# Learning DFAs from Sparse Data 



## Kristian Guillaumier

Supervisor: Dr. John Abela

Department of Artificial Intelligence, Faculty of ICT
University of Malta

This dissertation is submitted for the degree of Doctor of Philosophy


L-Università ta' Malta

## University of Malta Library - Electronic Thesis \& Dissertations (ETD) Repository

The copyright of this thesis/dissertation belongs to the author. The author's rights in respect of this work are as defined by the Copyright Act (Chapter 415) of the Laws of Malta or as modified by any successive legislation.

Users may access this full-text thesis/dissertation and can make use of the information contained in accordance with the Copyright Act provided that the author must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the prior permission of the copyright holder.

## Declaration of Authentication

Student Name and ID: Kristian Guillaumier, 69478M<br>Course: Doctor of Philosopy<br>Title of Thesis: Learning DFAs from Sparse Data

I hereby declare that I am the legitimate author of this thesis and that it is my original work. Except where specific reference is made to the work of others, the contents of this dissertation are original. No portion of this work has been submitted in support of an application for another degree or qualification of this or any other university or institution of higher education. I hold the University of Malta harmless against any third party claims with regard to copyright violation, breach of confidentiality, defamation, and any other third party right infringement.

As a Ph.D. student, as per Regulation 49 of the Doctor of Philosophy Regulations, I accept that my thesis be made publicly available on the University of Malta Institutional Repository. This dissertation contains 58,513 words excluding appendices.

29th January 2020

## Acknowledgements

Working on a Ph.D. involves quite a bit of work, and I have been fortunate enough to have the support of my family and friends all along the way.

My sincerest gratitude goes to my supervisor and friend, Dr. John Abela, who was always there when the seas got too deep. I cannot possibly count all the Saturday afternoons he set aside for me to meet and discuss new ideas, and help me move forward. I am forever thankful for his immense dedication.

Towards the end of my work, I suffered from a rather unfortunate health problem which prevented me from functioning on top of my game. During this ordeal, I found the support of Prof. Alexiei Dingli, my Head of Department at the time, as well as Ms. Jacqueline Fenech at Human Resources whose understanding helped make my work possible.

It is no exaggeration to say that none of this would have been possible without the support of my wife, Rita, who shared this trying experience (and my incessant mood swings) without even a hint of complaint. Thank you for your grace, patience, and for believing - there is truly no place like home. My heart belongs to my son, Jamie, who never fails to put a smile on my face. Thank you for understanding while daddy was 'writing his book', and for distracting me when I needed it most. You are the overwhelming pride of my life.

Kindness and a good heart go a long way. This I learnt from my parents to whom this thesis is dedicated. I know that you have been waiting for this for a long, long time, and that it means to you as much as it means to me. It is done now. I love you for everything.


#### Abstract

Regular inference is the task of identifying a Deterministic Finite-State Automaton (DFA) from a training set of positive and negative strings of a regular language. This problem is known to become increasingly more difficult as the size of the target DFA grows larger and as the training data becomes sparser. One of the most studied algorithms for this task is the Evidence-Driven State Merging (EDSM) algorithm due to Rodney Price which emerged as a winning algorithm in the international Abbadingo One competition of 1997.

We focus on 'Abbadingo-style' problems (learning DFAs over binary alphabets with training sets of various densities), and we present the results of a comprehensive analysis of how, and more importantly when, EDSM succeeds or fails in identifying either the exact target DFA, or a low-error hypothesis with respect to a test set. We do this by creating thousands of problem instances to study their characteristics, as well as the behaviour of the state merging process. To support this analysis, we have developed an extensive software framework consisting of highly optimised, and parallelised, implementations of state merging algorithms, as well as several visual and statistical analysis tools.

Motivated by the results and insights we obtained following this analysis, we propose three methods each having the aim of improving on the generalisation rate of EDSM on Abbadingo-style problems when the training data is sparse. Our first method involves the development of an ensemble of monotonic, greedy heuristics which, together, are able to outperform EDSM. This method is inspired by Wolpert and Macready's No Free Lunch theorems which roughly state that, in general, no single heuristic will dominate all others over all problem instances. This is indeed supported by the empirical evidence we have gathered during our experimentation on various problem configurations. Associated with the ensemble of heuristics, we have also identified a method which enables us to predict, with a high degree of confidence, which of the individual heuristics in the ensemble results in a low or zero-error hypothesis.


Our second approach, which we call the Delta Graph method, is based on the observation that when a greedy heuristic selects sequences of merges, the initial ones are especially critical. When a wrong choice is made, finding the target DFA becomes impossible and the likelihood of finding a low-error hypothesis will be greatly reduced. This method involves constructing and non-monotonically searching in a structure representing a highly condensed subspace of possible merges. This structure contains several short sequences of merges, where, with high experimental probability, at least one of them will consist of correct merges leading to the target. These merges establish enough constraints on a partial hypothesis that, when extended with a label-matching heuristic, will lead to the target DFA or a low-error hypothesis.

Typical evolutionary approaches in DFA learning operate by attempting to evolve a target DFA either as a transition matrix or by partitioning the states of a Prefix Tree Acceptor (PTA). In our third method, we present an alternative approach which, instead, evolves short sequences of merges selected from a subset of high state-reduction merges. As in the Delta Graph method, these short sequences of merges establish enough constraints on a hypothesis, that when extended with a label-matching heuristic, will, with high experimental probability, lead to the target DFA or a low-error hypothesis.

To ensure a common baseline for comparison, our methods are evaluated on target DFAs and training sets which have been constructed according to the Abbadingo One competition procedures. Our results show that each of the methods we have developed outperforms EDSM. For example, on 64 -state target DFA problems and symmetrically structurally complete training sets at the sparsest density set by the Abbadingo One competition, while EDSM identifies low-error DFAs approximately $15 \%$ of the time, our ensemble, Delta Graph, and evolutionary methods do so about $26 \%, 43 \%$, and $56 \%$ of the time respectively. We also obtain considerably better generalisation rates on problem instances which are highly adversarial to EDSM.

## Contents

1 Introduction ..... 1
1.1 A Simple Example ..... 3
1.2 Grammatical Inference Competitions ..... 5
1.2.1 The Abbadingo One DFA Learning Competition ..... 6
1.3 Motivation and Research Hypotheses ..... 7
1.4 Overview of Results and Contributions ..... 11
1.5 Organisation of this Document ..... 18
2 Preliminaries ..... 21
2.1 Sets ..... 21
2.2 Strings and Languages ..... 23
2.2.1 Ordering of Strings ..... 24
2.3 Formal Languages ..... 25
2.4 Finite-State Automata ..... 29
2.4.1 Non-Deterministic Finite-State Automata ..... 30
2.4.2 Deterministic Finite-State Automata ..... 34
2.4.3 The Depth of a DFA ..... 34
2.4.4 Canonical Automata ..... 35
2.4.5 DFA Minimisation ..... 35
2.5 Summary ..... 44
3 Characterising the Search Space of DFA Learning ..... 45
3.1 Introductory Concepts ..... 46
3.1.1 Quotient Automata ..... 46
3.1.2 Structural Completeness and Characteristic Sets ..... 46
3.1.3 The Maximal Canonical Automaton ..... 48
3.1.4 The Prefix Tree Acceptor ..... 50
3.1.5 The Augmented Prefix Tree Acceptor ..... 51
3.2 Derived Automata ..... 52
3.3 Some Important Results ..... 52
3.4 The Border Set and its Properties ..... 58
3.5 Summary ..... 62
4 A Survey of State Space Search Algorithms ..... 64
4.1 Preliminaries ..... 65
4.1.1 Costs ..... 65
4.1.2 Time and Space Complexity, Completeness, and Admissibility ..... 65
4.1.3 Open and Closed Lists ..... 66
4.2 Blind Search Algorithms ..... 67
4.2.1 Depth-First Search ..... 67
4.2.2 Breadth-First Search ..... 68
4.2.3 Iterative Deepening ..... 68
4.2.4 Uniform-Cost Search ..... 70
4.3 Informed Search Algorithms ..... 71
4.3.1 Greedy Search ..... 71
4.3.2 A* Search ..... 72
4.3.3 Iterative Deepening A* Search ..... 74
4.3.4 Branch and Bound Techniques ..... 75
4.3.5 Beam Search ..... 76
4.4 Observations So Far ..... 78
4.5 Estimating the Cost of Backtracking ..... 78
4.5.1 Pitfalls and Refinements ..... 81
4.5.2 Ideas for Improvement ..... 82
4.6 Summary ..... 86
5 DFA Learning Algorithms ..... 87
5.1 State Merging Algorithms ..... 88
5.1.1 A Simple State Merging Example ..... 88
5.1.2 Merge Paths and Graphs ..... 90
5.1.3 The State Merging Operation ..... 90
5.1.4 Trakhtenbrot-Bardzin's and Gold's Algorithm ..... 95
5.1.5 Regular Positive and Negative Inference ..... 96
5.1.6 Price's Evidence Driven State Merging ..... 102
5.1.7 Variants of EDSM ..... 112
5.1.8 DFA Learning using Minimum Description Length ..... 118
5.1.9 Parallel Beam Search (PBS, SAGE) ..... 119
5.1.10 Ed-Beam ..... 122
5.1.11 TBW-EDSM ..... 123
5.1.12 Some Remarks Regarding 'Search Wrappers' ..... 124
5.2 Other DFA Learning Methods ..... 124
5.2.1 Genetic Search and Swarm Intelligence ..... 124
5.2.2 DFA Learning as Constraint Satisfaction ..... 128
5.2.3 Graph (Vertex) Colouring ..... 129
5.2.4 $\quad$ Satisfiability Solvers ..... 130
5.2.5 Connectionist Approaches ..... 133
5.3 Summary ..... 134
6 Properties of State Merging ..... 135
6 6.1 DFAs are Partitions, States are Blocks ..... 135
6.2 General Properties ..... 137
6.2.1 The Initial Partition ..... 137
6 6.2.2 Properties of the Join and Merge Operations ..... 137
6.2.3 Permitting, Blocking, Included, and Root Merges ..... 142
6.2.4 Orphaned States ..... 148
6.3 Properties of Merge Paths ..... 150
6.4 Some Remarks Regarding EDSM ..... 158
6.5 Colour-Compatible Merges ..... 162
6.6 Ending Remarks ..... 166
7 Baseline Experiments and Methodology ..... 170
7.1 How Heuristics or DFA Learning Algorithms are Evaluated andAnalysed172
7.2 Oracle-Assisted Heuristics and Paths ..... 175
7.3 Glossary of Heuristics ..... 176
7.4 The Abbadingo One Setup ..... 177
7.4.1 Creating Target DFAs ..... 177
7.4.2 Creating Training Sets ..... 178
7.4.3 Creating Problem Instances ..... 180
7.5 Baseline Experiments ..... 181
7.5.1 Expected APTA Sizes and Merges ..... 181
7.5.2 Presentation of Results ..... 183
7.5.3 Baseline EDSM and W-EDSM Performance ..... 184
7.5.4 Analysis of Oracle-Assisted Heuristics ..... 184
7.5.5 Non-Structurally Complete Training Sets ..... 186
7.5.6 Getting the First Merges Right ..... 186
7.5.7 Reduction Curves and DFA Compression ..... 188
7.5.8 The APTA Reduction Table ..... 190
7.5.9 Overlap Between Ideal Merge Paths and Merges in the APTA Reduction Table ..... 193
7.5.10 Colour-Compatible Merge Positions ..... 195
7.5.11 Merge Path Lengths ..... 197
7.5.12 Summary of Results ..... 198
7.6 Methodology ..... 199
7.6.1 An Ensemble of Heuristics ..... 199
7.6.2 The Delta Graph ..... 204
7.6.3 Evolving Initial Merge Sequences ..... 210
7.7 Summary ..... 215
8 Evaluation and Results ..... 218
8.1 Overview of the Experimental Setup ..... 220
8.1.1 Adversarial Setups ..... 221
8.2 Evaluating the Ensemble of Heuristics ..... 222
8.2.1 Adversarial Setups ..... 224
8.2.2 Performance on Unrestricted Problem Instances ..... 226
8.2.3 Observations and Discussion ..... 227
8.3 Evaluating the Delta Graph ..... 233
8.3.1 Adversarial Setups ..... 237
8.3.2 Discussion and Observations ..... 238
8.4 Evaluating the Genetic Algorithm ..... 244
8.4.1 Adversarial Setups ..... 248
8.4.2 Discussion and Observations ..... 249
8.5 Ending Remarks ..... 254
9 Conclusions and Future Work ..... 259
9.1 Achievements and Contributions ..... 260
9.1.1 Summary of Results ..... 261
9.1.2 An Ensemble of Heuristics ..... 263
9.1.3 Adversarial Problem Instances ..... 263
9.1.4 Label-Matching Heuristics ..... 264
9.1.5 Colour-Compatible Merges ..... 264
9.1.6 The APTA Reduction Table ..... 264
9.1.7 Getting the First Merges Right ..... 265
9.1.8 The Genetic Algorithm ..... 265
9.2 Future Work ..... 266
9.3 Concluding Remarks ..... 270
A Complete Results ..... 285
A. 1 Glossary of Heuristics ..... 286
A. 2 32-State Target DFA Instances ..... 289
A.2.1 n32d607e1024.sqlite ..... 289
A.2.2 n32d607e128_GA.sqlite ..... 290
A.2.3 n32d607e512_EdsmFailing.sqlite ..... 290
A.2.4 n32d607e64_EdsmFailing.sqlite ..... 291
A.2.5 n32d607e512_NotStructComp.sqlite ..... 293
A.2.6 n32d607e64_NotStructComp.sqlite ..... 293
A.2.7 n32d607e512_NoLoops.sqlite ..... 294
A.2.8 n32d607e64_NoLoops.sqlite ..... 295
A. 3 64-State Target DFA Instances ..... 296
A.3.1 n64d1e1024.sqlite ..... 296
A.3.2 n64d1e64_GA.sqlite ..... 298
A.3.3 n64d1e512_Unrestricted.sqlite ..... 298
A. 4 128-State Target DFA Instances ..... 300
A.4.1 n128d1e512.sqlite ..... 300
B State Reduction Rates ..... 301
C State Space Search Algorithms ..... 304
D Contents of the Submitted Media ..... 310

## List of Abbreviations

APTA Augmented Prefix Tree Acceptor - a tree constructed from both the positive and negative examples in the training set. The tree corresponds to a DFA containing accepting, rejecting, and unlabelled states. 51

AV-K Aldous-Vazirani/Knuth - a method due to Cloteaux and Valentin that combines Knuth-estimation with the GWTW algorithm to estimate properties of search trees. 83

BFS Breadth-First Search - a tree or graph search algorithm. 68
BnB Branch and Bound - a technique to solve optimisation problems. 75

BS Border Set - a set of automata in a lattice or space of automata which cannot be generalised further. 60

DFA Deterministic Finite-State Automaton. See also FSA. 34

DFS Depth-First Search - a tree or graph search algorithm. 67

EDSM Evidence-Driven State Merging - a state merging heuristic developed by Rodney Price. 7

FSA A Finite-State Automaton - an abstract machine used to recognise or generate strings in a language. 29

FSM See FSA. 29

GA Genetic Algorithm - a metaheuristic for solving optimisation problems inspired by natural selection. 125,210

GWTW Go With the Winners - is a method due to Aldous and Vazirani that requires a polynomial number of samples to estimate the deepest leaf node in a tree. 83

IDA* Iterative Deepening A* Search - a tree or graph search algorithm. 74

IDS Iterative Deepening Search - a tree or graph search algorithm. 68

LAT Lattice - a lattice of automata constructed from an MCA. 52

MCA Maximal Canonical Automaton - corresponds to the automaton having the largest number of useful states to recognise the strings in the training set it has been constructed for. An MCA is typically non-deterministic. 48

NFA Non-Deterministic Finite-State Automaton. See also FSA. 30

NFL No Free Lunch - the no free lunch theorems for optimisation by Wolpert and Macready. 199

PBS Parallel Beam Search - technique used by Juillé for DFA learning. 7

PTA Prefix Tree Acceptor - a tree constructed from only the positive examples in the training set. The tree corresponds to a DFA containing accepting and unlabelled states. 50

RPNI Regular Positive and Negative Inference - a state merging algorithm due to Oncina and Garcia. 96

S-EDSM Shared Evidence-Driven State Merging - a state merging heuristic due to Spina. 113

SAGE Self-Adaptive Greedy Estimate - a parallel beam search algorithm due to Juillé and Pollack. 119

SAT Satisfiability - depending on the context, refers to the Boolean satisfiability problem, or the property of a formula being satisfiable. 2

SCD Smallest Consistent DFA - this a decision problem of whether there exists a DFA having at most $n$ states which recognises the positive strings and rejects the negative strings in a given training set. 2

UCS Uniform-Cost Search - a tree or graph search algorithm. 70

W-EDSM Windowed-EDSM - uses a windowing technique to improve the runtime performance of EDSM. 112

## Chapter 1

## Introduction

Several real-world objects can be represented as sequences of symbols. For instance, a musical composition can be thought of as a sequence of notes, and DNA sequences are strings over the alphabet A, C, G, and T. Likewise, English sentences are sequences of characters over an alphabet or words in a dictionary. Unfortunately, in many cases we do not have a formal description of how these sequences should be put together, but we instead have several examples of correctly constructed instances. For example, while we might not have an exact definition of what makes baroque music what it is, we have large collections of musical compositions in that style. These ideas lead to the concept of grammatical inferenct $\rrbracket^{1}$ which is the task of learning a formal grammar, or class description, from examples of strings which belong and do not belong to the language that it specifies.

In this dissertation, we are specifically interested in regular inference where we are concerned with identifying deterministic finite-state automata (DFAs) over binary alphabets from training sets consisting of both positive and negative strings. To concretise this idea with an example, suppose that we are told that the strings $a b, a a b, a a a b$, and $a a a a b$ all belong to some unknown regular language $L$, and the strings $\lambda, a, b, a a, b b$, and $a a a$ do not belong to it. After examining this information, we may reasonably deduce that the regular language is $L=a a^{*} b$ while noting that there are, in fact, infinitely many languages that are consistent with

[^0]our training data ${ }^{2}$. Nonetheless, for the time being, we will unburden ourselves with this detail and just assume that the unknown language is vaguely defined as the most 'straightforward' one.

At the risk of stating the obvious, the task of finding an automaton that is consistent with a training set is usually trivial. All that needs to be done is to construct one that exactly encapsulates the training set, and thereby learning it by rote3. In the context of learning, however, such a result is of little use - the automaton would be perfectly capable of correctly classifying all the strings in the training set but would be unable to generalise to unseen examples. Using Occam's principle, a more interesting and useful problem is that of finding an automaton which not only is consistent with the training data but also has the minimum number of states. For typical problem instances, these minimum state automata would necessarily contain cycles which would recognise infinite languages that correspond to more general hypotheses. Unfortunately, this task is NPcomplete [Gol78, Ang78]. Reformulated as a decision problem, the question is: given positive and negative sets $S_{+}$and $S_{-}$respectively, does there exist a DFA with at most $n$ states such that $S_{+}$is in the language and $S_{-}$is not? This decision problem is generally referred to as the smallest consistent DFA problem (SCD) dlH10].

Showing that our problem is NP-complete involves (i) showing that the problem is in the class NP, and (ii) that any NP-hard problem can be transformed into our problem in polynomial time Showing that the the problem is in NP follows from the fact that a set of strings $S_{+}$and $S_{-}$can be checked for consistency with an automaton having at most $n$ states in polynomial time. The next step in showing NP-completeness involves reducing the Boolean satisfiability (SAT) problem to our decision problem. This is indeed possible, and the reader is referred to dlH10 for this reduction. We also suggest [Gol78], Ang78], and [Pit89] for a comprehensive study on the asymptotic complexity of the regular inference problem.

[^1]
### 1.1 A Simple Example

In this section we examine the basic method of how many algorithms operate without introducing any complex notation or subtleties. The idea can be summarised as follows:

- There is a target regular language $L$, specified by a DFA, which is unknown to us.
- The language $L$ is defined over a binary alphabet $\Sigma=\{a, b\}$.
- The only information we are given about the language $L$ is a finite set of strings that belong to it (positive training examples) and a finite set of strings that do not (negative training examples). Together, these are referred to as the training set.
- A prefix tree acceptor (an acyclic DFA, see Section 3.1.4) corresponding to the most specific hypothesis is constructed from this training set.
- A pair of states is selected from this hypothesis (usually using a greedy heuristic) and those states are merged together according to certain criteria and constraints. This will create a new hypothesis that is more compact and, typically, introduces cycles which make it more general than the one we started with. The criteria by which a merge is performed, ensure that the resulting hypothesis remains consistent with the training data (it does not mislabel training strings). If the merge results in an inconsistent hypothesis, we say that the merge is invalid, discard it, and choose a different one.
- This non-backtracking process of pair selection and merging will be repeated until there are no more valid merges possible. At this point, we have constructed a final hypothesis which is, hopefully, either the target DFA or one which has a low-error with respect to some test set.

We illustrate the process with the example shown in Figure 1.1. Starting from (i) we have a very specific hypothesis that recognises a finite language, and out of all the possible states that may be merged, we select the pair $q_{1}$ and $q_{3}$. Merging them gives us DFA (ii) which is now more compact, more general, and
also recognises an infinite language due to the introduction of cycles. Furthermore, and critical to our task, all the strings accepted by DFA (i) are also accepted by that in (ii). At every step we ensure that our new hypothesis is consistent with the training data, and continue until no further merge is possible. We hope that the final hypothesis is the target DFA we are looking for.


Figure 1.1: A very simple state merging example.

### 1.2 Grammatical Inference Competitions

Learning DFAs from training data has been an active research area for over four decades. During this time, a number of challenges were proposed by the academia with the intention of promoting the development of state-of-the-art algorithms, and to encourage new research in the area.

In 1997, the Abbadingo One DFA learning competition was made public by Barak Pearlmutter and Kevin Lang Abb97, LPP98. This consisted of a set of benchmark problems (of varying difficulty) for researchers to develop and test new algorithms with. The two winning algorithms by Rodney Price and Hugues Juillé represented a significant improvement over existing method $\xi^{5}$. Soon after Abbadingo One, the Gowachin learning competition was launched in 1998 by Kevin Lang, Barak Pearlmutter, and François Coste [Gow98]. The primary difference between Gowachin and Abbadingo One was that Gowachin allowed researchers to generate their own problem instances in terms of target DFA and training set size, as well as to introduce noise in the training data. In 2004, the GECCO competition was organised by Simon Lucas to learn DFAs from noisy samples [GEC04]. David Combe, Colin de la Higuera, and Jean-Christophe Janodet organised the Zulu competition for learning DFAs from membership queries CdIHJ10. The aim of the competition was to develop methods which are able to obtain the best classification rates from a fixed number of queries to an Oracle. The StaMinA competition for learning DFAs with large alphabets was published in 2010 by Walkinshaw et al., where the hardness of each benchmark problem was determined by the sparsity of the training data as well as the alphabet size $\mathrm{WBD}^{+} 10$. The winning algorithm by Marijn Heule and Sicco Verwer represented a significant improvement over the state-of-the-art on the challenging setting of large alphabets HV13.

The grammatical inference community has also organised other competitions focusing on other types and classes of problems. In 2004, the Omphalos competition was organised by Brad Starkie, François Coste, and Menno van Zaanen to learn context-free languages. Tenjinno was organised by Bradford Starkie, Menno Zaanen, and Dominique Estival which was held in conjunction with the

[^2]International Colloquium on Grammatical Inference (ICGI) in 2006 [SZE06]. The primary aim of the competition was to promote the development of grammatical inference algorithms for use in machine translation systems. The Sequence Prediction Challenge, or SPiCe, was a competition organised in 2006 for guessing the next symbol in a sequence [SPI16, $\mathrm{BEL}^{+} 17$. Specifically, the challenge involved 15 problems made up of a combination of synthetic and real-world data, where the aim is to predict the five most likely next symbols in a prefix. Sicco Verwer, Rémi Eyraud, and Colin de la Higuera organised the Probabilistic Automata Learning Competition (PAutomaC) in 2012. This was the first online challenge for learning non-deterministic probabilistic finite state machines from artificial data generated using either Hidden Markov Chains or probabilistic automata [PAu12, VEdIH14].

### 1.2.1 The Abbadingo One DFA Learning Competition

In this dissertation, we are interested in learning DFAs over binary alphabets from sparse training data. As such, the procedure we will use to generate problem instances for experimentation and analysis will follow that specified in the Abbadingo One competition (the Gowachin challenge also used this procedure).

The Abbadingo One competition presented the experimental setup for sixteen benchmark problems where the task is to infer a target DFA from a given training set. A learner would be considered to have solved one of the problems if it is able to identify a DFA which performs with an error rate of no more than $1 \%$ on a test set ${ }^{6}$. This test set is, of course, kept secret - our hypothesis is sent to an 'Oracle' which simply returns a pass/fail answer. Participants are allowed multiple submissions to the Oracle as long as they do not "flood" the system with more than 25 requests in a day [Abb97]. The difficulty of each of the sixteen classes of problems is distinguished by the size of the target DFA and the density or sparseness of the training set given. Large DFAs with sparse training sets are harder to identify than small DFAs having denser training sets [LPP98. Out of the sixteen problems, four of them were already known to be easily solvable and the competition focused on the remaining twelve harder ones. The following table

[^3]shows the winning algorithms for each of these twelve problem instances.

|  |  | Training Set Density |  |  |
| :---: | :---: | :---: | :---: | :---: |
|  | III dense | II | $\mathrm{I}_{\text {sparse }}$ |  |
| DFA Size | 64 states | Juillé-PBS | Juillé-PBS | Juillé-EDSM+Search |
|  | 128 states | Juillé-PBS | Juillé-PBS | Unsolved |
|  | 256 states | Price-EDSM | Juillé-EDSM | Unsolved |
|  | 512 states | Price-EDSM | Price-EDSM | Unsolved |

Table 1.1: Results of the Abbadingo One DFA learning competition [PP98].

Juillé-XBS is a parallel beam search technique developed by Hugues Juillé et al. JP98b, Price-EDSM is an evidence-driven greedy heuristic by Rodney Price [LPP98], Juillé-EDSM represents the case where both Juillé's PBS and Price's EDSM solved the problem, and Juillé-EDSM+Search is a method that augments Price's EDSM with Juille's search technique to perform better on sparse data sets. Both Juillé's and Price's methods will be described later on in Chapter 5.

### 1.3 Motivation and Research Hypotheses

Grammatical inference is a hard problem [Gol78, Ang78, and advances in the area are bound to contribute to both the theoretial and applied machine learning community. Moreover, the practical applications are many, with excellent surveys conducted by de la Higuera dlH05, Stevenson and Cordy [SC14], and Wieczorek Wie17. A few highlights include:

- In robotics, work has been done in map learning [RS93, Rie99].
- Grammatical inference techniques have been applied to several structural pattern recognition problems such as fingerprint identification, object recognition in industrial settings, and image contour analysis Mic86, BS90.
- Induction of automata has been applied to the task of speech recognition and generating natural language models [GSVG90, TDH00, SZE06].
- In bioinformatics, identifying patterns in DNA or protein sequences is a major area of study. Grammatical inference methods have been applied to the task of DNA sequence classification and the analysis of protein structure $\left[\mathrm{SBH}^{+} 94, \mathrm{WRS}^{+} 99, ~ \mathrm{PCD} 19\right]$.
- In data mining, we see grammar learning algorithms used to learn user behaviour from web navigational patterns [BL00.
- In software engineering, work has been done in the fields of inference of general purpose or domain-specific programming languages, and the analysis of execution traces [CRML73, ABL02, DJA08].

As recently as the StaMinA learning competition, EDSM is considered to be a state-of-the-art algorithm for identifying DFAs over binary alphabets from sparse training data HV13. As such, we will focus our attention on studying the behaviour of EDSM, as well as develop methods to improve on its generalisation rate. Our two research hypotheses are:

1. Can EDSM be combined with other monotonic, greedy heuristics to produce an ensemble that will out-perform EDSM on Abbadingo-style problem instances?

Discussion: A monotonic, greedy heuristic makes merge choices starting from an APTA until it reaches a final hypothesis which cannot be reduced further. These choices represent a single, non-backtracking path in the search space of merges, and different heuristics which reach different final hypotheses on the same problem instance, correspond to different inductive biases in the search. We conjecture that an ensemble of such heuristics is viable approach, since the size of the final hypothesis (we are searching for the minimum state DFA) can be used to choose which one of the hypotheses returned by the ensemble is most likely to be correct. Furthermore, such a method is appealing since the computational overhead is a linear function in the number of heuristics in the ensemble.
2. To what extent can non-monotonic search methods, such as 'search wrapper' techniques and genetic algorithms, be further improved over EDSM to perform better on Abbadingo-style problem instances?

Discussion: Non-monotonic search methods have been extensively investigated by the regular inference community. For example, Oliveira and Silva's BIC backtracking algorithm poses DFA learning as a constraint satisfaction problem [OMS98, Juillé and Pollack's SAGE wraps a stochastic search around a state merging algorithm JP98b, and Lang's Ed-Beam wraps a deterministic search around EDSM [Lan99]. As a result of both Cicchello's work [Cic02], as well as our preliminary experiments, we know that the initial merge choices made are especially critical since they constrain a hypothesis in ways which will either considerably increase the likelihood of finding a good solution, or make it much less likely (if not impossible). To investigate this research hypothesis, we will study the characteristics of merges in the entire space of merges to determine whether we can identify a subspace of high-quality merges in which we can search for short sequences of such high-quality merges. These sequences would then be extended with a labelmatching heuristic, such as EDSM, to obtain a final hypothesis.

Our work can be further broken down into the following sub-tasks:

1. Replicate the Abbadingo One procedure for creating target DFA instances and training sets. Our procedure needs to be validated against published results to ensure that we are comparing like with like.
2. In order to perform a comprehensive analysis of state merging algorithms and EDSM, we need to experiment on a large number of problem instances of various configurations. To support this, we need to develop highly optimised implementations of these algorithms, as well as a suite of statistical analysis and visualisation tools.
3. Implement the EDSM algorithm to establish a baseline for its generalisation rate. Using this baseline, we can compare the behaviour and performance of our methods.
4. Thoroughly investigate current state-merging techniques in DFA learning with an emphasis on the EDSM heuristic. Under which condition does EDSM work? When and why does it fail?
5. Study the search space of DFA learning in order to better understand how the state merging process works.
6. Identify whether and which characteristics of the target DFAs and training data affect the outcomes of a learning algorithm. For instance, to which degree do training sets which are not structurally complete (see Section 3.1.2) negatively affect the generalisation rate of EDSM?
7. Whenever a state merging algorithm is converging to the target DFA (or a close approximation of it), the number of states in the current hypothesis is being reduced at each step. Is this rate of reduction consistent over different problem instances? Can this rate of reduction be used to help prune the number of state pairs that need to be considered at every step of the merging process?
8. When EDSM is scoring merges for selection, several merges may be tied with the same score, and a tie-breaking policy is required. If an 'Oracle' is used to help break ties optimally, can we establish an experimental upper bound on the performance of EDSM?
9. Every merge made, typically, assigns a labelling to a previously unlabelled state in the hypothesis thereby establishing constraints on future merges. It follows that selecting 'good' merges (an unlabelled state has been labelled properly) during the initial steps of the merging process will increase the likelihood of identifying the exact target DFA or a low-error hypothesis. To which extent are these initial merges in a sequence important?
10. Is there a relationship between the length of a merge path (the sequence of merges starting from the initial to the final hypothesis) and the likelihood of identifying either the target DFA or a low-error hypothesis?
11. Compare the merge sequences selected by an ideal, Oracle-assisted heuristic
to those selected by EDSM to determine when, where, and why EDSM is selecting an incorrect merge.

### 1.4 Overview of Results and Contributions

In this dissertation we have performed a comprehensive analysis of the behaviour the EDSM state merging algorithm. To support this analysis, we have developed an extensive software framework consisting of highly optimised implementations of state merging algorithms as well as statistical analysis and visualisation tools. Using our framework, we created thousands of Abbadingo-style problem instances and investigated both their characteristics and the behaviour of many state merging algorithms when applied to them. Motivated by the results we have obtained by this analysis, we developed three distinct methods which significantly improve on the generalisation rate of EDSM.

## Analysis of State Merging and EDSM

- We studied the proportion of labelled states in the prefix tree acceptors for problem instances at the lowest Abbadingo density for target DFAs ranging from 32 to 512 states. We determined that, as the size of the target DFA increases, the proportion of labelled states decreases - even at the same density (see Table 7.4). In other words, at the lowest Abbadingo density, learners are, proportionally, given fewer strings to infer the target from as the size of the target DFA increases.
- Each state in an APTA maps to a state in the target DFA of the corresponding problem instance. This observation is the basis of graph colouring DFA learning techniques such as Coste and Nicolas' CN97] and Costa Florêncio and Verwer's [CFV12] vertex colouring methods which attempt to find such a mapping. Specifically, if we are searching for an $n$-state target, there are $n$ colours which need to be assigned to the states in the APTA subject to constraints. Of course, during DFA learning, when the target DFA is unknown to us, this mapping is not known. However, for the purpose of analysis (rather than for learning), this mapping can be obtained from an

Oracle and is useful to determine when a heuristic such as EDSM makes an incorrect, colour-incompatible merge choice - such a merge would be incorrect because we would be mapping distinct states in the target DFA to the same state. While discussing colour-compatible merges in detail in Section 6.5, we emphasise that we are not using colour-compatibility for the purposes of DFA learning like in CN97] or CFV12 but, rather, as a tool to reason about what good and bad merges are (in the sense that a sequence of colour-compatible merges represents the best-case behaviour, and always leads from the APTA to the exact target DFA).

- Using 'Oracle-assisted' colour-compatible merges, we used an optimal, nondeterministic, tie-breaking strategy for EDSM to empirically establish an upper bound to its best-case performance. For instance, for 64 -state target DFA problems at the lowest density and a random tie-breaking strategy, EDSM will identify a low-error hypothesis $15 \%$ of the time over 1024 problem instances. On the other hand, if an optimal tie-breaking strategy is used (EDSM always selects a colour-compatible merge from the highest scoring rank of merges), we obtain a rate of about $24 \%$ (see Table 7.5). This establishes a limit of how much we can (experimentally) expect EDSM to improve if we focus on resolving ties more effectively.
- During analysis, we can use an Oracle to help is only select colour-compatible merges which always lead to to the exact target DFA. This allows us to construct what we refer to as ideal merge paths which can be compared to the paths constructed by EDSM. This allows us to determine when, where, and why EDSM selects a wrong pair of states to merge. Furthermore, we can determine the likelihood with which the correct, colour-compatible merge is outside of the first rank of highest EDSM scoring merges (i.e. EDSM has no hope of ever selecting the correct merge).
- In CK02, Cic02, Cicchello et al. discuss the importance of selecting the first merges in a sequence of merges correctly. In Section 7.5.6, we develop on this idea by using colour-compatible merges to determine the effect of getting the first $k$ merges in a sequence correct. For 32-state target DFA
problems at the lowest training set density, while EDSM finds low-error hypotheses at a rate of about $16 \%$ over 1024 problem instances, ensuring that the first eight merges are colour-compatible, increases our success rate to about $55 \%$. These results suggest that expending effort in ensuring that the initial merges are correct, will considerably increase our likelihood of discovering low-error hypotheses.
- We have identified three cases which adversely affect the performance of EDSM. These cases are when (i) the training data is not symmetrically structurally complet $\epsilon^{7}$ with respect to the target DFA, (ii) when the target DFA does not contain loop transitions from some state to itself, and (iii) when the training data yields a prefix tree acceptor, where the highest EDSM scoring rank of tied merges does not contain a colour-compatible merge (i.e. the first merge selected by EDSM is guaranteed to be wrong). Our experiments show that there is a considerable likelihood that randomly constructed problem instances (using the Abbadingo procedure) will result in at least one of these adversarial scenarios (see Table A.13), and that any of these cases negatively impact EDSM's generalisation rate (see Chapter 8).
- Using a non-deterministic 'Oracle-assisted' heuristic, which always selects colour-compatible merges, we experimentally determined that for problem instances of $n$-state target DFAs, the exact target is never found in fewer than $n+1$ merge steps, and that low-error hypotheses are always found in close to $n+1$ steps (see Sections 7.5.4 and 7.5.11). Moreover, we experimentally determined that there is some merge path length $n+k$ for some $k \geq 1$ after which EDSM will never find a low-error hypothesis. This observation is useful to us in two ways. Firstly, the length of a merge path is a good proxy for the quality (in terms of error over a test set) of a hypothesis returned by a state merging algorithm. Secondly, when performing a non-monotonic search, we can confidently abandon path extensions (and save computational

[^4]effort) which have grown too long as they would, almost certainly, end with high-error hypotheses.

- Whenever a merge is performed, the size of the hypothesis decreases. By analysing the state reduction rates in merge paths which lead to the target DFA, we notice that the initial merges chosen in these paths are always 'high state reduction' merges. For example, in Section 7.5.7 and Table B.1, we consider 64 -state target DFA problems, and observe that when we are converging to the correct target DFA, each of the first eight merges in the path reduce the size of the hypothesis by at least an average of 200 states. This result is useful to us because, in the set of all possible merges available at any point, there are significantly more low reduction merges than there are high reduction merges. This allows us to simply ignore these low reduction merges in the initial steps of the construction of a merge path and concentrate our searches on the far smaller pool of high reduction merges.
- Determining the exact number of states which will be reduced at each and every merge step is computationally expensive (as the merge needs to be actually performed). To deal with this, we have developed the APTA Reduction Table structure which contains the set of merges in an APTA which have reduced its size by at least some number of states (see Section 7.5.8). While the merges in this structure do not correspond exactly to the high reduction merges we discussed earlier, the process of constructing the table is substantially more efficient than computing the 'true' reductions at each step, the reduction table still contains far fewer merges than the set of all possible merges, and, more importantly, our experiments show that the table still contains enough good, colour-compatible, merges to get us past the critical first merge steps when constructing a merge sequence (see Section 7.5.9).


## An Ensemble of Heuristics ${ }^{8}$

The first method we have developed in this dissertation is inspired by Wolpert and Macready's No Free Lunch Theorems for Optimisation [WM97] which roughly state that, in general, no single heuristic (inductive bias) will dominate all others over all data sets. During our experimentation we realised that while certain greedy heuristics had worse success rates than others, they were still able to identify low-error hypotheses when the 'better' ones could not and vice-versa. Consider a heuristic called Reduction which scores merges according to how many states have been reduced in the hypothesis (the more states reduced, the higher the score). While this heuristic performs worse than EDSM, it is still able to identify low-error hypothesis on problem instances where EDSM does not. When experimenting with 1024 problem instances of 32 -state target DFAs at the lowest training set density, we observe that EDSM finds a low-error hypothesis in 161 out of the 1024 cases ( $\approx 16 \%$ ) while Reduction does so in only 64 cases $(\approx 6 \%)$. However, either EDSM or Reduction manages to find a low-error hypothesis in 191 out of the 1024 experiments $(\approx 19 \%)$. In other words, the symmetric difference is not empty, and is illustrated in the Venn diagram in Figure 1.2 below.


Figure 1.2: Venn diagram showing the performance of EDSM and Reduction.

Furthermore, while we were studying the effects of target DFA and training set structure on the learning process, we noticed that training sets extracted from targets having certain characteristics were adversarial to some greedy heuristics. For example, in Section 7.6.1, we see that when the target DFA does not contain

[^5]loop transitions from a state to itself, the performance of EDSM is greatly reduced, while in Table 8.8 we see that a carefully selected ensemble of heuristics mitigates this problem. These observations led to the development of an ensemble of heuristics which, together, outperform EDSM to varying, yet significant, degrees. For example, for 64 -state problems at the lowest density, EDSM can discover lowerror hypotheses in roughly $15 \%$ of 1024 problem instances, while the ensemble of heuristics succeeds in $26 \%$ of the same problems. Moreover, the ensemble of heuristics is much less negatively affected than EDSM when the training set is not symmetrically structurally complete with respect to the target DFA.

## The Delta Graph

Our second method is based on the observation that when a greedy, monotonic, heuristic is selecting a sequence of merges, the initial ones are especially critical. When a wrong choice is made, there will be no hope of ever finding the target DFA, and the chances of finding a low-error hypothesis will also be greatly reduced. On the other hand, getting the first few merges right can be shown to constrain the hypothesis well enough to allow us to proceed with a much greater likelihood of succeeding. Here, we present a non-monotonic algorithm which significantly reduces the size of the search space, and allows us to focus our efforts on finding a good sequence of initial merges. In brief, the method involves:

- Building a so called APTA Reduction Table (see Section 7.5.8) consisting of high-reduction merges. This structure contains significantly fewer merges than the entire space of possible merges for a problem instance.
- Building a subspace of short merge sequences, called the Delta Graph, containing quotient DFAs (see Section 3.1.1). We do this by only considering merges in the APTA reduction table and using problem-specific parameters to constrain the size of the subspace.
- Searching in the Delta Graph (the subspace of merges) for a sequence of merges which, when extended using a label matching heuristic, results in a low-error hypothesis.

The exact method for constructing the Delta Graph is given in Section 7.6.2,
and the method is thoroughly evaluated in Section 8.3 where we observe that the Delta Graph method gives significantly better results than EDSM. For example, for 64 -state target DFA problems at the lowest density, EDSM has a success rate of around $15 \%$ over 1024 problem instances, while our method succeeds $43 \%$ of the time. Moreover, this method is much less impacted by problem instances which are highly-adversarial to EDSM. In our evaluation we also compare our results to those reported for SAGE and Ed-Beam by Lang in Lan99.

## An Evolutionary Algorithm

Previous attempts at using evolutionary algorithms to deal with the grammatical inference problem for regular languages focused on either directly evolving a representation of the target DFA, or partitioning the states of an initial hypothesis into a quotient DFA (see Definition 3.1). The literature suggests that these methods do not perform especially well for target DFAs having many states Dup94, LR05.

In our third method, we present an evolutionary approach which rather than searching for the target DFA directly, evolves an initial, short sequence of highquality merges over a reduced, yet expressive, region of the search space. In this method, a fixed-length chromosome is constructed representing a sequence of merges selected from a high state reduction subspace of merges called the APTA reduction table. The fitness of a chromosome is then measured by extending the sequence of merges it contains using the W-EDSM (see Section 5.1.7) label matching heuristic as shown in Figure 1.3 below. Specifically, the fitness is the size of the resulting hypothesis, where smaller hypotheses correspond to fitter chromosomes, and a fitness of zero indicates that we have identified a DFA equal in size to the target DFA.


Figure 1.3: A chromosome consists of a sequence of $k$ merges, and the fitness is evaluated by extending it using W-EDSM.

While the properties of the state merging operator ensure that several operations such as crossover and mutation are straightforward and very efficient (merge order is unimportant, see Chapter (6), the fitness function is computationally expensive and can hinder scalability for very large target DFAs. Notwithstanding this, we observe very promising results. For 32 -state problems at the lowest density, EDSM has a low-error hypothesis success rate of around $14 \%$, while our method succeeds in over $66 \%$ of the cases. The generalisation rate of our evolutionary method also scales well (unlike other evolutionary methods found in the literature) for 64 -state target problems where we manage to find low-error hypotheses in around $56 \%$ of all our test cases, whereas EDSM manages to do so around $14 \%$ of the time. This method is also significantly more resilient to problem instances which are highly adversarial to EDSM. These include, but are not limited to, cases when the training data is not symmetrically structurally complete, and when the target DFA does not contain any loop transitions from a state to itself.

### 1.5 Organisation of this Document

This document is logically organised in two parts. The first chapters are dedicated to covering all the background concepts related to the regular inference problem, reviewing the relevant literature, and describing the behaviour of several state merging algorithms including EDSM and its variants. In the second part, we describe our methodologies, their implementation details, provide a thorough evaluation and discussion, and conclude this dissertation.

## Chapter 2; Preliminaries

The mathematical notation and key concepts related to sets, strings, finite state automata, and regular languages are covered here.

## Chapter 3: Characterising the Search Space of DFA Learning

Here, we introduce important properties, theorems, and formal concepts related to the search space of DFA learning.

## Chapter 4: A Survey of State Space Search Algorithms

In this chapter, we examine several state space search techniques which will be important both when describing existing state merging algorithms, as well as when designing new ones. We also discuss methods which will be useful for understanding the characteristics of search spaces.

## Chapter 5: DFA Learning Algorithms

Here we give a detailed description of several state merging algorithms, heuristics, and their variants. Algorithms including Trakhtenbrot+Bardzin [TB73], RPNI [OG92], EDSM [LPP98], SAGE [JP98b], and Ed-Beam Lan99] are covered here. We also introduce techniques such as Blue-Fringe and windowing to improve search performance. The second part of the chapter is dedicated to covering other methods including the application of evolutionary techniques, SAT solving, recurrent neural networks, and graph colouring to our inference problem.

## Chapter 6: Properties of State Merging

In this chapter, we expand on the properties of the state merging operation, as well as on those of the EDSM scoring heuristic. We will also discuss the interactions between merges, the characteristics of merge sequences, and describe colour-compatible merges as a tool for understanding how merge selection works. The concepts covered in this chapter will be important during the design of the three methods we are proposing later in this dissertation.

## Chapter 7: Baseline Experiments and Methodology

We describe the Abbadingo One experimental setup, create a number of baseline data sets for evaluation, study the effectiveness and behaviour of EDSM, confirm
to which extent the first merges in a merge path are critical, analyse the behaviour of merge paths, correlate low-error hypotheses to the length of merge paths, identify problem instances which are adversarial to EDSM, and introduce the APTA reduction table structure. The remainder of the chapter is dedicated to providing a detailed account of each of the three methods we develop in this dissertation.

## Chapter 8: Evaluation and Results

Each of the three methods we have developed in this dissertation is evaluated against large data sets of problem instances, where we use EDSM and W-EDSM as baselines. Our methods are also evaluated against training sets which have been designed to be highly adversarial to EDSM. Finally, our results are discussed and summarised with respect to the strengths and weaknesses of each.

## Chapter 9: Conclusions and Future Work

Our achievements and contributions are framed with respect to the research hypotheses we presented in this chapter. We conclude this dissertation by proposing the future directions this work leads to.

## Chapter 2

## Preliminaries

Sets, strings, finite-state machines, and formal languages are the foundations upon which most of our work will be based. Consequently, in this chapter, we will cover the key concepts, operations, and notation which we will be using throughout this document while ensuring that any notation we use is consistent with that found in the literature. We highly recommend [Kel95], Lin01, and [dlH10] for an excellent and thorough exposition on these preliminary topics.

### 2.1 Sets

Capital Roman letters such as $P, Q, S$, and $X$ will be used to refer to sets.
$|S|$ denotes the cardinality or size of the set $S$.
$\mathcal{P}(S), P(S), 2^{S}$, or $2^{|S|}$ are used to denote the power set of the set $S$.

Given the sets $P$ and $Q$, set subtraction is denoted by $P \backslash Q$ or $P-Q$ and yields the set containing the elements in $P$ but not in $Q$.
$P \subseteq Q$ and $P \subset Q$ denote that $P$ is a subset of $Q$, and that $P$ is a proper subset of $Q$ respectively.

Likewise, $P \supseteq Q$ and $P \supset Q$ denote that $P$ is a superset of $Q$, and that $P$ is a proper superset of $Q$ respectively.

A partition $\pi$ of a set $S$ is a collection of disjoint subsets, called blocks, whose
union is $S$. As an example, all the five partitions $\pi_{1}, \ldots, \pi_{5}$ of the set $S=\{1,2,3\}$ are shown in Figure 2.1.


Figure 2.1: All the partitions of the set $S=\{1,2,3\}$.

## Definition 2.1: Partitions of a Set

A partition of a finite set $S$ is a set of $k$ subsets, called blocks, denoted by $\pi=\left\{B_{1}, \ldots, B_{k}\right\}$ such that:

$$
S=B_{1} \cup B_{2} \cup \ldots \cup B_{k}
$$

and

$$
B_{i} \cap B_{j}=\emptyset \mid 1 \leq i \leq k, 1 \leq j \leq k, i \neq j .
$$



Figure 2.2: Blocks in a partition.
A block containing $n$ elements is called an $n$-block, and given some $s \in S$, the block $B$ that contains the element $s$ for some partition $\pi$ is denoted by $B(s, \pi)$. So if $S=\{a, b, c\}$ and $\pi=\{\{a\},\{b, c\}\}$, then $B(b, \pi)=\{b, c\}$.

Consider two partitions $\pi_{1}$ and $\pi_{2}$ of some set $S$. The partition $\pi_{1}$ is said to be finer than $\pi_{2}$, if every block in $\pi_{1}$ is a subset of some block in $\pi_{2}$. This relation is written as $\pi_{1} \preceq \pi_{2}$. Dually, the partition $\pi_{2}$ is said to be coarser than $\pi_{1}$ and we write $\pi_{1} \succeq \pi_{2}$. In Figure 2.1, every block in the partition $\pi_{1}$ is a subset of some block in $\pi_{2}$, so $\pi_{1}$ is finer than $\pi_{2}$. This 'finer-than' relation possesses the characteristics of subset inclusion, where $\pi \preceq \pi$ is always true (reflexive), $\pi_{1} \preceq \pi_{2}$ and $\pi_{2} \preceq \pi_{3}$ implies that $\pi_{1} \preceq \pi_{3}$ (transitive), and $\pi_{1} \preceq \pi_{2}$ and $\pi_{2} \preceq \pi_{1}$ implies $\pi_{1}=\pi_{2}$ (antisymmetric). In other words, the relation is a partial order over the set of all partitions of a set and gives rise to a lattice. An example of a lattice for the partitions of the set $S=\{1,2,3,4\}$ is shown in the Hasse diagram in Figure 2.3 below.


Figure 2.3: The lattice for the set $S=\{1,2,3,4\}$.

### 2.2 Strings and Languages

$\Sigma$ denotes a finite set of symbols and is called an alphabet.
$\lambda$ denotes the empty string.
$\Sigma^{*}$ is called the Kleene closure and denotes the set of all strings including $\lambda$ that can be generated over the alphabet $\Sigma$, while $\Sigma^{+}$is called the positive closure and denotes the set of all strings excluding $\lambda$. We elaborate on these operations in Section 2.3.

The lower Roman letters such as $a, b$, and $c$ denote symbols in the alphabet, while
the higher Roman letters such as $u, v$, and $w$ denote strings in $\Sigma^{*}$. $|u|$ denotes the length of the string $u$.

The concatenation of the strings $u$ and $v$ is denoted by $u \cdot v$ or simply by $u v$.

The string $u$ is a prefix of $v$ if $\exists w \mid u w=v$, and $u$ is a suffix of $v$ if $\exists w \mid w u=v$.

A language $L$ is a subset of $\Sigma^{*}$.

Given the languages $L_{1}$ and $L_{2}$, language subtraction denoted by $L_{1} \backslash L_{2}$ or $L_{1}-L_{2}$ results in the language containing all the strings in $L_{1}$ but not in $L_{2}$.

Given the languages $L_{1}$ and $L_{2}$, language concatenation denoted by $L_{1} \odot L_{2}$ is the language $L=\left\{x y \mid x \in L_{1}, y \in L_{2}\right\}$.

A positive sample of some language $L$ denoted by $S_{+}$is a subset set of strings belonging to $L$. In other words, $S_{+} \subseteq L$.

A negative sample of some language $L$ denoted by $S_{-}$is a subset set of strings not belonging to $L$. In other words, $S_{-} \subseteq\left(\Sigma^{*}-L\right)$.
$\operatorname{Pref}(s)$ denotes the set of all prefixes of the string $s$. Likewise, $\operatorname{Pref}(S)=$ $\{u \mid \exists v, u v \in S\}$ denotes the set of all prefixes of all the strings in $S$.
$L / u=\{v \mid u v \in L\}$ denotes the left quotient of $L$ by $u$. Then $L / u \neq \emptyset$ iff $u \in$ $\operatorname{Pref}(L)$. As an example, consider $L=\{\lambda, a, a a, a a a, a a a a, \ldots\}$ and let $u=a$. $L / a$ is then $\{\lambda, a, a a, a a a, a a a a, \ldots\}$. In other words, $L / a$ is the set of all suffixes that complete $a$ to give a word in $L$ (in this example, $L$ and $L / a$ happen to be the same set).

### 2.2.1 Ordering of Strings

Let $\leq_{\text {alpha }}$ denote a total order on letters in the alphabet $\Sigma$. Various orders can be defined over strings $\Sigma^{*}$. For example:

## Definition 2.2: Prefix Ordering of Strings [dlH10]

Prefix order is denoted by $\leq_{p r e f}$ and is defined as:

$$
x \leq_{\text {pref }} y \text { if } \exists w \in \Sigma^{*} \mid y=x w
$$

In other words, $x \leq_{\text {pref }} y$ when $x$ is a prefix of $y$. For example, $a \leq_{\text {pref }}$ $a b \leq_{\text {pref }} a b c \leq_{\text {pref }} a b c d$. We note the limitation of prefix ordering: the string $a b$ and $c d$ are incomparable.

## Definition 2.3: Lexicographic Ordering of Strings dlH10]

Lexicographic order is denoted by $\leq_{l e x}$ and is defined as:

$$
x \leq_{l e x} y \text { if } x \leq_{p r e f} y \text { or }\left(x=u a w, y=u b z \mid a \leq_{a l p h a} b\right)
$$

In other words, $x \leq_{l e x} y$ when the strings are in prefix order, or they share a common prefix $u$ and the next character $a$ in $x$ is $\leq_{a l p h a}$ the next character $b$ in $y$. Consider, $a b \leq_{l e x} a c$. Here the common prefix is $a$, the next characters are $b$ and $c$ respectively, and $b \leq_{a l p h a} c$. Both the tails $w$ and $z$ are $\lambda$.

Definition 2.4: Length-Lexicographic Ordering of Strings [dlH10]
Length-lexicographic order is denoted by $\leq_{l e n g t h-l e x}$ and is defined as:

$$
x \leq_{\text {length-lex }} y \text { if }|x|<|y| \text { or }\left(|x|=|y| \text { and } x \leq_{\text {lex }} y\right)
$$

Note that for strings $a a b$ and $a b$, we see that $a a b \leq_{l e x} a b$ but $a b \leq_{l e n g t h-l e x} a a b$.

### 2.3 Formal Languages

A language is called a formal language if the strings which belong to it are wellformed with respect to a specific set of rules called a grammar. As an example, consider the language defined over the alphabet $\Sigma=\{a, b\}$ and the rule: contains all the strings consisting of zero or more a's followed by zero or more b's. The
strings $\lambda, a, b, a a, a b b$ all belong to the language, whereas the string $a b b a a$ does not because it does not conform to our grammar rule. Of course, for all but the most trivial languages, using English (or any other natural language) descriptions to define grammars may result in undesired ambiguity. To resolve this, our grammars must be defined formally.

## Definition 2.5: Formal Grammars Cho56]

A grammar $G$ is a 4 -tuple $G=(N, \Sigma, P, S)$ where:

- $N$ is a finite set of non-terminal symbols. Non-terminal symbols are ones which may be replaced or expanded, and are denoted by uppercase letters such as $A, B$, and $C$.
- $\Sigma$ is a finite set of terminal symbols. Terminal symbols are the alphabet of our language and may not be replaced or expanded. Like alphabets, terminal symbols are denoted by lower-case letters such as $a, b$, and $c$.
- $P$ is a set of production rules. A production rule has the form ( $\Sigma \cup$ $N)^{*} N(\Sigma \cup N)^{*} \rightarrow(\Sigma \cup N)^{*}$ where $*$ is the Kleene star (see Definition 2.6). The rule is read as: zero or more terminal or non-terminal symbols, followed by a single non-terminal symbol, followed by zero or more terminal or non-terminal symbols, may be rewritten as zero or more terminal or non-terminal symbols. Each rule describes how to replace or expand non-terminal symbols.
- $S \in N$ is a non-terminal symbol which is distinguished as the starting symbol.


## Definition 2.6: Kleene Star [EFT96]

Kleene star, Kleene closure, or the Kleene operator is a unary operation on a set of strings or on an alphabet which is denoted by the $*$ symbol, and is defined as follows:

- Let $S$ be a set of strings or symbols.
- Let $S^{0}=\{\lambda\}$, and let $S^{1}=S$ (basis).
- $S^{i+1}=\left\{u v: u \in S^{i}\right.$ and $\left.v \in S\right\}$ for $i>0$ (recursion).
- The Kleene star $S^{*}$ on the set $S$ is:

$$
S^{*}=\bigcup_{i \geq 0} S^{i}=S^{0} \cup S^{1} \cup S^{2} \cup \ldots
$$

In other words, $S^{*}$ is the set of strings obtained by concatenating zero or more elements of $S$. We note that there are an infinite number of strings in the set $S^{*}$ since there is no restriction on the length of the strings in it. As an example, consider the set $\Sigma=\{a, b\}$. Then $\Sigma^{*}$ is the set of all possible strings over the alphabet $\Sigma$ and gives $\{\lambda, a, b, a a, a b, b a, b b, a a a, \ldots\}$.

## Definition 2.7: Kleene Plus [EFT96]

Kleene plus, or positive closure is a unary operation on a set of strings or on an alphabet which is denoted by + and is defined similarly to Kleene star but excludes the empty string $\lambda$. In other words, $S^{+}=S^{*}-\{\lambda\}$, or equivalently:

$$
S^{+}=\bigcup_{i \geq 1} S^{i}=S^{1} \cup S^{2} \cup \ldots
$$

Consider the grammar $G=(N, \Sigma, P, S)$ where $N=\{S, A\}, \Sigma=\{a, b\}$, and the production rules are:

1. $S \rightarrow \lambda$,
2. $A \rightarrow \lambda$,
3. $S \rightarrow a S$,
4. $S \rightarrow b A$, and
5. $A \rightarrow b A$.

The string aabbb may be generated by the following sequence of production rule applications beginning with the non-terminal starting symbol: $S \xrightarrow{3} a S \xrightarrow{3}$ $a a S \xrightarrow{4} a a b A \xrightarrow{5} a a b b A \xrightarrow{5} a a b b b A \xrightarrow{2} a a b b b \lambda=a a b b b$ where the numbers above the arrows indicate which production was used to replace a non-terminal symbol. This grammar is the formal specification of the contains all the strings consisting of zero or more a's followed by zero or more b's rule we mentioned earlier.

In 1956, Chomsky described a containment hierarchy, called the Chomsky hierarchy, which defines four classes of formal grammars in decreasing order of complexity Cho56. This complexity is based on the forms the production rules of the grammar are permitted to have. These four classes of grammars are called Type 0 , Type 1, Type 2, and Type 3 grammars which respectively recognise recursively enumerable, context-sensitive, context-free, and regular languages. The containment of these grammars is illustrated in Figure 2.4, and the forms that the production rules for each class of grammar can take are shown in Table 2.1. We note that there are two types of regular grammar forms: right-regular grammars, and left-regular grammars. A regular grammar is strictly either right or left-regular.


Figure 2.4: The Chomsky hierarchy.

| Grammar | Language recognised | Production rule form |
| :--- | :--- | :--- |
| Type 0 | Recursively enumerable | $\alpha A \beta \rightarrow \gamma$ |
| Type 1 | Context-sensitive | $\alpha A \beta \rightarrow \alpha \gamma \beta$ |
| Type 2 | Context-free | $A \rightarrow \alpha$ |
|  | Regular (right) | $A \rightarrow x$ and $A \rightarrow x B$ |
| Type 3 | Regular (left) | $A \rightarrow x$ and $A \rightarrow B x$ |
|  |  |  |
| Where: |  |  |
| - $\quad x$ is a single terminal symbol. |  |  |
| $-A$ and $B$ are non-terminal symbols. |  |  |
| - $\alpha$ and $\beta$ are possibly empty strings of terminals and/or non-terminals. |  |  |
| $-\gamma$ is a non-empty string of terminals and/or non-terminals. |  |  |

Table 2.1: Production rule forms in the Chomsky hierarchy.

### 2.4 Finite-State Automata

A finite-state automaton (FSA), also called a finite-state machine (FSM), is an abstract machine used to recognise or generate strings in a language. FSAs are a mathematical model consisting of a finite number of states, transitions between them, and actions. FSAs are usually illustrated as graphs where one of the states is distinguished as the starting state and any state may be marked as an accepting one. A starting state is visualised by a null arrow incident to it, and an accepting state is drawn with a double outline. For instance, in the FSA shown in Figure
2.5, the starting state is $q_{0}$, and the state $q_{1}$ is labelled as an accepting state. This FSA recognises the strings $\{a, a a, a a a, a a a a, \ldots\}$ since for every string in this set, a walk in the graph starting at the starting state always reaches an accepting state.


Figure 2.5: The finite-state automaton for the language $a^{+}$.

When an FSA is used to recognise strings in a language (i.e. it is able to answer a yes/no question as to whether a string is in a language or not) it is called an acceptor. When an FSA is used to generate strings in a language it is called a generator. It should be noted that while a generator can generate all the possible strings in a language, this process will never terminate when the language under consideration is infinite (as is typically the case).

We note that there exists a one-to-one correspondence between regular grammars (described in Section 2.3) and the languages recognised by an FSA HMU07. Specifically, regular languages are equivalently defined by both regular grammars and FSAs.

### 2.4.1 Non-Deterministic Finite-State Automata

A non-deterministic finite-state automaton (NFA) is an automaton where each state may have zero, one, or more transitions corresponding to a symbol in the alphabet. We can formally define NFAs as follows:

## Definition 2.8: Non-Deterministic Finite-State Automata Lin01, dlH10

A non-deterministic finite-state automaton is a 5 -tuple $\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F\right\rangle$ where:

- $\Sigma$ is an alphabet, and $Q$ is a finite set of states.
- $q_{0} \in Q$ is the initial state (some NFAs define a set $I \subseteq Q$ of initial states whereas here we are only concerned with NFAs having one initial state).
- $\delta: Q \times(\Sigma \cup\{\lambda\}) \rightarrow 2^{Q}$ is a transition function (here we are including empty transitions whereas in some applications, the transition function of NFAs excludes the empty transition).
- $F \subseteq Q$ is the set of final states.

In grammatical inference problems, it is common to label final states as being of two types - final-accepting states $F_{A}$ and final-rejecting states $F_{R}$. States which are neither accepting states nor rejecting states are called unknown, neutral, or unlabelled states. In these cases $F=F_{A} \cup F_{R}$ and our automaton would then be defined as a 6 -tuple $\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F_{A}, F_{R}\right\rangle$. Such automata are usually called unbiased automata since they embed both positive and negative information AS94b. In our diagrams, unknown states will be denoted by a single black outline, final-accepting states will be denoted by a double black outline, and final-rejecting states by a thick black outline. To illustrate, consider the NFA show in Figure 2.6, where:

- $\Sigma=\{a, b\}$
- $Q=\left\{q_{0}, q_{1}, q_{2}\right\}$
- $q_{0}=q_{0}$
- $\delta=\left\{\begin{array}{l}\delta\left(q_{0}, a\right) \rightarrow q_{1} \\ \delta\left(q_{1}, b\right) \rightarrow q_{1} \\ \delta\left(q_{1}, b\right) \rightarrow q_{2}\end{array}\right\}$
- $F_{A}=\left\{q_{2}\right\}, F_{R}=\emptyset$


Figure 2.6: A non-deterministic finite-state automaton.

NFAs can also have $\lambda$-transitions. A $\lambda$-transition, sometimes called an empty transition, allows the automaton to 'move' from one state to another without consuming any input (when recognising) or without emitting any output (when generating). An example NFA with a $\lambda$-transition is shown in Figure 2.7 below.


Figure 2.7: A finite-state automaton with an empty transition.

Transition functions may be composed. With respect to the NFA shown in Figure 2.6, processing the string $a b b$ starting from the state $q_{0}$ leads us to the state $q_{2}$. This is equivalent to the composition $\delta\left(\delta\left(\delta\left(q_{0}, a\right), b\right), b\right) \rightarrow q_{2}$. Since $q_{2}$ is a state in $F_{A}$, we say that the string $a b b$ is accepted by the NFA. For conciseness, we also use the notation $\delta(q, u)$ to represent the composition of transition functions when starting from state $q$ after processing the string $u$ (i.e. we use recursion). Using this notation for the example we just presented, we would write $\delta\left(q_{0}, a b b\right) \rightarrow q_{2}$.

## Definition 2.9: Languages Recognised by an Automaton [dlH10]

The language $L_{F_{A}}(\mathcal{A})$ recognised by an automaton $\mathcal{A}$ is the set of all strings $u \in \Sigma^{*}$ where $\delta(q, u) \in F_{A}$ starting from $q_{0}$. For brevity, $L(\mathcal{A})$ is sometimes used to denote $L_{F_{A}}(\mathcal{A})$.

## Definition 2.10: Languages Recognised by Rejection [dlH10]

The language $L_{F_{R}}(\mathcal{A})$ recognised by rejection by an automaton $\mathcal{A}$ is the set of all strings $u \in \Sigma^{*}$ where $\delta(q, u) \in F_{R}$ starting from $q_{0}$.

It should be noted that even though the sets $F_{A}$ and $F_{R}$ are disjoint, there may still be strings that belong to both $L_{F_{A}}(\mathcal{A})$ and $L_{F_{R}}(\mathcal{A})$. An example of this is shown in Figure 2.8 where the string $a b$ is accepted and rejected by the same NFA.


Figure 2.8: The string $a b$ is both accepted and rejected by the NFA.

## Definition 2.11: Consistent NFAs [dlH10]

An NFA $\mathcal{A}$ is consistent if $L_{F_{A}}(\mathcal{A}) \cap L_{F_{R}}(\mathcal{A})=\emptyset$. The preceding example shown in Figure 2.8, shows how the NFA is not consistent since the string $a b$ is ambiguously both accepted and rejected by it.

## Definition 2.12: Equivalent NFAs [Lin01]

Two NFAs $\mathcal{A}$ and $\mathcal{A}^{\prime}$ are equivalent if $L_{F_{A}}(\mathcal{A})=L_{F_{A}}\left(\mathcal{A}^{\prime}\right)$ and $L_{F_{R}}(\mathcal{A})=$ $L_{F_{R}}\left(\mathcal{A}^{\prime}\right)$.

The acceptance of a string $u=a_{1} a_{2} \ldots a_{l}$ by an automaton $\mathcal{A}$ defines a (possibly non-unique) sequence of $l+1$ states $q_{0}, \ldots, q_{l}$ such that $q_{0}$ is the starting state, $q_{l}$ is a final state, and $q_{i+1} \in \delta\left(q_{i}, a_{i+1}\right)$ for $0 \leq i \leq l-1$ defines the successor state. These $l+1$ states are said to be reached by the path $p=q_{0} \rightarrow q_{1} \rightarrow \ldots \rightarrow q_{l}$ for string $u$, whereas the $l$ transitions are said to be exercised by it. Given a set of strings $S$, the set of transitions exercised by the set is the union of all the transitions exercised for each $u \in S$.

A language $L(\mathcal{A})$ accepted by antomaton $\mathcal{A}$ is the set of strings accepted by $\mathcal{A}$. A state $q \in Q$ of an automaton $\mathcal{A}$ is said to be useful if there is a string $u \in L(\mathcal{A})$ for which the state $q$ can be reached for some acceptance of $u$. In other words, $q$ is on the state path that $u$ takes when being accepted. The state is
otherwise said to be useless, meaning that it is not reached by any string $u \in L(\mathcal{A})$.

### 2.4.2 Deterministic Finite-State Automata

A deterministic finite-state automaton (DFA is an automaton $\mathcal{A}$, where $\forall q \in Q$ and $\forall a \in \Sigma$, each transition $\delta(q, a)$ has at most one member for each symbol $a \in \Sigma$. If each transition $\delta(q, a)$ has exactly one member, we call the deterministic automaton complete.

Definition 2.13: Deterministic Finite-State Automata [Lin01, dlH10]
A deterministic finite-state automaton is a 5 -tuple $\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F\right\rangle$ where:

- $\Sigma$ is an alphabet, and $Q$ is a set of states.
- $q_{0} \in Q$ is the initial state.
- $\delta: Q \times \Sigma \rightarrow Q$ is a transition function.
- $F \subseteq Q$ is the set of final states.

Again, it is useful to make a distinction between final-accepting states, and final-rejecting states in DFAs which would then be defined as a 6 -tuple $\mathcal{A}=$ $\left\langle\Sigma, Q, q_{0}, \delta, F_{A}, F_{R}\right\rangle$. Similarly, such an automaton would embed both positive and negative information and is called an unbiased automaton AS94b. Additionally, while the distinction between a DFA and an NFA has practical implications, they both have exactly the same expressive power. Any NFA can be converted into a DFA using the method of power set construction although it should be noted that this construction could result in an exponential increase in the number of states required Lin01].

### 2.4.3 The Depth of a DFA

The depth of a DFA is the maximum over all states of the length of the shortest string which leads to that state. More specifically, let $q_{1}, q_{2}, \ldots \in Q$ be the states in a DFA, and let $\phi: Q \rightarrow \mathbb{N}$ be a function which returns the length of the shortest
string from the starting state leading to some state in $Q$. The depth of a DFA is $\max \left(\phi\left(q_{1}\right), \phi\left(q_{2}\right), \ldots\right)$.

### 2.4.4 Canonical Automata

A canonical automaton $\mathcal{A}(L)$ is the deterministic automaton having the minimal number of states required to recognise a language $L$. It should be noted that any automaton recognising $L$ having the same minimal number of states as $\mathcal{A}(L)$ is isomorphic to $\mathcal{A}(L)$.

## Definition 2.14: Canonical Automata DMV94]

Let $\mathcal{A}(L)=\left\langle\Sigma, Q, q_{0}, \delta, F\right\rangle$ be the automaton having the minimal number of states required to recognise the language $L . \mathcal{A}(L)$ is called the canonical automaton of $L$ and is defined as follows:

- $Q=\{L / u: u \in \operatorname{Pref}(L)\}$,
- $q_{0}=L / \lambda$,
- $F=\{L / u: U \in L\}$, and
- $\delta(L / u, a)=L / u a$ where $u, u a \in \operatorname{Pref}(L)$.

Any automaton can be reduced to its equivalent canonical form, and this reduction is useful for a variety of reasons. These include:

- The obvious computational and space efficiencies when dealing with automata having fewer states,
- It is unique, and
- It allows us to target only one automaton when learning a language.


### 2.4.5 DFA Minimisation

Minimisation is a technique used to obtain a canonical DFA from some input DFA. The process involves removing and merging states from the original DFA such that
the language it accepts (or rejects) is not affected. The following types of states are considered during this process:

1. Unreachable states - useless states that are are not reachable from the starting state. A state $q$ is unreachable if no string $u \in \Sigma^{*}$ exists such that $q=\delta\left(q_{0}, u\right)$.
2. Non-distinguishable states - two states $p$ and $q$ are indistinguishable if $\delta(p, u) \in$ $F$ implies that $\delta(q, u) \in F$, and $\delta(p, u) \notin F$ implies that $\delta(q, u) \notin F$, for all $u \in \Sigma^{*}$. Equivalently, if $\exists u \in \Sigma^{*}$ such that $\delta(p, u) \in F$ and $\delta(q, u) \notin F$ then $p$ and $q$ are distinguishable by the string $u$.

In the DFA shown in Figure 2.9, states $q_{4}$ and $q_{5}$ are unreachable from the starting state $q_{0}$. Clearly, a DFA without unreachable states is simply the original DFA with the unreachable states as well as their incident and adjacent transitions deleted. The algorithm to identify unreachable states is shown in Algorithm 2.1 at the end of this section. We can now turn our attention to identifying states that are indistinguishable. Distinguishability and indistinguishability have the properties of equivalence relations. That is, if states $p$ and $q$ are indistinguishable, and states $q$ and $r$ are indistinguishable, then $p$ and $r$ must also be indistinguishable indeed, all the states $p, q, r$ are indistinguishable. An algorithm called Mark that finds all distinguishable states is adapted from [in01] to accommodate both accepting and rejecting final states in unbiased automata, and is given in Algorithm 2.2 at the end of this section. To improve readability, we make use of a helper function TypeOf(State) $\rightarrow$ \{Accepting, Rejecting, Unknown $\}$ that given some state, tells us whether it is a final accepting state (i.e. state $\in F_{A}$ ), a final rejecting state (i.e. state $\in F_{R}$ ), or otherwise.


Figure 2.9: A DFA having the unreachable states $q_{4}$ and $q_{5}$.
Consider the DFA shown in Figure 2.10 from Lin01] to illustrate the behaviour of Algorithm 2.2;


Figure 2.10: An example showing how the Mark algorithm works.

Mark starts by eliminating unreachable states from the DFA. This DFA has no unreachable states, so it remains as it is. Next, Mark generates all unordered and distinct pairs of states, and marks them as distinguishable when their states have different types. In our example, Table 2.2 shows which pairs of states are marked as distinguishable.

| Pair | State types | Action |
| :--- | :--- | :--- |
| $\left(q_{0}, q_{1}\right)$ | (unlabelled, unlabelled) | Unmarked. |
| $\left(q_{0}, q_{2}\right)$ | (unlabelled, unlabelled) |  |
| $\left(q_{0}, q_{3}\right)$ | (unlabelled, unlabelled) |  |
| $\left(q_{0}, q_{4}\right)$ | (unlabelled, accepting) | Mark as distinguishable. |
| $\left(q_{1}, q_{2}\right)$ | (unlabelled, unlabelled) |  |
| $\left(q_{1}, q_{3}\right)$ | (unlabelled, unlabelled) |  |
| $\left(q_{1}, q_{4}\right)$ | (unlabelled, accepting) | Mark as distinguishable. |
| $\left(q_{2}, q_{3}\right)$ | (unlabelled, unlabelled) | Unmarked. |
| $\left(q_{2}, q_{4}\right)$ | (unlabelled, accepting) | Mark as distinguishable. |
| $\left(q_{3}, q_{4}\right)$ | (unlabelled, accepting) |  |

Table 2.2: State pairs in the DFA shown in Figure 2.10 are marked as distinguishable.

Finally, the algorithm iterates through all remaining unmarked pairs, and for each one, generates new successor pairs using $\delta$ and $a \in \Sigma$. If a new pair is marked as distinguishable, so is the original. The process is repeated until no new pairs have been marked. To illustrate, we consider the unmarked pair $\left(q_{0}, q_{1}\right)$ and generate new successor pairs using $\delta$ for each possible symbol as shown in Table 2.3 .

| Sym | Pair | New Pair | Action |
| :--- | :--- | :--- | :--- |
| $a$ | $\left(q_{0}, q_{1}\right)$ | $\left(\delta\left(q_{0}, a\right), \delta\left(q_{1}, a\right)\right)=\left(q_{1}, q_{2}\right)$ | $\left(q_{1}, q_{2}\right)$ is unmarked, so the origi- <br> nal pair $\left(q_{0}, q_{1}\right)$ remains unmarked <br> too. |
| $b$ | $\left(q_{0}, q_{1}\right)$ | $\left(\delta\left(q_{0}, b\right), \delta\left(q_{1}, b\right)\right)=\left(q_{3}, q_{4}\right)$ | $\left(q_{3}, q_{4}\right)$ is marked, so we mark the <br> original pair $\left(q_{0}, q_{1}\right)$ too. |

Table 2.3: Generating successors for the pair $\left(q_{0}, q_{1}\right)$.

This procedure can be proven to find all distinguishable pairs of states in any DFA Lin01, and yields the following pairs of distinguishable pairs of states in our example: $\quad\left(q_{0}, q_{1}\right),\left(q_{0}, q_{2}\right),\left(q_{0}, q_{3}\right),\left(q_{0}, q_{4}\right),\left(q_{1}, q_{4}\right),\left(q_{2}, q_{4}\right),\left(q_{3}, q_{4}\right)$. The remaining indistinguishable pairs are $\left(q_{1}, q_{2}\right),\left(q_{1}, q_{3}\right),\left(q_{2}, q_{3}\right)$. Using our (in)distinguishability equivalence relation, states $q_{1}, q_{2}, q_{3}$ are then all indistinguishable from one another. We can now wrap up our DFA minimisation task using a method called Reduce. The basis for Reduce is the partitioning of our set of states $Q$ into blocks that satisfy the following partitioning properties:

1. Any state $q \in Q$ appears in exactly one block, and
2. All the states in a block are indistinguishable from each other, and
3. States in a different block are distinguishable.

Since all the states in a block are indistinguishable, they can be 'grouped' into one state without affecting the language recognised by our DFA. Continuing our example where we found $q_{1}, q_{2}$, and $q_{3}$ to be indistinguishable, we would then get the following partition: $\left\{\left\{q_{0}\right\},\left\{q_{1}, q_{2}, q_{3}\right\},\left\{q_{4}\right\}\right\}$. The procedure for Reduce is
presented in Algorithm 2.3 at the end of this section. Applying Reduce to our example, we create the states for the blocks $\left\{q_{0}\right\},\left\{q_{1}, q_{2}, q_{3}\right\},\left\{q_{4}\right\}$ of our partition:

... and derive the following transitions for $\mathcal{A}^{\prime}$ :

$$
\delta^{\prime}=\left\{\begin{array}{ll}
\delta^{\prime}\left(q_{0}, a\right) & \rightarrow q_{1,2,3} \\
\delta^{\prime}\left(q_{0}, b\right) & \rightarrow q_{1,2,3} \\
\delta^{\prime}\left(q_{1,2,3}, a\right) & \rightarrow q_{1,2,3} \\
\delta^{\prime}\left(q_{1,2,3}, b\right) & \rightarrow q_{4} \\
\delta^{\prime}\left(q_{4}, a\right) & \rightarrow q_{4} \\
\delta^{\prime}\left(q_{4}, b\right) & \rightarrow q_{4}
\end{array}\right\}
$$

...to finally obtain our minimised DFA $\mathcal{A}^{\prime}$ :


We refer readers to Lin01 for a proof of correctness for the Reduce procedure, as well as to Alq97 for a further discussion on the minimisation of unbiased automata.

Algorithm 2.1 Finding unreachable states ([Lin01])
Input: A DFA $\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F_{A}, F_{R}\right\rangle$.
Output: A set of unreachable states in $\mathcal{A}$ from $q_{0}$.

Reachable $\leftarrow\left\{q_{0}\right\}$
Current $\leftarrow\left\{q_{0}\right\}$
repeat
Next $\leftarrow\}$
for $q \in$ current do
for $c \in \Sigma$ do
Next $\leftarrow \operatorname{Next} \cup\{p: p=\delta(q, c)\}$
end for
end for
// Don't visit states we know to be reachable.
Current $\leftarrow$ Next - Reachable
// States in Current are definitely reachable.
Reachable $\leftarrow$ Reachable $\cup$ Current
until Current is empty
return $Q \backslash$ Reachable

```
Algorithm 2.2 Mark (adapted from Lin01)
Input: A DFA \(\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F_{A}, F_{R}\right\rangle\).
Output: A set of pairs of distinguishable states.
    1: Remove all unreachable states. // Using Algorithm 2.1
    // Generate all unordered pairs \((p, q)\) such that \(p \neq q\).
    / / Note that we have \(\frac{n^{2}+n}{2}\) unordered pairs by the triangle number method.
    / / So we get \(\frac{n^{2}+n}{2}-n=\frac{n^{2}-n}{2}\) unordered pairs such that \(p \neq q\).
    // If the states in the pair \((p, q)\) have different types, they are distinguishable.
    for \(i=1 \ldots|Q|\) do
        for \(j=i+1 \ldots|Q|\) do
            \(p \leftarrow Q[i]\)
            \(q \leftarrow Q[j]\)
            if type \(O f(p) \neq\) type \(O f(q)\) then
                Mark the pair \((p, q)\) as distinguishable.
            end if
            end for
    end for
    repeat // Mark remaining pairs.
    for \(i=1 \ldots|Q|\) do \(\quad / /\) Loop through all unordered and distinct pairs.
            for \(j=i+1 \ldots|Q|\) do
                \(p \leftarrow Q[i]\)
                \(q \leftarrow Q[j]\)
                if \((p, q)\) is already marked then continue // Skip already marked.
                for \(c \in \Sigma\) do
                    \(p_{c} \leftarrow \delta(p, c)\)
                    \(q_{c} \leftarrow \delta(q, c)\)
                    if \(\left(p_{c}, q_{c}\right)\) is marked as distinguishable then
                    Mark the pair \((p, q)\) as distinguishable too.
                    end if
                end for
        end for
    end for
    until No new pairs have been marked as distinguishable
    return All pairs that have been marked as distinguishable.
```

```
Algorithm 2.3 Reduce (adapted from [Lin01])
```

Input: A DFA $\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F_{A}, F_{R}\right\rangle$.
Output: A minimal DFA $\mathcal{A}^{\prime}=\left\langle\Sigma, Q^{\prime}, q_{0}^{\prime}, \delta^{\prime}, F_{A}^{\prime}, F_{R}^{\prime}\right\rangle$ where $L(\mathcal{A})=L\left(\mathcal{A}^{\prime}\right)$.

Use Mark to find all the distinguishable and indistinguishable states.
Using our 'partitioning properties' Partition the state set $Q$ into blocks $B$.
// Create a new state for each block of indistinguishable states.
: for each block $\left\{q_{i}, q_{j}, \ldots, q_{k}\right\} \in B$ do
Create a state $q_{i, j, \ldots, k}$ in $\mathcal{A}^{\prime}$
end for
: // Construct transitions.
for each transition $\delta(q, a) \rightarrow q^{\prime}$ in $\mathcal{A}$ do
$\left\{q_{i}, q_{j}, \ldots, q_{k}\right\} \leftarrow$ the set that $q$ belongs to
$\left\{q_{l}, q_{m}, \ldots, q_{n}\right\} \leftarrow$ the set that $q^{\prime}$ belongs to
// Without creating duplicate transitions...
Add the rule $\delta^{\prime}\left(q_{i, j, \ldots, k}, a\right) \rightarrow q_{l, m, \ldots, n}$ to $\mathcal{A}^{\prime}$
end for

14: // Initial state.
15: The initial state in $\mathcal{A}^{\prime}$ is the one that contains $q_{0}$ in $\mathcal{A}$.

16: // Final states.
17: The accepting states in $\mathcal{A}^{\prime}$ are the ones that contain the states $q \in F_{A}$ in $\mathcal{A}$.
18: The rejecting states in $\mathcal{A}^{\prime}$ are the ones that contain the states $q \in F_{R}$ in $\mathcal{A}$.

19: return $\mathcal{A}^{\prime}$

### 2.5 Summary

In this section, we reviewed the notation and concepts relating to sets, strings, and finite-state machines which we will use throughout this dissertation. We also discussed regular languages, and described the types of finite-state machines which are relevant to our work. The algorithm for reducing an automaton to its canonical form has been elaborated on since it is one of the basic components necessary for constructing and studying data sets conforming to the Abbadingo One competition procedure.

## Chapter 3

## Characterising the Search Space of DFA Learning

In this chapter, we investigate the search space associated with learning DFAs, primarily referring to the work of Dupont et al. DMV94 and de la Higuera dlH10, as well as that by Fu et al. FB75. The concepts covered in this section constitute the 'building blocks' of DFA learning algorithms. We will be discussing:

- Derived and quotient automata: given an automaton and some partition of its states, that partition can be used to derive another automaton from the first.
- The structural completeness of a training set with respect to an automaton: this property represents a condition which is required for the exact identification of the automaton.
- The maximum canonical automaton: the most specific NFA (hypothesis) embedding the positive strings in a training set.
- The prefix tree acceptor and the augmented prefix tree acceptor: the most specific DFA (hypothesis) respectively embedding, either the positive strings in a training set, or the positive and negative strings in a training set.
- The search space of automata and its border set (the deepest automata in the search space such that no further generalisation would result in an automaton consistent with the training set).
- We also discuss a number of important properties regarding these concepts and structures.


### 3.1 Introductory Concepts

### 3.1.1 Quotient Automata

Consider an automaton $\mathcal{A}$ having states $Q$, as well as a partition $\pi$ on those states. This partitioning gives rise to an automaton $\mathcal{A} / \pi$ called a quotient automaton which is formally defined as follows:

## Definition 3.1: Quotient Automata [DMV94]

If $\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F\right\rangle$ is an automaton, then $\mathcal{A} / \pi=\left\langle\Sigma, \bar{Q}, \bar{q}_{0}, \bar{\delta}, \bar{F}\right\rangle$ is an automaton derived from $\mathcal{A}$ with respect to the partition $\pi$ and is called a quotient automaton. Quotient automata are defined as follows:

- $Q^{\prime}=\{B(q, \pi) \mid q \in Q\}$,
- $q_{0}^{\prime}=B\left(q_{0}, \pi\right)$,
- $\delta^{\prime}: Q^{\prime} \times \Sigma \rightarrow 2^{Q^{\prime}} \mid \forall B, B^{\prime} \in Q^{\prime}$ and $\forall a \in \Sigma, B^{\prime} \in \delta^{\prime}(B, a)$ is defined iff $\exists q, q^{\prime} \in Q, q \in B, q^{\prime} \in B^{\prime}$ and $q^{\prime} \in \delta(q, a)$.
- $F^{\prime}=\left\{B \in Q^{\prime} \mid B \cap F \neq \emptyset\right\}$.

The states in $Q$ that are in the same block $B$ of $\pi$ are said to be merged together. As an example, consider the automaton shown in Figure 3.1 (i) and the partition $\pi=\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}, q_{4}\right\}\right\}$. We say that the automaton 3.1 (ii) is derived from 3.1 (i) with respect to $\pi$ and the states $q_{3}$ and $q_{4}$ are merged together.

### 3.1.2 Structural Completeness and Characteristic Sets

A positive sample $S_{+}$is said to be structurally complete with respect to an automaton $\mathcal{A}$ if there exists an acceptance of the strings in $S_{+}$by $\mathcal{A}$ such that DMV94:

(i)

(ii)

Figure 3.1: A DFA (i) and the derived quotient automaton (ii) with respect to the partition $\pi=\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}, q_{4}\right\}\right\}$.

- Every transition of $\mathcal{A}$ is exercised, and
- Every final state of $\mathcal{A}$ is used as an accepting state.

Consider the DFA shown in Figure 3.2. The sample $S_{+}=\{a a, b a, a a a, a a b a a\}$ is not structurally complete with respect to it since the final state $q_{0}$ is not reached by any string in the sample, and the sample $S_{+}=\{\lambda, a a, a a a, a a b\}$ is not structurally complete either since the transition $\delta\left(q_{0}, b\right)=q_{1}$ is never exercised. On the other hand, $S_{+}=\{\lambda, a a, b a, a a a, a a b\}$ is structurally complete since all the final states are reached and all the transitions in the DFA are exercised. We can extend the concept of structural completeness to symmetrical structural completeness ${ }^{1}$ where in the case of biased automata (see Section 2.4.2), we also take into account a negative sample $S_{-}$. Specifically, we require that every final rejecting state is reached by some string in $S_{-}$and that every transition is exercised by some string in $S=S_{+} \cup S_{-}$Alq97, dlH10.


Figure 3.2: This DFA is structurally complete with respect to $S_{+}=$ $\{\lambda, a a, b a, a a a, a a b\}$ but not with respect to $S_{+}=\{\lambda, a a, a a a, a a b\}$.

[^6]Consider a class of languages $\mathcal{C}$. The characteristic set for a learning algorithm and a language $L \in \mathcal{C}$ is the set of strings $S_{+} \subseteq L$ and $S_{-} \cap L=\emptyset$ such that whenever the algorithm runs with the input $\left\langle S_{+}, S_{-}\right\rangle$it outputs a correct representation of $L$ Gol67, Dup96, OG02]. Moreover, the representation found does not change if new strings are added to $S_{+}$or $S_{-}$GLVdP12]. We note that the language identification in the limit model Gol67 establishes that an algorithm identifies a class of languages $\mathcal{C}$ in the limit ${ }^{2}$ if and only if every language in the class has associated with it a characteristic set for that algorithm. An algorithm for constructing a polynomial characteristic set ${ }^{3}$ for certain classes of state merging algorithms may be found in GLVdP12.

### 3.1.3 The Maximal Canonical Automaton

A maximal canonical automaton (MCA) with respect to a positive sample $S_{+}$ is denoted by $\mathcal{M C A}\left(S_{+}\right)$and corresponds to the acyclic automaton having the largest number of useful states that recognises the strings in $S_{+}$[DMV94].

One may observe that $\mathcal{M C \mathcal { C }}\left(S_{+}\right)$is a star-shaped automaton having one branch for each string in $S_{+}$and is almost always non-deterministic. As an example, consider $S_{+}=\{a a, a b, b b, a b b\}$ which gives the MCA shown in Figure 3.3. Note that each state in an MCA corresponds to every non-empty prefix of $S_{+}$plus the state $q_{\lambda}$.

[^7]
## Definition 3.2: Maximal Canonical Automaton dlH05]

Let $S_{+}=\left\{s_{1}, s_{2}, \ldots, s_{n}\right\}$ be a positive sample. Let $\mathcal{M C \mathcal { A }}\left(S_{+}\right)=\left\langle\Sigma, Q, q_{0}, \delta, F\right\rangle$ denote the maximal canonical automaton of $S_{+}$which is defined as follows:

- $\Sigma$ is the alphabet over which $S_{+}$is defined,
- $Q=\left\{q_{u}^{i}: u \in \operatorname{PREF}\left(s_{i}\right)\right.$ and $\left.u \neq \lambda\right\} \cup\left\{q_{\lambda}\right\}$,
- $q_{0}=q_{\lambda}$,
- $\delta\left(q_{\lambda}, a\right)=\left\{q_{a}^{i}: a \in \operatorname{Pref}\left(s_{i}\right)\right\}, \forall i \in[n], \forall a \in \Sigma$,
- $\delta\left(q_{u}^{i}, a\right)=\left\{q_{u a}^{i}: u a \in \operatorname{Pref}\left(s_{i}\right)\right\}, \forall i \in[n], \forall a \in \Sigma$,
- $F=\left\{q_{s_{i}}\right\}, \forall i \in[n]$.


Figure 3.3: The MCA for $S_{+}=\{a a, a b, b b, a b b\}$. The superscript indicates the index of the string in $S_{+}$that passes through the state, whereas the subscript indicates the prefix of the string in $S_{+}$that passes through it.

### 3.1.4 The Prefix Tree Acceptor

A prefix tree acceptor (PTA) with respect to a positive sample $S_{+}$is denoted by $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$and corresponds to the quotient automaton $\mathcal{M C \mathcal { A }}\left(S_{+}\right) / \pi$ where $\pi$ is defined as $B(q, \pi)=B\left(q^{\prime}, \pi\right) \operatorname{iff} \operatorname{Pref}(q)=\operatorname{Pref}\left(q^{\prime}\right)$. In other words, $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$is obtained by merging the states in $\mathcal{M C A}\left(S_{+}\right)$that have a common prefix Dup94, [CN97. An alternative and equivalent definition is shown in Definition 3.3 next. It can be seen that this construction makes $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$a deterministic automaton. The PTA for $S_{+}=\{a a, a b, b b, a b b\}$ is shown in Figure 3.4 and shows how the states in the corresponding MCA are merged together.


Figure 3.4: The PTA for $S_{+}=\{a a, a b, b b, a b b\}$. The superscript indicates the index of the string in $S_{+}$that passes through the state, whereas the subscript indicates the prefix of the string in $S_{+}$that passes through it.

## Definition 3.3: Prefix Tree Acceptor [dlH05]

A prefix tree acceptor embedding a positive sample $S_{+}$is denoted by $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$ and is defined as follows:

- $Q=\left\{q_{u}: u \in \operatorname{Pref}\left(S_{+}\right)\right\}$,
- $q_{0}=q_{\lambda}$,
- $\forall u a \in \operatorname{Pref}\left(S_{+}\right)$and $\forall a \in \Sigma, \delta\left(q_{u}, a\right)=q_{u a}$,
- $F=\left\{q_{u}: u \in S_{+}\right\}$.


### 3.1.5 The Augmented Prefix Tree Acceptor

Intuitively, an augmented prefix tree acceptor (APTA is the superposition of $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$and $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{-}\right)$and is denoted as $\mathcal{A} \mathcal{P} \mathcal{A} \mathcal{A}(S)$ where $S=\left\langle S_{+}, S_{-}\right\rangle$. The example in Figure 3.5 shows the PTA (i) for $S_{+}=\{a, b, a a b, a b b\}$, the PTA (ii) for $S_{-}=\{b a, b b a, b a a\}$, and the APTA (iii) which is the superposition of (i) and (ii). A formal definition of an APTA is given in Definition 3.4.

(i)

(ii)

(iii)

Figure 3.5: The superposition of $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$and $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{-}\right)$is an APTA.

## Definition 3.4: Augmented PTA, adapted from dlH05]

The augmented prefix tree acceptor embedding a sample $S=\left\langle S_{+}, S_{-}\right\rangle$, denoted by $\mathcal{A} \mathcal{P} \mathcal{A}(S)$ or $\mathcal{A} \mathcal{P} \mathcal{A}\left(S_{+}, S_{-}\right)$, is the $\operatorname{DFA}\left\langle\Sigma, Q, q_{0}, \delta, F_{A}, F_{R}\right\rangle$ having:

- $Q=\left\{q_{u}: u \in \operatorname{Pref}\left(S_{+} \cup S_{-}\right)\right\}$,
- $q_{0}=q_{\lambda}$,
- $\forall u a \in \operatorname{Pref}\left(S_{+} \cup S_{-}\right)$and $\forall a \in \Sigma, \delta\left(q_{u}, a\right) \rightarrow q_{u a}$,
- $F_{A}=\left\{q_{u}: u \in S_{+}\right\}$.
- $F_{R}=\left\{q_{u}: u \in S_{-}\right\}$.


### 3.2 Derived Automata

Let $\Pi(\mathcal{M C \mathcal { A }})=\left\{\pi_{1}, \pi_{2}, \ldots, \pi_{n}\right\}$ denote the set of all possible state partitions of an MCA. We say that a partition $\pi_{b}$ directly derives from a partition $\pi_{a}$ and write $\pi_{a} \preceq \pi_{b}$ whenever $\pi_{b}$ can be constructed from $\pi_{a}$ as follows [CF03, DMV94:

$$
\exists b_{1}, b_{2} \in \pi_{a}, b_{1} \neq b_{2}, \pi_{2}=\left(\pi_{1}-\left\{b_{1}, b_{2}\right\}\right) \cup\left\{b_{1} \cup b_{2}\right\}
$$

We note that that the set of partitions $\Pi$ is a complete lattice under the $\preceq$ relation, and denote the lattice of partitions of an MCA by $\mathcal{L A} \mathcal{T}(\mathcal{M C A})$. The notation $\mathcal{D} \mathcal{F} \mathcal{A}(\mathcal{A})$ will be used to refer to the set of all DFAs derivable from a automaton $\mathcal{A}$. Coste and Fredouille remark that even though $\overline{\mathcal{L A T}}(\mathcal{M C \mathcal { A }})$ is a lattice of partitions, $\mathcal{D} \mathcal{F} \mathcal{A}(\mathcal{A})$ is not a lattice under the $\preceq$ relation, and using the term 'lattice' in this context is incorrect CF03.

### 3.3 Some Important Results

Property 1 (dlH05). If $\mathcal{A} / \pi_{a} \preceq \mathcal{A} / \pi_{b}$ then $L\left(\mathcal{A} / \pi_{a}\right) \subseteq L\left(\mathcal{A} / \pi_{b}\right)$. That is, if the quotient automaton $\mathcal{A} / \pi_{b}$ derives from $\mathcal{A} / \pi_{a}$, every string in $L\left(\mathcal{A} / \pi_{a}\right)$ is also in $L\left(\mathcal{A} / \pi_{b}\right)$.

Proof. Let $\mathcal{A} / \pi_{a}=\left\langle\Sigma, Q, q_{0}, \delta, F\right\rangle, \mathcal{A} / \pi_{b}=\left\langle\Sigma, Q^{\prime}, q_{0}^{\prime}, \delta^{\prime}, F^{\prime}\right\rangle$, and consider any string $u=a_{1} a_{2} \ldots a_{n}$ in $L\left(\mathcal{A} / \pi_{a}\right)$. For any such string:

- Clearly there must be a path $q_{0} \xrightarrow{a_{1}} q_{1} \xrightarrow{a_{2}} \cdots q_{n-1} \xrightarrow{a_{n}} q_{n}$ in $\mathcal{A} / \pi_{a}$.
- By the construction of $\pi_{b}$ there must also be a path $q_{0}^{\prime} \xrightarrow{a_{1}} q_{1}^{\prime} \xrightarrow{a_{2}} \cdots q_{n-1}^{\prime} \xrightarrow{a_{n}}$ $q_{n}^{\prime}$ in $\mathcal{A} / \pi_{b}$.
- Furthermore, $q_{0} \in B\left(q_{0}^{\prime}, \pi_{b}\right)$, and $q_{n} \in B\left(q_{n}^{\prime}, \pi_{b}\right) \in F^{\prime}$.
- It then follows that any string in $L\left(\mathcal{A} / \pi_{a}\right)$ is also in $L\left(\mathcal{A} / \pi_{b}\right)$.

Theorem 1 ([DMV94]). Let $S_{+}$be a positive sample of any regular language $L$, and $\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F\right\rangle$ be an automaton accepting exactly $L$. If $S_{+}$is structurally complete with respect to $\mathcal{A}$, then $\mathcal{A}$ is in $\mathcal{L A T}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$.

Proof. We first show that there is some partition $\pi$ such that $\mathcal{M C \mathcal { C }}\left(S_{+}\right) / \pi$ is isomorphic to $\mathcal{A}$, and secondly we show that the final states in $\mathcal{M C \mathcal { A }}\left(S_{+}\right) / \pi$ are exactly the final states in $\mathcal{A}$.

- Let $S_{+}=\left\{s_{1}, s_{2}, \ldots, s_{m}\right\}$ be a sample which is structurally complete with respect to $\mathcal{A}$ where $m=\left|S_{+}\right|$.
- Let $s_{i}=a_{1}^{i} a_{2}^{i} \cdots a_{\left|s_{i}\right|}^{i}$ where $1 \leq i \leq m$ be a string in $S_{+}$.
- For each string $s_{i} \in S_{+}$we can construct a state sequence $p^{i}=q_{0}^{i}, q_{1}^{i}, \cdots, q_{\left|s_{i}\right|}^{i}$ consisting of $\left(\left|s_{i}\right|+1\right)$ states where each $q_{0}^{i}$ corresponds to the starting state in $\mathcal{A}$, the states $q_{\left|s_{i}\right|}^{i}$ each correspond to some final state in $\mathcal{A}$, and $q_{j+1}^{i} \in$ $\delta\left(q_{j}^{i}, a_{j+1}^{i}\right)$ for $1 \leq i \leq m$ and $0 \leq j \leq\left|s_{i}\right|-1$. Observe how this sequence is an acceptance (path) of the string $s_{i} \in S_{+}$by $\mathcal{A}$.
- Construct $\mathcal{M C \mathcal { A }}\left(S_{+}\right)=\left\langle\Sigma, \bar{Q}, \bar{q}_{0}, \bar{\delta}, \bar{F}\right\rangle$ as follows:
- Create one initial state $\bar{q}_{0}$ embedding the states $q_{0}^{i}$ of each sequence $p^{i}$ for $s_{i} \in S_{+}$.
- The transition function $\bar{\delta}$ is defined as $q_{j+1}^{i} \in \delta\left(q_{j}^{i}, a_{j+1}^{i}\right) \Longleftrightarrow \bar{q}_{j+1}^{i} \in$ $\delta\left(\bar{q}_{j}^{i}, a_{j+1}^{i}\right)$ for $1 \leq i \leq m$ and $0 \leq j \leq\left(\left|s_{i}\right|-1\right)$. In other words, there is a transition in $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$for each 'hop' in our sequences $p^{i}$.
$-\bar{F}$ is constructed as $\bar{F}=\left\{\bar{q}_{\left|s_{i}\right|}^{i} \mid 1 \leq i \leq m\right\}$.
- We next define a function $\phi$ that maps the states in $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$to those in $\mathcal{A}$ as follows:
$-\phi: \bar{Q} \rightarrow Q$ where $\phi\left(q_{j}^{i}\right)=q \in Q$ whenever $q=q_{j}^{i}$ for $1 \leq i \leq m$ and $0 \leq j \leq\left|s_{i}\right|$.
- Define a partition $\pi$ over the states of $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$as follows: $B(\bar{q}, \pi)=$ $B^{\prime}\left(\bar{q}^{\prime}, \pi\right)$ iff $\phi(\bar{q})=\phi\left(\bar{q}^{\prime}\right)$. In other words, two states are in the same partition if they map to the same state in $\mathcal{A}$.
- Result 1: This definition of $\pi$ implies that $\mathcal{M C \mathcal { A }}\left(S_{+}\right) / \pi$ is isomorphic to $\mathcal{A}$ since the structural completeness of $S_{+}$guarantees that the transitions of $\mathcal{M C \mathcal { A }}\left(S_{+}\right) / \pi$ correspond exactly to those in $\mathcal{A}$.
- The final states $\widehat{F}$ in $\mathcal{M C \mathcal { A }}\left(S_{+}\right) / \pi$ are given by $\widehat{F}=\{B \in \pi \mid \exists \bar{q} \in B: \phi(\bar{q}) \in F\}$. This means that the final states in the quotient automaton are those blocks which have one of the states corresponding to a final state in $\mathcal{A}$.
- The second part of our definition of structural completeness ensures that $\forall q \in F, \exists i$ such that $q_{\left|s_{i}\right|}^{i}=q$ for $i \leq i \leq m$ (i.e. for every final state in $\mathcal{A}$, there is at least one string in $S_{+}$that reaches it).
- Result 2: It therefore follows that $\widehat{F}$ is exactly equal to $F$.
- Results 1 and 2 complete the proof.

Theorem 2 ([DMV94]). Let $S_{+}$be a positive sample of any regular language $L$, and $\mathcal{A}(L)=\left\langle\Sigma, Q, q_{0}, \delta, F\right\rangle$ be the canonical automaton that accepts $L$. If $S_{+}$is structurally complete with respect to $\mathcal{A}(L)$, then $\mathcal{A}(L)$ is in $\mathcal{L A} \mathcal{T}\left(\mathcal{P} \mathcal{T}\left(S_{+}\right)\right)$.

Proof. This proof is similar to the previous one but adapted to accomodate that both the canonical automaton and the prefix tree acceptor are deterministic.

- Let $S_{+}=\left\{s_{1}, s_{2}, \ldots, s_{m}\right\}$ be a sample which is structurally complete with respect to $\mathcal{A}(L)$ where $m=\left|S_{+}\right|$.
- Let $s_{i}=a_{1}^{i} a_{2}^{i} \cdots a_{\left|s_{i}\right|}^{i}$ where $1 \leq i \leq m$ be a string in $S_{+}$.
- For each string $s_{i} \in S_{+}$we can construct a state sequence $p^{i}=q_{0}^{i}, q_{1}^{i}, \cdots, q_{\left|s_{i}\right|}^{i}$ consisting of $\left(\left|s_{i}\right|+1\right)$ states where each $q_{0}^{i}$ corresponds to the starting state in $\mathcal{A}(L)$, the states $q_{\left|s_{i}\right|}^{i}$ each correspond to some final state in $\mathcal{A}(L)$, and $q_{j+1}^{i}=\delta\left(q_{j}^{i}, a_{j+1}^{i}\right)$ for $1 \leq i \leq m$ and $0 \leq j \leq\left|s_{i}\right|-1$. Observe how this sequence is an acceptance (path) of the string $s_{i} \in S_{+}$by $\mathcal{A}(L)$, and how this sequence is unique because the canonical automation is deterministic.
- Construct $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)=\left\langle\Sigma, \bar{Q}, \bar{q}_{\lambda}, \bar{\delta}, \bar{F}\right\rangle$ as follows:
- Define an equivalence between states $q_{j}^{i}$ in a sequence $p^{i}$ of $s_{i} \in S_{+}$ as: $q_{j}^{i}=q_{k}^{l}$ iff $\operatorname{Pref}\left(q_{j}^{i}\right)=\operatorname{Pref}\left(q_{k}^{l}\right)=\omega$ and denote each grouping of equivalent states by $\bar{q}_{\omega}$.
- The initial state in $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$is $\bar{q}_{\lambda}$.
$-q_{j+1}^{i}=\delta\left(q_{j}^{i}, a_{j+1}^{i}\right) \Longleftrightarrow \bar{q}_{\omega \cdot a_{j+1}^{i}}=\delta\left(\bar{q}_{\omega}, a_{j+1}^{i}\right)$ where $1 \leq i \leq m$ and $0 \leq j \leq\left(\left|s_{i}\right|-1\right)$.
$-\bar{F}=\left\{\bar{q}_{\omega} \mid \omega \in S_{+}\right\}$.
- We next define a function $\phi$ that maps the states in $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$to those in $\mathcal{A}(L)$ as follows:
$-\phi: \bar{Q} \rightarrow Q$ where $\phi\left(\bar{q}_{\omega}\right)=q=\delta\left(q_{0}, \omega\right)$.
- Define a partition $\pi$ over the states of $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$as follows: $B\left(\bar{q}_{\omega}, \pi\right)=$ $B^{\prime}\left(\bar{q}_{\omega^{\prime}}, \pi\right)$ iff $\phi\left(\bar{q}_{\omega}\right)=\phi\left(\bar{q}_{\omega^{\prime}}\right)$. In other words, two states are in the same partition if they map to the same state in $\mathcal{A}(L)$.
- Result 1: This definition of $\pi$ implies that $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right) / \pi$ is isomorphic to $\mathcal{A}(L)$ since the structural completeness of $S_{+}$guarantees that the transitions of $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right) / \pi$ correspond exactly to those in $\mathcal{A}(L)$.
- The final states $\widehat{F}$ in $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right) / \pi$ are given by $\widehat{F}=\{B \in \pi \mid \exists \bar{q} \in B: \phi(\bar{q}) \in F\}$. This means that the final states in the quotient automaton are those blocks which have one of the states corresponding to a final state in $\mathcal{A}$.
- The second part of our definition of structural completeness ensures that $\forall q \in F, \exists i$ such that $q_{\left|s_{i}\right|}^{i}=q$ for $i \leq i \leq m$ (i.e. for every final state in $\mathcal{A}$, there is at least one string in $S_{+}$that reaches it).
- Result 2: It therefore follows that $\widehat{F}$ is exactly equal to $F$.
- Results 1 and 2 complete the proof.

Theorem 3 ([DMV94]). If an automaton $\mathcal{A}$ belongs to $\mathcal{L A} \mathcal{T}\left(\mathcal{M C A}\left(S_{+}\right)\right)$then $S_{+}$ is structurally complete with respect to $\mathcal{A}$.

Proof. Consider:

- By construction, $S_{+}$is clearly structurally complete with respect to $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$.
- By definition of $\mathcal{L A} \mathcal{T}$, any automaton $\mathcal{A} / \pi$ derived from $\mathcal{M C \mathcal { C }}\left(S_{+}\right)$for some $\pi$ is in $\mathcal{L A} \mathcal{T}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$. Additionally, $S_{+}$will be structurally complete with respect to any such derived quotient automaton by definition of how quotient automata are constructed.

Theorem 4 ([DMV94). Let $S_{+}$be a positive sample. Let $\mathbb{A}$ be the set of automata such that $S_{+}$is structurally complete with respect to each $\mathcal{A} \in \mathbb{A}$. The set $\mathbb{A}$ is equal to $\mathcal{L A} \mathcal{T}\left(\mathcal{M C A}\left(S_{+}\right)\right)$.

Proof. We will use the standard procedure of showing that two sets $A$ and $B$ are exactly equal by determining that any element in $A$ is in $B$, and that any element in $B$ is in $A$. The results of Theorem 1 together with those of Theorem 3 are sufficient to show this:

- By Theorem 1, if $S_{+}$is structurally complete with respect to an automaton $\mathcal{A}$ then $\mathcal{A}$ is in $\mathcal{L A} \mathcal{T}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$.
- By Theorem 3 if an automaton $\mathcal{A}$ belongs to $\mathcal{L A} \mathcal{T}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$then $S_{+}$is structurally complete with respect to $\mathcal{A}$.

Property $2\left([\right.$ DMV94 $) . \mathcal{L A} \mathcal{T}\left(\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)\right) \subseteq \mathcal{L} \mathcal{A} \mathcal{T}\left(\mathcal{M C} \mathcal{A}\left(S_{+}\right)\right)$.
Proof. $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$may be defined (see Section 3.1.4) as a quotient automaton of $\mathcal{M C A}\left(S_{+}\right)$where $\pi$ is given by $B(q, \pi)=B\left(q^{\prime}, \pi\right)$ iff $\operatorname{Pref}(q)=\operatorname{Pref}\left(q^{\prime}\right)$.

## Observation

It should be noted that $\mathcal{L A} \mathcal{T}\left(\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)\right)$is, in general, properly included in $\mathcal{L A} \mathcal{T}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$, so searching in it will result in a reduction in the size of our search space.

Property 3 ([DMV94]). There exist positive samples $S_{+}$for which some languages can only be represented as NFAs in $\mathcal{L A} \mathcal{T}\left(\mathcal{M C A}\left(S_{+}\right)\right)$.

Proof. We show this by presenting a case. Consider the NFA shown in Figure 3.6 (ii) which accepts the language $\left(b a^{*} a\right)^{*}$. The sample $S_{+}=\{b a a\}$ is structurally complete with respect to this NFA but not with the minimal DFA (iii) for the same language.


Figure 3.6: $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$, minimal NFA, and minimal DFA for $S_{+}=\{b a a\}$.

## Observation

As a consequence of Property 3, some languages cannot be identified from $S_{+}$if we restrict our searches to DFAs only. This is because $S_{+}$may be structurally complete with respect to the minimal NFA but not with respect to the minimal DFA for the same language.

Property 4 ([DMV94). There exist positive samples $S_{+}$for which the set of languages in $\mathcal{L A} \mathcal{T}\left(\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)\right)$is properly included in the set of languages in $\mathcal{L A T}\left(\mathcal{M C A}\left(S_{+}\right)\right)$.

Proof. From Property 2, we know that the set of languages in $\mathcal{L A T}\left(\mathcal{P T} \mathcal{A}\left(S_{+}\right)\right)$ is included in the set of languages in $\mathcal{L A} \mathcal{T}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$. We now need to show that in some cases this inclusion is proper, and there may exist automata in $\mathcal{L A} \mathcal{T}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$that have no equivalent automaton in $\mathcal{L A} \mathcal{T}\left(\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)\right)$.

Consider the language $L=\left(b a+b(a a)^{*}\right)$ which may be represented by the NFA shown in Figure 3.7 (iii) and its corresponding canonical automaton $\mathcal{A}(L)$ shown in Figure 3.7(iv). Also consider the sample $S_{+}=\{b a, b a a\}$ of the language, and $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$shown in Figure 3.7 (i) and $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$shown in Figure 3.7 (ii).

The NFA may be derived from $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$because $S_{+}$is structurally complete with respect to it, and moreover it is obvious by merging the states $\left(q_{1}, q_{3}\right)$ in $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$. Note, however, that it is impossible to derive the NFA from $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$. Also, since $S_{+}$is structurally complete with respect to the NFA but not with respect to $\mathcal{A}(L)$, no derivation can exist to obtain it from $\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)$.

### 3.4 The Border Set and its Properties

Recall that $\mathcal{L A} \mathcal{T}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$, which also contains $\mathcal{U}$, contains all the automata that recognise languages for which $S_{+}$is a structurally complete sample. In this sense, it would seem that $\mathcal{U}$ may be a candidate target automaton, although it (and indeed many other automata) would clearly constitute an over-generalisation. To deal with this, we introduce the notion of a negative sample $S_{-} \subseteq\left(\Sigma^{*}-L\right)$ giving $S=\left\langle S_{+}, S_{-}\right\rangle$. We now require our inference task to identify an automaton


Figure 3.7: $\mathcal{M C \mathcal { A }}\left(S_{+}\right), \mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right), \mathrm{NFA}$, and minimal DFA for $S_{+}=\{b a, b a a\}$. which is consisten $\mathrm{t}^{4}$ with both $S_{+}$and $S_{-}$, as well as having the minimal number of state ${ }^{5}$

## Definition 3.5: Antistrings [DMV94]

An antistring $\overline{a s}$ in a lattice or space of automata is a set of automata where each element is not related by $\preceq$ to any other element in $\overline{a s}$. In other words, no automaton in the antistring may be derived from any other one.

## Definition 3.6: Maximal Depth [DMV94]

An automaton $\mathcal{A}$ is said to be at its maximal depth in a lattice or space if there is no other automaton $\mathcal{A}^{\prime}$ which may be derived from $\mathcal{A}$ such that $L\left(\mathcal{A}^{\prime}\right) \cap S_{-}=\emptyset$. In other words, we may not derive any other automaton which is also compatible with the positive and negative sample.

[^8]
## Definition 3.7: The Border Set in an MCA [DMV94]

The border set $\mathcal{B S}_{\left.S_{+}, S_{-}\right\rangle}\left(\mathcal{M C A}\left(S_{+}\right)\right)$of a lattice is the antistring in $\mathcal{L A T}$ ( $\left.\mathcal{M C} \mathcal{A}\left(S_{+}\right)\right)$where each element is at its maximal depth. For brevity, and when the context is clear or irrelevant, we will refer to the border set by $\mathcal{B} \mathcal{S}_{\mathcal{M C A}}$.

Definition 3.8: The Border Set in a PTA, adapted from [DMV94] and CF03]

The border set $\mathcal{B S} \mathcal{S S}_{\left\langle S_{+}, S_{-}\right\rangle}\left(\mathcal{P} \mathcal{A} \mathcal{A}\left(S_{+}\right)\right)$of a space of DFAs is the antistring in $\mathcal{D F} \mathcal{A}\left(\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)\right)$where each element is at its maximal depth. For brevity, and when the context is clear or irrelevant, we will refer to the border set by $\mathcal{B} \mathcal{S}_{\mathcal{P T A} \mathcal{A}}$.

It is now apparent that the purpose of $S_{-}$is to control generalisation, and the minimal consistent automaton would be in the border set. Any automata 'shallower' than the border set would surely be an under-generalisation, and any automata 'deeper' than the border set would be an over-generalisation since they are inconsistent with $S_{-}$.

Property 5 ([DMV94]). $\mathcal{B} \mathcal{S}_{\mathcal{P} \mathcal{T A}_{\mathcal{A}}} \subseteq \mathcal{B} \mathcal{S}_{\mathcal{M C A}}$ for some $S=\left\langle S_{+}, S_{-}\right\rangle$.
Proof. From Property2, we already know that any automaton in $\mathcal{D F} \mathcal{A}\left(\mathcal{P T} \mathcal{A}\left(S_{+}\right)\right)$ is in $\mathcal{L A \mathcal { A }}\left(\mathcal{M C \mathcal { A }}\left(S_{+}\right)\right)$. Now we wish to show that every automaton in $\mathcal{B} \mathcal{S}_{\mathcal{P T \mathcal { A }}}$ is in $\mathcal{B} \mathcal{S}_{\mathcal{M C A}}$. Suppose an automaton $\mathcal{A}$ exists that is in $\mathcal{B} \mathcal{S}_{\mathcal{P} \mathcal{T A}_{\mathcal{A}}}$ but not in $\mathcal{B} \mathcal{S}_{\mathcal{M C A}}$. Since $\mathcal{A} \in \mathcal{B} \mathcal{S}_{\mathcal{P T \mathcal { A }}}$, then $\mathcal{A}$ is consistent with $S$, and no further derivations are possible from $\mathcal{A}$ which would be consistent with $S$. If $\mathcal{A} \notin \mathcal{B} \mathcal{S}_{\mathcal{M C A}}$, then $\mathcal{A}$ is either inconsistent with $S$, or some derivation must exist whose resulting automaton would be consistent with $S$. This contradicts our assumption.

## Observation

For emphasis, we note that the opposite result $\mathcal{B S} \mathcal{M C \mathcal { A }} \subseteq \mathcal{B} \mathcal{S}_{\mathcal{P} \mathcal{A} \mathcal{A}}$ is clearly not always true. Since an automaton $\mathcal{A} \in \mathcal{B S} \mathcal{M C A}_{\mathcal{A}}$, while being consistent with $S$ as well as having no further derivations possible, might not be in $\mathcal{L A} \mathcal{T}\left(\mathcal{P} \mathcal{T} \mathcal{A}\left(S_{+}\right)\right)$in the first place.

Property 6 (DMV94). There may be several distinct languages represented by the automata belonging to $\mathcal{B S}_{\mathcal{M C A}}$.

Proof. We prove this property by presenting an example. Consider the sample $S_{+}=\{b a a\}$ and $S_{-}=\{b b\}$. The maximal canonical automaton constructed from $S_{+}$is shown in Figure 3.8 (i). The two automata shown in Figures 3.8 (ii) and (iii) belong to the border set and represent the distinct languages $(a \mid b a)^{*}$ and $((a \mid b) a)^{*}(a \mid b)$ respectively.

(i)

(ii)

(iii)

Figure 3.8: $\mathcal{M C \mathcal { A }}\left(S_{+}\right)$constructed for $S_{+}=\{b a a\}$, and two DFAs in the border set.

Property 7 ([DMV94]). There may exist NFAs belonging to $\mathcal{B S} \mathcal{M C \mathcal { A }}$ that contain fewer states than the minimal consistent DFAs.

Proof. This follows from the fact that given a minimal consistent DFA recognising a language, there may exist NFAs having fewer states that recognise the same language.

Property 8 ([DMV94]). All DFAs in $\mathcal{B S}_{\mathcal{M C A}}$ are minimal for the language they accept.

Proof. Let $\mathcal{A} \in \mathcal{B S} \mathcal{S C \mathcal { A C A }}_{\mathcal{A}}$ be a DFA accepting the language $L$ which is consistent with a positive and negative sample. Assume that $\mathcal{A}$ is not isomorphic to the canonical automaton $\mathcal{A}(L)$, then:

- Both $\mathcal{A}$ and $\mathcal{A}(L)$ are consistent with the sample.
- DFA minimisation involves merging non-distinguishable states defined by some partitioning $\pi$.
- This means that $\exists \pi \mid \mathcal{A} / \pi=\mathcal{A}(L)$.
- It then follows that $\mathcal{A} \ll \mathcal{A}(L)$, and $\mathcal{A}(L)$ would be deeper in the lattice than $\mathcal{A}$.
- This is impossible because $\mathcal{A}$ is in the border set and there cannot be any other consistent DFA derived from it that is deeper.


## Observation

This property also holds for the DFAs in $\mathcal{B} \mathcal{S}_{\mathcal{P} \mathcal{A} \mathcal{A}}$ due to Property 5 .

### 3.5 Summary

In this chapter, we have described derived automata, quotient automata, structural completeness, prefix tree acceptors, lattices, search spaces, and the border set. These concepts are the basis of many DFA learning algorithms. We also covered some important results from DMV94 and dlH05 regarding the search space of DFA learning. These include:

- Merging states in an automaton $\mathcal{A}$ to obtain $\mathcal{A}^{\prime}$, implies that the language recognised by $\mathcal{A}$ is a subset of the language recognised by $\mathcal{A}^{\prime}$. Merging generalises an automaton.
- If a positive sample is structurally complete with respect to a DFA, then the DFA is guaranteed to be in the lattice constructed from the maximal canonical automaton of the sample.
- If a positive sample is structurally complete with respect to the canonical (minimal) DFA, then the canonical automaton is guaranteed to be in the search space $\mathcal{D F} \mathcal{A}$ constructed from the prefix tree acceptor of sample.
- If an automaton belongs to the lattice constructed from the maximal canonical automaton of a positive sample, then the sample is structurally complete with respect to that automaton.
- Consider the set of all automata with which a positive sample is structurally complete with. That set of automata is exactly equal to the lattice constructed from the maximal canonical automaton of that sample.
- The search space constructed starting from the prefix tree acceptor of a positive sample is included in the lattice constructed from the maximal canonical automaton of that sample.
- The border set in the space of DFAs constructed from the prefix tree acceptor of a sample is included in the border set of the lattice constructed from the maximal canonical automaton of the same sample.
- The automata in a border set may represent distinct languages.
- All the DFAs in a border set are minimal for the language they recognise.


## Chapter 4

## A Survey of State Space Search

## Algorithms

As discussed in the previous chapter, the task of identifying a target DFA can be construed as a search in a space of DFAs starting from some initial hypothesis until we reach our goal. In this chapter, we cover a variety of state space search algorithms which involve searching for a goal in a space that is organised as a tree or a graph. Focusing on properties such as scaleability, completeness, and admissibility, we will pay special attention to the search techniques which are relevant to several DFA learning algorithms including Oncina and García's RPNI [OG92, Juillé and Pollack's SAGE [JP98b], and Lang's Ed-Beam [Lan98. We suggest [RN03 as a good introduction on state space search as well as CLRS09] for a comprehensive description and analysis of these algorithms.

## Note

Pseudo-code for all the algorithms referred to in this chapter may be found in Appendix C

### 4.1 Preliminaries

### 4.1.1 Costs

When searching through a state space (the set of all states reachable from an initial state by any sequence of actions [RN03]) structured as a tree or graph, moving from one nod $\underbrace{1}$ to another one involves traversing a path. This is analogous to executing an action or a plan. These paths have a cost associated with them which we shall denote by $g(n)$, where $n$ is the node our path is leading us to from the initial state. Of course, if $n$ is the initial state itself, $g(n)=0$ since the cost of travelling between a node and itself is zero. We illustrate the cost of an action in Figure 4.1. The task of a state space search algorithm usually involves finding the minimum cost path starting from an initial state to a goal state (the cost being the sum of the edge weights in the path).


Figure 4.1: Path costs to the states $s 1$, and $s 2$ and $s 3$.

### 4.1.2 Time and Space Complexity, Completeness, and Admissibility

The time and space complexity of state space search algorithms is studied in terms of the branching factor $b$ of the tree, and the depth $d$ of the goal we are

[^9]seeking. Additionally, the properties of search completeness and admissibility are important RN03:

- Completeness means that the algorithm will return a solution (the goal) if one exists.
- Admissibility means that the algorithm is guaranteed to return the optimal solution. Sometimes admissibility is referred to as optimality.


### 4.1.3 Open and Closed Lists

Typically, graph search algorithms maintain internal data structures that store lists of nodes which are yet to be visited and/or nodes which have already been visited. The data structure that stores the list of nodes to be visited is called the open list and is used by the algorithm to determine which nodes to visit next CLRS09. Certain spaces may involve nodes which are repeated or involve cycles (the state space could very well be a graph instead of a tree). These repetitions or cycles may cause an algorithm to enter infinite loops or cause the search to grow exponentially in the size of the graph as shown in Figure 4.2. Because of this, some implementations maintain a closed list of nodes which have already been visited so as not to expand them again. [RN03] suggest three countermeasures to deal with such repetitions in increasing order of effectiveness and computational expense:

1. When considering the successors of a node (we are expanding the children of the node), ignore the ones which are identical to the parent since we have already processed it. While this approach is not sufficient in the general case, there are several applications where this method is enough.
2. Do not create paths that contain cycles. This requires that the current path is remembered (usually using a hash table) and reject children that already appeared in the path.
3. Remember all the states that we have already visited and do not visit them again. Special care must be taken as this requires $b^{d}$ memory to store the
visited nodes (where $b$ and $d$ are the branching factor and depth, respectively).


Figure 4.2: Failure to implement countermeasures can cause searches to consider an exponential number of nodes.

### 4.2 Blind Search Algorithms

### 4.2.1 Depth-First Search

Depth-first search (DFS) is a tree or graph search algorithm that starts from some initial state and explores as far along a path as possible before backtracking. This process is repeated until a goal node is found. If the depth is bounded, DFS will eventually enumerate all the nodes in the tree or graph. DFS may be expressed as in Algorithm C.1 (in Appendix C). We can now make some observations about the method. The algorithm is not complete when the state space is infinite. In fact, DFS will only be complete when the state space is finite and we maintain a closed list to protect from the infinite expansion of nodes along a path (because of cycles). DFS is certainly not admissible as it is not complete. Since we are enumerating all the nodes in the tree, the time complexity of DFS is $b^{d}$. The space complexity is $b \times d$ since we are only storing information related to one single path down the tree. This is illustrated in Figure 4.3 below.


Figure 4.3: The space complexity of DFS is $b \times d$. The open list is coloured.

### 4.2.2 Breadth-First Search

Breadth-first search ( $\overline{\mathrm{BFS}}$ ) is similar to DFS but rather than expanding nodes in a depth-wise manner, it expands nodes on a 'level-by-level' basis. As far as the algorithm is concerned, it is identical to DFS with the exception that expanded nodes are added to the back of the open list rather than at the front. In other words, BFS uses a queue as an open list rather than a stack. This is shown in Algorithm C.2. Since the goal state must exist at some level in the state space, BFS will eventually reach it and is hence complete. Furthermore, if the costs of the actions in the state space are all the same (i.e. all the edge weights have the same value), BFS is admissible as it will terminate at the shallowest goal node (i.e. cheapest path cost) in the space. A case where BFS is not admissible because of varying cost values is shown in Figure 4.4. The time complexity of BFS is $b^{d}$ and so is the space complexity: the open list in BFS maintains all the nodes at a given level, and the level $d$ containing the goal has $b^{d}$ nodes.

### 4.2.3 Iterative Deepening

BFS is both complete and admissible when all the costs are the same. This is at the expense of exponential time and space complexity. Meanwhile DFS has the promise of linear space complexity at the cost of completeness and admissibility. The objective of iterative deepening search (IDS) is to combine aspects of both methods to give us the completeness and admissibility of BFS but using much less memory. The general idea is to perform a DFS in the state space up to some level


Figure 4.4: BFS is not admissible when action costs are different.
$n \leq d$. If we do not find the goal, we reiterate by performing another DFS up to a deeper level $(n+1) \leq d$. This process is repeated until the goal is found.

We can see that at every iteration $n$ we would have to incur the expense of redoing all the work we have already done to reach iteration $(n-1)$. While this reasoning is correct, we note that it is not so detrimental. Consider a state space with a branching factor $b=2$. At level $n$, the algorithm has to contend with $2^{n}$ nodes. This necessitates redoing all the work at level $(n-1)$ which contains $2^{n-1}$ nodes. This, in turn, necessitates redoing the work at level $(n-2)$ which contains $2^{n-2}$ nodes, and so on until we reach the root node. This means that when working on level $n$, the sum of the nodes in all the preceding levels $(n-1),(n-2), \ldots, 1$ is at most twice the number of nodes in level $n$. This overhead is then bounded by a constant factor, and may be visualised in Figure 4.5 below. Extending this, we may see that this overhead becomes even less significant when the branching factor is higher. The pseudo-code for IDS is given in Algorithm C.3, and we conclude our discussion by making the following observations:

- The time complexity, like BFS, is $b^{d}$. However, like DFS, the space complexity is $b \times d$.
- Among other applications, IDS is very useful in online scenarios since early iterations execute very quickly giving us results immediately with further refinements as the algorithm iterates on.
- Like BFS, IDS is complete and only admissible when the action costs (edge weights) are the same.
- Unless we implement cycle detection or maintain a closed list, IDS may be trapped in an infinite loop.


Figure 4.5: The overhead of IDS is bound by a constant factor.

### 4.2.4 Uniform-Cost Search

If the cost of our actions is not the same, then algorithms such as BFS or IDS will no longer be admissible. This can be dealt with by using a min-queue/heap (described in [For64]) as an open list rather than a plain queue or stack where the nodes are kept sorted by their lowest cost $g$. This modification is called uniform-cost search (UCS), and we note that its behaviour is identical to Dijkstra's single-source shortest path algorithm [CLRS09. The pseudo-code for the method is given in Algorithm C.4, and we proceed to illustrate its behaviour using the simple example shown in Figure 4.6. The min-queue is ordered by cost $g$ which is shown in superscript for each node in the example:

- $Q=\left\{A^{(0)}\right\}$.
- $Q$ is not empty, pop $A, A$ is not goal.
- Expand $A, Q=\left\{B^{(1)}, C^{(10)}\right\}$.
- $Q$ is not empty, pop $B, B$ is not goal.
- Expand $B, Q=\left\{E^{(2)}, D^{(3)}, C^{(10)}\right\}$.
- $Q$ is not empty, pop $E, E$ is not goal.
- Expand $E, Q=\left\{D^{(3)}, C^{(10)}\right\}$.
- $Q$ is not empty, pop $D, D$ is not goal.
- Expand $D, Q=\left\{F^{(4)}, C^{(10)}\right\}$.
- $Q$ is not empty, pop $F, F$ is goal. Done.


Figure 4.6: An example to illustrate a uniform-cost search. The goal states $C$ and $F$ are coloured.

Uniform cost search is complete (since all nodes are considered) and it is also admissible [CLRS09]. Admissibility follows from the fact that we are always dequeueing the node having the minimum path cost. If there are two goal nodes $n$ and $n^{\prime}$ where $n$ has a lower cost than $n^{\prime}$, then $n$ would be popped off and discovered before the suboptimal goal $n^{\prime}$. Unfortunately, while UCS is both complete and admissible, both the time and space complexities of the algorithm are $b^{d}$.

### 4.3 Informed Search Algorithms

### 4.3.1 Greedy Search

Greedy searching involves using a monotonic heuristic to guide a search. An example of a greedy search applied to the travelling salesman problem would be to always select the closest unvisited city from the current one. Some characteristics of greedy searching include:

- It can get trapped oscillating between nodes unless a closed list is used.
- The quality of the results is arbitrary.
- If we have a perfect heuristic (which is unlikely) the algorithm will move directly and optimally to the goal.

While such an algorithm is very efficient both in terms of time and space, it is easy to see how, in practice, the technique is neither complete nor admissible. We illustrate this using a simple example where we wish to find the lowest-cost path in the tree shown in Figure 4.7. In this example, the greedy search immediately makes a mistake by moving to the node $C$ instead of the node $B$ because it has a lower cost. This mistake makes it impossible to identify the lowest-cost path $A \rightarrow B \rightarrow D$.

Greedy search


Path cost found $=20+1=21$


Figure 4.7: Greedy search is neither complete nor admissible.

### 4.3.2 A* Search

$A^{*}$ search is an extension to UCS that uses a heuristic to make the search more efficient. Unlike the case we have seen earlier, we note that A* still guarantees completeness and admissibility under attainable conditions [CLRS09. Recall that UCS will expand all the nodes having a $g$ value less than the cost of the optimal goal $g(o p t)$, where opt is the optimal goal node. This causes the algorithm to expand a great deal of nodes that 'point away' from the minimum cost goal.

These suboptimal paths will be eventually detected and discarded but only until after we have paid the computational expense of visiting them.

Let $h(n)$ be a heuristic function that estimates the cost of a search from the node $n$ to a goal. A heuristic is said to be admissible if it underestimates the true cost of an action - in other words, it is a lower bound RN03. The following are examples of scenarios and admissible heuristics from Rum13:

- Consider a map where each node located in Euclidian space represents a city, and the edges represent connections between adjacent cities. A possible heuristic is one which chooses the shortest distance between the node $n$ and the goal. Observe how this heuristic is admissible because it underestimates the true cost of travelling between $n$ and the goal (by triangle inequality).
- Consider a $3 \times 3$ tile game with 8 tiles and 1 empty slot. We wish to arrange our tiles in a 'winning' arrangement (the goal). A heuristic would be to count the number of misplaced tiles in our current state (tile arrangement). This is admissible because a misplaced tile would need to move at least one slot before finding its correct position in the winning arrangement.
- Consider the same $3 \times 3$ tile game. Yet another admissible heuristic would be the sum of the Manhattan distances between all the tiles at their current position and their target position in the winning arrangement. This heuristic is still an underestimate because Manhattan distance allows for tiles to slide over each other which is clearly not possible in the real world (i.e. the Manhattan distance is a best case scenario). While this is also an underestimate, it is an even stronger one than our previous tile-counting method. In search terminology, we say that this heuristic dominates the other. Dominant admissible heuristics result in more efficient implementations of the $\mathrm{A}^{*}$ algorithm.

Recall that UCS is designed to minimise the objective function $f(n)=g(n)$. In A* we instead minimise the objective function $f(n)=g(n)+h(n)$. The algorithm for $A^{*}$ is identical to that for UCS but uses the revised objective function. We conclude by making the following observations:

- UCS can be thought of as a special case of A* where $\forall n: h(n)=0$. Note how this is a perfectly valid (but ineffective) admissible heuristic (underestimate).
- UCS expands all the nodes having $f(n)=g(n)+0<g(o p t)$ whereas A* expands all the nodes having $f(n)=g(n)+h(n)<g(o p t)$. In general, there will be much fewer nodes in the set $g(n)+h(n)<g(o p t)$ than there will be in $g(n)+0<g(o p t)$ making A* more efficient.
- Using the same argument as BFS, we can show that $\mathrm{A}^{*}$ is complete, and if the heuristic is admissible, it can be shown that A* is admissible too [RN03.
- In the worst case, we could be given a heuristic $h=0$ degenerating A* to UCS. In this case, both the time and space complexity would be $b^{d}$. We emphasise that the time and space requirements of A* are highly dependent on the quality of the heuristic.


### 4.3.3 Iterative Deepening A* Search

Like we did in IDS, we can use a similar iterative deepening strategy with $\mathrm{A}^{*}$ to give us iterative deepening $A^{*}$ (IDA*). Here, a number of iterations are performed, where, for each iteration, we perform a depth-first search cutting off a path when its total cost $f(n)=g(n)+h(n)$ exceeds a path length threshold which is guided by an admissible heuristic. This means that at every iteration of our depth-first search we will not blindly explore all nodes up to a given depth but we will instead only explore the most promising ones. The threshold is set up as follows:

- Initially the threshold is $f(s)=g(s)+h(s)=0+h(s)=h(s)$ where $s$ is the starting node.
- If the goal is not found within the threshold, set it to the minimum value of $f(n)$ which exceeded the previous threshold. Repeat until the goal is found.

The advantage of IDA* over A* is that it always requires $b \times d$ space while still being complete and admissible (when the heuristic is admissible). The pseudocode for IDA* is given in Algorithm C.5.

### 4.3.4 Branch and Bound Techniques

The idea behind branch and bound (BnB) is to make searches more efficient by discarding paths in a search which we know will give worse results than a currently known solution. Initially, BnB sets an upper bound $U$ to $\infty$ and discards any nodes whose $g(n)>U$. This means that, at the start, all paths in the search tree are considered. Whenever a goal is reached at a node $n^{\prime}$, the upper bound is revised to $U=g\left(n^{\prime}\right)$. This new upper bound follows since any open path that is more expensive will certainly lead to a worse goal than $n^{\prime}$. Whenever a goal whose cost is cheaper than our current $U$, we update it to reflect this, and go on until the search is complete. Consider the example shown in Figure 4.8 where we wish to find the lowest cost path from the initial node $S$ to the goal $G$. The following illustrates the first steps that a BnB approach would take in this example:

- Initial path in min-queue $Q=\left\{S^{(g=0)}\right\}$, and $U=\infty$.
- Dequeue path, and add successors $Q=\left\{S \rightarrow A^{(g=3)}, S \rightarrow B^{(g=4)}\right\}$.
- Dequeue path, and add successors $Q=\left\{S \rightarrow B^{(g=4)}, S \rightarrow A \rightarrow B^{(g=8)}, S \rightarrow\right.$ $\left.A \rightarrow C^{(g=9)}\right\}$.
- Dequeue path, and add successors $Q=\left\{S \rightarrow B \rightarrow D^{(g=6)}, S \rightarrow A \rightarrow\right.$ $\left.B^{(g=8)}, S \rightarrow B \rightarrow A^{(g=9)}, S \rightarrow A \rightarrow C^{(g=9)}\right\}$.
- Dequeue path, and add successors $Q=\left\{S \rightarrow A \rightarrow B^{(g=8)}, S \rightarrow B \rightarrow\right.$ $\left.A^{(g=9)}, S \rightarrow A \rightarrow C^{(g=9)}, S \rightarrow B \rightarrow D \rightarrow F^{(g=10)}, S \rightarrow B \rightarrow D \rightarrow C^{(g=14)}\right\}$.
- Continue dequeuing and adding successors until, eventually, we will dequeue the path $S \rightarrow B \rightarrow D \rightarrow F \rightarrow G^{(g=13)}$ which reaches the goal. We now update our upper bound $U$ from $\infty$ to 13, and discard any partial path that is more expensive than 13 since it will certainly lead to a more expensive path to the goal.
- We proceed and continuously revise $U$ and prune the space until we exhaust the queue.

It is also useful to introduce a lower bound to the idea of BnB to further prune the search space. Given an admissible heuristic $h(n)$, we can use $f(n)=g(n)+h(n)$


Figure 4.8: Branch and bound example.
as this lower bound like we have done in A* and IDA*. This means that the only modification we have to make to the scheme we previously described is to base the min-queue on $g(n)+h(n)$ rather than on $g(n)$ alone. The use of upper and lower bounds to restrict the search space this way is called admissible pruning.

### 4.3.5 Beam Search

Beam search is a search technique that may be implemented as a variant of algorithms such as BFS, UCS and A*. The primary advantage of beam search is that it fixes a bound on memory usage to make searches scale better for large problems. In its simplest form, beam search behaves like BFS where the algorithm expands all the children of a given node and trims that list to a fixed, predetermined size. The algorithm uses an objective function (whether it is $g(n)$ or $g(n)+h(n)$ ) to trim the open list to a size $k$ called the beam width of best partial results. Hopefully, our objective function, which is likely based on a heuristic, will permit our 'narrowing down' process to still allow for a path to reach a good goal. Of course, applying this trimming means that the search is neither complete nor admissible and we will call this inadmissible pruning. We illustrate this in Figure 4.9, and show the pseudo-code for beam searching in Algorithm C.6.

We can observe that setting the beam width to $k=1$ makes our search a plain greedy search, while setting $k=\infty$ makes our search complete. With regards to space complexity, we can see that the variables beam and set in Algorithm
C. 6 store $k$ nodes and $k \times b$ nodes respectively. This makes our overall ${ }^{2}$ space requirements $k \times b$. The time complexity depends on the data structure we use to implement the min-queue. Assuming that our queue and dequeue operations are implemented in $\mathcal{O}\left(\log _{2}(n)\right)$ time, our overall requirements would be $d \times b \times k \times$ $\log _{2}(k \times b)$. For completeness, we describe a version of beam search in Algorithm C. 7 which also maintains a closed list.

In conclusion, we refer the reader to variants of beam search such as beam stack search due to [ZH05] that combines backtracking with beam search to ensure both completeness and admissibility, divide-and-conquer beam stack search, and beam search using limited discrepancy backtracking [FK05].


Figure 4.9: An illustration of a beam search.

[^10]
### 4.4 Observations So Far

We can now list some useful observations that will guide us when developing our own state space search methods:

- We can modify our algorithms to avoid exponential space complexity while maintaining both completeness and admissibility.
- For an algorithm to be admissible it must be complete, and all complete state space search algorithms have exponential time complexity. In order to scale, we must prune the search space (such as what we did in beam search) hoping that we have not discarded our goal. The design of our heuristics is of paramount importance.


### 4.5 Estimating the Cost of Backtracking

In this section, we discuss the work of Knuth who proposes a Monte Carlo sampling approach to estimate the complexity of backtracking searches Knu74. The methods and results shown here are useful to us in two ways. Firstly, Juillé et al.'s competition-winning parallel beam search is inspired by this idea to drive its heuristic JP98b. Secondly, and more importantly, estimating properties of the search spaces we are dealing with will help us develop a deeper insight into the problem.

Knuth argues that it appears that many combinatorial optimisation problems can only be handled by exhaustively searching through all possibilities. Of course, the consequence of such an exhaustive search is that a slight increase in the size of the parameters of the problem will likely result in the total running time increasing by several orders of magnitude. The corollary is that slight optimisations could very well result in substantial improvements in performance. As an example, consider the game of Instant Insanity where we have four cubes each having one of the colours red, green, blue, and white on their sides as shown in Figure 4.10, The game is to arrange all the four cubes such that when put next to each other, the top, bottom, front, and back faces have each of the four colours appearing once. In Figure 4.10 below, the top face of the cubes has the colours $W R G W$
with the colour white appearing twice. This results in an invalid configuration (the bottom face is RRGB with red appearing twice which is another violation).


Figure 4.10: The Instant Insanity cubes.

A single six-sided cube may be arranged in 24 different ways laying on a table: six sides each laying on the table and four rotations for each. Given that there are four cubes in the game, a brute force search would require us to consider $24^{4}=331,776$ possible configurations. There are several strategies available to reduce the size of the search. For instance, we may observe that, by symmetry, each configuration is equivalent to one of seven others as shown in Figure 4.11.


Figure 4.11: Instant Insanity configurations are symmetrical.

In general, such a search problem is abstractly defined as finding all the sequences $\left(x_{1}, x_{2}, \ldots, x_{n}\right)$ that satisfy some property $P_{n}\left(x_{1}, x_{2}, \ldots, x_{n}\right)$.

For Instant Insanity $n=4$, the value $x_{k}$ is the placement (configuration) of the $k^{\text {th }}$ cube, and $P_{4}\left(x_{1}, x_{2}, x_{3}, x_{4}\right)$ is the property that all the four cubes have colours appearing only once on each side. In other words, a configuration $\left(x_{1}, x_{2}, x_{3}, x_{4}\right)$ that satisfies the condition $P_{4}$ is a sequence that represents a path down the search tree to a valid configuration.

So ( $x_{1}, x_{2}, \ldots, x_{k}$ ) for $0 \leq k<n$ represents a partial solution (incomplete path down the search tree), and $P_{k}\left(x_{1}, x_{2}, \ldots, x_{k}\right)$ is the satisfaction property for the partial solution up to $x_{k}$. We now wish that, if $P_{k+1}$ holds, then $P_{k}$ is implied. More specifically, $P_{k+1}\left(x_{1}, x_{2}, \ldots, x_{k}, x_{k+1}\right) \Longrightarrow P_{k}\left(x_{1}, x_{2}, \ldots, x_{k}\right)$. In other words, if a solution $\left(x_{1}, x_{2}, \ldots x_{k}\right)$ does not satisfy $P_{k}$, then no extension $\left(x_{1}, x_{2}, \ldots x_{k}, x_{k+1}\right)$ of it can possibly satisfy $P_{k+1}$. By induction, no extension $\left(x_{1}, x_{2}, \ldots x_{k}, \ldots, x_{n}\right)$ up to $n$ can satisfy the property $P_{n}$.

Suppose that the property $P_{k}$ is designed in such a way so that it always returns true when $k<n$, and only returns the correct true/false answer when $k=n$. This is tantamount to the weakest possible definition of $P_{n}$ and would result in a brute force search. On the other hand, the strongest possible definition would be when $P_{k}\left(x_{1}, x_{2}, \ldots, x_{k}\right)$ is true if and only if there exists a sequence $x_{k+1}, \ldots, x_{n}$ that satisfies $P_{n}\left(x_{1}, x_{2}, \ldots, x_{k}, x_{k+1}, \ldots, x_{n}\right)$. This means that the partial solution $\left(x_{1}, x_{2}, \ldots, x_{k}\right)$ is certainly a prefix of a valid solution. In the general case, stronger properties are much more expensive to compute and/or are not always evident. Unfortunately, the choice of these properties greatly affects the running time of a search, so we proceed by discussing Knuth's mechanism to estimate this cost Knu74.

Let $c\left(x_{1}, x_{2}, \ldots, x_{k}\right)$ be the cost associated with executing the partial solution up to $x_{k}$. So when $k=n$, we would obtain the cost for the complete solution. Knuth uses $c()$ to denote the cost when $k=0$, and uses $T$ to denote the tree of all possibilities explored by the algorithm. This gives $T=$ $\left\{\left(x_{1}, x_{2}, \ldots, x_{k}\right) \mid k \geq 0\right.$ and $P_{k}\left(x_{1}, x_{2}, \ldots, x_{k}\right)$ holds $\}$. The total cost of computing the entire backtracking tree is given by $\operatorname{cost}(T)=\sum_{t \in T} c(t)$. An estimation of the entire backtracking cost is obtained using a Monte Carlo approach based on random samplings of the tree. For each partial solution $\left(x_{1}, x_{2}, \ldots, x_{k}\right)$ for $0 \leq k<n$, a random extension $x_{k+1}$ is chosen from a set $S$ of candidates where $P_{k+1}$ holds, and the total cost is estimated as shown in Algorithm 4.1.

The algorithm will compute the estimate $C=c()+d_{0} c\left(x_{1}\right)+d_{0} d_{1} c\left(x_{1}, x_{2}\right)+\cdots+$ $d_{0} d_{1} \ldots d_{k-1} c\left(x_{1}, x_{2}, \ldots, x_{k}\right)$ where $d_{k}$ is a function of $\left(x_{1}, x_{2}, \ldots, x_{k}\right)$ which gives the number of extensions $x_{k+1}$ satisfying the property $P_{k+1}$ (i.e. $|S|$ in Algorithm 4.1). By the law of large numbers, if the mean result of multiple estimates is taken,

```
Algorithm 4.1 Estimating the cost of backtracking (adapted from [Knu74)
```

Input: A state space search tree.
Output: The estimated backtracking cost.
$k \leftarrow 0 \quad / /$ Current depth of random walk.
$D \leftarrow 1 \quad / /$ Auxiliary variable.
$C \leftarrow c() \quad / /$ The estimated cost.
$S \leftarrow$ All the extensions $x_{k+1}$ such that $P_{k+1}\left(x_{1}, x_{2}, \ldots, x_{k}, x_{k+1}\right)$ holds
$d_{k} \leftarrow|S| \quad / / d_{k}$ will be 0 when $k=n$ because S would be empty.
if $d_{k}=0$ then
return $C$
end if

9: $x_{k+1} \leftarrow \mathrm{~A}$ random element in $S \quad / /$ Random walk.
10: $D \leftarrow d_{k} \times D \quad / / D$ will be the product of all edge degrees encountered.
11: $C \leftarrow C+c\left(x_{1}, x_{2}, \ldots, x_{k+1}\right) \times D$
2: $k \leftarrow k+1$

13: go to 4 // Extend solution.
we will approach the true expected value (the cost of the backtracking search). We refer the reader to Knuth's technical report Knu74 for the proofs of the validity of this estimate.

### 4.5.1 Pitfalls and Refinements

Trying to infer characteristics about an entire search tree based on knowledge gathered by exploring a single path warrants skepticism. In fact, it is easy to construct an adversarial scenario to demonstrate failure. Consider an experiment proposed by Knuth which produces a result of 1 ( $99.9 \%$ of the time), and produces a result of $1,000,001$ ( $0.1 \%$ of the time). The expected value is 1,001 while sampling the space would almost invariably always yield a 1 . As illustrated by

Pur78, similar pathological cases would occur when the search tree is severely unbalanced ${ }^{3}$

A method to deal with this is to introduce a bias in our algorithm where, rather than selecting an extension $x_{k+1}$ at random, we choose one that "investigates more interesting parts of the tree". Specifically, we alter the procedure in step 9 in Algorithm 4.1 which selects an extension in $S$ with a probability of $\frac{1}{d_{k}}$ with one which selects the $j^{\text {th }}$ item in $S$ with a probability of $\operatorname{Pr}(j)$ where $\operatorname{Pr}(1)+$ $\operatorname{Pr}(2)+\ldots+\operatorname{Pr}\left(d_{k}\right)=1$. Of course, determining the choices for $\operatorname{Pr}(j)$ is a complex problem in itself but, irrespective of what the choices are, it can be shown that (i) the expected value of $C$ will still be $\operatorname{cost}(T)$, and (ii) one of the choices for $\operatorname{Pr}(j)$ will be the perfect choice. This idea forms the basis of importance sampling in statistics, and again we refer the reader to Knu74 for proofs regarding these properties as well as other techniques for improving estimates such as tree transformations and stratified sampling.

Out of all the methods attempted by the author, the one which was found to be the most promising in terms of results, efficiency, and complexity is when "stupid moves" which lead to underestimates are avoided. The gist is that all terminal nodes that have non-terminal siblings are discarded ensuring that we would never pick a node having $d_{k}=0$ unless all siblings also lead to such a terminal situation. Even so, special care must be taken. Consider a 'tall, skinny' binary tree having depth $d$ where every node has one terminal child and one nonterminal child except at the deepest level. Such a tree would have exactly $2 d+1$ nodes. Unfortunately, using our strategy of eliminating 'stupid moves' would make our estimator explore the deepest possible branch contributing an exponentially large term to the estimator which would then result in a gross overestimate.

### 4.5.2 Ideas for Improvement

In Pur78, Purdom suggests a method that allows for backtracking when the algorithm reaches leaf nodes. While his results were very promising, and, in certain

[^11]cases, represent an improvement by an order of magnitude, the method may require an exponential amount of backtracking to occur. Alternatively, Chen Che92 proposes a stratified sampling approach that reduces the variance in results over what Knuth obtains at the cost of requiring domain-specific knowledge about the problem at hand. In CV11, Cloteaux and Valentin extend Knuth's algorithm with Aldous and Vazirani's "Go with the Winners" GWTW method AV94 to obtain a fast estimator having low variance and high accuracy. Although an exact analysis of the bounds regarding the variances produced by Cloteaux and Valentin's algorithm still seems to be an open problem, their results show a remarkable improvement of between one and two orders of magnitude. We will conclude this section by describing their AV-K algorithm starting with GWTW. GWTW is designed to require a polynomial number of samples to determine the deepest leaf node in a tree with high accuracy and works as shown in Algorithm 4.2 below. The behaviour of GWTW is shown for three 'particles' (described in the algorithm) in Figure 4.12.


Figure 4.12: An example of the GWTW algorithm using three particles.

While GWTW is designed to determine the deepest nodes in the tree, we are specifically interested in the subtree which has been sampled to use it to get our size estimate from. Once AV-K has obtained the polynomially-sized sampled subtree $S$ from the original tree $T$ using a polynomial number of particles, it

[^12]```
Algorithm 4.2 "Go with the Winners" algorithm (adapted from [AV94)
```

Input: A tree $T$, and a number $n$ of 'particles'.
Output: The deepest leaf in the tree.

1: Place $n$ particles (a token, or abstract counting object) on the root node of $T$ depth $\leftarrow 0$
if all $n$ particles are on leaf nodes then return depth
end if

6: $I \leftarrow$ the set of non-leaf nodes containing any particles
$L \leftarrow$ the set of leaf nodes containing any particles

8: Spread all the particles in $L$ evenly among the nodes in $I$
9: Move each particle in $I$ from its node to a random child node

10: depth $\leftarrow$ depth +1
11: go to 3 // Next iteration.
proceeds by traversing $S$ to obtain the size estimate. Let $\operatorname{child}_{T}(v)$ be children of node $v$ in the original tree $T$, and $\operatorname{child}_{S}(v)$ be children of node $v$ in the GWTW sampled tree $S$. The size estimator $G(v)$ of a tree rooted at $v$ is recursively defined by the equation:

$$
G(v)= \begin{cases}1, & \text { if } v \text { is a leaf node } \\ \frac{\left|c h i l d_{T}(v)\right|}{\left|c h i l d_{S}(v)\right|} \sum_{c \in c h i l d_{S}(v)} G(c), & \text { otherwise }\end{cases}
$$

The actual subtree that has been sampled from the previous example is shown in Figure 4.13 with the subtree shaded and the node labels representing the estimate of the number of leaves in the subtree at that node. We can now finish off making the following observations:

- While Knuth proved that his method will approach the corrected expected value (size of the tree), the variance can be exponentially large. In the 'tall, skinny' tree case, an exponential number of trials are required to find the deepest nodes and avoid an underestimate Pur78.
- Knuth notes that variance may be lowered if we bias the sampling probabilities of child nodes and, while a perfect importance function provably exists, determining it is at least as hard as the tree sizing problem itself.
- In the AV-K method, using $\mathcal{O}\left(d^{c}\right)$ particles where $d$ is the depth of the tree $T$ and $c$ is some constant, the size of the sampled tree $S$ will be $\mathcal{O}\left(d^{c+1}\right)$ thus requiring polynomial time in the depth of $T$.
- We refer the reader to Cloteaux and Valentin's technical report [CV11 for more information regarding these observations and their experimental results.


Figure 4.13: The subtree sampled by GWTW and size estimates.

### 4.6 Summary

In this chapter we have covered a number of tree and graph searching algorithms which are relevant to our work. The concepts behind breadth-first search, beam search, and state space pruning are prevalent in many DFA learning algorithms. We have also briefly discussed the idea of using heuristics to greedily guide our search to find a goal as well as the pitfalls associated with the method. We will return to Knuth's ideas related to backtracking and estimating the complexity of search spaces in the next chapter, where we cover the concept of wrapping greedy heuristics with search to improve the likelihood of discovering good solutions.

## Chapter 5

## DFA Learning Algorithms

This chapter is organised in two main parts. In the first, we focus on a class of algorithms called state merging algorithms as well as their augmentation with various non-monotonic search strategies. The purpose of this chapter is to discuss how these algorithms work, identify their strengths and weaknesses, and serve as a platform on which to build our own methods. The general strategy behind state merging algorithms is to construct an initial hypothesis from a training set, and repeatedly merge pairs of states for compacting and generalisation until a final hypothesis is reached.

In the second part of this chapter, we will cover some alternatives to state merging. Our regular inference task involves searching in a space of DFAs which are consistent with a training set. Based on some definition of optimality (e.g. identifying the minimum state DFA), this task may then be construed as searching for an optimal solution in a hypothesis space. This is the inductive learning task in machine learning, where we have available to us an abundance of techniques which deal with such problems. As such, it is natural that the grammatical inference community has considered these methods when designing better algorithms and considering new approaches. Here, we survey the application of evolutionary techniques, SAT solving, recurrent neural networks, and graph colouring to our inference problem, and while we assume a general understanding of these topics, we refer readers to [Gol89], Mar09], [ $\overline{\mathrm{BBH}^{+} 09}$, HDB96], Agg18], and [CLRS09] for good introductions on them.

### 5.1 State Merging Algorithms

A state merging algorithm typically starts by constructing a PTA corresponding to the most specific hypothesis which is consistent with the training data. The algorithm then proceeds by repeatedly merging pairs of states together to 'fold' the automaton into a more compact and general hypothesis. A merge is said to be valid, compatible, or consistent if the resulting DFA is consistent with the training data. If a merge produces a DFA which is not consistent with the training data, we say that that merge is invalid, incompatible, or inconsistent and it is discarded. Whenever there are several valid merges in a DFA, a heuristic is used to choose which one out of these to proceed with. After a choice is made, this selection process is repeated until the algorithm reaches a DFA where all the merge choices available to it are inconsistent with the training data (and thus would constitute an over-generalisation). At this point, the algorithm has constructed a final hypothesis in the border set (see Section 3.4) and it is returned. The role of the negative training examples should be apparent: without negative examples, a state merging algorithm would converge to the universal automaton.

### 5.1.1 A Simple State Merging Example

Consider the regular language $L=a^{*} b^{*}$ shown in the DFA in Figure 5.1 (i) below, and the training set $S_{+}=\{\lambda, a, a a, a b, b, b b\}$ and $S_{-}=\{b a, b a a, b a b\}$. This training set results in the APTA shown in Figure 5.1 (ii). The sequence of merges $\left(q_{1}, q_{3}\right) \rightarrow$ $\left(q_{2}, q_{6}\right) \rightarrow\left(q_{2}, q_{4}\right) \rightarrow\left(q_{5}, q_{7}\right) \rightarrow\left(q_{5}, q_{8}\right)$ starting from the APTA leads to the target DFA and is illustrated in Figure 5.2.

(i)


Figure 5.1: The DFA (i) recognising the regular language $L=a^{*} b^{*}$, and the APTA (ii) for the training set $S_{+}=\{\lambda, a, a a, a b, b, b b\}$ and $S_{-}=\{b a, b a a, b a b\}$.





Figure 5.2: The path from the APTA to the DFA for $L=a^{*} b^{*}$.

### 5.1.2 Merge Paths and Graphs

Suppose that, rather than making a single merge choice, we instead expand every possible one available to us. We would obtain a merge graph where each vertex is a hypothesis and each edge is a merge. A sequence of choices a greedy algorithm makes is a merge path or merge sequence in this graph, starting from an initial hypothesis to a final hypothesis in the border (see Chapter 3). We note that this exercise of expanding each possible choice is a purely conceptual one as doing so in practice would be infeasible. Figure 5.3 shows a hypothetical merge graph from a PTA to the universal automaton showing several merge choices available. We can clearly see that a sequence of merge choices defines a path in the graph from the PTA to a DFA in the border set.


Figure 5.3: A hypothetical (and partial) merge graph showing a path from the initial hypothesis to a DFA. Blue arrows represent the merge choices made along the chosen path.

### 5.1.3 The State Merging Operation

Merging is usually described as an operation where two states $q_{i}$ and $q_{j}$ are merged together in some DFA $\mathcal{A}$. Equivalently, we may say that this involves merging two blocks $B_{i}$ and $B_{j}$ in the partition $\pi$ of some quotient automaton $\mathcal{A} / \pi$ to obtain a partition $\pi^{\prime}$ for the derived automaton $\mathcal{A} / \pi^{\prime}$. Clearly, not all pairs of states or
blocks can be merged together. A merge may be casually ${ }^{1}$ accepted or rejected depending on either of these conditions being satisfied:

1. If the starting hypothesis is an APTA: a state in the resulting DFA cannot be both accepting and rejecting. Equivalently, a block in the resulting partition cannot contain both accepting and rejecting states.
2. If the starting hypothesis is a PTA: the resulting DFA must be consistent with the training data. In other words, the resulting DFA cannot accept strings in the negative training data.

Furthermore, merging two states may yield an automaton that is not deterministic. We can see how naively merging the states $q_{1}$ and $q_{3}$ in Figure 5.4 results in non-determinism. This non-determinism is not desirable, so our operation must be designed to avoid this. We will call this procedure merging for determinism where merging a pair of states $q_{i}$ and $q_{j}$ involves:

1. Finding all the transitions incident to $q_{j}$ and redirect them to $q_{i}$.
2. Folding the state $q_{j}$ into $q_{i}$.
3. Recursively merging their successors to correct any non-determinism.


Figure 5.4: Merging the state $q_{1}$ and $q_{3}$ causes non-determinism.

[^13]Consider the example shown in Figure 5.5 where we wish to merge the states $q_{1}$ and $q_{3}$. In step 1 , we first redirect the transition $\delta\left(q_{0}, b\right) \rightarrow q_{3}$ to $q_{1}$. We next fold the state $q_{3}$ into $q_{1}$ to obtain the non-deterministic automaton shown at step 2 . This needs to be corrected, so our final step is to recursively merge the successors of $q_{1}$ and $q_{3}$ to obtain a deterministic automaton.

(step 0 - initial DFA)

(step 2 - fold states q1 and q3)

(step 1 - redirect transition)

(step 3 - recursively merge successors)

Figure 5.5: A simple example showing a merge for determinism.

We can now formalise merging for consistency and determinism as follow ${ }^{2}$

## Definition 5.1: Merging for Determinism (adapted from [CN97])

Let $\mathcal{A} / \pi$ be a quotient automaton and $B_{i}, B_{j} \in \pi$ be two blocks which we wish to merge together. The merge for determinism operation $M\left(\pi, B_{i}, B_{j}\right)=$ $\pi^{\prime}$ which merges the two blocks in $\pi$ to obtain a new partition $\pi^{\prime}$ of the deterministic quotient automaton $\mathcal{A} / \pi^{\prime}$ is defined as follows:

- Let $J$ be an operation which joins two blocks $B_{i}$ and $B_{j}$ in a partition $\pi$. This operation is defined as follows: $J\left(\pi, B_{i}, B_{j}\right)=\left(\pi \cup\left(B_{i} \cup\right.\right.$ $\left.\left.B_{j}\right)\right)-\left\{B_{i}, B_{j}\right\}$. We can see that the join operation simply replaces two blocks by their union and the resulting quotient automaton may not be deterministic.
- Deterministic merging:
- Let $\leadsto$ denote the binary relation over pairs of blocks in a partition $\pi$ where $\left(B_{i}, B_{j}\right) \leadsto\left(B_{k}, B_{l}\right)$ holds whenever there exists $a \in \Sigma$ and $q_{i} \in B_{i}$ and $q_{j} \in B_{j}$ such that $B_{k}=B\left(\delta\left(q_{i}, a\right), \pi\right)$ and $B_{l}=$ $B\left(\delta\left(q_{j}, a\right), \pi\right)$. In other words, if $\left(B_{i}, B_{j}\right) \leadsto\left(B_{k}, B_{l}\right)$, then merging $\left(B_{i}, B_{j}\right)$ implies that we also have to merge $\left(B_{k}, B_{l}\right)$.
- Let $\sim^{*}$ denote the transitive closure of $\leadsto$.
- To merge the blocks ( $B_{i}, B_{j}$ ) for determinism we then have to perform $J\left(\pi, B_{x}, B_{y}\right)$ on all the blocks $\left(B_{x}, B_{y}\right)$ in the set $\left\{\left(B_{i}, B_{j}\right)\right\} \cup$ $\left\{\left(B_{k}, B_{l}\right):\left(B_{i}, B_{j}\right) \leadsto\left(B_{k}, B_{l}\right) \in \sim^{*}\right\}$.

[^14]
## Definition 5.2: Valid Joins and Merges

A join or a merge is said to be invalid if in the resulting partition $\pi$, there exists some block $B$ such that $\exists q_{i} \in B$ and $\exists q_{j} \in B$ where $q_{i} \in F_{A}$ and $q_{j} \in F_{R}$. In other words, a join or a merge is invalid if the resulting partition contains a block having mixed accepting/rejecting states. A join or a merge is otherwise valid.

The pseudo-code for the deterministic merge operation (detmerge) is presented in Algorithm 5.1 next. Unless explicitly specified, whenever we refer to a merge in a DFA, we are always merging for determinism.

```
Algorithm 5.1 detmerge - deterministic merge (adapted from [LDD08])
Input: An APTA \(\mathcal{A}=\left\langle\Sigma, Q, q_{0}, \delta, F_{A}, F_{R}\right\rangle\), a state partition \(\pi\) of \(Q\), and the blocks
    \(B_{i}, B_{j} \in \pi\) to be merged.
Output: The updated partition with blocks \(B_{i}\) and \(B_{j}\) merged for determinism.
    \(\pi \leftarrow\left(\pi \cup\left(B_{i} \cup B_{j}\right)\right)-\left\{B_{i}, B_{j}\right\} \quad / /\) Join \(B_{i}\) and \(B_{j}\) then validate.
    if \(\left\{B_{i} \cup B_{j}\right\}\) has mixed accepting/rejecting states then return Invalid Merge
    // The new block might cause non-determinism.
    while FindNonDeterminism \(\left(\mathcal{A}, \pi,\left\{B_{i} \cup B_{j}\right\}\right) \rightarrow\left(B_{k}, B_{l}\right)\) exists do
        \(\pi \leftarrow \operatorname{detmerge}\left(\mathcal{A}, \pi, B_{k}, B_{l}\right) \quad / /\) Recursive merge.
        if \(\pi\) is an invalid merge then return Invalid Merge
    end while
    return \(\pi\)
    function FindNonDeterminism \((\mathcal{A}, \pi, B)\)
        for \(a \in \Sigma\) do
            if \(\exists B_{k}, B_{l} \in \pi \mid p \in B, q \in B, \delta(p, a) \rightarrow B_{l}, \delta(q, a) \rightarrow B_{k}\) is defined then
            return \(\left(B_{k}, B_{l}\right)\)
            end if
        end for
        return Not Found
    end function
```


### 5.1.4 Trakhtenbrot-Bardzin's and Gold's Algorithm

Trakhtenbrot and Bardzin [TB73] and Gold Gol78 independently proposed an algorithm which converges to a minimal DFA when the training data is uniformly complete (that is, the training data consists of all strings whose length is less-than-or-equal-to the depth of the target DFA). The algorithm shown in Algorithm 5.2, follows Lopéz and García's presentation and is commonly referred to as TBG [LG16]. The procedure requires determining whether two states are obviously distinguishable. Given an APTA $\mathcal{A}$ constructed from the training data, the distinctness of two states $p$ and $q$ is defined as follows:

$$
o d(p, q, \mathcal{A})=\text { True } \Leftrightarrow \exists s \in \Sigma^{*}:\left\{\begin{array}{l}
\phi(\delta(p, s)), \phi(\delta(q, s)) \in\{0,1\}, \text { and } \\
\phi(\delta(p, s)) \neq \phi(\delta(q, s))
\end{array}\right.
$$

Where:

$$
\phi(s)= \begin{cases}1, & \text { if } s \in F_{A} \\ 0, & \text { if } s \in F_{R} \\ ?, & \text { otherwise }\end{cases}
$$

It should be noted that the TBH algorithm is not deterministic. In Algorithm 5.2. this can be seen in line 4 which chooses any state in $(R \odot \Sigma-R)$, and in line 14 which chooses any indistinguishable state. These operations are typically made deterministic by selecting the first state in canonical order (length-lexicographic) among the choices available.

## Limitations of the Method

We conclude our discussion on the TBH algorithm by stating the obvious - while the algorithm is guaranteed to find the minimum canonical DFA in polynomial time, the requirement that the training data be uniformly complete makes it unsuitable for many practical applications. Furthermore, if the training data is not uniformly complete, the algorithm may fail to produce a consistent DFA in which case the APTA is returned as a fallback (i.e. it fails to generalise).

Algorithm 5.2 TBG algorithm (adapted from [LG16])
Input: Two disjoint sets $S_{+}$and $S_{-}$.
Output: An automaton consistent with $S$.
$\mathcal{A} \leftarrow \operatorname{APTA}\left\langle\Sigma, Q, q_{0}, \delta, F_{A}, F_{R}\right\rangle$ embedding $S_{+}$and $S_{-}$
$\mathrm{R} \leftarrow\{\lambda\}$
while $\exists q \in R \odot \Sigma-R: \forall p \in R, o d(p, q, \mathcal{A})=$ True do
Choose $q$ $R \leftarrow R \cup\{q\}$
end while
$\bar{Q} \leftarrow R$
$\bar{q}_{0} \leftarrow \lambda$
for $p \in R$ do
if $p \in F_{A}$ then $\bar{F}_{A} \leftarrow \bar{F}_{A} \cup\{p\}$
if $p \in F_{R}$ then $\bar{F}_{R} \leftarrow \bar{F}_{R} \cup\{p\}$
for $a \in \Sigma$ do
if $p a \in R$ then $\bar{\delta}(p, a)=p a$
else $\bar{\delta}(p, a)=$ any $q \in R$ such that $o d(p a, q, \mathcal{A})=$ False
end for
end for

7: $\mathcal{A}^{\prime} \leftarrow\left\langle\Sigma, \bar{Q}, \bar{q}_{0}, \bar{\delta}, \bar{F}_{A}, \bar{F}_{R}\right\rangle$
8: if $\mathcal{A}^{\prime}$ is consistent with $\left\langle S_{+}, S_{-}\right\rangle$then return $\mathcal{A}^{\prime}$
else return $\mathcal{A}$

### 5.1.5 Regular Positive and Negative Inference

Regular Positive and Negative Inference ( RPNI) is a state merging algorithm due to Oncina and García which returns a generalisation in all circumstances OG92. Moreover, the algorithm is guaranteed to output the target DFA when the sample $S_{+}$and $S_{-}$is characteristic (see Section 3.1.2). The procedure, described in

Algorithm 5.3, starts from an APTA constructed from the positive and negative samples $\left\langle S_{+}, S_{-}\right\rangle$and greedily merges states until no further processing is possible. During processing, RPNI distinguishes states as being either red, blue, or uncoloured (white) as shown below:


Figure 5.6: Red, blue, and white states.

During initialisation, the starting state in the APTA is red, the states immediately succeeding it are blue, and the rest are whit $3^{3}$. It should also be noted, that the states in the APTA are ordered and processed canonically (in length-lexicographic order). This is shown in the APTA constructed from $S_{+}=$ $\{a a a, a a b a, b b a, b b a b a\}$ and $S_{-}=\{a, a a b, b b\}$ in Figure 5.7.


Figure 5.7: The APTA constructed from $S_{+}=\{a a a, a a b a, b b a, b b a b a\}$ and $S_{-}=$ $\{a, a a b, b b\}$ showing red, blue, and white states.

[^15]```
Algorithm 5.3 RPNI (from [LG16])
```

Input: Positive and negative strings $S_{+}$and $S_{-}$.
Output: A DFA $\mathcal{A}$ consistent with $S_{+}$and $S_{-}$.

```
    // The states \(q_{0}, q_{1}, \ldots\) in \(\mathcal{A}\) are in canonical order where \(q_{0}=\lambda\).
    \(: \mathcal{A} \leftarrow\) Build APTA from \(S_{+}\)and \(S_{-}\)
    \(R E D \leftarrow\left\{q_{0}\right\}\)
    \(B L U E \leftarrow R E D \odot \Sigma-R E D\)
    while \(B L U E \neq \emptyset\) do
        \(q \leftarrow\) First state in \(B L U E\) in canonical order
        \(B L U E \leftarrow B L U E-\{q\}\)
        merged \(\leftarrow\) false
        for \(p \in R E D\) in canonical order do
            if \(\operatorname{detmerge}(\mathcal{A}, p, q)\) is valid then // Deterministic, Algorithm 5.1
                merged \(\leftarrow\) true
                \(\mathcal{A} \leftarrow \operatorname{detmerge}(\mathcal{A}, p, q)\)
                Break for loop
            end if
        end for
        if merged \(=\) false then
        \(R E D \leftarrow R E D \cup\{q\}\)
        end if
        \(B L U E \leftarrow R E D \odot \Sigma-R E D\)
    end while
    return \(\mathcal{A}\)
```


## A Simple Example

We conclude our discussion of RPNI with an example adapted from dlH10, where we attempt to learn the DFA from the sample $S_{+}=\{a a a, a a b a, b b a, b b a b a\}$ and $S_{-}=\{a, b b, a a b\}$. We first create the APTA, mark the initial state as red, and all of its successors blue:


Figure 5.8: The APTA constructed from $S_{+}=\{a a a, a a b a, b b a, b b a b a\}$ and $S_{-}=$ $\{a, b b, a a b$,$\} .$

For convenience, we will label all the states in canonical order (length-lexicographic) of the prefix they accept to help us visualise how states are chosen for merging:


Figure 5.9: The states in the APTA labelled in canonical order.

Now, $R E D=\left\{q_{0}\right\}$ and $B L U E=\left\{q_{1}, q_{2}\right\}$ so using our state ordering we attempt to merge $q_{0}$ with $q_{1}$. This merge is not valid, there are no further red states to attempt, so we promote $q_{1}$ to red and recompute the blue set:


Figure 5.10: Merging the states $q_{0}$ and $q_{1}$ is inconsistent with $S_{-}$.

Now $R E D=\left\{q_{0}, q_{1}\right\}$ and $B L U E=\left\{q_{2}, q_{3}\right\}$. Using our state ordering, we choose the blue state $q_{2}$ to merge into the red state $q_{0}$. This merge is invalid, so we attempt to merge the blue state $q_{2}$ with the next red state $q_{1}$. This merge is now valid, so we accept it and recompute the blue set:


Figure 5.11: Merging the states $q_{1}$ and $q_{2}$.

Now RED $=\left\{q_{0}, q_{1 / 2}\right\}$ and BLUE $=\left\{q_{3}, q_{4}\right\}$. Using our state ordering, we choose the blue state $q_{3}$ to merge into the red state $q_{0}$. This merge is invalid, so we attempt to merge the blue state $q_{3}$ with the next red state $q_{1 / 2}$. Since this merge is also invalid and there are no further red states, we promote the blue state $q_{3}$ to red and recompute the blue set:


Figure 5.12: Promoted $q_{3}$ to red and recomputed the blue set.

Now $R E D=\left\{q_{0}, q_{1 / 2}, q_{3}\right\}$ and $B L U E=\left\{q_{4}, q_{5}, q_{6}\right\}$. Using our state ordering, we choose the blue state $q_{4}$ to merge into the red state $q_{0}$. This merge is invalid, so we attempt to merge the blue state $q_{4}$ with the next red state $q_{1 / 2}$. This merge is now valid, so we accept it and recompute the blue set:


Figure 5.13: Merging the states $q_{1 / 2}$ and $q_{4}$.

Now $R E D=\left\{q_{0}, q_{1 / 2 / 4}, q_{3 / 7}\right\}$ and $B L U E=\left\{q_{5}, q_{6 / 9}\right\}$. Using our state ordering, we choose the blue state $q_{5}$ to merge into the red state $q_{0}$. This merge is valid, so we accept it and recompute the blue set:


Figure 5.14: Merging the states $q_{0}$ and $q_{5}$.

Now $R E D=\left\{q_{0 / 5}, q_{1,2,4}, q_{3 / 7}\right\}$ and $B L U E=\left\{q_{6 / 9}\right\}$. Using our state ordering, we choose the blue state $q_{6 / 9}$ to merge into the red state $q_{0 / 5}$. This merge is invalid, so we attempt to merge the blue state $q_{6 / 9}$ with the next red state $q_{1 / 2 / 4}$. This merge is now valid, so we accept it and obtain the final DFA with no more blue states:


Figure 5.15: Merging the states $q_{1 / 2 / 4}$ and $q_{6 / 9}$.

We show the final DFA omitting state numbers and colours for clarity:


Figure 5.16: The completed example omitting colours.

### 5.1.6 Price's Evidence Driven State Merging

In this section we describe Rodney Price's Abbadingo-winning Evidence Driven State Merging (EDSM) algorithm [LPP98. Several variants of EDSM have been proposed since its development and will be discussed in a separate section.

## Motivation

By examining the partial merge graph shown Figure 5.17 it is clear that:

- There may be more than one path from the initial hypothesis to the target DFA.
- Selecting a 'wrong' merge is catastrophic since, once it has been made, a path to the true target concept may not exist $\|^{4}$.

The primary motivation behind EDSM is that, since making correct decisions is so critical to our success, we should make them based on the most evidence available to us. Unlike the merge selection strategy of the algorithms introduced earlier, EDSM uses the training data to determine the merge order rather than some predefined one (such as breadth-first in RPNI). In the next sections, we define and describe the evidence used by Price in [LPP98].

[^16]

Figure 5.17: Merge paths from the APTA to hypotheses in the border set. The dark arrows represent the merge choices made to reach the target DFA.

## Transition Trees

Transition trees are structures which are useful for visualising the behaviour of EDSM's merge selection process. A transition tree can be constructed starting from any state $q$ in a DFA, and lists all the possible string suffixes following that state. A transition tree $\mathcal{T}$ built from a state $q$ in an automaton $\mathcal{A}$, is recursively constructed as follows:

- The root state of the tree is the state $q$.
- For every state $t \in \mathcal{T}$ connect the states $\left\{q_{a}=\delta(t, a): \forall a \in \Sigma\right\}$. In other words, for every state in the transition tree, connect the states in $\mathcal{A}$ that are adjacent to that state in the tree (this is the recursive step).
- If a node is added to the tree but already appears in the tree, mark it as a leaf node in the tree and return (this is the closure).

As an example, consider the true target concept $L=a b^{*} a$, and the training set $S_{+}=\{a a, a b a, a b b a, a b b b a\}$ and $S_{-}=\{\lambda, a a a, b a, a b b b\}$. This training set is symmetrically structurally complete and gives rise to the APTA shown in Figure 5.18. Typically, a transition tree is constructed and rooted at some state $q$. Each unique path from that root node $q$ to an accepting or rejecting state $q_{f}$ in the
tree represents a suffix of a string in the training set. If the ending state $q_{f}$ is an accepting state then the path $q \rightarrow \ldots \rightarrow q_{f}$ represents the suffix of a string in $S_{+}$, whereas if $q_{f}$ is a rejecting state then the path $q \rightarrow \ldots \rightarrow q_{f}$ represents the suffix of a string in $S_{-}$. The transition tree rooted at the state $q_{1}$ in the APTA shown in Figure 5.18 is shown in Figure 5.19, and has the following paths from the root to final leaf states:

1. $q_{1} \rightarrow q_{3}\left(\right.$ suffix $a$ in $\left.S_{+}\right)$,
2. $q_{1} \rightarrow q_{3} \rightarrow q_{6}$ (suffix $a a$ in $S_{-}$),
3. $q_{1} \rightarrow q_{4} \rightarrow q_{7}\left(\right.$ suffix $b a$ in $\left.S_{+}\right)$,
4. $q_{1} \rightarrow q_{4} \rightarrow q_{8} \rightarrow q_{9}$ (suffix bba in $S_{+}$),
5. $q_{1} \rightarrow q_{4} \rightarrow q_{8} \rightarrow q_{10}$ (suffix bbb in $S_{-}$), and
6. $q_{1} \rightarrow q_{4} \rightarrow q_{8} \rightarrow q_{10} \rightarrow q_{11}$ (suffix bbba in $S_{+}$).


Figure 5.18: The APTA for $S_{+}=\{a a, a b a, a b b a, a b b b a\}, S_{-}=\{\lambda, a a a, b a, a b b b\}$.


Figure 5.19: The transition tree from state $q_{1}$.

## The Evidence Metric

Price defined his evidence metric as follows: A merge's score is the sum over equivalence classes of the following quantity: if there are conflicting labels in the class, minus infinity; if there are no labels in the class, zero; otherwise, the number of labels minus one. We subtract one because the first label in the class establishes the correct label for the class, but is not checked [LPP98].

We can visualise how the evidence score of the merge $q_{1}$ with $q_{4}$ in the APTA shown in Figure 5.18 is computed by superposing the two transition trees and count the number of state labels that match. These matches and the resulting score is illustrated in Figure 5.20 below.

The intuition behind EDSM's heuristic can now be summarised as follows: The more labels a merge groups together, the more likely it will be that that merge is correct. In practice, an implementation to compute the EDSM score of a merge without needing to build and examine transition trees can be derived from Price's definition itself:

1. Merge the states in the DFA $\mathcal{A}$.
2. If the merge is not compatible, return $-\infty$.
3. This merge will give rise to a quotient DFA $\mathcal{A}^{\prime}$ where each state is an equivalence class of the states in the original DFA $\mathcal{A}$ (the states are in the same
$q_{4}$

Transition tree from
$q_{1}$


Figure 5.20: Computing the EDSM score by superposing transition trees. block of the partition).
4. Step 2 guarantees that there will never be any mixed accepting/rejecting states in a single equivalence class.
5. Inspect the equivalence classes (blocks of states) in $\mathcal{A}^{\prime}$, and sum the following value for each one:
(a) If the equivalence class contains only unlabelled states, contribute 0 to the score, else
(b) Contribute number of labelled states minus one to the score.

To score the merge between the states $q_{1}$ and $q_{4}$ in the APTA shown in Figure 5.18, we start by actually performing the merge to get the DFA in Figure 5.21.


Figure 5.21: The DFA obtained after merging the states $q_{1}$ and $q_{4}$ in the APTA shown in Figure 5.18.

The merge is consistent with the training data, and the resulting DFA has six states which correspond to the following equivalence classes: $\left\{q_{0}\right\},\left\{q_{1}, q_{4}, q_{8}, q_{10}\right\}$, $\left\{q_{2}\right\},\left\{q_{3}, q_{7}, q_{9}, q_{11}\right\},\left\{q_{5}\right\}$, and $\left\{q_{6}\right\}$. The computation of the sub-scores for each equivalence class as well as the total EDSM score is shown in Table 5.1 below.

| Equiv. class | Label counting | Score contribution |
| :---: | :---: | :---: |
| $\left\{q_{0}\right\}$ | Only contains unlabelled state $q_{0}$ | 0 |
| $\left\{q_{1}, q_{4}, q_{8}, q_{10}\right\}$ | $q_{1}, q_{4}, q_{8}$ are unlabelled, and $q_{10}$ is rejecting (i.e. 1 labelled state) | 1 labelled state $-1=\mathbf{0}$ |
| $\left\{q_{2}\right\}$ | Only contains unlabelled state $q_{2}$ | 0 |
| $\left\{q_{3}, q_{7}, q_{9}, q_{11}\right\}$ | $q_{3}, q_{7}, q_{9}, q_{11}$ are all accepting (i.e. 4 labelled states) | 4 labelled states -1 $=\mathbf{3}$ |
| $\left\{q_{5}\right\}$ | Only contains one rejecting state $q_{5}$ (i.e. 1 labelled state) | 1 labelled state $-1=\mathbf{0}$ |
| $\left\{q_{6}\right\}$ | Only contains one rejecting state $q_{6}$ (i.e. 1 labelled state) | 1 labelled state $-1=\mathbf{0}$ |
|  | Score $=\operatorname{Sum}(0+0+3+0+0)=$ | 3 |

Table 5.1: The computation of the EDSM score for the merge $\left(q_{1}, q_{4}\right)$ in the APTA shown in Figure 5.18.

## The Algorithm

Algorithm 5.4 shows how EDSM is used to greedily construct a merge path starting from an APTA to a final hypothesis.

Algorithm 5.4 EDSM (adapted from [LPP98])
Input: A positive sample $S_{+}$and a negative sample $S_{-}$.
Output: A hypothesis DFA.

```
\(\mathcal{A} \leftarrow\) Construct an APTA from \(S_{+}\)and \(S_{-}\)
merges \(\leftarrow\) Compute all valid detmerges between all pairs of states in \(\mathcal{A}\)
while merges \(\neq \emptyset\) do
    \(\mathcal{A} \leftarrow\) Perform detmerge having the highest EDSM score
    merges \(\leftarrow\) Compute all valid detmerges between all pairs of states in \(\mathcal{A}\)
end while
return \(\mathcal{A}\)
```


## Tie-Breaking

When EDSM is selecting merges, it is quite possible that two or more merges are tied with same score. Several tie-breaking strategies may be found in the literature including breaking ties randomly, using a relative depth-based heuristic, or a combination of both. In Spi04, Spina describes the following relative depthbased heuristic to resolve ties:

$$
\operatorname{Score}=\operatorname{Score}+\left(0.9+\left(-0.01 \times \operatorname{MAx}\left(\operatorname{DEPth}\left(q_{i}\right), \operatorname{DEPTH}\left(q_{j}\right)\right)\right)\right)
$$

## A Simple Example

We illustrate a complete run of EDSM using training set $S_{+}=\{a a a, a a b a, b b a$, $b b a b a\}$ and $S_{-}=\{a, b b, a a b, a b a\}$. This training set results in the starting APTA shown in Figure 5.22 below.


Figure 5.22: The APTA constructed for $S_{+}=\{a a a, a a b a, b b a, b b a b a\}$ and $S_{-}=$ $\{a, b b, a a b, a b a\}$.

Out of all the possible valid merges, we select the one between the states $q_{2}$ and $q_{3}$ which has the highest randomly tie-broken EDSM score of 2 . The new DFA would correspond to the state partition $\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}, q_{3}\right\},\left\{q_{4}\right\},\left\{q_{5}, q_{7}\right\}\right.$, $\left.\left\{q_{6}\right\},\left\{q_{8}\right\},\left\{q_{9}, q_{10}\right\},\left\{q_{11}\right\},\left\{q_{12}\right\}\right\}$. This DFA is shown in Figure 5.23 where each state is annotated with its EDSM score contribution, and are then renumbered in breadth-first order for clarity.


Figure 5.23: The current hypothesis after merging the states $q_{2}$ and $q_{3}$.

We proceed by examining all the possible valid merges available, and select the one between the states $q_{1}$ and $q_{5}$ (in the renumbered DFA in Figure 5.23) which has the highest randomly tie-broken EDSM score of 2 . The new DFA would correspond to the state partition $\left\{\left\{q_{0}\right\},\left\{q_{1}, q_{5}, q_{8}\right\},\left\{q_{2}, q_{7}, q_{9}\right\},\left\{q_{3}\right\},\left\{q_{4}\right\},\left\{q_{6}\right\}\right\}$ and is shown in Figure 5.24 annotated with the EDSM score contribution, and the states, again, renumbered for clarity.


Figure 5.24: The current hypothesis after merging the states $q_{1}$ and $q_{5}$.

We next select the merge between the states $q_{2}$ and $q_{4}$ which has the highest randomly tie-broken EDSM score of 1 . The new DFA would correspond to the state partition $\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}, q_{4}\right\},\left\{q_{3}\right\},\left\{q_{5}\right\}\right\}$ and is shown in Figure 5.25 below annotated with the EDSM score contribution.


Figure 5.25: The current hypothesis after merging the states $q_{2}$ and $q_{4}$.

Our final step is to select the merge between the states $q_{0}$ and $q_{3}$ which has the highest randomly tie-broken EDSM score of 1 . The final hypothesis is a three-state DFA corresponding to the state partition $\left\{\left\{q_{0}, q_{3}\right\},\left\{q_{1}, q_{4}\right\},\left\{q_{2}\right\}\right\}$ and is shown in Figure 5.26 .


Figure 5.26: The final hypothesis obtained after merging the states $q_{0}$ and $q_{3}$.

We conclude by observing that, in this example, we used the same training sample $S_{+}$and $S_{-}$that we started with in the example for RPNI (Section 5.1.5). Both algorithms inferred a DFA having three states albeit for different languages.

### 5.1.7 Variants of EDSM

## Blue-Fringe

The reference implementation of EDSM considers all the pairs of states in the current hypothesis as candidates for merging and picks the one having the highest score. While this has the advantage that all possible pairings of states are taken into account (we are not restricting the search), this procedure is computationally expensive due to the large number of merge pairs to consider ${ }^{5}$. In LPP98, Lang et al. propose a variant of EDSM that places a restriction on the merge order identical to blue-fringe state merging algorithms - that is, only pairs of red/blue states are considered, scored and selected. The idea is twofold: (a) the set of pairings is smaller, and (b) since blue states are always the root of a tree in the DFA, the merge procedure can be implemented more efficiently. Lang et al. make the following observations:

- The unrestricted, reference implementation is slightly more effective than the blue-fringe variant. This is attributed to the fact that the reference implementation works with a larger pool of candidates.
- There are still many instances where the blue-fringe variant succeeds and the reference implementation fails. The failures of the two implementations are "seemingly uncorrelated".
- Because of this, both implementations may be made to work on the same problem, and the smaller of the two resulting DFAs is returned.


## Windowed EDSM (W-EDSM)

Another alternative suggested by Lang et al. in [LPP98 to minimise the computational expense incurred by evaluating each possible merge step, is a windowing approach referred to as Windowed-EDSM (W-EDSM). In W-EDSM, only the states that lie within a distance $w$ of the root in a breadth-first ordering of the states in the hypothesis are considered. This distance $w$ is referred to as the window size. This change has the following effects:

[^17]- Performance may be hurt by overlooking high-scoring merges which involve states that are deeper than the window allows the algorithm to 'see'. The authors, however, argue that this event is "relatively rare".
- The running time is much better than the reference implementation of EDSM.
- The recommended window size $w$ is twice the size of the target DFA. This means that we would need to know the size of the target DFA a priori.

We adapt the W-EDSM procedure from CK02] next and refer readers to CK03, and [Spi04 for further discussion on W-EDSM:

- In breadth-first order, create a window of nodes in the current DFA. The recommended size of the window is twice the size of the target DFA.
- Evaluate all possible merge pairs within the window. Ties are broken randomly.
- Merge the pair of nodes that has the highest calculated score.
- If the merge reduces the size of the window, in breadth-first order, include the number of nodes needed to regain a window of size twice the target DFA.
- If a merge is not possible within the given window, increase the size of the window by a factor of two.
- Terminate when no more merges are possible.


## Shared Evidence Driven State Merging (S-EDSM)

The primary weakness of EDSM is that its merge ordering heuristic relies on a score whose significance may be weak early in the run of the algorithm (support for this was studied by Ciccello and Kremer when working on MW-EDSM in CK02 and confirmed by our experimental results presented later). S-EDSM, due to Abela et al., attempts to address this by defining an alternative state ordering based on the concept of shared evidence between different valid merges [ACS04]. The idea is to increase evidence in the situations where EDSM has little or none.

A central question in S-EDSM is concerned with how individual valid merges affect other valid merges in a hypothesis. Specifically, if $m_{1}$ and $m_{2}$ are both valid merges in a DFA $\mathcal{A}$, and we first perform the merge $m_{1}$ to obtain $\mathcal{A}^{\prime}$, does $m_{2}$ remain a valid merge in $\mathcal{A}^{\prime}$ ? This question gives rise to the notion of pairwise compatibility for merges which we proceed by defining next.

## Definition 5.3: State Compatibility [ACS04, Spi04]

Two states $q$ and $q^{\prime}$ are said to be state incompatible if both the states are labelled and those labels are different. Specifically, a state $q$ is incompatible with $q^{\prime}$ if $q \in F_{A}$ and $q^{\prime} \in F_{R}$, or if $q \in F_{R}$ and $q^{\prime} \in F_{A}$. The states $q$ and $q^{\prime}$ are otherwise state compatible.

## Definition 5.4: Pairwise Merge Compatibility ACS04, Spi04

Consider some DFA $\mathcal{A}$, as well as two valid merges $m_{1}$ and $m_{2}$. Let $\mathcal{A}_{1}$ be the hypothesis obtained after performing $m_{1}$ in $\mathcal{A}$, and $\mathcal{A}_{2}$ be the hypothesis obtained after performing $m_{2}$ in $\mathcal{A}$. The merges $m_{1}$ and $m_{2}$ are said to be pairwise compatible if for each $q \in \mathcal{A}$, then $q \in \mathcal{A}_{1}$ is state compatible with $q \in \mathcal{A}_{2}$. We denote this relation by $m_{1} \uparrow m_{2}$.

If two merges $m_{1}$ and $m_{2}$ are not pairwise compatible, we say that they are pairwise incompatible and write $m_{1} \downarrow m_{2}$.

We will use the notation $m_{1} \uparrow\left\{m_{2}, m_{3}, \ldots\right\}$ to express that $m_{1}$ is pairwise compatible with $m_{2}, m_{3}, \ldots$.

An example of pairwise compatible merges is shown in Figure 5.27. Both the merges $\left(q_{2}, q_{3}\right)$ and $\left(q_{4}, q_{5}\right)$ are valid in the DFA shown in (i), and if we perform $\left(q_{2}, q_{3}\right)$ first, the merge $\left(q_{4}, q_{5}\right)$ is still possible and valid. The same is true if we had started with $\left(q_{4}, q_{5}\right)$. As a counterexample, consider Figure 5.28, Both merges $\left(q_{0}, q_{1}\right)$ and $\left(q_{4}, q_{6}\right)$ are valid, but after performing $\left(q_{0}, q_{1}\right)$, the merge $\left(q_{4}, q_{6}\right)$ is no longer possible. At this point, we may observe that the pairwise compatibility induces a binary relation between merges and that the relation is symmetric ( $m_{1} \uparrow$
$m_{2}$ implies $m_{2} \uparrow m_{1}$ ) ACS04, Spi04.


Figure 5.27: Pairwise compatible merges in S-EDSM.


Figure 5.28: Pairwise incompatible merges in S-EDSM.

Consider a scenario where there are six possible merges in some DFA. These merges together with the other merges they are pairwise compatible with, as well
as their hypothetical EDSM scores are shown Table 5.2. If we had to use EDSM as our heuristic, we would select $m_{1}$ which has the highest score of 8 and proceed along that merge path. By making this choice, the merges $m_{3}, m_{4}, m_{5}$, and $m_{6}$ become invalid as only $m_{2}$ is pairwise compatible with the merge $m_{1}$ we just made.

| Merge | $\uparrow$ | EDSM score |
| :---: | :---: | ---: |
| $m_{1}$ | $\left\{m_{2}\right\}$ | 8 |
| $m_{2}$ | $\left\{m_{1}, m_{4}\right\}$ | 6 |
| $m_{3}$ | $\left\{m_{4}, m_{5}\right\}$ | 5 |
| $m_{4}$ | $\left\{m_{2}, m_{3}, m_{5}\right\}$ | 4 |
| $m_{5}$ | $\left\{m_{3}, m_{4}\right\}$ | 4 |
| $m_{6}$ | $\}$ | 2 |

Table 5.2: Pairwise compatible merges.

S-EDSM makes its merge choice by taking into account the pairwise compatibility of the available merges as follows: the S-EDSM score for a merge is equal to the EDSM score of the merge plus the EDSM score of each other merge it is pairwise compatible with. This is shown in Table 5.3 where the EDSM scores are shown in superscript.

| Merge | $\uparrow$ | S-EDSM score |
| :---: | :---: | ---: |
| $m_{1}^{(8)}$ | $\left\{m_{2}^{(6)}\right\}$ | $8+6=\mathbf{1 4}$ |
| $m_{2}^{(6)}$ | $\left\{m_{1}^{(8)}, m_{4}^{(4)}\right\}$ | $6+8+4=\mathbf{1 8}$ |
| $m_{3}^{(5)}$ | $\left\{m_{4}^{(4)}, m_{5}^{(4)}\right\}$ | $5+4+4=\mathbf{1 3}$ |
| $m_{4}^{(4)}$ | $\left\{m_{2}^{(6)}, m_{3}^{(5)}, m_{5}^{(4)}\right\}$ | $4+6+5+4=\mathbf{1 9}$ |
| $m_{5}^{(4)}$ | $\left\{m_{3}^{(5)}, m_{4}^{(4)}\right\}$ | $4+54+=\mathbf{1 3}$ |
| $m_{6}^{(2)}$ | $\}$ | $2=\mathbf{2}$ |

Table 5.3: How merges are scored using S-EDSM.

Using this strategy, we see that $m_{4}$ has the highest S-EDSM score of 19 and would be selected as the next merge instead of $m_{1}$ which would have been chosen by EDSM. We can now proceed by making a few observations:

- S-EDSM tries to maximise the number of merges which remain valid after a particular merge has been made. Recall that, earlier on we mentioned that performing a merge implies that certain merges and paths no longer remain possible. By maximising the options that remain open, we may say that S-EDSM is a risk-averse search strategy.
- In practice, S-EDSM is actually performing a lookahead. To score a merge, EDSM just needs to 'look into' its partition. In S-EDSM, in order to determine which merges are pairwise compatible with the one being considered, we have to first execute the merge and then inspect its results in the next level.
- Spina reports that, because of this lookahead, S-EDSM is much more computationally expensive than EDSM Spi04. To alleviate this, the author suggests considering a subset of the valid merges to work with. This can be determined by:
- Either only considering valid merges that are within a percentage bracket of the highest EDSM score, or
- Only considering a top percentage of merges from the valid ones ordered in descending EDSM score.
- For sparse training sets, EDSM has to resolve many ties, whereas in S-EDSM, Spina suggests that there are fewer. Spina also suggests that S-EDSM might actually be useful as an EDSM tie-breaker Spi04.

We conclude our discussion on S-EDSM by summarising the principal results in ACS04 and Spi04:

- It has been observed that, in several instances, the merge paths selected by EDSM and S-EDSM are identical. Even when this is not the case, the paths may still lead to the same final hypothesis.
- Abela et al. measured the performance of S-EDSM in terms of both the classification rate of the hypotheses identified, as well as their sizes. On average, the classification rate of S-EDSM is consistently better than EDSM.

Specifically, S-EDSM did not perform worse than EDSM, and in a number of instances outperformed it.

- As training sets become sparser, S-EDSM performs better than EDSM.
- Spina also described and justified other merge interactions, including mutually compatible merges, merge coverage, and merge dominance. As published in ACS04] and Spi04, S-EDSM only uses pairwise compatibility to support merge selection. The other interactions suggested remain a matter for further investigation.


### 5.1.8 DFA Learning using Minimum Description Length

In AJ06, Adriaans and Jacobs describe an interesting study regarding the use of the Minimum Description Length (MDL) principle for DFA induction ${ }^{6}$. The method described by the authors is outlined as follows:

- An APTA is constructed from the positive and negative strings in the training set.
- Merge pairs are selected according to the blue-fringe method described in Section 5.1.7 above.
- The algorithm uses MDL to determine which red/blue states will be merged together - all the red/blue state combinations are scored using the MDL heuristic, and the best one is chosen.
- This process is repeated until all the states in the hypothesis are red where we would have converged to a final hypothesis.

So far, the method is identical to running the EDSM algorithm in the bluefringe framework with the exception that MDL is used to score merge pairs rather than EDSM. While referring readers to [AJ06] for a complete derivation, the MDL score computed as the sum of (i) the number of bits required to encode the model (the model code), and (ii) the number of bits required to encode the sample given that model (the data-to-model code). To approximate the model code, the

[^18]authors use the following argument: Suppose an $n$-state DFA is defined over an alphabet $\Sigma$, and is functionally complete (i.e. there is an outgoing transition for each symbol for each state). There are then $n \times|\Sigma|$ transitions, each having $n$ possible destinations, giving $n^{n \times|\Sigma|}$ possibilities. Each state can either be accepting or rejecting, giving $2^{n}$ possibilities. For each permutation of states $2 \ldots n$ there is an equivalent DFA. The length of an index identifying a specific DFA (model) is then $\log _{2}\left(\frac{2^{n} \times n^{n \times|\Sigma|}}{(n-1)!}\right)$. Adriaans and Jacobs propose five variations to determine the length of the data-to-model code. An example considering only the positive strings in the training set is given by $\log _{2}\binom{m+}{d+}$, where $m+$ is the size of the model (computed in the previous step), and $d+$ is the number of positive strings in the training set. The heuristic can now be described as: the lower the sum of the model code and the data-to-model code, the better the merge is assumed to be.

Using one or more variants of the data-to-model code computation proposed, the MDL method was able to solve a subset of the Abbadingo One competition problems that EDSM was also able to solve. Specifically, EDSM was able to solve four more classes of problems which MDL was not able to.

### 5.1.9 Parallel Beam Search (PBS, SAGE)

In [JP98a, JP98b], Juillé and Pollack describe a stochastic search algorithm which uses a beam search (see Section 4.3.5) to explore the most promising parts of the search space and is inspired by Knuth's ideas on estimating backtracking costs (see Section 4.5). The algorithm considers several merge paths each starting from an initial state, and follows a sequence of partial solutions (the DFAs obtained by merging) that eventually lead to a final solution (the hypothesis DFA in the border set). The method, called Self-Adaptive Greedy Estimate (SAGE), is described in Algorithm 5.5, and is broken down into two components: the construction phase, and the competition phase.

## The Construction Phase

Let PE denote a processing element which performs the following task:

- Start from the PTA $\mathcal{A}_{0}$.
- Select a random merge to obtain some $\mathcal{A}_{1}$ from the set of all possible valid merges (children) of $\mathcal{A}_{0}$.
- Select a random merge to obtain some $\mathcal{A}_{2}$ from the set of all possible valid merges of $\mathcal{A}_{1}$.
- Repeat this process until we select some merge to get $\mathcal{A}_{n}$ which cannot be expanded any further (no further merges are possible because $\mathcal{A}_{n}$ will be a DFA in the border set). This sequence $\mathcal{A}_{0} \rightarrow \ldots \rightarrow \mathcal{A}_{n}$ represents a random merge path that identifies the hypothesis $\mathcal{A}_{n}$.
- Assign a score to the hypothesis. This score is computed with respect to the "problem objective function". In our case, this is the size of the hypothesis DFA $\mathcal{A}_{n}$ (this follows from the definition of the Abbadingo success criterion, Section (1.2).

Given the starting PTA, a PE is created for each merge possible from it (i.e. for each 'child node' of the PTA in the merge graph) to score that merg ${ }^{7}$ ? This procedure is illustrated in Figure 5.29. Once all the nodes are scored by their respective PEs, the construction phase for the current level is complete and we proceed to the competition phase.

## The Competition Phase

We now have scores for each merge available at a given level, and proceed by selecting the best $w$ alternatives (the beam width) from them. In Figure 5.29 we would select the highest scoring $w$ DFAs from those labelled $1,2,3, \ldots n$. We now repeat the construction phase for each of the alternatives to process the next level. This (construction $\rightarrow$ competition) process is repeated until no further merges are possible.

[^19]

Figure 5.29: Constructing and scoring random merge paths in SAGE.

```
Algorithm 5.5 Self-Adaptive Greedy Estimate (adapted from [JP98a, [JP98b])
Input: A prefix tree acceptor \(\mathcal{A}\), and a beam width \(w\).
Output: A hypothesis DFA \(\mathcal{H}\).
    candidates \(\leftarrow\) An empty max-queue of DFAs ordered by \(\mathcal{A}\).score
    beam \(\leftarrow \mathrm{A}\) list containing \(\mathcal{A}\)
    while beam \(\neq \emptyset\) do
        \(q \leftarrow\) A new max-queue \(\quad / /\) Queue is ordered by \(\mathcal{A}\).score .
        for each \(\mathcal{A}^{\prime}\) in beam do
        merges \(\leftarrow\) validDetMerges \(O f\left(\mathcal{A}^{\prime}\right)\)
        if \(\mid\) merges \(\mid=0\) then
            candidates \(\leftarrow\) candidates \(\cup\left\{\mathcal{A}^{\prime}\right\} \quad / /\) No more merges possible.
            else
                for \(\mathcal{A}_{\text {next }} \in\) merges do // Score each merge.
                    \(\mathcal{A}_{\text {next }}\).score \(\leftarrow \operatorname{score}\left(\mathcal{A}_{\text {next }}\right) ;\) q.enqueue \(\left(\mathcal{A}_{\text {next }}\right) \quad / /\) See Section 5.1.9
                end for
            end if
        end for
        beam \(\leftarrow \emptyset \quad / /\) Populate the beam.
        while \(q \neq \emptyset\) and \(w>\mid\) beam \(\mid\) do
            beam \(\leftarrow\) beam \(\cup\{q\).dequeue ()\(\} \quad / /\) Dequeue max of \(\mathcal{A}\).score.
        end while
    end while
    return best \(O f\) (candidates)

\subsection*{5.1.10 Ed-Beam}

In Lan98 and Lan99, Lang describes a method and implementation which combines elements of Juillé and Pollack's search strategy using Price's EDSM heuristic to obtain better generalisation rates using less computational resources. Straight from Lang, the observation is that "Not only are Price's improved merge ordering and Juille's search over initial merges good ideas in isolation, but each idea benefits from the other one when they are combined". This led to the development of a method which combines beam search and EDSM called Ed-Beam.

The first component in Ed-Beam is the function ed-fold which is a blue-fringe implementation of EDSM that is guided by a binary control string which is infinitely right-padded with zeroes. At every merge step, the function consults the next bit in the control string and if a 0-bit is encountered, the highest scoring merge according to the heuristic will be chosen. If, on the other hand, a 1-bit is encountered, that highest scoring merge will be marked as unusable for the remainder of the path construction. When ed-fold completes a path, it returns the size of the hypothesis it discovered as a score. Ed-Beam wraps this function in a beam search where at each step, the algorithm has a population of \(p\) of control strings of length \(l\). Each of these \(p\) control strings is extended with a 0 or a 1 to give us \(2 p\) control strings of length \(l+1\). These control strings are evaluated using ed-fold and trimmed to the best \(p\) for the next iteration. A Standard ML implementation of this method may be found in Lan99.

Evaluation was made on 810 benchmark problems having between 4 and 21 states used to label 20 binary strings of length 30 . Rather than providing the labels of the 20 strings (which would correspond to the labels of the states reached at the end of the string), a label is provided for each of the states visited by any given string. Due to the common prefixes between the strings, there would be roughly 550 state labels in a given training set. During evaluation, Lang allotted a maximum of 900 seconds \(]^{8}\) to the algorithm which found provably optimal solutions for each of the 810 benchmark problems. Ed-Beam is considered to be a state-of-the-art technique for learning DFAs on Abbadingo-style problem instances HV13].

\footnotetext{
\({ }^{8} \mathrm{On}\) a 450 MHz Pentium II machine.
}

\subsection*{5.1.11 TBW-EDSM}

In Cic02, Cicchello describes a limited-search approach to learning DFAs and studies its behaviour when training sets are sparse. The method is based on the observation that, should the initial merge choices made by a state merging algorithm be correct, they would be sufficient to constrain the hypothesis well enough such that a heuristic such as EDSM can then proceed with a greater likelihood of identifying the target DFA or a close approximation of it. The author verifies this by using a modified version of W-EDSM (referred to as MW-EDSM) to determine which levels in the merge path construction are the most crucial.

Supported by these findings, Cicchello presents the TBW-EDSM algorithm (Trakhtenbrot-Barzdin + windowed EDSM) which attempts to correctly identify the first merge to perform at the APTA level. After this first merge is selected, the algorithm proceeds using standard W-EDSM. Algorithm 5.6 illustrates this procedure.
```

Algorithm 5.6 TBW-EDSM ([Cic02])
Input: The training data $S_{+}$and $S_{-}$, a window size $t$.

```

Output: A hypothesis DFA \(\mathcal{A}\).
for \(i \in\{1, \ldots, t\}\) do \(\mathcal{A} \leftarrow\) The APTA from \(S_{+}\)and \(S_{-}\) \(m \leftarrow\) The \(i^{\text {th }}\) valid detmerge in the breadth-first traversal of \(\mathcal{A}\) \(\mathcal{A} \leftarrow\) Execute the detmerge \(m\) on the APTA \(\mathcal{A}\) \(\mathcal{A} \leftarrow\) Complete \(\mathcal{A}\) using W-EDSM
end for
return The smallest completed hypothesis \(\mathcal{A}\) encountered

To evaluate the method, 300 Abbadingo-style problems were generated having an average target DFA size of 63.75 . In each case, a training set consisting of 1422 strings was used. This corresponds to a density which is slightly lower than Abbadingo density 1 problems for a nominal size of 64 states. For comparison, W-EDSM was given 30 attempts at each problem, and TBW-EDSM was set to search in the first 30 breadth-first merges ( \(t=30\) in Algorithm 5.6). TBW-EDSM
succeeded in finding low-error hypotheses in 24 out of the 300 problems, while W-EDSM only managed to do so in 17 cases. The TBW-EDSM search strategy also managed to outperform W-EDSM for identifying hypotheses having a more lenient error rate of between \(95 \%\) and \(99 \%\) on a test set.

\subsection*{5.1.12 Some Remarks Regarding 'Search Wrappers’}

State merging algorithms driven by a greedy heuristic such as EDSM, are very computationally efficient since they do not perform any search beyond scoring the merges available at any current point along a merge sequence. As such, their performance limited by the quality of heuristics used BO05. Juillé and Pollack's SAGE, Lang's Ed-Beam, and Cicchello's TBW-EDSM are usually described as specialised procedures that 'wrap' a greedy heuristic (typically, EDSM) in a search [Lan98, HV13]. For instance, SAGE wraps a stochastic search around a state merging algorithm, and Ed-Beam wraps a deterministic search around EDSM. Wrapping heuristics such as EDSM in a search has also been applied to the task of identifying deterministic real-time automata \({ }^{9}\) VdWW12. We refer readers to BO05 for a survey of several other approaches to using search techniques (guided by a heuristic) to explore the space of possible sequences of merges.

\subsection*{5.2 Other DFA Learning Methods}

\subsection*{5.2.1 Genetic Search and Swarm Intelligence}

Genetic search is widely understood to be a powerful metaheuristic, and has been frequently deployed to tackle grammatical inference problems. We see several attempts in Dup94, LR05], BL05], [SB07], and [TE11]. Additionally, and of special importance to us, there have been efforts to directly evaluate the performance of evolutionary methods with respect to EDSM on Abbadingo-style problem instances [LR03]. Each of these approaches and corresponding conclusions have their own nuances, and while we refer readers to the authors' work for the complete methods, we outline the implementation of a simple genetic algorithm

\footnotetext{
\({ }^{9}\) Where the training data consists of sequences of events generated by a real-time system.
}

\section*{Chromosome representation:}
- Let \(\{0,1, \ldots, n-1\}\) be the indices of the \(n\) states in our starting prefix tree acceptor.
- A partition \(\pi\) is encoded as a sequence of \(\left(i_{0}, i_{1}, \ldots, i_{n-1}\right)\) integers where each integer \(i_{j} \in\{0,1, \ldots, m-1\} \mid 0 \leq j<n\) denotes the block assigned to the state \(j\), where \(m\) is the size of the target DFA. In other words, \(i_{p}=i_{q} \Longleftrightarrow\) \(B(p, \pi)=B(q, \pi)\). As an example, consider the chromosome \((2,2,0,1,2,1)\) which represents the partition \(\{\{0,1,4\}\{2\}\{3,5\}\}\).
- We can see that this representation is closed for the mutation and crossover operators defined next - i.e. the resulting chromosomes are representations of valid partitions.

\section*{Mutation:}
- Mutation simply involves selecting two random indices in a chromosome and swapping their values. An example is shown next:


\section*{Crossover:}
- Crossover involves selecting a random index \(r\) common to both parents, and the values at the indices \(\leq r\) in both parents is swapped to obtain two offspring. We, again, illustrate with an example:


The fitness function is defined as follows:
- Assign a fitness of \(\infty\) if the quotient automaton induced by the partition is inconsistent with the negative samples.
- Otherwise, since we want to minimise the size of the DFA induced by the partition, the fitness is the number of distinct blocks in the partition (as each block corresponds to a state in the hypothesis).

This approach essentially involves 'packing' states into a partition of a quotient DFA, and forms the basis of many evolutionary methods for inferring regular languages. An alternative would be to evolve the target DFA directly as can be seen in LR05]. Here, Lucas and Reynolds use the transition matrix of the target DFA as the chromosome representation (they evolve an \(n\)-state DFA rather than a partition) and apply several operators on it to explore the search space. We note that, the transition matrix by itself is not sufficient to represent a DFA since the final/accepting state labels would be missing. Rather than encoding the state labels in the chromosome itself, which would increase the size of the search space by a factor of \(2^{n}\), they infer the types from the training set. Each state in the candidate hypothesis encoded by the transition matrix is labelled according to how many strings in the training set reach it. If more positive strings reach a state in the candidate hypothesis than negative ones, the state is labelled as accepting. Otherwise, the state is labelled as rejecting. The authors call this scheme smart state labelling. The framework used by Lucas and Reynolds is that of a multistart, random hill-climber which does not require a crossover operator. Several kinds of mutation operators are described where the most basic one involves choosing a random location in the transition table (the destination of a state) and replacing it with another state selected from a uniform distribution of all states. The fitness function is simply a measure of how many strings in the training set are classified correctly by the candidate hypothesis.

We summarise this part by discussing some of the results obtained using variants of the methods we described earlier:
- For small Abbadingo-style DFAs, [LR03 report that their evolutionary approach outperforms EDSM for sparse data but is 'clearly' outperformed by

EDSM for automata having 32 states or more.
- Similar results may be found in [BL05] with EDSM performing better than the GA on larger target DFA problems. Interestingly, their method allows for hypotheses to expand (specify) and compress (generalise) throughout the run of the algorithm, whereas EDSM only compresses. We concur with the authors that the method warrants more study.
- The method presented in [R05] allows for noise in the data set, but is again outperformed by EDSM (and its variants) for target DFAs at or larger than 32 states.

In CU12, Chivilikhin and Ulyantsev describe an Ant Colony Optimisation, or ACO CDM91, approach to learning finite state machines (FSMs). The proposed method works as follows:
- Initially, the graph that the artificial ants search in is empty. A node corresponding to a randomly constructed FSM is created and added to the graph, and all the ants are placed on this node.
- Node expansion and selection:
- With some probability \(p\), an ant constructs new edges in the graph by performing a number of mutations (a change in transition end state and/or change the transition action) of the current node/FSM it is on (without replacement). The ant moves to the best new node it has created.
- With a probability \(1-p\), the ant will select a next node from the existing successors. Selection is stochastic, based on the classical ACO pheromone formula [CDM91].
- The size of the graph is controlled as follows:
- Each ant is given a maximum number of steps to perform without an increase in fitness value.
- The whole colony is given a maximum number of steps to perform without an increase in overall fitness value.
- As ants traverse the graph, the pheromone levels are being updated and evaporated as in classical ACO.
- The fitness of a node in the graph is computed as the error over a test set.

The authors experimented with the "Alarm Clock Problem" on FSMs having four states. The average number of fitness evaluations used to identify the target FSM was measured over 1000 runs of their ACO algorithm, as well over 1000 runs of a GA for the same problem Tsa10. Their results show that the GA they implemented required more than twice the number of fitness evaluations than the ACO needed (117,977 vs. 53,944 fitness evaluations). A second "John Muir food trail" scenario was evaluated on FSMs having seven states, where the ACO identified two valid solutions after 143 and 221 million fitness evaluations. The authors report that, on both the problems, their ACO either outperformed, or worked as well as the GA they compared their method with.

\subsection*{5.2.2 DFA Learning as Constraint Satisfaction}
\(B I C\) is an algorithm proposed by Oliveira and Silva which uses the principle constraint satisfaction to identify the smallest DFA consistent with a training set OMS98. The method is outlined as follows:
- Consider a target DFA having the states \(Q\), and an APTA \({ }^{10}\), constructed from some training set, having the states \(Q^{\prime}\).
- It follows that there exists a mapping \(f: Q^{\prime} \rightarrow Q\) which determines which state in the APTA corresponds to which state in the target DFA.
- Since we are looking for the minimum DFA, we want the set of states \(Q\) in the mapping \(f\) to be of minimum cardinality.
- A search algorithm identifies a mapping function \(f\) subject to a number of constraints:

\footnotetext{
\({ }^{10}\) Oliveira and Silva refer to this as an LFDFA (loop free automaton).
}
- If two states in the APTA are type incompatible (accepting/rejecting), then they do not correspond to the same state in the DFA obtained by the mapping function.
- If two states \(q_{i}\) and \(q_{j}\) are mapped to the same state and have successor states \(q_{k}\) and \(q_{l}\) for some symbol, then \(q_{k}\) and \(q_{l}\) must also be mapped together in the same state (this is merging for determinisation).
- BIC is a search algorithm which explores a tree of mapping assignments and backtracks when an assignment results in a conflict. The method improves over earlier efforts (such as that by Biermann BBP75), by being able to perform dependency directed backtracking [RN03] and thereby considerably improve the efficiency of the algorithm.

On randomly constructed problem instances, BIC was able to identify the target DFA in "very little time" for all problems where the target had up to 11-12 states, and became, progressively, less effective on larger ones. We note that the exact nature of the method (in the sense that BIC identifies the exact target DFA) makes its unsuitable for the types of Abbadingo-style problems we are considering in this dissertation. Nonetheless, the authors argue that the techniques they described will be "extremely effective in a variety of other situations". This is indeed the case. We see similar ideas being used when graph colouring and Boolean satisfiability techniques are applied to the DFA learning problem next.

\subsection*{5.2.3 Graph (Vertex) Colouring}

An interesting approach to deal with the regular inference problem is to transform it into an instance of a graph colouring problem. In CN97, Coste and Nicolas describe such a method with encouraging results. The general idea is to:
- Assign a distinct colour to every state in the target DFA.
- Every state in the APTA constructed from some training set corresponds to an unknown colour in the target DFA. Moreover, the states in the APTA correspond to the nodes in the graph colouring problem.
- Two vertices in the graph are connected by an edge if they are not compatible (i.e. they must have distinct colours). In other words, these edges represent a number of inequality constraints.
- If two states are merged together, then their successors (for the same transition label) need to be merged too in order to satisfy merging for determinism (see Section 5.1). This represents a number of equality constraints \(\sqrt{11}\).
- Inferring an \(n\)-state hypothesis is equivalent to finding an \(n\)-colouring of the graph subject to these constraints.
- The authors use DSATUR Bré79] for this task.
- Once a colouring is obtained, states in the APTA having the same colour are merged together to return a hypothesis.
- Experimentation was made on very small target DFA instances with the authors acknowledging that harder problems such as those in the Abbadingo One competition still need to be studied.

In [CFV12], Costa Florêncio and Verwer note that while inequality constraints translate to graph colouring "in a very natural way" (they are edges in the graph), the equality constraints greatly complicate matters. Specifically, in the method used by Coste and Nicolas, the equality constraints are propagated dynamically and, as such, the graph colouring instance is being continuously modified on the fly. Costa Florêncio and Verwer propose the first method to translate both types of constraints into a pure graph colouring instance. They also discuss the complexity bounds on the learning task, as well as a family of algorithms suitable for identifying either the exact target DFA or an approximation of it.

\subsection*{5.2.4 Satisfiability Solvers}

One of the primary motivators for using Boolean satisfiability (SAT) methods to infer DFAs is that the techniques are well understood and mature. Specifically,

\footnotetext{
\({ }^{11}\) We note that these constraints were mentioned earlier in Section 5.2.2 when discussing Oliveira and Silva's constraint satisfaction algorithm OMS98.
}

SAT solvers exploit advanced search techniques such as conflict analysis and intelligent back-tracking [HV13].

The general approach is to translate a DFA learning problem into a conjunctive normal form (CNF) formula and use a SAT solver to deal with it. In HV10, Heule and Verwer describe a direct encoding method to encode DFAs as CNF. The procedure is outlined as follows: (i) translate DFA identification into a graph colouring problem, (ii) translate the graph colouring problem into SAT, and (iii) use a SAT solver on the problem instance. While, at the time of writing, the procedure used by the authors represented a state-of-the-art translation, the method nonetheless results in an encoding of \(\mathcal{O}\left(|C|^{2}|V|^{2}\right)\) clauses. The authors report that this encoding is too hard for current SAT solvers to deal with ( \(C\) is the number of colours, and \(V\) is the number of vertices in the graph/DFA). An alternative compact encoding is proposed and evaluated which requires \(\mathcal{O}\left(|C|^{2}|V|\right)\) clauses. The algorithm is summarised as follows:
1. Find a large clique \(L\) of vertices in the graph representing an APTA.
2. Initialise a set of colours \(C\) such that \(|C|=|L|\).
3. Construct a CNF formula by translating the APTA.
4. Use a SAT solver to solve the formula (the authors used PicoSAT [Pic]).
5. If the formula is unsatisfiable, add a colour to \(C\) and go back to step 3.
6. Return the DFA from the formula found in step 4 .

We summarise the setup and results obtained by HV10 next:
- At the time of writing, state-of-the-art SAT solvers can deal with problems of up to 5 million clauses, whereas more difficult problems in Abbadingo One require upwards of 100 million clauses to represent. Because of this, the authors suggest applying a "few steps" of EDSM to compact the initial tree (APTA) and proceed with their SAT method.
- Experiments were conducted for DFAs whose sizes range between 16 and 21 states, and compared their results against benchmark EDSM-based implementations. For target DFAs in this size class, their SAT method was found
to be very competitive with, and in some cases, outperformed the benchmark implementations.

In HV13, Heule and Verwer refine on their earlier work to develop the StaMinA competition \({ }^{12}\) winning algorithm dfasat. The procedure is outlined as follows:
- The DFA identification problem is encoded into SAT.
- Since the resulting encoding is too large for state-of-the-art SAT solvers, a number of greedy EDSM merge steps are performed. Each merge step reduces the size of the DFA, and when the DFA is sufficiently small, the SAT method can be applied.
- Of course, this makes the result dependent on the performance of EDSM. Since EDSM is not well suited to the kind of problems described in the StaMinA competition, a new heuristic is proposed which is more suitable for these kinds of problem instances.
- Since greedy heuristics can lead to suboptimal solutions, the computed evidence score is randomised and run several times before returning the best solution found. In other words, the randomised evidence score is computed as RandomisedScore \(=\operatorname{Random}() \times \operatorname{ActualScore}\), where Random () is a function which returns a random value between 0.0 and 1.0.
- Using this randomised scoring procedure results in many possible candidate solutions which the authors refer to as an ensemble of automata (inspired by Dietterich [Die00]). The ensemble of automata are then used to classify an unseen string \(s\) as follows:
\[
\operatorname{CLASS}(s)= \begin{cases}1 & \text { if } \geq 50 \% \text { of automata accept } s \\ 0 & \text { otherwise }\end{cases}
\]

On the hardest problems in the StaMinA competition having a 50 -symbol alphabet, EDSM achieves an accuracy score of \(52 \%\), whereas dfasat improves

\footnotetext{
\({ }^{12}\) Recall that the StaMinA competition was concerned with learning for DFAs having large alphabets \(\mathrm{WBD}^{+} 10\).
}
to \(95 \%\). This represents a substantial improvement over the state-of-the-art in software model DFA identification.

\subsection*{5.2.5 Connectionist Approaches}

Our research indicates that using recurrent neural networks (RNNs) and other connectionist approaches mostly seem to work for inferring very simple languages. The general idea is based on the fact that an RNN trained with a set of labelled strings, not only learns the behaviour of the grammar (automaton), but also its state representation. In other words, the RNN internally encodes a finite state automaton CSSM89. Attempts to exploit this result may be found in the work done by [GSC \({ }^{+} 90\), WK91], ZGS93, and AS94a. The typical approach is:
- Train an RNN to classify the strings in some set \(S=\left\langle S_{+}, S_{-}\right\rangle\). Usually, the RNN has a one-bit output \([0,1]\) corresponding to whether the input string is accepting or rejecting.
- Once the network is trained, a clustering (such as kNN or hierarchical) operation is performed on the activations of the hidden units in the network.
- Starting with the initial single-point clusters, it can be shown that there is a one-to-one correspondence between those initial cluster points and the states in the prefix tree acceptor.
- The two closest clusters are merged together (AS94a uses centroid Euclidean distance).
- This merging of clusters is performed repeatedly until we obtain an inconsistent automaton.
- The final automaton is then the last consistent automaton found.

Unfortunately, beyond a sense that the technique only works for small target DFA instances, we cannot seem to find an extensive evaluation of these methods. Nonetheless, we find the approach interesting, and a possible source of inspiration.

\subsection*{5.3 Summary}

In the first part of this chapter, we have described a variety of state merging algorithms which we classify as either monotonic, greedy heuristics (e.g. RPNI, EDSM), or greedy heuristics augmented with search (SAGE, Ed-Beam, TBWEDSM). This distinction will be useful later on when we will be describing our attempts at learning DFAs when training data is sparse. We note that in our survey we have excluded exhaustive backtracking algorithms such as mmm BF72, BBP75, OE96, bica OMS98, and exbar Lan99 due to their impracticality when dealing with large DFAs such as those described in the Abbadingo One competition. We refer readers interested in these methods to Cic02, where a good overview of these methods may be found. Finally, we suggest [CK03] as an alternative survey of these methods and results, as well as dlH05 for a more general exposition. In [BO05], Bugalho and Oliveira also present a very good summary of several search techniques which wrap greedy heuristics such as EDSM.

The second part of this chapter was dedicated to introducing a number of techniques as alternatives to the state merging algorithms we have discussed earlier. While being very innovative, a number of them seem to suffer from the fact that their performance does not scale well for larger Abbadingo-style target DFAs. Nonetheless, the principles behind several of them will be immensely useful to us. For instance, the ideas behind the graph colouring methods discussed here allow us to develop techniques which will help us better understand and visualise which merges in a hypothesis are beneficial and which ones are not. Likewise, understanding how current evolutionary methods work allows us to develop alternative approaches to deal with harder problems and obtain better results.

\section*{Chapter 6}

\section*{Properties of State Merging}

In this chapter, we are concerned with the properties of the state merging operator used by many of the algorithms described in Chapter 5. These include merge identity, merge ordering, permitting and blocking merges, included and root merges, and so called orphaned states. In this discussion, we also cover a number of properties related to the EDSM heuristic, as well as those of sequences or paths of merges in the search space. We conclude by defining colour-compatible merges, and investigating how these can be used to describe what 'good' merges are, and also as a tool for analysing merge sequences. The properties investigated in this chapter are relevant to the design of the three algorithms we are proposing in this dissertation.

\subsection*{6.1 DFAs are Partitions, States are Blocks}

Recall that a quotient automaton is an automaton induced by some partitioning of states with respect to some reference automaton (quotient automata have been defined in Section 3.1.1). As an example, suppose we have the DFA shown in Figure 6.1 (i) and partition its states as follows: \(\pi=\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}, q_{4}\right\}\right\}\). From this partitioning, we derive the corresponding quotient DFA shown in Figure 6.1 (ii) where we see that the states \(q_{3}\) and \(q_{4}\) are now in the same block.


Figure 6.1: A DFA (i) and its corresponding quotient DFA (ii) for the partition \(\pi=\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}, q_{4}\right\}\right\}\). The states \(q_{3}\) and \(q_{4}\) have been merged together.

Quotient automata allow us to reason about DFAs, states, and merges as follows:
- A DFA is a partition with respect to some reference DFA (in most cases, this reference DFA is the APTA). In the example shown in Figure 6.1, the partition \(\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}, q_{4}\right\}\right\}\) with respect to the APTA/DFA (i) is the DFA (ii).
- In this sense, states in a DFA are synonymous with blocks in a partition.
- When discussing the merge operation in Section 5.1.3, we defined a merge as the union of one or more pairs of blocks in a partition subject to some constraints. Back to the example in Figure 6.1, we can say that merging the states \(q_{3}\) and \(q_{4}\) maps the initial partition \(\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{\boldsymbol{q}_{\boldsymbol{3}}\right\},\left\{\boldsymbol{q}_{\boldsymbol{4}}\right\}\right\}\) to the partition \(\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{\boldsymbol{q}_{\mathbf{3}}, \boldsymbol{q}_{4}\right\}\right\}\).

Reasoning about DFAs, states, and merges this way has the following implications:
- Partitions and blocks are sets, and a merge corresponds to the union of blocks in a partition. This allows us to rely on the algebraic properties of set union when discussing merges and their properties.
- After a merge is executed, the derived DFA has a different set of states. Using block notation, we can refer to the states in the derived DFA in terms of the states in the initial DFA. For instance in Figure 6.1, the block \(B\left(\pi, q_{3}\right)\) corresponds to the state \(\left\{q_{3}\right\}\) in (i) while also referring to the 'compound' state \(\left\{q_{3}, q_{4}\right\}\) in (ii). As a matter of fact, in DFA (ii) we can say that states \(q_{3}\) and \(q_{4}\) are equivalent.
- This reasoning allows us to interchangeably refer to DFAs as partitions, and states as blocks.

\subsection*{6.2 General Properties}

\subsection*{6.2.1 The Initial Partition}

The initial partition \(\pi\) of some DFA having states \(Q\) is the partition \(\pi=\{\{q\}\) : \(\forall q \in Q\}\). In other words, the initial partition contains one distinct block per state in the DFA. In the context of initial partitions, the DFA in question is usually an APTA. As an example, consider the APTA shown in 6.1 (i). The initial partition constructed from this APTA is \(\pi=\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}\right\},\left\{q_{4}\right\}\right\}\), where every state in the APTA is in its own block in the partition.

\subsection*{6.2.2 Properties of the Join and Merge Operations}

\section*{A Recap of the Join and Merge Operations}

Recall that, in Section 5.1.3, we defined the join operation on a pair of blocks in a partition as the union of those blocks. Specifically, the join operation is \(J\left(\pi, B_{i}, B_{j}\right)=\left(\pi \cup\left(B_{i} \cup B_{j}\right)\right)-\left\{B_{i}, B_{j}\right\}\). As can be seen in Figure 6.2 below, the quotient automaton arising following a join could very well be non-deterministic.


Figure 6.2: The DFA (i) corresponds to the initial partition \(\pi=\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\}\right.\), \(\left.\left\{q_{3}\right\},\left\{q_{4}\right\}\right\}\). Joining the two blocks \(\left\{q_{1}\right\}\) and \(\left\{q_{3}\right\}\), results in the partition \(\pi=\) \(\left\{\left\{q_{0}\right\},\left\{q_{1}, q_{3}\right\},\left\{q_{2}\right\},\left\{q_{4}\right\}\right\}\) which corresponds to the NFA (ii).

To resolve this non-determinism, a merging for determinisation constraint was defined in Section 5.1.3. In order to merge a pair of blocks such that the resulting quotient automaton is deterministic, in addition to joining the pair of block in question, we also have to recursively join all the pairs of blocks established by the determinisation constraint. Back to the example in Figure 6.2, to resolve the non-determinism caused by joining the blocks \(\left\{q_{1}\right\}\) and \(\left\{q_{3}\right\}\), we also have to join the blocks \(\left\{q_{2}\right\}\) and \(\left\{q_{4}\right\}\). In other words, to merge a pair of blocks \(\left(B_{i}, B_{j}\right)\) for determinism, we have to join all the block pairs in a set \(X=\left\{\left(B_{i}, B_{j}\right), \ldots\right\}\) which is defined by the deterministic merge operation.

\section*{Identity Joins}

Joining a block in a partition with itself is a null operation, and will be referred to as an identity join. Specifically \(J\left(\pi, B_{i}, B_{i}\right)=\pi\). This follows from the definition of the join operation which replaces a pair of blocks by their union, and that the union of a set with itself is the same set.

\section*{Ordering of Blocks in a Join}

Joining some block \(B_{i}\) with a block \(B_{j}\) results in the same partition as joining the block \(B_{j}\) with \(B_{i}\). Specifically, \(J\left(\pi, B_{i}, B_{j}\right)=J\left(\pi, B_{j}, B_{i}\right)\). This follows since set
union commutes.

\section*{Ordering of Join Operations is Unimportant}

Starting with a partition \(\pi\), and a number of join operations between blocks in \(\pi\), we would obtain the same partition regardless of the order in which the join operations are performed. If \(B_{i}, B_{j}, B_{k}\), and \(B_{l}\) are blocks in a partition, then \(J\left(J\left(\pi, B_{i}, B_{j}\right), B_{k}, B_{l}\right)=J\left(J\left(\pi, B_{k}, B_{l}\right), B_{i}, B_{j}\right)\). This follows since the join operation is a union of two blocks (sets of states), and set union is associative. Moreover, since set union is also associative on an arbitrary finite number of sets Hal60, this property will hold for any ordering of any number of join operations. This behaviour is illustrated in Figure 6.3 below.


Figure 6.3: Ordering of join operations is unimportant.

\section*{Identity Merges}

Merging \({ }^{1}\) a block in the partition of a quotient DFA with itself is a null operation, and will be referred to as an identity merge. This follows from:
- A merge \(M\left(\pi, B_{i}, B_{j}\right)\) involves joining the blocks \(\left(B_{i}, B_{j}\right)\) as well as any further blocks to resolve non-determinism.
- Since, in this case, \(B_{i}=B_{j}\), the transitions outgoing \(B_{i}\) are the same as those outgoing \(B_{j}\). Moreover, since the partition \(\pi\) corresponds to a DFA, it follows that there is no further non-determinism to correct for (i.e. we only need to perform the first join and there are no further ones needed).
- Joining a block with itself results in the same partition.

\footnotetext{
\({ }^{1}\) Whenever we refer to merge operations, we are always referring to deterministic merge operations.
}

\section*{The Set of Joins Required by a Merge Remains Unchanged}

Let \(X\) be the set of block pairs which need to be joined together in order to perform a merge \(m\) for determinism. Suppose, that \(\pi\) is the partition obtained following any number of merges starting at the partition of an APTA. The set of block pairs \(X\) which need to be joined together by the merge \(m\) remains unchanged for any \(\pi\). In other words, the set of block pairs which need to be joined by a merge, does not change after any sequence of merges. This is illustrated in Figure 6.4 where:
- The partition for DFA (i) is \(\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}\right\},\left\{q_{4}\right\},\left\{q_{5}\right\}\right\}\).
- Should we wish to perform the merge \(m\) between the blocks \(\left\{q_{1}\right\}\) with \(\left\{q_{2}\right\}\) for determinism, we need to join all the block pairs \(X=\left\{\left(\left\{q_{1}\right\},\left\{q_{2}\right\}\right),\left(\left\{q_{4}\right\},\left\{q_{5}\right\}\right)\right\}\).
- Suppose, we instead merge the block \(\left\{q_{1}\right\}\) with \(\left\{q_{3}\right\}\) to obtain the DFA (ii) having the partition \(\left\{\left\{q_{0}\right\},\left\{q_{1}, q_{3}\right\},\left\{q_{2}\right\},\left\{q_{4}\right\},\left\{q_{5}\right\}\right\}\).
- Performing the merge \(m\) between the blocks \(\left\{q_{1}\right\}\) with \(\left\{q_{2}\right\}\) in this new partition still requires us to join the same block pairs in \(X\) and gives the DFA (iii).


Figure 6.4: The set of block pairs which need to be joined by a merge does not change after any sequence of merges.

To show that this property holds, suppose that the set of joins \(X\) required to perform some merge in a partition \(\pi\) does change should the same merge is performed in some other partition \(\pi^{\prime}\). By definition of the merge operation, the set of joins which needs to be performed by a merge is only affected by the transitions
in the quotient automaton. This means that \(X\) can only change if transitions have been added and/or removed during the sequence of operations that led to \(\pi^{\prime}\). This is impossible, by the construction of quotient automata (see Definition 3.1. As an example, consider the APTA shown in Figure 6.5 and note that all the transitions in the APTA also exist in the derived DFA, and that no new ones have been introduced.


Figure 6.5: All the transitions in an APTA also exist in a derived DFA.

\section*{Ordering of Merge Operations is Unimportant}

Starting with a partition \(\pi\), and a number of (deterministic) merge operations between blocks in \(\pi\), we would obtain the same partition regardless of the order in which the merge operations are performed. If \(B_{i}, B_{j}, B_{k}\), and \(B_{l}\) are blocks in a partition, then \(M\left(M\left(\pi, B_{i}, B_{j}\right), B_{k}, B_{l}\right)=M\left(M\left(\pi, B_{k}, B_{l}\right), B_{i}, B_{j}\right)\). This follows from the previous properties that join order is unimportant, and that the set of joins which needs to be performed by a merge remains unchanged:
- Suppose that \(X\) is the set of joins required to perform the merge \(M\left(\pi, B_{i}, B_{j}\right)\), and that \(Y\) is the set of joins required to perform the merge \(M\left(\pi, B_{k}, B_{l}\right)\).
- Since the sets of joins \(X\) and \(Y\) remain unchanged in any partitioning of the same states, in any case, we are performing all the joins in \(X \cup Y\) which can be applied in any order to arrive at the same partition.

The unimportance of merge ordering is illustrated in an example in Figure 6.24 at the end of this chapter. Here we see how different permutations of the same merges lead us to exactly the same DFA.

\subsection*{6.2.3 Permitting, Blocking, Included, and Root Merges}

Consider the DFA shown in Figure 6.6. Note that both the merges \(\left(q_{1}, q_{2}\right)\) and \(\left(q_{1}, q_{3}\right)\) are valid in the DFA. However, if we first execute the merge \(\left(q_{1}, q_{2}\right)\), then the merge \(\left(q_{1}, q_{3}\right)\) is now no longer possible since the compound state \(\left\{q_{1}, q_{2}\right\}\) in the resulting DFA is accepting whereas the state \(q_{3}\) is rejecting. We say that the merge \(\left(q_{1}, q_{2}\right)\) blocks the merge \(\left(q_{1}, q_{3}\right)\). On the other hand, after performing \(\left(q_{1}, q_{2}\right)\), the merge \(\left(q_{3}, q_{4}\right)\) is still possible and we say that the merge \(\left(q_{1}, q_{2}\right)\) permits the merge \(\left(q_{3}, q_{4}\right)\). Now, suppose that we perform the merge \(\left(q_{0}, q_{1}\right)\) in the same APTA to obtain the partition \(\left\{\left\{q_{0}, q_{1}, q_{2}\right\},\left\{q_{3}, q_{4}\right\}\right\}\). Note that to satisfy the determinisation constraint, performing this merge implies that \(\left(q_{0}, q_{2}\right),\left(q_{1}, q_{2}\right)\), and \(\left(q_{3}, q_{4}\right)\) have also been merged together. We say that these merges are included in the original merge \(\left(q_{0}, q_{1}\right)\). Any merge out of all the possible merges in a DFA which is not included in any other merge is referred to as a root merge.


Figure 6.6: This DFA illustrates cases of permitting, blocking, included, and root merges.

\section*{Definition 6.1: Permitting and Blocking Merges}

Suppose that \(m_{1}\) and \(m_{2}\) are both valid merges in some DFA. If after merging \(m_{1}\) the merge \(m_{2}\) remains a valid merge, we say that \(m_{1}\) permits \(m_{2}\) and write \(m_{1} \multimap m_{2}\). On the other hand, if after performing \(m_{1}\) the merge \(m_{2}\) becomes invalid, we say that \(m_{1}\) blocks \(m_{2}\) and write \(m_{1}\) to \(m_{2}\). The set \(\mathcal{B L} \mathcal{K}_{m}\) of all merges blocked by some merge \(m\) is \(\mathcal{B} \mathcal{L} \mathcal{K}_{m}=\left\{m_{k}: m\right.\) fo \(\left.m_{k}\right\}\).

\section*{Definition 6.2: Included Merges}

Let \(\pi_{1}\) be the partition obtained after merging \(m_{1}\) in \(\pi\), and let \(m_{2}=\left(q, q^{\prime}\right)\) be some other merge. Merge \(m_{1}\) includes the merge \(m_{2}\), written as \(m_{1} \rightarrow m_{2}\), whenever \(m_{1} \neq m_{2}\) and \(B(q, \pi) \neq B\left(q^{\prime}, \pi\right)\) and \(B\left(q, \pi_{1}\right)=B\left(q^{\prime}, \pi_{1}\right)\). In other words, the merge \(m_{1}\) includes some other distinct merge \(m_{2}\) whenever the pair of states \(m_{2}\) were not in the same block before the merge \(m_{1}\) but are in the same block afterwards. Whenever a merge \(m_{1}\) does not include a merge \(m_{2}\), we write \(m_{1} \nrightarrow m_{2} . \mathcal{I N C}_{m}=\left\{m_{k}: m \rightarrow m_{k}\right\}\) is the set of merges included in the merge \(m\).

\section*{Definition 6.3: Root Merges}

Let \(\mathcal{M}=\left\{m_{1}, m_{2}, \ldots, m_{k}\right\}\) be the set of all the valid merges in some DFA. The root merges \(\mathcal{R T}\) of a DFA are a subset of \(\mathcal{M}\) which only contains merges which are not included in any other merge in \(\mathcal{M}\). Specifically:
\[
\mathcal{R T}=\mathcal{M}-\left(\mathcal{I N C}_{m_{1}} \cup \mathcal{I N C} \mathcal{C}_{m_{2}} \cup \ldots \cup \mathcal{I N} \mathcal{N}_{m_{k}}\right)
\]

Consider the DFA shown in Figure 6.6 earlier. Its merges, blocks, inclusions, and roots are shown in Table 6.1. Suppose we construct a graph where every node is a valid merge, and edges are drawn between included merges and blocked merges. We call such a graph a merge relation graph. An example of a merge relation graph is shown in Figure 6.7 where blue edges with arrowheads denote inclusion, red dashed edges denote blocking, and shaded nodes (merges) are root merges. Inclusions are directed since the relation is not symmetric while blocks are undirected since the relation is symmetric (discussed later).
\begin{tabular}{llll}
\hline Merge & Partition & Blocks & Includes \\
\hline \hline\(\left(q_{0}, q_{1}\right)^{*}\) & \(\left\{\left\{q_{0}, q_{1}, q_{2}\right\},\left\{q_{3}, q_{4}\right\}\right\}\) & \(\left(q_{0}, q_{3}\right),\left(q_{0}, q_{4}\right),\left(q_{1}, q_{3}\right),\left(q_{1}, q_{4}\right)\) & \(\left(q_{0}, q_{2}\right),\left(q_{1}, q_{2}\right),\left(q_{3}, q_{4}\right)\) \\
\(\left(q_{0}, q_{2}\right)\) & \(\left\{\left\{q_{0}, q_{2}\right\},\left\{q_{1}\right\},\left\{q_{3}\right\},\left\{q_{4}\right\}\right\}\) & \(\left(q_{0}, q_{3}\right),\left(q_{0}, q_{4}\right)\) & - \\
\(\left(q_{0}, q_{3}\right)^{*}\) & \(\left\{\left\{q_{0}, q_{3}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{4}\right\}\right\}\) & \(\left(q_{0}, q_{1}\right),\left(q_{0}, q_{2}\right),\left(q_{1}, q_{3}\right)\) & - \\
\(\left(q_{0}, q_{4}\right)^{*}\) & \(\left\{\left\{q_{0}, q_{4}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}\right\}\right\}\) & \(\left(q_{0}, q_{1}\right),\left(q_{0}, q_{2}\right),\left(q_{1}, q_{4}\right)\) & - \\
\(\left(q_{1}, q_{2}\right)\) & \(\left\{\left\{q_{0}\right\},\left\{q_{1}, q_{2}\right\},\left\{q_{3}\right\},\left\{q_{4}\right\}\right\}\) & \(\left(q_{1}, q_{3}\right),\left(q_{1}, q_{4}\right)\) & - \\
\(\left(q_{1}, q_{3}\right)^{*}\) & \(\left\{\left\{q_{0}\right\},\left\{q_{1}, q_{3}\right\},\left\{q_{2}\right\},\left\{q_{4}\right\}\right\}\) & \(\left(q_{0}, q_{1}\right),\left(q_{0}, q_{3}\right),\left(q_{1}, q_{2}\right)\) & - \\
\(\left(q_{1}, q_{4}\right)^{*}\) & \(\left\{\left\{q_{0}\right\},\left\{q_{1}, q_{4}\right\},\left\{q_{2}\right\},\left\{q_{3}\right\}\right\}\) & \(\left(q_{0}, q_{1}\right),\left(q_{0}, q_{4}\right),\left(q_{1}, q_{2}\right)\) & - \\
\(\left(q_{2}, q_{3}\right)\) & Invalid merge & & \\
\(\left(q_{2}, q_{4}\right)\) & Invalid merge & \\
\(\left(q_{3}, q_{4}\right)\) & \(\left\{\left\{q_{0}\right\},\left\{q_{1}\right\},\left\{q_{2}\right\},\left\{q_{3}, q_{4}\right\}\right\}\) & - & - \\
\hline
\end{tabular}

Table 6.1: Partitions, blocks, roots, and inclusions for the merges in the DFA shown in Figure 6.6. Asterisks denote root merges.


Figure 6.7: A merge relation graph. Blue edges with arrowheads denote inclusion, red dashed edges denote blocking, and shading indicates a root merge.

\section*{Note}

This idea of permitting and blocking merges was introduced earlier as pairwise compatible and pairwise incompatible merges when we discussed S-EDSM in Section 5.1.7.

\section*{Blocking is Symmetric}

Suppose that \(m_{1}\) and \(m_{2}\) are two valid merges in a partition \(\pi\). If \(m_{1}\) to \(m_{2}\), then \(m_{2}\) fo \(m_{1}\). This follows from the argument:
- Suppose we perform the merge \(m_{1}\) followed by the merge \(m_{2}\) to obtain \(\pi\).
- By our hypothesis, the partition \(\pi\) must be invalid (since \(m_{2}\) has been blocked by \(m_{1}\) ).
- Since the order of merges is unimportant, performing the merges in reverse order results in exactly the same invalid partition \(\pi\).

Symmetry of the blocking relation is illustrated in an example in Figure 6.25 which may be found at the end of this chapter.

\section*{Blocking is Not Transitive}

Suppose that \(m_{1}, m_{2}\), and \(m_{3}\) are valid merges. If \(m_{1}\) to \(m_{2}\) and \(m_{2}\) to \(m_{3}\), it does not follow that \(m_{1}\) to \(m_{3}\). This property can be shown by constructing a simple counterexample. Let \(m_{1}=\left(q_{1}, q_{4}\right), m_{2}=\left(q_{1}, q_{3}\right)\), and \(m_{3}=\left(q_{1}, q_{2}\right)\) in the DFA shown in Figure 6.8 (i). Performing the merge \(m_{1}=\left(q_{1}, q_{4}\right)\) blocks \(m_{2}=\left(q_{1}, q_{3}\right)\), and performing the merge \(m_{2}=\left(q_{1}, q_{3}\right)\) blocks \(m_{3}=\left(q_{1}, q_{2}\right)\). However, performing the merge \(m_{1}=\left(q_{1}, q_{4}\right)\) does not block the merge \(m_{3}=\left(q_{1}, q_{2}\right)\).


Figure 6.8: The blocking relation is not transitive.

\section*{All Combinations of Merges in a Valid Sequence are Permitting}

Suppose that \(h=\pi_{0} \rightarrow \ldots \rightarrow \pi_{k}\) is a valid sequence of merges (a merge path), and let \(m\) and \(m^{\prime}\) be any two merges in \(h . m \multimap m^{\prime}\) holds for any \(m, m^{\prime} \in h\). This
follows from the fact that the order of merges is unimportant: any permutation of merges in \(h\) starting from \(\pi_{0}\) will lead us to exactly the same partition \(\pi_{k}\). So if the sequence of merges in \(h\) is valid, then so is any other permutation of it, and no merge in the permutation will block any other one.

\section*{Merge Inclusion is Not Symmetric}

Given any two valid merges \(m_{1}\) and \(m_{2}\), if \(m_{1} \rightarrow m_{2}\), it does not follow that \(m_{2} \rightarrow m_{1}\). We can see that this property holds by examining the inclusions shown in Table 6.1: the merge \(\left(q_{0}, q_{1}\right)\) includes \(\left(q_{0}, q_{2}\right)\) but not vice-versa.

\section*{Merge Inclusion is Transitive}

If \(m_{1} \rightarrow m_{2}\) and \(m_{2} \rightarrow m_{3}\), then \(m_{1} \rightarrow m_{3}\). This property follows from the (transitive closure) definition of the state merging operation given in Definition 5.1.3. Any merges included in some merge \(m\) correspond to one or more join operations required by \(m\). If merging \(m_{1}\) requires us to also perform all the joins in \(m_{2}\), and merging \(m_{2}\) requires us to also perform all the joins in \(m_{3}\), then the merge \(m_{1}\) implies that we also have to perform all the joins in \(m_{3}\).

\section*{The Border Set w.r.t. Root Merges}

Consider an APTA, the border set in the search space, and the set of root merges in the APTA. There exist border set DFAs in the search space which cannot be reached using only the root merges of the APTA. This can be shown by examining the border set DFA 19 in Figure 6.9. The root merges in the APTA are \(\left(q_{0}, q_{1}\right)\), \(\left(q_{0}, q_{3}\right),\left(q_{0}, q_{4}\right),\left(q_{1}, q_{3}\right)\), and \(\left(q_{1}, q_{4}\right)\). It is impossible to reach the border set DFA 19 without performing the included (non-root) merge \(\left(q_{0}, q_{2}\right)\). As a matter of fact, \(\left(q_{0}, q_{2}\right)\) is only included in the root merge \(\left(q_{0}, q_{1}\right)\), but performing \(\left(q_{0}, q_{1}\right)\) excludes the border set DFA from ever being reached. We also note that this border set DFA is minimal \((n=2)\).

Furthermore, suppose that \(m_{1} \rightarrow m_{2}\). Let \(\mathcal{B S}\left(m_{1}\right)\) be the set of border set DFAs reachable after performing \(m_{1}\), and let \(\mathcal{B S}\left(m_{2}\right)\) be the set of border set DFAs reachable after performing \(m_{2}\). There exist border set DFAs reachable after \(m_{2}\) that are not reachable if we perform \(m_{1}\) even though \(m_{1}\) includes \(m_{2}\). In Figure
6.9, the border set DFA 20 is reachable by \(\left(q_{1}, q_{2}\right)\) but not by \(\left(q_{0}, q_{1}\right)\) even though \(\left(q_{0}, q_{1}\right) \rightarrow\left(q_{1}, q_{2}\right)\).


Figure 6.9: Merge graph with highlighted (thick) paths to the border set DFA 19.

\section*{Blocked Merges and Inclusion}

If \(m_{1} \rightarrow m_{2}\), then the set of merges blocked by \(m_{2}\) must be a subset of the merges blocked by \(m_{1}\). To see that this property holds, suppose that the set of merges blocked by \(m_{2}\) is not a subset of the merges blocked by \(m_{1}\). This means that there exists some merge \(m\) which is invalid (blocked) after performing \(m_{2}\) but remains valid after performing \(m_{1}\). This is impossible since, by our hypothesis, merging \(m_{1}\) implies that we also have to merge \(m_{2}\), and if \(m\) is blocked by \(m_{2}\), then it must also be blocked by \(m_{1}\). We can see an example of this by examining the merges and blocks shown in Figure 6.7. The merge \(\left(q_{0}, q_{1}\right)\) blocks
the merges \(\left\{\left(q_{0}, q_{3}\right),\left(q_{1}, q_{3}\right),\left(q_{0}, q_{4}\right),\left(q_{1}, q_{4}\right)\right\}\), and the merge \(\left(q_{0}, q_{2}\right)\) blocks the merges \(\left\{\left(q_{0}, q_{3}\right),\left(q_{0}, q_{4}\right)\right\}\). Observe that \(\left(q_{0}, q_{1}\right) \rightarrow\left(q_{0}, q_{2}\right)\), and that \(\left\{\left(q_{0}, q_{3}\right)\right.\), \(\left.\left(q_{0}, q_{4}\right)\right\} \subseteq\left\{\left(q_{0}, q_{3}\right),\left(q_{1}, q_{3}\right),\left(q_{0}, q_{4}\right),\left(q_{1}, q_{4}\right)\right\}\) holds.

It follows that, if \(m_{1} \rightarrow m_{2}\) and \(m_{2}\) blocks some set of merges, then \(m_{1}\) must also block those same merges. Consider, yet again, the merges and blocks shown in Figure 6.7, where the merge \(\left(q_{0}, q_{2}\right)\) blocks the merges \(\left\{\left(q_{0}, q_{3}\right),\left(q_{0}, q_{4}\right)\right\}\). Since \(\left(q_{0}, q_{1}\right)\) includes \(\left(q_{0}, q_{2}\right)\), then \(\left(q_{0}, q_{1}\right)\) will also block \(\left\{\left(q_{0}, q_{3}\right),\left(q_{0}, q_{4}\right)\right\}\). This result also allows us to deduce that if some merge \(m_{1}\) blocks some other merge \(m_{2}\), then \(m_{2}\) must also be blocked by any other merge that includes \(m_{1}\). Back to Figure 6.7. we can see that the merge \(\left(q_{1}, q_{2}\right)\) blocks the merges \(\left\{\left(q_{1}, q_{3}\right),\left(q_{1}, q_{4}\right)\right\}\). Since \(\left(q_{1}, q_{2}\right)\) is included in \(\left(q_{0}, q_{1}\right)\) then \(\left(q_{0}, q_{1}\right)\) will block \(\left\{\left(q_{1}, q_{3}\right),\left(q_{1}, q_{4}\right)\right\}\) as well.

\subsection*{6.2.4 Orphaned States}

Examining the sequence of merges starting from an APTA to a border set DFA (final hypothesis) shown in Figure 6.10, we can observe that the state \(q_{1}\) remains unlabelled throughout.


Figure 6.10: A path leading to a DFA having unlabelled states.

The merge relation graph constructed from the above APTA is shown in Figure 6.11 next, where we omit the inclusion relation as it is unnecessary in our current argument. We also annotate the merge relation graph to show which merges would label the state \(q_{1}\) as either accepting or rejecting.


Figure 6.11: The merge relation graph for the merges in the APTA shown in Figure 6.10. The annotations show which merges would label the state \(q_{1}\) and how.

In the merge sequence shown in Figure 6.10, we first choose to merge the states \(\left(q_{0}, q_{4}\right)\). Doing so blocks the merge \(\left(q_{1}, q_{4}\right)\) so we:
1. Remove the merge \(\left(q_{0}, q_{4}\right)\) which we just performed, as well as all of its edges.
2. Remove the blocked merge ( \(q_{1}, q_{4}\) ) , as well as all of its edges.

We note that, so far, it is still possible to label the state \(q_{1}\) as accepting via \(\left(q_{1}, q_{5}\right)\) and rejecting via \(\left(q_{1}, q_{3}\right)\) as shown in Figure 6.12 .


Figure 6.12: The merge relation graph after performing the merge \(\left(q_{0}, q_{4}\right)\).

In our path, we next merge the states \(\left(q_{0}, q_{5}\right)\). This blocks \(\left(q_{1}, q_{5}\right)\), so we:
1. Remove the merge \(\left(q_{0}, q_{5}\right)\) we performed, as well as its edges.
2. Remove the blocked merge \(\left(q_{1}, q_{5}\right)\), as well as its edges.

It is now impossible for \(q_{1}\) to become accepting, as the only merge left that would label it is \(\left(q_{1}, q_{3}\right)\) which makes it rejecting. This is shown in Figure 6.13 below.


Figure 6.13: The merge relation graph after performing the merge \(\left(q_{0}, q_{5}\right)\). Now, \(q_{1}\) can never become an accepting state.

In our path, we finally merge \(\left(q_{2}, q_{3}\right)\). This blocks \(\left(q_{1}, q_{3}\right)\), so we:
1. Remove the merge \(\left(q_{2}, q_{3}\right)\) we performed, as well as all its edges.
2. Remove the blocked merge \(\left(q_{1}, q_{3}\right)\), as well as all its edges.

We can now see that no remaining merge can ever assign a label to \(q_{1}\), as the only merge that remains is \(\left(q_{4}, q_{5}\right)\) which does not affect its labelling. This means that \(q_{1}\) would remain unlabelled no matter which other merges are made from this point onwards. We say that the state is orphaned with respect to the sequence of merges we chose.

\section*{Observation}

Suppose we construct a merge relation graph, and to each merge (node) we attach information about which unlabelled states that merge would label and, also, whether each of those unlabelled states become accepting or rejecting by that merge. If we perform a merge that makes an unlabelled state accepting, we can simplify the graph by removing any merges (and their edges) which would make that state rejecting (and vice-versa). Furthermore, suppose that two merges \(m_{1}\) and \(m_{2}\) both label some unlabelled state \(q\). If the label assigned by \(m_{1}\) to \(q\) differs from the one assigned by \(m_{2}\), then \(m_{1}\) and \(m_{2}\) clearly block each other.

\subsection*{6.3 Properties of Merge Paths}

Consider a merge graph starting at an APTA and containing a border set of DFAs. Furthermore, suppose that \(E\) is any one of the DFAs in the border set, and that
\(P_{E}\) is the set of all paths in the merge graph starting at the APTA and leading to the border set DFA \(E\). Since merge ordering is unimportant, while each path in \(P_{E}\) is, nominally, a sequence of merges, we can actually treat any path as a set of merges. Moreover, as discussed earlier, no two merges in a path are blocking.

\section*{Definition 6.4: The Fundamental Set}

The fundamental set \(U_{E}\) with respect to some border set DFA \(E\) is the set of merges consisting of the union of all the merges \(m\) in all the paths \(P_{E}\) that lead to \(E\) :
\[
U_{E}=\bigcup_{m \in P_{E}} m
\]

There are \(n\) ! permutations of a fundamental set containing \(n\) merges. Since merge ordering is unimportant, while each of these permutations is a syntactically distinct path, each of them are equivalent (in the sense that they lead to the same DFA). Clearly, any ordering of merges in \(U_{E}\) will lead to the border set DFA E.

The significance of the fundamental set is that it is representative of, and executing its merges in any order serves as a substitute for, any of the paths in \(P_{E}\) that lead to a border set DFA \(E\). It then follows that it is sufficient to execute all the merges in a fundamental set (in any order) to reach the respective border set DFA. We will elaborate on this property later.

\section*{Definition 6.5: A Fundamental Graph}

Let \(\mathcal{B S}\) be a border set, let \(E\) be some DFA in \(\mathcal{B S}\), let \(U_{E}\) be the fundamental set of merges leading to \(E\), and let \(\bar{U}_{E}\) be any ordering of the merges in \(U_{E}\). \(\bar{U}_{E}\) is then a path whose nodes are DFAs starting from the APTA leading to the border set DFA \(E\), and whose edges are the merges. Moreover, let \(G=\left\{\bar{U}_{E} \mid E \in \mathcal{B S}\right\}\) be the set of all paths starting from the APTA to each of the DFAs in the border set.

A fundamental graph is a graph whose vertices are the set of all nodes (DFAs) in the paths in \(G\), and whose edges are the set of all edges (merges) in the paths in \(G\). In other words, a fundamental graph is constructed by superposing all the paths in \(G\). We note that, since we can have several syntactically distinct paths \(\bar{U}_{E}\) to a border set DFA, there will, likewise, be many syntactically distinct, yet equivalent, fundamental graphs. An example of a fundamental graph is shown in Figure 6.16.

\section*{Definition 6.6: The Significant Set}

The significant set \(I_{E}\) with respect to some border set DFA \(E\) is a set of merges consisting of the intersection of all the merges \(m\) in all the paths \(P_{E}\) that lead to \(E\) :
\[
I_{E}=\bigcap_{p \in P_{E}} p
\]

Similar to the observation we made for fundamental sets, there are \(n\) ! permutations of a significant set containing \(n\) merges. Since merge ordering is unimportant, while each of these permutations is a syntactically distinct sequence of merges, each of them are equivalent (in the sense that they lead to the same DFA).

The significant set represents the minimum set of merges, applied in any order, which are necessary to reach the respective border set DFA. However, executing all of them is not sufficient to reach it. We will elaborate on this property later.

\section*{Definition 6.7: The Common Set}

Let \(U_{i}\) and \(U_{j}\) be two fundamental sets (paths) to the border set DFAs \(i\) and \(j\) respectively. The common set, denoted by \(C_{i, j}\) is the set of merges obtained by the intersection of \(U_{i}\) and \(U_{j}\).

\section*{Definition 6.8: The Critical Merge}

Let \(U_{i}\) and \(U_{j}\) be two fundamental sets whose common set is \(C_{i, j}\). Suppose that we perform all the merges \(C_{i, j}\) in any order. We call any merge following these a critical merge due to the fact that this choice will certainly exclude either of the border set DFAs \(i\) or \(j\) from ever being reached.

We will now concretise these concepts using an example. Consider the merge graph shown below in Figure 6.14 which has five border set DFAs corresponding to the DFAs \(1,14,16,19\), and 20. In this example, the border set DFA 1 has the following eight paths leading to it:
\[
\begin{aligned}
& \left(q_{0}, q_{1}\right) \\
& \left(q_{1}, q_{2}\right) \rightarrow\left(q_{0}, q_{1}\right) \\
& \left(q_{0}, q_{2}\right) \rightarrow\left(q_{3}, q_{4}\right) \rightarrow\left(q_{0}, q_{1}\right) \\
& \left(q_{1}, q_{2}\right) \rightarrow\left(q_{3}, q_{4}\right) \rightarrow\left(q_{0}, q_{1}\right)
\end{aligned}
\]
\[
\begin{aligned}
\left(q_{0}, q_{2}\right) & \rightarrow\left(q_{0}, q_{1}\right) \\
\left(q_{3}, q_{4}\right) & \rightarrow\left(q_{0}, q_{1}\right) \\
\left(q_{3}, q_{4}\right) & \rightarrow\left(q_{0}, q_{2}\right) \rightarrow\left(q_{0}, q_{1}\right) \\
\left(q_{3}, q_{4}\right) & \rightarrow\left(q_{1}, q_{2}\right) \rightarrow\left(q_{0}, q_{1}\right)
\end{aligned}
\]


Figure 6.14: A merge graph starting from APTA of \(S_{+}=\{a a\}\) and \(S_{-}=\{b, a b\}\). Border set DFAs are shown in blue.

All these paths are the set \(P_{1}\), and the union of the individual merges in these paths gives us the fundamental set \(U_{1}=\left\{\left(q_{0}, q_{1}\right),\left(q_{0}, q_{2}\right),\left(q_{1}, q_{2}\right),\left(q_{3}, q_{4}\right)\right\}\). Recall, that applying all the merges in \(U_{1}\) in any order is sufficient to reach the border set DFA 1. The significant set is the intersection of the pairs in \(P_{1}\) and results in \(I_{1}=\left\{\left(q_{0}, q_{1}\right)\right\}\). Executing the merges in \(I_{1}\) in any order are necessary but not always sufficient to reach the border set DFA 1. Finally, consider the two fundamental sets \(U_{1}=\left\{\left(q_{0}, q_{1}\right),\left(q_{0}, q_{2}\right),\left(q_{1}, q_{2}\right),\left(q_{3}, q_{4}\right)\right\}\) and \(U_{19}=\left\{\left(q_{0}, q_{2}\right),\left(q_{1}, q_{3}\right),\left(q_{1}, q_{4}\right),\left(q_{3}, q_{4}\right)\right\}\). Their intersection is the common set
\(C_{1,19}=\left\{\left(q_{0}, q_{2}\right),\left(q_{3}, q_{4}\right)\right\}\). Any merge following those in \(C_{1,19}\) is critical as illustrated in Figure 6.15 below.


Figure 6.15: A critical merge.

Back to the example in Figure 6.14, we now construct the five fundamental sets to their respective border set DFAs. After choosing any ordering of merges for each of the five fundamental sets, we can construct the fundamental graph shown in Figure 6.16 where we also show the resulting critical merges. The orderings chosen in this example are:
- \(\bar{U}_{1}=\left(\left(q_{0}, q_{1}\right),\left(q_{0}, q_{2}\right),\left(q_{1}, q_{2}\right),\left(q_{3}, q_{4}\right)\right)\)
- \(\bar{U}_{14}=\left(\left(q_{0}, q_{3}\right),\left(q_{1}, q_{4}\right)\right)\)
- \(\bar{U}_{16}=\left(\left(q_{0}, q_{4}\right),\left(q_{1}, q_{3}\right)\right)\)
- \(\bar{U}_{19}=\left(\left(q_{0}, q_{2}\right),\left(q_{1}, q_{3}\right),\left(q_{1}, q_{4}\right),\left(q_{3}, q_{4}\right)\right)\)
- \(\bar{U}_{20}=\left(\left(q_{0}, q_{3}\right),\left(q_{0}, q_{4}\right),\left(q_{1}, q_{2}\right),\left(q_{3}, q_{4}\right)\right.\)


Figure 6.16: A fundamental graph constructed by superposing the paths to each of the 5 border set DFAs in the merge graph in Figure 6.14. The red arrow shows the point where a critical merge is about to be made. Dotted edges and loops are redundant identity merges.

\section*{The Sufficiency of Fundamental Sets}

The sufficiency of the merges in a fundamental set \(U_{E}\) to reach a border set DFA \(E\) follows from the following argument:
- Let \(P\) be the set of merges of some path starting from the APTA to the border set DFA \(E\).
- By definition of a fundamental set, \(P\) is a subset of \(U_{E}\).
- Since merge ordering is unimportant, we know that any permutation of \(U_{E}\) leads to the same DFA. This also includes the permutation starting with the merges in \(P\) which lead to \(E\) (the remaining merges in the permutation following the merges \(P\) will be identity merges).
- This means that any permutation of \(U_{E}\) will lead to (and is therefore sufficient to reach) \(E\).

\section*{All Merges in a Fundamental Set are Permitting}

If \(p\) and \(q\) are any two merges in a fundamental set \(U_{E}\), then \(p \multimap q\) for all \(p, q \in U_{E}\). This follows, since:
- By construction, the border set DFA \(E\) is a valid DFA (see Chapter 3), so any sequence of merges leading to it consists of valid merges.
- The merges (in any order) in a fundamental set are sufficient to reach the border set DFA \(E\).
- As discussed in Section 6.2 earlier, any two merges in any permutation of a valid sequence of merges are permitting.

\section*{The Necessity of Significant Sets}

The merges in a significant set \(I_{E}\) are always necessary to reach a border set DFA \(E\). This follows from the definition of significant sets: a significant set is the intersection of merges in every path that leads to a border set DFA and, hence, no path can ever lead to a border set DFA without using each of those merges.

\section*{The Insufficiency of Significant Sets}

The merges in a significant set \(I_{E}\) is not sufficient to reach a border set DFA \(E\). This property can be shown by example. Consider the merge graph in Figure 6.14 and the border set DFA 19. The significant set is \(I_{19}=\left\{\left(q_{1}, q_{3}\right),\left(q_{0}, q_{2}\right)\right\}\). While these two merges are required by any path that leads to the border set DFA 19, they are not sufficient to reach it by themselves.

\subsection*{6.4 Some Remarks Regarding EDSM}

When constructing a merge path, the EDSM heuristic scores all the candidate merges \({ }^{2}\) at every step, and selects the one having the highest score. If there is more than one merge having the highest score, the tie is, typically, broken randomly. In [LPP98], we see two slightly different yet, for the purposes of how EDSM actually selects merges, equivalent descriptions of how the score is computed:
1. "We award one point for each state label which, as a result of a merge, undergoes an identity check and turns out to be okay. Any mismatch results in a negative overall score.", and later
2. "A merge's score is the sum over equivalence classes of the following quantity: if there are conflicting labels in the class, minus infinity; if there are no labels in the class, zero; otherwise, the number of labels minus one. We subtract one because the first label in the class establishes the correct label for the class, but is not checked."

Consider the example shown in Figure 6.17, which shows a sequence of merges starting at an APTA:
- The first merge in APTA (i) is between the states \(\left(q_{3}, q_{4}\right)\), and gives the DFA shown in (ii). This merge has an EDSM score of 1 since we have a single label match between the rejecting states \(q_{3}\) and \(q_{4}\). This result is consistent with both descriptions of how EDSM scores merges.

\footnotetext{
\({ }^{2}\) If a windowing strategy is used, this scheme is slightly different. See Section 5.1.7
}
- The second merge is between the states \(\left(q_{0}, q_{2}\right)\) in DFA (ii) resulting in the DFA and partition (iii). Regarding this second merge:
- The state \(q_{0}\) is now merged with the state \(q_{2}\), and both of them are accepting. Using the first description of the EDSM score above, the score of this merge is 1 due to the single label-match.
- On the other hand, following the second explanation of how a merge is scored, we would need to inspect all the equivalence classes (blocks in the partition (iii)), and sum the counts (less 1) of the labelled states in each equivalence class. This would give an EDSM score of 2.


Figure 6.17: The computation of EDSM scores along a merge path.

Recall that a merge for determinism involves joining several blocks of states in the partition of a quotient \(\mathrm{DFA}^{3}\). Assuming that a merge is valid, every block in the resulting partition will not have mixed accepting/rejecting states. The computed label of a block (which corresponds to the label of a 'compound' state in the corresponding quotient DFA ) is determined by its contents:
- If a block only contains unlabelled states, then the type of the block is unlabelled too.
- If a block contains at least an accepting state, then the block is accepting (since we are assuming that the merge is valid, there are no mixed labelling in a block).

\footnotetext{
\({ }^{3}\) The join operation and deterministic merging were defined in Section 5.1.3
}
- If a block contains at least a rejecting state, then the block is rejecting.

Suppose that TypeMatch is a Boolean function which, given a pair of blocks, returns True if the computed labels (as described above) of the blocks are either both accepting or both rejecting, and returns False otherwise. Furthermore, let \(X\) be the set of join operations between pairs of blocks which need to be performed by a deterministic merge. The EDSM score of a merge following the first description in [LPP98] may be computed as:
\[
\operatorname{EDSM}=\sum_{\left(B, B^{\prime}\right) \in X} \begin{cases}1, & \text { if } \operatorname{TypeMatch}\left(B, B^{\prime}\right)  \tag{6.1}\\ 0, & \text { otherwise }\end{cases}
\]

The second description of the EDSM scoring function in LPP98] requires us to inspect equivalence classes (blocks in a partition). Assuming that a merge is valid (i.e. blocks do not contain mixed label states), the EDSM score of a merge, resulting in a partition \(\pi\) containing our equivalence classes, may be given by:
\[
\begin{equation*}
\operatorname{EDSM}(\pi)=\sum_{B \in \pi} \max ((\# \text { labels in } B)-1,0) \tag{6.2}
\end{equation*}
\]

\section*{EDSM and Cumulative EDSM}
- For every join operation, the size of a partition (the number of blocks) decreases by 1 (since two blocks will become one), and the EDSM score will increment by 1 when the computed labels of the blocks are, either both accepting, or both rejecting. An illustration of this is shown in Figure 6.18.
- It then follows that the value given by Equation 6.2 may be equivalently obtained by subtracting the total number of labelled states in a partition by the total number of blocks whose computed label is either accepting or rejecting. We also note that the total number of labelled states in a partition is exactly equal to the size of the training set (by construction, there is one labelled state in the APTA per string in the training set), and that this
number of labelled states does not change in any partitioning of those states (the join operations simply group states together). In Figure 6.18, we can see that the number and type of the labelled states (not blocks) in a partition do not change after any number of join operations. If LabelledBlocks is a function which takes a partition and returns the number of blocks whose computed label is either accepting or rejecting, then Equation 6.2 may be replaced by the Equation 6.3 below:
\[
\begin{equation*}
\operatorname{EDSM}(\pi)=\mid \text { Training Set } \mid-\operatorname{LabelledBLocks}(\pi) \tag{6.3}
\end{equation*}
\]
- Following our first observation, we can also deduce that, in a merge path starting from an APTA up to some DFA/partition, the sum of the EDSM scores given by Equation 6.1 at each merge step in the path, is equal to the value we obtain by Equations 6.2 or 6.3 . An example of this may be seen on several permutations of merge sequences in Figure 6.26 at the end of this chapter.
- In this sense, we will be referring to the score obtained by Equation 6.1 as the EDSM score, and refer to the score obtained by Equation \(6.2 \sqrt{6.3}\) as the cumulative EDSM score up to the DFA/partition in the sequence.


Figure 6.18: A number of joins performed in a partition. Only state labelings are shown: \(\mathrm{A}=\) Accepting, \(\mathrm{R}=\) Rejecting, and \(\mathrm{U}=\) unlabelled.

It should be noted that using, either the EDSM score given by Equation 6.1, or the cumulative EDSM score given by Equation 6.26 .3 results in exactly the
same inductive bias - the merge ordering obtained by both scoring methods is identical. To see why, consider the merge sequence shown in Figure 6.19 where we have two merge choices at the DFA \(X\), and the sequence of merges up to \(X\) has a cumulative EDSM score of C-EDSM. Suppose that the merge giving DFA \(A\) has a score of ScoreA, and that the merge giving DFA \(B\) has a score of ScoreB. The cumulative EDSM score up to DFA \(A\) will be C-EdSm + ScoreA, whereas the cumulative EDSM score up to DFA \(B\) will be C-EdSm + ScoreB. In other words, with respect to both merge choices, C-EDSM is a constant. Hence, although the EDSM score and the cumulative EDSM score values are different, the ordering given by both for the purposes of merge selection will be the same.


Figure 6.19: EDSM and cumulative EDSM scores order merges identically.

Finally, suppose that we are given the task of identifying an \(n\)-state target DFA from \(m\) training examples, and that all the states in the target DFA are labelled (as required by Abbadingo One). Using Equation 6.3, we can deduce that the sum of the EDSM scores (the cumulative EDSM score) starting from the partition of the APTA to the partition of the target DFA (which contains \(n\) labelled blocks) must be \(m-n\).

\subsection*{6.5 Colour-Compatible Merges}

By construction, each state in an APTA corresponds to exactly one state in the target DFA. In other words, if we had to assign a distinct colour to every state in the target DFA, then every state in the APTA will map to one of those colours. We can see an example of this in Figure 6.20, where every state in the target (i) has a distinct colour, and every state in the APTA (ii) constructed for \(S_{+}=\{b b, a b b\}\) and \(S_{-}=\{b, b a, a a\}\) maps to the corresponding state and colour in the target

DFA. Crucially, the quotient automaton obtained by partitioning the states in the APTA by colour corresponds exactly to the target DFA. Moreover, if the training set is symmetrically structurally complete, the set of all states in the APTA of any given colour (e.g. all the green states) will contain at least one labelled state and, therefore, it is impossible to have mixed accepting/rejecting labels in a set of states having the same colour. In Section 5.2.3, we have made reference to how Coste and Nicolas [CN97], and Costa Florêncio and Verwer [CFV12] have used this observation to pose DFA learning as a graph colouring problem, where inferring an \(n\)-state hypothesis corresponds to the assignment of \(n\) colours to the states of the APTA subject to constraints.

(i)


Figure 6.20: A coloured target DFA (i) and APTA (ii).

Suppose that, for the purpose of analysing DFA learning algorithms (i.e. not in 'real-world' DFA learning), an Oracle reveals the correct APTA colours to us. We can use this information to know whether a merge choice made by a heuristic, such as EDSM, is the correct one or not. For example, in Figure 6.21, we can see that while both the merges \(\left(q_{2}, q_{5}\right)\) and \(\left(q_{1}, q_{4}\right)\) are both valid merges, \(\left(q_{2}, q_{5}\right)\) is colour-compatible and leads to the target DFA, while \(\left(q_{1}, q_{4}\right)\) is not colourcompatible and, now, the target can never be identified. In this sense, the idea of colour-compatibility can be used to determine which are the good and bad merge choices.


Figure 6.21: Both the merges resulting in (ii) and (iii) are valid. However, the merge resulting in (iii) is not colour-compatible since red state \(q_{1}\) is 'mixed' with the blue state \(q_{4}\).

To be clear, knowing whether a merge is colour-compatible or not, requires us to consult an Oracle and, therefore, cannot be used for DFA learning. Nonetheless, strictly in the context of analysing the behaviour of learning algorithms such as EDSM, this colour-compatibility of merges is useful since it allows us to find answers to the following questions \({ }^{4}\)
- During a merge step, we can determine whether a colour-compatible merge exists in the set of highest EDSM-scoring merges. How often does this happen? If a good, colour-compatible merge does not exist in the highestscoring merges, how 'far away' is it? An illustration showing this scenario is shown in Figure 6.22.
- Can we construct a subspace of the space of all merges possible such that,

\footnotetext{
\({ }^{4}\) In the next chapter, we describe several experiments that answer these questions.
}
with high experimental probability, that subspace still contains a large number of colour-compatible merges which lead to a good solution?
- Can we compare the merge sequences made by EDSM to the 'ideal' sequences of colour-compatible merges leading to the target DFA to identify exactly when, and why, EDSM makes a mistake? An illustration showing this scenario is shown in Figure 6.23.
- Which properties of colour-compatible merge sequences leading to the target DFA can we discover? How can these properties be used to better understand how EDSM works, and how can they be used to develop new DFA learning methods?

\section*{Definition 6.9: Colour-Compatible Merges}

A colour-compatible merge is a deterministic merge where all the states in every block in the resulting partition have the same colour. Specifically:

Let \(C\) be the set of distinct target DFA colours, \(\rho: Q \rightarrow C\) be the function which maps the states \(Q\) in an APTA to a colour in \(C\), and \(\pi\) be the partition (quotient automaton) obtained after performing some merge \(m\). The merge \(m\) is colour-compatible if \(\forall b \in \pi, \forall q, q^{\prime} \in b: \rho(q)=\rho\left(q^{\prime}\right)\).


Figure 6.22: An illustration showing how a colour-compatible merge could be 'outside' of the highest-scoring merges (merges are ordered by score).


Figure 6.23: An illustration showing where a heuristic, such as EDSM, made the wrong, colour-incompatible merge choice. The target DFA can never be reached after this choice is made. The green path is a hypothetical sequence of merges performed by EDSM, and the blue path is the sequence of colour-compatible merges leading to the target.

\subsection*{6.6 Ending Remarks}

The properties and concepts we have discussed in this chapter allow us to better understand the structure of the search space, as well as the behaviour of the EDSM heuristic. The unimportance of merge ordering, blocking and included merges, the cumulative property of the EDSM score, and the colour-compatibility of merges will be especially important. In the next chapter, we will perform a number of experiments on Abbadingo-style problem instances which include establishing baseline results for evaluation, studying the properties of so called 'Oracle-assisted' merges, estimating where good colour-compatible merges can be found in a subspace of merges, and determining the extent to which the first merges in a sequence are important. In this chapter we will also present the three DFA learning methods we have developed in this dissertation.


Figure 6.24: An example showing how merge ordering is unimportant. Dotted loops are identity merges.


Figure 6.25: An example showing how the blocking relation is symmetric.


Figure 6.26: The sum of EDSM scores. Dotted loops are identity merges.

\section*{Chapter 7}

\section*{Baseline Experiments and Methodology}

In this chapter, we start by describing the exact Abbadingo One setup which will be used to create target DFAs and benchmark training sets. We then proceed by performing a number of experiments to help us better understand the behaviour of the EDSM heuristic, as well as how the construction of target DFAs and training sets affect our learning task. In these experiments we:
- Determine the typical sizes of APTAs, as well as the expected proportion of labelled states in an APTA with respect to both the total number of strings possible and the size of the APTA itself. This investigation is important to determine whether the proportion of labelled strings (at density 1 ) is stable across target DFA sizes. Furthermore, we study the effect on generalisation rate should these proportions change.
- Describe so called Oracle-assisted heuristics, and how they will be used to study the behaviour of ideal paths to the exact target DFA.
- Empirically identify the baseline performance of EDSM, W-EDSM, and a variant of EDSM which uses an optimal, Oracle-assisted tie-breaking strategy. The results of these experiments will be used to compare our methods against.
- Study the experimental likelihood of randomly generating a training set which is not structurally complete using the Abbadingo One competition
procedure. We also investigate to which extent structural incompleteness negatively affects EDSM.
- Investigate the effect of getting the first \(k\) merges in a merge sequence correct (i.e. the first \(k\) merges are colour-compatible).
- Whenever a merge is performed, the size (number of states) of the hypothesis decreases. We study the rates at which a hypothesis is 'compressing' when a state merging algorithm is converging to the true target DFA. Identifying the rate of reduction is important to determine whether and which low state reduction merges can be avoided when building a merge sequence.
- Develop the APTA Reduction Table structure which contains high state reduction merges in an APTA. We further determine the expected number of colour-compatible merges in the table. Our reasoning is that, while the reduction table contains considerably fewer merges than the total number of merges possible, it is still likely to contain several merges in common with ideal merge paths to the target DFA (created using an Oracle-assisted heuristic). Focusing the initial steps of a search in the space corresponding to the merges contained in the table is likely to result in several short sequences of colour-compatible merges which constrain a hypothesis well enough that, when extending using a label matching heuristic, would perform better than EDSM.
- Study the relationship between the length of a merge sequence constructed using a greedy heuristic and the error of the resulting hypothesis with respect to a test set. We determine that the exact \(n\)-state target DFA is never found in fewer than \(n+1\) merge steps, and that, using heuristics such as EDSM, the error of a hypothesis is proportional to the length of a merge sequence (as the length of a merge sequence increases, so does the error of the hypothesis).

Motivated by the results we have obtained following this analysis, the remainder of this chapter is dedicated to providing a detailed description of three DFA learning methods we have developed in this dissertation. The performance of each of these methods will be evaluated in the next chapter.
1. Our first method is the development of an ensemble of monotonic, greedy heuristics which are, collectively, able to identify low-error hypotheses on problem instances where EDSM does not. Each heuristic in the ensemble has been identified experimentally and represents a different inductive bias used in the search.
2. The second method, which we call the Delta Graph, involves constructing a structure corresponding to a subspace of the merges in the space of quotient DFAs for the problem instance. Experimentally, we find that this structure contains several short sequences of colour-compatible merges which, when extended using a label matching heuristic such as W-EDSM, results in a much higher likelihood of identifying a lower error hypothesis compared to EDSM.
3. Thirdly, we describe an evolutionary algorithm which, rather than attempting to evolve a representation of the target DFA directly or packing the states of an APTA into the partition of a quotient automaton, attempts to identify initial sequences of high-quality, colour-compatible merges. These sequences constrain a hypothesis well enough to allow, with high experimental probability, a label matching heuristic to identify the exact target DFA or a low-error hypothesis.

\subsection*{7.1 How Heuristics or DFA Learning Algorithms are Evaluated and Analysed}

The procedure used to evaluate the performance of a heuristic or DFA learning algorithm is illustrated in Figure 7.1 and is outlined as follows:
- Create a large (representative) number of Abbadingo problem instances consisting of target DFAs, and their corresponding training and test sets (the exact procedure is described in Section 7.4 below).
- For each problem instance, construct an APTA from the training set.
- Use a heuristic (or DFA learning algorithm) such as EDSM to construct a merge path from every APTA to a final hypothesis.
- Determine the error of a hypothesis with respect to the test set, as well as other features such as the size of the hypothesis DFA. If test string reaches an unlabelled (orphaned) state in a hypothesis, or is otherwise unparsable, that test string is classified as being rejected by the hypothesis. This behaviour is consistent with the principle that unparsable strings do not belong to the language.
- Aggregate and analyse these features over all the problem instances. For instance, an important metric is the number of hypotheses over the total number of problem instances which have an error of \(\leq 1 \%\) over the test set (the Abbadingo winning criterion).


Figure 7.1: Overview of how problem instances are constructed, and how heuristics are evaluated.

To facilitate the aggregation and the analysis of results, all problem instances and merge path extensions (one merge path per heuristic/algorithm) are stored in SQLite database format SQL.
- Each database contains an Experiments and a Paths table:
- The Experiments table contains all the problem instances including target DFAs, as well as their corresponding training sets, test sets, and the APTA. In this table, we also store other features about a problem instance such as the proportion of positive and negative strings in the training set.
- The Paths table contains a merge path extension for each merge heuristic and for each experiment in the Experiments table (i.e. there is a one-to-many relationship between the Experiments and the Paths table). Again, in the Paths table, we store other features such as the heuristic/algorithm used to construct the paths, and the error of the final hypothesis identified by the heuristic.
- The exact schema of the database, including the actual database files themselves may be found in the media accompanying this dissertation.
- The databases containing all the experimental runs referred to in this dissertation (problem instances, merge path constructions, and results) are described in Table 7.1 below.
\begin{tabular}{ll} 
Database & Description \\
\hline \hline 32-state target DFAs, all training sets at density 1 ( 607 strings) \\
\hline n32d607e1024.sqlite & \begin{tabular}{l}
1024 problem instances. Training sets are symmetri- \\
cally structurally complete.
\end{tabular} \\
& 512 problem instances. Training sets are chosen such \\
n32d607e512_EdsmFailing.sqlite & that none of the highest scoring EDSM merges (ties) \\
& in the APTA are colour-compatible. \\
n32d607e512_NotStructComp.sqlite & 512 problem instances. Trainings sets are at not sym- \\
& metrically structurally complete. \\
& 512 problem instances. Target DFAs do not contain \\
n32d607e64_EdsmFailing.sqlite & loop transitions. \\
& 64 problem instances. Training sets are chosen such \\
& that none of the highest scoring EDSM merges (ties) \\
& in the APTA are colour-compatible.
\end{tabular}
\begin{tabular}{|c|c|}
\hline n32d607e64_NotStructComp.sqlite & 64 problem instances. Trainings sets are at not symmetrically structurally complete. \\
\hline n32d607e64_NoLoops.sqlite & 64 problem instances. Target DFAs do not contain loop transitions. \\
\hline n32d607e128_GA.sqlite & 128 problem instances. Training sets are symmetrically structurally complete \\
\hline \multicolumn{2}{|l|}{64-state target DFAs, all training sets at density 1 (1,521 strings)} \\
\hline n64d1e1024.sqlite & 1024 problem instances. Training sets are symmetrically structurally complete. \\
\hline n64d1e512_Unrestricted.sqlite & 512 problem instances. May or may not: be symmetrically structurally complete, have loops, have colourcompatible in first EDSM-scoring rank. \\
\hline n64d1e64_GA.sqlite & 64 problem instances. Training sets are symmetrically structurally complete. \\
\hline \multicolumn{2}{|l|}{128-state target DFAs, all training sets at density 1 (4,382 strings)} \\
\hline n128d1e512.sqlite & 512 problem instances. Training sets are symmetrically structurally complete. \\
\hline
\end{tabular}

Table 7.1: Description of databases containing the experiments referred to in this chapter.

\subsection*{7.2 Oracle-Assisted Heuristics and Paths}

We will refer to any heuristic which uses any useful information not in the training set as an Oracle-assisted heuristic. An example of such a heuristic would be one which always selects colour-compatible merges (defined in Section 6.5) which lead to the exact target DFA. In this sense, the merge choices made by the Oracleassisted heuristic are always 'good'. Oracle-assisted heuristics are useful because they allow us to identify exactly when and where a heuristic makes a wrong choice. Any path constructed using such a heuristic will be called an Oracle-assisted merge path, and any shortest merge path starting from an APTA to the exact target will be called an ideal merge path.

\subsection*{7.3 Glossary of Heuristics}

In this chapter, we will tabulate and discuss the performance of several heuristics. For presentation purposes, we will be using the following short names to refer to each of them. Heuristics prefixed with a \(\dagger\) are those assisted by an Oracle.
\begin{tabular}{ll} 
EDSM & Reference EDSM as described in LPP98. \\
W-EDSM & \begin{tabular}{l} 
Reference EDSM with windowing as described in \\
Reduction
\end{tabular} \\
\hline Selects merges which reduce the size of the current hy- \\
pothesis most. \\
W-Reduction & Same as reduction but with windowing. \\
\(\dagger\) EDSM-TieCC & \begin{tabular}{l} 
Reference EDSM where an Oracle breaks ties by select- \\
ing a colour-compatible merge if it exists.
\end{tabular} \\
\(\dagger\) Full-EDSM & \begin{tabular}{l} 
Reference EDSM where an Oracle fully labels the states \\
in the starting APTA.
\end{tabular} \\
\(\dagger\) Col-EDSM & \begin{tabular}{l} 
Reference EDSM where an Oracle fully colours each \\
state in the starting APTA.
\end{tabular} \\
\(\dagger\) FullCol-EDSM & \begin{tabular}{l} 
Reference EDSM where an Oracle fully labels and \\
colours each state in the starting APTA.
\end{tabular}
\end{tabular}
\(\dagger\) Col \(k\)-EDSM \(+\mathbf{W}\) - An Oracle selects the highest EDSM scoring colourEDSM compatible merge for the first \(k\) merge steps, then proceeds with W-EDSM.
\(\dagger\) Col \(k\)-W-EDSM \(+\mathbf{W}\) - An Oracle selects the highest W-EDSM scoring colourEDSM compatible merge for the first \(k\) merge steps, then proceeds with W-EDSM.

\subsection*{7.4 The Abbadingo One Setup}

\subsection*{7.4.1 Creating Target DFAs}

The procedure described in LPP98] for creating an \(n\)-state target DFA is summarised as follows:
1. Create a graph containing \(\frac{5}{4} n\) nodes.
2. Label each node in the graph as either an accepting or rejecting state with equal probability.
3. For each node in the DFA, create one transition per symbol to another random node. In Abbadingo One, the alphabet is binary, and the symbols are the letters \(a\) and \(b\).
4. Randomly select a node from the graph and make it the starting state.
5. Minimise the graph (DFA) using the Moore minimisation method as described in Section 2.4.5.
6. Accept the minimised graph as the target if its depth is exactly \(\left(2 \log _{2} n-2\right)\), otherwise repeat the procedure until this condition is satisfied. The depth of a DFA is defined in Section 2.4.3.

It should be noted that this method will yield a target DFA whose size is close to, but not necessarily equal to, the desired target size \(n\). The depth will, however, always be exactly equal to \(\left(2 \log _{2} n-2\right)\). Lang et al. note that the size variation is practically inconsequential, although any variance in depth would greatly complicate the construction of a training set from our DFA [LPP98]. The complete algorithm for creating 'Abbadingo-style' target DFAs is given in Algorithm 7.1.
```

Algorithm 7.1 Creating an Abbadingo-style target DFA ([LPP98]).

```
Input: The size \(n\) of the desired target DFA, and symbols the alphabet of the DFA.
Output: A minimised DFA \(\mathcal{A}\) containing \(m\) states and a depth of exactly \(\left(2 \log _{2} n-2\right)\).
    \(m\) is close to the desired input size \(n\) (i.e. roughly \(n\) states).
    // Continue trying until the depth property is satisfied.
    while true do
        \(\mathcal{A} \leftarrow \mathrm{A}\) new DFA with \(\frac{5}{4} \times n\) states
        // Create the transitions (without replacement).
        for each state \(s\) in \(\mathcal{A}\) do
            for each symbol \(a \in\) symbols do
                    \(s^{\prime} \leftarrow\) A random state in \(\mathcal{A}\)
                    Create a transition in \(\mathcal{A}\) from state \(s\) to \(s^{\prime}\) for symbol \(a\)
            end for
        end for
        Let the starting state be a random state in \(\mathcal{A}\)
        Minimise \(\mathcal{A}\)
```

        if depthOf(\mathcal{A})=\operatorname{round}(2\times\mp@subsup{\operatorname{log}}{2}{}(n)-2) then // Depth property.
            return \mathcal{A}
        end if
    end while
    ```

\subsection*{7.4.2 Creating Training Sets}

Having created a target DFA of approximately size \(n\), we now proceed to describe the method used to generate a training set of a given density from it:
1. Uniformly, and without replacement, draw a training set \(D \subseteq S\), where \(S\) is the set of all binary strings having length 0 to \(2 \log _{2} n+3\) inclusively.
2. Classify all the strings in \(D\) using the target DFA.
3. The size of the training set \(D\) is determined by using a density parameter \(p=\{1,2,3,4\}\) to linearly interpolate between two values \(L\) and \(U\), these
respectively being the approximate lower and upper bounds on the sample complexity. When \(p=1\), the training set is sparsest, and when \(p=4\), the training set is densest. This gives us \(|D|=L+\frac{p}{4}(U-L)\).
4. The lower bound \(L\) is given by \(L=\log _{2}\left(\frac{2^{n} n^{2 n}}{(n-1)!}\right)\), where \(\frac{2^{n} n^{2 n}}{(n-1)!}\) is the number of possible \(n\)-state DFAs. For completeness sake, we derive this bound thusly:
(a) Every state can be accepting or rejecting \((\times 2)\).
(b) Every transition labelled with an \(a\) can go to any of the \(n\) states \((\times n)\).
(c) Every transition labelled with a \(b\) can go to any of the \(n\) states \((\times n)\).
(d) This gives us \(2 n n\) possibilities for each state.
(e) Repeated for \(n\) states, this gives us \((2 n n)^{n}=2^{n} n^{n} n^{n}=2^{n} n^{2 n}\) possibilities.
(f) However, the order in which we choose states is not important, hence we get \(\frac{2^{n} n^{2 n}}{(n-1)!}\) possibilities.
(g) It is worth noting that for practical considerations, \(L=\log _{2}\left(\frac{2^{n} n^{2 n}}{(n-1)!}\right)\) may be algebraically simplified to \(L=\sum_{k=1}^{n-1} \log _{2}\left(\frac{2 n}{1-\frac{k}{n}}\right)+\log _{2}\left(2 n^{2}\right)\).
5. The designers of Abbadingo use the learning curves for the TrakhtenbrotBarzdin algorithm (whose success depends on the training set being uniformly complete) to obtain the value for the upper bound \(U\).

\section*{Note regarding DFA and training set construction}

While our target DFAs and data sets are constructed according to published specifications, there could still be the possibility of variances due to discrepancies in implementation. As a sanity check, we measure our results against those of Lang's "Evidence Based State Merging with Search " Lan98] who, in turn, measured his against previously published results. In various configurations, we can confirm that our implementation and results are well within the ballpark.

\section*{Note regarding 32-state target problems}
- The Abbadingo One competition does not specify the size bounds or density values for 32 -state problems.
- To obtain this, we select a training set size which would give roughly the same EDSM performance as that for 64 -state problems at density 1 .
- The 'density 1' we get for 32 -state problems is 607 strings which corresponds to circa \(3.7 \%\) of the \(16 n^{2}-1\) strings possible.

\subsection*{7.4.3 Creating Problem Instances}

All the problem instances created for the experiments described in this chapter will have the following characteristics:
- Recall that we may request a target DFA of size \(n\) but get one close to \(n\) instead. In our experiments, all target DFA sizes generated will have exactly the number of states \(n\) requested. The Abbadingo construction depth requirement of ( \(2 \log _{2} n-2\) ) will always be honoured.
- Unless otherwise specified, all training sets will be symmetrically structurally complete with respect to the target. Cases when the training set is not symmetrically structurally complete will be studied separately.
- Unless otherwise specified, the number of positive and negative strings in any training set will not be allowed to differ by more than \(20 \%\). This will avoid pathological cases where a training set would be overwhelmed by strings of a single class.
- All training sets are at Abbadingo density 1.
- Testings sets will consist of 1,800 strings which do not appear in the training set.
- A low-error hypothesis is one which has a misclassification rate of no more than \(1 \%\) over the testing set.

\subsection*{7.5 Baseline Experiments}

\subsection*{7.5.1 Expected APTA Sizes and Merges}

In Table 7.3 we show the expected number of states in an APTA as well as the number of possible merge pairs in it for various target DFA sizes at an Abbadingo density of 1 (the minimum, maximum, and mean is computed over 1024 Abbadingo problem instances). These figures emphasise the large number of state pairs available when merging. Recall that the Abbadingo One competition does not specify the number of training strings at the sparsest density for 32 -state problems. We select a value of 607 strings that would result in approximately the same EDSM generalisation rate for 64 -state problems at density 1 .
\begin{tabular}{crccc}
\hline & \multicolumn{4}{c}{ 1024 Iterations, Density 1 } \\
Setup & & Minimum & Maximum & Mean \\
\hline \hline 32 states & States: & 2,401 & 2,599 & 2,502 \\
\((607\) strings \()\) & Merge Pairs: & \(2,881,200\) & \(3,376,101\) & \(3,129,399\) \\
\hline 64 states & States: & 7,036 & 7,323 & 7,180 \\
\((1,521\) strings \()\) & Merge Pairs: & \(24,749,130\) & \(26,809,503\) & \(25,774,398\) \\
\hline 128 states & States: & 22,257 & 22,838 & 22,596 \\
\((4,382\) strings \()\) & Merge Pairs: & \(247,675,896\) & \(260,775,703\) & \(255,275,186\) \\
\hline 256 states & States: & 64,355 & 65,365 & 64,761 \\
\((11,255\) strings \()\) & Merge Pairs: & \(2,070,750,835\) & \(2,136,258,930\) & \(2,096,979,099\) \\
\hline 512 states & States: & 200,701 & 202,348 & 201,466 \\
\((32,500\) strings \()\) & Merge Pairs: & \(20,140,345,350\) & \(20,472,255,378\) & \(20,294,369,699\) \\
\hline
\end{tabular}

Table 7.3: Expected APTA table sizes and merge pairs.

Table 7.4 shows the proportion of the mean size of an APTA and the labelled states in it with respect to the \(16 n^{2}-1\) possible strings LPP98] for an \(n\)-state target DFA at density 1. The mean APTA sizes are computed over 1024 randomly created Abbadingo problem instances. We observe that:
- As the size of the target DFA increases, the proportion of labelled states (the size of the training set) in the corresponding APTA decreases with respect to the total number of strings possible. For example, for 64 -state target problems, the Abbadingo One competition specifies that 1,521 (2.3\%) strings
out of the 65,535 strings are labelled. On the other hand, for 512 -state target problems, the competition only specifies that \(32,500(0.8 \%)\) strings out of the \(4,194,303\) strings are labelled.
- As the size of the target DFA increases, the proportion of the labelled states in the APTA decreases with respect to the size of the APTA itself. For example, for 64 -state target problems, the Abbadingo One competition specifies that \(1,521(21.2 \%)\) strings/states out of the 7,180 states in an averagesized APTA are labelled. On the other hand, for 512-state target problems, the competition only specifies that \(32,500(16.1 \%)\) strings/states out of the 201,466 states in the APTA are labelled.
\begin{tabular}{|c|c|c|c|c|}
\hline \multirow[b]{2}{*}{Target Size \(n\)} & \multicolumn{3}{|c|}{1024 Iterations, Density 1} & \multirow[b]{2}{*}{Labelled States (in APTA)} \\
\hline & APTA Height
\[
2 \log _{2}(n)+3
\] & Total Strings
\[
16 n^{2}-1
\] & APTA Size (mean) & \\
\hline \multirow[t]{2}{*}{32 states} & 13 & 16,383 & 2,502 & 607 \\
\hline & & & \(15.3 \%\) of total & \[
\begin{gathered}
3.7 \% \text { of total } \\
24.3 \% \text { of APTA }
\end{gathered}
\] \\
\hline \multirow[t]{2}{*}{64 states} & 15 & 65,535 & 7,180 & 1,521 \\
\hline & & & 11\% of total & \[
\begin{aligned}
& 2.3 \% \text { of total } \\
& 21.2 \% \text { of APTA }
\end{aligned}
\] \\
\hline \multirow[t]{2}{*}{128 states} & 17 & 262,143 & 22,596 & 4,382 \\
\hline & & & 8.6\% of total & \[
\begin{aligned}
& 1.7 \% \text { of total } \\
& 19.4 \% \text { of APTA }
\end{aligned}
\] \\
\hline \multirow[t]{2}{*}{256 states} & 19 & 1,048,575 & 64,761 & 11,255 \\
\hline & & & 6.2\% of total & \[
\begin{aligned}
& 1.1 \% \text { of total } \\
& \text { 17.4\% of APTA }
\end{aligned}
\] \\
\hline \multirow[t]{2}{*}{512 states} & 21 & 4,194,303 & 201,466 & 32,500 \\
\hline & & & 4.8\% of total & \[
\begin{gathered}
0.8 \% \text { of total } \\
16.1 \% \text { of APTA }
\end{gathered}
\] \\
\hline
\end{tabular}

Table 7.4: Proportion of the labelled states in an APTA, and size of the APTA with respect to the total number of strings.

\subsection*{7.5.2 Presentation of Results}

The majority of the tables shown in this chapter (such as those in Tables 7.5 and 7.6), as well as those in the following ones will be presented as follows:
\(\left.\begin{array}{ll}\text { Header } & \begin{array}{l}\text { Table headers will show the size of the target DFA we } \\ \text { are searching for, the number of strings in the training }\end{array} \\ \text { set (these would correspond to Abbadingo at density 1), } \\ \text { and the total number of experiments run to obtain the ag- } \\ \text { gregated results. The filename of the SQLite database file } \\ \text { which contains the experiments used is also shown in this } \\ \text { header }\end{array}\right\}\)

\footnotetext{
\({ }^{1}\) The SQLite databases are described in Table 7.1 above.
}

\subsection*{7.5.3 Baseline EDSM and W-EDSM Performance}

In this section, we show the experimental likelihood of finding low-error hypotheses, hypotheses which are close in size to the target (within \(\pm 1\) state), and finding the exact target when using EDSM and W-EDSM as a heuristid \({ }^{2}\). We also show the mean and median sizes of the hypotheses discovered by each heuristic. Using an Oracle-assisted tie breaker, we can empirically establish an upper bound on the performance of EDSM and W-EDSM. In other words, this shows how EDSM behaves if it always makes an optimal tie-breaking choice.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{n32d607e1024.sqlite} \\
\hline \multicolumn{6}{|c|}{32-State Target, 607 Strings, 1024 Experiments} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM & 15.7\% & 19.6\% & 4.3\% & 75 & 90 \\
\hline W-EDSM & 14.1\% & 17.78\% & 3.6\% & 77 & 92 \\
\hline EDSM-TieCC & 25.5\% & 26.4\% & 16.6\% & 71 & 88 \\
\hline \multicolumn{6}{|c|}{n64d1e1024.sqlite} \\
\hline \multicolumn{6}{|c|}{64-State Target, 1,521 Strings, 1024 Experiments} \\
\hline EDSM & 15.2\% & 17.7\% & 0.6\% & 162 & 200 \\
\hline W-EDSM & 13.8\% & 16.5\% & 0.5\% & 170 & 207 \\
\hline EDSM-TieCC & 24\% & 22.3\% & 9.2\% & 154 & 199 \\
\hline \multicolumn{6}{|c|}{n128d1e512.sqlite} \\
\hline \multicolumn{6}{|c|}{128-State Target, 4,382 Strings, 512 Experiments} \\
\hline W-EDSM & 22.7\% & 15.4\% & 0\% & 377 & 511 \\
\hline W-EDSM-TieCC & 27.5\% & 18.2\% & 7.4\% & 359 & 502 \\
\hline
\end{tabular}

Table 7.5: Baseline EDSM and W-EDSM performance.

\subsection*{7.5.4 Analysis of Oracle-Assisted Heuristics}

In Table 7.6, we show the results obtained when constructing merge sequences using the Full-EDSM, Col-EDSM, and FullCol-EDSM Oracle-assisted heuristics on 1024 problem instances of 64 -state target DFAs at density 1. By examining these results, as well as inspecting the merge paths constructed using these heuristics, we can make the following important empirical observations:

\footnotetext{
\({ }^{2}\) For performance reasons, all 128 -state target DFA problem instances will be evaluated using windowed heuristics (as described in [LPP98]).
}
- Col-EDSM always finds the exact target in \(n+m\) merge steps where \(n\) is the size of the target and \(m \geq 1\).
- Full-EDSM can over-generalise (finding a DFA smaller than the target) leading to situations where the target DFA is not found.
- Col-EDSM constrains merges more than Full-EDSM does. When the training set is symmetrically structurally complete, Col-EDSM encapsulates both the correct label of every state in the APTA and also exactly which merges must be merged in which.
- In our experiments, FullCol-EDSM always finds the exact target in \(n+1\) merge steps where \(n\) is the size of the target DFA.
- We observe that the exact target DFA is never found in fewer than \(n+1\) merge steps. We suspect that this is a strong lower bound.
- Low-error hypotheses are always found close to \(n+1\) merge steps, and never further than \(n+\frac{n}{2}\) steps.
- The length of a merge path is a very good proxy for whether the hypothesis will be a low-error one. If the path is too long, we can be almost certain that the hypothesis will have a high error against the testing set (we elaborate on this observation in Section 7.5.11.
\begin{tabular}{lccccc}
\multicolumn{5}{c}{ n64d1e1024.sqlite } \\
& 64-State Target, & 1,521 Strings, \(\mathbf{1 0 2 4}\) & Experiments \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline Full-EDSM & \(100 \%\) & \(98.6 \%\) & \(31.3 \%\) & 64 & 64 \\
Col-EDSM & \(100 \%\) & \(100 \%\) & \(100 \%\) & 64 & 64 \\
FullCol-EDSM & \(100 \%\) & \(100 \%\) & \(100 \%\) & 64 & 64 \\
\hline
\end{tabular}

Table 7.6: Analysis of Oracle-assisted heuristics.

\section*{Ideal Merge Paths}

We call merge paths which find the target DFA in \(n+1\) merge steps, ideal merge paths.

\subsection*{7.5.5 Non-Structurally Complete Training Sets}

When creating a random Abbadingo-style training set, there is a considerable likelihood that it is not symmetrically structurally complete with respect to its corresponding target. In this section, we estimate this likelihood as well as its effect on EDSM. In Table 7.7 we can see that as the size of the target DFA increases, the likelihood of constructing a training set which is not structurally complete increases noticeably. The experiments which follow in Table 7.8 also show that, in such cases, the performance of EDSM is adversarially impacted. Moreover, and as expected, when the training set is not structurally complete, the target DFA can never be found since it is not in the search space to begin with (see Chapter (3). We could find exact targets using the FullCol-EDSM heuristic only when we have been lucky enough that the full labelling caused the training set to become symmetrically structurally complete.
\begin{tabular}{lccc}
\hline Setup & \#Iterations & \#Struct. Incompl. & Observed Likelihood \\
\hline \hline 32 States, 607 Strings & 2048 & 935 of 2048 & \(46 \%\) \\
64 States, 1,521 Strings & 2048 & 1255 of 2048 & \(61 \%\) \\
128 States, 4,382 Strings & 2048 & 1486 of 2048 & \(73 \%\) \\
256 States, 11,255 Strings & 1024 & 908 of 1024 & \(89 \%\) \\
\hline
\end{tabular}

Table 7.7: The likelihood of generating a non-structurally complete training set.
\begin{tabular}{lccccc}
\hline \multicolumn{4}{c}{ 32-State Target, \(\mathbf{6 0 7}\) Strings, Aggregated Over \(\mathbf{2 5 6}\)} & Problem Instances \\
Heuristic & \(\leq 1 \%\) & Error & \(\pm 1\) Target & Exact Target & Mean Size
\end{tabular} Median Size 9

Table 7.8: The effect of non-structurally complete training sets on EDSM.

\subsection*{7.5.6 Getting the First Merges Right}

In Cic02, Cicchello studies the importance of getting the initial sequence of merges correct. Specifically, the author reports that selecting the first few merges
in a merge path correctly, increases the likelihood of identifying a low-error hypothesis. We build on this idea by observing the expected performance improvement when using an Oracle-assisted heuristic that selects the highest EDSM-scoring colour-compatible merge for the first \(k\) merge steps. After these \(k\) merges have been performed, the remainder of the path is extended using W-EDSM. As expected, as \(k\) increases so does the likelihood of discovering low-error hypotheses. Every merge made will establish constraints on the following ones (unlabelled states become labelled) and when these merges are correct, those constraints will allow the heuristic to perform better. Table 7.9 shows how performance improves as the first eight merges are guaranteed to be correct for 32-state target DFA problems. The same improvement may be observed for 64 and 128 -state target DFAs (see Tables A. 10 and A. 14 in Appendix A.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{n32d607e1024.sqlite} \\
\hline \multicolumn{6}{|c|}{32-State Target, 607 Strings, 1024 Experiments} \\
\hline Heuristic & \(\leq 1 \% \mathrm{Err}\) & \(\pm 1\) Tgt & Exact Tgt & Mean Size & Median Size \\
\hline EDSM (baseline) & 15.7\% & 19.6\% & \(4.3 \%\) & 75 & 90 \\
\hline Col01-EDSM + W-EDSM & 20\% & 25.9\% & 4.8\% & 69 & 84 \\
\hline Col02-EDSM + W-EDSM & 28.8\% & 37.1\% & 8.1\% & 60 & 49 \\
\hline Col03-EDSM + W-EDSM & 31.7\% & 41.3\% & 8.5\% & 54 & 38 \\
\hline Col04-EDSM + W-EDSM & 39.1\% & 50.4\% & 9.9\% & 48 & 33 \\
\hline Col05-EDSM + W-EDSM & 43.4\% & 57.9\% & 11.9\% & 43 & 32 \\
\hline Col06-EDSM + W-EDSM & 51.2\% & 64.1\% & 14.8\% & 40 & 32 \\
\hline Col07-EDSM + W-EDSM & 51.4\% & 66.6\% & 15.5\% & 38 & 32 \\
\hline Col08-EDSM + W-EDSM & 54.9\% & 71.3\% & 17.3\% & 36 & 32 \\
\hline
\end{tabular}

Table 7.9: The effect of getting the first merges right (the generalisation rate improves as \(k\) increases).

A significant difference between our method and Cicchello's is that Cicchello used a limited search to identify the correct merge to perform at each level. Such a search could very well have failed. We argue that just to analyse merge behaviour, this risk and computational expense is not needed. In our case, since we are using colour-compatibility, identifying a correct merge is trivial. Of course, colourcompatibility depends on an Oracle which, while being useful in our analysis, is not available in actual algorithms applied to real-world problems.

\subsection*{7.5.7 Reduction Curves and DFA Compression}

\section*{Clarification}

The 'reduction' and 'compression' terminology introduced in this chapter should not be confused with similarly named principles in Minimum Description Length (MDL) as used by Adriaans and Jacobs in AJ06.

In this section, we consider merge paths of length \(n+1\) from an APTA to the exact target of size \(n\) which have been constructed using the FullCol-EDSM heuristic. These correspond to the 'best case' shortest paths consisting of only colour-compatible merges leading to the exact target DFA. We examine the following characteristics:
- The number of states reduced in the hypothesis at every one of the \(n+1\) merge steps.
- The rates at which hypothesis sizes are being 'compressed' starting from the APTA to the final \(n\)-state target.

These experiments allow us to analyse paths in terms of both the number of states reduced, and hypothesis sizes at each step of an ideal path. By examining the results in Figures 7.2, 7.3, and 7.4 we can see that when a heuristic is successfully converging to the target there is an expected state reduction and hypothesis size at each merge step (the exact values may be found in Table B.1 in Appendix B). It follows that at every merge step, we only need to focus on the subset of merges which will reduce the hypothesis size to roughly within the minimum and maximum bounds (and close to the average value). The most important observation is possibly that merges which result in very low reductions are avoided in the initial merges of successful paths. For example, by inspecting Figure 7.3, we can see that merges reducing fewer than 64 states are never selected in the first eight merge steps of an ideal path (i.e. the intersection between 64 on the \(y\)-axis and 8 on the x -axis is below the minimum, red-dotted plot).


Figure 7.2: Number of states reduced in \(n+1\) steps in a FulCol-EDSM path.

True Reduction from APTA to Target in \(\mathrm{n}+1\) Steps
64-State Target, 1,521 Strings, 1024 Experiments, Heuristic=FullCol-EDSM


Figure 7.3: Number of states reduced in \(n+1\) steps in a FulCol-EDSM path (state reduction on \(\log _{2}\) scale).


Figure 7.4: Hypothesis size compression in \(n+1\) steps in a FulCol-EDSM path.

\subsection*{7.5.8 The APTA Reduction Table}

The reduction in states at every merge step along an ideal merge path to the target DFA allows us to determine which low reduction merges to ignore during merge selection. Since there are considerably fewer high reduction merges than there are low reduction ones (there are more states closer to the leaves of the APTA), we can design algorithms which focus their search on this much smaller pool of merges. Unfortunately, determining this reduction at each and every merge step comes at a high computational cost as it requires us to actually perform the merge itself. To deal with this, we propose the APTA reduction table structure which consists of the set of valid merges in an APTA which have reduced its size by at least some number of states and have at least some EDSM score. In the experiments which follow, we observe that:
- While the merges in the APTA reduction table do not correspond exactly to the 'true' high reduction merges we discussed earlier, the table needs to be constructed only once for a problem instance as opposed to determining the actual reductions for every single merge at every step along the merge path.
- The APTA reduction table contains far fewer merges than the set of all possible merges at every merge step.
- The table contains enough good, colour-compatible, merges to get us past the critical first merge steps when constructing a merge sequence.

\section*{Definition 7.1: The True Reduction of a Merge}

Let \(g\) be the size of a DFA, and \(h\) be the size of the DFA obtained by performing some valid merge \(m\). The true reduction of the merge \(m\) is \((g-h)\).

\section*{Definition 7.2: The APTA Reduction Table}

Let \(\mathcal{M}\) be the set of valid merges in an APTA. Let \(R: \mathcal{M} \rightarrow \mathbb{N}\) be a function which gives the true reduction of some merge, and \(E: \mathcal{M} \rightarrow \mathbb{N}\) be a function which gives the EDSM score of some merge. The APTA reduction table for parameters \(\alpha\) and \(\beta\) is the set of merges \(\{m \in \mathcal{M}: R(m) \geq \alpha\) and \(E(m) \geq \beta\}\).

In Table 7.10, we show the expected sizes of the reduction tables for various target size problems, and \(\alpha\) and \(\beta\) configurations. We note the substantial reduction in the number of merges in the table compared to all the possible ones in the APTA. For example, for 128 -state target problems, there are about \(255 \times 10^{6}\) possible merges in the APTA (see Table 7.3), whereas in the reduction table for \(\alpha \geq 160\) and \(\beta \geq 1\) there are only 1,658 which correspond to roughly \(0.001 \%\) of these possible merges (see Table 7.10). The 'Avg. CC Merges' columns show the average number of colour-compatible merges in the table for the corresponding configuration. In the same example, out of the 1,658 merges, the table contains an average of 63 colour-compatible ones. The implication is that there are still a large number of high reduction colour-compatible (good) merges in this vastly reduced search space. This idea is illustrated in Figure 7.5 .

32-State Target, 607 Strings, 256 Iterations
Average merges in APTA: 3,129,399
\begin{tabular}{ccccc}
\(\alpha \geq\) & \multicolumn{2}{c}{\(\beta \geq 0(\mathrm{~min}\). EDSM) } & \multicolumn{2}{c}{\(\beta \geq 1\) (min. EDSM) } \\
(min. reduction) & Avg. Merges & Avg. CC Merges & Avg. Merges & Avg. CC Merges \\
\hline \hline 30 & \(2,313(0.074 \%)\) & 145 & \(770(0.025 \%)\) & 78 \\
25 & \(3,520(0.113 \%)\) & 205 & \(1,031(0.033 \%)\) & 99 \\
20 & \(5,982(0.192 \%)\) & 347 & \(1,498(0.048 \%)\) & 137 \\
\hline
\end{tabular}

64-State Target, 1,521 Strings, 256 Iterations
Average merges in APTA: 25,774,398
\begin{tabular}{ccccc}
\(\alpha \geq\) & \multicolumn{2}{c}{\(\beta \geq 0(\mathrm{~min}\). EDSM) } & \multicolumn{2}{c}{\(\beta \geq 1\) (min. EDSM) } \\
(min. reduction) & Avg. Merges & Avg. CC Merges & Avg. Merges & Avg. CC Merges \\
\hline \hline 60 & \(3,699(0.014 \%)\) & 132 & \(1,291(0.005 \%)\) & 74 \\
55 & \(4,586(0.018 \%)\) & 159 & \(1,496(0.006 \%)\) & 88 \\
50 & \(5,768(0.022 \%)\) & 198 & \(1,752(0.007 \%)\) & 94 \\
\hline
\end{tabular}

128-State Target, 4,382 Strings, 256 Iterations
Average merges in APTA: 255,275,186
\begin{tabular}{ccccc}
\(\alpha \geq\) & \multicolumn{2}{c}{\(\beta \geq 0(\mathrm{~min}\). EDSM) } & \multicolumn{2}{c}{\(\beta \geq 1\) (min. EDSM) } \\
(min. reduction) & Avg. Merges & Avg. CC Merges & Avg. Merges & Avg. CC Merges \\
\hline \hline 200 & \(1,774(0.001 \%)\) & 53 & \(988(0.000 \%)\) & 46 \\
180 & \(2,369(0.001 \%)\) & 70 & \(1,249(0.000 \%)\) & 54 \\
160 & \(3,461(0.001 \%)\) & 90 & \(1,658(0.001 \%)\) & 63 \\
\hline
\end{tabular}

Table 7.10: APTA reduction table sizes for various target sizes, and \(\alpha\) and \(\beta\) configurations.


Figure 7.5: An illustration of the APTA reduction table (not to scale).

\subsection*{7.5.9 Overlap Between Ideal Merge Paths and Merges in the APTA Reduction Table}

In this section we study the number of merges in an ideal path which also exist in the APTA reduction table. Specifically:
- We construct a large number of problem instances.
- For each instance, we construct an ideal merge path leading to the exact target DFA in \(n+1\) merge steps using the FullCol-EDSM heuristic.
- For each instance, we construct the corresponding APTA reduction table for some \(\alpha\) and \(\beta\).
- We count the number of merges in the ideal merge path which also exist in the reduction table. This is illustrated in Figure 7.6.

The significance of this experiment is that if \(k\) merges in an ideal path also exist in the APTA reduction table, then we know that only searching within the merges of the reduction table would be enough to get us through the first \(k\) steps in a merge path which leads to the target (recall that merge ordering is unimportant). Of course, the number of merges in a path which will also be in the table is dependent on the values of the \(\alpha\) and \(\beta\) values we choose. The results we obtain
following this experiment are promising. In Table 7.11 we observe that there is a very high likelihood that the reduction table constructed for some \(\alpha\) and \(\beta\) (which is considerably smaller than the number of all valid merges) contains a large number of colour-compatible merges in the ideal FullCol-EDSM path which leads to the target. For example, more than \(96 \%\) of FullCol-EDSM paths constructed for 32 and 64 -state problems have at least 9 colour-compatible merges in their corresponding APTA reduction table.
\begin{tabular}{ll}
\(\mathbf{3 2}\) States, \(\mathbf{6 0 7}\) Strings, & \(\mathbf{6 4}\) States, \(\mathbf{1 , 5 2 1}\) Strings, \\
\(\mathbf{1 0 2 4}\) Iterations, \(\alpha \geq 25\) and \(\beta \geq 1\) & \(\mathbf{1 0 2 4}\) Iterations, \(\alpha \geq 60\) and \(\beta \geq 1\) \\
\hline \hline Min CC Merges in Table: 5 & Min CC Merges in Table: 3 \\
Max CC Merges in Table: 24 & Max CC Merges in Table: 26 \\
Avg CC Merges in Table: 16 & Avg CC Merges in Table: 16 \\
\hline\(\geq 0\) CC merges \(1024 / 1024(100.00 \%)\) & \(\geq 0\) CC merges 1024/1024 \((100.00 \%)\) \\
\(\ldots\) & \(\ldots\) \\
\(\geq 7\) CC merges \(1016 / 1024(99.22 \%)\) & \(\geq 7\) CC merges \(1012 / 1024(98.83 \%)\) \\
\(\geq 8\) CC merges \(1014 / 1024(99.02 \%)\) & \(\geq 8\) CC merges \(996 / 1024(97.27 \%)\) \\
\(\geq 9\) CC merges \(\mathbf{1 0 0 0 / 1 0 2 4 ( \mathbf { 9 7 . 6 6 \% } )}\) & \(\geq 9\) CC merges 986/1024 (96.29\%) \\
\(\geq 10\) CC merges \(976 / 1024(95.31 \%)\) & \(\geq 10\) CC merges \(967 / 1024(94.43 \%)\) \\
\(\geq 11\) CC merges \(944 / 1024(92.19 \%)\) & \(\geq 11\) CC merges \(946 / 1024(92.38 \%)\) \\
\(\geq 12\) CC merges \(910 / 1024(88.87 \%)\) & \(\geq 12\) CC merges \(915 / 1024(89.36 \%)\) \\
\(\ldots\) & \(\ldots\) \\
\hline
\end{tabular}

Table 7.11: Number of FullCol-EDSM colour-compatible merges which also exist in the APTA reduction table.


Figure 7.6: Measuring the overlap of merges in an ideal merge path and its corresponding APTA reduction table.

\subsection*{7.5.10 Colour-Compatible Merge Positions}

We will refer to the highest scoring EDSM merges (ties) in an APTA reduction table as rank 1 merges, the next highest scoring merges rank 2 , and so on. When EDSM makes a wrong choice it is because it has selected a merge which is not colour-compatible. The problem is that if a colour-compatible merge exists in the first rank, we do not know where it is, and breaking ties randomly may cause us to select the wrong one. Moreover, a colour-compatible merge may not exist within the first EDSM rank at all. To determine where a colour-compatible merge is in the reduction table, at every merge step we sort our reduction table by EDSM score (descending) and study the position of the first colour-compatible merge in the table. To study this behaviour, we create 1024 Abbadingo-style problem instances for 64 -state target DFA problems at density 1 (1,521 strings). For each of these instances, we:
1. Construct an ideal merge path using FullCol-EDSM,
2. Construct several APTA reduction tables for \(\alpha \geq\{60,50,40,30,20\}\) and \(\beta \geq\{0,1\}\),
3. Sort the table by EDSM score descending, and
4. Analyse the position of a colour-compatible merge in the table (for each of the \(\alpha\) and \(\beta\) configurations we created).

In the configuration \(\alpha \geq 60\) and \(\beta \geq 1\) we get:
- At merge step 1 (the very first merge in the APTA): a colour-compatible merge can be found within the first 6 merges of the table (sorted by EDSM) in \(87 \%\) of the 1024 experiments.
- At merge step 2: a colour-compatible merge can be found within the first 6 merges of the table in \(84 \%\) of the 1024 experiments.
- At merge step 3: a colour-compatible merge can be found within the first 6 merges of the table in \(86 \%\) of the 1024 experiments.
- At merge step 4: a colour-compatible merge can be found within the first 6 merges of the table in \(88 \%\) of the 1024 experiments.

In this experimental setup (64-state target DFAs), we also see that using \(\beta \geq 0\) gives slightly better colour-compatible positions (higher in the table) than when \(\beta \geq 1\). However, the table is much bigger. Additionally, using lower values of \(\alpha\) (more merges, bigger tables) does not increase the likelihood that a colourcompatible merge appears at a higher position. This is true when analysing positions at progressively deeper merge steps. These observations allow us to estimate where we can expect colour-compatible merges to appear in the reduction table for various problem setups (and within which EDSM rank). In other words, we can determine \(\alpha\) and \(\beta\) configurations which will result in a very high likelihood that colour-compatible merges may be found within the first \(m\) merges in the table.

\section*{Note}

This value \(m\) is used to guide our choice of branching factor when using the Delta Graph method which we describe later on in Section 7.6.2.

\subsection*{7.5.11 Merge Path Lengths}

Our experiments show that when EDSM manages to find low-error hypotheses, it does so in close to \(n+1\) merge steps where \(n\) is the size of the target DFA we are looking for \({ }^{3}\). Conversely, when merge paths grow too long, we can be exceedingly confident that they will lead to a high-error hypothesis. The following tables show the minimum, maximum, and average path lengths (in merges) when the EDSM \({ }^{4}\) heuristic manages to find hypotheses with \(\leq 1 \%\) error (low error), and \(\geq 5 \%\) or \(\geq 10 \%\) error (high error). This relationship can also be seen in Figure 7.7 where we plot the path length against error for the EDSM heuristic on 64 -state target problems.
\begin{tabular}{cccc}
\hline \multicolumn{4}{c}{ n32d607e1024.sqlite } \\
32-State Target, 607 Strings, 1024 & Experiments, Heuristic=EDSM \\
Error & Min. Path Length & Max Path Length & Avg. Path Length \\
\hline \hline Low \(\leq 1 \%\) & 31 & 38 & 33 \\
High \(\geq 5 \%\) & 43 & 107 & 90 \\
High \(\geq 10 \%\) & 52 & 107 & 92 \\
\hline
\end{tabular}
\begin{tabular}{cccc}
\hline \multicolumn{4}{c}{ n64d1e1024.sqlite } \\
64-State Target, 1,521 & Strings, & 1024 & Experiments, Heuristic=EDSM \\
Error & Min. Path Length & Max Path Length & Avg. Path Length \\
\hline \hline Low \(\leq 1 \%\) & 62 & 69 & 65 \\
High \(\geq 5 \%\) & 89 & 223 & 194 \\
High \(\geq 10 \%\) & 112 & 223 & 200 \\
\hline
\end{tabular}
\(\qquad\)
n128d1e512.sqlite
128-State Target, 4,382 Strings, 512 Experiments, Heuristic=W-EDSM
\begin{tabular}{cccc} 
Error & Min. Path Length & Max Path Length & Avg. Path Length \\
\hline \hline Low \(\leq 1 \%\) & 124 & 134 & 129 \\
High \(\geq 5 \%\) & 192 & 543 & 477 \\
High \(\geq 10 \%\) & 278 & 543 & 501 \\
\hline
\end{tabular}

Table 7.12: Hypothesis error increases with path length.

\footnotetext{
\({ }^{3}\) Also see Section 7.5.4.
\({ }^{4}\) We use W-EDSM for 128 -state target problems.
}


Figure 7.7: Merge path length against hypothesis error for EDSM on 64-state targets at density 1 (as path length grows, so does the error). Data summarised from n64d1e1024.sqlite.

\subsection*{7.5.12 Summary of Results}

In this section, we will summarise the results we obtained in the previous sections and discuss both their implications and how they can be exploited:
- In our experiments, we have observed that low-error hypotheses are aways found in close to \(n+1\) merge steps, and that whenever a merge sequence gets too long, the likelihood of finding such a low-error hypothesis decreases considerably. The implication is that candidate extensions which are getting too long can be aborted early in order to improve the online performance of non-monotonic search strategies.
- We have also observed that when creating random problem instances, there is a considerable likelihood that the training set is not structurally complete with respect to the target DFA. This allows us to study how algorithms behave in this specific adversarial condition.
- In Section 7.5.6, we have studied the importance of getting the first merges in an extension right. The implication is that designing methods which focus on these initial merges will have an overwhelmingly positive impact in terms of either finding the exact target DFA or a low-error hypothesis.
- We also studied the rate at which states are being reduced at each merge step in a sequence which leads to the target DFA. This allowed us to determine that the initial merges in a sequence are always high-reduction merges. Furthermore, these results directly contributed to the APTA reduction table structure which is used in both the Delta Graph and genetic algorithm approached which we will discuss in next sections.
- After defining the APTA reduction table, we have performed experiments to determine its size, its composition, and the usefulness of the merges it contains. This allowed us to ascertain that searching for short initial sequences of merges within this subspace of merges is a promising approach.
- We also examined the positions of colour-compatible merges in the APTA reduction table when its merges are ordered by EDSM score. This allowed us to determine the likelihood that a colour-compatible merge is not among the highest-scoring rank of merges and allows us to fine-tune the hyperparameters which will be needed in the Delta Graph.

\subsection*{7.6 Methodology}

\subsection*{7.6.1 An Ensemble of Heuristics}

The inspiration for this method comes from Wolpert and Macready's No Free Lunch Theorems for Search and Optimisation [WM96, WM97] supported by empirical evidence we have gathered following several experiments. The No Free Lunch (NFL) argument may be summarised as follows:
- A search algorithm attempts to find good solutions by evaluating candidates in a search space.
- A specific search algorithm may perform well on a certain instance but poorly on another.
- In general, the performance of two search algorithms will be identical when 'averaged out' over all possible problem instances.
- In other words, certain algorithms will perform better than others depending on the characteristics of the problems they are confronted with.

Our DFA learning task involves identifying the smallest DFA which is consistent with the training data. As such, it would seem reasonable to prefer merges that reduce the size of the hypothesis most at each merge step. Consider a simple heuristic, which we call Reduction (not to be confused with reduction or compression in MDL [AJ06]), that greedily selects the next merge in a hypothesis based on how many states are reduced after that merge is performed. A greater reduction in states gives a higher score and ties are broken randomly. Over 1024 problem instances of 32 -state target DFAs and training sets at density 1 ( 607 strings), we obtain the result: \({ }^{[5]}\) shown in Table 7.13 ,
\begin{tabular}{lccccc}
\hline \multicolumn{5}{c}{ n32d607e1024.sqlite } & \\
32-State Target, & 607 Strings, & 1024 & Experiments & \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM & \(15.7 \%\) & \(19.6 \%\) & \(4.3 \%\) & 75 & 90 \\
Reduction & \(6.1 \%\) & \(7.7 \%\) & \(1.8 \%\) & 88 & 97 \\
EDSM or Reduction & \(18.7 \%\) & \(22.8 \%\) & \(5.6 \%\) & 73 & 89 \\
\hline
\end{tabular}

Table 7.13: The performance of EDSM, Reduction, and EDSM or Reduction.

Concretely, EDSM finds hypotheses with \(\leq 1 \%\) error in 161 out of the 1024 instances ( \(15.7 \%\) ), reduction does so in 62 instances (6.1\%), and an ensemble of either EDSM or reduction does so in 191 instances (18.7\%). These results arranged in a Venn diagram are shown in Figure 7.8 next. It is evident that, while reduction performs worse than EDSM, the symmetric difference is not empty (i.e. reduction manages to find low-error hypotheses in cases when EDSM cannot).

\footnotetext{
\({ }^{5}\) Presented as described in Section 7.5.2
}


Figure 7.8: Venn diagram showing the performance of EDSM and reduction.

We also study the effects of how the structure of the target DFAs and the subsequent construction of training sets affects the performance of EDSM. Consider two sets of 256 problem instances each. In one set, none of the target DFAs are allowed to have loops (a transition from a state to itself), and in the other all target DFAs have at least one loop. We now measure the performance of EDSM on these two setups and tabulate the results in Table 7.14 below.
\begin{tabular}{|c|c|c|c|c|c|}
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM-No-Loop & 10.55\% & 12.5\% & 1.17\% & 81 & 92 \\
\hline EDSM-All-Loop & 16.41\% & 24.22\% & 4.3\% & 71 & 88 \\
\hline
\end{tabular}

Table 7.14: Comparing the performance of EDSM when the target DFA has no loop transitions against when the target DFA has at least one loop transition.

These results indicate that the absence of loops is highly adversarial to EDSM, and it is beneficial to consider alternative heuristics which are able to perform better in such cases. Concretely, we observe that the performance on targets having no loop transitions drops to \(\approx 64 \%\) compared to EDSM on targets having at least one loop transition. This observation coupled with the results showing that reduction can indeed find low-error hypotheses when EDSM cannot, supports the idea of combining multiple heuristics together.

\section*{The Ensemble of Heuristics}

\section*{Primitives}

We will use Edsm to denote the EDSM score of a merge, C-EdSm denotes the cumulative EDSM score of a merge starting from the APTA, RED denotes the number of states reduced by a merge, and Size denotes the number of states in (size of) the hypothesis after a merge.

Suppose we first order the merges in an APTA \({ }^{6}\) by EDSM score descending, and then order them by the score from different heuristic \(\mathrm{H}=(\mathrm{C}-\mathrm{EdSm}+1)+\) \(\log _{e}(\operatorname{RED}+1)\). In Table 7.15 below, we list the top scoring 10 merges in the APTA for each of these two orderings, and make the following observations:
- The highest EDSM score is 6 , and there is only one merge ( \(q_{36}, q_{81}\) ) having this score. Unfortunately, this merge is not colour-compatible, so using EDSM, the first merge in the APTA will be wrong.
- On the other hand, we can see that using H as a heuristic orders the merges differently. Now, the highest H-scoring merge \(\left(q_{0}, q_{23}\right)\) is colour-compatible.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{3}{|c|}{Merges ordered by EDSM} & \multicolumn{3}{|c|}{Merges ordered by H} \\
\hline State Pair & EDSM Score & Col. Compat. & State Pair & H Score & Col. Compat. \\
\hline \(\left(q_{36}, q_{81}\right)\) & 6 & & \(\left(q_{0}, q_{23}\right)\) & 11.71 & Yes \\
\hline \(\left(q_{0}, q_{23}\right)\) & 5 & Yes & \(\left(q_{36}, q_{81}\right)\) & 11.63 & \\
\hline \(\left(q_{80}, q_{135}\right)\) & 5 & & \(\left(q_{80}, q_{135}\right)\) & 11.48 & \\
\hline \(\left(q_{45}, q_{135}\right)\) & 5 & Yes & \(\left(q_{45}, q_{135}\right)\) & 10.86 & Yes \\
\hline \(\left(q_{64}, q_{146}\right)\) & 4 & & \(\left(q_{0}, q_{81}\right)\) & 10.62 & \\
\hline \(\left(q_{23}, q_{81}\right)\) & 4 & & \(\left(q_{2}, q_{22}\right)\) & 10.53 & \\
\hline \(\left(q_{108}, q_{109}\right)\) & 4 & & \(\left(q_{23}, q_{81}\right)\) & 9.94 & \\
\hline \(\left(q_{0}, q_{81}\right)\) & 4 & & \(\left(q_{108}, q_{109}\right)\) & 9.78 & \\
\hline \(\left(q_{2}, q_{22}\right)\) & 4 & & \(\left(q_{64}, q_{146}\right)\) & 9.26 & \\
\hline \(\left(q_{3}, q_{108}\right)\) & 3 & & \(\left(q_{2}, q_{108}\right)\) & 9.20 & \\
\hline
\end{tabular}

Table 7.15: The top scoring merges in an APTA ordered by EDSM and ordered by the heuristic H. Dotted lines delineate EDSM ranks.

\footnotetext{
\({ }^{6}\) Constructed from a density 1 training set drawn from a 32 -state target DFA.
}

In a state merging algorithm, the score obtained by a heuristic is used to select merges (by ordering them) to build a merge path starting from the APTA to a final hypothesis. A different heuristic will score merges differently and would, potentially, build a merge path to another final hypothesis. In other words, different heuristics correspond to different inductive biases of state merging algorithms. While experimenting with various combinations of Edsm, C-Edsm, Red, and Size scores, we observed that several of them were able to identify the exact target DFA or low-error hypotheses when EDSM was not able to (these results are presented and discussed in the next chapter). The following list describes the various combinations of EDSM with reduction which we will use in our ensemble of heuristics.

Our search algorithm, based on the ensemble of heuristics, will denoted by \(\mathrm{H} 1 / \mathrm{H} 2 / \mathrm{H} 3 / \ldots\), where the hypothesis returned would be the one having the fewest number of states. In the case when there are multiple hypotheses having the same minimum number of states, a random one among them is returned.
\[
\begin{aligned}
& \mathrm{A} 1:(\mathrm{EdSm}+1) \times \mathrm{Log}_{2}(\mathrm{Size}) \\
& \text { A2 : }(\operatorname{EdSm}+1) \times \log _{10}(\operatorname{Size}) \\
& \mathrm{A} 3:(\operatorname{EdSm}+1) \times \log _{e}(\operatorname{Size}) \\
& \mathrm{C} 1:(\operatorname{EdSm}+1) \times \log _{2}(\operatorname{RED}+1) \\
& \mathrm{C} 2:(\operatorname{EdSm}+1) \times \log _{10}(\operatorname{Red}+1) \\
& \mathrm{D} 1:(\mathrm{EDSm}+1)+\log _{2}(\operatorname{Red}+1) \\
& \mathrm{C} 3:(\operatorname{EDSM}+1) \times \log _{e}(\operatorname{RED}+1) \\
& \text { D2: }(\operatorname{EDSM}+1)+\log _{10}(\operatorname{RED}+1) \\
& \mathrm{D} 3:(\operatorname{EDSM}+1)+\log _{e}(\operatorname{RED}+1) \\
& \text { E1: }(\mathrm{C}-\mathrm{EdSm}+1) \times \mathrm{Log}_{2}(\text { Size }) \\
& \text { F1: }(\mathrm{C}-\mathrm{EdSm}+1)+\mathrm{Log}_{2}(\mathrm{Size}) \\
& \text { E2 : }(\mathrm{C}-\mathrm{EdSm}+1) \times \log _{10}(\text { Size }) \\
& \mathrm{E} 3:(\mathrm{C}-\mathrm{EdSm}+1) \times \mathrm{Log}_{e}(\mathrm{Size}) \\
& \text { G1: }(\mathrm{C}-\operatorname{EdSm}+1) \times \log _{2}(\operatorname{RED}+1) \\
& \mathrm{H} 1:(\mathrm{C}-\mathrm{EDSM}+1)+\mathrm{Log}_{2}(\mathrm{RED}+1) \\
& \mathrm{G} 2:(\mathrm{C}-\operatorname{EdSm}+1) \times \log _{10}(\operatorname{Red}+1) \\
& \mathrm{H} 2:(\mathrm{C}-\operatorname{EdSm}+1)+\log _{10}(\operatorname{Red}+1) \\
& \text { G3: }(\mathrm{C}-\operatorname{EdSm}+1) \times \operatorname{LOG}_{e}(\operatorname{Red}+1) \\
& \mathrm{H} 3:(\mathrm{C}-\mathrm{EdSm}+1)+\log _{e}(\mathrm{Red}+1)
\end{aligned}
\]
\[
\begin{aligned}
& \text { I1 : }(\operatorname{RED}+1) \times \log _{2}(\operatorname{EdSm}+1) \quad \mathrm{J} 1:(\operatorname{RED}+1)+\log _{2}(\operatorname{EdSm}+1) \\
& \text { I2 : }(\operatorname{Red}+1) \times \log _{10}(\operatorname{EdSm}+1) \quad \mathrm{J} 2:(\operatorname{Red}+1)+\log _{10}(\operatorname{EdSm}+1) \\
& \mathrm{I} 3:(\operatorname{Red}+1) \times \log _{e}(\operatorname{EdSm}+1) \quad \mathrm{J} 3:(\operatorname{Red}+1)+\log _{e}(\operatorname{EdSm}+1) \\
& \mathrm{K} 1:(\operatorname{RED}+1) \times \log _{2}(\mathrm{C}-\operatorname{EdSm}+1) \\
& \mathrm{L} 1:(\mathrm{RED}+1)+\mathrm{Log}_{2}(\mathrm{C}-\mathrm{EdSm}+1) \\
& \mathrm{K} 2:(\operatorname{RED}+1) \times \log _{10}(\mathrm{C}-\operatorname{EdSm}+1) \\
& \mathrm{L} 2:(\operatorname{Red}+1)+\log _{10}(\mathrm{C}-\operatorname{EdSm}+1) \\
& \mathrm{K} 3:(\operatorname{RED}+1) \times \operatorname{LoG}_{e}(\mathrm{C}-\operatorname{EdSm}+1) \\
& \mathrm{L} 3:(\operatorname{RED}+1)+\operatorname{LoG}_{e}(\mathrm{C}-\operatorname{EdSm}+1)
\end{aligned}
\]

\section*{Teams and Ensembles of Automata}

We note that our ensemble of heuristics should not be confused with either García et al.'s teams of automata method GVdPLR10] or the ensemble of automata method used by Heule and Verwer in dfasat [Die00, HV13]. Both of these methods rely on the approach of randomising the merge selection bias of an algorithm to infer a team or ensemble of automata. Once a team of automata have been learnt for a given problem instance, strings in a test set are classified by either taking a fair vote, a weighted vote, or a using the smallest automaton in the team found. In the fair vote case, a label is assigned to a test string according to the one assigned by the majority of the votes in the team. In the weighted vote case, the vote is weighted inverse to the size of the DFA found.

In contrast to the aforementioned approaches, the heuristics in our ensemble are predetermined for all problem instances and there is no random component in the selection of merges (with the exception of tie-breaking).

\subsection*{7.6.2 The Delta Graph}

The Delta Graph method involves constructing and searching in a highly condensed (and much shallower) version of the search space of merges. Experimentally, this subspace of merges can be shown to contain a sufficient number of colour-compatible merge sequences that constrain hypotheses well enough to allow heuristics such as EDSM to perform better. The motivation for this method follows from the results we have obtained earlier in Section 7.5 of this chapter. Specifically:
- Short initial sequences of colour-compatible merges constrain partial hypotheses well enough to make a huge difference in the likelihood of discovering low-error hypotheses.
- In greedy, monotonic heuristics such as EDSM, ties are usually broken at random. It then follows that a colour-compatible merge may be (catastrophically) missed during this tie break. Moreover, the set of highest scoring, tied merges may not even contain a colour-compatible merge at all making even the luckiest tie break futile.
- Exploring the entire merge space using backtracking is not feasible. However, the much smaller pool of merges in the APTA reduction table has been experimentally shown to contain a sufficient number of colour-compatible merges to extract a good initial sequence from (see Section 7.5.9).

\section*{Delta Graph Construction}

Our strategy will involve searching for an initial sequence of merges in a graph (a subspace of quotient DFAs) which is constructed as follows:
- Every node in the graph is a partial hypothesis (quotient DFA) where the root node is the APTA.
- Nodes are expanded by considering only merges in an APTA reduction table constructed for some \(\alpha\) and \(\beta\). In other words, the edges in the graph are merges selected from the APTA reduction table which lead to other partial hypotheses.
- To keep the size of the graph in check:
- Graphs will be constrained to have a maximum depth \(\delta\).
- Not all valid merges in the APTA reduction table at a node are expanded. The pool of valid merges at a node are sorted descending by some score (such as EDSM) and only the top \(b\) merges are executed at the node (also see Figure 7.10). The parameter \(b\) is an integer greater than 0 and will be called the branching factor of the graph.
- The branching factor \(b\) is only honoured up to some depth limit \(f: 1 \leq\) \(f \leq \delta\). If a node is deeper than \(f\) only the highest scoring merges (ties) are expanded. We call the parameter \(f\) the branching limit.
- Nodes/hypotheses are reused to save computation and keep the graph as small as possible (if we do not reuse nodes, we would construct a tree instead of a graph).

The parameters \(\alpha, \beta, \delta, b\), and \(f\) are summarised next, and their relationship is illustrated in Figure 7.9. An illustration of node expansion and the behaviour of the branching factor is show in Figure 7.10. Figure 7.11 shows an annotated partial Delta Graph constructed for a small example where we can see how a path in the graph from an APTA to a colour-compatible leaf represents a sequence of good colour-compatible merges. We can also observe that it is possible (and for most setups, quite likely) for the graph to contain multiple colour-compatible leaf hypotheses.


Figure 7.9: The relationship between the \(\delta, b\), and \(f\) parameters of a Delta Graph.
\(\alpha\) and \(\beta\)
When constructing the Delta Graph, only merges in the reduction table for the APTA associated with the problem instance are considered. \(\alpha\) is the minimum state reduction that a merge must make to be considered for inclusion in the table. \(\beta\) is the minimum EDSM score for the merge to be allowed into the table.

Graph depth \(\delta\)
The maximum depth in merges of the Delta Graph which will be constructed.

Branching factor \(b\) The number of merges to expand at each node. At the current node, the top scoring \(b\) merges in the APTA reduction table are expanded to lead into the next nodes in the graph.

Branching limit \(f \quad\) This controls up to how many merge levels in the graph the branching factor \(b\) will be considered (i.e. the first \(f\) merge levels in the graph will have a branching factor \(b\), and partial hypotheses deeper than \(f\) will be expanded using all the top-scoring ties at that node/hypothesis).


Figure 7.10: Illustration of how nodes are expanded in the Delta Graph.

\section*{Hypothesis Construction}

Obtaining a hypothesis from a Delta Graph involves extending every leaf in the graph (which represents an initial sequence of merges) with W-EDSM and remembering the one which leads to, either the smallest hypothesis, or the one closest in size to the target 7 . If a leaf is extended to a hypothesis which is exactly equal in size to the target, that extension (sequence of merges) is immediately returned and no further leaves are considered. Since the experiments in Section 7.5.11show that longer paths are likely to lead to high-error hypotheses, we abort paths which are

\footnotetext{
\({ }^{7}\) Depending on whether we know the size of the target or not.
}
longer than \(n+\frac{n}{2}\) merge steps. This represents a pessimistic bound after which we can confidently expect a merge sequence to perform badly. Should all the leaves be aborted, we fall back to W-EDSM for that problem instance.


Figure 7.11: A partial Delta Graph constructed up to depth 6. Blue edges represent colour-compatible merges, and blue nodes represent colour-compatible hypotheses. Some nodes and edges have been omitted for clarity.

\section*{Delta Graph Sizes and Analysis}

Since the computational cost associated with the Delta Graph method is highly dependent on the size of the graph and the number of leaves it contains, we analyse the characteristics of such graphs constructed for various problem instances and graph configurations. Furthermore, recall that a colour-compatible leaf in a graph represents a sequence of good colour-compatible merges starting from the APTA. In Table 7.16, we show the mean values for graphs sizes, number of leaves, and number of colour-compatible leaves. We also show the experimental likelihood of a graph having at least one colour-compatible leaf. The results we obtain are very promising. For the parameter configurations shown in the tables below, we not only have a good chance of discovering a good colour-compatible sequence (the 'any leaf is CC' column) but we also, on average, discover many colour-compatible
sequences of length \(\delta\) (the 'mean CC leaves') column. Moreover, the number of leaves which need expansion is feasible on modest consumer-grade hardware.

Delta Graphs for 32-State Targets, 607 Strings, 256 Iterations
\begin{tabular}{lcccc} 
Delta Graph & \begin{tabular}{c} 
Mean \\
Size
\end{tabular} & \begin{tabular}{c} 
Mean \\
Leaves
\end{tabular} & \begin{tabular}{c} 
Mean \\
CC Leaves
\end{tabular} & \begin{tabular}{c} 
Any Leaf \\
is CC?
\end{tabular} \\
\hline \hline\(\delta=3, b=6, f=3, \alpha \geq 25, \beta \geq 1\) & 193 & 155 & 17 & \(82 \%\) \\
\(\delta=3, b=12, f=3, \alpha \geq 25, \beta \geq 1\) & 1,232 & 1,103 & 84 & \(93 \%\) \\
\(\delta=4, b=10, f=3, \alpha \geq 25, \beta \geq 1\) & 1,654 & 905 & 30 & \(87 \%\) \\
\hline
\end{tabular}
\begin{tabular}{lcccc}
\hline \multicolumn{5}{c}{ Delta Graphs for 64-State Targets, \(\mathbf{1 , 5 2 1}\) Strings, 256 Iterations } \\
Delta Graph & Mean & Mean & Mean & Any Leaf \\
Construction Parameters & Size & Leaves & CC Leaves & is CC? \\
\hline \hline\(\delta=3, b=8, f=3, \alpha \geq 60, \beta \geq 1\) & 400 & 338 & 29 & \(81 \%\) \\
\(\delta=3, b=10, f=3, \alpha \geq 60, \beta \geq 1\) & 725 & 634 & 43 & \(82 \%\) \\
\(\delta=3, b=12, f=3, \alpha \geq 60, \beta \geq 1\) & 1200 & 1072 & 58 & \(85 \%\) \\
\(\delta=4, b=6, f=3, \alpha \geq 60, \beta \geq 1\) & 388 & 205 & 9 & \(67 \%\) \\
\(\delta=4, b=8, f=3, \alpha \geq 60, \beta \geq 1\) & 867 & 469 & 15 & \(73 \%\) \\
\(\delta=4, b=10, f=3, \alpha \geq 60, \beta \geq 1\) & 1583 & 869 & 24 & \(77 \%\) \\
\hline
\end{tabular}

Delta Graphs for 128-State Targets, 4,382 Strings, 256 Iterations
\begin{tabular}{lcccc} 
Delta Graph & \begin{tabular}{c} 
Mean \\
Construction Parameters
\end{tabular} & \begin{tabular}{c} 
Mean \\
Lize
\end{tabular} & \begin{tabular}{c} 
Mean \\
Le Leaves
\end{tabular} & \begin{tabular}{c} 
Any Leaf \\
is CC?
\end{tabular} \\
\hline \hline\(\delta=3, b=12, f=3, \alpha \geq 180, \beta \geq 1\) & 1029 & 910 & 66 & \(90 \%\) \\
\(\delta=4, b=10, f=3, \alpha \geq 180, \beta \geq 1\) & 1309 & 677 & 30 & \(79 \%\) \\
\(\delta=5, b=10, f=3, \alpha \geq 180, \beta \geq 1\) & 2226 & 961 & 22 & \(78 \%\) \\
\hline
\end{tabular}

Table 7.16: Delta graph sizes for 32,64 , and 128 -state target DFA problems for various parameter configurations.

\subsection*{7.6.3 Evolving Initial Merge Sequences}

Previous attempts at using evolutionary techniques for regular inference such as those found in Dup94, LR05, and Wie17 are based on either directly evolving a DFA, or partitioning the states of an APTA. In this section, we propose a genetic algorithm (GA) which, instead, evolves an initial sequence of high quality merges, and see that this method reduces the problem to subset selection which is a well
explored area YH98, Wol15. The argument supporting this method is similar to that for the Delta Graph method we described earlier. Getting the initial sequence of merges is crucial for obtaining high quality hypotheses, and an analysis of the contents of the APTA reduction table with respect to the merges in a FullColEDSM path shows that a properly constructed reduction table contains a large number of merges in common with the ideal FullCol-EDSM path for the instance.

\section*{Chromosome Representation}

A chromosome is a vector of state pairs of length \(k\) where every state pair represents the states to be merged. Since merge order is irrelevant, the design of our crossover and mutation operators is straightforward and efficient. The \(k\) merges in the chromosome represent a subset of merges in the reduction table. Specifically, if the APTA reduction table contains \(p\) merges where \(p \gg k\), our task is to select \(k\) merges out of \(p\) such that those \(k\) merges are an initial sequence which leads to a low-error hypothesis. Note that executing some merge \(\left(q_{i}, q_{j}\right)\) in the chromosome may make another merge \(\left(q_{k}, q_{l}\right)\) an identity merge (see general properties of merging in Chapter 6). This would happen when the partition \(\pi\) obtained after merging \(\left(q_{i}, q_{j}\right)\) would cause \(B\left(q_{k}, \pi\right)=B\left(q_{l}, \pi\right)\). Furthermore, if one of the merges in a chromosome blocks another merge, then the entire merge sequence represented by the chromosome is invalid (see Definition 6.1). Population setup, crossover, and mutation operators are designed such that a chromosome would never contain either identity merges or blocking merges.

The initial population then consists of chromosomes each filled with a random selection of \(k\) valid, non-blocking, and non-identity merges selected from the APTA reduction table.

\section*{Fitness Function}

Starting from the APTA, all the \(k\) merges in a chromosome are executed resulting in a partial hypothesis which is then completed/extended using W-EDSM to reach a final hypothesis. The score of the chromosome is:

This means that a hypothesis with a fitness of 0 is equal to the size of the target, and the genetic algorithm attempts to minimise this score.


Figure 7.12: A chromosome consists of \(k\) merges selected from the APTA reduction table. The fitness is evaluated by extending the chromosome using W-EDSM.

\section*{Selection}

Deterministic tournament selection is used to select parents for crossover BT97. If \(t\) is the size of a tournament, \(t\) chromosomes are selected randomly from the population and the fittest is returned. In the interest of diversity, if identical parents are selected for crossover, the selection procedure is attempted again (up to a limit).

\section*{Crossover and Mutation}

Two parent chromosomes are selected using tournament selection. The \(k+k\) merges from both parents are pooled into a set (eliminating duplicates) and an offspring is constructed by randomly selecting \(k\) non-identity and non-blocking merges from the pool. Mutation simply involves deleting a random merge from a chromosome and replacing it with a different non-identity and non-blocking merge from the APTA reduction table. Crossover is illustrated in Figure 7.13 below.
```

Parent 1:
Parent 2: }\begin{array}{lllllll|l|l|l|l|l|ll}{\textrm{m}1}\&{\textrm{m}2}\&{\textrm{m}3}\&{\textrm{m}4}\&{\textrm{m}5}\&{\textrm{m}6}\&{\textrm{m}7}\&{\textrm{m}8}<br>{\hline}

```
(1) \(\sqrt{ }\)

Pool:

\(\qquad\)

Figure 7.13: Illustration of the crossover operation.

\section*{Stopping Criterion}

The GA terminates either when a chromosome has been extended to a hypothesis whose size is equal to the size of the target DFA (the fitness of the chromosome is 0 ), or when a maximum number of generations have elapsed. In this latter case, the best (smallest) hypothesis is returned as the final hypothesis.

\section*{Population Make Up}

Populations are assembled for the next generation according to a crossover rate \(c \%\), a mutation rate \(m \%\), and an elitism \({ }^{8}\) rate \(e \%\). Parents are selected using tournament selection to contribute \(c \%\) of the new population using crossover. These new chromosomes are mutated at with a probability \(m\). The top scoring \(e \%\) of the current population are copied into the new population. If \(c \%+e \%<1.0\), the remainder of the new population is filled with randomly created chromosomes to maintain the population size desired. A flowchart illustrating the entire process is shown in Figure 7.14.

\footnotetext{
\({ }^{8}\) Elitism involves carrying over a number of the best chromosomes from the current population into the next.
}


Figure 7.14: A flowchart illustrating the genetic algorithm.

\section*{Summary of Algorithm Parameters}
\(\alpha\) and \(\beta \quad\) A chromosome consists of merges selected from the reduction table for the APTA associated with the problem instance. \(\alpha\) is the minimum reductions that a merge must make to be considered for inclusion in the table. \(\beta\) is the minimum EDSM score for the merge to be allowed into the table.

Population size \(p \quad\) This is the number of candidate chromosomes in any given population.

Max. generations \(g\) This corresponds to a pessimistic stopping criterion. The genetic algorithm will terminate when it has either discovered a merge path having a score of 0 , or when \(g\) generations have elapsed. In this case, the best path found so far is returned.

Tournament size \(t\) The number of chromosomes which will 'compete' in a tournament to select the best for crossover.

Chromosome size \(k\) The length in merges of a chromosome.
\begin{tabular}{ll} 
Crossover rate \(c\) & \begin{tabular}{l} 
The percentage of the new population which will be ob- \\
tained using crossover.
\end{tabular} \\
Mutation rate \(m\) & \begin{tabular}{l} 
The probability that an offspring generated using \\
crossover will be mutated.
\end{tabular} \\
Elitism rate \(e\) & \begin{tabular}{l} 
The percentage of the fittest chromosomes in the current
\end{tabular} \\
population which will be carried over into the next.
\end{tabular}

\subsection*{7.7 Summary}

In this chapter, we have run several experiments whose results guided the development of our ensemble of heuristics, Delta Graph, and genetic algorithm. The following is a synopsis of our findings:
- When studying the structure of the APTAs constructed from Abbadingostyle training sets, we observed that, as the size of the target DFA increases, the proportion of labelled strings in the APTA decreases. In other words, at the same training set density, larger DFAs have proportionally fewer labelled strings to infer the target DFA from.
- We introduced the concept of Oracle-assisted heuristics and ideal merge paths. We have used these ideas to:
- Determine the rates at which states are being reduced in a hypothesis when a state merging algorithm is converging to the target DFA.
- Determine an experimental upper bound on the performance of EDSM using a non-deterministic tie-breaking strategy (i.e. if it exists, a colourcompatible merge is selected from the highest scoring EDSM merges).
- Observe that the exact \(n\)-state target DFA is never found in fewer than \(n+1\) merge steps. Furthermore, we also observed that when EDSM identifies a low-error hypothesis, it always does so in close to \(n+1\) merge steps and never further than \(n+\frac{n}{2}\) merge steps. This result is used by the Delta Graph to abandon leaf extensions which have grown
too long, since, in expectation, they would almost certainly lead to a high-error hypothesis.
- We observed that likelihood of generating a training set which is not structurally complete is high, and that it increases proportionally with the size of the target DFA. We also determined that such training sets negatively affect the performance of EDSM to a considerable degree. In the next chapter, we will evaluate how well each of our three methods perform in this situation.
- We have experimentally established the performance of EDSM on Abbadingo problem instances as a baseline for comparing our methods against.
- We established the positive effect of getting the first \(k\) merges in a merge sequence correct (i.e. the first \(k\) merges are colour-compatible). Searching for short sequences of colour-compatible merges is the basis for both the Delta Graph and evolutionary methods we have developed.
- Inspired by the No Free Lunch theorems, and using a simple Reduction heuristic (which assigns higher scores to merges which reduce the most states in a hypothesis), we have determined that multiple heuristics can be successfully combined in an ensemble.
- Our experiments on the expected state reduction at each merge step along an ideal merge path show that, at each of these steps, low reduction merges are avoided when a heuristic is converging to the target DFA. These results inspired the APTA reduction table structure which we have shown to:
- Have considerably fewer merges than the entire set of merges possible at each step along a merge path.
- Contain colour-compatible merges at high enough positions (in the reduction table when sorted, descending, by EDSM score) to be used as branching factors in the Delta Graph method.
- Still contain a large number of colour-compatible merges which can be identified by our Delta Graph and our genetic algorithm.
- We observed the relationship between the length of a merge sequence constructed using EDSM and the error of the resulting hypothesis with respect to a test set. Specifically, the error of a hypothesis is proportional to the length of a merge sequence (as the length of a merge sequence increases, so does the error of the hypothesis).

\section*{Chapter 8}

\section*{Evaluation and Results}

In this chapter, we evaluate the performance of the ensemble of heuristics, Delta Graph, and evolutionary methods we introduced in the previous chapter. We start by describing the Abbadingo One experimental setup, specify how our results will be presented, and describe the problem instances we will be evaluating with. We will also evaluate our three methods on problem instances which are adversarial to EDSM. The exact experiments we carry out for each of our three methods are shown in Table 8.1, 8.2, and 8.3 respectively.

\footnotetext{
32-state target DFAs Symmetrically structurally complete training sets at density 1:
- 1024 problem instances.

Evaluate adversarial cases (see Section 8.1.1 below):
- 512 problem instances, training sets not structurally complete.
- 512 problem instances, immediately EDSM-failing training sets.
- 512 problem instances, no loops in target DFA.

64 -state target DFAs Symmetrically structurally complete training sets at density 1 :
}
- 1024 problem instances.

Evaluate unrestricted cases (see Section 8.2.2 below):
- 512 unrestricted problem instances.

128 -state target DFAs Symmetrically structurally complete training sets at density 1 :
- 512 problem instances.

Table 8.1: The problem instances used to evaluate the ensemble of heuristics method.

32-state target DFAs Symmetrically structurally complete training sets at density 1:
- 1024 problem instances.

Evaluate adversarial cases:
- 512 problem instances, training sets not structurally complete.
- 512 problem instances, immediately EDSM-failing training sets.
- 512 problem instances, no loops in target DFA.

64 -state target DFAs Symmetrically structurally complete training sets at density 1:
- 1024 problem instances.

Evaluate unrestricted cases (also compare with Ed-Beam and SAGE):
- 512 unrestricted problem instances.

128 -state target DFAs Symmetrically structurally complete training sets at density 1 :
- 512 problem instances.

Table 8.2: The problem instances used to evaluate the Delta Graph method.
\begin{tabular}{ll}
\hline 32-state target DFAs & Symmetrically structurally complete training sets at density 1: \\
& - 128 problem instances. \\
& Evaluating adversarial cases: \\
& - 64 problem instances, training sets not structurally complete. \\
& - 64 problem instances, immediately EDSM-failing training sets. \\
& - 64 problem instances, no loops in target DFA. \\
64-state target DFAs & Symmetrically structurally complete training sets at density 1: \\
& - 64 problem instances.
\end{tabular}

Table 8.3: The problem instances used to evaluate the genetic algorithm.

The data collected for each individual set of experiments are stored in SQLite databases as described in Section 7.1, and may be found on the media accompanying this dissertation (these results are also aggregated in Appendix A). Table 8.4 below shows which SQLite database corresponds to which problem configuration we are evaluating in this chapter. After presenting the results we obtained, we conclude with a discussion of these results, and highlight the strengths and weaknesses of each method.
\begin{tabular}{|c|c|c|c|}
\hline Database & Ensemble & Delta Graph & Genetic Algorithm \\
\hline \multicolumn{4}{|l|}{Evaluation on symmetrically structurally complete training sets} \\
\hline n32d607e1024.sqlite & \(\bullet\) & - & \\
\hline n64d1e1024.sqlite & - & - & \\
\hline n128d1e512.sqlite & & \(\bullet\) & \\
\hline n32d607e128_GA.sqlite & & & \(\bullet\) \\
\hline n64d1e64_GA.sqlite & & & - \\
\hline \multicolumn{4}{|c|}{Evaluation on adversarial setups} \\
\hline n32d607e512_EdsmFailing.sqlite & - & - & \\
\hline n32d607e512_NotStructComp.sqlite & & \(\bullet\) & \\
\hline n32d607e512_NoLoops.sqlite & & & \\
\hline n32d607e64_EdsmFailing.sqlite & & & \(\bullet\) \\
\hline n32d607e64_NotStructComp.sqlite & & & \(\bullet\) \\
\hline n32d607e64_NoLoops.sqlite & & & - \\
\hline \multicolumn{4}{|c|}{Evaluation on unrestricted setups} \\
\hline n64d1e512_Unrestricted.sqlite & - & - & \\
\hline
\end{tabular}

Table 8.4: The sets of experiments used to evaluate the ensemble of heuristics, Delta Graph, and genetic algorithm.

\subsection*{8.1 Overview of the Experimental Setup}

While our experimental setup is identical to that described in Chapter 7, we reproduce it below for the reader's convenience:
- The Abbadingo method will be used to generate target DFAs whose sizes will be exactly the number of states \(n\) requested. The Abbadingo construction depth requirement of \(\left(2 \log _{2} n-2\right)\) will always be honoured.
- Unless otherwise specified, training sets will be symmetrically structurally complete with respect to the target. Cases when the training set is not symmetrically structurally complete will be evaluated separately.
- Unless otherwise specified, the number of positive and negative strings in any training set will not be allowed to differ by more than \(20 \%\). This will avoid pathological cases where a training set would be overwhelmed by strings of a single class.
- All training sets are at Abbadingo density 1 with the number of training strings taken from [LPP98]. We define training sets for 32-state target DFA problems at density 1 as having exactly 607 strings. Our motivation for this was described in Chapter 7.
- The size of the testing sets for all problem instances (32, 64 , and 128 -states) will be exactly 1,800 strings as in the Abbadingo One competition.

\section*{Presentation of Results}

The results obtained for each experimental run are presented exactly as described in Section 7.5.2. In summary, we show the percentage of experiments where we have obtained \(\leq 1 \%\) error on a testing set, the percentage of experiments where we found hypotheses whose sizes are within \(\pm 1\) states of the target, the percentage of experiments where we found the exact target, and the mean and median hypothesis sizes.

\section*{Hardware and Platform}

All the experiments were run on a Mid-2015 MacBook Pro 15-inch, quadcore, 2.5 ghz Intel Core i7 with 16 GB DDR3 RAM running macOS High Sierra 10.13. Algorithms are implemented in Go version 1.11.5 GoL.

\subsection*{8.1.1 Adversarial Setups}

To further analyse the behaviour and performance of our methods, we will also be experimenting with data sets which are highly adversarial to EDSM. These include:
1. Non-structurally complete training sets: training sets are constructed such that either not all states in the target are reached, or not all transitions are exercised. In Section 7.5.5, we have presented a number of experiments showing the extent with which such training sets negatively affect EDSM.
2. Immediately EDSM-failing training sets: training sets are constructed such
that the set of highest EDSM scoring merges (ties) in the APTA does not contain a colour-compatible merge. In other words, the first choice made by EDSM will be guaranteed to be the wrong one.
3. No loops in target DFAs: target DFAs will not contain any loops (transitions from a state to itself). As discussed in Section 7.6.1, we observe an appreciable decrease in EDSM's performance on problem instances where the target DFA does not contain loops.

\subsection*{8.2 Evaluating the Ensemble of Heuristics}

Our ensemble of heuristics will be evaluated as follows:
- Problem instances:
- \(1024 \times 32\)-state target DFA problem instances (n32d607e1024.sqlite), \(1024 \times 64\)-state target DFA problem instances (n64d1e1024.sqlite), and \(512 \times 128\)-state target DFA problem instances (n128d1e512.sqlite) each having a symmetrically structurally complete training set at density 1 with respect to the target DFA size.
- \(512 \times 32\)-state target DFA problem instances where the training set is not structurally complete (n32d607e512_NotStructComp.sqlite).
- \(512 \times 32\)-state target DFA problem instances where the target DFA does not contain loop transitions from a state to itself (n32d607e512_NoLoops.sqlite).
- \(512 \times 32\)-state target DFA problem instances where the set of highest EDSM-scoring merges (ties) do not contain a colour-compatible merge (n32d607e512_EdsmFailing.sqlite).
- \(512 \times 64\)-state target DFA problem instances with no restrictions on structural completeness, presence of loop transitions, or colour-compatible merges in the highest EDSM-scoring ties (n64d1e512_Unrestricted.sqlite).
- The heuristics in the ensemble are described in Section 7.6.1, and are named A1, A2, ... L3. Since the ensemble contains 33 heuristics, to improve online performance, each heuristic is evaluated using a windowed implementation.
- A1/A2/A3/... will be used to denote the best result obtained from heuristic A1 or A2 or A3....
- For performance reasons, W-EDSM (rather than EDSM) is used as a baseline for comparison and evaluation on 128 -state target DFA problem instances.

An illustration of how the ensemble of heuristics is used to find a hypothesis is shown in Figure 8.1, and its performance on symmetrically structurally complete training data (showing EDSM and/or W-EDSM as a baseline) is tabulated in Table 8.5,


Figure 8.1: An illustration showing how the ensemble of heuristics method will be evaluated.
\begin{tabular}{lccccc}
\hline \multicolumn{5}{c}{ n32d607e1024.sqlite } \\
32-State Target, \(\mathbf{6 0 7}\) Strings, \(\mathbf{1 0 2 4}\) Experiments & \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM (E) & \(15.7 \%\) & \(19.6 \%\) & \(4.3 \%\) & 75 & 90 \\
W-EDSM (WE) & \(14.1 \%\) & \(17.78 \%\) & \(3.6 \%\) & 77 & 92 \\
Reduction (E) & \(6.1 \%\) & \(7.7 \%\) & \(1.8 \%\) & 88 & 97 \\
W-Reduction (WR) & \(5.9 \%\) & \(7.8 \%\) & \(1.3 \%\) & 89 & 99 \\
\hline E/WE/R/WR & \(21.8 \%\) & \(24.7 \%\) & \(6.8 \%\) & 71 & 88 \\
A1/A2/... & \(28.4 \%\) & \(30.3 \%\) & \(13.4 \%\) & 67 & 83 \\
E/WE/R/WR/A1/... & \(29.5 \%\) & \(31.5 \%\) & \(14 \%\) & 66 & 82 \\
\hline
\end{tabular}
\begin{tabular}{lccccc}
\hline \multicolumn{6}{c}{ n64d1e1024.sqlite } \\
64-State Target, & 1,521 Strings, & 1024 & Experiments \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM (E) & \(15.2 \%\) & \(17.7 \%\) & \(0.6 \%\) & 162 & 200 \\
W-EDSM (WE) & \(13.8 \%\) & \(16.5 \%\) & \(0.5 \%\) & 170 & 207 \\
Reduction (R) & \(2.4 \%\) & \(3 \%\) & \(0.1 \%\) & 204 & 219 \\
W-Reduction (WR) & \(2.5 \%\) & \(2.8 \%\) & \(0.1 \%\) & 205 & 221 \\
\hline E/WE/R/WR & \(19 \%\) & \(20.7 \%\) & \(1 \%\) & 158 & 199 \\
A1/A2/.. & \(24.3 \%\) & \(24.5 \%\) & \(4.1 \%\) & 149 & 193 \\
E/WE/R/WR/A1/.. & \(25.8 \%\) & \(25.9 \%\) & \(4.2 \%\) & 147 & 193 \\
\hline
\end{tabular}
\(\qquad\)
n128d1e512.sqlite
128-State Target, 4,382 Strings, 512 Experiments
\begin{tabular}{lccccc} 
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline W-EDSM (WE) & \(22.7 \%\) & \(15.4 \%\) & \(0 \%\) & 377 & 511 \\
W-Reduction (WR) & \(2 \%\) & \(1.6 \%\) & \(0 \%\) & 519 & 558 \\
\hline WE/WR & \(23.2 \%\) & \(16.4 \%\) & \(0 \%\) & 375 & 511 \\
A1/A2/.. & \(30.3 \%\) & \(22.9 \%\) & \(1.4 \%\) & 342 & 455 \\
WE/WR/A1/.. & \(30.7 \%\) & \(23.4 \%\) & \(1.4 \%\) & 340 & 521 \\
\hline
\end{tabular}

Table 8.5: The performance of the ensemble of heuristics on 32,64 , and 128 -state target problems on symmetrically structurally complete training sets.

\subsection*{8.2.1 Adversarial Setups}

The ensemble of heuristics is also evaluated against the adversarial setups described in the previous section. Tables 8.6, 8.7, and 8.8 show the results obtained
by the ensemble of heuristics on training sets which are not structurally complete, immediately failing training sets, and target DFAs with no loops respectively.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{5}{|l|}{Non-Structurally Complete Training Sets (n32d607e512_NotStructComp.sqlite)} & omp.sqlite) \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM (E) & 8.4\% & 15.2\% & 0\% & 76 & 90 \\
\hline W-EDSM (WE) & 9.6\% & 14.1\% & 0\% & 78 & 91 \\
\hline Reduction (R) & 2.3\% & 5.3\% & 0\% & 91 & 97 \\
\hline W-Reduction (WR) & 2.5\% & 4.7\% & \(0 \%\) & 92 & 99 \\
\hline E/WE/R/WR & 12.9\% & 18.9\% & 0\% & 72 & 87 \\
\hline A1/A2/... & 16.8\% & 21.7\% & 0\% & 70 & 85 \\
\hline E/WE/R/WR/A1/... & 18.2\% & 23.2\% & 0\% & 68 & 83 \\
\hline
\end{tabular}

Table 8.6: The performance of the ensemble of heuristics when training data is not symmetrically structurally complete.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{Immediately Failing Training Sets (n32d607e512_EdsmFailing.sqlite) 32-State Target, 607 Strings, 512 Experiments} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM (E) & 0.2\% & 0.2\% & 0\% & 92 & 93 \\
\hline W-EDSM (WE) & 0.2\% & 0.2\% & 0\% & 94 & 96 \\
\hline Reduction (R) & 1.6\% & 2.3\% & 0.2\% & 96 & 98 \\
\hline W-Reduction (WR) & 1.6\% & \(2 \%\) & 0.2\% & 97 & 100 \\
\hline E/WE/R/WR & 1.8\% & 2.3\% & 0.4\% & 89 & 92 \\
\hline A1/A2/... & 3.5\% & 3.9\% & 0.6\% & 85 & 90 \\
\hline E/WE/R/WR/A1/... & \(3.5 \%\) & 4.1\% & 1\% & 85 & 90 \\
\hline
\end{tabular}

Table 8.7: The performance of the ensemble of heuristics when the highest scoring merges in the APTA do not contain a colour-compatible merge.

No Loops in Target DFA (n32d607e512_NoLoops.sqlite)
32-State Target, 607 Strings, 512 Experiments
\begin{tabular}{lccccc} 
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM (E) & \(8 \%\) & \(13.5 \%\) & \(2 \%\) & 82 & 92 \\
W-EDSM (WE) & \(8.4 \%\) & \(13.1 \%\) & \(1.2 \%\) & 83 & 95 \\
Reduction (R) & \(3.3 \%\) & \(5.1 \%\) & \(0.6 \%\) & 93 & 99 \\
W-Reduction (WR) & \(2.7 \%\) & \(4.9 \%\) & \(0.4 \%\) & 94 & 100 \\
\hline E/WE/R/WR & \(12.3 \%\) & \(16.6 \%\) & \(2.3 \%\) & 78 & 91 \\
A1/A2/.. & \(20.3 \%\) & \(22.5 \%\) & \(6.8 \%\) & 74 & 89 \\
E/WE/R/WR/A1/.. & \(20.5 \%\) & \(22.7 \%\) & \(6.8 \%\) & 73 & 88 \\
\hline
\end{tabular}

Table 8.8: The performance of the ensemble of heuristics when the target DFA does not contain any loops.

\subsection*{8.2.2 Performance on Unrestricted Problem Instances}

In this section, we present the results obtained when applying the ensemble of heuristics to problem sets which follow the Abbadingo specifications without any restrictions. This means that: the training data may or may not be symmetrically structurally complete, the highest scoring merges (first rank ties) in the APTAs constructed from the training data may or may not contain colour-compatible merges, the target DFAs may or may not contain loops, there is no requirement on the proportions of positive and negative strings in the training set, the target DFA has exactly the number of states \(n\) specified, and the target DFA has a depth of exactly \(\left(2 \log _{2} n-2\right)\). An exact breakdown of the composition of these experiments may be found in Section A.3.3 of Appendix A. As can be seen in the results shown in Table 8.9 below, the ensemble of heuristics gives an improvement of about \(1.7 \times\) over EDSM.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{Unrestricted Problem Instances (n64d1e512_Unrestricted.sqlite)} \\
\hline \multicolumn{6}{|c|}{64-State Target, 1,521 Strings, 512 Experiments} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean & Median \\
\hline EDSM (E) & 12.1\% & 12.5\% & \(0 \%\) & 162 & 197 \\
\hline W-EDSM (WE) & 9.8\% & 12.1\% & \(0 \%\) & 170 & 205 \\
\hline Reduction (R) & 1.4\% & \(0 \%\) & \(0 \%\) & 205 & 218 \\
\hline W-Reduction (WR) & 1.6\% & 0\% & \(0 \%\) & 207 & 219 \\
\hline E/WE/R/WR & 14.1\% & 15.4\% & \(0 \%\) & 159 & 196 \\
\hline A1/A2/... & 19.5\% & 18.9\% & 1\% & 156 & 196 \\
\hline E/WE/R/WR/A1/... & 20.3\% & 19.7\% & 1\% & 153 & 193 \\
\hline
\end{tabular}

Table 8.9: The performance of the ensemble of heuristics on unrestricted Abbadingo-style problem instances.

\subsection*{8.2.3 Observations and Discussion}

The observations drawn from these experiments are:
- In Table 8.5, we observe that, when training sets are symmetrically structurally complete with respect to the target DFA, the ensemble of heuristics considerably outperforms EDSM and W-EDSM in all cases. The experimental likelihood of identifying low-error hypotheses improves as follows:
- 32-states: EDSM \(\mathbf{1 5 . 7 \%} \rightarrow\) Ensemble \(\mathbf{2 9 . 5 \%} \approx 1.9 \times\).
- 64-states: EDSM \(\mathbf{1 5 . 2 \%} \rightarrow\) Ensemble \(\mathbf{2 5 . 8 \%} \approx 1.7 \times\).
- 128-states: W-EDSM \(\mathbf{2 2 . 7 \%} \rightarrow\) Ensemble \(\mathbf{3 0 . 7 \%} \approx 1.4 \times\).
- While the ensemble of heuristics outperforms EDSM and W-EDSM in all cases, its relative effectiveness decreases as the size of the target DFA increases. While we conjecture that this phenomenon is related to the proportion of labelled states in an APTA with respect to its size (see Table 7.4), this matter needs to be investigated further.
- In the symmetrically structurally complete training set case, the likelihood of the ensemble of heuristics finding the exact target increases considerably. For instance, in Table 8.5 we see that, for 32 -state problems, EDSM manages to find the target in \(4.3 \%\) of the cases, while the ensemble of heuristics manages
in \(14 \%\) of the same problem instances. This represents an improvement of \(\approx 3.3 \times\). The same observation can be made for 64 and 128 -state target problems.
- Non-structurally complete training sets also adversarially affect the performance of the ensemble of heuristics. However, we see that the ensemble of heuristics still performs much better than either EDSM or W-EDSM in these cases. In Table 8.6, we see that EDSM can identify low-error hypotheses in \(8.4 \%\) of the cases, while the ensemble of heuristics can do so at a rate of \(18.2 \%(\approx 2 \cdot 2 \times)\).
- Recall that when training sets are randomly constructed, there is a high likelihood that they are not symmetrically structurally complete (see Section 7.5.5). Because of this, and the fact that the ensemble of heuristics performs much better than EDSM both when the training data is and is not symmetrically structurally complete, the ensemble of heuristics will perform even better on unrestricted data sets. This is verified by the results shown in Table 8.9,
- As expected, when the training sets are not structurally complete, neither method can ever find the target (see Chapter 3).
- The immediately failing training set cases are designed such that a colourcompatible merge would not exist in the highest EDSM-scoring rank (set of ties). As such, EDSM's first choice will always be wrong and the heuristic never manages to identify the exact target, and only finds a low-error hypothesis in \(0.2 \%\) of 512 problem instances. In these cases, the ensemble of heuristics performs better ( \(3.5 \%\) out of 512 instances).
- We see a considerable improvement in the adversarial cases when the target DFA does not contain any loops. In Table 8.8 , we see that, for 32 -state target DFA problems, EDSM finds low-error hypotheses in \(8 \%\) of all 512 problem instances, whereas the ensemble of heuristics does so in \(20.5 \%\) of the same cases.
- Figure 8.2 shows the experimentally derived likelihood with which each heuristic in the ensemble finds low-error hypotheses over 1024 problem instances for a 64 -state target at density 1 setup. We note that several heuristics, individually, perform better than EDSM, while the heuristics E1, E2, and E3 never manage to find a low-error hypothesis. This anomaly requires further investigation. Similar results are seen for 32 and 128-state problem setups. Furthermore, in Figure 8.3, we see that performing the same analysis on non-structurally complete data set shows that individual heuristics in the ensemble perform better than EDSM to an even greater extent.
- Table 8.10 shows an extract of the error rates obtained for each of the 1024 problem instances per heuristic for 64 -state target, density 1 problems. We can clearly see instances where at least one of the heuristics manages to find low-error hypotheses when EDSM, W-EDSM, Reduction, and W-Reduction do not. On instances 11 and 37, we see that several heuristics in the ensemble manage to identify a hypothesis with an error of \(\leq 1 \%\) on the test set (and EDSM does not), while on instance 70 , none of them do. The complete set of results may be found in the 'n64d1e1024_Ensemble.xlsx' Microsoft Excel sheet on the accompanying media.
- Analysing these results in a correlation matrix in Figure 8.4, shows the interrelations between heuristics in the ensemble. For example, the heuristics A1, A2, and A3 are very strongly correlated, and that there is a weaker, yet notable, correlation between each heuristic A1, A2, A3, ..., L3 to either EDSM or reduction. These strong correlations may allow us to optimise the runtime performance of the ensemble by omitting heuristics which are highly correlated with others (since we would expect almost identical performance between highly correlated heuristics). The complete data tables and correlation matrices may be found in the file 'n64d1e1024_Ensemble_Correlation.xlsx' on the media accompanying this document.
- The computational cost compared to EDSM is a linear function of heuristics in the ensemble.


Figure 8.2: Abbadingo win success rate per ensemble heuristic. Showing results for 64 -state targets, density 1 ( 1,521 strings), 1024 problem instances per heuristic. The heuristics in the ensemble which individually outperformed EDSM are shown in green.


Figure 8.3: Abbadingo win success rate per ensemble heuristic when training data is not structurally complete. Showing results for 32 -state targets, density 1 (607 strings), 64 problem instances per heuristic. The heuristics in the ensemble which individually outperformed EDSM are shown in green.
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline 苟 & － & \％ & \％ & － & ヘ & ® & ర & กิ & \(\because\) & \(\vec{\square}\) & คิ & D3 & 血 & 㑭 & 箇 & 层 & N & 等 & च & กิ & \％ & \＃ & ヘิ & 甲 & \(\exists\) & \(\exists\) & ๓ & 5 & \(\stackrel{\sim}{\sim}\) & \(\stackrel{\square}{\sim}\) & \(\sqrt{3}\) & \％ & 9 & コ & ㄲ & 9 & E
W
易 &  & & E
\＃
\＃
0
0
3
3 \\
\hline 11 & 0.46 & 0.46 & 0.45 & 0.46 & 0.47 & 0.45 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.00 & 0.46 & 0.47 & 0.48 & 0.46 & 0.45 & 0.46 & 0.01 & 0.01 & 0.01 & 0.01 & 0.00 & 0.01 & 0.22 & 0.22 & 0.22 & 0.46 & 0.47 & 0.47 & 0.46 & 0.46 & 0.46 & 0.46 & 0.47 & 0.46 & 0.44 & 0.07 & 0.47 & 0.45 \\
\hline 37 & 0.11 & 0.12 & 0.12 & 0.12 & 0.11 & 0.12 & 0.00 & 0.00 & 0.00 & 0.10 & 0.09 & 0.10 & 0.12 & 0.11 & 0.11 & 0.12 & 0.13 & 0.12 & 0.38 & 0.37 & 0.36 & 0.10 & 0.09 & 0.10 & 0.41 & 0.40 & 0.40 & 0.43 & 0.42 & 0.42 & 0.42 & 0.43 & 0.43 & 0.43 & 0.42 & 0.42 & 0.04 & 0.21 & 0.41 & 0.43 \\
\hline 43 & 0.47 & 0.47 & 0.48 & 0.47 & 0.47 & 0.47 & 0.46 & 0.49 & 0.48 & 0.46 & 0.47 & 0.49 & 0.46 & 0.47 & 0.46 & 0.47 & 0.47 & 0.47 & 0.45 & 0.45 & 0.45 & 0.47 & 0.48 & 0.48 & 0.45 & 0.46 & 0.46 & 0.00 & 0.01 & 0.01 & 0.44 & 0.46 & 0.44 & 0.01 & 0.00 & 0.01 & 0.46 & 0.47 & 0.01 & 0.00 \\
\hline 50 & 0.49 & 0.49 & 0.49 & 0.48 & 0.48 & 0.48 & 0.01 & 0.01 & 0.01 & 0.01 & 0.00 & 0.00 & 0.50 & 0.50 & 0.49 & 0.48 & 0.49 & 0.49 & 0.05 & 0.05 & 0.05 & 0.01 & 0.01 & 0.00 & 0.49 & 0.48 & 0.48 & 0.48 & 0.48 & 0.47 & 0.49 & 0.48 & 0.49 & 0.47 & 0.48 & 0.48 & 0.38 & 0.48 & 0.48 & 0.47 \\
\hline 66 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.05 & 0.05 & 0.05 & 0.01 & 0.01 & 0.01 & 0.05 & 0.05 & 0.05 & 0.01 & 0.01 & 0.01 & 0.27 & 0.29 & 0.27 & 0.01 & 0.01 & 0.01 & 0.18 & 0.19 & 0.19 & 0.28 & 0.29 & 0.28 & 0.28 & 0.28 & 0.27 & 0.27 & 0.27 & 0.27 & 0.01 & 0.01 & 0.29 & 0.28 \\
\hline 70 & 0.50 & 0.49 & 0.49 & 0.49 & 0.48 & 0.49 & 0.50 & 0.52 & 0.51 & 0.49 & 0.50 & 0.49 & 0.49 & 0.49 & 0.48 & 0.48 & 0.49 & 0.51 & 0.47 & 0.47 & 0.46 & 0.49 & 0.49 & 0.49 & 0.49 & 0.49 & 0.47 & 0.47 & 0.50 & 0.50 & 0.50 & 0.48 & 0.48 & 0.48 & 0.49 & 0.49 & 0.49 & 0.48 & 0.48 & 0.48 \\
\hline 88 & 0.00 & 0.00 & 0.01 & 0.00 & 0.01 & 0.01 & 0.00 & 0.00 & 0.01 & 0.00 & 0.00 & 0.00 & 0.07 & 0.06 & 0.06 & 0.00 & 0.00 & 0.00 & 0.06 & 0.06 & 0.06 & 0.00 & 0.00 & 0.01 & 0.49 & 0.47 & 0.49 & 0.49 & 0.48 & 0.47 & 0.48 & 0.49 & 0.48 & 0.48 & 0.49 & 0.48 & 0.00 & 0.01 & 0.48 & 0.49 \\
\hline 109 & 0.48 & 0.48 & 0.47 & 0.48 & 0.48 & 0.49 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.48 & 0.47 & 0.49 & 0.48 & 0.49 & 0.47 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.50 & 0.50 & 0.49 & 0.49 & 0.49 & 0.50 & 0.49 & 0.48 & 0.48 & 0.50 & 0.49 & 0.49 & 0.45 & 0.01 & 0.50 & 0.50 \\
\hline & － & 8.1 & & Ext & act & t fro & Om & ＇n6 & \(64 d 1\) & 1e10 & 024 & Ens & sem & mble & e． x & X＇ & ， & wing & g t & e & erro & r & oun & d & t & 2 & dec & ma & 1 & ace & & obt & ain & & per & he & ri & tic & in & he \\
\hline
\end{tabular}


Figure 8.4: The correlation matrix showing the relationship between individual heuristics in the ensemble and hypothesis error (rounded to 2 decimal places). The complete set of results may be found in the 'n64d1e1024_Ensemble_Correlation.xlsx' Microsoft Excel sheet on the accompanying media.

\subsection*{8.3 Evaluating the Delta Graph}

The Delta Graph method will be evaluated as follows:
- Problem instances:
- \(1024 \times 32\)-state target DFA problem instances (n32d607e1024.sqlite), \(1024 \times 64\)-state target DFA problem instances (n64d1e1024.sqlite), and \(512 \times 128\)-state target DFA problem instances (n128d1e512.sqlite) each having a symmetrically structurally complete training set at density 1 with respect to the target DFA size.
\(-512 \times 32\)-state target DFA problem instances where the training set is not structurally complete (n32d607e512_NotStructComp.sqlite).
- \(512 \times 32\)-state target DFA problem instances where the target DFA does not contain loop transitions from a state to itself (n32d607e512_NoLoops.sqlite).
- \(512 \times 32\)-state target DFA problem instances where the set of highest EDSM-scoring merges (ties) do not contain a colour-compatible merge (n32d607e512_EdsmFailing.sqlite).
\(-512 \times 64\)-state target DFA problem instances with no restrictions on structural completeness, presence of loop transitions, or colour-compatible merges in the highest EDSM-scoring ties (n64d1e512_Unrestricted.sqlite). We will also be using this data set to compare our results with those reported for Ed-Beam and SAGE by Lang in Lan99.
- The Delta Graph will be evaluated against several depth, branching factor, branching limit, and APTA reduction table configurations.
- We will study the behaviour of a special case of the Delta Graph called the 'GFirst' variant. This enables us to measure the benefit of identifying several colour-compatible initial merge sequences that the Delta Graph method allows.
- For performance reasons, W-EDSM (rather than EDSM) is used as a baseline for comparison and evaluation on 128-state target DFA problem instances.
- An illustration of how the Delta Graph is used to identify a heuristic is shown in Figure 8.5


Figure 8.5: An illustration showing how the Delta Graph method will be evaluated.

As described in the previous chapter, the Delta Graph requires us to extend leaf hypotheses in the graph using W-EDSM and score each leaf with:
\[
\text { Score }=\mathrm{AbS}(\text { HypothesisSize }- \text { TARgetSize })
\]
... which we wish to minimise (a score of 0 means that we found a hypothesis equal in size to the target). When running our experiments, we are in possession of the target DFA. This means that during the construction of the Delta Graph we are privy to the colour-compatibility of the merges and the hypotheses in it (which is not available in the 'real world'). As a runtime optimisation, we will only expand leaves which are colour-compatible. This allows us to only consider good leaves as starting points and avoid needlessly processing leaves which would, with high probability, lead to high-error hypotheses. In the 'real world', we wouldn't be in possession of the target for colouring purposes and would therefore need to extend all leaves. In the worst case, using this shortcut can only result in an under estimate of the actual performance of the Delta Graph method since we are only processing a subset of the actual leaves in the graph. Also recall that, using this method, when we are expanding leaves using W-EDSM, we abort paths longer than \(n+\frac{n}{2}\) merge steps since it is virtually certain that they will lead to
a high-error hypothesis. Should all the leaves be aborted in such a manner, the Delta Graph method would have failed and we fall back to W-EDSM to get a result.

The performance of the Delta Graph method, compared to EDSM and/or WEDSM as a baseline, for various configurations is shown in Table 8.11. 'GFirst' is a variant of the Delta Graph which only extends the first colour-compatible leaf it encounters (if any exists) rather than expanding them all \({ }^{11}\). Finally, Delta Graph configurations will be named using the following scheme: Graph \(/ \delta / \alpha / \beta / b / f\), where \(b\) is the branching factor, \(f\) is the branching limit, \(\delta\) is the maximum depth of the graph, \(\alpha\) is the minimum reduction of merges in the APTA reduction table, and \(\beta\) is the minimum EDSM score of merges in the APTA reduction table.

\section*{Delta Graph Parameter Selection}

The parameters chosen to run the Delta Graph with (the reduction table \(\alpha\) and \(\beta\), and the depth, branching factor, and branching limit) were selected following an analysis similar to what we have done in Section 7.5.8. We chose combinations of parameters to, experimentally, improve the likelihoods that (i) the size of the Delta Graph is manageable (in terms of the number of nodes and leaves it contains), (ii) there are several colour-compatible leaf nodes, and (iii) when the APTA reduction table is sorted by EDSM, colour-compatible merges may be found within the branching factor of the Delta Graph.

\footnotetext{
\({ }^{1}\) The significance of this variation will be explored in the discussion section of this method.
}
\begin{tabular}{lccccc}
\hline \multicolumn{6}{c}{ n32d607e1024.sqlite } \\
32-State Target, \(\mathbf{6 0 7}\) Strings, & 1024 & Experiments \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM & \(15.7 \%\) & \(19.6 \%\) & \(4.3 \%\) & 75 & 90 \\
W-EDSM & \(14.1 \%\) & \(17.78 \%\) & \(3.6 \%\) & 77 & 92 \\
\hline GFirst \(/ 3 / 25 / 1 / 6 / 3\) & \(28.9 \%\) & \(38.6 \%\) & \(7.3 \%\) & 59 & 46 \\
GFirst \(/ 3 / 25 / 1 / 12 / 3\) & \(31.3 \%\) & \(42.4 \%\) & \(8.8 \%\) & 55 & 38 \\
GFirst \(/ 4 / 25 / 1 / 10 / 3\) & \(35.3 \%\) & \(46.6 \%\) & \(9.6 \%\) & 52 & 34 \\
\hline Graph \(/ 3 / 25 / 1 / 6 / 3\) & \(40.1 \%\) & \(52.2 \%\) & \(10.4 \%\) & 55 & 33 \\
Graph \(/ 3 / 25 / 1 / 12 / 3\) & \(50.7 \%\) & \(65.2 \%\) & \(14.4 \%\) & 47 & 32 \\
Graph \(/ 4 / 25 / 1 / 10 / 3\) & \(49 \%\) & \(62.2 \%\) & \(13.6 \%\) & 49 & 32 \\
\hline
\end{tabular}
\begin{tabular}{lccccc}
\hline \multicolumn{6}{c}{ n64d1e1024.sqlite } \\
64-State Target, & 1,521 Strings, & 1024 & Experiments \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM & \(15.2 \%\) & \(17.7 \%\) & \(0.6 \%\) & 162 & 200 \\
W-EDSM & \(13.8 \%\) & \(16.5 \%\) & \(0.5 \%\) & 170 & 207 \\
\hline GFirst \(/ 3 / 60 / 1 / 12 / 3\) & \(27.3 \%\) & \(30.6 \%\) & \(1.7 \%\) & 133 & 111 \\
GFirst \(/ 4 / 60 / 1 / 10 / 3\) & \(31.2 \%\) & \(33.1 \%\) & \(1 \%\) & 128 & 98 \\
\hline Graph \(/ 3 / 60 / 1 / 12 / 3\) & \(42.9 \%\) & \(47 \%\) & \(2.2 \%\) & 113 & 65 \\
Graph \(/ 4 / 60 / 1 / 10 / 3\) & \(40.1 \%\) & \(44.8 \%\) & \(1.7 \%\) & 118 & 67 \\
\hline
\end{tabular}
n128d1e512.sqlite
128-State Target, 4,382 Strings, 512 Experiments
\begin{tabular}{lccccc} 
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline W-EDSM & \(22.7 \%\) & \(15.4 \%\) & \(0 \%\) & 377 & 511 \\
\hline GFirst \(/ 3 / 180 / 1 / 12 / 3\) & \(27.1 \%\) & \(18.6 \%\) & \(0 \%\) & 341 & 388 \\
GFirst \(/ 4 / 180 / 1 / 10 / 3\) & \(33.2 \%\) & \(23.6 \%\) & \(0.2 \%\) & 309 & 241 \\
\hline Graph \(/ 3 / 180 / 1 / 12 / 3\) & \(48.6 \%\) & \(38.1 \%\) & \(0.4 \%\) & 272 & 140 \\
Graph \(/ 4 / 180 / 1 / 10 / 3\) & \(47.3 \%\) & \(36.3 \%\) & \(0.4 \%\) & 280 & 148 \\
Graph \(/ 4 / 180 / 1 / 12 / 3\) & \(49.4 \%\) & \(39.1 \%\) & \(0.4 \%\) & 275 & 134 \\
\hline
\end{tabular}

Table 8.11: The performance of various delta graph configurations on 32, 64, and 128-state target problems with symmetrically structurally complete training sets.

\subsection*{8.3.1 Adversarial Setups}

We also evaluate the Delta Graph against the adversarial setups described earlier. Tables 8.12, 8.13, and 8.14 show the results obtained by the Delta Graph when training sets are not structurally complete, immediately failing training sets, and target DFAs with no loops respectively.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{32-State Target, 607 Strings, 512 Experiments} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM & 8.4\% & 15.2\% & \(0 \%\) & 76 & 90 \\
\hline W-EDSM & 9.6\% & 14.1\% & \(0 \%\) & 78 & 91 \\
\hline Graph/3/25/1/6/3 & 25.2\% & 41.8\% & \(0 \%\) & 58 & 35 \\
\hline Graph/3/25/1/12/3 & 34.2\% & 57.6\% & 0\% & 48 & 32 \\
\hline Graph/4/25/1/10/3 & 31.8\% & 53.9\% & \(0 \%\) & 51 & 32 \\
\hline
\end{tabular}

Table 8.12: The performance of the Delta Graph when training data is not symmetrically structurally complete.
\begin{tabular}{|c|c|c|c|c|c|}
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM & 0.2\% & 0.2\% & 0\% & 92 & 93 \\
\hline W-EDSM & 0.2\% & 0.2\% & 0\% & 94 & 96 \\
\hline Graph/3/25/1/6/3 & 20.9\% & \(31.4 \%\) & 4.1\% & 71 & 92 \\
\hline Graph/3/25/1/12/3 & 35.4\% & 50.6\% & 5.1\% & 56 & 34 \\
\hline Graph/4/25/1/10/3 & 32.6\% & 46.5\% & 5.5\% & 59 & 35 \\
\hline
\end{tabular}

Table 8.13: The performance of the Delta Graph when the highest scoring merges in the APTA do not contain a colour-compatible merge.
\begin{tabular}{lccccc}
\hline \multicolumn{5}{c}{ No Loops in Target DFA (n32d607e512_NoLoops.sqlite) } \\
32-State Target, \(\mathbf{6 0 7}\) Strings, & 512 & Experiments & \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM & \(8 \%\) & \(13.5 \%\) & \(2 \%\) & 82 & 92 \\
W-EDSM & \(8.4 \%\) & \(13.1 \%\) & \(1.2 \%\) & 83 & 95 \\
\hline Graph \(/ 3 / 25 / 1 / 6 / 3\) & \(29.3 \%\) & \(42.6 \%\) & \(7.4 \%\) & 63 & 41 \\
Graph \(/ 3 / 25 / 1 / 12 / 3\) & \(41.8 \%\) & \(57.4 \%\) & \(9.2 \%\) & 51 & 32 \\
Graph \(/ 4 / 25 / 1 / 10 / 3\) & \(38.9 \%\) & \(54.3 \%\) & \(7 \%\) & 54 & 32 \\
\hline
\end{tabular}

Table 8.14: The performance of the Delta Graph when the target DFA does not contain any loops.

\subsection*{8.3.2 Discussion and Observations}

The observations drawn from these experiments include:
- In Table 8.11, we observe that, when training sets are symmetrically structurally complete with respect to the target DFA, the Delta Graph considerably outperforms EDSM and W-EDSM in all cases. The experimental likelihood of identifying low-error hypotheses improves as follows (showing the best-case Delta Graph configuration):
- 32-states: EDSM \(\mathbf{1 5 . 7 \%} \rightarrow\) Graph \(/ 3 / 25 / 1 / 12 / 350.7 \% \approx 3.2 \times\).
- 64-states: EDSM \(\mathbf{1 5 . 2 \%} \rightarrow\) Graph \(/ 3 / 60 / 1 / 12 / 342.9 \% \approx 2.8 \times\).
- 128-states: W-EDSM \(\mathbf{2 2 . 7 \%} \rightarrow\) Graph \(/ 4 / 180 / 1 / 12 / 349.4 \% \approx 2.2 \times\).
- While the Delta Graph outperforms EDSM and W-EDSM in all cases, its relative effectiveness decreases as the size of the target DFA increases. While we conjecture that this phenomenon is related to the proportion of labelled states in an APTA with respect to its size (see Table 7.4), this matter needs to be investigated further.
- In the symmetrically structurally complete training set case, the likelihood of the Delta Graph finding the exact target increases considerably. For example, for 32 -state problems, EDSM manages to find the target in \(4.3 \%\) of the cases, while the Delta Graph manages in \(14.4 \%\) of the cases (Graph/3/25/1/12/3).

This represents an improvement of about \(3.3 \times\). A similar observation can be made for 64 and 128 -state target problems.
- Non-structurally complete training sets also adversarially affect the performance of the Delta Graph. However, we see that the Delta Graph still performs much better than EDSM or W-EDSM in these cases. In Table 8.12, we see that EDSM can identify low-error hypotheses in \(8.4 \%\) of the cases, while the Delta Graph can do so at a rate of \(34.2 \%(\approx 4.1 \times)\) when using the Graph/3/25/1/12/3 configuration.
- Recall that when training sets are randomly constructed, there is a high likelihood that they are not symmetrically structurally complete (see Section 7.5.5). Because of this, and the fact that the Delta Graph performs much better than EDSM both when the training data is and is not symmetrically structurally complete, the Delta Graph method will perform even better on unrestricted data sets. This is verified by the results shown in Table 8.15 where the Delta Graph is evaluated against unrestricted problem instances.
- As expected, when the training sets are not structurally complete neither method can ever find the target (see Chapter 3).
- The immediately failing training set cases are designed such that a colourcompatible merge would not exist in the highest EDSM-scoring rank (set of ties). As such, EDSM's first choice will always be wrong and the heuristic never manages to identify the exact target, and only finds a low-error hypothesis in \(0.2 \%\) of 512 problem instances. In these cases, the Delta Graph performs considerably better ( \(35.4 \%\) out of 512 instances) using the Graph/3/25/1/12/3 configuration. In this case, we can also see that the Delta Graph outperforms the ensemble of heuristics by almost an order of magnitude.
- The noticable improvement of the Delta Graph method on the immediately failing problem instances may be attributed to the fact that the branching factor chosen allows the search to explore merges beyond the highest EDSMscoring rank.
- In the adversarial data set which contains target DFAs without any loops, we also see a notable improvement. In Table 8.14, we see that for 32 -state target DFA problems, EDSM finds low-error hypotheses in \(8 \%\) of all 512 problem instances, whereas the Graph/3/25/1/12/3 does so in \(41.8 \%\) of the same cases.
- The computational cost of the Delta Graph method is primarily dependent on the number of leaves which need to be extended. In turn, the number of leaves depends on the graph construction parameters: the number of leaves will increase as \(\delta, b\), and/or \(f\) increase. A worst case can be determined using the \(\delta\) and \(b\) parameters when \(f=\delta\) and the graph is a tree. In practice, however, we see that due to node reuse and the fact that the values for \(\delta\), \(b\), and \(f\) do not need to be large to obtain good results, the processing time required by the method is well within what can be achieved on contemporary consumer-level hardware.

\section*{Ordering Leaves for Extension}

The Delta Graph returns a hypothesis by extending every leaf using W-EDSM, and either returns the first extension which leads to a hypothesis equal in size to the target, or the one closest \({ }^{2}\). It is then clear that we should extend the most promising leaves first to avoid processing the remaining ones after having already found a good extension. Consider problem instances consisting of 64 -state target DFA and training sets at density 1. In the Delta Graph size analysis in Section 7.6.2 we have seen that graphs constructed using the Graph/3/12/3/60/1 setup have an average of 1072 leaves out of which 58 are colour-compatible. Should we be able to order the leaves in such a way that those 58 colour-compatible ones are close to the 'top', we can experimentally establish a window of leaves to consider and discard the rest expecting that they will not be colour-compatible.

During our Delta Graph analysis we have identified that when Delta Graphs are seven or more merges deep, the cumulative EDSM score at a leaf is highly correlated with whether that leaf is colour-compatible or not. In other words,

\footnotetext{
\({ }^{2}\) As an optimisation, the Delta Graphs aborts extensions longer than \(n+\frac{n}{2}\) merges. If all leaves are aborted, the Delta Graph falls back to W-EDSM for the instance.
}
sorting leaves by cumulative EDSM score descending has a very high likelihood of moving the colour-compatible leaves towards the top. Unfortunately, constructing Delta Graphs at such a depth is prohibitively expensive. At the depths of 3 or 4 we are working with, the cumulative EDSM is not sufficient to discriminate the colour-compatibility of leaves. Identifying an alternative proxy for leaf quality remains a matter for further investigation.

\section*{Cost/Benefit}

A compromise exists when selecting the parameters to construct Delta Graphs. Consider the results obtained in Table 8.11 for problem instances consisting of 64-state target DFAs with symmetrically structurally complete training sets at density 1 :
- Graph \(/ 3 / 60 / 1 / 12 / 3\) finds low-error hypotheses in \(42.9 \%\) of the 1024 problem instances while Graph/4/60/1/10/3 does so in \(40.1 \%\) of the same instances.
- On the same hardware, Graph/4/60/1/10/3 takes less than half the time to find a hypothesis compared to Graph/3/60/1/12/3 (although the graph is deeper, the branching factor is smaller).
- Additionally, a Graph/3/60/1/12/3 configuration has an average of 1072 leaves while a Graph/4/60/1/10/3 configuration has an average of 869 leaves. This means that there are \(\approx 20 \%\) fewer path extensions to process when using the latter set of parameters.
- A compromise in terms of generalisation rate performance and online performance may be made based on these observations. For instance, for a modest decrease in performance over Graph/3/60/1/12/3, the other configuration requires less than half the time to find a hypothesis.

\section*{Not all Colour-Compatible Sequences are Created Equal}

Again, consider the results obtained in Table 8.11 for problem instances consisting of 64 -state target DFAs with symmetrically structurally complete training sets at density 1 :
- Col03-EDSM+W-EDSM guarantees a high EDSM-scoring initial sequence of three colour-compatible merges and finds low-error hypotheses \(27.4 \%\) of the time with a median hypothesis size of 113 states (see Table A. 10 in Appendix (A).
- Experimentally, Graph/3/60/1/12/3 can find an initial sequence of three colour-compatible merges only \(85 \%\) of the time (see Delta Graph size analysis in Section 7.6.2 but, yet, can find low-error hypotheses at a rate of \(42.9 \%\) (shown in Table 8.11). This is much better compared to the \(27.4 \%\) for Col03-EDSM+W-EDSM. Additionally, it can find a median hypothesis size virtually identical in size to the target.
- Moreover, Graph \(/ 3 / 60 / 1 / 12 / 3\) finds low-error hypotheses at almost the same rate (42.9\%) that Col07-EDSM+W-EDSM does (43.8\%, see Table A.10) while only exploring three merges deep rather than seven. Median hypothesis sizes are also almost identical.
- This phenomenon may be accounted for by the fact that, on average, about 58 leaves in the graph are colour-compatible partitions (see Delta Graph size analysis in Section 7.6.2. This means that the graph contains at least 58 distinct colour-compatible sequences of length three. If one sequence of three colour-compatible merges does not constrain the hypothesis well enough for the remaining \(W\)-EDSM extension to do a good job, there is scope for the remaining 57 to do so.
- This observation is supported by the fact that our alternative search strategy GFirst/3/60/1/12/3 (which just picks any one out of the 58 three-sequence of colour-compatible merges), while performing substantially better than plain EDSM, performs worse than Graph/3/60/1/12/3 which, potentially, explores all existing colour-compatible sequences of length 3 .

\section*{Comparison to Ed-Beam/B_SWET/SAGE}

Lang's Ed-Beam algorithm (discussed in Section 5.1.10) uses a similar approach of sampling the search space in the hope of constructing extensions which lead
to smaller DFAs. Targets had between 4 and 21 states and training sets were composed of roughly 550 strings. The actual data set (originally constructed by Arlindo Oliveira) is referred to by a web link in the paper which, unfortunately, no longer works. Although this testing setup makes comparing results with ours very difficult, we note that the Ed-Beam method is functionally similar to the earlier B_SWET \({ }^{3}\) method by Lang himself in Lan98. This is of interest to us since Lang's evaluation of B_SWET used the Abbadingo procedure to generate problem instances. Moreover, the sizes of the target DFAs and training sets are comparable to ours. Sampled over 500 Abbadingo-style 64-state target problem instances at a density of 1, B_SWET discovered low-error hypotheses in \(\approx 21 \%\) of the cases whereas the blue-fringe reference implementation of EDSM managed to do so in \(\approx 10 \%\) of the 500 cases \({ }^{4}\). Lang also compared his results to SAGE (see Section 5.1.9), which, in its best configuration, managed to discover low-error hypotheses in \(\approx 14 \%\) of those cases.

To compare like with like, we constructed a suite of 512 experiments for 64state target problems at the same density as Lang's experiments following the Abbadingo specifications without any restrictions. This means that, like in Lang's case, the training data may be symmetrically structurally complete or not, it may result in APTAs where the first rank EDSM scoring merges contain a colourcompatible merge or not, the target DFAs may or may not contain loops, and we place no requirement on the proportions of positive and negative strings in a training set \({ }^{5}\). The only difference in setup between our method and Lang's is that we used the reference implementation of windowed EDSM as a baseline, whereas he used the reference implementation of blue-fringe EDSM. As can be seen in the results shown in Table 8.15, the baseline EDSM performance of both our experimental setups are virtually identical ensuring that our results can be compared reliably. Here we see that using a \(10 \%\) generalisation rate for EDSM as a baseline, the BSWET makes an improvement of about \(2.1 \times\), SAGE makes an improvement of about \(1.4 \%\), and Graph/3/60/1/12/3 makes an improvement of

\footnotetext{
\({ }^{3}\) Beam, Searching Wrapper, dETerministic.
\({ }^{4}\) The results we show are approximations as we have parsed them out of charts in Lan98. Unfortunately, the results were not tabulated to get the exact values.
\({ }^{5}\) An exact breakdown of the composition of these experiments may be found in Section A.3.3
}
about \(3.4 \times\). We note that, as recently as [HV13], Ed-Beam/B_SWET is considered to be a state-of-the-art search strategy when applied to Abbadingo-style problems.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{Unrestricted Problem Instances (n64d1e512_Unrestricted.sqlite)} \\
\hline \multicolumn{6}{|c|}{64-State Target, 1,521 Strings, 512 Experiments} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean & Median \\
\hline \multicolumn{6}{|c|}{Results from [Lan98]} \\
\hline EDSM (baseline, blue-fringe) & \(\approx 10 \%\) & - & - & - & - \\
\hline B_SWET ( \(\mathrm{p}=32, \mathrm{r}=8\) ) & \(\approx 21 \%\) & - & - & - & - \\
\hline SAGE ( \(\mathrm{p}=1024\) ) & \(\approx 14 \%\) & - & - & - & - \\
\hline \multicolumn{6}{|c|}{Our Results} \\
\hline EDSM (baseline, windowed) & 9.8\% & 12.1\% & 0\% & 170 & 205 \\
\hline Graph/3/60/1/12/3 & 34.2\% & 39.8\% & 0.2\% & 121 & 70 \\
\hline Graph/4/60/1/10/3 & 30.1\% & 35.6\% & 0.6\% & 128 & 77 \\
\hline
\end{tabular}

Table 8.15: Comparing the performance of the Delta Graph method to Lang's B_SWET/Ed-Beam, and his implementation of Juillé's SAGE.

\subsection*{8.4 Evaluating the Genetic Algorithm}

The genetic algorithm will be evaluated as follows:
- Problem instances:
- \(128 \times 32\)-state target DFA problem instances (n32d607e128_GA.sqlite), and \(64 \times 64\)-state target DFA problem instances (n64d1e64_GA.sqlite) each having a symmetrically structurally complete training set at density 1 with respect to the target DFA size.
- \(64 \times 32\)-state target DFA problem instances where the training set is not structurally complete (n32d607e64_NotStructComp.sqlite).
- \(64 \times 32\)-state target \(D F A\) problem instances where the target DFA does not contain loop transitions from a state to itself (n32d607e64_NoLoops.sqlite).
- \(64 \times 32\)-state target DFA problem instances where the set of highest EDSM-scoring merges (ties) do not contain a colour-compatible merge (n32d607e64_EdsmFailing.sqlite).
- The fitness function used by the GA involves extending candidate solutions to a final border hypothesis using W-EDSM. The resulting DFA size is then used as the score we wish to minimise. The evaluation of this fitness turns out to be a major computational bottleneck, and the application of this method to 128 -state problems is not feasible on the hardware available to us.
- We will study the behaviour of a special case of the GA called the 'Blind' variant. This variant selects parents blindly (without considering their fitness) and allows us to measure effectiveness of the fitness function.
- We will also compare the effectiveness of GA with a random search (merges are selected randomly).
- An illustration of how the GA is used to identify a heuristic is shown in Figure 8.6


Figure 8.6: An illustration showing how the genetic algorithm will be evaluated.

The collection of parameters used by the GA to evolve initial sequences are given the following short names:

Random Randomly selects merges at each step to construct a path (random search).

Blind/n32/v1

GA/n32/v1

GA/n32/v2

GA/n32/v3

GA/n64/v1

GA/n64/v2

GA on 32-state target DFAs at density 1. Pop. size \(=100\), max. generations \(=50\), chrom. length \(=8\), tournament size \(=5, \alpha=25, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

Best of two runs of GA/n32/v1.

GA on 32-state target DFAs at density 1. Pop. size \(=200\), max. generations \(=100\), chrom. length \(=8\), tournament size \(=5, \alpha=25, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

GA on 64 -state target DFAs at density 1. Pop. size \(=100\), max. generations \(=50\), chrom. length \(=8\), tournament size \(=5, \alpha=25, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

GA on 64 -state target DFAs at density 1. Pop. size \(=100\), max. generations \(=50\), chrom. length \(=8\), tournament size \(=5, \alpha=60, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

GA/n64/v3

GA/n64/v4
Best of GA/n64/v1, v2, and v3.
GA/n64/v5
GA on 64 -state target DFAs at density 1. Pop. size \(=200\), max. generations \(=50\), chrom. length \(=6\), tournament size \(=5, \alpha=60, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

Table 8.17 shows the performance of the GA using the various configurations described above on 32 and 64 -state target DFA problems on symmetrically structurally complete training data.
n32d607e128_GA.sqlite
32-State Target, 607 Strings, 128 Experiments
\begin{tabular}{lccccc} 
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM & \(16.4 \%\) & \(21.1 \%\) & \(3.9 \%\) & 72 & 86 \\
\hline Random & \(0 \%\) & \(0 \%\) & \(0 \%\) & 776 & 776 \\
Blind/n32/v1 & \(0 \%\) & \(0 \%\) & \(0 \%\) & 81 & 81 \\
\hline GA/n32/v1 & \(46.9 \%\) & \(63.3 \%\) & \(9.4 \%\) & 39 & 32 \\
GA/n32/v2 & \(62.5 \%\) & \(80.5 \%\) & \(16.4 \%\) & 35 & 32 \\
GA/n32/v3 & \(68.8 \%\) & \(87.5 \%\) & \(16.4 \%\) & 34 & 32 \\
\hline
\end{tabular}
\begin{tabular}{lccccc}
\hline \multicolumn{5}{c}{ n64d1e64_GA.sqlite } \\
64-State Target, & 1,521 Strings, \(\mathbf{6 4}\) Experiments & \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM & \(12.5 \%\) & \(21.9 \%\) & \(1.6 \%\) & 161 & 199 \\
\hline \(\mathrm{GA} / \mathrm{n} 64 / \mathrm{v} 1\) & \(17.2 \%\) & \(21.9 \%\) & \(0 \%\) & 113 & 105 \\
\(\mathrm{GA} / \mathrm{n} 64 / \mathrm{v} 2\) & \(14.1 \%\) & \(25 \%\) & \(1.6 \%\) & 131 & 113 \\
\(\mathrm{GA} / \mathrm{n} 64 / \mathrm{v} 3\) & \(26.6 \%\) & \(32.8 \%\) & \(1.6 \%\) & 110 & 94 \\
\(\mathrm{GA} / \mathrm{n} 64 / \mathrm{v} 4\) & \(37.5 \%\) & \(42.2 \%\) & \(3.1 \%\) & 94 & 79 \\
\(\mathrm{GA} / \mathrm{n} 64 / \mathrm{v} 5\) & \(56.3 \%\) & \(59.4 \%\) & \(4.7 \%\) & 85.6 & 64 \\
\hline
\end{tabular}

Table 8.17: The performance of various GA configurations on 32, and 64-state target problems on symmetrically structurally complete training sets.

\subsection*{8.4.1 Adversarial Setups}

The GA is, again, evaluated against the adversarial setups described earlier. Tables \(8.18,8.19\), and 8.20 show the results obtained by the GA when training sets are not structurally complete, immediately failing training sets, and target DFAs with no loops respectively.
\begin{tabular}{lccccc}
\hline \multicolumn{6}{c}{ Non-Structurally Complete Training Sets (n32d607e64_NotStructComp.sqlite) } \\
32-State Target, & 607 Strings, \(\mathbf{6 4}\) Experiments \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM & \(9.4 \%\) & \(14.1 \%\) & \(0 \%\) & 75 & 86 \\
W-EDSM & \(10.9 \%\) & \(15.6 \%\) & \(0 \%\) & 76 & 89 \\
\hline GA/n32/v1 & \(40.6 \%\) & \(57.8 \%\) & \(0 \%\) & 41 & 32 \\
GA/n32/v2 & \(56.3 \%\) & \(73.4 \%\) & \(0 \%\) & 36 & 32 \\
\hline
\end{tabular}

Table 8.18: The performance of the GA when training data is not symmetrically structurally complete.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{Immediately Failing Training Sets (n32d607e64_EdsmFailing.sqlite)} \\
\hline \multicolumn{6}{|c|}{32-State Target, 607 Strings, 64 Experiments} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM & \(0 \%\) & \(0 \%\) & \(0 \%\) & 90 & 92 \\
\hline W-EDSM & 0\% & 0\% & 0\% & 93 & 95 \\
\hline GA/n32/v1 & 42.2\% & 56.3\% & 7.8\% & 43 & 32 \\
\hline GA/n32/v2 & 62.5\% & 79.7\% & 18.8\% & 37 & 32 \\
\hline
\end{tabular}

Table 8.19: The performance of the GA when the highest scoring merges in the APTA do not contain a colour-compatible merge.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{No Loops in Target DFA (n32d607e64_NoLoops.sqlite)} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM & 6.3\% & 6.3\% & \(0 \%\) & 84 & 92 \\
\hline W-EDSM & 6.3\% & 6.3\% & \(0 \%\) & 89 & 96 \\
\hline GA/n32/v1 & 37.5\% & 45.3\% & 4.7\% & 46 & 36 \\
\hline GA/n32/v2 & 46.9\% & 62.5\% & 7.8\% & 41 & 32 \\
\hline
\end{tabular}

Table 8.20: The performance of the GA when the target DFA does not contain any loops.

\subsection*{8.4.2 Discussion and Observations}

The observations drawn from these experiments include:
- In Table 8.17, we observe that, when training sets are symmetrically structurally complete with respect to the target DFA, the GA considerably outperforms EDSM in all cases. The experimental likelihood of identifying low-error hypotheses improves as follows (showing the best-case GA configuration):
\[
\begin{aligned}
& \text { - 32-states: } \mathrm{EDSM} \mathbf{1 4 . 1 \%} \rightarrow \mathrm{GA} / \mathrm{n} 32 / \mathrm{v} 3 \mathbf{6 8 . 8 \%} \approx 4.7 \times \\
& \text { - } 64 \text {-states: } \mathrm{EDSM} \mathbf{1 2 . 5 \%} \rightarrow \mathrm{GA} / \mathrm{n} 64 / \mathrm{v} 5 \mathbf{5 6 . 3 \%} \approx 4.5 \times
\end{aligned}
\]
- Tuning the GA search parameters (at the cost of computational effort) highly affects the performance of the method. This can be seen when comparing the effectiveness of the GA/n64/v1 and GA/n64/v5 parameter sets.
- In the symmetrically structurally complete training set case, the likelihood of the GA finding the exact target increases considerably. For example, for 32 -state problems, EDSM manages to find the target in \(3.9 \%\) of the cases, while the GA manages in \(16.4 \%\) of the cases (GA/n32/v3). This represents an improvement of about \(4.2 \times\). A similar observation can be made for \(64-\) state target DFA problems.
- The fact that GA/n32/v2 performs noticeably better than GA/n32/v1 (best of two runs with the same parameters) is a strong indicator that the population sizes and maximum generations we have chosen in the GA/n32/v1 configuration are too conservative and restrict the search. This is confirmed by GA/n32/v3 which yields the best results after doubling the size of the initial population and the maximum number of generations allowed to run.
- Training sets which are not structurally complete also adversarially affect the performance of the GA. However, we see that the GA still performs much better than EDSM in these cases. In Table 8.18, we see that EDSM can identify low-error hypotheses in \(9.4 \%\) of the cases, while the GA can do so at a rate of \(56.3 \%(\approx 6 \times)\) when using the GA/n32/v2 configuration. These results are also better than what we have observed for the ensemble of heuristics and Delta Graph methods.
- Recall that when training sets are randomly constructed, there is a high likelihood that they are not symmetrically structurally complete (see Section 7.5.5). Because of this, and the fact that the GA performs much better than EDSM both when the training data is and is not symmetrically structurally complete, we can expect the GA to perform even better on unrestricted data sets. This is a similar observation to what we made when discussing the ensemble of heuristics and Delta Graph.
- As expected, when the training sets are not structurally complete, neither method can ever find the target (see Chapter 3).
- In Table 8.19, we see that, unlike for EDSM, the GA is not negatively affected in the adversarial case where none of the first rank merges do not contain
a colour-compatible merge. This can be attributed to the fact that the GA has the opportunity to discover several sequences of colour-compatible merges. A case for this has been made in the "Not all Colour-Compatible Sequences are Created Equal" discussion earlier. Furthermore, the sequences identified by the GA are longer (and thus establish more constraints) than those identified by the Delta Graph method.
- Likewise, the GA performs much better than EDSM when the target DFA does not contain any loops. In Table 8.20, we can see that while EDSM can find low-error hypotheses at a rate of \(6.3 \%\), the configuration GA/n32/v2 can do so \(46.9 \%\) of the time. Moreover, in \(7.8 \%\) of the same problem instances, the GA is able to find the exact target DFA while EDSM can never do so.
- An analysis of the convergence rate of GA/n32/v1 in 'n32d607e128_GA.sqlite' shows that \(\leq 1 \%\) error hypotheses can be found in as little as 5 generations and in an average of 23 generations. Recall, that the GA terminates when we either find a DFA which is exactly equal in size of the target or when the maximum number of generations have elapsed. Interestingly, in \(27 \%\) of the cases, GA/n32/v1 managed to find a hypothesis exactly equal in size to the target DFA but had an error greater than \(1 \%\). If our criterion for success is finding DFAs equal to the size of the target, this method is even more promising. Figure 8.7 shows the rates at which the GA converges for five different problem instances. Here we can see that the 32 -state target DFA problems P1 and P4 have converged to a fitness of zero (i.e. the GA identified a 32-state hypothesis), problems P2 and P5 converged prematurely, and P3 has been terminated before it had time to properly settle at some local minimum.
- The computational cost associated with the GA is dependent on the population size \(p\) and the maximum number of generations \(g\) allowed. In the worst case, we would need to perform \(p \times g\) fitness evaluations (i.e. W-EDSM extensions) until the algorithm terminates. On the hardware available to us, problem instances much larger than 64 states are impractical.


Figure 8.7: Fitness against generations for five problem instances showing different convergence scenarios.

\section*{Notes Regarding the 'Blind' Genetic Algorithm}

The blind implementation of the GA works as follows:
- We start with a population of \(i\) randomly created chromosomes.
- Two parents are randomly selected for mating (i.e. a fitness function is not used for selection). This process is illustrated in Figure 8.8
- The resulting offspring is mutated according to the mutation rate parameter specified.
- Unless the offspring already exists, it is added to the population (increasing its size).
- The selection, crossover, and mutation procedure is repeated until the population grows up to a final size \(f\).
- Each of the \(f\) chromosomes in the final population is extended with WEDSM and the smallest hypothesis is returned by the algorithm.


Figure 8.8: Deterministic tournament selection (i) vs. 'blind' selection (ii).

GA/n32/v1 has a population size of 100 chromosomes, and is allowed to run for at most 50 generations. This means, that throughout the lifetime of the GA, we would compute a maximum of \(100 \times 50=5000 \mathrm{~W}\)-EDSM extensions (fitness evaluations). Blind/n32/v1 has a final size of 5000 chromosomes meaning that it requires at least the same number of W-EDSM extensions as GA/n32/v1. The primary difference between the two methods is that the selection procedure of the 'normal' GA is directed by the fitness, whereas in the 'blind' GA, selection is random. The fact that the fitness-directed GA performs significantly better ( \(46.9 \%\) low-error rate, \(\mathrm{GA} / \mathrm{n} 32 / \mathrm{v} 1\), Table 8.17 ) than the blind variant ( \(0 \%\) lowerror rate) enables us to, experimentally, confirm that the fitness function chosen is indeed guiding our search efficiently. As a matter of fact, the blind GA performs no better than a random search in the space of valid merges. Nonetheless, we note that, while both the blind GA and a random search never find low-error hypotheses (as shown in Table 8.17), the mean/median sizes of the DFAs found by the blind GA (81/81 states) are much smaller than those found by a random search (776/776 states). This indicates that starting searches with high-reduction, non-zero EDSM-scoring merges \({ }^{6}\) biases the search towards smaller final DFAs.

\footnotetext{
\({ }^{6}\) Chromosomes are an initial sequence of merges drawn from the APTA reduction table.
}

\section*{Comparison to Other Evolutionary Approaches}

The evolutionary technique used by Dupont in Dup94 involves searching for an optimal partitioning of states of an initial hypothesis constructed from training examples. Dupont attempted to identify 15 languages, 7 of which are the so called Tomita languages [Tom82], while the remaining ones are languages having similar complexity (in terms of minimum DFA size). The results obtained were comparable to those using RPNI which was the state-of-the-art at the time. The largest target DFAs that were dealt with had 5 states and 8 transitions which are much smaller than the target DFAs we are considering in this dissertation. As such, comparing our method to Dupont's is not practical.

On the other hand, our results are most comparable to those obtained by Lucas and Reynolds in [LR05] since we both use exactly the same Abbadingo problem instance creation procedure, as well as a similar experimental protocol. When applied to noise-free problem instances of between 4 and 16 states, their method performed better than EDSM. Unfortunately, as the target DFA sizes started growing to close to 32 -state targets, their method was outperformed by EDSM even at a density of 3,275 training strings. In contrast, we have shown that our method considerably outperforms EDSM even on DFAs having up to 64 states \({ }^{7}\) and on much sparser training data (we 'only' used 607 training strings for 32 -state problems, and 1,521 training strings for 64 -state problems). Moreover, we have also shown that our method is less sensitive to setups which are highly adversarial to EDSM. We conclude this section by noting that Lucas and Reynolds' implementation can deal noisy training sets, whereas this requirement was not within the scope of our work

\subsection*{8.5 Ending Remarks}

In this chapter, we have evaluated the performance of each of the three methods we proposed on data sets consisting of thousands of Abbadingo-style problem

\footnotetext{
\({ }^{7}\) We have not tested our GA method on targets larger than 64 states only due to runtime performance concerns, and not because we expect that the generalisation performance would suffer.
}
instances. Our evaluation strategy was similar to that found in the literature (for example Lan99], [ACS04, and [LR05]). We have also created data sets which are adversarial to EDSM to determine how our methods perform in these cases. Using each of our three methods, we have observed appreciable improvements over EDSM using symmetrically structurally complete training sets, as well as on all the adversarial cases. We complete this summary with a discussion of the strengths and weaknesses of each method.

\section*{The Ensemble of Heuristics}
- Strengths:
- Considerably outperforms EDSM on symmetrically structurally complete training sets, as well as in the adversarial cases when either the training set is not structurally complete, or when the target DFA has no loop transitions from a state and itself.
- The bounds on computational effort are well understood (it is a linear function of the number of heuristics in the ensemble).
- We established that identifying new monotonic, greedy, heuristics is a promising research direction.
- Parallelisation is straightforward.
- Weaknesses:
- In the adversarial case when the highest EDSM-scoring merges (ties) in the APTA do not contain good, colour-compatible merges, the performance of the ensemble of heuristics is underwhelming.
- This ensemble method gives a notable improvement over EDSM but is weaker than the non-monotonic Delta Graph and GA methods (on the other hand, the runtime performance of the ensemble of heuristics is much better).
- Further notes:
- While the ensemble of heuristics outperforms EDSM and W-EDSM on all data sets of Abbadingo-style problem instances we experimented
with, the relative decrease in performance as the size of the target DFA increases needs to be further investigated.

\section*{The Delta Graph}
- Strengths:
- Considerably outperforms EDSM on symmetrically structurally complete training sets, as well as in all the three adversarial setups.
- The median size of the hypotheses identified by the Delta Graph is equal to or very close to that of the target DFA. The significance of this is that the Delta Graph is able to identify smaller DFAs than EDSM can.
- Parallelisation is straightforward.
- Weaknesses:
- For best performance, graph construction parameters need to be tuned for each class of problem. A complete grid-search for the best parameters is computationally expensive.
- While still being computationally feasible on consumer grade hardware, the Delta Graph requires more time to evaluate than a monotonic heuristic such as EDSM, or the ensemble of heuristics.
- Further notes:
- The Delta Graph method was partially inspired by the fact that getting the initial merges in a sequence correct (colour-compatible), increases the likelihood that extending the sequence using a label-matching heuristic will lead to smaller or lower error hypotheses. Our results show that we can obtain very good results in as few as three merges.
- We have experimentally shown that the Delta Graph contains several initial sequences of colour-compatible merges (whose length is the depth of the Delta Graph). In our "Not all Colour-Compatible Sequences are Created Equal" discussion, we have seen how this is advantageous, and results in an improvement beyond what we would expect should we
simply identify a single sequence of colour compatible-merges of the same length.
- While the Delta Graph outperforms EDSM and W-EDSM on all data sets of Abbadingo-style problem instances we experimented with, the relative decrease in performance as the size of the target DFA increases needs to be further investigated.

\section*{The Genetic Algorithm}
- Strengths:
- Considerably outperforms EDSM on symmetrically structurally complete training sets, as well as in all the three adversarial setups.
- The GA can, potentially, identify longer colour-compatible sequences than the Delta Graph can.
- Multiple runs (random restarts) improve performance.
- The median size of the hypotheses identified by the GA is equal to or very close to that of the target DFA. The significance of this is that the GA is able to identify smaller DFAs than EDSM can.
- In the 32 and 64-state target DFA cases we evaluated, the size of the search space is not affected by either the size of the target DFA we are looking for, or the size of the training set (the APTA reduction table sizes for suitable \(\alpha\) and \(\beta\) values are almost identical).
- Parallelisation is straightforward.
- Weaknesses:
- On consumer-grade hardware, the method does not scale to very large DFAs (more than 64 states) due to the computational cost of the fitness function.
- For best performance, GA parameters need to be tuned for each class of problem. Since the algorithm takes long to run, searching for the good parameters is very computationally expensive.
- The GA requires much more time to evaluate than any of the other methods we considered in this dissertation.
- Further notes:
- While the GA outperforms EDSM and W-EDSM on all data sets of Abbadingo-style problem instances we experimented with, the relative decrease in performance as the size of the target DFA increases needs to be further investigated.

\section*{Chapter 9}

\section*{Conclusions and Future Work}

One of the primary aims of the Abbadingo One competition designers was to promote the development of DFA learning algorithms that perform better than the state-of-the-art on larger DFAs and sparser training sets Abb97, LPP98. In this sense, their objective has been met with overwhelming success - the amount of work and contributions to the area since the competition had been announced is considerable, and has served as an inspiration to us.

A significant amount of our work was dedicated to taking a 'from first principles' approach to the DFA learning problem. This involved both understanding the behaviour of state merging algorithms, and also developing an extensive framework on which to implement and experiment with many existing algorithms as well as our own. Creating all this scaffolding from scratch is worthwhile for several reasons. It helps us to understand the minutiae of each learning algorithm, to finely control and optimise the performance of our data structures and algorithms, and to produce outputs for visualisation and examination in any format we wish (this is extremely useful for understanding the conditions under which learning algorithms succeed or fail). In this chapter, we conclude this dissertation by framing our achievements and contributions in terms of the research hypotheses we set out with. Notwithstanding the promising results we have achieved in the area, there is future work to be done. We complete our dissertation by presenting a number of ideas regarding the future directions our work can take.

\subsection*{9.1 Achievements and Contributions}

The primary aims of this dissertation were to:
- Find heuristics which are able to identify low or zero-error hypotheses on problem instances when EDSM does not, and
- Design non-monotonic, yet computationally feasible, DFA search algorithms, and determine the extent with which they improve on EDSM.

We feel that main aims of this dissertation have been met successfully. A number of heuristics, based on various combinations of EDSM and state reduction, were proposed. These heuristics represent different inductive biases, and are able to identify either the target DFA or a low-error hypothesis in cases when EDSM does not. Subsequently, we used these heuristics in an ensemble which outperforms EDSM. We have also developed two non-monotonic search strategies, namely the Delta Graph and the genetic algorithm, that also outperform EDSM to an even greater extent. While ensuring that our evaluation methodology is consistent with the ones found in the literature (such as Lan99], Cic02], and [Spi04), we have also considered several types of problem instances which adversarially affect EDSM.

The subtasks we presented in Section 1.3 have been also tackled. We have replicated the Abbadingo One procedure for creating problem instances, we have performed many experiments on large sets of problem instances to reliably support the approaches that we proposed, we implemented EDSM as a baseline for comparison and studied the conditions under which it succeeds or fails, and in Chapters 6 and 7, we have studied important characteristics of state merging algorithms and performed several experiments to better understand the problem at hand. These included studying the characteristics of target DFAs and training sets to determine which ones are adversarial to our learning task, observe state reduction rates along ideal merge paths to help us focus non-monotonic searches on a subspace of high reduction merges, determine the extent to which the first merges in a sequence are critical, and performing other statistical tests on problem instances and sequences of merges (such as correlating the identification of low-error hypotheses with the length of merge sequences).

\subsection*{9.1. \(\quad\) Summary of Results}

The results we obtained for each method, next to EDSM/W-EDSM as a baseline, are summarised in Tables 9.1, 9.2, and 9.3 for comparison. Here we show the outcomes for the best parameter configurations we have attempted for the Delta Graph and the GA.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{n32d607e1024.sqlite \(\dagger\), n32d607e128_GA.sqlite \(\ddagger\)} \\
\hline \multicolumn{6}{|l|}{32-State Target, 607 Strings, 1024 Experiments \(\dagger\), 128 Experiments \(\ddagger\)} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM \(\dagger\) & 15.7\% & 19.6\% & 4.3\% & 75 & 90 \\
\hline W-EDSM \(\dagger\) & 14.1\% & 17.78\% & 3.6\% & 77 & 92 \\
\hline Ensemble \(\dagger\) & 29.5\% & 31.5\% & 14\% & 66 & 82 \\
\hline Delta Graph \(\dagger\) & 50.7\% & 65.2\% & 14.4\% & 47 & 32 \\
\hline GA \(\ddagger\) & 68.8\% & 87.5\% & 16.4\% & 34 & 32 \\
\hline
\end{tabular}

Table 9.1: A summary of the results for 32 -state target problems.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{n64d1e1024.sqlite \(\dagger\), n64d607e64_GA.sqlite \(\ddagger\)} \\
\hline \multicolumn{6}{|l|}{64-State Target, 1,521 Strings, 1024 Experiments \(\dagger\), 64 Experiments \(\ddagger\)} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM \(\dagger\) & 15.2\% & 17.7\% & 0.6\% & 162 & 200 \\
\hline W-EDSM \(\dagger\) & 13.8\% & 16.5\% & 0.5\% & 170 & 207 \\
\hline Ensemble \(\dagger\) & 25.8\% & 25.9\% & 4.2\% & 147 & 193 \\
\hline Delta Graph \(\dagger\) & 42.9\% & 47\% & 2.2\% & 113 & 65 \\
\hline GA \(\ddagger\) & 56.3\% & 59.4\% & 4.7\% & 85.6 & 64 \\
\hline
\end{tabular}

Table 9.2: A summary of the results for 64 -state target problems.
\begin{tabular}{lccccc}
\hline & \multicolumn{5}{c}{ n128d1e512.sqlite } \\
& 128-State Target, & 4,382 Strings, \(\mathbf{5 1 2}\) & Experiments & \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline W-EDSM & \(22.7 \%\) & \(15.4 \%\) & \(0 \%\) & 377 & 511 \\
\hline Ensemble & \(30.7 \%\) & \(23.4 \%\) & \(1.4 \%\) & 340 & 521 \\
Delta Graph & \(49.4 \%\) & \(39.1 \%\) & \(0.4 \%\) & 275 & 134 \\
\hline
\end{tabular}

Table 9.3: A summary of the results for 128 -state target problems.

Tables 9.4 , 9.5, and 9.6 below show the performance of each method when used on
our three adversarial data sets. Again, we present our results next to EDSM/WEDSM as a baseline.
\begin{tabular}{lccccc}
\hline \multicolumn{6}{c}{ Non-Structurally Complete Training Sets (n32d607e64_NotStructComp.sqlite) } \\
& 32-State Target, \(\mathbf{6 0 7}\) Strings, \(\mathbf{6 4}\) Experiments \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline \hline EDSM & \(9.4 \%\) & \(14.1 \%\) & \(0 \%\) & 75 & 86 \\
W-EDSM & \(10.9 \%\) & \(15.6 \%\) & \(0 \%\) & 76 & 89 \\
\hline Ensemble & \(21.9 \%\) & \(25 \%\) & \(0 \%\) & 65 & 75 \\
Delta Graph & \(35.9 \%\) & \(62.5 \%\) & \(0 \%\) & 43 & 32 \\
Genetic Algorithm & \(56.3 \%\) & \(73.4 \%\) & \(0 \%\) & 36 & 32 \\
\hline
\end{tabular}

Table 9.4: A summary of the performance of our methods when training data is not symmetrically structurally complete.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{Immediately Failing Training Sets (n32d607e64_EdsmFailing.sqlite)} \\
\hline \multicolumn{6}{|c|}{32-State Target, 607 Strings, 64 Experiments} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM & \(0 \%\) & \(0 \%\) & \(0 \%\) & 90 & 92 \\
\hline W-EDSM & 0\% & 0\% & \(0 \%\) & 93 & 95 \\
\hline Ensemble & 1.6\% & 3.1\% & 1.6\% & 83 & 88 \\
\hline Delta Graph & 39.1\% & 54.7\% & 9.4\% & 57 & 32 \\
\hline Genetic Algorithm & 62.5\% & 79.7\% & 18.8\% & 37 & 32 \\
\hline
\end{tabular}

Table 9.5: A summary of the performance of our methods when the highest EDSMscoring merges in the APTA do not contain a colour-compatible merge.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{No Loops in Target DFA (n32d607e64_NoLoops.sqlite) 32-State Target, 607 Strings, 64 Experiments} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact Target & Mean Size & Median Size \\
\hline EDSM & 6.3\% & 6.3\% & \(0 \%\) & 84 & 92 \\
\hline W-EDSM & 6.3\% & 6.3\% & \(0 \%\) & 89 & 96 \\
\hline Ensemble & 15.6\% & 15.6\% & 7.8\% & 77 & 90 \\
\hline Delta Graph & 43.8\% & 60.9\% & 7.8\% & 49 & 32 \\
\hline Genetic Algorithm & 46.9\% & 62.5\% & 7.8\% & 41 & 32 \\
\hline
\end{tabular}

Table 9.6: A summary of the performance of our methods when the target DFA does not contain any loops.

\subsection*{9.1.2 An Ensemble of Heuristics}

During the experimentation phase of this dissertation, we established that various combinations of EDSM and reduction can be used to create an ensemble of monotonic, greedy heuristics which performs better than EDSM alone. Indeed, individual heuristics in the ensemble actually performed better than EDSM by themselves (over a large set of identical problem instances). Importantly, since the number of heuristics in the ensemble is fixed, the method scales well for larger target DFA sizes at their corresponding training set density. The ensemble of heuristics was also able to considerably improve over EDSM's generalisation rate in several adversarial scenarios. These results are promising, and support the search for other heuristics which are able to find low-error hypotheses on problem instances when others (such as EDSM) do not.

\subsection*{9.1.3 Adversarial Problem Instances}

When studying the characteristics of the structure of target DFAs and training sets, we have described a number of scenarios which are highly adversarial to EDSM. These arise when:
- The training set is not structurally complete with respect to the target DFA,
- The highest EDSM-scoring merges in the APTA (the first rank of ties) do not contain a colour-compatible merge, and
- The target DFA does not contain loop transitions from a state to itself.

We have also observed that, using the Abbadingo method to create problem instances, the likelihood of encountering these problematic situations is quite significant. For example, in Table A. 13 of Appendix A, we see that, on 64-state target DFA problem instances, there is a \(59 \%\) likelihood that a training set is not symmetrically structurally complete, a \(42 \%\) likelihood that there is no colourcompatible merge in the first EDSM rank, and a \(20 \%\) likelihood that the target DFA does not contain loop transitions from a state to itself. Each of the three methods we proposed in this dissertation is able to deal with these adversarial scenarios better than EDSM.

\subsection*{9.1.4 Label-Matching Heuristics}

It well understood that the performance of a label-matching heuristic such as EDSM decreases as the training data becomes sparser [LPP98, Spi04]. In our work, we have elaborated on this idea by determining to which extent tie-breaking is critical (using the EDSM-TieCC heuristic in Section 7.5.3). Inspired by the work in Cic02, we further studied the significance and impact getting the initial sequence of merges right. These observations supported the development of our ensemble, Delta Graph, and genetic algorithm which perform better on sparse training data.

\subsection*{9.1.5 Colour-Compatible Merges}

We have used colour-compatible merges as a tool for defining what 'good' merges are. In Chapter 7, this allowed us to pinpoint exactly where mistakes are being made along a merge path (and study them), and where these colour-compatible merges are expected to be found in our pool of candidate merges. Moreover, Oracle-assisted heuristics, such as FullCol-EDSM, allow us to analyse the characteristics of good merge sequences which lead to the exact target DFAs and understand how merges are selected. Our Delta Graph and GA methods were both inspired by the results we have obtained from these experiments.

\subsection*{9.1.6 The APTA Reduction Table}

The analysis of the rate of reduction in states along ideal, FullCol-EDSM, paths allows us to exclude merges from our search space which deviate from the expected rate of reduction. More specifically, our experiments show that the initial merges in a path are almost never low reduction ones. Initially ignoring such low reduction merges (of which there are overwhelmingly many) results in a pruning of the search space which, nonetheless, has been shown to contain many colour-compatible merges. The Delta Graph and GA methods we have developed would not be practical if we had to consider the entire space of merges as candidates. Moreover, studying the contents of the reduction table in terms of the colour-compatible merges it contains, allowed us to estimate where these
colour-compatible merges exist in the reduction table. This allows us to explore beyond the first EDSM scoring rank. These results were also used to determine the branching factor of the Delta Graph.

\subsection*{9.1.7 Getting the First Merges Right}

Getting the initial merges in a path right is critical to reaching a low-error hypothesis or the actual target DFA (see Cic02, and Chapter 7). In this dissertation, we have proposed the Delta Graph and GA methods to find such initial sequences. The performance we obtained using both methods is considerably better than what is possible using EDSM, SAGE, or Ed-Beam (results are shown in the discussion of the Delta Graph in Section 8.3.2, as well as in Lan98). Moreover, both methods are highly resilient to problem configurations which are adversarial to EDSM.

\subsection*{9.1.8 The Genetic Algorithm}

In the literature, we have identified several attempts which use genetic algorithms to find smallest consistent DFAs Dup94, LR05, Wie17. These methods involve evolving a representation of the target DFA by either 'packing' the states of the APTA into an \(n\)-block partition corresponding to a quotient of the target automaton (where \(n\) is the size of the target DFA), or by constructing the transition matrix of the target DFA. Based on the observation that the initial merges in a sequence greatly affect the quality (in terms of size) of the hypothesis identified, we have developed a technique which evolves short initial sequences of high-quality merges. When these short sequences are extended, we observe a higher likelihood of identifying the target DFA or a close approximation of it. Compared with the results reported by previous attempts at using genetic algorithms on similar Abbadingo-style problem instances, our method performs and scales (in terms of target DFA and training set size) considerably better.

\subsection*{9.2 Future Work}

Research in regular inference is ongoing and its application to real-world problems is far-reaching \({ }^{1}\). Consequently, our search for efficient monotonic, greedy heuristics is ongoing with promise shown by our ensemble of heuristics method. Additionally, while the augmentation of such heuristics with a more exhaustive, non-monotonic search yields noticeably better results, the computational cost in doing so is substantial. Based on the experiments, methods, and results we have obtained in our work, we have determined the following areas which could result in further important advancements in the field:
- Implementations of the state merging operator are computationally expensive. It then follows that sets of experiments would have to be smaller, experimenting on larger DFAs becomes harder, and broader searches become impractical. At every step along a merge path, a large number of merges need to be attempted to determine their validity and their score. Whenever a merge is executed, the hypothesis modified by that merge needs to be rolled back to its original state before another one of the candidates is attempted. Efficient implementations of the state merging operator, typically, employ a book-keeping strategy to track changes and undo them before the next merge is tried. This necessitates a significant amount of memory allocations and writes. A low-level profiling of the costs associated with constructing merge paths shows us that the majority of the effort is taken up by these memory operations. While our implementation takes this into account, it can be improved further on at least two fronts. Firstly, the outcomes of a large number merges which are being evaluated at a current step along a merge path remain unchanged during the next step. Consider two states being merged: if the two subgraphs rooted at them do not change after some other merge has been performed, their previous evaluation will not be affected and can be carried forward. Secondly, it is much cheaper to perform one large memory allocation and read/write operation rather than making several smaller ones. Our implementation preallocates working

\footnotetext{
\({ }^{1}\) We have covered several real world applications of grammatical inference in Section 1.3
}
memory for merges so allocation is not an issue. However, tracking changes and undoing them is still required. During experimentation, we notice that changes are fragmented throughout the preallocated memory block which requires scanning through the modifications and copying many small portions of memory. We are considering a more efficient data structure to represent partitions which minimises this fragmentation and allows for a single, larger memory copy rather than many small ones.
- In many cases, parallelisation of merge path construction is trivial and, indeed, all of our implementations are concurrent. Unfortunately, since memory is typically a shared resource on a single machine, the overwhelming overhead associated with allocation, and reading and writing, hinders scaleability due to this bottleneck. In addition to the strategies described previously, the distribution of computational effort across networks of possibly lowerpowered machines is compelling (rather than relying on single memory, and multiple faster cores/CPUs).
- While the parameters we have chosen to construct Delta Graphs and run our GAs with give us promising results, they may be suboptimal. A grid search through the parameter space is impractical due to the large number of experiments which would need to be run in order to identify them. In this dissertation, we have performed several statistical tests to determine these parameters and identify, in expectation, which ones would perform best (e.g. colour-compatible merge positions, APTA reduction table analysis, etc...). Nonetheless, further statistical testing and analysis is almost certain to lead to more focused search spaces which are both smaller, as well as contain more good merges.
- When constructing deeper Delta Graphs, we have observed that the cumulative EDSM score of a leaf hypothesis in the graph is a very strong indicator of whether that hypothesis is good (colour-compatible) or not. This would allow us to order leaf extension by the most promising ones first and improve running time. Unfortunately, this property has been found to be true at graph depths which are beyond practical (but still considerably shallower
than the depth of the entire search space). For instance, at the Delta Graph depths of 3 and 4 which we have attempted, this correlation breaks down. We are still actively searching for an alternative proxy for leaf/hypothesis quality.
- Due to practical considerations, especially due to the computational cost associated with the merge operation, we have not experimented with sufficiently large data sets for target DFAs having 256 states and more. The behaviour of our methods on such targets and large data sets needs addressing and may be possible by implementing the ideas we have discussed previously.
- Our comparison to TBW-EDSM and Ed-Beam has been made with respect to the results published by the respective authors. A more compelling analysis can be made by implementing the actual algorithms and evaluating their performance against the exact problem instances we have used ourselves.
- The results we obtained from the ensemble of heuristics methods clearly show that there is scope for discovering new heuristics which are able to find lowerror hypotheses when others such as EDSM do not. We are currently in the preliminary stages of designing genetic programs to evolve these heuristics similarly to how they have been applied to SAT solving [BEDP08] and bin packing [BHK06]. This technique has merit because the inference problems we are dealing with have all the hallmarks PLM08] of those where genetic programs have performed well: (i) we can generate significant amounts of test data, (ii) there is a straightforward way to measure the performance of a hypothesis, (iii) small improvements can be measured (making evolutionary based techniques possible), and (iv) approximate solutions are acceptable.
- A detailed investigation of merge relation graphs (blocking, inclusion, and labelling), discussed in Chapter 3, can help us understand how merges interact with each other and which ones carry high risk. For example, detecting which and when merges will result in orphaned states (states which will remain unlabelled in the final hypothesis) is exact and straightforward. Unfortunately, a preliminary investigation indicates, that, in the vast majority
of the cases, such orphaning happens very late in the merging process making a backtracking strategy infeasible as it would be too deep. Rather than discarding merges when we definitely know that they would result in orphaned states (when it is too late), we are investigating an entropy-based metric to determine the likelihood of a merge becoming so, and postpone it due to the risk involved. A similar idea has been explored by Coste and Nicholas in CN98. The relationships and constraints in the merge relation graph can also be used to develop a constraint satisfaction system similar to the work done by Oliveira and Silva in OMS98.
- In 1998, shortly after the Abbadingo One competition was completed, the Gowachin [Gow98] server was made available to provide on-demand benchmark problems for DFA learning algorithms \({ }^{2}\). In addition to allowing users to request test problems for specific target DFA sizes and training set densities, Gowachin also allows for a degree of noise to be injected in the training data. While in this dissertation, we have not considered noisy training sets, algorithms such Habrard et al.'s ALERGIA HBS03] may serve as a starting point for extending our algorithms to support such scenarios.
- Inspired by Heule and Verwer's work on the StaMinA competition-winning algorithm, we are investigating the use of our methods to learning DFAs having higher cardinality alphabets. Our Delta Graph and GA methods are able (with high experimental likelihood) to identify good short initial sequences of merges which can be used to compact the initial APTA as a preprocessing step of an algorithm such as dfasat [HV13]. Likewise, we can also experiment with the candidate hypotheses returned by the ensemble of heuristics, Delta Graph, and genetic algorithm methods by treating them as an ensemble of automata [Die00] to classify unseen strings.

\footnotetext{
\({ }^{2}\) At the time of writing, the Gowachin service is no longer running.
}

\subsection*{9.3 Concluding Remarks}

The grammatical inference community has been active for decades, and interest in the area does not appear to be slowing down. This is evidenced by the number of learning competitions that have been made public since Abbadingo One in 1997, as well as the consistent output of researchers both in the International Conference on Grammatical Inference (ICGI) and elsewhere.

Although a shift in focus from purely Abbadingo-style problems in the direction of practical applications of grammatical inference, studying other classes of languages, and other types of problems is apparent, we felt that we could still contribute more to the area. We have done this by taking a 'from first principles' approach by performing an extensive, experimental analysis on the behaviour of state merging on Abbadingo-style problems. This enabled us to develop a number of methods which outperform both EDSM as well as EDSM + Search methods that are considered to be the state-of-the-art in DFA learning. In this sense, we feel that we have successfully achieved our original goals. Of course, during the course of our work, we have also raised more questions than we have answered. While our work in the area of DFA learning is ongoing, it is our sincere hope that the findings made in this dissertation will serve to inspire others.

\section*{Bibliography}
[Abb97] Abbadingo One: DFA Learning Competition. http://abbadingo. cs.nuim.ie, 1997. Accessed: 2017-05-19.
[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining Specifications. In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, page 4-16, New York, NY, USA, 2002. Association for Computing Machinery.
[ACS04] John Abela, François Coste, and Sandro Spina. Mutually Compatible and Incompatible Merges for the Search of the Smallest Consistent DFA. In International Colloquium on Grammatical Inference, pages 28-39. Springer, 2004.
[Agg18] C.C. Aggarwal. Neural Networks and Deep Learning: A Textbook. Springer International Publishing, 2018.
[AJ06] Pieter Adriaans and Ceriel Jacobs. Using MDL for Grammar Induction. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, Grammatical Inference: Algorithms and Applications, pages 293-306, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
[Alq97] Rene Alquezar. Symbolic and Connectionist Learning Techniques for Grammatical Inference. PhD thesis, Universitat Politécnica de Catalunya, 1997.
[Ang78] D. Angluin. On the Complexity of Minimum Inference of Regular Sets. Inform. Control, 39(3):337-350, 1978.
[AS94a] R. Alquezar and A. Sanfeliu. A Hybrid Connectionist-Symbolic Approach to Regular Grammatical Inference Based on Neural Learning and Hierarchical Clustering, pages 203-211. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.
[AS94b] R. Alquezar and A. Sanfeliu. Incremental Grammatical Inference From Positive And Negative Data Using Unbiased Finite State Automata. In In Proceedings of the ACL'02 Workshop on Unsupervised Lexical Acquisition, pages 291-300, 1994.
[AV94] D. Aldous and U. Vazirani. "Go with the winners" Algorithms. In Proceedings 35th Annual Symposium on Foundations of Computer Science, pages 492-501, Nov 1994.
\(\left[\mathrm{BBH}^{+} 09\right]\) A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.
[BBP75] A. W. Biermann, R. I. Baum, and F. E. Petry. Speeding up the Synthesis of Programs from Traces. IEEE Transactions on Computers, C-24(2):122-136, Feb 1975.
[BEDP08] Mohamed Bader-El-Den and Riccardo Poli. Generating SAT Localsearch Heuristics Using a GP Hyper-heuristic Framework. In Proceedings of the Evolution Artificielle, 8th International Conference on Artificial Evolution, EA'07, pages 37-49, Berlin, Heidelberg, 2008. Springer-Verlag.
\(\left[\mathrm{BEL}^{+} 17\right]\) Borja Balle, Rémi Eyraud, Franco M. Luque, Ariadna Quattoni, and Sicco Verwer. Results of the Sequence PredIction ChallengE (SPiCe): a Competition on Learning the Next Symbol in a Sequence. In Sicco Verwer, Menno van Zaanen, and Rick Smetsers, editors, Proceedings of The 13th International Conference on Grammatical Inference, volume 57 of Proceedings of Machine Learning

Research, pages 132-136, Delft, The Netherlands, 05-07 Oct 2017. PMLR.
[BF72] A. W. Biermann and J. A. Feldman. On the Synthesis of FiniteState Machines from Samples of Their Behavior. IEEE Transactions on Computers, C-21(6):592-597, June 1972.
[BHK06] E. K. Burke, M. R. Hyde, and G. Kendall. Evolving Bin Packing Heuristics with Genetic Programming. In Proceedings of the 9th International Conference on Parallel Problem Solving from Nature, PPSN'06, pages 860-869, Berlin, Heidelberg, 2006. Springer-Verlag.
[BL00] Jose Borges and Mark Levene. Data Mining of User Navigation Patterns. In Web Usage Analysis and User Profiling, pages 92-112. Springer, 2000.
[BL05] Josh Bongard and Hod Lipson. Active Coevolutionary Learning of Deterministic Finite Automata. J. Mach. Learn. Res., 6:1651-1678, December 2005.
[BO05] Miguel Bugalho and Arlindo Oliveira. Inference of Regular Languages using State Merging Algorithms with Search. Pattern Recognition, 38:1457-1467, 092005.
[Bré79] Daniel Brélaz. New Methods to Color the Vertices of a Graph. Commun. ACM, 22(4):251-256, April 1979.
[BS90] H. Bunke and A. Sanfeliu. Syntactic and Structural Pattern Recognition: Theory and Applications. World Scientific Pub Co Inc, 1990.
[BT97] Tobias Blickle and Lothar Thiele. A Comparison of Selection Schemes used in Evolutionary Algorithms. Evolutionary Computation, 4:361-394, 1997.
[CdlHJ10] David Combe, Colin de la Higuera, and Jean-Christophe Janodet. Zulu: An Interactive Learning Competition. In Anssi Yli-Jyrä, András Kornai, Jacques Sakarovitch, and Bruce Watson, editors,

Finite-State Methods and Natural Language Processing, pages 139146, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
[CDM91] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed Optimization by Ant Colonies. 011991.
[CF03] François Coste and Daniel Fredouille. What is the Search Space for the Inference of Non Deterministic, Unambiguous and Deterministic Automata? Research Report RR-4907, INRIA, 2003.
[CFV12] Christophe Costa Florêncio and Sicco Verwer. Regular inference as vertex coloring. In Nader H. Bshouty, Gilles Stoltz, Nicolas Vayatis, and Thomas Zeugmann, editors, Algorithmic Learning Theory, pages 81-95, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
[Che92] Pang C. Chen. Heuristic Sampling: A Method for Predicting the Performance of Tree Searching Programs. SIAM J. Comput., 21(2):295-315, April 1992.
[Cho56] Noam Chomsky. Three Models for the Description of Language. IRE Transactions on Information Theory, 2:113-124, 1956.
[Cic02] Orlando Cicchello. A New Limited Search Approach to Learning Abbadingo-Style Finite State Automata. Master's thesis, The Faculty of Graduate Studies, University of Guelph, 2002.
[CK02] Orlando Cicchello and Stefan C. Kremer. Beyond EDSM. In Pieter W. Adriaans, Henning Fernau, and Menno van Zaanen, editors, ICGI, volume 2484 of Lecture Notes in Computer Science, pages 37-48. Springer, 2002.
[CK03] Orlando Cicchello and Stefan C. Kremer. Inducing Grammars from Sparse Data Sets: A Survey of Algorithms and Results. J. Mach. Learn. Res., 4:603-632, December 2003.
[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, 3rd edition, 2009.
[CN97] François Coste and Jacques Nicolas. Regular Inference as a Graph Coloring Problem. In In Workshop on Grammar Inference, Automata Induction, and Language Acquisition (ICML' 97, pages 9-7, 1997.
[CN98] François Coste and Jacques Nicolas. Inference of Finite Automata: Reducing the Search Space with an Ordering of Pairs of States. In Claire Nédellec and Céline Rouveirol, editors, Machine Learning: ECML-98, pages 37-42, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
[CRML73] S. Crespi-Reghizzi, M.A. Melkanoff, and L. Lichten. The Use of Grammatical Inference for Designing Programming Languages. Communications of the ACM, 16(2):83-90, 1973.
[CSSM89] Axel Cleeremans, David Servan-Schreiber, and James L. McClelland. Finite State Automata and Simple Recurrent Networks. Neural Comput., 1(3):372-381, September 1989.
[CU12] Daniil Chivilikhin and Vladimir Ulyantsev. Learning Finite-State Machines with Ant Colony Optimization. In Marco Dorigo, Mauro Birattari, Christian Blum, Anders Lyhne Christensen, Andries P. Engelbrecht, Roderich Groß, and Thomas Stützle, editors, Swarm Intelligence, pages 268-275, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
[CV11] B. Cloteaux and L. Valentin. Counting the Leaves of Trees. Congressus Numerantium, 207:129-139, 2011.
[Die00] Thomas G. Dietterich. Ensemble methods in machine learning. In Multiple Classifier Systems, pages 1-15, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
[DJA08] A. Dubey, P. Jalote, and S. K. Aggarwal. Learning Context-free Grammar Rules from a Set of Programs. IET Software, 2(3):223240, June 2008.
[dlH05] Colin de la Higuera. A Bibliographical Study of Grammatical Inference. Pattern Recogn., 38(9):1332-1348, September 2005.
[dlH10] Colin de la Higuera. Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, New York, NY, USA, 2010.
[DMV94] P. Dupont, L. Miclet, and E. Vidal. What is the Search Space of the Regular Inference? In Rafael C. Carrasco and Jose Oncina, editors, Grammatical Inference and Applications, volume 862 of Lecture Notes in Computer Science, pages 25-37. Springer Berlin Heidelberg, 1994.
[Dup94] Pierre Dupont. Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG Method, pages 236245. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.
[Dup96] Pierre Dupont. Incremental Regular Inference. In Proceedings of the Third ICGI-96, pages 222-237. Springer, 1996.
[EFT96] H.D. Ebbinghaus, J. Flum, and W. Thomas. Mathematical Logic. Undergraduate Texts in Mathematics. Springer New York, 1996.
[FB75] King-Sun Fu and Taylor L. Booth. Grammatical Inference: Introduction and Survey. IEEE Transactions on SMC, 5:409-423, 1975.
[FK05] David Furcy and Sven Koenig. Limited Discrepancy Beam Search. In Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI'05, pages 125-131, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.
[For64] G. E. Forsythe. Algorithms. Commun. ACM, 7(6):347-349, June 1964.
[GEC04] GECCO: Learning DFAs from Noisy Samples. http://cswww. essex.ac.uk/staff/sml/gecco/NoisyDFA.html, 2004.
[GJ79] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman \& Co., New York, NY, USA, 1979.
[GLVdP12] Pedro García, Damián López, and Manuel Vázquez de Parga. Polynomial Characteristic Sets for DFA Identification. Theoretical Computer Science, 448:41-46, 2012.
[GoL] The Go Programming Language. https://golang.org. Accessed: 2019-06-06.
[Gol67] E. Mark Gold. Language Identification in the Limit. Information and Control, 10(5):447-474, 1967.
[Gol78] E. Mark Gold. Complexity of Automaton Identification from Given Data. Inf. Control., 37(3):302-320, 1978.
[Gol89] David E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.
[Gow98] The Gowachin DFA Learning Benchmark. http://www.irisa.fr/ Gowachin/, 1998. Accessed: 2017-05-19.
[Grü04] Peter Grünwald. A Tutorial Introduction to the Minimum Description Length Principle. ArXiv, math.ST/0406077, 2004.
[GSC \(\left.{ }^{+} 90\right]\) C Lee Giles, Guo-Zheng Sun, Hsing-Hen Chen, Yee-Chun Lee, and Dong Chen. Higher Order Recurrent Networks and Grammatical Inference. In Advances in Neural Information Processing Systems, pages 380-387, 1990.
[GSVG90] Pedro García, Encarna Segarra, Enrique Vidal, and Isabel Galiano. On the use of the Morphic Generator Grammatical Inference (MGGI) Methodology in Automatic Speech Recognition. International Journal of Pattern Recognition and Artificial Intelligence, 04(04):667-685, 1990.
[GVdPLR10] Pedro García, Manuel Vázquez de Parga, Damián López, and José Ruiz. Learning Automata Teams. In José M. Sempere and Pedro García, editors, Grammatical Inference: Theoretical Results and Applications, pages 52-65, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
[Hal60] Paul Halmos. Naive Set Theory. Van Nostrand, 1960. Reprinted by Springer-Verlag, Undergraduate Texts in Mathematics, 1974.
[HBS03] Amaury Habrard, Marc Bernard, and Marc Sebban. Improvement of the State Merging Rule on Noisy Data in Probabilistic Grammatical Inference. In Proceedings of the 14th European Conference on Machine Learning, ECML'03, pages 169-180, Berlin, Heidelberg, 2003. Springer-Verlag.
[HDB96] Martin T. Hagan, Howard B. Demuth, and Mark Beale. Neural Network Design. PWS Publishing Co., Boston, MA, USA, 1996.
[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages and Computation. Pearson Addison-Wesley, Upper Saddle River, NJ, 3 edition, 2007.
[HV10] Marijn J. H. Heule and Sicco Verwer. Exact DFA Identification Using SAT Solvers, pages 66-79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
[HV13] Marijn J. H. Heule and Sicco Verwer. Software Model Synthesis using Satisfiability Solvers. Empirical Software Engineering, 18(4):825-856, 2013.
[JP98a] Hugues Juillé and Jordan B. Pollack. A Sampling-based Heuristic for Tree Search Applied to Grammar Induction. In Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI '98/IAAI '98, pages 776-783, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.
[JP98b] Hugues Juillé and Jordan B. Pollack. A Stochastic Search Approach to Grammar Induction. In ICGI, pages 126-137. Springer Verlag, 1998.
[Kel95] Dean Kelley. Automata and Formal Languages: An Introduction. Prentice Hall, 1995.
[Knu74] Donald E. Knuth. Estimating the Efficiency of Backtrack Programs. Technical report, Stanford, CA, USA, 1974.
[Lan98] Kevin J. Lang. Evidence Driven State Merging with Search, 1998.
[Lan99] Kevin J. Lang. Faster Algorithms for Finding Minimal Consistent DFAs. Technical report, 1999.
[LDD08] Bernard Lambeau, Christophe Damas, and Pierre Dupont. StateMerging DFA Induction Algorithms with Mandatory Merge Constraints, pages 139-153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
[LG16] Damián López and Pedro García. On the Inference of Finite State Automata from Positive and Negative Data, pages 73-112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
[Lin01] Peter Linz. An Introduction to Formal Languages and Automata. Jones \& Bartlett Publishers, 2001.
[LPP98] Kevin Lang, Barak Pearlmutter, and Rodney Price. Results of the Abbadingo One DFA Learning Competition and a New EvidenceDriven State Merging Algorithm, 1998.
[LR03] Simon M. Lucas and T. Jeff Reynolds. Learning DFA: Evolution Versus Evidence Driven State Merging. In Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2003, 8 - 12 December 2003, Canberra, Australia, pages 351-358, 2003.
[LR05] S. M. Lucas and T. J. Reynolds. Learning Deterministic Finite Automata with a Smart State Labeling Evolutionary Algorithm. 2005.
[Mar09] Victor W. Marek. Introduction to Mathematics of Satisfiability. Chapman \& Hall/CRC, 1st edition, 2009.
[Mic86] Laurent Miclet. Structural Methods in Pattern Recognition. Springer-Verlag, 1986.
[OE96] Arlindo L. Oliveira and Stephen Edwards. Limits of Exact Algorithms for Inference of Minimum Size Finite State Machines. In Setsuo Arikawa and Arun K. Sharma, editors, Algorithmic Learning Theory, pages 59-66, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
[OG92] Jose Oncina and Pedro García. Identifying Regular Languages In Polynomial Time. In Advances in Structural and Syntactic Pattern Recognition, Volume 5 of the Series in Machine Perception and Artificial Intelligence, pages 99-108. World Scientific, 1992.
[OG02] Jose Oncina and Pedro García. A Polynomial Algorithm To Infer Regular Languages. 082002.
[OMS98] Arlindo L. Oliveira and Joao P. Marques-Silva. Efficient Search Techniques for the Inference of Minimum Size Finite Automata. In In Proceedings of the 1998 South American Symposium on String Processing and Information Retrieval, Santa Cruz de La Sierra, pages 81-89. IEEE Computer Society Press, 1998.
[PAu12] Probabilistic Automata Learning Competition. http://ai. cs.umbc.edu/icgi2012/challenge/Pautomac/, 2012. Accessed: 2019-10-10.
[PCD19] Mateusz Pyzik, François Coste, and Witold Dyrka. How to measure the topological quality of protein parse trees? In Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek, editors, Proceedings of

The 14th International Conference on Grammatical Inference 2018, volume 93 of Proceedings of Machine Learning Research, pages 118138. PMLR, feb 2019.
[Pic] PicoSAT Solver. http://fmv.jku.at/picosat/. Accessed: 2017-05-19.
[Pit89] Leonard Pitt. Inductive Inference, DFAs, and Computational Complexity. In Proceedings of the International Workshop on Analogical and Inductive Inference, AII '89, pages 18-44, London, UK, UK, 1989. Springer-Verlag.
[PLM08] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd, 2008.
[Pur78] Paul W Purdom. Tree Size by Partial Backtracking. SIAM Journal on Computing, 7(4):481-491, 1978.
[Rie99] A. Rieger. Inferring Probabilistic Automata from Sensor Data for Robot Navigation. Forschungsberichte des Lehrstuhls VIII, Fachbereich Informatik der Universität Dortmund. 1999.
[RN03] Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Pearson Education, 2 edition, 2003.
[RS93] R.L. Rivest and R.E. Schapire. Inference of Finite Automata Using Homing Sequences. Information and Computation, 103(2):299 347, 1993.
[Rum13] Wheeler Ruml. Greedy search, with an example. https://www. youtube.com/watch?v=9Vx0m-YiCWg, 2013. Accessed: 2019-12-30.
[SB07] Hooman Shayani and Peter J. Bentley. A More Bio-plausible Approach to the Evolutionary Inference of Finite State Machines. In Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO '07, pages 2937-2944, New York, NY, USA, 2007. ACM.
[SBH \(\left.{ }^{+} 94\right]\) Yasubumi Sakakibara, Michael Brown, Richard Hughey, I. Saira Mian, Kimmen Sjölander, Rebecca C. Underwood, and David Haussler. Stochastic Context-Free Grammars for tRNA modeling. \(N u-\) cleic Acids Research, 22:5112-5120, 1994.
[SC14] Andrew Stevenson and James R. Cordy. A Survey of Grammatical Inference in Software Engineering. Sci. Comput. Program., 96(P4):444-459, December 2014.
[Spi04] Sandro Spina. Merge Interactions - A New Heuristic for Automata Learning. Master's thesis, Department of Computer Science and A.I., Faculty of Science, University of Malta, 2004.
[SPI16] The Sequence PredictIction ChallengE (SPiCe). https://spice. lis-lab.fr, 2016. Accessed: 2019-10-10.
[SQL] SQLite. https://www.sqlite.org. Accessed: 2019-12-30.
[SZE06] Bradford Starkie, Menno Zaanen, and Dominique Estival. The Tenjinno Machine Translation Competition. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, Grammatical Inference: Algorithms and Applications, pages 214-226. Springer Berlin Heidelberg, 092006.
[TB73] B. A. Trakhtenbrot and Ya. M. Barzdin. Finite Automata, Behavior and Synthesis. North Holland, Amsterdam, 1973.
[TDH00] Franck Thollard, Pierre Dupont, and Colin de la Higuera. Probabilistic DFA Inference Using Kullback-Leibler Divergence and Minimality. In Proceedings of the Seventeenth International Conference on Machine Learning, ICML '00, pages 975-982, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
[TE11] Fedor Tsarev and Kirill Egorov. Finite State Machine Induction Using Genetic Algorithm Based on Testing and Model Checking. In Proceedings of the 13th Annual Conference Companion on Genetic
and Evolutionary Computation, GECCO '11, page 759-762, New York, NY, USA, 2011. Association for Computing Machinery.
[Tom82] M. Tomita. Dynamic Construction of Finite Automata from Examples using Hill-Climbing. In Proceedings of the Fourth Annual Conference of the Cognitive Science Society, pages 105-108, Ann Arbor, Michigan, 1982.
[Tsa10] F. Tsarev. Method of Finite-state Machine Induction from Tests with Genetic Programming. pages 31-36, 2010.
[VdWW12] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. Efficiently Identifying Deterministic Real-time Automata from Labeled Data. Machine Learning, 86(3):295-333, 2012.
[VEdlH14] Sicco Verwer, Rémi Eyraud, and Colin de la Higuera. PAutomaC: a Probabilistic Automata and Hidden Markov Models Learning Competition. Machine Learning, 96(1):129-154, 2014.
\(\left[\mathrm{WBD}^{+} 10\right]\) Neil Walkinshaw, Kirill Bogdanov, Christophe Damas, Bernard Lambeau, and Pierre Dupont. A framework for the competitive evaluation of model inference techniques. In Proceedings of the First International Workshop on Model Inference In Testing, MIIT '10, page 1-9, New York, NY, USA, 2010. Association for Computing Machinery.
[Wie17] Wojciech Wieczorek. Grammatical Inference - Algorithms, Routines and Applications, volume 673 of Studies in Computational Intelligence. Springer, 2017.
[WK91] Raymond L. Watrous and Gary M. Kuhn. Induction of Finite-state Automata Using Second-order Recurrent Networks. In Proceedings of the 4 th International Conference on Neural Information Processing Systems, NIPS'91, pages 309-316, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
[WM96] David Wolpert and William Macready. No Free Lunch Theorems for Search. 1996.
[WM97] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Optimization. Trans. Evol. Comp, 1(1):67-82, April 1997.
[Wol15] Mark A. Wolters. A Genetic Algorithm for Selection of Fixed-Size Subsets with Application to Design Problems. Journal of Statistical Software, 68(c01), 2015.
[WRS \(\left.{ }^{+} 99\right]\) Jason Tsong-Li Wang, Steve Rozen, Bruce A. Shapiro, Dennis E. Shasha, Zhiyuan Wang, and Maisheng Yin. New Techniques for DNA Sequence Classification. Journal of Computational Biology, 6(2):209-218, 1999.
[YH98] Jihoon Yang and Vasant G. Honavar. Feature Subset Selection Using a Genetic Algorithm. IEEE Intelligent Systems, 13(2):44-49, March 1998.
[ZGS93] Zheng Zeng, Rodney M. Goodman, and Padhraic Smyth. Learning Finite State Machines With Self-Clustering Recurrent Networks. Neural Computation, 5(6):976-990, 1993.
[ZH05] Rong Zhou and Eric A. Hansen. Beam-Stack Search: Integrating Backtracking with Beam Search, 2005.

\section*{Appendix A}

\section*{Complete Results}

In this appendix, we aggregate the results across several thousands of problem instances for 32,64 , and 128 -state target DFAs at a training set density of 1 . Running the genetic algorithm on 128 -state target DFAs and larger is impractical, so GA results are only available for target DFAs having 32 and 64 states. Adversarial setups for 32 -state target DFA problems are also shown here. For quick reference, the captions of the subsections and tables in this appendix correspond to the SQLite file names included in the media accompanying this dissertation.

General notes regarding the problem instances in this appendix:
- Abbadingo One does not specify density values for 32 -state problems (the number of strings in the training set). To obtain this, we experimentally identify the percentage of the total \(16 n^{2}-1\) strings which would give roughly the same EDSM performance as that for 64 -state problems at density 1 . This corresponds to 607 training strings.
- Per the Abbadingo method, we may request a target of size \(n\) but get one close to \(n\) instead. All target DFAs in our experiments will have exactly the number of states \(n\) requested. The Abbadingo construction depth requirement of \(\left(2 \log _{2} n-2\right)\) will always be honoured.
- All training sets will be symmetrically structurally complete with respect to the target. Cases when the training set is not symmetrically structurally complete are tabulated separately.
- Unless otherwise specified, the proportion of positive and negative strings in any training set will not be allowed to differ by more than \(20 \%\). This will avoid pathological cases where a training set would be overwhelmed by strings of a single class.
- All training sets are at Abbadingo density 1.
- Testings sets will consist of 1,800 strings which do not appear in the training set.

\section*{A. 1 Glossary of Heuristics}
\begin{tabular}{ll} 
EDSM & Reference EDSM implementation LPP98]. \\
W-EDSM & \begin{tabular}{l} 
Reference EDSM implementation with windowing as de- \\
scribed in LPP98.
\end{tabular} \\
Reduction & \begin{tabular}{l} 
Selects merges which reduce the size of the current hy- \\
pothesis most.
\end{tabular} \\
W-Reduction & Same as reduction but with windowing. \\
EDSM-TieCC & \begin{tabular}{l} 
Reference EDSM where an Oracle breaks ties by select- \\
ing a colour-compatible merge if it exists.
\end{tabular} \\
W-EDSM-TieCC & \begin{tabular}{l} 
Reference W-EDSM where an Oracle breaks ties by se- \\
lecting a colour-compatible merge if it exists.
\end{tabular} \\
Full-EDSM & \begin{tabular}{l} 
Reference EDSM where an Oracle fully labels the states \\
in the starting APTA.
\end{tabular} \\
Col-EDSM & \begin{tabular}{l} 
Reference EDSM where an Oracle fully colours each \\
state in the starting APTA.
\end{tabular} \\
FullCol-EDSM & \begin{tabular}{l} 
Reference EDSM where an Oracle fully labels and \\
colours colours each state in the starting APTA.
\end{tabular} \\
\hline
\end{tabular}

Colk-EDSM \(+\mathbf{W}\) - An Oracle selects the highest EDSM scoring colourEDSM compatible merge for the first \(k\) merge steps, then proceeds with W-EDSM.

Colk-W-EDSM+W- An Oracle selects the highest W-EDSM scoring colourcompatible merge for the first \(k\) merge steps, then proceeds with W-EDSM.

E/WE/R/WR Best of EDSM, W-EDSM, Reduction, and WReduction.

A1/A2/... Best of the A1 ... L3 ensemble heuristics.
E/WE/R/WR/A1/. . Best of EDSM, W-EDSM, Reduction, W-Reduction and the A1 ... L3 ensemble heuristics.

GFirst/ \(\delta / \alpha / \beta / b / f\)
Delta Graph where only the first colour-compatible leaf is expanded and \(\delta\) is the depth of the graph, \(\alpha, \beta\) are the minimum reduction and minimum EDSM score to build the APTA reduction table with, \(b\) is the branching factor, and \(f\) is the branching limit.

Graph \(/ \delta / \alpha / \beta / b / f \quad\) Delta Graph where \(\delta\) is the depth of the graph, \(\alpha, \beta\) are the minimum reduction and minimum EDSM score to build the APTA reduction table with, \(b\) is the branching factor, and \(f\) is the branching limit.

Random
Randomly selects valid merges at each step to construct a path (random search).

Blind/n32/v1 'Blind' GA on 32-state target DFAs at density 1. Parents are randomly (blindly) selected for mating without using a fitness function. Starting pop. size \(=100\), final pop. size \(=5000\), chrom. length \(=8, \alpha=25, \beta=1\), mutation rate \(=1 \%\).

GA/n32/v1

GA/n32/v2

GA/n32/v3

GA/n64/v1

GA/n64/v2

GA/n64/v3

GA/n64/v4
GA/n64/v5

GA on 32-state target DFAs at density 1. Pop. size \(=100\), max. generations \(=50\), chrom. length \(=8\), tournament size \(=5, \alpha=25, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

Best of two runs of GA/n32/v1.

GA on 32-state target DFAs at density 1. Pop. size \(=200\), max. generations \(=100\), chrom. length \(=8\), tournament size \(=5, \alpha=25, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

GA on 64 -state target DFAs at density 1. Pop. size \(=100\), max. generations \(=50\), chrom. length \(=8\), tournament size \(=5, \alpha=25, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

GA on 64 -state target DFAs at density 1. Pop. size \(=100\), max. generations \(=50\), chrom. length \(=8\), tournament size \(=5, \alpha=60, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

GA on 64 -state target DFAs at density 1. Pop. size \(=100\), max. generations \(=50\), chrom. length \(=6\), tournament size \(=5, \alpha=60, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

Best of GA/n64/v1, v2, and v3.
GA on 64 -state target DFAs at density 1. Pop. size \(=200\), max. generations \(=50\), chrom. length \(=6\), tournament size \(=5, \alpha=60, \beta=1\), crossover \(=80 \%\), mutation \(=1 \%\), and elite \(=10 \%\).

\section*{A. 2 32-State Target DFA Instances}

\section*{A.2.1 n32d607e1024.sqlite}

Table A. 2 shows the baseline, Oracle-assisted, first-merge assisted, ensemble, and Delta Graph results obtained over 1024 problem instances having symmetrically structurally complete training data.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{1024 Experiments, n32d607e1024.sqlite} \\
\hline \multicolumn{6}{|l|}{32-State Target, 607 Strings, Symmetrically Structurally Complete} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{|c|}{Baseline} \\
\hline EDSM & 15.7\% & 19.6\% & 4.3\% & 75 & 90 \\
\hline W-EDSM & 14.1\% & 17.78\% & \(3.6 \%\) & 77 & 92 \\
\hline Reduction & 6.1\% & 7.7\% & 1.8\% & 88 & 97 \\
\hline W-Reduction & 5.9\% & 7.8\% & 1.3\% & 89 & 99 \\
\hline \multicolumn{6}{|c|}{Oracle-Assisted} \\
\hline EDSM-TieCC & 25.5\% & \(26.4 \%\) & 16.6\% & 71 & 88 \\
\hline Full-EDSM & 98.9\% & 99.5\% & 56.1\% & 32 & 32 \\
\hline Col-EDSM & 100\% & 100\% & 100\% & 32 & 32 \\
\hline FullCol-EDSM & 100\% & 100\% & 100\% & 32 & 32 \\
\hline \multicolumn{6}{|c|}{Getting the First Steps Right, Oracle-Assisted} \\
\hline Col01-EDSM + W-EDSM & 20\% & 25.9\% & 4.8\% & 69 & 84 \\
\hline Col02-EDSM+W-EDSM & 28.8\% & 37.1\% & 8.1\% & 60 & 49 \\
\hline Col03-EDSM + W-EDSM & 31.7\% & 41.3\% & 8.5\% & 54 & 38 \\
\hline Col04-EDSM + W-EDSM & 39.1\% & 50.4\% & 9.9\% & 48 & 33 \\
\hline Col05-EDSM + W-EDSM & 43.4\% & 57.9\% & 11.9\% & 43 & 32 \\
\hline Col06-EDSM + W-EDSM & 51.2\% & 64.1\% & 14.8\% & 40 & 32 \\
\hline Col07-EDSM + W-EDSM & 51.4\% & 66.6\% & 15.5\% & 38 & 32 \\
\hline Col08-EDSM + W-EDSM & 54.9\% & 71.3\% & 17.3\% & 36 & 32 \\
\hline \multicolumn{6}{|c|}{Ensemble} \\
\hline E/WE/R/WR & 21.8\% & 24.7\% & 6.8\% & 71 & 88 \\
\hline A1/A2/... & \(28.4 \%\) & \(30.3 \%\) & 13.4\% & 67 & 83 \\
\hline E/WE/R/WR/A1/... & 29.5\% & \(31.5 \%\) & 14\% & 66 & 82 \\
\hline \multicolumn{6}{|c|}{Delta Graph, First CC Variant} \\
\hline
\end{tabular}
\begin{tabular}{lccccc}
\hline GFirst \(/ 3 / 25 / 1 / 6 / 3\) & \(28.9 \%\) & \(38.6 \%\) & \(7.3 \%\) & 59 & 46 \\
GFirst \(/ 3 / 25 / 1 / 12 / 3\) & \(31.3 \%\) & \(42.4 \%\) & \(8.8 \%\) & 55 & 38 \\
GFirst \(/ 4 / 25 / 1 / 10 / 3\) & \(35.3 \%\) & \(46.6 \%\) & \(9.6 \%\) & 52 & 34 \\
\hline & & Delta Graph & & & 33 \\
\hline Graph \(/ 3 / 25 / 1 / 6 / 3\) & \(40.1 \%\) & \(52.2 \%\) & \(10.4 \%\) & 55 & 32 \\
Graph \(/ 3 / 25 / 1 / 12 / 3\) & \(50.7 \%\) & \(65.2 \%\) & \(14.4 \%\) & 47 & 32 \\
Graph \(/ 4 / 25 / 1 / 10 / 3\) & \(49 \%\) & \(62.2 \%\) & \(13.6 \%\) & 49 & 3 \\
\hline
\end{tabular}

Table A.2: Aggregated results from n32d607e1024.sqlite.

\section*{A.2.2 n32d607e128_GA.sqlite}

Table A. 3 shows the genetic algorithm results obtained over 128 problem instances having symmetrically structurally complete training data.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{\multirow[t]{2}{*}{128 Experiments, n32d607e128_GA.sqlite}} \\
\hline & & & & & \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{|c|}{Baseline and Genetic Algorithm} \\
\hline EDSM & 16.4\% & 21.1\% & \(3.9 \%\) & 72 & 86 \\
\hline Random & \(0 \%\) & 0\% & 0\% & 776 & 776 \\
\hline Blind/n32/v1 & 0\% & 0\% & 0\% & 81 & 81 \\
\hline GA/n32/v1 & 46.9\% & 63.3\% & 9.4\% & 39 & 32 \\
\hline GA/n32/v2 & 62.5\% & 80.5\% & 16.4\% & 35 & 32 \\
\hline GA/n32/v3 & 68.8\% & 87.5\% & \(16.4 \%\) & 34 & 32 \\
\hline
\end{tabular}

Table A.3: Aggregated results from n32d607e128_GA.sqlite.

\section*{A.2.3 n32d607e512_EdsmFailing.sqlite}

Table A. 4 shows the baseline, Oracle-assisted, ensemble, and Delta Graph results obtained over 512 problem instances having adversarial training data which is symmetrically structurally complete but none of the first merge ties contain any colour-compatible merges.

Adversarial Setup, 512 Experiments, n32d607e512_EdsmFailing.sqlite 32-State Target, 607 Strings, Symmetrically Structurally Complete
\begin{tabular}{lccccc} 
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{c}{ Baseline } \\
\hline EDSM & \(0.2 \%\) & \(0.2 \%\) & \(0 \%\) & 92 & 93 \\
W-EDSM & \(0.2 \%\) & \(0.2 \%\) & \(0 \%\) & 94 & 96 \\
Reduction & \(1.6 \%\) & \(2.3 \%\) & \(0.2 \%\) & 96 & 98 \\
W-Reduction & \(1.6 \%\) & \(2 \%\) & \(0.2 \%\) & 97 & 100 \\
\hline \multicolumn{6}{c}{ Oracle-Assisted } \\
\hline EDSM-TieCC & \(0.4 \%\) & \(0.2 \%\) & \(0 \%\) & 91 & 93 \\
FullCol-EDSM & \(100 \%\) & \(100 \%\) & \(95.9 \%\) & 32 & 32 \\
\hline & \multicolumn{6}{c}{ Ensemble } \\
\hline E/WE/R/WR & \(1.8 \%\) & \(2.3 \%\) & \(0.4 \%\) & 89 & 92 \\
A1/A2/... & \(3.5 \%\) & \(3.9 \%\) & \(0.6 \%\) & 85 & 90 \\
E/WE/R/WR/A1/... & \(3.5 \%\) & \(4.1 \%\) & \(1 \%\) & 85 & 90 \\
\hline & Delta & Graph & & \\
\hline Graph/3/25/1/6/3 & \(20.9 \%\) & \(31.4 \%\) & \(4.1 \%\) & 71 & 92 \\
Graph/3/25/1/12/3 & \(35.4 \%\) & \(50.6 \%\) & \(5.1 \%\) & 56 & 34 \\
Graph/4/25/1/10/3 & \(32.6 \%\) & \(46.5 \%\) & \(5.5 \%\) & 59 & 35 \\
\hline
\end{tabular}

Table A.4: Aggregated results from n32d607e512_EdsmFailing.sqlite.

\section*{A.2.4 n32d607e64_EdsmFailing.sqlite}

Table A.5 shows the baseline, Oracle-assisted, ensemble, Delta Graph, and genetic algorithm results obtained over 64 problem instances having adversarial training data which is symmetrically structurally complete but none of the first merge ties contain any colour-compatible merges.

\section*{Note}

The problem instances in this section are created and evaluated in exactly the same way as those in the previous section. The only difference is that there are 64 problem instances rather than 512 to accommodate for the fact that the genetic algorithms take much longer to run.

Adversarial Setup, 64 Experiments, n32d607e64_EdsmFailing.sqlite 32-State Target, 607 Strings, Symmetrically Structurally Complete
\begin{tabular}{|c|c|c|c|c|c|}
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{|c|}{Baseline} \\
\hline EDSM & 0\% & 0\% & 0\% & 90 & 92 \\
\hline W-EDSM & 0\% & 0\% & 0\% & 93 & 95 \\
\hline Reduction & 0\% & 0\% & 0\% & 96 & 99 \\
\hline W-Reduction & 0\% & 0\% & 0\% & 97 & 100 \\
\hline \multicolumn{6}{|c|}{Oracle-Assisted} \\
\hline EDSM-TieCC & 0\% & 0\% & 0\% & 90 & 94 \\
\hline FullCol-EDSM & 100\% & 100\% & 100\% & 32 & 32 \\
\hline \multicolumn{6}{|c|}{Ensemble} \\
\hline E/WE/R/WR & 0\% & 0\% & 0\% & 88 & 91 \\
\hline A1/A2/... & 1.6\% & 3.1\% & 1.6\% & 84 & 89 \\
\hline E/WE/R/WR/A1/... & 1.6\% & \(3.1 \%\) & 1.6\% & 83 & 88 \\
\hline \multicolumn{6}{|c|}{Delta Graph} \\
\hline Graph/3/25/1/6/3 & \(23.4 \%\) & \(39.1 \%\) & 4.7\% & 67 & 91 \\
\hline Graph/3/25/1/12/3 & 37.5\% & 59.4\% & 15.6\% & 51 & 32 \\
\hline Graph/4/25/1/10/3 & 39.1\% & 54.7\% & 9.4\% & 57 & 32 \\
\hline \multicolumn{6}{|c|}{Genetic Algorithm} \\
\hline GA/n32/v1 & 42.2\% & \(56.3 \%\) & 7.8\% & 43 & 32 \\
\hline GA/n32/v2 & \(62.5 \%\) & 79.7\% & 18.8\% & 37 & 32 \\
\hline
\end{tabular}

Table A.5: Aggregated results from n32d607e64_EdsmFailing.sqlite.

\section*{A.2.5 n32d607e512_NotStructComp.sqlite}

Table A. 6 shows the baseline, ensemble, and Delta Graph results obtained over 512 problem instances having adversarial training data which is not symmetrically structurally complete.

Adversarial Setup, 512 Experiments, n32d607e512_NotStructComp.sqlite 32-State Target, 607 Strings, Not Symmetrically Structurally Complete
\begin{tabular}{ccccc} 
Heuristic \(\leq 1 \%\) Error & \(\pm 1\) Target Exact Mean Size & Median Size \\
\hline Baseline \\
\hline
\end{tabular}
\begin{tabular}{lccccc}
\hline EDSM & \(8.4 \%\) & \(15.2 \%\) & \(0 \%\) & 76 & 90 \\
W-EDSM & \(9.6 \%\) & \(14.1 \%\) & \(0 \%\) & 78 & 91 \\
Reduction & \(2.3 \%\) & \(5.3 \%\) & \(0 \%\) & 91 & 97 \\
W-Reduction & \(2.5 \%\) & \(4.7 \%\) & \(0 \%\) & 92 & 99 \\
\hline
\end{tabular}
\begin{tabular}{lccccc}
\multicolumn{6}{c}{ Ensemble } \\
\hline E/WE/R/WR & \(12.9 \%\) & \(18.9 \%\) & \(0 \%\) & 72 & 87 \\
A1/A2/.. & \(16.8 \%\) & \(21.7 \%\) & \(0 \%\) & 70 & 85 \\
E/WE/R/WR/A1/.. & \(18.2 \%\) & \(23.2 \%\) & \(0 \%\) & 68 & 83 \\
\hline \multicolumn{6}{c}{ Delta Graph } \\
\hline Graph/3/25/1/6/3 & \(25.2 \%\) & \(41.8 \%\) & \(0 \%\) & 58 & 35 \\
Graph/3/25/1/12/3 & \(34.2 \%\) & \(57.6 \%\) & \(0 \%\) & 48 & 32 \\
Graph/4/25/1/10/3 & \(31.8 \%\) & \(53.9 \%\) & \(0 \%\) & 51 & 32 \\
\hline
\end{tabular}

Table A.6: Aggregated results from n32d607e512_NotStructComp.sqlite.

\section*{A.2.6 n32d607e64_NotStructComp.sqlite}

Table A. 7 shows the baseline, ensemble, Delta Graph, and genetic algorithm results obtained over 64 problem instances on adversarial training data which is not symmetrically structurally complete.

\section*{Note}

The problem instances in this section are created and evaluated in exactly the same way as those in the previous section. The only difference is that there are 64 problem instances rather than 512 to accommodate for the fact that the genetic algorithms take much longer to run.

Adversarial Setup, 64 Experiments, n32d607e64_NotStructComp.sqlite 32-State Target, 607 Strings, Not Symmetrically Structurally Complete
\begin{tabular}{lccccc} 
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{c}{ Baseline } \\
\hline EDSM & \(9.4 \%\) & \(14.1 \%\) & \(0 \%\) & 75 & 86 \\
W-EDSM & \(10.9 \%\) & \(15.6 \%\) & \(0 \%\) & 76 & 89 \\
Reduction & \(3.1 \%\) & \(4.7 \%\) & \(0 \%\) & 90 & 97 \\
W-Reduction & \(3.1 \%\) & \(4.7 \%\) & \(0 \%\) & 91 & 98 \\
\hline \multicolumn{6}{c}{ Ensemble } \\
\hline E/WE/R/WR & \(15.6 \%\) & \(20.3 \%\) & \(0 \%\) & 70 & 84 \\
A1/A2/.. & \(20.3 \%\) & \(25 \%\) & \(0 \%\) & 67 & 77 \\
E/WE/R/WR/A1/.. & \(21.9 \%\) & \(25 \%\) & \(0 \%\) & 65 & 75 \\
\hline \multicolumn{6}{c}{ Delta Graph } \\
\hline Graph/3/25/1/6/3 & \(28.1 \%\) & \(50 \%\) & \(0 \%\) & 52 & 32 \\
Graph/3/25/1/12/3 & \(35.9 \%\) & \(62.5 \%\) & \(0 \%\) & 43 & 32 \\
Graph/4/25/1/10/3 & \(34.4 \%\) & \(57.8 \%\) & \(0 \%\) & 47 & 32 \\
\hline \multicolumn{6}{c}{ Genetic Algorithm } \\
\hline GA/n32/v1 & \(40.6 \%\) & \(57.8 \%\) & \(0 \%\) & 41 & 32 \\
GA/n32/v2 & \(56.3 \%\) & \(73.4 \%\) & \(0 \%\) & 36 & 32 \\
\hline
\end{tabular}

Table A.7: Aggregated results from n32d607e64_NotStructComp.sqlite.

\section*{A.2.7 n32d607e512_NoLoops.sqlite}

Table A. 8 shows the baseline, ensemble, and Delta Graph results obtained over 512 problem instances having adversarial training data where none of the target DFAs have any loop transitions.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{Adversarial Setup, 512 Experiments, n32d607e512_NoLoops.sqlite} \\
\hline & 32-State & Target, 607 & Strings & & \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{|c|}{Baseline} \\
\hline EDSM & 8\% & 13.5\% & \(2 \%\) & 82 & 92 \\
\hline W-EDSM & 8.4\% & 13.1\% & 1.2\% & 83 & 95 \\
\hline Reduction & 3.3\% & 5.1\% & 0.6\% & 93 & 99 \\
\hline W-Reduction & 2.7\% & 4.9\% & 0.4\% & 94 & 100 \\
\hline \multicolumn{6}{|c|}{Ensemble} \\
\hline E/WE/R/WR & 12.3\% & 16.6\% & 2.3\% & 78 & 91 \\
\hline A1/A2/... & 20.3\% & 22.5\% & 6.8\% & 74 & 89 \\
\hline E/WE/R/WR/A1/... & 20.5\% & 22.7\% & 6.8\% & 73 & 88 \\
\hline \multicolumn{6}{|c|}{Delta Graph} \\
\hline Graph/3/25/1/6/3 & 29.3\% & 42.6\% & 7.4\% & 63 & 41 \\
\hline Graph/3/25/1/12/3 & 41.8\% & 57.4\% & 9.2\% & 51 & 32 \\
\hline Graph/4/25/1/10/3 & 38.9\% & 54.3\% & 7\% & 54 & 32 \\
\hline
\end{tabular}

Table A.8: Aggregated results from n32d607e512_NoLoops.sqlite.

\section*{A.2.8 n32d607e64_NoLoops.sqlite}

Table A. 9 shows the baseline, ensemble, Delta Graph, and genetic algorithm results obtained over 64 problem instances having adversarial training data where none of the target DFAs have any loop transitions.

\section*{Note}

The problem instances in this section are created and evaluated in exactly the same way as those in the previous section. The only difference is that there are 64 problem instances rather than 512 to accommodate for the fact that the genetic algorithms take much longer to run.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{Adversarial Setup, 64 Experiments, n32d607e64_NoLoops.sqlite} \\
\hline \multicolumn{6}{|c|}{32-State Target, 607 Strings} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{|c|}{Baseline} \\
\hline EDSM & 6.3\% & 6.3\% & 0\% & 84 & 92 \\
\hline W-EDSM & 6.3\% & 6.3\% & 0\% & 89 & 96 \\
\hline Reduction & 1.6\% & 1.6\% & 0\% & 97 & 99 \\
\hline W-Reduction & 1.6\% & 1.6\% & 0\% & 98 & 101 \\
\hline \multicolumn{6}{|c|}{Ensemble} \\
\hline E/WE/R/WR & 6.3\% & 6.3\% & 0\% & 81 & 92 \\
\hline A1/A2/... & 15.6\% & 15.6\% & 7.8\% & 77 & 90 \\
\hline E/WE/R/WR/A1/... & 15.6\% & 15.6\% & 7.8\% & 77 & 90 \\
\hline \multicolumn{6}{|c|}{Delta Graph} \\
\hline Graph/3/25/1/6/3 & 26.6\% & 46.9\% & 4.7\% & 61 & 35 \\
\hline Graph/3/25/1/12/3 & 43.8\% & 60.9\% & 7.8\% & 49 & 32 \\
\hline Graph/4/25/1/10/3 & 42.2\% & 56.3\% & 9.4\% & 54 & 32 \\
\hline \multicolumn{6}{|c|}{Genetic Algorithm} \\
\hline GA/n32/v1 & 37.5\% & \(45.3 \%\) & 4.7\% & 46 & 36 \\
\hline GA/n32/v2 & 46.9\% & 62.5\% & 7.8\% & 41 & 32 \\
\hline
\end{tabular}

Table A.9: Aggregated results from n32d607e64_NoLoops.sqlite.

\section*{A. 3 64-State Target DFA Instances}

\section*{A.3.1 n64d1e1024.sqlite}

Table A. 10 shows the baseline, Oracle-assisted, first-merge assisted, ensemble, and Delta Graph results obtained over 1024 problem instances having symmetrically structurally complete training data.

\section*{1024 Experiments, n64d1e1024.sqlite}

64-State Target, 1,521 Strings, Symmetrically Structurally Complete
\begin{tabular}{|c|c|c|c|c|c|}
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{|c|}{Baseline} \\
\hline EDSM & 15.2\% & 17.7\% & 0.6\% & 162 & 200 \\
\hline W-EDSM & 13.8\% & 16.5\% & 0.5\% & 170 & 207 \\
\hline Reduction & \(2.4 \%\) & \(3 \%\) & 0.1\% & 204 & 219 \\
\hline W-Reduction & 2.5\% & 2.8\% & 0.1\% & 205 & 221 \\
\hline \multicolumn{6}{|c|}{Oracle-Assisted} \\
\hline EDSM-TieCC & 24\% & 22.3\% & 9.2\% & 154 & 199 \\
\hline Full-EDSM & 100\% & 98.6\% & 31.3\% & 64 & 64 \\
\hline Col-EDSM & 100\% & 100\% & 100\% & 64 & 64 \\
\hline FullCol-EDSM & 100\% & 100\% & 100\% & 64 & 64 \\
\hline \multicolumn{6}{|c|}{Getting the First Steps Right, Oracle-Assisted} \\
\hline Col01-EDSM + W-EDSM & 19\% & 21\% & 1.3\% & 159 & 205 \\
\hline Col02-EDSM + W-EDSM & 22.9\% & 26.7\% & 0.8\% & 144 & 172 \\
\hline Col03-EDSM + W-EDSM & 27.4\% & 29.7\% & 1.2\% & 132 & 113 \\
\hline Col04-EDSM + W-EDSM & \(32.8 \%\) & 35\% & 1.7\% & 121 & 86 \\
\hline Col05-EDSM + W-EDSM & 36.3\% & 38.3\% & \(2 \%\) & 110 & 76 \\
\hline Col06-EDSM + W-EDSM & 40.6\% & 43\% & 1.7\% & 102 & 67 \\
\hline Col07-EDSM + W-EDSM & 43.8\% & 44.4\% & 1.2\% & 96 & 66 \\
\hline Col08-EDSM + W-EDSM & 47.5\% & 49.2\% & 2.5\% & 89 & 65 \\
\hline \multicolumn{6}{|c|}{Ensemble} \\
\hline E/WE/R/WR & 19\% & 20.7\% & 1\% & 158 & 199 \\
\hline A1/A2/... & 24.3\% & 24.5\% & 4.1\% & 149 & 193 \\
\hline E/WE/R/WR/A1/... & 25.8\% & 25.9\% & 4.2\% & 147 & 193 \\
\hline \multicolumn{6}{|c|}{Delta Graph, First CC Variant} \\
\hline GFirst/3/60/1/12/3 & 27.3\% & \(30.6 \%\) & 1.7\% & 133 & 111 \\
\hline GFirst/4/60/1/10/3 & 31.2\% & \(33.1 \%\) & 1\% & 128 & 98 \\
\hline \multicolumn{6}{|c|}{Delta Graph} \\
\hline Graph/3/60/1/12/3 & 42.9\% & 47\% & 2.2\% & 113 & 65 \\
\hline Graph/4/60/1/10/3 & 40.1\% & 44.8\% & 1.7\% & 118 & 67 \\
\hline
\end{tabular}

Table A.10: Aggregated results from n64d1e1024.sqlite.

\section*{A.3.2 n64d1e64_GA.sqlite}

Table A. 11 shows the genetic algorithm results obtained over 64 problem instances having symmetrically structurally complete training data.
\begin{tabular}{lccccc}
\hline \multicolumn{5}{c}{ 64 Experiments, n64d1e64_GA.sqlite } \\
64-State Target, 1,521 Strings, Symmetrically Structurally Complete \\
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{5}{c}{ Baseline and } & Genetic \\
Algorithm \\
\hline EDSM & \(12.5 \%\) & \(21.9 \%\) & \(1.6 \%\) & 161 & 199 \\
GA/n64/v1 & \(17.2 \%\) & \(21.9 \%\) & \(0 \%\) & 113 & 105 \\
GA/n64/v2 & \(14.1 \%\) & \(25 \%\) & \(1.6 \%\) & 131 & 113 \\
GA/n64/v3 & \(26.6 \%\) & \(32.8 \%\) & \(1.6 \%\) & 110 & 94 \\
GA/n64/v4 & \(37.5 \%\) & \(42.2 \%\) & \(3.1 \%\) & 94 & 79 \\
GA/n64/v5 & \(56.3 \%\) & \(59.4 \%\) & \(4.7 \%\) & 85.6 & 64 \\
\hline
\end{tabular}

Table A.11: Aggregated results from n64d1e64_GA.sqlite.

\section*{A.3.3 n64d1e512_Unrestricted.sqlite}

Table A. 12 shows the results obtained over 512 problem instances where the target DFAs and the training sets are allowed to be created exactly following the Abbadingo specifications without any restrictions. In other words, the training data may be symmetrically structurally complete or not, it may result in APTAs where the first rank EDSM scoring merges contain a colour-compatible merge or not, the target DFAs may or may not contain loops, and we place no requirement that the proportion of positive and negative strings in the training set is less than or equal to \(20 \%\). The exact characteristics of this set of 512 problem instances are shown in Table A. 13 below.

512 Experiments, n64d1e512_Unrestricted.sqlite
64-State Target, 1,521 Strings
\begin{tabular}{lccccc} 
Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{c}{ Baseline } \\
\hline EDSM & \(12.1 \%\) & \(12.5 \%\) & \(0 \%\) & 162 & 197 \\
W-EDSM & \(9.8 \%\) & \(12.1 \%\) & \(0 \%\) & 170 & 205 \\
Reduction & \(1.4 \%\) & \(0 \%\) & \(0 \%\) & 205 & 218 \\
W-Reduction & \(1.6 \%\) & \(0 \%\) & \(0 \%\) & 207 & 219 \\
\hline \multicolumn{6}{c}{ Ensemble } \\
\hline E/WE/R/WR & \(14.1 \%\) & \(15.4 \%\) & \(0 \%\) & 159 & 196 \\
A1/A2/.. & \(19.5 \%\) & \(18.9 \%\) & \(1 \%\) & 156 & 196 \\
E/WE/R/WR/A1/... & \(20.3 \%\) & \(19.7 \%\) & \(1 \%\) & 153 & 193 \\
\hline \multicolumn{6}{c}{ Delta } \\
& Graph & & & \\
\hline Graph/3/60/1/12/3 & \(34.2 \%\) & \(39.8 \%\) & \(0.2 \%\) & 121 & 70 \\
Graph/4/60/1/10/3 & \(30.1 \%\) & \(35.6 \%\) & \(0.6 \%\) & 128 & 77 \\
\hline
\end{tabular}

Table A.12: Aggregated results from n64d1e512_Unrestricted.sqlite.
\begin{tabular}{|c|c|c|}
\hline \multicolumn{3}{|l|}{Characteristics of Unrestricted Experiments 64-State Target, 1,521 Strings, 512 Experiments} \\
\hline Property & \# Instances & Likelihood \\
\hline Training set is symmetrically structurally complete & 212 of 512 & 41\% \\
\hline Training set is not symmetrically structurally complete & 300 of 512 & 59\% \\
\hline CC merge exists in first EDSM rank in APTA & 298 of 512 & 58\% \\
\hline No CC merge in first EDSM rank in APTA & 214 of 512 & 42\% \\
\hline Target DFA contains loops & 412 of 512 & 80\% \\
\hline Target DFA does not contain loops & 100 of 512 & 20\% \\
\hline Proportion of +ve and -ve in training set \(\leq 20 \%\) & 383 of 512 & 75\% \\
\hline Proportion of + ve and -ve in training set \(>20 \%\) & 129 of 512 & 25\% \\
\hline Table A.13: Characteristics of the target DFA n64d1e512_Unrestricted.sqlite. & and traini & data in \\
\hline
\end{tabular}

\section*{A. 4 128-State Target DFA Instances}

\section*{A.4.1 n128d1e512.sqlite}

Table A. 14 shows the baseline, Oracle-assisted, first-merge assisted, ensemble, and Delta Graph results obtained over 512 problem instances having symmetrically structurally complete training data.
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multicolumn{6}{|c|}{512 Experiments, n128d1e512.sqlite} \\
\hline \multicolumn{6}{|l|}{128-State Target, 4,382 Strings, Symmetrically Structurally Complete} \\
\hline Heuristic & \(\leq 1 \%\) Error & \(\pm 1\) Target & Exact & Mean Size & Median Size \\
\hline \multicolumn{6}{|c|}{Baseline} \\
\hline W-EDSM & 22.7\% & 15.4\% & \(0 \%\) & 377 & 511 \\
\hline W-Reduction & 2\% & 1.6\% & \(0 \%\) & 519 & 558 \\
\hline \multicolumn{6}{|c|}{Oracle-Assisted} \\
\hline W-EDSM-TieCC & 27.5\% & 18.2\% & 7.4\% & 359 & 502 \\
\hline \multicolumn{6}{|c|}{Getting the First Steps Right, Oracle-Assisted} \\
\hline Col01-W-EDSM+W-EDSM & 23.2\% & 15.6\% & 0.4\% & 368 & 505 \\
\hline Col02-W-EDSM + W-EDSM & 25.8\% & 17.6\% & 0.2\% & 345 & 406 \\
\hline Col03-W-EDSM + W-EDSM & 30.5\% & 21.3\% & 0.4\% & 320 & 264 \\
\hline Col04-W-EDSM + W-EDSM & \(33.6 \%\) & 23.6\% & 0.2\% & 296 & 213 \\
\hline Col05-W-EDSM+W-EDSM & 38.5\% & 27.1\% & 0.8\% & 270 & 188 \\
\hline Col06-W-EDSM+W-EDSM & 40.6\% & 27.7\% & 0.6\% & 256 & 173 \\
\hline Col07-W-EDSM+W-EDSM & 43.4\% & 30.9\% & 0.8\% & 233 & 157 \\
\hline Col08-W-EDSM+W-EDSM & 48\% & \(33.2 \%\) & 0.6\% & 207 & 146 \\
\hline \multicolumn{6}{|c|}{Ensemble} \\
\hline WE/WR & 23.2\% & 16.4\% & 0\% & 375 & 511 \\
\hline A1/A2/... & 30.3\% & 22.9\% & 1.4\% & 342 & 455 \\
\hline WE/WR/A1/... & 30.7\% & 23.4\% & 1.4\% & 340 & 521 \\
\hline \multicolumn{6}{|c|}{Delta Graph, First CC Variant} \\
\hline GFirst/3/180/1/12/3 & 27.1\% & 18.6\% & 0\% & 341 & 388 \\
\hline GFirst/4/180/1/10/3 & \(33.2 \%\) & 23.6\% & 0.2\% & 309 & 241 \\
\hline \multicolumn{6}{|c|}{Delta Graph} \\
\hline Graph/3/180/1/12/3 & 48.6\% & 38.1\% & 0.4\% & 272 & 140 \\
\hline Graph/4/180/1/10/3 & 47.3\% & 36.3\% & 0.4\% & 280 & 148 \\
\hline Graph/4/180/1/12/3 & 49.4\% & 39.1\% & 0.4\% & 275 & 134 \\
\hline
\end{tabular}

Table A.14: Aggregated results from n128d1e512.sqlite.

\section*{Appendix B}

\section*{State Reduction Rates}

In this appendix, we consider 1024 problem instances having 64 -state target DFAs at the lowest Abbadingo density of 1,521 strings. We use the FullCol-EDSM heuristic (see Section 7.3) to build merge paths leading to the exact target in exactly \(n+1\) merge steps (where \(n=64\) is the size of the target DFA). At each of the \(n+1=65\) merge steps, we show the minimum, maximum, and mean number of states reduced over the 1024 runs. From these results, we observe that there is an expected minimum number of states reduced at each merge step. For example, we can see that for the first eight merge steps in the table (steps 0 to 7 delimited by the dotted line), on average, merges having fewer than 200 states are never considered.
\begin{tabular}{|c|c|c|c|}
\hline \multicolumn{4}{|l|}{\begin{tabular}{l}
1024 Experiments, Heuristic=FullCol-EDSM \\
64-State Target, 1,521 Strings \\
\(\mathrm{n}+1=65\) Merge Steps to Exact Target
\end{tabular}} \\
\hline Merge Step & Minimum & Maximum & Mean \\
\hline 0 & 208 & 2539 & 749.19 \\
\hline 1 & 171 & 2620 & 454.54 \\
\hline 2 & 148 & 990 & 352.67 \\
\hline 3 & 107 & 972 & 301.59 \\
\hline 4 & 106 & 669 & 265.17 \\
\hline 5 & 89 & 607 & 240.48 \\
\hline 6 & 77 & 569 & 221.08 \\
\hline 7 & 72 & 426 & 206.1 \\
\hline 8 & 67 & 402 & 193.56 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|}
\hline 9 & 54 & 432 & 184.54 \\
\hline 10 & 56 & 375 & 174.57 \\
\hline 11 & 54 & 344 & 165.15 \\
\hline 12 & 53 & 396 & 158.04 \\
\hline 13 & 51 & 292 & 150.11 \\
\hline 14 & 46 & 292 & 143.63 \\
\hline 15 & 43 & 284 & 137.29 \\
\hline 16 & 36 & 284 & 133.06 \\
\hline 17 & 36 & 246 & 127.76 \\
\hline 18 & 35 & 251 & 123.76 \\
\hline 19 & 33 & 233 & 118.96 \\
\hline 20 & 45 & 214 & 115.7 \\
\hline 21 & 33 & 201 & 111.2 \\
\hline 22 & 31 & 221 & 107.2 \\
\hline 23 & 35 & 209 & 104.3 \\
\hline 24 & 30 & 200 & 100.19 \\
\hline 25 & 30 & 198 & 97.04 \\
\hline 26 & 32 & 182 & 94.06 \\
\hline 27 & 29 & 183 & 90.82 \\
\hline 28 & 29 & 179 & 88.36 \\
\hline 29 & 29 & 175 & 84.61 \\
\hline 30 & 27 & 174 & 82.18 \\
\hline 31 & 26 & 188 & 79.46 \\
\hline 32 & 25 & 163 & 77.2 \\
\hline 33 & 26 & 148 & 74.64 \\
\hline 34 & 25 & 189 & 71.83 \\
\hline 35 & 23 & 144 & 69.64 \\
\hline 36 & 22 & 157 & 67.66 \\
\hline 37 & 21 & 128 & 65 \\
\hline 38 & 21 & 139 & 63.14 \\
\hline 39 & 21 & 145 & 60.97 \\
\hline 40 & 20 & 137 & 58.47 \\
\hline 41 & 17 & 154 & 56.44 \\
\hline 42 & 16 & 131 & 54.18 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|}
\hline 43 & 14 & 127 & 52.16 \\
\hline 44 & 13 & 125 & 49.4 \\
\hline 45 & 11 & 103 & 47.36 \\
\hline 46 & 11 & 108 & 45.48 \\
\hline 47 & 10 & 118 & 43.26 \\
\hline 48 & 9 & 108 & 41.3 \\
\hline 49 & 9 & 92 & 38.83 \\
\hline 50 & 9 & 106 & 37.01 \\
\hline 51 & 7 & 82 & 35.04 \\
\hline 52 & 6 & 72 & 32.68 \\
\hline 53 & 6 & 77 & 30.62 \\
\hline 54 & 5 & 80 & 28.71 \\
\hline 55 & 5 & 70 & 26.61 \\
\hline 56 & 4 & 62 & 24.35 \\
\hline 57 & 4 & 56 & 22.04 \\
\hline 58 & 4 & 62 & 19.94 \\
\hline 59 & 4 & 59 & 17.85 \\
\hline 60 & 4 & 48 & 15.44 \\
\hline 61 & 3 & 37 & 13.01 \\
\hline 62 & 2 & 33 & 10.32 \\
\hline 63 & 1 & 17 & 7.44 \\
\hline 64 & 1 & 15 & 5.19 \\
\hline
\end{tabular}

Table B.1: Minimum, maximum, and mean reduction of states at each merge step over 1024 experiments for 64 -state target DFA problem instances.

\section*{Appendix C}

\section*{State Space Search Algorithms}

This appendix contains the pseudo-code for the state space search algorithms described in Chapter 4.

Algorithm C. 1 Depth-first search
Input: A tree \(T\), root node \(s\), and a goal node goal.
Output: A goal state, or not found.
```

$S \leftarrow$ A new stack containing $s \quad / /$ This is the open list.
while $S \neq \emptyset$ do
node $\leftarrow S . p o p()$
if node $=$ goal then
return goal // Or return the path to goal.
else
children $\leftarrow$ node.expand ()
S.push(children) // Push to the top of the stack.
end if
end while

```

11: return goal not found

Algorithm C. 2 Breadth-first search
Input: A tree \(T\), root node \(s\), and a goal node goal.
Output: A goal state, or not found.
\(Q \leftarrow \mathrm{~A}\) new queue containing \(s\)
while \(Q \neq \emptyset\) do
node \(\leftarrow Q\).dequeue()
if node \(=\) goal then
return goal // Or return the path to goal. else
children \(\leftarrow\) node.expand ()
Q.enqueue(children) // Enqueue at the end of the queue.
end if
end while
return goal not found
```

Algorithm C. 3 Iterative deepening search
Input: A tree $T$, root node $s$, and a goal node goal.
Output: A goal state, or not found.
for $n=1$ to $\infty$ do
goal $\leftarrow$ Perform a DFS up to level $n$
if goal is found then
return goal
end if
end for

```
```

Algorithm C. 4 Uniform-cost search
Input: A tree $T$, root node $s$, and a goal node goal.
Output: A goal state, or not found.
1: $Q \leftarrow \mathrm{~A}$ min-queue containing $s \quad / /$ Queue is sorted by $g(n)$.
while $Q \neq \emptyset$ do
node $\leftarrow Q$. .popMin()
if node $=$ goal then
return goal // Or return the path to goal.
else
children $\leftarrow$ node.expand()
Q.push(children)
end if
end while

```
    return goal not found
```

Algorithm C. 5 Iterative deepening A* search
Input: A tree $T$, root node $s$, and a goal node goal.
Output: A goal state, or not found.
threshold $\leftarrow g(s)$
while true do
goal, threshold $\leftarrow \operatorname{IDA} \_\operatorname{DFs}(s$, goal, threshold $)$
if goal is found then
return goal
end if
end while
return not found
function Ida_DFs( $n$, goal, threshold)
cost $\leftarrow g(n)+h(n) \quad / /$ Returns a goal node, and the new threshold.
if cost $>$ threshold then
return nil, cost
end if
if $n=$ goal then
return $n$, threshold
end if
nextThreshold $\leftarrow \infty$
for each child in n.children() do
newNode, newThreshold $\leftarrow I D A \_D F S($ child, goal, threshold)
if newNode $\neq$ nil then
return nodeNode, threshold
end if
nextThreshold $\leftarrow \min$ (nextThreshold, newThreshold)
end for
return nil, nextThreshold
end function

```
```

Algorithm C. 6 Beam search
Input: A tree $T$, root node $s$, a goal node goal, and a beam width $k$.
Output: A goal state, or not found.
1: beam $\leftarrow \mathrm{A}$ list containing $s$
while beam $\neq \emptyset$ do
// Generate and test children.
set $\leftarrow$ A new min-queue // Queue is sorted by $f(n)$.
for each node in beam do
for each child in node.expand () do
if child $=$ goal then
return child
else
set $\leftarrow \operatorname{set} \cup\{$ child $\}$
end if
end for
end for
14: // Populate the beam.
15: $\quad$ beam $\leftarrow \emptyset$
16: $\quad$ while set $\neq \emptyset$ and $k>\mid$ beam $\mid$ do
17: $\quad$ beam $\leftarrow$ beam $\cup\{$ set.dequeue ()$\} \quad / /$ Dequeue minimum $f(n)$.
18: end while
: end while

```
    return goal not found

Algorithm C. 7 Beam search with closed list
Input: A tree \(T\), root node \(s\), a goal node goal, and a beam width \(k\).
Output: A goal state, or not found.
```

visited $\leftarrow \mathrm{A}$ hash table containing $s \quad / /$ This is our closed list
beam $\leftarrow \mathrm{A}$ list containing $s$
while beam $\neq \emptyset$ do
// Generate and test children.
set $\leftarrow$ A new min-queue // Queue is sorted by $f(n)$.
for each node in beam do
for each child in node.expand() do
if child $=$ goal then
return child
else
set $\leftarrow \operatorname{set} \cup\{$ child $\}$
end if
end for
end for

```
    // Populate the beam.
    beam \(\leftarrow \emptyset\)
    while set \(\neq \emptyset\) and \(k>\mid\) beam \(\mid\) do
        state \(\leftarrow\) set.dequeue ()\(\quad / /\) Dequeue minimum \(f(n)\).
        if state \(\notin\) visited then
                        if visited is not full then
                    visited \(\leftarrow\) visited \(\cup\{\) state \(\}\)
                    beam \(\leftarrow\) beam \(\cup\{\) state \(\}\)
        else
            return goal not found // Out of memory.
            end if
        end if
        end while
    end while
    return goal not found

\section*{Appendix D}

\section*{Contents of the Submitted Media}

The attached media has the following structure and contents:
\begin{tabular}{ll} 
/Charts & \begin{tabular}{l} 
Charts and the corresponding data in Microsoft Excel \\
format showing the performance of each heuristic in the \\
ensemble over 1024 problem instances. Also includes
\end{tabular} \\
a correlation matrix showing the relationship between \\
different heuristics in the ensemble.
\end{tabular}```


[^0]:    ${ }^{1}$ Sometimes referred to as Grammatical Induction, Language Identification, or Syntactic Pattern Recognition.

[^1]:    ${ }^{2}$ Assuming that there is no bound on the language description.
    ${ }^{3}$ Depending on the context, such an automaton would be the Maximal Canonical Automaton or the Prefix Tree Acceptor, whose purpose and construction will be discussed later on.
    ${ }^{4}$ By virtue of Cook's theorem, showing that the Boolean satisfiability SAT problem can be transformed into our problem in polynomial time is enough GJ79.

[^2]:    ${ }^{5}$ These we will be covered in detail later in this dissertation

[^3]:    ${ }^{6}$ From here onwards, any hypothesis which has an error of $\leq 1 \%$ on some test set will be referred to as a low-error hypothesis.

[^4]:    ${ }^{7}$ The concept of structural completeness will be covered later on. For the time being it is sufficient to point out that training sets which are not structurally complete are quite likely to occur and the situation is adversarial to heuristics such as EDSM.

[^5]:    ${ }^{8}$ Not to be confused with the ensemble of automata method used by Heule and Verwer in dfasat Die00, HV13].

[^6]:    ${ }^{1}$ From this point onwards, whenever we refer to the structural completeness of some DFA with respect to a sample $S=\left\langle S_{+}, S_{-}\right\rangle$, we always mean it to be in the symmetrical sense.

[^7]:    ${ }^{2}$ Language identification in the limit is a formalism due to Gold which roughly states that a learner will make a finite number of mistakes before converging to a correct solution Gol67.
    ${ }^{3}$ The size of the characteristic set is polynomial with respect to the size of the target DFA.

[^8]:    ${ }^{4}$ The automaton accepts the strings in $S_{+}$and rejects those in $S_{-}$. A consistent automaton is sometimes called a compatible automaton.
    ${ }^{5}$ We have remarked on the complexity of this task in Chapter 1 .

[^9]:    ${ }^{1}$ When speaking of state space search algorithms, the objects we are searching for are typically called 'states'. Here we will refer to them as nodes in order to avoid confusion with states in automata.

[^10]:    ${ }^{2}$ We are not keeping a closed list in this procedure.

[^11]:    ${ }^{3}$ As opposed to some ideal scenario. For instance, Knuth's method would be very accurate in estimating the size of a complete binary tree of height $h$ having $2^{h}-1$ nodes after visiting only $2 h-1$ nodes Pur78.

[^12]:    ${ }^{4}$ Named after Aldous' and Vazirani's, and Knuth's hybrid method.

[^13]:    ${ }^{1}$ A learning algorithm may accept or reject a merge based on additional criteria.

[^14]:    ${ }^{2}$ We also refer the reader to CN97 for more details regarding these constraints.

[^15]:    ${ }^{3}$ In practice, a white state is just one that is neither red nor blue.

[^16]:    ${ }^{4}$ On the other hand, an unbounded backtracking strategy is foolish since it would imply an exhaustive search.

[^17]:    ${ }^{5}$ There are $\frac{\left(n^{2}-n\right)}{2}$ pairings in an $n$-state hypothesis.

[^18]:    ${ }^{6}$ We recommend Grü04] as a good introduction to MDL.

[^19]:    ${ }^{7}$ The authors note that this procedure is highly parallelisable and is amenable to distributed implementations.

