ETS Learning of Kernel Languages

by

John M. Abela

B.Sc. (Mathematics and Computer Science), University of Malta, 1991.

M.Sc. (Computer Science), University of New Brunswick, 1994.

A Thesis Submitted in Partial Fulfillment of

the Requirements for the Degree of

Doctor of Philosophy

in the Faculty of Computer Science

Supervisor:	Lev Goldfarb, Ph.D., Faculty of Computer Science, UNB.
Examining Board:	Joseph D. Horton, Ph.D., Faculty of Computer Science, UNB.
	Viqar Husain, Ph.D., Faculty of Mathematics and Statistics, UNB.
	Maryhelen Stevenson, Ph.D., Faculty of Electrical and Computer Eng., UNB. (Chairperson)
External Examiner:	Professor Vasant Honovar, Artificial Intelligence Laboratory, Iowa State University.

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

November, 2002

The woods are lovely, dark, and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.

– Robert Frost
Stopping by the woods on a snowy evening

To my wife, Rachel,
to our children, Conrad and Martina,
to our parents,
and, last but not least, to Kaboose.

Abstract

The Evolving Transformations Systems (ETS) model is a new inductive learning model proposed by Goldfarb in 1990. The development of the ETS model was motivated by need for the unification of the two competing approaches that model learning – numeric (vector space) and symbolic. This model provides a new method for describing classes (or concepts) and also a framework for learning classes from a finite number of positive and negative training examples. In the ETS model, a class is described in terms of a finite set of weighted transformations (or operations) that act on members of the class. This thesis investigates the ETS learning of *kernel languages*. Kernel languages, first proposed by Goldfarb in 1992, are a subclass of the regular languages. A kernel language is specified by a finite number of weighted transformations (string rewrite rules) and a finite number of string called the *kernels*. One of the aims of this thesis is to show the usefulness and versatility of using distance, induced by the transformations, for both the class description of formal languages and also for directing the learning process. To this end, the author adopted a pragmatic approach and designed and implemented a new ETS learning algorithm - *Valletta*. Valletta learns multiple-kernel languages, with both random and misclassification noise, and has a user-defined inductive bias. This allows the user to indicate which ETS hypotheses (descriptions) are preferred over others. Valletta always finds an ETS language description that is consistent with the training examples - if one exists. Since ETS is a new model, few tools were available. A number of new tools were therefore purposely developed for this thesis. These include a string-edit distance function, *Evolutionary Distance*, a technique for reducing strings to their normal forms modulo a non-confluent string rewriting system, new refined formal

definitions of transformations system (TS) descriptions of formal languages, and a distance-driven search technique for Valletta's search engine. The usefulness of Valletta is demonstrated on a number of examples of learning kernel languages. Valletta performed very well on all the datasets and always converged to the correct class description in a reasonable time. This thesis also argues that the choice of representation (i.e. the encoding of the domain of discourse) and the choice of the inductive preference bias of a learning algorithm are, in general, crucial choices. ETS is not a learning algorithm but, rather, a learning model. In the ETS model, the representation (or encoding) of the domain of discourse and the preference inductive bias of an ETS learning algorithm are not fixed. The user chooses the representation and the inductive preference bias, consistent with the ETS model, that he or she deems appropriate for the learning task at hand. On the other hand, learning algorithms such as backpropagation neural networks fix the representation (vectors) and have a fixed inductive preference bias that cannot be changed by the user. This helps to explain why such neural networks perform badly on some learning problems.

Acknowledgements

I do not know where to start! Lev Golfarb, friend, mentor, and my Ph.D. supervisor comes first. I must thank Lev deeply for his patience, support, advice, and above all, inspiration over the years. His unwavering faith in the ETS Model inspired and encouraged me throughout. Next come my wife, Rachel, and our children Conrad and Martina. They all had to make many personal sacrifices while I away from home on my frequent, and often lengthy, visits to New Brunswick. I distinctly remember the occasions I would be freezing on a cold November day in Fredericton while they were still having barbecues on the beach back home in Malta. When Conrad was younger he would often come to me while I was pounding away at the keyboard and say "*Dada, could you please draw a chou-chou train with three coaches and with an elephant in the wagon at the back - please - so I can colour it?*". Mixing science and family was not always easy. I must thank my mother May, for putting up with my endless complaining - especially at the end, all of my brothers and sisters, and Rachel's parents, Teddy and Louise, for taking the kids away at the weekends so I could finish my work. Very special thanks also go to Prof. Joseph Horton for advice, encouragement, and inspiration. Prof. Horton is also the chairman of my Ph.D. committee. I would also like to thank Prof. Dana W. Wasson who is also on my Ph.D. committee, Dr. Bernie Kurz, the graduate-studies advisor, the Dean of Computer Science, Prof. Jane Fritz, and also all the staff at the Faculty Office. I thank my friend Alex Mifsud for advice and useful discussion. Finally, last but not least, I wish to thank my friends and fellow graduate colleagues Dmitry Korkin and Oleg Golubitsky for endless hours of discussion and much useful advice.

Contents

Abstract **iv**

Acknowledgements **vi**

List of Figures **xiv**

List of Tables **xv**

Preliminary Notation **1**

1 Introduction **2**

1.1 Background 2

1.2 The ETS Inductive Learning Model 5

1.2.1 Some Preliminary Definitions 10

1.2.2 Transformations Systems 11

1.2.3 Evolving Transformations Systems 12

1.2.4 Class Description in the ETS Model 13

1.2.5 Inductive Learning with ETS 15

1.3 Research Objectives 17

1.4 Thesis Organization 21

Part I - Setting The Scene **23**

2 Preliminaries **24**

2.1 Relations, Partial Orders, and Lattices 25

2.2 Strings, Formal Languages, and Automata 29
2.3 Reduction Systems 35
2.4 String-Rewriting Systems 40
2.4.1 Definitions and Notation 40
2.4.2 Length-Reducing String-Rewriting Systems 44
2.4.3 Congruential Languages 45
2.5 Pattern Recognition 47
2.6 Overview of Computational Learning Theory (CoLT) 49
2.6.1 What is learning after all? 50
2.6.2 Gold's results 53
2.6.3 The Inductive Learning Hypothesis 55
2.6.4 Probably Approximately Correct Learning 56
2.6.5 The PAC Learning Model 57
2.6.6 Inductive Bias 59
2.6.7 Occam's Razor 61
2.6.8 Other Biases 61
2.7 Grammatical Inference 62
2.7.1 The Grammatical Inference Problem 62
2.7.2 Some GI Techniques 66
2.8 String Edit Distances 70
2.8.1 Notes and Additional Notation 78

3 Kernel Languages **81**
3.1 TS Class Descriptions for Formal Languages 82
3.1.1 String Transformations Systems 83
3.1.2 String TS Class Descriptions of Formal Languages 86
3.1.3 Examples of String TS Class Descriptions 88
3.1.4 The Role of the Attractors in TS Class Descriptions 98
3.1.5 The Role of the Distance Function 100
3.1.6 Comparison with Other Forms of Description 103
3.1.7 Summary 106

3.2 Kernel Languages . . . 107
3.2.1 Preliminary Definitions . . . 109
3.2.2 Kernel Languages . . . 115
3.3 Evolutionary Distance (EvD) . . . 120
3.3.1 TS Descriptions for Kernel Languages . . . 125
3.3.2 Some Properties and Applications of Kernel Languages . . . 126

4 The GSN Learning Algorithm 128
4.1 Background . . . 128
4.2 Overview of the GSN Algorithm . . . 130
4.3 Results Obtained by the GSN Algorithm . . . 139
4.4 Problems with the GSN Algorithm . . . 141

Part II - *Valletta*: A Variable-Bias ETS Learning Algorithm 148

5 The *Valletta* ETS Algorithm 149
5.1 Overview . . . 150
5.1.1 How *Valletta* differs from the *GSN* Algorithm . . . 152
5.1.2 How *Valletta* Works — An Example . . . 155
5.1.3 Kernel Selection . . . 168
5.1.4 How *Valletta* Works — In Pictures . . . 171
5.2 *Valletta* in Detail . . . 176
5.2.1 The Pre-processing Stage . . . 176
5.2.2 An Algorithm for Global Augmented Suffix Trie Construction 183
5.2.3 The Search Lattice . . . 186
5.3 How Valletta Learns . . . 188
5.4 Computing the f function . . . 196
5.5 Reducing C^+ and C^- to their Normal Forms . . . 208
5.5.1 Feature Repair . . . 216
5.6 Summary and Discussion . . . 217

6 *Valletta* Analysis **219**
6.1 Time Complexity of the Preprocessing Stage 219
6.2 Time Complexity of String Reduction 221
6.3 Time Complexity of Computing f 223
6.4 Convergence 225

7 Experimentation, Testing, and Results **228**
7.1 Valletta's Testing Regimen 229
7.2 The *Darwin* Search Engine 236
7.3 Testing with the *GSN* DataSets 241
7.4 Learning in the Presence of Noise 243
7.5 The Monk's Problems 246
7.5.1 A Discussion of the Results 254
7.6 Comparison with the Price EDSM Algorithm 256
7.7 Representation and Bias 262
7.7.1 What is Representation? 263
7.7.2 Is Representation Important? 266
7.8 Analysis of the Results 273

8 Conclusions and Future Research **285**
8.1 Conclusions 285
8.2 Contributions of this Thesis 288
8.3 Future Research 292
8.3.1 Extensions to *Valletta* 292
8.3.2 A Distance Function for Recursive Features 295
8.3.3 ETS Learning of Other Regular Languages 296
8.3.4 Open Questions 297
8.4 Closing Remarks 298

Bibliography **299**

A Using Valletta **310**

B *Valletta*'s Inductive Bias Parameters **314**

C Training Sets used to test *Valletta* **317**

D GI Benchmarks **323**

E GI Competitions **325**

E.1 The Abbadingo One Learning Competition 325

E.2 The Gowachin DFA Learning Competition 326

F Internet Resources **327**

F.1 Grammatical Inference Homepage 327

F.2 The *pseudocode* LaTeX environment 327

G The Number of Normal Forms of a String **329**

H Kernel Selection is NP-Hard **332**

H.1 The Kernel Selection Problem 332

H.2 Transformation from MVC 333

I Trace of GLD Computation **335**

Vita

List of Figures

1.1 Class description in the ETS model. 7
1.2 Learning in the ETS model. 9
1.3 The correct metric structure for the language $a^n b^n$. 14
1.4 Optimization of the f function. 16

2.1 The Hässe diagram for the lattice P. 28
2.2 A DFA that accepts the language ab^*a. 34
2.3 Properties of reduction systems. 39
2.4 The error of the hypothesis h with respect to the concept c and the distribution $\mathcal{D}$. 57
2.5 The Prefix Tree Acceptor for the strings bb, ba, and aa. 67
2.6 Learning DFAs through state merging 68
2.7 String distance computation using *GLD*. 77

3.1 The pre-metric space embedding of the language a^*b. 89
3.2 Closest Ancestor Distance between the strings *abbab* and *acbca*. . . . 122
3.3 Distances between the normal forms of *0*, *110*, and*0010010*. 123
3.4 Why *EvD* satisfies the triangle inequality. 124

4.1 The pre-metric space embedding of the language $a^n b^n$. 132
4.2 The f_1 and f_2 functions. 134
4.3 Basic architecture of the GSN algorithm. 136
4.4 Adding a new dimension to the simplex. 137
4.5 Line graphs of the GSN results. 140

4.6 Why we need to find the kernel k. 147

5.1 High-level flowchart of *Valletta* showing the main loops. 157
5.2 The GAST built from the strings: *abccab*, *cabc*, and *cababc*. 159
5.3 The search lattice for the strings: *abccab*, *cabc*, and *cababc*. 162
5.4 The search tree built by *ETSSearch*. 163
5.5 The parse graphs for the strings: *abccab*, *cabc*, and *cababc*. 167
5.6 *Valletta*'s kernel selection procedure. 169
5.7 Normal Form Distance (NLD). 170
5.8 How *Valletta* Works — The Pre-Processing Stage 171
5.9 How *Valletta* Works — The Learning Stage 172
5.10 How *Valletta* Works — Computing f_2 and f_3. 173
5.11 How *Valletta* Works — Computing f_1. 174
5.12 How *Valletta* Works — String Reduction 175
5.13 The suffix tree for the string *010101*. 178
5.14 The suffix trie for the string *010101*. 180
5.15 The GAST for the strings: *010101*, *00101*, and *11101*. 182
5.16 The record structure of each GAST node. 184
5.17 The partially completed GAST for the string *aab*. 185
5.18 The search lattice built from the strings in R_{C^+}. 187
5.19 The completed search tree for the set $R_{C^+} = \{a, b, c, d\}$. 189
5.20 How Valletta expands the search. 195
5.21 Computing the distance between the normal forms. 198
5.22 Promoting the kernels used in f_3 computation. 199
5.23 A depiction of how the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ functions work. 205
5.24 A depiction of the kernel selection process. 206
5.25 Computing the new f_1 function. 207
5.26 The Edit-Graphs for the strings *1110100* and *00010101*. 208
5.27 A parse of the string *000101110100* using non-confluent features. . . 210
5.28 The parse graph structure showing the cross-over nodes. 212
5.29 Removal of redundant nodes in parse graph reduction. 213

5.30 Removal of redundant edges in parse graph reduction. 213
5.31 How *feature repair* works. 216

6.1 The search tree created from the strings $\{a, b, ab, ca, bc, cab\}$. 226

7.1 Screen dump of *Valletta* when learning of A1101 was completed. . . . 233
7.2 The search tree created by Valletta for the *a302* dataset. 234
7.3 High-level flowchart of the Darwin genetic algorithm search engine. . 237
7.4 Comparing the running times of the Valletta and GSN algorithms. . . 242
7.5 The new method for computing f_1 used for Valletta. 245
7.6 Some robots of the Monk's Problems. 246
7.7 The Alphabet used to encode the Monk datasets. 249
7.8 The alphabet used by MDINA. 252
7.9 The DFAs produced by the EDSM algorithm for different $0\{1\}^*$ datasets.257
7.10 The DFA produced by the EDSM algorithm for the *bin01* dataset. . . 259
7.11 The DFA produced by the EDSM algorithm for the *kernel01* dataset. 260
7.12 Enumerating the search space. 271
7.13 Breakdown of running time by procedure for the *a701* dataset. 273
7.14 Breakdown of running time by procedure for *a702* dataset. 274
7.15 Breakdown of running time by procedure for *a703* dataset. 274
7.16 The search tree created by Valletta for the *a302* dataset. 276
7.17 The behaviour of the f, f_1, and f_2 functions for the *a703* dataset. . . 278

8.1 A TCP/IP Farm for parallelizing *Valletta*. 295
8.2 A DFA for the regular language ab^*a. 296

A.1 The screen dump of Valletta during the learning process. 311

G.1 The reduction of the string *ababababa* modulo the feature set $\{aba, bab\}$.330

H.1 How to transform *Minimum Vertex Cover* to *Kernel Selection*. 334

List of Tables

2.1 The order relation for the lattice P. 28
2.2 String Edit Distance between the strings *abcb* and *acbc*. 71
2.3 Empty Distance Matrix for the strings *acbcb* and *abca*. 73
2.4 Completed Distance Matrix for the strings *acbcb* and *abca*. 74

4.1 A training set for the language $a^n b^n$. 131
4.2 The transformations discovered by the ETS learning algorithm. . . . 131
4.3 The main steps of the GSN ETS learning algorithm. 138
4.4 The training examples used to test the GSN learning algorithm. . . . 139

5.1 The training set for the language K_1 158
5.2 The Repeated Substrings array for the strings: *abccab*, *cabc*, and *cababc*.160
5.3 The set R_{C^+} created from the strings: *010101*, *00101*, and *11101*. . . 186
5.4 The normals forms of each independent segment. 211

7.1 Results obtained from testing Valletta. 232
7.2 A comparison of the results obtained for *Valletta* and *Darwin*. 240
7.3 The GSN datasets used for *Valletta/GSN* comparison. 241
7.4 The published results for the Monk's Problems. 248
7.5 The kernels discovered by the Mdina algorithm. 253
7.6 The *kernel01* training set. 261
7.7 Some strings from $a^n b^n$ and their Gödel Numbers. 265
7.8 A trace of the f, f_1, and f_2 functions for the *a703* dataset. 277

I.1 Distance matrix after GLD computation of *aba* and *abbba*. 342

Preliminary Notation

The following notational conventions will be used throughout this thesis.

$\mathbb{R}$ denotes the set of real numbers.

$\mathbb{N}$ denotes the set of non-negative integers. $\mathbb{N}^+$ denotes the positive integers.

In the case when upper-case Roman or Greek letters are used:

- $A \subset B$ denotes normal subset inclusion,
- $|A|$ denotes the cardinality of the set A.

In the case when lower-case Roman or Greek letters are used:

- $x \subset y$ denotes x *is a factor (substring) of* y,
- $|a|$ denotes the length of the string a.

Unless explicitly stated otherwise, Σ always denotes a *finite* alphabet of symbols and ε always denotes the empty string.

For any given set S, $\mathcal{P}(S)$ denotes the power set of S.

$\emptyset$ denotes the empty set or the empty language over Σ. Which of the two will be clear from the context.

The terms *class* and *concept* are used interchangeably.

Chapter 1

Introduction

The aim of this first chapter is to provide the motivation and background behind the research undertaken for this thesis. This chapter also contains a brief overview of Lev Goldfarb's ETS inductive learning model, a listing of the primary research objectives, and a discussion of the organization of the thesis.

1.1 Background

Evolving Transformations System (ETS) is a new inductive learning model proposed by Goldfarb [41]. The main objective behind the development of the ETS inductive learning model was the unification of the two major directions being pursued in Artificial Intelligence (AI), i.e. the *numeric* (or vector-space) and *symbolic* approaches. In Pattern Recognition (PR), analogously, the two main areas are the *decision-theoretic* and *syntactic/structural* approaches [16]. The debate on which of the two is the best approach to model intelligence has been going on for decades - in fact, it has been called the 'Central Debate' in AI [113]. In the very early years of AI, McCulloch and Pitts proposed simple neural models that manifested adaptive behaviour. Not much later, Newell and Simon proposed the *physical symbol systems*

paradigm as a framework for developing intelligent agents. These two approaches more-or-less competed until Minsky and Papert published their now famous critique of the perceptron, exposing its limitations. This shifted attention, and perhaps more importantly funding, towards the symbolic approach until the 1980s when the discovery of the *Error Back Propagation* algorithm and the work of Rumelhart *et al* reignited interest in the connectionist approach. Today, the debate rages on with researchers in both camps sometimes showing an almost childish reluctance to appreciate, and more importantly address, the other side's arguments and concerns. This long standing division between these two approaches is more than just about technique or competition for funding. The two sides differ fundamentally in how to think about problem solving, understanding, and the design of learning algorithms.

Goldfarb, amongst others, has long advocated the unification of the two competing 'models' [40, 41, 42, 45, 47, 49]. Goldfarb is not alone in his conviction that the single most pressing issue confronting cognitive science and AI is the development of a unified inductive learning model. Herman von Helmholtz [133], and John von Neumann [134] both insisted on the need for a unified learning model. In the Hixon Symposium in 1948, von Neumann spoke about the need for a '*new theory of information*' that would unite the two basic but fundamentally different paradigms — *continuous* and *discrete*. In the very early 1990's Goldfarb introduced his *Evolving Transformations Systems* (*ETS*) inductive learning model. In the ETS model, geometry (actually distance) plays a pivotal role. A class in a given domain of discourse O is specified by a small non-empty set of prototypical members of the class and a distance function defined on O. The set of objects that belong to the class is then defined to be all objects in O that lie within a small distance from one of the 'prototypes' of the class. Objects in the class are therefore close to each other. The distance function is a measure of dissimilarity between objects and is usually

taken to be the minimum cost (weighted) sequence of transformations (productions) that transforms one object into another. The assignment of a weight, a non-negative real number, to each transformation is what brings in continuity to the production system (symbolic) model [41]. Learning in the ETS model reduces to the problem of finding the set of transformations and the respective weights that yield the optimal metric structure. At each stage in the learning process, an ETS algorithm discovers, or rather *constructs*, new transformations out of the current set until class separation is achieved — hence the *evolving* nature of the model.

It must be emphasized that the ETS model in not a *learning algorithm* but, rather, a *learning formalism*. Unlike the connectionist model, it is not tied to just one particular method, i.e. vectors, of representing the objects in the domain of discourse. This flexibility is desirable since it allows that practitioner to choose the representation that gives the best class description. This point is discussed in Chapter 8. One of the aims of this thesis is to show how and why the ETS model allows for a much more compact, economical, and more importantly, *relevant* form of class description especially in the presence of noise. Also, unlike many learning algorithms, the ETS model does not assume a particular *inductive preference bias* (see Chapter 2). In other words, the ETS model does not fix any preference for one hypothesis over another. This versatility allows, in theory, for the construction of ETS learning algorithms for every conceivable domain. Learning algorithms such as *Candidate Elimination*, *ID3*, and even *Back-Propagation*, all have a built-in inductive preference bias that cannot be changed by the user. The implication is that some classes cannot be learned. This is an important, but very often misunderstood or even ignored, point which is discussed in Chapter 7.

In this thesis the author presents an ETS learning algorithm for *kernel languages*. Kernel languages are a subclass of the regular languages introduced by Goldfarb

in [49]. The algorithm, which is called *Valletta* after Malta's capital city and the author's home town, is completely *distance-driven*, i.e. the distance function directs the search for the correct class description. It appears that ETS algorithms are unique in this regard. Valletta is a variable-bias algorithm in the sense that the user can select an inductive preference bias[1] (i.e. a preference for certain hypotheses over others) before the learning process starts.

1.2 The ETS Inductive Learning Model

This section introduces and discusses the Evolving Transformations Systems (ETS) inductive learning model. The number of formal definitions and notation have been kept down to the absolute minimum. This is because the main objective of this section is to introduce the main ideas behind the model. In particular, to indicate how classes (or concepts) can be described using transformations systems and also how learning is achieved in the model. To this end, only the most important definitions and notation have been included. The ETS model has undergone significant development since its inception. During the preparation of this thesis, the author's colleagues in the *Machine Learning Group* at UNB undertook the formal development of the ideas contained in this section [54]. This has resulted in changes to the main definitions and notation. In this thesis, however, we shall be faithful to the definitions and notation used by Goldfarb in his papers on the ETS Model [44, 45, 46, 47, 48, 49].

One of the main ideas in the ETS model is that the concept of class distance plays an important, even critical, role in the definition and specification of the class. Given a domain of discourse O, a class C in this domain can be specified by a non-empty finite subset of C which we call the set of *attractors*, and which we denote by A, a

[1] See Section 2.6.5 for a definition.

non-negative real number δ, and by a distance function d_C. The set of all objects in O that belong to C is then defined to be:

$$\{o \in O \mid d_C(a, o) < \delta,\, a \in A\}.$$

In other words, the class consists precisely of those objects that are a distance of δ or less from some attractor. We illustrate with a simple example. Suppose we want to describe (i.e. specify) the class (or concept) *Cat*. Let O be the set of all animals, A a finite set of (prototypical) cats, δ a non-negative real number, and d_{Cat} a distance function defined on the set of all animals. Provided that A, δ, and d_{Cat} are chosen appropriately, the set of all cats is then taken to be the set of all animals that are a distance of δ or less from any of the attractors, i.e. the set of prototypical cats. This is depicted below in Figure 1.1. In our case, the set A contains just one prototypical cat although, in general, a class may have many prototypes. All animals that are in the δ-neighbourhood of this cat are classified as cats.

This idea borrows somewhat from the theory of concepts and categories in psychology (see Section 2.6 of Chapter 2). The reader is also referred to [102] for a discussion of Eleanor Rosch's theory of concept learning known as *Exemplar Theory*. Objects are classified together if they are, in some way, *similar*. In our example, all the animals that are cats are grouped together since the distance between any cat and the prototype is less than the threshold δ. In other words, an animal is a cat if it is similar to the cat prototype. The less distance there is between two animals, the more similar they are — i.e. distance is a measure of dissimilarity.

Some clarification of the above example is in order. It is not clear how to define a distance on the set of animals in order to achieve the correct specification of the class *cat*. Of course, one does not actually define the distance function on the set of animals but rather on their *representation*, i.e. the set of animals is mapped into

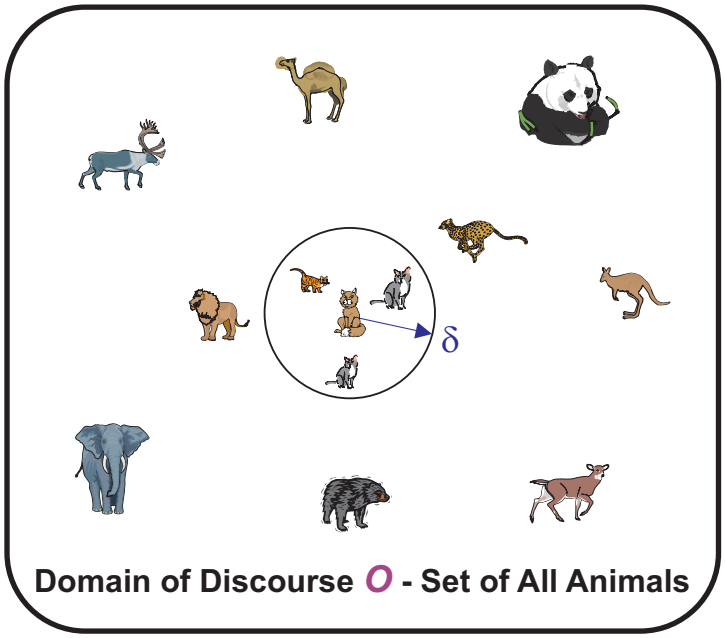


Figure 1.1: Class description in the ETS model.

some mathematical structure such as strings, trees, graphs, or vectors. The distance function is then defined on this set. It cannot be over-emphasized that there is a fundamental distinction between the set O of all animals and its representation, i.e. the numeric or symbolic encoding of the elements of O. The issue of representation is an important one. The reader is referred to Chapter 7 for a discussion. If one were to represent the animals by their genome, i.e. the string containing the DNA sequence, then it is conceivable that one could develop a string-edit distance function that would achieve the above. This can only be done, of course, if one assumes that the set of all strings that are DNA sequences of cats is a computable language. If a language is computable then it must have a finite description and be described by means of a grammar, automaton, Turing machine, etc. This is not asking too much. In machine learning it is always assumed that the class to be learned is computable — since otherwise it would not have a finite description. To summarize, in the ETS model a class C in a domain of discourse O is specified by a finite number of

prototypical instances of the class, the attractors, and by a distance function d_C such that all the members of the class lie within a distance of δ or less from an attractor, where δ is fixed for C.

The ETS model, however, is not just about class description, but also about learning class descriptions of classes from finite samples to obtain an *inductive class description*[2]. To give an overview of how this is done we must first give a working definition of the learning problem.

Definition 1.1 (The Learning Problem — An Informal Definition). *Let O be a domain of discourse and let $\mathcal{C}$ be a (possibly infinite) set of related classes in O. Let C be a class in $\mathcal{C}$ and let C^+ be a finite subset of C and C^- be a finite subset of O whose members do not belong to C. We call C^+ the **positive** training set and C^- the **negative** training set. The learning problem is then to find, using C^+ and C^-, a class description for C.*

Of course, in practice, this may be impossible since if the number of classes in $\mathcal{C}$ is infinite, then C^+ may be a subset of infinitely many classes in $\mathcal{C}$. In other words, no finite subset, on its own, can characterize an infinite set (see Chapter 2). We therefore insist only on finding a class description for some class $C' \in \mathcal{C}$ such that C' *approximates* C. This depends, of course, on our having a satisfactory definition of what it means for a class to approximate another. In essence, learning in the ETS model reduces to finding a distance function (defined in terms of a set of weighted transformations) that achieves *class separation*, i.e. a distance function such that the distance between objects in C^+ is zero or close to zero while the distance between an object in C^+ and an object in C^- is greater than zero. An ETS algorithm achieves this by iteratively modifying a distance function such that the objects in C^+ start moving towards each other while, at the same time, ensuring that the distance from

[2]Or *inductive class representation (ICR)*.

an object in C^+ to any object in C^- is always greater than some given threshold. This is depicted in Figure 1.2. The members of C^+ are, initially, not close together.

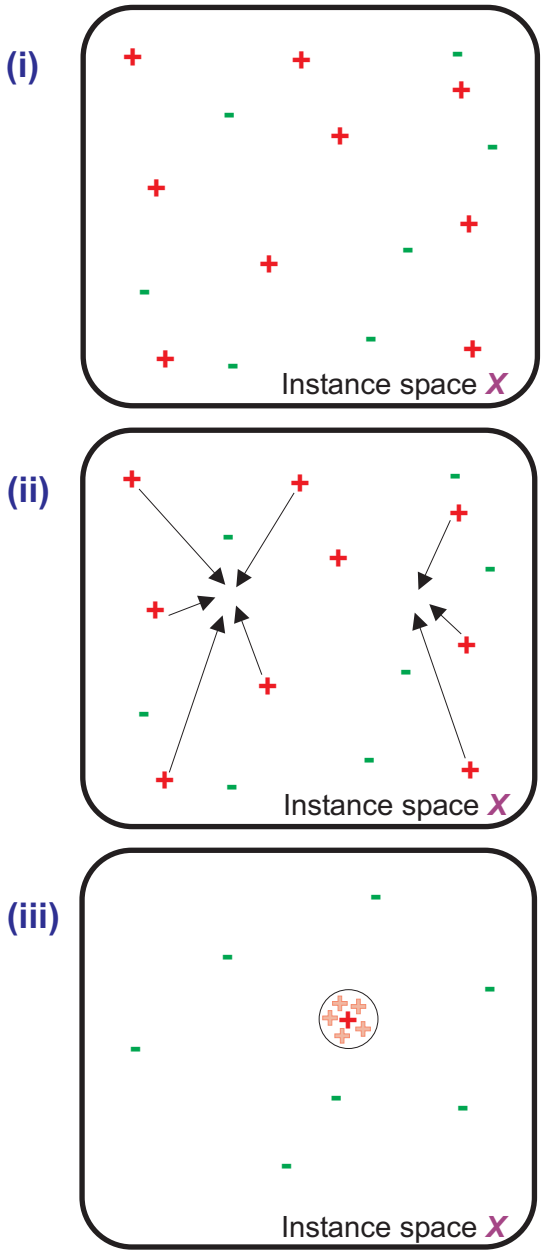


Figure 1.2: Learning in the ETS model.

As the learning process progresses, the members of C^+ start moving towards each other until, finally, all the members of C^+ all lie in a δ-neighbourhood.

1.2.1 Some Preliminary Definitions

The following definitions of *metric space*, *pre-metric space*, and *pseudo-metric* are those favoured by Goldfarb and appear in many of his papers. The reader is referred to [40] for an exposition.

Definition 1.2 (Metric Space).

A ***metric space*** *is a pair* (A, d) *where* A *is a set and* d *is a non-negative, real-valued mapping,*

$$d : A \times A \rightarrow \mathbb{R}^+ \cup \{0\},$$

that satisfies the following axioms:

1. $\forall a \in A,\ d(a, a) = 0,$
2. $\forall a_1, a_2 \in A,\ a_1 \neq a_2,\ d(a_1, a_2) > 0,$
3. $\forall a_1, a_2 \in A,\ d(a_1, a_2) = d(a_2, a_1),$ *and*
4. $\forall a_1, a_2, a_3 \in A,\ d(a_1, a_3) \leq d(a_1, a_2) + d(a_2, a_3).$ □

Definition 1.3 (Pre-metric Space).

A ***pre-metric space*** *is a pair* (A, d) *where* A *is a set and* d *is a non-negative, real-valued mapping,*

$$d : A \times A \rightarrow \mathbb{R}^+ \cup \{0\},$$

that satisfies the following axioms:

1. $\forall a \in A,\ d(a, a) = 0,$
2. $\forall a_1, a_2 \in A,\ d(a_1, a_2) \geq 0,$
3. $\forall a_1, a_2 \in A,\ d(a_1, a_2) = d(a_2, a_1),$ *and*
4. $\forall a_1, a_2, a_3 \in A,\ d(a_1, a_3) \leq d(a_1, a_2) + d(a_2, a_3).$ □

Definition 1.4 (Pseudo-metric Space).

*A **pseudo-metric space** is a pair (A, d) where A is a set and d is a non-negative, real-valued mapping,*

$$d : A \times A \rightarrow \mathbb{R}^+ \cup \{0\},$$

that satisfies the following axioms:

1. $\forall a \in A$, $d(a, a) = 0$, *and*
2. $\forall a_1, a_2 \in A$, $d(a_1, a_2) = d(a_2, a_1)$. □

Notes to Definitions. A pre-metric space is therefore identical to a metric space except that the distance between two *distinct* elements of A can be zero — i.e. for some $a_1, a_2 \in A$, $a_1 \neq a_2$, $d(a_1, a_2)$ can be zero. Note, therefore, that the definitions for *metric space* and *pre-metric space* differ only in Axiom 2. A pseudo-metric space, on the other hand, places much less restrictions on the distance function d. In a pseudo-metric space, we only require that for any $a \in A$, the distance $d(a, a)$, i.e. from a to itself, is zero. We also require the so-called *symmetry* axiom, i.e. for any pair $a_1, a_2 \in A$, the distance $d(a_1, a_2)$ is the same as $d(a_2, a_1)$. The pseudo-metric, therefore, does not have to satisfy the triangle inequality and this allows the distance between two non-identical objects to be zero.

1.2.2 Transformations Systems

Definition 1.5 (Transformation System). (From [49])

*A **transformations system (TS)**, $\mathcal{T} = (\mathbf{O}, \mathbf{S}, \mathbf{D})$, is a triple where $\boldsymbol{O}$ is a set of homogeneously structured objects, $\mathbf{S} = \{S_i\}_{i=1}^m$ is a finite set of transformations (substitution operations) that can transform one object into another, and $\boldsymbol{D}$ is a competing family of distance functions defined on $\boldsymbol{O}$.*

Notes to Definition 1.5. The definition of transformations system is meant to capture the idea that objects are built (or rather composed) from primitive objects and that any object can be transformed into any other object by the inserting, deleting, or substitution of primitive or complex objects. For example, if the set of objects is the set of strings over some alphabet Σ, the transformations would be string insertion, deletion, and substitution operations, i.e. rewrite rules. We always assume that the set of transformations is *complete*, i.e. it allows any object to transformed into any other object. The set of objects O is any set of structured objects such as strings, trees, graphs, vectors, etc. The set O is called the *domain of discourse* and its members are called *structs*. The set D is a family of *competing* distance functions[3] defined on O. Each transformation is assigned a *weight*, usually a non-negative real number. The distance between two objects $a, b \in O$ is typically taken to be the minimum weighted cost over all sequences of transformations that transform a into b. The distance functions are called *competing* since one has to find the set of weights that minimize the pairwise distance of the objects in the class. This point is elaborated upon in Chapter 3.

1.2.3 Evolving Transformations Systems

Definition 1.6 (Evolving Transformation System). (From [49])
*An **Evolving Transformations System** is a finite or infinite sequence of transformations systems,*

$$\mathcal{T}_i = (O, S_i, D_i), \;\; i = 1, 2, 3, \ldots$$

where $S_{i-1} \subset S_i$.

Notes to Definition 1.6. An ETS is therefore a finite or infinite sequence of TS's with a common set of structured objects. Each set of transformations S_i, except S_0

[3]Not necessarily metrics.

is obtained from S_{i-1} by adding to S_{i-1} one or several new transformations. The set of transformations S_i, therefore, *evolves* through time.

We now proceed to see how transformations systems can be used to:

1. describe classes in O, even in the presence of noise, and

2. learn the classes in O from some training examples.

1.2.4 Class Description in the ETS Model

In the ETS model, a class C in a domain of discourse O is specified by a finite subset, A, of prototypical members of the class and by a distance function d_C. This distance function is that associated (or rather, *induced*) by a set of weighted transformations. The set A is called the set of *attractors*. The set of objects belonging to C is then defined to be

$$\{o \in O \,|\, d_C(a, o) < \delta,\, a \in A\}.$$

Using distance to specify and define the class gives us enormous flexibility. We illustrate with a simple example. Suppose the domain of discourse is the set, Σ^*, of all strings over the alphabet $\Sigma = \{a, b\}$. In this case the transformations are rewriting rules, i.e. insertion, deletion, and substitution string operations. Consider, as an example, the following set of transformations and its weight vector:

Transformation	Weight
$a \leftrightarrow \varepsilon$	0.5
$b \leftrightarrow \varepsilon$	0.5
$aabb \leftrightarrow ab$	0.0

The transformation $a \leftrightarrow \varepsilon$ denotes the insertion/deletion of the character a while $aabb \leftrightarrow ab$ denotes the substitution (in both directions) of the string $aabb$ by the string ab. The reader should note that the first two transformations are assigned a

non-zero weight while the last transformation is assigned a zero weight. Also, the set of transformations is complete, i.e. any string in Σ^* can be transformed into any other string in Σ^*. Now suppose we wanted to describe the context-free language $L = a^n b^n$. We can accomplish this by letting the set of attractors be equal to $\{ab\}$, i.e. the set containing just one attractor — the string ab. We then define the distance function d_L to be the minimum cost over all sequences of transformations that transform one string into another. The cost of a sequence is the sum of the weights of the transformations in the sequence. For example, to transform the string $aaabb$ into the string ab one can first delete an a using the transformation $a \leftrightarrow \varepsilon$ and then replace $aabb$ by ab using the transformation $aabb \leftrightarrow ab$. Note that the cost of this sequence is 0.5. The attractor ab and distance function d completely specify the language (or class) $a^n b^n$. Any string in the language can be transformed into any other string in

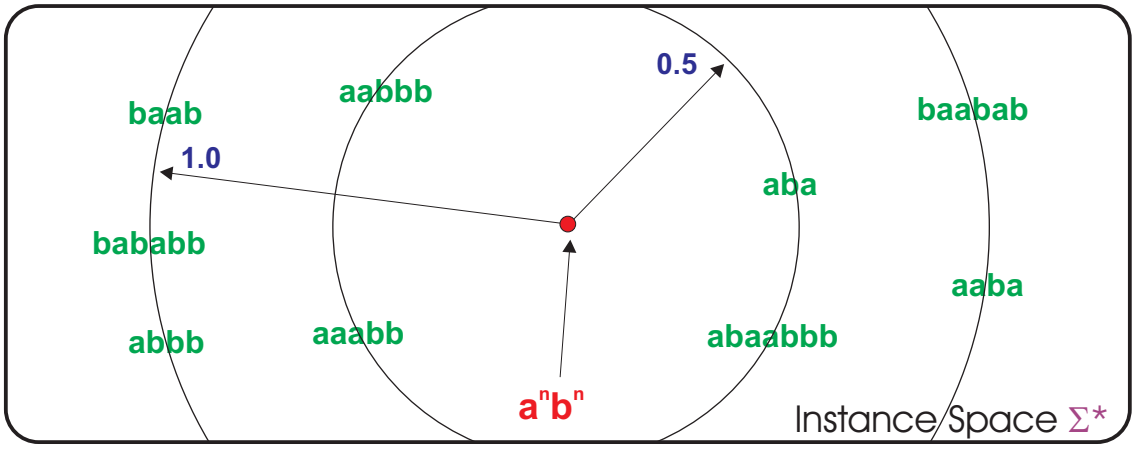


Figure 1.3: The correct metric structure for the language $a^n b^n$.

the same language using only the zero-weighted transformation $aabb \leftrightarrow ab$. All the strings in the language, therefore, have a pair-wise distance of zero. A string in the language and another string not in the language will have a pair-wise distance greater than zero. For example, as shown above, the distance from the string $aaabb$, which is not in the language, to the string ab, which belongs to the language, is 0.5. We

say that the distance function d_L gives the *correct metric structure*[4] for the language (class) L. This is depicted in Figure 1.3.

Is it easy to see that the distance function d_L gives us a measure of how 'noisy' a string is. The more 'noise', i.e. spurious characters, the string has, the further away it is from a string in L. As we shall see in Chapter 3, this method of class description gives us a very natural and elegant way for handling noisy languages and it is well known that noise occurs very often in real-world Pattern Recognition (PR) problems [16].

1.2.5 Inductive Learning with ETS

Learning in the ETS Model reduces to searching for the distance function that yields the correct metric structure. Now since the distance function is itself defined in terms of a set of transformations together with its weight vector, learning, in essence, involves searching for the correct set of transformations and then finding the optimal set of weights. As with all learning problems, one is given a finite set C^+ of objects that belong to some unknown class C and a finite set C^- of objects that do not belong to C. The task is then to take these training examples and infer a description of a class C' such that *approximates* C (see Chapter 2). An ETS algorithm discovers the correct metric structure by optimizing the following function:

$$f = \frac{f_1}{c + f_2}, \tag{1.1}$$

where f_1 is the minimum distance (over all pairs) between C^+ and C^-, f_2 is the average pair-wise *intra-set* distance in C^+, and c is a small positive real constant to avoid divide-by-zero errors. The aim here is to find the distance function such that the distance between any two objects in C^+ is zero or close to zero while the distance between an object in C^+ and an object in C^- is appropriately greater

[4] In general, d_L may be a metric, pre-metric, or a pseudo-metric.

than zero. We therefore try to maximize f_1 and, more importantly, to minimize f_2. When the value of f exceeds a pre-set threshold t we say that we have achieved *class separation* and, hence, the correct metric structure. During learning, an ETS algorithm uses the value of f to direct the search for the correct set of transformations, i.e. the set that describes the class C. Figure 1.4, below, shows a depiction of the optimization of the f function. An ETS learning algorithm iteratively builds new

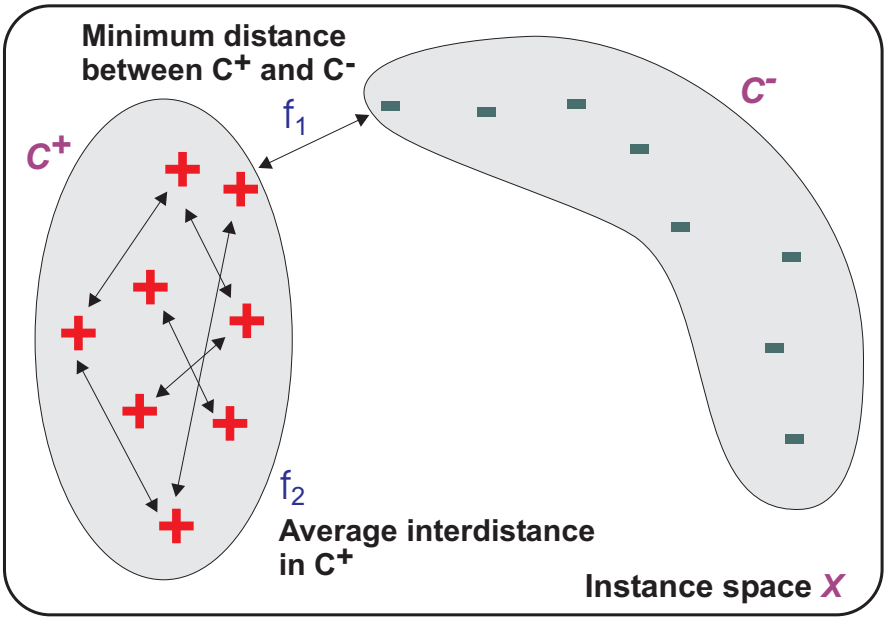


Figure 1.4: Optimization of the f function.

transformations systems until it discovers the set of transformations and the weight vector that give class separation. The ETS algorithm, therefore, creates an evolving transformations system (ETS) — a sequence of transformations systems (TS's). Each TS in the sequence is built from the TS preceding it through the addition of new transformations until, finally, a TS is found that gives the correct metric structure. The reader is referred to [49, 92] for an exposition and also to Chapter 4 in which the GSN ETS learning algorithm is discussed.

1.3 Research Objectives

In the early 1990's, two Master's students at UNB who where working closely with Lev Goldfarb, implemented the first ETS inductive learning algorithm. In his Master's thesis Santoso [107] described a basic algorithm for ETS inductive learning and introduced a new string-edit distance function, *Generalized Levensthein Distance*, or *GLD*, that was used to describe a subclass of regular languages called *kernel languages*. Nigam [92], together with Lev Goldfarb, then developed and implemented the first grammatical inference[5] (GI) algorithm that uses ETS principles. This algorithm, hereafter referred to as the *GSN algorithm*, was the first implementation ever of the ideas of Lev Goldfarb. The GSN algorithm was the first algorithm to describe classes in terms of a distance function and to use distance to direct the search for the correct class description. The domain chosen by Nigam and Goldfarb to develop and test the algorithm was kernel languages. A kernel language consists of all those strings over some given alphabet Σ that can be obtained by inserting, anywhere, in any order, and any number of times, any string from a finite set of strings called the *features* into a non-empty string called the *kernel*. The only restriction being that no feature can be a substring of any other feature or of the kernel. This domain was an example of a *structurally unbounded environment* (see Chapter 4). The concept of a structurally unbounded environment was proposed by Goldfarb to describe those environments that cannot be hard-coded into a learning algorithm. This prevents 'cheating' by the learning algorithm. The GSN algorithm did very well and, *prima facie*, the results seeming nothing less than spectacular. The algorithm learned all of the training classes from very small training sets even in the presence of noise. The author felt that the results obtained from the GSN algorithm most definitely warranted further investigation. To this end the author undertook to conduct further

[5] see Section 2.7 in Chpater 2 for a definition.

development of the GSN algorithm in order to answer the following questions:

1. Is the GLD distance function suitable for the class description of kernel languages?

2. Could the GSN algorithm learn in the presence of more noise?

3. What is the time and space complexity of the GSN algorithm?

4. What is the *inductive preference bias* of the GSN algorithm?

5. Can the GSN algorithm be modified to learn multiple-kernel languages?

The answers to the above questions can be found in Chapter 4. Nigam did not analyse the time and space complexity of his algorithm. This is because his main thesis objective was to present a 'proof of concept', i.e. to demonstrate the viability of implementing an ETS grammatical inference algorithm. One problem with the GSN algorithm is that, although still polynomial, computation of the f function is still very compute intensive. This is because computing the f function requires a total number of distance computations that is quadratic in the cardinality of the training set and where each distance computation is itself quadratic in the length of the two strings. This means that as the size of the training set is increased and the strings get longer, the time required for computing f increases considerably. A number of problems were also identified with the GLD distance function itself and also with the learning strategy used by the GSN algorithm. A discussion can be found in Chapter 4. Although the GSN algorithm had some problems it was still felt that it merited further development and investigation. Many researchers in the grammatical inference community, including Miclet [86], have advocated the development of different approaches to the GI problem. The GSN algorithm employs a fundamentally new learning model, ETS, and in general, was very promising. It did

not seem to have any problems which could not conceivably be overcome. The GSN algorithm was therefore the starting point of the research undertaken for this thesis. The initial aims of the research undertaken for this thesis were: to continue further development of the GSN algorithm, to address the problems that were identified, and also to extend the algorithm so it would learn larger concept classes[6] with more noise. The primary research objective can be stated as:

To investigate the role of distance for the purpose of the class description and the ETS inductive learning of kernel languages.

We decided, after much deliberation, to restrict the learning domain, i.e. the class of languages learnt by the algorithm, to kernel languages. This class of languages is a structurally unbounded environment and, it turns out, has practical applications. It was also decided that our new ETS learning algorithm would consider multiple-kernel languages, with more noise, with larger training sets and longer strings, and with much less restrictions on the positive training strings. For reasons that are discussed later on in this thesis, learning multiple-kernel languages is much harder than the case when the language has only one kernel. All practical applications of kernel languages that we came across were, as a matter of fact, multiple-kernel. The problems identified with the GLD distance function used by the GSN algorithm meant also that a new string-edit distance algorithm that allowed the correct description of kernel languages had to be developed. To this end we had to refine and continue development of the definitions and the theory of TS descriptions of formal languages and then to develop TS descriptions of kernel languages with particular attention given to the case when the language is noisy. The GSN algorithm has a fixed *prefer-*

[6] A set of related classes.

ence inductive bias (see Chapter 2) and this means that some perfectly valid kernel languages cannot be learnt. We decided very early on that the new algorithm would have variable inductive bias. This would allow the user to change the inductive bias according to the application. It eventually became clear that, rather than modifying the GSN algorithm, a new algorithm would have to be developed. The new algorithm was called *Valletta*. Valletta is loosely based on the GSN algorithm but uses a completely new distance function, a new method for computing the f function, a new pre-processing stage, and a new search strategy that allows for a variable inductive bias. It must be stressed that Valletta is a means to an end. The main objective of this thesis was not to produce an artifact but rather to investigate the role of distance in the class description and learning of kernel languages. The main aim of Valletta is to investigate the feasibility of using distance to direct the learning process itself and to identify the issues and problems involved in such a task. We, of course, gave due attention and importance to the time and space complexity of Valletta.
To summarize, in order to achieve the main research objective we had to consider the following secondary objectives:

1. Refine the definitions of TS descriptions of formal languages.

2. Define formally the class of kernel languages and study their properties. Also, to try and find practical applications of kernel languages.

3. Generalized Levensthein Distance (GLD) had a number of properties that made it unsuitable for describing kernel languages. The new algorithm therefore required a new string edit distance function, and an efficient algorithm to implement it, that would address the problems with GLD.

4. Develop a new learning strategy that could learn multiple-kernel languages.

5. Show that the new algorithm always finds a TS description consistent with the training examples (if one exists).

6. Comparison with other methods.

1.4 Thesis Organization

This thesis is divided into two parts.

Part I — Setting the Scene

As its name suggests, Part I contains background material and also the theory developed for the *Valletta* algorithm described in Part II. Chapter 2, *Preliminaries*, contains the background material necessary for understanding the remainder of the thesis. Chapter 2 includes only material which was deemed absolutely necessary for understanding this thesis. Chapter 3, *Kernel Languages*, introduces and discusses a subclass of the regular languages first proposed by Lev Goldfarb. In this chapter Goldfarb's original definitions are expanded and refines and also includes a discussion of some of the interesting properties of this class of languages. Updated definitions for Transformations System (TS) descriptions for formal languages can also be found in Chapter 3. In Chapter 4, *The GSN Algorithm*, we discuss the Goldfarb, Santoso, and Nigam ETS inductive learning algorithm and list its main problems. The GSN algorithm was the starting point of the research undertaken for this thesis.

Part II — Valletta: A Variable-Bias ETS Learning Algorithm

Part II of this thesis presents the *Valletta* ETS inductive learning algorithm for kernel languages. Chapter 5, *The Valletta ETS Algorithm*, starts off with a listing of the design objectives for Valletta and then proceeds to a detailed discussion of how Valletta

works. The various data structures and techniques developed for Valletta, including the new string edit distance function used by the algorithm, are also discussed in this chapter. Chapter 6, *Valletta Analysis*, contains an analysis of Valletta's time and space complexity. In this chapter we shall also see that Valletta will always find a TS description consistent with a valid, i.e. structurally complete, training set. In Chapter 7, *Valletta Results*, we discuss the results obtained from the testing regimen that was designed for Valletta and also compare Valletta's performance with that of other grammatical inference algorithms. In Chapter 8, *Conclusions*, the author draws some conclusions from his experience in developing and implementing the Valletta algorithm and also discusses the results obtained. In this chapter we also discuss if and how the research objectives were met. Chapter 8 also contains a number of recommendations for future research, including improvements and enhancements to Valletta, as well a discussion of some related open questions.

The reader is advised to read Chapter 2 before any of the other chapters. This chapter contains important background material and will save the reader the effort of consulting the various references for this material. Some of the material and notation in Chapter 2 is new and, indeed, probably unique to this thesis. Chapter 3, in which we formally introduce and discuss transformations system (TS) descriptions for formal languages and kernel languages, as well as Chapter 4, where we discuss GSN ETS inductive learning algorithm, can be skipped at first reading. The reader who wants to get a quick, general overview of the ideas and results contained in this thesis should first read Chapters 1 and 5 and then proceed to Chapters 7 and 8.

Part I

Setting The Scene

Computer Science is no more about computers

than astronomy is about telescopes.

E. W. Dijkstra

Chapter 2

Preliminaries

The aim of this chapter is to present the basic ideas, notions, definitions, and notation that are necessary for understanding the material in this thesis. Most of the material can be found in standard undergraduate textbooks but some of the definitions and notation are unique to this thesis. In particular, the reader is advised to read Sections 2.2 (Strings, Languages, and Automata), 2.3 (Reduction Systems), 2.4 (String Rewriting Systems), and 2.8 (String Edit Distances) since these contain ideas, definitions, and notation that are either non-standard or developed purposely for this thesis. Section 2.6 contains a brief synoptic survey of the principal concepts, results, and problems in Computational Learning Theory (CoLT) and Section 2.7 presents the main ideas in Grammatical Inference (GI) theory. The reader may choose to skip either section if he or she is familiar with the topic. It was envisaged that the reader may have to refer to this chapter regularly when reading the rest of this thesis and therefore, apart from providing numerous references, the author adopted a direct style — listing the main ideas and definitions and, as much as possible, avoiding surplus detail.

2.1 Relations, Partial Orders, and Lattices

The intuitive notion of a relationship between two elements of a set is succinctly captured by the mathematical notion of a *binary relation*. This section contains the main definitions relevant to this thesis. The reader is referred to the excellent book by Davey and Priestley [23] where most of the definitions come from. For all of the definitions in this section, P always denotes an arbitrary (finite or infinite) set.

Definition 2.1. *A **binary relation**, denoted by $\rightarrow$, is any subset of the Cartesian product $P \times P$. For any binary relation $\rightarrow \subset P \times P$:*

$$domain(\rightarrow) \stackrel{\text{def}}{=} \{\, a \mid \exists b,\ (a,b) \in \rightarrow \},\ \textit{and}$$
$$range(\rightarrow) \stackrel{\text{def}}{=} \{\, b \mid \exists a,\ (a,b) \in \rightarrow \}.$$

□

Although most authors prefer to use the notation $a \sim b$ to denote '*a is related to b*', the alternative notation $a \rightarrow b$ is used in this thesis. This is to emphasize that '*a is related to b*' does not necessarily imply that '*b is related to a*' and also because this is the standard notation used in the study of reduction systems and string-rewriting systems.

Definition 2.2. *The **inverse relation** to $\rightarrow$, denoted by $\rightarrow^{-1}$, is defined in the following manner: $\rightarrow^{-1} \stackrel{\text{def}}{=} \{\, (b,a) \mid (a,b) \in \rightarrow \}$.* □

For reasons of clarity the symbol $\leftarrow$ will henceforth be used to denote the inverse of the relation $\rightarrow$. This is because most of the relations considered in this thesis are those in string-rewriting systems where, for any two strings a and b, $a \rightarrow b$ means '*a is related to b if b can be obtained from a by replacing a substring x in a with the string y*'. Arrows, therefore, are useful because they indicate the direction of the replacement and eliminate (or perhaps reduce) confusion.

Definition 2.3. *A relation $\rightarrow$ is a **partial order** if $\rightarrow$ is reflexive, anti-symmetric, and transitive. If $\rightarrow$ is a partial order on P then P is a **partially ordered set** or **poset**.* □

Definition 2.4. *Any two elements $x, y \in P$ are called **comparable** (under $\rightarrow$) if either $x \rightarrow y$ or $y \leftarrow x$ or $x = y$.* □

Definition 2.5. *If $\rightarrow$ is a partial order on P, then $\rightarrow$ is called a **total order** if any two elements in P are comparable. In this case P is called a **linearly ordered set** or a **totally ordered set**.* □

Definition 2.6. *Let $\rightarrow$ be a partial order on P. In this thesis a **chain** is a finite sequence of elements of P, $p_0, p_1, p_2, ..., p_n$, such that $p_i \rightarrow p_{i+1}$ for $0 \leq i < n$.* □

Definition 2.7. *Let $\rightarrow$ be a partial order on P and let $x, y \in P$. We say that y is **covered** by x (or x **covers** y), and write $x \rightarrowtail y$ or $y \leftarrowtail x$, if $x \rightarrow y$ and $x \rightarrow z \rightarrow y$ implies $x = z$. We call $\rightarrowtail$ the **covering relation** on $\rightarrow$. If $\exists\, z \in P$ such that $x \rightarrow z \rightarrow y$ then we write $x \rightsquigarrow y$ and say x **does not cover** y.* □

For example, in the totally ordered set $(\mathbb{N}, \leq)$, where $\mathbb{N}$ is the set of natural numbers, $m \rightarrowtail n$ if and only if n = m + 1. In the case of $(\mathbb{R}, <)$, where $\mathbb{R}$ is the set of reals, there are no pairs x, y such that $x \rightarrowtail y$. Note that we insist that the covering relation is irreflexive.

Definition 2.8. *Let $\rightarrow$ be a partial order on P and let $Q \subseteq P$. Q is called a **down-set** (or alternatively a **decreasing set** or **order ideal**) if whenever $x \in P$ and $y \in Q$ and $y \rightarrow x$ then $x \in Q$. An **up-set** (or alternatively an **increasing set** or **order filter**) is defined analogously.* □

Definition 2.9. *Let P be a partially ordered set and let $Q \subseteq P$. Then*

*(a) $a \in Q$ is a **maximal element** of Q if $a \rightarrow x$, $x \in Q$ implies $a = x$;*

*(b) $a \in Q$ is the **greatest** (or **maximum**) element of Q if $a \rightarrow x \ \forall x \in Q$, and in that case we write $a = max(Q)$.* □

A **minimal** element of Q, the **least** (or **minimum**) element of Q and $min(Q)$ are defined dually. One should note that Q has a greatest element only if it has precisely **one** maximal element, and that the greatest element of Q, if it exists, is unique (by the anti-symmetry of $\rightarrow$).

Definition 2.10. *Let P be a partially ordered set. The greatest element of P, if it exists, is called the **top element** of P and is denoted by $\top$ (pronounced 'top'). Similarly, the least element of P, if it exists, is called the **bottom element** and is denoted by $\perp$.* □

Definition 2.11. *Let P be a partially ordered set and let $S \subseteq P$. An element $x \in P$ is called an **upper bound** of S if $x \rightarrow s \forall s \in S$. A **lower bound** is defined similarly. We denote the set of all upper bounds of S by S^u and the set of all lower bounds of S by S^l.*

*One should note that, since $\rightarrow$ is transitive, then S^u and S^l are always an up-set and a down-set respectively. If S^u has a least element, x, then x is called the **least upper bound** (or the **supremum**) of S. Similarly, if S^l has a largest element, x, then x is called the **greatest lower bound** (or **infimum**) of S.* □

Recall from above that, when they exist, the top and bottom elements of P are denoted by $\top$ and $\perp$ respectively. Clearly, if P has a top element, then $P^u = \{\top\}$ and therefore $sup(P) = \top$. Likewise $inf(P) = \perp$ whenever P has a bottom element. **Notation**. In this thesis the following notation will be used: $x \vee y$ (read as 'x **join** y' in place of $sup(x, y)$ and $x \wedge y$ (read as 'x **meet** y') in place of $inf(x, y)$. Similarly, $\bigvee S$ and $\bigwedge S$ are used to denote $sup(S)$ and $inf(S)$ respectively.

Definition 2.12. *Let P be a non-empty partially ordered set. Then P is called a* ***lattice*** *if $x \vee y$ and $x \wedge y$ exist. If $\bigvee S$ and $\bigwedge S$ exist $\forall S \subseteq P$, then P is called a* ***complete lattice****. If P has maximal and minimal members then it follows from the definition of infimum and supremum that these must be unique. Such a lattice is called a* ***bounded lattice****.* □

Example 2.1. *Let $P = \{a, b, c, d, e, f\}$ and let the relation $\rightarrow$ be defined as shown in Table 2.1. Figure 2.1 shows the Hässe diagram for the lattice P.*

a $\rightarrow$ b	c $\rightarrow$ e
a $\rightarrow$ c	d $\rightarrow$ f
b $\rightarrow$ d	e $\rightarrow$ f
c $\rightarrow$ d	

Table 2.1: The order relation for the lattice P.

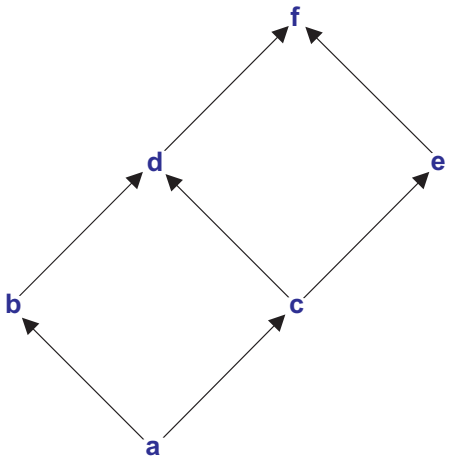


Figure 2.1: The Hässe diagram for the lattice P.

2.2 Strings, Formal Languages, and Automata

This section contains the basic definitions and notation for strings, languages, and automata as used throughout the thesis. The main purpose of this section is to establish notation and, although it is assumed that the reader is familiar with the above concepts, it is still recommended that this section is read since some of the notation is non-standard and only found in literature on string-rewriting systems, string combinatorics, and grammatical inference. The reader may wish to consult [59, 62, 79, 115] for expositions.

Many of the definitions below come directly, or are adapted, from [79]. Yet others come from [103] and some are indeed unique to this thesis.

Notation 2.13. *Let Σ be a finite*[1] ***alphabet**. Its elements are called **letters**, **characters**, or **symbols**. A **string** over the alphabet Σ is a finite sequence of characters from Σ.* □

*(a) We denote by ε the **empty string** (the sequence of length 0).*

*(b) Σ^* denotes the set of **all possible strings** over Σ. Σ^* is the **free monoid** generated by Σ under the usual operation of string concatenation with the empty string ε as the identity. Σ^+ denotes the set of **all non-empty strings**, i.e. $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$.*

(c) We use the usual exponent notation to denote multiple concatenation of the same string. For any string $s \in \Sigma^*$, $s^0 = \varepsilon$, $s^1 = s$, and $s^n = s^{n-1} \cdot s$, where $\cdot$ denotes string concatenation. If $s = ab$, then $s^3 = ababab$. Note that we often use parentheses to identify the string being repeated: $(ab)^3 = ababab$ while $ab^3 = abbb$.

[1] In this thesis, attention is restricted to finite alphabets only

(d) *We denote the* ***length*** *of a string* $s \in \Sigma^*$ *by* $|s|$. *Formally,* $|\varepsilon| = 0$, $|a| = 1$ *for* $a \in \Sigma$, *and* $|sa| = |s| + 1$ *for* $a \in \Sigma, s \in \Sigma^*$.

(e) Σ^n *denotes the set of* ***all strings of length n*** *and* $\Sigma^{\leq n}$ *denotes the set of* ***all strings of length less than or equal to n***.

(f) *For any string* $s \in \Sigma^*$, $s[i]$ *denotes the* i*th character of* s *where* $1 \leq i \leq |s|$.

(g) *For any* $a \in \Sigma$ *and for any* $s \in \Sigma^*$ *we denote by* $|s|_a$ *the number of occurrences of the character* a *in* s. *For any subset* $A \subseteq \Sigma$ *and for any* $s \in \Sigma^*$ *we denote by* $|s|_A$ *the number of characters in* s *that belong to* A. *Therefore,* $|s|_A = \sum_{a \in A} |s|_a$.

(h) *We denote by alph(s) the subset of* Σ *that contains exactly those characters that occur in* s. *Therefore,* $alph(s) \stackrel{\text{def}}{=} \{a \,|\, a \in \Sigma, |s|_a \geq 1\}$.

(i) *A string* $x \in \Sigma^*$ *is said to be a* ***substring*** *of another string* $y \in \Sigma^*$ *if* $\exists u, v \in \Sigma^*$ *such that* $y = uxv$. *Notice that 'is a substring of' is a binary relation on* Σ^* *that induces a partial order on* Σ^*. *If* $x \neq y$ *then we say that* x *is a* ***proper substring*** *of* y. *If* x *is a substring of* y *then we write* $x \subset y$. ***s[i..j]*** *denotes the substring of* s *that starts at position* i *and ends at position* j. *Notice that, by convention,* $\varepsilon \subset s$, $\forall s \in \Sigma^*$. *The set of all substrings of* s *is denoted by* $\textbf{\textit{Substrings}}^*\textbf{\textit{(s)}} \stackrel{\text{def}}{=} \{x \in \Sigma^* | \, x \subset s\}$.

(j) *A string* $x \in \Sigma^*$ *is said to be a* ***prefix*** *of another string* $y \in \Sigma^*$ *if* $\exists v \in \Sigma^*$ *such that* $y = xv$. *If* $x \neq y$ *then* x *is said to be a* ***proper prefix*** *of* y. *We denote by* $s^{(i)}$ *the* ***prefix of length i*** *of* s. *The* ***set of all prefixes*** *of* s *is denoted by* $\textbf{\textit{Prefix}}^*\textbf{\textit{(s)}} \stackrel{\text{def}}{=} \{s^{(i)} | \, 1 \leq i \leq |s|\} \cup \{\varepsilon\}$.

(k) *A string* $x \in \Sigma^*$ *is said to be a* ***suffix*** *of some other string* $y \in \Sigma^*$ *if* $\exists v \in \Sigma^*$ *such that* $y = vx$. *If* $x \neq y$ *then* x *is said to be a* ***proper suffix*** *of* y. *We*

denote by $s_{(i)}$ *the* ***suffix of length i*** *of* s*. The* ***set of all suffixes*** *of* s *is denoted by* $\textbf{\textit{Suffix}}^*\textbf{\textit{(s)}} \stackrel{\text{def}}{=} \{s_{(i)} | \, 1 \leq i \leq |s|\} \cup \{\varepsilon\}$.

(l) *A set of strings* $S \subset \Sigma^*$ *is said to be* ***substring free*** *if no string in* S *is a substring of some other string in* S*. Formally,* S *is substring free if* $Substrings^*(s) \cap S = \{s\}, \forall s \in S$*.* ***Prefix free*** *and* ***suffix free*** *sets of strings are defined similarly.*

(m) *A string* $x \in \Sigma^*$ *is said to be* ***primitive*** *if it is not a power of some other string in* Σ^**. I.e. if* $x \neq \varepsilon$ *and* $x \neq y^n$ *for some* $y \in \Sigma^*$ *and some* $n > 1$.

(n) *A string* $s \in \Sigma^*$ *is called a* ***square*** *if it is of the form* xx *where* $x \in \Sigma^+$*. A string* s *is said to* ***contain a square*** *if one of its substrings is a square; otherwise, it is called* ***square-free****.*

(o) *A string* $x \in \Sigma^*$ *is said to be a* ***subsequence*** *of some other word* $y \in \Sigma^*$ *if* $x = a_1 a_2 a_3 \cdots a_n$*, with* $a_i \in \Sigma$*,* $n \geq 0$*, and* $\exists z_0, z_1, z_2, \cdots, z_n \in \Sigma^*$ *such that* $y = z_0 a_1 z_1 a_2 \cdots a_n z_n$*. A subsequence of a string* S *is therefore any sequence of characters that is in the same order as it appears in* s*.*

(p) *Two strings* $x, y \in \Sigma^*$ *are said to be* ***conjugate*** *if* $\exists u, v \in \Sigma^*$ *such that* $x = uv$ *and* $y = vu$ *for* $u \neq \varepsilon$ *and* $v \neq \varepsilon$*.*

(q) *Let* $u, v \in \Sigma^+$ *be two non-empty strings and let* u *have two distinct occurrences as substrings in* v*. Clearly then, there must exist strings* x*,* y*,* x'*, and* y' *such that the following must hold:*

$$w = xuy = x'uy' \text{ with } x \neq x'$$

The two occurrences of u *either* ***overlap****, are* ***disjoint****, or are consecutive (****adjacent****). Let us examine each possibility in turn. Without loss of generality, suppose that* $|x| < |x'|$*. Then*

- $|x'| > |xu|$. *For this to be true there must exist some* $z \in \Sigma^+$ *such that* $x' = xuz$ *and* $w = xuzuy'$. *The two occurrences of* u *are therefore clearly* ***disjoint***.
- $|x'| = |xu|$. *This means that* $x' = xu$ *and therefore* $w = xuuy'$ *contains a square. The two occurrences of* u *are* ***adjacent***.
- $|x'| < |xu|$. *The second occurrence of* u *starts before the first ends. The occurrences of* u *are said to* ***overlap***. □

The problem of finding overlapping occurrences of the same substring within a given string will arise later on in our discussion of *kernel languages* (see Chapter 3). The following lemma will prove to be a useful and interesting result.

Lemma 2.1. *Let* $w \in \Sigma^*$ *be a string over* Σ. *Then* w *contains 2 overlapping occurrences of a non-empty string* u *if and only if* w *contains a substring of the form* avava, *where* $a \in \Sigma$ *and* $v \in \Sigma^*$. ■

The reader is referred to [79, page 20] for the proof. Any string of the form *avava* is said to **overlap** (with itself). According to Lemma 2.1, a string has two overlapping occurrences of a substring if and only if it contains a substring of the form *avava*. This result is useful since it allows for an efficient procedure for searching for overlapping substrings in strings.

Let us now turn our attention to sets of strings. The subsets of Σ^* are called **(formal) languages**. A language can be finite or infinite. If the language is infinite then we are interested mainly whether or not is has a *finite description*. This description can take many forms — a grammar, a regular expression, a finite state automaton, a Turing machine, etc. These descriptions are used for *specifying*, *generating*, and *recognizing* formal languages. In Chapter 3, a new form of description for formal languages [115], the *Transformations System (TS) Description* is introduced. This description was developed for the purpose of *learning* formal languages.

In this thesis we are concerned primarily with *regular languages*. Regular languages are the simplest languages in the Chomsky hierarchy and have been the subject of much study [3, 127]. For a finite alphabet Σ, the class of regular languages over Σ is precisely the class of regular sets over Σ, i.e. the smallest class of subsets of Σ^* that contains all the finite subsets and is closed under the operations of union, concatenation, and Kleene star (*). Regular languages can be specified (also generated and recognized) by left linear grammars, right linear grammars, regular expressions, non-deterministic finite state automata, and deterministic finite state automata. The latter are important since a unique deterministic finite state automaton that has a minimal number of states exists for each regular language. This gives us a canonical description of the regular language. The reader may wish to consult [115, 56] for further details and an exposition.

Definition 2.14 (Finite State Automata).

(a) A ***deterministic finite state automaton (DFA)*** $\mathcal{A}$ *is specified by the 5-tuple* $\mathcal{D} = \langle Q, \Sigma, \delta, s, F \rangle$ *where*

- Q *is a finite set of* ***states****,*
- Σ *is a finite alphabet,*
- $\delta : Q \times \Sigma \to Q$ *is the* ***transition function****,*
- $s \in Q$ *is the* ***start state****, and*
- $F \subseteq Q$ *is the set of* ***accepting states****.*

(b) The transition function $\delta : Q \times \Sigma \to Q$ *of a DFA can be extended to* $Q \times \Sigma^*$ *as follows:*

$$\delta(q, \varepsilon) = q$$
$$\delta(q, wa) = \delta(\delta(q, w), a).$$

*(c) A string $x \in \Sigma^*$ is **accepted by** $\mathcal{D}$ if $\delta(s, x) \in F$ and*

*$L(\mathcal{D}) \stackrel{\text{def}}{=} \{x \in \Sigma^* | x \text{ is accepted by} \mathcal{D}\}$ is called the **language accepted by** $\mathcal{D}$.*

□

Notice that if q is a state and a is an alphabet symbol then the transition function ensures that $|\delta(q, a)| = 1$, i.e. $\mathcal{D}$ can reach only one state from q after reading a. This is what makes $\mathcal{D}$ deterministic. We can also define a **nondeterministic finite state automaton (NFA)** by appropriately modifying the definition of DFA. The transition function is changed to $\delta : Q \times \Sigma \to 2^Q$ and extended to $Q \times \Sigma^*$ as follows:

$$\delta(q, \varepsilon) = \{q\}$$
$$\delta(q, wa) = \cup_{p \in \delta(q,w)} \delta(p, a).$$

It turns out that for every regular language R, there exists a DFA $\mathcal{D}$ and an NFA $\mathcal{A}$ such that both $\mathcal{D}$ and $\mathcal{A}$ both recognize R. NFAs are usually easier to work with since if $\mathcal{A}$ is an NFA of n states that accepts a regular language R, a corresponding DFA, $\mathcal{D}$, that also accepts R, may have up to 2^n states. Figure 2.2 shows the minimal DFA for the language associated with the regular expression ab^*a.

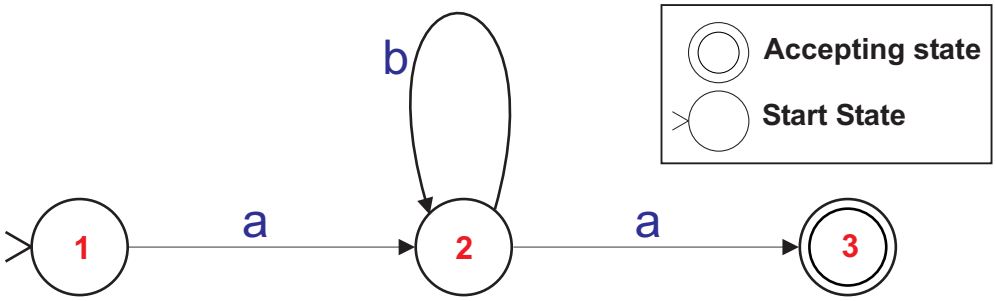


Figure 2.2: A DFA that accepts the language ab^*a.

2.3 Reduction Systems

The Norwegian mathematician and logician Axel Thue [123] considered the following problem: Suppose one is given a set of objects and a set of rules (or transformations) that when applied to a given object yield another object. Now suppose one is given two objects x and y. Can x be transformed into y? Is there perhaps another object z such that both x and y can be transformed into z?

In the case when the objects are strings, this problem became known as the *word problem*. Thue published some preliminary results about strings over a finite alphabet. Although Thue restricted his attention to strings he did suggest, however, that one might be able to generalize this approach to more structured combinatorial objects such as trees, graphs, and other structured objects. This generalization was later developed and the result was *reduction systems*. Reduction systems are so called because they describe, in an abstract way, how objects are transformed into other objects that are, by some criterion, *simpler* or *more general*. As discussed in Chapter 1, in ETS theory we also want to capture the idea of a set of *structs*, or structured objects, that are generated from a finite subset of simple (i.e. irreducible) structs using operations (or transformations) that transform one struct into another. This is essentially the opposite process of reduction. This notion and reduction systems fall under the general name of *replacement systems* [67]. Replacement systems are now an important area of research in computer science and have applications in automated deduction, computer algebra, formal language theory, symbolic computation, theorem proving, program optimization, and now of course, also machine learning.

This section is important since the definitions, notation, and techniques presented here are used throughout the thesis — in particular in the definitions of string-rewriting systems later on in Section 2.4 and kernel languages in Chapter 3. The reader is referred to [12, Chapter 1] and [63] for expositions.

Definition 2.15 (Reduction System). [12, page 10]

Let S be a set and let $\rightarrow$ be a binary relation on S. Then:

*(a) The structure $R = (S, \rightarrow)$ is called a **reduction system**. The relation $\rightarrow$ is called the **reduction relation**. For any $x, y \in S$, if $x \rightarrow y$ then we say that x **reduces** to y.*

*(b) If $x \in S$ and there exists no $y \in S$ such that $x \rightarrow y$, then x is called **irreducible**. The set of all elements of S that are irreducible with respect to $\rightarrow$ is denoted by **IRR(R)**.*

*(c) For any $x, y \in S$, if $x \overset{*}{\longleftrightarrow} y$ and y is irreducible, then we say that y is a **normal form** of x. Recall that $\overset{*}{\longleftrightarrow}$ is the reflexive, symmetric, and transitive closure of $\rightarrow$.*

(d) For any $x \in S$, we denote by $\Downarrow_{\mathbf{R}} (\mathbf{x}) \overset{\text{def}}{=} \{y \in S \mid x \overset{*}{\longleftrightarrow} y, y \text{ is irreducible}\}$, the set of normal forms of x modulo R.

*(e) If $x, y \in S$ and $x \overset{*}{\rightarrow} y$, then x is an **ancestor** of y and y is a **descendant** of x. If $x \rightarrow y$ then x is a **direct ancestor** of y and y is a **direct descendant** of x.*

*(f) If $x, y \in S$ and $x \overset{*}{\leftrightarrow} y$ then x and y are said to be **equivalent**.* □

Notation 2.16 (Reduction System). [12, page 10]

Let $R = (S, \rightarrow)$ be a reduction system. Then:

(a) For each $x \in S$:

*Let $\mathbf{\Delta}(\mathbf{x})$ denote the set of **direct descendants** of x with respect to $\rightarrow$. Thus, $\Delta(x) \overset{\text{def}}{=} \{y \mid x \rightarrow y\}$. Also, let $\Delta^+(x) \overset{\text{def}}{=} \{y \mid x \overset{+}{\rightarrow} y\}$ and $\Delta^*(x) \overset{\text{def}}{=} \{y \mid x \overset{*}{\rightarrow} y\}$. Thus, $\Delta^*(x)$ is the set of descendants of x modulo $\rightarrow$.*

(b) *For each $A \subseteq S$:*

*Let $\boldsymbol{\Delta}(\mathbf{A})$ denote the set of **direct descendants** of A with respect to $\rightarrow$. Thus, $\Delta(A) = \cup_{x \in A} \Delta(x)$. Also, let $\Delta^+(A) \stackrel{\text{def}}{=} \cup_{x \in A} \Delta^+(x)$ and $\Delta^*(A) \stackrel{\text{def}}{=} \cup_{x \in A} \Delta^*(x)$ Thus, $\Delta^*(A)$ is the set of descendants of the subset A modulo $\rightarrow$.*

(c) *For each $x \in S$:*

*Let $\nabla(\mathbf{x})$ denote the set of **direct ancestors** of x with respect to $\rightarrow$. Thus, $\nabla(x) \stackrel{\text{def}}{=} \{y \,|\, y \rightarrow x\}$. Also, let $\nabla^+(x) \stackrel{\text{def}}{=} \{y \,|\, y \stackrel{+}{\rightarrow} x\}$ and $\nabla^*(x) \stackrel{\text{def}}{=} \{y \,|\, y \stackrel{*}{\rightarrow} x\}$. Thus, $\nabla^*(x)$ is the set of ancestors of x modulo $\rightarrow$.*

(d) *For each $A \subseteq S$:*

*Let $\nabla(\mathbf{A})$ denote the set of **direct ancestors** of A with respect to $\rightarrow$. Thus, $\nabla(A) \stackrel{\text{def}}{=} \cup_{x \in A} \nabla(x)$. Also, let $\nabla^+(A) \stackrel{\text{def}}{=} \cup_{x \in A} \nabla^+(x)$ and $\nabla^*(A) \stackrel{\text{def}}{=} \cup_{x \in A} \nabla^*(x)$ Thus, $\nabla^*(A)$ is the set of ancestors of the subset A modulo $\rightarrow$.*

(e) *Note that $\stackrel{*}{\longleftrightarrow}$ is an equivalence relation on S. For each $s \in S$ we denote by $[s]_R$ the equivalence class of s mod(R). Formally, $[s]_R \stackrel{\text{def}}{=} \{\, y \,|\, y \stackrel{*}{\longleftrightarrow} S \,\}$. Also, for any $A \subseteq S$, $[A] \stackrel{\text{def}}{=} \cup_{x \in A} [x]_R$.* □

Definition 2.17. [12, page 10]

Let R be a reduction system.

(a) *The **common ancestor problem** is defined as follows:*

Instance: *$x, y \in S$.*

Problem: *Is there a $w \in S$ such that $w \stackrel{*}{\rightarrow} x$ and $w \stackrel{*}{\rightarrow} y$? In other words, do x and y have a common ancestor?*

(b) *The **common descendant problem** is defined as follows:*

Instance: *$x, y \in S$.*

Problem: *Is there a* $w \in S$ *such that* $x \xrightarrow{*} w$ *and* $y \xrightarrow{*} w$*? In other words, do* x *and* y *have a common descendant?*

(c) The ***word problem*** *is defined as follows:*

Instance: $x, y \in S$.

Problem: *Are* x *and* y *equivalent under* $\xleftrightarrow{*}$*?* □

In general these problems are undecidable [12]. However, there are certain conditions that can be imposed on reduction systems in order for these questions to become decidable.

Lemma 2.2. *Let* $(S, \rightarrow)$ *be a reduction system such that for every* $x \in S$*,* x *has a unique normal form. Then* $\forall x, y \in S$*,* $x \xleftrightarrow{*} y$ *if and only if the normal form of* x *is identical to the normal form of* y*.*

Proof of Lemma 2.2 Let $x, y \in S$ and let x' and y' denote the normal forms of x and y respectively.
$\Rightarrow$ Suppose that $x \xleftrightarrow{*} y$ and $x' \neq y'$. Then $x \xleftrightarrow{*} y'$ since $x \xleftrightarrow{*} y$ (by assumption) and $y \xleftrightarrow{*} y'$ (by definition). Now y' is irreducible (by definition) and therefore x has two distinct normal forms: x' and y'. This is a contradiction.
$\Leftarrow$ Suppose that x and y have a common normal form z. Then, by definition, $x \xleftrightarrow{*} z$ and $y \xleftrightarrow{*} z$. The results follows from the symmetry and transitivity of $\xleftrightarrow{*}$ ■

The proof of this lemma was omitted in [12]. The above result means that if for all $x, y \in S$ we have an algorithm to check if $x = y$ (very easy for strings), and also an algorithm to compute the unique normal forms of x and y, then the word problem becomes *always* decidable.

Definition 2.18. [12, page 11]
Let R *be a reduction system.*

(a) *R is* ***confluent*** *if* $\forall w, x, y \in S$, $w \xrightarrow{*} x$ *and* $w \xrightarrow{*} y$ *implies that* $\exists z \in S$ *such that* $x \xrightarrow{*} z$ *and* $y \xrightarrow{*} z$.

(b) *R is* ***locally confluent*** *if* $\forall w, x, y \in S$, $w \to x$ *and* $w \to y$ *implies that* $\exists z \in S$ *such that* $x \xrightarrow{*} z$ *and* $y \xrightarrow{*} z$.

(c) *R is* ***Church-Rosser*** *if* $\forall x, y \in S$, $x \xleftrightarrow{*} y$ *implies that* $\exists z \in S$ *such that* $x \xrightarrow{*} z$ *and* $y \xrightarrow{*} z$. □

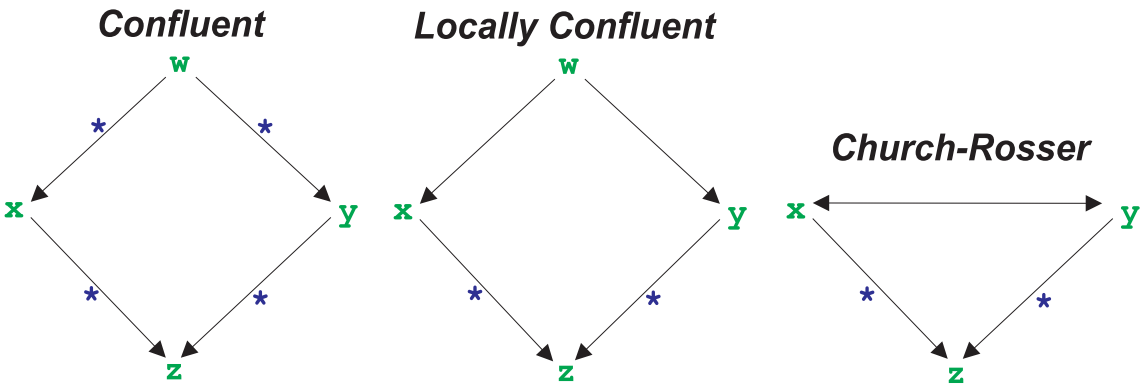


Figure 2.3: Properties of reduction systems.

Definition 2.19. [12, page 12]

Let R be a reduction system. The relation $\to$ *is* ***noetherian*** *if there is no infinite sequence* $x_0, x_1, x_2, \cdots \in S$ *such that* $x_i \to x_{i+1}$ *for* $i \geq i$. *If R is confluent and* $\to$ *is noetherian then R is* ***convergent***. □

If R is a reduction system and $\to$ is noetherian then we are assured that at least one normal form exists. If R is finite the word problem and the common descendant problem are decidable. Furthermore, if R is noetherian and convergent then, for every $s \in S$, $[s]_R$ has one unique normal form. In addition, if R is convergent then R is confluent if and only if R is locally confluent (see proof of Theorem 1.1.13 in [12]).

2.4 String-Rewriting Systems

A string-rewriting system T is a set of *rewriting rules* of the form (l, r) where $l, r \in \Sigma^*$ for some finite alphabet Σ. The reduction system associated with T is $R = (\Sigma^*, \rightarrow_T)$ where $\rightarrow_T$ is the *reduction relation* induced by T. If $(l, r) \in T$ implies that $(r, l) \in T$ then T is called a *Thue System* otherwise it is called a *semi-Thue System*. In recent years there has been a resurgence of interest in Thue systems [7, 12, 13, 63, 67]. This interest is perhaps due to the advances made in computer algebra, automated deduction and symbolic computation in general [13]. There have also been a number of new results in the theory of replacement systems and this has spurned on more research. In this thesis we are concerned primarily with string-rewriting systems that induce reduction relations that are noetherian and, in particular, those that have only length-reducing rules, i.e where $|l| > |r| \, \forall (l, r) \in T$. This property is desirable since it ensures that for any string $x \in \Sigma^*$, the normal forms of x exist and are computable. It turns out that string-rewriting systems can be used to (partially) specify a subclass of formal languages called Kernel Languages. This topic is discussed in Chapter 3.

2.4.1 Definitions and Notation

It is assumed that the reader is familiar with the main definitions of Reduction Systems and the associated notation from Section 2.3.

Definition 2.20 (String-Rewriting Systems). *Let Σ be a finite alphabet.*

*(a) A **string-rewriting system** T on Σ is a subset of $\Sigma^* \times \Sigma^*$ where every pair $(l, r) \in T$ is called a **rewrite rule**.*

*(b) The **domain** of T is the set $\{l \in \Sigma^* \mid \exists r \in \Sigma^* \, and \, (l, r) \in T\}$ and denoted by **dom(T)**. The **range** of T is the set $\{r \in \Sigma^* \mid \exists l \in \Sigma^* \, and \, (l, r) \in T\}$ and denoted by **range(T)**.*

(c) *When T is finite the **size** of T, which we denote by $\|T\|$, is defined to be the sum of the lengths of the strings in each pair in T. Formally,* $\|T\| \stackrel{\text{def}}{=} \sum_{(l,r)\in T}(|l| + |r|)$.

(d) *The **single-step reduction relation** on Σ^*, $\rightarrow_T$, induced by T is defined as follows: for any $x, y \in \Sigma^*$, $x \rightarrow_T y$ if and only if $\exists u, v \in \Sigma^*$ such that $x = ulv$ and $y = urv$. In other words, $x \rightarrow_T y$ if and only if the string y can be obtained from the string x by replacing the substring l in x by r to obtain y.*

*The **reduction relation** on Σ^* induced by T, which we denote by $\stackrel{*}{\rightarrow}_T$, is the reflexive, transitive closure of $\rightarrow_T$.*

(e) *$R_T = \{\Sigma^*, \rightarrow_T\}$ is the reduction system induced by T.*

(f) *The **Thue Congruence** generated by T is the relation $\stackrel{*}{\longleftrightarrow}_T$ — i.e. the symmetric, reflexive, and transitive closure of $\rightarrow_T$. Any two strings $x, y \in \Sigma^*$ are **congruent mod(T)** if $x \stackrel{*}{\longleftrightarrow}_T y$. For any string $w \in \Sigma^*$, the (possibly infinite) set $[w]_T$, i.e. the equivalence class of the string w mod(T), is called the **congruence class** of w (mod(T)).*

(g) *Let S and T be two string-rewriting systems. S and T are called **equivalent** if they generate the same Thue congruence, i.e. if $\stackrel{*}{\longleftrightarrow}_S = \stackrel{*}{\longleftrightarrow}_T$.* □

Notes to Definition 2.20. For any string-rewriting system T on Σ, the pair $(\Sigma, \rightarrow_T)$ is a reduction system. T is a finite set of string pairs (rules) of the form (l, r). Each rule can be interpreted to mean 'replace l by r'. The reduction relation induced by T, $\rightarrow_T$, is usually much larger than T itself since it contains not just the rules of T but also all those strings pair (x, y) such that, for some $a, b \in \Sigma^*$, $y = arb$ is obtained from $x = alb$ by a single application of the rule (l, r). In practice, for obvious reasons, $\rightarrow_T$ is infinite.

Many of the properties of reduction systems discussed in Section 2.3 apply also to R_T. In particular, if T is a string-rewriting system on Σ and $R_T = (\Sigma^*, \rightarrow_T)$ is the reduction system induced by T, then, for any two strings $x, y \in \Sigma^*$:

- $\rightarrow_T$ is **confluent** if whenever $w \xrightarrow{*}_T x$ and $w \xrightarrow{*}_T y$ for some $w \in \Sigma^*$, then $\exists z \in \Sigma^*$ such that $z \xrightarrow{*}_T z$ and $y \xrightarrow{*}_T z$. T is therefore confluent if whenever any 2 strings have a common ancestor they also have a common descendant.

- $\rightarrow_T$ is **Church-Rosser** if whenever $x \xleftrightarrow{*}_T x$ then $\exists z \in \Sigma^*$ such that $z \xrightarrow{*}_T z$ and $y \xrightarrow{*}_T z$. Informally, $\rightarrow_T$ is Church-Rosser if any pair of equivalent strings has a common descendant.

- $\rightarrow_T$ is **locally confluent** if whenever $w \rightarrow_T x$ and $w \rightarrow_T y$ for some $w \in \Sigma^*$, then $\exists z \in \Sigma^*$ such that $z \xrightarrow{*}_T z$ and $y \xrightarrow{*}_T z$. In other words, $\rightarrow_T$ is locally confluent whenever any two strings have a common direct ancestor they also have a common descendant.

It is important to note that the above are not if-and-only-if conditions. For any two strings x and y, x and y having a common descendant does not necessarily imply that x is equivalent to y or that they have a common ancestor. Consider, as an example, the string-rewriting system $T = \{(ax, z), (ay, z)\}$ where $\Sigma = \{a, b, x, y, z\}$. The strings axb and ayb have a common descendant since $axb \rightarrow_T zb$ and $ayb \rightarrow_T zb$ but clearly cannot have a common ancestor.

As from this point onwards, purely in the interests of brevity and clarity, we shall omit the subscript T and simply use $\rightarrow$, $\xrightarrow{*}$, and $\xleftrightarrow{*}$ instead of $\rightarrow_T$, $\xrightarrow{*}_T$, and $\xleftrightarrow{*}_T$.

Definition 2.21 (Orderings on Σ^*). *Let $\rhd$ be a binary relation on Σ.*

*(a) If T is a string-rewriting system on Σ, $\rhd$ is said to be **compatible with T** if $l \rhd r$ for each rule $(l, r) \in T$.*

(b) $\rhd$ *is a* ***strict partial ordering*** *if it is irreflexive, anti-symmetric, and transitive.*

(c) If $\rhd$ *is a strict partial ordering and if,* $\forall x, y \in \Sigma^*$*, either* $x \rhd y$*, or* $y \rhd x$*, or* $x = y$*, then* $\rhd$ *is a* ***linear ordering****.*

(d) $\rhd$ *is* ***admissible*** *if,* $\forall x, y, a, b \in \Sigma^*$*,* $x \rhd y$ *implies that* $axb \rhd ayb$*. In other words, left and right concatenation preserves the ordering.*

(e) $\rhd$ *is called* ***well-founded*** *if it is a strict partial ordering and if there is no infinite chain* $x_0 \rhd x_1 \rhd x_2 \cdots$*. If* $\rhd$ *is well-founded but also linear then it is a* ***well-ordering****.* □

Notes to Definition 2.21. It turns out that if T is a string-rewriting system on Σ then $\rightarrow_T$ is noetherian if and only if there exists an admissible well-founded partial ordering $\rhd$ on Σ^* that is compatible with T. (Lemma 2.2.4 in [12]). This is useful because, for reasons outlined previously, we want to consider only string-rewriting systems that are noetherian. For any string-rewriting system T, in order to establish whether $\rightarrow_T$ is noetherian we need only find (or construct) an admissible well-founded partial ordering that is compatible with T. In our case we usually opt for the **length-lexicographical ordering**, i.e. where strings are ordered according to length first and then lexicographically.

Notice also that for any string-rewriting system T, the set of direct descendants of a string $x \in \Sigma^*$ modulo T, $\Delta(x)$, is finite. This is true even if $\rightarrow_T$ is not noetherian and follows from the fact that any string $x \in \Sigma^*$ has a finite number of substrings and therefore the rules in T can only be applied in a finite number of ways. On the other hand, if $\rightarrow_T$ is noetherian, then $\forall x \in \Sigma^*$, the set of all descendants of x, $\Delta^*(x)$, is finite. This follows by König's Infinity Lemma.

Definition 2.22 (Normalized String-Rewriting Systems). *Let T be a string-rewriting system on Σ. T is* ***normalized*** *if, for every rule $(l, r) \in T$,*

(a) $l \in IRR(T - \{(l, r)\})$, and

(b) $r \in IRR(T)$. □

Notes to Definition 2.22. Informally, T is normalized if and only if, for each rule (l, r) in T, the left-hand side l can only be reduced by the rule (l, r) itself and the right-hand side r is irreducible. If T is a string-rewriting system that is not normalized, i.e. it contains rules whose right-hand side that is reducible, there is a polynomial time algorithm that on input T will output a string-rewriting system T' such that T' is normalized and equivalent to T [12, page 47]. Unless otherwise stated, all string-rewriting systems that are considered from now on are normalized.

2.4.2 Length-Reducing String-Rewriting Systems

A string-rewriting system T is called *length-reducing* if $(l, r) \in T$ implies that $|l| > |r|$. In other words the left hand side of a rule is always longer than the right hand side. The obvious implication of this property is that when a rule is applied to a string x the resulting string x' is always strictly shorter than x. Recall that if S is any string-rewriting system on Σ then $\rightarrow_S$ is noetherian if and only if there exists an admissible well-founded partial ordering $\rhd$ on Σ^* that is compatible with S. Therefore, let $\rhd$ be the length-lexicographical partial order on Σ^*. Since $l \rhd r$ clearly holds $\forall (l, r) \in T$ and also since $\rhd$ is admissible and well-founded, then we can conclude $\rightarrow_T$ is noetherian. There are three particular types of length-reducing used in this thesis.

Definition 2.23 (Monadic String-rewriting Systems).

Let T be a string-rewriting system on Σ. T is called ***monadic*** *if T is length-reducing*

and $|r| = 1$ *or* $r = \varepsilon$, $\forall (l, r) \in T$.

Definition 2.24 (Special String-rewriting Systems).

Let T *be a string-rewriting system on* Σ. T *is called* ***special*** *if* T *is length-reducing and* $r = \varepsilon$, $\forall (l, r) \in T$.

Definition 2.25 (Trivial String-rewriting Systems).

Let T *be a string-rewriting system on* Σ. T *is called* ***trivial*** *if* $r = \varepsilon$, *and* $l = a, a \in \Sigma$, $\forall (l, r) \in T$. □

2.4.3 Congruential Languages

Let us now investigate, informally, the possibility (and later on the feasibility) of using string-rewriting systems to define, and also test for membership of, formal languages.

Let T be a string-rewriting system over an alphabet Σ. How can we use T to define a proper language $L \subset \Sigma^*$? In other words, are there any interesting, non-trivial languages *induced by* T? Of course, we must define exactly what it means for a language L to be induced by a string-rewriting system T. Let us examine some possibilities.

- Let L_1 be the set of all irreducible strings modulo T, i.e. $L_1 = \mathrm{IRR(T)}$.
- Let L_2 be the union of all the equivalence classes with respect to T.

Observe that L_1 can be finite or infinite depending on T. It turns out that the set $IRR(T)$, i.e. the set of all irreducible strings modulo some string-rewriting system T, is a regular language and a finite state automaton that recognizes L_1 can be constructed in polynomial time from T [12, Lemma 2.1.3, page 37]. Whether such languages are useful or not is open to discussion but a review of the literature does not reveal any particular use. L_2, it turns out, is Σ^* itself. It appears, therefore, that

T by itself is rather limited for the purpose of defining formal languages. Suppose, however, that we use T together with a finite number of equivalence (congruency) classes modulo T.

Definition 2.26 (Congruential Languages). [80]

Let T be a finite string-rewriting system on Σ. A ***congruential language*** *is any finite union, C, of congruency classes of T.* □

A congruential language is specified by the pair (T, C). Since both T and C are finite sets, we have a finite description of the language. Congruential languages have been subject to study by various researchers [10, 11, 13, 80]. One interesting result is that all NTS languages are congruential [11]. A context-free grammar is said to be NTS if the set of sentential forms it generates is unchanged when the rules are used both ways. It is quite common, and also sensible, to restrict attention to congruential languages where T has only length-reducing rules. This will ensure that $\rightarrow_T$ is noetherian and this guarantees that the normal forms for each string exist and are computable. An example of congruential languages where T has only length reducing rules is given below.

Example 2.2. *An Example of a Congruential Language*

- *Let $\Sigma = \{a, b\}$ and let $T = \{(ab, \varepsilon), (\varepsilon, ab)\}$. This is the Dyck Language of matching parenthesis.*

A class of languages, *Kernel Languages*, that is very similar to congruential languages but which also differ in a number of important ways, is described in Chapter 3.

2.5 Pattern Recognition

Humans have perceptive and cognitive abilities that we very often take for granted - that is until we try to replicate these abilities on a computer. It is only then that we realize how powerful and complex human perceptive and cognitive abilities are. Human have absolutely no problem in differentiating between a picture of a horse and that of a rabbit, or perhaps between the smell of an onion and that of an orange. However, it is not easy to program these tasks on a digital computer. This difficulty is probably because (From [27])

> 'each pattern usually contains a large amount of information, and the recognition problems typically have an inconspicuous, high-dimensional, structure.'

It has always been a dream of computer scientists and engineers to make a computer recognize things that we recognize unconsciously [6]. The main objective of pattern recognition, therefore, is to make inferences from perceptual data. In pattern recognition we are concerned, amongst other things, with (From [26])

> 'the assignment of a physical object or event to one of several pre-specified categories'

To this end pattern recognition techniques make use of tools from statistics, probability, computational geometry, algebra, formal language theory, machine learning, signal processing, and algorithm design. Pattern recognition is thus of central importance to artificial intelligence and has far-reaching applications in engineering, science, medicine, and business. The advances made in the last half century, such as speech recognition, allow humans to interact more freely and effectively with humans and the natural world.

Historically, the two major approaches to pattern recognition were the statistical (or decision theoretic) and the syntactic (or symbolic) approaches. The advent of

artificial neural networks in the 1980*s* introduced yet another vector space based, connectionist, approach. A description and discussion of the three approaches is found in [108]. We conclude this of-necessity brief section by identifying the *main* issues in pattern recognition:

Classification The problem of classification is basically that of assigning a given object to a predefined class (concept, or category). This is something that humans do very well. In order for a machine to perform satisfactory classification it must have, in some form, a representation "spaces" for objects and classes of objects as well as a computational procedure for determining whether a given object belong to the class/category or not.

Class Representation and Structure If a machine is to perform correct classification it must, in some form, be able to store a representation of each class of objects. What is the best way to represent classes? (see Chapter 8). How will the latter choice affect the noise handling capabilities? This choice of class representation is related to the choice of object representation "space". One must be careful to distinguish between real-world objects such as faces, sounds, or smells and their numeric or symbolic encoding. For artificial neural networks, for example, the class learned during the training procedure is represented within the given "architecture" by the final set of weights. The objects themselves are encoded as vectors over the reals. In case of syntactic pattern recognition, the objects are represented as strings over some alphabet, while the class is represented as the corresponding grammar, i.e. as a set of production rules (involving, of course, some secondary alphabet) [33].

2.6 Overview of Computational Learning Theory (CoLT)

The process of learning, whether by animal or machine, has long fascinated philosophers, psychologists and, more recently, cognitive scientists and computer scientists. In fact, the ability to learn is considered by some to be of the most fundamental attributes of intelligent behaviour [85]. Yet it is difficult to find a precise definition of what learning is. The Collins English Dictionary defines learning as '*the modification of behaviour though practice, training, and experience*'. This is perhaps too informal. The area of *Machine Learning* concerns itself with answering the question *What is Learning?*, as well as with the development of powerful and efficient algorithms that learn from examples. In this section we shall not concern ourselves with machine learning algorithms or techniques. The aim is to discuss the main issues in the theoretical aspects of machine learning and present the main results relevant to this thesis. In particular, machine learning theory is concerned with issues such as: (from [88])

- What is learnable? and, more importantly, what is not?,
- Under which conditions is a particular learning algorithm assured of learning successfully?
- How many training examples are required to learn a class (or concept)?,
- How does learning performance vary with the number of training examples provided?,
- When and how can prior knowledge held by the learner guide the process of generalizing from examples? Can prior knowledge be helpful?,

- How can the learner automatically alter its representation to improve its ability to represent and learn the target function (or class)?,
- What is the best strategy for choosing a useful training experience, and how does the choice of this strategy alter the complexity of the learning problem?,
- Learning in the presence of 'noise', and
- The time and space complexity of learning.

Various models and theories that deal with the above issues have been proposed. The area is today called CoLT — short for *Computational Learning Theory*. CoLT is not concerned with learning algorithms *per se* but rather with the issues mentioned above.

2.6.1 What is learning after all?

The most logical starting point is to define the learning problem. The main machine learning textbooks and, for that matter, also the main CoLT textbooks, do not seem to give much importance to an exact and formal definition of what learning really is. Most CoLT textbooks pose the learning problem as a *function approximation* problem. The problem with using this framework, and a treatment of this issue is outside the scope of this thesis, is that it does not give a satisfactory answer to the question 'What is a class (or concept)?' To put it another way, *What does a learning algorithm (or process) learn?* Many in the machine learning community are quite comfortable with the assumption that a learning algorithm (or process) learns a membership (or characteristic) function[2]. In other words, the learning entities learns how to classify unknown instances as either belonging or not belonging to a given

[2]an *indicator* function in the case of *yes/no* classification

concept[3]. This tell us nothing about the concept itself — i.e. its *description* or *representation*. There is evidence, from experiments performed by researchers in the cognitive sciences [8], that humans learn more than just a classification function. It is widely held that when a human learns a concept a *representation* of that concept then exists in that person's brain. If a human has never seen a cat and is shown a number of cats as training examples, the human then learns to identify any animal as being or not being a cat but also, and significantly, the human can *describe* cats in terms of their attributes. The author distinctly remembers seeing a pencil drawing of a rhinoceros drawn by an 18^{th} century German artist. What was extraordinary about this drawing, apart from the fact that it was a very good reproduction of a rhinoceros, was that the artist had allegedly never seen a rhinoceros but had made the drawing from a verbal description of a rhinoceros he got from a friend who had visited Africa. Humans can describe and communicate what they learn and intelligent activity (including language and thought) appears to involve the manipulation of concepts (or classes) in the mind [31]. It is the author's belief that a theory of machine (or animal) learning must address the issues of the formation, learning, and more importantly, the *representation* of concepts[4]. It is clear that humans do more than just learn concepts. They can combine concepts (whether learned or innate) together to form new ones, make plausible inferences from the concepts they have learned, modify and apply each concept in different ways according to the context, and use concepts to predict future action or events. It is for these reasons that the author and his colleagues of the *Machine Learning Group* at UNB believe that a theory of machine or animal learning should start with a theory of concepts and categories (perhaps in conjunction with the cognitive science community). Once this has been accomplished, then one can concentrate on the development of a formal

[3]in the case of a *fuzzy* concept, a degree of membership is returned

[4]in this thesis the terms *concept* and *class* are used interchangeably

theory of learning. The author's colleagues of the *Machine Learning Group* at the University of New Brunswick, at the time this thesis was being written, were working this topic.

Concept learning can, informally, be characterized as follows (from [88]):

Given:

- a domain (or instance space) $\mathcal{X}$,
- a set of classes (or concepts) $\mathcal{C} \subset \mathcal{P}(\mathcal{X})$,
- a finite subset c^+ of some $c \in \mathcal{C}$, called the *positive training set*, and
- a finite subset c^- of objects not in c, called the *negative training set*.

For each concept $c \in \mathcal{C}$, let I_c denote the *indicator function* of c, i.e.

$$\forall x \in \mathcal{X},\ I_c(x) = 1 \text{ if } x \in c \text{ and } I_c(x) = 0 \text{ if } x \notin c.$$

Thus I_c is a boolean-valued function over the set of instances. Also, let T denote the union of the positive and negative training examples, i.e. $T = c^+ \cup c^-$. It is also assumed that T is drawn at random from $\mathcal{X}$ according to some general probability distribution $\mathcal{D}$. The **learning task** is then to find a learning algorithm L such that, $\forall c \in \mathcal{C}$ and on input D, L outputs a hypothesis[5] $h(x)$ such that $h(x) = I_c(x)$.

It must be emphasized that there is a fundamental distinction between the set $\mathcal{X}$ and the symbolic or numeric encoding of its elements. Suppose, as an example, that we want to learn the concept *cat*. In this case $\mathcal{X}$ could be the set of all animals. Clearly, any learning algorithm requires that the animals are encoded as elements of some mathematical structure (whether numeric or symbolic), i.e. as vectors, strings, graphs, bitmaps, etc. An important issue, which is very often overlooked or even

[5] a boolean-valued function over $\mathcal{X}$

ignored, is whether the particular encoding chosen affects learning. This issue is discussed in Chapter 8. Also, many in the CoLT community assume only that the class or concept to be learned is simply a computable set under the particular encoding of $\mathcal{X}$. In our example, if the set of all animals $\mathcal{X}$ is encoded as strings over some finite alphabet Σ, then the set of all cats (encoded as strings) is a computable language over Σ, i.e. it has a finite description. This is a basic but important assumption since if, under a particular encoding, the class of cats is not computable, then learning the class is then clearly impossible. No other assumptions are made about the structure of the class. It turns out that, as posed above, the learning problem is, in general, unsolvable. This is, trivially, because any infinite (computable) class cannot be characterized by any of its finite subsets. In other words no finite subset of any infinite set has enough 'information' that allows us to infer with absolute certainty to which set it belongs. For example, given a finite set of strings, S, that we are told belongs to some infinite language L, we cannot inductively infer L from S with absolute certainty. This is because S is also a subset of infinitely many other languages apart from L. This issue was addressed by E.M. Gold in the 70's.

2.6.2 Gold's results

E.M. Gold was the first to examine (symbolic) learnability formally. In his now seminal 1976 paper, *Language Identification in the Limit*, Gold proved a number of important results about learning infinite sequences of strings with unbounded length. Gold was concerned mainly with what was learnable 'in the limit'. Gold restricted his attention to learning languages from finite sets of strings. He gave no importance to the finiteness or even the bounds of the data. Gold published some beautiful and deep results about this matter but an in-depth discussion of these is outside the scope of this thesis. Gold considered only grammars in a class $\mathcal{A}$ which he called

'admissible'. The class $\mathcal{A}$ is defined as follows:

Definition 2.27 (Admissible Classes of Grammars).
Let Σ be a finite alphabet. A class of grammars $\mathcal{A}$ is called ***admissible*** *if:*

(a) $\mathcal{A}$ is denumerable, and

(b) for any $s \in \Sigma^*$ and for any $G \in \mathcal{A}$, it is decidable whether or not $s \in L(G)$.

Note that the Chomsky classes of regular, context-free, and context-sensitive grammars are all admissible. Gold's main results can be paraphrased as follows:

- No admissible class of grammars that generate a superfinite class of languages is identifiable in the limit from positive training examples only, and
- Any admissible class of grammars that generate a superfinite class of languages is identifiable in the limit from positive and negative training examples.

A superfinite class of languages is one that includes at least one infinite language. In this context, 'identification in the limit' means, informally, that as the number of the training examples tends to infinity, we can identify with absolutely certainty the language from which the examples are drawn. One implication of Gold's results is that we can never be *absolutely sure* that we have learned the right class — unless, of course, we keep adding training examples and let the size of the training set tend to infinity. We might, of course, happen to hit on the right hypothesis, i.e. when $h(x) = I_c(x)$, but we can never be really be absolutely sure. The only thing we can ensure is that of finding a hypothesis $h(x)$ that is *consistent* with the training set D, i.e.

$$\forall x \in D, h(x) = 1 \text{ if } x \in c^+ \text{ and } h(x) = 0 \text{ if } x \in c^-,$$

and then hoping that $h(x)$ approximates I_c over the unseen instances of $\mathcal{X}$. Of course, the more 'representative' (of the class) our training data is the more confident we can feel that our learned hypothesis *approximates* the class c. But how can we be sure? In general, we cannot. We just have to make an assumption. This assumption is called the *Inductive Learning Hypothesis*.

2.6.3 The Inductive Learning Hypothesis

We noted that the main objective of learning is to find a hypothesis that is identical to the indicator function, I_c, of the concept c over the entire set of instances $\mathcal{X}$. However, the only information[6] we have about the concept c is the finite set of training examples. But, as was argued in the previous section, we can never be sure that our learned hypothesis is identical to the indicator function of c. The best we can hope to achieve, therefore, is to find a hypothesis consistent with the training set D and then assume that it approximates well (what exactly is meant by 'approximates' is explained later) the concept c over the other unseen instances. This assumption is now stated:

Inductive Learning Hypothesis (ILH) *Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over the unobserved examples.* From [88].

The above definition of the ILH is rather informal. It is not exactly clear what is meant by 'sufficiently large set of training examples'. Having said that, the main point here is that if a hypothesis is consistent with the training examples then *one can assume* that it will also do well with (i.e. classify correctly) the other unseen instances from $\mathcal{X}$. The inductive learning hypothesis assumes that the training and

[6] we might have other information about the concept c but the argument will still hold

test examples are drawn from the same general distribution $\mathcal{D}$. This is a fundamentally an unprovable hypothesis unless addition assumptions are made about the target concept [90]. The ILH is considered to be a central and critical assumption in CoLT. Before proceeding to discuss the PAC (*Probably Approximately Correct*) Learning model let us first outline a number of problems and issues that arise from our formulation of the learning problem:

- Absolutely correct learning is impossible. This follows from Gold's results.
- For every finite training set D there may be many (possibly infinite) hypothesis consistent with D. Which one does one choose?
- How can one choose the best hypothesis, i.e. the hypothesis that best approximates the concept being learned?
- Under which conditions does the Inductive Learning Hypothesis hold?

The PAC model, which is discussed next, was designed to address (and hopefully overcome) these problems.

2.6.4 Probably Approximately Correct Learning

Although there is no guarantee that the learning procedure L will find the *right* hypothesis, i.e. $h = c$ [7], we can, at least, insist that the learned hypothesis is *close to*, or *approximates* the concept to be learned. If the set of instances, $\mathcal{X}$, is finite, we can measure how 'close' h is to the concept c by taking the symmetric difference. In practice, $\mathcal{X}$ is infinite and therefore this method is not valid. Informally, the true error of h is precisely the error rate we expect when applying h to future unseen instances of $\mathcal{X}$ drawn according to the probability distribution $\mathcal{D}$. This error is now defined formally (from [88]).

[7] for convenience let $h = \{x \in \mathcal{X} \mid h(x) = 1\}$

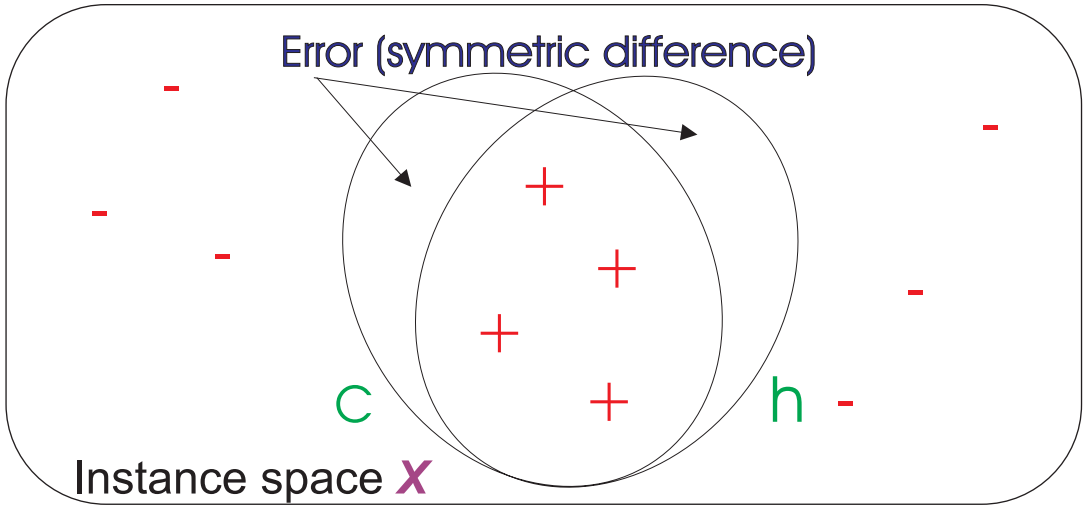


Figure 2.4: The error of the hypothesis h with respect to the concept c and the distribution $\mathcal{D}$.

Definition 2.28. *The **true error**, which is denoted by $error_{\mathcal{D}}(h)$, of the hypothesis h with respect to the target concept c and the distribution $\mathcal{D}$ is the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.*

$$error_{\mathcal{D}}(h) = Pr_{x \in \mathcal{D}}[c(x) \neq h(x)]$$

Figure 2.4 shows this error in graphical form. The error of the hypothesis h with respect to the concept c and the distribution $\mathcal{D}$ is the probability that a randomly chosen instance of $\mathcal{X}$ falls into the region where h and c disagree.

2.6.5 The PAC Learning Model

In contrast to Gold, Valiant took the view that it was more productive to examine what was feasibly learnable rather than what was absolutely learnable. The main differences between Gold's paradigm and Valiant's PAC model are:

- PAC considers only fixed-length strings, and so requires only a finite training set.

- It has a polynomial bound on the time required to accomplish learning (the polynomial bound is not on the number of training examples though).
- The form of concept description to be learned needs to be specified.

We can now proceed to define PAC learning.

Definition 2.29. *Consider a concept class C defined over a set of instances $\mathcal{X}$ of length n and a learner L using hypothesis space H. C is PAC-learnable by L using H if for all $c \in C$, distributions $\mathcal{D}$ over $\mathcal{X}$, an ϵ such that $0 < \epsilon < 1/2$, and δ such that $0 < \delta < 1/2$, learner L will with probability at least (1-δ) output a hypothesis $h \in H$ such that $error_{\mathcal{D}}(h) \leq \epsilon$, in time (and space) polynomial in $1/\epsilon$, $1/\delta$, n, and size(c)*[8].

Notes to Definition 2.29.

- The learner L must, with arbitrary high probability (1-δ), output a hypothesis having arbitrary low error (ϵ).
- It must do so efficiently — in time polynomial in $1/\epsilon$, $1/\delta$, n, and size(c).
- PAC-learnability depends on the number of training examples required by the learner. The smaller ϵ and δ are, the more training examples are required. As ϵ and δ approach 0, the number of training examples required tends to the cardinality of the concept c.
- It is interesting to note that the time and space complexity of PAC-learning is not expressed as a function of the number of training examples required but of $1/\epsilon$, $1/\delta$, n, and $size(c)$. This means that if we want a small error with high probability we require very large training sets. What would really be

[8] one may assume the Kolmogorov complexity of c

better is if the number of training examples is polynomial to the size (structural complexity) of the class. It must be noted, however, that, if the hypothesis space is finite, then the number of training examples required is logarithmic to the cardinality of the hypothesis space and also that it is possible to prove that every concept is polynomially PAC-learnable [90].

Buntine, amongst others, criticized the PAC model on a number of grounds. In his paper *A Critique of the Valiant Model* [17] he explains that the PAC model does not use either common sense or practical experience in a number of ways. It uses worst-case rather than average-case analysis and does not accommodate preferences about hypothesis.

2.6.6 Inductive Bias

A learning algorithm considers a set of hypothesis during learning. This is called the hypothesis space of the algorithm. Questions that arise are:

- What if the hypothesis required is not in hypothesis space?
- How can we ensure that the hypothesis we seek is in the hypothesis space considered by the learning algorithm?
- If there are more than one hypothesis consistent with the training set, which one do we choose?

The answer to the first two questions is that we can never be absolutely sure that the hypothesis we seek lies in the hypothesis space considered by the algorithm. Of course, we could design our learning algorithm to consider the space of every possible hypothesis but this is clearly not feasible since the space would be too large. In practice, some given information about the concept to be learned allows us to design

an algorithm that considers a hypothesis space that contains the required hypothesis. The third question is more thorny. We cannot really have a foolproof way for choosing one out of many consistent hypotheses that is 'right' in the sense that we can be sure that it best approximates the required hypothesis. As always, we have to make some assumptions. These assumptions are called the *Inductive Bias* of the learning algorithm. The inductive preference bias can be loosely defined as *the minimal set of assumptions made by any learning algorithm regarding the:*

- **Inductive Language Bias**: *the choice of the hypothesis space (The Inductive Language Bias), and*
- **Inductive Preference Bias**: *any preference of one hypothesis consistent with the training set over another (The Inductive Preference Bias).*

Note that the choice of the hypothesis space also includes the choice of concept representation. For instance, if we are learning a regular language we can search in the space of DFAs, or the space of regular grammars, etc. It has been stated many times in CoLT literature that it is totally futile to have an unbiased learner. In this thesis, unless otherwise stated, when we say 'the bias' or 'the inductive bias' we mean the inductive preference bias. CoLT maintains that a fundamental property of inductive inference is:

a learner that makes no a priori assumptions regarding the identity of the target concepts has no rational basis for classifying any unseen instances.

Many in CoLT consider concept learning as searching the chosen hypothesis space for one (or more) hypothesis consistent with the training set. Clearly, the method chosen to enumerate the hypothesis space during the search (assuming the hypothesis space is countable) dictates which hypothesis will be chosen over other - assuming

the learning algorithm stops when it finds a hypothesis consistent with the training examples.

2.6.7 Occam's Razor

One of the popular inductive preference biases employed is that known as 'Occam's Razor'. William of Occam was a British monk, logician, and philosopher who is alleged to have said '*entities should not be multiplied beyond what is necessary*'. This can be paraphrased as *if one has to choose between competing hypotheses, one should always choose the simplest*. Of course, what is the 'simplest' depends on the context and the measure of simplicity used. This bias works in most cases. For example, if we are learning a regular language we might use the bias that chooses a DFA with the least number of states, or cycles, or perhaps the regular grammar with the least number of rules.

2.6.8 Other Biases

Other preference biases have been proposed and employed in various learning algorithms. An example is the so-called *Epicurus's Bias*. This is based on the *Epicurus principle of multiple explanations* which essentially says that if more than one theory (or hypothesis) is consistent with the observations, then keep all theories. This is the bias employed by Mitchell's Version Space inductive learning algorithm. It must be said, however, that this bias is impractical in many cases since it turns out that, very often, the number of hypotheses consistent with the training examples is exponential to the size of the training set.

2.7 Grammatical Inference

Grammatical Inference (GI) addresses the following problem: given a set of strings that belong to some unknown formal language L, and possibly a set of strings that do not belong to L, can a learning machine or agent infer L — i.e output a structural description of L?. Apart from the obvious theoretical interest, this problem is also of practical importance. GI is a integral part of syntactic and structural pattern recognition and it has yielded many useful and interesting tools and methodologies for structural learning. GI, it turns out, is by no means a simple problem. The first grammatical inference algorithms date back to the sixties and are actually older than the concept of 'machine learning' itself although today GI is considered as a sub-area of machine learning. This section discusses the GI problem, presents some rather depressing hardness and undecidability results, and gives a brief synoptic survey of the more successful GI techniques.

2.7.1 The Grammatical Inference Problem

Grammatical inference is an instance of inductive inference which can be described as the task of discovering syntactic structures from a corpus of training examples. In this case the corpus of training examples is usually a finite set of strings, T, over some fixed, finite, alphabet Σ which includes strings from some unknown language L and possibly other strings that do not belong to L. The task of the learning machine or agent is to infer L from T — i.e. to discover the rewrite rules that describe L. The output is either a grammar for L or some other structural description of L such as an automaton, expression, etc. Grammatical inference encompasses not only the theory but also the methods and techniques for learning grammars from sets of training examples.

Although the name *grammatical inference* suggests that it is a grammar that

is the output of the learning algorithm, this need not be the case — any other structural description of a language such as automata, expression, TS description, etc, can be used. The theory and techniques of grammatical inference have also been applied to the task of learning languages of other structures such as 2-dimensional strings (array grammars), trees, and graphs [86]. The most obvious (and traditional) field of application of grammatical inference has been Syntactic Pattern Recognition [33, 108]. In the last two decades, however, grammatical inference techniques have also been applied to applications such as Speech Recognition, Natural Language Processing, Biological Sequence Analysis, Gene Analysis, Image Processing, Machine Vision, and Cryptography to name but a few [132].

Recall from Section 2.6 that any finite sample of strings of an infinite language does not uniquely define (characterize) the language since a finite set of strings may be associated with an infinite number of languages. This observation alone means that our informal definition of the GI problem is impractical since inferring, with absolute certainty, a grammar of a language from a finite set of samples from the same language is, in theory, impossible. This rather obvious fact led early researchers in grammatical inference and computational learning theory (CoLT) to propose somewhat more modest and constrained formalizations of the objectives of grammatical inference. Given a set, $C = C^+ \cup C^-$, of training examples, where C^+ contains strings belonging to some unknown language L, and C^- contains strings not in L, the learning machine or agent has to output a grammar G (or some other structural description of a language) such that $L(G)$ *approximates* L — given some definition of approximation as discussed in the previous section on CoLT. Another requirement is that the set of positive training examples, C^+, must be *structurally complete*, i.e. it contains enough information to allow us to successfully approximate L. A formal definition of completeness is the following:

Definition 2.30. From [16, page 254]

A sample I is said to be structurally complete with respect to a grammar G over an alphabet Σ if

(a) $I \subset L(G)$,

(b) $\text{alph(I)} = \Sigma$ *(i.e. all the letters in* Σ *are contained in* I*), and*

(c) every rule of G *is used at least once in the generation of the strings of* I. □

Definition 2.31. From [16, page 254]

A sample I is said to be structurally complete with respect to a DFA A over an alphabet Σ if

(a) $I \subset L(A)$,

(b) $\text{alph(I)} = \Sigma$ *(i.e. all the letters in* Σ *are contained in* I*),*

(c) every transition of A *is used in at least one string* $s \in I$,

(d) every accepting state is used to accept at least one string $s \in I$. □

The definition of structural completeness often depends on the class of languages under consideration, the representation used to describe the languages, and the inference algorithm. The Valletta algorithm, described in Chapter 5, uses a different definition of structural completeness.

The Theoretical and Practical Complexity of Grammatical Inference

Although GI dates back to the sixties and the area is arguably older than machine learning itself there are still no universal, efficient algorithms that solve the problem. This, we feel, is due more to the inherent hardness and complexity of the GI problem rather on the amount of effort made over the last four decades. The somewhat

depressing hardness results have discouraged many researchers from working in this area. In a book published in 1990, Miclet [16] wrote:

> Historically, grammatical inference has always been closely related to pattern recognition, except for a pure theoretical branch, which is not really connected to algorithms. It is now clear that the interest of researchers in this domain is decreasing, the publications are less and less numerous; the years 1970-80 were the most fruitful, but since the early eighties, articles or congress communications on the topic have been rare.

The main negative theoretical evidence for learning DFAs can be summarized as follows: (from [29])

(a) Finding the smallest DFA consistent with a set of positive and negative training examples is, in general, NP-hard [4, 38].

(b) The minimal DFA consistency problem cannot be approximated within any polynomial of the size of the optimal solution [98].

(c) Approximate inference of finite automata from sparse labelled examples is NP-Hard if an adversary chooses both the target machine and the training set [70].

The results for context-free grammars are even worse [86, 76]. In this case even the ability to ask equivalence queries does not guarantee exact identification in polynomial time. One must emphasize, however, that these are general results. Most researchers have concentrated on finding the conditions under which the negative results can be overcome. The GI community is divided into those researchers who are interested in the theoretical possibility of learning a grammar and those who are mainly interested in developing algorithms for solving a particular practical problem. Of course, the latter should be aware of the results obtained by the former. Many GI algorithms have been developed over the last three decades. Many algorithms

restrict the learning domain, i.e. the class of formal languages that can be learned, and therefore, as Miclet noted in [86], these algorithms are difficult to compare. We conclude this section with a review of some of the techniques used in GI.

2.7.2 Some GI Techniques

One of the most popular techniques used for learning DFAs is the *state merging* technique. This technique was inspired by the procedure that is used to find the unique minimal DFA from any given DFA — the so-called *Moore optimization* procedure. It was Trakhtenbrot and Barzdin [127] who first proposed an $O(mn^2)$ state-merging algorithm for constructing the minimal DFA consistent with a structurally complete training set that contains both positive and negative training examples, where m is the size of the PTA, or prefix-tree automaton, built from the positive training examples and n is the size of the target DFA. Trakhtenbrot's and Barzdin's algorithm required, however, that the training set included all the strings of up to length l where l depends on the target DFA and is equal to $2n - 1$. It is clear that the number of training examples becomes prohibitively large as n grows. Many other state-merging algorithms were developed. Probably the most well-known is the state-merging algorithm that won the Abbadingo DFA learning competition in 1998. The Abbadingo DFA learning competition involved learning from both positive and negative training examples. The target DFAs, the training examples, and the testing strings were drawn from a uniform random distribution. The reader is referred to Appendix E for a description. The winning algorithm was developed by Ron Price [73] and is described as an evidence-driven state merging (EDSM) algorithm. This algorithm is considered to be the current state-of-the-art in DFA learning algorithms. State-merging algorithms work by first building a prefix-tree acceptor (PTA) for the positive training set. This is essentially a DFA, containing no cycles, that accepts only the strings

in C^+. A prefix tree acceptor for the strings bb, ba and aa is shown in Figure 2.5 below.

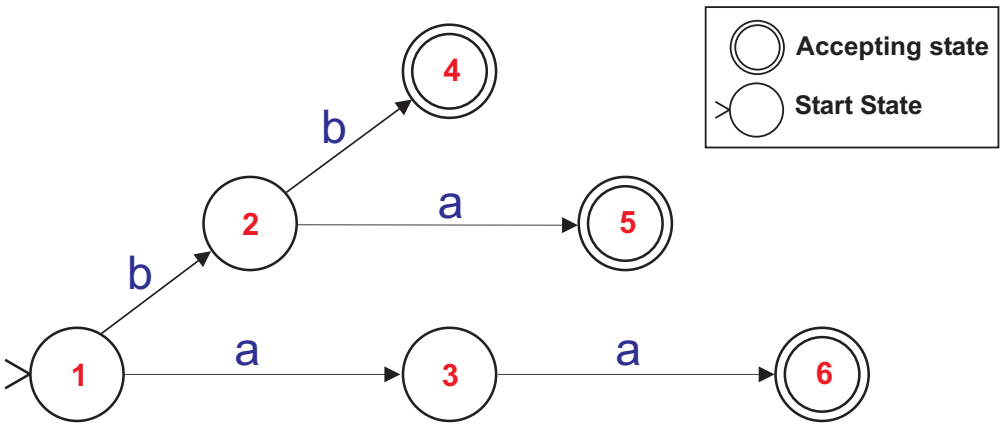


Figure 2.5: The Prefix Tree Acceptor for the strings bb, ba, and aa.

The algorithm then generalizes by defining an appropriate equivalence relation on the set of states and merging equivalent states — cycles are thus created. The equivalence relation is similar to that used in the Moore optimization procedure [115]. In essence, two states are considered equivalent if no suffix[9] leads from them to differing labels. Cycles are created only if they do not allow the resulting DFA to accept strings in C^-. The set of negative training examples is therefore necessary since it prevents over-generalization. The reader is referred to [73] which contains an exposition of how the algorithm works and many useful references.

[9] i.e. k-tail

Figure 2.6 below, uses a very simple example to show how state-merging basically works. It starts by building a PTA from C^+ and then merge states, creating cycles in the process. Price's EDSM algorithm employs the strategy that it is best to perform first those merges that are supported by the most *evidence* [73].

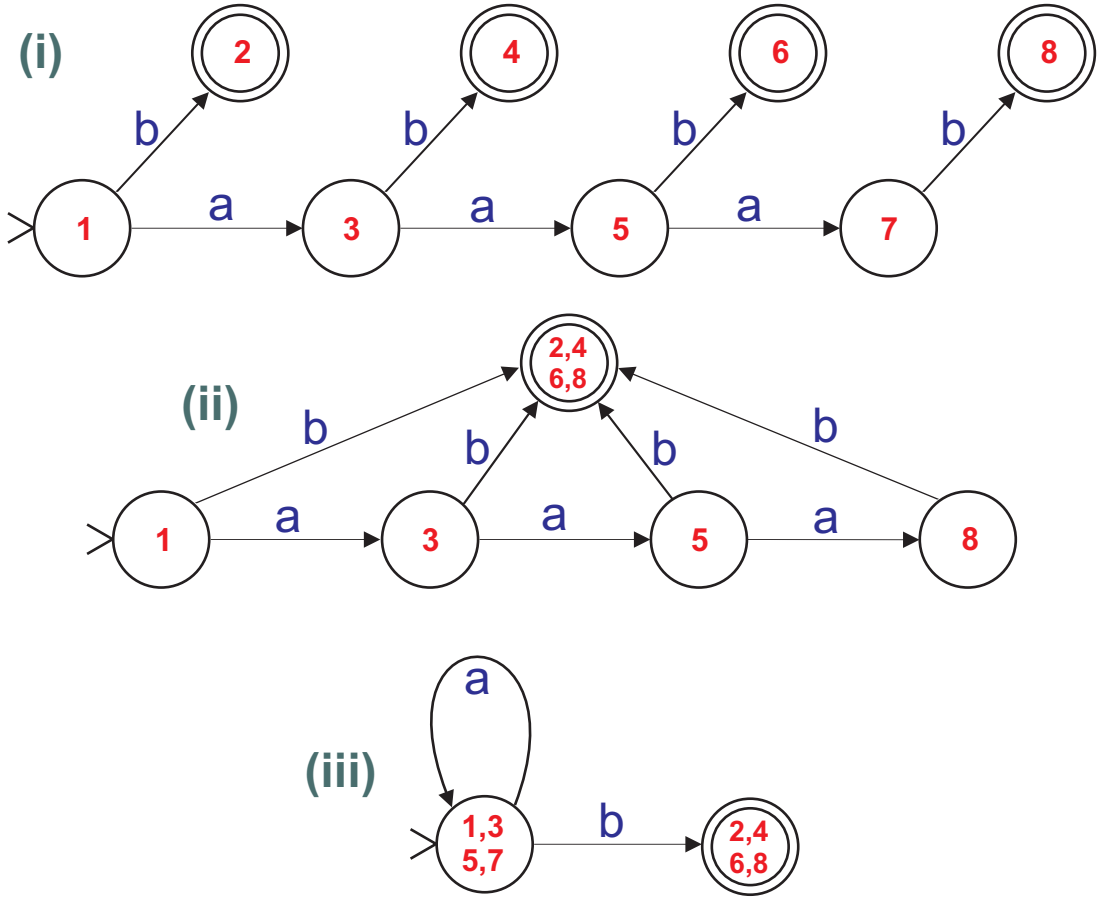


Figure 2.6: Learning DFAs through state merging

Figure 2.6 (i) shows the Prefix Tree Automaton (PTA) for the set of strings $S = \{b, ab, aab, aaab\}$. Note that $L(PTA) = S$. (ii) shows the DFA obtained by merging the states 2, 4, 6, and 8. This does not yield any generalization. (iii) shows the DFA obtained after merging states 1, 3, 5, and 7. This introduces a cycle in the DFA and therefore results in generalization.

Various other techniques for GI have been developed. References [86, 76] contain an excellent survey of the main methods developed over the last three decades. Miclet in [86] also discusses an extension of the classical, i.e. Chomsky, grammatical models to the case were the grammar's rewrite rules are associated with a 'probability' of rewriting. This gives us *stochastic* grammars and automata. These have been used extensively in pattern recognition [16, 33] and various methods have been developed for their inference. Stochastic grammars and automata have appealed to many researchers and practitioners in pattern recognition because they have some ability for handing noise. This point is discussed later on in Chapter 3.

2.8 String Edit Distances

In Chapter 1 we discussed the ETS inductive learning model and we saw that the notion of distance plays a central role. In fact, the ETS model uses distance;

(a) to classify an unknown object as belonging or not belonging to a class C by measuring its 'distance' to another object already known to belong to C,

(b) to partially define the class C itself (the *class description* of C is partially specified using a distance function), and also

(c) to direct the learning process.

This thesis investigates the application of the ETS model to the problem of grammatical inference — i.e. learning a language from a finite subset. In grammatical inference the set of objects, i.e. the instance space, is the set, Σ^*, of all strings over some alphabet Σ and the classes are the formal languages over Σ. It is therefore evident that in order to describe the classes and to test for class membership we clearly require an appropriate definition of the 'distance between two strings' and also an effective procedure for computing this distance. It turns out that there are many ways of computing the distance between two strings. In this section we shall discuss briefly the more popular 'traditional' methods used in many applications for computing such distance and we shall then argue why such methods are inadequate and, therefore, why it is necessary to develop new distance functions for strings.

Two methods that traditionally have been used to compute the *string edit distance* between two strings $s_1, s_2 \in \Sigma$, denoted by $d(s_1, s_2)$, are listed below [71].

String edit distance is defined to be the minimum number of character edit transformations (i.e. insertion, deletion, and substitution of single characters) that transform s_1 into s_2.

Weighted String Edit Distance is defined to be the minimum cost over all sequences of character edit transformations (as above) that transforms s_1 into s_2 (in the case when each edit transformation is assigned a *cost*, i.e. a positive real number).

Table 2.2 shows the edit transformations that transform the string $s_1 = abcb$ into the string $s_2 = acbc$. D_b denotes deletion of the character b and I_c denotes insertion of the character c. The string edit distance between s_1 and s_2 is therefore 2 since a minimum of 2 character insertions and/or deletion are required to transform *abcb* into *acbc*. In this example the set of character edit operations (rewrite rules), which we denote by O, is $\{a \leftrightarrow \varepsilon, b \leftrightarrow \varepsilon, c \leftrightarrow \varepsilon\}$.

a	b	c	b	I_c
a	D_b	c	b	c

Table 2.2: String Edit Distance between the strings *abcb* and *acbc*.

In the case of weighted string edit distance a *weight vector*, which we denote by ω, stores the weights of the character edit operations. We illustrate with an example.

Example 2.3. *Let $s_1 = aaaab$, $s_2 = baaaa$, the set of operations is $O = \{a \leftrightarrow \varepsilon, b \leftrightarrow \varepsilon\}$ and $\omega = \{0.1, 0.9\}$.* □

In Example 2.3 an insertion or deletion of the character a has a weight of 0.1 and an insertion or deletion of the character b has a weight of 0.9. The weighted string edit distance is 0.8 since the least edit cost is achieved by first deleting 4 a's from s_1 to obtain the string b and then inserting 4 a's to obtain *baaaa*. The reader should note that, in this case, the weighted string edit distance is not equal to the normal (unweighted) string edit distance since the least number of edit operations required to transform s_1 into s_2 is 2 (a deletion and an insertion of b). The above example also illustrates the fact that weighted string edit distance does not necessarily have

to use the same edit sequence (i.e. sequence of rewrite rules) that is used by normal (unweighted) string edit distance. String edit distance and weighted string edit distance are usually called *Levensthein distance* and *weighted Levensthein distance* respectively after V.I. Levensthein [77]. Levensthein is generally recognized as the first to define string edit distance and to propose a method for its computation.

The string edit distance problem is an extensively studied topic. This is undoubtedly due to the fact that string edit distance has numerous applications including: file comparison [64], spelling correction [58], searching for similarities amongst biosequences [91], speech processing [105], and error correcting codes [96]. The reader is referred to [57, 106] for a comprehensive list of applications, references, and an exposition.

Various algorithms for both Levensthein distance and weighted Levensthein distance have been developed since Levensthein's seminal paper in 1965. Most use dynamic programming techniques and run in $O(mn)$ time, where m and n are the lengths of the two strings, although some are asymptotically faster. One of the most widely used algorithms for string edit distance is the dynamic programming algorithm proposed by Wagner and Fischer in [135]. Before describing this algorithm we first show that the string edit distance problem can be described by a recurrence relation.

Notation 2.32. *Let $s_1, s_2 \in \Sigma^*$ be two strings over some finite alphabet Σ. $D(i, j)$ denotes the string edit distance between $s_1^{(i)}$ and $s_2^{(j)}$.* □

$D(i, j)$, therefore, is precisely the distance between the first i characters of s_1 and the first j characters of s_2. The following recurrence relation defines the string edit distance problem (Proof in [57]).

- $D(0, 0) = 0$,
- $D(i, 0) = i$,

- $D(0, j) = j$, and
- $D(i, j) = min\{D(i-1, j) + w(\varepsilon, s[i]), D(i, j-1) + w(s[j], \varepsilon), D(i-i, j-1) + w(s[i], s[j])\}$,

where $s[i]$ and $s[j]$ are the ith and jth characters of s_1 and s_2 respectively and $w(a, b)$ is the cost (or weight) of substituting the character a by the character b. Computing the string edit distance between s_1 and s_2 is therefore equivalent to filling in the values in Table 2.3. This is accomplished by Algorithm 2.8.1 — a simple, quadratic-time dynamic-programming algorithm proposed by Wagner and Fischer in [135]. The algorithm is very short and elegant and can easily be parallelized. It consists of two preprocessing loops that fill in the first row and the first column of the distance matrix and two nested loops that compute the rest of the matrix.

The algorithm runs in $o(m, n)$ time where $m = |s_1|$ and $n = |s_2|$. Table 2.4 shows the distance matrix after termination of the algorithm. The weights used were 1 for insertion/deletion (of any character) and 2 for substitution (of any two characters). Using this weight vector the algorithm finds the minimum number of edit operations that transform s_1 into s_2.

	j	0	1	2	3	4	5
i		ε	a	c	b	c	b
0	ε						
1	a						
2	b						
3	c						
4	a						

Table 2.3: Empty Distance Matrix for the strings *acbcb* and *abca*.

Various extensions to the Wagner and Fischer algorithm have been proposed including one algorithm that uses only $O(n)$ space. The reader is referred to [57] and [114] for a discussion.

Algorithm 2.8.1: LEVDIST(x, y)

comment: Weighted String-Edit Distance Computation

comment: x and y are the input strings

$n \leftarrow len(x)$
$m \leftarrow len(y)$
comment: $D_{n,m}$ is the distance matrix

$D_{0,0} \leftarrow 0$
comment: Initialize first row and first column

for $i \leftarrow 1$ **to** m
  do $D_{i,0} \leftarrow D_{i-1,0} + w(x(i), \varepsilon)$
for $j \leftarrow 1$ **to** n
  do $D_{0,j} \leftarrow D_{0,j-1} + w(\varepsilon, y(j))$
comment: Compute the rest of the distance matrix

for $i \leftarrow 1$ **to** m
do { **for** $j \leftarrow 1$ **to** n
do { $deletioncost \leftarrow D_{i-1,j} + w(x(i), \varepsilon)$
$insertioncost \leftarrow D_{i,j-1} + w(\varepsilon, y(j))$
$subscost \leftarrow D_{i-1,j-1} + w(x(i), y(j))$
$D_{i,j} \leftarrow min(deletioncost, insertioncost, subscost)$

	j	0	1	2	3	4	5
i		ε	a	c	b	c	b
0	ε	0	1	2	3	4	5
1	a	1	0	1	2	3	4
2	b	2	1	2	1	2	3
3	c	3	2	1	2	1	2
4	a	4	3	2	2	2	3

Table 2.4: Completed Distance Matrix for the strings *acbcb* and *abca*.

Weighted or unweighted Levensthein distance, in the case when only single-character insertion, deletion, and substitution operations are allowed, can therefore be computed with reasonable efficiency and little programming effort. Some researchers have considered other types of operations such as *block moves* [124] and *transpositions* (i.e. swapping adjacent characters) [95].

The string edit distances we have considered above have one important characteristic in common — the *only* operations allowed are single-character insertion, deletion, and substitution operations. This type of distance functions is not adequate for the class description of all but the most simple languages. In the next chapter we shall be looking at how string edit distances can be used to describe formal languages and the fundamental inadequacy of this type of string distances will become apparent. What is somewhat surprising is that an extensive search of the literature on string edit distances revealed that nobody, apparently, considered string edit distance where multi-character edit operations are allowed. This is somewhat surprising since, as Fu pointed out in [33], many classes (in syntactic pattern recognition) cannot be described using only single-character edit operations. Also, in molecular biology, it is widely assumed that neighbouring characters in biological sequences are highly correlated and in proteins, in particular, characters in different parts of the amino-acid sequence are correlated. In spite of this, many researchers in bioinformatics still use Levensthein distance in applications such as protein sequence classification. The reader should note that single-character edit operations are, in essence, *context independent*. The single-character edit operation $a \leftrightarrow \varepsilon$ allows the character a to be inserted (or deleted) *anywhere* in the string. On the other hand, a multi-character edit operation such as $aba \leftrightarrow aa$ allows for the insertion (or deletion) of the character b only between two occurrences of the character a. Clearly, systems that use the latter type of operations have more *descriptive* (or *expressive*) power.

The apparent lack of interest and effort in the development of distance functions that use multi-character edit operations may be due to the following reasons:

(a) All the applications of string edit distance that the author came across do not involve the class description and learning of formal languages. In particular, they do not involve learning a class description that is defined in terms of the

edit operations. This idea is central to the ETS inductive learning model. It may be the case that most researchers have no use for such operations, or that perhaps they have not yet discovered the power of multi-character operations.

(b) It can also be the case that multi-character string edit operations are avoided because of the perceived intractability of computing string edit distance when using such operations. The theory of string-rewriting systems (see Section 2.4) shows that the problem of transforming one string to another using multi-character edit operations is, in general, undecidable and becomes decidable, and therefore computable, only if certain restrictions are imposed. In fact, it appears that, even after imposing certain restrictions on the type of operations allowed in order to make the word problem decidable, computing string edit distance when allowing for multi-character operations may well be NP-Hard.

In spite of the above, in the next Chapter we will demonstrate the usefulness of multi-character edit operations and the associated distance functions for the purpose of specifying certain classes of formal languages. We shall introduce a new string edit distance called *evolutionary distance (EvD)* that will be used to specify and learn *kernel languages* — a sub-class of the regular languages.

In his Masters thesis Nigam [92] proposed an ETS learning algorithm that used a polynomial-time dynamic-programming string edit distance algorithm that allowed weighted multi-character insertion and deletion edit operations which he called *macro operations*. This string distance was called *Generalized Levensthein Distance (GLD)* and had been proposed by Goldfarb and Santoso [107]. GLD did allow for the learning of certain classes of kernel languages but it had a number of important limitations. In spite of its limitations, GLD performed well on the class of languages considered by Nigam in his thesis. GLD runs in $O(nm)$ where n and m are the lengths of the two strings. The GLD algorithm differs from conventional Levensthein distance in that

it uses two additional data structures — the so-called *match matrices*. Each match matrix, one for each string, is used to store the position that each feature ends. This is illustrated in Figure 2.7 below. For example, the cell 4, 3 in the top match matrix contains a marker symbol since the feature *ab* (shown in red) ends at position 3 in the string *cabccbab*.

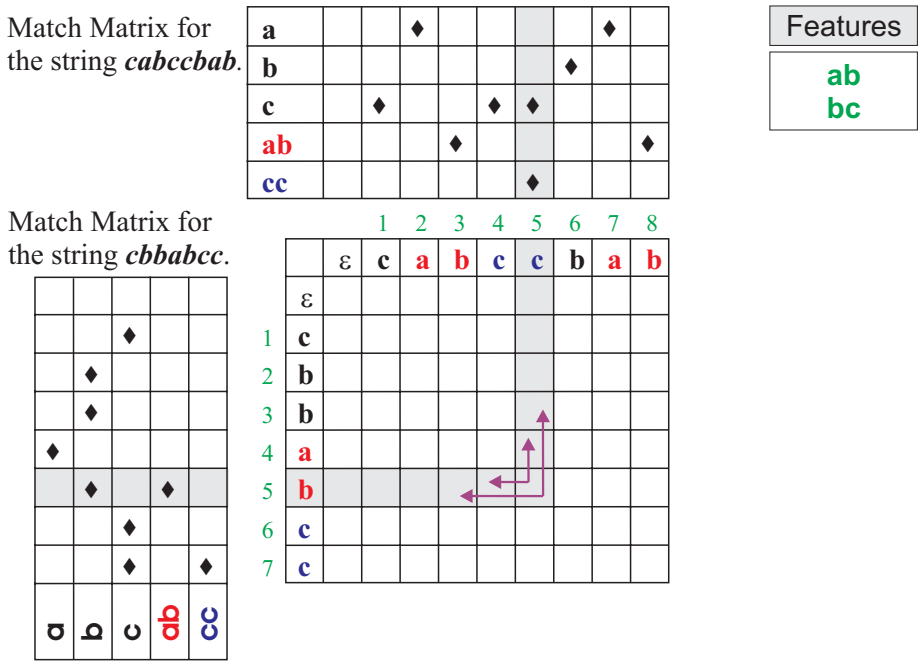


Figure 2.7: String distance computation using *GLD*.

The match matrices are constructed in a pre-processing stage and are used by the GLD algorithm when the distance matrix is being filled. For example, when computing the value of $D_{5,5}$, the algorithm queries the match matrices and discovers that feature *cc* (in blue) ends in position 5 of the string *cabccbab* and feature *ab* (in red) ends in position 5 of the string *cbbabcc*. The algorithm therefore computes the value in $D_{5,5}$ by finding the minimum of:

(a) $D_{5,4}$ plus the cost of inserting a c,

(b) $D_{4,5}$ plus the cost of deleting a b,

(c) $D_{5,3}$ plus the cost of inserting the feature cc,

(d) $D_{3,5}$ plus the cost of deleting the feature ab,

(e) $D_{4,4}$ in the case of a match.

A detailed explanation of how GLD works can be found in [107]. As already stated, GLD has a number of important limitations. In Chapter 3, when we discuss kernel languages, we shall see that any useful string edit distance function must be a pre-metric or a metric, i.e. the triangle inequality must hold. We will see that, in the presence of noise, the triangle inequality property plays a critical role and if the distance function violates this property we will not be able to describe the class (language). Also, testing for class membership becomes impossible. GLD is not a metric and not even a pre-metric but a pseudo-metric. In Chapter 4, where we discuss the GSN algorithm used by Nigam in his Masters thesis, we shall show that GLD, in some cases, does not return the minimum cost edit distance and also that, in general, it is not suitable for the class description of kernel languages.

This section concludes by fixing some miscellaneous (but useful!) notation and making a number of observations.

2.8.1 Notes and Additional Notation

Notation 2.33. *Let O be a set of string edit operations (single or multi-character), and ω be a weight vector for O. We denote by $\delta^{\omega}_{\mathbf{O}}$ to be distance function associated with O and ω. We call δ^{ω}_{O} the distance function* ***induced*** *by O and ω.* □

Note that we are not specifying how the distance is computed. We assume that the method of computing the distance between two strings is fixed. If, for example, we

fix the method for computing string edit distance to be the minimum cost over all edit sequences that transform one string into another, then, for any strings s_1 and s_2, $\delta_O^\omega(s_1, s_2)$ is the minimum cost over all edit sequences that transform s_1 into s_2 where the set of edit operations is O and the associated weight vector is ω.

Unweighted Levensthein distance is a special case of weighted Levensthein distance since if we assign a weight of 1 to each single-character deletion/insertion and a weight of 2 to each single-character substitution, $\delta_O^\omega()$ returns the minimum number of edit operations required to transform one string into another.

String edit distance functions are often metrics but not always necessarily so. If the set of operations O contains at least one operation with a zero weight then the corresponding distance function cannot be a metric since it would be possible to transform one string into another, non-identical, string using only zero-weighted operations. The distance between the two non-identical strings would then be zero. For example, if $s_1 = ab$, $s_2 = ba$, the set of operations is $O = \{a \leftrightarrow \varepsilon, b \leftrightarrow \varepsilon\}$ and $\omega = \{0.0, 1.0\}$, then s_1 can be transformed into s_2 using only two applications of the zero-weighted operation $a \leftrightarrow \varepsilon$. In this case, δ_O^ω is a pre-metric.

Definition 2.34 (Equivalent Distance Functions). *Let O be a set of edit operations and let ω and ω' be two weight vectors. Then δ_O^ω and $\delta_O^{\omega'}$ are called* ***equivalent*** *if, for any $s_,s_2 \in \Sigma^*$, both $\delta_O^\omega(s_1, s_2)$ and $\delta_O^{\omega'}(s_1, s_2)$ generate the same set of minimum cost edit sequences.* □

Informally, two string distance functions are equivalent if, for any $s_1, s_2 \in \Sigma^*$, a small change in weights does not change the minimum cost sequences that transform s_1 into s_2. In Example 2.3, if we change the weight vector $\omega = \{0.1, 0.9\}$ into $\omega' = \{0.09, 0.91\}$, the sequence of edit operations that yields the minimum cost does not change — i.e. we still would have to delete the 4 a's before the b and insert 4 a's after the b. In practice we usually fix O and let $\omega \in \Omega$ where Ω is some fixed sub-

space of $\mathbb{R}_+^n$ where n is the dimension of the weight vectors (and also the cardinality of O). We then call $\delta_O^\Omega() = \{\delta^\omega() \,|\, \omega \in \Omega\}$ a **family** of string distance functions. It is sometimes also desirable to make the operation weights 'compete' against each other by restricting Ω to be the n-1 dimension unit simplex [41] where $n = |O|$. In this case

$$\Omega \stackrel{\text{def}}{=} \{\omega = (w_1, w_2, \ldots, w_n) \,|\, \Sigma_{i=1}^n w_i = 1\}.$$

Chapter 3

Kernel Languages

The class of kernel languages is a subclass of the regular languages proposed by Goldfarb in [49]. A kernel language consists of all those strings over some given alphabet Σ that can be obtained by inserting, anywhere, in any order, and any number of times, any string from a finite set of strings called the *features* into a non-empty string called the *kernel*. The only restriction being that no feature can be a substring of any other feature or of the kernel. This domain is an example of a *structurally unbounded environment* (see Chapter 4). The concept of a structurally unbounded environment was developed by Goldfarb to describe those environments that cannot be 'hard-coded' into a learning algorithm. This prevents 'cheating' by the learning algorithm. In this chapter we first present updated definitions for Transformations System (TS) descriptions of formal languages and follow these with the first formal definitions for kernel languages, which are then extended to include languages with multiple kernels. We discuss some important properties of kernel languages such as confluence and closure, and present some interesting results. We also discuss how noise is handled and present some practical applications of kernel languages.

3.1 TS Class Descriptions for Formal Languages

The traditional structures used to specify, generate, and recognize formal languages are phrase-structure grammars (PSGs), finite-state and pushdown automata, regular expressions etc. These structures were not designed, nor intended, for grammatical inference. Their main weakness is their inherent awkwardness in handling noisy languages and, as Bunke explained in [16], noise is very often present in real-world classes. In grammatical inference, and indeed in machine learning as a whole, we are concerned with such issues such as the form and type of class description (an issue often ignored by many in the machine learning community) and also the issue of determining class membership. In practice, classes (or concepts) are very often 'noisy' and we therefore require forms of class descriptions and procedures to test for class membership that can handle noisy classes in a simple and elegant manner.

Recall from Chapter 1 that, in the ETS inductive learning model, classes of objects are defined in terms of a finite number of members of the class, a set of weighted transformations, and a distance function defined in terms of the transformations. When the domain is Σ^*, the objects (or *structs*) are strings, the transformations are rewrite rules, and the distance is usually a string edit distance such as Levensthein distance. In this section we discuss a new form of description of formal languages — the *Transformations System (TS) Class Description*. TS class descriptions for formal languages were introduced by Goldfarb in [47]. This paper described how one can specify a formal languages using a finite set of strings from the language, a set of rewrite rules (or transformations) and a weight vector associated with the rules. The string distance function induced by the rules and their respective weights is then used to define the class and to test for class membership. In this section we expand upon Goldfarb's paper by updating and refining the definitions and by addressing issues such as; how to best handle noisy languages, a comparison with other forms for

describing formal languages, and a discussion of the desirable properties of the string distance function. A number of examples are also provided in order to illustrate the basic concepts.

3.1.1 String Transformations Systems

A *String Transformations System* (STS) is a special instance of a Transformations System (TS)[1] where the set of objects (or structs) is the set, Σ^*, of all strings over some finite alphabet Σ, the transformations are string rewrite rules, and the distance functions are string edit distance functions defined on Σ^*. It must be pointed out that the definitions that follow are somewhat different from those that appeared in Goldfarb's original paper [47]. In particular, we have generalized the definition of the distance function and introduced new structures for dealing effectively with noise and for testing for class membership. In the definitions that follow Σ will always denote a finite alphabet, ε the empty string, and $s_1 \rightarrow_r s_2$ denotes the application of the rewrite rule r on string s_1 to obtain the string s_2. For example, if $s_1 = cac$ and $r = (a, b)$, then $s_2 = cbc$. Each rewrite rule is denoted by a pair such as (a, b) which means *replace ab by b*. A *chain* is a finite sequence of n rewrite rules, $r_1, r_2, \ldots, r_n$ that is applied to a string s_1 to obtain some other string s_2. In other words, a chain is an *edit sequence*. In the case when each transformation is assigned a weight, i.e. a non-negative real number, the *cost* of a chain is usually taken to be the sum of the weights of the transformations in the edit sequence. For all the definitions in this section, the set of transformations R is always *complete* — every string in Σ^* can be transformed into any other string in Σ^*. This ensures that the word problem is always decidable. Also, R is always a Thue system, i.e. $(a, b) \in R \Rightarrow (b, a) \in R$. Each rewrite rule can therefore be applied 'in both directions'.

[1] The reader is referred to Chapter 1 for the relevant definitions.

Definition 3.1 (String Transformations System). *Let Σ be a finite alphabet. A* ***String Transformations System****, $\mathcal{M} = (R, F, n, \phi)$ is a 4-tuple where;*

(a) R, the set of ***transformations****, is a finite string-rewriting system over Σ with $|R| = m$. R is indexed by the set $\{1, 2, \cdots, m\}$ with $ord(r)$, $r \in R$, being the indexing bijection of R into $\{1, 2, \cdots, m\}$. We call* $\mathbf{ord(r)}$ *the ordinal value of the rewrite rule $r \in R$. Also,*

- *for any $a \in \Sigma, (a, \varepsilon) \in R$, and*
- *for any $s_1, s_2 \in \Sigma^*$, $(s_1, s_2) \in R \Rightarrow (s_2, s_1) \in R$.*

(b) $F = \{\Delta_\omega\}_{\omega \in \Omega}$ is a family of ***distance functions****, $\Delta_\omega : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}_+$ where Ω is the set of all m by n (n is fixed for $\mathcal{M}$) matrices over $\mathbb{R}$ such that $\forall \omega \in \Omega$,*

$$\sum_{i=1}^{m} \omega_{i,j} = 1, for\ all\ 1 \leq j \leq n.$$

For every $\omega \in \Omega$ and for every rewrite rule $r \in R$, we denote by w_r the vector consisting of row $ord(r)$ in ω, i.e. $w_r = (\omega_{ord(r),1}, \omega_{ord(r),2}, \cdots, \omega_{ord(r),n})$. We call w_r the ***weight vector for r****. For any two strings $x, y \in \Sigma$ we define $\Delta_\omega(x, y)$ as follows: Let $\pi = s_0 \rightarrow_{r_1} s_1 \rightarrow_{r_2} s_2 \cdots \rightarrow_{r_v} s_v$ where $s_0 = x$, $s_v = y$, and $r_i \in R$, $i \leq i \leq v$, denote the chain (edit sequence) of length v that transforms x into y and let $\nu = w_{r_1}, w_{r_2}, \cdots, w_{r_v}$ denote the corresponding sequence of weight vectors of the rewrite rules in π. Then*

$$\Delta_\omega(x, y) \stackrel{\text{def}}{=} min_{\pi \in \Pi}\, \phi(w_{r_1}, w_{r_2}, \cdots, w_{r_q}),\ q \in \mathbb{N},$$

where Π is the set of all chains that transform x into y and $\phi : (\mathbb{R}_+^n)^\infty \rightarrow \mathbb{R}$ is a function (fixed for $\mathcal{M}$) that maps sequences of n-dimensional vectors into $\mathbb{R}$ such that the following always hold;

- $\Delta_\omega(x, y) = \Delta_\omega(y, x)$, *and*
- $\Delta_\omega(x, x) = 0$. □

Notes to Definition 3.1.

(i) We insist that, for every symbol $a \in \Sigma$, the rewrite rule (a, ε) is in R. This means that R contains as a subset the trivial string-rewriting system over Σ, i.e. all the single-letter insertion/deletion operations[2]. An important implication of this is that R is *complete*, i.e. any string in Σ^* can be transformed into any other string. The word problem is therefore always decidable. It also follows that for any $x, y \in \Sigma^*$, $x \rightarrow_R y$ always holds and, therefore, there exists at least one chain (edit sequence) $s_0 \rightarrow_{r_1} s_1 \rightarrow_{r_2} s_2 \cdots \rightarrow_{r_v} s_v$ where $s_0 = x$, $s_v = y$ and $r_i \in R$, $0 \leq i \leq v$, for some $v \in \mathbb{N}$. To obtain *one* such chain one first reduces x to ε using single-letter deletions and then builds y from ε using single-letter insertions. In practice, we usually find the chain and an ω that minimizes $\Delta_\omega(x, y)$. This is precisely why we often refer to F as a family of *competing* distance functions. This issue is expanded upon later on.

(ii) We also insist that, for any $x, y \in \Sigma^*$, $(s_1, s_2) \in R$ implies that $(s_2, s_1) \in R$, i.e. all rewrite rules are two-way. R, therefore, is a Thue system and $\rightarrow_R^*$ is the Thue congruence induced by R.

(iii) Since, for any $x, y \in \Sigma^*$, $\Delta_\omega(x, y) = \Delta_\omega(y, x)$, then, necessarily, for any $(a, b) \in R$, $w_{(a,b)} = w_{(b,a)}$. In other words, the weight vector for replacing a by b must be identical to that for replacing b by a since, otherwise, the symmetry condition $\Delta_\omega(x, y) = \Delta_\omega(y, x)$ would not always hold.

(iv) In most cases we let $n = 1$, i.e. we assign each rewrite rule a single non-negative real weight (a 1-dimension vector). However, for some applications it may be desirable, or even necessary, to assign multi-dimensional vectors as weights. For example, a rewrite rule may have a *deletion* cost, a *similarity* cost, and a

[2]We use the words *operation* and *rewrite rule* interchangeably

penalty. In this case, each transformation is assigned a weight vector from $\mathbb{R}^3$. Although we shall not elaborate here, we have tried to make the definition of string distance as general as possible in order to allow the practitioner as much flexibility as possible. The reader is referred to Example 3.2 below.

(v) Notice that, for any two strings $x, y \in \Sigma^*$, we do not specify how the distance function $\Delta_\omega(x, y)$ is computed. We only insisted that it is a *pseudo-metric*[3]. We could have defined $\Delta_\omega(x, y)$ to be the length of the shortest path or as the path of least cost but this would restrict the generality, and therefore the usefulness, of the distance function. We must also point out that, for noisy languages, we require that the distance function obeys the triangle inequality condition, i.e. for any three strings $a, b, c, \in \Sigma*$, $d(a, c) \leq d(a, b) + d(b, c)$. For noiseless languages a pseudo-metric suffices but in noisy languages the lack of the triangle inequality property, in general, makes correct classification impossible. This issue will be discussed later on in this section and in the next chapter.

The examples of formal languages specified by string transformations systems that we discuss later on in this section should convince the reader why we need the distance function to be as general as possible.

3.1.2 String TS Class Descriptions of Formal Languages

We now investigate the possibility, or rather the feasibility, of using string transformations systems (STSs) to describe (i.e. specify) formal languages. Unlike formal grammars, which were designed to specify, generate, and recognize formal languages, STS descriptions of formal languages were developed by Goldfarb [47] primarily to describe formal languages, especially in the presence of noise, and also, as we shall see in Part II of this thesis, for *learning* languages.

[3]See the Introduction of Chapter 2 for a definition.

Definition 3.2 (String TS Class Description).

*Let Σ be a finite alphabet. A **String TS Class Description** over Σ is a 3-tuple $(\mathcal{M}, A, \omega)$ where:*

(a) $\mathcal{M} = (R, F, n, \phi)$ be a string transformations system over Σ as in Definition 3.1.

*(b) A is a set of pairs of the form (s, δ_s) where $s \in \Sigma^*$ and $\delta_s \in \mathbb{R}$ is a non-negative real called the **delta-neighborhood value** for s. A is called the set of **attractors**.*

(c) ω is a weight matrix in Ω as in Definition 3.1. □

Definition 3.3. *The **language specified** by the string TS class description $(\mathcal{M}, A, \omega)$, denoted by $\mathbf{L}\langle \mathcal{M}, \mathbf{A}, \omega \rangle$ is the set of all strings $x \in S$ such that, for some $(a, \delta_a) \in A$, the distance between x and a is less than or equal to the delta-neighbourhood value for a. Formally,*

$$L\langle \mathcal{M}, A, \omega \rangle \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid \Delta_\omega(x, a) \leq \delta_a \, for \; some \; (a, \delta_a) \in A\}$$

□

Definition 3.4 (STS Language).

*A language $L \subseteq \Sigma^*$ is called an **STS Language** when $L = L\langle \mathcal{M}, A, \omega \rangle$ for some string TS description $(\mathcal{M}, A, \omega)$.* □

Informally, an STS language is specified by the set of attractors and their corresponding delta-neighbourhood values, the set of transformations (rewrite rules), and the weight vectors. The distance function induced by the transformations and their weights is used to determine class membership. This is achieved by computing the distance of the unclassified string x to each of the attractors. If, for some attractor, (a, δ), $d(a, x) \leq \delta$, then x is considered as belonging to the language. Notice that each attractor has its own delta-neighbourhood value.

3.1.3 Examples of String TS Class Descriptions

We now present some examples of TS class descriptions of formal languages. Some are for regular languages, some for context-free languages, and one is for a real-world language.

Examples of Regular Languages

Example 3.1 (The Regular Language a^*b).

This is language L_8 of the GI benchmarks listed in Appendix D. A grammar for this language is $G_{3.1} = (\Sigma, N, S, P)$ where $\Sigma = \{a, b\}$, $N = \{S\}$, and P contains the following productions:

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow b \end{aligned}$$

$L(G_{3.1}) = \{b, ab, aab, aaab, aaaab, \cdots\}$. A string TS class description for this language is $(\mathcal{M} = (R, F, n, \phi), A, \omega)$ where: $n = 1$,

$$R = \begin{bmatrix} a & \leftrightarrow & \varepsilon \\ b & \leftrightarrow & \varepsilon \\ ab & \leftrightarrow & b \end{bmatrix}, \qquad \omega = \begin{bmatrix} 0.5 \\ 0.5 \\ 0 \end{bmatrix}, \qquad A = \{(b, 0)\}, \qquad \phi(\nu_\pi) = \sum_{r \in \pi} w_r.$$

where π is a minimum-cost path that transforms one string into another. □

Notes to Example 3.1. $L(G_{3.1})$ is a very simple regular language. We use it to illustrate the main ideas involved in specifying formal languages using string transformations systems. Notice that we have three transformations: *insert/delete a*, *insert/delete b*, and *replace a by b*. The first two transformations are assigned a non-zero weight while the third is assigned a weight of zero. It is easy to see that any string in the language can be transformed into any other string in the language using only the zero-weighted operation $ab \leftrightarrow b$. The distance between any two strings in the language is therefore always zero. For this language we define the distance

between two strings x, y to be the minimum *cost* over all chains (edit sequences) that transform x into y. For $L(G_{3.1})$, the cost of a chain is defined to be the sum of all the weights of the transformations in the chain. Formally,

$$\Delta_\omega(x, y) \stackrel{\text{def}}{=} min_{\pi \in \Pi} \phi(\nu_\pi) \text{ with } \phi(\nu_\pi) = \sum_{r \in \pi} w_r.$$

where Π is the set of all chains that transform the string x into the string y (using only the transformations in R) and $\nu\pi$ is the sequence of weights associated with the chain π. To test for membership of the language one need only compute the distance between the unknown string z and the attractor b. As explained above, the distance between b and any other string in the language is zero while the distance between b and *any* string not in the language is greater than zero. Consider for example the string aba. To transform this string to b one can use the following chain: $aba \rightarrow_{(ab,b)} ba \rightarrow_{(a,\varepsilon)} b$. In fact, *any* chain that transforms a string not in $L(G_{3.1})$ to b, for that matter, to any string in $L(G_{3.1})$, *must* include at least one non-zero weighted transformation (rewrite rule).

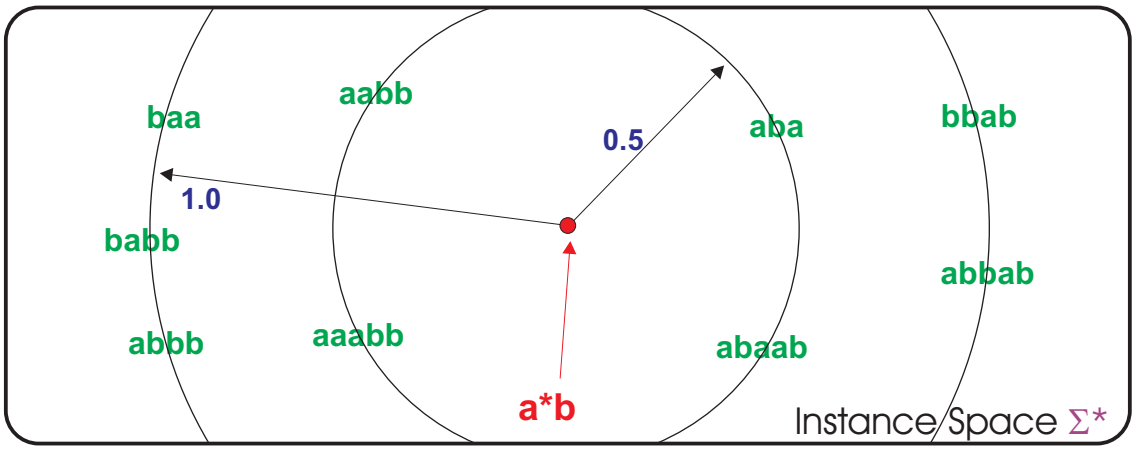


Figure 3.1: The pre-metric space embedding of the language a^*b.

Figure 3.1 shows the pre-metric space embedding of $L(G_{3.1})$. Note that all the strings in the language are contained in a single point in the pre-metric space. Any string

not in the language is proportionately far away from this point depending on the number of non-zero weighted required to transform it into a string in $L(G_{3.1})$.

Example 3.2 (The Regular Language $\mathbf{a}^* \cup \mathbf{b}^*$).

A grammar for this language [47] *is* $G_{3.2} = (\Sigma, N, S, P)$ *where* $\Sigma = \{a, b\}$, $N = \{S, A, B\}$, *and* P *contains the following productions:*

$$\begin{aligned} S &\rightarrow A \\ S &\rightarrow B \\ A &\rightarrow a \\ A &\rightarrow aA \\ A &\rightarrow a \\ A &\rightarrow aA \\ B &\rightarrow b \\ B &\rightarrow bB \end{aligned}$$

$L(G_{3.2}) = \{a, aa, aaa, \ldots b, bb, bbb, \ldots\}$.

A string TS class description for this language is $(\mathcal{M} = (R, F, n, \phi), A, \omega)$ *where:* $n = 1$,

$$R = \begin{bmatrix} a & \leftrightarrow & \varepsilon \\ b & \leftrightarrow & \varepsilon \\ aa & \leftrightarrow & a \\ bb & \leftrightarrow & b \end{bmatrix}, \qquad \omega = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.0 \\ 0.0 \end{bmatrix}, \qquad A = \{(a,0)(b,0)\}, \qquad \phi(\nu_\pi) = \sum_{r \in \pi} w_r.$$

□

Notes to Example 3.2. This language was chosen because it has two attractors. Strings in $L(G_{3.2})$ can be transformed, using only the zero-weighted rewrite rules, into one, and only one of the attractors. Notice also that both delta-neighbourhood values are set to 0, i.e no noise is allowed.

Example 3.3 (A Simple Kernel Language).

This is a simple kernel language. *Strings in the language* $L_{3.3}$ *are obtained by inserting into the* ***kernel string*** cac, *anywhere and any number of times, the* ***feature string*** bab.

$L(G_{3.3}) = \{cac, babcac, cbabac, cacbab, cababcbab \cdots\}$. *The inserted feature strings are shown in red. A regular expression for* $L_{3.3}$ *is* $(\mathbf{bab})^*\mathbf{c}(\mathbf{bab})^*\mathbf{a}(\mathbf{bab})^*\mathbf{c}(\mathbf{bab})^*$.

A string TS class description for this language is $(\mathcal{M} = (R, F, n, \phi), A, \omega)$ *where:* $n = 2$,

$$R = \begin{array}{|rcl|}\hline a & \leftrightarrow & \varepsilon \\ b & \leftrightarrow & \varepsilon \\ c & \leftrightarrow & \varepsilon \\ bab & \leftrightarrow & \varepsilon \\ \hline \end{array}, \qquad \omega = \begin{array}{|c|c|}\hline 0.33 & 0.0 \\ 0.33 & 0.0 \\ 0.33 & 0.0 \\ 0.0 & 1.0 \\ \hline \end{array}, \qquad A = \{(cac, 0.5)\}, \qquad \phi(\nu_\pi) = \frac{D}{S+\epsilon},$$

where;

$$D = \sum_{r \in \pi} \omega_{(ord(r),1)}, \text{ and} \tag{3.1}$$

$$S = \sum_{r \in \pi} \omega_{(ord(r),2)}. \tag{3.2}$$

ϵ *is a small positive real constant to prevent a divide-by-zero error when* S *is zero, and* π *is a minimum deletion cost chain (edit sequence).* □

Notes to Example 3.3. $L_{3.3}$ is an example of a *kernel language*. Kernel languages, a subclass of the regular languages, are defined formally and discussed later on in this chapter. In essence, the kernels play the role of the attractors. A kernel language is specified by a set of strings K called the kernels and another set of strings F called the features. The only requirement is that F is substring-free, i.e. no feature is a substring of another feature, and also that no feature is a substring of the kernel.

Example 3.3 is different from all the previous examples in that;

(a) it allows for noise,

(b) each rewrite rule is assigned a weight vector rather than a scalar as before,

(c) the delta-neighbourhood of the kernel is non-zero.

Note that, as in the previous examples, any *noiseless* string in the language $L_{3.3}$ can be transformed into any other noiseless string in $L_{3.3}$ using only the zero-weighted rewrite rule $bab \leftrightarrow \varepsilon$. The reason that the STS description of $L_{3.3}$ is rather more involved than those of the previous examples is because this particular STS description can handle 'noise' in the language — i.e spurious characters added to the strings of $L_{3.3}$. To see how we first discuss the distance function used. In all of our previous examples each rewrite rule is assigned a weight (a scalar). This weight is *deletion cost* of the particular rewrite rule. String distance was then defined to be the minimum cost over all chains that transform one string into the other. Any string $s \in \Sigma^*$ was then said to belong to the language if the distance between S and one of the attractors was zero. This scheme works well but is inadequate when the language is noisy. Suppose, for example, that we want to allow noise and that we wish to classify strings that have some noise to be classified as belonging to $L_{3.3}$. Consider the string *babcbabacbbabbabb*. The characters of the kernel are shown in black, the feature *bab* in red, and the noise is shown in blue. To transform this string into the kernel *cac* one must first delete all occurrences of the feature *bab*. This can be done with zero cost. One then must delete the 'noise' using the non-zero weighted rewrite rule $b \leftrightarrow \varepsilon$. Since the amount of noise is relatively small we may consider it reasonable to classify this string as belonging to $L_{3.3}$. We can achieve this by setting the delta-neighbourhood value of the kernel *cac* to small positive real value, say 1.33. For any string $s \in \Sigma^*$, if the distance from s to *cac* is less than or equal to 1.33 the

string is classified as belonging to $L_{3.3}$. This sounds reasonable enough except that it does not always work. Consider the string *acbca*. The minimum cost distance to the kernel *cac* is, in fact, 1.33. However, this string cannot reasonably be considered to be in $L_{3.3}$. Not only is the kernel *cac* not contained in *acbca* but, also, the kernel is not even a subsequence. Furthermore, the feature *bab* does not occur in this string. This example illustrates the problem with using minimum cost edit distance. This type of distance function measures the *difference* between two strings rather than the *similarity*. In addition, it measures the *absolute* difference. This type of distance was used by Nigam [92] in his ETS learning algorithm for kernel languages. A discussion of the problems of this kind of distance in found in Chapter 4 where we also discuss Nigam's algorithm. The definition of distance we used in the STS description for $L_{3.3}$ avoids this problem. We use a new method for computing string distance that we call *normalized string edit distance*. Each rewrite rule is assigned a weight vector, $w_r = (\omega(ord(r), 1), \omega(ord(r), 2))$, rather than a scalar. The first component is the *deletion cost* of the rewrite rule while the second is the *similarity value*. Distance between any two strings $s_1, s_2 \in \Sigma^*$ is then computed in the following manner:

(a) First find an edit sequence (chain) that transforms s_1 into s_2 and that has the minimum deletion cost. Let D equal to the sum of the deletion costs of the rules in the chain.

(b) Let S be equal to the sum of the similarity values of the rules in the same chain.

(c) The distance is then computed by dividing D by S;

$$\frac{D}{S + \epsilon}.$$

We illustrate with an example. Consider again the string *babcbabacbbabbabb*. To transform this string into the kernel *cac* we first delete the 4 occurrences of the

feature *bab* and then the 'noise' by deleting the two *b*'s. The rewrite rule $bab \leftrightarrow \varepsilon$ has a deletion cost of 0 and a similarity cost of 1 while the rewrite rule $b \leftrightarrow \varepsilon$ has a deletion cost of 0.33 and a similarity cost of 0. The distance is therefore,

$$\frac{0.66}{4.0 + \epsilon} = 0.165$$

This method of computing distance avoids the problems mentioned earlier. The use of distance allows an elegant and very natural way to deal with noisy languages. The reader should note, that unlike the case with stochastic grammars [33, 137], one does not need special rules to deal with the noise. All the noise is handled by the primitive single-character insertion/deletion rewrite rules which are always present since, by definition, R is complete. In the example above we have not shown how to detect features inserted inside other features. This is accomplished by using the EvD distance function that is introduced and discussed later on in Section 3.3.

Examples of Context-Free Languages

Example 3.4 (The Language $\mathbf{a^i b^i}$, $i > 0$).

A grammar for this language [126, page 322] *is* $G_{3.4} = (\Sigma, N, S, P)$ *where* $\Sigma = \{a, b\}$, $N = \{S\}$, *and* P *contains the following productions:*

$$\begin{aligned} S &\rightarrow ab \\ S &\rightarrow aSb \end{aligned}$$

$L(G_{3.4}) = \{ab, aabb, aaabbb, \ldots\}$.

A String TS class description for this language is $(\mathcal{M} = (R, F, n, \phi), A, \omega)$ *where:* $n = 1$,

$$R = \begin{bmatrix} a & \leftrightarrow & \varepsilon \\ b & \leftrightarrow & \varepsilon \\ aabb & \leftrightarrow & ab \end{bmatrix}, \qquad \omega = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.0 \end{bmatrix}, \qquad A = \{(\varepsilon, 0)\}, \qquad \phi(\nu_\pi) = \sum_{r \in \pi} w_r.$$

Notes to Example 3.4. This language is interesting because it has two attractors. Notice that each string in the language can be transformed into the attractor ab using only the zero-weighted rewrite rule $aabb \leftrightarrow ab$. The attractor ε is included only because it belongs to the language. We call this type of attractor a *dormant attractor*.

Example 3.5.

This language consists of all strings over the alphabet $\Sigma = \{a, b\}$ *in which* a *occurs as many times as* b. *A grammar for this language* [101, Exercise 1.3] *is* $G_{3.5} = (\Sigma, N, S, P)$ *where* $\Sigma = \{a, b\}$, $N = \{S\}$, *and* P *contains the following productions:*

$$\begin{aligned} S &\rightarrow ab \\ S &\rightarrow ba \\ S &\rightarrow SS \\ S &\rightarrow aSb \\ S &\rightarrow bSa \end{aligned}$$

A String TS class description for this language is $(\mathcal{M} = (R, F, n, \phi), A, \omega)$ *where:* $n = 1$,

$$R = \begin{bmatrix} a & \leftrightarrow & \varepsilon \\ b & \leftrightarrow & \varepsilon \\ ab & \leftrightarrow & \varepsilon \\ ba & \leftrightarrow & \varepsilon \end{bmatrix}, \qquad \omega = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.0 \\ 0.0 \end{bmatrix}, \qquad A = \{(ab, 0)(\varepsilon, 0)\}, \qquad \phi(\nu_\pi) = \sum_{r \in \pi} w_r.$$

□

Notes to Example 3.5. This example is interesting because its only attractor is the empty string ε. The reader should note how compact and economical the description is when compared to the corresponding grammar.

An Example from Pattern Recognition

Example 3.6.

The human electrocardiogram (ECG) can be described by a simple regular language that consists of strings that are a concatenation of the substrings prbtb, prbtbb, *and* prbtbbb. *The characters* p, r, *and* t *are the waveform primitives for the pulses and* b *is the waveform primitive for the quiescent times [126, page 118]. A grammar for this language is* $G_{3.6} = (\Sigma, N, S, P)$ *where* $\Sigma = \{p, r, t, b\}$, $N = \{S, A, B, C, D, E, H\}$, *and* P *contains the following productions:*

$$\begin{array}{lcllcllcl} S & \to & pA & S & \to & pA & S & \to & pA \\ C & \to & tD & C & \to & tD & C & \to & tD \\ E & \to & b & E & \to & b & E & \to & b \\ H & \to & b & H & \to & b & H & \to & b \end{array}$$

A String TS class description for this language is $(\mathcal{M} = (R, F, n, \phi), A, \omega)$ *where:* $n = 1$,

$$R = \left[\begin{array}{rcl} p & \leftrightarrow & \varepsilon \\ r & \leftrightarrow & \varepsilon \\ t & \leftrightarrow & \varepsilon \\ b & \leftrightarrow & \varepsilon \\ prbtb & \leftrightarrow & \varepsilon \\ prbtbb & \leftrightarrow & \varepsilon \\ prbtbbb & \leftrightarrow & \varepsilon \end{array}\right], \quad \omega = \left[\begin{array}{c} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \\ 0.0 \\ 0.0 \\ 0.0 \end{array}\right], \quad A = \{(\varepsilon, 0)\}, \quad \phi(\nu_\pi) = \sum_{r \in \pi} w_r.$$

□

Notes to Example 3.5. Human ECG signals can be encoded as strings over the alphabet $\{p, r, t, b\}$. A regular expression for this language is $\{prbtb+prbtbb+prbtbbb\}^*$. This is a kernel language with *prbtb*, *prbtbb*, and *prbtbbb* as the features and ε as the sole kernel. Again, the reader should note how compact and elegant the TS description is when compared to the grammar for the same language. The language contains no noise and, therefore, the delta-neighbourhood value of the attractor ε is set to 0.

One important point we must make is that the distance function should be designed in such as way so as to prevent the misclassification of strings that contain features inside other features. For example, consider the string *prbprbtbtb*. This string contains the feature *prbtb* inside another occurrence of the same feature. Unless the string distance function is designed properly, this string will be classified as belonging to the language. One can first delete the inner occurrence of the feature *prbtb* and then delete the outer one to obtain the kernel ε. This happens if the distance function allows what we call *deletion with concatenation*, i.e if deletion of an inner substring is followed by the concatenation of the remaining parts of the string. One of the problems of the GLD distance function used by Nigam in his ETS learning algorithm for kernel languages was precisely this. GLD allows deletion with concatenation and therefore recognizes strings that contain features inserted inside other features even though they do not belong to the language. *Evolutionary distance (EvD)*, which we introduce in the next section, addresses (and solves) this problem.

Goldfarb's paper [47] also includes a TS description of another real-world language. This is the language over the alphabet $\{a, b, c, d, e\}$ that consists of strings that are string encodings (chain-codes) of telocentric and submedian chromosomes. This language is context-free and was first described in [33]. The context-free grammar for this language has 20 production rules while the TS description has only five weighted rewrite rules.

3.1.4 The Role of the Attractors in TS Class Descriptions

It can be argued that the most important component of a TS description of a formal language is the set of transformations R. Note that, in each of the examples of TS descriptions that we presented, each set of transformations contained a set of zero-weighted transformations and a set of non-zero-weighted transformations. The zero-weighted transformations, which we shall henceforth denote by $\mathbf{R_0}$, are the *features* of the language. The non-zero transformations, which we shall henceforth denote by $\mathbf{R_P}$, are usually the *primitive* (i.e. single-character) insertion/deletion rewrite rules. The primitive transformations are included to ensure that whole set is complete — i.e. every string in Σ^* can be transformed into every other string in Σ^*. The transformations in $\mathbf{R_P}$ also are responsible for handling 'noise'. After reviewing the examples of TS descriptions it should become evident to the reader that the zero-weighted transformations (or rewrite rules) play a primary role in the specification of the language since they capture the similarity between the strings of the language. The attractors play a secondary but critical role. The attractors act as *templates* or *prototypes* of the strings in the language. Without the attractors, specifying the language would become very difficult — if not impossible. We illustrate with an example. Let the set of transformations R be $\{b \leftrightarrow \varepsilon, a \leftrightarrow \varepsilon, aabb \leftrightarrow ab\}$ and let the corresponding weight vector be $(0.5, 0.5, 0.0)$. The only rewrite rule that has a zero weight is $aabb \leftrightarrow ab$. Note that R can be used to describe the language $a^n b^n$, for $n \geq 1$ by defining the language to be all those strings in Σ^* that can be transformed into the attractor ab using only the zero-weighted rewrite rule $aabb \leftrightarrow ab$. We cannot, however, define the language only in terms of this rewrite rule. This is because the rewrite rule $aabb \leftrightarrow ab$ induces an equivalence relation on Σ^* where the language $a^n b^n$ is but one equivalence class. We must therefore have a method for excluding the other equivalence classes. Notice that R_0 is the Thue system consisting of the

single rewrite rule $aabb \leftrightarrow ab$ and $\overset{*}{\longleftrightarrow}_{R_0}$ is the Thue congruence generated by R — i.e. the symmetric, reflexive, and transitive closure of $\leftrightarrow_{R_0}$. Now, let $\sim \subset \Sigma^* \times \Sigma^*$ be a relation on Σ^* defined as follows:

$$\sim \overset{\text{def}}{=} \{(a, b) \in \Sigma^* \times \Sigma^* \mid a \overset{*}{\longleftrightarrow}_{R_0} b\}.$$

In other words, two strings in Σ^* are related if one can be transformed into the other using the rewrite rule in R_0, i.e. $aabb \leftrightarrow ab$. What are the equivalence classes of $\sim$?. Each equivalence class consists of all those strings that can be transformed into each other using this rewrite rule. Notice that R_0 is obviously confluent and therefore each equivalence class contains only one irreducible string (normal form). Each equivalence class can therefore be specified by its (unique) normal form. The language $a_n b_n$ is one particular equivalence class and is precisely all the strings that have ab as their normal form. Another equivalence class is the set of strings that can be reduced to the normal form abb, i.e. $\{abb, aabbb, aaabbbb, aaaabbbbb, \ldots\}$. In fact, there are infinitely many equivalence classes. In general, this scenario is always the case. R_0, which contains the zero-weighted transformations, induces a natural equivalence relation on Σ^*. The role of the attractors is then to specify which equivalence classes are included in the language. This is required even when R is confluent. Sometimes, particularly when R is non-confluent, the attractors are used to specify a proper subset of an equivalence class. Consider, for example, the kernel language that consists of all strings over the binary alphabet that can be obtained from the kernel *101* by inserting anywhere, any number of times, and in any order, the features *00*, *11*, and *010*. In this case $R = \{0 \leftrightarrow \varepsilon, 1 \leftrightarrow \varepsilon, 00 \leftrightarrow \varepsilon, 11 \leftrightarrow \varepsilon, 010 \leftrightarrow \varepsilon\}$. The corresponding weight vector is $(0.5, 0.5, 0, 0, 0)$. R_0 therefore contains the transformations $00 \leftrightarrow \varepsilon$, $11 \leftrightarrow \varepsilon$, and $010 \leftrightarrow \varepsilon$. In this case R_0 is non-confluent and the congruence class that contains the kernel *101* also contains the normal form *01*. This is because both strings are normal forms of the string *00101*. In the case of $a^n b^n$ the language is, in fact, a

congruence class. But, as we have also seen, this is not always the case. The point here is that using attractors allows us to specify languages that are not necessarily *congruential*[4]. These examples demonstrate that, in general, STS languages are not necessarily congruential.

3.1.5 The Role of the Distance Function

We have seen that the set of transformations R and the set of attractors A play different but complementary roles in the specification of STS languages. We now investigate the role of the distance function. We must emphasize before proceeding any further that the distance function is not just useful for the purpose of class description if the language (i.e the class) is noisy. Even in the case of crisp (i.e. 'noiseless') there is much to be gained by using a TS description. We shall see also in Part II of this thesis that distance plays an important role during the learning process independently of whether the language we are trying to learn is crisp or noisy. At the moment, however, we shall discuss only the role of distance in class description.

In might appear, at first glance, that in the case of *crisp* (or noiseless) languages, the distance function is much less important since the TS description of the language then functions as a *propositional description* — i.e. a method for determining whether or not a string belongs to the language in question. One may argue that this is precisely what we need — but we beg to differ. In ETS theory, unknown objects are classified on the basis of their distance to one of the attractors of a class. In the case of strings, we determine whether a string belongs to a language or not depending on its distance to one of the attractors of the language. If the language is noiseless, the delta-neighbourhood values of the attractors of the language are set to zero. This

[4]Recall from Chapter 2 that a language is congruential if it is the union of a finite number of congruence classes of a Thue system.

means that a string belongs to a language only if the distance to one of the attractors is zero. In such cases, it is true that the TS description of the language acts like a propositional description such as a formal grammar. A TS description has one important advantage over formal grammars and other similar forms of description. TS descriptions allow for a more compact and informative description. Consider the following example:

Example 3.7.

$L_{3.7}$ = the language consisting of all strings over $\Sigma = \{a, b, c\}$ that contain the substring abc. See [115] for a formal grammar for this language.

A regular expression for $L_{3.7}$ is $\{a + b + c\}^* abc\{a + b + c\}^*$.

A string TS class description for this language is $(\mathcal{M} = (R, F, n, \phi), A, \omega)$ where: $n = 2$,

$$R = \left[\begin{array}{ccc} a & \leftrightarrow & \varepsilon \\ b & \leftrightarrow & \varepsilon \\ c & \leftrightarrow & \varepsilon \\ abc & \leftrightarrow & abc \end{array}\right], \qquad \omega = \left[\begin{array}{c|c} 0.33 & 0.0 \\ 0.33 & 0.0 \\ 0.33 & 0.0 \\ 0.0 & 1.0 \end{array}\right], \qquad A = \{(abc, 1.0)\}, \qquad \phi(\nu_\pi) = \frac{D}{S+\epsilon},$$

where;

$$D = \frac{\sum_{r \in \pi} \omega_{(ord(r),1)}}{max(|s_1|, |s_2|)}, \text{ and} \tag{3.3}$$

$$S = \sum_{r \in \pi} \omega_{(ord(r),2)}. \tag{3.4}$$

ϵ is a small positive real constant to prevent a divide-by-zero error when S is zero, and π is a minimum deletion cost chain (edit sequence). □

In Example 3.7, D is weighted Generalized Levensthein distance (GLD) that is normalized by dividing by the length of the longer of the two strings. S is simply the

sum of all the similarity values of the transformations in the GLD chain. The transformation $abc \leftrightarrow abc$ requires some clarification. In normal circumstances replacing a substring by itself seems futile but these so-called *match operations* are used in all Levensthein string-edit distance algorithms and their derivatives [57, 114]. In fact, the Levensthein distance algorithm and its derivatives all use the basic (primitive) single-character match operations. These are assigned a zero weight and allow the algorithm to progress when the ith character of the first string matches the jth character of the second string. In our case we are using the match operation to detect if both strings contain the substring abc. Given any string $s \in \Sigma^*$, then;

(a) If s contains abc as a substring, the distance to the kernel abc will be always less than 1. More precisely, it will be in the interval $[0, 1[$.

(b) If s does not contain the string abc then the distance to the kernel abc will be greater than 1.

Example 3.7 is very interesting in that it shows the flexibility of using distance for class description. We use only one transformation to describe the language. Everything else is considered as 'noise'. The reader is urged to compare the above TS description with the rather cumbersome grammar for the same language found in [115]. As Goldfarb explained in [47], grammars must be able to build any string in the language while a TS description captures only the similarity between the strings in the language.

When the language is noisy, as is the case with the majority of real-world applications, traditional propositional descriptions such as grammars and automata become unsuitable. It is for this reason that stochastic grammars and stochastic automata were developed [33, 26, 108]. TS descriptions allow for a more compact and versatile description of a noisy formal language than do stochastic automata and grammars since we can tailor the distance function to suit the domain. In particular, we can

control the *amount* and the *type* of noise. Let us go back to Example 3.3. In this example we showed how we can vary the values of the δ-neighbourhoods of the attractors in order to vary the amount of noise in the language. This is because we used a distance function that normalized the noise by dividing the cost of the deletions by the similarity value of the chain. In other words, the distance function does not just measure the difference between two strings but also the similarity. This allows us to measure the 'relative' noise. It is not clear how this can be done with stochastic grammars. Moreover, one can also vary the weights of the primitive operations to control the *type* of noise. For instance, decreasing the deletion cost of the character b will results in strings with more b's being accepted as being in the language.

In all the examples we presented, the distance function we chose was always a pre-metric. In general, metrics are not suitable since a metric function does not allow two distinct objects to have a pair-wise distance of zero. The choice therefore lies between a pseudo-metric and a pre-metric. The only difference between the two is that a pseudo-metric does not have to observe the triangle inequality. This, in fact, turns out to be a problem when the language is noisy. When a language is noisy, one cannot correctly specify a class using a pseudo-metric. This problem is discussed in detail in the next chapter when we examine the GLD distance function as used by Nigam in the GSN algorithm.

3.1.6 Comparison with Other Forms of Description

In syntactic and structural pattern recognition *structural representation* is used to encode the objects in a domain of discourse. In this type of representation complex objects are recursively built from primitive objects. It is therefore not surprising that formal grammars have historically been used for pattern class description. This is probably because formal language theory is well known and understood and because

grammars describe how complex patterns can be built from simpler, more primitive, constituents. The recognition procedures are based on the well-known concept of language parsing for which many algorithms exist. Many researchers have questioned the use of formal grammars in syntactic pattern recognition. In a well known textbook on pattern recognition, Watanabe wrote [137, page 438]:

> Phrase-structure grammar has become so prestigious within a short period of time that to many people grammar nowadays means exclusively phrase-structure grammar. Correspondingly, many of those who try to make a grammatical theory of pictures take it for granted that they have to emulate the the phrase-structure grammar to make a really modern theory of pictures. To understand the strong points and shortcomings of such an attempt, it will be necessary to consider the reason why the phrase-structure grammar has become so popular so quickly. The intellectual attractiveness of this grammatical theory derives from its kinship with the proof theory developed by the logicians and automata theory. Certainly, the founders of this type of grammar were strongly influenced by the then popular new stream of logical development associated with such fascinating names as Frege, Russel, Gödel, Post, and Turing, among others. The influence of this proof theory on this language theory is so evident that even now there are still people who feel that this theory tries to force a framework which is natural in another entirely different field on the field of language. (Chomsky himself argued that phrase-structure grammars are inadequate to account for the class of grammatical sentences in a natural language.) But the analogy between grammaticalness and provability (computability) is so beautiful and so enticing that we are tempted to believe that there is something deeper than mere coincidence. But, people tend to forget that this key point of phrase-structure grammar has no role to play in is adaptation to the picture theory. The phrase-structure grammar is supposed to produce grammatically correct sentences and only grammatically correct sentences. But the grammatical, generative rules assumed in a picture theory often produce pictures that can never occur in practice. Furthermore, there are many pictures that occur in practice that cannot be produced by those generative rules. Even limiting ourselves to a certain well-defined family of pictures, we find there is no such thing as "pictureness" corresponding to "grammaticalness" or "provability" or "computability". A grammatical theory is a descriptive theory, but it derives its utility partly from

> its normative aspect. In picture applications, however, there are not useful normative concepts. If a single picture appears that violates our grammatical rules, we cannot reject the picture because it is ungrammatical, but we have to modify the rules. The result will probably produce many pictures which we do not want to include in our family of pictures.

Watanabe is not alone in his belief that phrase-structure grammars are not always suitable for syntactic pattern recognition. Tanaka [116] and Goldfarb [47] both addressed the issue in their papers. The main problems of phrase-structure grammars and, for that matter, other forms of description such as automata, are:

Noisy Classes Grammars and automata do not handle noise properly. Stochastic grammars and stochastic automata were developed to handle the problem. In stochastic grammars, probabilities are assigned to each production. Special productions are also added to handle the noise. In essence one has to 'learn' the noise too and find special productions to handle the noise. The resulting class descriptions are often cryptic and difficult to understand. Moreover, few techniques have been developed for the inference of stochastic grammars. The probabilities assigned to the productions depend very much on the training sets. This makes the approach very sensitive to changes in the sets of training examples. TS descriptions use distance to handle noisy classes. The primitive transformations are then used to handle the noise in the strings. One can then design a distance function for the domain in question. As we have already seen in this section, the distance function can be tailored for the various types, and amounts, of noise.

Class Description Grammars, and automata in particular, are not very good class descriptions. Grammar descriptions of formal languages can be cumbersome. This is because, as Goldfarb explains in [47], grammars must be able to build *all* the strings in the languages while in TS descriptions, the zero-weighted

transformations always capture only the *similarity* between the strings of the language. This was demonstrated in Example 3.7, the language the of all strings that contain the substring *abc*. The grammar for the language is cumbersome and uninformative. The TS description contains just one single transformation $abc \leftrightarrow abc$. This transformation captures the regularity in the language. Everything else is treated as noise and is handled by the primitive single-character transformations. The distance function was chosen to allow for a very compact and information class description.

3.1.7 Summary

Roughly speaking, string transformations systems (STSs) are a generalization of Thue systems — we add a weight to each rewrite rule and define an appropriate string distance function. We have seen how string transformations systems can be used to specify and define formal languages by mean of a TS description. In particular we have shown how, in many cases, TS descriptions give us a much more compact and informative class descriptions than formal grammars. In the case of noisy languages, we can tailor the distance function to control the amount and type of noise. We have also argued that TS descriptions have distinct advantages over other forms of description such as stochastic grammars and automata.

One important topic that is not discussed is the *expressive power* of string transformations systems. McNaughton dealt with the expressive power of Thue systems in [80]. He restricted his attention to Church-Rosser Thue systems, i.e. those were the reduction relation induced by the rewrite rules is confluent. In this thesis we consider only rewrite-rules that do not allow variables (non-terminal symbols). This allows us to describe classes of regular and context-free languages. McNaughton showed that the introduction of variables allows for the description of more complex languages.

3.2 Kernel Languages

In this section we introduce and discuss the subclass of regular languages we call *Kernel Languages*. A kernel language over a finite alphabet Σ is specified by the pair $\langle K, F\rangle$ where $K \subset \Sigma^*$ is a finite, non-empty, set of strings called the set of *kernels* and $F \subset \Sigma^+$ is a finite, non-empty, and substring-free set of strings called the set of *features*. Informally, the strings in the kernel language specified by $\langle K, F\rangle$ are precisely those strings that can be obtained (generated) by inserting features from F anywhere, in any order, and any number of times, into the kernel strings of K. We only require that;

(a) features are not inserted inside other features,

(b) no feature is a substring of any other feature, i.e. F is substring-free, and

(c) no kernel contains a feature as a substring.

We illustrate with an example. Consider the set of features $F = \{ba, ab\}$ and the set of kernels $K = \{bb, bc\}$. The following strings in $L\langle K, F\rangle$, the language generated from K and F, are obtained by successive insertions of features in the kernels: (kernels are shown in red)

bb	bc
$babb$	$bcab$
$babbab$	$abbcab$
$babab\mathbf{b}ab$	$abbbac\mathbf{c}ab$
$babab\mathbf{b}a\mathbf{b}ab$	$abbbab\mathbf{a}c\mathbf{c}ab$
$abbab\mathbf{b}a\mathbf{b}\mathbf{b}ab$	$baabbbab\mathbf{a}c\mathbf{c}ab$
$abbab\mathbf{b}a\mathbf{b}\mathbf{b}ab\mathbf{a}bab$	$baabbbab\mathbf{a}c\mathbf{c}abba$

Notice that features can be inserted anywhere in a kernel but not inside another feature. We must point out, however, that this does not necessarily mean that a string in $L\langle K, F\rangle$ cannot contain substrings such as $aabb$ which can be formed by the insertion of the feature ab inside another occurrence of the same feature. The reason

for this is because this feature can also be formed from the features ab and ba as follows: $bb \rightarrow_{ab} abbb \rightarrow_{ba} baabbb$ to obtain the string $baabbb$ which, of course, contains $aabb$ as a substring. Testing for membership in a kernel language is achieved either by checking if a given unknown string x can be generated from one of the kernels by a sequence of feature insertions or, alternatively, by (nondeterministically) deleting features from x to obtain a kernel. Note that the latter procedure is equivalent to computing the normal forms of x modulo the special semi-Thue system, R_F, that consists exactly of $|F|$ rules of the form (f, ε), $f \in F$. The set of rewrite rules of R_F is therefore indexed by F. To determine if x belongs to $L\langle K, F\rangle$ we then need only check whether one of the normal forms belongs to K.

We continue this section by first presenting some important preliminary definitions and then proceeding to the main definitions of kernel languages. We then look at various types of kernel languages such as confluent and non-confluent kernel languages, trivial kernel languages, non-congruential kernel languages, and kernel languages with single or multiple kernels. We also present some interesting results about, and properties of, kernel languages. Kernel languages were introduced by Goldfarb as a non-trivial class of languages for benchmarking learning algorithms. It turns out, however, that kernel languages have a number of real-world applications which we present in this section. In this section we also present a new string-edit distance function, Evolutionary Distance (EvD), which we developed in response to the problems we identified with GLD distance. EvD can handle all types of noise, including classification noise, and was successfully used in Valletta for learning multiple-kernel languages.

3.2.1 Preliminary Definitions

Definition 3.5 (s-Deletion). *Let $s \in \Sigma$ be a symbol in Σ and let $x \in \Sigma^+$ be a non-empty string over Σ. We denote by $x|_s$ the string obtained from x by deleting all occurrences of s.* □

When we (informally) defined kernel languages in the introduction to this section we insisted that the features can be inserted anywhere in the kernels but not inside other features. We therefore need to ensure that when the strings of the language are generated, features are not inserted inside other features and, equivalently, when testing for membership in a given kernel language, strings that contain features inside other features are not accepted unless they can be reduced to some kernel in K. Our formal definition of kernel languages must accommodate this requirement. To achieve this we introduce a new symbol, θ, that is not in alphabet Σ. We call θ the *placeholder symbol* and its role is to prevent features being inserted inside other features. To this end we also define two types of string-rewriting rules; *θ-substitutions* and *θ-reductions*.

For the remaining part of this section we fix the following notation:

(a) Σ denotes a finite alphabet,

(b) θ, the *placeholder symbol*, is a symbol not in Σ, and

(c) Γ denotes the *extended alphabet* $\Sigma \cup \{\theta\}$.

Definition 3.6 (θ-Substitutions). *Let F be a non-empty finite subset of Σ^+ and let $f \in F$. We define the binary relation $\overset{\theta}{\rightarrow}_f \subset \Gamma^* \times \Gamma^*$ to be the set of all ordered pairs of strings $(x, y) \in \Gamma^* \times \Gamma^*$ such that y can be obtained from x by replacing (substituting) a θ in x by $\theta f \theta$. Formally, for any $x, y \in \Gamma^*$, $x \overset{\theta}{\rightarrow}_f y$ if and only if for some $u, v \in \Gamma^*$, $x = u\theta v$ and $y = u\theta f\theta v$. We call $\overset{\theta}{\rightarrow}_f$ the **θ-substitution relation induced by (the feature) f**. Also, let*

- $\overset{\theta}{\rightarrow}_f^{\,*}$ *denote the reflexive transitive closure of* $\overset{\theta}{\rightarrow}_f$ *and,*
- $\overset{\theta}{\rightarrow}_f^{\,+}$ *denote the transitive closure of* $\overset{\theta}{\rightarrow}_f$.

We now abstract further by hiding the string f *used in the* θ*-substitution, insisting only that the string belongs to the given set* F*. We accomplish this by letting* $x \overset{\theta}{\rightarrow}_F y$ *denote the set of all pairs of strings,* (x, y), $x, y \in \Gamma^*$ *such that* y *is obtained from* x *by replacing a* θ *in* x *by* $\theta f \theta$ *for some* $f \in F$*. We call* $\overset{\theta}{\rightarrow}_F$ *the* θ***-substitution relation induced by F****. Formally;*

$$\overset{\theta}{\rightarrow}_F \overset{\text{def}}{=} \bigcup_{f \in F} \overset{\theta}{\rightarrow}_f .$$

Furthermore, let

- $\overset{\theta}{\rightarrow}_F^{\,*}$ *denote the reflexive transitive closure of* $\overset{\theta}{\rightarrow}_F$ *and,*
- $\overset{\theta}{\rightarrow}_F^{\,+}$ *denote the transitive closure of* $\overset{\theta}{\rightarrow}_F$.

Finally, we denote by $\boldsymbol{G_F}$ *the string-rewriting system consisting of all the rules of the form* $(\theta, \theta f \theta)$ *where* f *is a feature. Formally,*

$$G_F \overset{\text{def}}{=} \{(\theta, \theta f \theta) \mid f \in F\}.$$

We call G_F *the* θ***-substitution system induced by F****.* □

The following are some obvious but useful results. The proofs are omitted.

Proposition 3.1. *Let* x *and* y *be any two strings in* Γ^**. Then*

(a) if $x \overset{\theta}{\rightarrow}_F^{\,+} y$ *then* $|x| < |y|$,

(b) if $x \overset{\theta}{\rightarrow}_F^{\,*} y$ *then* $|x| \leq |y|$*, and*

(c) if $x \overset{\theta}{\rightarrow}_F^{\,*} y$ *and* $|x| = |y|$*, then* $x = y$.

Notice that $\overset{\theta}{\rightarrow}_F^*$ is the reduction relation induced by G_F. Note also that $\overset{\theta}{\rightarrow}_F^*$ is most certainly not noetherian. This is because every rewrite rule in G_F is length-increasing and $\forall x, y \in \Gamma^*$ such that $x \overset{\theta}{\rightarrow}_f^* y$, $f \in F$, y always contains the left-hand side of the rule, i.e. θ. This means that *any* of the rules in G_F can be applied in an endless sequence. It does turn out, however, that $\overset{\theta}{\rightarrow}_F^*$ is always confluent — irrespective of our choice of F. We now state this result as a theorem.

Theorem 3.2. *Let $F \subset \Sigma^*$ be any finite set of strings over Σ. Then $\overset{\theta}{\rightarrow}_F^*$ is confluent.*

Informally, this result is true because if, for any $x, y \in \Gamma^*$, $x \overset{\theta}{\rightarrow}_F^* y$, then we can apply the rewrite rules (θ-substitutions) used to transform x into y in *any order* — i.e. any permutation of the re-write rules will do. We illustrate with an example. Let $\theta c\theta c\theta$ be a string in Γ^* and let ab and ba be strings in F. Then: (the inserted feature is shown in red)

$$\theta c\theta c\theta \overset{\theta}{\rightarrow}_{ab} \theta c\theta \textcolor{red}{ab} \theta c\theta \overset{\theta}{\rightarrow}_{ab} \theta \textcolor{red}{ba} \theta c\theta ab\theta c\theta$$
$$\theta c\theta c\theta \overset{\theta}{\rightarrow}_{ba} \theta \textcolor{red}{ba} \theta c\theta c\theta \overset{\theta}{\rightarrow}_{ba} \theta ba\theta c\theta \textcolor{red}{ab} \theta c\theta$$

The order the rewrite rules are applied is not important. We now state this result as a lemma and present a proof. We can then proceed directly to proving Theorem 3.2.

Lemma 3.3. *Let x, y be two strings over Γ such that $x \overset{\theta}{\rightarrow}_F^+ y$. Then there exists a finite chain, $s_0 \overset{\theta}{\rightarrow}_{f_1} s_1 \overset{\theta}{\rightarrow}_{f_2} s_2 \ldots \overset{\theta}{\rightarrow}_{f_n} s_n$, for some $n > 0$, such that $s_0 = x$, $s_n = y$, and $f_i \in F$ for $1 \leq i \leq n$. Let $\mathcal{I}$ be the indexing set $1, 2, \ldots, n$ that indexes the features used in the chain. Then for every bijection of $\mathcal{I}$ onto itself we get the chain, $s_0 \overset{\theta}{\rightarrow}_{f_{i_1}} s'_1 \overset{\theta}{\rightarrow}_{f_{i_2}} s'_2 \ldots \overset{\theta}{\rightarrow}_{f_{i_n}} s'_n$, and $s'_n = s_n = y$.*

To prove Lemma 3.3 we require the following result.

Lemma 3.4. *Let x, y be two strings over Γ such that $x \overset{\theta}{\rightarrow}_F^2 y$. Let $x \overset{\theta}{\rightarrow}_{f_1} w \overset{\theta}{\rightarrow}_{f_2} y$ be a 2-chain where f_1 and f_2 are two, not necessarily distinct, features in F used to*

transform x into y. Then $x \xrightarrow{\theta}_{f_2} w' \xrightarrow{\theta}_{f_1} y$ for some $w' \in \Gamma^*$ (i.e. the order of the θ-substitutions is changed).

Proof of Lemma 3.4 If $f_1 = f_2$ then the result is trivially true. Suppose then that $f_1 \neq f_2$. The two rewrite rules can be applied in two possible ways:

(a) The rules can be applied on two distinct θ characters in the source string. In this case the result is obviously true since, irrespective of which rewrite rule is applied first, the target string will always be the same.

(b) The rules can be applied in sequence with the second rule applied to a θ character created by the first rule. Again, in this case, the target string will be the same irrespective of the order in which the rules are applied. This is shown below.

$$\theta \rightarrow \theta a\theta \rightarrow \theta a\theta b\theta$$
$$\theta \rightarrow \theta b\theta \rightarrow \theta a\theta b\theta$$

■

Lemma 3.4 shows that any sequence of two θ-substitutions can be applied in *any order*. As a result of this, $\xrightarrow{\theta}{}^*_F$ is always locally confluent. We now show, by induction, that this result also applies to sequences of any finite length. This will be the proof of Lemma 3.3

Proof of Lemma 3.3 Let F be a set of features, $x, y \in \Gamma^*$, and $x \xrightarrow{\theta}{}^*_F y$ Suppose the result is true for any n-chain of θ-substitutions. We therefore have:

$$x \xrightarrow{\theta}_{f_1} x_1 \xrightarrow{\theta}_{f_2} x_2 \xrightarrow{\theta}_{f_3} \ldots \xrightarrow{\theta}_{f_n} y$$

Consider now the $(n+1)$-chain obtained by adding another rewrite f_{n+1}. We therefore get the following $(n+1)$-chain

$$x \xrightarrow{\theta}_{f_1} x_1 \xrightarrow{\theta}_{f_2} x_2 \xrightarrow{\theta}_{f_3} \ldots \xrightarrow{\theta}_{f_n} y \xrightarrow{\theta}_{f_{n+1}} y'$$

By Lemma 3.4 we can swap the last two rules to obtain

$$x \xrightarrow{\theta}_{f_1} x_1 \xrightarrow{\theta}_{f_2} x_2 \xrightarrow{\theta}_{f_3} \ldots \xrightarrow{\theta}_{f_{n+1}} y'' \xrightarrow{\theta}_{f_n} y'$$

We can do this again by swapping the rules $\xrightarrow{\theta}_{f_{n-1}}$ and $\xrightarrow{\theta}_{f_{n+1}}$ to obtain

$$x \xrightarrow{\theta}_{f_1} x_1 \xrightarrow{\theta}_{f_2} x_2 \xrightarrow{\theta}_{f_3} \ldots \xrightarrow{\theta}_{f_{n+1}} y'' \xrightarrow{\theta}_{f_n} y'$$

We can continue this process until $\xrightarrow{\theta}_{f_{n+1}}$ become the first rewrite rule applied in the chain. The argument is sound since, as shown below, when $\xrightarrow{\theta}_{f_{n+1}}$ is moved to any position in the chain, the remaining segments of the chain can be permuted — by the induction hypothesis. All possible permutations of the $(n+1)$-chain are therefore covered.

$$x \xrightarrow{\theta}{}^{i}_{F} x' \xrightarrow{\theta}_{f_{n+1}} x'' \xrightarrow{\theta}{}^{j}_{F} y', \quad i+j=n$$

■

Proof of Theorem 3.2 Suppose that for $x, y \in \Gamma^*$, there is a $z \in \Gamma^*$ such that $z \xrightarrow{\theta}{}^{*}_{F} x$ and $z \xrightarrow{\theta}{}^{*}_{F} y$ Let $\mathcal{C}_1$ denote the chain that transforms z into x and let $\mathcal{C}_2$ denote the chain that transforms z into y. Consider the chain $\mathcal{C}_1 \circ \mathcal{C}_2$ where $\circ$ denotes the union of chains. This chain transforms z into x and then into some z'. The chain $\mathcal{C}_2 \circ \mathcal{C}_1$ transforms z into y and then into some z''. By Lemma 3.3 $z' = z''$ and hence $\xrightarrow{\theta}{}^{*}_{F}$ is confluent. ■

Definition 3.7 (θ-Reductions). *Let F be a finite non-empty subset of Σ^+, and $f \in F$. We define the binary relation $\overset{\theta}{\rightharpoonup}_f \subset \Sigma^* \times \Gamma^*$ to be the set of all ordered pairs of strings over Γ such that $y \in \Gamma^*$ can be obtained from $x \in \Sigma^*$ by replacing (substituting) any f in x by θ. Formally, for any $x \in \Sigma^*$, $y \in \Gamma^*$, $x \overset{\theta}{\rightharpoonup}_f y$ if and only if for some $u, v \in \Gamma^*$, $x = urv$ and $y = u\theta v$. We call $\overset{\theta}{\rightharpoonup}_f$ the **θ-reduction relation induced by (the feature) f**. Also, let*

- $\overset{\theta}{\rightharpoonup}_f^*$ *denote the reflexive transitive closure of* $\overset{\theta}{\rightharpoonup}_f$ *and,*
- $\overset{\theta}{\rightharpoonup}_f^+$ *denote the transitive closure of* $\overset{\theta}{\rightharpoonup}_f$.

*As before, we now abstract further by hiding the string f used in the θ-reduction, insisting only that the string belongs to the given set F. We accomplish this by letting $x \overset{\theta}{\rightharpoonup}_F y$ denote the set of all pairs of strings, $(x, y) \in \Gamma^* \times \Gamma^*$ such that y is obtained from x by replacing some $f \in F$ in x by θ. We call $\overset{\theta}{\rightharpoonup}_F$ the **θ-reduction relation induced by F**. Formally;*

$$\overset{\theta}{\rightharpoonup}_F \overset{\text{def}}{=} \bigcup_{f \in F} \overset{\theta}{\rightharpoonup}_f .$$

Furthermore, let

- $\overset{\theta}{\rightharpoonup}_F^*$ *denote the reflexive transitive closure of* $\overset{\theta}{\rightharpoonup}_F$ *and,*
- $\overset{\theta}{\rightharpoonup}_F^+$ *denote the transitive closure of* $\overset{\theta}{\rightharpoonup}_F$.

Finally, we denote by $\boldsymbol{R}_F$ the string-rewriting system consisting of all the rules of the form (f, θ) where f is a feature.

$$R_F \overset{\text{def}}{=} \{(f, \theta) \mid f \in F\}$$

*We call R_F the **θ-reduction system induced by F**.* □

We note that R_F is not necessarily length-reducing, always length non-increasing, and also always noetherian. Unlike G_F, the string-rewriting system R_F can be non-confluent. Using a similar argument to that used in the proof of Lemma 3.3, we can also show that the rewrite rules can be applied in any order.

3.2.2 Kernel Languages

Definition 3.8 (Kernel Language Description). *A **kernel language description**, which we denote by $\mathcal{K}$, is a 4-tuple $\langle \Sigma, \theta, K, F \rangle$ where;*

(a) Σ is a finite alphabet, θ is a 'placeholder' symbol not in Σ, and Γ is the extended alphabet $\Sigma \cup \{\theta\}$,

*(b) K is a non-empty finite subset of Σ^* we call the **kernel set**, and*

*(c) F, the set of **features**, is a non-empty finite subset of Σ^+.* □

Definition 3.9 (Kernel Language). [Generative Definition]
Let $\mathcal{K} = \langle \Sigma, \theta, K, F \rangle$ be a kernel language description and let G_F and $\xrightarrow{\theta}_F$ be the θ-substitution system and the single-step reduction relation, respectively, induced by F. Also, let K_θ be a finite set of strings over Γ that is indexed by K. $K_\theta \subset \Gamma^*$ is obtained from K by inserting a θ at the front, end, and between each letter of each $k \in K$. Formally, for every $k_i \in K$, $1 \leq i \leq |K|$, the corresponding $k_i^\theta \in K_\theta$ is obtained from k_i as follows:

$$k_i^\theta = \theta k_{i_1} \theta k_{i_2} \ldots \theta k_{i_n} \theta.$$

*The **kernel language, $L^G(\mathcal{K})$, generated by** $\mathcal{K}$ is then defined as follows;*

$$L^G(\mathcal{K}) \stackrel{\text{def}}{=} \bigcup_{k^\theta \in K_\theta} \Delta^*_{G_F}(k^\theta)|_\theta.$$

□

Notes to Definition 3.9. K_θ is the set of strings consisting exactly of the strings in K but with a θ added to the front, end, and between each letter. K_θ, therefore, is indexed by K. Also, for each $k_i^\theta = \theta s_1 \theta s_2 \ldots \theta s_n \theta$, $k_i^\theta \in K_\theta$, $\Delta^*_{G_F}(k^\theta)$ is the set of descendants of k_i^θ modulo the string rewriting system G_F. $\Delta^*_{G_F}(k^\theta)$, therefore, consists exactly of those strings that can be obtained from k_i^θ by replacing a θ with a $\theta f \theta$ for some $f \in F$.

We illustrate with an example. Let $\mathcal{K} = \langle \Sigma, \theta, K, F \rangle$ be a kernel language description where $\Sigma = \{a, b, c\}$, $K = \{cc\}$, and $F = \{ab, ba\}$. We then obtain K_θ from K by adding θ's as described above to obtain $K_\theta = \{\theta c \theta c \theta\}$. We build the strings in the language by applying, non-deterministically, the rules in $R_F = \{\theta \rightarrow \theta ab\theta, \theta \rightarrow \theta ba\theta\}$.

$$\theta c\theta c\theta \rightarrow_{ab} \theta c\theta \mathbf{ab} \theta c\theta \rightarrow_{ba} \theta \mathbf{ba} \theta c\theta \mathbf{ab} \theta c\theta$$

The inserted features are shown in red. We then remove the θ symbols to obtain a string in the language.

$$\theta \mathbf{ba} \theta c\theta \mathbf{ab} \theta c\theta|_\theta = \mathbf{ba}c\mathbf{ab}c$$

The definition of a kernel language we have just presented is *generative* in nature in the sense that it can be used to generate a kernel language from its description. We now proceed to a *propositional* description of a kernel language. A propositional description allows us, for any string $x \in \Sigma^*$, to check whether or not it belongs to some given kernel language. The two definitions, therefore, serve different but complementary purposes. We also show that, indeed, they are equivalent.

Definition 3.10 (Kernel Language). [Propositional Definition]
Let $\mathcal{K} = \langle \Sigma, \theta, K, F \rangle$ be a kernel language description and let R_F and $\xrightarrow{\theta}_F$ be the θ-reduction system and the single-step reduction relation, respectively, induced by F. The ***kernel language, $L^R(\mathcal{K})$, recognized by*** *$\mathcal{K}$ is then defined as follows;*

$$L^R(\mathcal{K}) \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid \Downarrow_{R_F} (x)|_\theta \cap K \neq \emptyset\}$$

where $\Downarrow_{R_F}(x)|_\theta$ is the set of all normal forms of the string x with all occurrences of θ removed and K is the kernel set in $\mathcal{K}$. □

Notes to Definition 3.10. The strings in $L^R(\mathcal{K})$ are the strings in Σ^* whose normal forms modulo R_F (and with the θ's removed) include a string in K. Testing for membership in $L^R(\mathcal{K})$ therefore involves first reducing the strings to their normal forms modulo R_F. The normal forms are strings over Γ^*. We then delete the θ's and check if one of the normal forms is in the set of kernels. If yes, the string belongs to the language.

We now show that although the two definitions play different roles, the languages they describe are the same.

Theorem 3.5. *Let $\mathcal{K} = \langle \Sigma, \theta, K, F \rangle$ be a kernel language description. Then $L^G(\mathcal{K}) = L^R(\mathcal{K})$.*

Proof of Theorem 3.5 In essence, what we have to show is that any number of rewrite rules in G_F can be 'undone' with any number of rewrite rules in R_F and vice versa. This is trivially true since for every rule $\theta \rightarrow \theta f \theta$ in G_F there is a corresponding rule $f \rightarrow \theta$ in R_F. The two rules can be combined to produce $\theta \rightarrow \theta f \theta \rightarrow \theta\theta\theta$. The θ's can then be removed to obtain the original string. Since the rewrite rules in both G_F and R_F can be applied in any order, it follows that any sequence of rewrite rules in G_F

$$\xrightarrow{\theta}_{f_1}, \xrightarrow{\theta}_{f_2}, \ldots, \xrightarrow{\theta}_{f_n},$$

can be 'undone' by the corresponding sequence of rewrite rules from R_F

$$\xrightarrow[\theta]{}_{f_1}, \xrightarrow[\theta]{}_{f_2}, \ldots, \xrightarrow[\theta]{}_{f_n},$$

and vice-versa.

We first show that $L^G(\mathcal{K}) \subseteq L^R(\mathcal{K})$. Let $s \in L^G(\mathcal{K})$. Then there exists a string $s' \in \Gamma^*$ such that $s = s'|_\theta$ and for some $k \in K$, $k \xrightarrow{\theta}{}^*_F s'$. By our argument above, $s \xrightarrow{\theta}{}^*_F k'$, where $k' \in \Gamma^*$ and $k'|_\theta = k$. Then $s \in L^R(\mathcal{K})$. A analogous argument can be used to show that $L^R(\mathcal{K}) \subseteq L^G(\mathcal{K})$ and hence $L^G(\mathcal{K}) = L^R(\mathcal{K})$. ■

From this point henceforth, we shall always consider a set of features and a set of kernels minimal in the sense that a proper subset of either of them does not generate the same language. We now formalize this idea and then proceed to show that if the set of kernels is minimal then it uniquely specifies the kernel language.

Definition 3.11 (Minimal Kernel Language Description).
*Let $\mathcal{K} = \langle \Sigma, \theta, K, F \rangle$ be a kernel language description. $\mathcal{K}$ is called **minimal** if there does not exist $\mathcal{K}' = \langle \Sigma, \theta, K', F' \rangle$ with $F' \subset F$ or $K' \subset K$ and such that $L^G(\mathcal{K}') = L^G(\mathcal{K})$.* □

In other words, a kernel language description is minimal if the removal of a kernel or feature changes the kernel language that is specified by the description.

Lemma 3.6. *Let $\mathcal{K} = \langle \Sigma, \theta, K, F \rangle$ and $\mathcal{K}' = \langle \Sigma, \theta, K', F \rangle$ be two minimal kernel language descriptions. If $L^G(\mathcal{K}) = L^G(\mathcal{K}')$, then $K = K'$.*

Definition 3.12. *For any two sets A and B, SymDiff(A, B) denotes the symmetric difference of A and B.*

$$\mathit{SymDiff}(A, B) \stackrel{\text{def}}{=} (A - B) \cup (B - A)$$

□

Proof of Lemma 3.6 Suppose $K \neq K'$. Then there exists some $k \in \mathit{SymDiff}\,(K, K')$. Without loss of generality, suppose the $k \in K$ (and therefore $k \notin K'$). This implies that $k \notin L^G(\mathcal{K}')$. This is because k is, by definition, irreducible modulo F and since k is not a kernel in K' it cannot belong to $L^G(\mathcal{K}')$. Therefore $L^G(\mathcal{K}') \neq L^G(\mathcal{K})$. This is a contradiction and therefore $K = K'$. ■

The above result leads to the question: *Does a (minimal) set of features uniquely specify a kernel language?* Despite quite significant effort on the author's part we still do not have a definite answer to this question. Our intuition and conventional wisdom during the time this thesis was being written points to a '*yes*' answer but a formal proof has, so far, eluded us. This is because one could, conceivably, build strings from the kernels using one set of features and then reduce the resulting strings back to the kernels modulo another set of features. It can easily be shown however, that if the kernel is the null string, then the set of features has to be unique.

Before proceeding to introduce and discuss TS descriptions of kernel languages we must first have a string edit distance function that can used for this purpose.

3.3 Evolutionary Distance (EvD)

In order to define TS descriptions for kernel languages we need a string-edit distance function that avoids the problems with Generalized Levensthein Distance (GLD). The problems with GLD are discussed in Chapter 4. In brief, GLD is not suitable for the description of kernel languages because it cannot detect features inserted inside other features. Moreover, GLD violates the triangle inequality and, as well shall see in Chapter 4, this presents a problem when dealing with noisy languages. With GLD it is difficult to incorporate mechanisms to handle noise. A feature that is corrupted by the insertion of a single character is considered as noise. This can affect the convergence of the learning algorithm. The main objective behind the development of EvD was to devise a distance function that could be used for TS description of kernel languages. In particular, the new distance function had to:

(a) Be able to detect features inserted inside other features (this is important since strings that contain features inserted into other feature must not be classified as belonging to the language),

(b) be able to handle both confluent and non-confluent kernel languages with or without noise,

(c) be able to handle kernel languages with multiple kernels, and

(d) provide an efficient method for testing for membership.

EvD satisfies all the above requirements. In addition, it can be computed in linear time for confluent languages, is a pre-metric and satisfies the triangle inequality, and can handle relatively high levels of noise.

Various other string edit distance functions were considered. All were rejected because they did not satisfy the above requirements. Amongst the string edit distance

that were considered was *True Edit Distance* (TED). TED is defined to be the least cost, over all possible sequences of weighted transformations, that transforms one string into another. The reader will be forgiven for thinking that this definition is the same as that for GLD. GLD, however does not really return the least cost. This is because it is sometimes the case that one may have to add some characters before computing distance. GLD does not do this. Consider the string *cccabbacccc* and the feature *abcba*. The path of least cost to the string *cccccccc* is to first insert a *c* to obtain *ccccabcbacccc* and then to delete the feature *abcba*. GLD cannot do this and therefore does not really find a sequence of transformations of least cost. TED is, in general undecidable. This is because, with zero-weighted transformations, there is no bound on the number of transformations that need to be inserted in a string before the string is transformed into the other. This means that TED is equivalent to the word problem.

We also considered *Closest Ancestor Distance* (CAD) before finally settling for EvD. CAD is computationally tractable since only deletions are allowed. There is therefore a bound on the number of deletions even if the transformations are zero-weighted. The distance between two strings is defined to be the distance to the closest ancestor using only deletions. In other words, CAD is the the least cost, over all possible sequences of weighted deletions, that transform both strings to a third string — (the *ancestor*).

We illustrate with an example. Figure 3.2, overleaf, shows how the strings *abbab* and *acbca* are transformed into the common ancestor *aba*. The idea of using CAD seemed intuitive but it turned out that CAD cannot properly describe kernel languages. Consider the strings *010* and *01110* drawn from the kernel language that has ε as the sole kernel and the (zero-weighted) features *11* and *010*. The CAD distance is zero

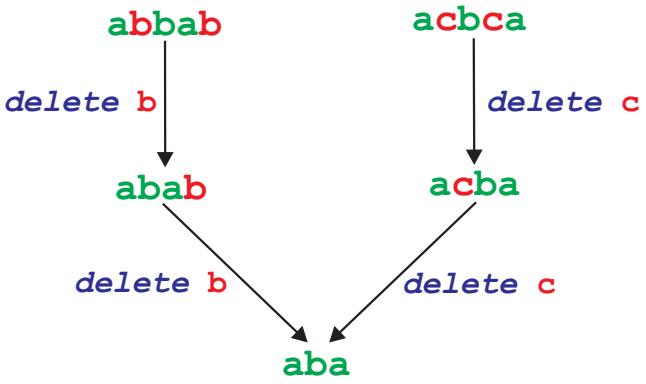


Figure 3.2: Closest Ancestor Distance between the strings *abbab* and *acbca*.

since one need only delete the feature 11 from the second string to obtain *010*. This is not the correct distance. The string *01110* does not belong to the language. CAD, therefore, cannot be used to describe kernel languages. We then decided to consider defining the distance between two strings to be the weighted Levensthein distance between the normal forms. Given two strings and a set of features F we first reduce the two strings to their normal forms modulo F and then compute the distance between the normal forms. The problem with this method is that the resulting distance function is pseudo-metric and violates the triangle inequality. Consider the strings $a =$ '*110*', $b =$ '*0010010*' and $c =$ '*0*'. Figure 3.3, overleaf, shows the pair-wise distances between the normal forms of the three strings.

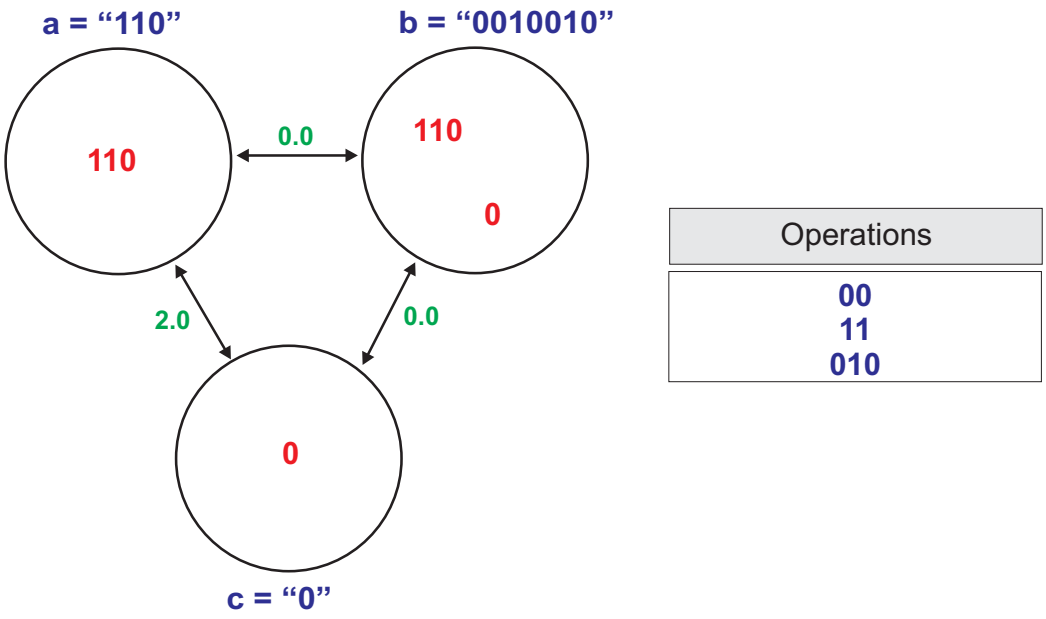


Figure 3.3: Distances between the normal forms of *0*, *110*, and*0010010*.

Note that:

$$d(a, c) = 2,\ d(a, b) = 0, \text{ and } d(b, c) = 0$$

and therefore:

$$d(a, c) = 2 > d(a, b) + d(b, c).$$

This violates the triangle inequality. EvD avoids this problem by passing the set of kernels as parameters to the distance function. The parameters passed to EVD are:

(a) A set of features F. These induce a set of zero-weighted rewrite rules and the θ-reduction relation R_F.

(b) A set of primitive single-character rewrite rules P. These are assigned non-zero weights.

(c) A set of kernels K.

Given two strings s_1 and s_2, $EvD(s_1, s_2)$ is then computed as follows:

- Reduce s_1 and s_2 to their normal forms modulo F using the θ-reduction relation R_F. We denote the two sets of normal forms (with the θ's removed) by $\Downarrow_F (s_1)$ and $\Downarrow_F (s_1)$ respectively.
- Find the WLD[5] distance from $\Downarrow_F (s_1)$ to K, call it d_1. Find the WLD distance from $\Downarrow_F (s_2)$ to K, call it d_2.
- $EvD(s_1, s_2) = d_1 + d_2$

Notice that d_1 =WLD($\Downarrow_F (s_1), K$) and d_2 =WLD($\Downarrow_F (s_2), K$). Both $\Downarrow_F (s_1)$ and K are sets. To find the minimum distance one has to compute the WLD distance over all pairs and take the minimum. EvD satisfies the triangle inequality since, given

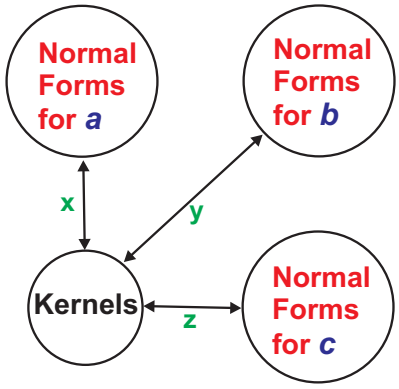


Figure 3.4: Why EvD satisfies the triangle inequality.

three strings a, b, and c with $WLD(\Downarrow_F (a), K) = x$, $WLD(\Downarrow_F (b), K) = y$, and $WLD(\Downarrow_F (c), K) = z$, then, as can be seen in Figure 3.4 above,

$$EvD(a, b) = x + y,\ EvD(b, c) = y + z, \text{ and } EvD(a, c) = x + z.$$

The triangle inequality $EvD(a, c) \leq EvD(a, b) + EvD(b, c)$ holds since:

$$x + z \leq x + 2y + z.$$

[5]Weighted Levensthein Distance.

The θ-reduction relation induced by F, R_F, is used in order to detect features inserted inside other features. In Chapter 5 we describe an efficient procedure for reducing the strings to their normal forms modulo R_F. The computational complexity of EvD might seem, at first glance, to be much worse than that for GLD. One must bear in mind, however, that GLD computation is quadratic to the length of the unreduced strings. For EvD we compute the WLD, with only single-character operations, between the normal forms and the kernels. These are usually much shorter than the original strings.

3.3.1 TS Descriptions for Kernel Languages

The TS description of a kernel language associated with a kernel language description $\mathcal{K} = \langle \Sigma, \theta, K, F \rangle$ is string TS description where:

(a) The set of kernels K is the set of attractors,

(b) the set of rewrite rules are of the form $f \leftrightarrow \varepsilon$ where $f \in F$, and

(c) the distance function is EvD.

Other than the above, a TS description of a kernel language is, in all other respects, a standard TS description as discussed earlier in this chapter. Each kernel is assigned its own delta-neighbourhood value. In kernel language TS descriptions, the rewrite rules induced by the features are always zero-weighted. The *primitive* single-character deletion rewrite rule are non-zero weighted and are used by the EvD function to compute the distance between the normal forms. EvD uses θ-reductions in order to detect features inserted inside other features. Example 3.3 on page 91 is a typical example of a kernel language description.

3.3.2 Some Properties and Applications of Kernel Languages

The set of features of a kernel language can induce a confluent or non-confluent reduction relation. Confluent kernel languages are much easier to work with since each string has got only one normal form and, thus, the normal form can be computed in linear time. EvD computation in the case of a confluent set of features is much faster than GLD. Non-confluent kernel languages are more difficult to handle because of the many normal forms that each string can have. This depends on the number of overlapping features. In order to get some idea of the degree of 'non-confluence' of a kernel language we devised the following measure:

$$\tau = \frac{\nu}{{}^{n}P_2}$$

where ν is the number of overlapping features. This gives us a measure of how 'non-confluent' a kernel language is. If τ is between 0.6 and 1 we say the language is *strongly non-confluent*. We noticed, when running tests on the Valletta algorithm, that a high τ number almost always resulted in an increase in the time taken for EvD computation. This is not surprising given that the number of normal forms of a string increases substantially if the string is reduced modulo a set of features with a high τ number.

The set of kernel languages is, in general, not closed under union. Consider the languages described by $\mathcal{K}_1 = \langle \Sigma, \theta, K, F \rangle$ and $\mathcal{K}_2 = \langle \Sigma, \theta, K', F' \rangle$. Note that both languages are defined over the same alphabet. The union of $\mathcal{K}_1$ and $\mathcal{K}_2$ is not necessarily a kernel language.

If two kernel languages share the same set of kernels but have a different set of features, their union is not a kernel language. Rather, it is a subset of the kernel language $\langle \Sigma, \theta, K, F \cup F' \rangle$. This is because the union will contain strings that contain only one set of features but not both.

On the other hand, if two kernel languages share the same set of features but have a different set of kernels, their union is the kernel language $\langle \Sigma, \theta, K \cup K', F \rangle$.

Some kernel languages have only single-character features. These are called *trivial kernel languages*. This is because the string rewriting system induced by the features is trivial (see Chapter 2). Trivial kernel languages are quite useful. We found out that they can represent binary functions defined over a domain where objects are described by discrete-valued attributes. We call this type of kernel languages *OAV* (*Object Attribute Value*) *Languages*. If fact, we successfully re-encoded the Monk's Problems datasets [122] (See Chapter 7) as strings in a OAV languages and used Valletta to successfully learn the classes. As we shall see in Chapter 7, with trivial kernel languages we can even describe classes defined on these domains which are difficult or cumbersome to describe in either CNF or DNF. Kernel languages, it appears, are not 'trivial' after all.

We also found other real-world applications of kernel languages. The class of n-bit parity binary strings is a kernel language. So is the human ECG language described earlier on page 96. We have no doubt that there are more. Kernel languages can also be used to describe 'features' in chain-code picture languages [2]. In fact, they are rather suited for this application since they can used to remove 'noise' from the features.

Chapter 4

The GSN Learning Algorithm

4.1 Background

The very first application of ETS theory was in grammatical inference. In his Masters thesis, Nigam [92], together with Led Goldfarb who was his Masters supervisor, developed an ETS inductive learning algorithm that learned kernel languages with a single kernel. This work was a continuation of earlier work done by Santoso [107], again in a Masters thesis under Goldfarb's supervision. Goldfarb choose such a class of languages for implementing his ETS theory because the class of kernel languages is an example of a *structurally unbounded environment* (*SUE*). A SUE is, in essence, a set of classes that cannot be enumerated by varying some finite number of numeric, but discrete, parameters. In other words, a learning algorithm for a structurally unbounded environment cannot 'cheat' by hard-coding the environment in the algorithm itself. Goldfarb first proposed the term structurally unbounded environment in [45]. In Goldfarb's own words:

> Having assumed (the) pattern learning as a basic attribute of an intelligent process, one can then proceed to define a minimum requirement for the environment learning process. What is the importance of the 'minimum re-

quirement' for environment? The history of AI and pattern recognition suggests that learning in some environments is "easy" and thus little insight into the most interesting parts of the learning process is gained, while learning in some other environments is simply impossible on computational grounds. The requirement that I propose can be called structural unboundedness of the environment. Informally, an environment is called **structurally unbounded** if no finite sets of "features", or parameters, is sufficient for specifying all classes of events in the environment. I do not have a formal definition of the structural unboundedness that would apply to all environments, but one should stress that the family of event classes in such environments cannot be enumerated by varying some finite number of parameters — i.e. no "closed form" description of the environment exists. Thus, for example, an infinite family, $\{x, y, z \,|\, x+y = 5m, y+3z = n\}_{n,m\in\mathbb{N}}$ is not a structurally unbounded environment.

The class of kernel languages is an example of a structurally unbounded environment. The set of all possible features and the set of all possible kernels cannot be encoded into the learning algorithm. In algorithms such as the *Candidate Elimination* and *ID3* [88] the set of features (in this case the *attributes*) is known beforehand. During learning, these algorithms do not construct new features but describe the class in terms of the initial set of features. The output of the Candidate Elimination algorithm is a *Version Space* [88] and that of the ID3 algorithm is a decision tree. On the other hand, learning in a structurally unbounded environment, such as kernel languages, requires the learning algorithm (or agent) to *discover* (or rather *construct*) the features during the learning process. The learning algorithm cannot be given all the possible features to start with. Kernel languages, therefore, were an ideal environment for which to develop and test the ideas developed by Goldfarb throughout the years. The algorithm described and analysed in this chapter is that developed by Goldfarb, Santoso, and Nigam. This algorithm will henceforth be referred as the *GSN algorithm*. The GSN algorithm is an ETS inductive learning algorithm that learns a kernel language from a relatively small number of positive and

negative examples. The output of the algorithm is a simple form of a TS description. The results obtained by the GSN algorithm seemed nothing less than spectacular to the author when he discussed the algorithm with Sandeep Nigam in 1992. The algorithm appeared to solve what seemed to be a rather difficult combinatorial problem with uncanny ease. Given five or six strings from a kernel language and a similar number of strings not in the language the GSN algorithm, even in the presence of noise, discovers the features of the kernel language in a very short time. The GSN algorithm worked by constructing a set of weighted features such that, under the distance function induced the features, the interdistance between the strings in C^+ is zero or close to zero while the distance between the strings in C^+ and those in C^- is appropriately large. This is called *class separation*. When the author had finished his Masters thesis, he decided that the GSN algorithm was fascinating and was worth a closer look. The starting point of this thesis was therefore the GSN algorithm. The reader is referred to the Masters theses of Nigam [92] and Santoso [107] for a detailed description. The purpose of this Chapter is to give a brief synoptic description of the GSN algorithm and to identify and discuss its limitations and problems.

4.2 Overview of the GSN Algorithm

The actual GSN algorithm is described in detail in Sandeep Nigam's Masters thesis [92]. The reader is referred to Nigam's thesis for a more detailed description and a discussion of the various implementation issues. The main purpose of this section is to outline the salient features of algorithm. The algorithm is relatively simple in concept although its actual implementation was somewhat complex. This was mainly because it was optimized for speed and heuristics were added in order to improve the running time.

Typical input to the GSN algorithm is the following:

(a) A set C^+ of positive training examples,

(b) a set C^- of negative training examples, and

(c) the *learning threshold* value T.

Informally, given the above input, the algorithm tries to minimize the interdistance in C^+ and to maximize the distance between C^+ and C^-. As explained in Chapter 1 this will result in class separation and generalization. Suppose, for instance, that the language to be learned by some ETS learning algorithm is the context-free language $a^n b^n$ and that the input to the algorithm consists of the following training strings:

$\mathbf{C^+}$	$\mathbf{C^-}$
ab	*ba*
aabb	*aba*
aaabbb	*aaaaa*
aaaabbbb	*bbaba*
aaaaabbbbb	*aaabb*
	ababab
	abaabbb

Table 4.1: A training set for the language $a^n b^n$.

If the ETS learning algorithm has the proper inductive preference bias it will discover the following set of weighted transformations:

Transformation	**Weight**
$a \leftrightarrow \varepsilon$	0.5
$b \leftrightarrow \varepsilon$	0.5
$aabb \leftrightarrow ab$	0.0

Table 4.2: The transformations discovered by the ETS learning algorithm.

Notice that if string distance is defined to be the *minimum cost* of transforming any string in Σ^*, where $\Sigma = \{a, b\}$, into any other string then class separation has been achieved. This is because, given the above set of weighted transformations, *any* string in the language $a^n b^n$ can be transformed, with zero cost, into any other string in the same language using only the (zero-weighted) transformation $aabb \leftrightarrow ab$ while the distance between strings in $a^n b^n$ and other strings not in $a^n b^n$ would depend on the number of the non-zero weighted transformations required to transform one string into another. This is depicted in Figure 4.1 below.

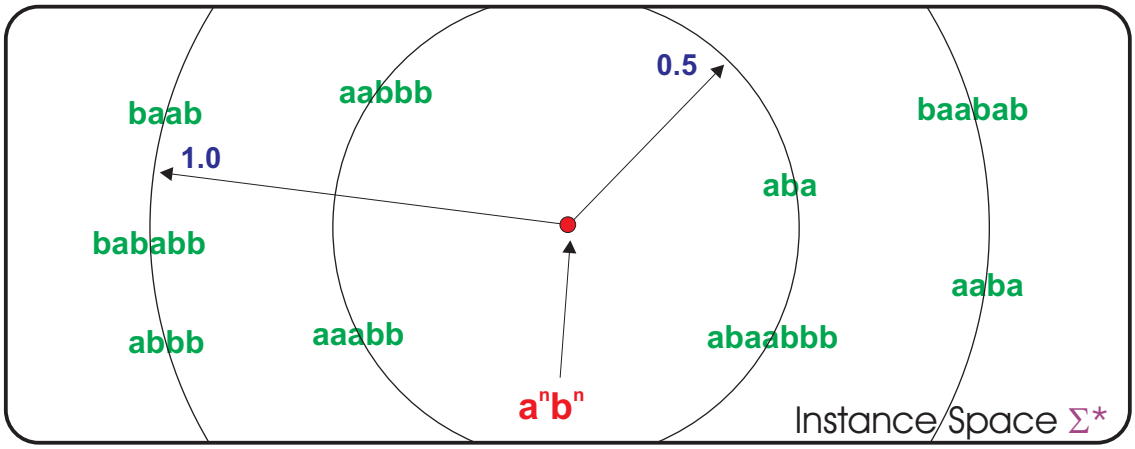


Figure 4.1: The pre-metric space embedding of the language $a^n b^n$.

Figure 4.1 shows the pre-metric space embedding of Σ^* where the distance used is that induced by the transformations in Table 4.2. All strings in the language $a^n b^n$ have a (pair-wise) interdistance of zero and the language $a^n b^n$ therefore appears as a single point in Figure 4.1. The other strings in Σ^* appear proportionately far away from the point that represents $a^n b^n$. The string *aabbb*, for example, can be transformed into the string *aabb* (which belongs to $a^n b^n$) by the single deletion of a character b, i.e. by means of the transformation $b \rightarrow \varepsilon$ which has a weight of 0.5. This is the path of minimum cost that transforms *aabbb* into *aabb*. The string *aabbb* is therefore shown in Figure 4.1 at a distance of 0.5 from the language $a^n b^n$. The

GSN algorithm learns by discovering, or rather *constructing*, a set of transformations (i.e. string insertions, deletions, and substitutions) such that, under the distance function induced by this set of transformations, all strings in C^+ have (pairwise) zero distance and the distance between strings in C^+ and those in C^- is appropriately large. We now proceed to examine the GSN algorithm in more detail and discuss some implementation issues.

The string distance function used by the GSN algorithm is *Generalized Levensthein Distance* (GLD). The main idea behind GLD was to extend Levensthein distance to cater for multi-character transformations and therefore allow for the computation of the minimum-cost edit-path between two strings using multi-character transformations. Conventional Levensthein distance (see Section 2.8) allows only for insertions, deletions, and substitutions of single characters and is therefore unsuitable for the description and learning of kernel languages. GLD uses a very similar dynamic programming technique that runs in $O(nm)$ time, where m and n are the lengths of the two strings, to find the minimum weighted cost of transforming one string into another. GLD is discussed in some detail in Section 4.4 where its main problems and limitations are identified and discussed. In spite of its problems, GLD worked well in the GSN algorithm and did not affect the learning of the examples of kernel languages considered by Nigam in his thesis.

The principal objective of the GSN algorithm is the maximization of the function;

$$f = \frac{f_1}{c + f_2}, \tag{4.1}$$

where f_1 is the minimum distance (over all pairs) between C^+ and C^-, f_2 is the average pair-wise *intra-set* distance in C^+ and c is a small positive real constant to avoid divide-by-zero errors. Formally, if O is a set of transformations, ω the weight

vector associated with O, and d_ω the distance function induced by O and ω, then

$$f_2 = \frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_\omega(s_i, s_j), \text{ where } s_i, s_j \in C^+. \tag{4.2}$$

The value of f_2 is therefore the average distance taken over all possible pairs in C^+. The value of f_1 is computed as follows:

$$f_1 = min\{d_\omega(s_1, s_2) \mid s_1 \in C^+, s_2 \in C^-\}. \tag{4.3}$$

The value of f_1 is therefore the minimum distance taken over all pairs of strings (s_1, s_2) where $s_1 \in C^+$ and $s_2 \in C^-$.

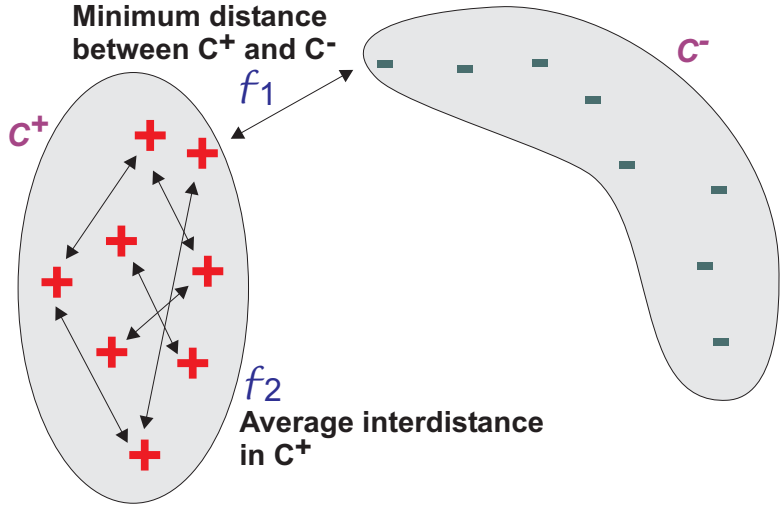


Figure 4.2: The f_1 and f_2 functions.

Figure 4.2 is a depiction of the computation of the f_1 and f_2 functions. During learning, the average intra-set distance in C^+, i.e. f_2, gets progressively smaller until the value of f_2 is zero or very close to zero and class separation is achieved. It must be pointed out that both f_1 and f_2 each require a polynomial number of distance computations. More precisely, the number of distance computations required to compute f_1 is $O(|C^+|\,|C^-|)$ while the number of distance computations required to find the

value of f_2 is $O(|C^+|^2)$. Bearing in mind that the distance computation itself is also quadratic in the length of the two strings, it is easy to see that for longer strings, i.e. longer than 100 characters, and for large training sets, the computation of f_1 and f_2 is, although still polynomial, rather compute-intensive.

The GSN algorithm optimizes the f function by iteratively constructing new transformations from its current set of transformations until finally, under the distance function induced by the final set of transformations, f exceeds the threshold T. This threshold is a positive real constant which is input by the user. The value of T depends of the amount of noise in the training set — the more noise, the lower a value of T is required. The initial set of transformations is usually the set of single-letter insertion/deletion transformations — the so-called *primitive* transformations. The algorithm then builds new transformations from this basic set of transformations and continues this construction process until f exceeds the threshold T. The GSN algorithm itself consists of two main loops; the *learning* loop and the *feature construction* loop — as depicted in Figure 4.3 overleaf. The feature construction loop runs inside the learning loop. The learning loop is a *conditional* loop — it keeps iterating until learning is achieved. At each pass through the learning loop new features are constructed (by the feature construction loop) out of the current set of features by concatenation on the left and right. Suppose, for example that the alphabet is $\{a, b, c\}$ and that the current set of features is ab and ca. The feature construction loop constructs new operations by concatenating, on both sides of the current features, the characters of the alphabet to obtain the following set of new features; $\{ab, ca, aab, bab, cab, aba, abb, abc, aca, bca, cca, caa, cab, cac\}$.

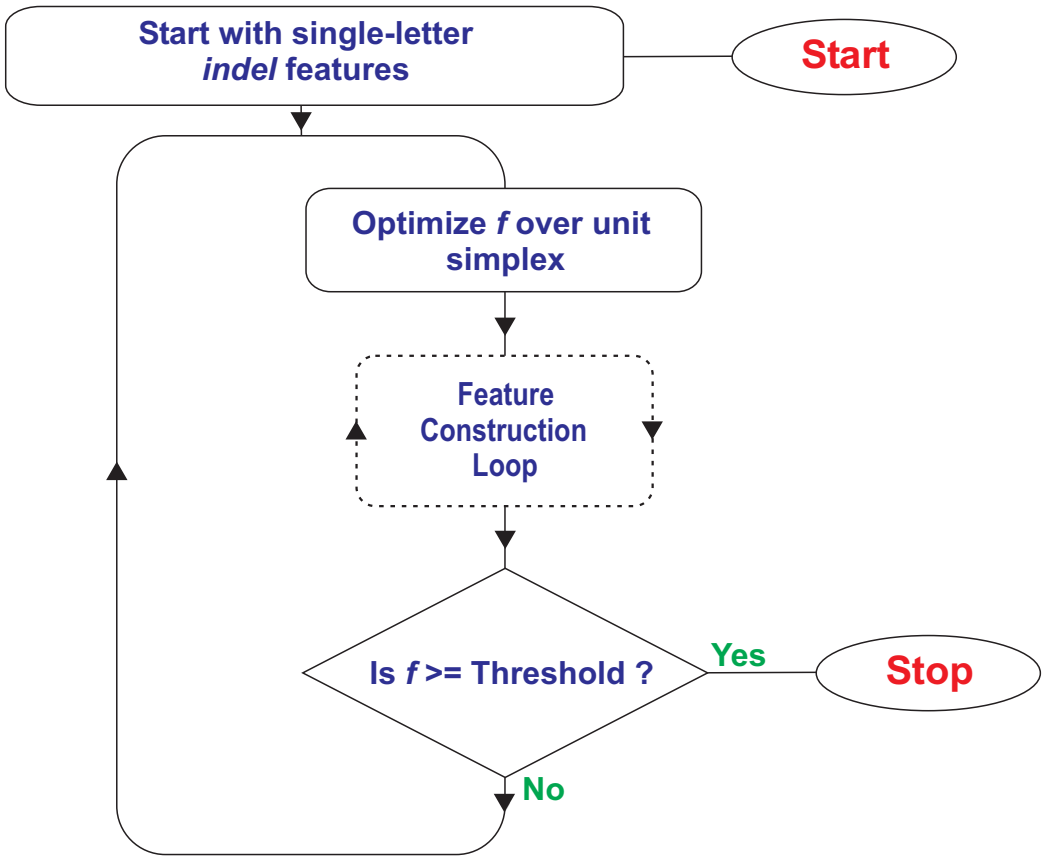


Figure 4.3: Basic architecture of the GSN algorithm.

The algorithm then adds each new feature, one by one, to the current set of features and optimize the weights over the unit simplex. Any of the new features that register a decrease in f_2 are retained and added to the current feature set. The architecture of the GSN is therefore conceptually quite simple. The outer loop controls learning while the inner loop controls feature construction. The unit simplex is the subset of $\mathbb{R}^n$ where the sum of the components of each vector is always 1. Notice that the unit simplex is always of dimension $n-1$ for $\mathbb{R}^n$. Every time a new feature is added the dimension of the simplex is increased by 1. This is depicted in Figure 4.4 below. It must be pointed out that the GSN algorithm does not perform an exhaustive search of the unit simplex but computes the function f at the vertices,

mid-points of the edges and other specific points of the simplex. It finds the set of weights that maximizes f. The sum of the weights of the transformations is always 1. This makes the weights 'compete' against each other. The reader is referred to Nigam's Master thesis [92] for the exact procedure for selecting the weights.

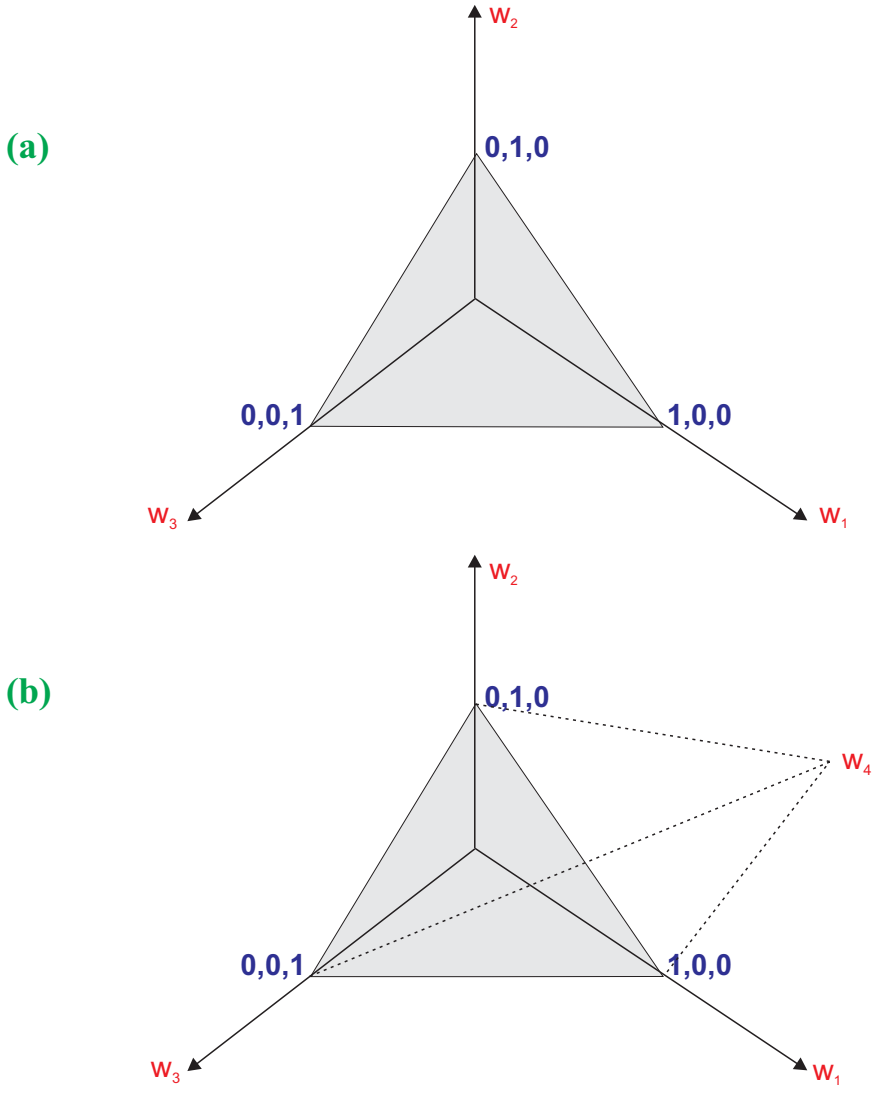


Figure 4.4: Adding a new dimension to the simplex.

GSN — Main Steps		
	(1)	Compute the values of f_2 for each of the m vertices of the basic simplex.
	(2)	Promote (to the next) stage all the 1-letter features for which f_2 is maximal.
	(3)	From the promoted 1-letter features form all possible 2-letter features using left and right concatenation of the letters of the alphabet.
Matching Test	(4)	For each of the constructed 2-letter features check if the feature is present in all the strings of C^+.
	(5)	**IF** none of the features passes the matching test **THEN**, the feature construction stage is completed and the promoted 1-letter feature is added to the current set of features **ELSE** promote (to the next stage) all 2-letter features which passed the matching test.
	(6)	Add each of the promoted *n+1*-letter features one-at-a-time to the current set of features and compute the value of f_2 for the weight vector $(1/m, 1/m, \ldots, 1/m, 0)$.
	(7)	Promote all 2-letter features for which f_2 is minimal.
	(8)	Form from the promoted 2-letter features all possible 3-letter features (by left and right concatenation). Promote 3-letter features on the basis of the matching test in step (4).
	(9)	Continue the formation of new features until the value of f_2 cannot be reduced further.

Table 4.3: The main steps of the GSN ETS learning algorithm.

Table 4.3, reproduced from Nigam's thesis, lists the main steps of the GSN algorithm. Notice that the GSN algorithm *constructs* the features as opposed to *searching* for them. At each stage, the number of new features considered by the feature construction loop is $2.|\Sigma|.|O|$ where Σ is the alphabet and O is the current set of features. The *matching test* is performed in each iteration of the learning loop. This is somewhat compute-intensive. A completely different approach was adopted for the *Valletta* algorithm.

4.3 Results Obtained by the GSN Algorithm

The GSN algorithm was implemented in Modula-2 on a Sun Sparcstation 2 UNIX workstation. Nigam tested the algorithm on numerous examples but only three are documented in his thesis. The training data sets are listed in Table 4.4 below.

GSN — Training Examples			
Problem	$\|C^+\|$	$\|C^-\|$	*Description*
nigam01	5	4	Kernel: *cccc* Features: *ee, ded* Average Length: 2.5 chars No noise, 17.3 seconds runtime. Average length of strings: 28 characters.
nigam01n	5	4	Kernel: *cccc* Features: *ee, ded* Average Length: 2.5 chars $\approx 5\%$ noise, 17.0 seconds runtime. Average length of strings: 28 characters.
nigam02	6	5	Kernel: *ccdbeebdccdbeebd* Features: *dd, ede, cdbebdc* Average Length: 4 chars No noise, 90.2 seconds runtime. Average length of strings: 59 characters.
nigam02n	6	5	Kernel: *ccdbeebdccdbeebd* Features: *dd, ede, cdbebdc* Average Length: 4 chars $\approx 10\%$ noise, 95.2 seconds runtime. Average length of strings: 59 characters.
nigam03	8	5	Kernel: *bddaccaa* Features: *bddabcdadd, bddaaceeabcdd, eadededeabcd* Average length of features: 11.7 chars. No noise, 10 minutes 57.5 seconds runtime. Average length of strings: 103 characters.
nigam03n	8	5	Kernel: *bddaccaa* Features: *bddabcdadd, bddaaceeabcdd, eadededeabcd* Average length of features: 11.7 chars. $\approx 10\%$ noise, 10 minutes 59.0 seconds runtime. Average length of strings: 103 characters.

Table 4.4: The training examples used to test the GSN learning algorithm.

The results in Table 4.4 suggest that the running time of the GSN algorithm

suffers somewhat as the size of the training set, the feature length, and the length of the training strings increases. We shall discuss this issue and give possible reasons for it in the next section. As the line graph in Figure 4.5 shows, there seems to be some correlation between the average string length of the training examples and the running time. Also, the *nigam03* data set had large features (10-13) and given GSN's rather laborious feature construction method, it is no surprise that this data set took well over 10 minutes to train. On the other hand, the presence of noise did not seem to affect the algorithm and training times increased only slightly.

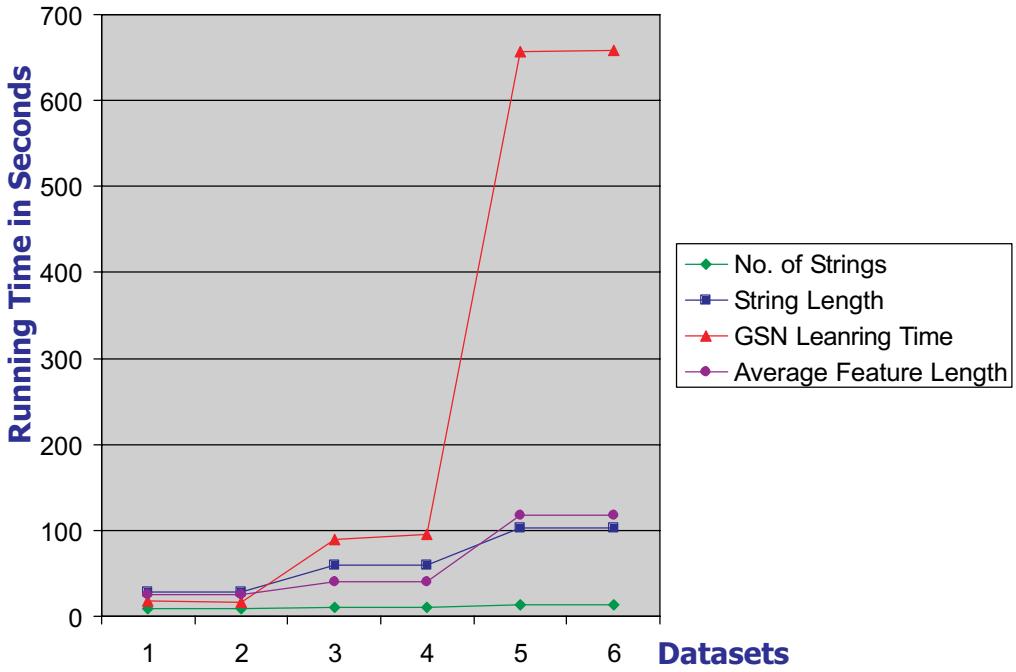


Figure 4.5: Line graphs of the GSN results.

The GSN algorithm did, in the author's opinion, perform rather well given that it was the very first grammatical inference algorithm that implemented the ETS inductive learning model. The GSN algorithm also manifested a healthy robustness

to noise and, in each case, managed to find the correct class description for each problem. This is not to say that there was no room for improvement to the GSN algorithm. In fact, the remainder of this chapter is dedicated to the problems that were identified with the algorithm and also a discussion of the number of ways in which it could be improved. This critique served as a basis for the *Valletta* ETS inductive learning algorithm which is introduced and discussed in Part II of this thesis.

4.4 Problems with the GSN Algorithm

An analysis of the GSN algorithm identified the following problems and/or concerns:

Restricted Learning Domain The GSN algorithm can only learn kernel languages that have exactly one kernel in their description. Multiple-kernel languages cannot be learned by the GSN algorithm and there does not seem to be an obvious way to modify the algorithm so it can also learn multiple-kernel languages. All examples of 'practical' kernel languages that the author came across were all multiple-kernel languages.

Structural Completeness The GSN algorithm assumes that a positive training set C^+ is *structurally complete* if and only if each feature of the unknown language is present, i.e. occurs as a substring, in every string of C^+. This is perhaps too restrictive. Of course, such a restriction can be exploited in order to accelerate the learning process. In fact, the GSN algorithm cannot learn a kernel language unless each feature occurs in every string of C^+. The *matching test* of the GSN algorithm's learning loop ensures that only features that occur in every training string of C^+ are considered. This gives the algorithm a very effective stopping criterion. Each

primitive feature is built (by left and right concatenation) until it no longer passes the matching test. *Valletta* uses a much more relaxed definition of what constitutes structural completeness.

Running time Although it is true that the GSN algorithm did manage to learn all the languages in its training regimen it manifested a sensitivity to large features and to long training strings (i.e. over 100 characters). This is evident from the chart in Figure 4.5. The author only had access to the test results documented in Nigam's thesis and did not have access to a running version of the GSN algorithm. Attempts to re-compile the algorithm on a machine running the Linux operating system proved unsuccessful. However, the author used a popular statistical package to find the strongest correlation between the running time and string length, size of C^+, and the average feature length and it resulted that the strongest correlation was between the running time and the average feature length. It must be emphasized that, since only three test datasets were available, the author is not claiming that the tests are conclusive. The purpose of this exercise was to try to discover which of the three most influenced GSN's running time. The results suggest that the GSN algorithm's depends mainly on the length of the features. This is probably because the feature construction loop considers all candidate features that can be constructed by left and right concatenation of the characters of the alphabet. At each pass of the learning loop, $2.|\Sigma|.|O|$ candidate features are considered where Σ is the alphabet and O is the current set of features. The algorithm then checks to see if each candidate feature exists in every string in C^+ (the *matching test*) and then computes f for each of the candidate features that pass the matching test. Those that pass are then added to O. Feature construction is, therefore, rather involved. Also, computing f involves a quadratic (in the size of the training set) number of distance computation with

each distance computation itself quadratic in the length of the two strings. Although always polynomial, computation of f can quickly become intractable when the size of the training set becomes larger and when the length of the strings in C^+ increases. These problems were addressed in *Valletta* by using a completely new method to compute f and by avoiding simplex optimization completely.

Noise The GSN algorithm did learn kernel languages with noisy training sets but the amount of noise was restricted to a maximum of around 10%. In the GSN data sets the amount of noise was usually very small and Nigam did not distinguish between *feature noise* and *kernel noise*. This topic will be discussed in Chapter 5. In addition, GSN requires that each feature occurs, in uncorrupted form, in *every* string of C^+.

Inductive Preference Bias The inductive bias of the GSN algorithm was not specified by its creators. An analysis of the algorithm revealed that it has a strong preference for shorter features. This is because the algorithm builds the features from the primitive single-letter transformations. It is not clear whether the GSN algorithm actually enumerates the search space (of feature sets) and whether it will always find a TS description for any given training set.

Generalized Levensthein Distance (GLD) GLD has a number of problems. These problems are listed below. Each problem is then discussed in detail.

(a) In some cases it does not return the minimum distance.

(b) As a consequence of the above, GLD violates the triangle inequality. This has important implications.

(c) It is not suitable for kernel languages since it cannot determine if a feature has been inserted inside another feature.

Problem 1 — GLD does not always find the minimum distance.

The main idea behind Levensthein string-edit distance and, for that matter, most string edit distance functions is to find the minimum number of edit operations (transformations) that transform one string into the other or, in the case of weighted edit operations, the edit sequence of minimum cost. It turns out however that, in some cases, GLD does not return the minimum cost. Consider the kernel language over the alphabet $\Sigma = \{a, b\}$ with the empty string ε as the kernel and the string *abbabba* as the sole feature. The set of transformations in the TS description of the language are:

$$\{a \leftrightarrow \varepsilon, b \leftrightarrow \varepsilon, abbabba \leftrightarrow \varepsilon\}$$

and the corresponding weight vector is $(0.5, 0.5, 0)$. Let $x, y, z \in \Sigma^*$ be three strings where:

$$\text{x} = \text{————}\text{abbabba}\text{————}$$

$$\text{y} = \text{————}\text{abbab}\mathbf{a}\text{ba}\text{————}$$

$$\text{z} = \text{————}\varepsilon\text{————}$$

In other words, x is a string that contains the feature *abbabba*, y is the same string but with this feature corrupted by the insertion of an extra character a (shown in red), and z is the string with the feature deleted. Now suppose that x and z belong to our kernel language. The GLD distance from y to z cannot be zero since GLD will delete the corrupted feature character by character. This requires 8 single-character deletions and, hence, $GLD(y, z) = 4$. On the other hand the distance from y to x is only 0.5 since y can be transformed to x through the deletion of the extra character a. This means that the distance from y to z is not returned correctly. This problem has important implications as we shall now see.

Problem 2 — GLD violates the triangle inequality.

An obvious implication of the above example is that GLD violates the triangle inequality. In the above example, $GLD(y, x) = 0.5$, $GLD(x, z) = 0$, and $GLD(y, z) = 4$. Therefore,

$$GLD(y, x) + GLD(x, z) < GLD(y, z).$$

At this point one may well ask, *Do we really need the triangle inequality property anyway*? The answer is yes since without the triangle inequality we cannot guarantee correct classification and, perhaps more critically, correct specification of the kernel language. Let us consider again the above example. Suppose y was an unknown string and suppose we wanted to test it for membership in our kernel language. If we computed the GLD distance from y to z we would have probably rejected the string. If, on the other hand, we computed the distance from y to x we probably would have accepted the string. This shows that it is not possible to specify a noisy kernel language using only a finite set of strings (i.e the kernels) since a noisy string may have a GLD distance of d_1 to one of the kernels but a distance of d_2 to another string in the same language where $d_2 < d_1$ and where d_1 is larger than the δ-neighbourhood value and d_2 smaller. To summarize, if the triangle inequality property is not satisfied, the set of kernels is not enough to specify a noisy kernel language.

Problem 3 — GLD is not suitable for Class Description.

Another problem with GLD is that it is not suitable for class description of kernel languages. By definition, in kernel languages, features cannot be inserted inside other features. In certain cases, GLD cannot detect if a feature is inserted inside another feature. An example of when this problem occurs can be found in Appendix I. This appendix also contains a trace of the GLD computation between two given strings. Because of this problem with GLD, it is possible to classify strings that do not belong

to a particular kernel language as belonging to the language.

Final Class Description When the GSN algorithm terminates it returns a set of features (there can be more than one such set) that allowed f to meet or exceed the threshold T set by the user. The GSN algorithm does not return the kernel of the unknown language. This turns out to be problem since, without the kernel, the TS class description will be incomplete. Nigam assumed that once the set of features, and hence the distance function, is found, classification of unknown strings could be achieved by computing the distance from an unknown string to one of the strings in C^+. In other words, C^+ acts as the set of attractors in the TS description. This, however, only works if the language is noiseless. If $C+$ contains noisy strings then the unknown string might not be classified correctly. This is because the distance between the unknown string and a noisy string in C^+ might be less than the δ-neighbourhood value but the distance between the unknown string and a noiseless string in C^+ might be larger than the same δ-neighbourhood value. This problem is illustrated with a simple example. Suppose that, for some unknown language L that has a kernel k and string distance function d, the GSN algorithm discovers a set of features F from a set of training examples C^+. Suppose we are told that the language is noisy and, with this in mind, we set the delta-neighbourhood value to 1.0 — i.e. any string that has a distance of less than or equal to 1.0 to a string in C^+ will be classified as belonging to the L. Now suppose that x is a noisy string in C^+ with $d(k, x) = 1$, y some other string in Σ^* but not in L, and the distance from x to y is 0.5. This results in y being classified as belonging to the language when this should clearly not be the case since;

$$0 < d(x, k) \leq 1, \quad d(x, y) = 1, \quad \text{and} \quad d(y, k) > 1.$$

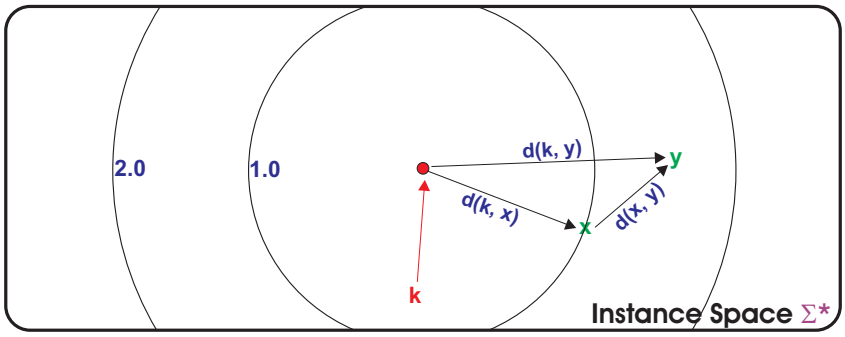


Figure 4.6: Why we need to find the kernel k.

This problem is depicted in Figure 4.6 above. It is therefore clear that if s is any string in C^+, it is not possible to classify an unknown string k based on its distance to s *unless one can be absolutely sure that* s *is noiseless*. The implication of the above is that C^+ cannot be used for the classification of unknown strings when learning is completed unless C^+ contains no noise or unless we can identify the noiseless strings in C^+. The latter can be done by computing the interdistance in C^+ *after* learning is completed and identifying the set of noiseless strings, i.e the set of string with a pair-wise interdistance of zero. A number of experiments conducted by the author proved that this procedure works in practice but only if the number of noisy strings is very small and only if the language has only one kernel. The lesson learnt here is that an ETS learning algorithm for kernel languages must find the kernels and not just the features of the target language.

Although the GSN algorithm did have its problems, it still performed, in the author's opinion, remarkably well. The principle aim of the GSN algorithm was to demonstrate that an ETS learning algorithm was feasible, i.e Goldfarb, Nigam, and Santoso wanted to present a *proof of concept*. This critique of the algorithm and the GLD distance function must be viewed in this context. The *Valletta* ETS algorithm described in Part II of this thesis addresses these problems.

Part II

Valletta: A Variable-Bias ETS Learning Algorithm

Science is a discipline in which even
a fool of this generation should be
able to go beyond the point reached
by a genius of the last.
Scientific folklore, 20th century AD.

Chapter 5

The *Valletta* ETS Algorithm

The aim of this chapter is to introduce and discuss the *Valletta* ETS inductive learning algorithm. Section 1 contains an overview of the algorithm, a listing of the main differences between Valletta and the GSN algorithm, as well as a complete trace of how the algorithm learns a very simple kernel language. The remaining sections of this chapter describe in detail the various parts of the algorithm including the preprocessing stage, string reduction, computing the f function, EvD computation, the search engine, and the kernel selection procedure. These sections also contain a description of the data structures that were purposely developed for Valletta. The reader who is only interested in getting an overall idea of Valletta's architecture and operation may, at first reading, choose to read only Section 1 and skip the remaining sections. The algorithms are described using the excellent *pseudocode* LaTeX style from the University of Waterloo (see Appendix F). The actual code is significantly more elaborate. The main aim is to help the reader understand how each algorithm works. Only the main data structures and main programming constructs are shown. To this end extensive use of diagrams and tables is made. The time and space complexities of each of the algorithms are discussed in Chapter 6.

5.1 Overview

The main objective behind the development of the Valletta algorithm was to investigate the feasibility or otherwise of applying the ETS model to a grammatical inference problem. In particular, to see if and how distance can be used to direct the learning process and also how such an algorithm performs in the presence of noise. The development of Valletta also helped to identify and study the issues that arise in the design and implementation of such ETS algorithms. A secondary objective was to design an algorithm that addressed the problems identified with the GSN algorithm and that can learn a much broader class of kernel languages, with more noise, and more efficiently. The end result is an algorithm that is different from the GSN algorithm although it is still based on the ETS model. Valletta's learning strategy is based on the observation that the set of features that partially specifies[1] an unknown kernel language K must necessarily be a subset of the set of all repeated[2] non-overlapping substrings in the positive training set C^+ — assuming that the strings in C^+ were drawn at random (according to a uniform distribution) from K and that every feature occurs at least twice in C^+. In contrast, the GSN algorithm required that each feature occurred, in uncorrupted form, in *every* string of C^+. Valletta, therefore, has a much more relaxed definition of what constitutes the *structural completeness* of a set of training examples. Valletta was designed from the beginning to learn multiple kernel languages and not just single kernel languages. This is because all examples of naturally occurring kernel languages that were encountered were all multiple-kernel. It turns out that learning multiple-kernel languages is much more difficult than learning single kernel languages. With single-kernel languages we need only find a set of features. On the other hand, with multiple-kernel languages we also have to find the

[1] A kernel language is specified by a set of features, a set of kernels, and a distance function.

[2] I.e. substrings that occur at least twice in C^+.

kernels and, of course, we do not know the number of kernels. Besides the obvious computational complexity this problem also poses an interesting question. Should we find a TS description that minimizes the number of features or the number of kernels? Valletta can be instructed to find TS descriptions that minimize either the number of features or the number of kernels. Valletta has what is called a *variable inductive preference bias*. This means that Valletta allows the user to choose which hypotheses (i.e. TS descriptions) are preferred over others. This is, arguably, an advantage over other learning algorithms that do not allow the user to change the algorithm's preference bias.

Valletta has two main stages. The *pre-processing stage* searches for all repeated substrings in C^+ and stores them in a repeated substring list R_{C^+}. The *learning stage* then finds a set of features (from R_{C^+}) and a set of kernels that gives class separation, i.e. a set of features that optimizes the function

$$f = \frac{f_1}{c + f_2},$$

where f_1 is the minimum EvD (over all pairs) between C^+ and C^-, f_2 is the average pair-wise *intra-set* EvD in C^+, and c is a small positive real constant to avoid divide-by-zero errors. Valletta's learning stage builds a structure, the *search tree*, where each node represents a feature set. Valletta expands this tree only on the basis of f_2 (the reason for this is explained later on in Section 5.3). This means that Valletta's search for the set of features that describes the unknown kernel language K is completely directed by f_2. No other criteria were used to direct the learning process. It would have been relatively simple to add some heuristics to accelerate the search but this was resisted since one of the aims behind Valletta's development was to see if, indeed, distance on its own can direct the search for the TS description of K. Valletta's search strategy is more efficient than that used by the GSN algorithm. At each stage of the learning loop, the GSN algorithm builds new features by left and right concatenation

of the current set of features. This is computationally expensive. On the other hand, Valletta assumes that the TS description of K is a subset of R_{C^+} and considers only subsets from this set. The search space is still, of course, very large. Indeed, the search space is $\mathcal{P}(R_{C^+})$, the power set of R_{C^+}. However, not all elements of $\mathcal{P}(R_{C^+})$ are considered by Valletta. The target set of features must be *substring-free*. In other words, no feature can be a substring of any other feature. Valletta, therefore, selects only the *valid* (i.e. substring-free) feature sets from $\mathcal{P}(R_{C^+})$. Valletta does this efficiently using a data structure called the *search lattice*.

Valletta uses a new string-edit distance function called *Evolutionary Distance* (*EvD*). EvD is suitable for describing kernel languages since it can detect features inserted inside other features. As we saw in Chapter 3, the idea behind EvD is that, given two strings and a set of features F, the distance between two strings can be taken to be the weighted Levensthein distance (WLD) between the normal forms (modulo R_F) of the two strings and a given set of kernels. One important advantage of this technique is that the normal forms are usually much shorter than the actual strings and this results in significantly shorter computation times. The main problem is, of course, how to efficiently reduce the strings to their normal form modulo F. In Valletta this is done using a special data structure called the *parse graph*. EvD works by first building the parse graphs for the two strings and then extracting the normal forms from the parse graphs. The EvD procedure then computes the weighted Levensthein distance between the normal forms and the set of kernels that is passed as a parameter.

5.1.1 How *Valletta* differs from the *GSN* Algorithm

Valletta differs from the GSN algorithm in a number of key areas. The inherent complexity of learning multiple-kernel languages necessitated a completely different

approach to ETS learning. With multiple-kernel languages, we must search for a set of features and a set of kernels not knowing beforehand the number of features or kernels. The main differences between the two algorithms are listed below:

Learning Domain Valletta's *learning domain*, i.e. the class of kernel languages that can be learned by the algorithm, is much broader than that of GSN. Valletta can learn multiple-kernel languages as well as single-kernel languages with both *misclassification* and *random* noise.

Structural Completeness The GSN algorithm requires that each feature must occur in uncorrupted form at least once in each of the strings in C^+. Valletta uses a much more relaxed definition of structural completeness. Valletta requires only that each feature occurs in uncorrupted form at least n times in the entire training set where $n \geq 2$ is a positive integer input by the user.

Inductive Bias Valletta also has a much broader inductive bias than the GSN algorithm. Valletta is a *variable preference bias* algorithm. In other words, the user can, to a certain extent, instruct Valletta to choose certain hypotheses (i.e. TS class descriptions) over others. With Valletta, the user can also choose to minimize the number of features or the number of kernels. The user can also specify the maximum length of the kernels, the minimum length of the features, and various other parameters.

Evolutionary Distance Valletta uses a new string-edit distance function called *Evolutionary Distance* (*EvD*). EvD was inspired both by the theory of string-rewriting systems and by the structure of kernel languages. EvD is a pre-metric and addresses the problems associated with GLD.

Feature-Repair One problem with GSN is that if a feature is corrupted with just one extra character, the whole feature is then considered as noise. The EvD

algorithm has the ability to perform *feature-repair*. All features that are corrupted with a number of extra characters that is not more than 25% of their length are automatically 'repaired' by the EvD algorithm. The gives much quicker convergence for noisy languages. The problem is that searching for corrupted features can be, in general, computationally expensive.

Pre-processing Stage Valletta incorporates a pre-processing stage. The pre-processing stage finds all non-overlapping repeated substrings in C^+ and builds a number of data structures which are then used to guide the learning process. The preprocessing stage defines the *search space* of the learning stage.

Search Strategy The GSN algorithm searches for a TS description that is consistent with the training set by constructing a set of features from an initial set of primitive, i.e. single-character, features by concatenating the current set of features on the left and on the right to obtain new features. Each new feature constructed in this manner is then added to the current set of features and the f function is computed for different points in the unit simplex. Valletta does not 'build' the features since all 'candidate' features are found by the pre-processing stage. Valletta finds the optimal set of features by building a search tree in which each node represents a valid feature set. Valletta chooses only valid feature sets by consulting a data structure called the *search lattice*. This is built from the list of repeated substrings created in the pre-processing stage. Nodes in the search tree are expanded (i.e. by adding child nodes) only on the basis of the value of f_2 (average C^+ inter-distance). The search process is, therefore, entirely distance-driven. The new search strategy allows for a much more efficient search and also allows the user to easily modify the inductive preference bias of the algorithm and thereby give preference to certain hypotheses over others.

5.1.2 How *Valletta* Works — An Example

In this section a very simple kernel language, together with a very small set of training examples from the same language, is used to demonstrate how Valletta's works. The kernel language example is used to demonstrate and illustrate the different procedures used at each stage of the algorithm.

General Description and Architecture

Valletta is an ETS inductive learning algorithm that learns kernel languages with or without noise. Valletta consists of two stages – the *preprocessing stage* and the *learning stage*. In the preprocessing stage Valletta finds all non-overlapping repeated substrings in the C^+ training set. Each string in C^+ is scanned and all non-overlapping substrings, of any length, that occur at least twice in C^+ are identified. The preprocessing stage is required since, unlike the GSN algorithm, Valletta does not build the features incrementally through left and right concatenation of the basic primitive features (see Chapter 4). Valletta's design is based on the premise that the TS description of the language it is trying to learn must necessarily be a subset of the set of all non-overlapping repeated substrings in C^+. All learning algorithms make some assumptions as to the *structural completeness* [33] of the training sets. For example, most grammatical inference algorithms that output a grammar as a class description assume that each production in the grammar of the unknown language L was used at least once in the generation of the positive training set C^+. Analogously, many GI algorithms that learn DFAs assume that each transition in the DFA of the unknown language L was used at least once in generation of C^+ [28]. Valletta only assumes that each feature was used n times in the generation of the strings in C^+ where $n \geq 2$. The value of n is set by the user. Recall that, in the GSN algorithm, Nigam considered the positive training set C^+ to be structurally complete, and therefore valid as a

training set, only if each feature occurred, in uncorrupted form (i.e. without noise), in *every* string of C^+. Valletta's requirements for structural completeness are therefore somewhat more relaxed. This, as can be expected, makes learning harder since the search space of possible TS descriptions is much larger. Assuming that the training set contains at least two occurrences of each feature, it immediately follows that the set of features must therefore be a subset of the set of all non-overlapping repeated substrings in C^+. Valletta's preprocessing stage finds all non-overlapping repeated substrings in C^+ using a structure called the *Global Augmented Suffix Trie* ($GAST$). This structure, described in detail later on, allows us to find all the non-overlapping repeated substrings in C^+ in time quadratic to the size of C^+ in the worst case. Once the GAST is built, Valletta stores all the repeated substrings in a structure called the *search lattice*. The search lattice is used by the learning stage to efficiently find *valid* (substring-free) sets of features. The preprocessing stage, i.e. the construction of the GAST and the search lattice, is performed only once by Valletta. When the preprocessing stage is completed, Valletta proceeds to the learning stage. The learning stage uses the search lattice to identify only valid feature sets and constructs a search tree, *TR*, whose nodes correspond to valid feature sets. Valletta then uses a distance-driven hybrid A*/Beam tree-search technique to find a set of features that maximizes the function;

$$f = \frac{f_1}{f_2 + c},$$

where f_2 is the average pair-wise distance of strings in C^+ and f_1 is the minimum distance between C^+ and C^-. Valletta's search for the set of features that maximizes f is completely directed by f_2. No other criterion is used. No heuristics were added in order to accelerate the search. This was purposely done in order to investigate whether or not f_2 can be used to direct the search (see the thesis objectives in Chapter 1).

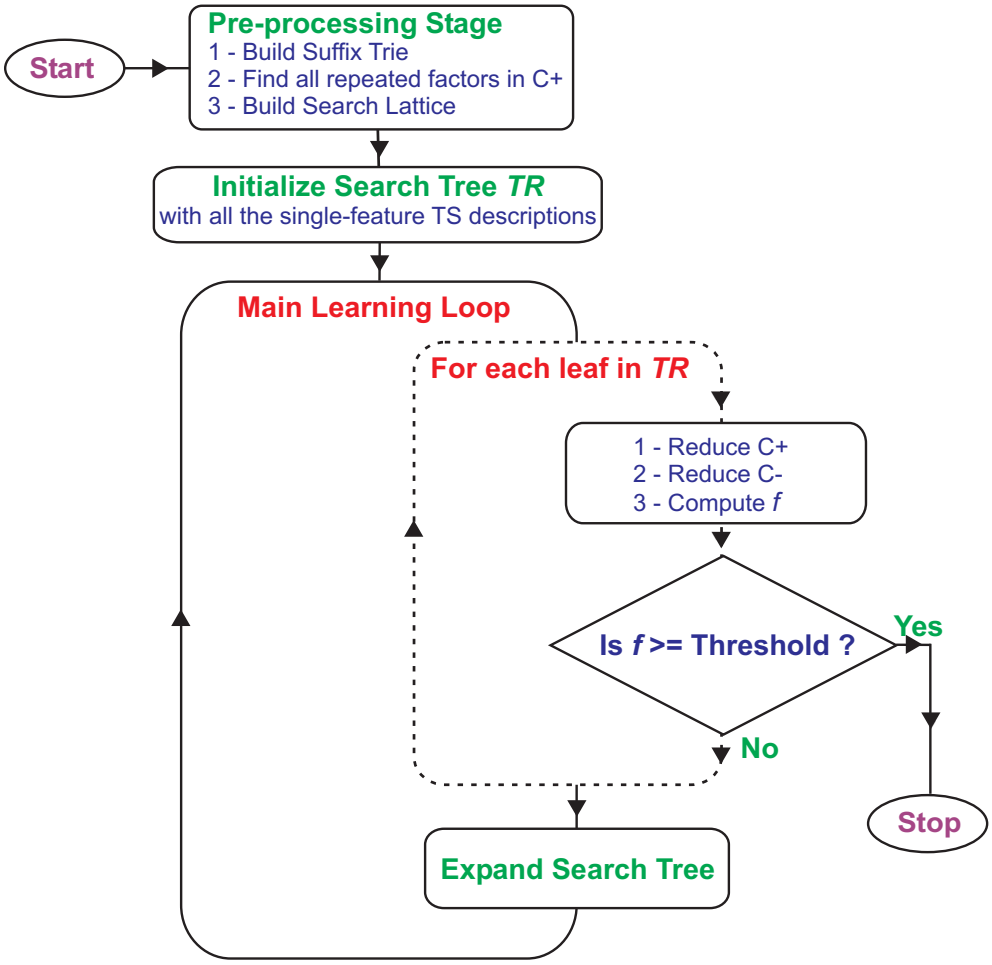


Figure 5.1: High-level flowchart of *Valletta* showing the main loops.

At each stage of the learning process Valletta decides on the best way to expand TR, i.e. which feature sets to consider, by expanding those nodes (feature sets) for which f_2 is minimal. It must be pointed out that, given the set R_{C^+} of all the repeated substrings in C^+, the search space of the algorithm, i.e. the set of all valid feature sets, is a proper subset of $\mathcal{P}(R_{C^+})$ and is usually very large. Figure 5.1 shows a high-level flowchart of the Valletta algorithm showing only the main loops. Let us now consider a set of training examples from a very simple kernel language, K_1. The training examples are used to present a 'trace' of Valletta as it tries to find a TS class

description. K_1 is defined on the alphabet $\{a, b, c\}$ as follows:

$$R = \begin{bmatrix} \text{a} & \leftrightarrow & \varepsilon \\ \text{b} & \leftrightarrow & \varepsilon \\ \text{c} & \leftrightarrow & \varepsilon \\ \text{ab} & \leftrightarrow & \varepsilon \end{bmatrix}, \qquad \omega = \begin{bmatrix} 0.33 \\ 0.33 \\ 0.33 \\ 0 \end{bmatrix}, \qquad A = \{(cc, 0)\}, \qquad \phi(\nu_\pi) = \sum_{r \in \pi} w_r.$$

where ϕ is defined to be the cost of the *evolutionary path of least cost*, i.e. EvD, between two strings. Evolutionary string distance (EvD) was defined and discussed in Chapter 3[3]. An algorithm for computing EvD is presented later on in this chapter. Note that the primitive single-characters are assigned a non-zero weight and are used to handle noise. The transformation $ab \leftrightarrow \varepsilon$ associated with the feature ab is assigned a zero-weight and is used to transform any string in the language into any other string in the language with zero cost. The language K_1 consists of all strings that can be

$\mathbf{C}^+$	$\mathbf{C}^-$
abccab	*ccbcba*
cabc	*caabcc*
cababc	*bcbcba*

Table 5.1: The training set for the language K_1

obtained from the kernel cc by inserting, anywhere, and any number of times, the sole feature ab. Table 5.1 shows a very small training set for K_1. C^+ consists of three strings drawn from K_1 while C^- consists of three random strings. The sizes of the training sets and the lengths of the strings were chosen to be as small as is reasonably possible. In the interest of simplicity, it is assumed that a positive training set for K_1 is structurally complete if every string contains the feature ab at least once.

[3]See Chapter 3 for definitions of ϕ, ν, and π.

The Pre-processing Stage

Valletta's pre-processing stage accepts as input C^+ and builds three data structures;

(a) The **Global Augmented Suffix Trie** (**GAST**),

(b) the **Repeated-Substrings List**, $\mathbf{R_{C^+}}$, and

(c) the **Search Lattice** $\mathbf{L_{C^+}}$.

As explained in the beginning of this chapter, Valletta's learning stage uses these data structures to efficiently search for the set of features that maximizes the function f.

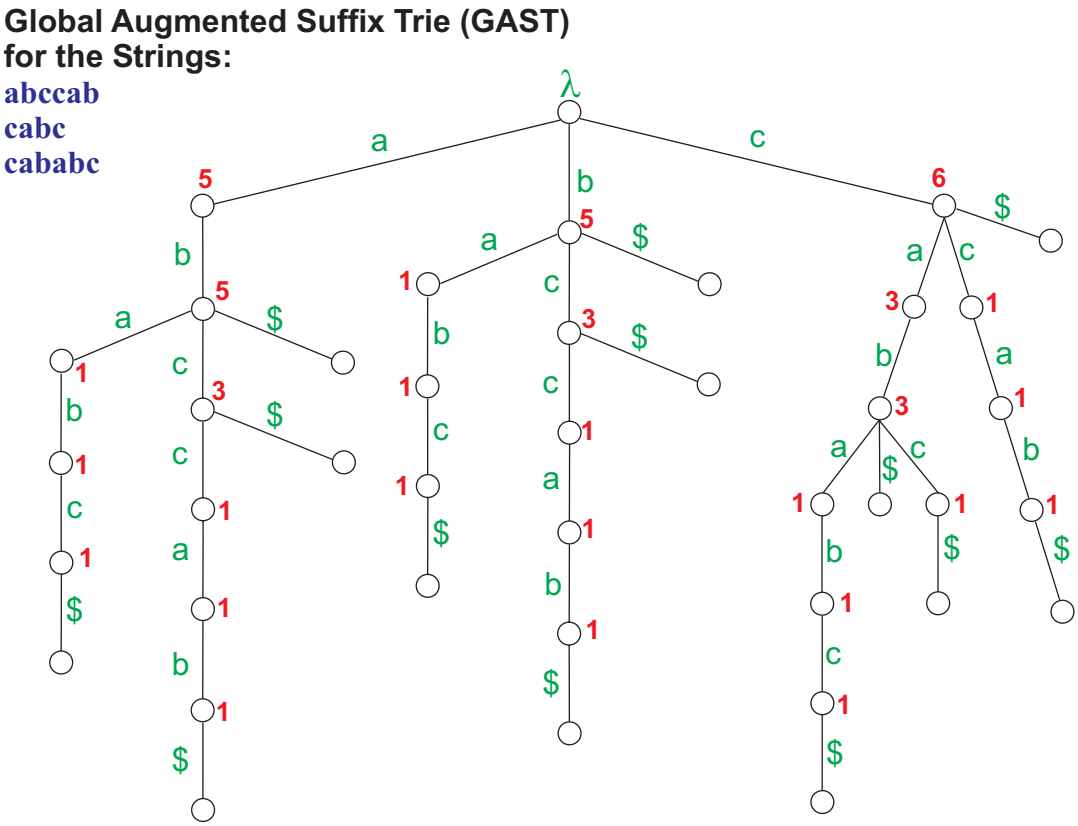


Figure 5.2: The GAST built from the strings: *abccab*, *cabc*, and *cababc*.

The pre-processing stage starts by reading in C^+. It then builds the *Global Augmented Suffix Trie* or *GAST*. A GAST is a tree data structure that stores all the

possible suffixes of a given set of strings. It is based on the concept of a suffix trie [57]. The GAST data structure, as well as the algorithm for its construction, are defined and discussed in the next section. In essence, a GAST for a set of strings S consists of a tree with labelled edges where every path from the root to a node (including the leaves) forms a string (obtained by concatenating the edge labels of the path) that is a suffix of some string in S. The GAST built from C^+ is shown in Figure 5.2 above. The GAST also stores, at each node, the number of times the substring (obtained by concatenating the edge labels of the path from the root to that node) occurs in S. The \$ symbol that appears at each leaf of the GAST in Figure 5.2 plays a special role and can be ignored by the reader for now.

After Valletta's preprocessing stage builds the GAST, it extracts, using a simple recursive tree-traversal algorithm, all the non-overlapping repeated substrings that occur n times (for some fixed positive integer n input by the user). The substrings are stored in the *Repeated Substrings List* R_{C^+}. In our case R_{C^+} contains all the non-overlapping repeated substrings that occur at least three times in C^+.

Substring	**Repetitions**	**Occurrences**
a	5	3
b	5	3
c	6	3
ab	5	3
bc	3	3
ca	3	3
abc	3	3
cab	3	3

Table 5.2: The Repeated Substrings array for the strings: *abccab*, *cabc*, and *cababc*.

In Table 5.2, the column with the heading **Repetitions** holds the total number of times the corresponding substring was found in C^+ while the column with the heading **Occurrences** stores the total number of distinct strings in C^+ that substring

occurred in.

Notice that it is assumed that C^+ is structurally complete. In other words we are assuming that each feature in the TS class description of K_1, which is unknown to Valletta, occurs at least once in every string of C^+. We can therefore can be sure that the set of features in the target TS class description is a subset of the strings shown in Table 5.2. No other information about the class is provided to Valletta. Valletta does not 'know' what the set of features or the set of kernels are. Valletta also does not 'know' that the TS class description of K_1 has only one kernel and only one feature. Valletta must discover the class description of K_1 on its own. In our example K_1 is a very simple language and the reader should have no difficulty inferring the class description directly from the given training sets. The main purpose of K_1, however, is to demonstrate how Valletta works.

The last task performed by the pre-processing stage is the construction of the *search lattice* from R_{C^+}. The search lattice, which we shall denote by L_{C^+} is the lattice $(R_{C^+}, \triangleright)$ where $\triangleright$ is the covering length-lexicographical relation on Σ^*. The reader is spared from the formal details for now. The search lattice is discussed in detail in Section 5.2.1. Figure 5.3, overleaf, shows the search lattice constructed from the strings in Table 5.2. One interesting property of the search lattice is that, for any given node x, all the descendants of x always include x as a substring. If x belongs to a feature set, then none of the descendants of x can be in the same feature set. This property allows Valletta to efficiently choose *valid* feature sets, i.e. sets of strings from R_{C^+} such that no feature is a substring of another in the same set. In essence, every valid feature set in R_{C^+} is an *anti-chain*[23] in the search lattice.

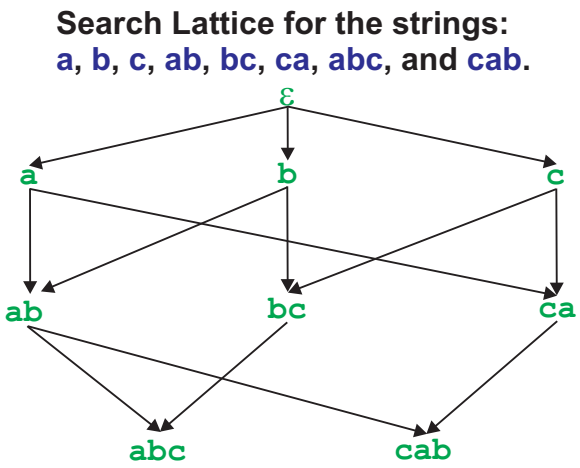


Figure 5.3: The search lattice for the strings: *abccab*, *cabc*, and *cababc*.

The Learning Stage

The input to the learning stage is the set of repeated substrings R_{C^+} and the search lattice L_{C^+}. Recall that, as previously explained, R_{C^+} completely specifies the search space of the learning stage. This is because we know that the set of features in the TS description of K_1 must be a subset of R_{C^+} — not just any subset but, rather, a substring-free subset of R_{C^+}. The learning stage must therefore enumerate, in some way, the substring-free subsets of R_{C^+} and find the subset that gives class separation. One must point out that the set of all substring-free subsets of R_{C^+} is usually very large and a brute-force search is therefore out of the question. Valletta's search strategy is that of finding a TS description consistent with the training examples by searching through the set of all valid feature sets in R_{C^+} using a distance-driven hybrid A*/Beam [93] search technique. Valletta enumerates the search space by constructing a search tree, denoted by TR, and shown below in Figure 5.4. The search tree shown in Figure 5.4 is the actual search tree built by procedure *ETSSearch* from the strings in Table 5.2. Each node in the search tree is labelled with a string from R_{C^+}. The feature set associated with any node in the tree is the set of strings from

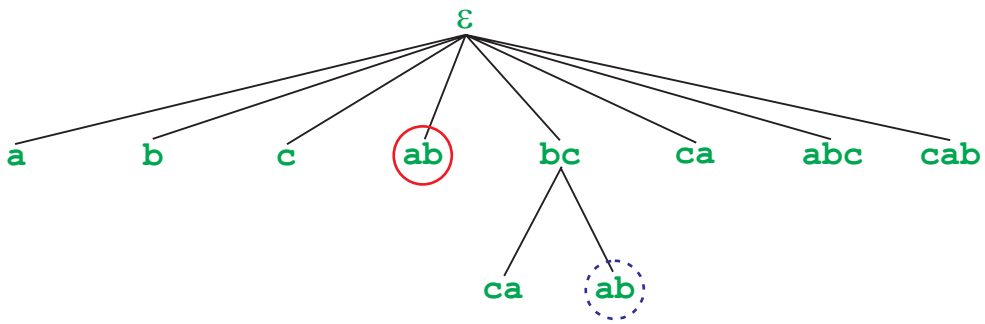


Figure 5.4: The search tree built by *ETSSearch*.

R_{C^+} that are node labels in the path from that node to the root. For example, the node shown circled with a broken blue circle in Figure 5.4 is associated with the feature set $\{bc, ab\}$, which is precisely the set of strings from R_{C^+} that are node labels in the path from that node to the root. Each node in the search tree, therefore, represents a feature set and the aim is to build a search tree such that, in the shortest time possible, a node is found that represents the set of features in the TS description of the unknown target language. Search tree construction proceeds as follows. *ETSSearch* initially creates a search tree that has exactly one level with a node for every string in R_{C^+}. *ETSSearch* then computes that value of f for each leaf. If, for all of the leaves, the value of f does not exceed the threshold, *ETSSearch* then expands the tree by adding new leaves for those nodes for which f_2 is minimal. In our example, the value of f exceeded the threshold for the node labelled ab shown circled in red. This is because ab is the sole feature for the language K_1 and so, almost immediately, *ETSSearch* discovered the correct TS description of K_1. In practice, however, it is usual that the search tree grows to be many levels deep before a node that represents the correct feature set is found.

The above is a basic overview of how Valletta learns. We have seen how Valletta finds the features of the target language but not how the kernels are found. The actual learning process is discussed in detail in Section 5.3. A number of points,

however, require some clarification:

Firstly, the search process is completely *distance-driven*. The decision on which leaves are expanded is taken solely on the basis of f_2, i.e the average pair-wise EvD in C^+. During each iteration of the learning loop, *ETSSearch* expands those nodes for which f_2 is minimal. It then computes the value of f for each of the new leaves.

Secondly, when a node is chosen for expansion (i.e. by adding new leaves), the labels of the new leaves are chosen from R_{C^+} after consulting the search lattice L_{C^+}. This is to ensure that every node represents a valid, i.e. substring-free, feature set. Consider again the tree in Figure 5.4. When the node labelled *bc* is expanded, we want to ensure that the leaves are not labelled with strings in R_{C^+} that are substrings of *bc* or vice versa. The search lattice is used to prevent this from happening.

Thirdly, Valletta employs a search strategy that is based on finding the set of features of the unknown target language rather than the set of kernels. We have seen that each node in the search tree represents a set of features and that Valletta computes the value of f for each node and then decides which nodes to expand based on the value of f. Computing the value of f involves the computation of pair-wise EvD both within C^+ as well as between C^+ and C^-. Unlike the GSN algorithm, Valletta does not use GLD but, rather, EvD which was introduced in Chapter 3. Recall that GLD has only one parameter, the set of features (i.e the transformations), but EvD has two parameters — the set of features F and the set of kernels K. Computing the EvD between two strings involves reducing both strings to their normal forms modulo $\rightarrow_F^*$ (i.e. the reduction relation induced by F), and then computing the weighted Levensthein distance between the normal forms and the set of kernels. The problem here is, of course, *where do we get the set of kernels from*? The answer to this question is that, given a feature set F, the set of kernels must be a subset of the normal forms of C^+ modulo F. Informally, this is because,

given a finite set of strings from a kernel language K, if we delete all occurrences of all features from these strings, we should end up with the kernels of K. This sounds simple enough except that in practice we do not know how many kernels K has in its description. The presence of noise in the strings serves to complicate matters since the normal forms of C^+ will also include noise and it will not be apparent which is a 'clean' kernel and which it not. Moreover, if F is not confluent things get even worse since every string in C^+ can have many normal forms and it is, in general, not clear which of these are the kernels. The problem of identifying the kernels from the normal forms of C^+ is called the *kernel selection problem* and is discussed in detail in Section 5.4 where an algorithm that performs this task is also discussed. This problem did not arise with the GSN algorithm since Goldfarb and Nigam did not consider kernel languages with multiple kernels. Valletta computes the value of f for a set of features F as follows:

(a) All strings in C^+ and C^- are reduced to their normal forms modulo the θ-reduction relation R_F. This process is called *reduction*. If R_F is non-confluent each string may have many distinct normal forms.

(b) All pair-wise WLDs between all the normal forms are computed and stored in the *distance matrix*. The value of f_3 is then computed from the distance matrix. The function f_3 is defined to be the average NFD (normal form distance) in C^+. The normal forms used in f_3 computation are promoted as '*candidate kernels*'

(c) From amongst the candidate kernels, Valletta finds a set of kernels for C^+ that minimizes f_2. This process is called *kernel selection*.

(d) Finally, the set of kernels obtained in (b) is used to compute f_1 and hence f.

It must pointed out that reduction, f_3 computation, and kernel selection are relatively complex procedures. These procedures are discussed separately, and in some detail, in

Section 5.5 and Section 5.4 respectively. This section concludes with a brief synoptic overview of these procedures.

Reducing C^+ and C^- to their Normal Forms

Reducing all strings in C^+ and C^- to their normal forms modulo a set of features F involves deleting all occurrences of all the features in F from each of the strings. When the normal forms are computed we can then proceed to the process of kernel selection. The problem of reduction sounds simple enough but turns out to be a rather difficult combinatorial problem. We must be able to detect features inserted inside other features and, if F is non-confluent and therefore contains features that overlap with each other, there may be many different ways in which we can delete the features from a given string. To make things even more interesting we also have to accommodate noisy strings.

Valletta uses a specially developed data structure called a *parse graph*. The parse graph of a string is an acyclic directed graph in which nodes represent positions in the string and the edges are labelled with occurrences of features within the strings. Figure 5.5, overleaf, shows the parse graphs built from the C^+ training set of the kernel language K_1. A parse graph contains two special nodes labelled S and \$ corresponding to the beginning and the end of the string respectively. All the other nodes are labelled with a positive integer corresponding to a position in the string (counting from the left). The edges are labelled with substrings that contain features. All edges, except those incident to S or \$, start or end at nodes that represent a position in the string where a feature has been found or where a feature ends. To obtain all the normal forms of the string we enumerate all the paths in the parse graph. In the worst case, the number of paths can be exponential in the size of the graph. The parse graph gives us all the normal forms of the string and also allows us

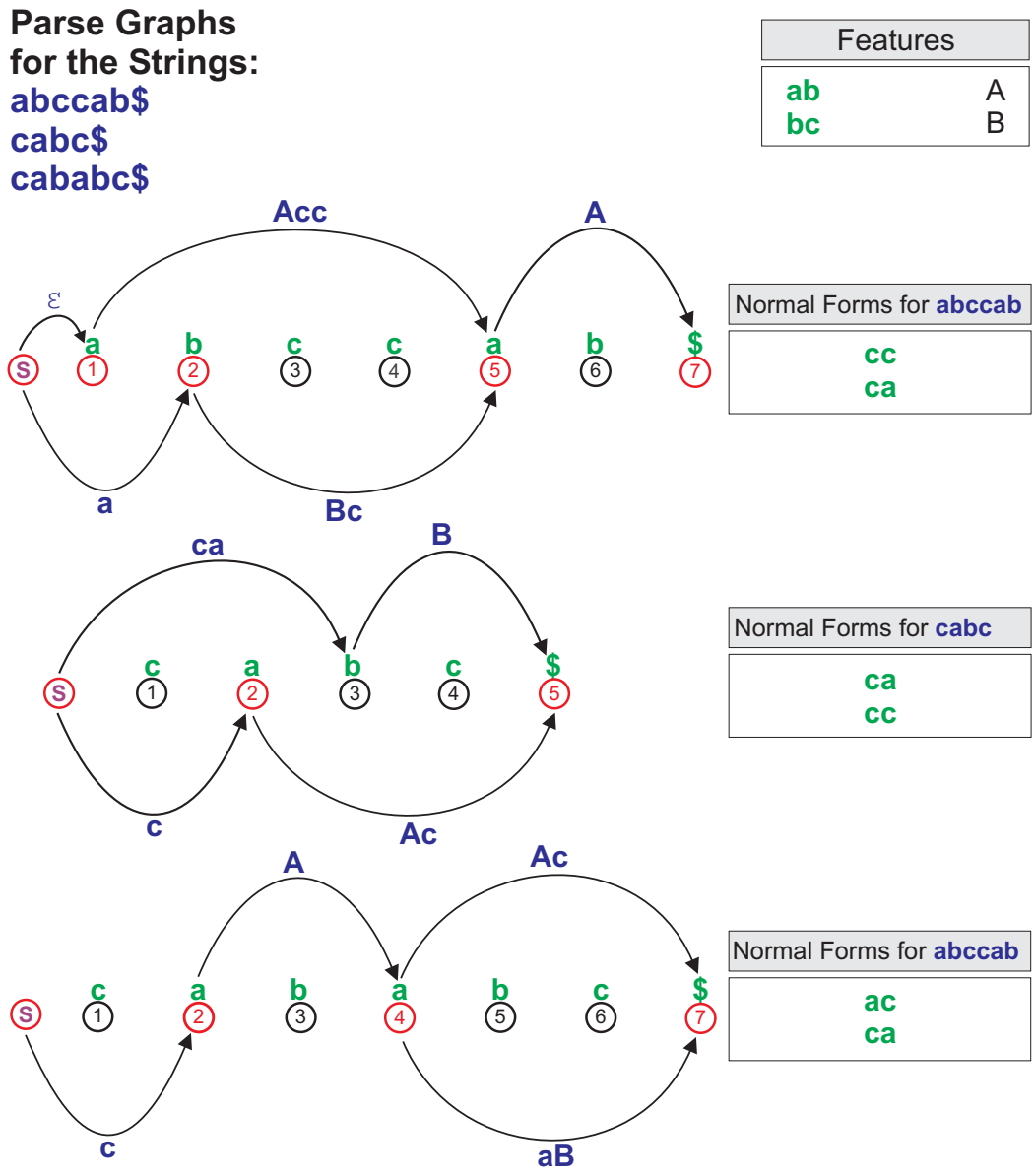


Figure 5.5: The parse graphs for the strings: *abccab*, *cabc*, and *cababc*.

to avoid reducing strings that have features inserted inside other features. A formal discussion of parse graphs, their properties, their construction, and the associated algorithms is found in Section 5.5. Section 5.5 also contains a discussion of how parse graphs can be used to handle noisy strings. A technique called *feature repair* is used to 'repair' features that are corrupted by noise. This technique greatly increases

Valletta's tolerance to noisy training sets.

5.1.3 Kernel Selection

When the strings in C^+ and C^- are reduced to their normal forms, the next step is to find a set of kernels that can be used as a parameter to the EvD function. Once a set of kernels is found, we can use EvD to compute f_2, the average pair-wise distance in C^+, and f_1, the minimum distance between C^+ and C^-. Having computed f_2 and f_1 we can then compute f. The kernel selection procedure is based on the premise that, given the set of features in the TS description of an unknown kernel language, the set of kernels must be a subset of the set of normal forms of C^+ modulo the set of features. The problem is that, when the language is not confluent, each string in C^+ can have many normal forms and if noise is present the normal forms will be corrupted. It will therefore not be clear which of the normal forms of C^+ are the kernels of the language that we are trying to learn and which are not. When we reduce C^+ we obtain a collection of sets of normal forms, $\mathcal{N} = \{\Downarrow_F (s_1), \Downarrow_F (s_2), \ldots, \Downarrow_F (s_n)\}$ where $s_1, s_2, \ldots, s_n$ are the strings in C^+ and for any $s_i \in C^+$, $\Downarrow_F (s_i)$ denotes the normal forms of s_i modulo the set of features F. Given this collection of sets of normal forms we have to find a set of kernels that minimizes f_2. We know that each $\Downarrow_F (s_i)$ must contain at least one kernel, except when the language is noisy in which case $\Downarrow_F (s_i)$ might not contain an uncorrupted kernel. In general, each of the normal forms can be a kernel. It is obvious that, when choosing our set of kernels, it makes sense, in general, to choose strings that are normal forms of as many strings in C^+ as possible. This is because we assume that the number of strings in C^+ is always much greater than the number of kernels in the TS description of the unknown language. The sets of normal forms in $\mathcal{N}$ will therefore have strings in common. One would therefore expect that one could perform some sort of statistical analysis of the normal

forms of C^+ and thereby be able to choose the most likely kernels. Although this is true in concept, the kernel selection problem actually turns out to be NP-Hard (see Appendix H). The kernel selection problem, as posed, is a subproblem of the *Minimum Hitting Set* problem [35, page 222]. In the minimum hitting set problem we are given a collection C of subsets of a set S and then asked to find the smallest subset $S' \subseteq S$ such that S' contains at least one element from each subset in C. The minimum hitting set problem is provably NP-Hard (transformation from *Vertex Cover*). As with many NP-Hard problems it is often the case that one can find a good approximation algorithm.

Figure 5.6 shows the collection of normal forms obtained after reduction of C^+ of the kernel language K_1. The set of features used for reduction is $\{ab, bc\}$. In

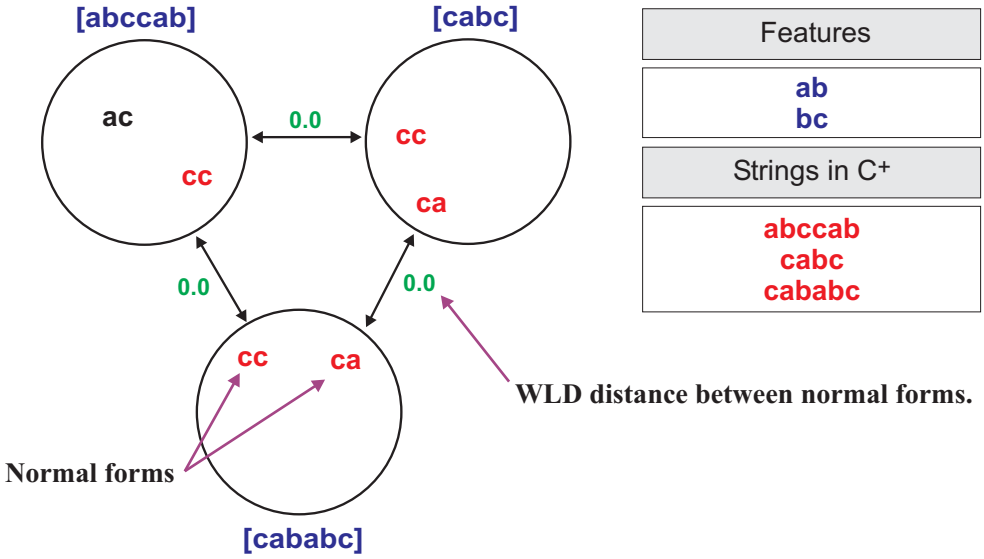


Figure 5.6: *Valletta*'s kernel selection procedure.

Figure 5.6, for example, the string cc is a normal form of all the strings in C^+ while the string ac (shown in black) is a normal form of only one string. It is therefore prudent to assume that cc is much more likely to be a kernel than ac. In fact, as can

be seen in Figure 5.6, if *cc* is chosen as the kernel, the average EvD pair-wise distance in C^+, and hence f_2 would be 0. Valletta's kernel selection algorithm, described in Section 5.4, is an approximation algorithm that makes use of the function f_3 — the average normal form distance (NFD) between the strings in C^+. The NFD distance between two strings is defined to be the minimum distance (over all pairs) between the normal forms of the two strings. This is depicted in Figure 5.7.

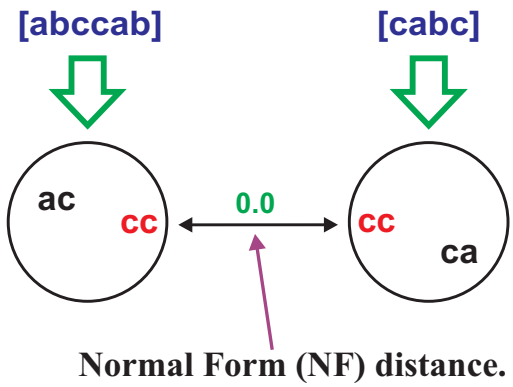


Figure 5.7: Normal Form Distance (NLD).

As we shall see in Section 5.4, f_3 gives us measure of the '*entropy*' within C^+ and is effectively used to select the 'candidate kernels' from amongst the normal forms in $\mathcal{N}$.

5.1.4 How *Valletta* Works — In Pictures

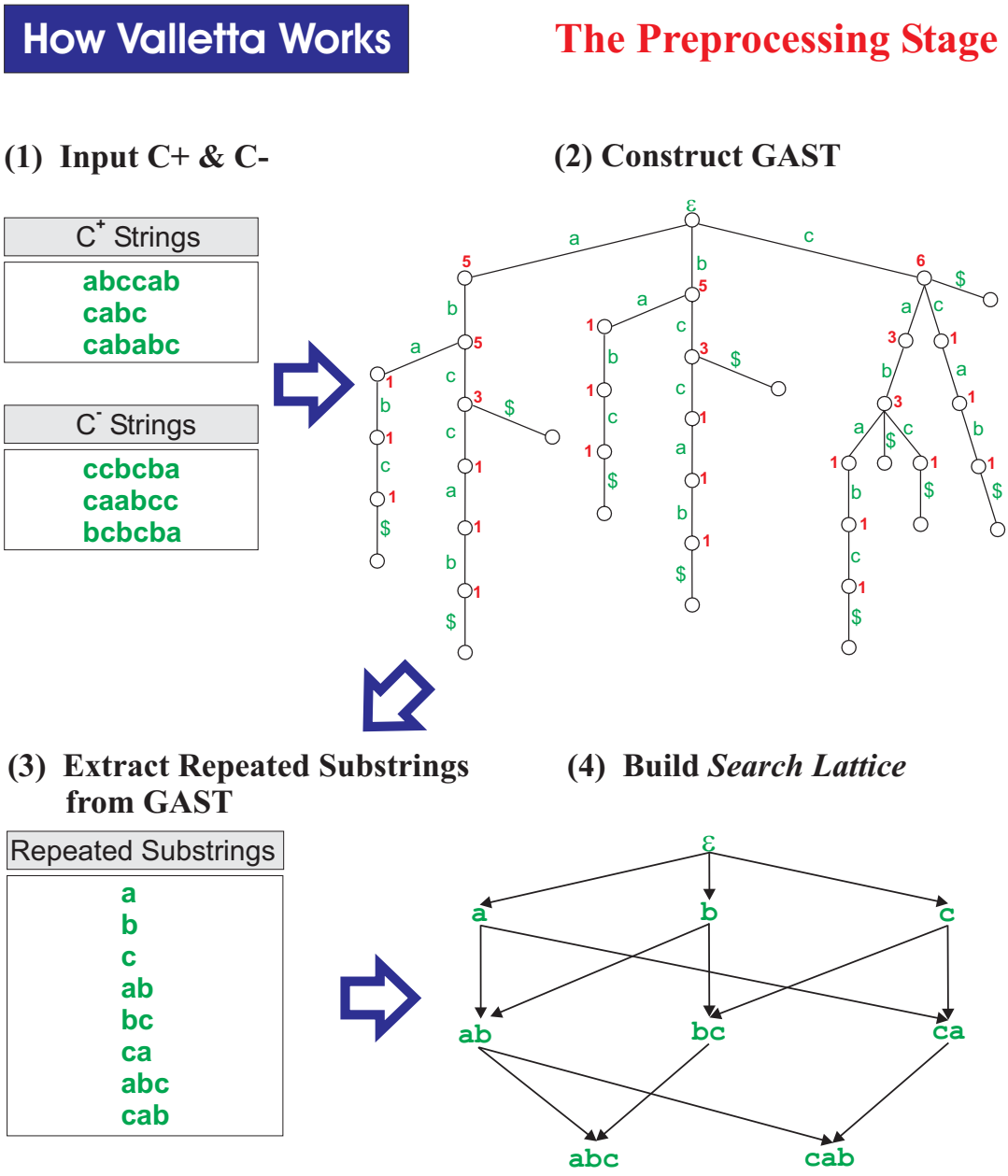


Figure 5.8: How *Valletta* Works — The Pre-Processing Stage

The preprocessing stage is performed just once — before the actual learning starts. The preprocessing stage produces a list, R_{C^+}, containing all the non-overlapping repeated substrings in C^+. The *Search Lattice* is built from this list.

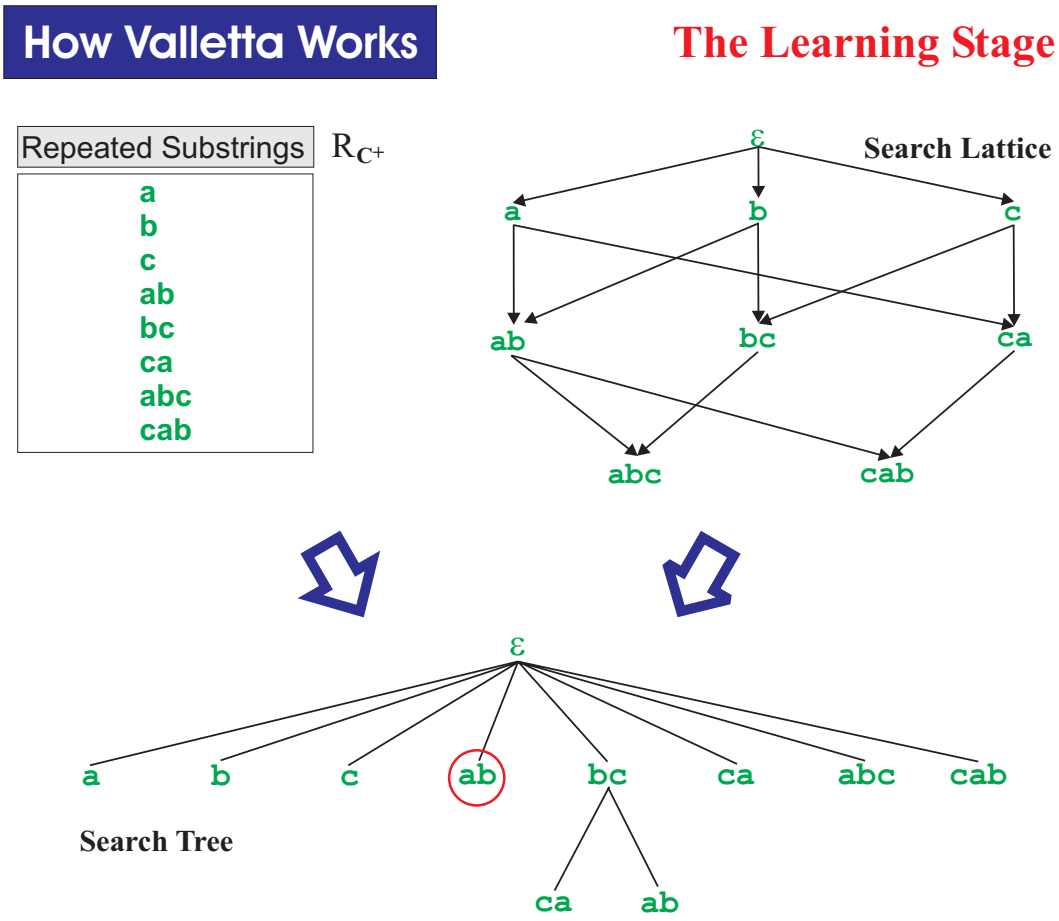


Figure 5.9: How *Valletta* Works — The Learning Stage

The learning process builds a search tree in which each node represents a set of features. Each node is labelled with a string from R_{C+}. The set of features associated with a node is defined to be the strings that are node labels in the path from that node to the root. In Figure 5.9, the node inside the red circle represents the set of features $\{abc, ba, ca\}$. When new nodes are added to the search tree, the search lattice is consulted to ensure that the new nodes represent only valid feature sets. The values of f_1, f_2, f_3, and f are computed for each node added to the tree. Learning stops when a value of f for a node exceeds the threshold t.

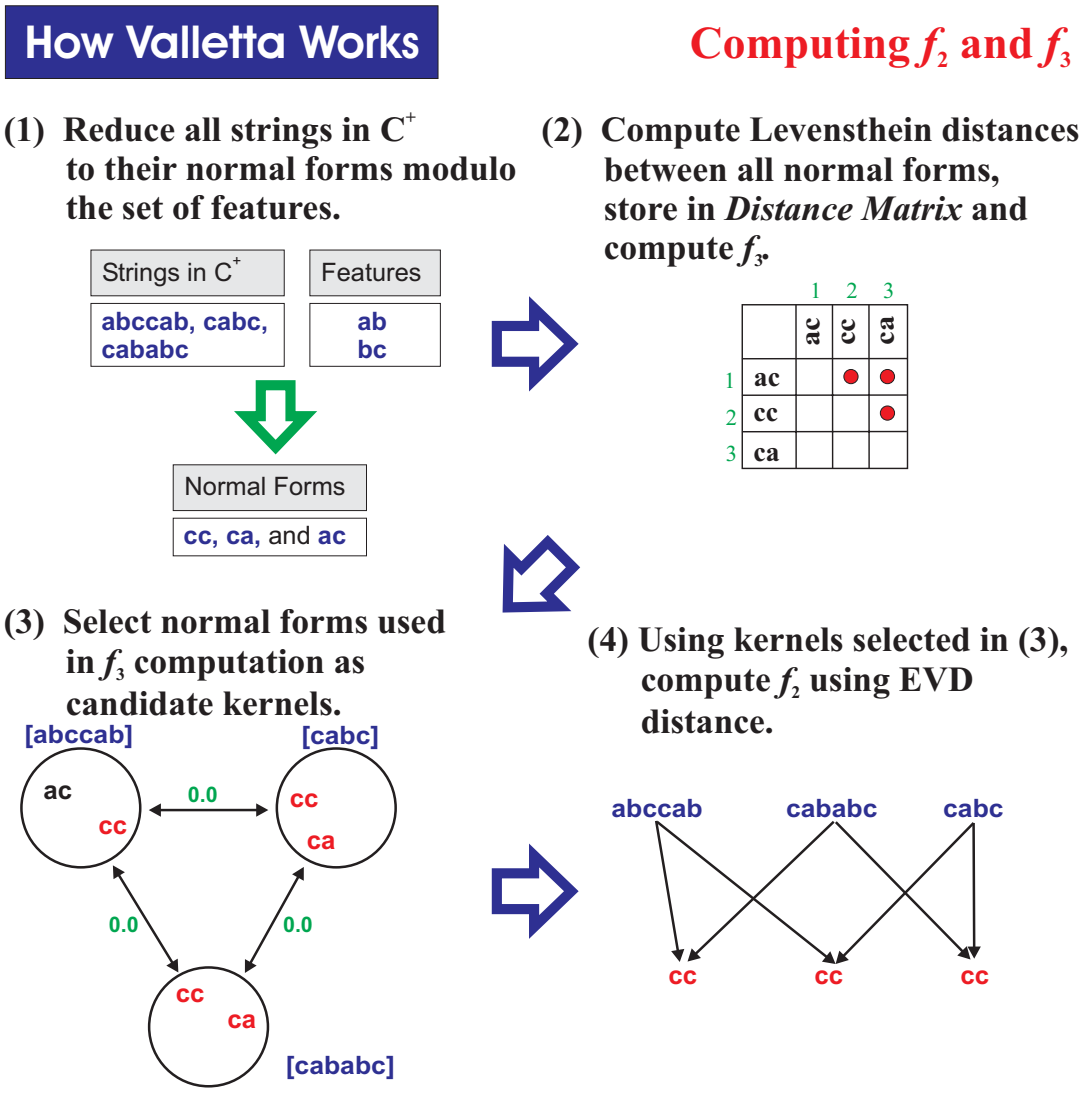


Figure 5.10: How *Valletta* Works — Computing f_2 and f_3.

Computing f_2 and f_3 involves first reducing all the strings in C^+ to their normal forms modulo the given set of features. The Levensthein distance between all pairs of normal forms is then computed and the results stored in the *distance matrix*. The value of f_3 is computed directly from the distance matrix. The normal forms that are used in the computation of f_3 are then selected as *candidate kernels*. The kernel selection procedure then finds a set of kernels that minimizes f_2.

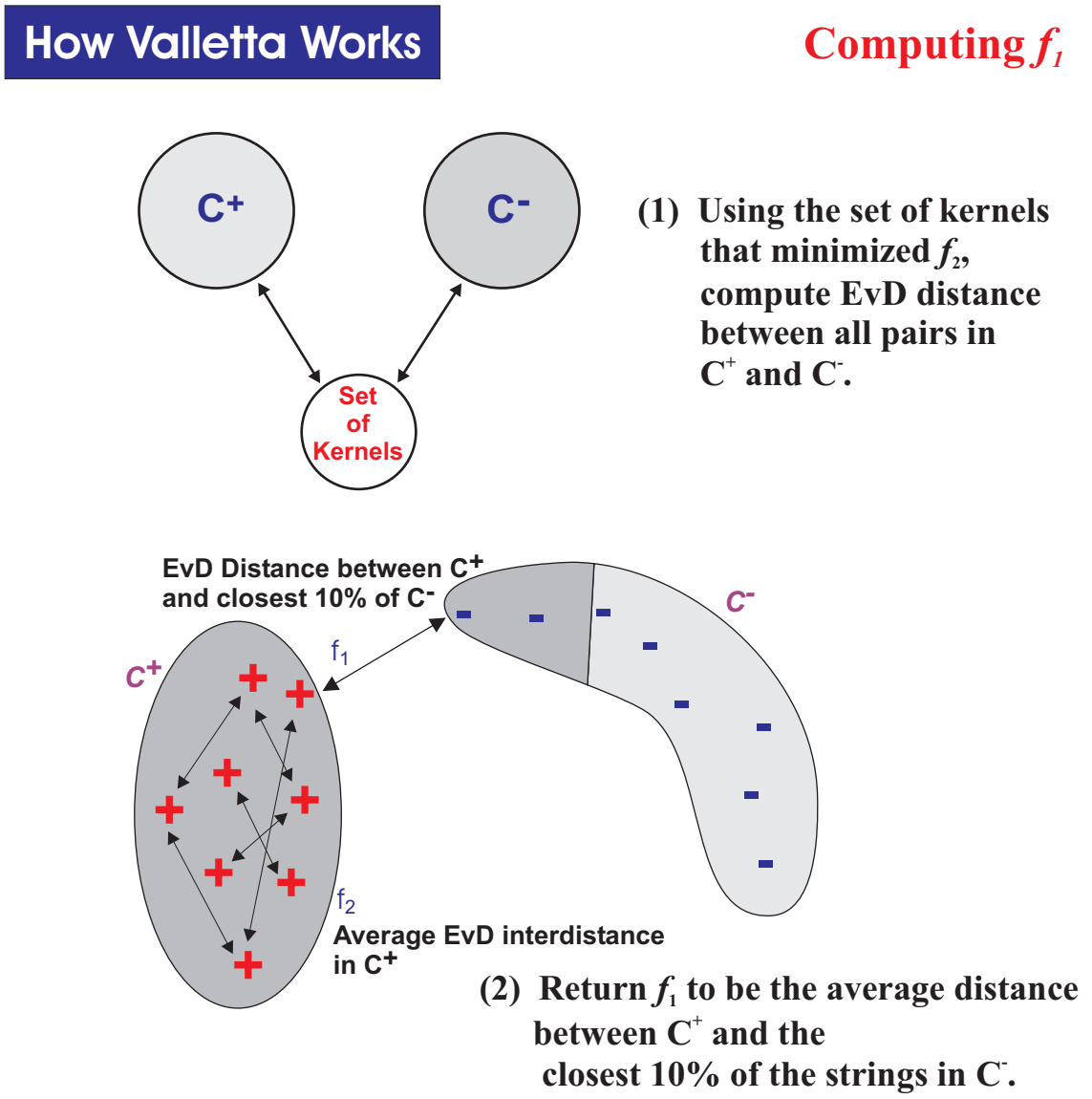


Figure 5.11: How *Valletta* Works — Computing f_1.

Once a set of kernels that minimizes f_2 is found the value of f_1 is computed. For Valletta, f_1 is defined to be the EvD (using the set of kernels as a parameter) between C^+ and the closest 10% of the strings in C^-. To achieve this, the EvD between all pairs in C^+ and C^- must be computed first. The new method of computing f_1 was required in order to correctly handle misclassification noise.

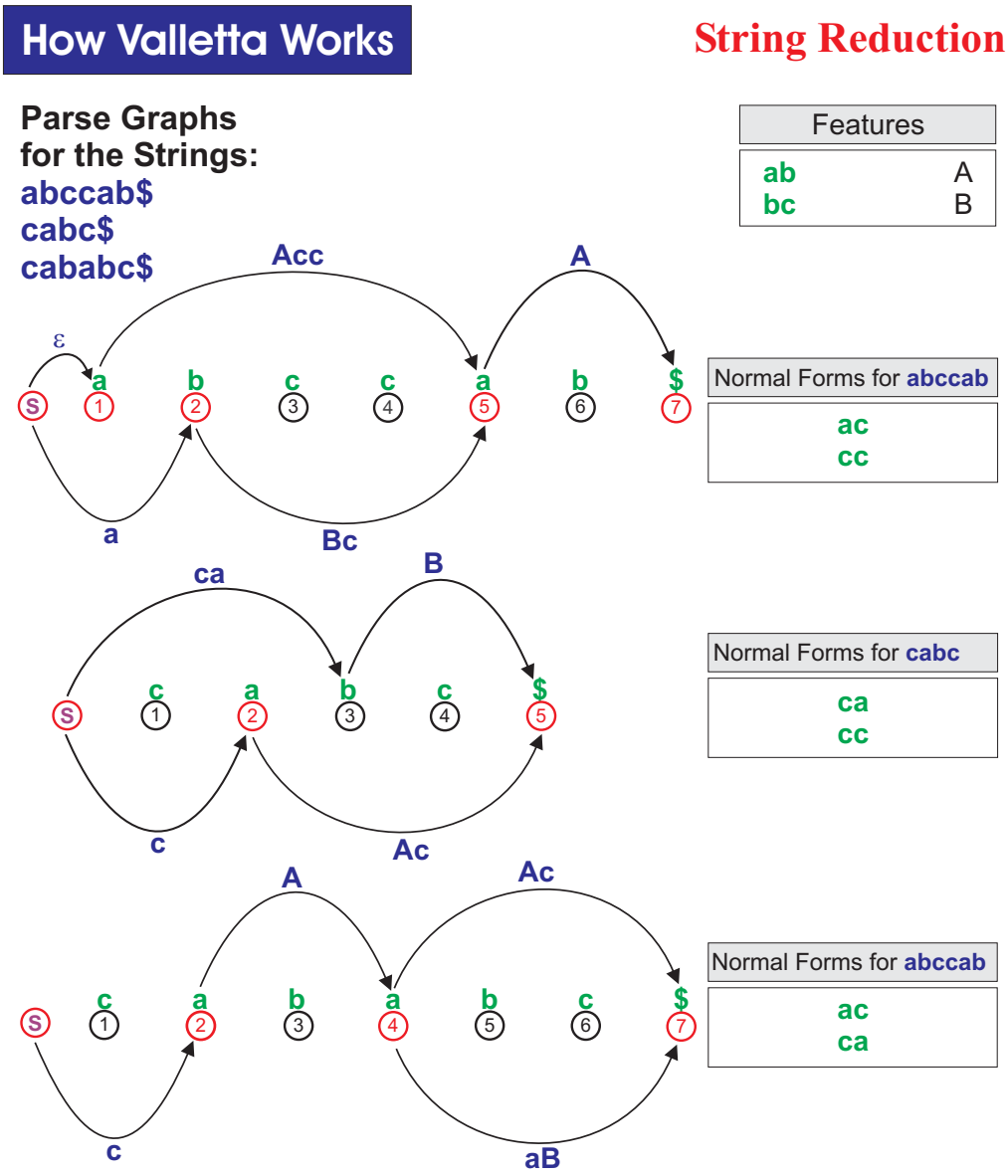


Figure 5.12: How *Valletta* Works — String Reduction

In general, there can be many different ways in which to reduce a string modulo a given set of features F. If F is confluent, each string has only one normal form but if F is non-confluent each string may have many normal forms. String reduction is achieved using a special data structure called a *parse graph*. The paths in the parse graph represent the normal forms of the string.

5.2 *Valletta* in Detail

5.2.1 The Pre-processing Stage

Recall from the previous section that Valletta's preprocessing stage performs two tasks:

(a) It finds the set of *candidate features*, and

(b) it builds the *search lattice*.

The set of candidate features is built by extracting from C^+ all the non-overlapping repeated substrings. This is achieved by constructing a data structure called the *Global Augmented Suffix Trie* or *GAST* for short. The search lattice, which we shall define and discuss later in Section 5.2.3, is then built from the GAST. The preprocessing stage has time and space complexity that is, in the worst case, quadratic in the size of the training set and is performed only once. When the preprocessing stage is complete the actual learning process commences.

Finding Candidate Features

The GAST data structure is based on a simple but very versatile and extensively used data structure called the *suffix tree*. The suffix tree of a string was first proposed by McCreight [82] in 1976 and is, in essence, a compact index of the string's vocabulary, i.e. the set of all its distinct, non-empty substrings. Suffix trees have been widely studied and have many varied uses [57, 82, 114]. One classic application is the *substring problem*. Given a string s of length m the problem is that of determining whether or not s contains some string x of length n where n is assumed to be less than m. It turns out that once a suffix tree for s is built then this problem can be solved in $O(n)$ time. This means that even if s is very long, the search for x is performed

in time that is proportional only to the length of x. A review of texts and literature in computational biology revealed that suffix trees are widely used. This is because a suffix tree exposes the internal structure of a string and building a suffix tree in a preprocessing stage very often allows for faster and more efficient string processing algorithms. In the substring problem, for example, building the suffix tree for a string allows subsequent searches for *any* substring to be computed very efficiently. Suffix trees are also used for string matching, keyword construction, recognizing DNA contamination, finding common substrings, and many other applications [57, 114]. The reader is referred to [57] for an exposition and references. The versatility of suffix trees and the fact that the algorithms for suffix tree construction use, on average, subquadratic time motivated the author to consider suffix trees as a basis for searching for the candidate features in C^+.

Definition 5.1 (Suffix Tree). Modified from [57, page 90]

Let s be a string of length m over some finite alphabet Σ and let \$ be special symbol not in Σ. The ***suffix tree*** *of s, which we denote by $\mathcal{T}_s$ is a rooted, directed tree such that:*

(a) $\mathcal{T}_s$ has exactly m leaves numbered 1 to m.

(b) Each internal node, other than the root, has at least two children.

(c) Edges incident to internal nodes (incoming edges) are labelled with a non-empty substring of s.

(d) Edges incident to the leaves are labelled with either a non-empty substring of s with the special symbol \$ or with just \$.

(e) No two edges out of the same node can have edge-labels beginning with the same character.

(f) For any leaf i, the concatenation of the edge-labels on the path from the root to the leaf i yields the suffix of s that starts at position i, i.e. s[i..m].

Definition 5.2.

Let s be a string over a finite alphabet Σ and let $\mathcal{T}_s$ be the suffix tree for s. Then:

- *The **label** of a path from the root to a node i, is the concatenation, in order, of the substrings labelling the edges of the path.*
- *The **path-label** for a node is the label of the path from the root to the node.*
- *For any substring x of s, the **locus** of x is the node in $\mathcal{T}_s$ whose path-label is x.*

Figure 5.13 shows the suffix tree for the string *010101*. Note that the concatenation of the edge-labels on the path from the root to the leaf numbered 1 yields the full string *010101* while the path from the root to the leaf numbered 5 gives us the suffix of *010101* that starts at position 5, i.e. *01*.

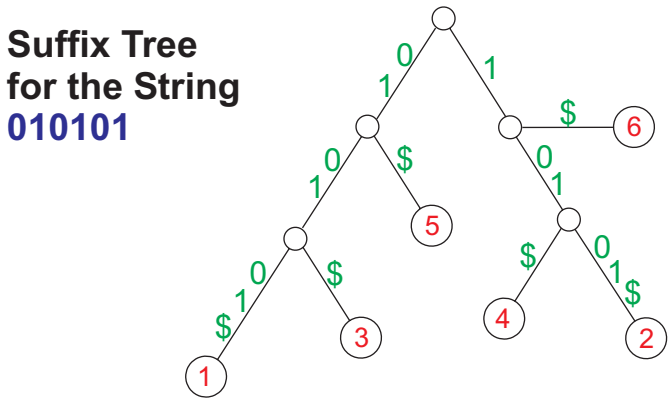


Figure 5.13: The suffix tree for the string *010101*.

There is a slight problem with the definition of suffix tree that we presented above. The problem is that if one suffix of s matches a prefix of s then no suffix tree that obeys our definition is possible. This is because the path for the first suffix would

not end at a leaf. Consider, for example, the string *010101*. Note that the suffix *01* is a prefix of the suffix *0101* and, therefore, the path that spells out *01* cannot end in a leaf. This problem is overcome by adding the special character (not in Σ) to the end of s. We denote this special character by \$. The string therefore becomes s\$. This ensures that no suffix is a prefix of any other suffix and therefore the suffix tree[4] for the string exists. It is precisely for this reason that a \$ symbol appears added at the end of every suffix of the string *010101* in Figure 5.13. From now on we shall assume that the special character \$ is always added to a string before the suffix tree is constructed.

A myriad of linear time and space suffix-tree construction algorithms have been developed in the last three decades [57, 128, 129, 138]. The technique used by Valletta for searching for the non-overlapping repeated substrings in C^+ employs a special type of suffix tree known as the *Suffix Trie*[5]. The reason for our choice of such a data structure will become apparent later.

Definition 5.3 (Suffix Trie).

Let s be a string of length m over some finite alphabet Σ. The ***suffix trie*** *of s, which we denote by $\mathcal{I}_s$, is a suffix tree such that each edge is labelled with a* ***single*** *character from Σ and sibling edges have distinct characters.*

Figure 5.14 shows the suffix tree for the string *010101*. Note that each node can have a maximum of $|\Sigma|$ children.

The number 3 (in purple) next to the locus of the string *01* denotes the number of non-overlapping occurrences of *01* in *010101*. This point will be elaborated upon later on. The total number of nodes, and therefore also edges, of a suffix trie can, in the worst case, be quadratic in the length of the string. The time and space complexity

[4]The suffix tree for a string s is unique (up to isomorphism of graphs) [114].

[5]The word *trie* was coined by Fredkin [114] from 'information re*trie*val'.

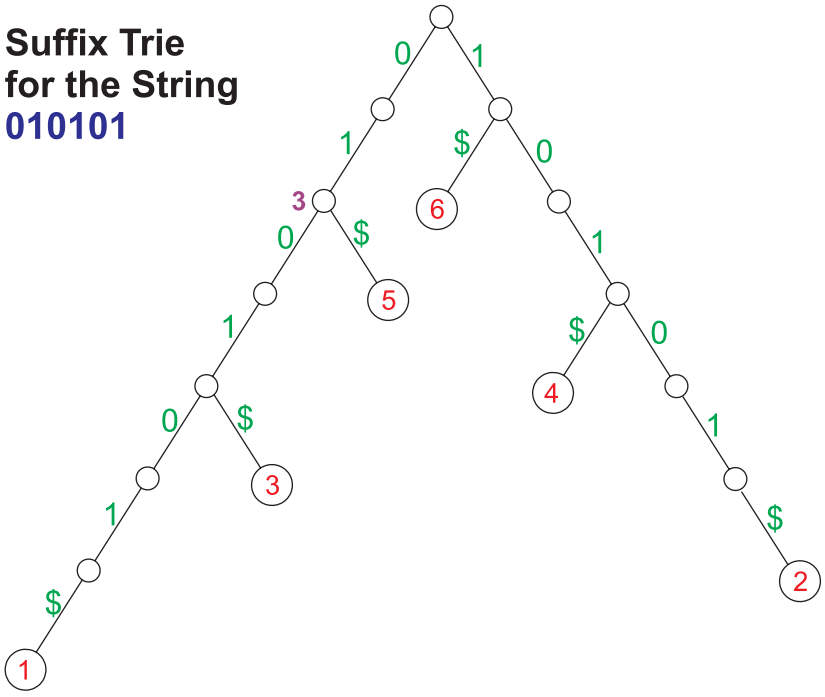


Figure 5.14: The suffix trie for the string *010101*.

of building a suffix trie is therefore also, in the worst case, quadratic in the length of the string. In spite of this, we opted for suffix tries instead of suffix trees. This was because suffix tries allow for a simple and reasonably efficient solution to the following problem: Suppose we are given a string s and we have to find, and count the number of occurrences of, all the distinct, non-overlapping, repeated substrings in the given string s. It turns out that this can easily be achieved by adding an integer counter to each node i of the trie. This counter records the number of non-overlapping occurrences of the substring whose locus is the node i. When the trie is being built, the counter is incremented every time the node is visited. In the suffix trie in Figure 5.14, for example, there are three suffixes that start with the prefix *01*, the locus of *01* is therefore visited three times during the construction of the trie. This is the reason for the number 3 shown next to the locus of the substring *01* in Figure 5.14. The numbers for the other nodes are not shown. The substrings, and

the number of occurrences of each substring, can therefore be computed during the actual construction of the trie. When the trie is completed, a traversal (in any order) of the trie yields a list of substrings and, for each substring, the number of occurrences that substring. This is accomplished by writing down the path-label and the counter of each node i during the traversal of the trie. We shall see soon that a suffix trie for a string s of length m can be built in at most $(m^2 - m)/2$ steps. However, we need to find not just the non-overlapping repeated substrings of one string but rather those in all of C^+. To accomplish this we used a new data structure, based on the suffix trie, that allows us to achieve this goal.

Definition 5.4. *Let S be a finite set of strings over a finite alphabet Σ and let the longest string in S be of length m. A* ***Global Augmented Suffix Trie*** *for S, which we denote by $\mathcal{G}_S$ is rooted, directed tree such that:*

(a) Each internal node, other than the root, has a minimum of two and a maximum of $|\Sigma|$ children.

(b) Each edge is labelled with a symbol from $\Sigma \cup \$$.

(c) Edges incident to the leaves are labelled with \$.

(d) $\mathcal{G}_S$ has depth $m + 1$.

(e) Sibling edges have different edge labels.

(f) For any $s \in S$ and for any $1 \leq i \leq |s|$, there exists a unique leaf l such that the concatenation of the edge-labels on the path from the root to the leaf l yields the suffix of s that starts at position i, i.e. $s[i..m]$.

Figure 5.15 shows the GAST for the strings *010101*, *00101*, and *11101*. The number to the left of each node represents the number of non-overlapping occurrences

of the substring whose locus is the node in question. The number on the right of each node represents the number of strings in C^+ in which the substring occurred. For instance, the node whose path-label is *10* (indicated in Figure 5.15 by a purple arrow), has the number 4 on the left and the number 3 on the right. This means that the substring *10* occurred four times in three different strings. The reader may recall that, in suffix tries, the leaves are labelled with a number corresponding to the starting position of the suffix (i.e. the path-label of the leaf) in the string. With GAST we are dealing with not just one string but with a set of strings and therefore this numbering of the leaves is not relevant.

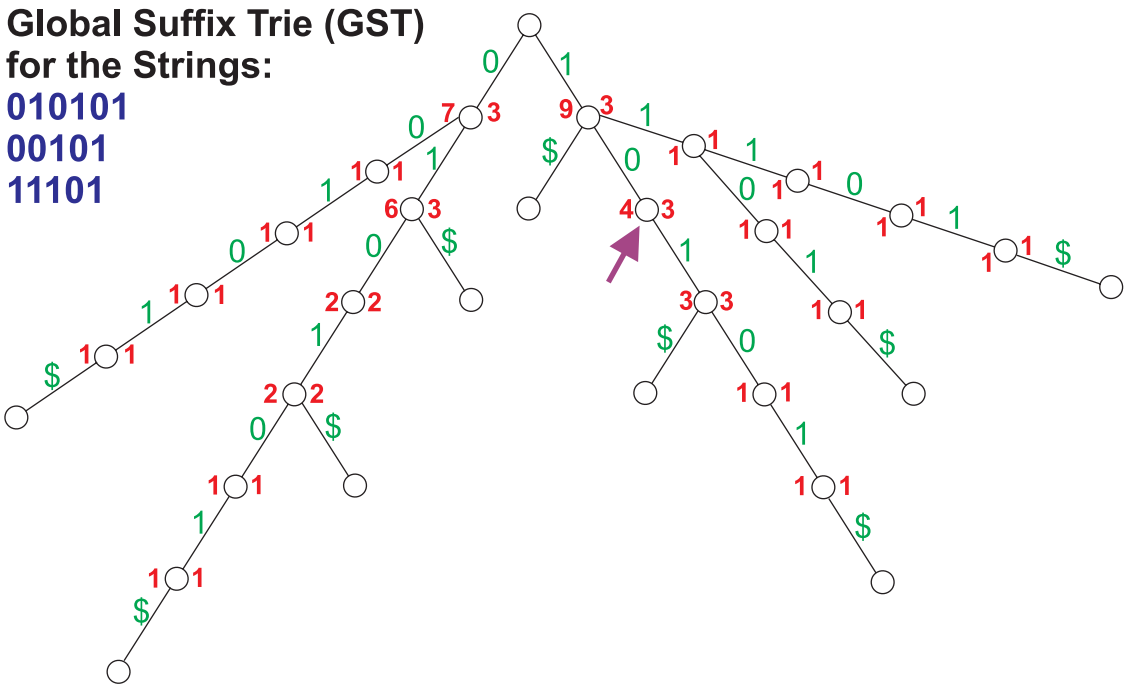


Figure 5.15: The GAST for the strings: *010101*, *00101*, and *11101*.

5.2.2 An Algorithm for Global Augmented Suffix Trie Construction

Algorithm 5.2.1: BUILDGAST($File$)

procedure INSERTSTRING(s)
comment: Process string and update trie
$n \leftarrow len(s)$
for $i \leftarrow 1$ **to** n
do {
  $currnode \leftarrow root$
  $x \leftarrow s(i..n)$
  for $j \leftarrow i$ **to** n
  do {
    $z \leftarrow Ordinal(x(i))$
    if $n(currnode).p(s) = NULL$
      then $CreateNewNode$
      else {
        $currnode \leftarrow n(currnode).p(s)$
        if $overlapcondition =$ **false**
          then $incr\ n(currnode).reps$
      }
  }
}

main
comment: open C^+ file, read strings, and insert in trie
while not $EndOfFile$
do {
  $s \leftarrow NextStringInFile$
  INSERTSTRING(s)
}

Notes to Algorithm 5.2.1, BuildGAST.

- The GAST constructed by *BuildGAST* is stored in an array of records. Each record corresponds to a node in the GAST. The records have the following structure:

  n(i).elabel, character: this is the label of the incoming edge.

  n(i).reps, integer: this stores the number of non-overlapping repetitions of the string whose locus is node i.

  n(i).p(1..n), array of pointers: this is an array of pointers to the children of node i.

- The main procedure simply reads in the C^+ training set and passes each string to the *InsertString* procedure which processes the string and updates the trie. *EndOfFile* is a Boolean variable that is true when the end of the file being read is reached. *NextStringInFile* returns the next string in the C^+ file.

- The procedure *InsertString* has two main loops. The outer loop iterates through the suffixes of s, the input string. The inner loop iterates through the characters in each suffix, x, starting from the front to the back. The inner loop updates the trie as follows. It first sets the value of *currnode*, a pointer variable, to point to the root of the trie. For each character in the suffix, $x(j)$, the value of $n(i).p(z)$ is checked (z is the ordinal value of the character $x(j)$). If the value of the pointer is null then this means that the algorithm has never encountered a suffix that has a prefix equal to the path-label of that node. In this case a new node is created. If the pointer is not null, then the *currnode* pointer is set to the current node and the *n(i).reps* counter is incremented.

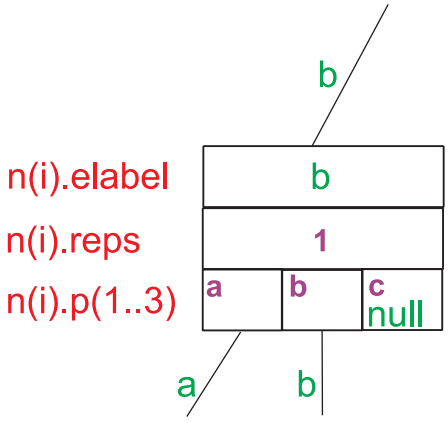


Figure 5.16: The record structure of each GAST node.

The operation of the *BuildGAST* algorithm is illustrated with a simple example. Figure 5.17 shows the partially completed GAST for the string *aab*. (a) shows the

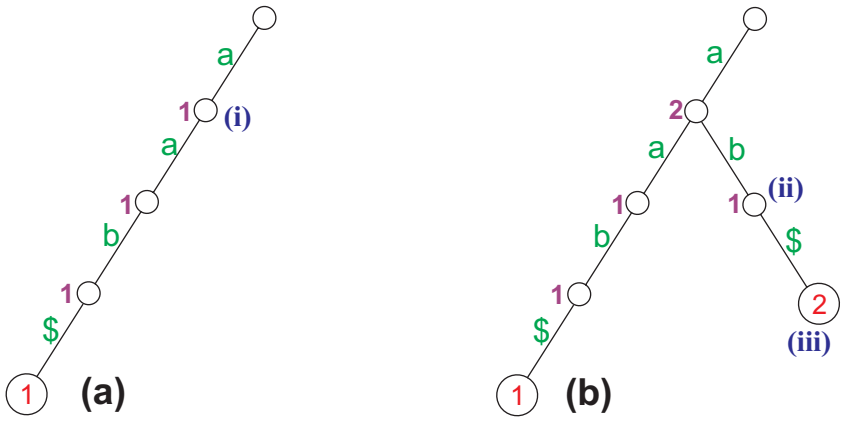


Figure 5.17: The partially completed GAST for the string *aab*.

trie after the suffix *aab*, i.e the whole string has been added. The number 1, in purple, next to each node represents the number of repetitions of the path-label of that node. When the suffix *ab* is processed the algorithm discovers the root's first subtree (corresponding to the character 'a') is not empty. It therefore goes to the node labelled (i) and increments the counter $n(i).reps$. This will record the fact that the substring *a* has been found twice in the string. It then discovers that node (i) does not have a left subtree (corresponding to the character *b*). A new node (ii) is created and the counter n(ii).reps is set to 1, recording the fact that the substring *ab* was found twice so far. The node (iii), labelled with the $ symbol, is then added since every suffix must terminate with this symbol.

Once the GAST is constructed all that remains is to extract the repeated substrings. Since each node stores the number of times the suffix associated with that node has been found in C^+, this process is performed by a simple recursive depth-first tree-traversal algorithm. The repeated substrings that are extracted from the GAST are stored in the list R_{C^+}.

5.2.3 The Search Lattice

The last procedure carried out by the preprocessing stage is the construction of the search lattice L_{C^+} from the set of repeated substrings R_{C^+}. The search lattice is a directed acyclic graph. It contains a distinguished node called the root and labelled with the null string ε. Each node is labelled with a distinct string from R_{C^+}. Given any two nodes, a and b, the edge (a, b) exists in L_{C^+} if $a \triangleright b$.

Substring	SPos	Reps
0	7	3
1	9	3
00	1	1
01	6	3
10	4	3
11	1	1
001	1	1
010	2	2
101	3	3
110	1	1
111	1	1
0010	1	1
0101	2	2
1110	1	1
00101	1	1
11101	1	1
01010	1	1
10101	1	1
010101	1	1

Table 5.3: The set R_{C^+} created from the strings: *010101*, *00101*, and *11101*.

The covering relation $\triangleright$ is defined as follows:

Definition 5.5. *Let $a, b \in \Sigma^*$ be any two strings. We say that $a \triangleright b$ if* b *can be obtained from* a *by the left or right concatenation of a non-empty string x, i.e. $b = xa$ (left concatenation) or $b = ax$ (right concatenation).*

L_{C^+} is therefore the lattice $(R_{C^+}, \triangleright)$. One interesting property of the search lattice

is that, for any given node n, the labels of all nodes that are descendants of n are substrings of its label. The search lattice is used by Valletta to efficiently build substring-free subsets of R_{C+}. Given a set of features F (a subset of R_{C+}), Valletta expands F by considering only nodes in R_{C+} that are not descendants of the strings in F. Table 5.3, on the previous page, lists the repeated substrings found by Valletta for the strings *010101*, *00101*, and *11101*.

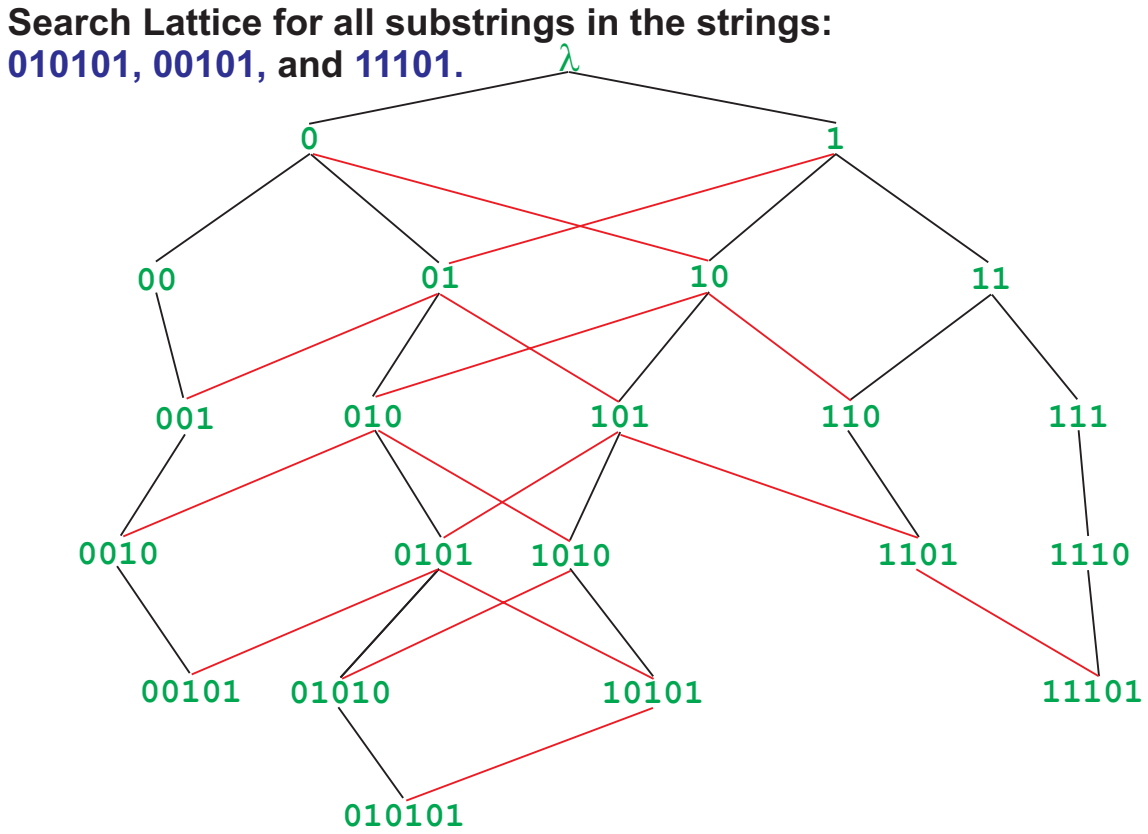


Figure 5.18: The search lattice built from the strings in R_{C+}.

Figure 5.18 shows the search lattice constructed from these strings. Black edges denote concatenation on the right and red edges denote concatenation on the left. The algorithm used for the construction of the search lattice is relatively simple and has been omitted.

5.3 How Valletta Learns

When the preprocessing stage is completed, Valletta invokes the search engine and learning starts. The input to the learning stage is the set of repeated substrings R_{C^+} and the search lattice L_{C^+}. As previously explained, R_{C^+} completely specifies the search space of the learning stage. This is because the set of features in the target TS description must be a subset of R_{C^+}. Not just any subset, in fact, but a substring-free subset of R_{C^+}. As also explained in the previous section, the search lattice L_{C^+} allows us to efficiently choose only substring-free subsets of R_{C^+}. Valletta employs a search technique that searches through the space of all the valid (i.e. substring-free) subsets of R_{C^+}. It must be emphasized that, although Valletta considers all of the search space, it does not necessarily have to search all of the space. It stops when it finds a TS description consistent with the set of training examples. If, however, Valletta does not find a valid TS description, it will continue searching until it enumerates the whole search space. This is an important point since it means that Valletta will always find a TS description consistent with the training sets if one exists.

Overview

Valletta's search engine enumerates all the substring-free subsets of R_{C^+} by building a *search tree*, which we denote by TR. Before learning begins, the value of f is computed for each of the strings in R_{C^+}. In other words, each of the strings in R_{C^+} is considered as a feature set with just one feature. The strings in R_{C^+} are then sorted (in ascending order) according to the value of f_2. All the strings in R_{C^+} are then added to TR as children of the root node. Before learning starts, therefore, TR is of depth 1 and has a root node and $|R_{C^+}|$ children.

In TR, each node is labelled with a string from R_{C^+}. The feature set associated with a node in TR is the set of strings from R_{C^+} that are node labels in the path

from that node to the root. Figure 5.19 shows the completed search tree for the set of repeated substrings $R_{C^+} = \{a, b, c, d\}$. For example, the node shown circled in red is associated with the feature set $\{c, a, d\}$ which is precisely the set of strings from R_{C^+} that are node labels in the path from that node to the root. Each node in the TR, therefore, represents a valid feature set and the aim is to build a search tree such that, in the shortest time possible, a node is found that represents the set of features in the target TS description. When adding children to a node in TR,

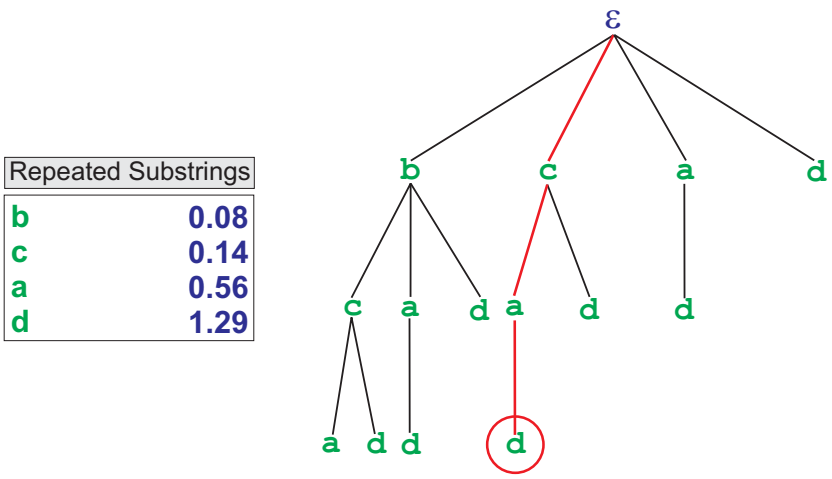


Figure 5.19: The completed search tree for the set $R_{C^+} = \{a, b, c, d\}$.

the search engine considers only strings in R_{C^+} that have a value of f_2 that is *larger* than the node being expanded. This prevents TR being expanded with nodes that represent the same feature set as a node already in the tree and also ensures that strings with a low value of f_2 are added first. A child node is also not given a label that would result in a feature set that is not substring-free. The search lattice is used to ensure that this does not happen. When a node n in TR is being expanded, the search engine consults the search lattice L_{C^+} and excludes all strings in R_{C^+} that are descendants of the feature set associated with n. This makes the whole process simple and efficient. As can be seen in Figure 5.19, if the search process continues,

the *TR* will eventually be completed. The completed search tree *TR* will contain a unique node for every valid feature set in R_{C^+}.

Search tree construction is performed by the *ETSSearch* procedure. *ETSSearch* initially creates a search tree that has exactly one level with a node for every string in R_{C^+}. *ETSSearch* then computes that value of f for each leaf. If, for all of the leaves, the value of f does not exceed the threshold, *ETSSearch* then expands the tree by adding new leaves to those nodes for which f_2 is minimal. The tree search technique used by the Valletta search engine is loosely based on a generic graph search technique described by Nilsson in [93]. This simple but effective technique is shown in Algorithm 5.3.1 below.

Algorithm 5.3.1: GRAPHSEARCH(*void*)

comment: **Generic Graph Search algorithm [93, page 142]**
* OPEN and CLOSED are two lists
* TR is the search tree built by the algorithm

TR $\leftarrow n_0$
comment: Initially, TR consists solely of the start node n_0

OPEN $\leftarrow n_0$
CLOSED $\leftarrow \emptyset$
while OPEN $\neq \emptyset$
do
- ***a)*** Get first node from OPEN
- ***b)*** Remove it from OPEN
- ***c)*** Put it on CLOSED
- ***d)*** Call it n
- **if** n is a goal
  - **then** Exit with **SUCCESS**
  - **else**
    - **1.** Expand node n by generating a set $\mathcal{M}$ of successors. Install each successor as a child of n in TR
    - **2.** Reorder the list OPEN, either according to some arbitrary scheme or according to heuristic merit.

Exit with **FAILURE**

One of the main advantages of this technique is that it can be used to perform best-first, depth-first, and breadth-first searches. In breadth-first search the nodes are simply put at the end of the OPEN list (FIFO) and are not reordered. For a depth-first search the nodes are added at the beginning of the OPEN list (LIFO). For best-first search the list OPEN is reordered according to some heuristic merit of the nodes [93]. The simplicity and elegance of this graph search technique and was the inspiration for Valletta's search engine. The actual algorithm differs from Nilsson's algorithm in the number of ways.

(a) Three lists are used and not two — ACTIVE, PENDING, and CLOSED.

(b) Expanding the tree is done in two separate stages.

(c) The tree construction procedure continues as long as the lists ACTIVE and PENDING are non-empty.

When the algorithm is run it first creates a tree with ε in the root and with all the strings in R_{C^+} as children. These nodes are added to ACTIVE. The other lists, PENDING and CLOSED, are initially set to null. The list ACTIVE stores nodes that have just been added to the tree. The value of f of each of these new nodes must therefore be computed. In each iteration of the main loop the algorithm performs the following procedures:

(a) The value of f for each node in ACTIVE is computed and the status of the node is changed to PENDING (i.e. the node is deleted from ACTIVE and added to PENDING).

(b) The list PENDING is then reordered, in ascending order, according to the value of f_2. PENDING will then have the nodes with the smallest values of f_2 at the top.

(c) The top I nodes in PENDING are expanded by adding two strings from R_{C^+}. The search lattice is used to ensure that the new child nodes represent valid feature sets. The new nodes are added to ACTIVE. If, for any node in PENDING, no new child nodes can be added, the node is deleted from PENDING and added to CLOSED.

(d) The algorithm then expands a further J nodes based on the value of f_2 and also the value of *n.pass*. For each node n, the value of *n.pass* records the last time (iteration number) the node was expanded. The algorithm subtracts the value of *n.pass* of each node from the variable *pass*, which stores the number of current iteration. The greater the difference the more likely that the node is expanded. This ensures that *every* node in the tree will eventually be expanded and prevents Valletta from making *TR* deeper and deeper as it searches of the target TS description.

The pseudocode of Valletta's search engine, Algorithm 5.3.2, is shown overleaf. For any node n in the search tree, $path(n)$ denotes the set of node labels in the path from n to the root. The function $path(n)$, therefore, returns the feature set associated with the node n.

Notes and Observations

The values of I and J are set by the user. The variable I specifies how many of the top nodes, i.e. those with the smallest values of f_2, in PENDING are expanded and J specifies the number of other nodes that are expanded in each pass. Typically, I is given a value in the range 2 to 4 and J a value in the range 4 to 12. Every time a node is expanded, a maximum of two children are added. When a node n in *TR* is expanded, only features in R_{C^+} with a larger value of f_2 than the node label of n can be added. This ensures that each node in *TR* uniquely represents a feature set.

Algorithm 5.3.2: ETSSEARCH(L)

comment: **Valletta's Search Engine**

comment: The parameter L is the Search Lattice

TR $\leftarrow \lambda$

comment: Initially, TR consists solely of the root node λ

Create a child to λ for every node (string) in L
Add every string in L to ACTIVE
Set PENDING $\leftarrow \emptyset$ and CLOSED $\leftarrow \emptyset$
$pass \leftarrow 0$
while ACTIVE*and*PENDING $\neq \emptyset$
do
- $pass \leftarrow pass + 1$
- For all n in ACTIVE
  - **do**
    - $K \leftarrow path(n)$
    - Compute f for K
    - $n.score \leftarrow f_2 + f_3$
    - $n.pass \leftarrow pass$
    - **if** $F \geq$ THRESHOLD
      - **then** Exit with **SUCCESS**
    - Remove n from ACTIVE and add it to PENDING
- Sort ACTIVE in descending order according to *score*
- **For** n = the top I nodes in PENDING
  - **do**
    - Add two children n_1 and n_2 to TR from L checking that $path(n_1)$ and $path(n_2)$ are valid feature sets.
    - **if** No such children can be found
      - **then**
        - Remove n from ACTIVE
        - Add n to CLOSED
      - **else** Add n_1 and n_2 to ACTIVE.
- **For** n = the top J nodes in PENDING with $n.pass \leq pass - 2$
  - **do**
    - Add two children n_1 and n_2 to TR from L checking that $path(n_1)$ and $path(n_2)$ are valid feature sets.
    - **if** No such children can be found
      - **then**
        - Remove n from ACTIVE
        - Add n to CLOSED
      - **else** Add n_1 and n_2 to ACTIVE.

Exit with **FAILURE**

If there are no more strings in TR that have a larger value of f_2 than the label of the node, the node is deleted from PENDING and added to the CLOSED list. The value of J is important because any value greater than zero ensures that Valletta will enumerate the whole search space and is thus guaranteed to find a TS description that is consistent with the training examples (if one exists). Valletta does not stop until the lists ACTIVE and PENDING are empty and all the nodes are in the list CLOSED. This happens *only* if the search tree is complete and, thus, all possible valid feature sets have been considered.

The search for the target TS description is completely distance-driven. After each iteration of the main loop, the value of f is computed for all the new nodes in the ACTIVE list. These nodes are then removed from ACTIVE and added to PENDING. The nodes in PENDING are then sorted, in ascending order, according to the value of f_2. Various other criteria for sorting PENDING were tried but f_2 worked best and returned the fastest convergence. Valletta was run on some datasets using the value of f_1 for sorting PENDING but this made the algorithm very sensitive to the choice of C^-.

Valletta searches for the features and not for the kernels. Once it has found a set of features it searches for a set of kernels amongst the normal forms that minimizes the value of f_2. An obvious alternative strategy is to try to find the kernels directly from C^+. This is theoretically possible since the kernels are subsequences of the strings. The approach was considered but the number of subsequences in a string is super-polynomial. This makes such an approach impractical. Valletta's search engine is essentially a variation of the A^* and the *Beam* search techniques that are popular in AI applications [93]. The value of the parameter I determines the size

of the 'beam'. Figure 5.20 is a depiction of how Valletta expands the search tree.

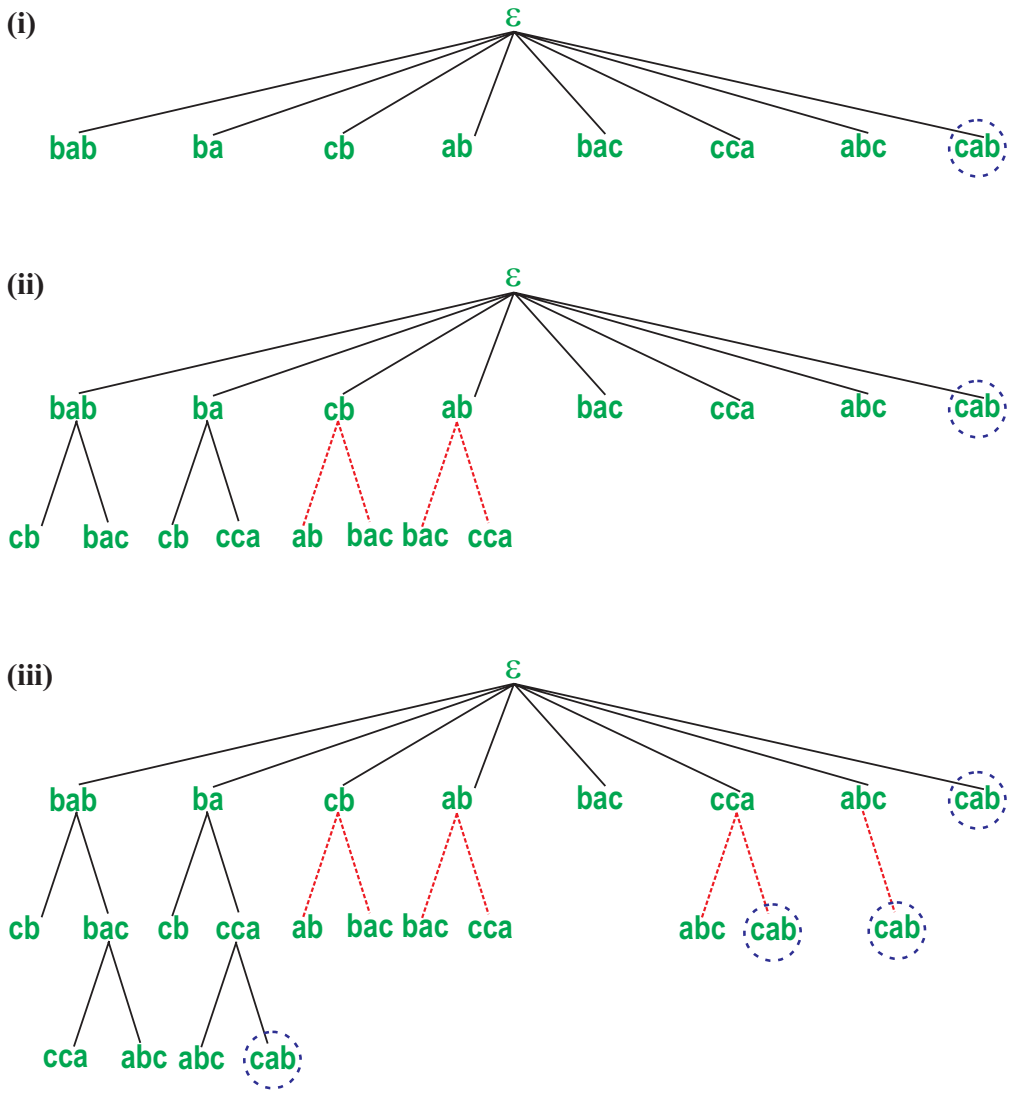


Figure 5.20: How Valletta expands the search.

In this case both I and J were set to 2. The search tree is expanded in two stages. Valletta first expands the two nodes with the smallest values of f_2 and then expands two other nodes based on the value of $n.pass$ which, for each node, stores the number of the pass (iteration of the main learning loop) that node was last expanded. The edges of the nodes expanded in the second stage are shown in red.

Two of the author's main concerns when developing Valletta were whether it was possible to find an effective and efficient method for reducing the strings in C^+ and C^- modulo a given set of features and, also, an efficient procedure for finding the target set of kernels within the set of normal forms. These two issues are discussed in the next two sections.

5.4 Computing the f function

At every iteration of the learning loop, Valletta computes the value of f for each feature set that is being considered by the algorithm. In the GSN algorithm, computing the value of f involved first computing f_2 by finding the average pair-wise distance in C^+ and then computing f_1 by finding the minimum distance between all pairs (a, b) where $a \in C^+$ and $b \in C^-$. This procedure is relatively straight-forward since all the languages that can be learned by the GSN algorithm are single-kernel. The case for multiple-kernel languages is much more complex. This is because, given a feature set F, Valletta must find a set of kernels that optimizes f. This is, in general, not a trivial task. If C^+ is reduced to its normal forms modulo F, then the required set of kernels is a subset of this set of normal forms. This is because if we remove the features from the strings in C^+, all that is left, in theory, should be the kernels. All the normal forms of C^+, therefore, are *candidate kernels*. The problem is that, if F is not confluent, then the number of normal forms may be very large and trying all possible subsets is, for obvious reasons, out of the question. The presence of noise further complicates the task since some of the kernels will be corrupted. Moreover, it was initially not clear how to compute the value of f_2 in the case of multiple kernels. Although these problems seemed insurmountable at first, simple and elegant solutions were eventually found. Valletta computes the value of f as follows:

(a) Reduce both C^+ and C^- to their normal forms modulo the θ-reduction relation

R_F. A θ-reduction relation is used in order to detect features inserted inside other features. After reduction, each string $s \in C^+$ has a set of normal forms. We denote by $\Downarrow(s)$ the set of all normal forms of the string s (modulo R_F) with all occurrences of θ removed.

(b) Compute the Weighted Levensthein Distance (WLD) between all pairs of the normal forms. The results are stored in the *Distance Matrix*.

(c) Compute the value of f_3. The function f_3 is defined to be the average pair-wise distance in C^+ computed between the sets of normal forms. In other words, for all pairs of strings $s_1, s_2 \in C^+$, we find the minimum WLD between the normal forms in $\Downarrow(s_1)$ and $\Downarrow(s_2)$. The average is taken over all such pairs in C^+.

(d) Identify the normal forms that were used to find the value of f_3. Given any two string $s_1, s_2 \in C^+$, we identify the normal form in $\Downarrow(s_1)$ and the normal form in $\Downarrow(s_2)$ that return the minimum WLD. We *promote* all such normal forms as *candidate kernels*.

(e) From the promoted set of candidate kernels find the set of kernels K that minimizes f_2. In our case f_2 is defined to be the average EvD in C^+ with K being one of the parameters (F is the other parameter). This process, it turns out, is NP-Hard. We use an approximation algorithm.

(f) Compute the value of f_1, i.e the minimum EvD between C^+ and C^-. We use a modified definition of f_1 in order to handle misclassification noise. This is discussed in Chapter 7.

(g) The value of f is then computed in the usual way, i.e.

$$f = \frac{f_1}{c + f_2}.$$

We now discuss each of the above steps in detail. Reduction of C^+ and C^- to their normal forms modulo R_F is discussed in the next section.

Distance Matrix Computation

Distance matrix computation involves computing the weighted Levensthein distance (WLD) between all pairs of normal forms of the strings in C^+. This process is important since the pair-wise distances are then used in the computation of both f_2 and f_3. Of course, there is no need to compute the WLD between *all* pairs of normal forms. Valletta first constructs a list of the all the *unique* normal forms and then computes the distance between all the pairs in this list. This is depicted in Figure 5.21 below. The actual number of distance computations performed is $\frac{n(n-1)}{2}$

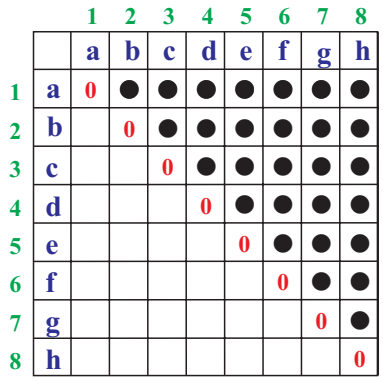


Figure 5.21: Computing the distance between the normal forms.

where n is the number of unique normal forms of C^+. In practice, a normal form can occur in many strings and in the actual implementation, special pointers, that record to which string in C^+ each normal form belongs, are maintained. It must be pointed out that the WLD is computed on the normal forms and not on the actual strings in C^+. Normal forms are usually much shorter than the strings in C^+ since all the features in F have been deleted. Using the distance matrix allows us to have an efficient method for computing f_2 and f_3 since the WLD between any two normal

forms is computed only once and stored in the matrix. Also, the WLD algorithm uses only single-character operations and can therefore be optimized for speed.

Computing f_3.

When the distance matrix is complete, the value of f_3 can be computed. This is the average Normal Form Distance (NFD) between the strings in C^+. For each pair of strings $s_1, s_2 \in C^+$, we find the minimum WLD over all pairs of normal forms in $\Downarrow(s_1) \times \Downarrow(s_2)$. Notice that, in this case, we are finding the minimum distance between two sets — not two strings. Also, we do not need to perform the actual distance computations since the distance matrix contains the pre-computed WLDs between all normal forms. For each pair of strings $s_1, s_2 \in C^+$ we identify the pair of normal forms that return the minimum distance between $\Downarrow(s_1)$ and $\Downarrow(s_2)$. This is depicted in Figure 5.22 below.

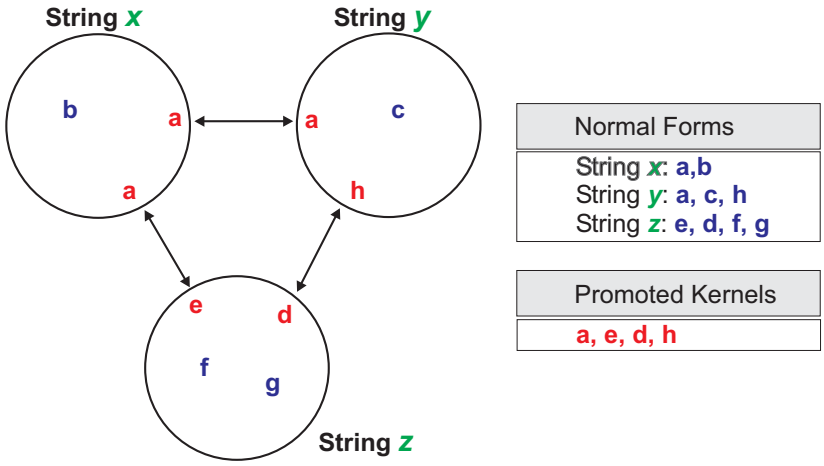


Figure 5.22: Promoting the kernels used in f_3 computation.

The function f_3 gives us a measure of the '*entropy*', as-it-were, of the set C^+. We also use f_3 to identify which of the normal forms are potential kernels and which can be safely discarded. In Figure 5.22, C^+ contains the strings x, y, and z. The normal forms are:

String	Normal Forms
x	$\Downarrow(x) = \{a, b\}$
y	$\Downarrow(y) = \{a, c, h\}$
z	$\Downarrow(z) = \{e, d, f, g\}$

The minimum distance between $\Downarrow(x)$ and $\Downarrow(z)$, for example, is returned by (a, e). These normal forms (shown in red) are therefore retained as candidate kernels. During the computation of f_3 we record which normal forms return the minimum distance and how many times. Returning to Figure 5.22, note that the normal form a is used twice and $\{e, d, h\}$ are used once. The normal forms $\{b, c, f, g\}$ are not used and are discarded. The function f_3, therefore, plays two important roles; *(i)* it returns a measure of the *entropy* of C^+ and *(ii)* it identifies those strings that are most likely to be kernels. The normal forms used in the computation of f_3, shown in red in Figure 5.22, are then *promoted* as candidate kernels. The rest are discarded. The set of promoted kernels is denoted by PR. In our example $PR = \{a, e, d, h\}$.

String	Promoted
x	$\{a\}$
y	$\{a, h\}$
z	$\{e, d\}$

Kernel Selection

The next step in the computation of f is finding a subset of the candidate kernels in PR that optimizes f. This was probably the biggest problem faced by the author in developing Valletta. The set PR is the union of the sets of promoted normal forms of each string in C^+. The task at hand is to find a set of kernels that will minimize f_2. The function f_2 is defined to be the average, over all pairs, EvD in C^+. Recall from Chapter 3 that EvD between two strings s_1 and s_2 involves reducing both strings to their normal forms modulo the set of feature F and then computing weighted Levensthein distance from each set of normal forms to the set of kernels K. It therefore follows that if each of the sets of normal forms of the two strings contain at least one kernel in K, the EvD will then be zero. Given the sets of candidate features (these are subsets of PR), we therefore have to find a subset of PR that contains at least one candidate feature in each set. If we could find such a set the value of f_2 would be zero. This problem can, in general, be easily solved by taking one string from each set but this is not possible in our case, since we cannot allow the number of kernels to be equal to the number of strings in C^+. We know that each kernel occurs more than once in C^+ and the number of kernels is the target TS description is assumed to be much less than the number of strings in C^+. The user is also allowed to enter a bound on the number of kernels that are allowed in the final TS description. We therefore have to consider subsets of PR of various sizes, subject to this bound, until we find the one that minimizes f_2. In the example of Figure 5.22, for instance, the set of candidate kernels that returns a value of zero for f_2 is $\{a, d, e, h\}$. This is called the *kernel selection* problem. The difficulty of the problem is further compounded by the fact that, if the training sets contain noisy strings, the normal forms will contain noise.

The kernel selection problem, as posed, is a subproblem of the *Minimum Hitting Set* NP-Hard problem [35, page 222], and is also itself NP-Hard (see also Appendix H. It is evident that we cannot possibly consider all the subsets of *PR*. In Valletta, the inductive bias is not fixed. The user can choose to give preference to TS descriptions with many kernels and few features and vice versa. The user does this by setting the value of the parameter *MaxKer*. *MaxKer* stores the maximum number of kernels allowed in the final TS description. The problem of kernel selection highlighted the fundamental difficulty of learning multiple-kernel languages. All practical kernel languages encountered all had multiple-kernels. It was therefore important to find an efficient way of performing kernel selection that still allows Valletta to find the target TS description.

The approximation algorithm that was developed for Valletta is based on the following assumptions:

(a) The training set is assumed to be structurally complete.

(b) The number of kernels in the target language can be anything from 1 up to the value of *MaxKer*, which is set by the user. It is assumed that the number of strings in C^+ is much larger than *MaxKer* since otherwise there would, statistically, not be enough evidence for choosing one kernel over another. Kernels must occur at least *MKO*[6] times in the training set. These assumptions are not unreasonable. Every learning algorithm makes assumptions about the 'uniformity' of the training set. For instance, a training set of 100 strings drawn from a two-kernel language should not contain 99 occurrences of one kernel and just one of the other.

(c) When the algorithm discovers the target set of features the set normal forms of C^+ will contain the target set of kernels. Also, after f_3 computation the target

[6] A value entered by the user. MKO stands for Minimum Kernel Occurrences.

kernels would be used to find the minimum distances between the set of normal forms of the strings in C^+.

Given the above assumptions, it is possible to design an approximation algorithm that considers various sets of kernels and computes the value of f_2 for each set. It then returns the set of kernels that returned the smallest value of f_2. It must be stressed that the algorithm does not, and could not possibly, consider all possible sets of kernels. The number of all possible subsets of PR is, of course, $2^{|PR|}$. The approximation algorithm considers only a relatively small number of set of kernels based on a set of carefully chosen heuristics. The main idea behind the approximation algorithm used for kernel selection is that number of kernels in the target TS description is always much smaller than the number of strings in C^+. It is not claimed that this idea is always valid. In fact, it is quite possible that the algorithm will fail on certain datasets in spite of the fact that it worked very well with all of the datasets used to develop and test Valletta.

Kernel Selection Approximation Algorithm

(a) Reduce PR to $2 \times MaxKer$. This is done by assigning, to each normal form in PR, a score based on the number of times that normal form is used to return the minimum distance in the computations of f_3. The justification here is that if a normal form is used many times to return the minimum EvD between pairs of strings in C^+, it is more likely to be a kernel. The normal forms in PR are sorted in ascending order according to this score and the top 2.MaxKer are promoted as *candidate kernels*.

(b) Take the reduced PR and consider only set of kernels of various sizes returned by the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ functions. The values of α and β are entered by the user.

(c) Compute the value of f_2 for each of the above kernel sets and return the kernel set that has the smallest value of f_2.

The $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ functions are defined as follows:

$$\mathbb{S}_\alpha(|PR|) = |PR|.\frac{\alpha}{(log_\mu.|PR|)^\pi} \tag{5.1}$$

$$\mathbb{S}_\beta(|PR|) = |PR|.\frac{\beta}{(log_\mu.|PR|)^\pi} \tag{5.2}$$

The values of α, β, μ, and π are entered by the user and form part of the inductive bias of the algorithm. $|PR|$ denotes the cardinality of the set PR (after reduction). These values allow the user to change the inductive bias to suit the domain. The user can opt to give preference to TS descriptions with more kernels and less features, etc.

The role of the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ functions is to select a manageable number of sets of candidate kernels from PR. Suppose, for example, that $|PR| = 50$. Suppose also that $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ return the values of 5 and 4 respectively. We then consider all possible

subset of sizes 1, 2, 3, 4, 5 and also 47, 48, 49, and 50. The motivation was our observation of the function ${}^nC_R$, i.e. the function that returns the binomial coefficients. This function is very large when r approaches $\frac{n}{2}$. On the other hand, the values returned by the function are relatively small when r is close to 1 or close to n, i.e the 'fringes'. We therefore wanted to design a function that considers only all possible subsets of sizes close to 1 and close to n. This behaviour is depicted in Figure 5.23 below.

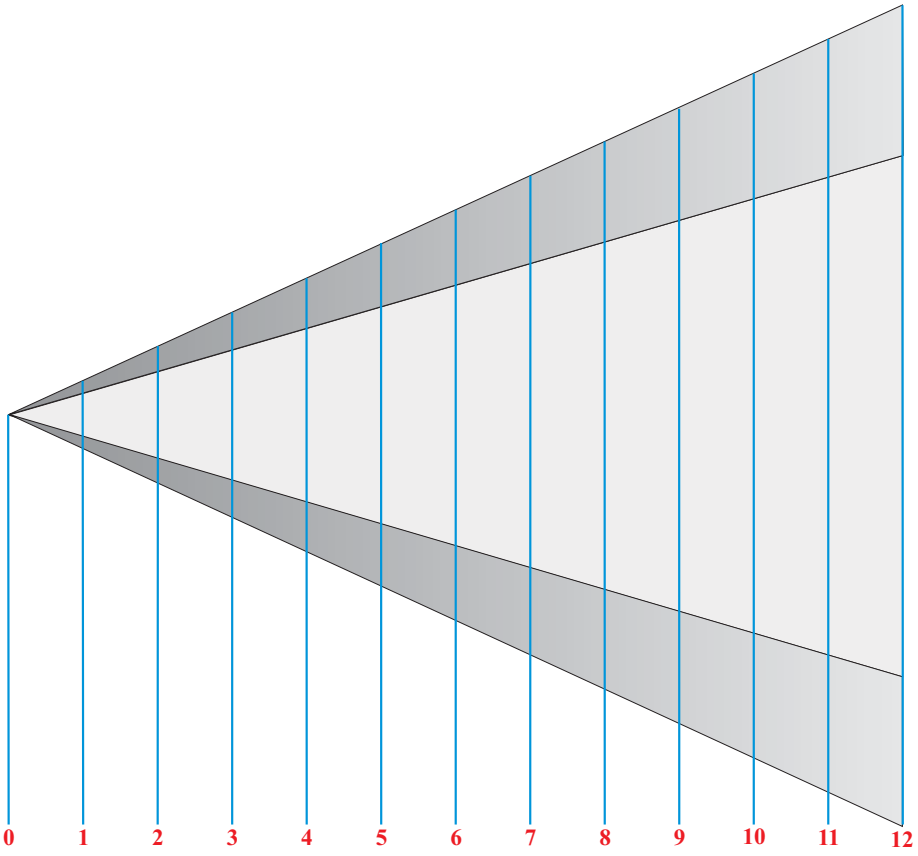


Figure 5.23: A depiction of how the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ functions work.

The values of α and β control the number of subsets of PR of sizes close to 1 and close to $|PR|$ respectively that are considered as kernel sets. If $|PR| = 32$, for example,

then $\mathbb{S}_\alpha(|PR|)$ returns 3 and $\mathbb{S}_\beta(|PR|)$ return 2. We therefore consider all possible subsets of PR of sizes 1, 2, 3, 31, and 32. The parameters in the case are $\alpha = 0.12$, $\beta = 0.10$, $\pi = 7$, and $\mu = 24$. If the user decides to give preference to TS descriptions with a small number of kernels then the value of β can be set to zero or very close to zero. The values of μ and pi can be any positive integers although we found out, through experimentation, that the optimal values of μ are in the range from 12 to 32 and those of π from 2 to 12.

Figure 5.24 shows the values, with $|PR|$ ranging from 1 to 64, of $\mathbb{S}_\alpha(|PR|)$ with $\mu = 24$, $\pi = 7$, and $\alpha = 0.10$. The reader should note how the function has high gain

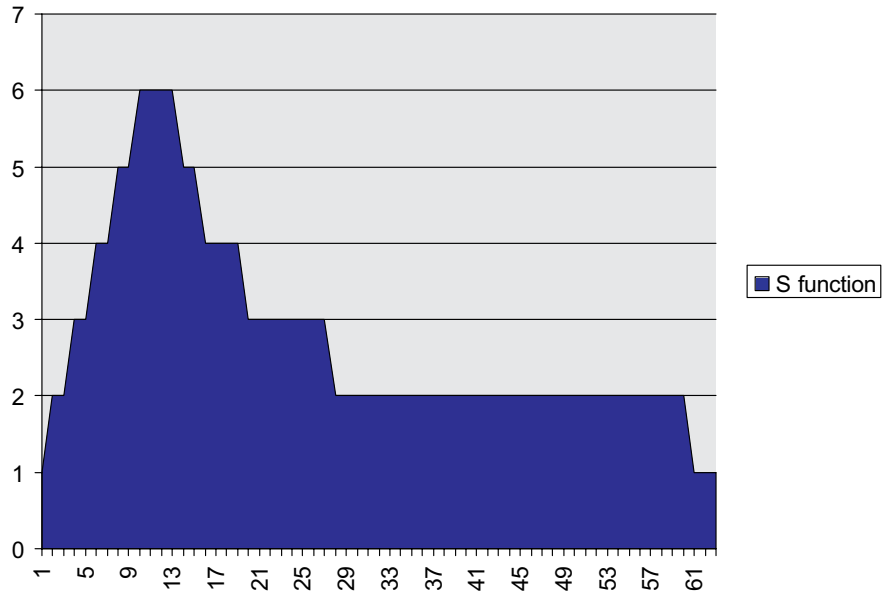


Figure 5.24: A depiction of the kernel selection process.

around $|PR| = 12$. In this case, if $\mathbb{S}_\alpha = \mathbb{S}_\beta$, all possible subsets of PR are considered. The shape of the function can be changed by modifying the values of μ and π. It was not possible to conduct an in-depth analysis of the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ functions. These functions were devised after much experimentation on the MathCAD® mathematical

analysis software package. The results obtained have to be corroborated with further research. The kernel selection approximation algorithm based on these two functions worked extremely well in Valletta. In every case, this kernel selection technique found the correct set of kernels without having to consider all the possible subsets of PR. The values of α and β depend only on the cardinality of the training sets.

The kernel selection returns the subset of PR that returned the smallest value of f_2. The next step is then to compute f_1. This involves computing the EvD between the set of normal forms of C^+ and those of C^- using the set of kernels returned by the kernel selection procedure. For Valletta we used a different definition of f_1 to the one used by GSN. In GSN, f_1 was defined to be the minimum pair-wise distance between C^+ and C^-. If the training set contain misclassification noise, i.e. if, for example, C^- contains a string belonging to the target class C, then f_1 would, clearly, be zero. In order to avoid this problem and allow Valletta to handle misclassification noise, f_1 is defined to be the distance between C^+ and the closest 10% of string in C^-. This is depicted in Figure 5.25. The computation of f_1 is discussed in Chapter 7. When f_1 and f_2 have been computed, the value of f is computed in the usual way:

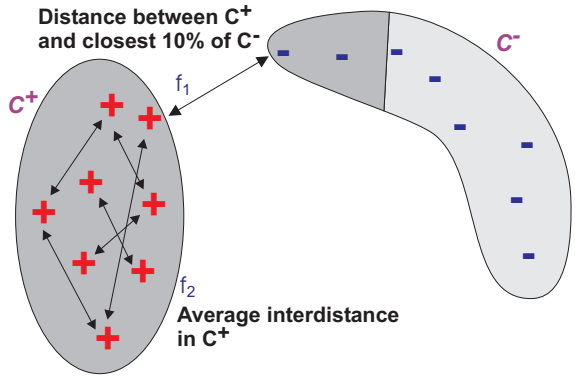


Figure 5.25: Computing the new f_1 function.

$$f = \frac{f_1}{c + f_2}.$$

5.5 Reducing C^+ and C^- to their Normal Forms

Reducing a string modulo a given set of features F appears, *prima facie*, to be a very simple procedure. All one needs to do is to delete all occurrences of the features from the string and thus obtain the kernel. If the set of feature is confluent, the process can be performed in $O(n)$ time where n is the length of the string. If, however, the set of features is non-confluent, then there may be many different ways in which the features can be deleted. The presence of noise further complicates matters since the features themselves or even the kernel may be corrupted. Figure 5.26 shows the *edit-*

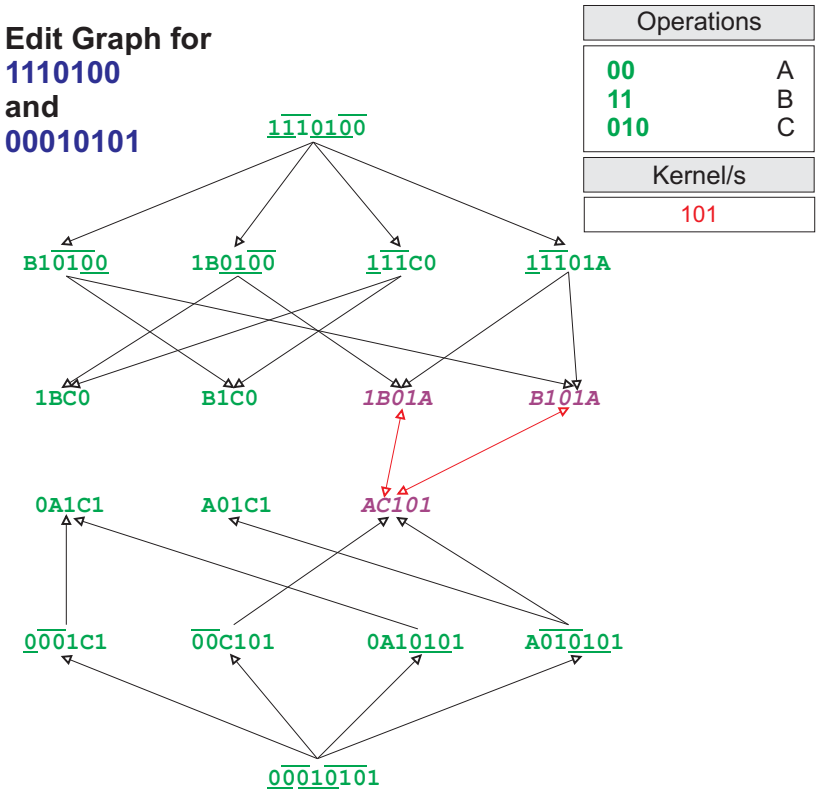


Figure 5.26: The Edit-Graphs for the strings *1110100* and *00010101*.

graphs of the string *1110100* and *00010101* modulo the set of features $\{00, 11, 010\}$. An edit graph is a structure that shows all possible ways that the features can be deleted from the strings to obtain the normal forms. In the edit graph a feature

is not actually deleted from the string but replaced by a symbol (in our case an upper case character) called the *alias*. The is necessary since it provides a means for detecting features inserted inside other features. The alias places the role of the θ symbol in θ-reduction relations. In our case, the strings were drawn from the kernel language with the string *101* as the kernel and set of features as above. Since the features overlap between themselves there is more than one way of deleting the features from the strings. Removing the features in every possible way there will result in more than one normal form for each string. This is depicted in Figure 5.26. It turns out, not surprisingly, that edit graphs grow super-polynomially in the size of the strings. It is therefore not practical to pursue such an approach as a means for string reduction. From very early in Valletta's development it was evident that an efficient method for dealing with what looked like a difficult combinatorial problem was required. In some pathological cases the number of normal forms can be very large. The number of normal forms a string can have is, of course, bound from above by the total number of possible subsequences in the same string. A number of experiments showed that feature sets with high τ numbers, long strings, and low-cardinality alphabets, sometimes conspired to make such pathological cases occur. Valletta, however, depended on having an efficient procedure for reducing strings to their normal forms. The string reduction algorithm used by Valletta uses a data structure called a *parse graph*. A parse graph (of a string) is a structure that stores all the possible ways a string can be reduced modulo a set of confluent or non-confluent features. A parse graph is a directed acyclic graph with two distinguished nodes labelled START and FINISH. Each node is assigned a non-negative integer. The number of nodes is $n + 2$ where n is the length of the string. The START node is assigned the integer 0 and the FINISH node is assigned the number $n + 1$. All other nodes are associated with a position (starting from the left) in the string and are

assigned the appropriate integer. All the edges in the parse graph are labelled with a substring from $\{A \cup \Sigma\}^*$ where Σ is the string alphabet and A is the set of alias symbols that are assigned to the features. Figure 5.27 shows the parse graph of the string *000101110100* modulo the non-confluent set of features $\{00, 11, 010\}$. Every

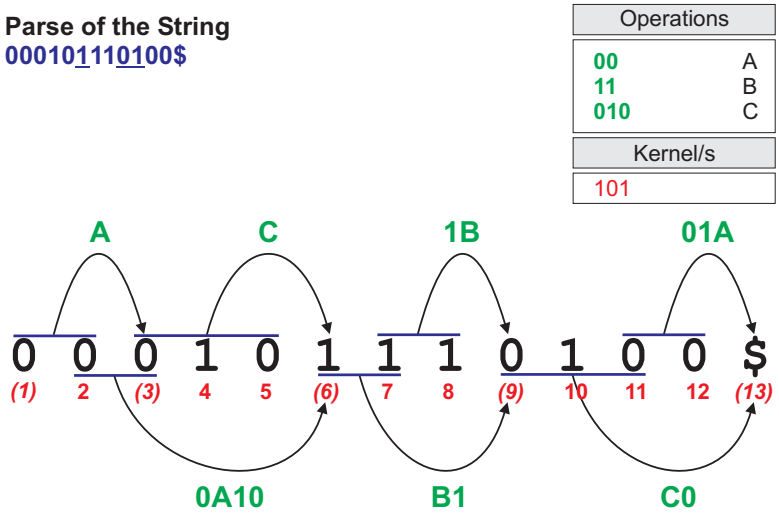


Figure 5.27: A parse of the string *000101110100* using non-confluent features.

node that is incident to an edge is either the position in the string where a feature is found or the position where a feature ends. Such nodes are called *target* nodes. Note that only one feature can start in any given position in the string. In other words, we can only find one feature in F that starts at position i in the string. If this were not the case then there would be features that are prefixes of other features. Parse graphs are useful structures because they store all the possible ways a string can be parsed. They also can be built in linear time and use linear space. From the parse graph in Figure 5.27 it should be evident that one can obtain all the normal forms of the strings by considering all the paths from the START to the FINISH nodes. This can be done using a simple recursive depth-first search algorithm. The normal form associated with a path is obtained by concatenating all the edge labels in the path and then removing the alias symbols. Another important type of node is the

cross-over node. Cross-over nodes are so called because they represent target nodes that are not crossed by edges. Formally, a cross-over is a target node n_i (i denotes the node number) such that there is no edge (n_j, n_k) where $j < i < k$. Nodes 0 and $n + 1$ are, by definition, cross-over nodes. Cross-over nodes are important because they define the *independent segments* of the string. The independent segments are the substrings where the first character is a cross-over node and the last character is the character before the next cross-over node. Consider again Figure 5.27. The cross-over nodes are 1, 6, 9, and 13. The independent segments of the string $s =$ *000101110100* are therefore $s[1, 5]$, $s[6, 8]$ and $s[9, 12]$. An interesting property of the independent segments of a string is that each can be parsed separately. It is possible to extract the normal forms for the first segment and then of the second segment, and so on. Let $I_1, I_2, \ldots, I_n$ be the normal forms of the n independent segments of a string. The normal forms of the whole string can then be obtained by taking, in order, all concatenations in the 'Cartesian product' $I_1 \times I_2 \times \ldots \times I_n$. In Figure 5.27, for example, the normal forms of the independent segments are:

Independent segment$^+$	**Normal forms$^-$**
$s[1, 5]$	ε, *010*
$s[6, 8]$	*1*
$s[9, 12]$	*0*, *01*

Table 5.4: The normals forms of each independent segment.

We then concatenate all pairs of normal forms of the first and second independent segments to obtain a new set of normal forms which we denote by $I_1 \times I_2$ and then concatenate all pairs in $(I_1 \times I_2) \times I_3$. The normal forms obtained from this method are the normal forms of the whole string. The normal forms of the string *000101110100* are *10*, *101*, *01010*, and *010001*. The idea here is that one can find the normal forms of each independent segment separately (perhaps in parallel) and then perform all

the possible concatenations in order to obtains the set of normal forms of the whole string. This is depicted in Figure 5.28.

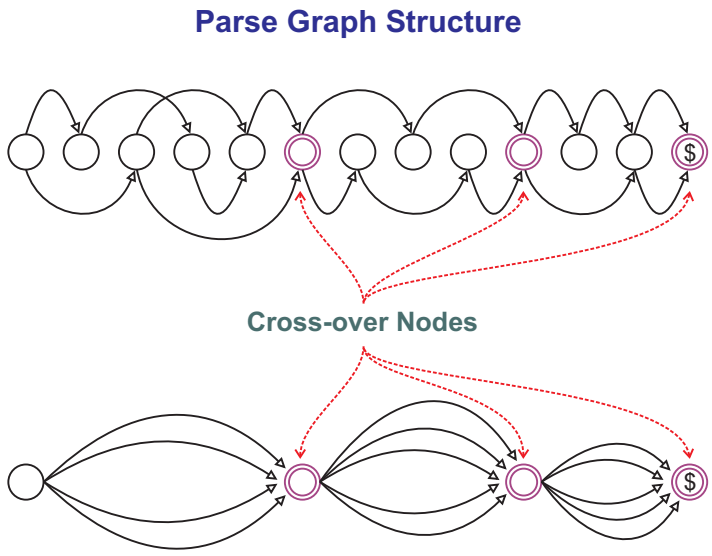


Figure 5.28: The parse graph structure showing the cross-over nodes.

Notice also that the number of normal forms obtained by this method is equal to the number of paths in the complete n-partite graph where n is the number of independent segments and where partition i has $|I_i|$ nodes. This number can, in general, be very large.

A number of experiments carried out with both random strings and strings from actual kernel languages revealed that although, in general, the number of paths in the parse graph was usually relatively large, the number of *unique* normal forms was usually much smaller. A number of important optimizations were therefore incorporated into the string reduction algorithm. The idea was to find a way of extracting all the unique normal forms in the parse graph without considering all possible paths. Notice, for example, that in Figure 5.27 both paths between nodes 6 and 9 yield the same normal form. One of the edges is therefore redundant. It turns out that it is possible to reduce the size of the parse graph by merging certain nodes

and by removing redundant edges. It also turns out that, for Valletta, we do not really need all the normal forms but only those that are less than a certain length (depending on the length of the longest kernel). Before extracting the normal forms, a special preprocessing stage removes the redundant nodes and edges in the parse graph. Consider the simple example in Figure 5.30.

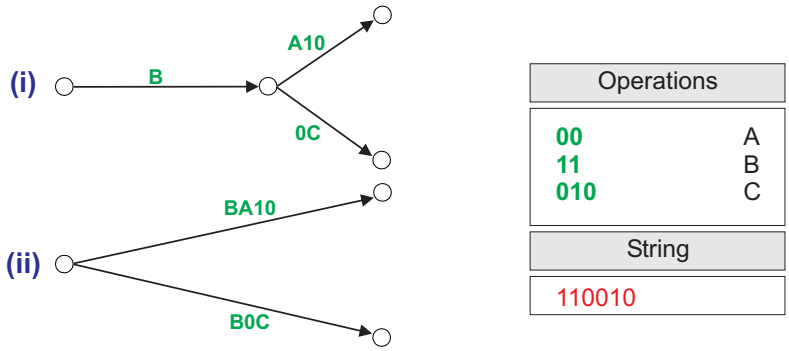


Figure 5.29: Removal of redundant nodes in parse graph reduction.

Figure 5.29 shows how the string *110010* can be reduced using the set of features $\{00, 11, 010\}$. Each feature found in the string is replaced by an upper case letter (called the *alias*) in the parse graph. As can be seen the middle node is redundant and can be removed. The edges are then labelled with the normal forms of the paths that passed through the redundant node.

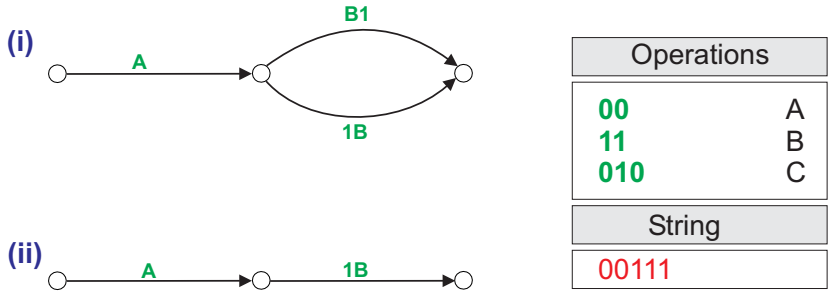


Figure 5.30: Removal of redundant edges in parse graph reduction.

It is sometimes also the case that the parse graph contains redundant edges. An

example of this is shown in Figure 5.30 which shows how the string *00111* can be reduced. In this case, an edge can be removed since the normal forms of both paths are identical. We shall not discuss the rather involved algorithm for parse graph reduction. Our aim here is to outline the general technique. Parse graph reduction is only necessary when the set of features is non-confluent. When the features are confluent the strings have only one normal form and this can be found in linear time.

When parse graph reduction is completed Valletta extracts the normal forms using a simple, recursive, depth-first search of the reduced parse graph. The number of unique normal forms is usually quite manageable. It was observed that for small cardinality alphabets (≤ 3) and strongly non-confluent features with a high τ number (i.e. many features share suffixes and prefixes), the number of unique normal forms can be quite large, sometimes many hundreds. This happens because with small cardinality alphabets the probability of occurrence of each feature is higher. This is a well-known result from the theory of word combinatorics [79]. The normal form extraction algorithm used for Valletta therefore includes yet another optimization. The algorithm does not consider normal forms that are of length $2n + 1$ or more where n is the length of the longest kernel in the set of kernels that is passed as a parameter to the EvD function. This is because, once the algorithm has found at least one normal form of length $2n + 1$ or less, it is useless to consider normal forms that are longer than $2n + 1$ since it can be shown that these cannot possibly have a shorter weighted Levensthein distance to the kernel. This is true only if all insertions/deletion have the same cost. Consider *any* two strings. The weighted Levensthein distance between the two strings is bound from above by $m + n$ where m and n are the lengths of the two strings[7]. This is because, in the worst case, one has to first delete all of one string and then build the second string using character

[7]We are assuming that the cost of insertion and deletion is 1.

insertions. In EvD we have to find the normal form that is the shortest distance to the set of kernels. What we did for Valletta is that, during normal form extraction (from the parse graph), we keep track of the smallest distance between the set of kernels and the normal forms found so far. If this value is x, we then ignore all normal forms that are of length $n + x$.

Valletta's string reduction algorithm is able to extract the normal forms of the strings in C^+ and C^- in reasonable time. Even for strings that are approximately 400 characters long and with non-confluent features, reduction never takes more than a few seconds at most. In summary, the string reduction procedure used is as follows:

(a) Build parse graph of the string,

(b) reduce the parse graph by removing redundant edges and nodes, and

(c) traverse the reduced parse graph considering only normal forms of a given length.

In the implementation of Valletta the actual algorithm for string reduction was rather involved and took many weeks to develop and debug. The string reduction algorithm worked quite well although the topic is my no means closed. Further investigation into the theory and techniques of string reduction is required and the author remains convinced that more efficient methods can be found.

5.5.1 Feature Repair

One of the problems of GLD was that if a feature has an extra character inserted somewhere along its length, the whole feature would be considered as noise. This problem limited the number of noisy features allowed in a string. We solved this problem in Valletta by modifying the string reduction algorithm to handle what we called *feature noise*. Feature noise consists of spurious characters inserted *inside* the features. When the string reduction algorithm is building the parse graph it tried to find the features that occur inside the input string. When *feature repair* is enabled, the algorithm also searches for features that have been corrupted by the insertion of extra characters. The number of extra characters that are allowed inside the features is determined by the *MFN* parameter entered by the user. *MFN* is a real number in the interval $[0, 1]$. A value of 0.25, for instance, means that up to 25% of the feature (by length) can be corrupted. In Figure 5.31, the feature *ab* is corrupted by the insertion of an extra '*c*'. When building the parsee graph the algorithm identifies the

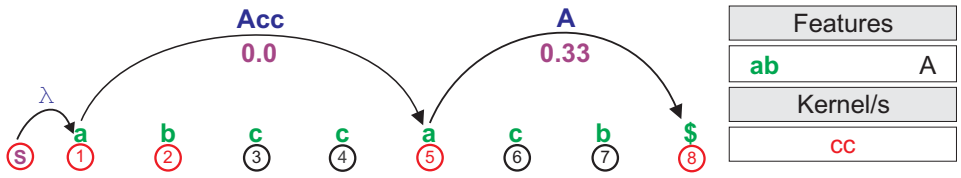


Figure 5.31: How *feature repair* works.

substring *acb* as being a corrupted version of the feature *ab* and assigns a weight of 0.33 (the cost of deleting the 'c') to the edge. When the normal forms are extracted from the parse graph, the weights of the edges are added together and assigned to each normal form. This value is called the *feature repair cost* of the normal form. The EvD between two strings is then the WLD between the normal forms and the set of kernels plus the feature repair costs. The feature repair facility worked well and allowed significantly higher levels of noise in C^+. The only problem is that the

computational complexity of parse graph construction is increased since the algorithm has to search for corrupted versions of the features.

5.6 Summary and Discussion

The complexity of learning multiple-kernel languages necessitated a new approach to ETS learning of kernel languages. Valletta is a more complex algorithm than the GSN algorithm precisely because the environment is more complex. Multiple kernels, non-confluence, and noise increase the number of normal forms, distance computation, the search space, and hence learning. Valletta has to search not only in the space of all possible features (i.e the repeated substrings) but also in the space of all possible kernels (i.e. all the normal forms of C^+). In addition, Valletta uses a much more relaxed definition of structural completeness. This makes the search space much larger. The addition of noisy strings complicated matters further. The two main concerns that arose during Valletta's development was whether it was possible to find efficient algorithms for string reduction and also for kernel selection. The string reduction algorithm, which uses parse graphs, works quite well. Kernel selection is, in general, NP-Hard. The approximation algorithm used is based on the premise that some kernels are more likely to be in the target TS descriptions than others and also the assumption that the training set is uniform in the sense that the kernels are uniformly distributed in the training data.

Most learning algorithms exploit some partial order on the search space to direct the search. Valletta uses only distance, or rather, an ordering based on distance. Valletta borrows some ideas from both the $A*$ and *Beam* search techniques. It expands the search solely on the basis of the value of f_2. No other heuristics are used. As the results in Chapter 7 show, this approach gives good results but this is

not to say that the search technique cannot be improved further (See Chapter 8).

Another of Valletta's features is the pre-processing stage. The pre-processing stage identifies the search space of the learning stage. Preprocessing can yield a lot of information and drastically reduce the time and space complexity of the task at hand. This is a lesson learned from Bioinformatics [57]. The preprocessing stage produces the set of repeated substrings R_{C^+} and the search lattice L_{C^+}. Valletta's search space is the set of all anti-chains in L_{C^+}.

Valletta, like the GSN algorithm does not really optimize the f function. It stops searching when f exceeds a user-input threshold value. This, of course, is not equivalent to optimizing f over the search space of all possible TS descriptions that are consistent with C^+.

Chapter 6

Valletta Analysis

The aim of this chapter is to analyse the Valletta algorithm. In particular, to discuss the time and space complexities of the various stages of the algorithm and also its ability to converge to a TS description consistent with a set of training examples that is structurally complete.

6.1 Time Complexity of the Preprocessing Stage

The preprocessing stage performs the following independent procedures:

(a) The construction of the global augmented suffix trie (GAST).

(b) Traversal of the GAST in order to extract all non-overlapping repeated substrings in C^+.

(c) Construction of the search lattice.

1. Construction of the GAST

The construction of the GAST is performed, on average, in $O(n \log n\, |C^+|)$ time where n is the length of the longest string in C^+. In the worst case this bound is

$O(n^2\,|C^+|)$. The GAST construction algorithm used by Valletta is loosely based on the suffix trie construction algorithm discussed in [114]. The main difference is that Valletta does not build the suffix trie of a single string but of all the strings in C^+. It also records additional information on the number of times a suffix has been found and in how many strings. As discussed in [114, Page 201], it is possible to reduce the worst case upper bound to $O(n\,log^2\,n\,|C^+|)$ by using the technique proposed by Apostolico and Preparata (see references in [114]). The author did not bother with further optimization of GAST algorithm since it worked very well and constructed the GAST in a few seconds even for the largest datasets.

It is possible to use suffix trees instead of suffix tries to obtain the same results. Suffix tree can be constructed in linear time and are therefore asymptotically faster but the algorithms for their construction are substantially more complex.

2. Traversal of the GAST

After the GAST is constructed, Valletta's preprocessing stage then traverses the GAST in order to extract all non-overlapping repeated substrings. As with all tree traversal algorithms, the time complexity of this step is linear in the size of tree. The author conducted a number of experiments on sets of strings of various cardinalities, of various lengths, and over different alphabets. It was observed that with larger alphabets, the GAST may grow to contain a number of nodes that is quadratic in the size of the set of strings. The size of a set of strings is here defined to be the sum of the lengths of the strings in the set. However, the GAST is built from C^+ and not from random strings. For obvious reasons, the strings in C^+ contain a lot of regularity (repeated substrings) and this ensures that the GASTs remain relatively small. The *a702* dataset, which was the largest dataset used to test Valletta, contained 128 strings with lengths of up to 304 characters and generated a GAST with 1.5 million

nodes in just under 18 seconds[1]. The size of the training set was 18,253 characters.

3. Construction of the Search Lattice

The final task performed by the preprocessing stage is the construction of the search lattice. The depth of the search lattice is, on average $O(log_m\, n)$ where n is the number of nodes in the lattice and m is the cardinality of the alphabet. Search lattice construction is performed, on average, in $O(n\, log_m\, n)$ time. In the worst case this can be asymptotically worse, $O(n^2)$, although it is very unlikely that this bound is ever encountered in practice.

6.2 Time Complexity of String Reduction

As we saw in the previous chapter, string reduction is a problem that has a very simple description but that does not have a simple solution. The string reduction algorithm used in Valletta was refined and improved over a number of months until it performed the task quite well. It is difficult to give asymptotic bounds on the time complexity of string reduction. If the set of features is confluent, reduction of a string to the sole normal form can be performed in linear time. If the features are non-confluent, then there are many factors that can influence the time complexity. In the worst case, the number of normal forms can be super-polynomial to the length of the string but this happens in extremely pathological cases (see Appendix G). If the set of features has a high τ number (see Chapter 3) and is therefore strongly non-confluent and the alphabet is of small (i.e. 2-4) cardinality then, if the strings are relatively long (i.e. > 100 characters), the number of normal forms can, in some cases, be large (many hundreds or even thousands). A number of experiments performed on both randomly generated strings and also on strings from a kernel language confirmed

[1]On a Pentium II PC running at 800MHz.

that the cardinality of the alphabet has a strong influence. This is because, features over a small alphabet have a high probability of occurring in random strings over the same alphabet. The length of the strings in C^+ was not really a factor. In the *bin02* dataset, for instance, when C^+ was reduced using the target set of features $\{00, 11, 010\}$, it yielded 2125 normal forms of which 482 were unique. The longest string in C^+ was 85 characters long and was reduced to 59 different normal forms. The maximum number of normal forms for any string in C^+ was 82. The negative training set C^-, which consisted of randomly generated strings, yielded 509 normal forms. The number of normal forms always remained relatively low. Both C^+ and C^- each had 64 strings. The algorithm used in Valletta to perform string reduction of a string s of length n had three stages:

(a) Construction of the match matrix. This is performed in $O(nm)$ time where m is the number of features.

(b) Construction of the parse graph. This procedure is $O(nm)$

(c) Traversal of the parse graph in order to extract the normal forms. This is performed by a simple depth-first recursive algorithm. The time complexity depends on the number of normal forms and the τ number of the set of features. In practice it turned out that string reduction was always performed very quickly.

The string-reduction algorithm also included a number of optimizations, like *parse graph reduction* and *edge-merging* (See Chapter 5), which reduced the overall complexity. As far as the author is concerned this topic is still an open area. Although the string reduction algorithm performed very well, the author remains convinced that faster and more efficient methods can be developed.

6.3 Time Complexity of Computing f

Given a set of features F, the computation of the f function involves the following steps:

(a) Reduction of both C^+ and C^- modulo the set of features F,

(b) Computation of the distance matrix,

(c) Computation of f_3,

(d) Kernel Selection and computation of f_2, and

(e) Computation of f_1.

Reduction of the strings in C^+ and C^- is discussed in the previous section. The computation of distance matrix involves computing the weighted Levensthein distance (WLD) between all pairs of normal forms of each string in C^+. The number of WLD computations is therefore bound from above by

$$\sum_{s \in C^+} | \Downarrow_F (s)|^2.$$

where $\Downarrow_F (s)$ is set of normal forms (modulo F) for the string s. The time complexity of WLD is quadratic in the size of the two strings. It must be pointed out that in the actual implementation it is not necessary to compute the WLD between all pairs of normal forms but rather between all pairs of *unique* normal forms. This gives a significant time saving since the set of normal forms of C^+ usually contains many repeated strings.

Computation of f_3, the average WLD between the sets of normal forms of the strings in C^+ is quadratic in $|C^+|$ and N, the number of normal forms of C^+. For f_3 we do not need to compute the actual distances since the WLD between all pairs of

normal forms can be found in the distance matrix. The algorithm to compute f_3 accesses the distance matrix in order to find the distance between any two normal forms.

Kernel selection, as is shown in Appendix H, is NP-Hard. An exhaustive search for the set of kernels that minimizes f_2 is therefore out of the question. The approximation algorithm that is used in Valletta considers a number of kernel sets that depends on the value of the functions $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$. The reader is referred to page 206 for a discussion of these functions. The number of different kernel sets considered by the approximation algorithm depends on the values of the parameters of the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$. These parameters are input by the user. The role of the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ is to reduce the number of kernel sets considered by the algorithm to a manageable level. The kernel selection procedure returns the set of kernels that give the minimum value of f_2. Valletta therefore has to compute the value of f_2 for every kernel set considered by the kernel selection procedure. It must be emphasized, however, that when computing f_2 it is not necessary to perform the actual distance computation since, as was the case with f_3, the WLDs between all pairs of normal forms are stored in the distance matrix. The kernel selection procedure returns the kernel set that minimizes f_2. Finally, computation of f_1 involves computing the minimum WLD between the set of normal forms of C^+ and the set of normal forms of C^-. This step is performed in $O(|C^+|.|C^-|)$ time.

6.4 Convergence

Valletta's search engine uses what can best be described as a hybrid *distance-driven Beam/A* search technique* to find a TS description that is consistent with the training set. One desirable attribute of this technique is that it is relatively easy to show that Valletta always finds a TS description that is consistent with the training set if one exists and if the training set is structurally complete. It should be evident to the reader that if the training set is structurally complete then the target set of features is a subset of the set of all the non-overlapping repeated substrings in C^+, i.e. the *candidate features*. The algorithm finds a feature set that is consistent with the training set. Of course, as with every learning algorithm, it cannot be guaranteed that the algorithm finds the target set of features since, in general, there may be many set of features (TS descriptions) that are consistent with the training data. Valletta always finds one such TS description if it exists. This is because Valletta's search technique enumerates the search space. It is therefore clear that if a consistent TS description exists it is eventually found. To show that this is indeed true we have to show that:

(a) The completed search tree TR contains a node for each of the valid (substring-free) features sets in the set of repeated substrings R_{C^+}. Each valid feature set is an *anti-chain* [59] in the search lattice R_{C^+}.

(b) Valletta continues expanding the search tree TR until either a valid TS description is found or until TR is complete.

Before the construction of the search tree begins, Valletta first computes the value of f for each of the candidate features and then orders the list R_{C^+} in ascending order according to the value of f_2. This ordering of R_{C^+} is then fixed for the learning process. The search tree TR is then initialized to a tree with a single level consist-

ing of the root node and $|R_{C+}|$ children with each leaf labelled with a unique string from R_{C+}. In TR, every node is labelled with a string from R_{C+}. The feature set associated with any given node is the set of strings that are labels of the nodes on the path from that node to the root. When a node is expanded, i.e. by adding child nodes, only strings in R_{C+} that have a larger value of f_2 are added.

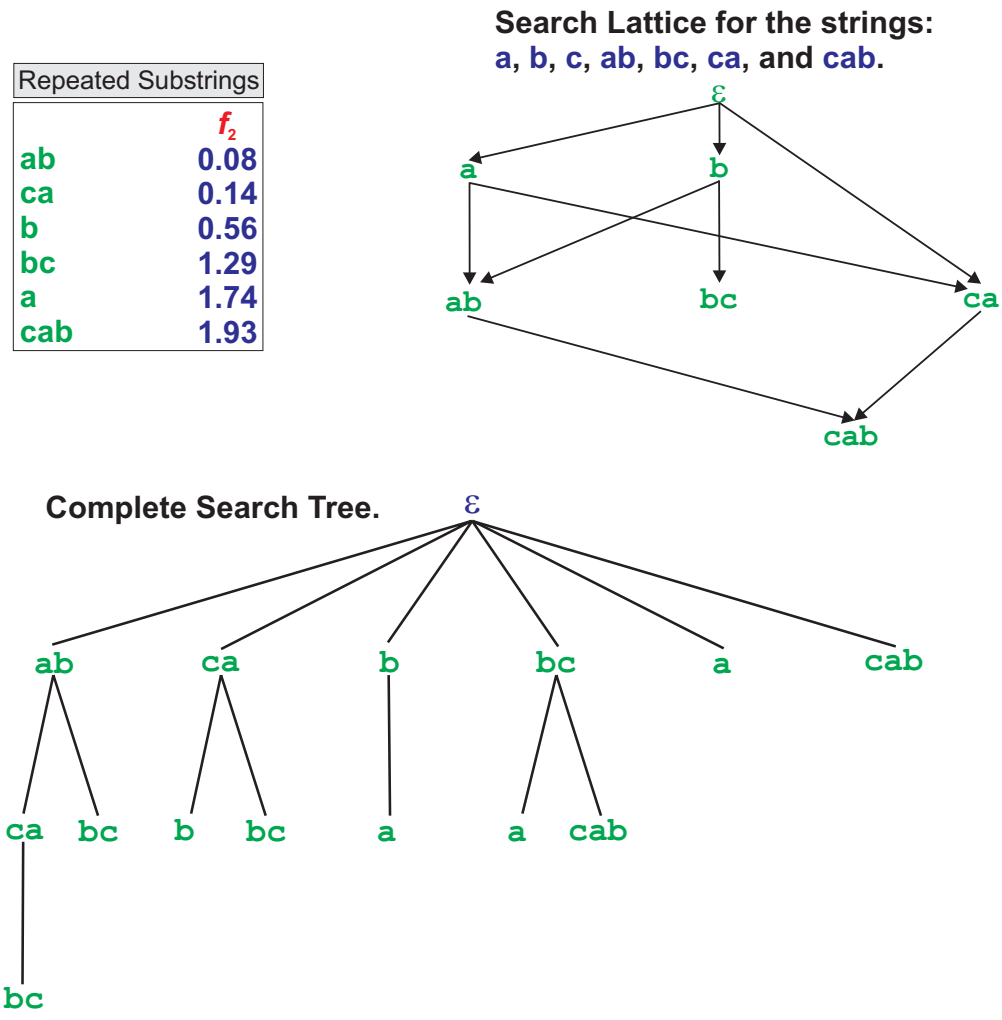


Figure 6.1: The search tree created from the strings $\{a, b, ab, ca, bc, cab\}$.

This ensures that each node in TR represents a unique feature set. In addition, the search lattice L_{C+} is consulted to ensure that the new node represents a valid

feature set. The nodes in TR represent feature sets that are anti-chains in the search lattice L_{C^+}. This follows from the fact that each feature set is substring free. The complete search tree, therefore, contains *all* valid set of features.

As we saw in Chapter 5, a node remains in the ACTIVE and PENDING lists until no more child nodes can be added. When this condition is reached the node is deleted from the ACTIVE or PENDING lists and added to the CLOSED list. A node is added to the CLOSED list only when either no more strings remain in R_{C^+} that have a larger value for f_2 or when all the remaining strings in R_{C^+} that have a larger value f_2 cannot be added since it would result in an invalid feature set. Once added to the CLOSED list a node remain there until the algorithm terminates. Valletta keeps on adding nodes to the search tree until all nodes are in the CLOSED list. When all the nodes are in the CLOSED list, the tree is complete and contains all the valid subsets of R. This means that Valletta always finds a valid TS description that is compatible with the set of training examples if one exists. Figure 6.1 shows the complete search build by Valletta on a very small test dataset. Valletta was modified for this test so it does not stop when it finds a TS description consistent with the training sets but continues expanding the tree until the tree was complete. Note how each node is expanded by adding child nodes labelled with strings in R_{C^+} that have a larger value of f_2 than the parent. Note also that each node in the completed search tree represents an anti-chain in the search lattice L_{C^+}. In practice, of course, Valletta does not build the complete search tree since this is usually very large.

Chapter 7

Experimentation, Testing, and Results

In this penultimate chapter we document, discuss, and analyse the results obtained from the tests and experiments there were conducted with Valletta. Valletta was tested with both artificial and 'real-world' datasets, including datasets created by other researchers to test their learning algorithms. Four groups of datasets were used for testing Valletta and evaluating its performance:

- A suite of datasets that was purposely designed to test Valletta's performance, robustness, learning in the presence of noise, and also to aid the debugging process.
- The GSN datasets used by Nigam in his Master thesis (see Section 7.3).
- The datasets for the *Monk's Problems* (see Section 7.5).
- A dataset created for investigating a Backpropagation neural network's performance on the parity problem (see Section 7.7).

The datasets were carefully chosen so as to show how Valletta performs in the presence of different types of noise, and whether it is robust, i.e. whether a change of the training set affects its performance. In order to demonstrate that Valletta's distance-driven search technique was better than average, the author also designed and implemented a Genetic Algorithm (GA) search engine. The GA was tested with all the datasets in the first group and its performance was compared to that of Valletta. This is discussed in Section 7.2. The author also downloaded the code for the EDSM DFA learning algorithm that won the Abbadingo DFA learning competition (See Appendix E). The EDSM algorithm is compared to Valletta in Section 7.6. A sample of the datasets used to test Valletta can be found in Appendix C. For obvious reasons, not all the datasets are included in Appendix C.

7.1 Valletta's Testing Regimen

In order to test Valletta, evaluate its performance, and also to aid in the debugging process, a suite of 12 datasets was produced. The datasets were chosen to be as heterogenous as possible. The datasets were generated by a small program, *tset*, which given a number of input parameters from the user, automatically generated positive and negative training sets. The user must enter the number of strings in both C^+ and C^-, the minimum and maximum string length, the set of kernels and the set of features, and a random number seed. Every effort was made in order to make the training sets as varied as possible. Some are confluent and some non-confluent. The number of characters in the alphabet varies from 2 to 11. The number of kernels varies from 1 to 3 and the number of features varies from 2 to 7. The length of the features varies from 2 to 13 characters. The length of the strings varies from 3 up to approximately 400 characters and the number of strings in the training sets varies from around 10 to several thousand. The training sets, listed below, also contain

all the types of noise that Valletta can handle, i.e. feature noise, kernel noise, and misclassification noise.

Training datasets used to test Valletta

bin01 Binary alphabet, medium sized training set (≈ 35), single kernel, no noise.
Kernel: *101*.
Features: *11*, *00*, and *010*.
Non-confluent.

bin01n Binary alphabet, medium sized training set (≈ 35), single kernel, 2% misclassification noise.
Kernel: *101*.
Features: *11*, *00*, and *010*.
Non-confluent.

bin02 Binary alphabet, large sized training set (> 64), single kernel, no noise.
Kernel: *101*.
Features: *11*, *00*, and *010*.
Non-confluent.

a301 Three-letter alphabet $\{a, b, c\}$, medium sized training set (≈ 45), one kernel, no noise.
Kernel: *abbcb*.
Features: *bab*, *abba*, *cacba*, and *bbcab*.
Non-confluent.

a302 Three-letter alphabet $\{a, b, c\}$, medium sized training set (≈ 45), two kernels, no noise.
Kernels: *abbcb*,*cccbbc*.
Features: *bab*, *abba*, and *cacc*.
Non-confluent.

a302n Three-letter alphabet $\{a, b, c\}$, medium sized training set (≈ 45), two kernels, 12% noise.
Kernels: *abbcb*,*cccbbc*.
Features: *bab*, *abba*, and *cacc*.
Non-confluent.

a701 Seven-letter alphabet $\{a, b, c, d, e, f, g\}$, medium-sized training set (≈ 40), two kernels, no noise.
Kernels: *gadcffb*,*acbcffgcbbe*.
Features: *bfg*, *gacd*, *acefb*, *fbacd*, *ggfcbd*, *cdfba*, and *acdeeegb*.
Non-confluent.

a702 Seven-letter alphabet $\{a, b, c, d, e, f, g\}$, large training set (≈ 128), two kernels, no noise.
Kernels: *gadcffb*,*acbcffgcbbe*.
Features: *bfg*, *gacd*, *acefb*, *fbacd*, *ggfcbd*, *cdfba*, and *acdeeegb*.
Non-confluent.

a703 Seven-letter alphabet $\{a, b, c, d, e, f, g\}$, small training set (≈ 24), two kernels, no noise.
Kernels: *gadcffb*,*acbcffgcbbe*.
Features: *bfg*, *gacd*, *acefb*, *fbacd*, *ggfcbd*, *cdfba*, and *acdeeegb*.
Non-confluent.

a703n Seven-letter alphabet $\{a, b, c, d, e, f, g\}$, small training set (≈ 24), two kernels, 10% noise.
Kernels: *gadcffb*,*acbcffgcbbe*.
Features: *bfg*, *gacd*, *acefb*, *fbacd*, *ggfcbd*, *cdfba*, and *acdeeegb*.
Non-confluent.

a1101 Eleven-letter alphabet $\{a, b, c, d, e, f, g, h, i, j, k\}$, small training set ($\approx 15$), three kernels, no noise.
Kernels: *hgaidcfkfb*, *fkighbdafkfb*, *jahbcfikgcbde*.
Features: *aih*, *degai*, *acefb*, *bfjbace*, *cjkcbd*, *cdfba*, and *facdiikhgeeb*.
Non-confluent.

a1101n Eleven-letter alphabet $\{a, b, c, d, e, f, g, h, i, j, k\}$, small training set ($\approx 15$), three kernels, no noise.
Kernels: *hgaidcfkfb*, *fkighbdafkfb*, *jahbcfikgcbde*.
Features: *aih*, *degai*, *acefb*, *bfjbace*, *cjkcbd*, *cdfba*, and *facdiikhgeeb*.
Non-confluent.

During the course of Valletta's development and debugging numerous other datasets were created and used for testing and debugging. The above are the 'official' testing datasets.

Results

Valletta found the correct TS descriptions all the kernel languages of each the training sets. In this section we document and discuss briefly the results obtained but refrain from an analysis of the results. This is done later on in Section 7.8. Table 7.1 shows a listing of the training sets and the results obtained. The column

Problem	**Σ**	**$\|C^+\|$**	**Len**	**$\|C^-\|$**	**Len**	**Reps**	**Nodes**	**Pass**	**Time**
bin01	2	6	9.3	8	21.0	25	47	4	11.5
bin01n	2	6	10.7	8	21.0	27	55	6	15.2
bin02	2	63	44.2	63	29.6	47	162	10	2,080.4
a301	3	29	75.6	15	49.8	118	67	5	305.4
a301b	3	29	75.6	15	49.8	118	97	9	539.2
a302	3	14	31.1	15	49.8	40	61	4	38.2
a302n	3	14	32.1	15	49.8	34	67	5	44.9
a701	7	41	72.4	15	104.2	17	83	8	233.5
a702	7	128	142.6	32	105.3	15	179	11	5,831.3
a703	7	24	87.3	15	104.2	18	89	8	217.0
a703n	7	24	89.3	15	104.2	21	153	9	578.2
a1101	11	15	181.0	11	97.8	18	89	8	195.3
a1101n	11	15	183.4	11	97.8	21	113	9	273.6

Table 7.1: Results obtained from testing Valletta.

with the heading $\mathbf{\Sigma}$ shows the alphabet size. The next four columns show the cardinality of C^+, the average length of the strings in C^+, the cardinality of C^-, and the average length of the strings in C^- respectively. The column with the heading **Reps** shows the total number of non-overlapping repeated substrings found by Valletta's preprocessing stage. This value is important since it gives an idea of the size of the

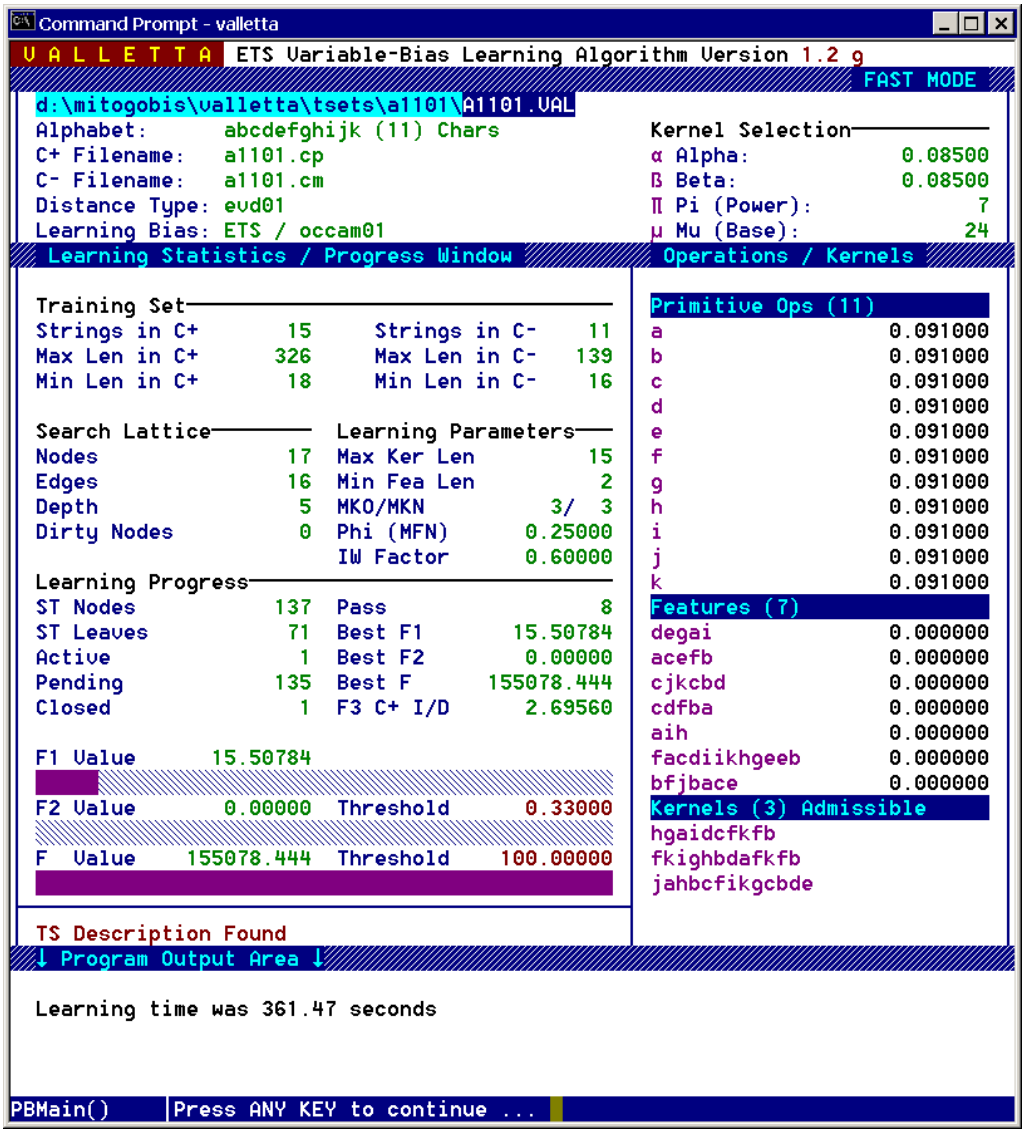


Figure 7.1: Screen dump of *Valletta* when learning of A1101 was completed.

search space. The reader is reminded that, given a set R of repeated substrings, the search space is the power set $\mathcal{P}(R)$. The column with the heading **Nodes** shows the number of nodes in Valletta's search tree when learning was completed. This number represents the total number of different feature sets (TS descriptions) considered by Valletta. The last two columns contain the number of passes (iterations) performed by Valletta's learning stage before a TS description consistent with the training sets

was found and the total time in seconds that elapsed before the final TS description was found. Figure 7.1 shows a screen dump of Valletta after it had completed learning on the *A1101* dataset. The A1101 dataset is interesting because it has 3 kernels, 7 features (with the longest feature being 13 characters long), and very long training strings (> 300 characters) and yet Valletta found the correct class description in just over 6 minutes. The TS description found by Valletta, i.e. the features and kernels, is displayed on the left of the screen. Valletta also displays the values of f, f_1 and f_2 and a bar graph of these values as well as various other learning statistics and parameters.

Some Observations

- For each of the 12 datasets, Valletta discovered the correct class description after considering only a very small part of the total search space. In all cases Valletta found the correct TS description after considering less than 200 different class descriptions. Valletta's distance-driven search proved to be very efficient.

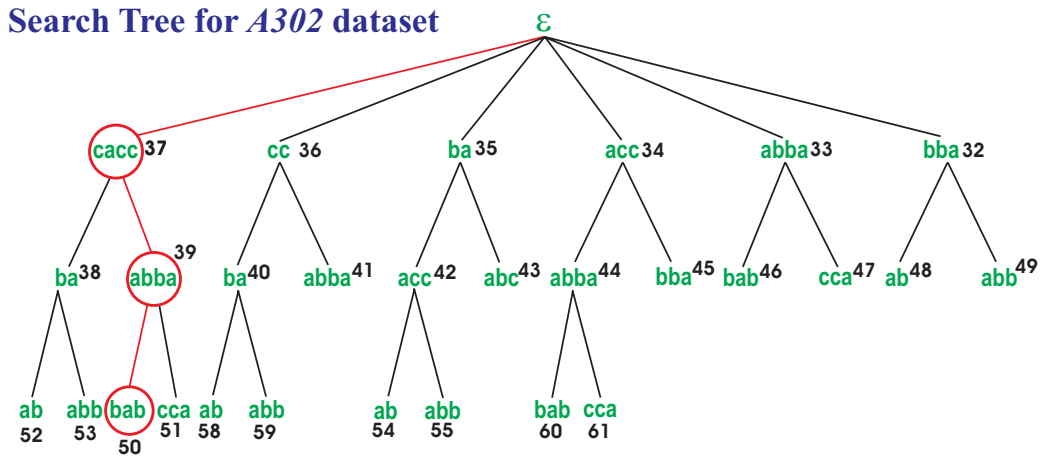


Figure 7.2: The search tree created by Valletta for the *a302* dataset.

As explained in Chapter 5, the search tree is expanded only on the basis of the

value of f_2, the average pair-wise distance in C^+. To put things into perspective, the smallest search space, i.e. that of the *bin01* dataset, contained 2^{15} different feature sets. Figure 7.2 shows the actual search tree built by Valletta for the *a302* dataset. The nodes shown circled in red represent the final TS description. For this dataset Valletta created a search tree with 61 nodes. The number shown in black to the left of each node is the actual node number. Due to restricted space, only the nodes in the first level of the tree that have children are shown.

- Valletta took just over 1.5 hours to find the correct class description for the *a702* dataset. This was not because the search is inefficient. In fact, the number of different feature sets considered by Valletta before it found the correct class description was 179. This was only slightly more than that for the *a703* dataset which was drawn from exactly the same kernel language. The *a702* dataset, however, contained many more strings (128 in C^+) and the average length of the strings was much longer. This adversely affected the learning time. This issue is discussed in Section 7.8.

- The number of features and/or the number of kernels did not seem to affect Valletta. Valletta learned kernel languages with 7 features and 3 kernels with no problems at all.

- Valletta performed much better on alphabets of higher (≥ 4) cardinality. This is because, with high cardinality alphabets, the pre-processing stage finds fewer candidate features in C^+. This issue is discussed in Section 7.8

- The values of the I and J parameters were both set to 3 for all datasets. Valletta therefore always expanded the top three nodes in the PENDING list and three other nodes chosen according to the heuristic described in Chapter 5.

7.2 The *Darwin* Search Engine

As we noted in the previous section, Valletta learned the correct TS description of each kernel language in the testing datasets after considered a relatively very small number of feature sets (i.e. TS descriptions). Valletta therefore manifested very quick convergence to the target TS description. As has been emphasized many times in this thesis, Valletta's search is entirely distance-driven. At each iteration of the learning loop only the value of the f_2 function is used to determine which features are considered next. Valletta's performance is analysed later on in Section 7.8. In this section we discuss the *Darwin* genetic algorithm (GA) search engine which was added to Valletta in order to perform comparisons between Valletta's distance-driven search and the *adaptive search* technique used by genetic algorithms [88]. Valletta's results left us in no doubt that distance-driven search performs much better than random search. The reason why the Darwin GA engine was implemented was because the author wanted to compare Valletta's search technique to another search technique that is in widespread use. Valletta's modular design made it relatively easy to replace the distance-driven search engine with the GA engine. Exactly the same pre-processing stage is used together with the same distance function, EvD. Moreover, the same methods of computing f, f_1, and f_2 are retained. All that was changed was the actual search technique. In this way, one could compare the two different search techniques — with all other things being equal. Darwin actually forms part of Valletta — the user is given the option to decide which search technique to use — ETS distance-driven search or the GA engine.

The Darwin search engine uses conventional GA adaptive search techniques. Valletta's pre-processing stage first finds all the candidate features in the usual manner as described in Chapter 5. Darwin then creates an initial population of 256 chromosomes. Each chromosome is a string in which each character corresponds to one of

the candidate features. In other words, if the preprocessing stage finds n candidate

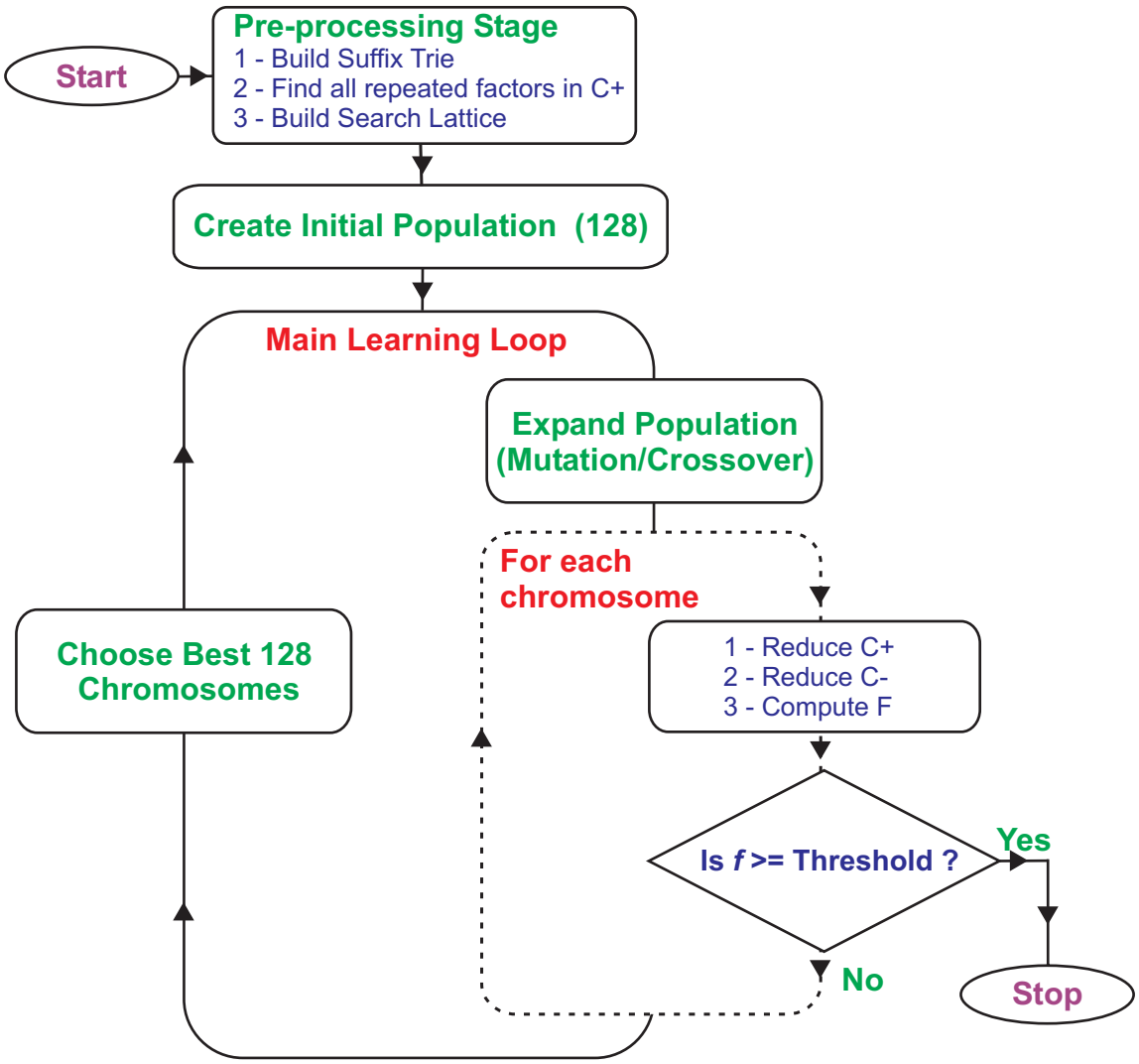


Figure 7.3: High-level flowchart of the Darwin genetic algorithm search engine.

features, then the chromosomes are strings over an alphabet of n characters. Each chromosome, therefore, represents a set of features. After the initial set of chromosomes is created (i.e. the initial population), Darwin then enters into the main learning loop. This loop is repeated until learning is achieved or until a specified number of iterations is made without the algorithm registering an improvement in the value of f_2. This is depicted in Figure 7.3. In the main learning loop the set of

chromosomes is expanded via the *mutation* and *crossover* genetic operations. The mutation operator is a unary operator. It acts on one chromosome and makes random changes to a number of randomly selected character positions. A new chromosome is thus created. The crossover operator is a binary operator. It acts on two chromosomes and selects a random position in each of the chromosomes. The first part of the first chromosome is joined to the second part of the second chromosome to create a new chromosome. The second part of the first chromosome is joined to the first part of the second chromosome to create another new chromosome. Let us illustrate with an example. Suppose that the pre-processing stage finds the following set of 7 candidate features:

$$00,\ 101,\ 1011,\ 0110,\ 11,\ 010,\ 111001.$$

The chromosomes are then drawn from the set of all strings over the alphabet $\Sigma = \{a, b, c, d, e, f, g\}$. Note that $|\Sigma| = 7$. Now consider the chromosome acd. This represents the set of features $\{00, 1011, 11\}$. Suppose that through mutation the new chromosome afd is created (i.e the c was changed into an f). The new chromosome represents the set of features $\{00, 010, 11\}$. To see how crossover works consider the following two chromosomes:

$$\text{acde and fgbd.}$$

Suppose that the 3^{rd} and the 4^{th} positions are randomly chosen in the first and second chromosomes respectively. Each chromosome is now divided into two parts. Crossover then entails joining the first part of the first chromosome to the second part of the second and the second part of the first chromosome to the first part of the second. This creates the two new chromosomes:

$$\text{acd and fgbde.}$$

The fitness function used in Darwin is actually f_2 itself. It must be pointed out, however, that Darwin does not use f_2 to decide which feature sets to consider. This is done by the mutation and crossover operators we just described. Valletta, on the other hand, uses f_2 to decide which features are to be added to the current set of features. This is a fundamental and important difference. Darwin's search is not distance-driven — distance is used only as a stopping criterion. Valletta uses distance to direct the search and also as a stopping criterion, i.e. to determine when a correct class description has been found.

Darwin expands the population of chromosomes by adding 128 new chromosomes through mutation and a further 128 chromosomes through crossover. The value of f is then computed for each of the chromosomes and the fittest 256 chromosomes, i.e. the 256 chromosomes with the lowest values of f_2 are selected. The rest are discarded. This cycle is repeated until either a TS description valid with the training sets is found or a pre-specified number of cycles (generations) is made without an improvement in the value of f_2 being registered.

The results of the test run with the Darwin GA engine are listed in Table 7.2 overleaf. Darwin was run on each of the 12 datasets used test Valletta. Darwin managed to correctly learn each of the kernel languages and found the target TS descriptions but took much longer. Due to the stochastic nature of genetic algorithms Darwin would often return very different times on the same dataset. The initial population is chosen randomly and, consequently, the running times varied greatly. Darwin would sometimes find the correct class description in the initial population, albeit very rarely, and sometimes it would takes many hours and many generations to find the same class description. The times shown in Table 7.2 are the averages taken over 5 separate runs. The 5^{th} column shows the average number (mean) of generations

(populations) that was generated by Darwin before the correct class description was found. The last column shows the number of different valid feature sets (TS descriptions) considered by Darwin. Darwin automatically discarded invalid feature sets, i.e. feature sets that were not substring-free[1]. Darwin still managed to find the correct class description after considering a relatively small number class descriptions, i.e. it considered a very small part of the total search space. Valletta, however, did much better. Valletta started the search from a single point in the search space and then used distance (the value of f_2) to direct the search. On the other hand, Darwin started the search in 256 different locations in the search space and then used genetic operators to modify the chromosomes and thereby broaden the search.

Problem	Valletta	St Nodes	Darwin	Generations	Chromosomes
bin01	13 secs	58	10 mins	18	2,648
bin01n	15 secs	64	10 mins	17	2,502
a301	23 mins	176	2 hrs	28	4,115
a302	51 secs	73	1 hr 34 mins	55	8,065
a302n	55 secs	67	1 hr 48 mins	55	8,103
a701	48 mins	163	2 hrs 19 mins	32	4,696
a702	1 hr 37 mins	179	43 hrs 55 mins	33	4,855
a703	11 mins	149	3 hrs 11 mins	29	4,243
a703n	12 mins	153	3 hrs 32 mins	29	4,271
a1101	6 mins	137	2 hrs 34 mins	23	3,374
a1101n	7 mins	163	2 hrs 53 mins	23	3,381

Table 7.2: A comparison of the results obtained for *Valletta* and *Darwin*.

[1]I.e. when a feature is a substring of another feature.

7.3 Testing with the *GSN* DataSets

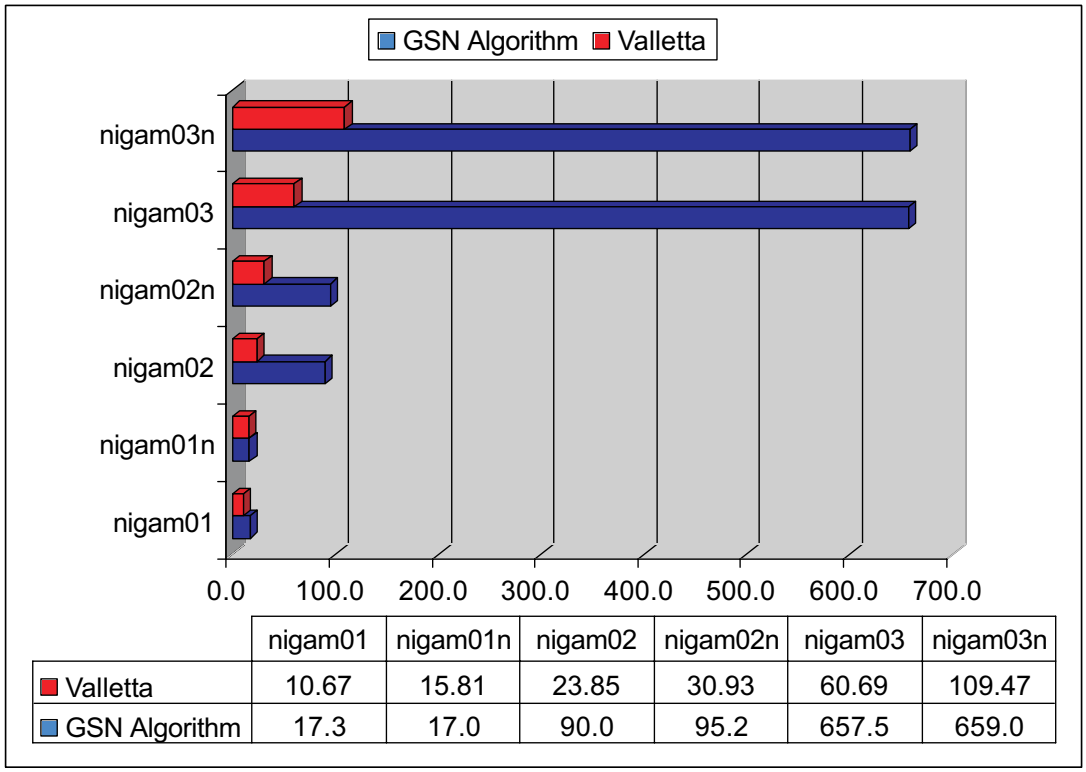
In addition to the datasets that were created purposely for Valletta, Valletta was also tested on the original GSN datasets as used by Nigam in his Masters thesis [92]. The GSN datasets comprise 6 training sets covering three confluent single-kernel languages. Each language has two training sets — one with noise and the

The GSN Datasets

Problem	$\|C^+\|$	$\|C^-\|$	*Description*
nigam01	5	4	Kernel: *cccc* Features: *ee, ded* Average Length: 2.5 chars Single kernel, Confluent, No noise. Average length of strings: 28 characters.
nigam01n	5	4	Kernel: *cccc* Features: *ee, ded* Average Length: 2.5 chars Single kernel, Confluent, $\approx$ 5% noise. Average length of strings: 28 characters.
nigam02	6	5	Kernel: *ccdbeebdccdbeebd* Features: *dd, ede, cdbebdc* Average Length: 4 chars Single kernel, Confluent, No noise. Average length of strings: 59 characters.
nigam02n	6	5	Kernel: *ccdbeebdccdbeebd* Features: *dd, ede, cdbebdc* Average Length: 4 chars Single kernel, Confluent, $\approx$ 10% noise. Average length of strings: 59 characters.
nigam03	8	5	Kernel: *bddaccaa* Features: *bddabcdadd, bddaaceeabcdd, eadededeabcd* Average length of features: 11.7 chars. Single kernel, Confluent, No noise. Average length of strings: 103 characters.
nigam03n	8	5	Kernel: *bddaccaa* Features: *bddabcdadd, bddaaceeabcdd, eadededeabcd* Average length of features: 11.7 chars. Single kernel, Confluent, $\approx$ 10% noise. Average length of strings: 103 characters.

Table 7.3: The GSN datasets used for *Valletta*/*GSN* comparison.

other with no noise. Table 7.3 lists the training sets and the properties of each of the languages. The number of training examples in all the training sets is very small - less than 13 strings in C^+ and C^- combined. All the languages are confluent. This means that no two features overlap, i.e. no prefix of any feature is a suffix of any other feature. Nigam probably chose only confluent sets of features because of the added complications of making the GLD function work with a non-confluent set of features. Learning confluent languages is easier, of course, since there is always one normal form for each string. Valletta easily learned all the languages in the



	nigam01	nigam01n	nigam02	nigam02n	nigam03	nigam03n
Valletta	10.67	15.81	23.85	30.93	60.69	109.47
GSN Algorithm	17.3	17.0	90.0	95.2	657.5	659.0

Figure 7.4: Comparing the running times of the Valletta and GSN algorithms.

GSN datasets, even those with noise, and took significantly less time than the GSN algorithm. It must be strongly emphasized that the comparison between the running times of the two algorithms must be taken in the proper context. The GSN algorithm was implemented in Modula-2 on a Sun Sparcstation-2 running the UNIX operating

system while Valletta was implemented in PowerBASIC®, a dialect of BASIC, and run on a PC running Microsoft Windows 2000®. The actual running times, therefore, are not comparable. It is evident from the graph in Figure 7.4, however, that Valletta is much less sensitive to the increase in the length of the training strings and the feature length. As pointed out in Chapter 4, we observed that the GSN algorithm is particularly sensitive to the length of the features. As was then explained, this is due to the feature-construction technique used by this algorithm. Valletta does not build the features but identifies all candidate features in a pre-processing stage. The search space of Valletta's learning stage is completely determined by the pre-processing stage. At each pass of the learning loop, the GSN algorithm constructs new features from the current set of features through left and right concatenation of the characters in the alphabet. With larger alphabets and large feature sets this process very quickly becomes very computationally intensive.

Another thing that must be pointed out is that, with the GSN datasets, Valletta was instructed to consider only single-kernel TS descriptions. This was easily done since Valletta is a variable-bias algorithm and the user can change the algorithm's inductive bias by modifying a number of input parameters. This was done because we knew that the target languages have only one kernel.

7.4 Learning in the Presence of Noise

Noise is present in most real-world pattern recognition and machine learning applications [16]. It was therefore one of the main objectives that Valletta should be able to learn in the presence in noise. In GI problems noise usually takes the form of spurious characters inserted in the strings. Three different types of noise were considered. Each type required a different technique to handle it. The three types of noise are:

(a) **Feature Noise** This consists of extra characters inserted *inside* the features

of a kernel language.

(b) **Kernel Noise** All other spurious characters are classified as kernel noise.

(c) **Misclassification Noise** This is the case when strings belonging to the target language are inserted in C^-.

Let us illustrate the difference between feature and kernel noise with the following example. Consider the kernel *aabaa*, the feature *bab* (shown in red), and the string $s = aabbabaa$. The string s contains one instance of the feature *bab* and contains no noise. Now consider also the following strings:

$$aabbcabaa$$

$$acabbabaa$$

$$aabbabcaa$$

The first string contains an extra character c (shown in blue) inserted *inside* the feature *bab*. This is feature noise. The second string contains an extra c inserted in the kernel and not in any feature. This is kernel noise. The third string contains an extra c inserted at the end of the feature *bab*. Since the extra character is not inside the feature this is also considered to be kernel noise. As explained in Chapter 5, Valletta deals with feature and kernel noise in different ways. Feature noise is detected and removed during the reduction of the strings in C^+ to their normal forms. This process is called *feature repair*. Kernel noise is not removed and remains in the normal forms. Kernel noise is detected and measured when Levensthein distance is computed between the normal forms. This process was also explained in Chapter 5. Valletta, therefore, not only distinguishes between the two types of noise but also handles them in different ways. Feature repair is a very important facility in Valletta since without this facility, if a feature has just one extra character, the whole corrupted feature has to be considered as noise. This affects the value of f_2 and increases the

learning time. In some cases it can make learning impossible or make the algorithm find an incorrect TS description. This was a problem with the GSN algorithm and is the most probable reason why it could not handle too much noise.

Another problem with the GSN algorithm was that it could not handle misclassification noise. If a string that belongs to the target language is inserted in C^- then the value of f_1, i.e. the minimum distance between C^+ and C^- would be zero and therefore, the function

$$f = \frac{f_1}{\epsilon + f_2}$$

would obviously never exceed the threshold since f_1 is zero. In Valletta this problem was overcome by using a new method for computing f_1. Instead of letting f_1 be the

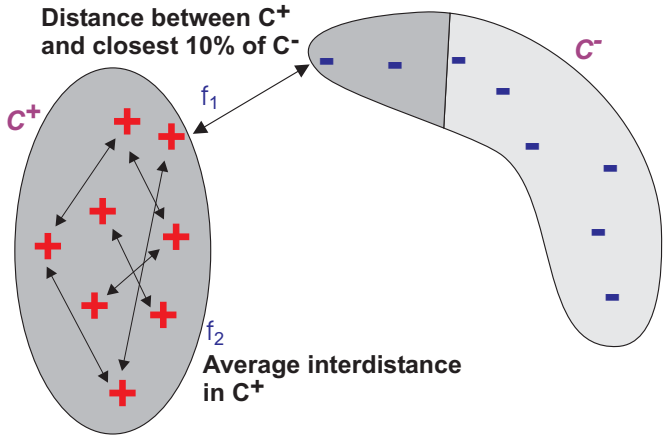


Figure 7.5: The new method for computing f_1 used for Valletta.

minimum distance, over all pairs, between C^+ and C^-, f_1 is defined to be the average distance between the strings in C^+ and the closest 10% of the strings in C^-. This is depicted in Figure 7.5 above. The percentage was chosen arbitrarily and can be changed if necessary. In the case when a small number of strings in the class C are added to C^- (i.e. misclassification noise), f_1 does not return zero. This new method worked extremely well on the Valletta datasets and was also successfully used for the *Monk's Problems* datasets discussed in the next section.

7.5 The Monk's Problems

In the summer of 1991 at the 2^{nd} European Summer School on Machine Learning, a number of researchers proposed a suite of simple machine learning problems they called the Monk's Problems[2]. The aim of the exercise was to test the leading machine learning algorithms on a common suite of three problems and to see which of the algorithms perform the best. The Monk's problems are set in an artificial robot domain. The robots in this domain are described by six different attributes [122].

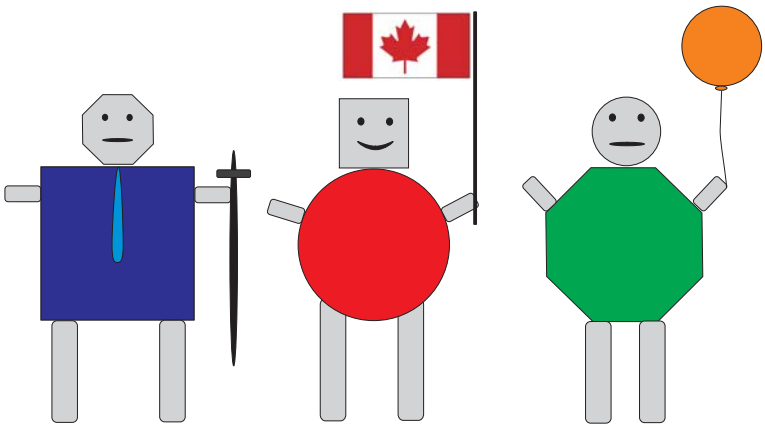


Figure 7.6: Some robots of the Monk's Problems.

x_1 : head_shape	$\in$	{round, square, octagon}
x_2: body_shape	$\in$	{round, square, octagon}
x_3 : is_smiling	$\in$	{yes, no}
x_4: holding	$\in$	{sword, balloon, flag}
x_5: jacket_colour	$\in$	{red, yellow, green, blue}
x_6: has_tie	$\in$	{yes, no}

There are three problems in the Monk's Problems. Each problem is a binary classification task, i.e. the learning algorithm must determine whether or not a robot belongs to a given class. Each problem consists of a logical description of the class and a training set that contains a proper subset of all the 432 possible robots in

[2]The school was held near the monastery of Corsendonk.

the domain. The learning task is then to generalize over these examples and, if the learning algorithm allows, to output a class description. The three classes are:

(a) **Monk1**:

(head_shape = body_shape) or (jacket_colour = red)

124 labelled examples were randomly selected from 432 possible robots. No misclassifications.

(b) **Monk2**:

exactly two of the six attributes have their *first* value

169 labelled examples were randomly selected from 432 possible robots. No misclassifications.

(c) **Monk3**:

(jacket_colour is green and holding sword) or

(jacket_colour is not blue and body_shape is not octagon)

122 labelled examples were randomly selected from 432 possible robots. 5% misclassifications.

Problem 3 is the only problem that contains noise and was included to test the performance of the algorithms in the presence of noise. Problem 2 is very similar to parity problems. Problems 1 and 3 are in DNF form and are therefore supposed to be solvable by symbolic learning algorithms such as ID3, AQ, etc. Problem 2 combines attributes in a way which makes it awkward to express in DNF or CNF.

Table 7.4, reproduced from [122] and shown overleaf, lists the results obtained by the different learning algorithms on the Monk's Problems datasets. As Thrun explains in [122], the experiments were performed by leading researchers, each of whom was an advocate of the algorithm he or she tested and, in many cases, the creator of the algorithm itself.

Learning Algorithm	**#1**	**#2**	**#3**
AQ17-DCI	**100%**	**100%**	94.2%
AQ17-HCI	**100%**	93.1%	**100%**
AQ17-FCLS		92.6%	97.2%
AQ17-NT			**100%**
AQ17-GA	**100%**	86.8%	**100%**
Assistant Professional	**100%**	81.3%	**100%**
mFoil	**100%**	69.2%	**100%**
ID5R	81.7%	61.8%	
IDL	97.2%	66.2%	
ID5R-Hat	90.3%	65.7%	
TDIDT	75.7%	66.7%	
ID3	98.6%	67.9%	94.4%
ID3, no windowing	83.2%	69.1%	95.6%
ID5R	79.7%	69.2%	95.2%
AQR	95.9%	79.7%	87.0%
CN2	**100%**	69.0%	89.1%
CLASSWEB 0.10	71.8%	64.8%	80.8%
CLASSWEB 0.15	65.7%	61.6%	85.4%
CLASSWEB 0.20	63.0%	57.2%	75.2%
PRISM	86.3%	72.7%	90.3%
ECOWEB leaf prediction	71.8%	67.4%	68.2%
ECOWEB l.p. & information utility	82.7%	71.3%	68.0%
Backpropagation	**100%**	**100%**	93.1%
Backpropagation with weight decay	**100%**	**100%**	97.2%
Cascade Correlation	**100%**	**100%**	97.2%

Table 7.4: The published results for the Monk's Problems.

The algorithms tested included the various decision tree learning algorithms such as ID3 and its variations, mFOIL — a rather interesting inductive learning system that learns Horn clauses using a beam search technique, and various neural networks such as Backpropagation and Cascade Correlation. No algorithm managed to correctly learn all three classes, although some came very close. In spite of the fact that the Monk's Problems are defined in a symbolic rather than a numeric domain, the best performing algorithms were, most surprisingly, the neural networks.

Testing Valletta on the Monk Datasets

Very early in Valletta's development we were looking for real-world kernel languages that could be used to test Valletta and to demonstrate it's usefulness. When we looked at the Monk's problems we discovered that the Monk's problems can quite easily be posed as GI[3] problems. In particular, each of the Monk's three classes of robots can be represented by a kernel language.

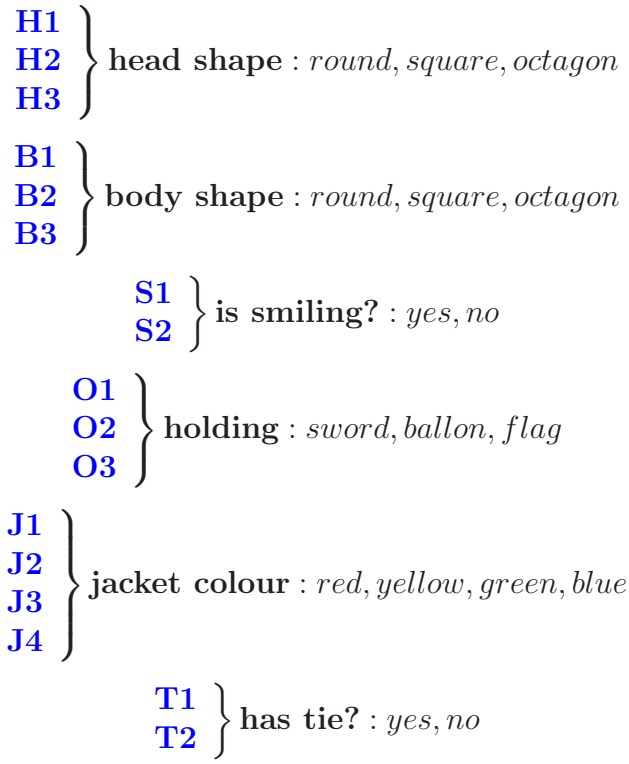


Figure 7.7: The Alphabet used to encode the Monk datasets.

Consider the encoding of the attribute values as shown in Figure 7.7 above. The alphabet used is $\Sigma = \{H, B, S, O, J, T, 1, 2, 3, 4\}$. Each attribute value is assigned a unique *tag* over Σ. For example the value *head_shape = square* is assigned the tag *H2*. The string **H2B3S1O2J3T2** represents a robot that has a square head, an

[3]Every learning problem could, in theory, be reduced to a Grammatical Inference problem

octagonal body, is smiling, holds a balloon, wears a yellow jacket, and has no tie. The class of robots in Problem 1 can be described by the following kernel language:

Kernels: **H1B1**, **H2B2**, **H3B3**, **H1B1J1**, **H2B2J1**, **H3B3J1**, **H1B2J1**, **H1B3J1**, **H2B1J1**, **H2B3J1**, **H3B1J1**, **H3B2J1**.

Features: **H1**, **H2**, **H3**, **B1**, **B2**, **B3**, **S1**, **S2**, **O1**, **O2**, **O3**, **J1**, **J2**, **J3**, **J4**, **T1**, **T2**.

The features H1, H2, H3, B1, B2, B3, and J1 are assigned a zero weight while the other features are assigned a non-zero weight. Normal Levensthein distance is then used. Each tag is, in essence, indivisible and is considered to be a single 'character'. It is easy to see that the kernels capture the description of the class. The values of the discriminating attributes of the class are all in the set of kernels. Each of the Monk's problems can be described by a confluent kernel language of this type. The above description is, admittedly, somewhat cumbersome but our aim was to demonstrate the versatility of kernel languages and also that of Valletta.

The Monk datasets were re-encoded as described above and used by Valletta to learn each of the Monk's problems. Valletta did manage to learn Problems 1 and 2 but took a very long time to do so — an average of 8 hours. Valletta could not learn Problem 3 with 100% accuracy and an analysis of this revealed that this was due to the fact the training set for Problem 3 was not structurally complete. This was because one of the kernels was not in the training sets. When a string containing the missing kernel was added to the dataset of Problem 3, Valletta successfully found the correct class description. Most of the algorithms tested on the Monk's problems took only a few minutes to learn and it was therefore decided to investigate why Valletta took so long.

It turned out that Valletta took such a long time to learn the Monk's problems because it has a much broader bias than was necessary for the Monk's problems. In

other words, it was considering a search space of TS descriptions that was very much larger than the space of possible TS descriptions that were consistent with the Monk datasets. Valletta did not know that each feature had to be exactly 2 characters or that the target kernel language was confluent. It considered TS descriptions that have features with 4 characters and more. This made Valletta consider thousands of invalid TS descriptions before the correct description was found. This is very often a problem with many learning algorithms and an issue that is discussed later on in the chapter. It is important that the inductive bias of the algorithm is not broader, i.e. larger, than necessary. In learning it is paramount that the learning algorithm designer or practitioner makes use of all the information available to him or her. The author therefore decided to design and implement a version of Valletta that learned only the subclass of kernel languages that can be used to describe problems such as those in the Monk's problems. It must be emphasized that was, by no means, cheating in any way. The new algorithm, which was called *Mdina*[4], after Malta's ancient capital, learns *trivial* (see Chapter 3) kernel languages, i.e. those where each rewrite rule is of the form $a \leftrightarrow \varepsilon$ for some character $a \in \Sigma$. Mdina has a much broader bias than that required for the Monk's problems but it bias is much smaller than Valletta's. Mdina can be used to learn all trivial languages and not just those that are kernel language encodings of problems such as the Monk's problems. Mdina is much faster than Valletta for a number of reasons. In trivial kernel languages each feature is a single character. Given a set of features F, $aleph(F)$, i.e. the characters used in F, cannot be used in the kernels. This is because, by definition, a kernel cannot contain a feature as a substring. It is therefore clear that, given a set of features F and a set of kernels K, then;

$$aleph(K) = \Sigma - aleph(F)$$

[4]Pronounced **Im-dina**.

In other words, the alphabet Σ is partitioned. One partition being the characters used for the features and its complement being the characters used in the kernels. The implication of this is that learning trivial kernel languages reduces to the problem of finding the correct partitioning of Σ that achieves class separation. The problem, of course, is how to do this efficiently. We abandoned the *tag* scheme used to test Valletta on the Monk datasets and re-encoded the Monk datasets using the alphabet shown in Figure 7.8 below.

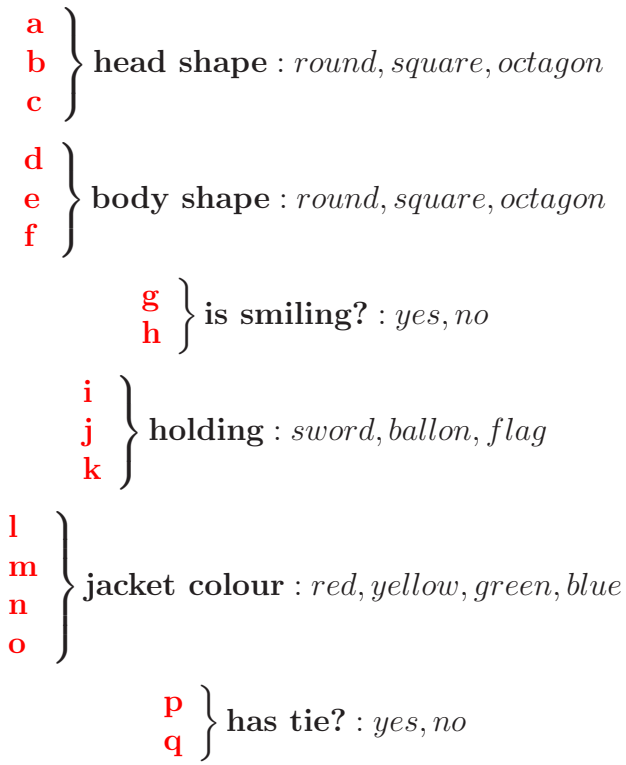


Figure 7.8: The alphabet used by MDINA.

Mdina is different from Valletta in a number of key areas:

(a) Mdina uses a very simple procedure for computing normal forms — it simply deletes the characters in $aleph(F)$ from the strings in C^+ and C^-. This is because the features are all single-character and the characters used in the set of features cannot occur in the kernels. This makes string reduction very fast.

(b) As a consequence of the above, Mdina does not need to build or construct features (like the GSN algorithm) or even search for valid feature sets (like Valletta). It simply considers different binary partitions of Σ.

(c) In order to handle noise properly, Mdina uses the following definition of f_1;

$\mathbf{f_1}$ is the average distance between C^+ and the closest 10% of the strings in C^-.

This is necessary to handle misclassification noise (See Section 7.4).

Mdina learned all the Monk's problems in a very short time. Problem 1 took just 8.6 seconds to solve. Problem 2 took 6.1 seconds and Problem 3 took 6.7 seconds to solve. For problem 3 we added one string, *aegilp*, to the training set in order to make the training set structurally complete. In other words, after the addition of *aegilp*, the training set of Problem 3 contained all the kernels.

Kernels discovered by Mdina		
Problem #1	Problem #2	Problem #3
ad	ad	dil
adl	ag	dim
ael	ai	din
afl	al	dl
bdl	ap	dm
be	dg	dn
bel	di	eil
bfl	dl	eim
cdl	gl	ein
cel	gp	el
cf	il	em
cfl	ip	en
	lp	fi
		fin

Table 7.5: The kernels discovered by the Mdina algorithm.

Table 7.5 lists all the kernels discovered by Mdina. In each case Mdina very quickly discovered the correct partitioning of Σ that gave class separation. In each case the

kernels discovered by Mdina were the kernels in the TS description of each of the Monk classes.

7.5.1 A Discussion of the Results

Mdina is a grammatical inference algorithm based on the ETS model that learns trivial kernel languages. Mdina uses distance as a stopping criterion as well as to direct the search process. Mdina was not designed for the Monk's problems and did not incorporate any knowledge about the Monk datasets. The reason it managed to learn all of the Monk's problems was because each class in the Monk's problem can be represented by a kernel language. This serves to demonstrate that kernel languages are quite versatile. This thesis contains other examples of 'real-world' kernel languages. It is quite possible that many classes in CNF or DNF form can be represented by kernel languages and, to the best of the author's knowledge and belief, Mdina is the first application of a GI algorithm to these types of problems.

The most important result, in the author's opinion, is Mdina's performance on Problem 3. Careful choice of the f_1 function allowed Mdina to easily learn the problem in spite of the 5% misclassification noise. The method used for computing f_1 was not chosen specifically for the Monk's problems but rather for all learning problems that include misclassification noise. Mdina does not have a variable inductive preference bias and *exactly the same preference bias* was used for all the three problems. This result served to highlight the elegant and economical way noisy classes are handled in the ETS Model. The features capture the regularity in the language and the 'noise' is handled by the distance function.

The technical report that describes the Monk's problems and the results obtained by the various algorithms [122] does not attempt to analyse or explain the results. The author feels the whole exercise served more to determine whether each algorithm had the correct inductive bias to learn each of the Monk's problem than to determine the actual learning ability of the various algorithms. Each of the algorithms listed in the report are widely used and have undoubtedly successfully been used for other learning tasks. The author thinks that the apparent inability of some of the algorithms to learn the Monk's problems is due more to their type of inductive bias rather than to anything else. The results also served to highlight the usefulness of kernel languages and also the versatility, flexibility, and power of the ETS model and the idea of using distance to define classes and to direct the search. A question that begs to be asked is: *Does Mdina have exactly the right inductive bias for the Monk's problems*? In order to answer this question the author wrote a program that considers all possible partitions of the alphabet Σ and that count the number of partitions of Σ that allow f to exceed the threshold. In other words, the program performs a brute-force search for all the TS descriptions of kernel languages that are consistent with each of the Monk datasets. It turned out that each dataset has around 4 to 5 TS descriptions consistent with it. The reason that Mdina found the correct TS description was because it gave preference to TS descriptions that have the least number of characters in the kernels. Mdina, therefore, employs a form of Occam's Razor as its inductive bias. It gives preference to TS descriptions with a fewer number of characters in the kernels.

7.6 Comparison with the Price EDSM Algorithm

The Evidence-Driven State Merging (EDSM) algorithm developed by Rodney Price [73] is currently recognized to be very much the state-of-the-art in DFA learning algorithms. The EDSM algorithm was the co-winner of the Abbadingo DFA learning competition (see Appendix E). The competition involved the learning of randomly generated DFAs with up to 512 states from a finite set of labelled training strings. The EDSM algorithm is an evidence-driven state-merging algorithm as described in Section 2.7 of Chapter 2. State-merging DFA learning algorithms work by first building a prefix-tree acceptor from the training set and then merging compatible states to create cycles and thus achieve generalization [127]. The author downloaded the EDSM code, recompiled it for the Windows 2000 platform, and conducted some experiments on the data sets used to test Valletta. This was done in order to compare the EDSM algorithm and Valletta in the areas of efficiency, class representation, robustness, and inductive bias. It must be emphasized that, as Bunke (and many others) pointed out, many GI algorithms are designed for specific domains and it is therefore often difficult, if not impossible, to compare the performance of two given algorithms. Strictly speaking, Valletta and EDSM are not really comparable. They are targeted at different domains and have different inductive biases. However, the author wanted to see how Valletta compares with a leading GI algorithm and also to see if any lessons can be learned. The Price EDSM algorithm was an ideal candidate. It won the Abbadingo DFA competition and is considered by many in the GI community to currently be the leading DFA learning algorithm. Valletta does not learn DFAs but kernel languages are regular so the author thought it would be interesting to compare the two algorithms on the same datasets. It is the author's belief that most DFA learning algorithms have too broad a bias. This issue is discussed in detail later on in Section 7.8 where the results obtained are analysed.

To get a feel of the EDSM algorithm the author first ran it on some randomly generated datasets of a very simple regular language – $0\{1\}^*$. Numerous small training sets of around 5 positive and 5 negative examples were tried. It was noticed that EDSM was very sensitive to the choice of the strings in C^-.

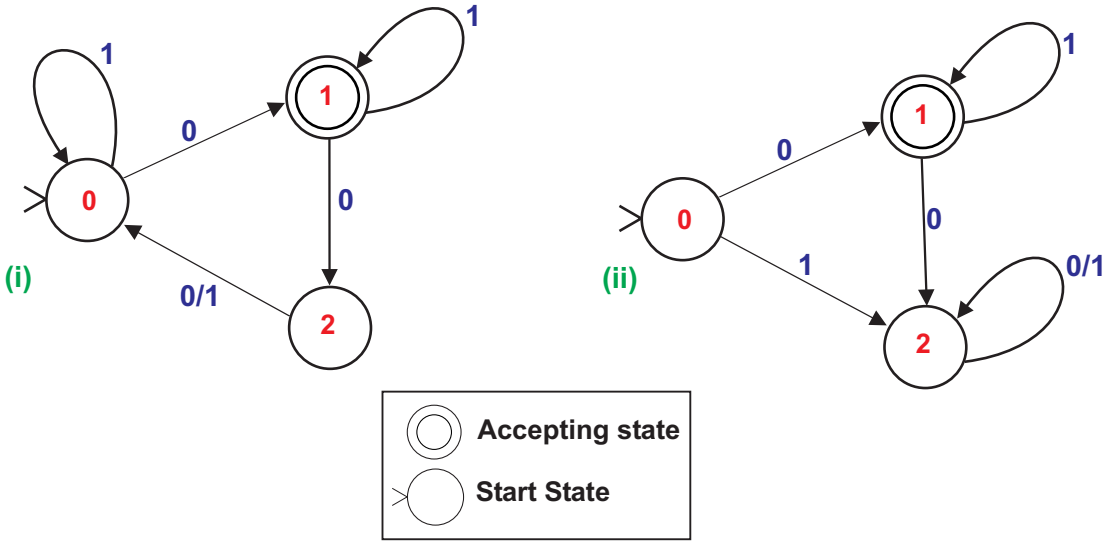


Figure 7.9: The DFAs produced by the EDSM algorithm for different $0\{1\}^*$ datasets.

Figure 7.9(i) shows the DFA output by EDSM from the following set of labelled strings[5].

A 0
A 01
A 011
A 0111
A 01111
R 010
R 0101
R 0110
R 00
R 0001

The DFA produced by EDSM from the above training set accepts the string *101*

[5] **A** denotes an *Accepted* string and **R** denoted a *Rejected* string

which is not in the language. New strings were then added, one at a time, to C^- and the DFA output by the EDSM algorithm was recorded. New strings were added to C^- until the target DFA was obtained. The strings added in C^- were carefully chosen so as to exclude the acceptance of strings known not to belong to the language. The string *101*, for example, is accepted by the DFA shown in Figure 7.9(i) when, clearly, this string does not belong to $0\{1\}^*$. This string was therefore added to C^-. EDSM finally produced the correct target DFA with the following training set:

A 011
A 0
A 0111
A 01111
A 0111111
A 011111111
A 01
R 0101
R 0110
R 11
R 1011
R 1010
R 000
R 0100
R 0001
R 01001
R 110
R 100
R 00

The results of these experiments strongly suggested that, in general, EDSM tends to under-generalize rather than over-generalize. This is not-surprising. EDSM was designed the class of regular languages. Its learning domain is therefore very large. The problem with such learning algorithms is that the training sets have to be quite large. This is not a criticism — simply an observation. This happens because a small training set tends to have many class descriptions (in this case DFAs) that

are consistent with the training examples. The algorithm therefore require large training sets in order to exclude the large number of incorrect, but very similar, class descriptions.

Since kernel languages are regular the author also tried some of the Valletta training sets on EDSM. Figure 7.10 shows the DFA produced by EDSM on the *bin01* training set. The strings in the training set are listed in Appendix C. It is quite obvious that EDSM did not output the correct DFA. This is because EDSM treated

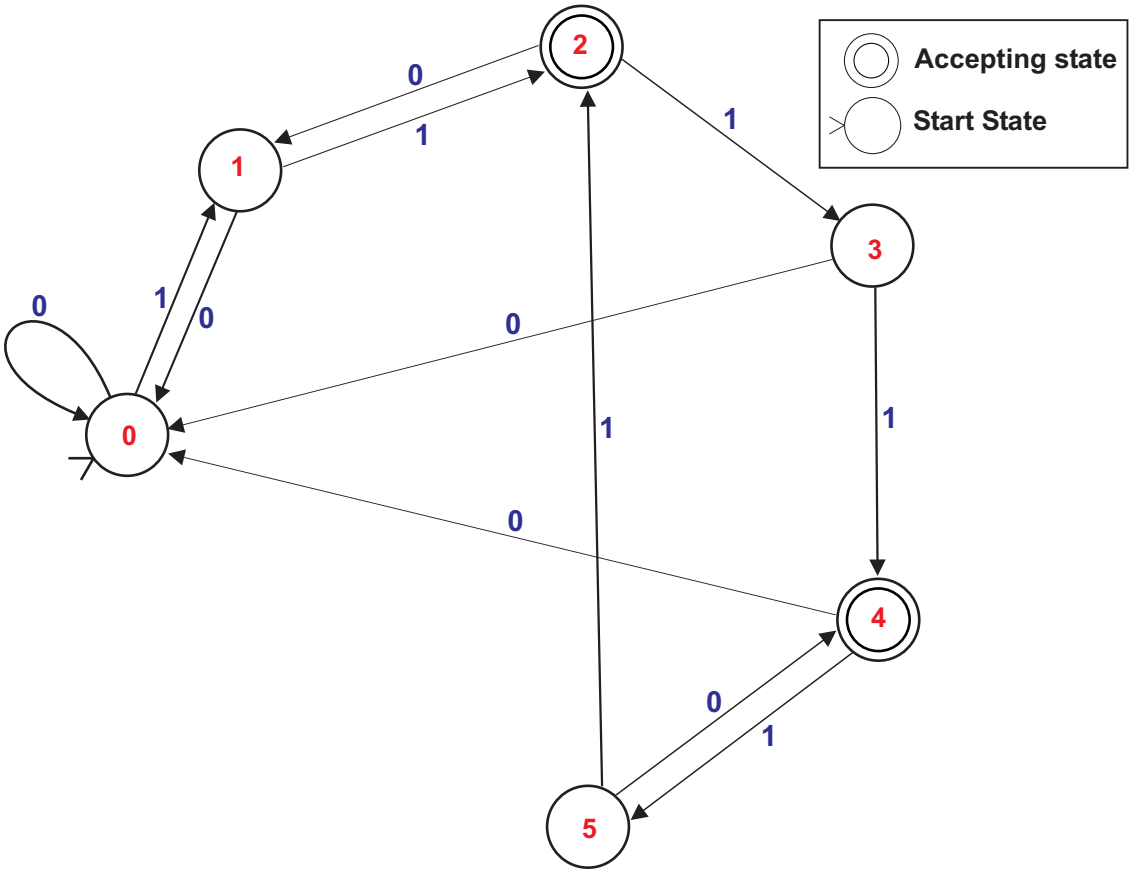


Figure 7.10: The DFA produced by the EDSM algorithm for the *bin01* dataset.

the class as a normal regular language. EDSM, unlike Valletta, does not implicitly incorporate information about kernel languages. The number of regular languages consistent with the training examples is very much larger than the number of kernel

languages consistent with the same set. For this reason EDSM would probably require much more training examples. The author also designed a small training set for a very simple kernel language, *kernel01*. The DFA produced by EDSM is shown in Figure 7.11.

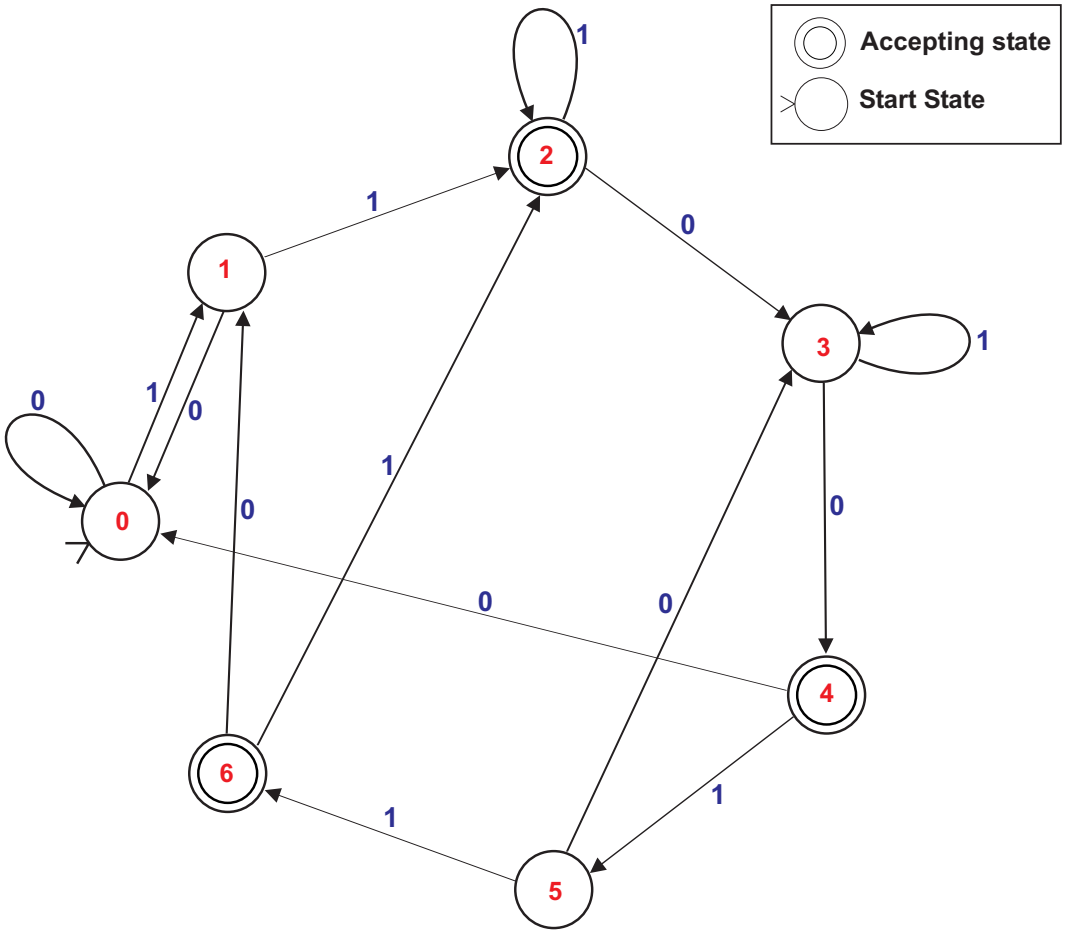


Figure 7.11: The DFA produced by the EDSM algorithm for the *kernel01* dataset.

Table 7.6, overleaf, lists the strings in the *kernel01* training set. Given the previously observed sensitivity of the EDSM algorithm to the strings in C^- we manually built C^-, which contains 6 strings, from strings that are, in general, longer than those in C^+. Valletta easily learned the correct class description even from such a small training set. The DFA produced by the EDSM algorithm was not the target DFA. It

C^+	C^-
00101010	100011010
1000111	10101010101
1010011	0000
1100101	00100000111
10001010	1010
	10111110

Table 7.6: The *kernel01* training set.

was not even similar to the target DFA. It accepts strings not in the kernel language. Many more new strings were added to C^- but the author could not manage convince EDSM to output the correct DFA. This is probably because the training set was used was structurally complete for Valletta but not for the EDSM algorithm. This issue is discussed later on in this chapter.

The EDSM experiments highlighted the importance of using a learning algorithm that is targeted at the right learning domain and that has the correct inductive bias (language bias and preference bias, see Chapter 2). Kernel languages are regular and are therefore, in theory, learnable by the EDSM algorithm. However, since the EDSM has such a large learning domain[6], it very likely requires very large training sets. The beauty of the ETS model is that the practitioner can incorporate the inductive bias he or she deems fit for the domain. There is no such thing as a universal learning algorithm — at least so far. Further analysis and a discussion of the results obtained is contained in Section 7.8.

[6]I.e the space of all hypotheses learnable by the algorithm.

7.7 Representation and Bias

It has been emphasized many times in this thesis that the ETS Model is a learning model and not a learning algorithm. The ETS Model does not impose a fixed *language bias* or *preference bias* on a learning problem. The language bias of a learning algorithm is the choice of representation of the domain of discourse, i.e. the numeric or symbolic encoding of the objects. The *inductive preference bias* of a learning algorithm is the preference of certain hypothesis over others. Let us illustrate both notions with an example. Suppose we want to learning a given regular language from a finite set of training examples. We can use a GI algorithm that accepts the strings as input or we can re-encode the strings as vectors and use an Artificial Neural Network (ANN). Both algorithms, supposedly, are learning the same class. Each, however, has a different language bias. Is there anything to be gained (or lost) by re-encoding the strings as vectors? The GI algorithm can try to find a class description consistent[7] with the training set in a number of different ways. It could search the space of all DFAs, or the space of all left-linear grammars, or even perhaps the space of all regular expressions. All are class descriptions of regular languages. Suppose we choose DFAs. We could then search for the DFA with the minimum number of states that is consistent with the training examples or perhaps the DFA with least number of cycles. This is called the inductive preference bias of the algorithm. Since, in general, there may be many class descriptions consistent with the training examples, a change of the inductive preference bias of the algorithm may result in a different class description being found. One of the main advantages (and, in the author's opinion, one of the most appealing features) of the ETS Model is that the user is not restricted to any particular language or preference bias. As explained in Chapter 1,

[7] I.e. a class description with accepts the positive training examples and rejects the negative training examples.

the ETS Model does insist that the domain of discourse is encoded as *structs*, which have a certain compositional structure, but this still allows a lot of flexibility for the user. The reader is referred to the recent technical report completed by the author's colleagues in the Machine Learning Group at the University of New Brunswick for an exposition [54]. With ETS, we can use strings, trees, graphs, matrices, etc. ETS also does not impose a preference bias on the designer of the learning algorithm. In this Section we shall attempt to argue the importance of not being restricted to a particular form of representation (language bias) and to a fixed preference bias. We shall also argue that the choice of representation affects the choice of inductive bias. In fact, a change of representation changes the learning problem itself. We also present arguments that support the claim that a change of representation can make a learning problem much harder unless we happen to know the correct inductive bias appropriate for that particular representation.

7.7.1 What is Representation?

The issue of representation deals with the encoding, into some mathematical structure, of the domain of discourse of some learning problem. We cannot over-emphasize the fundamental, and very important, distinction between the objects themselves in some domain of discourse and their representation, i.e. the numeric or symbolic encoding of the objects (as strings, numbers, vectors, graphs, etc.). Consider, for example, the set of all humans. A human can be encoded, or represented, in a number of ways;

(a) **Bitmap encoding**, i.e. a picture (binary image) of a human,

(b) **Genome**, a string containing the DNA sequence of a human,

(c) **2-D vector**, e.g. (weight, height), and

(d) **Attribute n-tuple**, e.g. race, complexion, colour of hair, etc.

Depending of what classes we want to consider in a given domain of discourse, an encoding may or may not be appropriate for the purpose of class description. It was shown in [108], for instance, that a 2-D vector (weight, height) is, in general, sufficient to distinguish the class of male humans from the class of female humans. In fact, when the vectors are plotted in the Cartesian plane it is easy to see that the two classes form two recognizable clusters. It is also conceivable that one can distinguish between males and females from bitmap representations. If, on the other hand, we want to consider the class of humans who have *minor thalassaemia*, a completely asymptomatic congenital condition of the blood, then such a 2-D vector will not do. Neither would a bitmap. The genome representation of the human would, in this case, be required. This is because people with minor thalassaemia are not externally distinguishable but their DNA contains the thalassaemia gene.

A representation may be better than another because it 'stores more information' relevant to the class in question. It has been argued that there is no form of representation that can 'store all the information' about a given object in the domain of discourse. The closest thing to such a perfect form of representation must surely be the 'transporter' device in the *Startrek* TV series. The transporter device can, allegedly, 'dissolve' a human into the constituent atoms, encode the human is some form, transfer this information to another location, and reconstruct the exact human, from different atoms, complete with identical DNA, thoughts, feelings, emotions, memory, experiences, etc.

Another argument why one particular representation may be better than another is that the regularities of a particular class might not be 'visible' under some representations [21]. We illustrate with an example.

String	Gödel Number
ab	4
aabb	900
aaabbb	889,350
aaaabbbb	8,687,348,670

Table 7.7: Some strings from $a^n b^n$ and their Gödel Numbers.

Table 7.7 shows, in the first column, four strings from the language $a^n b^n$. It should be easy to anyone with a basic knowledge of formal languages to guess from which language the strings are drawn. The second column shows the Gödel number of the same strings. All we have done is to re-encode the strings as natural numbers. We have used the Gödel number encoding of the strings. This mapping is deterministic and computable in polynomial time. The regularity that was previously visible in the strings now 'seems' to have disappeared. It is clearly more difficult for humans to 'guess' the correct class if the encoding is not 'right'. But why? Are learning algorithms also sensitive to the 'appropriate' choice of representation?
As Clark and Thornton explain in [21]:

> Some regularities enjoy only an attenuated existence in a body of training data. These are regularities whose statistical visibility depends on some systematic re-coding of the data. The space of possible re-coding is, however, infinitely large — it is the space of applicable Turing machines.

Many learning algorithms accept only one form of representation of the input data. This means that the practitioner must always use the same encoding (representation) of the domain of discourse. It is well known that artificial neural networks (ANNs) accept only vectors as input. The questions we ask is: *Does the choice of representation affect learning*? In other words, are learning algorithms sensitive to

re-encodings of the domain of discourse and do these re-encodings affect the learning process in general?

7.7.2 Is Representation Important?

In the early days of A.I. (i.e. 1960's to 1980's) most researchers agreed that representation was an important issue in cognitive science. Recently this assumption has been questioned. As Thornton explains in [119],

> Now there is growing disagreement over the relevance of representation and a steadily deepening polarization of views with respect to the necessity of employing it in cognitive machinery.

An argument very often used by those who do not see representation as a crucial issue is that if a class in a given domain of discourse can be learned in polynomial time under one encoding then it should be learnable in polynomial time in another encoding as long as the mapping from one encoding is deterministic and is computable in polynomial time. This argument, *prima facie*, makes sense. Suppose we have a GI algorithm that learns the language $a^n b^n$ from a finite number of examples in time polynomial in the size of the training set. If we can re-encode the strings as natural numbers using, for instance, the Gödel encoding of the strings, then there must be a polynomial algorithm to learn the same class. Before we present our objections and reservations to this argument we want to first clarify a number of points.

(a) It is always assumed that, under any reasonable encoding, a class is always *computable*. If, for example, our domain of discourse is the set of all animals and we want to consider the class of cats, then if we encode the animals as strings, the set of strings corresponding to cats must be a computable language. This assumption is fundamental since if a class is not computable (under some encoding) then it cannot have a finite description.

(b) If a class is computable and has a finite description under some encoding E then it is still computable and has a finite description under encoding E' as long as there is a deterministic and computable mapping from E to E'. If the language $a^n b^n$ is a computable language and has a finite description then the set of natural numbers which are Gödel encodings of the strings of this language must also be a computable set with a finite description. The finite description of $a^n b^n$ is a formal grammar while the description of the associated set of Gödel numbers is a set of μ-recursive functions. It is well known from the Theory of Computation that graphs can be encoded as strings and string as numbers. In fact, for every computable language, the set of Gödel number encodings of the strings in the language is itself a computable set [115].

The argument of those who do not see representation as an issue is that any representation can be used as long as the encodings form a computable set. This argument is used mostly by those in the connectionist community. This is probably because artificial neural networks, ANNs, only accept vectors as input. Although the above is, in theory, true we believe that researchers in the connectionist and machine learning communities who claim that representation is not an important issue are missing some important points.

We are not claiming that classes become 'unlearnable' if the representation is changed. What we are claiming is that the learning problem *changes if the representation changes*. Let us consider again the language $a^n b^n$ and the set of strings drawn from this language shown in Table 7.7. It is conceivable that one can design a simple GI algorithm that learns languages of this type. Now what if one is given only the Gödel encodings of this language? Does learning become harder? We claim that these are two different learning problems. If one encodes the strings as natural numbers, the class then changes too. In other words, the specification of the class is tied to the

form of representation. The set of natural numbers that are Gödel number encodings of the strings in $a^n b^n$ is no longer the class $a^n b^n$. This set of numbers is computable and does have a finite description (a set of μ-recursive functions) but might require a completely different learning algorithm with a different inductive bias and a different search strategy. This goes some way towards explaining why neural networks perform so badly in certain applications. It has been claimed that a neural network (of the appropriate size) can compute a computable function in the Euclidean space $\mathbb{R}^n$. In other words, given an ANN of the appropriate size, a set of weights exists that computes the function. Even if this is true the problems remain. When one chooses a particular architecture and size of a neural network one is putting an implicit bound on the size of the class description. Suppose, for example, that we have an ANN with n weights to learn a class C. We are then implicitly assuming that C can be described (or parameterized) with n real numbers. This issue has recently received attention from the connectionist community [75]. To demonstrate this we conducted a number of experiments with ANNs. We tried to train a Backpropagation neural network with the complete datasets for 8, 10, 12, and 16-bit even parity mapping. By *complete* we mean that for each n-bit parity we labelled each of the 2^n strings with a T (True) or F (False). The reason we tried complete datasets was to find the optimal network size, i.e. number of weights, for each of the mappings. For the 8-bit parity dataset we first tried an $8 \times 8 \times 2$ network. This did not converge after running for many hours. We then tried changing the various learning parameters but to no avail. We increased the number of hidden neurons to 16 and then to 32 and got very much the same results. We finally tried a network with 64 hidden neurons and the network converged within a few seconds. This network had $(8 \times 64) + (64 \times 2) = 640$ weights. The ANN for 10-bit parity converged in just over 3 mins and required 256 hidden neurons while the ANN for 12-Bit parity required 1024 hidden neurons and

converged in just under 12 minutes. We couldn't use Backpropagation to learn the 16-bit parity problem since the ANN software we used, Brainmaker®, was limited to 1024 hidden neurons. We then used a Cascade Correlation neural network to learn the 16-bit even parity mapping. The Cascade Correlation network changes its architecture during learning. We then tried to train the networks on incomplete data, sometimes with datasets that contained up to 85% of the 2^n possible strings. The networks never generalized correctly from incomplete data. This phenomenon has been observed many times [120]. We were not surprised at the neural network's inability to learn from incomplete parity data. There is absolutely no reason why a learning algorithm should generalize correctly from incomplete parity data *unless the algorithm has the right inductive bias*. This point is, most unfortunately, not always well understood. For any given training set of incomplete parity data, there might be thousands of mappings that are consistent with the training set. Why should then the neural network converge to the parity mapping? It will only do so if it has the correct inductive preference bias. Our experiments with the complete parity datasets demonstrated that a Backpropagation neural network can, in fact, represent the parity mapping. In other words, given an ANN of appropriate size, there exist a set of weights that can describe (or parameterize) the parity mapping. Generalization from incomplete training sets is a totally different issue. Neural networks do not learn the parity mapping precisely because they do not have the correct bias. Moreover, the inductive bias of a neural network is fixed and can only be changed slightly by modifying some learning parameters. It is therefore unlikely that anyone can convince a Backpropagation ANN to generalize correctly from incomplete examples. Some researchers, including Thornton [120], have proposed various reasons as to why this happens. Valletta, and for that matter all ETS learning algorithms, do not have a fixed inductive preference bias. The ETS model provides a general framework for

learning. The designer of the learning algorithm then incorporates the appropriate preference bias for the domain.

It turns out that, for any given n, the set of binary strings that are odd or even parity are a kernel language. For example, 4-bit even parity can be described by the following TS description:

Kernels: ε, 11, and 1111.

Transformations: $0 \rightarrow \varepsilon$ (Weight 0.0) and $1 \rightarrow \varepsilon$ (Weight 1.0)

We ran Valletta on a number of incomplete 16-bit parity datasets – *par01* had no noise and *par02* had 2% misclassification noise. In each case Valletta found the correct TS description is less than 3 minutes on a training set of several thousand strings. The training examples were the same that were used, unsuccessfully, to train the ANNs.

par01 16-bit parity-problem training set (even parity) used to train Back-propagation neural network. Binary alphabet, very large training set ($\approx 10,000$), multiple kernels, no noise, confluent.
Kernels: *11*, *1111*, *111111*, *11111111*, *1111111111*, *111111111111*, *1111111111111*, and *111111111111111*.
Features: *0*, and *1*.

par02 16-bit parity-problem training set (even parity) used to train Back-propagation neural network. Binary alphabet, large training set ($\approx 8,000$), multiple kernels, 2% misclassification noise, confluent.
Kernels: *11*, *1111*, *111111*, *11111111*, *1111111111*, *111111111111*, *1111111111111*, and *111111111111111*.
Features: *0*, and *1*.

Valletta, can learn the parity mapping even with very small, structurally complete datasets of less than 20 strings. We must emphasize that is because Valletta has the right inductive bias. When Valletta learns, it first considers the 'simple' TS descriptions, i.e. the TS description with the least number of features. It so happens

that the parity problem has a very simple TS description and this was very close to the Valletta's starting point in the search space. This is depicted in Figure 7.12 below. It might turn out that under some other representation, the target class description

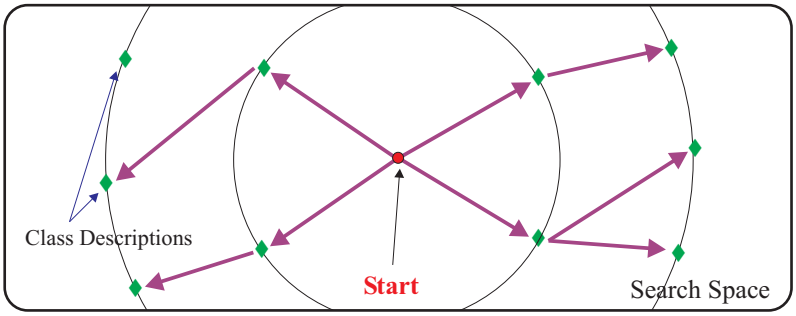


Figure 7.12: Enumerating the search space.

will be 'far away' from the starting point in the search space. The phenomenon was evident when the author was considering ETS learning of chain-code picture languages [2]. A picture such as a rectangle could be represented by its string contour encoding. It this case, the language of all strings that are contour encodings of rectangles (or any other figure) would be a context-sensitive language — sometimes with many productions. A GI learning algorithm that used Occam's bias would have a hard time learning these classes. This because the algorithm would first consider grammars with small number of productions before grammar with a large number of productions. The target grammar would therefore be 'far away' from the starting point in the search space. The same class of figures encoded as graphs could be represented by a very simple graph grammar. This would arguably be much closer to the starting point of the search if one used a learning algorithm with Occam's bias to search the space of graph grammars. We also strongly feel that the 'correct' inductive bias for a given learning problem depends on the choice of representation. If the representation is changed then a new, completely different bias might be required. Most learning algorithms search through the space of class

descriptions by first considering the simple class descriptions and progressing to more complex class descriptions. A DFA learning algorithm might first consider DFAs with 1 state, then DFAs with 2 states, and so on. In other words, the algorithm exploits an ordering of the space of class descriptions (i.e the search space). This bias is another example of Occam's bias. In essence, a learning algorithm's inductive preference bias specifies the method of enumerating the space of class descriptions. The algorithm stops when it finds a class description consistent with the training set. If one changes the method of enumerating the space, a different class description may be found. This is because, in general, the search space may contain several class descriptions consistent with the training set. In the case of parity problem, a neural network finds the first set of weights consistent with the set of incomplete parity data. It so happens that the mapping found by the network is not the correct (i.e. parity) mapping. This means that the network does not have the 'right' bias for the parity problem.

Wolpert [139] and many others have shown that no inductive bias can achieve a higher generalization accuracy than any other bias when considered over all classes in a given domain. In spite of this, it has been documented that certain bias do perform better than average on many real-world problems [118]. This strongly suggests that many real-world problems are homogenous in nature in that they require very similar inductive biases. This explains why certain learning algorithms such as ID3 do well on most applications. When learning algorithms do badly it is very often a case of incorrect inductive bias. This is why we designed Valletta to have a variable inductive preference bias. The user can, to a certain extent, change the algorithm's bias by modifying a number of parameters.

7.8 Analysis of the Results

Valletta's Overall Performance

The results of the tests carried out on Valletta using the purposely designed datasets showed that Valletta, in one particular case (the *a702* dataset), took just over 1.5 hours to find the correct class description. This is by no means catastrophic since artificial neural networks often require many hours of training time. In order to find out more why Valletta sometimes takes a few seconds to train and sometimes takes more than one hour, The author used a code profiler[8] to determine which sections of the code were taking up the running time. The results are shown in Figures 7.13, 7.14, and 7.15. The *a70x* datasets was chosen since the training examples of this dataset were drawn from the same kernel language. The dataset *a702* has a large number (≈ 128) of relatively long (average length 145 characters) in C^+ while, *a703* has 24 strings of average length (≈ 80 characters). The *a701* dataset has 41 strings of average length (≈ 87 characters). All the datasets have exactly the same C^- training set.

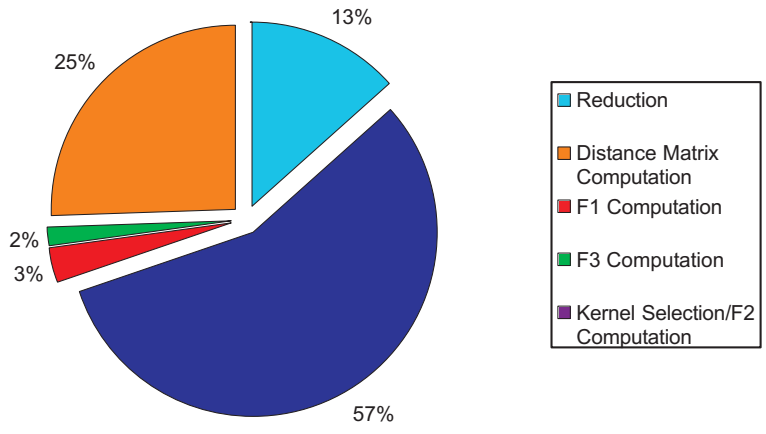


Figure 7.13: Breakdown of running time by procedure for the *a701* dataset.

[8]A tool which determines the CPU used by each procedure or function in a program.

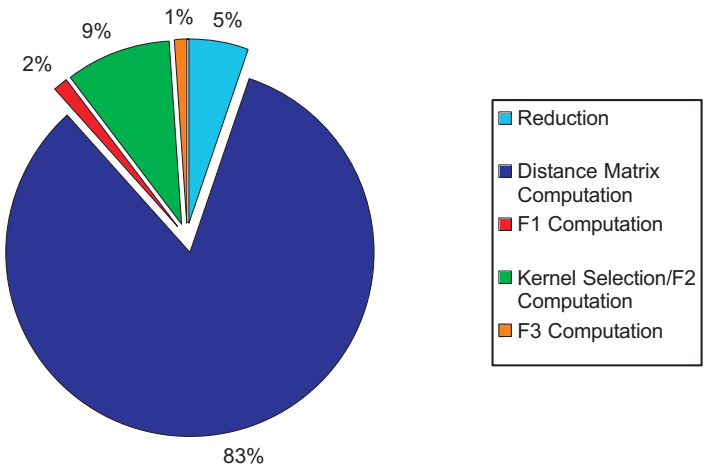


Figure 7.14: Breakdown of running time by procedure for *a702* dataset.

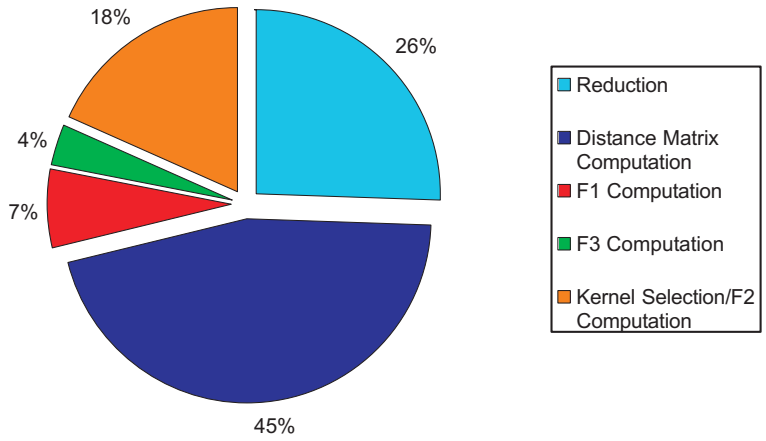


Figure 7.15: Breakdown of running time by procedure for *a703* dataset.

The results clearly show that Valletta's search time is completely unaffected by large training sets and longer strings. The number of different TS descriptions considered by Valletta in each of the case did not exceed 200. This is a very small part of the total search space. An analysis of the output of the code profiler revealed that, in the case of the *a702* dataset Valletta spent more than 80% of the time computing the distance matrix. The distance matrix stores the WLD between all pairs of normal

forms of C^+. Distance matrix computation is quadratic in the number of normal forms. In addition, each distance computation is quadratic in the length of the two strings. The whole process, although still quadratic, is still compute intensive. As the number of features increases and the length of the strings increase, the number of normal forms increases too. The pie charts produced by the code profiler revealed that as the cardinality of C^+ got larger and the strings got longer Valletta took more time to perform string reduction, kernel selection, and in particular, distance matrix computation. This is not of great concern. It should not be too difficult to modify Valletta in order to parallelize these processes. In fact, in Chapter 8, the author proposes a distributed version of Valletta that farms out these processes to a number of machines on a TCP/IP network. A number of tests that were conducted confirmed that this is feasible.

Search Tree Construction

As can be seen in Table 7.1 on page 232, Valletta constructed very small search trees for each of the datasets. The actual CPU required for learning each of the classes varied considerable but the size of the search trees was reasonably uniform. In the search tree each node represents a set of features. For each node, Valletta reduces the strings in C^+ to their normal forms (module the set of features associated with that node) and then search for a set of kernels from amongst the normal forms that minimizes f_2. Nodes are stored in the PENDING list (actually a priority queue) and this list is ordered according to the value of f_2 in each iteration of the main learning loop. The top I nodes are then expanded by adding new children. A further J nodes are then expanded according to the heuristic described in Chapter 5. Figure 7.16 (also shown on page 234) shows the actual search tree built by Valletta for the *a302*

dataset. The final search tree contained 61 nodes. Figure 7.16 does not show all the nodes in the first level of the tree — only those that have children. The number in black shown next to each node is the actual node number assigned by Valletta. The nodes shown encircled in red represent the feature set that returned a value of f_2 of zero. This was the target set of features. Note that the nodes that are expanded

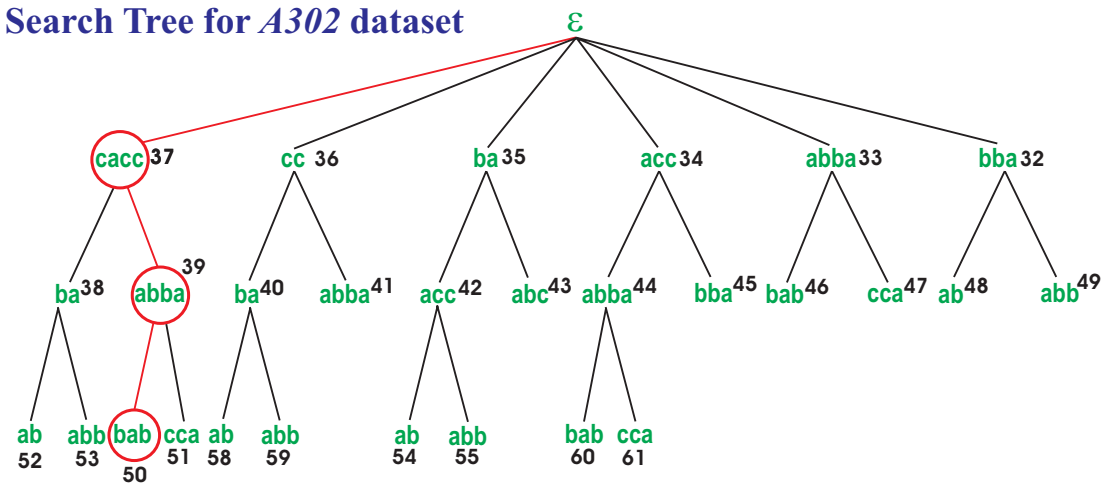


Figure 7.16: The search tree created by Valletta for the *a302* dataset.

by the algorithm are, in the majority of cases, nodes whose labels are either features in the target set of features or substrings of these features. The phenomenon was observed for all of the datasets.

During learning, Valletta maintains a record of the pass number and the values of the f, f_1, and f_2 functions for each node in the search tree. Every time the algorithm detects a reduction in the value of f_2 a log file is updated. Table 7.8, overleaf, shows that trace of the log file for the *a703* dataset. The first column shows the pass number (i.e. iteration of the main learning loop). The next three columns show the values of the f_1, f_2, and f functions respectively. The last column shows the set of features associated with the node that gave a reduction in f_2. Before learning starts, the value of the value of the *bestf2* variable is set to large real constant. Every time the values

of f_1, f_2, and f are computed for a node in the tree, the value of f_2 is compared to the value of *bestf2*. If it is less the log is updated and the value of bestf2 is set to the new value. As can be seen in Table 7.8, the set of features associated with a

Pass	f_1	f_2	f	**Features**
1	446.47	186.52	2.39	acd
1	444.20	178.69	2.48	ggfcbd
1	418.75	150.65	2.78	acdeeegb
3	305.36	130.16	2.34	cdfba, fbacd
3	384.44	118.65	3.24	fba, ggfcbd
3	396.00	112.03	3.53	acd, ggfcbd
3	300.38	92.60	3.24	fba, acdeeegb
3	337.30	86.65	3.89	ggfcbd, acdeeegb
4	303.56	77.64	3.91	cdfba, acefb, acd
4	384.79	72.92	5.27	bfg, fba, ggfcbd
4	300.38	63.58	4.72	bfg, fba, acdeeegb
4	380.60	62.12	6.12	acefb, fba, acdeeegb
4	386.64	58.58	6.60	acefb, ggfcbd, acdeeegb
4	293.95	49.91	5.89	fba, ggfcbd, acdeeegb
5	263.60	49.78	5.29	gfc, bfg, fba, acdeeegb
5	305.06	35.04	8.70	bfg, acefb, fba, acdeeegb
5	271.50	32.31	8.40	cdfba, acefb, ggfcbd, acdeeegb
5	230.88	29.21	7.90	fbacd, acefb, ggfcbd, acdeeegb
6	271.15	28.31	9.57	bacd, bfg, acefb, fba, acdeeegb
6	222.09	23.90	9.29	gfc, bfg, acefb, fba, acdeeegb
6	232.39	22.49	10.33	bacd, cdfba, acefb, ggfcbd, acdeeegb
6	242.51	13.75	17.64	bfg, cdfba, acefb, ggfcbd, acdeeegb
6	282.67	12.99	21.86	bfg, acefb, fba, ggfcbd, acdeeegb
7	222.28	8.10	27.43	gacd, bacd, cdfba, acefb, ggfcbd, acdeeegb
7	175.44	6.10	28.72	gacd, cdfba, fbacd, acefb, ggfcbd, acdeeegb
7	164.27	5.19	31.60	bfg, cdfba, fbacd, acefb, ggfcbd, acdeeegb
8	153.51	1.80	85.05	gac, fbac, bfg, cdfba, acefb, ggfcbd, acdeeegb
8	164.27	0.00	16427.00	gacd, bfg, cdfba, fbacd, acefb, ggfcbd, acdeeegb

Table 7.8: A trace of the f, f_1, and f_2 functions for the *a703* dataset.

reduction in the value of f_2 is almost always a subset of the target set of features or contains strings that are substrings of the target set of features. The target set of

features is shown in the last line of the table. This state of affairs occurred for every dataset used to test Valletta. Figure 7.17 shows a line graph of the values of f, f_1,

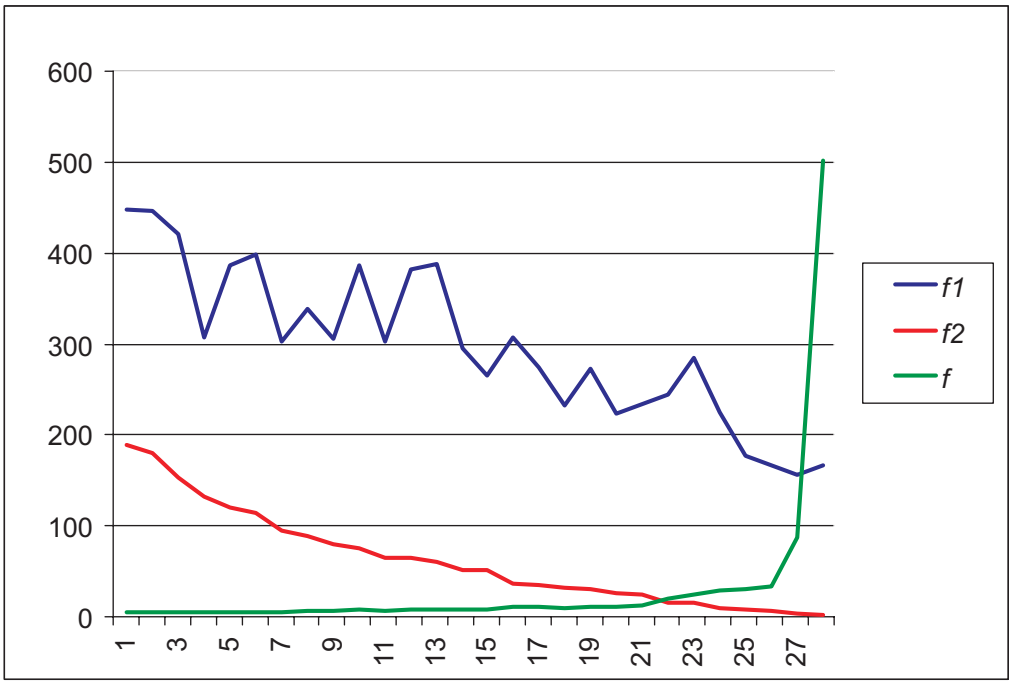


Figure 7.17: The behaviour of the f, f_1, and f_2 functions for the *a703* dataset.

and f_2 functions for the trace in Table 7.8. Note that a reduction in the value of f_2 does not necessarily imply an increase in the value of f_1. Note also that both f_2 and f decrease and increase monotonically respectively. The results obtained go some way in explaining why Valletta manifested relatively quick convergence to the target TS description. The nodes that were expanded, in the majority of cases, represented features set that contained strings that were either target features or substrings of the target features. Even if Valletta adds a string (to a set of features) that is a substring of a target feature, this almost always results in a reduction in the value of f_2. As can clearly be seen in Table 7.8, the set of features '*evolves*' (directed by the value of f_2) into the final set of features that minimizes f_2 and hence maximizes f.

Valletta *vs* Darwin

The efficiency of Valletta's distance-driven search engine in finding the correct TS description after considering only a small number of TS descriptions was beyond the author's initial expectations. It was for this reason that the author designed and implemented the Darwin GA search engine. The aim was to compare Valletta's distance driven search with another popular search technique. Darwin still returned good results but nowhere near those of Valletta. In other words, Valletta's distance-driven search proved much more efficient. Darwin often took many hours to train – even on datasets that Valletta took only a few seconds. The two methods employ completely different search techniques. Valletta starts the search in one single point in the search space and then uses distance to direct the search. Valletta's search is not monotonic and sometimes must backtrack but, in general, it converges to the correct class description very quickly. Darwin, like all genetic algorithm search engines, starts the search in a number of randomly selected locations in the search space and then uses mutation and crossover genetic operators to broaden the search. A genetic algorithm therefore performs a parallel search in the space of all chromosomes. Darwin uses exactly the same pre-processing stage and the same methods for computing f, f_1, and f_2 that are used by Valletta. The fitness function that was used was the value of f_2 for each chromosome. Valletta, in each case, converged to the target TS description after it considered less than 200 different class descriptions. Darwin required several thousand. In essence, the main difference between Valletta and Darwin is the search technique. Also, the stochastic nature of genetic algorithms makes it very difficult to incorporate an inductive preference bias. This is because the search through the hypothesis space is directed by the genetic operators — which are stochastic.

Valletta and the EDSM Algorithm

As we have noted before, Valletta and the EDSM algorithm are not, strictly speaking comparable. However, since kernel languages are, in fact, regular, the author ran the EDSM algorithm on some of Valletta's datasets. In spite of being a general purpose DFA learning algorithm, EDSM did rather badly on the datasets that it was asked to learn. This is not because EDSM is a bad learning algorithm. The learning of DFAs is, in general, very hard. The EDSM algorithm was meant to learn regular languages while Valletta was designed to learn a very specific sub-class. The EDSM algorithm therefore is meant for a much larger domain then Valletta. As a consequence, the EDSM algorithm requires much larger training sets. Let us illustrate this point with a simple example. Humans can learn to distinguish between dogs and cats very quickly because the two classes are well delineated. In other words there are just cats and dogs. A few examples of each class are sufficient for learning. If there were thousands of other species of animals that were neither cats nor dogs but somewhere in between then learning would clearly be much more difficult. Given a small (structurally complete) training set of strings drawn from a kernel language there might be a hundred or so different kernel language TS descriptions consistent with this training set. On the other hand, there might be several million DFAs consistent with this small training set. For this reason, algorithms with a very large learning domain, such as EDSM, require a large number of training examples to learn. More formally, the EDSM learns a very large concept class with a high VC dimension. An implication of this is that a large number of training examples are required so the algorithm can exclude the 'incorrect' class descriptions. The author noticed this in his experiments with the EDSM algorithm. This algorithm is very sensitive to the choice of strings in C^-. The author also feels that this is a strong argument against trying to develop general purpose learning algorithms that are 'too general'. With

ETS one can design the algorithm to suit the domain in question and then incorporate the most appropriate inductive preference bias. A leading researcher once told the author at a NIPS[9] meeting that "the most important thing in learning is to make the most of the data given to you". Many others seems to share this view. It must also be stressed that different learning algorithms require a different definition of what constitutes structural completeness. Valletta's structurally complete training sets were very small while those of EDSM were much larger. Valletta only required that each feature occurred at least twice in the whole training set while ESDM requires that each transition in the target DFA is used a number of times in the generation of C^+. A practitioner who wants to select an algorithm or technique for learning in a particular domain should therefore always ensure that:

(a) The data is *structurally complete* with regards to the selected learning algorithm, and

(b) the *inductive preference bias* of the algorithm is suitable for the domain.

Valletta and the Monk's Problems

The results obtained on the Monk datasets confirmed the importance of using an algorithm that is targeted on the right domain. When Valletta was first used to learn the Monk's problems, the algorithm would take several hours. The author then designed and implemented a special version of Valletta that was targeted at *trivial* kernel languages. In this type of kernel language the features all consist of a single character. These kernel language are called trivial because the set of features induces a trivial string rewriting system (see Chapter 2). The modified algorithm was called *Mdina*. Mdina successfully learned all of the Monk's problems including the third one. For the third problem, an extra string was added to the positive training set

[9] A connectionist group.

to make it structurally complete. Exactly the same inductive preference bias was used on all three problems. Mdina incorporated a new method of computing f_1, i.e. the distance between the positive and negative training sets. This new method is described in Section 7.4 and allowed Mdina to successfully handle misclassification noise.

Other Observations and Notes

- A number of colleagues asked the author: *Does Valletta perform a greedy search?* Valletta builds the feature sets by adding the features that reduce f_2 the most. This might lead one to think that Valletta is performing a greedy search. Such a technique cannot work, however, because, if the set of features is not confluent, then the order that the features are added to the feature set is critical. An investigation of how Valletta was expanding the search tree revealed that it was often the case that Valletta would expand features sets that had very small values of f_2 only for the expanded feature set to register an increase in f_2. This happens because, when the set of features is non-confluent, the order in which the feature set is built is crucial. This phenomenon is what prevents Valletta from conducting a purely monotonic search.

- Our experiments with neural networks on complete parity datasets confirmed that a neural network of the appropriate size and architecture can, in fact, represent mappings such as even parity. The reason the neural networks fail to generalize from incomplete parity training sets must be because they don't have the correct inductive bias. Neural networks of the right size converge from incomplete parity data but learn a mapping other than the parity mapping. There may, in theory, be several mappings consistent with an incomplete parity dataset. The ANN finds the first mapping consistent with the training

examples. This mapping is very often not the parity mapping. The results the author obtained reinforced his conviction that learning algorithms that have a variable inductive preference bias have a distinct advantage over those that have a fixed bias such as ID3, Backpropagation, etc. The author's own criticisms of the neural network approach are the following:

(a) ANNs seem to have a fixed inductive preference bias,

(b) They search a large hypothesis space. The VC dimension of the concept class is therefore usually quite large and a large number of examples are required.

(c) ANNs randomize their weights before learning starts. This means that the final set of weights, and therefore the mapping they learn, is not unique.

(d) The class description returned by ANNs is a set of weights. For most purposes this is unusable.

- It was observed that, in general, Valletta performed much better on alphabets of higher cardinality. This was because, in the case of large alphabets, features, indeed any strings, are statistically much less likely to occur in random strings. For example the string 101 is very likely to occur in random strings over the binary alphabet. The string *abcgaf*, on the other hand, is relatively rare in random strings over the alphabet $\{a, b, c, d, e, f, g\}$. This is well documented in textbooks on word combinatorics [79]. The implication of this is that it is much easier to see regularity in strings over larger alphabets. In fact, the *bin01* and *bin02* datasets were used to test Valletta precisely because the author wanted to test Valletta on kernel languages over binary alphabets. These were, by far, the hardest languages to learn. Very often, each string in C^+ would have numerous normal forms. In fact, *bin02* was used as a benchmark language.

- The *a701*, *a702*, and *a703* datasets were designed to test Valletta's robustness. Different researchers use different definitions of what constitutes robustness. We could, for example, call a learning algorithm robust if it satisfies the following conditions:

  (a) All structurally complete training sets produce the same class description

  (b) Once the learning algorithm discovers the target class description, adding more strings to C^+ should not make the algorithm converge to another class description.

  The first requirement depends on the inductive bias of the algorithm. Different structurally complete training sets can have different sets of TS descriptions that consistent with the training examples. Which TS description is found first depends on the inductive bias of the algorithm. Valletta and GSN do not really optimize the f function. They both stop when the find a TS descriptions that makes f exceed a pre-set threshold. This is not equivalent to optimization. The second condition is much more realistic. Valletta was tested on many different structurally complete training sets for the *a70x* kernel language and the algorithm always converged to the correct class description. Adding more training strings to a structurally complete training set that was previously used to successfully find the target TS description did not affect Valletta. Once it found the correct class description, the addition of more strings to C^+ did not effect performance.

Chapter 8

Conclusions and Future Research

All truths are easy to understand once they are discovered;

the point is to discover them.

Galileo Galilei

This final chapter contains three sections. The first section contains a summary of the conclusions that were reached in this thesis. The second section contains a list of the main contributions made, and finally, in the last section we discuss some of the future directions that the author is currently investigating with regard to pursuing this work.

8.1 Conclusions

The Role of Distance in Class Description

In this thesis the author has tried to show how and why, under the ETS hypothesis for class structure, distance allows for a compact and economical class description of formal languages. Conventional symbolic or propositional class descriptions of formal languages such as grammars and automata are not suitable for noisy classes. Neural

networks and other vector space methods can handle noisy classes better but provide poor descriptions of the class. A set of weights is not considered by many to be an appropriate, i.e meaningful, class description. String TS descriptions allow for a more compact form of description of noisy languages when compared to stochastic grammars and stochastic automata. Noise can take many forms. It can consist of spurious characters inserted, according to some distribution, into the strings of the language or it can take the form of misclassification of samples in the training sets. The features and kernels of the TS description capture the *regularity* of the language. Noise is handled by the distance function that forms an integral part of the description. In this thesis we have also seen how the distance function can be chosen to suit the concept class. The flexibility of the ETS model allows the learning algorithm designer to choose the distance function and the definitions of the f_1 and f_2 functions to suit the learning task at hand.

The Role of Distance in Learning

When the author first started work on this research he was reasonably confident that he could demonstrate the usefulness of distance for the purpose of class description. What was not clear in the beginning, was whether it was possible to show that distance can also be used to direct the learning process. Valletta was conceived, in part, in order to answer this question. The results obtained with Valletta suggest that distance can be used, on its own and without any other heuristics, to direct the search for the target class description. Valletta's distance-driven search engine found the correct class description after considering a relatively very small number of points in the search space. Valletta's search was not necessarily monotonic. Monotonicity can only be achieved if it can be guaranteed that, during training, any new feature added to the current set of features is a feature in the final class description. In other words, each new feature must belong to the final class description. This would allow

the feature set to be built one feature at a time. It is not clear whether this can always be done.

Representation *DOES* Matter

The issue of which is the best structural representation is perhaps one of the least understood areas in Machine Learning. By representation we mean the encoding of the domain of discourse into some mathematical structure such as scalars, vectors, strings, trees, graphs, etc. The choice of representation is important since a change of representation, i.e. the re-encoding of the domain of discourse, may require a completely different learning algorithm, with a different bias, and a different search technique. Moreover, the 'right' inductive bias depends on the choice of representation since the complexity of the class description changes if the representation changes. A class of figures drawn in the Cartesian plane may have a complex class description (context-sensitive grammar) when represented as strings (chain-codes) but may have a very simple class description (context-free graph grammar) when represented as graphs. Since most learning algorithms use a version of Occam's bias and consider simple class descriptions before complex ones, the time complexity of learning may be effected. A complex class description may be 'far away' from the learning algorithm's starting point in the space of class descriptions. Representation is also important since the regularities of a class may become 'invisible' if the set of instances is re-encoded. Learning would still, in theory, be possible but much harder. In this thesis we have seen that a change of representation may also change the requirements of structural completeness. The importance of this point, for machine learning in general, cannot be over-emphasized.

The Advantages of ETS

The ETS inductive learning model provides a general framework for learning and

should not be compared with specific learning algorithms such as ID3, Neural Networks, Candidate Elimination, etc. ETS does not impose a fixed language or preference bias on the learning algorithm designer. The main idea behind ETS is the use of distance for the purpose of class description and for directing the learning process (see above). One can design ETS learning algorithms for any domain. Unlike the case for neural networks, the designer of the ETS learning algorithm is not forced to use vectors to encode the instance space. This is an important advantage of ETS since it allows the designer to incorporate the inductive preference bias and the search technique he or she deems fit for the domain in question. This flexibility was demonstrated in the design of the Valletta algorithm which was conceived, in part, to address the problems that were identified with the earlier GSN algorithm. Valletta uses a new string edit distance function that was purposely developed for kernel languages. Valletta also uses new definitions for the f, f_1 and f_2 functions in order to handle multiple-kernel languages. The search process was entirely directed by the value of f_2 (average inter-distance in C^+). The inductive preference bias was chosen to suit the concept classes. This is not possible with most other learning algorithms. Many learning algorithms fail in certain domains (such as the Monk's problems) precisely because one cannot change their in-built inductive preference bias. With ETS this is not a problem. ETS also provides a natural method for coping with noisy languages. This was demonstrated by the examples of noisy kernel languages that were used to test Valletta as well as by the new method for computing the f_1 function that was developed in order to properly handle misclassification noise.

8.2 Contributions of this Thesis

(a) Refined and updated the definitions of Transformations System (TS) Descriptions of formal languages originally proposed by Goldfarb in [47]. In this thesis

we have seen how and why these TS descriptions allow a more compact and economical representation of formal languages. In particular, compared with other forms of description, TS descriptions provide a more elegant and practical method for describing noisy languages than do stochastic grammars and stochastic automata. Briefly, this is because with string TS descriptions the practitioner can tailor the distance function to suit the domain, quantity of noise, etc.

(b) Formal definitions for the class of regular languages called *kernel languages*. It was also shown that there are practical applications of this interesting class of languages.

(c) The development of a new string edit distance function, *Evolutionary Distance* (*EvD*), and an algorithm for its computation. EvD solves the problems that were identified with the GLD function. EvD is a metric and allows for the proper and correct description of kernel languages. EvD can distinguish between feature noise and kernel noise and incorporates a *feature-repair* facility that removes noise (spurious characters) from corrupted features. EvD can be computed in linear time if the set of features is confluent. EvD was then successfully used in the *Valletta* ETS learning algorithm for learning both noiseless and noisy kernel languages.

(d) A new ETS inductive learning algorithm — *Valletta*. Valletta addresses the problems that were identified in the analysis of the GSN algorithm and incorporates the following features:

- Valletta uses a pre-processing stage to efficiently find all non-overlapping repeated substrings in the C^+ training set and builds a data-structure, the *search lattice*, to enable the learning stage to efficiently construct feature

sets.

- Valletta uses a new data structure, the *parse graph*. The parse graph is used to reduce the strings in the training sets to their normal forms modulo a given set of confluent or non-confluent set of features.
- Valletta employs a new method for computing the f function. This new method was necessary since, unlike the GSN algorithm, Valletta learns multiple kernel languages. An approximation algorithm is used by Valletta for the NP-Complete problem of *kernel selection*. The new f function also allows for correct learning in the presence of misclassification noise which the GSN algorithm could not handle.
- Valletta uses a new search strategy and does not perform simplex optimization. The search strategy used is essentially a search in the space of all possible TS descriptions. Valletta builds a search tree from this space. The search space is completely specified by the set of repeated substrings built by the preprocessing stage. Valletta always learned the correct class description quickly in spite of the huge size of this space. In fact, Valletta always finds a TS description that is consistent with a structurally complete training set.
- Unlike many other learning algorithms such as *Candidate Elimination*, *ID3*, and *Back Propagation*, Valletta is a *variable-preference-bias* algorithm. This means that the user can select the appropriate inductive preference bias according to the application. This is an important advantage over other learning algorithms which have a fixed inductive preference bias.

(e) Valletta is completely *distance driven*. This means that, at each time step in the learning process, only the current distance function is used to direct the search.

No heuristics were incorporated into the search engine in order to accelerate the search. This was because we wanted to verify experimentally that distance can, in fact, be used to direct the learning process.

(f) A series of experiments using Valletta itself, artificial neural networks, the EDSM automata learning algorithm, the *Monk's Problems* datasets, and a version of Valletta that uses a genetic algorithm search engine called *Darwin* were conducted in order to show that:

- Distance can be used to direct the search for the correct TS description. This view is supported by the results that were obtained from comparing Valletta's distance-driven search to the adaptive search technique used by the Darwin genetic algorithm.
- The inductive bias of an algorithm is what determines whether an algorithm learns the class or not. It is meaningless to talk about the robustness of an algorithm since the robustness of a learning algorithm is a function of its inductive bias. The results of experiments conducted with the EDSM algorithm, the Monk's problems datasets, and incomplete parity datasets which were used to train neural networks, support this view.
- Representation ***is*** important. Changing the representation of the domain of discourse may change the learning problem and one may require a different algorithm, with a different search method, and indeed, a different inductive bias. The re-encoding of the objects in the domain of discourse can increase the time complexity of learning since the descriptive complexity of the class changes under different encodings. A change of representation means that the requirements of the structural completeness of training sets changes too. This has important implications for machine

learning since a training set may be structurally complete for one learning algorithm but not for the other.

8.3 Future Research

It is the author's belief that this thesis has only scratched the surface in the area of ETS learning of formal languages. The main aim behind the development of the Valletta algorithm was to provide an extended proof-of-concept of the ideas in Goldfarb's ETS theory. Throughout the preparation of this thesis, we frequently encountered issues and problem that could easily have been Ph.D. topics in their own right. It was sometimes not easy to remain focused on the central research objective. In this section we shall discuss some of the main issues and problems which require further investigation. Also included are some ideas for upgrading and refining the *Valletta* algorithm described in this thesis and also for modifying Valletta for learning other types of formal languages.

8.3.1 Extensions to *Valletta*

Enhancements to the Search Engine Valletta's search for the target TS description is directed only by the value of f_2. Nodes in the search tree, corresponding to set of features, and expanded if they have a small value for f_2. The features added are obtained from the search lattice. The search lattice is only used in order to ensure that any new nodes added to the search tree represent substring-free feature sets. In theory, given a node n in the search tree associated with a set of features F, one could perform a statistical analysis of the normal forms of C^+ modulo F and then choose new features based on the number of occurrences in the normal forms. Suppose, for example, the node n

represents the set of features $F = \{ab, cb\}$. After reducing C^+ modulo F we could expand F by examining the normal forms and finding the substrings in R_{C^+} that occur most frequently in the normal forms. This should significantly accelerate the search for the target TS description.

The Kernel Selection Procedure Finding a set of kernels that minimizes f_2 from amongst the normal forms (modulo a set of features) of C^+ was shown to be NP-Complete. The approximation algorithm used by Valletta selects the 'most likely' candidate kernels from the set of normal forms and then consider various subsets of the set of candidate kernels as determined by the α and β functions. This procedure worked well but is rather compute intensive. Unlike the GSN algorithm, which computes f_2 once, Valletta computes f_2 for many set of candidate kernels and then selects the set of kernels that minimized f_2. This procedure could conceivably be improved by using more complex heuristics to choose which set of normal forms are the most likely to be the target set of kernels. Given a set of normal forms, one could check, for each normal forms in the set, how many times it occurs in C^+ (i.e. in how many strings) and one could also exclude set of kernels which contain strings that are normal forms of the same strings. A number of such heuristics would improve the running time of the kernel selection procedure.

Incremental Learning Although Valletta does not have such a feature, it should be relatively straightforward to add an incremental learning facility to Valletta. Incremental learning is a term used to describe that ability of some learning algorithms to continue learning when the training set changes — usually through the addition of more training examples. This is, of course, what humans do very well. To incorporate incremental learning in Valletta one would have to add the facility to build a new search lattice from the modified training set and

then modify the search tree (i.e. the search space of TS descriptions) and then continuing the learning process.

Parallelization In Chapter 7 we noted that with large datasets, the number of different TS descriptions considered by Valletta was approximately the same as for the case with small datasets. Valletta took somewhat longer to learn because long strings have many normal forms when reduced modulo a set of features. This was particularly evident with the *A702* dataset. The computation of the distance matrix and the kernel selection process therefore took more time. A number of experiments conducted with a code profiler[1] confirmed that these two processes were taking up to 85% of the running time. One idea for dealing with this problem is that the computation of the these two processes can be farmed to different machines on a switched network. Another candidate for parallelization of the search engine itself. During learning, Valletta considers a number of different feature sets (TS descriptions) and computes the f function for each of them. As was seen in Chapter 7, the computation of the f function is rather compute intensive and therefore makes an ideal candidate for parallelization. Again, the computation of the f function for different feature sets can be farmed out to a number of different machines in a network. To investigate this, the author implemented a small TCP/IP network consisting of a PC running the main Valletta program and a number of daemons (slaves). This network is depicted in Figure 8.1. The implementation was straightforward since PowerBASIC includes a powerful TCP/IP Sockets API library. Tests showed that the communications overhead is very small compared to the significant increase in speed that is gained. The main Valletta daemon simply farms out the com-

[1] A software utility that records the CPU time of each function and/or code segment in a running program.

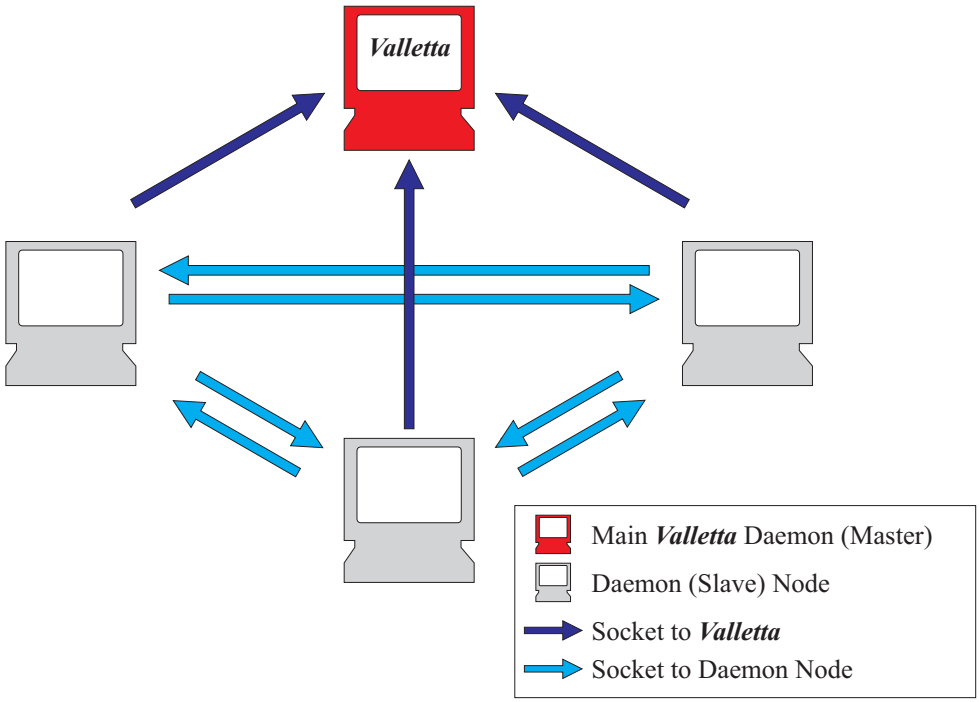


Figure 8.1: A TCP/IP Farm for parallelizing *Valletta*.

putation of f function for each of the feature sets to the slave daemons running on the other machines in the TCP/IP network. The daemons then return the values of f_1, f_2 and f_3 back to the main Valletta program.

8.3.2 A Distance Function for Recursive Features

The rewrite rules that we have considered in this thesis are of the form $a \leftrightarrow \varepsilon$, i.e. the right hand side is always the null string. This means that the rules allow only insertion and deletion of substrings. In general, the introduction of substitutions, i.e. rewrite with both sides being non-empty strings, have more expressive power and can describe languages that cannot be described using only insertions and deletions. For such rules, the most interesting are the recursive rewrite rule of the form $xyx \leftrightarrow y$ for $y \in \Sigma^+$ and $x, z \in \Sigma^*$. These rules are important because the right hand side is a substring of the left hand side and can therefore be applied many times —

hence the name *recursive features*. These rules can be used to describe classes of both regular and context-free languages. The language $a^n b^n$ is an example. The possibility of modifying Valletta to learn such languages depends on the development of string distance algorithms that can work with such rewrite rules and also on the development of pre-processing algorithm that identify candidate recursive features. We feel that both problems can be solved. Valletta would not require much more modification since we envisage that exactly the same distance-driven search technique can be used with these languages.

8.3.3 ETS Learning of Other Regular Languages

In this thesis we considered the learning of kernel languages. We believe that the ideas and algorithms used to learn kernel languages can be used, after being appropriately modified, to learn any regular languages. The idea here is that if one can characterize a regular language by all the cycle-free paths in the minimal canonical automaton for that language, then one could conceivably use rewrite rules that '*eat*' the cycles and thus be able to transform any string in the language into any other string in the same language. Let us demonstrate with a simple example. Consider the DFA of the regular language ab^*a in Figure 8.2 shown below. Note that there is only

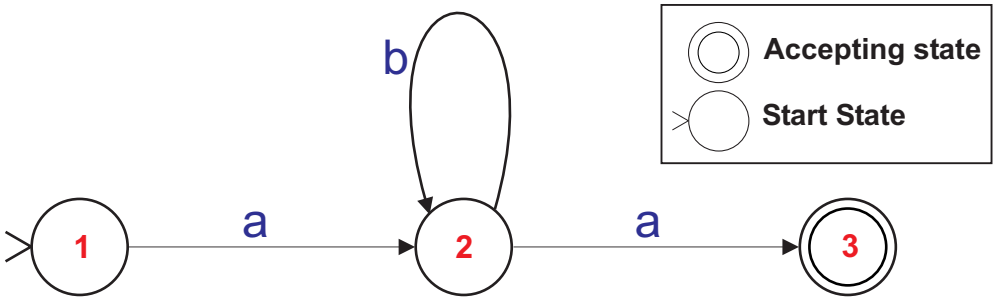


Figure 8.2: A DFA for the regular language ab^*a.

one cycle-free path in the automaton, $\lambda aa\lambda$. The symbol λ is here being used to

denote the ends of the string. One can then use the λ-*transformation* $\lambda ab \leftrightarrow \lambda a$ to transform any string in ab^*a into any other string in the same language. The role of such a rewrite rule is to '*eat*' the cycles. The λ symbol, which denotes the beginning of the string, prevents the rewrite rule being applied anywhere in the string. This idea is feasible since any finite automaton can be characterized by the set of all its k-tails [86]. One would, of course, need to develop a new distance function for these λ-transformations but, other than that, the Valletta algorithm should not require much modification.

8.3.4 Open Questions

(a) It is not exactly clear why Valletta converges so quickly to the correct class description. Distance proved to be suitable for directing the learning process. The results obtained in this thesis must be further corroborated by new research that investigates this issue.

(b) In this thesis we considered only rewrite rules that are 'variable free', i.e. no non-terminal symbols are used in the rule. Rewrite rules that allow variables are more powerful and can describe a broader class of languages than can rewrite rules defined on the terminal alphabet only [80]. The problem is whether it is feasible to design computationally tractable string edit distance algorithms and ETS learning algorithms for this type of rewrite rules. New research is required to investigate these issues.

(c) The number of unique normal forms modulo a given set of features is always much smaller than the number of paths in the parse graph. Is there an efficient polynomial-time procedure for extracting these normal forms?

(d) Is there an ETS learning algorithm that learns kernel languages monotonically?

8.4 Closing Remarks

The debate on which is the best approach for modelling intelligence, connectionist or symbolic, will undoubtedly rage on. In this thesis we presented a new technique for learning a subclass of the regular languages from a finite set of positive and negative training samples. Our algorithm is an implementation of Goldfarb's ETS Model. The ETS model is a unification of both the above competing approaches. It is hoped that this thesis convinces some readers that there is indeed 'life outside the connectionist and symbolic camps' and also that it encourages researchers in machine learning to develop ETS learning algorithms for other domains.

Bibliography

[1] Abela, John, *Topics in Evolving Transformation Systems*, Masters Thesis, Faculty of Computer Science, University of New Brunswick, Canada, **1994**.

[2] Abela, John, *Learning Picture Languages*, Technical Report TR-CS-9605, Department of Computer Science and Artificial Intelligence, University of Malta, **1996**.

[3] Angluin, D., *On the Complexity of Minimum Inference of Regular Sets*, Infomation and Control, Vol. 39, pp. 337-350, **1978**.

[4] Angluin, D., *Inference of Reversible Languages*, Journal of the Association of Computing Machinery, 29(3), pp. 741-765, **1982**.

[5] Anthony, M. and Biggs, N., *Computational Learning Theory*, Cambridge Univeresity Press, **1992**.

[6] Anzai, Y., *Pattern Recognition and Machine Learning*, Academic Press, **1992**.

[7] Baader, Franz and Nipkow, Tobias, *Term Rewriting and All That*, Cambridge University Press, **1998**.

[8] Barsalou, Lawrence W., *Cognitive Psychology - An Overview for Cognitive Scientists*, LEA Publishers, **1992**.

[9] Baxter, Jonathan, *A Model of Inductive Bias Learning*, Department of Systems Engineering, Research School of Information Science and Engineering, Australian National University, Canberra 0200, Australia, **1998**.

[10] Boasson, L., *Grammaires á Non-termineux separes*, International Colloqium on Automata, Langauges, and Programming, *Lecture Notes in Computer Science*, Vol. 85, pp. 109-118, Springer Verlag, Berlin/New York, **1980**.

[11] Boasson, L., and Senizergues G., *NTS Languages are Deterministic and Congruential*, Journal of Computer and System Sciences, Vol. 31, pp. 332-342, **1985**.

[12] Book, Ronald V. and Otto, Friedrich, *String-Rewriting Systems*, Springer-Verlag, **1993**.

[13] Book, Ronald V., *Thue Systems as Rewriting Systems*, J. Symbolic Computation, Vol. 3, pp. 39-68, **1987**.

[14] Boolos, George and Jeffrey, Richard, *Computability and Logic*, Cambridge University Press, **1989**.

[15] Briscoe, Garey and Caelli, Terry, *A Compendium of Machine Learning Volume 1: Symbolic Machine Learning*, Ablex Publishing Corporation, **1996**.

[16] Bunke, H. and Sanfeliu, A., (eds) *Syntactic and Structural Pattern Recognition - Theory and Applications*, World Scientific series in Computer Science, Vol. 7, **1990**.

[17] Buntine, Wray, *A Critque of the Valiant Model*, In *Proceedings of the Eleventh IJCAI*, Detroit, MI, Vol. 1, pp. 837-842, Morgan Kaufmann, **1989**.

[18] Carrasco, Rafael and Oncina, Jose (Eds.), *Grammatical Inference and Applications*, Springer-Verlag, **1994**.

[19] Chan, Tony Y. T., *Learning as Optimization: A Unified Paradigm*, Ph.D. Thesis, University of New Brunswick, **1991**.

[20] Cherkassky, Vladimir and Mulier, Filip, *Learning From Data Concepts, Theory and Methods*, Wiley Interscience, **1998**.

[21] Clarke, A. and Thornton, C., *Trading Spaces: Computation, Representation, and the Limits of Uniformed Learning*, Cognitive and Computing Sciences, University of Sussex, Brighton, UK, **1995**.

[22] Culbertson, Joseph C., *On the Futility of Blind Search*, Techical Report TR 96-18, Department of Computing Science, University of Alberta, Edmonton, Alberta, **1996**.

[23] Davey, B. A. and Priestley H. A., *Introduction to Lattices and Order*, Cambridge Mathematical Textbooks, **1990**.

[24] Dietterich, T. G., *Learning and Inductive Inference* in Edward A. Feigenbaum and Paul R. Cohen, editors, *The Handbook of Artificial Intelligence*, Vol. 3, Chapter 14, William Kaufmann, Inc., **1982**.

[25] Dreyfus, Hubert, *What Computers Can't Do*, First Edition, Harper and Row, **1972**.

[26] Duda, Richard and Hart, Peter, *Pattern Classification and Scene Analysis*, Wiley Interscience, **1973**.

[27] Duda R., Hart P., and Stork D., *Pattern Classification*, Second Edition, John Wiley & Sons, New York, ISBN 0471056693, **2001**.

[28] Dupont, P., *Regular Grammatical Inference from Positive and Negatives Samples by Genetic Search : the GIG method*, Lecture Notes in Artificial Intelligence, No. 862, Springer Verlag, Grammatical Inference and Applications, ICGI'94, pp 236–245, **1994**.

[29] Dupont, P., Miclet, L., and Vidal, E., *What is the Search Space of the Regular Inference* in Carrasco, Rafael and Oncina, Jose (Eds.), *Grammatical Inference and Applications*, Springer-Verlag, **1994**.

[30] Fodor, J. et al, *The Psychology of Language*, McGraw-Hill, New York, **1974**.

[31] Fodor, J., *The Language of Thought*, Thomas Y. Crowell, New York, **1975**.

[32] Fodor, J., *The Mind-Body Problem*, in *Scientific American*, 244, pp. 114-123, **1981**.

[33] Fu, King Sun, *Syntactic Pattern Recognition and Applications*, Prentice-Hall, **1982**.

[34] Gardner, H., *The Mind's New Science - A History of the Cognitive Revolution*, Basic Books, **1987**.

[35] Garey, Michael and Johnson, David, *Computers and Intractability A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, **1979**.

[36] Giles, J.R., *An Introduction to the Analysis of Metric Spaces*, Australian Mathematical Society Lecture Series 3, Cambridge University Press, **1987**.

[37] Gold, E.M., *Language Identification in the Limit*, Information and Control, Vol. 10, pp. 447-474, **1967**.

[38] Gold, E.M., *Complexity of Automaton Identification from Given Data*, Information and Control, Vol. 37, pp. 302-320, **1978**.

[39] Goldfarb, Lev, *A Unified Approach to Pattern Recognition*, Pattern Recognition, Vol. 17, No. 5, pp. 575-582, **1984**

[40] Goldfarb, Lev, *A New Approach to Pattern Recognition*, in L. N. Kanal and A. Rosenfield, editors, *Progress in Machine Intelligence and Pattern Recognition*, Vol. 2, North-Holland Publishing Company, Amsterdam, **1985**.

[41] Goldfarb, Lev, *On the Foundations of Intelligent Processes - 1: An Evolving Model for Patter Learning*, Pattern Recognition, 23, pp. 595-616, **1990**.

[42] Goldfarb, Lev, *A Unified Metric Model for Pattern Learning*, in *Proceedings IASTED International Symposium on Machine Learning and Neural Networks*, M. H. Hamza, editor, New York, October 10-11, pp. 96-99, **1990**.

[43] Goldfarb, Lev, *Why Do We Need the Auxilliary Vector Representation for the Metric Pattern Recognition Problem?*, Technical Report TR90-056, Faculty of Computer Science, University of New Brunswick, **1990**.

[44] Goldfarb, Lev, *A Working Characterization of Intelligence and a New Model*, Technical Report TR90-062, Faculty of Computer Science, University of New Brunswick, **1990**.

[45] Goldfarb, Lev, *Verifiable Characterization of an Intelligent Process*, Fourth UNB Artificial Intelligence Symposium, Lev Goldfarb and Bradford G. Nickerson, editors, pp. 67-80, Sept. 20-21, **1991**.

[46] Goldfarb, Lev, *What is Distance and why do we need the Metric Model for Pattern Learning*, Pattern Recognition, Vol. 25, No. 4, pp. 431-438, **1992**.

[47] Goldfarb, Lev, *Transformation Systems are More Economical and Informative Class Descriptions than Formal Grammars*, in *Proceedings of the 11th IAPR International Conference on Pattern Recognition*, Vol. 2, pp. 660-664, IEEE Computer Society Press, Los Angeles, **1992**.

[48] Goldfarb, Lev, *On some Mathematical Properaties of the ETS Model*, Technical Report TR93-079, Faculty of Computer Science, University of New Brunswick, UNB, September **1993**.

[49] Goldfarb, Lev, and Nigam, Sandeep, *The Unified Learning Paradigm - A Foundation for A.I.*, in Honovar V and Uhr L, editors, *Artificial Intelligence and Neural Networks: Steps Towards Principled Integration*, Academic Press, Boston MA, **1994**.

[50] Goldfarb, L., Abela, J., Bhavsar, V. C., Kamat, V. N., *Can a Vector-Space Based Learning Model Discover Inductive Class Generalization in a Symbolic Environment?*, Pattern Recognition Letters, 16, pp. 719-726, **1995**.

[51] Goldfarb, Lev, *Inductive Class Representation and its Central Role in Pattern Recognition*, in *Proceedings of the 1996 International Mutlidisciplinary Conference on Intelligent Systems, NIST*, Albus, J., Meystel, A., and Quintero, R., editors, U.S. Government Printing Office, Washington, Vol. 1, pp. 53-58, **1996**.

[52] Goldfarb, L. and Deshpande, S., *What is a Symbolic Measurement Process?*, in *Proceedings 1997 IEEE Conference on Systems, Machines, and Cybernetics*, IEEE Press, Vol. 5, pp. 4139-4145, **1997**.

[53] Goldfarb, Lev, and Hook, Jaroslav, *Why Classical Models for Pattern Recognition and not Pattern Recognition Models*, in *Proceedings of the International Conference on Advances in Pattern Recognition*, Plymouth, UK, Singh S., editor, Springer, London, pp. 405-414, **1999**.

[54] Goldfarb, Lev, Golubitsky, O., Korkin, D., *What is Structural Representation?*, Technical Report TR00-037, Faculty of Computer Science, University of New Brunswick, **2000**.

[55] Goldfarb, Lev, Golubitsky, O., Korkin, D., *What is Structural Representation in Chemistry?: Towards a Unified Framework for CADD*, Technical Report TR00-137, Faculty of Computer Science, University of New Brunswick, December **2000**.

[56] Gruska, Jozef, *Foundations of Computing*, International Thomson Computer Press, **1997**.

[57] Gusfield, Dan, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, **1997**.

[58] Hall, P.A., and Dowling, G.R., *Approximate String Matching*, Comput. Surveys, 12, pp. 381-402, **1980**.

[59] Hein, James L., *Discrete Structures, Logic, and Computability*, Jones and Bartlett Publishers, **1995**.

[60] Hermes, Hans, *Enumerability, Decidability, Computability*, Springer-Verlag, **1969**.

[61] Honovar, Vasant and Slutzki, Giora (Eds.), *Grammatical Inference*, Springer-Verlag, **1998**.

[62] Hopcroft, J., and Ullman, J., *Introduction to Automata Theory, Languages, and Machines*, Addison-Wesley, Reading, MA, **1979**.

[63] Huet, Gérard, *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, Journal of the ACM, vol. 27, No. 4, pp 797-821, Octoboer **1980**

[64] Hunt, J.W., and Szymasski, T.G., *An Algorithm for Differential File Comparison*, Comm. ACM, 20, pp. 250-353, **1977**.

[65] Hutchinson, Alan, *Algorithmic Learning*, Oxford University Press, **1994**.

[66] Jain, Sanjay et al, *Systems That Learn - An Introduction to Learning Theory*, MIT Press, **1999**.

[67] Jantzen, Matthias, *Confluent String Rewriting*, EATCS monographs on theoretical computer science, Springer-Verlag, **1988**.

[68] Johnson-Laird, Philip, *The Computer and The Mind*, Harvard University Press, **1988**.

[69] Kearns, Michael and Vazirani, Umesh, *An Introduction to Computational Learning Theory*, MIT Press, **1994**.

[70] Kearns, M. amd Valiant, L., *Crytographic Limitiations on Learning Boolean Formulae and Finite Automata*, Proc. of the 21st ACM Symposium on the Theory of Computing, pp 433-444, **1989**.

[71] Kruskal, B., and Sankoff, D., editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparision*, Addison-Wesley, **1983**.

[72] Lamberts, Koen and Shanks, David, *Knowledge, Concepts, and Categories*, MIT Press, **1997**.

[73] Lang, K., Pearlmutter, B. A., and Price, R., *Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm*, in Honovar, V. and Slutzki, G., (eds), *Grammatical Inference*, Fourth International Colloquium ICGI-98, Ames, Iowa, July **1998**, published by Springer-Verlag.

[74] Langley, Pat, *Elements of Machine Learning*, Morgan Kaufmann Publishers, **1996**.

[75] Lawrence S., Giles, L., Chung Tsoi, A., *What Size Neural Network Gives Optimal Generalization - Convergence Properties of Backpropagation*, Technical Report, UMIACS-TR-96-22 and CS-TR-3617, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, **1996**.

[76] Lee, Lilian, *Learning of Context-Free Languages: A Survey of the Literature*, Technical Report TR-12-96, Center for Research in Computing Technology, Harvard University, **1996**.

[77] Levensthein, V.I., *Binary Codes Capable of Correcting Deletions, Insertions, and Reversals*, Dok. Akad. Nauk. USSR, vol. 163, pp. 845-848, **1965**.

[78] Li, Ming, and Vitanyi, Paul, *An Introduction to Kolmogorov Complexity and Its Applications*, Second Edition, Springer, **1997**.

[79] Lothaire, M., *Combinatorics on Words*, Cambridge University Press, **1983**.

[80] MacNaughton, R., et al, *Church-Rosser Thue Systems and Formal Languages*, Journal of the ACM, vol. 35, pp. 324-344, **1998**.

[81] McNaughton, R., Narendran, P., and Otto, F., *Church-Rosser Thue Systems and Formal Languages*, Journal of the ACM, Vol. 35, No. 2, pp. 324-344, April **1988**.

[82] McCreight, E.M., *A Space-economical Suffix Tree Construction Algorithm*, Journal of the ACM, vol. 23, No. 2, pp. 262-272, **1976**.

[83] Michalski, Ryszard et al, *Machine Learning and Data Mining Methods and Applications*, John Wiley and Sons Ltd, **1998**.

[84] Minsky, Marvin. L., and Papert, Seymour, *Perceptrons*, expanded edition, MIT Press, Cambridge, MA, **1988**.

[85] Michalski, R., Carbonel, J., Mitchell, T., (eds) *Machine Learning - An Artificial Intelligence Approach*, Morgan Kaufmann Publishers Inc., **1983**.

[86] Miclet, L., and de Gentile, C., *Inférence Grammaticale à partir d'Exemples et de Contre-Exemples : deux Algorithmes Optimaux (BIG et RIG) et une Version Heuristique (BRIG)*, Actes des JFA-94, Strasbourg, France, pp. F1-F13, **1994**.

[87] Miclet, L., *Grammatical Inference* in Bunke, H. and Sanfeliu, A., (eds) *Syntactic and Structural Pattern Recognition - Theory and Applications*, World Scientific series in Computer Science, Vol. 7, **1990**.

[88] Mitchell, Tom M., *Machine Learning*, McGraw-Hill, 1997.

[89] Mitchell, Tom M., *Generalization as Search*, Artificial Intelligence, Vol. 18, pp. 203-226, **1982**.

[90] Natarajan, Balas, *Machine Learning A Theoretical Approach*, Morgan Kaufmann Publishers, **1991**.

[91] Needleman, S.B., and Wunsch, C.D., *A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins*, J. Mol. Biology, 48, pp. 443-453, **1970**.

[92] Nigam, Sandeep, *Metric Model Based Generalization and The Generalization Capabilities of Connectionist Models*, Masters Thesis, Faculty of Computer Science, University of New Brunswick, Canada, **1992**.

[93] Nilsson, N. J., *Artificial Intelligence: A New Synthesis*, Morgan Kauffmann, San Francisco, **1998**.

[94] Oliveira, A., and Silva, J., *Efficient Search Techniques for the Inference of Minimum Size Finite Automata*, Workshop on Automata Induction, Grammatical Inference, and Language Acquisition The Fourteenth International Conference on Machine Learning (ICML-97), Nashville, Tennessee, July, **1997**.

[95] Oommen, B.J., and loke, R.K.S., *Pattern Recognition of Strings with Substitutions, Insertions, Deletions, and Generalized Transpostions*, Pettern Recognition, vol. 30, No. 5, pp. 789-800, **1997**.

[96] Peterson, J.L., *Computer Programs for Testing and Correcting Spelling Errors*, Comm. ACM, vol. 23, pp. 676-687, December **1980**.

[97] Pitt, L., *Inductive Inference, DFA's, and Computational Complexity*, Lecture Notes in Artificial Intelligence, K. P. Jankte (editor), No. 397, Springer-Verlag, Berlin, pp. 18-44, **1989**.

[98] Pitt, L., and Warmuth, M., *The Minimum Consistent DFA Problem Cannot be Approximated Within Any Polynomial*, Tech. Report UIUCDCS-R-89, University of Illinois, **1989**.

[99] Quinlan, J. R., *Induction of Decision Trees*, Machine Learning, Vol. 1, pp. 81-106, **1986**.

[100] Randall Wilson, D., and Martinez, Tony R., *Bias and the Probability of Generalization*, in *Proceedings of the International Conference on Intelligent Information Systems (IIS'97)*, pp. 108-114, **1997**.

[101] Reverz, G. E., *Introduction to Formal Languuges*, McGraw-Hill, New York, **1982**.

[102] Rosch, Eleanor, H., *On the Internal of Perceptual and Semantic Categories*, in Timothy E. Morre, ed., *cognitive Development and the Acquisition of Language*, Academic Press, **1973**.

[103] Ron, Dana, *Automata Learning its Applications*, Ph.D. Dissertation, Hebrew University, **1995**.

[104] Sakakibara, Y., *Recent Advances in Grammatical Inference*, Theoretical Computer Science, 185, pp. 15-45, **1997**.

[105] Sakoe, H., and Chiba, S., *Continuous Word Recognition Based on Time Normalization Technique Using Dynamic Programming*, J. Acoustics Soc. Japan (in Japanese), vol. 29, pp. 483-490, **1971**.

[106] Sankoff, David and Kruskall, Joseph, *Time Warps, String Edits, And Macromolecules*, Addison-Wesley, London, **1983**.

[107] Santoso, W. B., *A Learning Algorithm for the Reconfigurable Learning Machine*, Master's thesis, University of New Brunswick, **1992**.

[108] Schalkoff, R., *Pattern Recognition - Statistical, Structural, and Neural Approaches*, John Wiley and Sons, Inc., **1992**.

[109] Sellers, P.H., *The Theory and Computation of Evolutionary Distances: Pattern Recognition*, J. Algorithms, 1, pp. 359-373, **1980**.

[110] Senizergues, G., *The Equivalence and Inclusion Problems for NTS Languages*, Journal of Computer and System Sciences, Vol. 31, pp. 303-331, **1985**.

[111] Simon, H. A., *Why Should Machines Learn?*, In R. S. Michalski, J.G. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*. Chapter 2, Tioga Publishing Co., **1983**.

[112] Sloman, Steven A., and Rips, Lance J., *Similarity and Symbols in Human Thinking*, MIT Press / Elsvier, **1998**.

[113] Stender, J., and Addis, T., (eds.) *Symbols versus Neurons*, IOS Press, Amsterdam, **1990**.

[114] Stephen, Graham A., *String Searching Algorithms*, Lecture Notes in Computing, vol. 3, World Scientific, **1994**.

[115] Sudkamp, Thomas, *Languages and Machines*, Addison Wesley Longman, **1997**.

[116] Tanaka, E., *Parsing and Error Correcting for String Grammars* in Bunke, H. and Sanfeliu, A., (eds) *Syntactic and Structural Pattern Recognition - Theory and Applications*, World Scientific series in Computer Science, Vol. 7, **1990**.

[117] Tanaka, E., *Theoretical Aspects of Syntactic Pattern Recognition*, Pattern Recognition, Vol. 28, No. 7, pp. 1053-1061, **1995**.

[118] Thornton, Christopher J., *Techniques in Computational Learning Algorithms*, Chapman & Hall Computing, **1992**.

[119] Thornton, Chris, *Re-presenting Representation*, Cognitive and Computing Sciences, University of Sussex, Brighton, UK, **1994**.

[120] Thornton, Chris, *Parity: The Problem that Won't Go Away*, Cognitive and Computing Sciences, University of Sussex, Brighton, UK, **1995**.

[121] Thornton, Chris, *There is No Free Lunch but the Starter is Cheap: Generalisation from First Principles*, Cognitive and Computing Sciences, University of Sussex, Brighton, UK, **1999**.

[122] Thrun, Sebastian, et al, *The Monk's Problems: A performance Comparison of Different Learning Algorithms*, Carnegie Mellon University, CMU-CS-91-197, December **1991**.

[123] Thue, A., *Probleme über Veränderungen von Zeichenreihen nach gegbenen Regeln*, Skr. Vid. Kristania, I Mat. Natuv. Klasse, No. 10, **1914**.

[124] Tichy, W.F., *The String-to-String Correction Problem with Block Moves*, ACM Transactions on Computer Systems, vol. 2, No. 4, pp. 309-321, November **1984**.

[125] Tomita, M, *Dynamic Construction of Finite Automata From Examples Using Hill Climbing*, Proc. of the 4th Annual Cognitive Science Conference, USA, pp. 105- 108, **1982**.

[126] Tou, J.T. and Gonzalez, R.C., *Pattern Recognition Principles*, Addison-Wesley, **1974**.

[127] Trakhtenbrot, B. and Brazdin Y., *Finite Automata: Behaviour and Synthesis*, North Holland Pub. Co., Amsterdam, The Netherlands, **1973**.

[128] Ukkonen, E., *Constructing Suffix Trees On-line in Linear Time*, in Leeuwen, J van. (ed), *Algorithms, Software, Architecture: Information Processing 92*, vol. 1, pp 484-492, Elsevier, Amsterdam, **1992**.

[129] Ukkonen, E., *On-line Construction of Suffix Trees*, Report A-1993-1, Department of Computer Science, University of Helsinki, Finland, ISBN 951-45-6384-0, February **1993**.

[130] Van Mechelen, Iven et al, *Categories and Concepts, Theoretical Views and Inductive Data Analysis*, Academic Press, **1993**.

[131] Verma, R., and Goldfarb, L., *A Metric Approach to Isolated Word Recognition*, The Fourth UNB Artificial Intelligence Symposium, Lev Goldfarb and Bradford G. Nickerson, editors, pp. 169-182, September 20-21, **1991**.

[132] Vidal, E., *Grammatical Inference: An Introductory Survey*, in Carrasco, R., and Oncina, J., (eds), *Grammatical Inference and Applications*, Second International Colloquium, ICGI-94, Alicante, Spain, September **1994**, published by Springer-Verlag.

[133] von Helmholtz, H., *The Origin and Correct Interpretation of our Sense Impressions*, in *Selected Writings of Hermann von Helmholtz*, R. Kahl, (ed.), Wesleyan University Press, **1971**.

[134] von Neumann, John, *The Computer and the Brain*, Yale University Press, **1958**.

[135] Wagner, R.A., and Fischer M.J., *The String-to-String Correction Problem*, Journal of the ACM, vol. 21, pp. 168-173, **1974**.

[136] Warmuth, Manfred K., *Towards Representational Independence in PAC Learning*, In K.P. Janke editor, *International Workshop: Analogical and Inductive Inference*, GDR, pp. 78-103, Lecture Notes in Artificial Intelligence (lncs) 397, Springer-Verlag, New York, **1989**.

[137] Watanabe, Satosi, *Pattern Recognition: Human and Mechanical*, Wiley Interscience, **1985**.

[138] Weiner, P., *Linear Pattern Matching Algorithm*, in *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pp. 1-11, **1973**.

[139] Wolpert, D., and Macready, W., *No Free Lunch Theorems for Search*, Unpublished MS, **1995**.

Appendix A

Using Valletta

Valletta was developed under *Microsoft Windows 2000*® and written in the *PowerBASIC Console Compiler*® (PBCC). PBCC is a dialect of BASIC that was designed for very fast server-based web applications that run under Windows NT/2000. It is produces only text (or console) mode applications and therefore does not have any GUI overhead. PBCC also has the advantage that it incorporates some very efficient string processing routines written in assembly language. In some tests we performed it was nearly always faster than the best C compilers. In addition, PBCC supports multi-threading, has in-built array sorting and searching, and produces only optimized 32-bit code. PBCC was perfect for implementing Valletta. Valletta, the program, is a 'no-frills' design but is still very user-friendly and easy-to-use. Valletta consists of just one single executable of just 100K in spite of the fact that the whole program required well over 5,500 lines of code. Running the program requires only loading and running the executable *valletta.exe*. When Valletta loads it asks the user for the filename of the *problem file*. Valletta then does the rest. During learning, Valletta automatically displays and updates a number of values that allow the user to monitor the progress of the learning process (see Figure A.1). These inlcude the values of f, f_1, and f_2 as well as search tree statistics.

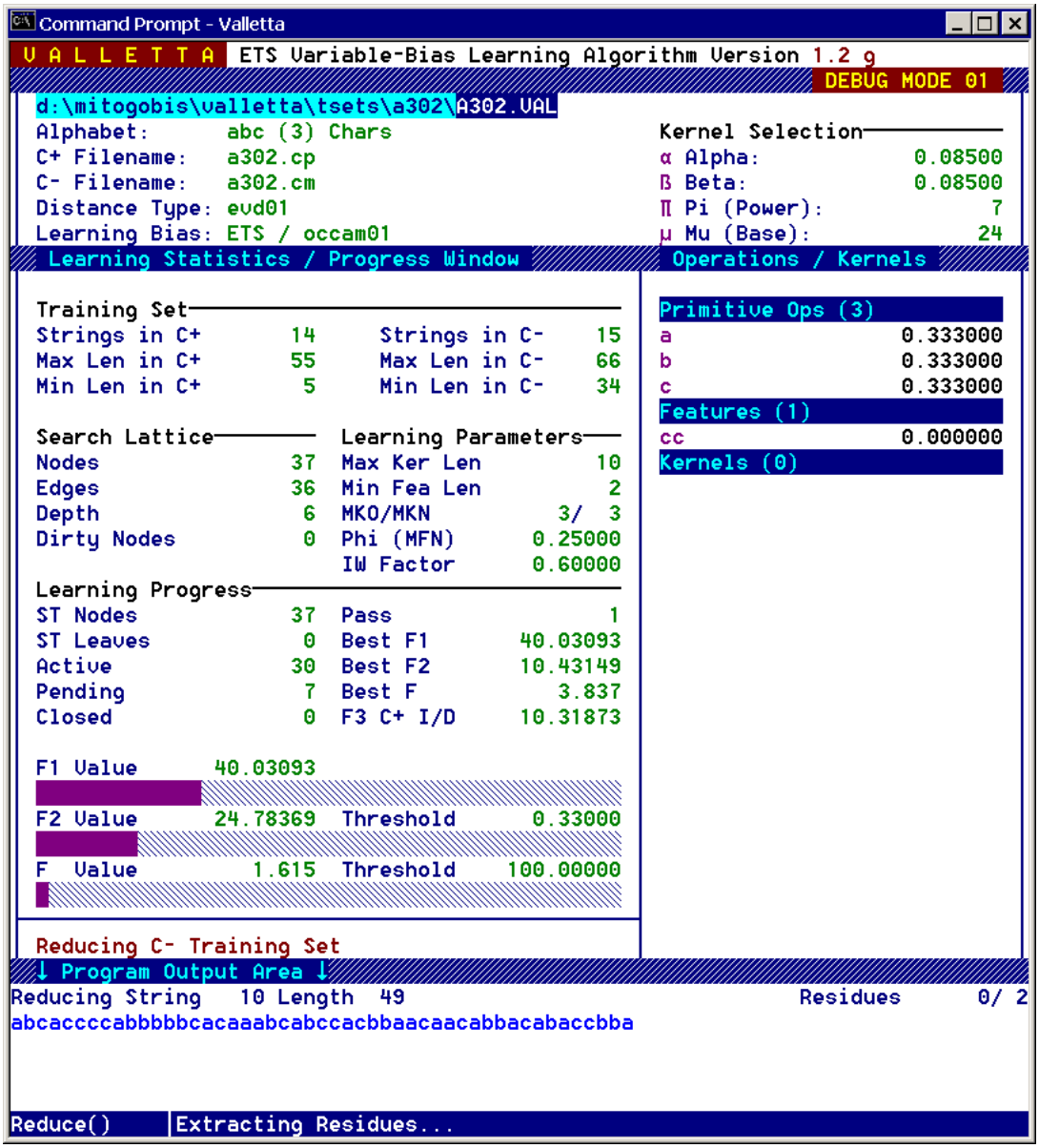


Figure A.1: The screen dump of Valletta during the learning process.

The problem file must have a filename that contains up to 8 characters and must have the extension "*.val*". The problem file contains the alphabet, the operating mode, the name of the inductive bias parameters file, and the type of search to be used by the Valletta's search engine. The composition and structure of the problem file is described at the end of this appendix.

The C^+ and C^- training set files must be prepared by the user before invoking Valletta and must have the same filename as the problem file but must have the extensions "*.cp*" and "*.cm*" respectively. Each file is a conventional ASCII text file containing the training strings with the carriage return code used as a delimiter - i.e. one string per line. The only other required file is file that contains the set of all non-overlapping substrings in C^+. This file have the same filename as the problem file but with the extension "*.rep*". The user would typically first create the training set files and then invoke the GAST program, *gast.exe*, in order to create the "*.rep*" file. The GAST program accepts as input a "*.cp*" file, constructs the global augmented suffix trie, and then finds all the non-overlapping repeated substrings. These are written to the "*.rep*" file. When this task is completed, Valletta can be invoked, the program file filename is entered, and learning can proceed.

The problem file consists of a simple ASCII text file that contains the following lines:

alphabet This is a string that contains the alphabet of symbols in lexicographical order. Note that *Valletta* checks the training sets to ensure that the strings contain only valid symbols.

bias The is the name of the *inductive bias parameters file* but without the *.inb* extension.

mode This parameter is a positive integer in the range 1 to 3 and specifying the *operating mode*. There are 3 distinct operating modes:

- ***0*** for *Fast Mode*. In this mode *Valletta* operates in the fastest possible manner.
- ***1*** for *Verbose Mode*. In this mode *Valletta* generates more screen output and writes logs.

- ***2*** for *Debug Mode 1*. In this mode *Valletta* as above but creates files containing the normal forms of all strings in C^+.
- ***3*** for *Debug Mode 2*. As above but also creates text file containing a trace of the computation of the f function, the interdistance matrices, etc. Used for debugging.

search type This is an integer that stores the type of search technique to be used by Valletta.

- ***0*** for distance-directed ETS search.
- ***1*** for search using the *Darwin* Genetic Algorithm (GA) search engine.
- ***2*** for a step-by-step ETS search - i.e the program pauses after step to allow the user to check the screen display. Used for debugging.

After the user enters the problem file filename, Valletta loads the C^+ and the C^- training files and checks their validity. It then loads the file containing the repeated substrings of C^+ and builds the search lattice. It then pauses for user input. When the user presses any key the learning process commences.

It is not usually practical to store the training set files in the same directory as Valletta. For this reason Valletta loads the path for the training set files and the inductive bias files from the *valletta.ini* file. This file is an ASCII text file that contains just one string - the path to the training set files, the bias files, and the log files. The log files are created during the learning process and contain a 'trace' of the learning process.

The composition and structure of the inductive bias parameters file is described in Appendix B.

Appendix B

Valletta's Inductive Bias Parameters

Valletta's inductive bias parameters are loaded from the file *problem.inb*. This files is a standard ASCII text file and contains the following parameters:

distance function This is an string value used to specify the type of distance function used for F_2 computation.

evd01 This is standard EvD distance. It does not perform feature repair.

evd02 This is EvD distance with feature repair.

MaxKerLen This is the maximum length any kernel of a TS description discovered by VALLETTA.

MinFLen The is the minimum length of a feature in a TS description discovered/considered by VALLETTA. Only features with length equal to or greater than **MinFLen** will be considered

FTH This is a positive real number and represent the f function threshold. Learning stops when this value is exceeded.

F2TH This is a positive real number and represent the f_2 function threshold. The target TS description must return a value of f_2 that is less than this threshold.

ø **O** This is a real number in the interval (0, 1]. This value is multiplied by $|C^+|$ to obtain *MKO*, the minimum number of times a kernel can occur in C^+.

χ **Chi** This is a real number in the interval (0, 1]. This value is multiplied by $|C^+|$ to obtain *MKN*, the maximum number of kernels in the target TS description.

α **Alpha** This is a real number in the interval (0, 1]. This value is passed as a parameter to the $\mathbb{S}_\alpha$ function which is used to control kernel selection. See Chapter 5.

β **Beta** This is a real number in the interval (0, 1]. This value is passed as a parameter to the $\mathbb{S}_\beta$ function which is used to control kernel selection. See Chapter 5.

π **Pi** This is a positive integer in the range 2-12. It is passed as a parameter to the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ functions which are used to control kernel selection. See Chapter 5.

μ **Mu** This is a positive integer in the range 2-32. It is passed as a parameter to the $\mathbb{S}_\alpha$ and $\mathbb{S}_\beta$ functions which are used to control kernel selection. See Chapter 5.

MFN . This is a real number in the interval [0, 1] and represents the maximum amount of noise allowed in a feature. This value is used for feature repair.

Notes:

The above parameters are used to specify Valletta's inductive preference bias and also to define the requirements of the structural completeness of the training sets. The value of *MaxKerLen*, *MinFLen*, ø, and χ are also used to define the structural

completeness requirements of C^+. The other parameters are used to control the algorithms inductive bias. The values of α and β can be changed by the user to give preference to TS descriptions that minimize or maximize the number of features and kernels.

Appendix C

Training Sets used to test *Valletta*

In this Appendix we list some of the datasets used to train Valletta.

bin01 Binary alphabet, medium sized training set (≈ 35), single kernel, no noise.

Kernel: *101*.

Features: *11*, *00*, and *010*.

Non-confluent.

bin01n Binary alphabet, medium sized training set (≈ 35), single kernel, 2% misclassification noise.

Kernel: *101*.

Features: *11*, *00*, and *010*.

Non-confluent.

bin02 Binary alphabet, large sized training set (> 64), single kernel, no noise.

Kernel: *101*.

Features: *11*, *00*, and *010*.

Non-confluent.

a301 Three-letter alphabet $\{a, b, c\}$, medium sized training set (≈ 45), one kernel,

no noise.

Kernel: *abbcb*.

Features: *bab*, *abba*, *cacba*, and *bbcab*.

Non-confluent.

a302 Three-letter alphabet $\{a, b, c\}$, medium sized training set (≈ 45), two kernels, no noise.

Kernels: *abbcb*,*cccbbc*.

Features: *bab*, *abba*, and *cacc*.

Non-confluent.

a302n Three-letter alphabet $\{a, b, c\}$, medium sized training set (≈ 45), two kernels, 12% noise.

Kernels: *abbcb*,*cccbbc*.

Features: *bab*, *abba*, and *cacc*.

Non-confluent.

a701 Seven-letter alphabet $\{a, b, c, d, e, f, g\}$, medium-sized training set (≈ 40), two kernels, no noise.

Kernels: *gadcffb*,*acbcffgcbbe*.

Features: *bfg*, *gacd*, *acefb*, *fbacd*, *ggfcbd*, *cdfba*, and *acdeeegb*.

Non-confluent.

a702 Seven-letter alphabet $\{a, b, c, d, e, f, g\}$, large training set (≈ 128), two kernels, no noise.

Kernels: *gadcffb*,*acbcffgcbbe*.

Features: *bfg*, *gacd*, *acefb*, *fbacd*, *ggfcbd*, *cdfba*, and *acdeeegb*.

Non-confluent.

a703 Seven-letter alphabet $\{a, b, c, d, e, f, g\}$, small training set (≈ 24), two kernels,

no noise.

Kernels: *gadcffb*,*acbcffgcbbe*.

Features: *bfg*, *gacd*, *acefb*, *fbacd*, *ggfcbd*, *cdfba*, and *acdeeegb*.

Non-confluent.

a703n Seven-letter alphabet $\{a, b, c, d, e, f, g\}$, small training set (≈ 24), two kernels, 10% noise.

Kernels: *gadcffb*,*acbcffgcbbe*.

Features: *bfg*, *gacd*, *acefb*, *fbacd*, *ggfcbd*, *cdfba*, and *acdeeegb*.

Non-confluent.

a1101 Eleven-letter alphabet $\{a, b, c, d, e, f, g, h, i, j, k\}$, small training set ($\approx 15$), three kernels, no noise.

Kernels: *hgaidcfkfb*, *fkighbdafkfb*, *jahbcfikgcbde*.

Features: *aih*, *degai*, *acefb*, *bfjbace*, *cjkcbd*, *cdfba*, and *facdiikhgeeb*.

Non-confluent.

a1101n Eleven-letter alphabet $\{a, b, c, d, e, f, g, h, i, j, k\}$, small training set ($\approx 15$), three kernels, no noise.

Kernels: *hgaidcfkfb*, *fkighbdafkfb*, *jahbcfikgcbde*.

Features: *aih*, *degai*, *acefb*, *bfjbace*, *cjkcbd*, *cdfba*, and *facdiikhgeeb*.

Non-confluent.

In the datasets that follow, • denotes the end-of-string marker.

A301

Features	*bab, abba, bbcab, cacba*
Kernel/s	*abbcb*
Remarks	Multiple kernel, medium training set, non-confluent, no noise.

C+ (**14 strings, 5/55**)

abbbbcabcbabbabba•

acacbcbbcb•

cacbcbbcabcacbccacbcabbaacacbcbabbbcabbabbcacbcabbabbcabbbcabbbababba

babbabbabcabbababbbcabbbcabbbabbbcababbaabbabbcabcacbcbbcab•

babbbcababbcabcacbcbabbcacbcbbcabbbcabcacbcbbbcabbbcabcbabbabbababbab

bbcabcacbc•bbcabbbcabcacbcababbbcacbcbabccacbcbbabcacbc•

abbaabbaaabbababcacbcbbbcabcacbcabbabbabbbcabbabcacbccbbcabbbbcababba

cacbcbabcacbc•

bbcababababbabbcabcacbcbabbcacbcbabbbcabbbcabbbbcabcacbccacbcbbcabbbc

abcbbcababbababbbcabbabbbbcababbabbcabcacbcbbcabcacbcbabbbcabcacbcb

bcab•

bbcabbababbaaabbacacbcabbabbbcabcacbcbabbababbababbaccacbcbbcabcacbcb

bcabbcacbcbbcabcacbcabbabbcabcacbcbbcab•

acacbccacbcbbbcabbcbababbabbababba•

acacbcabbabbabbbcabbabcacbcbabbaabbacacbcbbcabcacbccabbaabbacacbcbbab

babbabcacbcbbcabbabcacbcbab•

bbcabcacbccacbcababbabbcabbabbacacbccbabcacbcbcacbcbabbabbbcabbab•

cacbcbabbbcabcacbcaabbababbababbabbbcabcacbcbabbabbcabbbcabbbcabbabcb

abbabbbcababbababbaabbaabbacacbcabbaabbaabbabab•

abbabbcb•

bbcabbbcabaabbababbabbabbabcabbabbcabbcacbc•

abbcb•

aabbacacbcbbcabbabbaabbabbcabbcacbcbbcabcacbccabbababababbababbcacbcbbc

abcacbcbabcacbc•

cacbccacbccacbcacacbcbbababbabbcabbcacbcabbaabbacbbcabbbcabbbcabbabbc

acbccacbcbabcacbcabba•

bbcababbacacbcabbaacacbcabbacacbcbbabcacbcbbcabbbababbabbcabbabcbabba

bbbcabbabbbabbbcabbbcabbbcabcacbccacbcabbaabbabbcabbab•

cacbcabbaaabbabbcabbbbbcababbababcabbabbababbabbcabbab•

bbcabbbcababbaabbabbabbabbcabbbcababbacbcacbccacbcbabbababba•

cacbcababbabcacbcabbaccacbcbbabcacbc•

abbaabababbbabbbcabbbabcacbccbbcabbbcabbbbcababbababbabbbcab•

abbababababbababbcabcacbccacbcbbbcabcacbcbabbabbaabbabbcabcacbccbbcab

bbcabbabbcacbcbbcabbabbabcacbcbab•

bababbaabbacacbccacbcacacbcbbcababbababbababababbabbbcabbbcabbbcabccacb

cbababbabbabbbcababbabbcababbacacbc●
cacbcbbcabcacbcacacbccacbcbababbabbababbababbbabcacbcabbacacbcccacbcb
bcabcacbcbbbcabcacbcbabbbcabcacbcbbcabbab●
babcacbcabbaabbcabbababbababbbcabbabbaabbabbcabbabcacbcbbabcbabbbcabc
acbcabbacacbcbbbcabbbcababbababcacbc●
bababbabbcabcacbcabbcabbbcabbbcabbabbcacbcbcacbcbababbacacbccbababbab
bcabbcacbccacbcbbcababbabbcabbab●
abbababbbcababbcababbababbbbcababbabbabcacbccacbcccacbccacbcabbababba
cacbc●
babcacbcabbacacbcabbaacacbcbabcacbcbabbbbcabbbcabcacbcbabbabbcabcbbca
bbabbbcabbbcabbcacbcbabcacbcbabbabbabbbcabbbcab●

C- (15 strings, Min/Max Length 34/66)

bcbbbcccbcbbccbbcacacacccabbccabccbbbabbba●
cccabbcacbcaaacabcccbccbccbbabcbac●
abbbaabccaabcbcaccabccacacabbcccabaabcaabaaaacbcaabccabbccbbcaccbc●
aabbbbaaccbacbcaaacabbbbcbcacaaaaabbbabcacabccbacbbcbbcbcbca●
cabaacbaaacaacbccbcabacacaaaaccccaccccaccaaacbcbabbabbbcacacaabaa●
bbaacbacaacabaaaaacbabbbaaacbabbbabbacabcaacaccaacac●
baacaccbbbaabbcabbaaaaccaacbbbcabccabacacaccbbcccaaabacccbc●
bbbaabacacbcaabbbaaaaacacacbccbaabbcc●
accaabbabccbbcbbcaabaacaaaaaccaaaacaaaacbccaaca●
abcaccccabbbbbcacaaabcabccacbbaacaacabbacabaccbba●
babbcbcccacaaccbabcbccabcabcacaccaaaccaacccbaabaaab●
cabbabbbabccabbcbcccaacacabaacaccc●
ccacabacabbcabbbacccccabccaabcbcabbaaaaaacabacbacbaaacabbbcc●
acbbaccbccaaaabccababacbbccbcaaaccbbbbbbabcabbbc●
cbbaabaacaaaaaacbbccaabbaababababbbbccccbccac●

A302

Features	*bab, abba, cacc*
Kernel/s	*abbcb, cccbbc*
Remarks	Multiple kernel, small training set, non-confluent, no noise.

C+ (**14 strings, 5/55**)

ababbababbaccaccb●
abbcb●
bababababbabccaccbabba●
caccabbaacaccbbabbabbababccacccaccbabbacacccacc●
abbcabbab●
caccabcaccbcb●
abbabbcb●
caccbabccabbaccaccbabbababbacbabcaccabba●
bababbacbabccabbabbabbaccaccbabbab●
abbacaccccacccacccbabcabbababbbabbabbcaccabbacabbacacc●
babcabbaccbabbababbacbab●
babcacccabbababccaccccacccaccbbabbabbccaccabbababababba●
babccaccbabccaccbabcabbaabbababbacaccbabbaabbaccacccacc●
caccccacccabbaabbacabbabbabcaccbabbacaccccacccacccacc●

C- (**15 strings, Min/Max Length 34/66**)

bcbbbcccbcbbccbbcacacacccabbccabccbbbabbba●
cccabbcacbcaaacabcccbccbccbbabcbac●
abbbaabccaabcbcaccabccacacabbcccabaabcaabaaaacbcaabccabbccbbcaccbc●
aabbbbaaccbacbcaaacabbbbcbcacaaaaabbbabcacabccbacbbcbbcbcbca●
cabaacbaaacaacbccbcabacacaaaaccccacccccaccaaacbcbabbabbbcacacaabaa●
bbaacbacaacabaaaaacbabbbaaacbabbbabbacabcaacaccaacac●
baacaccbbbaabbcabbaaaaccaacbbbcabccabacacaccbbcccaaabacccbc●
bbbaabacacbcaabbbaaaaacacacbccbaabbcc●
accaabbabccbbcbbcaabaacaaaaaccaaaacaaaacbccaaca●
abcaccccabbbbbcacaaabcabccacbbaacaacabbacabaccbba●
babbcbcccacaaccbabcbccabcabcacaccaaaccaacccbaabaaab●
cabbabbbabccabbcbcccaacacabaacaccc●
ccacabacabbcabbbacccccabccaabcbcabbaaaaaacabacbacbaaacabbbcc●
acbbaccbccaaaabccababacbbccbcaaaccbbbbbbabcabbbc●
cbbaabaacaaaaaacbbccaabbaababababbbbccccbccac●

Appendix D

GI Benchmarks

The following are some public training samples for grammatical inference. The list was compiled by Pierre Dupont and is available at the *Homepage of the Grammatical Inference Community* (see Appendix F) and the *Gowachin Competition* web-site (see Appendix E). The training samples for these languages are available at these sites. The first set of 7 benchmarks was proposed by M. Tomita in [125]. All examples are over a two-letter alphabet and the size of the target DFAs ranges from 1 to 4. Notice that L_1 and L_2 are kernel languages.

L_1 a^*.

L_2 $(ab)^*$.

L_3 any sentence without an odd number of consecutive a's after an odd number of consecutive b's.

L_4 any sentence over the alphabet $\{a, b\}$ without more than two consecutive a's.

L_5 any sentence with an even number of a's and an even number of b's.

L_6 any sentence such that the number of a's differs from the number of b's by 0 modulo 3.

$\mathbf{L}_7$ $a^*b^*a^*b^*$.

A second set of 7 languages was compiled by L. Miclet and C. de Gentille [86] and by P. Dupont [28]. This set contains languages over a three-letter alphabet. The data was generated by random walks in the canonical DFA for each language and its compliment. For each language, 20 training sets were generated as just described and the resulting samples range in size from 1 to 156 strings. whereas their length varies from 1 to 1017 characters.

$\mathbf{L}_8$ a^*b.

$\mathbf{L}_9$ $(a^* + c^*)b$.

$\mathbf{L}_{10}$ $(aa)^*(bbb)^*$.

$\mathbf{L}_{11}$ any sentence with an even number of a's and an odd number of b's.

$\mathbf{L}_{12}$ $a(aa)^*b$.

$\mathbf{L}_{13}$ any sentence over the alphabet $\{a, b\}$ with an even number of a's.

$\mathbf{L}_{14}$ $(aa)^*ba^*$.

$\mathbf{L}_{15}$ $bc^*b + ac^*a$.

Another benchmark was proposed by A. Oliveira and J. Silva. This benchmark was developed in order to test the BIC algorithm [94]. The benchmark consists of 115 randomly generated DFAs whose size, after minimization, is between 3 and 19 states. A total of 575 training sets were generated, with each training set containing twenty strings with the length of each string being exactly 30 characters. Finally, the reader is referred to the Abbadingo and Gowachin regular-language inference competitions (see Appendix E). The web-sites for these competitions includes links to some training datasets (languages over a 2-letter alphabet only).

Appendix E

GI Competitions

E.1 The Abbadingo One Learning Competition

In 1996, Babak Pearlmutter and Kevin Lang posted a set of challenging DFA learning problems designed to allow researchers to test their favourite learning algorithms. This benchmark was formulated in the form of the Abbadingo One Learning Competition. The datasets were artificially generated (i.e., the target DFAs were randomly generated). The Abbadingo competition evoked significant interest and several people from all over the world participated in the competition. The eventual winners came up with algorithms that were significant improvements over the existing methods for learning DFAs. The competition is now over but one can still download the challenge problems. The Abbadingo homepage contains a number of articles on Abbadingo One as well as links to paper that describe the winning algorithms. The Abbadingo One homepage is located at:

http://abbadingo.cs.unm.edu/.

E.2 The Gowachin DFA Learning Competition

Following the success of Abbadingo One, Kevin Lang, Babak Pearlmutter, and Franois Coste teamed up to launch the Gowachin Learning Competition. In this competition, users are allowed to generate their own problem (by specifying the size of the target DFA, the number of training examples, and the noise level). The Gowachin homepage is located at:

http://www.irisa.fr/Gowachin/.

Appendix F

Internet Resources

F.1 Grammatical Inference Homepage

The semi-official *Homepage of the Grammatical Inference Community* is located at: http://www.univ-st-etienne.fr/eurise/gi/gi.html.
The website is maintained by Colin de la Higuera, Email: `cdlh@univ-st-etienne.fr`, and contains tutorials, links to data and program repositories, conference announcements and links, links to homepages of researchers in the GI community, a list of publications and technical reports, and links to journals and books.

F.2 The *pseudocode* LaTeX environment

All the algorithm pseudocode in this thesis was formatted using the excellent *pseudocode* LaTeX environment created by D.L. Kreher of the Department of Mathematical Sciences, Michigan Technological University, Houghton, Michingan and D.R. Stinson of the Department of Combinatorics and Optimization of the University of Waterloo, Waterloo, Ontario. The package is easy to use, produces an easy-to-read Pascal-like syntax and is easily customizable.

The style file *pseudocode.sty*, together with other required files and documentation can be downloaded from the following WWW site:

http://www.math.mtu.edu/ kreher.

Appendix G

The Number of Normal Forms of a String

If a set of features induces a confluent string rewriting system we are guaranteed to have only one normal form per string. If, on the other hard, the set of features is non-confluent then each string can have many normal forms. In some pathological cases the number of normal forms can be super-polynomial in the length of the string. In fact, the number of normal forms of a string (modulo a given set of features) is bound from above by a function that is exponential in the length of the string. This is because the number of ways in which the features can be deleted from a string is equal to the number of paths (from the *start* node to the *end* node) in the parse graph of the string. This can happen with a set of features that overlap with each other. The number of paths between the start node and the end node of the parse graph can be exponential in the number of nodes and hence the length of the string. In practice, the number of unique normal forms is usually much smaller since many paths yield the same normal form. Consider the string *000* and the feature *00*. The feature can be deleted from this string in two different ways, i.e. *0A* and *A0* where $A = 00$. In each case the normal form is the same, i.e. *0*. The fact that the number

of unique normal forms is usually much smaller than the number of paths in the parse graph leads the author to feel that it may be possible to develop an efficient algorithm that will extract all the unique normal forms from the parse graph without considering all possible paths. As described in Chapter 6, a number of heuristics

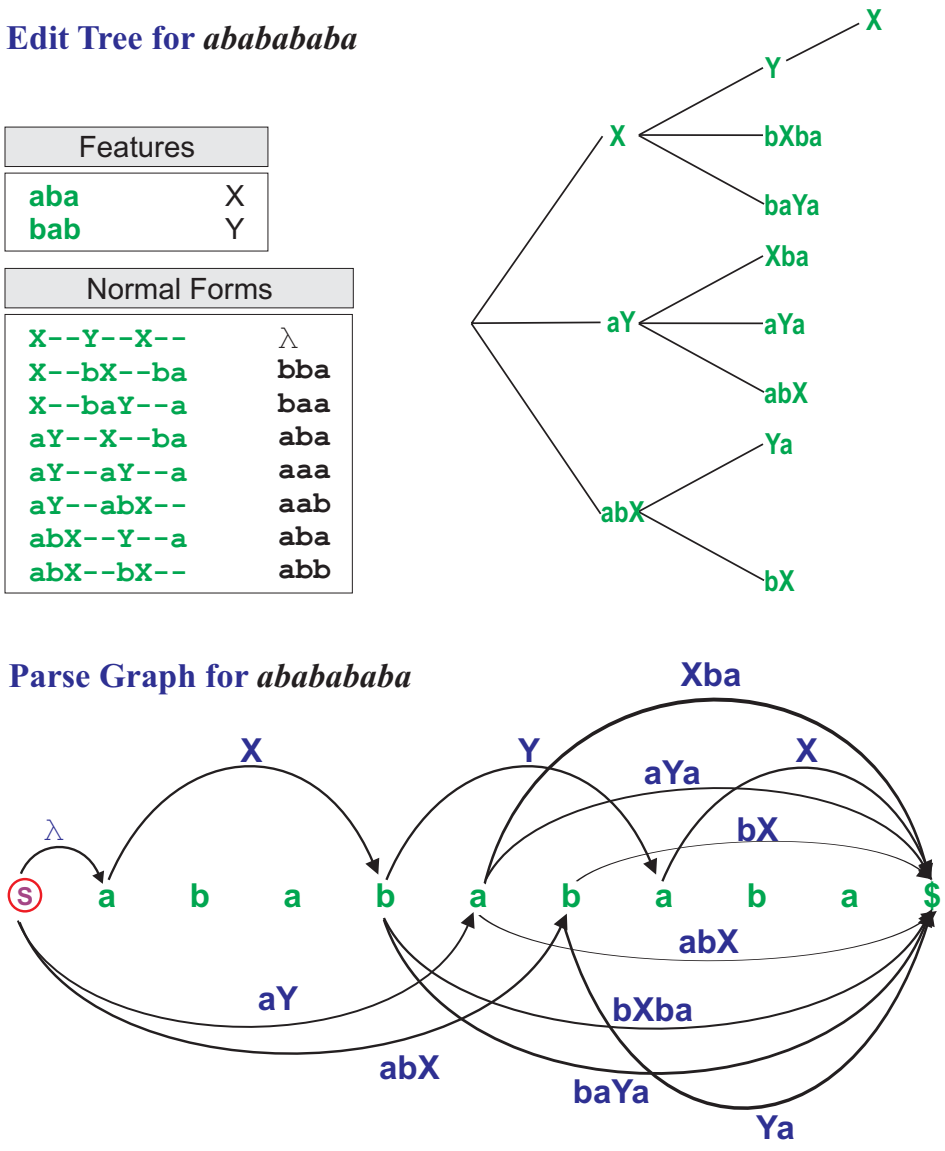


Figure G.1: The reduction of the string *abababa* modulo the feature set $\{aba, bab\}$.

were used in Valletta in order to speed up the search for the unique normal forms but the author feels that faster algorithms are possible.

It was observed that in certain cases, in particular when considering strings over two-symbol alphabets, the number of unique normal forms would approximate the total number of paths in the parse graph. A relatively short string (50-80 characters) would then have quite literally thousands of normal forms.

Consider the set of features $\{aba, bab\}$ and strings in $(ab)^+a$. The features aba and bab overlap with themselves and with each other. This feature set is therefore strongly non-confluent. In fact, it has a τ number (see Page 126) of 1.0 which is the highest possible. Figure G.1 shows the parse graph and the edit tree for a string in $(ab)^+a$. Note that, in this example, every path in the parse graph yields a unique normal from and the number of unique normals forms is therefore equal to the number of paths in the parse graph. A test on the string $(ab)^{25}a$ (51 character long) yielded 476,763 unique normal forms.

Consider also the set of features $\{ab, bc\}$ and the string $s = abc^k$ for some positive integer k[1]. Note that for each occurrence of abc in s, one can either delete an ab or a bc. This means that the number of normal forms is equal to the number of strings in $\{a, c\}^k$ which is exponential in k and therefore in the length of the string.

[1]Special thanks to Prof. J. Horton

Appendix H

Kernel Selection is NP-Hard

H.1 The Kernel Selection Problem

The strings in a kernel languages are generated from a finite set of *kernels* and a finite set of *features* by inserting, anywhere, in any order, and any number of times, the features in the kernels. This suggests that if one deletes the features from a given set of strings, what remain are the original kernels. This is only true if the set of features induces a confluent string rewriting system. If the set of features is non-confluent there then might be more than one way of deleting the features. This means that each string can have many normal forms of which one must be the kernel. The kernel selection problem (KS) is the task for finding the original kernels from the sets of normal forms of each string.

Suppose we have a set T of 5 training strings labelled 1, 2, 3, 4, 5 and a set of features F. Suppose also that the normal forms (modulo F) of the string 1 are $\{a, b, c\}$ where a, b, and c are strings (not symbols). Likewise, let us suppose that the normal forms of string 2 are $\{c, b, e\}$, of string 3 $\{f, a, d\}$, of string 4 $\{c, b, d\}$, and those of string 5 are $\{d, e, a\}$. Notice that some strings share normal forms. The set of normal forms N is therefore $\{a, b, c, d, e, f\}$.

In general, when learning kernel languages, we assume that the number of kernels in the target language is much smaller than the number of training strings. A learning algorithm therefore may (depending on its inductive bias) try to find a kernel language description of the training set that minimizes the number of kernels. We therefore have to try to find a minimal set of kernels (i.e. normal forms) such that each string is reduced, modulo F, to one of these kernels. In the above example, a minimal set of kernels is $\{a, c\}$. Another is $\{b, d\}$. Every instance of the kernel selection problem is therefore an instance of the *minimum hitting set* (MHS) problem which is provably NP-Hard. This, however, is not enough to show that kernel selection is NP-Hard — only that kernel selection is a subproblem of MHS. To show that KS is NP-Hard we use polynomial-time transformation from a known NP-Hard problem — *Minimum Vertex Cover* (MVC). In other words we show that if we have a subroutine to solve the KS problem we can also use this subroutine to solve the MVC problem and if our subroutine runs in polynomial time then MVC would then also have a polynomial-time solution.

H.2 Transformation from MVC

For any instance of MVC we transform the instance to KS as follows[1]. Given a graph $G = (V, E)$ we construct our instance of KS by creating a string axb for every edge $(a, b) \in E$. The strings so formed will be our C^+ training set. For the alphabet we shall use the vertex labels and a special symbol x. We construct the features by taking all strings of the form ax and xa for any $a \in V$. Each string axb, corresponding to the edge (a, b) can then be reduced in two ways: by removing ax to obtain the normal form b or by removing xb to obtain the normal form a. The normal forms are then the vertex labels of G. The problem of kernel selection is then equivalent to that

[1]Special thanks to Prof. J. Horton

MVC since if we obtain a minimum set of normal forms that 'cover' every string we get also a minimum set of vertices that cover every edge. Clearly the transformation can be done in polynomial time.

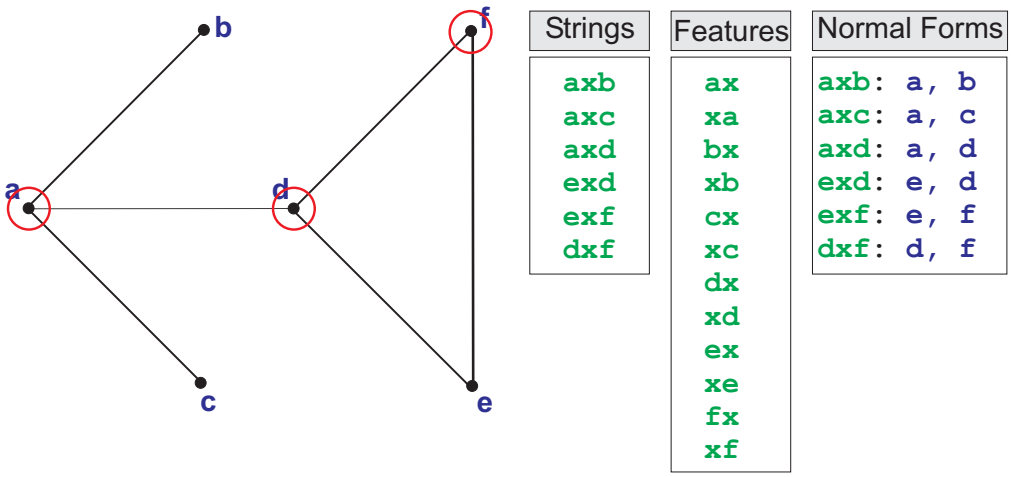


Figure H.1: How to transform *Minimum Vertex Cover* to *Kernel Selection*.

Figure H.1 shows an example of how the transformation is done. In this example $G = (V, E)$ where $V = \{a, b, c, d, e, f\}$ and $E = \{(a, b), (a, c), (a, d), (d, e), (d, f), (e, f)\}$. Note that the set of normal forms $\{a, d, e\}$ which are a minimal kernel set for the strings in the training set also represent a minimum vertex cover for G.

Appendix I

Trace of GLD Computation

Generalized Levensthein Distance or GLD is the string edit distance algorithm that was used by Nigam [92] for the GSN ETS inductive learning algorithm. In this appendix we have included a trace of the GLD algorithm as it computes the distance between the strings *aba* and *abbba*. The edit-operations transformations are shown below followed by the match matrices (see Chapter 4). The particular example is interesting because, although GLD computes the distance correctly, it shows the problem of using GLD for the description of kernel languages. If *aa*, *bb*, and *aba* are features of the kernel language that has ε, the empty string, as a kernel then *aba* belongs to the language but *abbba* does not since the latter string is obtained by inserting a feature inside another feature. Since *abbba* does not belong to the language then the distance to *aba*, which is in the language, should be greater than zero. In our case GLD returned a distance of 0 between the two strings. An analysis of the trace reveals that this happens because GLD performs what we call *deletion with concatenation*, i.e. when the feature is deleted the two remaining parts of the strings are joined together. The *EvD* distance algorithm (Chapter 3) avoids this problem by using θ-reduction.

Trace of GLD Computation

```
Operations            Weights
a                       0.5
b                       0.5
aa                      0.0
bb                      0.0
aba                     0.0

String 1 : aba
String 2 : abbba

MATCH matrix for string 1 : aba
 1 a     1   0   3   0   0   0   0   0   0   0
 2 b     0   2   0   0   0   0   0   0   0   0
 3 aa    0   0   0   0   0   0   0   0   0   0
 4 bb    0   0   0   0   0   0   0   0   0   0
 5 aba   0   0   1   0   0   0   0   0   0   0

MATCH matrix for string 2 : abbba
 1 a     1   0   0   0   5   0   0   0   0   0
 2 b     0   2   3   4   0   0   0   0   0   0
 3 aa    0   0   0   0   0   0   0   0   0   0
 4 bb    0   0   2   3   0   0   0   0   0   0
 5 aba   0   0   0   0   0   0   0   0   0   0

Computing Generalized Distance for aba and abbba

Row  1 Col  1
S2( 1)  Op( 1)=a      with Wgt= 0.50
Copying from DM( 1  0)= 0.50
Therefore W =  1.00  New v is  1.00
  S1( 1)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 0  1)= 0.50
  Therefore W =  1.00  New v is  1.00
    Op1 is  1=a        Op2 is  1=a
    Copying from DM( 0  0)= 0.00
    Therefore W =    0.00  New v is  0.00

Row  2 Col  1
S2( 2)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 1  1)= 0.00
```

```
Therefore W =  0.50  New v is  0.50
  S1( 1)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 0  2)= 1.00
  Therefore W =  1.50  New v is  0.50
    Op1 is  1=a        Op2 is  2=b
    Copying from DM( 0  1)= 0.50
    Therefore W =  100.49  New v is  0.50

Row  3 Col  1
S2( 3)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 1  2)= 0.50
Therefore W =  1.00  New v is  1.00
S2( 3)  Op( 4)=bb     with Wgt= 0.00
Copying from DM( 1  1)= 0.00
Therefore W =  0.00  New v is  0.00
  S1( 1)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 0  3)= 0.50
  Therefore W =  1.00  New v is  0.00
    Op1 is  1=a        Op2 is  2=b
    Copying from DM( 0  2)= 1.00
    Therefore W =  100.99  New v is  0.00
    Op1 is  1=a        Op2 is  4=bb
    Copying from DM( 0  1)= 0.50
    Therefore W =  100.49  New v is  0.00

Row  4 Col  1
S2( 4)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 1  3)= 0.00
Therefore W =  0.50  New v is  0.50
S2( 4)  Op( 4)=bb     with Wgt= 0.00
Copying from DM( 1  2)= 0.50
Therefore W =  0.50  New v is  0.50
  S1( 1)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 0  4)= 1.00
  Therefore W =  1.50  New v is  0.50
    Op1 is  1=a        Op2 is  2=b
    Copying from DM( 0  3)= 0.50
    Therefore W =  100.49  New v is  0.50
    Op1 is  1=a        Op2 is  4=bb
    Copying from DM( 0  2)= 1.00
    Therefore W =  100.99  New v is  0.50
```

```
Row  5 Col  1
S2( 5)  Op( 1)=a      with Wgt= 0.50
Copying from DM( 1  4)= 0.50
Therefore W =  1.00  New v is  1.00
  S1( 1)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 0  5)= 1.50
  Therefore W =  2.00  New v is  1.00
    Op1 is  1=a        Op2 is  1=a
    Copying from DM( 0  4)= 1.00
    Therefore W =    1.00  New v is  1.00

Row  1 Col  2
S2( 1)  Op( 1)=a      with Wgt= 0.50
Copying from DM( 2  0)= 1.00
Therefore W =  1.50  New v is  1.50
  S1( 2)  Op( 2)=b      with Wgt= 0.50
  Copying from DM( 1  1)= 0.00
  Therefore W =  0.50  New v is  0.50
    Op1 is  2=b        Op2 is  1=a
    Copying from DM( 1  0)= 0.50
    Therefore W =  100.49  New v is  0.50

Row  2 Col  2
S2( 2)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 2  1)= 0.50
Therefore W =  1.00  New v is  1.00
  S1( 2)  Op( 2)=b      with Wgt= 0.50
  Copying from DM( 1  2)= 0.50
  Therefore W =  1.00  New v is  1.00
    Op1 is  2=b        Op2 is  2=b
    Copying from DM( 1  1)= 0.00
    Therefore W =    0.00  New v is  0.00

Row  3 Col  2
S2( 3)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 2  2)= 0.00
Therefore W =  0.50  New v is  0.50
S2( 3)  Op( 4)=bb     with Wgt= 0.00
Copying from DM( 2  1)= 0.50
Therefore W =  0.50  New v is  0.50
  S1( 2)  Op( 2)=b      with Wgt= 0.50
  Copying from DM( 1  3)= 0.00
```

```
  Therefore W =  0.50  New v is  0.50
    Op1 is  2=b        Op2 is  2=b
    Copying from DM( 1  2)= 0.50
    Therefore W =    0.50  New v is  0.50
    Op1 is  2=b        Op2 is  4=bb
    Copying from DM( 1  1)= 0.00
    Therefore W =   99.99  New v is  0.50

Row  4 Col  2
S2( 4)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 2  3)= 0.50
Therefore W =  1.00  New v is  1.00
S2( 4)  Op( 4)=bb     with Wgt= 0.00
Copying from DM( 2  2)= 0.00
Therefore W =  0.00  New v is  0.00
  S1( 2)  Op( 2)=b      with Wgt= 0.50
  Copying from DM( 1  4)= 0.50
  Therefore W =  1.00  New v is  0.00
    Op1 is  2=b        Op2 is  2=b
    Copying from DM( 1  3)= 0.00
    Therefore W =    0.00  New v is  0.00
    Op1 is  2=b        Op2 is  4=bb
    Copying from DM( 1  2)= 0.50
    Therefore W =  100.49  New v is  0.00

Row  5 Col  2
S2( 5)  Op( 1)=a      with Wgt= 0.50
Copying from DM( 2  4)= 0.00
Therefore W =  0.50  New v is  0.50
  S1( 2)  Op( 2)=b      with Wgt= 0.50
  Copying from DM( 1  5)= 1.00
  Therefore W =  1.50  New v is  0.50
    Op1 is  2=b        Op2 is  1=a
    Copying from DM( 1  4)= 0.50
    Therefore W =  100.49  New v is  0.50

Row  1 Col  3
S2( 1)  Op( 1)=a      with Wgt= 0.50
Copying from DM( 3  0)= 0.00
Therefore W =  0.50  New v is  0.50
  S1( 3)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 2  1)= 0.50
```

```
  Therefore W =  1.00  New v is  0.50
  S1( 3)  Op( 5)=aba    with Wgt= 0.00
  Copying from DM( 0  1)= 0.50
  Therefore W =  0.50  New v is  0.50
    Op1 is  1=a        Op2 is  1=a
    Copying from DM( 2  0)= 1.00
    Therefore W =    1.00  New v is  0.50
    Op1 is  5=aba      Op2 is  1=a
    Copying from DM( 0  0)= 0.00
    Therefore W =   99.99  New v is  0.50

Row  2 Col  3
S2( 2)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 3  1)= 0.50
Therefore W =  1.00  New v is  1.00
  S1( 3)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 2  2)= 0.00
  Therefore W =  0.50  New v is  0.50
  S1( 3)  Op( 5)=aba    with Wgt= 0.00
  Copying from DM( 0  2)= 1.00
  Therefore W =  1.00  New v is  0.50
    Op1 is  1=a        Op2 is  2=b
    Copying from DM( 2  1)= 0.50
    Therefore W =  100.49  New v is  0.50
    Op1 is  5=aba      Op2 is  2=b
    Copying from DM( 0  1)= 0.50
    Therefore W =  100.49  New v is  0.50

Row  3 Col  3
S2( 3)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 3  2)= 0.50
Therefore W =  1.00  New v is  1.00
S2( 3)  Op( 4)=bb     with Wgt= 0.00
Copying from DM( 3  1)= 0.50
Therefore W =  0.50  New v is  0.50
  S1( 3)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 2  3)= 0.50
  Therefore W =  1.00  New v is  0.50
  S1( 3)  Op( 5)=aba    with Wgt= 0.00
  Copying from DM( 0  3)= 0.50
  Therefore W =  0.50  New v is  0.50
    Op1 is  1=a        Op2 is  2=b
```

```
    Copying from DM( 2  2)= 0.00
    Therefore W =   99.99  New v is  0.50
    Op1 is  1=a        Op2 is  4=bb
    Copying from DM( 2  1)= 0.50
    Therefore W =  100.49  New v is  0.50
    Op1 is  5=aba      Op2 is  2=b
    Copying from DM( 0  2)= 1.00
    Therefore W =  100.99  New v is  0.50
    Op1 is  5=aba      Op2 is  4=bb
    Copying from DM( 0  1)= 0.50
    Therefore W =  100.49  New v is  0.50

Row  4 Col  3
S2( 4)  Op( 2)=b      with Wgt= 0.50
Copying from DM( 3  3)= 0.50
Therefore W =  1.00  New v is  1.00
S2( 4)  Op( 4)=bb     with Wgt= 0.00
Copying from DM( 3  2)= 0.50
Therefore W =  0.50  New v is  0.50
  S1( 3)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 2  4)= 0.00
  Therefore W =  0.50  New v is  0.50
  S1( 3)  Op( 5)=aba    with Wgt= 0.00
  Copying from DM( 0  4)= 1.00
  Therefore W =  1.00  New v is  0.50
    Op1 is  1=a        Op2 is  2=b
    Copying from DM( 2  3)= 0.50
    Therefore W =  100.49  New v is  0.50
    Op1 is  1=a        Op2 is  4=bb
    Copying from DM( 2  2)= 0.00
    Therefore W =   99.99  New v is  0.50
    Op1 is  5=aba      Op2 is  2=b
    Copying from DM( 0  3)= 0.50
    Therefore W =  100.49  New v is  0.50
    Op1 is  5=aba      Op2 is  4=bb
    Copying from DM( 0  2)= 1.00
    Therefore W =  100.99  New v is  0.50

Row  5 Col  3
S2( 5)  Op( 1)=a      with Wgt= 0.50
Copying from DM( 3  4)= 0.50
Therefore W =  1.00  New v is  1.00
```

```
  S1( 3)  Op( 1)=a      with Wgt= 0.50
  Copying from DM( 2  5)= 0.50
  Therefore W =  1.00  New v is  1.00
  S1( 3)  Op( 5)=aba    with Wgt= 0.00
  Copying from DM( 0  5)= 1.50
  Therefore W =  1.50  New v is  1.00
    Op1 is  1=a        Op2 is  1=a
    Copying from DM( 2  4)= 0.00
    Therefore W =    0.00  New v is  0.00
    Op1 is  5=aba      Op2 is  1=a
    Copying from DM( 0  4)= 1.00
    Therefore W =  100.99  New v is  0.00

End of Computation
```

Final distance matrix is:

	ε	**a**	**b**	**b**	**b**	**a**
ε	**0.00**	0.50	1.00	0.50	1.00	1.50
a	0.50	**0.00**	0.50	0.00	0.50	1.00
b	1.00	0.50	**0.00**	0.50	0.00	0.50
a	0.00	0.50	0.50	0.50	0.50	**0.00**

Table I.1: Distance matrix after GLD computation of *aba* and *abbba*.

Table I.1 is the distance matrix after GLD computation is completed. The edit sequences that yields the minimum cost of transforming *aba* into *abbba* is shown in red.

Vita

John M. Abela jabel@cs.um.edu.mt

3, Kaless Street,
Sta. Venera,
HMR 16,
Malta
Date of Birth: 13^{th} November, 1961.

Academic Qualifications:

B.Sc. (Mathematics and Computer Science), University of Malta, 1991.
M.Sc. (Computer Science), University of New Brunswick, 1994.
Ph.D. program, Faculty of Computer Science, UNB, 1996-2001.

Recent Publications:

Abela, John, *Learning Picture Languages*, Technical Report TR-CS-9605, Department of Computer Science and Artificial Intelligence, University of Malta, **1996**.

Goldfarb, L., Abela, J., Bhavsar, V. C., Kamat, V. N., *Are Vector-Space Models Capable of Inductive Learning in a Symbolic Environment?*, In Proceedings of the 10^{th} Biennial Conference of the Computer Society of the Computational Study of Intelligence, Morgan Kauffman Pub., Palo Alto, Ca., **1995**.

Goldfarb, L., Abela, J., Bhavsar, V. C., Kamat, V. N., *Can a Vector-Space Based Learning Model Discover Inductive Class Generalization in a Symbolic Environment?*, Pattern Recognition Letters, 16, pp. 719-726, **1995**.

Working Experience:

Employed since 1994 as Assistant Lecturer with the Department of Computer Science and A.I., Faculty of Science and I.T., University of Malta, Valletta, Malta. Prior to that the author operated his own I.T. consultancy and software development business.