Parameterized Complexity and the Method of Test Sets

Christian Sloper
sloper@ii.uib.no
Cand. Scient. Thesis
Department of Informatics,
University of Bergen,
Norway

October 5, 2001

Contents

0.1 Introduction 4
0.2 Conventions 6
0.3 Acknowledgments 7

1 Classical complexity Theory 8
1.1 What is Complexity theory? 8
1.2 Time Complexity 8
1.3 Classes in classic complexity theory 9
1.4 Class **P** 9
1.5 Class **NP** 10
1.6 **NP**-Completeness 10
1.7 Unsolvable Problems? 11
1.7.1 Average-case analysis 11
1.7.2 Approximation 12
1.7.3 New computing technology 12
1.7.4 Heuristics 13
1.7.5 Fixed Parameter Tractability 13

2 Parameterized Complexity Theory 14
2.1 Parameterized Problems 14
2.2 The Idea 15
2.3 Fixed Parameter Tractability 15
2.4 The negative results 15
2.5 Theory or Practice 16
2.6 History of Parameterized Complexity 16

3 Positive Techniques 19
3.1 Introduction 19
3.2 Bounded Search Trees 19
3.2.1 Dominating Set on bounded degree graphs 20
3.2.2 Minimum DS on bounded degree Graphs 21
3.3 Reduction to problem kernel 22

3.3.1 VERTEX COVER 23
3.4 Combining the methods 24
3.5 Other techniques 27
3.5.1 Well-Quasi-Ordering 27
3.5.2 Bounded Tree-width and Path-width 27
3.5.3 Color Coding 27

4 Negative Techniques 29
4.1 Completeness Theory For PC 29
4.2 Short Turing Machine Acceptance 30
4.3 Polynomial time reductions 30
4.4 Reductions in FPC 31

5 Automata 33
5.1 Using Automata 33
5.2 The Deterministic Finite Automaton 33
5.2.1 Deterministic Finite Automaton Acceptance 34
5.3 The Nondeterministic Finite Automaton 34
5.3.1 Nondeterministic Finite Automaton Acceptance 35
5.3.2 Nondeterministic Finite Automaton Acceptance II . . . 36
5.4 The Myhill-Nerode Theorem 36
5.5 Negative results by Myhill-Nerode 38
5.6 Method of Test Sets 38
5.6.1 Why the parameter? 38
5.6.2 Only short strings are required 40
5.6.3 On the number of strings 42
5.6.4 Solving the promise problem 43

6 Efficient Method of Test Sets 47
6.1 Recognizing equivalence classes 47
6.2 Determining Signatures 48
6.3 Storing signatures 48
6.4 Comparing signatures 49
6.5 Shorter signaturetests 50
6.6 Building the automaton 'On-the-fly' 52
6.7 Example of 'On-the-fly' 53
6.8 The gain 56
6.9 A Space/Time tradeoff 57

7 Using randomization on method of test sets **59**
7.1 Creating a randomized algorithm . . . 60
7.2 Time and space-bounds . . . 61
7.2.1 Timeoptimal MTS (Algorithm 6.1) . . . 61
7.2.2 Spaceoptimal MTS (Algorithm 6.3) . . . 61
7.3 Empirical analysis . . . 61
7.4 Interpreting the results . . . 63
7.5 Size of the test set . . . 63
7.6 All is lost? . . . 65
7.7 Subclasses . . . 66
7.7.1 Bad Subclasses . . . 66
7.7.2 Good subclasses . . . 67
7.8 No duplicates . . . 68

8 Approximating Automata **70**
8.1 Approximation . . . 70
8.2 Defining the approximating languages . . . 71
8.3 The relationship between L, L^+, L^- and L^{RMTS} . . . 71
8.4 Empirical analysis . . . 72
8.5 Interpreting the results . . . 73
8.6 Improving approximation . . . 73

9 Tree-Automata **75**
9.1 Trees . . . 75
9.2 The tree automaton . . . 75
9.3 Simulating Finite Tree Automata . . . 77
9.3.1 Tree-Automaton Acceptance . . . 78

10 Future Work **79**
10.1 Generalizing the Method of Test set . . . 79
10.2 Method of Test Sets and Tree-width . . . 79
10.3 Test Set Compression . . . 80

0.1 Introduction

This thesis in the general field of algorithms and complexity theory will concentrate on the relative new field of parameterized complexity. The thesis is divided into two parts, part I (chapters 1 - 4) is a presentation of parameterized complexity, while the second part (chapters 5-9) is devoted to improving the method of test sets, a technique which is introduced in chapter 5. The last chapter (chapter 10) gives a list of problems for future studies.

Chapter 1 gives a presentation of "classic" complexity theory and then debates problems related to finding good algorithms for polynomially intractable problems.

Chapter 2 introduces parameterized complexity as a way of dealing with polynomial intractable problems. In addition to defining parameterized problems and the complexity class FPT (Fixed Parameter Tractable), it gives a brief history of Fixed Parameter Complexity and discusses the use of FPT in practice.

Chapter 3 also presents known techniques as it shows different methods of proving membership in FPT ("the positive toolkit"). The chapter includes examples of each method as well as proper pseudo-code algorithms for the methods.

Chapter 4 describes the negative toolkit in parameterized complexity, used to show hardness results.

Chapter 5 presents Finite Automaton theory and in particular the Myhill-Nerode Theorem upon which the rest of the thesis is indirectly based. It describes the Method of Test Sets (MTS) and gives two crucial Lemmas (Lemmas 5.9 and 5.10) that tie Myhill-Nerode and the Method of Test Sets together. These essential Lemmas were assumed but not fully proven in the recently published textbook on parameterized complexity [8].

Chapter 6 points out that the MTS presented in the book "Parameterized Complexity" [8] is a very slow version of the algorithm. With a little effort it is possible to get significant improvements in both time and space complexity. This chapter also discusses whether a space/time-tradeoff is possible for the MTS.

Chapter 7 is devoted to randomizing the method of test sets. This gives a polynomial time algorithm for the method of test sets, but it does not produce the correct automaton all the time. Much time has been spent on running experiments on the randomized algorithm and we present the results on how well the algorithm performs on several languages.

Chapter 8 shows that the automaton produced by the randomized method of test sets is an approximation of the automaton produced by the full method of test sets.

The degree of approximation has been tested on several languages and the results are given at the end of this chapter.

Chapter 9 describes a well-known generalization of the deterministic automata that accepts trees and not only strings. It also gives an example of how the tree-automaton works.

Chapter 10 describes which problems I would like to work with after I have finished this thesis.

0.2 Conventions

A few conventions are used consistently throughout the thesis:

$\mathbb{N}$ are the natural numbers.

Σ is always the alphabet of the language in question.

k is the parameter to the problem. In chapter 6,7,8 this is the same as "the promise".

Automata In figures displaying automata, grey nodes are considered to be accepting states while white nodes are considered to be rejecting states.

FPC is an abbreviation of Fixed Parameter Complexity.

FPT is an abbreviation of Fixed Parameter Tractable. A proper definition of this term is in chapter 2.

MTS is an abbreviation of Method of Test Sets.

Problem/Language In this thesis the words "problem" and "language" mean the same.

RMTS is an abbreviation of Randomized Method of Test Sets.

0.3 Acknowledgments

First and foremost I would like to thank my advisor Professor Jan Arne Telle. He has spent considerable energy and effort in advising me when I wrote this thesis and his teaching and guidance have been invaluable.

Second I would also thank Professor Marc Bezem for his help with the thesis and also for his help with the mathematics and generally acting as a second advisor.

I feel that it is important to thank the people who got me interested in algorithms and mathematics in the first place. Without the inspiration from these people I would probably not have continued with graduate studies. In particular Assoc. Prof. S Høyland and Assoc. Prof. F Manne spurred me into the direction of algorithms.

I would like to thank my family; my mother, my father and my brother. The support from them have meant a great deal and their unrockable faith in me have been comforting. Lastly I will thank my friends who have helped taking my mind of things and at least given me the indication that there is life outside the computer lab.

Chapter 1

Classical complexity Theory

1.1 What is Complexity theory?

Complexity theory has always tried to answer one simple question: *"What is easy to compute, and what is hard to compute?"*. While the question is easy to pose, decades of research has shown us that it is not so easy to answer.

A problem is considered computationally hard if it uses an unreasonable amount of time, space, memory or another resource to solve. A problem is considered computationally easy if it can be solved using little (relative to the input size) of the resource in question.

Complexity theory analyzes and groups problems into classes depending on their cost in the resource considered. Complexity theory can focus on any used resource, time and space being the most common. We will consider *time* as our main resource.

1.2 Time Complexity

Time complexity seeks to classify problems by their use of running time compared to the size of input data.

Time is of course a very fickle thing in the world of ever-increasing machines, where a child's game-console out-calculates any 15-year-old supercomputer. The concept of time in complexity theory is not measured by how much time passes, but on how many steps a computer uses to calculate answers.

The complexity is given as a function of the length of input-data. The function is a *worst-case analysis*, meaning that this function gives an upper bound on the running time. The algorithm never runs slower than the function indicates.

Definition 1.1 *Time Complexity*

Let A be an algorithm. The **running time** *or* **time complexity** *of A is the function $f : N \longrightarrow N$, where $f(n)$ is the maximum number of steps that A uses on any input of length n. If $f(n)$ is the running time of A, we say that A runs in time $f(n)$ and that A is an $f(n)$ time algorithm.*

1.3 Classes in classic complexity theory

A system of *time complexity classes* have been devised to classify problems according to their time complexity. Each of these classes contain the problems that are asymptotically equivalent in running time.

Definition 1.2 *Time complexity class*

Let $t : N \longrightarrow N$ be a function. The time complexity class *TIME*$(t(n))$, *is TIME*$(t(n))$ $=\{L \mid L$ *is a language decided by an* $O(t(n))$ *algorithm*[1]$\}$.

Differences in *polynomial* time are in this context considered to be small. Major differences in running time occur when we go from one function to a function *exponentially* different. This can be best illustrated by an example:

> Consider three algorithms of different running time: A linear n algorithm, a cubic n^3 algorithm and a exponential 2^n algorithm. Let the input size be $n = 90$. The linear algorithm finishes in 90 steps and the cubic in 729000 steps. While 729000 is a large number for a human, for a computer both numbers are so small that it would finish in mere seconds. A computer running the exponential algorithm however, running for 2^{90} steps, will not finish in the lifetime of this solar system (even executing billions of instructions per second).

As we have seen, grouping problems that are exponentially equal into classes is not unreasonable.

1.4 Class P

The class **P** is the class of languages that are solvable in polynomial time. The class is important because we consider this class to be roughly equivalent to the class of problems that can be considered computationally easy to solve.

[1]We assume that the reader has knowledge of big-O notation, if not we refer the reader to any book on the subject of algorithms or complexity theory

Even though some problems with polynomial running time have complexity cost that are completely unpractical, say $O(n^{100})$ or have hidden astronomical constants like 2^{100}, it has been shown that most polynomial algorithms can be reduced to a complexity level that is practical.

Definition 1.3 *Class* **P**

$$\mathbf{P} = \bigcup_{k \in \mathbb{N}} TIME(n^k)$$

1.5 Class NP

The class **NP** is another major class in the complexity theory. In **NP** we find all the problems that we can verify with polynomial algorithms. That is, if we have a suggested solution for a problem in **NP**, then we can check if it is a correct solution in polynomial time.

Definition 1.4 *Class* **NP**

$$\mathbf{NP} = \{L \mid \exists V, V \textit{ is a verifier for } L \textit{ and } V \in P\}$$

This class contains thousands of problems that are of huge importance in daily life computing. Polynomial algoritms to these problems would be of utmost interest, the problem is that no-one can find them. The grand question in complexity theory is:

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

It is obvious that $\mathbf{P} \subseteq \mathbf{NP}$, but is $\mathbf{NP} \subseteq \mathbf{P}$? Decades of research on this problem alone has left us with no answer. The general *intuition* is that the classes are different, but while many have tried, all attempts to produce a problem that is in **NP**, but provably not in **P**, have failed.

1.6 NP-Completeness

Even though scientists have been unable to answer $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, the research on **NP**-problems has not been without results.

In the 1970's Stephen Cook and Leonid Levin, independent of each-other, discovered that many problems in NP were linked together. They discovered that it is possible to transform in polynomial time all problems in **NP** to one particular **NP**-problem SAT (the famous Cook-Levin theorem), furthermore they discovered that SAT reduces to other problems in **NP**.

This discovery has enormous consequences as it implies that if any problem that transforms from SAT is solvable in polynomial time, then every problem in **NP** has a polynomial algorithm. This realization gives rise to the theory of **NP**-complete problems.

Definition 1.5 *NP-complete problem*
A language B is **NP***-complete if it satisfies the following:*

1. *B is in* **NP***, and*

2. *every language A in* **NP** *is polynomial time reducible to B.*

The idea of completeness-theory have been used elsewhere. As we shall see it will also be used in Fixed Parameter Complexity.

1.7 Unsolvable Problems?

Unless the unlikely $\mathbf{P} = \mathbf{NP}$ we have now encountered a horde of problems which require exponential time. Thus for anything but very small input size, these problems can be regarded as too time-consuming to solve.

This would not be a problem if these problems were purely theoretical constructs, only of interest to the gray-haired complexity-professor living in the university attic. The set of NP-hard problems unfortunately contains problems that is of importance not only to almost all parts of computer science, but also to the fields of finance, biology, meteorology and many,many others.

Researchers are usually not satisfied with having their problems classified as "NP-hard", they want answers and preferably efficient ones. Several different ideas have been exploited to provide the wanted answers, we introducde the 5 most prominent ones in the following subsections.

1.7.1 Average-case analysis

The worst-case analysis of an algorithm gives a guarantee on running time, but it also may be very deceiving. It may turn out that an algorithm with poor worst-case running time can beat "better" algorithms almost every time. The star example of this is QUICK-SORT, a sorting algorithm with one poor worst-case (input is already sorted in reverse order), but QUICK-SORT is far more efficient than other algorithms on average.

Finding algorithms for NP-hard problems with good average-case running time might solve the problem for many researchers, but unfortunately

average-case analysis is very mathematically taxing and it is often hard to determine what the average input is. Moreover, in many practical situations, like air traffic control, hospital operation equipment etc, having a good worst-case behaviour is life-critical.

1.7.2 Approximation

The realization that we will probably never find efficient exact algorithms for NP-hard problems spurred the research onto a new track. Producing algorithms that quickly attains near-optimal solutions for NP-hard problems were initially met with promising success. Problems like the NP-hard problem KNAPSACK was shown to have *very* good approximation algorithms and similar results soon followed. But for many other problems it was shown that it was NP-complete to find just a good approximation. Recent improvements in proof techniques [3] have further strenthened these negative results.

1.7.3 New computing technology

When we encountered the NP-hard problems we reached a barrier, a barrier which we have been unable to solve satisfactorily. Furthermore, most researchers agrees that we will never be able to break this barrier with the computers we have today.

The answer is simple, “build better machines”. Although the computing power of our machinery has grown enormously the last decades we are still operating on the same principle: One or a few processors sequentially carrying out instructions until the problem is solved. Faster computers will certainly help, but they cannot break the exponential barrier. No matter how fast they become and how many instructions they carry out per second, they cannot keep up with an exponential function. “Regular” computers just will not do.

Instead we need fundamental changes in the way we build computers, moving away from the single/few processor machines to machines operating on completely different architectures. Quantum computers and DNA molecular machines have been some of the proposals for these new architectures, but so far all proposals have had drawbacks. Either they are very difficult to implement (quantum computers) or they only represent a more powerful version of todays computers (molecular computers).

1.7.4 Heuristics

Heuristic algorithms are algorithms which rely on rules-of-thumb and experience of the programmer. Although successful in practice and largely celebrated by the general community, the heuristic algorithms have failed to gain acknowledgment from theoretical computer scientists. Because many heuristics cannot be analyzed and the only real “evidence” that they actually work comes from practical experience, many scientists regard the whole field of heuristics as “voodoo” and not scientifivally sound. In any case it is clear that the heuristic algorithms suffer from their unmathematical foundation which makes them unsuitable for solving problems in general.

1.7.5 Fixed Parameter Tractability

The field of parameterized complexity is one of the newest armaments in the battle against the intractable problems. Fixed Parameter Complexity concentrate on the parameterized problems and gives good algorithms for some of the problems, while proving that others probably do not have good parameterized algorithms. A proper introduction to Fixed Parameter Complexity can been found in the next chapter.

Chapter 2

Parameterized Complexity Theory

The complexity theory we have introduced so far deals with YES/NO-decision problems. Parameterized complexity examines (as its name implies) YES/NO decision problems with parameters. It tries to distinguish problems that have good behavior when parameterized from those where parameterizing the problems bring little help.

2.1 Parameterized Problems

A decision problem is generally of the form: "Given an instance I, does I have property P". By giving a restriction on the property P we get problems of the form : "Given an instance I, does I have a property P where P is dependent on k." It is easy think of natural examples of problems with this form, examples could be "Does a graph G have a DOMINATING SET of size k" or "Does a graph G have an INDEPENDENT SET of size k".These problems are known as parameterized problems:

Definition 2.1 *Parameterized languages*

A parameterized language L *is defined as* $L \subseteq \Sigma^* \times \Sigma^*$. *If* $< x, k >$ *is in a parameterized language* L *we call* k *the parameter. The parameter can be of any type, but we can without loss of generality restrict the parameter to the natural numbers* N. *Thus we refer to the parameterized problems as :* $L \subseteq \Sigma^* \times N$. *For a fixed* k, *we call* $L_k = \{< x, k > \mid < x, k >\in L\}$ *the* k*th* slice *of* L

2.2 The Idea

We will now try to find good algorithms for the parameterized problems. We already know that for an NP-complete problem we cannot get a fully polynomial algorithm for all values of the parameter, as this would lead directly to a polynomial time algorithm for the unparameterized problem (impossible if P≠NP). It is often easy to find $O(n^k)$ algorithms with a brute-force tactic,but these algorithms are usually uninteresting as the running time "explodes" when we increase either the main input or the parameter.

With a leap of imagination, however, it is possible to find algorithms with running times in the order of $f(k)n^{O(1)}$. These algorithms are much more interesting as increasing the main input size leads only to a polynomial increase in running time. We can even claim, that for *fixed* values of the parameter, we have a *polynomial* algorithm for the problem.

This may sound like a major breakthrough in complexity theory. It is indeed a major breakthrough, but it is not a reason to call in the marching bands, as the function $f(k)$ is invariably (as long as P≠NP) exponential. Not all problems have algorithms of this type either, this leads to a partitioning of the NP-complete problems. Those problems that have algorithms of type $f(k)n^{O(1)}$ are known as *Fixed Parameter Tractable* (FPT).

2.3 Fixed Parameter Tractability

A parameterized problem is considered fixed parameter tractable if there exists an algorithm that solves the problem and the running time is not exponential with regards to the main input.

Definition 2.2 *Fixed Parameter Tractable*

A parameterized language $L \subseteq \Sigma^* \times N$ is Fixed Parameter Tractable $\iff$ *There is an algorithm A, a constant α and a function $f : N \rightarrow N$ s.t:*

(a) *the running time of $A(< x, k >)$ is at most $f(k)|x|^\alpha$.*

(b) $< x, k > \in L \iff A(< x, k >) = 1$

2.4 The negative results

The problem with some of the methods mentioned at the end of chapter 1 is that they lack a way to prove negative results. Fixed Parameter Complexity, however, does not have this deficiency. FPC has its own completeness theory

not unlike the one introduced for NP in chapter 1. We will give a more thorough presentation of this side of FPC in chapter 4.

2.5 Theory or Practice

Fixed Parameter Complexity has been proclaimed as one of the ways to tackle NP-hard problems, and indeed it is. Some problems have been shown to have extremely good FPT-algorithms, allowing them to be solved efficiently for quite high values of the parameter. One of the best examples of such problems is the NP-Hard VERTEX COVER where the best known running time is $O(1.271^k + kn)$ [10, 15] making it useful for k as high as 200.

Proving that a language is FPT is, however, not a guarantee that Fixed Parameter Complexity can be utilized on this problem in practice. In addition to the usual traps when dealing with asymptotic complexities (high constants and high polynomials) we have the possibility that the function $f(k)$ is too large. Even after "significant" improvement, the first FPT-algorithm for VERTEX COVER had a $f(k)$ in the order of $f(k) = 2^{2^{500k}}$. This algorithm is clearly unusable for even $k = 1$.

2.6 History of Parameterized Complexity

There exists little or no written material on the history of parameterized complexity, but from what we have been able to piece together it seems that parameterized complexity owes its intellectual origins at leasts in part from algorithmic implications of the now famous Robertson-Seymour theorems on *graph minors.*

Definition 2.3 *Graph Minor*

A graph H is a minor of graph G, $H \leq_m G$, if a graph isomorphic to H can be obtained by repeatedly contracting edges of a subgraph of G. Contracting edge (u, v) means identifying u and v as a single vertex maintaining all neighbours.

Definition 2.4 *Obstruction Set*

Let F be a family of graphs closed under minors, that is if $G \in F \wedge H \leq_m G$ then $H \in F$.

The obstruction set for F,denoted by $OS(F)$ is defined to be the minor-minimal elements in the complement of F , that is $OS(F) = \{M : M \notin F \wedge (\forall G \notin \{M\} \cup F : G \not\leq_m M)\}$. It follows that $G \in F$ if and only if there does not exists $M \in OS(F)$ such that $M \leq_m G$, in other words $F = \{G : \forall M \in OS(F) : M \not\leq_m G\}$.

Robertson and Seymour finally proved Wagner's Conjecture which had been about since the early sixties. Wagner stated that for every infinite set of graphs $G1, G2, ...$ there exists $i < j$ such that $G_i \leq_m G_j$. The relation $\leq_m$ on finite graphs is reflexive, transitive and anti-symmetric and the conjecture says that the minor relation defines a well-partial-order on the class of all finite graphs, namely there exists no infinite descending chain (trivially true) and there is no infinite anti-chain. In other words, for any family of graphs F closed under minors the set $OS(F)$ is finite.

To prove this Robertson and Seymour introduced the concept of *tree-decomposition* and *tree-width*, and first showed that Wagner's conjecture held for bounded treewidth graphs.

Definition 2.5 *Tree-decomposition*
A tree-decomposition *of a graph $G = (V, E)$ is a pair*

$$(\{X_i \mid i \in I\}, T = (I, M))$$

where $\{X_i \mid i \in I\}$ is a collection of subsets of V, and T is a tree, such that:

1. $\bigcup_{i \in I} X_i = V$
2. $(u, v) \in E \Rightarrow \exists i \in I$ *with* $u, v \in X_i$
3. *For all vertices $v \in V, \{i \in I \mid v \in X_i\}$ induces a connected subtree of T.*

Definition 2.6 *Tree-width*
The width of a decomposition $(\{X_i \mid i \in I\}, T = (I, M))$ is $max_{i \in I}\{X_i - 1\}$. The tree-width *of a graph G is the minimum width over all tree-decompositions of G.*

It is in the algorithms for finding tree-decompositions of width $\leq k$ for fixed values of k that we can see the origin of parameterized complexity. The first algorithm for solving the problem [2] had time complexity $O(n^{k+2})$ and was based on dynamic programming. Then Robertson and Seymour introduced a divide-and-conquer algorithm [13] with running time $O(n^2)$, a remarkable improvement. At first it appears that the parameter no longer affects running time, but this is just O-deception. Since the parameter is fixed, O-notation hides the exponential $f(k)$ function in Robertson's and Seymour's algorithm.

The tree-width algorithms have been improved several times since then, but it was the jump from $O(n^{O(k)})$ to $O(n^{O(1)})$ that stirred the interest of

Fellows and Langston. Fellows and Langston observed that a similar algorithm could be devised for Vertex Cover[9]. Other results soon followed and Fellows with Downey explored the structures of Parameterized Complexity. Since then an ever growing body of scientists have contributed to the field.

In the last 10 years the discoveries and improvements in this field have been enormous. The algorithms for many problems have been improved to a point where they serve as the best algorithms we have for these problems. The future is promising indeed.

Chapter 3

Positive Techniques

3.1 Introduction

When designing algorithms for FPT we have to think differently than when designing algorithms for polynomial problems. Normally we avoid brute-force methods because we know that they will very often require exponential time. When we deal with **FPT** we already assume that some part of the problem involves exponential running time and we instead concentrate the effort in not avoiding, but *limiting* the exponential growth.

Since FPT is a relatively new field there is still a lot to be done in terms of defining what algorithmic techniques we have and what they can do. Of the techniques we do have, some are extremely technical and not all of them are practical.

In this chapter we will concentrate on two of the simpler methods which exhibit good practical results. We will give examples on the use of both techniques and show that they can be combined to give even better results. We will also give a brief introduction to other known techniques, but we will not give detailed examples of these.

3.2 Bounded Search Trees

This method is in itself easy to apply and gives good results on many useful problems. In fact one of the best known **FPT**-algorithm on VERTEX COVER is based on an algorithm using *Bounded Search Trees*.

All combinatorial problems can be solved by creating a search-space with all possible combinations. This is usually an exponential sized tree. The resulting algorithm is, obviously, not polynomial if we have to search the entire tree for a solution.

When applying the method of bounded search trees, the idea is to create a search-tree that in size *only depends* on the parameter k. Then if k is fixed the search-tree becomes constant in size and the search is efficient.

We will demonstrate how to use this method on the NP-complete problem DOMINATING SET. The general DOMINATING SET is not in **FPT** but many variations of it are. We will give an algorithm for the problem **DOMINATING SET ON BOUNDED DEGREE GRAPHS**.

Let $deg(v)$ be the degree of vertex v.

3.2.1 Dominating Set on bounded degree graphs

Input:	A Graph G,s.t.$\forall v \in V(G), deg(v) \leq c$, for a constant c
Parameter:	$k, k \in \mathbb{N}$.
Question:	Does G have a dominating set of size $\leq k$? (A *Dominating Set* is a set $V' \subseteq V(G)$ such that $\forall u \epsilon V(G), \exists v \in V'$ such that $uv \in E(G)$ or $u \in V'$)

Theorem 3.1 DOMINATING SET ON BOUNDED DEGREE GRAPHS *is solvable in time* $O((c+1)^k|V(G)|)$*, where c is the maximum degree of G.*

Proof. We will construct a search tree where each node has a maximum of $c+1$ children and we will mark each node in the tree with a set S which is the partial dominating set (DS) constructed at each step. Begin with a root-node marked with the empty set. Then select a vertex $v \in V(G)$ at random and argue as follows: In a complete DS, v must be dominated by a vertex $u \in V(G)$: Either $v = u$ or $uv \in E(G)$. We create one child for each of these possibilities and update S with the appropriate value of u. Since we are guaranteed that the vertex we selected cannot have more than c neighbors the root-node will have no more than $c+1$ children.

Then for any node in the tree except the root-node: Mark all vertices in G that are either in S or dominated by S. Select an unmarked vertex $v \in V(G)$ and repeat the argument above. Observe that v will have no neighbors in S, but can have neighbors that already are dominated. This implies that we will never add a vertex to S that is already a member of S. Again create a child for each possible value of u, mark the children with $S \cup \{u\}$. The selected vertex v will have $deg(v) \leq c$ and the tree-node will therefore have no more than $c+1$ children. (An example tree can be seen in figure 3.1)

For each new level in the tree, the size of the partial DS increases by one. On the height k, each tree-node will have a partial dominating set of cardinality k. Moreover, if a DS S^* of size $\leq k$ exists we will have found it

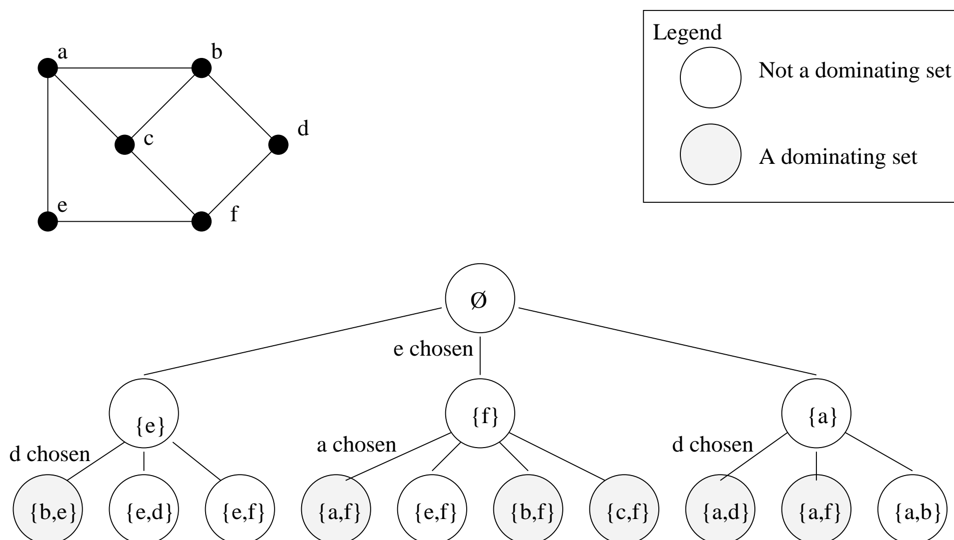


Figure 3.1: Example of a bounded search tree on a graph

by height k, since each vertex v selected randomly at each level of the tree must be dominated by some vertex in S^*, and the algorithm has tried all possibilities and then marked all subsequentially dominated vertices so they will not be selected later. The height of the tree is therefore limited to k and the fanout of each node in the tree is always $\leq c+1$. The total tree-size becomes $(c+1)^{k+1}$ with $(c+1)^k$ leaves. We can test the set on each leaf in linear time, yielding an algorithm with the running time: $O((c+1)^k|V(G)|)$. □

The algorithm can be seen on table 3.1.

3.2.2 Minimum DS on bounded degree Graphs

Corollary 3.2 *Minimum DS on bounded degree graphs is solvable in time* $O(V(G)(c+1)^{k+1})$ *where* k *is the size of the smallest DS in the input-graph.*

Proof. This immediately follows from the theorem; Instead of testing each leaf after the tree has reached a height of k, test each leaf after each step in the algorithm. If no DS is found in the leaves, increase the height of the tree by one and search again. This will always discover the smallest (one of the smallest) DS in the graph. □

Table 3.1: Algorithm for dominating set on bounded degree graphs

```
INPUT A graph G(V,E), where ∀v ∈ V, deg(v) ≤ c
PARAMETER Integer k
BEGIN Algorithm

Create Tree-node T.
T.Set = Empty    REM The partial dominating set
Integer h = 0    REM The height of the tree so far.

while h ≤ k
    For all leaves L in T
        For all vertices in v ∈ V
            IF v ∈ L.Set or (vw) ∈ E(G), w ∈ V
                Mark v
        End For
        u = random unmarked vertex.
        For all x neighbors to u
            Add a child-node M to L.
            M.Set = L.Set + {x}
        END For

        Add a childnode M to L.
        M.Set = L.Set + {u}

        Clear all marks in V.
    END For

    h = h + 1
REPEAT While

For all leaves L in T
    IF L.Set is Dominating Set in G.
        Return YES
END For

Return NO
END Algorithm
```

3.3 Reduction to problem kernel

Reduction to problem kernel is another straightforward tool for creating **FPT** algorithms. Although the algorithms are of a more complex nature than

Bounded Search Trees it usually yields algorithms with *additive* rather than *multiplicative* exponential function. The idea is to reduce the problem to a *problem kernel*, a smaller version of the problem were we can decide the answer for the original problem. If the size of the problem kernel is limited by the parameter k, then we can obtain a solution by solving the reduced problem with a brute-force algorithm.

These algorithms operate in two distinct steps:
1. Reduce the problem from a size-n problem to a size-$f(k)$ problem.
2. Solve the size-$f(k)$ problem using a brute-force algorithm.

If step #1 can be done in polynomial time we have a FPT-algorithm with running time of type $O(g(n) + h(k))$ where $g(n)$ is the polynomial algorithm for reducing the problem and $h(k)$ the exponential algorithm for solving the reduced problem. Observe the additive nature of the exponential function, a pleasing attribute that allows us to accept a larger exponential function while still keeping the running time low.

We will give an example of the problem-kernel-technique on the classical **NP**-complete problem VERTEX COVER.

3.3.1 VERTEX COVER

Input:	A Graph $G = (V, E)$,
Parameter:	$k \in \mathbb{N}$.
Question:	Does G have a Vertex Cover of size $\leq k$?
	(A vertex cover of a graph G is a subset $V' \subseteq V$
	where $\forall v_1 v_2 \in E$ either $v_1 \in V'$ or $v_2 \in V'$)

Theorem 3.3 *(Sam Buss[6])* VERTEX COVER is solvable in time $O(V(G)+ k^k)$

Proof. First we claim that when we select vertices for a vertex cover in a graph G and we do not select a vertex $v \in G$, then we must select all the neighbors of v in G.

It is easy to prove this claim by contradiction: Assume that we can form a vertex cover in G without selecting v or u and that v and u are neighbors ($vu \in E(G)$). Then we cannot have covered the edge vu and the assumption fails.

We use this claim above to argue that we must select all vertices with degree $> k$. If we do not select a vertex with degree $> k$, then we must select it's more than k neighbors and the VERTEX COVER becomes too large.

Now let $S = \{v \in V \mid deg(v) > k\}$. If $|S| > k$ then we reject, because S represents a set of vertices that must belong to any k-Vertex Cover of G. Let $k' = k - |S|$. k' is the number of vertices we have yet to select to our VERTEX COVER.

Let $H = G[V - S]$, the induced subgraph on $V - S$, which may contain vertices of degree 0. These vertices are never part of any VERTEX COVER and may be discarded from H. Let $H' = H - \{x \in H \mid deg(x) = 0\}$.

Now H' is limited to vertices of degree $\leq k$. Any one vertex we select from the graph could not cover the edges to more than k other vertices. Since we cannot select more than k' vertices, we cannot cover any graph with more than $k'(k+1)$ vertices, where $k' \leq k$. That is; if $|H'| > k'(k+1)$ then reject.

The problem has been reduced to finding a k'-VERTEX COVER in a graph of size $\leq k'(k+1)$. This can be done in $O(k^k)$ time by testing each of the $\binom{k'(k+1)}{k'}$ possible VERTEX COVERs.

The running time of the algorithm is, $O(n)$ to reduce the graph and then $O(k^k)$ to brute-force the solution from the reduced graph. Total of : $O(n + k^k)$.$\square$

From the above proof we deduce the algorithm in Table 3.2.

3.4 Combining the methods

Since the reduced kernel of a problem often is a problem of the same type or a similar **FPT**-problem we can avoid using a plain brute-force tactic to solve the kernel. Using a **FPT**-algorithm to further limit the use of brute force gives us a better exponential function. We will demonstrate this here by solving the kernel-problem from section 3.3 with a bounded search-tree algorithm for VERTEX COVER.

Theorem 3.4 *(Downey and Fellows [7])* VERTEX COVER *is solvable in time* $O(2^k|V(G)|)$

Proof. We will construct a binary tree where we label each node with a set $S \subseteq V$ and a graph H. S will be the partial VERTEX COVER constructed at each step of the algorithm and H will be the remaining graph at each step. Start with a root-node labeled with $S = \emptyset$ and $H = G$. Expand the binary tree as follows: Pick any edge $uv \in E$. This edge must be covered and this can be done by either selecting u or selecting v for the VERTEX

Table 3.2: Algorithm for Vertex Cover (Problem Kernel)

```
Input: Graph G(V,E), Integer k
BEGIN Algorithm

Integer c = 0. REMARK: Counting removed nodes

For all v ∈ V
    IF Degree(v) > k THEN
        Remove v from G.
        Remove all edges incident to v.
        c = c + 1
END For

IF c > k THEN
    RETURN No
END If

For all v ∈ V
    IF Degree(v) = 0 THEN
        Remove v from G.
    END If
END For

Let k' = k − c

IF |V| > k'(k + 1) THEN
    RETURN No
END If
REMARK: Brute force
For all possible size k' subsets S of V.
    IF S is VERTEX COVER of G THEN
        RETURN YES
    END If
END For

RETURN NO
END Algorithm
```

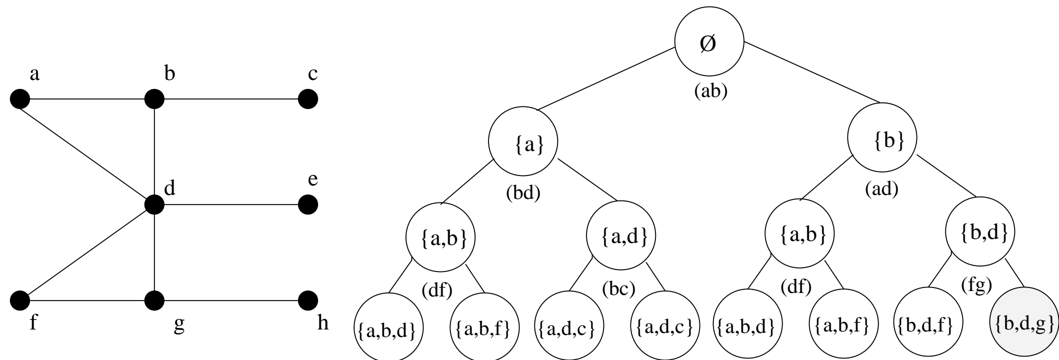


Figure 3.2: Example: A bounded search-tree for vertex cover(grey node is a vertex cover)

COVER. Add a child-node for each of the two possibilities (The binary property of the tree is apparent) labeling the two child-nodes with $S \cup \{u\}$ and $S \cup \{v\}$ respectively. Also mark the child-nodes with the remaining graph ($H' = H - \{v\}$ and $H' = H - \{u\}$ respectively). (An example of a tree can be seen in figure 3.4)

Repeat the expansion for each leaf in the tree in a breadth first order until we have created a complete tree of height k. Since we add one vertex to the partial solution S at each expansion the size of S at level k will also be k. It is clear that if there exists a VC S^* with size $\leq k$ we will have found it at level k, this is because the algorithm at each level picks an uncovered edge and tries both possibilities for covering it.

If any of the leaves is labeled with a graph H containing no edges, then a vertex cover of size $\leq k$ has been found (The corresponding value of S at that leaf).

The complexity of this algorithm is as follows: The cost of updating each node is linear in terms of the number of vertices in the graph. The number of nodes are 2^k which gives the total running time of $O(2^k|V(G)|)$.□

From the above proof we can deduce another algorithm for solving VERTEX COVER, the new algorithm can been seen in table 3.3.

Theorem 3.5 *(Balasubramanian, et al.[4]) VERTEX COVER can be solved in time $O(V(G) + 2^k k^2)$.*

Using the algorithm from section 3.3 we can reduce a $|V(G)|$-sized vertex cover problem to a $k(k + 1)$-sized vertex cover problem in time $O(V(G))$.

Then using the above algorithm on the reduced problem we get an algorithm that runs in time $O(V(G) + 2^k k^2)$. □

3.5 Other techniques

In this section we will give a very brief listing of other known techniques for proving FPT.

3.5.1 Well-Quasi-Ordering

The first general method for demonstrating fixed parameter tractability is based on the Robertson-Seymour results on graph minors and well-quasi-orderings. The proofs for these results run over 1500 pages and is very technical in nature. Well-quasi-ordering is a powerful tool, but the results are impractical as hidden constants of astronomical values frequently appear. The exponential parameter functions have improved, but this technique remain as a classification technique rather than a method for designing algorithms.

3.5.2 Bounded Tree-width and Path-width

Many problems are known to be solvable in polynomial time if we know their tree-width. The best known algorithm for computing tree-decomposition of width k runs in time $f(k)n$ (Observe the FPT-quality) where $f(k) = 2^{ck^2}$, due to Bodlaender [5]. Unfortunately this parameter function is too impractical and the algorithm has, as far as we know, not been implemented.

3.5.3 Color Coding

This general method is based on the use of k-perfect families of hash-functions derived from sophisticated derandomization techniques. It was developed by Alon, Yuster and Zwick [1]. This method works well for certain kinds of problems,such as finding small subgraphs in a graph. The results from this technique are often very practical in nature with low $f(k)$ functions and no hidden constants.

Table 3.3: Algorithm for solving Vertex Cover (Bounded Search-tree)

```
Input: Graph G(V,E), Integer k
BEGIN Algorithm

T = Root node in Tree.        REM The partial Vertex Cover
T.Set = ∅     REM The remaining graph
T.Graph = G

Integer c = 0. REMARK Height of tree

WHILE c ≤ k
    For all leaves L in Tree T
        Select edge uv from L.Graph at random
        Add a child-node C1 to L
        C1.Set = L.Set + {u}
        C1.Graph = G − {u}
        Add a child-node C2 to L
        C2.Set = L.Set + {v}
        C2.Graph = G − {v}
    END For

    c = c + 1
END While

For all leaves L in Tree T
     IF |E(L.Graph)| = 0 THEN
         RETURN YES
     END IF
END For

RETURN NO
END Algorithm
```

Chapter 4

Negative Techniques

The parameterized complexity theory deals mainly with NP-complete problems. Using the techniques mentioned in chapter 3 (and several others) we can prove that some of these problems are in FPT. There are, however, many problems where these techniques do not seem to work. Do we lack some important tool to prove that they all really are in FPT or can we prove that some of these problems are not fixed parameter tractable? This chapter deals with this question.

4.1 Completeness Theory For PC

Although we cannot prove it, the current conjecture is that some problems are not fixed parameter tractable. We support this conjecture the same way we support the conjecture that P≠NP, we build a completeness theory for it. By proving that there exist a problem which is related to *all* other problems in W[1..t] (the parameterized complexity classes analogous to NP) we create the first stepping stone for a very powerful argument. By showing that this complete problem is in FPT only if other problems are in FPT as well, we create a ring of problems with the property that if we can prove one in FPT all others are automatically in FPT(see observation 4.3). The argument is then that since some of these problems are so unstructured and "wild" there is very little chance they are in FPT and thus all the other complete problems cannot be in FPT.

The completeness theory for FPT is well under way and the classes of problems probably not in FPT are known[1] as $W[1] \subseteq W[2] \subseteq ... \subseteq W[t]$. The book "Parameterized Complexity"[8] contains a listing of problems in

[1] The origin for this naming is a technicality which is left to the interested reader, see the book "Parameterized Complexity"[8] for detailed information.

these classes.

4.2 Short Turing Machine Acceptance

As stated in the introduction there are many problems which we have not been able to prove to belong to FPT. Indeed some of the problems are so bizarre and unanalyzable that it seems unlikely that they are in FPT. One of these problems, which also may be viewed as the "core" problem in the completeness theory is known as the SHORT TURING MACHINE ACCEPTANCE (STMA).

Input:	A Nondeterministic Turing machine M.
Parameter:	$k, k \in \mathbb{N}$.
Question:	Is it possible for M to reach a halting state in at most k steps when started on an empty input tape?

This problem is trivially solvable in time $|M|^k$ by exploring all k-length execution paths, but as the execution pattern of Turing Machines vary as wildly as the ideas of the programmer it seems extremely unlikely that this problem has a pattern that can be exploited to get a fixed parameter algorithm.

Downey and Fellows has shown that any problem in W[1] can be reduced to STMA, in fact the proof for this is a miniaturization of Cook/Levin's proof for SAT. Moreover, they show that STMA$\in W[1]$. Similar problems exists for the other W[..] classes.

4.3 Polynomial time reductions

With one complete problem in place we can show that other problems are complete by performing reductions. This entails taking a complete problem A and "transforming" it to a new problem B. If this transformation can be done in polynomial time, then we say that A is *polynomial time reducible* to B. This implies that if the new problem B has a good algorithm then so must A since we can transform A to B. Since any language in the class reduces to the complete problem A, the whole class would have good algorithms.

Definition 4.1 *Polynomial time reducible*

Language A is polynomial time reducible *to language B, written $A \leq_p B$ if a polynomial time computable function $f : \Sigma^* \rightarrow \Sigma^*$ exists, where for every w,*

$$w \in A \iff f(w) \in B.$$

The function f is called the polynomial time reduction *of A to B. The following figure illustrates polynomial time reducibility.*

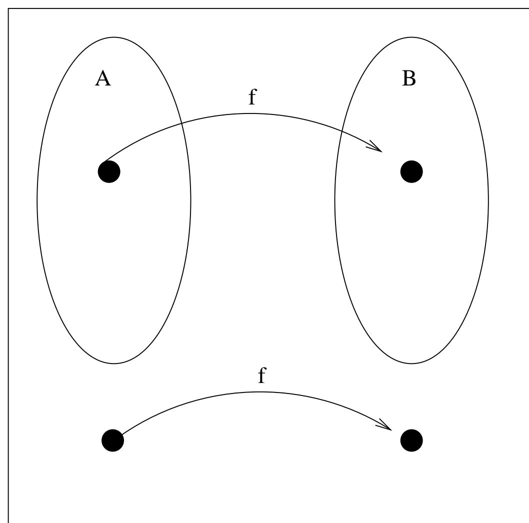


Figure 4.1: Polynomial time function f reducing A to B

4.4 Reductions in FPC

Reductions in FPC is very similar to pure polynomial time reductions. We require that the reduction can be done in polynomial time, but we also impose requirements on the parameter. It is important to keep the separation between the parameter and the main input.

Definition 4.2 *Parametric reduction*

A parametric reduction from a parameterized language L to a parameterized language L' is an algorithm that computes from input consisting of a pair (x, k), a pair (x', k') such that:

1. $(x, k) \in L \iff (x', k') \in L'$,
2. $k' = g(k)$ *is a function only of* k, *and*
3. *the computation is accomplished in time* $f(k)n^{O(1)}$, *where* $n = |x|$, *and* f *is an arbitrary function.*

Observation 4.3 *If L' and L are parameterized languages and L reduces to L' with the polynomial time function $f(k)$ for the main input and the arbitrary function $g(k)$ for the parameter input, then $L' \in FPT$ implies $L \in FPT$.*

Proof. If L' is in FPT there must exist an algorithm A that runs in time $h(k)n^{\alpha}$ that decides the language. We can transform any instance $< x, k >$ of L to an instance (x', k') of L' in polynomial time using the functions f and g. We can now devise an FPT-algorithm for L by first transforming it to L' and then deciding membership with algorithm A. The running time is $O(h(g(k))(f(n))^{\alpha})$ plus the polynomial time for the reduction, thus still in FPT.$\square$

Chapter 5

Automata

5.1 Using Automata

The Finite state Automaton (FA) is one of the simplest computational models, yet it is the source of a powerful method that exploits the computational powers of modern computers. This technique is called the method of test sets and the rest of the thesis will deal mainly with this technique.

This chapter will introduce the Finite Automaton and give background information leading up to the Method of Test Sets. Included are two Lemmas (Lemmas 5.10 and 5.9) that are crucial for the Method of Test Sets, but whose proofs were lacking in the textbook on parameterized complexity and have not appeared before in the litterature, as far as we know. The Method of Test Sets is then introduced with an example and analysis on its running time.

5.2 The Deterministic Finite Automaton

The deterministic finite automaton is a simple but powerful machine, it can be defined using a quintuple:

Definition 5.1 *Deterministic Finite Automaton (DFA)*

$M = (K, \Sigma, \delta, S, F)$,where

K	is a finite set called the set of states,
$S \in K$	is called the *initial* state,
$F \subseteq K$	is called the set of *accepting* states,
Σ	is the alphabet of the machine, and
$\delta : K \times \Sigma \mapsto K$	is a function called the *transition* function.

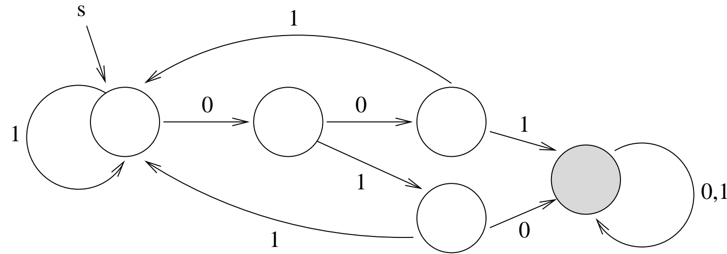


Figure 5.1: A DFA on the binary alphabet which will halt in the accepting state iff the input-string contains 001 or 010.

We consider DFA's to be physical entities which given an input-string s from the alphabet Σ will halt in an accepting state iff the string s is in the language defined by the DFA.

The machine will move from state to state according to the input string and the transition-function (The function is often represented with a graphical drawing of the DFA,see figure 5.1). It will move one step at a time, evaluating one symbol from the input-string s at each step and halt when it has evaluated all symbols in s. If the state where the machine halts is an accepting state then the input string is considered to be in the language described by the DFA. We can use this to create the following FPT-problem:

5.2.1 Deterministic Finite Automaton Acceptance

Input:	A string $s \in \Sigma^*$.
Parameter:	M, a DFA over Σ^*
Question:	Is $s \in L(M)$?

This problem is in FPT, since we can run the automaton M on the string s and obtain the answer in time $O(|s|C(\delta))$. Here $C(\delta)$ is the cost of computing the answer for the δ-function. The δ-function is usually represented as a table-lookup but other constructs could be imagined. However, even if our δ-function requires exponential time, the problem is still in FPT as the cost of the δ-function depends only on the parameter.

5.3 The Nondeterministic Finite Automaton

We will now introduce a variant of the DFA called the Non-deterministic Finite Automaton.

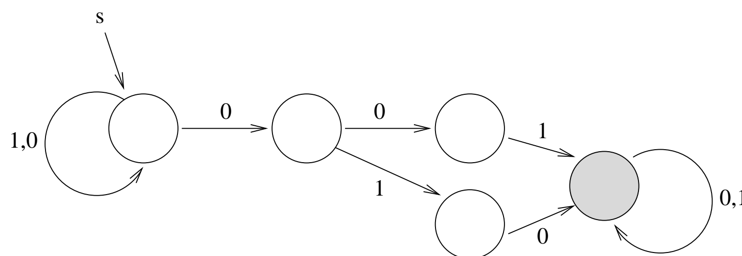


Figure 5.2: An NFA which halts in the accepting state iff the input string contains 001 or 010

Definition 5.2 *Nondeterministic Finite Automaton (NFA)*

The nondeterministic automata is a quintuple $M =< K, \Sigma, \Delta, S, F >$. *where* K,S,F, *and* Σ *are the same as for a deterministic automaton, but* $\Delta \subseteq K \times \Sigma \times K$ *is a relation called the* transition *relation.*

Instead of having the next state uniquely determined by the input-symbol and the transition function, the machine has now the option of choosing between states (See figure 5.2). The machine accepts a input-string s iff there exists some series of choices that will lead to an accepting state.

At first glance this machine seems much more powerful than the DFA, since it has the possibility of testing many different computational paths at each step.

It has, however, been shown by Rabin and Scott [12] that they are equivalent.

Theorem 5.3 *Every nondeterministic finite automaton is equivalent to a deterministic one. Furthermore, if M is a (nondeterministic) machine with n states, then we may compute in* $O(2^n)$ *steps a deterministic machine* M' *with at most* 2^n *states.*[1]

This can be used in the following FPT-problem:

5.3.1 Nondeterministic Finite Automaton Acceptance

Input:	A string $s \in \Sigma^*$.
Parameter:	M, a NFA with n states over Σ^*
Question:	Is $s \in L(M)$.

[1] We will not give the proof here, but the proof can be found in any book on automata-theory.

Theorem 5.4 *Nondeterministic Finite Automata Acceptance can be solved in time $O(2^n + |s|C(\delta))$ and is in FPT, here $C(\delta)$ is dependent on the parameter.*

Proof. By theorem 5.3 we could compute a DFA M' accepting the same language as M in time $O(2^n)$. Then we run M' on s to obtain the answer in time $O(|s|C(\delta))$. The total running time is thus $O(2^n + |s|C(\delta))$ and is in FPT since the exponential 2^n and the arbitrary $C(\delta)$ only depends on the parameter M.□

5.3.2 Nondeterministic Finite Automaton Acceptance II

Input:	M, an NFA with n states over Σ^*
Parameter:	A string $s \in \Sigma^*$.
Question:	Is $s \in L(M)$.

Observation 5.5 *Nondeterministic Finite Automaton Acceptance II can be solved in time $O(2^n + |s|C(\delta))$ which is exponential in running time.*

Proof. The problem is, from the view of the computer, essentially the same as Nondeterministic Finite State Acceptance. We can still build a deterministic machine M' in time $O(2^n)$ and run M' on s. The total running time is the same as the problem above, but now the exponential running time depends on the main input. The algorithm has exponential running time and therefore does not show membership in FPT. □

This is a very good example of what can happen if we switch the parameter and main input. Two problems with identical running time are not necessarily in the same complexity class. It is not necessarily obvious what should be the main input and what should be the parameter when analyzing a problem and the choice is, as we have seen, of great importance when it comes to the complexity analysis of the problem.

5.4 The Myhill-Nerode Theorem

The Myhill-Nerode theorem (by Nerode [11]) is of importance when we are trying to determine whether or not a specific problem is solvable by Finite State Machines. It can, however, also be used to create minimal machines for

problems where only an oracle and an upper bound on the number of states are known. This technique for creating a machine is called the Method of Test Sets.

First we need a definition of right congruence:

Definition 5.6 *Right Congruence*

(a) We call a binary relation R on a set Σ^* a *right congruence* (with respect to concatenation) iff

  (i) R is an equivalence relation on Σ^*

  (ii) $\forall x, y \in \Sigma^*, xRy$ iff $\forall z \in \Sigma^*, xzRyz$.

(b) For a language L, the *canonical right congruence*(induced by L) is the relation $\sim_L$ defined by

$$x \sim_L y \text{ iff for all } z \in \Sigma^*, xz \in L \text{ iff } yz \in L.$$

In short, a string x is related to a string y iff there is no string of length z that makes xz a member of L and yz not a member of L (or vice versa). We say that an equivalence relation R has finite index if it has only a finite number of equivalence classes.

Theorem 5.7 *Myhill-Nerode*

(a) *The following are equivalent over* Σ^*.

  (i) *L is a finite state language, accepted by a DFA.*

  (ii) *L is the union of a collection of equivalence classes of a right congruence of finite index over* Σ^*.

  (iii) $\sim_L$ *has finite index.*

(b) *Furthermore, any right congruence satisfying (ii) is a refinement*[2] *of* $\sim_L$.

We will not give the proof here, but we will give examples of how to use the theorem. First we will use it to prove that a language is not regular (a negative result) and then we will concentrate on the method of test sets, a technique to design finite state machines given an oracle to the regular language.

[2] An equivalence relation A is a refinement of an equivalence class F iff the equivalence classes of F are unions of equivalence classes of A

5.5 Negative results by Myhill-Nerode

The traditional way to prove that a language is not regular is to apply the pumping lemma technique. A straight forward and simple technique, but still more complicated than the following method.

We can use item (i) and (iii) of Myhill-Nerode to prove that a language is not a regular language.

$$L \text{ is finite state } \iff \sim_L \text{ has finite index}$$

If we can prove that $\sim_L$ has infinite index, then Myhill-Nerode implies that the language has an infinite number of equivalence classes and thus the language is not regular. We will give and example using the classical context-free language:

$$L = \{a^n b^n \mid n \geq 0\}$$

Theorem 5.8 $L = \{a^n b^n \mid n \geq 0\}$ *is not regular.*

Proof. $\forall n, m$ where $n \neq m$ we have $a^n \not\sim_L a^m$ because for $z = b^n$ then $a^n z \in L$ but $a^m z \notin L$. Thus, the language has an infinite number of equivalence classes and is (by Myhill-Nerode) not regular.□

5.6 Method of Test Sets

The method of test sets is actually a technique for solving the so-called "promise-problem".

The promise-problem

Input: A decision procedure ("oracle") for a regular language L.
Promise: L is regular and there is some DFA M accepting L with at most k states.
Question: Compute a DFA M' accepting L.

5.6.1 Why the parameter?

It is tempting to ask if we really require the bound on the number of states given as the parameter here. As the textbook "Parameterized complexity"[8] states, the answer is YES. If there were such an algorithm computing a DFA from simply an oracle, i.e. without the promise, then we could solve "the

Halting problem"[3], which is inherently unsolvable (by Turing). The proof for this, however, is not as simple as the one attempted in the textbook.

Lemma 5.9 *In general, there is no algorithm to construct an automaton for a regular language from a decision procedure for the language.*

Proof. Consider the encoding x of any Turing Machine. Let UTM be the Universal Turing Machine, a Turing machine capable of taking the description of a Turing machine as input and simulating it on a blank tape[4].
Define the language $L_x = \{1^k | \text{UTM halts on input } x \text{ in at most } k \text{ steps}\}$. If the UTM on x never halts then the language L_x is empty, otherwise it is a set, $L_x = \mathbb{N} - \{1^0, 1^1, .., 1^k\}$. Every L_x is obviously regular since it can be represented by an automata of the type seen in figure 5.3.

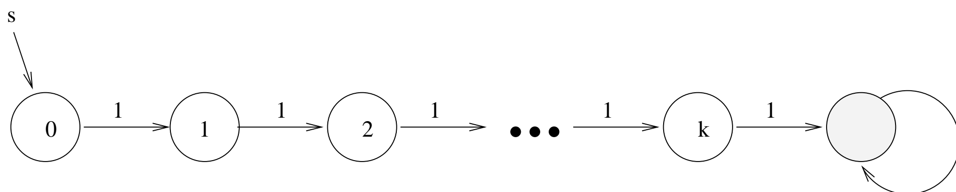


Figure 5.3: A finite state automaton accepting $L_x = \mathbb{N} - \{1^0, 1^1, .., 1^k\}$.

We can construct a decision procedure (Turing machine) A to test membership in L_x. Construct A by modifying the UTM, let UTM take x as input and in addition let it take a string $y = 1^t$, if A halts on x in less than t steps then $y \in L_x$ and otherwise $y \notin L_x$. Since A will always halt in $O(t)$ steps it is a decision procedure for L_x.

Now assume in contradiction to the Lemma that we can construct the finite automaton F_x for L_x. Then we have the equivalence:

UTM does not halt on $x \iff L_x$ is empty $\iff F_x$ has no reachable final state.

Since it is easy to check if F_x has a reachable final state, we can decide if UTM will halt on x, thus contradicting the unsolvability of the Halting problem. The assumption must be wrong and hence the Lemma holds. □

Note that in the above proof there is no bound on the states in F_x. As we shall see in section 5.6.4, if we are armed with a bound on the states we can construct an automaton for regular languages.

[3]See any book on complexity theory for a description and proof of the halting problem. We recommend "Introduction to the Theory of Computation"[14]

[4]The Halting Problem relates to the language {x : TM x halts on blank tape input}.

5.6.2 Only short strings are required

Using the promise that L is regular and the Myhill-Nerode Theorem we conclude that $\sim_L$ has finite index. Now we can show that we can determine the equivalence classes with strings of length at most $k-2$. The proof of this Lemma is crucial for the Method of Test Sets, but to our knowledge it has not appeared in full in the litterature.

Lemma 5.10 *Assume $L \subseteq \Sigma^*$ is recognized by some FSM $M(Q, \Sigma, \delta, F, q_0)$. Then $\forall v, u \in \Sigma^*$; if $v \not\sim_L u$ then $\exists z \in \Sigma^*$ s.t. $vz \in L \iff uz \notin L$ where $|z| \leq |Q| - 2$.*

Proof We define the relation $\delta^*(q, w)$ by induction. $\delta^*(q, a) = \delta(q, a)$ and $\delta^*(q, av) = \delta^*(\delta(q, a), v)$

We define inductively the sequence of equivalence relations $\sim_0$, $\sim_1$, $\sim_2$,..., $\sim_k$ over Q as follows:
Base case: $\forall q, q' \in Q$ we have $q \sim_0 q'$ iff $q \in F \iff q' \in F$.
Inductive step, $i > 0$: $\forall q, q' \in Q$, $q \sim_i q'$ iff $q \sim_{i-1} q' \wedge \forall a \in \Sigma\ \delta(q, a) \sim_{i-1} \delta(q', a)$.

This will gradually refine the equivalence classes until $\sim_{k+1} = \sim_k$. We can see that this stabilization will occur for a value $k \leq |Q| - 2$. We can safely ignore the special case when there are only rejecting or accepting states. These languages can be represented by a single state, and there exists no separating strings. Since no more than $|Q| - 1$ states can be equal under $\sim_0$ and and at least one equivalence class will be refined each step, this can continue for no more than $|Q| - 2$ steps.

If $\sim_{k+1} = \sim_k$, then no symbol $a \in \Sigma$ separates two states $q \sim_k q'$, i.e $\delta(q, a) \sim_k \delta(q', a), \forall a \in \Sigma$.

From this it follows that no string $w \in \Sigma^*$ separates q and q', as we now show by contradiction. Let $w = xa, a \in \Sigma$, be the shortest string that separates q and q', i.e $\delta^*(q, x) \sim_k \delta^*(q', x)$ and $\delta^*(q, xa) \not\sim_k \delta^*(q', xa)$. Let $q'' = \delta^*(q, x)$ and $q''' = \delta^*(q', x)$. We then have $q'' \sim_k q'''$ and $\delta(q'', a) \not\sim_{k+1} \delta(q''', a)$, contradicting $\sim_k = \sim_{k+1}$.

By Myhill-Nerode the equivalence-relations $\sim_k$ and $\sim_L$ are essentially the same by the relation between strings and states:
$\{u : \delta^*(q_o, u) = q\} \leftrightarrow q$

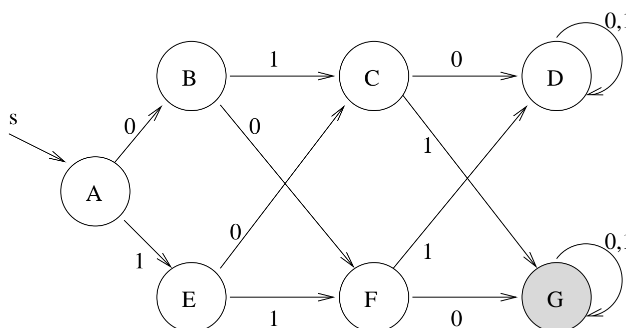
Assume as stated in the lemma that $u \not\sim_L v$. Let $q_u = \delta^*(q_0, u)$ and $q_v = \delta^*(q_0, v)$ and thus $q_u \not\sim_k q_v$.
Assume $q_u \sim_{i-1} q_v$ but $q_u \not\sim_i q_v (i \leq k)$. Then $\exists a_i \in \Sigma$ s.t. $\delta(q_u, a_i) \not\sim_{i-1}$

$\delta(q_v, a_i)$. Likewise $\exists a_{i-1} \in \Sigma$ s.t. $\delta^*(q_u, a_i a_{i-1}) \not\sim_{i-2} \delta^*(q_v, a_i a_{i-1})$. Continuing like this we get $a_i a_{i-1} .... a_1$ s.t $\delta^*(q_u, a_i a_{i-1} .... a_1) \not\sim_0 \delta^*(q_v, a_i a_{i-1} .... a_1)$. To conclude the proof: The string $z = a_i a_{i-1} .... a_1$ has length $\leq |Q|$ and distinguishes between u and v since $\delta^*(q_0, uz) \in F \iff \delta^*(q_0, vz) \notin F$, which means that $u a_i a_{i-1} .... a_1 \in L \iff v a_i a_{i-1} .... a_1 \notin L$.$\square$

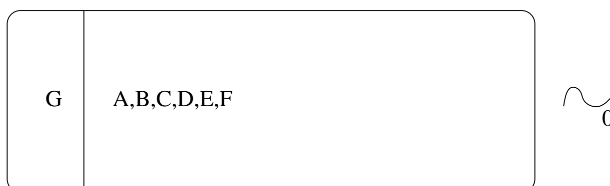
Example of Lemma 5.10

To illustrate the lemma 5.10 we will use it to gradually refine the equivalence classes of the automaton below, the series of figures show the gradual refinement of the equivalence-classes:

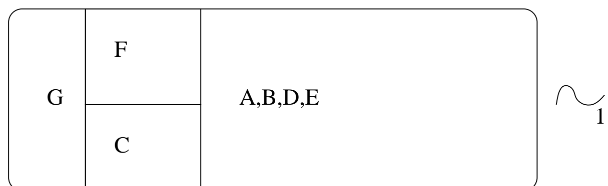
The automaton we are going to construct is given below.



For $\sim_0$ the equivalence classes are those states who are accepting states and those states who are not accepting states.



We refine $\sim_0$ to $\sim_1$ by checking if any symbol distinguish two equivalent states x and y by bringing x to one equivalence class and y to another. Here the symbol 0 will bring F to G whereas all other states remain in their old equivalence class on symbol 0 and similarly the symbol 1 brings only B to G.



We refine $\sim_1$ to $\sim_2$ the same way we refined $\sim_0$ to $\sim_1$. Symbol 0 brings B to F and E to C.

G	F	B	A,D
	C	E	

$\sim_2$

We refine $\sim_2$ to $\sim_3$ the same way we refined $\sim_0$ to $\sim_1$. Symbol 0 brings A to B while B remains.

G	F	B	A
	C	E	D

$\sim_3$

Since all states are separated we do not need to continue. All states could be separated with strings of length 3 or less, even though the smallest automaton has 7 states. For example the strings $x = \epsilon$ and $y = 010$, belonging to the equivalence class of states A and D respectively are distinguished by $z = 000$, and this string z is found by taking the sequence of distinguishing symbols as in the proof of Lemma 5.10.

5.6.3 On the number of strings

Lemma 5.11 *The number of strings in an alphabet Σ of length equal to k is $|\Sigma|^k$.*

Proof. We prove this by induction on k.

Base: $k = 0$, $|\Sigma|^0 = 1$, the empty string. Base ok.

IH: We assume that for $k-1$ the total number of strings is $|\Sigma|^{k-1}$.

IS: For each of the strings of length $k-1$, we can create a new one by adding a symbol from Σ. By IH we assume that there are $|\Sigma|^{k-1}$ strings of length $k-1$. We can add $|\Sigma|$ different new symbols to each one, the new total is thus $|\Sigma||\Sigma|^{k-1} = |\Sigma|^k$. Step ok.

Induction ok.□

Lemma 5.12 *The number of strings in an alphabet Σ of at most k is $\frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$.*

Proof. The number of strings in an alphabet Σ of length i is by Lemma 5.11 equal to $|\Sigma|^k$. The total number x of strings of length less than k is thus $\sum_{i=0}^{i=k} |\Sigma|^i$.

We can solve this:

$$\begin{aligned}
x &= \sum_{i=0}^{i=k} |\Sigma|^i \\
x &= |\Sigma|^0 + |\Sigma|^1 + |\Sigma|^2 + ... + |\Sigma|^k \\
|\Sigma|x &= |\Sigma|^1 + |\Sigma|^2 + |\Sigma|^3 + ... + |\Sigma|^{k+1} \\
|\Sigma|x - x &= |\Sigma|^{k+1} - |\Sigma|^0 \\
x(|\Sigma| - 1) &= |\Sigma|^{k+1} - |\Sigma|^0 \\
x &= \frac{|\Sigma|^{k+1} - 1}{|\Sigma| - 1}
\end{aligned}$$

And the proof is complete.□

5.6.4 Solving the promise problem

We are now ready to present the Method of Test Sets. We first give the result as presented in the textbook on parameterized complexity [8], that was based on a slightly weaker version of Lemma 5.10 which gave only the bound $|z| \leq |Q| = k$.

Theorem 5.13 *The promise problem is solvable and we can compute M' with at most k states accepting L. The running time is $O(|\Sigma|^{2k} * T(2k))$, where T(k) is the running time of the decision-procedure on input of length k.*

Proof. The naive algorithm for solving the promise problem assumes that any equivalence class can be represented by a string of length less or equal to k (trivially true) and that for any pair of strings $x \not\sim y$ there must exist a string z that separates the two($xz \in L \iff yz \notin L$) where $|z| \leq k$ (True by lemma 5.10). To decide if x is equivalent with y, we need to test all strings $z \in \Sigma^*$ with $|z| \leq k$ and see if $xz \in L$ iff $yz \in L$.

By building a table of size $\frac{|\Sigma|^{k+1}-1}{|\Sigma|-1} \times \frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$, one row and column for each string of length $\leq k$ (Lemma 5.12), we can test each string of length $\leq k$ with each string of length $\leq k$. Using the provided oracle we obtain and answer for each test. If two strings (rows in the table) agree on all tests, then the strings are in the same equivalence class in $\sim_L$ (by lemma 5.10). An example table can be seen in 5.1,for the language $L = (0 + 1)^*00$.

$\times$	ϵ	0	1	00	01	10	11	000	001	010	011	100	101	110	111
ϵ	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N
0	N	Y	N	N	N	N	N	N	N	N	N	Y	N	N	N
1	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N
00	Y	N	N	N	N	N	N	Y	N	N	N	Y	N	N	N
01	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N
10	N	Y	N	N	N	N	N	N	N	N	N	Y	N	N	N
11	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N
000	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N
001	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N
010	N	Y	N	N	N	N	N	N	N	N	N	Y	N	N	N
011	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N
100	Y	N	N	N	N	N	N	Y	N	N	N	Y	N	N	N
101	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N
110	N	Y	N	N	N	N	N	N	N	N	N	Y	N	N	N
111	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N

Table 5.1: Example: Table of test set answers for $L = (0+1)^*00$

We can now create a finite state machine using the following steps.

1. The states of M' is the equivalence classes of $\sim_L$.
2. The initial state is $[\lambda]_{\sim_L}$, the equivalence class containing the empty string.
3. The transition function δ of M' is defined by $\delta([a], x) = [ax])$ for $a, x \in \Sigma$.[5]
4. The accepting states of M' are those x with $[x] \subseteq L$.

This concludes our construction and it can be checked, even though a bit tedious, that $L(M') = L$.

Using the above algorithm we can construct a machine from table 5.1. Note that,

$[\epsilon] = [1] = [01] = [11] = [000] = [001] = [011] = [101] = [111]$
$[0] = [10] = [010] = [110]$
$[00] = [100]$

which gives us the equivalence classes. Using the algorithm we obtain the automaton in figure 5.6.4

[5]Notice that this is independent of a particular representative of an equivalence class. If $a \sim_L b$, then $\delta([a], x) = [ax] = [bx] = \delta([b], x)$.

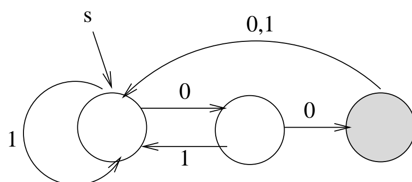


Figure 5.4: Example: Automaton built from the test set-answers to $L = (0+1)^*00$

We have to run the decision procedure a total of $\frac{|\Sigma|^{k+1}-1}{|\Sigma|-1} * \frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$ times with length less or equal to $2k$ (Once for each position in the table). Thus the running time is $O(|\Sigma|^{2k} * T(2k))$, where $T(k)$ is the running time of the decision procedure on input-size k. The algorithm as described is however not the most efficient. The following observation makes full use of Lemma 5.10 and gives an obvious improvement, while chapter 6 describes a less intuitive, but more powerful algorithm.

Observation 5.14 *It is sufficient to create a table of size* $\frac{|\Sigma|^{k}-1}{|\Sigma|-1} \times \frac{|\Sigma|^{k-1}-1}{|\Sigma|-1}$. *Thus the running time is* $O(|\Sigma|^{2k-3} * T(2k))$.

Since any state in the automaton can be reached by a string of length less or equal to $k-1$ (this since there are at most k states total) it is sufficient to use all strings of length $\leq k-1$ as rows in the table (a total of $\frac{|\Sigma|^{k}-1}{|\Sigma|-1}$, Lemma 5.12) , as this would guarantee us to find a representative for each equivalence class. Furthermore it is possible to separate two strings with a string of length less or equal to $k-2$(Lemma 5.10), thus we do not need more than $\frac{|\Sigma|^{k-1}-1}{|\Sigma|-1}$ columns in the table. Thus the total number of tests is $\frac{|\Sigma|^{k}-1}{|\Sigma|-1} \times \frac{|\Sigma|^{k-1}-1}{|\Sigma|-1}$ or $O(|\Sigma|^{2k-3} * T(2k))$. □

$\times$	ϵ	0	1
ϵ	N	N	N
0	N	Y	N
1	N	N	N
00	Y	N	N
01	N	N	N
10	N	Y	N
11	N	N	N

Table 5.2: Example: Table of test set answers (improved) for $L = (0+1)^*00$

To see the difference in practice compare the original test set-answers (Table 5.1) with the new table (Table 5.2) which takes this into account.

Observation 5.15 *The automaton computed by the method of test sets has the minimum number of states over any DFA for L.*

By Myhill-Nerode (Theorem 5.7) any congruence must be a refinement of the congruence $\sim_L$. Thus any other congruence will create a machine with more equivalence classes and thus more states. □

Chapter 6

Efficient Method of Test Sets

The method of test sets described in chapter 5 is a clear, intuitive way of describing the method of test sets. It is, however, not very practical because of the large array needed to store the table used to determine the different equivalence classes. The algorithm is also needlessly slow since it does nothing to prevent unnecessary calls to the oracle. We will in this chapter give an algorithm that runs both faster and uses considerably less space.

Recollect the promise problem:

Input:	A decision procedure (oracle) A for a regular language L.
Promise:	There is some DFA M accepting L with at most k states.
Question:	Compute an automaton M' accepting L.

Definition 6.1 *Oracle*

We will define an Oracle *for L as an independent entity, which given a string x can decide in constant time if $x \in L$.*

6.1 Recognizing equivalence classes

Remember that from Lemma 5.10 we can determine the equivalence class of a string $x \in \Sigma^*$ by testing the concatenation xz for all strings $z \in \Sigma^*$, $|z| \leq k-2$, where k is the promise. The strings $x, y \in \Sigma^*$ are in the same equivalence class if they answer the same on all z in the test. An equivalence class is thus uniquely determined by the answers given for these tests. We will call the sequence of these answers the *signature* of the equivalence class.

Definition 6.2 *Signature*

The signature of an equivalence class $[y]$ is the sequence of answers(yes or no) any string $x \in [y]$ will give when concatenated to all strings $z \in \Sigma^*, |z| \leq k-2$ and the result string xz is used as input to the oracle. Here the k is

the promise. In the signature the answers to the strings z are listed in the lexicographic order of these z in Σ.

6.2 Determining Signatures

To obtain the signature we must try all strings of length less than or equal to $k-2$.

Theorem 6.3 *We can determine for any language L with oracle, the signature of an equivalence class $[x]$, represented by the string x in time $O(|\Sigma|^{k-2})$.*

Proof. Enumerate all strings z in the alphabet Σ of length $\leq k-2$ in lexicographic order and for each one test "$xz \in L$?" with the oracle. The signature will be the sequence of answers. The total number of strings of length $\leq k-2$ are $\frac{|\Sigma|^{k-1}-1}{|\Sigma|-1}$ (by Lemma 5.12) and since the oracle can answer in constant time we get the total running time of $O(|\Sigma|^{k-2})$. □

Definition 6.4 *Signature test*

We define two types of signature tests:

1. *A full/long signature test consists of testing a string with all possible strings in the alphabet of length less than or equal to $k-2$.*
2. *A short signature test consists of testing a string with only a subset of strings that will uniquely determine which equivalence class it will fall into (see section 6.5).*

If a signature test is referred to without stating short or long/full we assume that it is a full test.

6.3 Storing signatures

One of the main problems of the naive algorithm is that it stores all answers for all possible strings in an array. Only when the array is complete will it determine the various equivalence classes of $\sim_L$. The array will be of size $O(|\Sigma|^{2k})$ which will very quickly grow to unmanageable size.

Definition 6.5 Signature Tree

A signature tree *is a binary tree where the nodes at each level are marked with a string from the test set*[1]

[1]Allthough any ordering of the test set will give a usuable tree(not necessarily the same), we will use the lexicographic ordering of the test set.

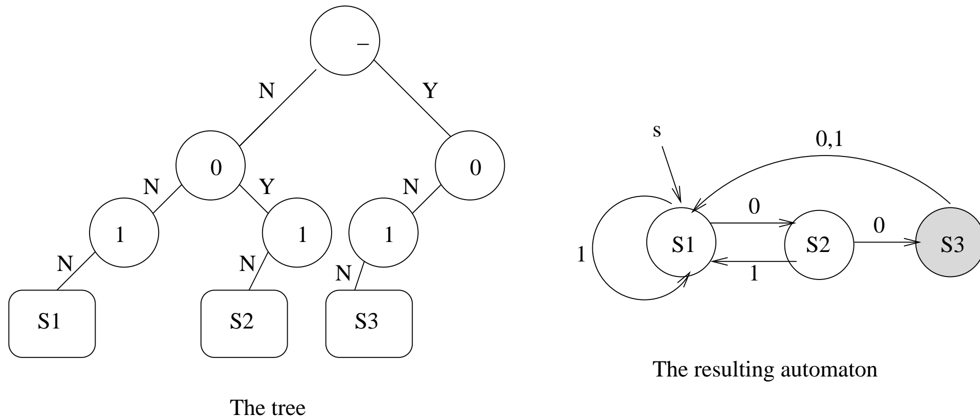


Figure 6.1: Full signature tree of language $\{x \mid x \text{ ends with } 00\}$ with promise $k = 3$. Each level in the tree represents one string in the test set. The strings marking each level is listed in lexicographic order $(\epsilon, 0, 1)$. Note how the leaves identifies the states in the resulting automaton.

Each node represents a test, a "NO"-answer branches left, while a "YES"-answer branches right. A path in the tree, from root to a leaf, represents the signature of a string.

We will store the signatures in a binary tree, where each level in the tree will represent a string $z \in \Sigma^*$, $|z| \leq k-2$. And the paths of tree will describe the signature of a string. (See figure 6.3).

Theorem 6.6 *The signatures can be stored in space* $O(k|\Sigma|^{k-2})$.

Proof. The tree will be of height $\frac{|\Sigma|^{k-1}-1}{|\Sigma|-1}$ (by Lemma 5.12), one level for each string of length $\leq k-2$. Since each path in the tree represents a different equivalence class the number of paths are limited to k. Each path in the tree is of length $\frac{|\Sigma|^{k-1}-1}{|\Sigma|-1}$, thus the space used is $k\frac{|\Sigma|^{k-1}-1}{|\Sigma|-1}$ or $O(k|\Sigma|^{k-2})$.□

6.4 Comparing signatures

The original algorithm compared signatures by checking the result, answer for answer. This takes, of course, exponentially long time when the signature is exponentially long. By issuing each leaf in the tree an Id encoding the unique signature (natural numbers from 0 to k is sufficient as Id-codes). Now each state can store the leaf Id instead of the signature. Comparing can then be

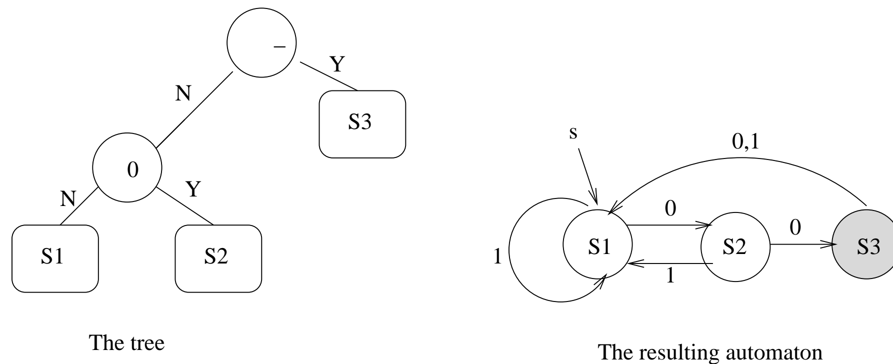


Figure 6.2: The contracted tree of language $\{x \mid x \text{ ends with } 00\}$. This tree was obtained by contracting all nodes in Figure 6.3 with degree 2 (one child). Observe that the test string 1 from the original test set is no longer used.

done by comparing the Id-number. We will use this later in the “on-the-fly” method.

6.5 Shorter signaturetests

We can take advantage of the knowledge that there will only be k different paths in the tree. When we have discovered k branches of the tree we no longer need to test the strings against all other strings. We know where the strings differ in answers and we can compute a much shorter test by contracting all nodes in the tree of degree 2(nodes with one child). This will leave us with a full binary tree with k leaves.(See figure 6.5) Note that this technique is not used in the next algorithms, but could be employed to give a speed-up to an implementation(it will,however, not affect the overall complexity of the next algorithms).

Lemma 6.7 *The height of a binary tree where all nodes has either* 2 *or* 0 *children and* l *leaves is less than or equal to* $(l-1)$.

Proof. We prove the lemma by induction on the number of leaves l.

Base: $l = 2$, internal nodes $= 1$, height $= 1$. Base case ok.

IH: Assume that a tree with $(l-1)$ leaves has height less than or equal to $(l-2)$.

IS: By the induction hypothesis the height of a $(l-1)$-leaf-tree has $(l-2)$ nodes. To grow in height we must remove a leaf on the highest level, adding a internal node with 2 leafes. This gives us a l-leaf tree with the height $(l-1)$. Induction ok. $\square$

Theorem 6.8 *If a signature tree with k paths is known, then we can construct in time $O(k|\Sigma|^{k-2})$ construct a tree such that subsequent signaturetests can be performed in time $O(k'-1)$, here k' is the number of equivalence classes (note that $k' \leq k$).*

We can replace all nodes of degree 2 with an edge, (if we move down the tree, testing at these nodes will always yield the same answer). The result will be a full binary tree. Since we can replace any node in constant time we can perform this contraction in time $O(k|\Sigma|^{k-2})$ (The maximum size of the tree, Theorem 6.6).

By lemma 6.7 we know that each path is no longer than $k' - 1$, and by running the test on the nodes left in the tree we can determine which equivalence class a string belongs to. □

The contracted tree can be viewed as a smaller test set, one where the redundant strings have been removed. We can use these proofs for another important observation.

Lemma 6.9 *The height of a binary tree where all nodes has either 2 or 0 children and l leaves is at least $\log\lceil l \rceil$.*

Proof by induction on l.

Base: T(1) has 2 leaves and height $1 \geq \log_2 2$. Base ok.

IH: Holds for 1...l

IS: l+1: The binary tree T with $l + 1$ leaves has two subtrees, T_1 and T_2. T_1 has l_1 leaves and T_2 has l_2 leaves. Either $l_1 \geq \lceil \frac{l}{2} \rceil$ or else $l_2 \geq \lceil \frac{l}{2} \rceil$. So depth of tree T:

$$\begin{aligned} depth(T) &= 1 + \max\{depth(T1), depth(T2)\} \\ &\geq_{IH} 1 + \max\{\lceil \log_2 l_1 \rceil, \lceil \log_2 l_2 \rceil\} \\ &\geq 1 + \log_2 \lceil \frac{l}{2} \rceil \geq \lceil \log l \rceil \end{aligned}$$

Proof complete.□

Theorem 6.10 *The minimal size of the test set S for any language L is at least $\log_2\lceil n \rceil$ and at most $n-1$, where n is the number of states in a minimal DFA for L.*

Assume in contradiction to the stated theorem that there exists a test set S for language L where $|S| < \log_2\lceil n\rceil$ strings. Then there must exist a contracted tree with height $|S|$, the contracted tree has n leaves and thus has height $\geq \log_2\lceil n\rceil$ (Lemma 6.9). We have reached a contradiction, hence the assumption must be wrong.

Similarly we assume in contradiction to the stated theorem that there exists a minimal test set S for language L where $|S| > l - 1$. The signature tree for this test set has n leaves and thus by Lemma 6.7 there exists a contracted tree with height at most $n - 1$. The levels in this contracted tree represent a smaller test set than S. Thus, S cannot be minimal and we have reached a contradiction. The assumption must be wrong. □

6.6 Building the automaton 'On-the-fly'

It is not efficient to build the automaton after testing all possible strings. It is far more efficient to test strings when you need them. By creating the automaton step by step and gradually discovering the states we can build an automaton using only $O(k'|\Sigma|)$ long tests, where k' is the number of equivalence classes in $\sim_L$.

The algorithm (seen in table 6.1) works by identifying the signature of the empty string. This equivalence class will be the start-state in the machine we are building. Then we will test each possible transistion from the state. We create the signature for each of the transition-strings and check if this signature is already discovered. If it is, then we can add to the transition-function using the already existing state. Otherwise we create a new state for this newfound signature. This new state is added to a queue, which ensures that we will later check the transitions from this state.

Working through the queue, doing the same as for the initial state, we gradually fill out the transition-function until it is complete and we have a minimal DFA for the language L.

Analyzing the algorithm we see that the while loop runs once for each state created and each while-loop calls the Signaturetest once for each symbol in the alphabet. Since this algorithm creates a minimal machine the number of states are limited to k', where $k' \leq k$. The total calls to the Signaturetest is $k'|\Sigma|$. As each signature test costs $O(|\Sigma|^{k-2})$ (Theorem 6.3) we get a total running time of $O(k'|\Sigma|^{k-1})$.

Table 6.1: “On-the-fly” Algorithm for the method of test sets (TimeOptimal)

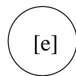
```
INPUT SignatureTest Subroutine
          Language Alphabet Σ.
BEGIN Algorithm

Set Q = ∅        The set of states
Set δ = ∅        The transitionfunction
Queue K = ∅      Unfinished states.

State q.
q.SignatureId = Signaturetest(ϵ)
q.Equivalencestring = ϵ
Q = Q ⋃{q}
K.Enqueue(q)

While K not empty
    State q = K.Pop
    For all a ∈ Σ
        SignatureId S = Signaturetest(q.equivalencestring + a)
        If ∃t ∈ Q s.t t.SignatureId = S Then
            δ(q, a) = t
        Else
            State v.
            v.SignatureId = S
            v.Equivalencestring = q.equivalencestring + a
            Q = Q ⋃{v}
            K.Enqueue(v)
            δ(q, a) = v
        End If
    End For
End While

END Algorithm
```

6.7 Example of ’On-the-fly’

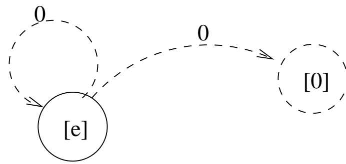
To better illustrate the idea of creating an automaton by the ’On-the-fly’-idea we have included this illustrated example. We will try to build an automaton for the regular language $L = (0 + 1)^{*}100$ where we give a promise that the language can be realized by an automaton with 4 states. Since the promise

(k) is 4, we know by Lemma 5.10 that the test set containing all strings of length $k-2=2$ can determine if two strings are equivalent. Thus the test set $S=\{\epsilon,0,1,00,01,10,11\}$ suffices.

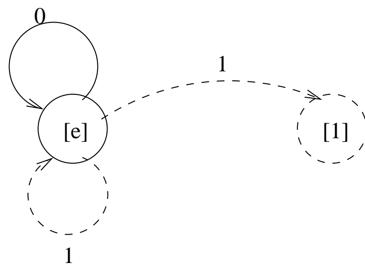
We start with the empty string ϵ and obtain the full signature for it, the signature will be a vector of 7 "NO"s. We denote the equivalence class and state, containg ϵ by $[\epsilon]$.



Now consider what happens if we are in $[\epsilon]$ and read a 0. Either we move to a new state with corresponding to equivalence class $[\epsilon 0]=[0]$ or we loop back to this state.

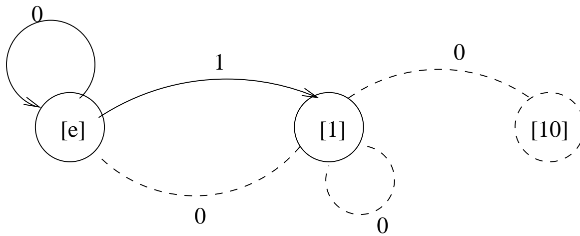


By obtaining the signature for 0,we discover that the signature for 0 is identical to that for ϵ, thus we make a loop back to $[\epsilon]$. Make the same argument for the other alphabetsymbol 1. Either it leads to a new state [1] or it loops back to $[\epsilon]$.



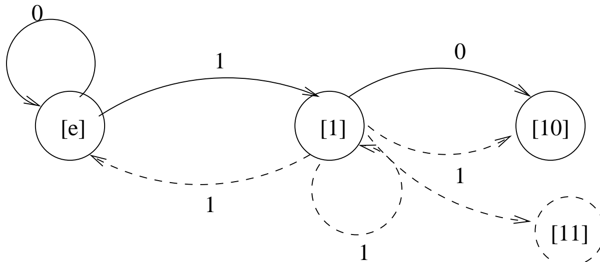
We discover that the suffix 00, which is in the test set, will take the string 1 into the language $(100\in L)$ while ϵ will remain outside the language $(00\notin L)$. Thus we know that ϵ and 1 are not equivalent and we create a new state [1]. Since $1\notin L$ we let the new state [1] be a rejecting state.

Now we have exhausted the symbols that can give transitions out of the state $[\epsilon]$. We move on to the next state [1]. We consider the string 10 which we get from concatenating 0 to the string 1 which is the representative of equivalence class [1]. The possible outcomes are shown as stapled lines below.

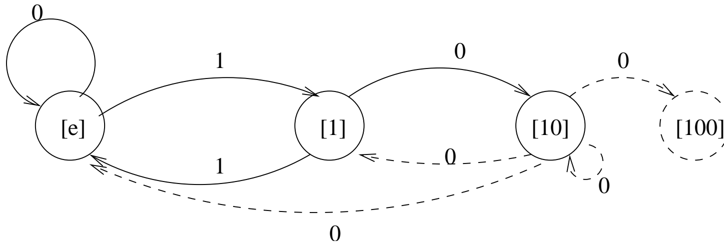


We obtain the signature for the string 10 and discover that we have not found this signature previously, thus we create a new state for it. The representative string for this state is [10] and since $10 \notin L$, [10] is a rejecting state.

Doing the same for the string 11, which we obtain by concatenating 1 to [1]'s representative string we discover that it has the same signature as [ϵ], thus we add a transition from [1] to [ϵ] on symbol 1.

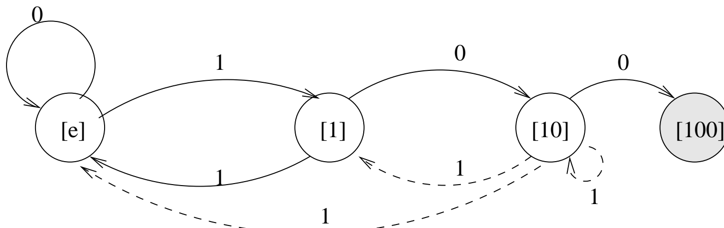


Moving on to the new state [10] we consider what happens when we read a 0 at this state.



We discover that this will give us a signature-string that we have not found yet. We create a new state and mark it with [100]. Since $100 \in L$ we mark this as an accepting state.

As before, we continue with testing the symbol 1 at state [10].



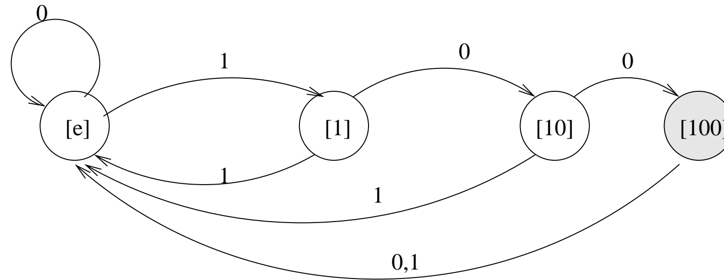
The signature is the same as the signature for the state [ϵ], thus we make a transition back to [ϵ].

There is now only one state where we don't know every transition, namely [100]. Let us consider what happens when reading 0 in this state.

We discover that the signature is identical to the signature of [ϵ]. We add an edge back to [ϵ].

To continue, we check the signature of the remaining possibility: Reading a 1 in state [100].

We discover that this signature is also the signature of [ϵ], and we add a transition from [100] to [ϵ] on 1.



There is no longer any states with unknown transitions. Thus the algorithm halts, having produced the minimal automaton for the language.

6.8 The gain

Using these techniques we get a substantial speedup in the algorithm. The naive algorithm uses $O(|\Sigma|^{2k})$ space and time. By using the tree-structure in section 6.3 we do not need more space than $O(k'|\Sigma|^{k-2})$ and using the technique described in section 6.6 and building the automaton "on the fly" we can make do with time $O(k'|\Sigma|^{k-1})$.

The difference in runningtime and space can be expressed as:

$$\text{The time factor } = \frac{\text{Naive algorithm}}{\text{New algorithm}} = \frac{O(|\Sigma|^{2k})}{O(k'|\Sigma|^{k-1})} = O\left(\frac{1}{k'}|\Sigma|^{k+1}\right)$$

$$\text{The space factor } = \frac{\text{Naive algorithm}}{\text{New algorithm}} = \frac{O(|\Sigma|^{2k})}{O(k'|\Sigma|^{k-2})} = O\left(\frac{1}{k'}|\Sigma|^{k+2}\right)$$

We can see that we have gained a substantial exponential factor in both time and space. The algorithm remains, however, in exponential time and is therefore very slow on large k.

6.9 A Space/Time tradeoff

Sacrificing time for space or space for time is a well known idea in algorithm theory. Often time is of the essence and space is abundant, other times the need for small space outweighs the need for a fast algorithm.

In the method of test sets we can sacrifice time to gain an enormous amount of space. This can be done by introducing a new way to decide if two strings $x, y \in \Sigma^*$ are in the same equivalence class ($[x] = [y]$). The subroutine in table 6.2 uses time $O(|\Sigma|^{k-2})$, but it does not require more space than a string of length k.

Table 6.2: Space-optimal subroutine for deciding if two strings are equivalent

```
Input: Strings x and y
Parameters: Oracle O, promise k
Question: Is [x] = [y] ?
BEGIN Algorithm SameEq

For i = 1 to (Σ^(k-1) - 1)/(Σ - 1)
    Obtain the i'th string of the alphabet z
    IF O(xz) ≠ O(yz) THEN
        Answer No.
    End IF
End For

Answer Yes.

End Algorithm
```

By changing the MTS algorithm in table 6.1 to the MTS algorithm in table 6.3 we get an algorithm that does not store the signatures at all, relying instead on testing for equivalence using subroutine SameEq in table 6.2 when needed. This new algorithm runs in $O(k^2|\Sigma|^{k-1})$ time, but uses only $O(k)$ space. We have sacrificed some time but use very little space.

Table 6.3: "On-the-fly" Algorithm for the method of test sets (Spaceoptimal)

INPUT Subroutine SameEq(x,y)(Table 6.2)
        Language Alphabet Σ
BEGIN Algorithm

Set $Q = \emptyset$ The set of states
Set $\delta = \emptyset$ The transition-function
Queue $K = \emptyset$ Unfinished states.

State q.
q.EqString $= \epsilon$
$Q = Q \bigcup \{q\}$
K.Enqueue(q)

While K not empty
    State $q = K$.Pop
    For all $a \in \Sigma$
        If There is $t \in Q$ such that SameEq(t.EqString,q.EqString+a) = YES Then
            $\delta(q, a) = t$
        Else
            State v.
            v.EqString $= q$.EqString $+ a$
            $Q = Q \bigcup \{q\}$
            K.Enqueue(v)
            $\delta(q, a) = v$
        End If
    End For
End While

END Algorithm

Chapter 7

Using randomization on method of test sets

The core idea behind Method of Test Sets is to generate a set of test-strings and use this to check which strings belong to which equivalence classes. While a complete Test Set contains all possible strings of length less or equal to a promise k, we are looking for strings that *separate* two strings x, y. Such strings effectively prove that x and y are not in the same equivalence class ($[x] \neq [y]$). If we don't find a string z that *separates* two strings x and y we know that $[x] = [y]$.

Definition 7.1 *Separating String*
For two strings $x, y \in \Sigma^*$, a string $z \in \Sigma^*$ is a separating string *if $xz \in L$ and $yz \notin L$ or vice-versa.*

Unfortunately the test set S of strings that are of length less or equal to $k - 2$ is exponentially large and testing all strings in S takes too much time for most values of k. When exhaustively testing all strings $z \in S$ is not an option we are forced to create a subset S' with the strings we will use to test with. If the set S' contains strings that separates any two strings x, y where $[x] \neq [y]$, then the automaton constructed using S' will be equal to the automaton constructed using S.

Since we have no way to determine if a string will be a separating string (without actually trying it) we will resort to randomly picking the strings for S'. This can be done in two ways, either we randomly guess each string without considering if we already guessed this string before, or we pick strings from the set S guaranteeing that S' does not contain duplicates.

Because of it's simplicity we will first try to randomly guess strings for S', but as we shall see, guaranteeing that S' does not contain duplicates gives a better result.

7.1 Creating a randomized algorithm

From chapter 6 we have two slightly different algorithms for the Method of Test Sets (the Time-optimal MTS Algorithm 6.1 and the Space-optimal Algorithm 6.3). Both algorithms can easily be adapted to randomization and we will test each of them on a series of regular languages.

The first algorithm stored the signatures in a tree. The levels in the tree represented a string of length less or equal to $k-2$. In a full tree all the strings of length less or equal to $k-2$ are present. In the randomized algorithm we can randomly pick C strings and insert them as new levels in the tree, where C is a prechosen integer.

The second algorithm can be randomized by changing the subroutine that determines if two strings are equivalent (Algorithm 6.2). Instead of iterating through all possible strings we randomly guess C strings and test with these, where C is a prechosen integer. Note that the randomized subroutine to test if two strings are equivalent. (see table 7.1) retains the space-saving properties of its deterministic counterpart. It has a running time of $O(Ck)$ as guessing a binary string of length k takes $O(k)$ time.

Table 7.1: Basic randomized subroutine for deciding if two strings are equivalent

```
Input: Strings x and y
Parameters: integer C, Oracle O, integer k
Question: Is [x] = [y] ?
BEGIN Algorithm

For 1 to C
    Guess binary string z, |z| ≤ k − 2
    IF O(xz) ≠ O(yz) THEN
        Answer No.
    End IF
End For

Answer Yes.

End Algorithm
```

7.2 Time and space-bounds

Both randomized algorithms are much faster than their counterpart as the time-consuming part of iterating through all possible steps have been replaced. Now the algorithms depend on how many strings have been picked as test-strings. If we let the number of test-strings be C we have the following running times.

7.2.1 Timeoptimal MTS (Algorithm 6.1)

The randomized version of this algorithm runs in time $O(k\Sigma C)$ as the algorithm tests each state (in all k) with all possible one-letter extensions from Σ. Each test takes $O(C)$ time as the tree is of height C.

The algorithm uses less space as the height of the signature-tree is C instead of $\frac{\Sigma^{k+1}-1}{\Sigma-1}$. Now the total space used is $O(kC)$ for the tree (C in height and k branches). Thus the total space used becomes $O(kC)$.

7.2.2 Spaceoptimal MTS (Algorithm 6.3)

The randomized version of this algorithm runs in time $O(k^3\Sigma C)$ as the algorithm 7.1 is runs in time $O(kC)$.

The algorithm uses the same amount of space as its non-randomized counterpart, that is $O(k)$.

7.3 Empirical analysis

We can empirically test each algorithm by generating automata with the randomized algorithm and compare them to an automaton produced by the full Method of test sets. By selecting various values for C we can get an idea of how many guesses are needed to have a reasonably good chance of producing a correct automaton. We have chosen C as a function of the promise k. Various values of C (k,$k\log_2 k$, k^2,$k^2\log_2 k$, k^3) have each been tested by generating[1] 250 automata and comparing them to a correct automata. The tables 7.2 and 7.3 show the percentage of correctly produced automata. The table 7.2 shows the results from the standard on-the-fly algorithm and the table 7.3 shows the result for the space-optimal algorithm.

[1]The number 250 was chosen empirically, after 250 runs of the algorithm the total result converged and varied only slightly

Language	min # states(k)	total # of strings	k	$k \log_2 k$	k^2	$k^2 \log_2 k$	k^3
$(0+1)^*11(0+1)^*$	3	3	64.8%	76.4%	97.6%	99.6%	100%
$000+010$	5	15	33.6%	78.8%	99.2%	100%	100%
$(0+1)^*0$	2	1	100%	100%	100%	100%	100%
$(000)^*$	4	7	8%	32.4%	76%	97.6%	100%
10^*1	4	7	10%	48.4%	83.6%	99.6%	100%
$(10)^*+(01)^*$	5	15	0.8%	12.4%	68%	98.8%	100%
$(0+1)^4$	6	31	1.6%	18.4%	58.4%	94.8%	99.6%
$1(0+1)^*1$	4	7	7.6%	43.2%	84.8%	98%	100%
$0+1$	3	3	67.2%	70%	98.8%	100%	100%
$binval <= 10$	7	63	0%	1.2%	15.6%	70.4%	97.2%
$binval <= 15$	6	31	0.8%	16%	64%	95.6%	99.6%
$binval <= 25$	9	255	0%	0%	2.4%	35.6%	94.4%

Table 7.2: Some results from randomizing the standard algorithm, each entry is percentage of correctly produced automatons

Language	min # states(k)	total # of strings	k	$k \log_2 k$	k^2	$k^2 \log_2 k$	k^3
$(0+1)^*11(0+1)^*$	3	3	66.4%	78.4%	97.6%	98.8%	100%
$000+010$	5	15	5.6%	72.8%	98.8%	100%	100%
$(0+1)^*0$	2	1	100%	100%	100%	100%	100%
$(000)^*$	4	7	0%	1.6%	34%	89.2%	100%
10^*1	4	7	0%	9.2%	59.2%	96.4%	100%
$(10)^*+(01)^*$	5	15	0%	1.2%	46.4%	97.2%	100%
$(0+1)^4$	6	31	0%	0.4%	16.8%	80%	99.6%
$1(0+1)^*1$	4	7	0.4%	15.2%	68%	98.4%	100%
$0+1$	3	3	20.4%	37.2%	90.4%	97.2%	100%
$binsum <= 10$	7	63	0%	0%	0%	30%	96.8%
$binsum <= 15$	6	31	0%	0.4%	16%	82.8%	100%
$binsum <= 25$	9	255	0%	0%	0%	2.8%	82.4%

Table 7.3: Some results using the space-optimal algorithm

7.4 Interpreting the results

What can we deduce from the experiments? First of all it is clear that some types of languages are better suited than others. A language like $(0+1)^*11(0+1)^*$ do exceptionally well while a language like "binval"[2] does not randomize well at all. Thus it is obvious that randomized test sets is only suitable on a subset of the regular languages.

It is also clear that the standard 'on-the-fly' algorithm does better than the one optimized for space. The intuitive explanation for this come from the fact that the space optimal algorithm generate many subsets of testing strings. If there is only a few strings that separate two equivalence classes, then there is very little chance for them to be picked for the test sets each time they are required.

Note that for small automata the values of k^2 and k^3 are greater than the number of all strings of appropriate length. Thus it would be more efficient to run the standard algorithm and obtain the correct machine.

Also note that the language $(0+1)^*0$ scores full marks since the required test set for this automaton is empty.

7.5 Size of the test set

The effectiveness of the algorithm depends on the choice of the size C of the test set S'. If we choose C too low, we will get a fast algorithm, but the probability for producing the correct machine will be low. On the other hand choosing C too high will result in an algorithm that produces the correct machine more often, but unfortunately runs much slower. We will show that the program in the worst case will run too slow if it should maintain a decent chance of success.

We will begin with calculating the general chance for finding one of these strings. It is important to notice that it may be more than one suitable string.

Theorem 7.2 *The probability for picking a string that separates two strings* x,y *is* $1-\left(1-\frac{B}{A}\right)^C$, *where* A *is the total number of possible strings,*B *is the number of separating strings and* C *is the size of the test set.*

Proof. The only case when we do not find a separating string is to not pick one of the B separating strings C times in a row. The probability for

[2]binval: the binary value of a string

not picking a string is obviously $\frac{A-B}{A}$ or $1-\frac{B}{A}$. This gives the probability for failure $(1-\frac{B}{A})^C$ and hence $1-(1-\frac{B}{A})^C$ for success.□

We will now give a formula that calculates the required size of the test set to limit the chance of failure to a constant level p.

Definition 7.3 *Error-margin*
The term error-margin *is here used as the probability for not finding a separating string. Thus , a p error-margin algorithm will fail with probability p.*

Theorem 7.4 *To achieve an error-margin of $p, 0 < p < 1$ for any k we must select a $C = \frac{\ln(p)}{\ln\left(1-\frac{B}{A}\right)}$ strings, where A is the total number of strings and B the number of separating strings.*

Proof. We can easily solve the mathematical equation:

$$\begin{aligned} \left(1-\frac{B}{A}\right)^C &= p \\ C\ln\left(1-\frac{B}{A}\right) &= \ln p \\ C &= \frac{\ln p}{\ln\left(1-\frac{B}{A}\right)} \end{aligned}$$

Proof complete.□

As $A > B$ the function $\ln\left(1-\frac{B}{A}\right)$ will drop towards zero, using Taylor-approximation we get:

$$\begin{aligned} f(k) &= f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + ... \\ a &= 1 \\ f(k) = \ln\left(1-\frac{B}{A}\right) &= \ln\left(1+(-\frac{B}{A})\right) \\ f(k) &= \ln(1) + \ln'(1)(1+(-\frac{B}{A})-1) + \frac{\ln''(1)}{2!}(1+(-\frac{B}{A})-1)^2 + ... \end{aligned}$$

When $A >> B$, the quadratic term $\frac{\ln''(1)}{2!}(1+(-\frac{B}{A})-1)^2 + ..$ becomes infinitesimal and can be dropped. We are left with :

$$\begin{aligned}
f(k) &\approx \ln(1) + \ln'(1)(1 + (-\frac{B}{A}) - 1) \\
f(k) &\approx 0 + \frac{1}{1}(-\frac{B}{A})) \\
f(k) &\approx -\frac{B}{A}
\end{aligned}$$

Combining this with the result from Theorem 7.4 and we get that when $A >> B$ (many states):

$$\begin{aligned}
C &= \frac{\ln(p)}{\ln\left(1 - \frac{B}{A}\right)} \\
\ln\left(1 - \frac{B}{A}\right) &\approx -\frac{B}{A} \\
C &\approx \frac{\ln(p)}{-\frac{B}{A}} \\
C &\approx -\frac{A}{B}\ln(p) \\
C &\approx \frac{A}{B}\ln\frac{1}{p}
\end{aligned}$$

From this we can draw some obvious conclusions:

Observation 7.5 *In the worst case situation when there are only one separating string we will need exponentially many guesses to obtain any constant error-margin p*

When there are only one separating string, then $B = 1$, using the above formula we get :

$$C \approx \frac{A}{B}\ln\frac{1}{p} = A\ln\frac{1}{p}$$

Since $\ln\frac{1}{p}$ is constant and A is exponential in the number of states (Lemma 5.12), C must grow exponentially as well. □

7.6 All is lost?

We have already concluded in observation 7.5 that the basic randomized algorithm is not practical on all input. The worst case situation where we

search for one string among all strings is simply too hard. As stated earlier in the chapter, it is not always true that there is only one separating string.

There might be more than one, perhaps even proportionally many to the total number of strings. Furthermore, this may not only be true for specific languages, but for whole subclasses of the regular languages.

What kind of gain can we hope for if we find such subclasses? We will now introduce a way to classify regular languages according how many separating strings there are for each pair of equivalence classes.

We will now try to identify some of these subclasses and try to prove whether or not the basic algorithm is suitable for these subclasses.

7.7 Subclasses

Definition 7.6 *Subclass*

A set S of regular languages is an $f(k)$-subclass if the minimal DFA with k states for any $L \in S$ has no pair of states where the separating strings are fewer than $f(k) \geq 1$.

7.7.1 Bad Subclasses

The randomized algorithm is not viable for any $f(k)$-subclass where $f(k)$ grows exponentially slower than the function $\frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$ (Lemma 5.12,The total number of strings), this leads to the notion of bad subclasses.

Definition 7.7 *Bad Subclass*

An $f(k)-$subclass is considered bad *if $f(k)$ grows exponentially slower than the function $g(k) = \frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$. That is $h(k) = \frac{g(k)}{f(k)}$ is exponential in k.*

Theorem 7.8 *When applying the randomized algorithm on a bad subclass we will have to guess exponentially (in the size of k) many strings to have any constant error-margin p.*

Proof. Using the formula $g(k) = \frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$ for A and any exponentially slower function $f(k)$ for B, we get that $\frac{A}{B} = \frac{g(k)}{f(k)}$ is an exponential function $h(k)$.

Inserting this into the formula from section 7.5, we get:

$$C \approx \frac{A}{B} \ln \frac{1}{p}$$

$$\frac{A}{B} = \frac{g(k)}{f(k)} = h(k)$$
$$C \approx h(k)\ln\frac{1}{p}$$

Which proves the theorem, since $\ln\frac{1}{p}$ is constant and $h(k)$ is exponential in k(number of states).□

7.7.2 Good subclasses

Conversely, in a subclass where there are very many separating strings for all equivalence classes, we can expect better running time for the randomized algorithm.

Definition 7.9 *Good Subclass*

An $f(k)$-subclass is considered good *if $f(k)$ grows exponentially equal to the function $g(k) = \frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$. That is $h(k) = \frac{g(k)}{f(k)}$ is a polynomial in k.*

Theorem 7.10 *When applying the randomized algorithm on a member of an $f(k)$-subclass that is considered* good*, it is sufficient to guess a polynomial (with regards to k) many strings to have a constant error-margin of p.*

Proof. Using the formula $g(k) = \frac{|\Sigma|^{k+1}-1}{|\Sigma|-1}$ for A and $f(k)$ for B. Since the class is considered good we know that $\frac{A}{B} = \frac{g(k)}{f(k)}$ is an polynomial function $h(k)$.

Inserting this into the formula from section 7.5, we get:

$$C \approx \frac{A}{B}\ln\frac{1}{p}$$
$$\frac{A}{B} = \frac{g(k)}{f(k)} = h(k)$$
$$C \approx h(k)\ln\frac{1}{p}$$

Which proves the theorem, since $\ln\frac{1}{p}$ is constant and $h(k)$ is polynomial in k (number of states).□

Table 7.4: Algorithm for creating a test set with no duplicates.

```
Input: Integer C
Parameters: Integer k, Alphabet Σ
Task: Generate C unique strings of length at most k
Subroutine: Generate_nth( n )
BEGIN Algorithm

Set  X of indexes Set  S of strings
For i = 0 to C − 1
    Guess number x between 0 and (Σ^{k−1} − 1)/(Σ − 1) − i
    y = |Z|, Z = {z|z ∈ X and z ≤ x}
    n = x + y
    X = X ⋃{n}
    S = S ⋃{ Generate_nth( n ) }
End For

Return S
End Algorithm
```

7.8 No duplicates

As mentioned earlier in this chapter the randomized selections for the test sets have not taken into account the possibility that there may be duplicates in the test set. This is not good as a duplicate is essentially useless. It can never separate a string that its duplicate cannot separate.

Creating a test set without duplicates is however not trivial. We have devised a algorithm (See figure 7.4) that in $O(Cn^2)$ time generates C unique strings of length at most n.

As we can see from Table 7.5 the results are much better with this algorithm. The reason we get so many results with full marks is that this will pick every teststring if the test set is larger than the total number of strings of length $\leq k$.

Language	min # states(k)	total # of strings	k	$k \log_2 k$	k^2	$k^2 \log_2 k$	k^3
$(0+1)^*11(0+1)^*$	3	3	100%	100%	100%	100%	100%
$000 + 010$	5	15	36.8%	82.4%	100%	100%	100%
$(0+1)^*0$	2	1	100%	100%	100%	100%	100%
$(000)^*$	4	7	10.4%	100%	100%	100%	100%
10^*1	4	7	11.2%	100%	100%	100%	100%
$(10)^* + (01)^*$	5	15	2%	19.2%	100%	100%	100%
$(0+1)^4$	6	31	2.4%	28%	100%	100%	100%
$1(0+1)^*1$	4	7	18%	100%	100%	100%	100%
$0+1$	3	3	100%	100%	100%	100%	100%
$binval <= 10$	7	63	0%	0.4%	27.2%	100%	100%
$binval <= 15$	6	31	1.6%	29.6%	100%	100%	100%
$binval <= 25$	9	255	0%	0%	0.8%	100%	100%

Table 7.5: Some results from randomizing the standard algorithm, each entry is percentage of correctly produced automatons

Chapter 8

Approximating Automata

From the test results in chapter 7 we can see that the randomized method(RMTS) rarely produced the correct automata. Somewhere along the line the "on-the-fly" fails to find one of the required separating strings. This leaves us with an unfinished automaton. Since MTS produces minimal automata and this automaton has less states it follows that the unfinished automaton does not accept the correct language. The automaton may, however, have produced an automaton very similar to the correct automaton, in a way it is an approximation of the correct automaton. In this chapter we will show this formally and then use empirical analysis to try to give an estimate for the degree of approximation.

8.1 Approximation

Definition 8.1 *Refinement/Coarsening*

A set of equivalence classes $L_\sim$ is a refinement *of another set of equivalence classes $L'_\sim$ iff the equivalence classes in $L'_\sim$ are unions of classes in $L_\sim$.*

If $L_\sim$ is a refinement *of $L'_\sim$, we say that $L'_\sim$ is a* coarsening *of $L_\sim$.*

Theorem 8.2 *Any set of equivalence classes $L'_\sim$ produced by RMTS when run on language L will be a coarsening of the equivalence classes $L_\sim$ produced by MTS when run on L.*

Proof. The equivalence classes in $L'_\sim$ are constructed by RMTS by gradually refinement. The algorithm starts with *one* equivalence class, a trivial coarsening of $L_\sim$.

Assume in contradiction to the statement of the theorem that RMTS will refine $L'_\sim$ such that it is no longer a coarsening of $L_\sim$.

This refinement can only happen if RMTS finds a string z that separates two strings x and y, where $[x] = [y]$ in $L_\sim$ and $xz \in L \iff yz \notin L$. Thus $[x] \neq [y]$ in $L'_\sim$.

This implies that MTS did not find the separating string z. MTS tests all strings of length less or equal to $k-2$, and by Lemma 5.10 we know that all necessary separating strings are of length less or equal to $k-2$. Thus z cannot exist.

We have a contradiction and the assumption must be wrong, hence RMTS cannot construct a $L'_\sim$ that is not a coarsening of $L_\sim$. //

Corollary 8.3 *If a set of equivalence classes $L'_\sim$ produced by RMTS when run on language L and a set of equivalence classes $L_\sim$ produced by MTS when run on L has the same size ($|L'_\sim| = |L_\sim|$), then $L'_\sim = L_\sim$.*

Assume that the corollary is not true. Then $L_\sim$ can not be a refinement of $L'_\sim$, since they have equal number of equivalence classes, but are not equal. This is impossible by the Theorem 8.2. Thus the assumption must be wrong and the corollary holds.

8.2 Defining the approximating languages

Given a set of equivalence-classes produced by RMTS it is not immediately obvious which classes should be the accepting states in the generated automaton. There may be coarsened states that contain both strings that belong to the language and strings that are not in the language. It is easy to define two languages L^+, L^- which approximate L.

L^+ is defined as the language obtained from the machine constructed by RMTS if we let the accepting states be those equivalence classes that contain at least one string x where $x \in L$.

L^- is defined as the language obtained from the machine constructed by RMTS if we let the accepting states be those equivalence classes that contain only strings belonging to L.

8.3 The relationship between L, L^+, L^- and L^{RMTS}

L^+ will accept all strings in L and probably more. It is clear that $L^+ \supseteq L$. A similar argument holds for L^-. L^- never accepts a string that is not in L, but rarely accepts all string in L. Hence $L^- \subseteq L$. We can now deduct that $L^- \subseteq L \subseteq L^+$.

Unfortunately it is impossible to decide if a state contains only strings in the language (without actually creating a correct automaton). Thus it is impossible to choose to create L^- or L^+. A more practical approach is to make the decision immediately upon discovery of a new equivalence class wether this state should be an accepting state. We will let an equivalence class be accepting iff the representative string of the equivalence class is in the language. We call this language L^{RMTS}.

Observation 8.4 *$L^- \subseteq L^{RMTS} \subseteq L^+$.*

Proof. We will prove this in two parts, first that $L^- \subseteq L^{RMTS}$ and then that $L^{RMTS} \subseteq L^+$ holds.

We will begin with proving $L^- \subseteq L^{RMTS}$ by contradiction. Assume in contradiction to the observation that $L^{RMTS} \subset L^-$. This implies that RMTS must select an equivalence class E containing only strings in L as a rejecting state. Since E contains only strings in L the representing string must be in L as well. Thus RMTS would choose E as an accepting state. We have reached a contradiction, hence the assumption must be wrong.

A similar argument is used to prove that $L^{RMTS} \subseteq L^+$. Assume in contradiction to the observation that $L^{RMTS} \supset L^+$. This implies that RMTS chose an equivalence class F containing no strings in L as an accepting state. Since F contains no strings in L, F's representative cannot be in L either. Thus RMTS would choose F as a rejecting state. We have reached a contradiction, hence the assumption must be wrong.□

8.4 Empirical analysis

We have defined a language L^{RMTS} for the randomized algorithm. Because of the randomization this language varies from run to run. Some times the randomized algorithm produces the correct automaton and hence $L^{RMTS} = L$, other times the randomized algorithm fails to find any separating strings and L^{RMTS} is either empty or accepts all strings.

To get an idea of how well the approximation works we can use empirical analysis. We can get a good idea of the level of approximation by creating many automata and test each of them by comparing their answer to the oracles answer on a set of input strings.

In our tests, seen in table 8.1, we have for each entry run the algorithm 250 times[1] and then tested each of the generated automaton with 250 randomly

[1]250 times was chosen empirically, after 250 test-runs the result had converged and only varied slightly after this.

Language	Promise (k)	total # of strings	k tests	$k \log_2 k$ tests	k^2 tests	$k^2 \log_2 k$ tests	k^3 tests
$(0+1)^*11(0+1)^*$	3	3	100%	100%	100%	100%	100%
$000+010$	5	15	61.44%	100%	100%	100%	100%
$(0+1)^*0$	2	1	100%	100%	100%	100%	100%
$(000)^*$	4	7	62.32%	100%	100%	100%	100%
10^*1	4	7	94.48%	100%	100%	100%	100%
$(10)^* + (01)^*$	5	15	68.15%	88.24%	100%	100%	100%
$(0+1)^4$	6	31	81.01%	52.52%	100%	100%	100%
$1(0+1)^*1$	4	7	78.54%	100%	100%	100%	100%
$0+1$	3	3	100%	100%	100%	100%	100%
binval <= 10	7	63	1.202%	12.67%	75.97%	100%	100%
binval <= 15	6	31	1.422%	35.61%	100%	100%	100%
binval <= 25	9	255	0.8704%	0.8752%	14.93%	100%	100%

Table 8.1: Approximation results

selected strings. The entry value is the percentage of correct answers given by the constructed automata on the selected randomly selected strings.

8.5 Interpreting the results

Comparing the approximation results(table 8.1) with the table with the randomized results (table 7.5) we can draw a few conclusions.

There seems to be a general gain in all entries. This is to be expected as most automata-languages have some strings in common. This is what happens in the languages $(0+1)^4$ and $000+010$, as these languages reject far more strings than they accept. Thus an unfinished automaton, one with no final state, happens to approximate very well (because it always answers “no” and the correct answer is usually “no”). This is not necessarily good news as we usually want an automaton that has a chance to accept the strings in the language.

Even taking this possible error into account we can see a good general result. Especially when we increase k we get a good results allthough it is clear that some languages (like binval) do badly even on approximation.

8.6 Improving approximation

It is probably possible to do even better by throwing away obviously error-prone automata. One could for example rule out generated machines with no

accept-states (or all accept states). This possibility has, however, not been explored in this thesis.

Chapter 9

Tree-Automata

As seen throughout the thesis, the method of test sets works on deterministic automata. Because of the extreme simplicity of the deterministic automata the set of languages recognized by deterministic automata, known as the regular languages, is severely limited. We can strengthen our automata by generalizing the input structure.

9.1 Trees

Definition 9.1 Labelled Trees

By a labelledtree*, we mean a directed, connected graph with no cycles. Each node has fanin bound by some constant f. Each node is labeled with a symbol from Σ. The node with fanout = 0 is known as the* root *node and any node with fanin = 0 is known as a* leaf*. We will denote a collection of such trees over Σ as Σ^{**}*

Examples of such trees are seen in figure 9.1

Observation 9.2 *A tree with fanin bounded by 1 is a string.*

It is this somewhat trivial observation that gives us the stepping stone from DFA to tree automaton. A deterministic automaton which recognizes strings can be thought of as a tree-automaton which operates on trees of fanin = 1. The concept of deterministic automaton can thus be generalized to a tree automaton capable of recognizing trees of any chosen fanin.

9.2 The tree automaton

The basic idea of the tree-automaton is to start at each leaf with the starting state. Then read the symbol at the leaf and process this symbol according

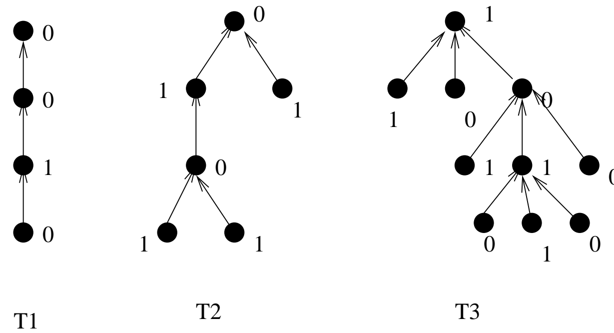


Figure 9.1: Trees over $\Sigma = \{0, 1\}$ with fanin 1, 2 and 3. Notice that the tree with fanin 1 is a string.

to the transition-function, this may cause a state change which is sent up through the tree. At internal nodes there may be more than one state propagated from the child-nodes, therefore our transition-function must be able to cope with several states at once. For an example of this idea, see Table 9.1 for the transitions function and Figure 9.2 for a evaluation of a tree using this transition function.

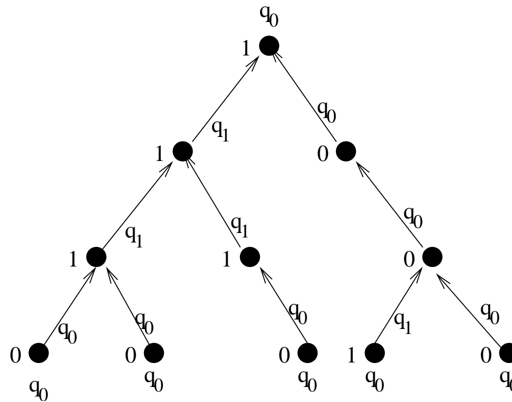


Figure 9.2: Evaluation of a tree using the transition-function in table 9.1, ending in state q_0 at the root.

Definition 9.3 Deterministic Finite Tree Automaton

A Tree Automaton *over* Σ *is a quintuple* $< Q, \Sigma, \delta, S, F >$ *and a bound* f. Q *is the set of states,* Σ *is the alphabet,* S *is the starting states and* F *are the final states of the automaton. The transition function* δ *is a function*

States	Symbols	δ(states,symbol)
q_0	0	q_0
q_0	1	q_1
q_1	0	q_1
q_1	1	q_1
q_0, q_0	0	q_0
q_0, q_0	1	q_1
q_0, q_1	0	q_0
q_0, q_1	1	q_1
q_1, q_0	0	q_0
q_1, q_0	1	q_0
q_1, q_1	0	q_0
q_1, q_1	1	q_1

Table 9.1: The transition-function in a tree-automaton with 2 states and fanin-bound = 2.

from $\bigcup_{1 \leq i \leq f}(\underbrace{Q \times \cdots \times Q}_{i \text{ times}} \times \Sigma)$ *to* Q.

9.3 Simulating Finite Tree Automata

In this section we determine how quickly we can determine if a tree is a language represented by the description of the tree automaton.

To simulate the run of a tree automaton on a tree T we can start at each leaf and move up the tree in a breadth-first manner. At each node, we use the propagated states from the nodes children and the nodes symbol in conjunction with the transition-table to calculate the next state. The cost in time at each node is dependent on the time to look up in the transition-table.

Lemma 9.4 *The size of the transition table in a tree automaton is* $\frac{|Q|^{f+1}-|Q|}{|Q|-1}\Sigma$.

Proof. There are $|\Sigma|$ table-entries for each combination of states. The possible combinations are limited by the fanin f at each tree-node. There can be from 1 to f states at each node. Thus the number of combinations can be expressed as:

$$\sum_{i=1}^{f} Q^i$$

Solving the expression:

$$\begin{aligned} x &= \sum_{i=1}^{i=f} |Q|^i \\ x &= |Q|^1 + |Q|^2 + |Q|^3 + ... + |Q|^f \\ |Q|x &= |Q|^2 + |Q|^3 + |Q|^4 + ... + |Q|^{f+1} \\ |Q|x - x &= |Q|^{f+1} - |Q|^1 \\ x(|Q| - 1) &= |Q|^{f+1} - |Q|^1 \\ x &= \frac{|Q|^{f+1} - |Q|}{|Q| - 1} \end{aligned}$$

And the proof is complete.□

9.3.1 Tree-Automaton Acceptance

Input:	A tree $t \in \Sigma^{**}$.
Parameter:	M, a Tree-Automaton
Question:	Is $t \in L(M)$.

Theorem 9.5 *Tree-Automaton Acceptance problem is in FPT as we can simulate a tree-automaton M on a Tree t in time $O(nC(\delta))$. Where n is the number of nodes in T and $C(\delta)$ is the cost of the transition-function.*

Given a description of a finite tree-automaton recognizing a language L, we can easily test if a tree $T \in L$. Start at each leaf and move up the tree in a breadth-first manner move up the tree. At each node, use the propagate states from its children and the nodes symbol in conjunction with the transition-table to calculate the next state. The cost in time at each node is dependent on the time to look up in the transition-table. The table is of size $\frac{|Q|^{f+1}-|Q|}{|Q|-1}\Sigma$ and the cost of each lookup $C(\delta)$ is thus $O(\frac{|Q|^{f+1}-|Q|}{|Q|-1}|\Sigma|)$ (naive linear lookup, could probably be done faster with binary search or a more complex storing procedure).

Thus the simulation takes $O(\frac{|Q|^{f+1}-|Q|}{|Q|-1}|\Sigma|n)$ time, where n is the number of nodes in the tree , f is the fanin and Q is the number of states in the automaton. This is obviously FPT as the $C(\delta)$ only depends on the parameter.□

Chapter 10

Future Work

This is a list of problem we would like to examine more closely.

10.1 Generalizing the Method of Test set

The method of testsets can be viewed more abstractly as “simplifying” a powerful machine (oracle) to a simpler machine model. Using the information given in the promise we can, by asking carefully selected questions, construct the simpler machine.

This could be formulated as the following problem:

GENERALIZED MTS

Input:	Given a decision procedure for a language B
Promise:	There exists a machine M' of type X, where $L(M') = B$ and M' has property P.
Question:	Construct M'?

It is clear that the property P must vary as we change to various machine models, but it is unclear how it varies.

10.2 Method of Test Sets and Tree-width

If a graph has bounded path-width (See definition 2.6) it is possible to give a partitioning of the vertices in the graph into bags, where the bags can be connected as a path using simple rules. Many graph-problems with bounded path-width have dynamic programming-algorithms which solve, given the bound as a parameter, many problems which for general graphs are NP-hard.

It is possible, allthough unpractical, to convert the dynamic programming-algorithms to deterministic automata. Therefore, the Method of Test Set and Randomized Method of Test can be applied to graph problems restricted to bounded pathwidth graphs.

However, graphs of bounded tree-width are much more general and of interest in practical settings, we would like to explore how graph problems with bounded tree-width can be solved using tree-automata. It is also of interest to explore the use of the Method of Test Sets on tree-automata and to check if the improvements we have made on the Method of Test Sets for string-automata can be applied to tree-automata as well.

The required lemmas for the method of test set: Lemma 5.10 which shows that only short strings are required for the test set and Lemma 5.9 which shows that the promise is needed, are as far as we know not proven for Tree-automata.

I would like to explore the use of Method of Test Set on tree-automata and too check if the improvements I have made on the Method of Test Set for string-automata can be applied to tree-automata.

I would also like to explore how problems on graphs with bounded tree-width can be solved using tree-automata.

10.3 Test Set Compression

Studying the tables of test set answer it is of interest to find the minimal table that still produced the correct automaton. This problem, which I believe may have a FPT-algorithm, can be formulated more generally as a matrix-problem.

TEST SET COMPRESSION

Input:	$M \times N$ (0,1)-matrix, where each row is unique in the matrix.
Parameter:	$k \in \mathbb{N}$
Question:	Are there k columns in the matrix that uniquely identifies the M rows, i.e

Bibliography

[1] N.Alon, R.Yuster, U.Zwick, “Color-coding: A new method of finding simple paths, cycles and other small subgraphs with large graphs,” in *Proceedings of the 26th Annual Symposium on the Theory of Computing*, ACM Press(1994) 326-335.

[2] S.Arnborg, D.G.Corneil, A. Proskurowski, Complexity of Finding Embeddings in a k-tree, *SIAM J.Alg and Discr. Methods 8(1987),277-287.*

[3] S. Arora, C.Lund, R.Motwani, M.Sudan, M.Szegedy, “Proof Verification and Intractability of Approximation Algorithms,” *Proceedings of the IEEE Symposium on the Foundations of Computer Science*,13-22,1992

[4] R. Balasubramanian, R.Downey, M.Fellows, V.Raman unpublished.

[5] H.L.Bodlaender,”A linear time algorithm for finding tree-decomposition of small tree-width,”*SIAM J. Comput.*, Vol.25 (1996) 1305-1317.

[6] Sam Buss, “Parameterized Complexity”, page 40. Springer-Verlag 1997

[7] Robert Downey, Mike Fellows, “Fixed-parameter tractability and completeness”, *Congressus Numerantium*, Vol.87 (1992) 161-187

[8] Robert Downey, Mike Fellows, “Parameterized Complexity”, Springer-Verlag 1997

[9] M. Fellows and M. Langston, “Fast search algorithms for graph layout permutation problems”,*Integration*,Vol.12(1991) 321-337.

[10] M.Hallett, G.Gonnet and U.Stege, “Vertex Cover Revisited: A Hybrid Algorithm of Theory and Heuristic,” manuscript, 1998.

[11] A. Nerode, “Linear automaton transformations’,” *Proc. Camb. Philos. Soc.*, Vol. 59 (1963) 291-296.

[12] M.Rabin and D.Scott, "Finite automata and their decision problems," *IBM J. Res.*, Vol. 3(2)(1959) 115-125

[13] N.Robertson and P.Seymour, Graph Minors XIII. The disjoint paths problem. Preprint.

[14] M. Sipser , "Introduction to the Theory of Computation", PWS Publishing Company, 1997

[15] U.Stege,"Resolving Conflicts in Problems in Computational Biochemistry," Ph.D. dissertation, ETH, 2000.