Automata-based algorithms for detecting minisatellites and satellites in DNA

Science is organized knowledge. Wisdom is organized life.

IMMANUEL KANT (1724 — 1804)

Corné de Ridder
Department of
Computer Science
University of Pretoria
Pretoria
South Africa, 002
`corne@reahl.org`

Abstract

This work is a significant extension of earlier research that was conducted in fulfilment of the requirements of an MSc degree in Computer Science at the University of Pretoria. The MSc research verified the hypothesis:

> *Finite automata can detect microsatellites effectively in deoxyribonucleic acid (DNA).*

The purpose of this thesis is to extend the above hypothesis to minisatellites and satellites in particular with regards to accuracy. Microsatellites, minisatellites and satellites are subsets of tandem repeats (TRs). A TR refers to two or more consecutive "motifs" contained in a genomic sequence. A perfect TR is defined as a string of nucleotides in which the motif is consecutively repeated at least twice. An approximate TR is a string of nucleotides where the motif is repeated consecutively at least twice, allowing for some differences between the instances.

Recent articles dealing with TR detection report that currently available TR detector algorithms do not exhaustively identify all possible TRs. Consequently some authors advocate that TRs from several TR detectors should be combined to ensure reliable TR detection.

In order to verify the hypothesis *Counting Finite Automata can detect minisatellites and satellites accurately in DNA* four related algorithms, collectively referred to as `FireSat`, are proposed. Of these $\texttt{FireSat}_1$, $\texttt{FireSat}_2$ and $\texttt{FireSat}_3$ have been implemented. The thesis compares the performance of these algorithms against the most relevant rival TR detecting software packages. It is found that $\texttt{FireSat}_3$ compares very well with the best of the available packages and frequently outperforms it. The conditions under which this occurs are highlighted.

The underlying principles of the fourth algorithm, $\texttt{FireSat}_{2'}$, have been implemented (although not the full algorithm), illustrating that $\texttt{FireSat}_{2'}$ would detect the exact same TRs as $\texttt{FireSat}_3$.

Although `FireSat` has default values, the user has several parameters that can be used to fine-tune her search. The objective is to allow the user to do an exhaustive search within the statistical constraints of her problem domain.

During the endeavour to develop algorithms that use finite automata for minisatellite and satellite detection in DNA, new types of automata were discovered (coined counting automata) that have the following properties:

- They can be used for tandem repeat detection to the same degree of accuracy as conventional finite automata.
- The languages they define can be classified in the Chomsky hierarchy.
- They can be implemented more efficiently than other finite automata, in terms of space.

Counting Finite Automata of Type 3 (CA_{T3}s) are defined as well as so-called Prototype CA_{T3}s (pCA_{T3}s). It is shown how these pCA_{T3}s can be cascaded to determine a Levenshtein based distance between two genetic substrings. A composition of CA_{T3}s, referred to as the Levenshtein Correspondence Automaton (LCA), is also proposed. The LCA is suitable for implementation on an FPGA or a GPU.

I am the wisest man alive for I know one thing, and that is that I know nothing.

— Socrates

Acknowledgements

"If I could say it in words there would be no reason to paint it." ... Edward Hopper.

Prof Kourie, I'll have to learn how to paint... I am falling short of words to express my gratitude towards you. Thank you for being part of my journey. Not only as an academic supervisor, but also as a life coach. Thank you for all the understanding, patience, support, encouragement, time and advice. Thank you for putting up with my nonsense.

A huge thanks to my husband Pieter and son Gustav, who implemented the theoretical ideas reported in this thesis. Thanks Pieter, for all the suggestions you've made and the hours you have spent assisting me. Thanks for your support over the years. Thanks Gustav for the finishing touches.

Thanks to my aunt, Lenca, who has supported me and believed in me over the years. Without your encouragement and loving support, I would not have been able to complete this PhD.

I experienced several challenges on my PhD journey. If I knew what was lying ahead of me, I would not have embarked on this journey. Self-discipline and perseverance are the two characteristics that made it possible for me to complete my PhD journey. A great thanks to my mum Elna, for teaching me these characteristics. Thanks also for your love and support.

One of the essential elements to study and write effectively is finding a good location. Thanks to Iwan for meeting this need by providing a study where I could sit and write for more than a year without distractions. Thanks also for your friendship, encouragement and support.

My mum also removed me from everyday life distractions — thank you so much for organizing the 10 days that we could spend in the country-side while I was writing. Thanks too, to my aunt Suna and cousin Derik for the 5 days near Dullstroom.

My thanks to my parents-in-law Koos and Gerda for their loving support — the food you have provided every second Monday for the past four months. Helen, thanks for editing my PhD.

Thanks to Oupa Jan, who kept me in his prayers and encouraged me throughout this journey.

The final month of this study had its own exceptional challenges. A special thanks to Nerina and my sister Elna who prayed me through this. Many thanks to Hendrien and Elsabe for their emotional, spiritual and energetic support.

A huge thanks to my sons, Franco and Gustav, who both supported me in a loving and protective way. Thanks to Anuschka for her encouragement. Thanks to Francois for his support and for verifying the comprehensibility of my *user guidelines*. Thanks to my loyal doggies, Liquorice, Sasha and Rex who were always near by when I was writing at home.

A special thanks to the late Prof Andries Barnard, who recognised my academic potential. Thanks to Daphne Becker who was always willing to assist me with my teaching duties, enabling me to spend more time doing research. Thanks to my late dad, Sam and my grandfather Willem, for the different positive contributions that they have made in my life.

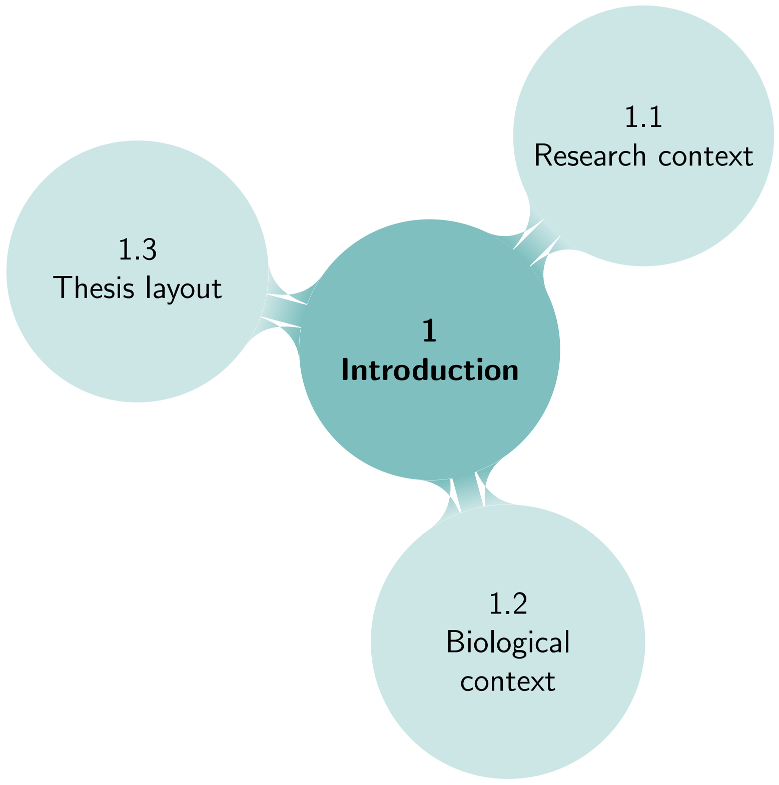
Many other people who I have not mentioned, have been influential in my life and in my work. I would not be who I am today without their loving support — my thanks to them for their guidance and help.

To my late husband Pieter and my sons Franco and Gustav.

Contents

List of Tables

List of Figures



1 Introduction

Without education, we are in a horrible and deadly danger of taking educated people seriously. ... G.K. Chesterton

---

Chapter 1 starts in Section 1.1 with a discussion of the research approach followed in conducting the research that was undertaken for this thesis. In Section 1.2, background to the biological issues that gave rise to the research is presented. The chapter ends with Section 1.3, which motivates the remainder of the thesis layout.

1.1 Research context

Glas et al. [2004] have analysed research undertaken in computing. They distinguish between three disciplines: Computer Science (CS), Software Engineering (SE) and Information Systems (IS). They mention that CS examines topics related to computer concepts at technical levels of analysis. This entails the formulation of processes *or* methods *or* algorithms. It also entails making use of mathematically-based conceptual analysis. CS does usually not rely on reference disciplines.[1]

Their analysis shows that the research approach adopted by about 80% of computer scientists is formulative; and about 90% of their research is self-referenced and self-reflective. With the exception of the research problem, whose origins is in the Bioinformatics discipline, and the statistical analysis of Chapter 9, the research described in this thesis relies on theorems and definitions that are mostly from the discipline of Computer Science. To this extent, it can be described as self-reflective. In addition, the research is formulative, contributing to the Computer Science knowledge framework in the following ways:

- New theoretical computer science machines called *counting automata (CAs)* are introduced. CAs of *type 1* (CA_{T1}), *type 2* (CA_{T2}) and *type 3* (CA_{T3}) are distinguished.
- *Prototype finite automata* (PFAs) are introduced. Deterministic PFAs (pDFAs), deterministic prototype CA_{T3}s (pCA_{T3}s) and non-deterministic CA_{T3}s (nPCA_{T3}s) are distinguished.
- The *cascade* operation on PFAs is defined.
- PFAs are cascaded to construct deterministic finite automata that detect tandem repeats in a brute force manner ($\mathtt{FireSat}_1$).
- PCA_{T3}s are cascaded to construct non-deterministic automata that detect tandem repeats by computing Levenshtein-based distances ($\mathtt{FireSat}_2$).
- pNCA_{T3}s are cascaded to construct non-deterministic counting automata that compute Levenshtein distances. In principle these can be used to detect TRs ($\mathtt{FireSat}_{2'}$).
- A Levenshtein Correspondence Automaton (LCA) is defined for the first time and used for detecting tandem repeats ($\mathtt{FireSat}_3$). It is built from a particular arrangement of CA_{T3}s.
- The *normalised Levenshtein Correspondence* measure, LC_n, is defined for the first time and used as a so-called *match score.*
- A *Recall-Precision* (RP) analysis is performed on output generated by $\mathtt{FireSat}_3$. The results obtained from the analysis were used to derive a so-called *match score function*—a function of the LC_n. The match score threshold function, multiplied by a threshold factor, is used to establish

[1]By "reference disciplines" Glas et al. [2004] mean other disciplines whose theories formed a basis for the research. Self-reference indicates reference to theories or papers in the discipline under examination. Economics is given as an example of a reference discipline for IS.

suitable nucleotide positions for the start and end of tandem repeat elements (TREs). To the best of my knowledge, this is the first time that such an RP analysis has been done on output generated by a tandem repeat detector for the purpose of optimising a fit.

- Utilizing some of the newly introduced theoretical concepts, four algorithms are introduced: $\texttt{FireSat}_1$; $\texttt{FireSat}_2$; $\texttt{FireSat}_{2'}$ and $\texttt{FireSat}_3$. Collectively they are referred to as `FireSat`. `FireSat` was implemented.

  In Chapter 9 it is demonstrated that in most circumstances, $\texttt{FireSat}_3$ and $\texttt{FireSat}_{2'}$ are somewhat more accurate minisatellite and satellite detectors than other existing TR detectors. From the detection results presented in Chapter 9 it is clear that $\texttt{FireSat}_2$ appears to be a highly accurate TR detector alternative, although practical challenges are presented by its implementation. $\texttt{FireSat}_1$ was not as effective at TR detection. However, $\texttt{FireSat}_1$ still compared favourably against other existing TR detectors whenever TRs with longer motif lengths should be detected.

The research approach followed in this thesis is therefore self-reflective and formative in much the same way as many other research undertakings within the Computer Science domain.

In addition, the research may be characterised as quantitative. Quantitative research can be either experimental or quasi-experimental (Goubil-Gambrell [1991]). The research conducted here is experimental in the sense that it includes the manipulation of variables that gives rise to various items of empirical[2] data. This data is then compared to data generated by functionally similar algorithms.

Since the proposed algorithms were implemented, it was possible to measure their effectiveness in a quantitative manner.[3] Using synthesised input data, the resulting output generated by various `FireSat` implementations was compared against the output generated by carefully selected contender algorithms. As described in Chapter 9, the comparison relies on statistical measures that are explained in detail.

Before undertaking the research for this thesis, I had previously investigated a selection of prominent software packages which are functionally similar to the algorithm developed here. Chapter 3 refers to these investigations and cites the associated publications which I either authored or co-authored. The chapter also

[2] Empiricism advocates the idea that observation and measurement is the core of the scientific endeavour (Trochim [2006]).

[3] $\texttt{FireSat}_{2'}$ was not fully implemented. Only the underlying theoretical principles of $\texttt{FireSat}_{2'}$ were implemented, illustrating that the $\texttt{FireSat}_{2'}$ NCA_{T3} can calculate both the LC and the LD.

summarises relevant algorithms whose implementations were not available on the web during those earlier investigations.

Put more generally, this study adopts a positivistic stance. Positivists prefer research methods that start with precise theories from which verifiable hypotheses can be extracted and tested in isolation (Easterbrook et al. [2007]). This research can also be characterised as having been conducted in a deductive manner. Deductive reasoning proceeds from the more general to the more specific—thus from theory to hypothesis to observation and (hopefully) to confirmation (Trochim [2006]). In the present case, these deductive phases have been realised as follows:

- *Theory:*
  Algorithms to detect tandem repeats have been proposed. They rely on new theoretical concepts such as the cascading operation that is applied to the prototype automata mentioned above. These newly introduced concepts, introduced in Chapter 2, are located within the context of pre-existing theoretical theorems and definitions from the field of Computer Theory[4]. In addition, the algorithms that offer solutions to the tandem repeat detection problem are assessed on the basis of theoretical principles from *Detection Theory* and entail an RP analysis, also mentioned above.

- *Hypothesis:*

  > *Finite Automata*[5] *can be used to accurately detect minisatellite and satellite tandem repeats*[6] *in DNA.*

  In order to investigate this research hypothesis, four different `FireSat` algorithms were developed that rely on four respective tailored automata. As mentioned above, $\texttt{FireSat}_1$ relies on PFAs, $\texttt{FireSat}_2$ relies on pCA$_{T3}$s, $\texttt{FireSat}'_2$ relies on pNCA$_{T3}$s and $\texttt{FireSat}_3$ makes use of CA$_{T3}$s. The extent to which each of these algorithms can accurately detect minisatellites and satellites is empirically explored.

- *Observation:* As previously mentioned, the suggested algorithms have been implemented and their output subsequently compared against the output of rival algorithms.

- *Confirmation:* By observing and comparing algorithm detection performance, an assessment was made of the extent to which the original hypothesis has been attained and of the shortcomings that exist.

[4] Computer Theory constitutes mathematical models that describe, with various degrees of accuracy, parts of computers, types of computers and similar machines (Cohen [1997]).

[5] See Chapter 2, Definition 2.3.1.

[6] Tandem repeats are defined in Chapter 2, Section 2.1 where there is also distinguished between minisatellites (see Definition 2.1.26) and satellites (see Definition 2.1.27).

1.2 Biological context

There were two sources of inspiration for this research. On the one hand, it was found that molecular biologists were visually scanning genetic sequences in order to detect specific patterns such as tandem repeats. On the other hand, in trial runs of various implemented algorithms that purport to detect repeats, output reported often differed significantly from one algorithm to the next. The differences were often traceable to differing interpretations of features that the sought-after repeats should possess.

This corresponds to the findings of Schaper et al. [2015], who motivate the development of *TRAL* (Tandem Repeat Annotation Library) by stating that tandem repeat detector algorithms generate heterogeneous, partially complementary as well as conflicting results. In addition, they maintain that currently available tandem repeat detection algorithms do not search exhaustively. One of the objectives of `FireSat` was to search as exhaustively as possible by providing parameter settings that enable the user to specify the precise features of repeats for which to search.

As in my earlier research, this current research endeavour contributes to Bioinformatics, a subset of Computational Biology. While the main focus of this thesis is the development of algorithms that contribute to the analysis of DNA, an understanding of the biological context of tandem repeats and of other competitive software packages is also required. Consequently, reflective research[7] has been conducted in order to enhance this understanding.

To provide readers who are computer scientists with some biological context, I repeat an earlier discussion to be found in De Ridder [2010].

Computational Biology is defined by BISTIC[8] as

> "...the development and application of data-analytical and theoretical methods, mathematical modeling and computational simulation techniques to the study of biological, social and behavioural systems."

Bioinformatics is defined by Luscombe et al. [2001] as

> "...the application of computational techniques to understand and organize the information associated with biological macromolecules."

[7]Reflective research includes the systematic and persistent inquiry into already existing knowledge (Kressel [1997]).

[8]The Biomedical Information Science and Technology Initiative Consortium of the National Institution of Health of the United States Of America (National Institute of Health of the United States of America. BISTIC Definition Committee [2000]).

Bergeron [2003] mentions that Bioinformatics distinguishes itself from other scientific endeavours in the sense that it focuses on the information encoded in the genes and how this information affects the universe of biological processes.

DNA is the polymer molecule[9] that stores genetic information of organisms (Paces [2001]). An organism's genetic information is encoded in DNA as a sequence of four different nitrogenous bases on a sugar-phosphate backbone. DNA can adopt various conformations, including the double helix structure. The four nitrogenous bases (nucleotides) are Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). The sequence of the four nitrogenous bases mirror each other in a predefined manner in each strand of the double helix mirror: Adenine on the one strand always binds with Thymine on the other, and Cytosine always binds with Guanine (Bergeron [2003]). Therefore, the sequence ATTGCA will occur as TAACGT on the complementary strand of the helix (Paces [2001]).

Examples of genetic databases that store nucleotide sequences include GenBank[10], DDBJ[11], EMBL[12], MGDB[13], GSX[14], NDB[15] (Bergeron [2003]) and GeneCards (Safran et al. [2002]).

DNA molecules are subject to numerous mutational events. One of the consequences of these events that can be detected by computationally analysing genome sequences, is tandem duplication. In tandem duplication a stretch of DNA, which we call a *motif*, is converted to one or more "copies", each following the preceding one in a contiguous fashion. These copies may or may not be exact.

A perfect tandem repeat is a string of nucleotides in a genomic sequence whose initial substring (of some arbitrary length), is followed by one or more adjacent exact copies of that substring.

In contrast, an approximate tandem repeat is a genomic sequence whose introductory substring (or motif) is followed by one or more substrings, of which at least one need not necessarily be an *exact* copy of the motif. The extent to which these non-exact copies may vary from the motif is limited, as will be discussed later in this thesis. A tandem repeat refers generically to either a perfect tandem repeat or an approximate tandem repeat.

Consecutive repetitive DNA sub-sequences are of relevance in biology for various reasons. In this regard Lim et al. [2012] mention that tandem repeats are essential in both biological and medical research. The identification of tandem repeats

[9]Polymer molecules are large molecules consisting of repeated chemical units joined together.
[10]One of the largest public sequence databases.
[11]DNA Data Bank of Japan.
[12]European Molecular Biology Laboratory.
[13]Mouse Genome Database.
[14]Mouse Gene Expression Database.
[15]Nucleic Acid Database.

in DNA has a particular significance in genetic research. Tandem repeats are important as genetic markers.

There are more than 30 inherited human diseases that can currently also be identified by detecting tandem repeats in DNA. The expansion of simple DNA repeats has been linked to heredity disorders in human beings, including Fragile X Syndrome, Mytonic Dystrophy, Huntington's disease, spinal and bulbar muscular atrophy, various Spinocerebellar Ataxias and Friedreich's Ataxia. These diseases are often referred to as repeat expansion diseases. The reason for this is that they are caused by long and highly polymorphic tandem repeats.[16]

Tetra- or penta-nucleotide tandem repeats in the human genome are the genetic markers used in DNA forensics. Since the number of adjacent repeat units varies from individual to individual, the number of copies of motifs can be used to identify an individual as well as relationships such as parent or grandparent relationships.

Tandem repeats are also used in population studies, in conservation biology as well as in multiple sequence alignments (Lim et al. [2012]). Kolpakov et al (2003) mention that the presence of repeats can be seen as a fundamental feature of genomes — a repeat is the simplest form of regularity in sequences. By analysing repeats first, clues are gained which may lead to the discovering of new biological phenomena (Kolpakov et al. [2003]). Note, that this is similar to how repeated words give a starting point to deciphering a script when written in an unknown language.

In the literature, a distinction is made between interspersed repeats and tandem repeats. Interspersed repeats refer to repeated DNA sequences located at dispersed regions in a genome. These repeats are also known as mobile elements or transposable elements. An interspersed repeat occurs if a stretch of DNA (sequence of nucleotides) is copied to a different location through DNA recombination (Lee [1996], Pestronk [2005]).

Tandem repeats in eukaryotic genomes are involved in various gene regulatory functions including participation in protein binding, affection of the chromatin structure[17] and in heat-shock inducible expression mechanisms (Kolpakov et al. [2003]). In higher organisms, tandem repeats are associated with recombination hot-spots[18]. A relationship has been established between the recombination in-

[16]Polymorphisms refer to variations in DNA sequences of individuals. Polymorphisms related to trinucleotide diseases occur if the number of tandem repeat elements expand far beyond their normal ranges (Fan and Chu [2007]).

[17] Chromatin designates the structure in which DNA exists within cells. Chromatin structure is determined and stabilized through the interaction of the DNA with the DNA-binding proteins (King [2012]).

[18] Recombination hotspots are short regions of around 2Kb in length that have significantly higher recombination rates than their neighbouring regions (Fearnhead [2006]).

tensity and the density of GT-repeats in human chromosomes. An increase of the male recombination level in regions rich in tandem repeats with a motif length between 10 and 100 base pairs (bps) has been observed (Kolpakov et al. [2003]).

Tandem repeats are also conserved in prokaryotes and in plasmids[19] as well as in genomic DNA. Furthermore a correlation has been observed between certain repeats and virulence factors of bacteria.

Given the importance of known and potential biological roles for repeats as briefly outlined above, it seems essential to develop an efficient and sensitive algorithm to detect these repeats, so that they may receive further study.

`FireSat` relies on the implementation of finite automata to search for repeats. In De Ridder [2010] three algorithms, referred to as $\texttt{Fire}\mu\texttt{Sat}_1$, $\texttt{Fire}\mu\texttt{Sat}_2$ and $\texttt{Fire}\mu\texttt{Sat}_3$ were proposed to search for microsatellites[20]. `FireSat` utilizes some of the principles of $\texttt{Fire}\mu\texttt{Sat}_2$ and $\texttt{Fire}\mu\texttt{Sat}_3$. In principle, `FireSat` can search for a repeating motif of *any* length, i.e. for tandem repeats in general.

One should note that there are numerous algorithms that search for microsatellites.[21] However, it is a computationally harder problem to find minisatellites[22] or satellites[23] in DNA. Clearly, the longer the motif length, the more characters need to be compared. Consequently, there are fewer algorithms that detect minisatellites and satellites in DNA.

1.3 Thesis layout

This thesis is organised as follows.

- In Chapter 2 definitions and background definitions from two subject disciplines, molecular biology and automata theory, relevant to the remainder of this thesis are provided. Definitions for PFAs, nPFAs as well as the *cascade* operation are included in this chapter.

[19] Prokaryotes are single cell organisms including bacteria (Bailey [2012]).
A plasmid is a small, circular unit of DNA that replicates within a cell independently of the chromosomal DNA. Plasmids are often found in bacteria. Certain plasmids can insert themselves into chromosomes at spots where there is a common sequence of nucleotides. Plasmids contain a small number of genes that code for proteins, and in particular for enzymes, some of which confer resistance to antibiotics. Plasmids are used in recombination DNA research, to transform bacterial cells (American Heritage Dictionary [2010]).

[20] Microsatellites are defined in detail in Chapter 2, Definition 2.1.20.

[21] A number of these implemented algorithms can be accessed at http://en.wikipedia.org/wiki/Microsatellite_genetics.

[22] Minisatellites are defined in detail in Chapter 2, Definition 2.1.26.

[23] Satellites are defined in detail in Chapter 2, Definition 2.1.27.

- Chapter 3 entails a literature overview. Reference is made to publications authored or co-authored by me where algorithms aiming to detect tandem repeats are discussed. Furthermore in Chapter 3 concept lattices are used to classify TR detecting algorithms.
- Chapter 4 introduces the concept of a *counting automaton*. It proposes definitions for three types of counting automata, CA_{T1}, CA_{T2} and CA_{T3}, and positions of the languages that they generate within the Chomsky hierarchy. These formalisms evolved by reflecting on prior experience in developing microsatellite detection algorithms described in De Ridder [2010]. Counters assigned to finite automata states in those algorithms were compared against pre-assigned integer values. It was realised that by generalising such counter comparisons in various ways, sublanguages within the Chomsky hierarchy could be defined.

  The automata previously used for microsatellite detection in $\texttt{Fire}_\mu\texttt{Sat}_1$ and $\texttt{Fire}_\mu\texttt{Sat}_3$ were in fact CA_{T3}s. At the time of publishing $\texttt{Fire}_\mu\texttt{Sat}$, CA_{T3}s have not been formally defined.
- Chapter 5 provides a number of underlying principles relevant to the `FireSat` variants.
- Chapter 6 presents $\texttt{FireSat}_1$: utilising PFAs together with the cascade operation.
- In Chapter 7, $\texttt{FireSat}_2$ and $\texttt{FireSat}_{2'}$ are introduced as alternative approaches to tandem repeat detection. $\texttt{FireSat}_2$ relies on the cascading of pCA_{T3}s, whereas $\texttt{FireSat}_{2'}$ cascades pNCA_{T3}s.
- $\texttt{FireSat}_3$, discussed in Chapter 8, describes how to compose CA_{T3}s to construct an LCA. This specialised automaton delivers the Levenshtein correspondence between two strings and is used for estimating tandem repeat elements.
- Chapter 9 applies detection theory and statistics to carry out an RP-analysis, resulting in a match score threshold function. Implementations of `FireSat` can use this function, modified by a user-selected match score factor, to estimate where tandem repeats occur in data. This chapter also reports on the accuracy of rival TR detectors in comparison to `FireSat`.
- Chapter 10 concludes this thesis and points to future research possibilities.

For the convenience of the reader after each chapter a bibliography of the chapter's references is provided.

Chapter References

American Heritage Dictionary. The American Heritage Science Dictionary. Houghton Mifflin Harcourt Publishing Company, 2010.

R. Bailey. The cell-cell structure. Online: http://biology.about.com/od/cellanatomy/a/eukaryprokarycells.htm, 2012.

B. P. Bergeron. *Bioinformatics Computing.* Prentice-Hall/ Professional Technical Reference: Upper Saddle River, NJ, 2003.

D. I. A. Cohen. *Introduction to Computer Theory.* Wiley: New York, 2nd ed. edition, 1997.

C. De Ridder. Flexible Finite automata-based algorithms for detecting microsatellites in DNA. Master's thesis, Department of Computer Science, University of Pretoria, July 2010.

S. Easterbrook, J. Singer, M-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. *Guide to Advanced Empirical Software Engineering*, 2007.

H. Fan and J-Y. Chu. A brief review of short tandem repeat mutation. *Genomic Proteomics Bioinformatics*, 5(1): 7—14, 2007.

P. Fearnhead. SequenceLDhot: detecting recombination hotspots. *Bioinformatics*, 22(24):3061–3066, 2006.

R. L. Glas, V. Ramesh, and I. Vessey. An analysis of research in Computing disciplines. *Communications of the ACM*, 47(6), June 2004.

P. Goubil-Gambrell. What do Practitioners need to know about Research Methodology? *Directions in Technical communications research*, 1991.

M.W. King. The medical biochemistry page. Online: http://themedicalbiochemistrypage.org, February 2012.

R. Kolpakov, G. Bana, and G. Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *PubMed Central Nucleic Acid Research* , 31(13):3672–3678, 2003.

K. Kressel. Practice-Relevant Research in Mediation: Toward a Reflective Research Paradigm. *Negotiation Journal*, 13(2):143–160, 1997.

F. Lee. *Interspersed repeats and Tandem Repeats in the Molecular Biology Web Book*, chapter 3. www.web-books.com, 1996.

K. G. Lim, C. K. Kwoh, L. Y. Hsu, and A. Wirawan. Review of tandem repeat search tools: a systematic approach to evaluating algorithmic performance. *Briefings in Bioinformatics*, May 2012.

N. M. Luscombe, D. Greenbaum, and M. Gerstein. What is Bioinformatics? A proposed definition and overview of the field. *Methods of Information in Medicine*, 40:346–358, 2001.

National Institute of Health of the United States of America. BISTIC Definition Committee. NIH working definition of Bioinformatics and Computational Biology. Online: http://www.bisti.nih.gov/CompBioDef.pdf, July 2000.

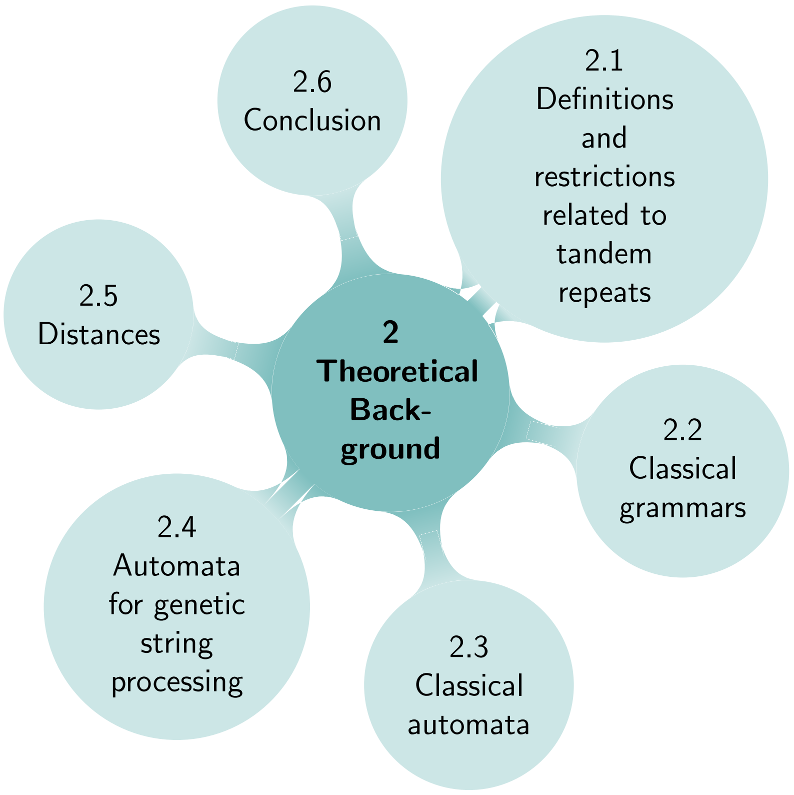
J. Paces. Bioinformatics: Tools for analysis of Biological sequences. *In: Proceedings of the Prague Stringology Conference 2001*, pages 50–58, 2001.

A. Pestronk. DNA repeat sequences and diseases. Online: http://www.neuro.wustl.edu/neuromuscular /mother/dnarep.htm, Sept 2005.

M. Safran, I. Solomon, O. Shmueli, M. Lapidot, S. Shen-Orr, A. Adato, U. Ben-Dor, N. Esterman, N. Rosen, I. Peter, T. Olender, V. Chalifa-Caspi, and D. Lancet. GeneCardsTM 2002: Towards a complete, object-oriented, human gene compendium. *Bioinformatics Applications Note*, 18(11):1542–1543, 2002.

E. Schaper, A. Korsunsky, A. Messina, R. Murri, J. Pečerska, H. Stockinger, S. Zoller, I. Xenarios, and M. Anisimova. TRAL: Tandem repeat annotation library. *Bioinformtics*, 31(18):3051–3053, 2015.

M. K. Trochim. Deduction and Induction. Online: http://www.socialresearchmethods.net, 2006.



2 Theoretical Background

“A good notation has a subtlety and suggestiveness which at times make it almost seem like a live teacher.” … Bertrand Russell

---

This chapter presents definitions originating from both molecular biology and from computer theory. The definitions are of relevance throughout the remainder of this thesis, contributing to an understanding of the context and of the proposed algorithms. In the context of this thesis a string (or word[1]) w is a sequence of characters drawn from some set of characters, say Σ, called an alphabet. In this text, $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$ unless otherwise stated. The length of a string w is denoted by $|w|$. Λ will be used to denote a special string, the so-called empty string. Note that $|\Lambda| = 0$. Also note that characters in a string will sometimes be placed in parenthesis to indicate relevant substrings[2], as for example in $(\mathtt{acg})\mathtt{act}(\mathtt{acg})\mathtt{acg}$.

The remainder of this chapter is laid out as follows.

- In Section 2.1 definitions and restrictions related to tandem repeats are provided. These definitions contribute towards the clarification of the *nature* of the tandem repeats to be detected. Within our context *nature* refers to

[1]The words *word* and *string* are used interchangeably throughout this text.
[2]A substring is a contiguous sequence of characters within a string.

the type of tandem repeats to be detected — implemented algorithms can detect either *perfect tandem repeats* (see Definition 2.1.1) or *approximate tandem repeats* (see Definition 2.1.4).

From Masombuka et al. [2010] and Schaper et al. [2015] it is clear that repeat detection algorithms that have the same objective, namely to detect tandem repeats, do not necessarily detect and/or report exactly the same tandem repeats. Thus, the *nature* of tandem repeats detected by various algorithms differ. One of the objectives of this research is to determine the extent to which `FireSat` is capable of detecting tandem repeats reported by other algorithms.

- The next two sections are devoted to classical concepts in theoretical computer science that are relevant to this work, namely grammars (Section 2.2) and automata (Section 2.3). Readers who have the necessary background may skip these sections.
- Section 2.4 proposes and defines so-called prototype finite automata. They are classical finite automata, limited in certain ways so as to be useful in the context of genetic string processing.
- In approximate string matching the so-called distance between a source and a destination string is calculated. Section 2.5 provides background to the calculation of distances. The same section also introduces the so called *Levenshtein Correspondence* (LC) as well as the *normalised LC* (LC_n).
- The chapter is concluded in Section 2.6.

Note that this thesis also proposes several variants of a *new* type of automaton, generically called a counting automaton. Since the present chapter is focussed on classical automata, material relating to these new types of automata is deferred until Chapter 4. It will be seen there that, though based on finite automata, definitions of counting automata allow for certain operations and conditions to be associated with the transitions. Chapter 4 will also relate the various resulting types of counting automata to the well-known Chomsky hierarchy. Though the Chomsky hierarchy is foundational to computer science language theory and should therefore be discussed in the current chapter that focusses on classical background material, it was decided to defer such a discussion to Chapter 4 as well.

2.1 Definitions and restrictions related to tandem repeats

In this section definitions and restrictions related to tandem repeats (TRs) are provided. The terminology introduced in this section is specifically used in Chapters 3, 5, 6, 7 and 8. In Section 2.1.1 definitions related to TRs are defined. Section 2.1.2 deals with motif error definitions. Definitions dependent on the repetitive motif length of a TR are presented in Section 2.1.3.

The origin of definitions given in this section varies — they can be divided roughly into four groups as follows:

- Definitions that were present in the literature at the time that I commenced with my TR (microsatellite) detection studies. These definitions are indicated with a $^\oplus$ next to their headers.
- Definitions that I proposed in De Ridder [2010] are indicated as such using a $^\oslash$.
- Some of the definitions are introduced in this thesis for the first time. Those are indicated with $^\odot$.
- Definitions that were originally presented in De Ridder [2010] but that have been modified according to new insight. These definitions are presented here accompanied by more than one of the applicable characters.

2.1.1 Pattern matching: tandem repeats

Terminology related to TRs is introduced in this section.

Definition 2.1.1. ***Perfect tandem repeat (PTR)*** $^\oplus$
A PTR is a string of nucleotides characterised by a certain introductory string followed by one or more exact copies of that introductory string. The introductory string is called the motif.[3] □

Example 2.1.2. *For the motif* `acg`, `acgacgacgacgacgacg` *is a PTR.* □

[3]Note that there is no biological explanation as to "why" the introductory string is called the motif. This thesis follows the convention of Rivals et al. [1995] that designates the introductory string as the motif. TR-detection is in a sense "fuzzy" — there may be more than one correct answer to TR detection. It is illustrated in the literature that there does not exist two TR detectors that detect exactly the same minisatellites and satellites (Masombuka et al. [2010], Schaper et al. [2015]).

Definition 2.1.3. ***Perfect tandem repeat element (PTRE)*** $^{\oslash}$
A PTRE is an exact copy of the introductory motif within a TR. $\square$

Definition 2.1.4. ***Approximate tandem repeat (ATR)*** $^{\oplus}$
An ATR is a string of nucleotides characterized by a certain motif that introduces the string followed by one or more adjacent "copies" of the motif. In the case of ATRs, at least one motif copy will not be exact. $\square$

Example 2.1.5. `(acg)act(acg)acg(acg)`.
Here, `acg` *is the motif. It is repeated "approximately" five times. The second "copy",* `act`*, is not exact since the* `g` *in the motif has been replaced by* `t`*.* $\square$

Definition 2.1.6. ***Approximate tandem repeat element (ATRE)*** $^{\oslash}$
An ATRE in an ATR is a non-exact copy of the introductory motif. $\square$

Note that whether or not a nucleotide string may be deemed to be an ATR is context-dependant or user-dependent—i.e. different contexts or users may have different tolerances for the number of inexact motif copies in an ATR and for the extent to which a given motif "copy" may differ from the motif itself. There are various so-called mutational events which cause the differences to arise. Furthermore, the extent of inexactness within an ATR clearly has to be within *certain bounds*. These themes will be addressed in Section 2.1.2.

Definition 2.1.7. ***Tandem repeat (TR)*** $^{\oplus}$
A TR is a string of nucleotides consisting of perfect tandem repeat elements (see Definition 2.1.3) and/or approximate tandem repeat elements (see Definition 2.1.6). $\square$

Definition 2.1.8. ***Tandem repeat element (TRE)*** $^{\oslash}$
A TRE is either a PTRE or an ATRE within a TR. $\square$

A TRE is thus any motif copy, exact or approximate, in a TR.

Example 2.1.9. *In the case of* `acg(acg)act(acg)acg`*,* `acg` *is a TRE that occurs*[4] *four times, and* `act` *is a TRE that occurs once.* $\square$

2.1.2 Motif error

Motif errors occur in ATREs.

[4]In this thesis the words *occurrence (noun)* and *occur(s) (verb)*, are interchangeably used to refer to *presence*, typically of a character or substring within a string or substring. The meaning should be clear from the context.

Definition 2.1.10. *Motif error* (ε) $^{\oplus}$
A motif error is an irregularity within an ATRE that may be attributed to a mutational event. The total number of motif errors in a given string is generally denoted by ε and is called the motif error of the string. Where clear from the context, the term motif error should be assumed to refer to the total number of motif errors in the string under consideration. □

The mutational events that give rise to motif errors are classified as *mismatches, deletions* or *insertions.*

In the examples below, a string ρ will represent a motif. ρ^p represents the string ρ repeated p times. ρ^p is thus a PTR.

A string u is considered similar to the string ρ^p if it can be written as $u = u_1u_2 \cdots u_p$ where each word u_k $(k = 1 \cdots p)$ is obtained by at most ε_{max} mutations on ρ.

In the examples to follow, assume that $\rho =$ `acgtac` and thus that $|\rho| = 6$. It is assumed that $u = u_1u_2 \cdots u_p$ is an ATR based on ρ^p. The examples show the set of mismatch strings, the set of deletion strings and the set of insertion strings, respectively, when $\varepsilon = 1$, in each string.

To enable accurate communication, within the context of TR detection in this thesis, it is meaningful to distinguish between:

- *Motif error* (ε): the number of motif errors occurring in ρ.
- *Motif error percentage* ($\varepsilon_{\%}$):

$$\varepsilon_{\%} = \frac{\varepsilon}{|\rho|} \times 100$$

- *Maximum number of motif errors* (ε_{max}): an integer value representing the maximum number of motif errors to be tolerated in ρ.
- *Maximum motif error percentage* ($\varepsilon_{max\%}$): a percentage value indicating the maximum percentage of nucleotides that an ATRE, deduced from a certain ρ, may have. Clearly the calculation of this value is equivalent to a maximum that is specified for $\varepsilon_{\%}$ and calculated in the same manner.

Definition 2.1.11. *Mismatches* $^{\oplus}$
A mismatch(es) occurs in the word u_k if $|u_k| = |\rho|$ and one or more bases, mismatch the bases, in the same respective positions of ρ. Thus the word "mismatch" is used to define the term mismatch. □

Example 2.1.12. *Assuming $\varepsilon_{max} = 1$, then u_k derived from $\rho =$ `acgtac`, with a mismatch are as follows:*

$$\begin{aligned} u_k &\in \{\texttt{xcgtac} \mid \texttt{x} : \{\texttt{c}, \texttt{g}, \texttt{t}\}\} \\ &\cup \{\texttt{axgtac} \mid \texttt{x} : \{\texttt{a}, \texttt{g}, \texttt{t}\}\} \\ &\cup \{\texttt{acxtac} \mid \texttt{x} : \{\texttt{a}, \texttt{c}, \texttt{t}\}\} \\ &\cup \{\texttt{acgxac} \mid \texttt{x} : \{\texttt{a}, \texttt{c}, \texttt{g}\}\} \\ &\cup \{\texttt{acgtxc} \mid \texttt{x} : \{\texttt{c}, \texttt{g}, \texttt{t}\}\} \\ &\cup \{\texttt{acgtax} \mid \texttt{x} : \{\texttt{a}, \texttt{g}, \texttt{t}\}\}. \end{aligned}$$

In all these cases $|u_k| = 6$. □

Definition 2.1.13. ***Deletions*** $^\oplus$
The word u_k derived from ρ where a deletion occurs, is the string ρ from which at least one nucleotide has been deleted. Thus the word "deletion" is used to define the term deletion.

□

Example 2.1.14. *The deletions of ρ when $\varepsilon_{max} = 1$ are given by:*

$$\begin{aligned} u_k &\in \{\texttt{cgtac}\} \\ &\cup \{\texttt{agtac}\} \\ &\cup \{\texttt{actac}\} \\ &\cup \{\texttt{acgac}\} \\ &\cup \{\texttt{acgtc}\} \\ &\cup \{\texttt{acgta}\}. \end{aligned}$$

Thus, in all these cases $|u_k| = 5$. □

Definition 2.1.15. ***Insertions*** $^\oplus$
A word u_k contains an insertion if at least one nucleotide is inserted in front of any position of ρ. Thus the word "insertion" is used to define the term insertion.

□

Example 2.1.16. *If $|\rho| = 6$ and $\varepsilon_{max} = 1$ then u_k can have insertions as follows:*

$$\begin{aligned} u_k &\in \{\texttt{xacgtac} \mid \texttt{x} : \{\mathsf{a},\mathsf{c},\mathsf{g},\mathtt{t}\}\} \\ &\cup \{\texttt{axcgtac} \mid \texttt{x} : \{\mathsf{a},\mathsf{c},\mathsf{g},\mathtt{t}\}\} \\ &\cup \{\texttt{acxgtac} \mid \texttt{x} : \{\mathsf{a},\mathsf{c},\mathsf{g},\mathtt{t}\}\} \\ &\cup \{\texttt{acgxtac} \mid \texttt{x} : \{\mathsf{a},\mathsf{c},\mathsf{g},\mathtt{t}\}\} \\ &\cup \{\texttt{acgtxac} \mid \texttt{x} : \{\mathsf{a},\mathsf{c},\mathsf{g},\mathtt{t}\}\} \\ &\cup \{\texttt{acgtaxc} \mid \texttt{x} : \{\mathsf{a},\mathsf{c},\mathsf{g},\mathtt{t}\}\}. \end{aligned}$$

In all these cases $|u_k| = 7$.

□

In the current context the destination string is taken to be a PTRE and the source string an ATRE.

Definition 2.1.17. ***Consecutive motif errors allowed*** ($\kappa_{max\%}$) $^{\odot}$
$\kappa_{max\%}$ *is a user-determined percentage value specifying the maximum allowable percentage of motif errors adjacent to one another within an ATRE.* **`FireSat`** *will only consider a substring to be a TR if the consecutive motif error,* $\kappa_{\%}$*, on all its ATREs is* $\leq \kappa_{max\%}$*. The counter* κ *keeps track of the number of motif errors occurring consecutively within an ATRE. Clearly* $\kappa_{\%} = \dfrac{\kappa}{|\rho|} \times 100$. □

Example 2.1.18. *Suppose* $\rho =$ `acgtcaaaaa`; $\varepsilon_{max\%} = 40\%$ *and* $\kappa_{max\%} = 40\%$. *Then* `acgtctttta` *may be considered a "copy" of* ρ. *However, if* $\kappa_{max\%} = 30\%$ *then* `acgtctttta` *may not be considered a "copy" of* ρ *since* $\kappa_{max\%} < \kappa\%$ *where* $\kappa\%$ *is the percentage of consecutive motif errors, namely* 40%. *On the other hand,* `aggtcttaag` *is a valid "copy" of* ρ*—even though it contains* 40% *mismatches (which does not exceed the percentage of permitted motif errors indicated by* $\varepsilon_{max\%} = 40\%$*), it also does not violate the requirement that* $\kappa_{max\%} \leq 30\%$ *because the errors are not adjacent.* □

The same distinctions, made for ε, presented in the itemized list after Definition 2.1.10, can be made for the metrics defined above.

Definition 2.1.19. ***The substring error*** ($\sigma\%$) $^{\oslash\ \odot}$
The substring error $\sigma\%$ *of a string is a percentage value, measuring the number of errors, weighted according to error type. If* p_m*,* p_d *and* p_i *are all equal to 1,* $\sigma\% = \varepsilon_{\%}$.

For a given motif, ρ*, and a given substring that has been partitioned into the form* $u = \rho u_2 \cdots u_p$*,* $\sigma\%$ *on* u_i *is computed as:*

$$\sigma\% = \frac{(n_d \times p_d) + (n_i \times p_i) + (n_m \times p_m)}{|\rho|} \times 100 \tag{2.1}$$

$$\tag{2.2}$$

where:

- n_d *is the number of deletions in* u*;*
- n_i *is the number of insertions in* u*;*
- n_m *is the number of mismatches in* u*;*
- p_m *is the penalty value on mismatches;*
- p_d *is the penalty value on deletions and;*
- p_i *is the penalty value on insertions.*

$\square$

The *maximum substring error allowed*, $\sigma_{max\%}$ is user determined $\sigma\%$. It serves as a threshold against which the substring error $\sigma\%$ is compared when `FireSat` detects TRs. `FireSat` will only consider a substring to be a TR if its substring error $\sigma\% \leq \sigma_{max\%}$.

Note that $\sigma\%$ and $\kappa\%$ can be computed for any two genetic strings, one designated as the source and the other as the destination — the definitions need not be limited to the TR context.

Within the context of TR detection, an alternative metric, indicating the differences between a PTRE (destination) and a TRE (source), is the Levenshtein Distance (LD). The definition of the LD is deferred until Section 2.5 where the Levenshtein Correspondence (LC) is also introduced.

2.1.3 TR classification

TRs are typically divided into microsatellites, minisatellites and satellites. The length of the repeating motif determines whether the TR is a microsatellite, minisatellite or satellite. The literature is not consistent in the classification of TRs according to repeating motif length (De Ridder [2010]).

The definitions adopted for the purposes of this thesis are as follows:

2.1.3.1 Microsatellites

Definition 2.1.20. ***Microsatellite*** $^\oplus$
A microsatellite is a TR whose motif length is restricted to the range $1 \leq |motif| \leq 6$. □

Microsatellites can be classified as follows:

Definition 2.1.21. ***Mononucleotide repeats*** $^\oplus$
A mononucleotide is a microsatellite consisting of a single nucleotide that is consecutively repeated. □

Example 2.1.22. $TR_a = \{\texttt{aa}, \texttt{aaa}, \texttt{aaaa}, \cdots\}$ *is the set of mononucleotide repeats with respect to* $\mathtt{a}$. □

Definition 2.1.23. ***Dinucleotide repeat*** $^\oplus$
A dinucleotide is a microsatellite where the repeated motif length is 2. □

Example 2.1.24. $TR_{ac} = \{\texttt{acac}, \texttt{acacac}, \texttt{acacacac} \cdots\}$ *is the set of dinucleotide repeats with respect to* `ac`. □

Definition 2.1.25. ***Trinucleotide, tetranucleotide, pentanucleotide and hexanucleotide repeats*** $^\oplus$
Trinucleotides, tetranucleotides, pentanucleotides and hexanucleotides are microsatellites whose repeated motif lengths are 3, 4, 5 and 6 respectively. □

2.1.3.2 Minisatellites and satellites

Definition 2.1.26. ***Minisatellite*** $^\oplus$
A minisatellite is a TR whose motif length is restricted to the range $7 \leq |motif| \leq 100$. □

Definition 2.1.27. ***Satellite*** $^\oplus$
A satellite is a TR whose motif length is restricted to the range $|motif| > 100$. □

2.1.4 TR-filters

TR-filters are concerned with the *nature* of the TRs to be output. In other words one can use these filters to specify how many of which type of TREs are required and/or allowed within a sequence of nucleotides for that sequence to be validly reported as a TR. Two TR-filters are defined below, namely the maximum number of consecutive ATREs and the minimum number of tandem repeat elements to be output. TR-filters differ from motif errors in the sense that guidelines pertaining to motif errors consider the nature of a single TRE whereas TR-filters consider the nature of a potentially detected TR that consists of several TREs.

In contrast to motif errors, TR-filters are not applicable when comparing two arbitrary genetic strings.

Definition 2.1.28. ***The Maximum consecutive* ATRE *filter* (α_{max})** $^{\oslash}$
The consecutive ATRE-filter, α_{max}, indicates the maximum number of ATREs that are allowed to occur next to each other. □

Thus, a string will only be classified as a TR if there are no more than α_{max} ATREs separating occurrences of PTREs. α keeps track of the number of consecutive ATREs. For a TR to be reported $\alpha \leq \alpha_{max}$.

Definition 2.1.29. ***The Minimum TR-length filter* (β_{min})** $^{\oplus}$
The TR-length filter indicates the minimum number of TREs that a TR should have if it is to be reported. Suppose β is the number of TREs in a TR. Then a TR will only be reported if $\beta \geq \beta_{min}$. □

In this section, definitions related to repeats were provided. Several of these definitions were proposed by De Ridder [2010]. The objective of introducing *new* definitions to the literature is to enable and extend precision when communicating about TRs in a way that was not previously possible.

2.2 Classical grammars

This section deals with classical grammars and the next with classical automata. The main sources used in these two sections are Cohen [1997], Sipser [2006] and Hopcroft and Ullman [1979]. Informal definitions are generally similar to definitions provided in Cohen [1997], whereas formal definitions are usually guided by Sipser [2006] or Hopcroft and Ullman [1979].

Readers who have a thorough background in mathematics and/or in theoretical computer science may skip both sections. They are given for completion to ensure that theoretical proposals given later fit in with classical theory.

Grammars generating languages belonging to different types of the Chomsky hierarchy are presented next to illustrate how the restrictions on productions influence languages and words generated. Although this thesis focusses more on machines (automata) than on grammars, Chapter 4 references the grammars defined here and illustrates *how* a *counting regular attribute grammar* (as defined by Sperberg-McQueen [2004]) can be converted into an automaton of a specific type, denoted by CA_{T3}.

Definition 2.2.1. ***Language***
A language is a set of words (or strings). Note that $\emptyset$ represents the empty language. $\square$

Definition 2.2.2. ***Kleene closure (Σ^*)***
The Kleene closure defines the language over an alphabet Σ such that any string of characters of Σ is a word in Σ^*. Note that the so-called empty string Λ is also considered to be in Σ^*. $\square$

Definition 2.2.3. ***Kleene plus closure (Σ^+)***
The Kleene plus closure is defined as $\Sigma^* \setminus \{\Lambda\}$. $\square$

Example 2.2.4.
If $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$ *then*
$\Sigma^* = \{\Lambda, \mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}, \mathtt{aa}, \mathtt{ac}, \mathtt{ag}, \mathtt{at} \cdots\}$ *and* $\Sigma^+ = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}, \mathtt{aa}, \mathtt{ac}, \mathtt{ag}, \mathtt{at} \cdots\}$.
If $\Sigma = \emptyset$ *then*
$\Sigma^* = \{\Lambda\}$ *and* $\Sigma^+ = \emptyset$. $\square$

Definition 2.2.5. ***Concatenation of languages***
If L_1 and L_2 are languages, then their concatenation, L_1L_2, is the language $\{w_1w_2 \mid w_1 \in L_1$ and $w_2 \in L_2\}$. $\square$

Example 2.2.6. *If* $L_1 = \{\mathtt{aa}, \mathtt{ac}\}$ *and* $L_2 = \{\mathtt{gc}\}$ *then* $L_3 = L_1L_2 = \{\mathtt{aagc}, \mathtt{acgc}\}$.
$\square$

Definition 2.2.7. ***Production***
A production has the form $\alpha \to \beta$ for strings α and β from $(V \cup \Sigma)^*$ where V and Σ are disjoint sets of so-called non-terminal and terminal characters respectively. $\square$

A production is also referred to as a *rewrite rule*. Thus a character substitution can recursively be performed to generate new character sequences. Production restrictions are determined by the type of grammar to be generated. This is illustrated in Sections 2.2.1, 2.2.2, 2.2.3 and Section 2.2.4. Note that $\to$ will exclusively be used in the statement of productions, whereas $\Rightarrow$ will exclusively be used to indicate stage changes in so-called derivations (see Example 2.2.20). The character "|" is used as follows to compact notation when two or more productions have the same right hand side:
$S \to ASC$ and $S \to \Lambda$ can be written as $S \to ASC|\Lambda$

Definition 2.2.8. ***Grammar***
A grammar is defined as a four tuple (V, Σ, R, S) where R is a set of production rules over the set of terminal characters Σ^* and the set of non-terminal characters, V (also referred to as variables). The special character $S \in V$ is designated as the start character of the grammar. $\square$

This text follows the convention of representing non-terminals as upper case characters.

Suppose w_1, α and w_2 are arbitrary strings in $(V \cup \Sigma)^*$. The operational meaning of a production rule $\alpha \to \beta$ in a grammar is that the substring α can be substituted by β in the string $w_1 \alpha w_2$ to produce the *derived* string $w_1 \beta w_2$.

Definition 2.2.9. ***Sentential form of a grammar*** $G = (V, \Sigma, R, S)$
A sentential form of the grammar $G = (V, \Sigma, R, S)$ *is its start character,* S*, as well as any other string in* $(V \cup \Sigma)^*$ *that is derivable from* S*.* $\square$

Example 2.2.10. *Consider the grammar with productions:*
$S \to ASC \mid \Lambda$
$A \to \mathtt{a}$
$C \to \mathtt{c}$

Then the following is one possible sequence of derivations:
$S \Rightarrow ASC \Rightarrow AASCC \Rightarrow \mathtt{a}ASCC \Rightarrow \mathtt{aa}SCC \Rightarrow \mathtt{aa}\Lambda CC \Rightarrow \mathtt{aa}\Lambda\mathtt{c}C \Rightarrow \mathtt{aa}\Lambda\mathtt{cc} \Rightarrow \mathtt{aacc}$.
Thus, each of the following is a sentential form:
$ASC,\ AASCC,\ \mathtt{a}ASCC,\ \mathtt{aa}SCC,\ \mathtt{aa}\Lambda CC,\ \mathtt{aa}\Lambda\mathtt{c}C,\ \mathtt{aa}\Lambda\mathtt{cc},\ \mathtt{aacc}$ $\square$

Definition 2.2.11. ***Language of the grammar*** $G = (V, \Sigma, R, S)$
The string $w \in \Sigma^*$ *is said to be in the language defined by* G *if and only if it can be derived by a sequence of substitutions that begins with* S *and ends with* w *(Kent et al. [1992]).* $\square$

Definition 2.2.12. $\boldsymbol{L'}$
If $L \subset \Sigma^*$ *then* L' *denotes all the words in* Σ^* *that do not belong to* L*.* $\square$

Definition 2.2.8 defines a grammar as a four tuple. Each grammar belongs to a specific level of the Chomsky hierarchy (discussed in Section 4.3). The level is determined by the kind of restrictions that apply to the grammar's productions. Each of the definitions below (2.2.13, 2.2.17, 2.2.18, 2.2.21 and 2.2.22) are in reference to a grammar (V, Σ, R, S). In each case, the production restrictions will be given, thereby highlighting the differences between the respective grammar types.

2.2.1 Regular grammars

Regular grammars (Definition 2.2.13) define regular languages. However, finite automata (Definition 2.3.1) and regular expressions (Definition 2.2.14) are alternative ways of defining regular languages.

Definition 2.2.13. ***Regular grammar***
A regular grammar has a finite set of rules, R, of the form $\alpha \rightarrow \beta$ where the productions are constrained in four ways:

1. $\alpha \in V$
2. $|\alpha| \leq |\beta|$
3. $\beta \in \Sigma^*$ *or* $\beta = x_i X$ *or* $\beta = X x_i$ *where* $X \in V$ *and* $x_i \in \Sigma^*$
4. *The single non-terminal on the right-hand side must appear either as the rightmost character in* every *production or as the leftmost character in* every *production rule.*

□

A regular grammar is called a right regular grammar (left regular grammar) if the single non-terminal on the right-hand side of productions is always the rightmost character (leftmost character, respectively) in every production[5].

Another way of defining a regular language is by using a regular expression (regex).

Definition 2.2.14. ***Regular expressions***

1. Λ *is a regex that defines the regular language* $\{\Lambda\}$.
2. *Suppose* $\mathtt{a} \in \Sigma$. *Then* $\mathtt{a}$ *is a regex that defines the regular language* $\{\mathtt{a}\}$.
3. *Let* r_1 *be a regex defining the regular language* L_1 *and let* r_2 *be a regex defining the regular language* L_2. *Then:*
   (a) r_1^* *is a regex defining* L_1^*.
   (b) $r_1 r_2$ *is a regex defining* $L_1 L_2$.
   (c) $r_1 + r_2$ *is a regex defining* $L_1 \cup L_2$.

□

Example 2.2.15. *If* r_1 *and* r_2 *are regexes representing the regular languages* $L_1 = \{\mathtt{ac}, \mathtt{gt}\}$ *and* $L_2 = \{\mathtt{aa}, \mathtt{gt}\}$, *then*
$r_1 + r_2$ *represents the language* $\{\mathtt{aa}, \mathtt{ac}, \mathtt{gt}\}$.
$r_1 r_2$ *represents the language* $\{\mathtt{acaa}, \mathtt{acgt}, \mathtt{gtaa}, \mathtt{gtgt}\}$. □

Definition 2.2.16. ***Regular language***
The language generated by a regular grammar or represented by a regular expression is a regular language. □

[5] A regular grammar is thus a specialised form of a linear grammar (Definition 2.2.18), which is, in turn, a specialised form of a context free grammar (Definition 2.2.17).

2.2.2 Context free grammars

Definition 2.2.17. ***Context free grammar (CFG)***
A CFG has a finite set of rules, R, of the form $\alpha \to \beta$, where
$\alpha \in V$*; and*
$\beta \in (V \cup \Sigma)^*$. □

A linear grammar is a context free grammar with an additional limitation on the right hand side of productions.

Definition 2.2.18. ***Linear grammar***
A linear grammar has a finite set of rules, R, of the form $\alpha \to \beta$ where:
$\alpha \in V$*; and*
either $\beta \in \Sigma^*$ or $\beta = x_i X y_i$ where $x_i, y_i \in \Sigma^*$ and $X \in V$. □

Definition 2.2.19. ***Context free language***
A language generated by a CFG is referred to as a context free language (CFL).
□

Example 2.2.20. *The CFL $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}} | \mathtt{n} \geq \mathtt{1}\}$ can be generated by the following CFG productions.*

$S \to ASC|\Lambda$
$A \to \mathtt{a}$
$C \to \mathtt{c}$

The derivation of the word aacc*, using these productions, is as follows:*
$\Rightarrow ASC$ *(using the production $S \to ASC$)*
$\Rightarrow AASCC$ *(using the production $S \to ASC$)*
$\Rightarrow \mathtt{a}ASCC$ *(using the production $A \to \mathtt{a}$)*
$\Rightarrow \mathtt{aa}SCC$ *(using the production $A \to \mathtt{a}$)*
$\Rightarrow \mathtt{aa}\Lambda CC$ *(using the production $S \to \Lambda$)*
$\Rightarrow \mathtt{aa}\Lambda\mathtt{c}C$ *(using the production $C \to \mathtt{c}$)*
$\Rightarrow \mathtt{aa}\Lambda\mathtt{cc}$ *(using the production $C \to \mathtt{c}$)*
$= \mathtt{aacc}$ □

2.2.3 Context sensitive grammars

Definition 2.2.21. ***Context sensitive grammar***
A CSG has a finite set of rules, R, of the form $\alpha A \beta \to \alpha \gamma \beta$, where

$A \in V$,
$\alpha, \beta \in (V \cup \Sigma)^*$, *and*
$\gamma \in (V \cup \Sigma)^+$.

□

Thus γ consists of concatenated terminals and non-terminals in any order but γ cannot be the empty word, Λ.

The name *context-sensitive* references the fact that α and β form the *context* of the non-terminal A and determine whether or not A can be replaced by γ. In the case of CFGs (Definition 2.2.17), the left hand side of productions consist only of one non-terminal without a context, which has to be taken into account.

2.2.4 Unrestricted grammars

Definition 2.2.22. ***Unrestricted grammar***
An unrestricted grammar has a finite set of rules, R, of the form $\alpha \rightarrow \beta$ where:
α and β are in $(V \cup \Sigma)^*$;
$\alpha \neq \Lambda$; and
α contains at least one element of V.
There are no other restrictions on productions. □

Unrestricted grammars characterise the so-called recursive enumerable languages.

2.3 Classical automata

The automata discussed in this section are well known in the literature. They are also referred to as theoretical machines or simply as machines. The simplest of these machines, deterministic finite automata (DFAs), are discussed in Section 2.3.1. Operations on deterministic DFAs relevant to Chapter 5 are given in Section 2.3.2. Section 2.3.3 covers definitions of other regular language acceptors. In Section 2.3.4, push down automata (PDAs) are defined and Turing machines are defined in Section 2.3.6. Note that definitions of these classical automata are taken from Cohen [1997]; Sipser [2006]; Hopcroft and Ullman [1979]; Sutner [2011b] and Sutner [2011a]. It is assumed that the reader is familiar with classical concepts from graph theory such as node, edge, path, cycle, tree, etc.

The concepts defined below will be referenced in Chapter 4 where CA_{T1}, CA_{T2} and CA_{T3} are defined and compared to the machines defined in this section.

2.3.1 Deterministic finite automata

Formally a DFA can be defined as follows:

Definition 2.3.1. ***Deterministic finite automata (DFA)***
A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, \mathcal{F})$ *where:*

- Q *is a non-empty finite set of states* $Q = \{q_0, q_1, q_2 ... q_n\}$.
- $q_0 \in Q$ *is designated to be the start state.*
- $\mathcal{F}$, *a subset of* Q, *is the set of final states.*
- $\Sigma = \{x_1, x_2, x_3 ... x_s\}$ *is a finite alphabet.*
- $\delta : Q \times \Sigma \to Q$ *is a possibly partial transition function.*

$\square$

Note that in the current context $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$.

Total and partial DFAs can be distinguished as follows:

Definition 2.3.2. ***Total Deterministic Finite Automata***
A total DFA is a DFA such that the transition function, δ*, is a total function. Thus, for every state in* Q *and character in* Σ *there is a transition to a state in* Q.

$\square$

Definition 2.3.3. ***Partial deterministic finite automata***
A partial DFA is a DFA such that the transition function, δ*, is a partial function. Thus, for a given state in* Q *there is not necessarily a transition on every character in* Σ. $\square$

Definition 2.3.4. ***DFA string acceptance***
Consider a DFA $(Q, \Sigma, \delta, q_0, \mathcal{F})$ *and a word* $w \in \Sigma^*$ *where* $w = a_0 a_1 \dots a_n$.

If there is a sequence of states, $q_{w0}, q_{w1}, q_{w2}, \dots q_{wn}$ *such that* $q_{w0} = q_0$, $q_{wn} \in \mathcal{F}$ *and* $\delta(q_{wi}, a_i) = q_{w(i+1)}$ *for* $i = 0 \dots n$ *then, and only then, the DFA is said to accept* w*. Otherwise the DFA is said to reject* w. $\square$

Definition 2.3.5. ***Language of a DFA***
The language of a DFA is the set of words that the DFA accepts. $\square$

Thus, if a DFA, either total or partial, is being used to test whether a string is in the DFA's language and it is found that $q_{wn} \notin F$ then the string is rejected as being in the DFA's language. If a partial DFA is being used and it is found that a transition cannot be made because $\delta(q_{wi}, a_i)$ is not defined, then the machine is said to crash. In this case, the string is also rejected as being in the DFA's language.

When important to distinguish between partial or total DFAs in this thesis, it will be done so explicitly.

It can be shown that the language of any DFA is a regular language. Conversely, as Kleene proved in 1956, any regular language (that is, a language defined by a regular expression) is also the language of some DFA (Cohen [1997]).

Definition 2.3.6. ***Transition graph of a DFA***
Automata are represented as directed graphs. The graph's nodes are DFA states, depicted as circles. The graph has labelled directed edges between states to represent the DFA transition function. The DFA start state is indicated by an in-edge that has no originating state. DFA final states are drawn as two concentric circles. The resulting graph is called a transition graph of the DFA. □

A word may be tested for membership of a DFA's language by checking whether there is a path in its transition graph from the start state to a final state. If there is, the word is said to be accepted by the DFA. Otherwise, it is rejected.

Example 2.3.7. *Figure 2.1 shows the transition graph of a total DFA. The DFA's alphabet is $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$. It accepts only one word, namely* `acg`*. Therefore, the language of the DFA is $\{\mathtt{acg}\}$. Note that the word Λ is not in the language, because $q_0 \notin F$. Also note that all words in Σ^+ other than* `acg` *end up in state $q_4 \notin F$, which is an example of a so-called sink state. Any partial DFA can be converted into a total DFA defining the same language by introducing a sink state as the destination state for all undefined transitions.* □

Definition 2.3.8. ***Cyclic DFA***
A cyclic DFA is a DFA whose transition graph has at least one cycle—i.e. there is at least one state p from which a sequence of edges loops back to state p. This includes the case where a single edge from p loops directly back into p [Watson [2010]]. □

Note that the transition graphs of all total DFAs are cyclic. (If this were not the case, then the DFA would have to have an infinite number of states, but that is counter to the definition of a DFA.) Typically, there will be at least one single-edged cycle at a sink state whenever no other cycles are present in a particular DFA.

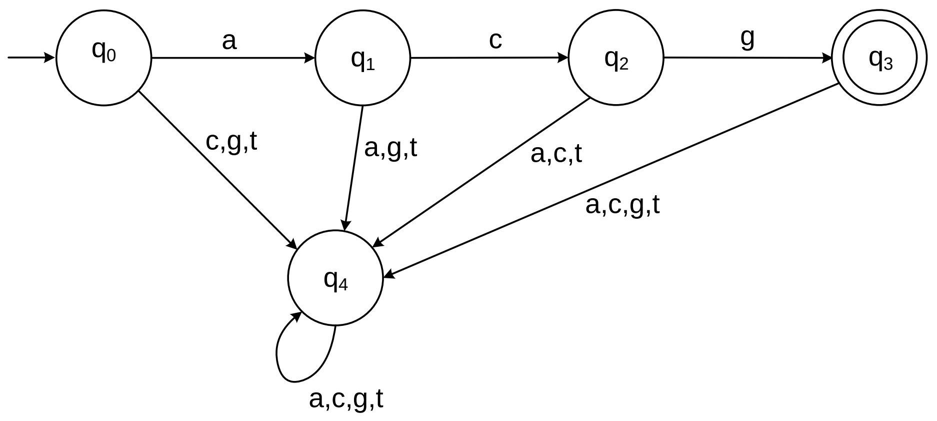


Figure 2.1: A total DFA whose language is {acg}.

Definition 2.3.9. ***Acyclic DFA (ADFA)***
An ADFA is a DFA without cycles. □

All ADFAs are therefore partial DFAs. An ADFA is thus made up of a collection of states and directed edges, each edge connecting one state to another in such a way that there is no path from some state, p, back to itself.

2.3.2 Operations on DFAs

Algorithms to determine the sum of DFAs and to convert a DFA into a right linear grammar and *vice-versa* are outlined below. These algorithms are of relevance in the remainder of this thesis—especially in Chapters 4, 5 and 7.

2.3.2.1 The sum operation on DFAs

The *Proof of Rule 2 of part 3 of Kleene's theorem* states that if there is a DFA called FA_1 that accepts the language defined by the regular expression r_1 and there is a DFA called FA_2 that accepts the language defined by the regular expression r_2, then there is a DFA that shall be called FA_3 that accepts the language defined by the regular expression $r_1 + r_2$. This statement can be proved by a constructive algorithm. While proving this statement a definition of how to determine the sum of two DFAs is given.

Definition 2.3.10. ***Sum(**FA_1, FA_2**)***
Starting with two machines, FA_1 consisting of states $q_0, q_1, q_2, \cdots$ and FA_2 consisting of states $x_0, x_1, x_2, \cdots$, a new machine FA_3 with states $z_0, z_1, z_2, \cdots$ is

built. Each state z_i in FA_3 can be seen as combining two states: one from FA_1, say q_j, and another from FA_2, say x_k. This will be expressed by saying that z_i is "q_j or x_k".

The start state of FA_3 is "q_{start} or x_{start}". If either the x part or the q part of the z state is a final state in one of the the original DFAs (FA_1 or FA_2), then the z state is a final state of FA_3.

When moving from one z state to another while reading a character from the input string, we investigate what happens to the q and x parts of z, and go to the corresponding new z part. If δ_1, δ_2 and δ_3 denote the transition functions of FA_1, FA_2 and FA_3, then it is possible to express the relationship as follows:

Suppose $z_i = q_j$ or x_k. Suppose too that $\delta_1(q_j, c) = q_{new}$, $\delta_2(x_k, c) = x_{new}$ and $\delta_3(z_i, c) = z_{new}$. Then $z_{new} = q_{new}$ or x_{new}. □

As there are only finitely many q's and finitely many x's, there can only be finitely many z's. Note that not every combination of state q and state x necessarily produces a reachable z state in FA_3 — some of the state combinations may thus be *useless* states.

To illustrate briefly the construction of $FA_3 = FA_1 + FA_2$ consider the following example:

Example 2.3.11. *Let FA_1 be the DFA whose language L_1 consists of a single word, i.e. $L_1 = \{\texttt{acg}\}$. FA_1 is given in Figure 2.2(a). Let FA_2 be the DFA whose language L_2 also consists of a single word, i.e. $L_2 = \{\texttt{ac}\}$. FA_2 is presented in Figure 2.2(b). Note that in terms of the definition of regular expressions (Definition 2.2.14),* `acg` *is a regex representing L_1 and* `ac` *is a regex representing L_2.*

FA_1 and FA_2 can be used to construct the transition table of FA_3 that accepts $L_3 = L_1 \cup L_2$, i.e. the regex `acg + ac`.

Thus, $FA_3 = (Q, \Sigma, \delta, z_0, F)$, where:
$Q = \{z_0, z_1, z_2, z_3, z_4\}$
$\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$
$F = \{z_3, z_4\}$ *and*
δ is provided by the transition table of FA_3.[6]

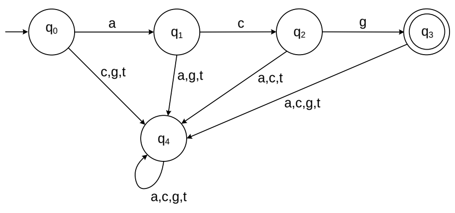
The sum machine, FA_3, is graphically depicted in Figure 2.2(c). □

Suppose that `ac` is input to FA_3. Then, from Table 2.1 it follows that $\delta(z_0, \texttt{a}) = z_1$ and $\delta(z_1, \texttt{c}) = z_3^+$. As a consequence, the conclusion is drawn that `ac` is in the language defined by $FA_1 + FA_2$.

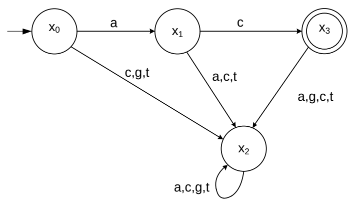
[6]The "+" character is used to indicate final states ($q_i \in F$) in transition tables representing δ.

States $\setminus \Sigma$	`a`	`c`	`g`	`t`
$z_0 = q_0$ or x_0	q_1 or $x_1 = z_1$	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$
$z_1 = q_1$ or x_1	q_4 or $x_2 = z_2$	q_2 or $x_3^+ = z_3^+$	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$
$z_2 = q_4$ or x_2	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$
$z_3^+ = q_2$ or x_3^+	q_4 or $x_2 = z_2$	q_3^+ or $x_2 = z_4^+$	q_4 or $y_4^+ = z_4^+$	q_4 or $x_2 = z_2$
$z_4^+ = q_3^+$ or x_2	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$	q_4 or $x_2 = z_2$

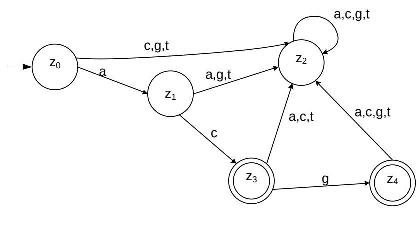
Table 2.1: Transition table for FA_3



(a) FA_1 — A DFA accepting `acg`.



(b) FA_2 — A DFA accepting `ac`.



(c) $FA_3 = FA_1 + FA_2$.

Figure 2.2: An illustration of DFA addition — $FA_3 = FA_1 + FA_2$

Using functional notation somewhat more loosely, the foregoing is expressed by the following sequence of equalities: $\delta(z_0, \texttt{ac}) = \delta(\delta(z_0, \texttt{a}), \texttt{c}) = \delta(z_1, \texttt{c}) = z_3^+$. Similarly $\delta(z_0, \texttt{acg}) = \delta(\delta(\delta(z_0, \texttt{a}), \texttt{c}), \texttt{g}) = \delta(\delta(z_1, \texttt{c}), \texttt{g}) = \delta(z_3^+, \texttt{g}) = z_4^+$. Thus `ac` and `acg` are accepted by the DFA presented in Figure 2.2(c).

Recall that a state of FA_3 is designated as final if it includes a final state of FA_1, or if it includes a final state of FA_2, or if it includes final states of both FA_1 and FA_2.

This rule is followed in Table 2.1. Thus, z_3 is a final state of FA_3, because it includes x_3^+ which is a final state of FA_2. On the other hand z_4^+ is a final state of FA_3, because it includes q_3^+ which is a final state of FA_1.

In general, a final state of FA_3 could include final states of both FA_1 and FA_2. However, there is no such final FA_3 state in the current example. Note that state z_2 is a sink state. For all the input characters of the alphabet $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$ FA_3 will remain in state z_2.

In general, it is always apparent which "old" final states are included in a "newly" constructed finite state — i.e. which "old" final state(s) cause the "new" final state to be final. In principle it is possible to make a partition between the respective final states of FA_3 based on the origin of the component final states of the original DFAs — FA_1 and FA_2 in this case. A set of final states derived from final states of both FA_1 and FA_2, if present, can be a third class of such a partition.

Exactly the same principle would hold, *pari passu*, if more than two DFAs were added together.

This observation was important for the development of FireμSat in De Ridder [2010], and is important for the development of FireSat$_2$ and FireSat$_{2'}$ presented in Chapter 7.

2.3.2.2 The product operation on DFAs

Definition 2.3.12. ***Product*** (FA_p, FA_q)

To construct FA_r which accepts $FA_p{\cdot}FA_q$ where FA_p consists of states $x_0, x_1, x_2, \cdots$ and FA_q consists of states $y_0, y_1, y_2, \cdots$, a new machine FA_r with states $z_0, z_1, z_2, \cdots$ is built.

FA_r is constructed by creating a z-state for every non-final state of FA_p. For every final state in FA_p a z-state should be established that expresses the option that processing is continuing on FA_p or that processing is beginning on FA_q. This implies more briefly that if a final state of FA_p is reached then processing is in $x_{something}$, which is a final state of FA_p or the processing on FA_p of the

input string is finished and processing has jumped to y_0 the start state of FA_q to commence processing on FA_q. Thus for every final state of FA_p there should be states $x_{something}$ or y_0 — the start state of FA_q. To proceed we should trace both machines for each character of Σ. Assume $z_j = x_i$ or y_0 where $x_i \in F$ of FA_p and y_0 is the start state of FA_q. Assume furthermore $a \in \Sigma$. If $\delta(x_i(a)) \to x_j$ and $\delta(y_0(a)) \to y_p$ then $z_2 = x_j$ or y_p. The components of the z-states of the product machines should all be traced for each character of Σ until no new z-states are generated. A state of the newly constructed FA_r (consisting of z-states) is a final state if and only if it consists of one or more final states of FA_q. There exist clearly only finitely many possibilities for the z-states of FA_r. Therefore FA_r is a finite machine. □

Cohen [1997] provides examples illustrating *how* to apply the product rule.

2.3.2.3 Right linear grammars derived from DFAs and *vice-versa*

Chapter 4 will illustrate *how* a *counting regular attribute grammar* (introduced by Sperberg-McQueen [2004]) can be derived from a CA_{T3} (a type of automaton proposed in this thesis) and *vice versa.* In Definition 2.3.14 the algorithm defining *how* to derive a DFA from a Right Linear Grammar is given. Definition 2.3.13 describes an algorithm that does the reverse: given a right linear grammar, it derives a language-equivalent DFA.

Definition 2.3.13. ***Derive a language-equivalent DFA from a right linear grammar***
Given a right linear grammar then a DFA can be derived as follows:
Step 1
Consider any right linear grammar of the form:

$$\begin{aligned} S &\to x_0 Z_0 \\ Y_1 &\to x_1 Z_1 \\ Y_2 &\to x_2 Z_2 \\ Y_3 &\to x_3 Z_3 \\ &\cdots \textit{ where } S, Y_j, Z_i \in V \textit{ and } x_i \in \Sigma. \end{aligned}$$

Step 2
For each Y_j, for each Z_i and for the start state, S, create a state with the corresponding name in q-format for the purposes of conformity see Figure 2.3.
Step 3
For every production $Y_j \to x_i Z_i$ draw a directed edge from q_{Yj} to q_{Zi} and label it with x_i. (If $Y_j = Z_i$, clearly the edge labelled x_j is a loop.)

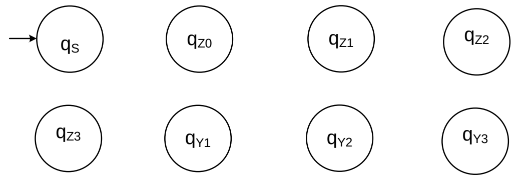


Figure 2.3: Step 2: Derivation of a DFA from a right regular grammar.

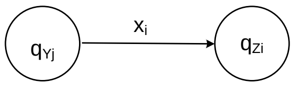


Figure 2.4: Step 3: Derivation of a DFA from a right regular grammar for any production $Y_j \rightarrow x_i Z_i$.

Step 4
For every production $Y \rightarrow x_j$ create a state $q_R \in F$ and draw an edge accordingly — see Figure 2.5.

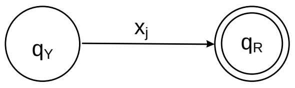


Figure 2.5: Step 4: Derivation of a DFA from a right regular grammar.

□

Definition 2.3.14. *Derive a language-equivalent right linear grammar from a DFA*
The DFA $(Q, \Sigma, \delta, q_0, F)$ can be used to derive a language-equivalent right linear grammar (V, Σ, R, S) as follows:

Step 1
Associate a non-terminal character with every state of the DFA.
Step 2
Let the grammar's start character (S) correspond with the DFA's start state.
Step 3
Let the grammar's set of terminal characters be the DFA's alphabet.
Step 4
For every transition $\delta(q, x) = p$, associate non-terminals, Y and Z, with states q and p respectively. Add the production $Y \rightarrow xZ$ to the grammar.

Step 5
For every state $r \in \mathcal{F}$, where R is the non-terminal associated with state r of the DFA, add the production $R \rightarrow \Lambda$.

□

2.3.3 Other regular language acceptors

As discussed above, regular grammars as well as DFAs are associated with regular languages in an exclusive fashion—i.e. the language of a regular grammar or DFA cannot be anything other than a regular language.

This subsection refers to various other machines that are also exclusively associated with regular languages. These are non-deterministic finite automata (NFAs), transition graphs for NFAs, Moore machines and Mealy machines. Definitions of these machines are provided next.

Note that both Moore and Mealy machines generate output and are referred to in the literature as transducers. As explained in Chapter 4, some of the formalisms proposed later in this thesis (specifically CA$_{T3}$s) have both Moore and Mealy machine characteristics.

Below, $\mathbb{P}(Q)$ denotes the power set of set Q—i.e. the collection of all subsets of Q. In addition, Σ_Λ denotes the set $\Sigma \cup \{\Lambda\}$.

Definition 2.3.15. ***(NFA) Non-deterministic finite automata***
An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, \mathcal{F})$ where:

- *Q is a non-empty finite set of states.*
- *$q_0 \in Q$ is designated to be the start state.*
- *A subset of Q represents the final states ($\mathcal{F}$).*
- *Σ is a finite alphabet.*
- *$\delta : Q \times \Sigma_\Lambda \rightarrow \mathbb{P}(Q)$ is a partial transition function.*

□

In the present context $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$.

Thus, an NFA is like a DFA except that it allows for a transition from a state to a *set of states* (i.e. more than 1 state) either on a given character or on the empty string. In the remainder of this thesis, reference to a finite automaton (FA) should be assumed to be to either a DFA or an NFA.

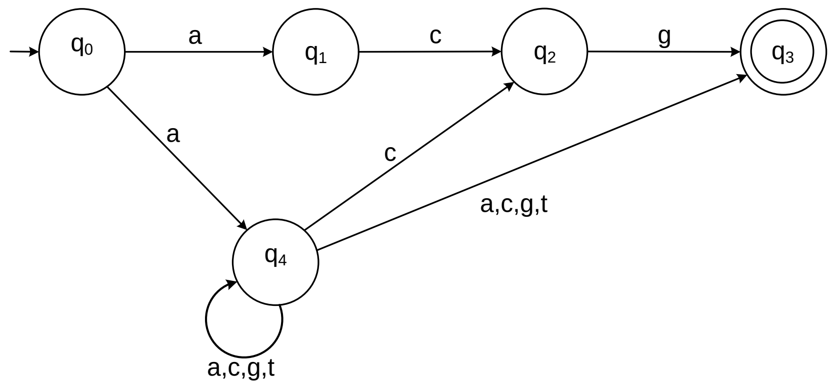


Figure 2.6: An NFA presenting three different ways to accept `acg`. Note that all words starting in `a` with a length of 2 or more are accepted by this NFA.

Figure 2.6 shows an NFA that accepts the word `acg` via three different paths. Note that there is no transition out of state q_3, the NFA's only final state. This means that for this NFA, δ is a partial function. In general, the transition function of an NFA may be either partial or total. Furthermore, it can be shown that a language-equivalent DFA can be derived from any NFA.

Definition 2.3.16. ***Moore machine***
A Moore machine is denoted by $Mo = \{Q, \Sigma, \Delta, \delta, \lambda, q_0\}$ *where:* $Q, \Sigma, \delta,$ *and* q_0 *have the same meaning as in the above definition of a DFA.* Δ *is the output alphabet.* $\lambda : Q \to \Delta$ *is a mapping from each state in* Q *to* Δ*. It represents the output associated with each state.* □

The output of Mo in response to input $a_1, a_2, \cdots, a_n, n \geq 0$, is given by
$\lambda(q_0), \lambda(q_1) \cdots \lambda(q_n)$, where $q_0, q_1, \cdots, q_n$ is the sequence of states such that
$\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$.

Notice that the Moore machine will give output of length $n + 1$ if n is the length of the input sequence. The input alphabet Σ need not be the same as the output alphabet Δ.

Definition 2.3.17. ***Mealy machine***
A Mealy machine is a six tuple $Me = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ *where* Q, Σ, δ *and* q_0 *are defined similarly to the formal definition of a DFA.* Δ *is the output alphabet and* $\lambda : Q \times \Sigma \to \Delta$ *is a mapping that gives the output associated with the transition from state* q *on input* a_i*.* □

The output of Me in response to input $a_1, a_2, \cdots, a_n$ is
$\lambda(q_0, a_1), \lambda(q_1, a_2), \cdots, \lambda(q_{n-1}, a_n)$, where $q_0, q_1, \cdots, q_n$ are states such that

$\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$. If the input sequence is of length n, then the output sequence is also of length n. The input alphabet Σ need not be the same as the output alphabet Δ.

Note that most sources do not explicitly state whether or not the transition function in Mealy and Moore machines, δ, should be regarded as total. Generally it appears to be that it is regarded as a total function. Note further that Moore and Mealy machines do not define a language of accepted words as a DFA does, since the notion of a final state is not defined. Instead, they map input strings to output strings.

2.3.4 Context free language acceptors

DFAs and NFAs were defined as acceptors of regular languages only. Context free languages (CFLs) cannot be accepted by DFAs. Non-deterministic Push Down Automata usually referred to simply as *Push Down Automata*[7] are classical machines accepting all CFLs. Deterministic Push Down Automata (DPDAs) accept a subset of the CFLs namely the *Deterministic Context Free Languages* (DCFLs). A formal definition of a DPDA is provided after the PDA definition. PDAs and conventions indicating *how* they are graphically displayed are informally described below.

- A finite alphabet Σ of possible input characters is assumed.
- An input tape is read. The tape is infinite in one direction. The string of input characters are initially on the tape. The first input character is in cell 1. The input is followed by blank cells on the tape. The character used to indicate a blank cell is Δ.
- An alphabet Γ of stack characters is assumed. Apart from Δ the characters of Γ do not have to correspond to those of Σ. The popping of Δ indicates an empty stack.
- A pushdown stack (infinite in one direction) is used. The stack is initially empty, containing Δs[8].
- There is exactly one *start state* that has only out-edges and no in-edges. (See Figure 2.7.)

[7]Throughout this thesis *Push Down Automata* (PDAs) refer to non-deterministic PDAs. The term *Deterministic Push Down Automaton* (DPDA) is used to distinguish a deterministic PDA.

[8]Instead of a pushdown stack some authors (including Salomaa [1985]) refer to pushdown tape. However, Cohen [1997]'s convention of referring to stack will be followed here, since *stack* is more descriptive and emphasizes the *last in-first out* functioning of the stack.

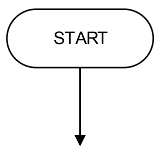


Figure 2.7: PDA *start state.*

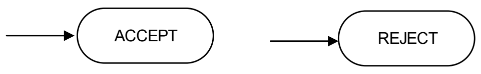


Figure 2.8: PDA *halt states.*

- There are two types of *halt states*, namely *accept* and *reject.* Halt states have in-edges and no out-edges (Figure 2.8).
- There are finitely many non-branching *push* states. *Push states* introduce characters onto the top of the stack. (See Figure 2.9.) Here x is any character in Γ, the stack alphabet.

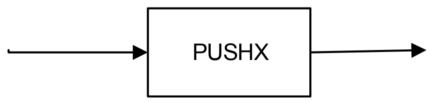


Figure 2.9: PDA *push state.*

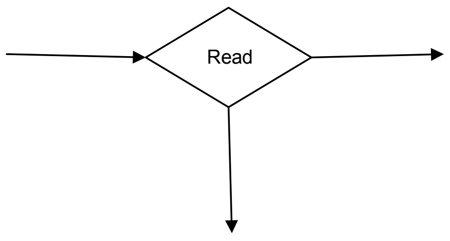


Figure 2.10: PDA *read state.*

- There are finitely many branching states of two types:
  1. *Read states*
     Read states read the next unused character from the tape. *Read states* may have out-edges labelled with characters from Σ_Δ[9]. There is no restrictions on duplication of labels. A label for each character of Σ or Δ is not required. See Figure 2.10.
  2. *Pop states*
     Pop states may have out-edges labelled with characters from Γ as well as Δ with no restrictions. *Pop states* remove characters from the top of stacks — Figure 2.11.

In the formal definition of a PDA, as before, $\Sigma_\Delta = \Sigma \cup \{\Delta\}$ and $\mathbb{P}(Q)$ refers to the power set of Q. Additionally, $\Gamma_\Delta = \Gamma \cup \{\Delta\}$. Δ is popped from the stack whenever the stack is empty.

Definition 2.3.18. ***Push down automata (PDAs)***
A pushdown automata is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \digamma)$ *where* Q, Σ, Γ *and* $\digamma$ *are all finite sets, and*

- *A non-empty finite set of states* $Q = \{q_0, q_1, q_2 \cdots q_n\}$.

[9] $\Sigma \cup \{\Delta\}$. Within this context Δ is the character indicating an empty tape or an empty stack.

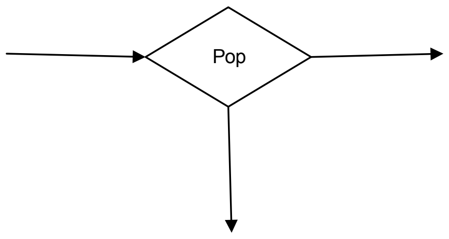


Figure 2.11: PDA *pop state*.

- $\Sigma = \{x_1, x_2 \cdots x_s\}$. *Is a finite set referred to as the input alphabet.*
- *A stack alphabet* $\Gamma = \{y_1, y_2, y_3 ... y_s\}$. *Note that* Σ *and* Γ *need not be disjoint.*
- $q_0 \in Q$ *is designated to be the start state.* q_0 *will be labelled* START *and represented as described above.*
- $F \subseteq Q$ *called the final states. The final accepting states will be labelled* ACCEPT.
- *A transition function* $\delta : Q \times \Sigma_\Delta \times \Gamma_\Delta \longrightarrow \mathbb{P}(Q \times \Gamma_\Delta)$. *The transition function is partial (Sipser [2006] and Cohen [1997]).*

□

The transition function δ can alternatively be presented as a tuple $(q_i, x_i, y_i, q_j, y_j)$. Where:

- $q_i \in Q$ is the current state.
- $x_i \in \Sigma_\Delta$ is the input triggering a transition.
- $y_i \in \Gamma_\Delta$ is currently at the top of the stack. y_i can be popped from the stack triggering the next transition.
- $q_j \in Q$ is the next state reached after branching has been triggered by either $x_i \in \Sigma_\Delta$ or $y_i \in \Gamma_\Delta$.
- $y_j \in \Gamma$ is pushed onto the stack if such an action is triggered by either $\delta(q_i, x_i)$ or $\delta(q_i, y_i)$.

Whereas a non-deterministic PDA allows different path options from which to choose, a DPDA is a PDA for which every input string has an unique path through the machine.

Definition 2.3.19. ***Deterministic push down automata (DPDA)***
A DPDA is defined as a PDA that also satisfies the following conditions:

- *For any $q \in Q, u_i \in \Sigma_\Delta, r_i \in \Gamma_\Delta$ the set $\delta(q, u_i, r_i)$ has at most one element.*
- *For any $q \in Q, r_i \in \Gamma$ if $\delta(q, \Delta, r_i) \neq \emptyset$ then $\delta(q, u_i, r_i) = \emptyset \ \forall u_i \in \Sigma$.*

□

Notation 2.3.20. ***Drawing PDAs***
PDAs are drawn by following the conventions described in the informal definition of a PDA. Read and pop states are labelled as such and depicted as diamond shapes. Start states and accept states are depicted as ellipses and labelled START and ACCEPT respectively. Push-states are rectangular, labelled with the word PUSH together with the character being pushed on the stack. Transitions from both read and pop states are depicted as labelled directed edges. A string or word from the alphabet that matches a path from the start state to a final state is said to be accepted by the PDA. □

If $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$ and $\Gamma = \{x\}$ then it is possible to draw a deterministic PDA that accepts the language $L = \{a^n c^n | n \geq 0\}$ as illustrated in Figure 2.12.

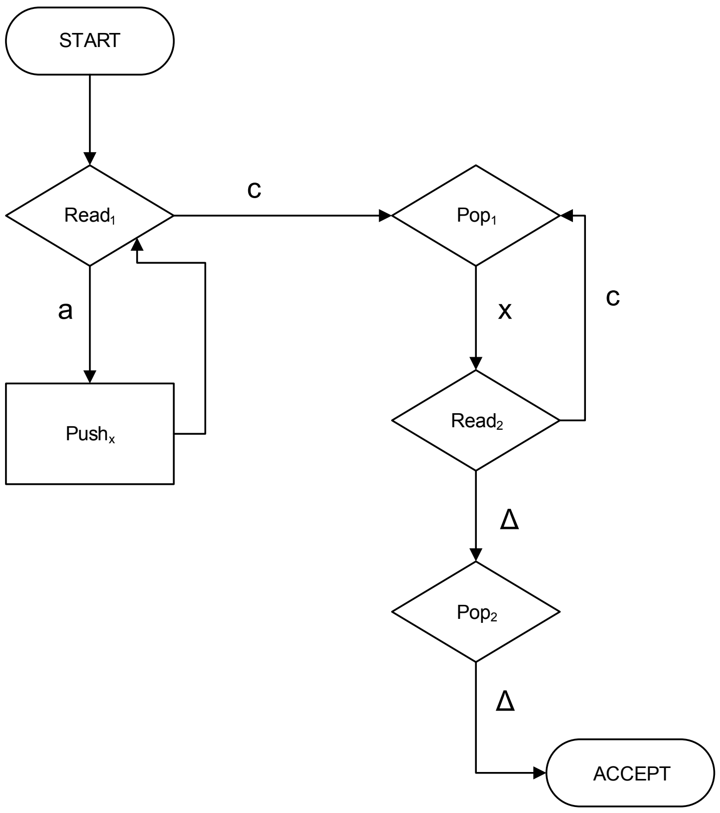


Figure 2.12: A DPDA accepting $L = \{\mathtt{a}^\mathtt{n}\mathtt{c}^\mathtt{n} \mid \mathtt{n} \geq \mathtt{1}\}$.

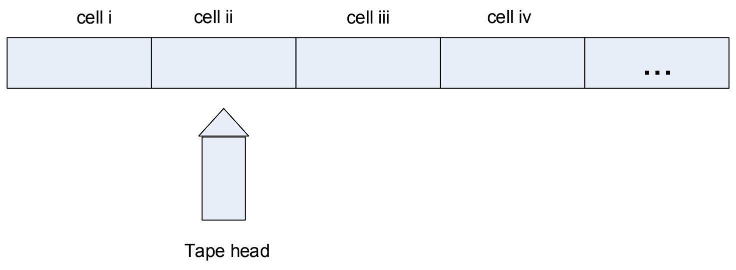


Figure 2.13: A TM tape and tape head.

2.3.5 Context sensitive language acceptors

Not being too formal, Cohen [1997] describes a Linear Bounded Automata (LBA) as a collection of six entities:

1. An alphabet Σ of input characters. The blank character Δ is not included in Σ.

2. A so-called tape that is divided into a sequence of numbered cells. Each of these cells contains one character or a blank. The input word is presented to the machine, one character per cell. The input begins at the leftmost cell, called cell i. The maximum number of cells on the input tape, if n is the length of the input string, is kn where k is a constant associated with the particular linear-bounded automaton. Cells that do not contain input, contain Δs (the character used to denote blanks). Note that LBAs are sometimes defined to use only cells with input. This implies $k = 1$. Equivalence classes of machines are implied — it is namely possible to compensate for the shorter tape by having a larger tape alphabet (Matuszek [1996]).

3. A tape head that can, in one step: read the content of a cell on the tape; replace that content with another character; and position the tape head to the next cell, either to the right or to the left of the cell it has just read. At the start of processing, the tape head begins by reading the input in cell i. The tape head cannot move left from cell i. If a LBA is given orders to move the tape head to the left of cell i, the machine will crash. A representation of the tape together with the tape head can be found in Figure 2.13.

4. An alphabet Γ is assumed that consists of characters that can be printed on the tape by the tape head. This can include characters from Σ. $\Delta \in \Gamma$. Note that Δ is referred to as the blank character — the only character that

is allowed to appear infinitely on the tape. There is referred to the printing of Δ as *erasing*.

5. Also assumed, is a final set of states and exactly one start state from which execution is begun. The start state may be re-entered during execution. One or more halt states may be included. The halt state causes execution to terminate upon entrance. The remainder of states do not have any specific function, only names such as q_1, q_2, q_3... or simply 123....

6. A program, is a set of rules that indicate how to change states, what to print on the tape, and where to move the tape head, depending on the current state and character just read. The program is depicted as a sequence of directed edges connecting states. Each edge is labelled with a triplet of information:

   $$(character, character, direction)$$

   The first character (either Δ or from Γ or Σ) is the character the tape head reads from the cell to which it is currently pointing. The second character is what the tape head prints in the cell before it is leaving. The third character represents the direction which the tape head should move — either to the *left* (L) or to the *right* (R).

It is possible to introduce a reject state. The reject state will be employed in a similar manner to a sink state of a DFA. However, instead of iterating until all the input on the tape has been read, if the reject state is entered then processing will stop and the string will be rejected. The formal definition, hereafter, makes provision for reject states. To terminate execution successfully the halt state should be entered. The word on the input tape is then said to be accepted by the LBA. A crash is said to occur if we are in the first cell and we try to move the tape head to the left. In the case of deterministic LBAs there does not exist a state that has two or more edges leaving it, which are labelled with the same first character.

Definition 2.3.21. ***Linear bounded automata***

A LBA is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, \Delta, q_0, F$ where Q, Σ and Γ are finite sets and:

1. *Q is the finite, non-empty set of states.*

2. *Σ is the input alphabet which does not contain the blank character Δ.*

3. *Γ is the tape alphabet where $\Delta \in \Gamma$ and $\Sigma \subset \Gamma$.*

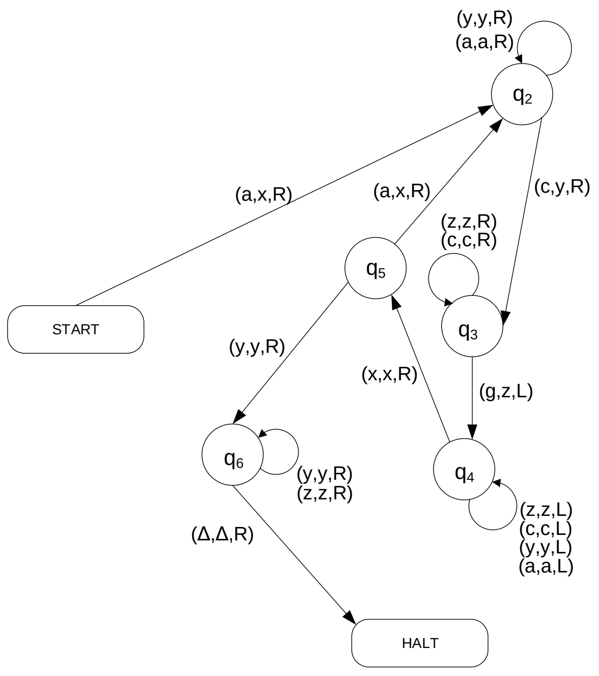


Figure 2.14: A TM accepting $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}}\mathtt{g}^{\mathtt{n}} \mid \mathtt{n} \geq \mathtt{1}\}$.

4. *$\delta : (Q \backslash F) \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is a partial transition function. L and R indicates the direction in which the tape head should move. L indicates move to the left and R move to the right. An undefined δ on the current state and character causes the machine to halt and reject, in other words, to crash.*

5. *$q_0 \in Q$ is the start state.*

6. *$F \subseteq Q$ is the set of accepting states or final states.*

□

A graphical representation of an LBA that accepts the context sensitive language $L = \{a^n c^n g^n | n \geqslant 1\}$ is given in Figure 2.14.

2.3.6 Recursively enumerable language acceptors

Recall that languages generated by unrestricted grammars are called recursively enumerable languages. Turing machines (TMs) are acceptors of recursively enumerable languages. They can be either deterministic or non-deterministic. The TMs considered in this thesis are single taped and deterministic[10].

Definition 2.3.22. ***Turing machine (TM)***
A TM is an LBA with an unbounded tape. □

nPDAs ($n \geq 2$) are equivalent in computing power to TMs. Consequently nPDAs ($n \geq 2$) are acceptors of recursive enumerable languages too.

Definition 2.3.23. ***Two-stack PDA (2PDA)***
A 2PDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \mathcal{F})$ *where* Q, Σ, Γ, q_0 *and* $\mathcal{F}$ *are the same as for an ordinary PDA. However,* δ *is defined as follows:*
$\delta : Q \times \Sigma_\Delta \times (\Gamma_\Delta) \times (\Gamma_\Delta) \longrightarrow \mathbb{P}(Q) \times \Gamma_\Delta \times \Gamma_\Delta$

□

It is not the intent to discuss all the theoretical machines and the types of languages they accept exhaustively. Recursive enumerable languages are accepted by other theoretical machines including Post machines[11] The objective of this chapter is to supply sufficient definitions to support the remainder of this thesis, specifically in Chapters 4, 6 and 7.

2.4 Automata for genetic string processing

In this section, background material is presented that is based on the classical material of the previous section, but has been adapted, defined and/or developed as part of this thesis. The concern is to define machines that will be used specifically in the context of genetic string recognition, together with the associated notation and operations. Prototype Finite Automata (pDFAs) and Counting Automata Type 3 (CA_{T3}) are new concepts introduced for the first time in this thesis. pDFAs are defined in this section. CA_{T3}s are defined in Chapter 4 where their functionality is illustrated as well. pDFAs, combined with the *cascade operation* (defined in this section) will be further explored in Chapter 6. In Chapter 7 pCA_{T3}s are cascaded to detect TRs.

In addition to distinguishing between an FA's start state (q_0) and final states $\mathcal{F}$, it is helpful to distinguish other state types in FAs that are used for TR recognition. The alphabet for such FAs should, of course, assumed to be $\{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$.

The definitions and notation below characterise to states that are reached when a motif error occurs.

[10]It can be shown that any non-deterministic TM can be simulated by a deterministic TM (Savitch [1970]).

[11]Emil Leon Post created the Post machine in 1936. The interested reader may consult Cohen [1997] for a definition and examples of Post machines.

Definition 2.4.1. ***State-types***
Suppose FA_L is defined over the alphabet Σ to accept L. FA_L is being used to check whether $u.\alpha.v \in L$ where $u, v \in \Sigma^*$ and $\alpha \in \Sigma$. Suppose further that the string u has been matched along a path so that state p has now been reached and suppose that $\delta(p, \alpha) = q$. Then the following terminology will be used to characterise state q with respect to string $u.\alpha.v$:

- *q is a* mismatch state *(denoted q_m) iff α is regarded as a mismatch in string $u.\alpha.v$.*
- *q is a* deletion state *(denoted q_d) iff string $u.\alpha.v$ is regarded as having had a character deleted between u and v.*
- *q is an* insertion state *(denoted q_i) iff α is regarded as an insertion in string $u.\alpha.v$.*
- *q is a* neutral state *(denoted q_n) iff it is neither a deletion, insertion nor mismatch state in respect of string $u.\alpha.v$.*

□

Notice that an ADFA has a finite set of paths from its start state to final states and each such path is associated with a string in the ADFA's language. Since there are no cycles in an ADFA, the length of each of these paths is finite. Thus, the language of an ADFA will always be a finite set of strings of finite length and conversely, given such a set of strings, an ADFA can be constructed to represent the set.

Because there are no cycles in an ADFA, the transition function, δ is necessarily a partial function. There will always be one or more states, q in an ADFA that may be regarded as terminal in the sense that the state has no outgoing transitions i.e. for all $a_i \in \Sigma$, $\delta(q, a_i)$ is not defined. $\perp$ is used to indicate $\delta(q, a_i)$ is undefined. $\perp$ is consequently used interchangeably — to indicate a sink state in total DFAs and an undefined δ in ADFAs. The meaning of $\perp$ will be clear from the context. In most practical situations, such terminal states are also final states of the ADFA. It is possible that an ADFA also has additional final states (Watson [2010]).

A prototype DFA is defined below in terms of an ADFA. For reasons that will become clear below, its terminating states as described above will be called *cascading* states.

The term *prototype* was chosen to suggest a *first* or *originating* DFA from which other DFAs are constructed. Thus *prototype* is used to emphasise that, in this thesis, these pDFAs often form only *part* of the full DFAs needed for TR recognition. pDFAs will be combined together (via the sum and cascade operations) to form TRE acceptors.

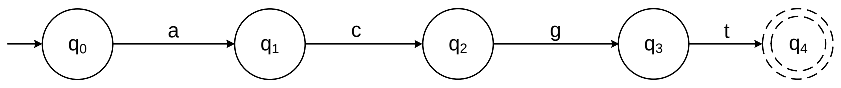


Figure 2.15: $pDFA_P(v, 0)$ where $v = \texttt{acgt}$.

Summing two pDFAs involves the straightforward application of the sum operation as defined above in Definition 2.3.10. However, finding the product of two pDFAs is much simpler than the general case of finding the product of two arbitrary DFAs. The product of two languages generated by regexes r_1 and r_2 is defined by using DFAs as presented in Definition 2.3.12. In the pDFA context the product operation is simpler and referred to as cascading. The cascading operation will be defined in Definition 2.4.4.

Let $v \subseteq \Sigma^*$; $|v| \leq 4$. Let the language of v, $L(v) = \{v\} \cup \{v_D\} \cup \{v_M\} \cup \{v_I\}$. Thus $L(v)$ is the set of strings resulting from one or more deletions, mismatches or insertions (but maximally $|v|$ insertions).

Definition 2.4.2. ***Prototype deterministic finite automata (pDFA)***
A pDFA is an ADFA $(Q, \Sigma, \delta, F, q_0)$ accepting $L \subseteq L(v)$. The final states of pDFAs are called cascading states and are indicated with dotted lines in transition graphs. □

It will be convenient to use the notation below to refer to certain specific pDFAs.

- $pDFA_P(v, 0)$ is a pDFA that reaches a cascading state after scanning v in its input string. Thus, $pDFA_P(v, 0)$ "accepts" exactly one PTRE of length $|v|$.
- $pDFA_D(v, e)$ is a pDFA that reaches one or more cascading states if and only if a substring has been read that is v less e deletions.
- $pDFA_M(v, e)$ reaches one or more cascading states if a substring has been read that is v, but with e mismatches.
- $pDFA_I(v, e)$ reaches one or more cascading states if a substring v, has been read, but with e insertions.

These definitions are illustrated by referring to their corresponding transition graphs. Assume that $\rho = \texttt{acgt}$ and $e = 1$.

Figure 2.15 represents $pDFA_P(v, 0)$.

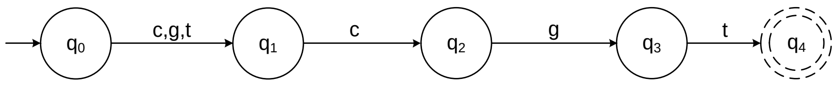


Figure 2.16: A pDFA allowing words where 1 mismatch occurs in the first position of $\upsilon =$ `acgt`. Note that words accepted are defined by the regex $(\texttt{c} + \texttt{g} + \texttt{t})\texttt{cgt}$.

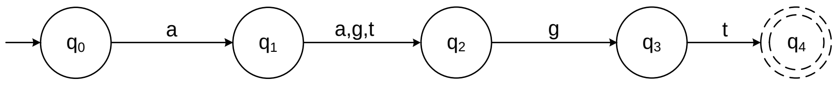


Figure 2.17: A pDFA allowing words where 1 mismatch occurs in the second position of $\upsilon =$ `acgt`. Note that words accepted are defined by the regex $\texttt{a}(\texttt{a} + \texttt{g} + \texttt{t})\texttt{gt}$.

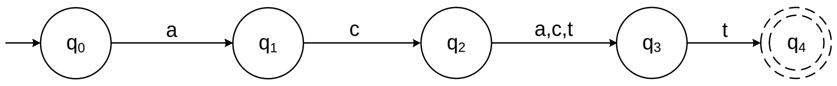


Figure 2.18: A pDFA allowing words where 1 mismatch occurs in the third position of $\upsilon =$ `acgt`. Note that words accepted are defined by the regex $\texttt{ac}(\texttt{a} + \texttt{c} + \texttt{t})\texttt{t}$.

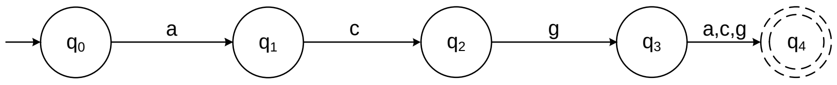


Figure 2.19: A pDFA allowing words where 1 mismatch occurs in the fourth position of $\upsilon =$ `acgt`. Note that words accepted are defined by the regex $\texttt{acg}(\texttt{a} + \texttt{c} + \texttt{g})$.

The next four pDFAs (in Figures 2.16, 2.17, 2.18 and 2.19) recognise v with a single mismatch, each in a different position. In principle, they could be added together to obtain $pDFA_M(v, e)$ where $e = 1$.

By adding the pDFA in Figure 2.15 to $pDFA_M(v, e)$ (presented in Figures 2.16, 2.17, 2.18 and 2.19), a pDFA is obtained that accepts v as well as all variants of v with $e = 1$ mismatch.[12] Such a pDFA is shown in Figure 2.20. Note that it has the characteristic that if the cascading state q_4 has been reached then $e = 0$; whereas if the cascading state q_{11} has been reached then $e = 1$ since 1 mismatch error has occurred within v.

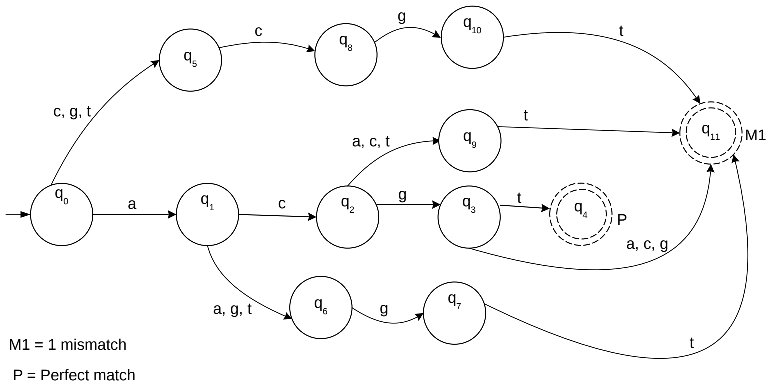


Figure 2.20: A pDFA accepting words where 1 mismatch occurs in any position of $v =$ `acgt` as well as $v =$ `acgt` itself.

For the purposes of detecting TRs in a memory- and runtime-effective way, the cascade operation is introduced below. The cascade operation is used in $\texttt{FireSat}_1$, $\texttt{FireSat}_2$ and $\texttt{FireSat}_{2'}$, discussed in Chapters 6 and 7 respectively.

A pDFA can be seen as a degenerate form of a special NFA that will be called a cascaded NFA. A cascaded NFA results from applying the so-called cascading operation on a cascaded NFA and a pDFA.

Definition 2.4.3. ***Cascaded NFA, N***

N is a five tuple $(Q_N, \Sigma, \delta_N, C_N, S_N)$ where

- *Q_N is a non-empty finite set of states $Q_N = \{q_0, q_1, q_2 ... q_n\}$.*
- *$S_N \in Q_N$ is designated to be the start state.*

[12]Note that the four different mismatch cascading states (Figure 2.20) were merged into one. The machine resultant from applying the algorithm inside Kleene's theory (Definition 2.3.10) strictly spoken, has four different *mismatch* cascading states.

- *C_N, a subset of Q_N, is the set of cascading states.*
- *$\Sigma = \{a_1, a_2, a_3 ... a_s\}$ is a finite alphabet.*
- *$\delta_N : Q_N \times \Sigma_\Lambda \rightarrow \mathbb{P}(Q_N)$ is a possibly partial transition function.*

$\square$

Assume we have N and P where N is the cascaded NFA and P is the pDFA:

$N(Q_N, \Sigma, \delta_N, C_N, S_N)$
$P(Q_P, \Sigma, \delta_P, C_P, S_P)$

Definition 2.4.4. ***Cascade*** ***($C(N, P)$)***
Let N be a cascaded NFA with C_N its cascading states. Let P be a pDFA with S_P its start state and C_P its cascading states.

Then $C(N, P)$ is the cascaded NFA $(Q_{\mathfrak{C}}, \Sigma, \delta_{\mathfrak{C}}, C_{\mathfrak{C}}, S_{\mathfrak{C}})$ where:

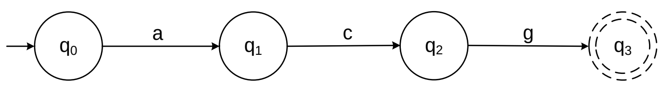
- *$Q_{\mathfrak{C}} = (Q_N - C_N) \cup (C_N \times Q_P)$ (the states of the newly cascaded machine, $\mathfrak{C}$);*
- *$C_{\mathfrak{C}} = C_N \times C_P$ (the new cascading states of $\mathfrak{C}$ are the cascading states of P) and;*
- *$S_{\mathfrak{C}} = S_N$*
- *With $\delta_{\mathfrak{C}}(q, a)$ defined for the different possibilities as follows:*

$$\begin{cases} \delta_N(q, a) \text{ if } q \in (Q_N - C_N) \wedge (\delta_N(q, a) \notin C_N) \\ \langle c_n, S_p \rangle \text{ if } (q \in (Q_N - C_N)) \wedge (\delta_N(q, a) = c_n) \wedge (c_n \in C_N) \\ \langle c_n, r_p \rangle \text{ if } (q = \langle c_n, q_P \rangle) \wedge (q \in (C_N \times Q_P) \wedge (\delta_p(q_p, a) = r_p) \end{cases}$$

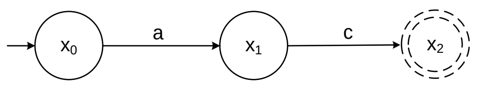
$\square$

The operation *cascade* as defined here should be distinguished from a *cascade of finite state transducers* often implemented within the context of natural language processing. Moore and Mealy machines are examples of *finite state transducers* (FSTs). In brief FSTs, refer to finite automata with output. When a pair of FSTs operate in a cascade it implies that:

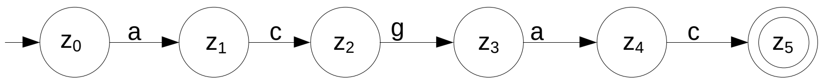
- the first FST maps an input string to a number of intermediate strings and;



(a) *pDFA*$_1$ — A pDFA accepting `acg`.



(b) *pDFA*$_2$ — A pDFA accepting `ac`.



(c) $FA_3 = pDFA_1.pDFA_2$.

Figure 2.21: An illustration of two pDFAs, *pDFA*$_1$ and *pDFA*$_2$, cascaded — $FA_3 = cascade(pDFA_1, pDFA_2)$

- the second FST maps the intermediate strings to a number of output strings (Kempe [2000]).

A detailed, formal discussion of a *cascade of FSTs* is beyond the scope of this thesis.

The *cascade* operation is defined for PFAs only. It is illustrated in Figure 2.21 where *pDFA*$_1$ (accepting `acg`) is cascaded to *pDFA*$_2$ (accepting `ac`) to construct FA_3 accepting `acgac` only.

The notations in 2.4.5, 2.4.6, 2.4.7 and 2.4.8 below are based on the assumption that the machine in question is to investigate whether u having the form $[\rho, u_2, u_3 \ldots u_n]$ is a substring of a genetic input string (gSeq). The notation was proposed in De Ridder [2010]. In that context, three algorithms to detect microsatellites were proposed. They are collectively known as `Fire`$_\mu$`Sat`. Here the same notation will be relevant in Chapters 6 and 7. The differences between `Fire`$_\mu$`Sat` and `FireSat` will be briefly outlined in that chapter.

Notation 2.4.5. $FA_P(\rho, 0)$ *is a DFA that reaches a final state after scanning a genetic input string (gSeq) and reaches an occurrence of* ρ *in gSeq. It reaches the final state again if* $u_2 = \rho$ *is in* u *a substring of gSeq, and again if* $u_3 = \rho$ *is encountered in* u*, etc. However,* $FA_P(\rho, 0)$ *goes to a sink state as soon as a character in* u *is encountered that indicates that* u *is not a PTR. Thus,* $FA_P(\rho, 0)$ *accepts a PTR with motif* ρ *of arbitrary length, entering the final states as many times as there are PTREs in the PTR.* □

Notation 2.4.6. *$FA_D(\rho, \varepsilon_{max})$ is a DFA that, upon scanning u, a substring of gSeq, reaches its first final state once the substring ρ has been read. $FA_D(\rho, \varepsilon_{max})$ continues to reach final states after scanning each word, u_i (where $i = 2 \cdots p$) provided that one of the following conditions hold: a) either $u_i = \rho$ or b) u_i is a word deduced from ρ that contains a maximum of ε_{max} deletions.* □

Notation 2.4.7. *$FA_M(\rho, \varepsilon_{max})$ is a DFA that functions analogously to $FA_D(\rho, \varepsilon_{max})$, except that it functions in terms of* mismatches *rather than deletions.* □

An example of $FA_M(\texttt{acg}, \texttt{1})$ is depicted in Figure 2.22. Initially, $FA_M(\texttt{acg}, \texttt{1})$ loops in states q_0, q_1 and q_2. Until `acg` is read. Thereafter, it contains cycles which will be completed as long as the PTRE `acg` or an ATRE that has been derived from `acg` as a mismatch in one nucleotide position, is being read. Any other input will cause $FA_M(\texttt{acg}, \texttt{1})$ to proceed to its sink state, namely q_7.

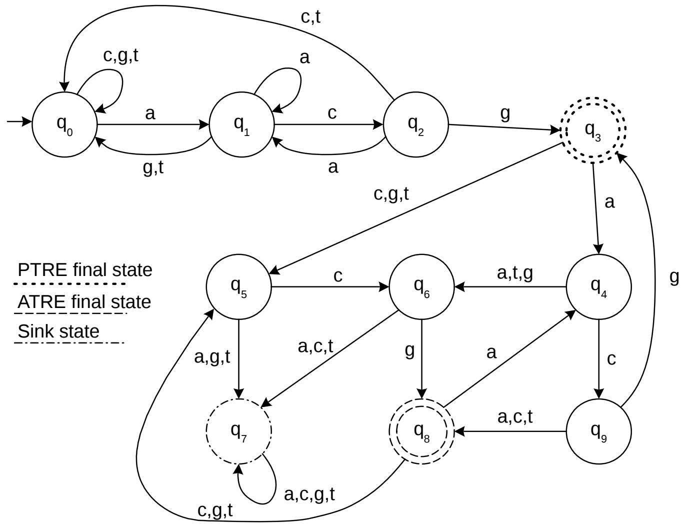


Figure 2.22: $FA_M(\texttt{acg}, \texttt{1})$.

Notation 2.4.8. *$FA_I(\rho, \varepsilon_{max})$ is a DFA that functions analogously to $FA_D(\rho, \varepsilon_{max})$ and $FA_M(\rho, \varepsilon_{max})$, except that it functions in terms of* insertions *rather than deletions or mismatches. Thus $FA_I(\rho, \varepsilon_{max})$ is a DFA that, upon scanning u, reaches its first final state once the substring ρ has been read. $FA_I(\rho, \varepsilon_{max})$ continues to reach final states after scanning each word, u_i (where $i = 2 \cdots p$) provided that one of the following conditions hold: a) either $u_i = \rho$ or b) u_i is a word deduced from ρ that contains a maximum of ε_{max} insertions.* □

2.5 Distances

To detect TRs, a Levenshtein Distance (LD) is calculated between a PTRE and a potential TRE in Chapter 7. For the same reason, in Chapter 8, the so called Levenshtein Correspondence (LC), is calculated. This section introduces the LC and the LC$_n$. Theoretical background, for calculating both the LD and the LC, is provided here.

In 1966, Levenshtein defined the *Levenshtein Distance* (LD) as provided in Definition 2.5.1. The LD is also commonly referred to as the edit distance. The LD is often calculated during approximate string matching. In such approximate string matching scenarios, some pre-assigned number, k, is used as an upper bound on the LD between the source and destination string. If the LD between these two strings is found to be less or equal to k then an approximate match is said to have occurred (Landau and Vishkin [1988]).

Automata in the literature are discussed in Section 2.5.1 that relate to LD computations. These automata characterise a given source string and have as language the set of destination strings that approximately match the source string because the associated LD is no more than a predetermined value, k. The final accepting state of automata can be used to determine the exact LD value for the source and destination strings.

Another approach that is used to calculate the LD is dynamic programming. In Section 2.5.2 this approach is explained. It will be seen that `FireSat`$_2$ calculates a Levenshtein Based Distance. This implies that `FireSat`$_2$ makes provision to calculate the distance between two strings allowing for mismatches, insertions and deletions. Mismatches and deletions are, however, not allowed after an insertion has occurred. Only a perfect match is allowed after an insertion. Consequently `FireSat`$_2$ cannot calculate an exact LD between two strings. The LD between two strings can be determined by `FireSat`$_{2'}$.

Definition 2.5.1. *Levenshtein Distance*
The Levenshtein distance (LD) between two strings (called source and destination strings) is the minimum number of insertion, deletion and mismatch operations required to transform the source into the destination (Levenshtein [1966]). □

The Levenshtein Correspondence (LC) is introduced in Section 2.5.3 and it is shown how this LC can also be determined using dynamic programming. The LC provides the theoretical underpinning of `FireSat`$_3$.

2.5.1 Automata for calculating the LD

Assume that k (the number of motif errors allowed) is given, as well as destination string d. All strings that differ from destination string d by an LD of at most k are considered to match d approximately.

A corresponding approximate string matching NFA for destination string d can be constructed using $k+1$ DFAs $(M_0, M_1, \cdots M_k)$, for $k+1$ different deviations from destination string d: one for *no error* (level 0), one for *one error* (level 1), $\cdots$ one for *k-errors* (level k). These $k+1$ automata are connected by transitions that represent the different edit operations (Holub [2010]). In Figure 2.23 a Levenshtein based NFA accepting the word `acgt` is illustrated. In this case $k = 2$.

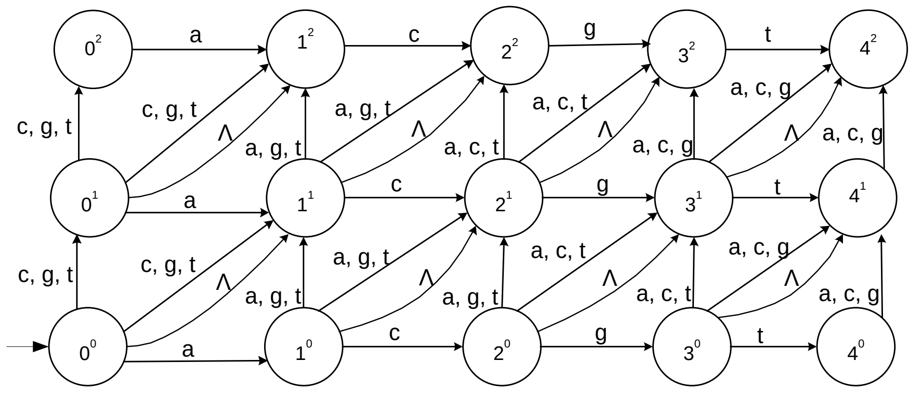


Figure 2.23: A Levenshtein automaton (NFA) for the word $\rho =$ `acgt` with a maximum edit distance of 2, allowing insertions before ρ, in between characters of ρ, and after ρ.

The start state of the Levenshtein automata presented in Figure 2.23 is in the lower left of the NFA. In Figure 2.23 ϵ indicates the number of errors that occurred en route to the state under consideration. Horizontal transitions represent perfect matches. Vertical transitions represent insertions. Diagonal transitions labelled with characters from Σ are mismatches whereas diagonal transitions labelled with Λ represent deletions. Given a Levenshtein automaton designed for a maximum edit distance of k and a specified source string, ρ, input over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$ will be accepted if and only if that input has an LD of at most k with respect to ρ. In the case of the automata presented in Figure 2.23, $k = 2$ and ρ is `acgt`. The input $d =$ `attt` will be accepted, and execution will end at state 4^2 indicating an LD of 2 between ρ and d. (Hjelmqvist [2012]).

In Figure 2.23, the string `cgt` can reach the final state 4^2 via the path $0^0 1^1 2^2 3^2 4^2$. This affirms that *maximal* LD between destination string `acgt` and source string `cgt` is 2. It does not indicate the actual LD. However, because Levenshtein automata are *nondeterministic*, there may be multiple paths mapping a given source string from an automaton's source state to a final state. In the present example, `cgt` can also be mapped onto the path $0^0 1^1 2^1 3^1 4^1$ to reach the final state 4^1, suggesting that the LD may be 1 or even less. However, since there is no path mapping `cgt` to final state 4^0, we can conclude that the *actual* LD is 1 and not 0.

For a given source string, there can even be more than one path to the *same* final state. This reflects the fact that the LD specifies the minimum edit distance between source and destination string only—not the actual errors that give rise to this minimum. Subsection 2.5.4 will briefly illustrate that there could be ambiguity with respect to the specific errors that give rise to a given LD.

Weights or penalties can be associated with the different edit operations and assigned to the corresponding transitions. In such a case, a path weight is associated with each accepted string. Should there be more than one path from the start to a final state for a given input string, then the corresponding path weights can be used to decide which sequence of edit operations to assume that resulted in the LD. Kurtz [1996] mentions that the weight function plays an important role within a biological context—by allocating appropriate weights to functions, matches that are biologically relevant can be made. Clearly, if no weights are allocated then all operations are assumed to be equally weighted.

Holub [2010] presents an alternative Levenshtein based automata, shown in Figure 2.24, that has fewer states than the previous one and that does not provide for insertions at the beginning and/or the end of ρ. Insertions are only allowed in between characters of ρ.

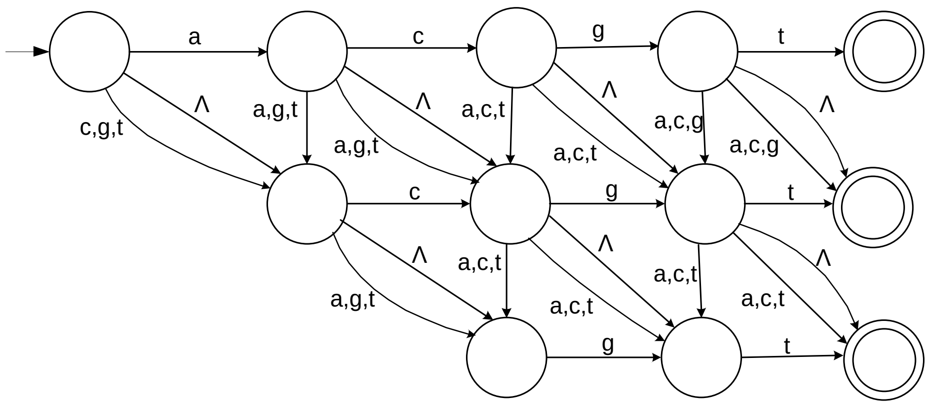


Figure 2.24: A Levenshtein automaton (NFA) for the word ρ = `acgt` with a maximum edit distance of 2, not allowing insertions before and/or after ρ.

Numerous algorithms have been developed to calculate the LD. In the foregoing, it has been illustrated *how* an NFA can be used to determine whether the LD is less than some maximal value characterising the associated NFA. Melichar [1996] showed that the number of states required by an equivalent DFA is at most $(k+1)!(k+2)^{m-2}$, where k is the upper bound on the edit distance allowed and m is the length of the source string.

2.5.2 Dynamic programming for calculating the LD

In addition to using automata, there are various other ways of calculating the LD. These include the recursive approach provided in Wagner and Fischer [1974] and Wagner and Lowrance [1975]. In this section, a dynamic programming algorithm (based on the algorithmic principles published by Navarro [2001]) is used to show how the LD can be calculated.

Let LD(s, d) refer to the LD to be computed between a source string s and a destination string d. A matrix, M, such as the one in Figure 2.25 has to be populated, where an entry in row i and column j, say $M_{i,j}$, represents the minimum number of operations required to match $s_{1\cdots i}$ to $d_{1\cdots j}$—i.e. $M_{i,j}$ represents $LD(s_{1\cdots i}, d_{1\cdots j})$, and consequently $M_{|s||d|}$ represents $LD(s, d)$.

Elements of the matrix can be recursively computed using the following formulae:

$$M_{0,0} = 0 \tag{2.3}$$

$$M_{i,0} = i \text{ for } i = 1 \ldots |s| \tag{2.4}$$

$$M_{0,j} = j \text{ for } j = 1 \ldots |d| \tag{2.5}$$

$$M_{i,j} = \begin{cases} M_{i-1,j-1} & \text{for } (i>0),(j>0) \text{ and } (s_i = d_j) \\ 1 + \min(M_{i-1,j-1}, M_{i,j-1}, M_{i-1,j}) & \text{for } (i>0),(j>0) \text{ and } (s_i \neq d_j) \end{cases} \tag{2.6}$$

These formulae can be explained as follows.

- $M_{0,0}$ is the LD between the empty string and itself, taken to be 0 as indicated in Equation (2.3).
- $M_{i,0}$ is the LD between the string $s_{1\ldots i}$ (a prefix of s of length i) and the empty string. Its value is i as indicated in Equation (2.4). Similarly, $M_{0,j}$ is the LD between the empty string and $d_{1\ldots j}$ (a prefix of d of length j). Its value is j as indicated in Equation (2.5).
- Given two non-empty strings $s_{1\cdots i}$ and $d_{1\cdots j}$, assume inductively that the LD between each pair of their respective prefixes has already been computed. If $s_i = d_j$ then clearly $M_{i,j} = M_{i-1,j-1}$ as indicated in the first case of Equation (2.6).

- To determine the LD between $s_{1 \dots i}$ and $d_{1 \dots j}$ when $s_i \neq d_j$, distinguish between the following possibilities:
  - If $s_i \neq d_j$ is attributed to a mismatch between s_i and d_j, then $M_{i,j} = 1 + M_{i-1,j-1}$, since $M_{i-1,j-1}$ is the LD between $s_{1..(i-1)}$ and $d_{1..(j-1)}$.
  - If $s_i \neq d_j$ is attributed to a deletion just after s_i, then $M_{i,j} = 1 + M_{i,j-1}$ since $M_{i,j-1}$ is the LD between $s_{1..i}$ and $d_{1..(j-1)}$.
  - If $s_i \neq d_j$ is attributed to s_i being an insertion just after s_{i-1}, then $M_{i,j} = 1 + M_{i-1,j}$ since $M_{i-1,j}$ is the LD between $s_{1..(i-1)}$ and $d_{1..j}$.

  The actual value of $M_{i,j}$ is now the minimum value of the three above possibilities. This is shown in the second case of Equation 2.6.

To illustrate these computations, consider the matrix, M, in Figure 2.25 in which $s =$ `acctg` and $d =$ `acttag` respectively. Note that the top left-most element, 0, corresponds with Equation (2.3); the rest of that first column corresponds with Equation (2.4) and the rest of the first row corresponds with Equation (2.5).

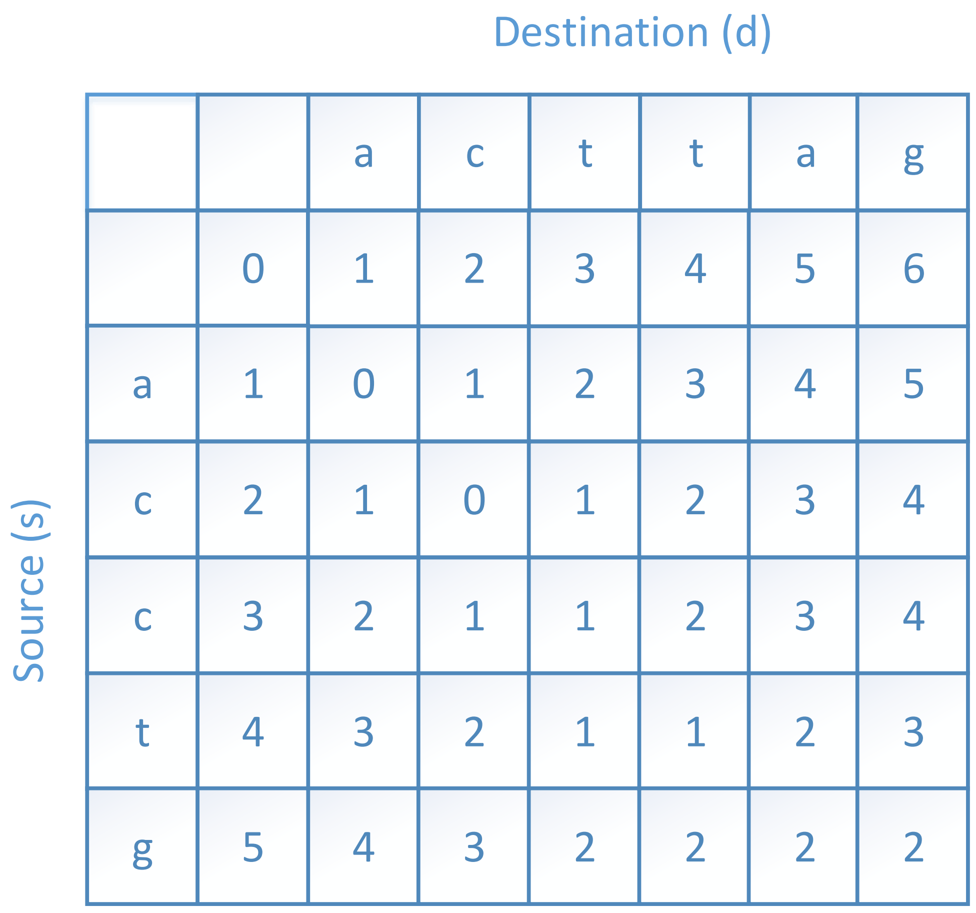
Since the first characters of s and d match (both being `a`) we allocate the value of $M_{1,1} = M_{0,0}$ as indicated by the first case of Equation (2.6). Thus $M_{1,1} = 0$.

In order to find $M_{1,2}$ the first character of the source string (s_1) is compared with the second character of the destination string (d_2). Since $s_1 \neq d_2$ (i.e. `a` $\neq$ `c`), the three possibilities implied in Equation (2.6) have to be considered:

1. If $s_1 \neq d_2$ is to be interpreted as a mismatch, this would mean that in a previous step d_1 was compared against the empty string at a cost given by $M_{1-1,2-1} = M_{0,1}$. The total cost of the mismatch that now occurs is therefore $M_{0,1} + 1 = 1 + 1 = 2$.

2. If $s_1 \neq d_2$ is to be interpreted as due to a deletion of something after s_1, this would mean that in evaluating d_1 against s_1 in a previous step, a minimum number of $M_{1,1}$ errors was found to have occurred, and now an additional error has been found when comparing d_2 against the empty string representing the deleted element after s_1. The total cost due to such a deletion would therefore be $M_{1,1} + 1 = 0 + 1 = 1$.

3. If $s_1 \neq d_2$ is to be interpreted as the insertion of s_1 after some prior source string prefix (in this case, the empty string), then the total cost due to such an insertion would be the cost of matching $d_1 d_2$ against the empty string (namely $M_{0,2}$) plus the cost of the insertion, s_1—i.e. the total cost is given by $M_{0,2} + 1 = 2 + 1 = 3$.


Destination (d)
Source (s)

		a	c	t	t	a	g
	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
c	2	1	0	1	2	3	4
c	3	2	1	1	2	3	4
t	4	3	2	1	1	2	3
g	5	4	3	2	2	2	2

Figure 2.25: Calculating the LD between the words `acttag` and `acctg`.

Of the three possibilities the cheapest option is number 2—the deletion option. Thus the value of entry $M_{1,2}$ becomes 1.

Within the context of TR-detection it is meaningful to normalise the LD. The normalised LD is referred to as the LD$_n$ here. The LD$_n$ is meaningful because it is difficult to determine if an LD, by itself, represents a high or a low degree of similarity when various lengths of ρ are considered.

Definition 2.5.2. ***The Normalised Levenshtein Distance*** $\oplus$
$LD_n(s, d)$, where s and d denote the relevant source and destination strings is

calculated as follows:

$$LD_n(s,d) = \frac{LD(s,d)}{max(|s||d|)} \tag{2.7}$$

□

Note that this definition corresponds to that of Doran and Van Wamelen [2010].

Dynamic programming entails recursive computation where the current value to be computed relies on the values already computed and recorded in a previous recursion. In the present case, the matrix has to be filled in, in such a way that, when the value in cell (i,j) is to be computed, then values for cells $(i-1,j)$, $(i,j-1)$ and $(i-1,j-i)$ have to be available. This can clearly be achieved by a row-wise left to right traversal of columns. It can also be achieved by a column-wise top-to-bottom traversal of rows. However, this can also be achieved by a so called *mirrored diagonally* based traversal. This is briefly explained in Chapter 8, Section 8.2.4.

2.5.3 Dynamic programming to calculate the LC

The LC between two strings is calculated in a similar manner to the LD calculation, again using dynamic programming principles. Again, there is more than one way of calculating the LC.

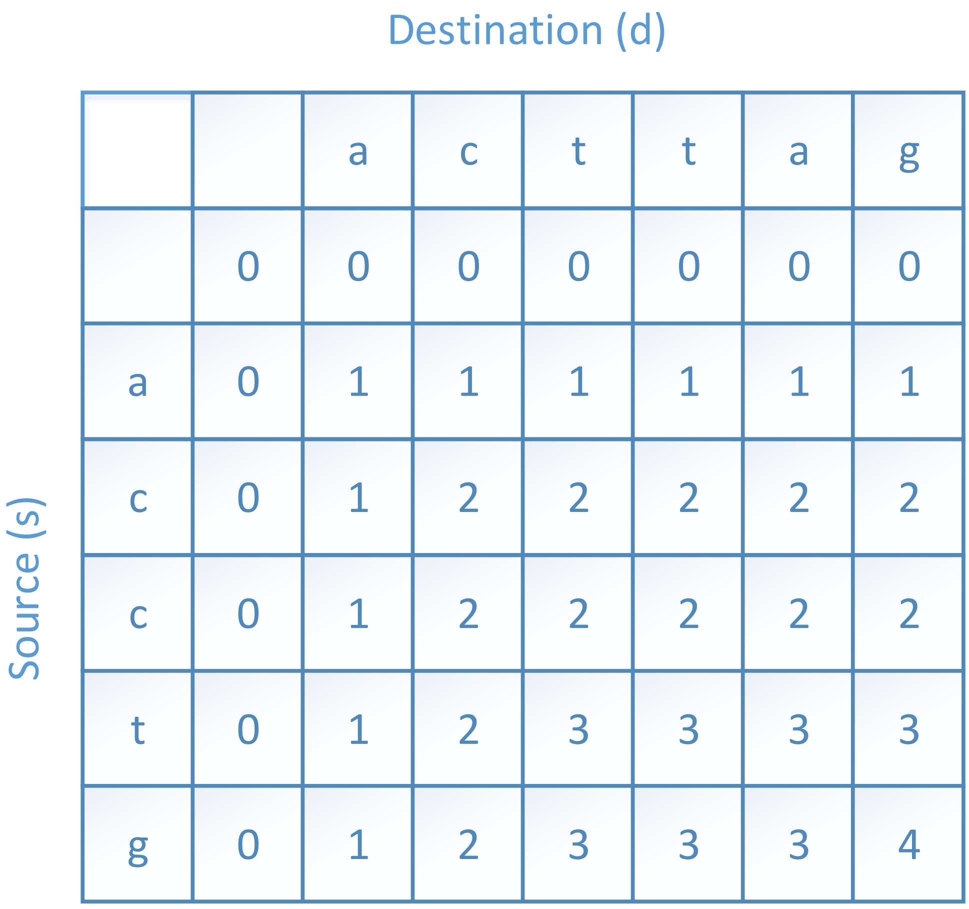
Instead of computing the minimum distance between two strings s and d like the LD does, the LC computes the maximum number of perfect matches between the two strings. Clearly the LD and the LC are related. Here is its formal definition:

Definition 2.5.3. ***Levenshtein correspondence***
$LC(s,d)$ denotes the Levenshtein correspondence *(LC) between a source string, s, and a destination string, d, and is defined as:*

$$max(|s|,|d|) - LD(s,d)$$

□

Thus $LC(s,d)$ refers to the maximum number of matches between s and d. To compute $LC(s,d)$, matrix M such as the one displayed in Figure 2.26, can be compiled.


Destination (d)
Source (s)

		a	c	t	t	a	g
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	2	2	2	2	2
c	0	1	2	2	2	2	2
t	0	1	2	3	3	3	3
g	0	1	2	3	3	3	4

Figure 2.26: Calculating the LC between the words `acttag` and `acctg`.

The contents of M is determined by an algorithm similar to that of Smith and

Waterman [1981][13]. The formula below is used to complete matrix M.

$$M_{0,0} = 0 \tag{2.8}$$

$$M_{i,0} = 0 \text{ for } i = 1 \ldots |s| \tag{2.9}$$

$$M_{0,j} = 0 \text{ for } j = 1 \ldots |d| \tag{2.10}$$

$$M_{i,j} = \begin{cases} 1 + M_{i-1,j-1} & \text{for } (i > 0), (j > 0) \text{ and } (s_i = d_j) \\ \max(M_{i-1,j-1}, M_{i,j-1}, M_{i-1,j}) & \text{for } (i > 0), (j > 0) \text{ and } (s_i \neq d_j) \end{cases} \tag{2.11}$$

Thus, as seen in Equations (2.8), (2.9) and (2.10), row 0 and column 0 are initialised with zeroes. This is because the LC between the empty string and each prefix of d is clearly zero, and likewise for each prefix of s.

Equation (2.11), needed to produce the remaining contents of M, has a *dual* relationship to its counterpart Equation (2.6) for computing the LD. When determining $M_{i,j}$, the maximum number of matches between $s_{1\cdots i}$ and $d_{1\cdots j}$, simply add 1 to $M_{i-1,j-1}$ if $s_i = d_j$. Otherwise determine the *maximum* of $M_{i-1,j}$, $M_{i,j-1}$ and $M_{i-1,j-1}$.

As an illustration, consider how to determine the maximum number of matches between $s_{1\cdots i}$ and $d_{1\cdots j}$. Proceed in a row-wise manner—consider first each column (from left to right) of row 1, then each column of row 2, etc.

In considering the first characters of s and d, i.e. $s_1 = d_1$, a match occurs, and thus $M_{1,1} = M_{0,0} + 1 = 0 + 1 = 1$, as indicated by the first case of Equation (2.11). Next, since $s_1 \neq d_2$, the second case of Equation (2.11) applies. The following three possibilities, therefore, have to be considered and the best outcome should be chosen:

1. As before, if $s_1 \neq d_2$ is to be interpreted as a mismatch, this would mean that in a previous step d_1 was compared against the empty string and the match count at that stage was therefore $M_{1-1,2-1} = M_{0,1}$. The total number of matches now would therefore be 0.

2. Similarly, if $s_1 \neq d_2$ is to be interpreted as due to a deletion of something after s_1, this would mean that in evaluating d_1 against s_1 in a previous step, a maximum number of $M_{1,1}$ matches had occurred, and now an error has been found when comparing d_2 against the empty string representing the deleted element after s_1. The total matches due to such a deletion would therefore be $M_{1,1} = 1$.

[13] This algorithm performs a local sequence alignment (entails the matching of a string s with substrings of d) and is guaranteed to find the optimal local alignment in correspondence with the scoring system (weight assignment) being used. It is a variation of an algorithm originally proposed by Needleman and Wunsch [1970]. A detailed discussion of the Smith-Waterman and the Needleman-Wunsh algorithms is beyond the scope of this thesis.

3. Finally, if $s_1 \neq d_2$ is to be interpreted as the insertion of s_1 after some prior source string prefix (in this case, the empty string), then the total number of matches due to such an insertion would be the number of matches of $d_1 d_2$ against the empty string (namely $M_{0,2}$)—i.e. the total matches would be $M_{0,2} = 0$.

Thus $M_{1,2}$ is 1, i.e. the maximum value of the three above scenarios, and that occurs when the source string is viewed as having something deleted after s_1.

As before, the entire matrix may be computed in this row-wise fashion, or in a column-wise fashion.

In this study, the LC will be computed in a context where a given motif ρ serves as the destination string and nucleotide sequences being read from the input are considered source strings. The source strings are thus evaluated as being TREs in relation to ρ. In such a context, it is reasonable to limit the length of the source string to $|S| = \lceil (1 + \frac{\varepsilon_{max\%}}{100}) \times |\rho| \rceil$.

It is hard to interpret the LC without referencing the length of the strings. For instance, an LC value of 10 might mean exact correspondence between two strings of length 10; it might mean that there are only 10 matches between two strings, each of length 100; it might mean there are 10 matches between a destination string of length 10 and a source string of length 15; etc. For that reason, it is desirable to *normalise* the value of an LC in relation to the length of the destination string. For example, in the first case cited above, the match rate is reflected by the computation $\frac{10}{10} = 1$; and in the second case it is $\frac{10}{100} = 0.1$. In the third case one might say that the beneficial effect of the 10 matches is negated by 5 insertions that have occurred, and compute the value of $\frac{10-5}{10} = 0.5$ to reflect the extent of matching between source and destination.

Clearly the LC and LD are closely related. This is illustrated in Definition 2.5.4.

Definition 2.5.4. ***The normalised Levenshtein correspondence defined i.t.o. the LD*** $^{\odot}$
$LC_n(s, d)$, where s and d denote the relevant source and destination strings is calculated as follows:

$$LC_n(s, d) = 1 - \frac{LD(s, d)}{max(|s|, |d|)} \tag{2.12}$$

□

Trial runs showed however, that an adapted version of the LC$_n$ is more suitable within the present context. Therefore the adapted LC$_n$, Definition 2.5.5, will be used in subsequent work.

Definition 2.5.5. ***The normalised Levenshtein correspondence*** $^{\odot}$
$LC_n(s,d)$, where s and d denote the relevant source and destination strings is calculated as follows:

$$LC_n(s,d) = \frac{LC(s,d) - |ins|}{|d|} \text{ and if } LC_n(s,d) < 0 \text{ then } LC(s,d) = 0 \qquad (2.13)$$

$\square$

Definition 2.5.6. ***The match score*** $^{\odot}$
$LC_n(s,d)$, where s and d denote the relevant source and destination strings, is calculated as follows:

$$match\ score = LC_n(s,d) \qquad (2.14)$$

$\square$

The *match score* is equal to the LC_n. In Chapter 9, after a *recall precision* (RP) analysis a *match score threshold function* and *match score threshold factor* are deduced. The *match score*, *match score threshold function* and *match score threshold factor* determine the *best fit* within the context of `FireSat` TR detection. Details of these concepts, including the RP-analysis, are postponed until Chapter 9.

2.5.4 Ambiguity

As indicated in Section 2.5.1, there may be more than one way of assigning mutation-types when aligning two strings to determine an LD or LC value. Consider, for example, the source string `cgtacac` and destination string `ccctagac`. The matrix for computing the LD is shown in Figure 2.27. The LD is 3, as displayed in the bottom right corner.

However, there are three different ways of assigning mutations to the source string to arrive at this LD value of 3. These are shown in Table 2.2. In the table, D indicates a deletion in the source string, P a destination string match with the source string and M a mismatch between the destination and source string.

In the first match pattern, a deletion is deemed to have occurred, followed by two mismatches. In the second match pattern, a deletion again occurs before two mismatches, but now in the second position of the source string. In the third match pattern, a mismatch is identified before the first deletion. Thereafter, another mismatch is reported.

Destination (d)

Source (s)

		c	c	c	t	a	g	a	c
	0	1	2	3	4	5	6	7	8
c	1	0	1	2	3	4	5	6	7
g	2	1	1	2	3	4	4	5	6
t	3	2	2	2	2	3	4	5	6
a	4	3	3	3	3	2	3	4	5
c	5	4	3	3	4	3	3	4	4
a	6	5	4	4	4	4	4	3	4
c	7	6	5	4	5	5	5	4	3

Figure 2.27: Calculating the LD between `cgtacac` and `ccctagac`.

2.6 Conclusion

In this chapter terminology, definitions and notations have been presented that are used throughout the remainder of this thesis. In the rare cases where additional terminology is required, it will be introduced at that stage. Note that algorithms are presented in Dijkstra's guarded command language (GCL) (Kourie and Watson [2012]; Vide et al. [2003] and Dahl et al. [1972]).

In Chapter 3 a literature overview of implemented algorithms searching TRs is presented.

Destination string	c	c	c	t	a	g	a	c
Source string		c	g	t	a	c	a	c
Match pattern 1	D	P	M	P	P	M	P	P
Source string	c		g	t	a	c	a	c
Match pattern 2	P	D	M	P	P	M	P	P
Source string	c	g		t	a	c	a	c
Match pattern 3	P	M	D	P	P	M	P	P

Table 2.2: Alternative match patterns giving the same LD value.

Chapter References

D. I. A. Cohen. *Introduction to Computer Theory*. Wiley: New York, 2nd ed. edition, 1997.

O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, volume 8. Academic Press, 1972. ISBN 0122005503.

C. De Ridder. Flexible Finite automata-based algorithms for detecting microsatellites in DNA. Master's thesis, Department of Computer Science, University of Pretoria, July 2010.

H.C. Doran and P.B. Van Wamelen. Application of the Levenshtein Distance Metric for the Construction of Longitudinal Data Files. *Educational Measurements: Issues and Practice*, 29(2):13—23, 2010.

S. Hjelmqvist. Fast, memory efficient Levenshtein algorithm. Online: http://www.codeproject.com/Articles/13525/Fast-memory-efficient-Levenshtein-algorithm, 2012.

J. Holub. *Finite Automata in pattern matching.* John Wiley & Sons inc., 1st edition, 2010. ISBN 978-0-470-50519-9.

J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory languages and computation.* Addison-Wesley publishing company, 1979.

A. Kempe. Reduction of intermediate alphabets in finite-state transducer cascades. In *TALN 2000 : 7e confrence annuelle sur le Traitement Automatique des Langues Naturelles*, pages 207—215, 2000. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.2744.

A. Kent, J.G. Williams, C. M. Hall, and R. Kent. *Encyclopedia of Computer Science and Technology*, volume 25 no 10, page 91. Marcel Dekker, Inc; New York and Basel, 1992.

D.G. Kourie and B.W. Watson. *The Correctness-by-Construction Approach to Programming.* Springer, Berlin Heidelberg, first edition, 2012.

S. Kurtz. Approximate string searching under weighted edit distance. In *Proceedings of the Third South American Workshop on String Processing*, Ottawa, Canada, 1996. Carlton University Press. URL http://www.zbh.uni-hamburg.de/pubs/pdf/Kur1996.pdf.

G.M. Landau and U. Vishkin. Fast String Matching with k Differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi: 10.1016/0022-0000(88)90045-1.

V.I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–708, 1966.

K. T. Masombuka, C. de Ridder, and D. G. Kourie. An investigation of software for minisatellite detection. In Fred Nicolls, editor, *Proceedings on the twenty-first PRASA symposium*, pages 171–176, November 2010. ISBN 978-0-7992-2470-2.

D. Matuszek. Linear-bounded automata. Online: http://www.seas.upenn.edu/ cit596/notes/dave/chomsky2.html, 1996.

B. Melichar. Space complexity of linear approximate string matching. In *Proceedings of the Prague Stringology Club Workshop '96*, pages 28—36, 1996. URL http://www.stringology.org/event/1996/p4.html.

G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31—88, 2001.

S. B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443—453, 1970.

E. Rivals, J. P. Delahaye, O. Delgrange, and M. Dauchet. A first step toward chromosome analysis by compression algorithms. In *Proceedings of the First International IEEE Symposium on Intelligence in Neural and Biological Systems*, pages 233–239, 1995.

A. Salomaa. *Computation and Automata* , volume 25 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 1985. ISBN 0 521 30245 5.

W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

E. Schaper, A. Korsunsky, A. Messina, R. Murri, J. Pečerska, H. Stockinger, S. Zoller, I. Xenarios, and M. Anisimova. TRAL: Tandem repeat annotation library. *Bioinformtics*, 31(18):3051–3053, 2015.

M. Sipser. *Introduction to the Theory of Computation*. COURSE TECHNOLOGY, CENGAGE Learning, 2 edition, 2006. ISBN 978-0-619-21764-8.

T.F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

C.M. Sperberg-McQueen. Notes on finite state automata with counters. Online: https://www.w3.org/XML/2004/05/msm-cfa.html, 2004.

K. Sutner. Context sensitive grammars, 2011a. URL https://www.cs.cmu.edu/~./FLAC/pdf/ContSens-6up.pdf.

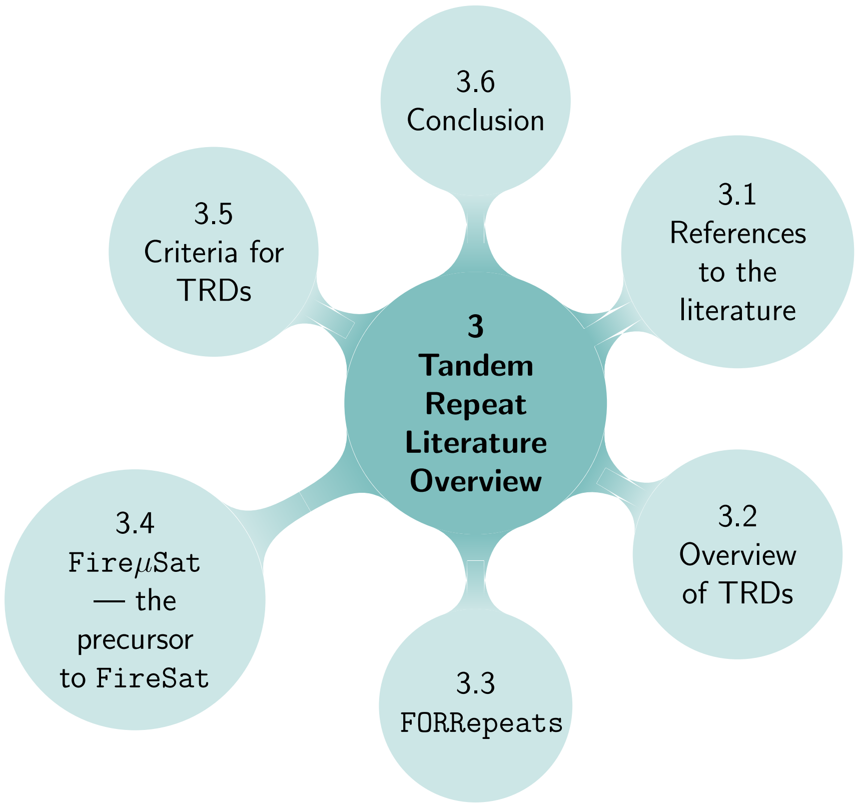
K. Sutner. Context free grammars, 2011b. URL http://www.cs.cmu.edu/~./FLAC/pdf/CFG.pdf.

C. M. Vide, V. Mitrana, and Gheorghe Paun. *Grammars and Automata for String Processing*. CRC Press, 2003.

R. A. Wagner and M. J. Fischer. The String-to-String Correction Problem. *J. ACM*, 21(1):168–173, Jan 1974. ISSN 0004-5411. doi: 10.1145/321796.321811. URL http://doi.acm.org/10.1145/321796.321811.

R. A. Wagner and R. Lowrance. An Extension of the String-to-String Correction Problem. *J. ACM*, 22(2):177–183, Apr 1975. ISSN 0004-5411. doi: 10.1145/321879.321880. URL http://doi.acm.org/10.1145/321879.321880.

B.W. Watson. *Constructing minimal acyclic deterministic finite automata*. PhD thesis, Faculty of Engineering, Built Environment and Information Technology, University of Pretoria, 2010.


3.6
Conclusion
3.5
Criteria for TRDs
3.1
References to the literature
3
Tandem Repeat Literature Overview
3.4
FireμSat — the precursor to FireSat
3.2
Overview of TRDs
3.3
FORRepeats

3 Tandem Repeat Literature Overview

"With everything that has happened to you, you can either feel sorry for yourself or treat what has happened as a gift. Everything is either an opportunity to grow or an obstacle to keep you from growing. You get to choose." ... Wayne W. Dyer

---

This work significantly extends De Ridder [2010] research[1] verifying that FAs can be used to effectively detect *microsatellites* in DNA. As mentioned in Chapter 1, the purpose of this thesis is to extend the above for *all* repeats—microsatellites, minisatellites and satellites. It will be seen—especially in Chapter 5—that the approach of `FireSat` differs significantly from that of `Fire`μ`Sat`, the algorithm developed as part of the MSc.

This chapter discusses the principal literature sources relevant to this thesis. It is laid out as follows:

- Section 3.1 provides references to repeat detection research literature. The first part focusses on research that has been either authored or co-authored

[1] http://upetd.up.ac.za/thesis/available/etd-08172010-202532/unrestricted/dissertation.pdf

by me. These sources may be consulted for back-references prior to 2014. The second part references material that summarises research subsequent to 2014.

- Detailed descriptions of TR detecting algorithms, referred to as TR-detectors (TRDs), will not be included here. (De Ridder [2010] describes the most prominent TRDs at the time of that study.) Instead, Section 3.2 gives a high-level overview of TRDs. It uses so-called formal concept lattices to reflect the interrelationship between various TRDs and their respective attributes. Therefore Section 3.2 also includes a brief introduction to formal concept lattices.

- `FORRepeats` [Lefebvre et al., 2003] is discussed in Section 3.3. The reason for focusing on this TRD is because, apart from Fire$_\mu$Sat and `FireSat`, it is the only one that employs finite automata to detect minisatellites.

- Thereafter, in Section 3.4, results that were obtained with Fire$_\mu$Sat are briefly reviewed. This provides some context for appreciating `FireSat` proposed in this thesis.

- The most important literature-based criteria to which repeat detection should yield are reviewed in Section 3.5.

- Section 3.6 concludes this chapter.

3.1 References to the literature

This section provides in Subsection 3.1.1 references to personal work related to TR detection. In Subsection 3.1.2 references to TR detection algorithms developed by other researchers are provided.

3.1.1 References to personal work

The objective of the first part of Chapter 2 of De Ridder [2010] was threefold:

1. to provide the reader with biological background required to understand the context of the research;

2. to introduce relevant biological terminology; and

3. to provide a literature overview that explained existing implemented algorithms for detecting repeats.

Neither the biological background nor the literature overview will be repeated here. Terminology essential to the comprehension of this thesis has been given in the previous chapter.

Although the focus of De Ridder [2010] was microsatellites, several packages detecting minisatellites and satellites (including `TRF`) were included as part of the literature review. In Chapter 3 of De Ridder [2010], a detailed discussion of two algorithms—Tandem Repeats Finder (`TRF`) and Search for Tandem Approximate Repeats (`STAR`)—is to be found. `TRF` detects microsatellites, minisatellites as well as satellites, whereas `STAR` only detects microsatellites. The literature usually relies on `TRF` for benchmarking purposes [Lim et al., 2012, Rivals, 2004, Schaper et al., 2015].

Masombuka (2008) completed his honours project under my supervision and discussed three algorithms for repeat detection. `Phobos`, `mreps` and `REPuter` are discussed in his mini-dissertation. Together with Masombuka and Kourie, I co-authored a paper entitled *An investigation of software for minisatellite detection.* The paper was presented at the *Pattern Recognition Association of South-Africa, 2010* (PRASA2010). We investigated four algorithms: `TRF`, `ATR-Hunter`, `Phobos` and `mreps` and compared their runtime as well as their output.

References to work related to repeat detection that I have either authored or co-authored are provided below. Note that these publications include, to a larger or a lesser extent, literature overviews of existing algorithms. Extensive literature overviews are to be found in De Ridder [2010], De Ridder et al. [2006b] and Masombuka et al. [2010].

- De Ridder et al. [2006a];
- De Ridder et al. [2006b];
- De Ridder [2010];
- Masombuka et al. [2010];
- De Ridder et al. [2011] and;
- De Ridder et al. [2013].

3.1.2 References to other research

Anisimova et al. [2015] and Schaper et al. [2015] maintain that an exhaustive search to detect all TRs is not possible. This is because TRs are context dependant — i.e. researchers have different opinions about whether a given string constitutes a TR, based on how many errors they are prepared to tolerate, whether they prioritise long motifs over shorter ones, whether they prioritise mismatches over indels etc. Thus, practically any string can be regarded as a TR if we allow a high enough tolerance. In the case of `FireSat`$_2$ and `FireSat`$_3$ it is possible (not necessarily meaningful) to allow the LD such that $|PTRE| = |\varepsilon_{max}|$. Consequently if a researcher knows exactly how many of which type of errors he/she wants to be accommodated, the penalties, threshold values and other parameters can be set accordingly and an exhaustive search becomes a possibility at the expense of space and runtime.

TRs are based on different algorithmic paradigms, as discussed in this chapter. Consequently there is a significant discrepancy between TR annotations adopted by different TRD-algorithms [Anisimova et al., 2015].

Indels introduce length variability between individual TREs. According to Anisimova et al. [2015], this increases the TR search space to $O(2^N N^3)$. Thus, it would seem that there has been some research conducted with a focus on investigating existing TRDs. It appears to focus in part on the validation of tandem repeats, statistically or otherwise. Examples include the following studies:

- Schaper et al. [2012] suggest the reconciliation of diverse predictions of current algorithms. They propose statistical criteria for measuring the quality of predicted TRs and for determining a maximum-likelihood estimation of TR divergence.

- Schaper et al. [2015] believe that it is essential to detect TRs by using a variety of TRDs. These TRs should then be combined for reliable TR annotation. They developed `TRA`, a tandem repeat annotation library[2]. Note that although Schaper et al. [2015] mention that six TRDs are integrated in `TRA`, they actually list the following TRDs:

  – `HHrepID` [Biegert and Schoöding, 2008];
  – `Phobos` [Mayer, Phobos 3.3.11, 2006-2010];
  – `TRED` [Sokol et al., 2007][3];
  – `T-REKS` [Jorda and Kajava, 2009];
  – `TRF` [Benson, 1999];
  – `TRUST` [Szklarczyk and Heringa, 2004]; and
  – `XSTREAM` [Newman and Cooper, 2007].

  Of these TRDs, `TRUST`, `HHrepID` and `XSTREAM` focus on TR-detection in protein sequences. `T-REKS` focuses on microsatellite detection.

[2]Available at `http://www.vital-it.ch/software/tral`.

[3]At the time of writing the `TRED` software was not accessible at the web page referenced in Sokol et al. [2007].

- In De Ridder et al. [2013] `T-REKS` is investigated as one of the rival TRDs to FireμSat. `TRF` as well as `Phobos` are considered as rival TRDs to `FireSat` and will be reported on in this thesis[4].

There are a variety of algorithms that detect TRs. Four concept lattices provide an overview of such algorithms in the next section.

3.2 Overview of TRDs

Archambault [2012] compiled a table[5] focussing on software that detects microsatellites. Note that several packages that do not aim to detect microsatellites *per se* are also included in her report. This table served as the primary data source for the lattice-based study described in this subsection.

Twenty nine TRDs are listed below, twenty six of which are reported by Archambault [2012].

Note that packages relating to repeat detection, but not to tandem repeat detection *per se* are not included in Table 3.1 and the subsequent classification. `IRF` (Inverted Repeats Finder) [Warburton et al., 2004], `REPuter` [Kurtz et al., 2001] and `SRF` (Spectral Repeat Finder) [Sharma et al., 2004] are examples of such excluded packages. Note further that `FireSat` is included in the lattices but not in the subsequent discussions.

3.2.1 Brief Introduction to FCA

Four so-called concept lattices have been constructed to give an overview of the repeat detection field. Concept lattices have been chosen as a concise and descriptive way of giving an overview of repeat-detecting software. The following information, drawn from Kourie and Watson [2012], is provided to introduce concept lattices briefly.

Concept lattices are defined in a computer science / mathematics subfield of study referred to as formal concept analysis (FCA). FCA considers sets of objects in relation to the attributes that the objects hold in common. A rich source of information about the inter-relationship between a set of objects that share certain attributes can be obtained by creating and studying a concept lattice representing these objects and their related attributes.

Concept lattices are rooted in set theory. A concept lattice is derived from a set of *attributes*, M, that characterise *objects* in a set, G. The starting point for

[4]Note that De Ridder [2010] extensively reports on `TRF`.

[5]The table is available at: http://qcbs.ca/toolbox/qcbs-wiki/

Name	Reference
ATRHunter	Wexler et al. [2004]
BWTrs	Pokrzywa and Polanski [2010]
ComplexTR	Hauth and Joseph [2002]
ExTRs	Krishan and Tang [2004]
`Fire`$_\mu$`Sat`	De Ridder et al. [2006a]
FireSat	De Ridder et al. [2011]
`FORRepeats`	Lefebvre et al. [2003]
IMeX	Mudunuri and Nagarajaram [2007]
INVERTRER	Lim et al. [2012]
Landau-algorithm*	Landau et al. [2001]
Microsatellite Identification Tool (MISA)	Thiel et al. [2003]
mreps	Kolpakov et al. [2003]
Msatcommander	Faircloth [2008]
MsatFinder	Thurston and Field [2005]
OMWSA	Du et al. [2007]
Phobos	Mayer [Phobos 3.3.11, 2006-2010]
QDD1 and QDD2	Meglécz et al. [2010] Ref incompl
SciRoKo	Kofler and Sclötterer [2007]
Simple Sequence repeat Identification Tool (SSRIT)	Temnykh et al. [2001]
Sputnik	Abajian [1994]
SSRscanner	Anwar and Khan [2006]
STAR	Rivals [2004]
STRING	Parisi et al. [2003]
Tandem Repeats Finder (TRF)	Benson [1999]
Tandem Repeat Occurrence Locator (TROLL)	Castelo et al. [2002]
TandemSWAN	Boeva et al. [2006]
T-REKS	Jorda and Kajava [2009]
TRStalker	Pellegrini et al. [2010]
W-SSRF	Sreenu et al. [2003]

Table 3.1: Representative list of TRDs. (Note: The algorithm marked by * was not named explicitly by Landau, its author. It will be referred to here as the `Landau-algorithm`.)

deriving concepts is to represent the objects and their associated attributes in a cross-table, called the *context*. Each row of the context represents an object in G, each column represents an attribute in M, and binary entries (true/false, or cross/blank, or 0/1) in the cells indicate whether a given attribute characterises the associated object. Figure 3.1 is an example of a context and will be discussed later.

A concept is defined by a pair of sets (A, B) such that $A \subseteq G$ and $B \subseteq M$. However, these sets are constrained in a specific way. The pair (A, B) may only be reckoned as a concept if A consists of all objects that possess all the attributes in B. Additionally B must be maximal, implying that there are no additional attributes shared by A that are not included in B.

Concept (A, B) therefore have maximal sets from the given context in the following sense: Suppose one picks a set of objects, say A, and discovers that all objects in A have attributes B in common. That does not guarantee that (A, B) is a concept. In order for (A, B) to be a concept, the following two conditions have to be met:

- No object, say d, that is not in A also possesses *all* the attributes in B (though d may posses some of the attributes in B).
- The objects in A do not *all* possess common additional attributes that are not in B.

Only if the above holds can one conclude that (A, B) is a concept. The concept (A, B) is said to have an *extent* of A and an *intent* of B.

Thus, each context is associated with a set of concepts. Moreover, FCA defines a partial ordering on concepts, namely concept $(A_1, B_1) \leq (A_2, B_2)$ if and only if $(A_1 \subseteq A_2)$. (It can be shown that, dually, $(A_1, B_1) \leq (A_2, B_2)$ if and only if $(B_2 \subseteq B_1)$.)

As a result of this ordering relationship on concepts, it can be shown that the set of concepts in a given context constitute a lattice[6]. Software packages are available to derive all concepts from a given context and to display the concepts in terms of their ordering in a so-called *line diagram* visually. Nodes in such a diagram represent concepts, and arcs connect direct successors with respect to the partial ordering defined on concepts.

The concept lattice line diagrams presented below were set up using the open source tool known as Concept Explorer. Concept Explorer was developed by Yevtushenko [2000]. Figure 3.2 is an example of the line diagram derived from

[6]Technically, this means that each set of concepts has a unique supremum and a unique infimum.

the context in Figure 3.1 and will be further discussed below. For the present, however, notice that labels are attached to nodes (concepts) in the line diagrams. These labels can be used to infer the intent and extent of each concept (node).

A shaded label at a concept refers to an attribute in the context and is called an *own attribute* of the concept. The intent of a given concept can be found by collecting together all the own attributes *at or above* that concept.

Similarly, each clear label at a concept refers to an object in the context and is called an *own object* of the concept. The extent of a concept can be found by tracing all the own objects *at or below* a given concept.

It is not the intention to contribute to the field of FCA in this thesis. The terminology and definitions provided above will be used where applicable. For a more extensive discussion of concept lattices the interested reader is referred to Kourie and Watson [2012].

In this text, line diagrams of several concept lattices will be presented and discussed. In each case, the set of objects will be a set of TRDs and the set of attributes will be a set of attributes that are associated with these TRDs.

3.2.2 The full TRD lattice

The TRDs discussed in this section were published during the period of 1994 until 2013. These algorithms and their various attributes considered below are shown in the cross-table of Figure 3.1 which serves as base context for the concept lattices to be constructed. The table's column headings are briefly described as follows.

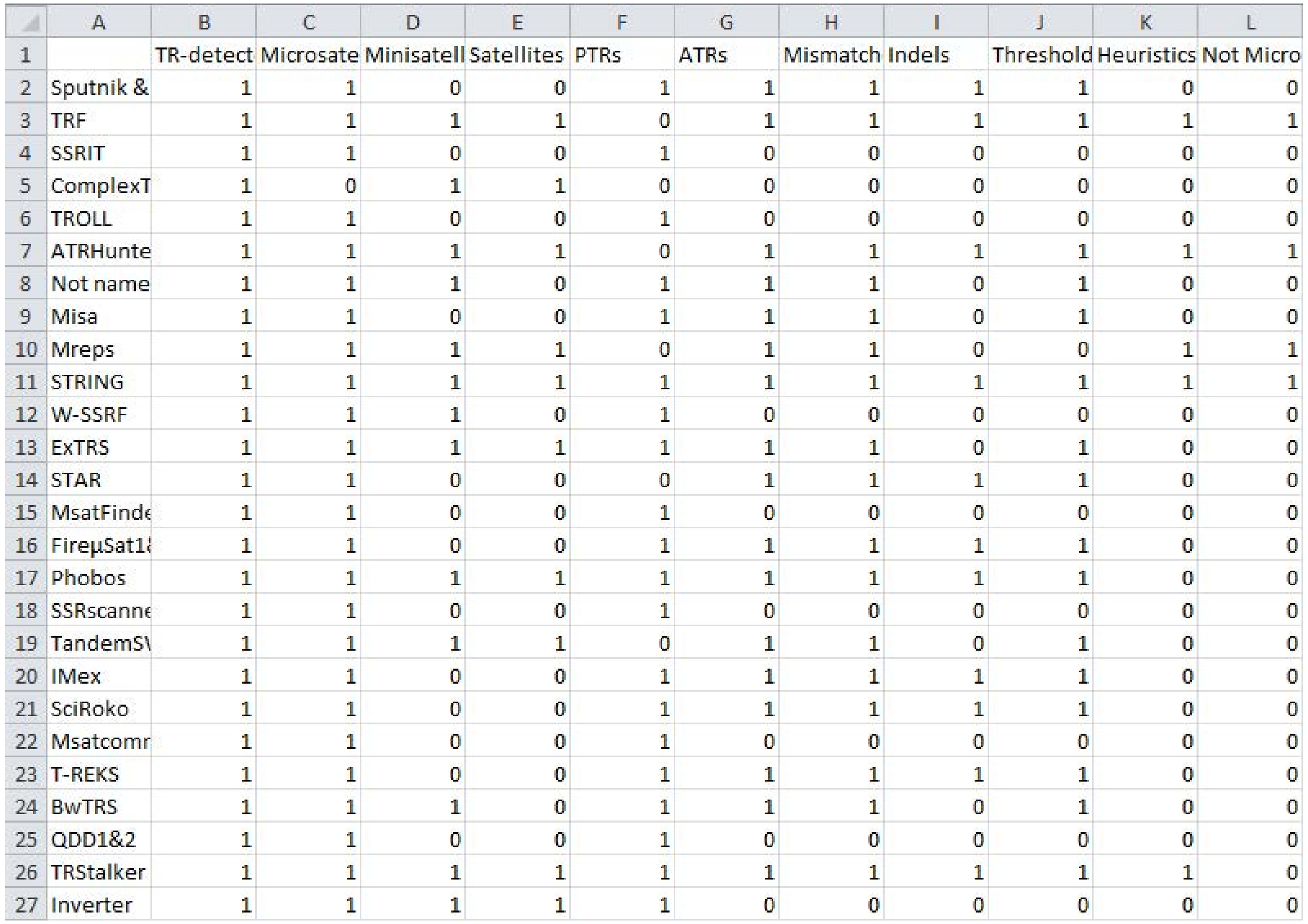
A The first column lists the 27 TRDs considered in the remainder of this section.

B *Microsatellites*
A *Microsatellite* is a TR whose repeated motif is restricted to $1 \leq |motif| \leq 6$.

C *Minisatellites*
A *Minisatellite* is a TR whose repeated motif is restricted to $7 \leq |motif| \leq 100$.

D *Satellites*
A *Satellite* is a TR whose repeated motif is $|motif| > 100$.

E *PTRs* (Perfect Tandem Repeats)
A *PTR* is a string of nucleotides characterized by a certain motif that introduces the string followed by one or more exact copies of the motif.



	A	B	C	D	E	F	G	H	I	J	K	L
1		TR-detect	Microsate	Minisatell	Satellites	PTRs	ATRs	Mismatch	Indels	Threshold	Heuristics	Not Micro
2	Sputnik &	1	1	0	0	1	1	1	1	1	0	0
3	TRF	1	1	1	1	0	1	1	1	1	1	1
4	SSRIT	1	1	0	0	1	0	0	0	0	0	0
5	ComplexT	1	0	1	1	0	0	0	0	0	0	0
6	TROLL	1	1	0	0	1	0	0	0	0	0	0
7	ATRHunte	1	1	1	1	0	1	1	1	1	1	1
8	Not name	1	1	1	0	1	1	1	0	1	0	0
9	Misa	1	1	0	0	1	1	1	0	1	0	0
10	Mreps	1	1	1	1	0	1	1	0	0	1	1
11	STRING	1	1	1	1	1	1	1	1	1	1	1
12	W-SSRF	1	1	1	0	1	0	0	0	0	0	0
13	ExTRS	1	1	1	1	1	1	1	0	1	0	0
14	STAR	1	1	0	0	0	1	1	1	1	0	0
15	MsatFinde	1	1	0	0	1	0	0	0	0	0	0
16	FireμSat1ŧ	1	1	0	0	1	1	1	1	1	0	0
17	Phobos	1	1	1	1	1	1	1	1	1	0	0
18	SSRscanne	1	1	0	0	1	0	0	0	0	0	0
19	TandemSV	1	1	1	1	0	1	1	0	1	0	0
20	IMex	1	1	0	0	1	1	1	1	1	0	0
21	SciRoko	1	1	0	0	1	1	1	1	1	0	0
22	Msatcomr	1	1	0	0	1	0	0	0	0	0	0
23	T-REKS	1	1	0	0	1	1	1	1	1	0	0
24	BwTRS	1	1	1	0	1	1	1	0	1	0	0
25	QDD1&2	1	1	0	0	1	0	0	0	0	0	0
26	TRStalker	1	1	1	1	1	1	1	1	1	1	0
27	Inverter	1	1	1	1	1	0	0	0	0	0	0

Figure 3.1: The TRD lattice source-data.

F *ATRs* (Approximate Tandem Repeats)
An *ATR* is a string of nucleotides characterized by a certain motif that introduces the string followed by one or more adjacent "copies" of the motif. In the case of ATRs, at least one motif copy will not be exact. Copies may include mismatches and/or indels — defined hereafter.

G *Mismatches*
A *Mismatch* is the replacement of a nucleotide in a PTRE with another nucleotide. The result is an ATRE that can be considered an approximate "copy" of the original PTRE.

H *Indels*
When two sequences are compared then an insertion in one sequence implies a deletion in the other. Therefore in Benson [1999], insertions and deletions are together referred to as *Indels*. An insertion in a PTRE results in an ATRE whose length is longer than that of the original PTRE. A deletion from a PTRE results in an ATRE that is shorter than the original PTRE. When using the word *indel*, reference is made to algorithms that have the capability to detect TRs containing both insertions and deletions.

I *Threshold/Confidence*
A *Threshold* or a *Confidence* is a value (usually an integer value) in a relational Boolean expression. Such expressions evaluate to either true or false and are used in algorithms to change the manner in which the algorithm executes, depending on whether or not the expression evaluates to true [McGraw-Hill and Parker, 2003]. In the context of TR-detection, a threshold is typically used to control the extent of the search. Thus, whenever too many indels or mismatches are identified in a substring that is potentially a TR, then the algorithm will no longer regard the substring as a potential TR and will aim to detect another TR.

J *Heuristics*
Generally speaking, a *Heuristic* is a "rule of thumb," or a good guide to follow when making decisions. Pearl [1984] defines heuristics in more precise terms, as strategies using readily accessible, though loosely applicable, information to control problem solving within either human beings or machines. In the context of computer science, heuristics (often statistics-dependent) aim to solve a problem more quickly than a classical algorithmic method might have. Heuristics are usually used if the search space is large and/or complicated. Thus by using heuristics, speed is often gained at the cost of precision. Heuristics can also find a non-exact solution when classical methods do not find any solution at all.

K *Not Micro*
The attribute *Not Micro* implies that although an algorithm has the ability to search for microsatellites the focus of the algorithm developers was specifically to search for TR minisatellites and/or satellites.

L The remaining columns in the table give the year of publication of the various algorithms. Note, however, that these attributes were not used to construct the formal concept lattices that are discussed below.

The line diagram of the concept lattice derived from this data is shown in Figure 3.2. The diagram gives an alternative view of how attributes and TRDs in Figure 3.1 map to one another.

Some of the information is quite easy to infer intuitively. For example, the top concept has an intent of {*TR- detectors*} and an extent consisting of all the objects listed in the rows of Figure 3.1 – i.e. all TRDs have the attribute *TR Detector*.

As another example, consider the concept with own object `ComplexTR`. This concept has an extent consisting of several other TRDs listed as own objects in concepts below it, and an intent of

{*Satellites, Minisatellites, TR-detectors*}

Further inspection shows that the own object at this concept, `ComplexTR`, is the only TRD in the data that does not have *Microsatellites* in its intent — i.e. it is the only TRD in the data that cannot detect microsatellites. Further such inferences could be made from this line diagram

However, because the diagram is rather cluttered, several alternative diagrams are given below that rely on smaller subsets of the attributes in Figure 3.1. Further discussion is with respect to these resulting less cluttered diagrams. In each case, however, the TRD `ComplexTR` will be excluded from the data.

3.2.3 A lattice based on motif-length classification

In this subsection Figure 3.3 is considered. This line diagram depicts all the considered TRDs in the dataset in relation to attributes that focus on motif length, namely: {*Microsatellites, Minisatellites, Satellites, PTRs, ATRs, Not Micro*}. Because fewer attributes are used than for Figure 3.2, it is simpler to cluster algorithms in terms of their capability to detect *Microsatellites* and/or *Minisatellites* and/or *Satellites*. It also displays whether the user has an option to select *PTRs* only.

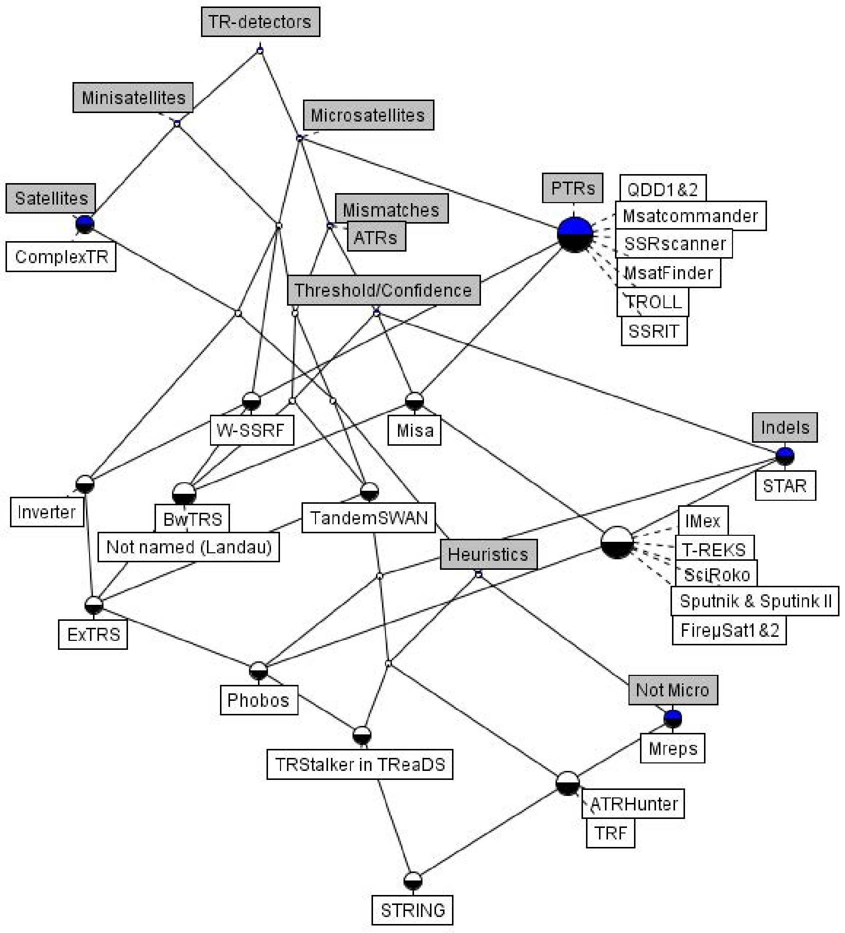


Figure 3.2: A line diagram built with attributes B to L listed in Figure 3.1.

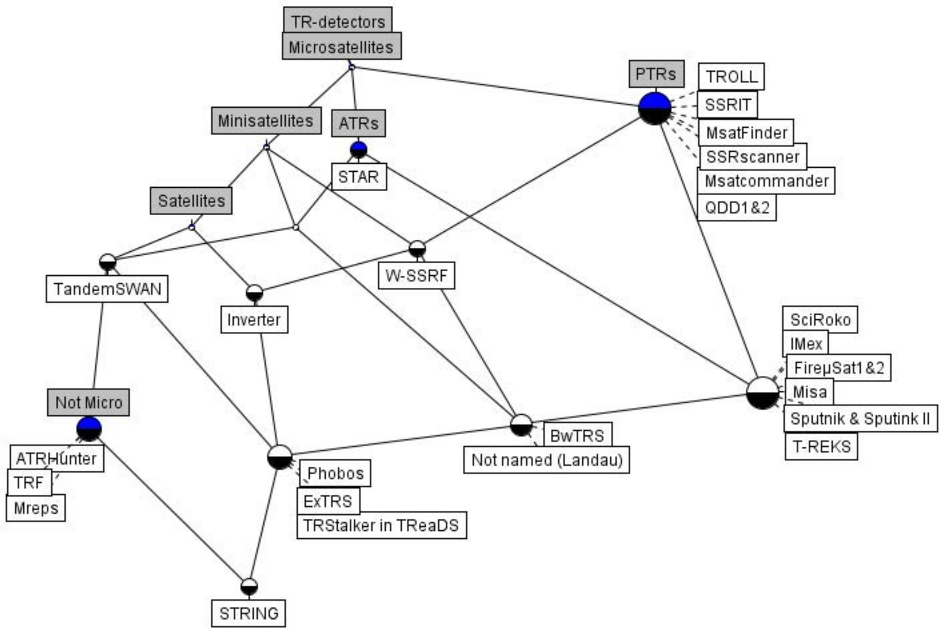


Figure 3.3: A line diagram of data in Figure 3.1 with attributes focussing on motif length, namely: {*TR-detectors*, *Microsatellites*, *Minisatellites*, *Satellites*, *PTRs*, *ATRs*, *Not Micro*}.

Just below the top concept (with own attribute *Microsatellite*) are two concepts whose own attributes are *Minisatellite* and *PTRs* respectively. The latter concept's intent is therefore:

{*Microsatellite, PTRs, TR-detectors*}

(i.e. the set consisting of its own attribute as well as attribute labels above it). This concept has six *own objects*, namely

{`QDD`, `MsatCommander`, `SSRscanner`, `TROLL`, `SSRIT`, `MsatFinder`}.

The extent of the same concept is the set consisting of its own objects as well as object labels below it, i.e.:

{`QDD`, `MsatCommander`, `SSRscanner`, `TROLL`, `SSRIT`, `MsatFinder`, `T-REKS`, `SciRoko`, `IMex`, `Fire`μ`Sat`, `Misa`, `W-SSRF`, `Inverter`, `Sputnik`, `Sputnik II`, `BwTRS`, `Landau-algorithm`, `TRStalker` in `TReads`, `Phobos`, `ExTRS`, `TRStalker`, `STRING`}.

The extent is thus the set of all the objects that have exactly and only the intent attributes in common, namely, they all detect *Microsatellites* and *PTRs*. Note that TRDs in the extent may have additional attributes not in the concept's intent. For example, the `Phobos` algorithm also has the attributes *Minisatellites*, *Satellites* and *ATRs* in its intent. This can be seen by collecting together all attributes encountered by following all paths upwards from the concept with `Phobos` as an own object.

As another example, consider the concept with own attribute *Not Micro* and own objects *TRF*, `mreps` and `ATRHunter`. Its intent is

{*Not Micro*, *Satellites*, *Minisatellites*, *Microsatellites*, *ATRs, TR-detectors*}

and its extent is

{`TRF`, `mreps`, `ATRHunter`, `STRING`}

The detection of *Microsatellites* is simpler than that of *Minisatellites* and *Satellites* as the length of the repetitive motif is smaller. Furthermore, when searching for *Minisatellites* or *Satellites* in a given DNA string, in practice the number of motif errors allowed in ATREs detected is often significantly larger than when searching for *Microsatellites*. This is because, for a given error tolerance based on some percentage of an ATRE's length, the number of actual errors allowed will be higher in the case of *Minisatellites* and *Satellites* compared to those resulting from *Microsatellites* with their shorter length motifs. Because of these factors, fewer algorithms cater for detecting *Minisatellites* and *Satellites* than for detecting *Microsatellites*. This can be seen in Figure 3.3.

There are three concepts in Figure 3.3 that do not have *Minisatellite* or *Satellite* in their intents, namely:

1. the concept whose intent is {*Microsatellite*, *PTRs*, *TR-detectors*} — i.e. the concept that has *PTRs* as an own attribute;
2. the concept whose intent is {*Microsatellite*, *ATRs*, *TR-detectors*} — i.e. the concept that has *ATRs* as an own attribute and `STAR` as own object
3. the concept below these two with the intent {*PTRs*, *ATRs*, *Microsatellites*, *TR-detectors*} but no own attribute.

Note that the attributes *Minisatellites* and *Satellites* are not in the intent of any of these three concepts. Thus, the algorithms that are own objects of these three concepts can be said to search for *Microsatellites* only — i.e. they are not capable of searching for *Minisatellites* or *Satellites*. There are a total of 13 such algorithms: 6 that search for *Microsatellites* that are *PTRs* only; 1 that searches for *Microsatellites* that are *ATRs* only (namely `STAR`); and 6 that search for *Microsatellites* that are either *PTRs* or *ATRs*.

The concept in Figure 3.3 with own object `W-SSRF` has an intent {*Minisatellites, Microsatellites, PTRs*}. It is the only concept whose intent has *Minisatellites* but excludes *Satellites* — i.e. it is the only concept representing algorithms that search for *Minisatellites* or *Microsatellites*, but not for *Satellites*. These algorithms constitute the concept's extent, namely {`Inverter, TRStalker, Phobos, ExTRs STRING, BwTRS, Landau-algorithm, W-SSRF`}.

The concept with own attribute *Satellites* has an extent with nine objects[7], namely
{`TandemSWAN`, `TRF`, `mreps`, `ATRHunter`, `TRStalker`, `STRING`, `Phobos`, `ExTRS`, `Inverter`}. These are the only algorithms that detect *Satellites*.

Although it is not always an objective of algorithms searching for *Satellites* or *Minisatellites* to search for *Microsatellites* too, most of the minisatellite and satellite detecting algorithms are capable of detecting some *Microsatellites* as well. Often, however, they do not do so as accurately as algorithms that focus on the detection of *Microsatellites* only.

Though algorithms in Figure 3.2 or 3.3 have been characterised as detecting *Microsatellites* (*Minisatellites* or *Satellites*, respectively) this should not be taken to mean that the algorithms necessarily detect the *complete* set of *Microsatellites*

[7]Recall that `FireSat` is not part of the discussion.

(*Minisatellites* or *Satellites*, respectively). For example, `W-SSRF` detects *Minisatellites* whose motif lengths are less than 10; `Inverter` detects *Satellites* whose motif lengths are limited to a range of less than 200.

It is clear from Figure 3.3 that `STAR`, the own object of the concept with the own attribute *ATRs*, is the only microsatellite detection algorithm that does not detect *PTRs per se*. Note, however, that `STAR` does detect *Indels*. The other algorithms that detect *Indels* and not *PTRs per se* are `mreps`, `ATRHunter` and `TRF`. These three algorithms have also been designed with the objective of detecting repeats with a repetitive motif (PTRE) length longer than 6.

3.2.4 A lattice based on error types and tolerances

The line diagram in Figure 3.4 relies on a different subset of attributes to the line diagram in Figure 3.3, namely:

{*TR-detectors*, *PTRs*, *ATRs*, *not Micro*, *Mismatches*, *Indels*, *Heuristics*, *Threshold/Confidence*}

Thus, the diagram distinguishes algorithms that identify *PTRs* only from those that identify both *PTRs* and *ATRs*. It also distinguishes between algorithms whose ATREs contain only *Mismatches* from those that tolerate both *Mismatches* and *Indels*. Finally, it indicates whether *Heuristics* and/or *Threshold/Confidence* values are used to determine *ATRs*.

By exploring the line diagram, one is easily able to answer questions such as the following in regard to the attributes of the algorithms under consideration:

- Does an algorithm only detect *PTRs*?
- Does an algorithm allow the user to detect *ATRs*?
- Does an algorithm allow the user to detect *Mismatches*?
- Does an algorithm allow the user to detect *Insertions* and *Deletions*?
- Does an algorithm rely on *Heuristics*?
- Does an algorithm rely on *Threshold / Confidence* values?

Of course, many more complex questions can also be answered by analysing such line diagrams. For example, one can infer information about relationships between the attributes of algorithms. Consider, for example, the concept in Figure 3.4 that has both *ATRs* and *Mismatches* as own attributes. This indicates that, for all the TRDs investigated here, it holds that whenever an algorithm detects *ATRs* it also has the ability to detect *Mismatches* too.

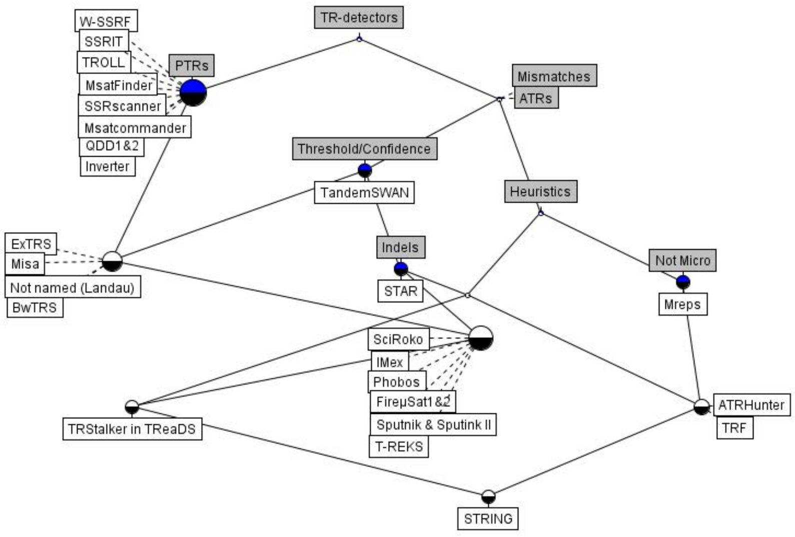


Figure 3.4: A line diagram of data in Figure 3.1 with attributes focussing on error types and tolerances, namely: {*TR-detectors, PTRs, ATRs, not Micro, Mismatches, Indels, Heuristics, Threshold/Confidence*}.

A number of algorithms search for repeats with *Mismatches* only. Figure 3.4 shows that `ExTRS`, `Misa`, the `Landau-algorithm`, `BwTRS`, `TandemSwan` and `mreps` allow for *Mismatches* only. This is the list of all algorithm labels below the *Mismatches* label in the figure, but that are not also below the *Indels* label.

To allow for *Mismatches* as well as *Indels* is a harder computational problem than simply allowing for *Mismatches* only. `STAR`, `Phobos`, `Sputnik`, FireμSat, `IMex`, `SciRoko`, `T-Reks`, `TRStalker`, `ATRHunter`, `STRING` and `TRF` allow for both *Mismatches* and *Indels* in TREs of TRs. These algorithms fall below the *Indels* label in the figure.

Whenever an algorithm allows for *ATREs* then, to ensure that only relevant repeats are reported, either *Heuristics* or a *Threshold/Confidence* value should be used. Figure 3.4 shows that certain algorithms use *Heuristics*, others use a *Threshold/Confidence* value, and a number of algorithms allow for the use of both *Heuristics* and a *Threshold/Confidence* values.

Krishan and Tang [2004] argue that *Heuristics* are incomplete in the sense that researchers who use them or who use probabilistic definitions to give a more abstract definition of TRs are thereby narrowing down the set of potential TRs in order to enhance runtime performance at the cost of accuracy. It will be seen that `FireSat` implements counting finite automata during the detection of repeats. `FireSat` provides the possibility for users to specify penalties and threshold values. Thus the statistical significance of results should be tuned using penalties and threshold values. The only heuristic that `FireSat` uses is to give mismatches priority above deletions and deletions in turn priority above insertions. Switches can however, be set to alter the mutation precedence order.

Here are further examples of inferences that can be made from the line diagram in Figure 3.4, sometimes also needing to refer to the previous line diagrams:

- The figure highlights the fact that *ATRs* and *Mismatches* are equivalent attributes in the sense that each algorithm that has one of these attributes will also have the other.
- All the algorithms listed here are able to detect a subset of `Microsatellites`. However `mreps`, the own object of the concept with the own attribute *Not Micro* (see Figure 3.3), was not developed for *Microsatellite* detection. The same holds for `ATRHunter`, `TRF` and `STRING` objects that form part of the extent of the concept with the own attribute *Not Micro*.
- `mreps` does not rely on *Threshold / Confidence* testing for TR-detection, but only on *Heuristics*.
- `TRF`, `ATRHunter` and `STRING` cater for *Indels*, but `mreps` does not.
- `TandemSWAN`, the own object of the concept with the own attribute *Threshold/Confidence*, does not use *Heuristics* during TR-detection at all.
- Three of the TR-detection algorithms catering for *Minisatellite* and *Satellite* detection implement both *Heuristics* and *Threshold/Confidence* values — `TRF`, `ATRHunter` and `STRING`.
- `INVERTER` is the only algorithm that caters for *Satellites* and *Minisatellites* that are only `PTRs` — In Figure 3.4, `INVERTER` is an own object of the concept with the own attribute *PTRs*.
- `W-SSRF` detects minisatellites consisting of *PTRs* only and has been designed to cater for microsatellite detection too.
- The concept in Figure 3.4 with *PTRs* as its own attribute has eight own objects, namely, `SSR-scanner`, `W-SSRF`, `TROLL`, `SSRIT`, `MsatFinder`, `Msatcommander`, `QDD` and `Inverter`. These algorithms search for *PTRs* only.
- From Figure 3.4 it is clear that several algorithms do not have the ability to detect *PTRs*. These are TRDs in the extents of concepts that do not include the *PTRs* attribute in their intents. For example, it can easily be

seen that `mreps` does not detect *PTRs* because the concept with own object `mreps` does not have *PTRs* in its intent. Similar reasoning applies to `STAR`, to `ATRHunter`, `TRF`, etc.

3.2.5 TRDs that search only for *PTRs*

The task of searching for *PTRs* only is relatively simple. It can be done in less than 40 lines of `Matlab/Octave` code[8].

Lim et al. [2012] assert that *PTR* detection provides a good basis for studying the intrinsic performance of repeat detecting tools since, in their opinion, *PTRs* are precise and thus not subject to varying interpretations. In De Ridder et al. [2013] I took a different view from Lim et al. [2012] with respect to the relevance of PTR detection for software comparison. Nevertheless, I thought it would be of interest to compare a number of software implementations in terms of PTR detection only. I report briefly below on running `Fire`μ`Sat`$_2$ (a predecessor of `FireSat`), `Inverter`, `T-reks`, in all cases searching only for microsatellite PTRs. The results were compared against PTR detection code implemented in `Matlab` mentioned above.

In Table 3.2, the findings of the trial runs are reproduced. The table gives the number of detected non-overlapping PTRs consisting of 20 or more base pairs, where $1 \leq |motif| \leq 6$ is the length of the motif[9]. The data used was precisely that which was originally used and reported on by Lim et al. [2012][10]. Table 3.2 reports on the number of PTRs detected as well as the runtime of the algorithms. From Table 3.2 and verification via data inspection it is clear that `ptrfind.m` (the `Matlab` script) and `Fire`μ`Sat`$_2$ are currently the most accurate and consistent approaches for *PTR* detection. `Fire`μ`Sat`$_2$ performs considerably faster than `Inverter`. `T-reks` turns out to be the fastest, but only detects about 66% of the PTRs found by `Fire`μ`Sat`$_2$ and the `Matlab` script.

Note that results generated by `Fire`μ`Sat`$_2$ as well as those generated by `T-reks` were post-processed to achieve only the non-overlapping PTR detections. The

[8]`Matlab` (`Octave` is `Matlab`'s freeware counterpart.) is a well-known interpreter. Downloadable `Matlab/Octave` code to find non-overlapping *PTRs* may be found at `www.dna-algo.co.za`. Scripts in these environments usually run more slowly than compiled executables. One could therefore expect a considerable speed-up if the code were to be rewritten as optimised C++ code (Andrews [2012]).

[9]The molecular biologist community expressed a need to analyse non-overlapping microsatellites as defined by Zhou et al. [2009] with motif-lengths in the range of $1 \leq |motif| \leq 6$. `Fire`μ`Sat`$_2$ has been extended to detect such microsatellites. Note that if $|motif| = 6$ then a maximum motif error of 2 ($\epsilon_{max} = 2$) is allowed.

[10]The `Saccharomyces_cerevisiae_S288c_chromosome.fasta` data is available as one contiguous file from `http://www.dna-algo.co.za`.

Table 3.2: Non-overlapping PTRs detected of length ≥ 20 for $1 \leq |motif| \leq 6$ on the data originally used by Lim et al. [2012].

	`ptrfind.m`	FireμSat$_2$	`T-REKS`	`Inverter`
PTRs detected	596	596	393	125
Time	>20min	05min20s	03min19	>20min

results generated by `Inverter` needed post-processing to extract only PTR detections for motif-lengths in the range $1 \leq |motif| \leq 6$.

As an aside, note that algorithms that can only detect perfect microsatellites often provide the additional functionality to facilitate the design of primers[11]. For example, `QDD2` and `Msatcommander` provide such a user option.

3.2.6 Selecting TRDs to benchmark `FireSat`

The lattices presented in Figures 3.3 and 3.4 facilitated the selection of rival algorithms to be used during trial runs of `FireSat`. The outcome of these trial runs will be reported in Chapter 6.

In considering length of the motif characteristics in respect of rival algorithms to be tested during trial runs, it was decided that these algorithms should be able to detect *Satellites*, *Minisatellites* as well as *Microsatellites*. Figure 3.3 showed that there are nine algorithms that can search for *Satellites*. These are now briefly considered.

1. `TRF` and `ATRHunter`: It was decided that *ATRs* must be tolerated by the algorithms selected for trial runs. In both Figure 3.3 and Figure 3.4, `TRF` and `ATRHunter` share the same intent, which includes the attribute *ATRs*. Since they are therefore similar algorithms, in the interests of economy it was decided to include only one of them for benchmarking purposes. Because `TRF` is commonly included for benchmarking purposes [Lim et al., 2012, Rivals, 2004, Schaper et al., 2015], it was decided to use `TRF` in the benchmarking to be done in this present study.

2. `mreps`: Although `mreps` does not allow for insertion and/or deletions (see Figure 3.4), it detects *Satellites*. (See Figure 3.3.) Previous trial runs [Masombuka et al., 2010] indicated that `mreps` is fast and relatively accurate during searches for *ATRs*. Therefore `mreps` was also included as a rival algorithm to be used here for benchmarking purposes.

[11]A primer is a strand of DNA that serves as a starting point for DNA synthesis — a new deoxyribonucleotide can only be added by DNA polymerases to a preformed primer strand that is hydrogen-bonded to the template [Cooper, 2000].

3. `Inverter`: This algorithm seemed to be of interest because it was published relatively recently (in 2012). `Inverter` searches for *PTR*s only. Note that only `Phobos` has a switch to search for *PTRs* only—`mreps` and `TRF` do not have that functionality. `Inverter` is, however, unavailable from the web and for that reason it was not included in trial runs.

4. `TRStalker`: This is a TRD embedded within `TReaDS` (*Tandem repeats discovery service*). `TreaDS` is a TR meta search engine that forwards user requests to selected TRDs that include `ATRHunter`, `mreps`, `TandemSWAM`, `TRF` or `TRStalker`. The detected TRs can be combined/merged to provide a so called TR *global view* [Pellegrini et al., 2012]. `TRStalker` is not a stand alone package and, for this reason, it will not be further considered for the purposes of selecting rival software.

5. The `TandemSWAN` web page was updated in January 2006. It offers a facility to process both `TRF` and `mreps` results in terms of statistical significance. Similarly to `mreps`, `TandemSWAM` only makes provision for detecting *Mismatches*. Like `Phobos` `TandemSWAM` uses a confidence measure during TR detection. (In Figure 3.4 `TandemSWAM` is the own object of the concept with the own attribute *Threshold/Confidence*.)

6. Note that `Phobos` caters for *Mismatches* and *Indels*. `Phobos` was updated in 2010. In Figure 3.4, the intent of the concept with `Phobos` as own object include *Threshold/Confidence*. In the interest of economy `TandemSWAM` will be omitted for benchmarking purposes in Chapter 9 while `Phobos` will be included.

7. The `ExTRS` software is not available from the web. Consequently it was decided to exclude `ExTRS` from trial runs. Like `TandemSWAN`, `ExTRS` was published more than 10 years ago and may be considered relatively old in the current context.

8. Similarly `STRING` is also relatively old (published in 2003) and is not available from the URL indicated in Parisi et al. [2003][12].

3.2.7 TRD computational techniques

The type of computational techniques that are used in the respective software packages differ. They include alignment (`TRF`, `ATRHunter`), periodicity (`mreps`), clustering (`T-Reks`) and compression algorithms (`STAR`). Details about these techniques may be found in relevant publications about the respective packages.

[12] `www.caspur.it/STRING/`

To the best of my knowledge, `FORRepeats` is the only algorithm, other than `FireSat`, that relies on finite automata to detect tandem repeats (focussing on minisatellite detection). For this reason, `FORRepeats` is discussed in the next section, Section 3.3. Note that `FORRepeats` is not available from the web.

3.3 FORRepeats

`FORRepeats` implements FAs to detect TRs. However, other than relying on finite automata to detect TRs, `FORRepeats` and `FireSat` do not have much else in common. In its first step, `FORRepeats` detects PTREs in large sequences. In its second step, TREs (allowing errors of the detected PTREs) are computed during a pairwise comparison between two extended *PTRs*. The details of the extension of exact repeats and the comparison thereof can be found in Lefebvre et al. [2003].

`FORRepeats` uses a heuristic method that is based on an FA called a factor oracle. The factor oracle data structure was introduced by Allauzen et al. [1999]. A factor oracle automaton is acyclic. It recognizes the so-called factors of a word p.

x is defined as a factor of a word p if and only if p can be written as uxv with $u, v \in \Sigma^*$. A factor x is a prefix of p if $p = xu$; x is a suffix of p if $p = ux$ where $u \in \Sigma^*$. x is a proper factor of p if x is a factor of p and is distinct both from p and from the empty word Λ.

The factor oracle automaton of a word p of length m, denoted by *Oracle(p)*, is a deterministic acyclic finite automaton $(Q, q_0, \Sigma, \mathcal{F}, \delta)$ where $Q = 0, 1, ..., m$ is the set of states; $q_0 = 0$ is the starting state; $\mathcal{F} \subseteq Q$ is the set of final states and δ is the transition function. The factor oracle of a word p of length m has the following properties:

- it has exactly $m + 1$ states,
- its number of transitions is in the range $[m, 2m - 1]$ and,
- it recognizes at least all the factors of p and possibly more words[13].

There is a bijection between the states of the oracle and the $m + 1$ prefixes of p (including the empty one). Each transition leading to a state i is labelled by $p[i]$, the i^{th} character of string p. Two kinds of transitions are distinguished, namely:

- Internal transitions: transitions from state i to state $i + 1$ and,

[13]Characterising the set of words recognized by the factor oracle is a research matter for researchers in the FA community.

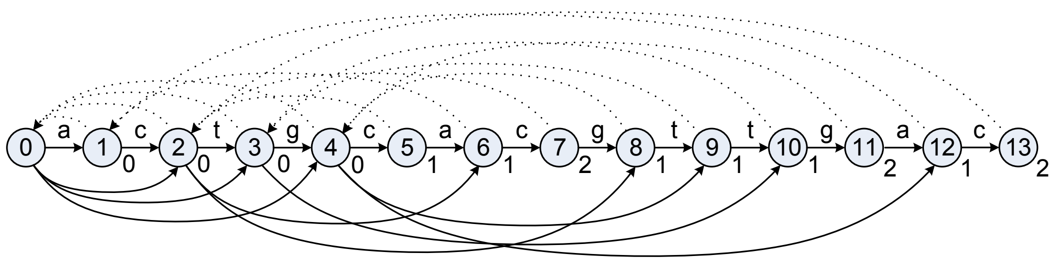


Figure 3.5: The oracle for the word ACTGCACGTTGAC.

- External transitions: transitions from state j to state i where $(j - i) > 1$.

For the above, the following must hold: $0 \leq i < j \leq m$. There are exactly m internal transitions. To store the oracle one needs to store only the word p and at most $m - 1$ external transitions without their labels. The factor oracle is a structure that is economical in terms of memory and runtime. The memory used is approximately 10.5 times the length of the sequence — the structure is linear in terms of memory. The construction of the structure is also linear in terms of runtime [Lefebvre et al., 2003]. Figure 3.5 gives an example of a factor oracle for the word `actgcacgttgac`.

Lefebvre et al. [2003] conducted experiments to determine the accuracy of `FORRepeats`. These experiments indicated that `FORRepeats` performs better at detecting TRs with a PTRE-length longer than 20 base pairs. When the repetitive motif is very short a large number of exact repeats can be found that will be time consuming to extend and may result in overlapping.

This finding about the performance of `FORRepeats` (namely that it is more accurate in detecting TRs with a longer PTRE-length than TRs with a shorter PTRE-length) is comprehensible when one considers its calculation technique. Firstly, PTREs are found. The search is extended to the left and right of these PTRs. However, the search is now for *ATRs* that comply with certain threshold values.

The proposers of `FORRepeats` present it as well-suited for the detection both of *Minisatellites* and of duplicated genes.

In contrast to `FORRepeats`, it will be seen that the algorithm proposed here, `FireSat`, constructs FAs and decorates the already constructed machines while iterating through the genetic input sequence. If allowable final states are reached during runtime and all threshold values (see Sections 2.1.2 and 2.1.4) are adhered to, then a TR is reported by `FireSat`. Thus, apart from also employing FAs to detect repeats, the approach of `FireSat` differs significantly from that of `FORRepeats`.

To provide some context of the algorithm proposed in this thesis, a brief discussion of the three predecessor algorithms proposed in De Ridder [2010] will follow in the next section.

3.4 `Fire`μ`Sat` — the precursor to `FireSat`

In De Ridder [2010] three slightly different algorithms to detect *Microsatellites* in DNA were proposed — `Fire`μ`Sat`$_1$, `Fire`μ`Sat`$_2$ and `Fire`μ`Sat`$_3$. Results were reported of tests run on the three respective algorithms. The algorithms were implemented in C++.

In terms of output, `Fire`μ`Sat`$_1$ and `Fire`μ`Sat`$_2$ reported exactly the same *Microsatellites*. Furthermore, they both report overlapping TRs. This feature could be regarded as over-reporting, for the following reason:

`Fire`μ`Sat`$_1$ and `Fire`μ`Sat`$_2$ will regard the string `acg acg acg acg acg` as a PTR with the introductory motif `acg` that is repeated 5 times. However, the string will also be regarded as containing a TR that has the introductory motif `cga` that is repeated 4 times (possibly treating `cg` as an ATRE deletion, depending on the tolerance settings being used). Finally, the same string may be regarded as a TR with an introductory motif of `gac` that is repeated 4 times. In addition, `acgacgacgacgacg` will be reported by `Fire`μ`Sat`$_1$ and `Fire`μ`Sat`$_2$ as a TR with a $|\text{PTRE}| = 6$ consisting of 3 TREs of which the last TRE is an ATRE containing 3 deletions.

The output of `Fire`μ`Sat`$_3$ differs from that of `Fire`μ`Sat`$_1$ and `Fire`μ`Sat`$_2$ in that it does not detect overlapping TRs *of the same* motif length. `Fire`μ`Sat`$_3$ does, however, detect all the TR regions detected by the other two algorithms.

The `Fire`μ`Sat` packages beat other competitive packages reported on in the literature in terms of runtime. De Ridder et al. [2013] report on details of this statement. These packages included `mreps` [Kolpakov et al., 2003], `IMEX` [Mudunuri and Nagarajaram, 2007], `STAR` [Rivals, 2004] and `TRF` [Benson, 1999].

In order for `Fire`μ`Sat`$_1$ and `Fire`μ`Sat`$_2$ to detect overlapping TRs as explained above, they traverse the genetic input string for each motif length n, n times. As a consequence, they also detect the longest possible TR, whereas other approaches might regard that particular region of the input string as embedding more than one TR, none of which are as long as the single TR identified by `Fire`μ`Sat`$_1$ and `Fire`μ`Sat`$_2$. `Fire`μ`Sat`$_3$ traverses the input string only once, but as a consequence, it does not always report the longest possible repeat to be detected. Also as a consequence, `Fire`μ`Sat`$_3$ is faster than both `Fire`μ`Sat`$_1$ and `Fire`μ`Sat`$_2$.

Whether the traversal approach of `Fire`μ`Sat`$_1$ and `Fire`μ`Sat`$_2$ should be followed in preference to the traversal approach of `Fire`μ`Sat`$_3$ depends therefore on whatever is valued most in the application context: execution time or the detection of the longest possible repeat within a certain region of the input string.

`FireSat` will employ both traversal approaches. It will be seen in Chapter 5 that `FireSat` relies on some of the principles introduced by FireμSat$_1$, FireμSat$_2$ and FireμSat$_3$. In the case of `FireSat`, a quicker runtime will always be at the cost of accuracy.

From the literature as well as from discussions with molecular biologists, a list of criteria was compiled to which algorithms should yield in order to search effectively for repeats. The compiled list has been presented in De Ridder et al. [2006b] and in De Ridder [2010]. Here, the original list of criteria is extended by three new criteria. The proposed criteria should be kept in mind during algorithm developments for TR detection.

3.5 Criteria for TRDs

The list of compiled criteria that will contribute to the successful development of software tools for the detection of repeats is presented below. The list constitutes criteria proposed by Benson [1999], a criterium suggested by Delgrange and Rivals [2004] and two of our own criteria. This list has already been published in De Ridder et al. [2006b] and De Ridder [2010]. Although these publications deal with *Microsatellites*, the same criteria hold for TRs in general. Benson [1999] suggests the following criteria that should be pursued during the development of an effective (in terms of runtime and memory complexity) TR detection algorithm:

1. The avoidance of full scale alignment matrix computations in the case of alignment algorithms.

2. No *a priori* knowledge should be required pertaining to the pattern, pattern size or number of copies of the TR.

3. No restrictions should exist regarding the size of the repeats that can be detected.

4. Percentage differences between adjacent copies should be used and *Mismatches* and *Indels* should be treated separately.

5. A consensus pattern for the smallest repetitive unit in the TR should be determined.

Note that in addition to Benson's criteria, the criterium suggested by Delgrange and Rivals [2004] is also endorsed, namely:

6. An exact algorithm should systematically detect significant TRs in a way that is independent of the motif.

De Ridder et al. [2006b] and De Ridder [2010] suggested two further criteria:

7. An algorithm that detects *Microsatellites* should be *flexible* in terms of penalties awarded to *Indels* and *Mismatches*. This criterium clearly holds for TR detection in general.

8. Software to detect *Microsatellites* should be *useable*, specifically in terms of output. By this we mean that analytically, biologically and statistically relevant output should be provided to the user. Furthermore, we suggest a hierarchical output that will enable the user to obtain the most relevant data easily. As in the case of [7.] this criterium holds for TR detection in general.

From studying the literature subsequent to 2010 it is clear that the originally presented criteria should be extended:

9. Software should provide an option to either report compound repeats, thus overlapping repeats, or to filter overlapping repeats. Schaper et al. [2015] point out that not all overlapping detections are redundant. Furthermore Schaper et al. [2015] define overlaps in two ways: Firstly, overlaps are two or more TR sequences from the same genetic input string that have in common some characters in the same genetic input string positions. The second definition refers to an overlapping pair of TRs that have a common ancestry of at least one pair of characters in alignments of multiple TR units for both TRs. The package *Tandem repeat annotation library* (`TRAL`) was proposed by Schaper et al. [2015] to filter identified clusters[14] in such a manner that the best TR representative from a cluster is output in line with user specifications.

10. Software should provide an option to detect *PTRs* only, or should indicate that it is not possible to detect *PTRs* with a certain package.

11. A length filter should be provided to avoid the output of redundant data[15]. This implies that users should have the option of indicating the minimum number of TREs that must occur in a reported TR.

[14]Within the genetic sequence context a cluster is a potential significant stretch of DNA.

[15]Redundant data within this context refers to TRs that users consider to be insignificant in their context. From Kolpakov et al. [2003] it is clear that longer TRs are generally considered to be of more significance than shorter ones.

The focus of `FireSat` is at present merely on detecting TRs accurately. To adapt `FireSat` in such a way that it complies to the above proposed criteria is left as a future research initiative.

3.6 Conclusion

This chapter provided references to existing literature, some of which I have authored or co-authored. References were also given to relevant software. An overview of algorithms detecting repeats was provided in terms of line diagrams for concept lattices. Rival algorithms against which `FireSat` (proposed in Chapters 6, 7 and 8 of this thesis) will be evaluated in Chapter 9 of this thesis were identified. Finally, a list of criteria was provided that will serve as a guideline for future development of `FireSat`.

In the next chapter, Chapter 4, definitions and theoretical concepts relevant to the algorithmic development of `FireSat` are considered.

Chapter References

C. Abajian. Sputnik: Source code published online. Online: http://espressosoftware.com/sputnik/sputnik_source.html, 1994.

C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: A new structure for pattern matching. *SOFSEM, Theory and Practice of Informatics, Lecture Notes in Computer Science*, 1725:295 – 310, 1999.

T. Andrews. Computation Time Comparison Between Matlab and C++ Using Launch Windows. 2012.

M. Anisimova, J. Pečerska, and E. Schaper. Statistical approaches to detecting and analyzing tandem repeats in genomic sequences. *Frontiers in Bioengineering and Biotechnology*, 3(2015):31, 2015. URL http://doi.org/10.3389/fbioe.2015.00031.

T. Anwar and A. U. Khan. SSRscanner: a program for reporting distribution and exact location of simple sequence repeats. *Bioinformation*, 1(3):89–91, 2006.

A. Archambault. Mining genomic data for tandem repeats. Online: http://qcbs.ca/wiki/bioinformatic_tools_to_detect_microsatellites_loci_from_genomic_data, 2012.

G. Benson. Tandem Repeats Finder. *Nucleic Acids Research*, 27(2):573 – 580, November 1999.

A. Biegert and J. Schoöding. De novo identification of highly diverged protein repeats by probabilistic consistency. *Bioinformatics*, 24(6), 2008.

V. Boeva, M. Regnier, D. Papatsenko, and V. Makeev. Short fuzzy tandem repeats in genomic sequences, identification, and possible role in regulation of gene expression. *Bioinformatics*, 22(6):676–684, 2006.

A. T. Castelo, W. Martins, and G. R. Gao. TROLL: Tandem Repeat Occurrence Locator. *Bioinformatics Applications Note*, 18(4):634–636, 2002.

G. M. Cooper. *The Cell: A Molecular Approach*. Sinauer Associates, Inc., 2nd ed. edition, 2000.

C. De Ridder. Flexible Finite automata-based algorithms for detecting microsatellites in DNA. Master's thesis, Department of Computer Science, University of Pretoria, July 2010.

C. De Ridder, D. G. Kourie, and B. W. Watson. FireμSat: an algorithm to detect microsatellites in DNA. In *Proceedings of the Prague Stringology Conference*, pages 137–150, August 2006a. ISBN 80-01-03533-6.

C. De Ridder, D. G. Kourie, and B. W. Watson. FireμSat: meeting the challenge of detecting microsatellites in DNA. In *Proceedings of the 2006 annual research conference of SAICSIT*, pages 247–256, 2006b. ISBN 1-59593-567-3.

C. De Ridder, P. V. Reyneke, B. W. Watson, O. Reva, and D. G. Kourie. Cascading Finite Automata for minisatellite detection. In *The 22nd Annual Symposium of the Pattern Recognition Association of South Africa*, pages 31–36, November 2011.

C. De Ridder, D.G. Kourie, B.W. Watson, T.R. Fourie, and P.V. Reyneke. Fine-tuning the search for microsatellites. *Journal of Discrete algorithms*, 20:21–37, 2013.

O. Delgrange and E. Rivals. STAR: an algorithm to search for tandem approximate repeats. *Bioinformatics*, 20(16): 2812–2820, June 2004.

L. Du, H. Zhou, and H. Yan. OMWSA:detection of DNA repeats using moving window spectral analysis. *Bioinformatics Applications Note*, 23(5):631 – 633, 2007.

B.C. Faircloth. msatcommander: detection of microsatellite repeat arrays and automated, locus-specific primer design. *Molecular Ecology Resources*, 8(1):92–94, 2008.

A.M. Hauth and D.A. Joseph. Beyond tandem repeats: Complex pattern structures and distant regions of similarity. *Bioinformatics*, 18(1):S31 – S37, July 2002.

J. Jorda and A.V. Kajava. T-REKS: identification of Tandem Repeats in sequences with a K-means based algorithm. *Bioinformatics*, 25:2632–2638, 2009.

R. Kofler and C. Sclötterer. Sciroko applications note. *Bioinformatics*, 23(13):1683–1685, 2007.

R. Kolpakov, G. Bana, and G. Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *PubMed Central Nucleic Acid Research* , 31(13):3672–3678, 2003.

D.G. Kourie and B.W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, Berlin Heidelberg, first edition, 2012.

A. Krishan and F. Tang. Exhaustive whole-genome tandem repeats search. *Bioinformatics*, 20(16):2702–2710, 2004.

S. Kurtz, J. Choudhuri, E. Ohlebush, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acid Research*, 29:4633–4642, 2001.

G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.

A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–326, 2003.

K. G. Lim, C. K. Kwoh, L. Y. Hsu, and A. Wirawan. Review of tandem repeat search tools: a systematic approach to evaluating algorithmic performance. *Briefings in Bioinformatics*, May 2012.

K. T. Masombuka, C. de Ridder, and D. G. Kourie. An investigation of software for minisatellite detection. In Fred Nicolls, editor, *Proceedings on the twenty-first PRASA symposium*, pages 171–176, November 2010. ISBN 978-0-7992-2470-2.

C. Mayer. Phobos: a tandem repeat search tool 2012, Phobos 3.3.11, 2006-2010. URL http://www.rub.de/ecoevo/cm/cm_phobos.htm.

McGraw-Hill and S.P. Parker. *McGraw-Hill Dictionary of Scientific and Technical Terms*. McGraw-Hill Companies, Inc, 6 edition, 2003.

E. Meglécz, C. Costedoat, V. Dubut, T. Malausa, and J.F. Martin. QDD: a user-friendly program to select microsatellite markers and design primers. *Bioinformatics*, 26(3):403 – 404, 2010.

S. B. Mudunuri and H. A. Nagarajaram. IMEx: Imperfect microsatellite extractor. *Bioinformatics*, 23(10):1181–1187, 2007.

A. M. Newman and J.B. Cooper. Xstream: A practical algorithm for identification and architecture modeling of tandem repeats in protein sequences. *BMC Bioinformatics*, 8, 2007. doi: 10.1186/1471-2105-8-382.

V. Parisi, V. De Fonzo, and F. Aluffi-Pontini. STRING. *Bioinformatics* , 19(14):1733–1738, 2003.

J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1984. ISBN 0-201-05594-5.

M. Pellegrini, M.E. Renda, and A. Vecchio. TRStalker: an efficient heuristic for finding fuzzy tandem repeats. *Bioinformatics*, 26:i358–i366, 2010.

M. Pellegrini, M.E. Renda, and A. Vecchio. Tandem repeats discovery service (TReaDS) applied to finding novel cis-acting factors in repeat expansion diseases. *BMC Bioinformatics*, 13(4):S3, 2012.

R. Pokrzywa and A. Polanski. BWTrs: A tool searching for tandem repeats in DNA sequences based on the Burrows-Wheeler transform . *Genomics*, 96:316 – 321, 2010.

E. Rivals. Eric Rivals's homepage. Online: http://www.lirmm.fr/ rivals/tete-en.html, 2004.

E. Schaper, A. V. Kajava, A. Hauser, and M. Anisimova. Repeat or not repeat? statistical validation of tandem repeat prediction in genomic sequences. *Nucleic Acids Research*, 2012. doi: 10.1093/nar/gks726. URL http://nar.oxfordjournals.org/content/early/2012/08/24/nar.gks726.abstract.

E. Schaper, A. Korsunsky, A. Messina, R. Murri, J. Pečerska, H. Stockinger, S. Zoller, I. Xenarios, and M. Anisimova. TRAL: Tandem repeat annotation library. *Bioinformtics*, 31(18):3051–3053, 2015.

D. Sharma, B. Issac, G.P. Raghava, and R. Ramaswamy. Spectral Repeat Finder (SRF): identification of repetitive sequences using Fourier transformation. *Bioinformatics*, 20(9):1405–1412, 2004.

D. Sokol, G. Benson, and J. Tojeira. Tandem repeats over the edit distance. *Bioinformatics*, 23(2):e30–e35, 2007. doi: 10.1093/bioinformatics/btl309. URL http://bioinformatics.oxfordjournals.org/content/23/2/e30.abstract.

V.B. Sreenu, G. Ranjitkumar, S. Swaminathan, S. Priya, B. Bose, M.N. Pavan, G. Thanu, J. Nagaraju, and H.A. Nagarajaram. MICAS: a fully automated web server for microsatellite extraction and analysis from prokaryote and viral genomic sequences. *Applied Bioinformatics Application Note*, 2(3), 2003.

R. Szklarczyk and J. Heringa. Tracking repeats using significance and transitivity. *Bioinformatics*, 20(suppl 1): i311–i317, 2004. doi: 10.1093/bioinformatics/bth911. URL http://bioinformatics.oxfordjournals.org/content/20/suppl_1/i311.abstract.

S. Temnykh, Lukashova A., S. Cartinhour, G. de Clerck, L. Lipovich, and S. Mccouch. Computational and experimental analysis of microsatellites in rice (oryza sativa l.): frequency, length variation, transposon associations, and genetic marker potential. *Genome research*, 11:1441 – 1452, 2001.

W. Thiel, R. Michalek, and A. Graner. Exploiting EST databases for the development and characterization of gene-derived SSR-markers in barley (Hordeum vulgare L.) 20. *Theoretical and applied genetics*, 106(3):411–422, 2003.

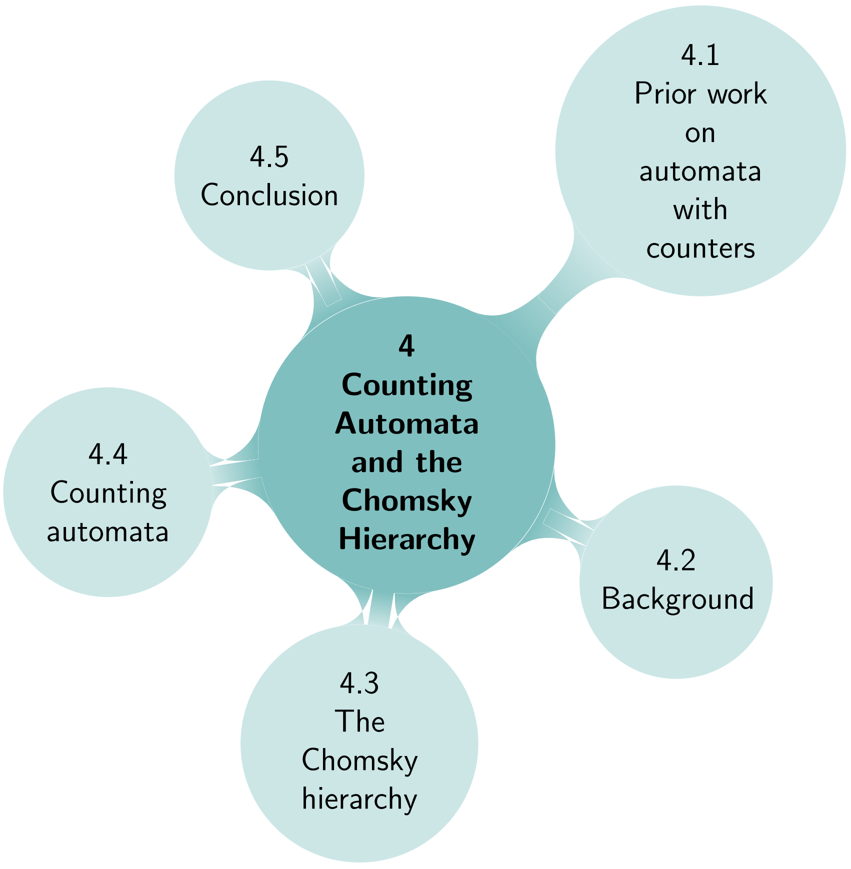
M. I. Thurston and D. Field. MsatFinder: Detection and characterisation of microsatellites. Online: http://www.genomics .ceh.ac.uk/msatfinder/, 2005.

P.E. Warburton, J. Giordano, Y. Cheung, Y. Gelfand, and G. Benson. Inverted Repeat Structure of the Human Genome:The X-Chromosome Contains a Preponderance of Large, Highly Homologous Inverted Repeats That Contain Testes Genes. *Genome research*, 14:1861–1869, 2004.

Y. Wexler, Z. Yakhini, Y. Kashi, and D. Geiger. Finding approximate tandem repeats in Genomic sequences. *Proceedings of the Eighth Annual International Conference on Computational Molecular Biology*, pages 223–232, 2004.

S.A. Yevtushenko. The correctness-by-construction approach to programming. In *Proceedings of the 7th national conference on Artificial Intelligence KII-2000. System of data analysis "concept explorer" (in Russian)*, pages 127–134, 2000.

H. Zhou, L. Du, and H. Yan. Detection of tandem repeats in DNA sequences based on parametric spectral estimation. *Trans. Info. Tech. Biomed.*, 13(5):747–755, September 2009. ISSN 1089-7771.


4.1
Prior work on automata with counters
4.5
Conclusion
4
Counting Automata and the Chomsky Hierarchy
4.4
Counting automata
4.2
Background
4.3
The Chomsky hierarchy

4 Counting Automata and the Chomsky Hierarchy

"The world is full of magic things, patiently waiting for our senses to grow sharper."
... William Butler Yeats

This chapter is concerned with a class of automata that extends on the class of DFAs by having counters associated with transitions. I call these Counting Automata (CAs) and define several variants, each being positioned at a specified level of the Chomsky hierarchy of languages. The idea of adding counters to finite automata originated while I was developing algorithms for microsatellite detection. $\texttt{Fire}_\mu\texttt{Sat}_1$ and $\texttt{Fire}_\mu\texttt{Sat}_3$ implement cyclic FAs that associated counters with states in a very informal manner. $\texttt{Fire}_\mu\texttt{Sat}$ counted for any microsatellite $u = u_1, u_2 \cdots u_i$ where ρ (u_1 in the current example) is the introductory motif, the number of perfect matches and different mutations occurring in u_j. At the time of proposing $\texttt{Fire}_\mu\texttt{Sat}$, a formal definition for CAs had not been proposed.

I have subsequently played with formalising the idea of CAs. I have noted the relationship between monotonic counters as operands in simple relational expressions governing the edges of CAs and the Chomsky hierarchy. I report on my findings in this chapter.

The first section of the chapter is an overview of alternative DFA extensions, found in the literature, that rely on counters.

Then Section 4.2 briefly gives the historical background that led to the notion of computability, the Turing Machine and, ultimately, to the Chomsky hierarchy. The Chomsky hierarchy is then discussed in Section 4.3. Even though these latter two sections cover classical topics, the material is provided here for completion.

In Section 4.4 a number of CAs are introduced and some of the languages accepted by these CAs are positioned within the Chomsky hierarchy. Section 4.5 concludes this chapter.

To keep things simple, preference is given to graphical illustrations of abstract machines instead of defining them formally in terms of mathematical expressions and/or transition tables. In the case of already established machines, these are drawn according to the conventions provided in Chapter 2, unless otherwise specified.

Note positioning in the Chomsky hierarchy of the automata presented in Section 4.1.1 is not discussed in this chapter. The reason is that the authors clarify that these automata are *only* regular language acceptors. For this reason they are called *counter extended* ***finite automata*** — thus occupying the same position in the Chomsky hierarchy as regular languages. Furthermore it will be seen that the automata presented in Section 4.1.2 are not intended to accept any language but rather to count occurrences of events. Classification of these automata within the Chomsky hierarchy is thus not plausible. In contrast to the above mentioned, different *types* of CAs, catering for 3 respective Chomsky hierarchy language-types, are introduced in this thesis. Section 4.4 provides details. For this reason it has been decided to postpone the background discussions of both *Computability* (Section 4.2) as well as the *The Chomsky Hierarchy* (Section 4.3) to just before the CA introduction and CA Chomsky classification.

4.1 Prior work on automata with counters

The works of previous authors that might be seen as relating to CA_{T3}s are briefly considered below.

4.1.1 Counter extended finite automata

Sperberg-McQueen defined *counter extended finite automata* (CEFAs) to accept languages defined by what he calls “regular expressions with integer-range exponents” (REIREs).

As an informal example of Sperberg-McQueen's syntax for a REIRE, consider the following:

$$(a\{3,4\}c\{0,1\})\{1,2\}$$

This regex defines all words where 3 or 4 `a`s are concatenated (the `a`$\{3,4\}$ part of the regex) followed by 0 or 1 `c`s (the `c`$\{0,1\}$ part of the regex). This pattern should be repeated at least once, but at most twice, this being indicated by the {1,2} part of the regex. The set of words defined by this regex is therefore given by:
{`aaa`, `aaaa`, `aaac`, `aaaac`, `aaaaaa`, `aaaaaaa`, `aaaaaac`, `aaacaaa`, `aaaaaaaa`, `aaaaaaac`, `aaaacaaa`, `aaacaaaa`, `aaacaaac`, `aaaaaaaac`, `aaaacaaaa`, `aaaacaaac`, `aaacaaaac`, `aaaacaaaac`}

He also proposes *counting regular attribute grammars* (CRAGs).

Example 4.1.1. *As a small example of a CRAG, the following shows the grammar for the language of the REIRE* $(a\{2,4\})$*:*

$$\begin{aligned} S &\rightarrow a\ A(1) && (4.1)\\ A(n) &\rightarrow \{n<4\}\ a\ A(n+1) && (4.2)\\ A(n) &\rightarrow \{n \geq 2\}\ \Lambda && (4.3)\end{aligned}$$

□

S is the start character and also the head of the production used to generate the first `a` of the word that is to be generated; the counter, n for the nonterminal A is set equal to 1 whenever production 4.1 is executed. When executing production 4.2, the value of `n` is checked against the condition $\{n < 4\}$ before another `a` is added to the word to be generated and the counter n is incremented. Whenever $n \geq 4$, production 4.2 will not be executed. The CRAG will proceed to the next production - production 4.3. The words generated by these productions are {`aa`, `aaa`, `aaaa`}. Words are generated by production 4.3 when it is executed and the condition $n \geq 2$ is met: i.e. when the number of `a`s is greater or equal to 2 and less or equal to 4 in the current working-string.

Example 4.1.2. *The CRAG of the language expressed by the REIRE*

$$(\mathtt{a}\{2,3\}\mathtt{c}\{1,4\})\{3,4\}$$

is as follows:

$$\begin{aligned} S &\rightarrow a\ A(1,1) && (4.4)\\ A(n,m) &\rightarrow \{n<3\}\ a\ A(n+1,m) && (4.5)\\ A(n,m) &\rightarrow \{n \geq 2\}\ c\ C(1,m) && (4.6)\\ C(n,m) &\rightarrow \{n<4\}\ c\ C(n+1,m) && (4.7)\\ C(n,m) &\rightarrow \{(m<4) \wedge (n \geq 1)\}\ a\ A(1,m+1) && (4.8)\\ C(n,m) &\rightarrow \{(m \geq 3) \wedge (n \geq 1)\}\ \Lambda && (4.9)\end{aligned}$$

□

The CRAG starts off with production 4.4 where both counters n and m are initialised to 1. In productions 4.4 and 4.5, n counts the number of as. In production 4.6 it is again initialised to 1 and it then counts the number of cs.

On the other hand, m keeps track of the number of repeated generations of $(a\{2,3\}c\{1,4\})$ and is incremented at production 4.8 where n is also reset to 1. Productions 4.8 and 4.9 ensure that a string from $(a\{2,3\}c\{1,4\})$ is repeated at least 3 and at most 4 times.

It will be seen below that I have adopted aspects of the CRAG notation in discussing the CAs. Conversely, since Sperberger-McQueen *does not* use a graphical representation to explain CEFAs, I will now present brief examples of CEFAs, borrowing from the graphical conventions that I introduced to illustrate CAs. However, CAs bear little resemblance to CEFAs and were conceived without reference to them. Initially the values of the respective counters (dX_i) are equal to 0. The convention is adopted that whenever a state is entered its counter is incremented with 1. Figure 4.1 graphically represents the CEFA whose lan-

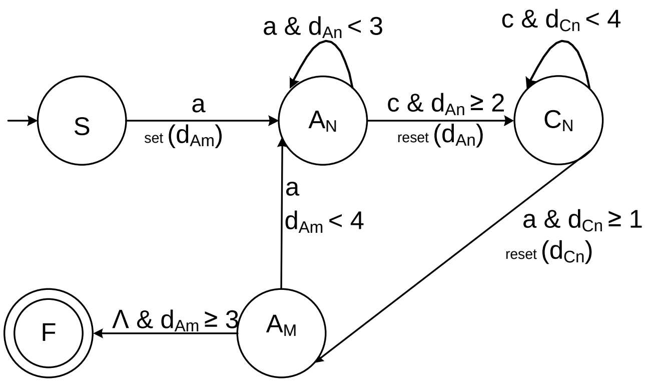


Figure 4.1: A CEFA representing $(a\{2,3\}c\{1,4\})\{3,4\}$.

guage corresponds to the REIRE $(a\{2,3\}c\{1,4\})\{3,4\}$ and to the language of the CRAG in Example 4.1.2. The relationship between the figure and the CRAG is as follows:

- d_{An} represents the n counter initialised by production 4.4 and incremented by production 4.5.
- The `set` function assigns a value of 1 to its parameter d_{Am}. `set(dAm)` represents the initialisation of the m counter of production 4.4.

- d_{Am} represents the counter that keeps track of the number of iterations through production 4.8.
- d_{Cn} holds count of the number of times production 4.7 is executed.
- The `reset` function returns the value of its parameter, d to 0. For example `reset(d_Cn)` assigns the value 0 to d_{Cn}. The *reset operation* is *always* executed after the simple relational expression, appearing above the *reset operation*, has been evaluated.

The figure should be interpreted as follows. The counters are assumed to be initialised to 0. From the start state, S, a transition to state A takes place upon the occurrence of character `a`, and counter d_{An} is incremented. For as long as $d_{An} < 3$ the occurrence of an `a` returns the machine to state A and increments d_{An}. If, in state A, when `c` occurs instead of `a`, and if $d_{An} \geq 2$, then a transition to state C takes place, and the counter d_{Cn} is incremented.

Similarly, in state C, the occurrence of `c` returns to the state and the occurrence of `a` takes the machine to state A_M. When the machine is in state A_M, if an `a` is occurring and $d_{Am} < 4$, the machine proceeds to state A_n. If in state A_M and Λ occurs while $d_{Am} \geq 3$, the machine proceeds to the final state F.

Hovland [2009a] and Hovland [2009b] describe work that relates to Sperberg-McQueen [2004]. The work includes regular expressions with numerical constraints and automata with counters. The membership problem is considered in terms of language classes for regular expressions with unordered concatenation and numerical constraints. Details may be found in the cited articles.

4.1.2 Event count automata

A further example of the use of counters in automata is seen in the notion of *Event count automata* (ECA), proposed by Chakraborty et al. [2005]. These automata capture arrival patterns by counting the number of data items that arrive in a unit interval of time, in the case where a suitable granularity of time has been fixed.

An ECA counter describes arrival patterns of the form $n_1, n_2 \cdots n_k$ where n_i denotes the number of items arriving before the automaton makes a move. A number of different moves can be made. To discuss these fully however, is beyond the scope of this thesis. The interested reader may consult Chakraborty et al. [2005]. Instead, an intuitive idea of ECAs is provided in Figure 4.2.

In this figure each state has an item *arrival mode* pattern, specified as an integer interval. For example, in state q_0 the mode is specified as $[1, 3]$, indicating that

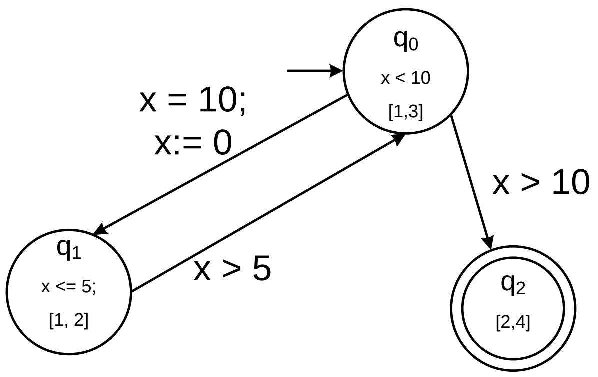


Figure 4.2: An Event Count Automata, from Chakraborty et al. [2005].

at least 1 and at most 3 items will arrive in each unit of time. If the total number of items that have arrived at the end of an interval is exactly 10 when the system is in state q_0, then state q_1 is entered and the counter x is set to 0. In the case where x is more than 10, state q_2 is entered from state q_0. State q_2 is never left.

Suppose that the following number of events arrive in each of 10 successive time units: 3 2 2 3 2 2 2 3 2 4. Using the form (q, n) to indicate that the automaton is in state n and the counter value is n, the automaton starts off as $(q_0, 0)$ and then runs as follows:

1. $\dashrightarrow (q_0, 3)$ (after the first time unit's arrival of 3 events)
2. $\dashrightarrow (q_0, 5)$ (after the second time unit's arrival of 2 events)
3. $\dashrightarrow (q_0, 7)$ (after the third time unit's arrival of 2 events)
4. $\dashrightarrow (q_0, 10)$ (after the fourth time unit's arrival of 3 events)
5. $\rightarrow (q_1, 0)$ (now move to state q_1 and set x to 0 as indicated on the edge by the characters x:=0)
6. $\dashrightarrow (q_1, 2)$ (after the fifth time unit's arrival of 2 events)
   $\dashrightarrow (q_1, 4)$ (after the sixth time unit's arrival of 2 events)
   $\dashrightarrow (q_1, 6)$ (after the seventh time unit's arrival of 2 events)
7. $\rightarrow (q_0, 6)$ (now move to state q_0 — the value of x remains as is)
8. $\dashrightarrow (q_0, 9)$ (after the eighth time unit's arrival of 3 events)
   $\dashrightarrow (q_0, 11)$ (after the ninth time unit's arrival of 2 events)
   $\rightarrow (q_2, 11)$ (move to the final state, q_2 — the value of x remains as is)

9. $\dashrightarrow (q_2, 15)$ (after the tenth time unit's arrival of 4 events)

Other automata concerning arithmetic are discussed in the next section.

4.1.3 Other automata involving arithmetic

Besides the work of Sperberg-McQueen [2004] and Chakraborty et al. [2005], I am aware of other work on automata that also involves the use of arithmetic and that, in this sense, is related to CAs defined below.

For example, in 1962 Schutzenberger [1962] published an article entitled *Finite Counting Automata* that he describes as "an attempt towards a classification of the (infinite) monoids of finite dimensional rational matrices which are the semidirect sum of finite monoids." It will be seen that this formalism does not relate to my investigation. Neither does the work of Klaedtke and Ruess that introduces the Parikh automaton [Cadilhac et al., 2012]; nor that of Karianto [2005] who extended Parikh automata. Detailed discussion of these formalisms is beyond the scope of this thesis.

4.2 A brief background of computability

In 1900, David Hilbert addressed a conference predicting mathematical problems that would be important in the forthcoming century. He predicted 23 areas that, in fact, turned out to be the major thrust of mathematics for the twentieth century. Although the computer itself was not one of his predictions, two areas he had predicted (*mathematically provable results and set theory*) turned out to be of seminal importance to computer science [Cohen, 1997]. He wanted the confusion in *set theory* to be resolved and, secondly, he was not satisfied that all the *mathematically provable results* were true.

Hilbert had in mind to formalise enough of classical arithmetic for doing analysis and simultaneously avoid paradoxes. Basically the idea was to start off with an axiomatizing of classical arithmetic and then to formalise it. During the formalisation process the traditional content of mathematics is removed. The remaining features are purely formal. The task of the formalist was to show that the resultant formalism provides a consistent as well as complete formal theory of classical arithmetic.

One of the central problems facing the formalist was that of finding a definite finitary formal procedure (algorithm) that could be used to unequivocally decide the provability of any claim in formalised mathematics. Such a procedure or algorithm should thus provide a *yes* or *no* answer on whether a statement complying with all the conditions of the formalism (set of axioms) is universally true or not. The described decision problem became known as Hilbert's *Entscheidungsproblem* [Cleland, 2004].

The paragraphs below offer a short discussion of the *status quo* of mathematical provability at the turn of the 19th century. It points to the contribution of set theory to prove one of the most philosophically important theorems in theoretical computer science. Thereafter, a brief discussion follows of the development of *computability* and *computers* as a partial consequence of work in these research fields.

4.2.1 Mathematically provable results

What statements have proofs and how can we generate these proofs?[1]. Church[2], Kleene[3], Post[4], Markov[5], Von Neuman[6] and Turing[7] worked on these problems—mostly independently. These mathematicians each provided different very simple sets of building blocks that seemed to be the "atoms" from which mathematical algorithms can be built, each constituting a similar (though slightly different) version of a universal model for algorithms—what could be referred to as a universal algorithm machine.

Gödel and Turing responded to the *Entscheidungdproblem* that Hilbert had posed in 1928. In 1931 Kurt Gödel proved his famous *Incompleteness Theorem*, showing

[1]Note that mathematical theories regarding the *nature* of the *proof* itself became a new branch of mathematics namely *metamathematics* [Minsky, 1967].

[2]Alonzo Church (1903 - 1995) is best known for the lambda calculus, the Church-Turing thesis, Frege-Church ontology and the Church-Rosser theorem.

[3]A number of mathematical concepts are named after Stephen Kleene, a student of Church. These concepts include Kleene hierarchy, Kleene algebra, the Kleene closure, Kleene's recursion theorem and the Kleene fixed point theorem. Kleene invented regular expressions too.

[4]Emil Post is known for developing Post-machines and truth tables independently of Wittgenstein and Pierce.

[5]Andrei Andreevich Markov is known for his work on stochastic processes, Markov chains and Markov processes.

[6]John von Neumann is best known for the Von Neumann computer architecture, self-replicating machines and stochastic computing. Note that Von Neumann contributed in several other fields too — foundations of mathematics, functional analysis, ergodic theory, geometry, topology and numerical analysis.

[7]Alan Turing, a student of Church, is known as the father of Theoretical Computer Science and Artificial Intelligence. Turing provided a formalisation of the concepts of *algorithm* and *computation* in terms of the so-called Turing machine (TM). The TM is a theoretical model of a general purpose computer. Besides being a mathematician and computer scientist, Turing was also a mathematical biologist, cryptanalyst and ultra distance runner.

that any mathematical system necessarily had statements that could neither be proved nor disproved. Gödel's incompleteness theorem illustrated that there is no solution to Hilbert's *Entscheidungsproblem.* Gödel showed that (in the formalism of Principia Mathematica) there are propositions $\mathfrak{U}$ such that neither $\mathfrak{U}$ nor $\neg\mathfrak{U}$ is provable (Turing [1936]).

Turing's invention of his universal computing machine, generally referred to as a Turing Machine (TM), was a response to the *Entscheidungsproblem* challenge too. Turing used the *halting problem*[8] to show why the *Entscheidungsproblem* could not be solved.

4.2.2 Set theory

Philosophically, the *halting problem* is one of the most important problems in theoretical computer science, since it demonstrates that not all problems are algorithmically solvable. The undecidability of the *halting problem* was proved, using a technique referred to as diagonalisation, discovered by the mathematician Georg Cantor in 1873 [Sipser, 2006]. Cantor was concerned with the measuring of two infinite sets. He asked the question: "If we have two infinite sets, how can it be determined if the one is greater than the other or if they are of the same size?" He developed the diagonalisation technique to answer this question. Results obtained by applying the diagonalisation technique are often counter-intuitive. (Details of the diagonalisation technique and the role that it plays in proving the *halting problem* can be found in Sipser [2006].) Note that there are other ways to prove the *halting problem* undecidable too. For example, Cohen [1997] and Harel and Feldman [2004] do so by mere logical (verbal) argument. Davis et al. [1994] prove the undecidability of the *halting problem* by using an algorithmic approach.

These findings destroyed all hope of achieving Hilbert's program of "mechanising" mathematics and even of deciding whether an arbitrarily selected problem can be solved mechanically (in other words, of determining which classes of problems are computable). Turing's model, however, employed such a simple set of mathematical instructions that a possibility existed of actually constructing a physical model of Turing's idea [Cohen, 1997]. Note that, although it is often said that the 1936 paper of Turing did not really affect the *practical* development of the computer, Minsky [1967] maintains that Turing's paper should be viewed against the intellectual background of a variety of ideas concerning descriptions and processes. In his 1936 paper, Turing provided an answer to the fundamental

[8]The halting problem is an undecidable problem for TMs. It can be shown that it is impossible to design a TM that takes an arbitrary TM, T, as input together with an arbitrary input string $u_1 \cdots u_z$, and then decides whether or not T will halt on that input. See, for example, Sipser [2006] or Mahesh [2013] for a proof of the undecidability of the halting problem.

question —*What processes can be described?* Turing's answer is related to algorithms, computability and computers. The word *description* definitely entails some language. Could a fixed language admit descriptions of all desirable processes? Could there be processes which are well-defined but cannot be described at all? These questions are associated with the idea of an algorithm—an effective procedure—for calculating the value of some quantity or finding a solution of some mathematical problem [Minsky, 1967].

Other fields of science were beginning to develop and propose mathematical problems of their own. One of these fields was linguistics. Chomsky [1956] proposed the well-known classification scheme for grammars in 1956.

4.3 The Chomsky hierarchy

Before the Chomsky hierarchy for formal grammars is presented, a brief background of Chomsky himself is in order.

Noam Chomsky has been teaching at the Massachusetts Institute of Technology (MIT) since 1955.

He published the Chomsky hierarchy[9] in 1956 in an article *Three models for the description of language* [Chomsky, 1956]. According to the 1992 edition of *Arts and Humanities Citation Index*, between 1980 to 1992 he was cited more often than any other living scholar. He was the eighth most cited source overall [Hughes, 2001, Robinson, 1979][10]. He is the author of over 100 books [Arnove, 2018].

Besides the Chomsky hierarchy, he is also credited as being the creator or co-creator[11] of the universal grammar (UG) theory. Chomsky published his original ideas of the UG theory in 1965 and 1966 [Chomsky, 1965, 1966]. Details about the UG theory are beyond the scope of this thesis. Furthermore Chomsky is the co-author of the Chomsky-Schützenberger theorem. Taking all of the foregoing into account, then, the following quote from Chomsky is noteworthy:

[9]The Chomsky hierarchy is occasionally also referred to as the Chomsky-Schützenberger hierarchy acknowledging Marcel-Paul Schützenberger (a French medical doctor and mathematician), who played a crucial role in the development of the theory of formal languages [Wilf et al., 1996].

[10]According to Hughes [2001] only Marx, Lenin, Shakespeare, Aristotle, the Bible, Plato, and Freud are cited more often in academic journals than Chomsky, who edges out Hegel and Cicero.

[11]The term universal grammar (UG) was borrowed by Chomsky from an earlier grammatical tradition that explicitly sought universal semantic roots of syntax (for example the 1660 Port-Royal *Grammaire génerale et raisonnée*) [Pesetsky, 2009].

> Science talks about very simple things, and asks hard questions about them. As soon as things become too complex, science can't deal with them . . . But it's a complicated matter: Science studies what's at the edge of understanding, and what's at the edge of understanding is usually fairly simple. And it rarely reaches human affairs. Human affairs are way too complicated [Krauss and Carroll, 2006].

In this chapter we define various *theoretical* machines and show where the languages accepted by these theoretical machines fit into the Chomsky hierarchy. Minsky [1967] points out that theoretical machines imply that we have to abstract away many realistic details and features of mechanical systems. CAs, proposed here, are indeed abstract.

Section 4.2 explained briefly *how* theoretical machines, especially the TMs, contributed towards the development of computers. Subsection 4.3.1 gives the Chomsky classification of formal grammars according to production rules. Each of the different types of formal grammars define classes of languages that can also be defined by appropriate types of theoretical machines.

4.3.1 Grammars in the Chomsky hierarchy

The Chomsky hierarchy distinguishes between four types of languages —Type 0, Type 1, Type 2 and Type 3. The classification is in line with the restrictions on the productions of the grammars generating these languages as defined in Section 2.2 of Chapter 2. Each of the language types has a corresponding theoretical machine (automaton) class.

Information about the Chomsky hierarchy is widely available. The primary sources used in this text are Cohen [1997], Hopcroft and Ullman [1979], Sipser [2006] and Salomaa [1985]. The following gives a brief overview of the hierarchy.

- **Type 0 grammars**
  Type 0 grammars are also referred to as unrestricted grammars, phrase-structured grammars or recursive enumerable (r.e.) grammars[12]. Definition 2.2.22 defines unrestricted grammars.
  Classical machines accepting Type 0 grammars

[12]The terms semi-Thue grammars, non-terminal-rewriting grammars and context-sensitive-with-erasing grammars are also used to refer to type 0 grammars [Cohen, 1997]. Recursive enumerable refers to enumerable sets. The elements of an enumerable set can be placed in a one-to-one correspondence with the set of natural numbers. Thus it is possible to index each element of a recursive enumerable set as the i^{th} element — it can be listed by the natural numbers. A set with the described properties is also referred to as countably infinite [Mahesh, 2013].

Languages generated by phrase-structured grammars are recognised by TMs (Definition 2.3.22) that read from and write on endless tapes. Other machines that accept Type 0 grammars include nPDAs with n > 1 (Definition 2.3.18) and Post Machines[13]. There can be distinguished between r.e. languages that are Turing decidable and those that are not Turing decidable. Turing decidable languages forms a subclass, discussed below, of the *Type 0* languages.

If any language L, is not Turing decidable it implies that even though a TM can always be constructed to accept all words in L, none of these will be able to reject all words in L' — i.e. for a subset of words in L' it will loop forever. Thus not all r.e. languages are recursive.

Recursive grammars: a subset of Type 0 grammars.
Recursive languages are Turing decidable — a language, L, is said to be Turing decidable if a TM, T, can be constructed such that all words in L are accepted by T and all words in L' (Definition 2.2.12) are rejected by T. An example of a recursive language is the problem of determining if two regular expressions with squaring describe the same set. If r is a regular expression over Σ then r^2 is a regular expression with squaring (Salomaa [1985]).

- **Type 1 grammars**
  Type 1 grammars or *context sensitive grammars* (CSGs, Definition 2.2.21) generate context-sensitive languages.

  Classical machines accepting Type 1 grammars
  Languages generated by CSGs are recognised by linearly-bounded automata (LBAs, Definition 2.3.21). LBAs constitute a subclass of TMs with a restriction on the length of the input tape. Clearly, acceptors of Type 0 languages can also accept Type 1 languages. Furthermore for every CSG there exists a specific LBA that accepts all words generated by that CSG and that crashes for all other words.

- **Type 2 grammars**
  Type 2 grammars or context free grammars (CFGs, Definition 2.2.17) generate context free languages (CFLs).

  Classical machines accepting Type 2 grammars
  Type 2 languages (CFLs) are accepted by PDAs (Definition 2.3.18). The complement of $\{\mathtt{a}^\mathtt{n}\mathtt{c}^\mathtt{n} \mid \mathtt{n} \geq \mathtt{1}\}$ is context free and can be accepted by a

[13]Post proposed the Post Machine in 1936, the same year in which Turing proposed the Turing machine. Cohen [1997] provides a definition of a Post machine.

PDA. Acceptors of Type 0 and Type 1 languages can also accept Type 2 languages.

Deterministic context free languages (DCFLs): a subset of languages generated by Type 2 grammars.
The DCFLs constitute the *proper* subset of the CFLs that can be accepted by deterministic PDAs (DPDAs). The language $\{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}} \mid \mathtt{n} \geq \mathtt{1}\}$ is deterministic context free and can be accepted by a DPDA. However the complement of $\{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}} \mid \mathtt{n} \geq \mathtt{1}\}$ is not *deterministic* context free and therefore cannot be accepted by a DPDA. It is, however, context free and can thus be accepted by a PDA (Definition 2.3.18). There are thus CFLs that can only be accepted by PDAs but not by DPDAs.

The fact that DPDAs accept the DCFLs is important from a practical point of view. It is inefficient to implement non-determinism in a compiler. Consequently programming languages are generally designed to be mostly deterministic context-free (although they may contain a few context-sensitive features ([Mahesh, 2013])). It is interesting to note that linear grammars (Definition 2.2.18) can generate some DCFLs (but not all) and they can generate some CFLs that are not DCFLs (but not all).

- **Type 3 grammars**
  Both *right linear grammars* and *left linear grammars* (see Definition 2.2.13) generate regular languages. Furthermore regular expressions (Definition 2.2.14) define only Type 3 languages.

  Classical machines accepting Type 3 grammars
  The simplest deterministic machines that accept regular languages are DFAs (Definition 2.3.1). Non-deterministic machines that accept regular languages are NFAs (Definition 2.3.15). In contrast to DPDAs and non-deterministic PDAs, DFAs and NFAs accept exactly the same set of languages, namely, regular languages. Of course, acceptors of Type 0, 1 and 2 languages can also accept Type 3 languages. An example of a regular language is $\mathtt{a}^*\mathtt{c}^*$ expressed as a regular expression.

Chapter 5 illustrates how to formulate regular expressions that define TRs. Note that *only* the most general machines accepting a particular language have been included in the foregoing discussion of Figure 4.3.

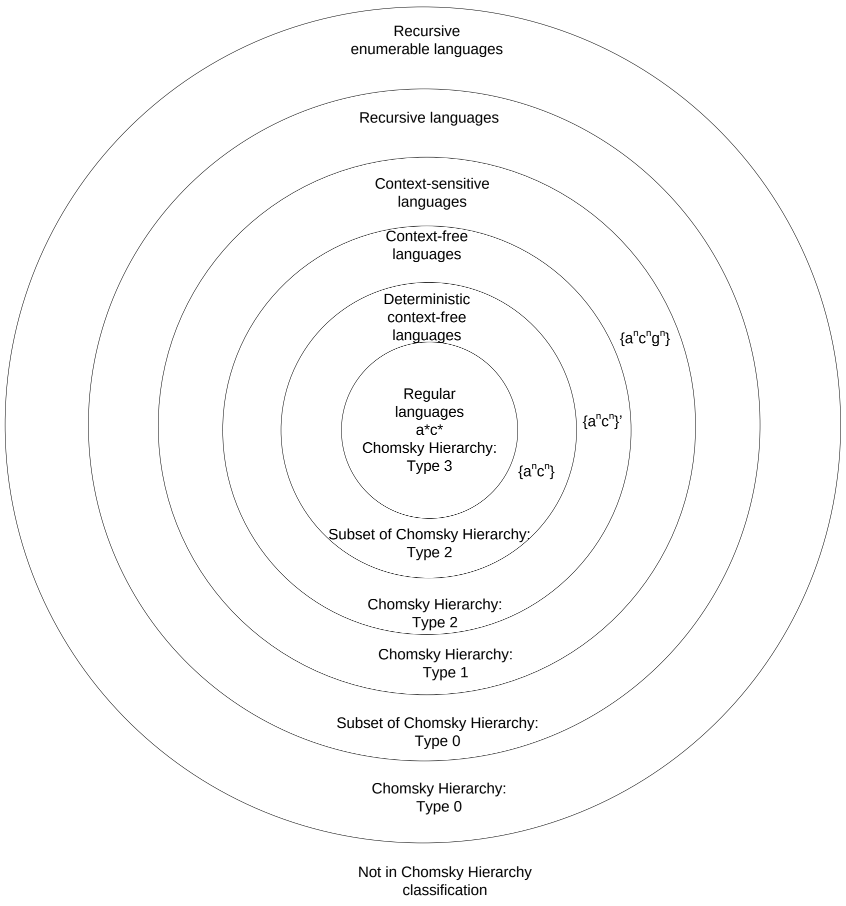

Recursive enumerable languages
Recursive languages
Context-sensitive languages
Context-free languages
Deterministic context-free languages
Regular languages
a*c*
Chomsky Hierarchy: Type 3
{a^n c^n}
{a^n c^n}'
{a^n c^n g^n}
Subset of Chomsky Hierarchy: Type 2
Chomsky Hierarchy: Type 2
Chomsky Hierarchy: Type 1
Subset of Chomsky Hierarchy: Type 0
Chomsky Hierarchy: Type 0
Not in Chomsky Hierarchy classification

Figure 4.3: A nested set formed by six classes of languages, adapted from Cohen [1997].

4.4 Counting automata

The definitions, advantages and capabilities of different CAs are considered here. The definition for CA_{T3} that is given in the first subsection serves as a basis for defining CA_{T2} and CA_{T1} in the subsections to follow.

- The focus is on CA_{T3} as a regular language acceptor — i.e. an acceptor of languages belonging to Type 3 of the Chomsky hierarchy. Because TRs are also regular languages, CA$_{T3}$ is the acceptor used in subsequent chapters of this thesis.

- CA$_{T2}$ is presented as an acceptor of a subset of CFLs (a subset of Type 2 of the Chomsky hierarchy).
- CA$_{T1}$ is explored as the acceptor of a subset of CSLs (Type 1 of the Chomsky hierarchy).

In Subsections 4.4.1, 4.4.5 and 4.4.6 it is assumed that simple relational expressions containing the operators $<, >, \leq, \geq$ and $=$ can be evaluated.

4.4.1 Counting automata type 3: CA$_{T3}$s

During the development of a TR-detection algorithm that implements DFAs, a need became apparent to define (for the first time), in the interest of memory management, something called a counting automaton of type 3, abbreviated to CA$_{T3}$.

CA$_{T3}$s accept all the regular languages. Definition 4.4.1 given below formally defines a CA$_{T3}$. The $T3$ subscript refers to regular languages as *Type 3* languages of the Chomsky hierarchy. While an appropriate definition for a CA$_{T3}$ was being formulated, it was realised that by modifying the definition slightly (specifically with respect to conditions of transition functions in terms of counter comparisons), languages belonging to various levels of the Chomsky hierarchy can be accepted. Consequently CA$_{T1}$ and CA$_{T2}$ are defined too in this chapter.[14]

Formally a CA$_{T3}$ can be defined as follows:

Definition 4.4.1. ***Counting automata type 3 (CA$_{T3}$)***
A CA$_{T3}$ is a seven-tuple,

$$M = (Q, q_0, \mathcal{F}, \Sigma, \varphi, R, \delta)$$

where the characters have the following meanings:

- $Q = \{q_0, q_1, q_2, \cdots q_n\}$ *is a non-empty finite set of states. Every* q_i *is a tuple —* $\langle p_i, d_i \rangle$ *where* d_i *is an integer counter and* p_i *is a state identifier. The initial value of* d_i *is zero.* d_i *is incremented every time state* $q_i = \langle p_i, d_i \rangle$ *is entered.*
- $q_0 = \langle p_0, d_0 \rangle \in Q$ *is designated to be the start state.*
- $\mathcal{F} \subseteq Q$ *is the set of final states.*

[14]Preliminary thoughts on CA$_{T0}$s are presented in Appendix A.

- $\Sigma = \{a_1, a_2, a_3, \cdots a_n\} \cup \{\Delta\}$ *is a finite alphabet where* Δ *is the null character.*
- $\varphi : Q \times \Sigma \to \mathbb{N}$ *is called the* threshold *function. It is a total function that associates an integer constant, called a threshold value, with each character and each state. The transition function's behaviour (given below) is governed by these threshold values. If no threshold value is explicitly indicated the default value is taken as* ∞.
- $R : Q \times \Sigma \to \{<, >, =, \leq, \geq\}$ *is called the* relational *function. It is a total function that associates a relational operator with each character at each state.*
- $\delta : Q \times \Sigma \to Q$ *is called the* transition *function.*

$$\delta(q_i, a_i) = \delta(\langle p_i, d_i\rangle, a_i) = \begin{cases} \langle p_k, d_k + 1\rangle, & \text{if } d_i \; R(q_i, a_i) \; \varphi(q_i, a_i). \\ \bot, & \text{otherwise.} \end{cases} \tag{4.10}$$

- *Where convenient, it will be assumed that the end of the input string is indicated by an infinite number of* Δ*s.*

□

In order to make the above definition clear, notice that $d_i \; R(q_i, s) \; \varphi(q_i, s)$ in Equation (4.10) is an expression that evaluates to a Boolean value. Suppose, for example, that $d_i = 5$, $R(q_i, s)$ evaluates to $<$ and $\varphi(q_i, s) = 10$. Then $d_i \; R(q_i, s) \; \varphi(q_i, s)$ is the expression $5 < 10$ and this expression evaluates to true.

From the arbitrary state $q_i = \langle p_i, d_i\rangle$ the transition function allows for one of two possible outcomes for a given character, a_i: either state $\langle p_k, d_k + 1\rangle$ if $d_i \; R(q_i, a_i) \; \varphi(q_i, a_i)$ evaluates to true, or to $\bot$ otherwise.[15] Every transition increments by one, upon entrance, the counter component of the state.

Thus, CA$_{T3}$s may be thought of as DFAs that have counters in each state and simple relational expressions on their edges. These simple relational expressions

[15]Some texts treat the transition function of an FA, denoted by $\delta : Q \times \Sigma \to Q$, as a total function and allow for the possibility that some transitions lead to a sink state, denoted by $\bot$ where $\bot \notin F$. The sink state is a dead end — i.e. for all $a_i \in \Sigma$, $\delta(\bot, a_i) = \bot$. Any DFA with such a sink state will thus be cyclic. In other texts δ is regarded as a possibly partial function, meaning that in some cases, $\delta(q, a_i)$ may not be defined. An ADFA per definition cannot have a sink state (because such a state always has a cycle back to itself), and consequently, the transition functions of ADFAs are partial. In this thesis $\delta(q, a_i) = \bot$ may be construed to mean either that the transition from state q on character a_i is not defined, or that the transition is to the sink state. The meaning will be clear from the context and will not affect the arguments made. The conventions regarding $\bot$ holds for CAs too.

rely on threshold values that are set according to functional requirements. Note that in the limit, a CA$_{T3}$ degenerates into a conventional DFA. This is the case when $R(q_i, a) =$ " $<$ " and $\varphi(q_i, a) \equiv \infty$ for all $q_i \in Q$ and all $a \in \Sigma$. The default relational expression (if nothing else is said) is that the state counter, d is less than ∞.

The purpose of introducing CA$_{T3}$s is to minimise memory requirements and at the same time provide for the recognition of languages such as those of the DFA depicted in Figure 4.6. This recognition comes at a slight runtime cost as seen below. Furthermore CA$_{T3}$s obtain statistical data during TR-detection.

Algorithm 4.4.2 indicates how to determine whether $w \in \mathcal{L}(M)$, where M is a CA$_{T3}$ whose language is $\mathcal{L}(M)$, and where $w = a_0a_1 \ldots a_j$ is a word of whose length is $|w|$. $\perp$ is used to indicate either a sink state of M (if δ is a total function) or the value of δ (if δ is a partial function). Equation 4.10 is actually

Algorithm 4.4.2 (Is $w \in \mathcal{L}(M)$?)**.**

```
$q, i := q_0, 0$;
{ Invariant: $(i \leq |w|) \wedge ((a_0 \ldots a_{j-1})$ has been processed ) }
do $((i < |w|) \wedge (q \neq \perp)) \rightarrow$
    $\oplus, t := R(q, a_i), \varphi(q, a_i)$;
    if $(d \oplus t) \rightarrow \langle p, d \rangle := \delta(q, a_i)$
    [] $\neg(d \oplus t) \rightarrow \langle p, d \rangle := \perp$
    fi;
    $i, q := i + 1, \langle p, d \rangle$
od;
$accept := ((i = |w|) \wedge (q \in F))$
{ post $(accept \Leftrightarrow w \in \mathcal{L}(M))$ }
```

□

applied when determining the value of $\delta(q, a_i)$. In order to apply Equation 4.10, $R(q, w[i])$ and $\varphi(q, w[i])$ have to be computed, yielding say relational operator $\oplus$ and threshold value t respectively. Then, assuming state $q = \langle p, d \rangle$, the new state is computed by considering the Boolean expression $(d \oplus t)$. Depending on whether this expression evaluates to true or false, the variables p and d representing the current state identifier and counter respectively have to be changed as prescribed by Equation 4.10.

Notation 4.4.3. ***Drawing CA$_{T3}$s***
CA$_{T3}$s can be drawn in a similar manner to DFAs. States are depicted as circles. The start state has an in-edge from no-where, whereas final states are two concentric circles. Transitions are depicted as labelled directed edges. In contrast to the edges of DFAs, which are labelled with characters from Σ only, the edges

of CA$_{T3}$s are labelled both with characters from Σ and with predicates (simple relational expressions as explained in Definition 4.4.1). These predicates evaluate either to true or false and determine, together with the input character, the destination state. □

Let $\Sigma = \{\mathtt{a}, \mathtt{c}\}$. A CA$_{T3}$ that accepts $L = \{\mathtt{a}^5\mathtt{c}^5\}$ can be drawn in two different ways as illustrated in Figures 4.4 and 4.5. In Figure 4.4 the transition on a cycling back to state q_0 has the expected condition $d_0 < 5$ replaced by $d_0 < \infty$. At an implementation level using the default restraint $d < \infty$ removes the need to check against an upper bound. This might sometimes result in fewer comparison operations and therefore in more efficient processing time. However, if an input word contains a very large number of as in the prefix—for example $\mathtt{a}^{100}\mathtt{c}^{107}$ — then the runtime can be improved by relying on the predicate $d_0 < 5$ instead, as in Figure 4.5. Note that similar trade-offs apply to using $d_1 < \infty$ as in Figure 4.4 instead of $d_1 < 5$ as in Figure 4.5.

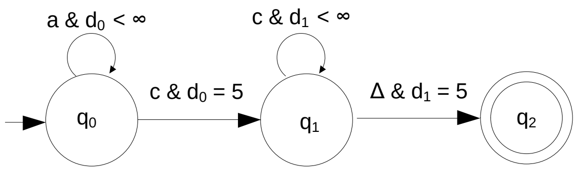


Figure 4.4: A CA$_{T3}$ accepting, the motif $\mathtt{a}^5\mathtt{c}^5$ using ∞ instead of an integer value.

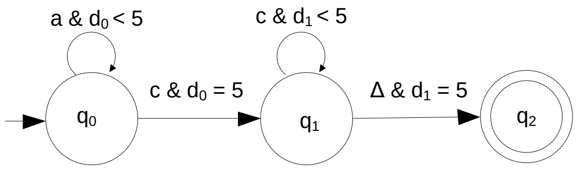


Figure 4.5: A CA$_{T3}$, with all the possible simple relational expressions against integer values included. Only the motif $\mathtt{a}^5\mathtt{c}^5$ is accepted.

The CA$_{T3}$ depicted in Figure 4.5 relies on the information in Table 4.1 and Table 4.2.

Table 4.1 gives the relational expression associated with each transition out of a state on each alphabet character. Thus, if d_i is the counter at state q_i, then the relational expression on a from state q_i is given by $(d_i\ R(q_i, \mathtt{a})\ \varphi(q_i, \mathtt{a}))$.

	a	c	Δ
q_0	$d_0 < 5$	$d_0 = 5$	$d_0 < \infty$
q_1	$d_1 < \infty$	$d_1 < 5$	$d_1 = 5$

Table 4.1: A table, expressing the predicates $(d_i\ R(q_i, a)\ \varphi(q_i, a))$ for the CA$_{T3}$ depicted in Figure 4.5, accepting $\mathtt{a}^5\mathtt{c}^5$.

δ	a	c	Δ
q_0	q_0	q_1	$\perp$
q_1	$\perp$	q_1	q_2

Table 4.2: A transition table, representing δ of the CA$_{T3}$ presented in Figure 4.5 that accepts $\mathtt{a}^5\mathtt{c}^5$.

Table 4.2 is the conventional DFA transition table showing the transitions to be taken from each alphabet character from each state. Of course, the transition to the indicated state should only be made when the associated predicate in Table 4.1 evaluates to true. If it is false, then the transition is represented by $\perp$. Table 4.2 also indicates a transition to $\perp$ if Δ (end of string) is encountered in state q_0 or if a is encountered in state q_1.

A conventional DFA accepting a^5c^5 is shown in Figure 4.6. It has 12 states, one of which is a sink state. If δ may be a partial function, then the sink state may be removed and the DFA becomes an ADFA.

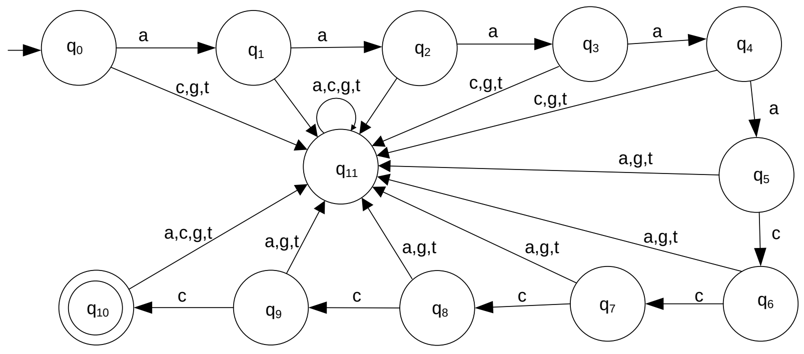


Figure 4.6: A DFA with 12 states over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$ accepting $\mathtt{a}^5\mathtt{c}^5$ only.

In general the number of states required for a DFA to accept $L = \{\mathtt{a}^n\mathtt{c}^n\}$ where $n \geq 1$ and n is a fixed value, is $2n + 2$. (If the sink state is omitted then $2n + 1$ states are required.) In contrast, a CA$_{T3}$ with two counters and two threshold values to recognise $L = \{\mathtt{a}^n\mathtt{c}^n\}$ has only three states (or four, if one includes the sink state). This is illustrated by the CA$_{T3}$ in Figure 4.7. The figure is a *skeleton* CA$_{T3}$ that accepts the language with one element, $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}}\}$, for *any pre specified* n. In any specific context, only the threshold values of the predicates have to change in line with the value assigned to n.

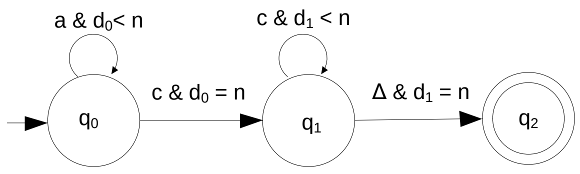


Figure 4.7: A CA$_{T3}$ accepting $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}}\}$ with the condition that n is a constant. Thus the presented CA$_{T3}$ can be seen as a skeleton CA$_{T3}$ where n serves as a place holder that should be a positive integer value.

A DFA is easily converted to a CA$_{T3}$, since each DFA transition can be viewed as a CA$_{T3}$ transition with a predicate specifying that the counter is unbounded (less than infinity). Thus, a CA$_{T3}$ can be constructed to accept any regular language. If the regular language contains substrings that are repeated a finite number of times, a CA$_{T3}$ to define it can usually be found with fewer states than a DFA.

Figure 4.8 depicts a CA$_{T3}$ that recognises the language $L = \{\mathtt{g}^*\mathtt{actactactg}^*\}$. The substring `actactact` is a typical example of a microsatellite (trinucleotide) that might be discovered in a genetic string.

4.4.2 PFAs

Definition 4.4.4. ***Prototype Counting Automata Type 3 (pCA$_{T3}$)***
A pCA$_{T3}$ is a pDFA that has been converted into a pCA$_{T3}$. □

Figure 4.9(a) illustrates a PFA reaching cascading states if `aaaa` and {`c`} are being read. The PFA displayed in Figure 4.9(a) is converted to an equivalent pCA$_{T3}$ depicted in Figure 4.9(b).

In Chapter 5 pCA$_{T3}$s are cascaded to construct CA$_{T3}$s. These CA$_{T3}$s detect TRs and gather statistical data about these detected TRs simultaneously. It will be seen that global counters are introduced, to update the number of perfect matches, mismatches, insertions and deletions. These global counters rely, for updating, on the counters of pCA$_{T3}$s constituting the CA$_{T3}$ relevant for a current detection.

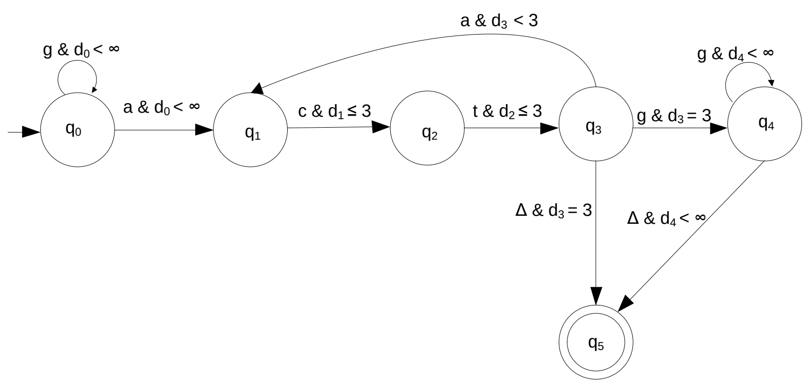
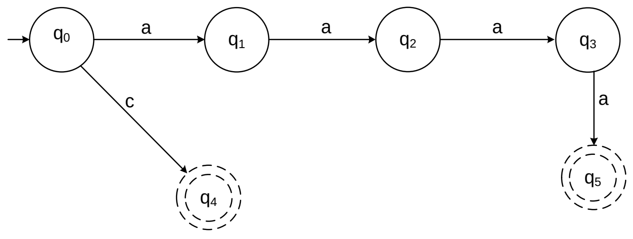
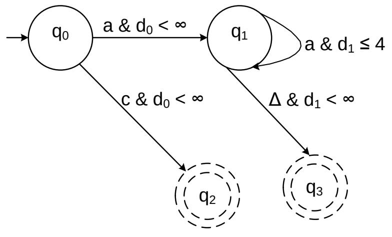


Figure 4.8: A CA_{T3} accepting $L = \{\texttt{g}^*\texttt{actactactg}^*\}$.



(a) pDFA($\texttt{aaaa}, \texttt{c}$)



(b) pCA_{T3}($\texttt{aaaa}, \texttt{c}$)

Figure 4.9: A pDFA and a pCA_{T3} catering for $L = \{\texttt{aaaa}, \texttt{c}\}$

4.4.3 CA$_{T3}$s relating to Moore and Mealy machines

Note that CA$_{T3}$s have both Moore machine (Definition 2.3.16) and Mealy machine (Definition 2.3.17) characteristics.

CA$_{T3}$s have Moore machine characteristics in the sense that an action is triggered when certain states are reached. In that case, a value is incremented whereas in the Moore machine case, some output is issued.

CA$_{T3}$s have Mealy machine characteristics in that their edges activate an action. In the CA$_{T3}$ case, the action is the verification of simple relational expressions, whereas in the Mealy machine case the action is the printing of certain output.

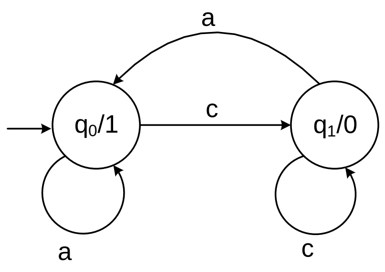


Figure 4.10: A Moore machine.

The output of the Moore machine presented in Figure 4.10 for the input $\mathtt{a}^5\mathtt{c}^5$ is 11111100000. Thus six 1s are consecutively output (corresponding to the 5 `a`s plus an additional 1 that is printed initially, irrespective of the input) followed by 5 consecutive 0s. The 0s are printed every time the character `c` is read. The output of the Moore machine thus allows one to deduce that $\mathtt{a}^5\mathtt{c}^5$ was input.

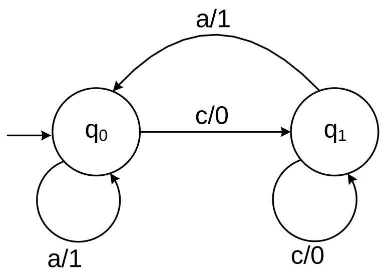


Figure 4.11: A Mealy machine.

The output of the Mealy machine, depicted in Figure 4.11, is similar to that of the Moore machine in Figure 4.10 apart from the fact that the Moore machine outputs an additional 1 in its first state. Instead of printing a 1 or 0 every time a state is entered, a Mealy machine prints the output on its edges. Note that, from analysing the output of either a Moore machine or a Mealy machine and being familiar with the respective machines one can, in some cases, deduce what the input string was.

4.4.4 CA$_{T3}$s and right linear grammars

Recall that Subsection 2.3.2.3 defined linear grammars (in Definition 2.2.18). It then went on to show how to derive a DFA from a right linear grammar that recognises the language generated by the grammar and *vice-versa.*

Recall also that Section 4.1 discussed CRAGs — attribute grammars proposed by Sperberg-McQueen [2004] for representing attribute grammars that contain counters and that generate regular languages. A CRAG can be used to define a right linear grammar that defines the language L. Analogously to Subsection 2.3.2.3, such a CRAG-based right linear grammar can be used to derive a language-equivalent CA$_{T3}$. To illustrate this claim consider Example 4.4.5.

Example 4.4.5. *Consider the CRAG generating* $L = \{\texttt{acgacgacg}(\texttt{acg})^*\}$*, defined below:*

$$S \to \{S_d \leq \infty\}\ aA\ (A_d + 1) \tag{4.11}$$
$$A \to \{A_d \leq \infty\}\ cC\ (C_d + 1) \tag{4.12}$$
$$C \to \{C_d \leq \infty\}\ gG\ (G_d + 1) \tag{4.13}$$
$$G \to \{G_d \leq \infty\}\ aA\ (A_d + 1) \tag{4.14}$$
$$G \to \{G_d \geq 3\}\ \Lambda \tag{4.15}$$

Notice that, aside from the counting annotations, the CRAG conforms to the definition of a (right) linear grammar. To convert this counting *right linear grammar to a CA$_{T3}$ that accepts L, the following steps should be followed. Note that the conversion steps are analogous to those specified in Definition 2.3.14 for deriving a DFA from a right linear grammar.*

- *Create a state for each non-terminal and add an additional final state. The state labelled S is the start state.*
- *For every production rule of the form*

  $\texttt{Non-terminal} \to \{\texttt{relational expression}\}\ \texttt{terminal Non-terminal}$

  draw a directed edge from the non-terminal state on the left to the non-terminal state on the right.

- *Label the drawn edge with the corresponding terminal and relational expression to be found on the left of the terminal, expressed in curly brackets.*
- *For a production with the form*

  $$\texttt{Non-terminal} \rightarrow \{\texttt{relational expression}\}\texttt{terminal}$$

  draw a directed edge to the final state and label it with the terminal and the corresponding relational expression (in curly brackets on the left of the terminal).
- *For a production*

  $$\texttt{Non-terminal} \rightarrow \{\texttt{relational expression}\}\Lambda$$

  draw an edge from the non-terminal state to the final state and label the edge with Δ[16] *as well as with the relational expression.*

By applying this constructive algorithm to the given counting right linear grammar, the CA_{T3} *in Figure 4.12 is obtained. To accommodate the non-terminals of the right linear grammar as state names, the convention of using* $q_i = \langle p_i, d_i \rangle$ *has been replaced by using the non-terminal name itself.*

$\square$

The reverse of the described constructive algorithm can also be applied—i.e. a CA_{T3} such as the one provided in Figure 4.13 can easily be converted into a counting right-linear grammar by following the steps below:

- *Step 1*
  Give all the states $q_i \in Q$ non-terminal subscript names from V.
- *Step 2*
  The start state of the CA_{T3} should be named S in the grammar.
- *Step 3*
  Note that all $x_i \in \Sigma$ of the CA_{T3} are terminals in Σ of the right linear counting grammar.
- *Step 4*
  For each edge $\delta(q_Y, x_i) = q_J$ where $x_i \in \Sigma$:

[16]In the case of CAs, Δ is assumed to follow after the last character of the input string. Note that Λ replaces Δ when the algorithm to *convert a* CA_{T3} *to a right linear counting grammar* is applied.

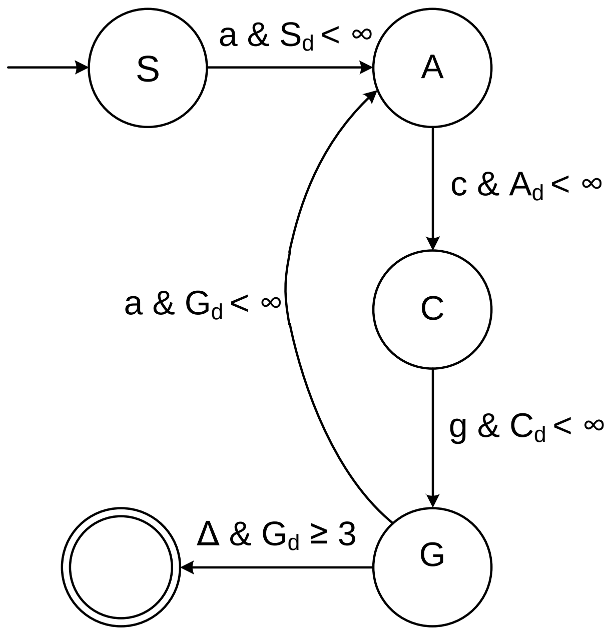


Figure 4.12: CA$_{T3}$ accepting $L = \{\texttt{acgacgacg(acg)}^*\}$.

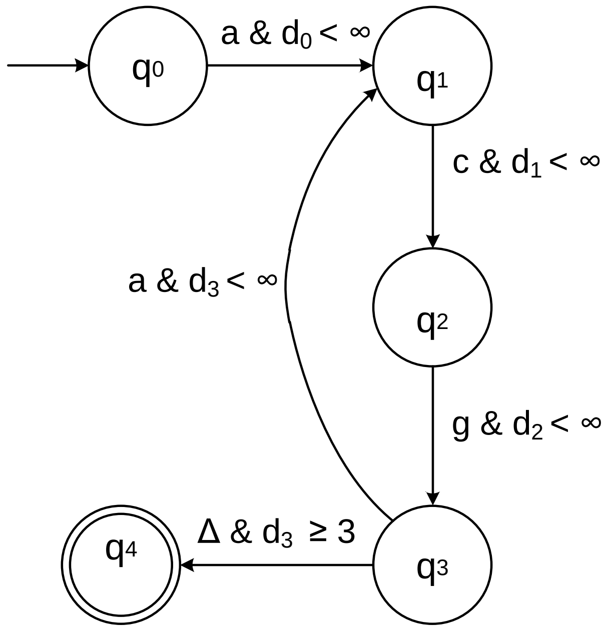


Figure 4.13: CA$_{T3}$ accepting $L = \{\texttt{acgacgacg(acg)}^*\}$.

1. Write down the associated simple relational expression in curly brackets.
2. Let every edge become a production: $Y \longrightarrow x_i J$.
3. Indicate counter incrementation with 1 on the right of the production.

- *Step 5*
  For all $q_i \in \digamma$ if $q_i \in \digamma$ is renamed I in the grammar add the production: $I \longrightarrow \Lambda$.

By applying the steps for *converting a CA$_{T3}$ to a right linear counting grammar* on the CA$_{T3}$ in Figure 4.13 the grammar below is obtained:

$$S \rightarrow \{S_d < \infty\}\ aA\ (A_d + 1) \tag{4.16}$$
$$A \rightarrow \{A_d < \infty\}\ cC\ (C_d + 1) \tag{4.17}$$
$$C \rightarrow \{C_d < \infty\}\ gG\ (G_d + 1) \tag{4.18}$$
$$G \rightarrow \{G_d < \infty\}\ aA\ (A_d + 1) \tag{4.19}$$
$$G \rightarrow \{G_d \geq 3\}\ \Lambda J (J_d + 1) \tag{4.20}$$
$$J \rightarrow \Lambda \tag{4.21}$$

From the preceding discussion it is clear that CA$_{T3}$s have the ability to decrease memory requirements if repetitions of characters or strings are part of the language under consideration. Therefore CA$_{T3}$s are suitable for tandem repeat detection where repetitive motifs need to be detected.

However, before exploring the functionality of CA$_{T3}$s for TR detection, modifications of CA$_{T3}$s are considered, namely CA$_{T2}$s and CA$_{T1}$s. CA$_{T2}$s have the ability to accept a subset of Type 2 of the Chomsky hierarchy of languages — i.e. a subset of the context free languages.

4.4.5 Counting automata type 2: CA$_{T2}$s

Recall that the set of DCFLs is a proper subset of the CFLs; that CFLs are classified as Type 2 in the Chomsky hierarchy classification; that DCFLs are accepted by DPDAs; that DCFLs are generated by CFGs and that DCFLs are of relevance during compiler design.

In this subsection, CA$_{T2}$s will be defined and compared to DPDAs and PDAs. This will highlight their abilities and limitations in recognising CFLs.

CA$_{T2}$s are defined almost similar to CA$_{T3}$s. Only the definition of the threshold function φ is changed. Formally,

Definition 4.4.6. *A CA_{T2} is represented by the same 7-tuple as a CA_{T3}, but the threshold function, φ is defined as follows:*
Let $D = \{d_{j1}, d_{j2}, \ldots d_{jk}\}$ be a set of counters of some set of states, $\{q_{j1}, q_{j2}, \ldots q_{jk}\}$. Let $\{P1, P2\}$ be a partition of $Q \times \Sigma$ (i.e. $P1 \cup P2 = Q \times \Sigma$, $P1 \cap P2 = \emptyset$) such that $0 \leq |P2| \leq 1$. Then:

$$\begin{aligned} \varphi : Q \times \Sigma &\rightarrow \mathbb{N} \cup D \\ \textit{where } \varphi(q_i, a) &= \begin{cases} n & (q_i, a) \in P1 \textit{ and } n \in \mathbb{N} \\ d_j & (q_i, a) \in P2 \textit{ and } d_j \in D \end{cases} \end{aligned}$$

□

Thus, the definition of a CA$_{T2}$ allows for a limited one pair of counters to be compared against each other. Specifically, the counter (d_i) of one state (q_i) is compared against the counter (d_j) of some other state (q_j) whenever a specific character of the alphabet occurs in state q_i. If the relevant relational expression evaluates to true then a transition is made to some new state (not necessarily to state q_j), i.e. $\delta(q_i, a) = q_k$; and if the relational expression evaluates to false then $\delta(q_i, a) = \perp$. In all other instances, CA$_{T2}$s state-to-state transitions are made on the basis of counter-against-threshold comparisons, as for CA$_{T3}$s. Thus, the same drawing conventions introduced for CA$_{T3}$ are used, except that a CA$_{T2}$ can have at most one counter against counter comparison. Thus every CA$_{T3}$ can be seen as a degenerate CA$_{T2}$ that has zero counter against counter comparisons.

The algorithm to determine if a word belongs to the language of a CA$_{T2}$ is similar to the algorithm for CA$_{T3}$s — Algorithm 4.4.2.

Karianto [2005] mentions that the idea of counting devices in automata theory is classic in nature. A DPDA "counts" on a stack in that information can be pushed or popped and the stack can be tested for being empty. CA$_{T2}$ counters can be leveraged to store and retrieve information in a way that resembles the operation of a stack. This is because CA$_{T2}$s are defined to allow for the comparison of two monotonic addition counters.

A CA$_{T2}$ can be considered equivalent to a DPDA with one stack where only one character can be pushed. Instead of pushing characters onto a stack, a CA$_{T2}$ increments an appropriate counter when a state is entered. Subsequently the values of two counters of a CA$_{T2}$ are compared.

This correspondence between a CA$_{T2}$ and a DPDA will now be illustrated by considering two DCFLs. The first is $L = \{a^n c^n \mid n \geq 1\}$ and the second is a so-called *oddpalindrome* language. It will be seen that the first language can be recognised by both a DPDA and a CA$_{T2}$, whereas the second can only be recognised by a DPDA.

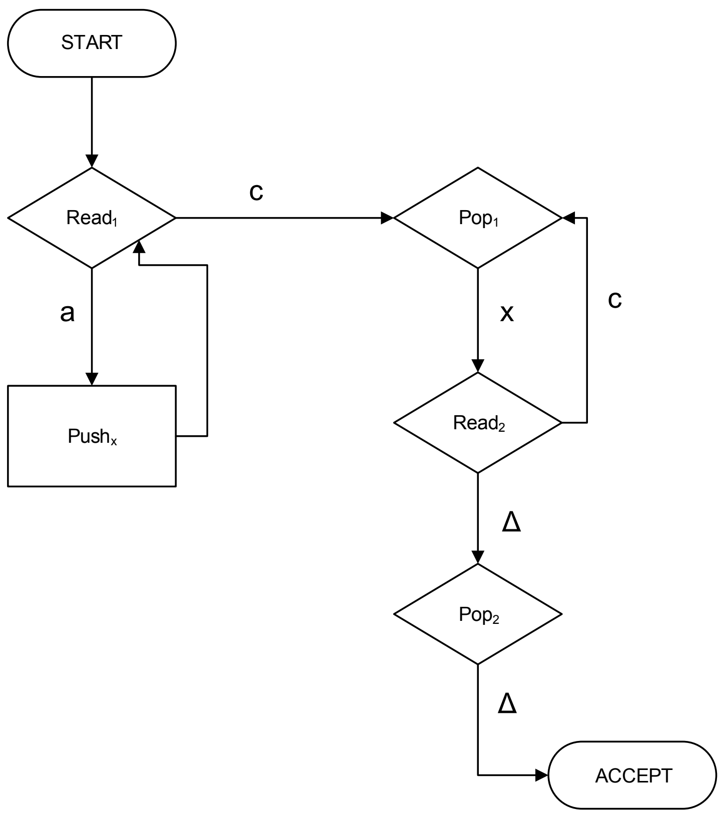


Figure 4.14: A DPDA accepting $L = \{\mathtt{a^n c^n} \mid \mathtt{n} \geq \mathtt{1}\}$.

4.4.5.1 A DPDA for $L = \{a^n c^n \mid n \geq 1\}$

L is accepted by the DPDA presented in Figure 4.14. (Note that there is no consensus on how to present PDAs graphically. Alternative ways of presenting DPDAs as well as PDAs can be found in Mahesh [2013] and in Sipser [2006].)

A word from L is accepted by pushing an `x` onto the stack for each `a` read from the first group of `a`s—the *Read*$_1$-*Push*$_x$-*loop*. Once the first `c` is read by *Read*$_1$, the DPDA branches to *Pop*$_1$. Another loop is required– -*Read*$_2$-*Pop*$_1$ where the remaining sequence of `c`s is read. The DPDA pops an `x` for each `c` read. If the number of `a`s is not equal to the number of `c`s then the DPDA crashes. (If the number of `a`s is greater than the number of `c`s an `x` will be popped in state *Pop*$_2$ causing the machine to crash. If the number of `a`s is less than the number of `c`s then a Δ will be popped in state *Pop*$_1$ causing the machine to crash.)

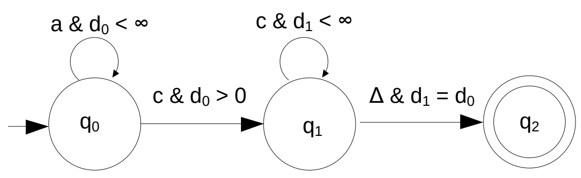


Figure 4.15: A CA_{T2} accepting $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}} \mid \mathtt{n} \geq \mathtt{1}\}$.

4.4.5.2 A $\mathbf{CA}_{T2}$ for $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}} \mid \mathtt{n} \geq \mathtt{1}\}$

Figure 4.15 depicts a CA_{T2} accepting $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}} \mid \mathtt{n} \geq \mathtt{1}\}$.

Note that the counters of states representing the occurrence of `a` and of `c` respectively should be equal.

A CA_{T2} requires less memory than a DPDA — it does not need a stack.[17]

4.4.5.3 A DPDA for an oddpalindrome

As another example comparing DPDAs and CA_{T2}s, consider the DPDA presented in Figure 4.16. It accepts strings of the form $w\mathtt{g}(\mathit{reverse}(w))$, where $w \in \{\mathtt{a}, \mathtt{c}\}^*$. This is an example of an *oddpalindrome* language. The DPDA starts off by reading `a`s and `c`s interchangeably in the Read_1 state. After an `a` has been read, an `x` is pushed onto the stack. After reading a `c`, an `y` is pushed on the same stack. After a push, the machine loops back to the Read_1 state. Thus there are two loops from the Read_1 state: one to the Push_y state; and another to the Push_x state. The machine runs by altering between the two loops processing the first part of the input string until the separator `g` is encountered.

When `g` is read in state Read_1, the DPDA branches to the only *Pop* state. Whenever a `y` is popped the machine branches to Read_2 where a `c` must be read to continue processing. Similarly, whenever an `x` is popped, the machine branches to Read_3 where an `a` should be read to continue processing.

In each case, the machine loops back to the *Pop* state to continue the read-pop cycle until the stack is empty. When the stack is empty a Δ is popped and the machine moves to Read_4 where Δ is read, thereby checking that the input is empty. Thereafter the machine proceeds to the *ACCEPT* state.

The memory of CA_{T2}s is not as specialised as that of DPDAs, since the latter have the ability to store a variety of characters on the stack. Thus *oddpalindrome with a separator* cannot be accepted by CA_{T2}s.

[17]In Addendum A it is explored *how* to compare two counters against one another.

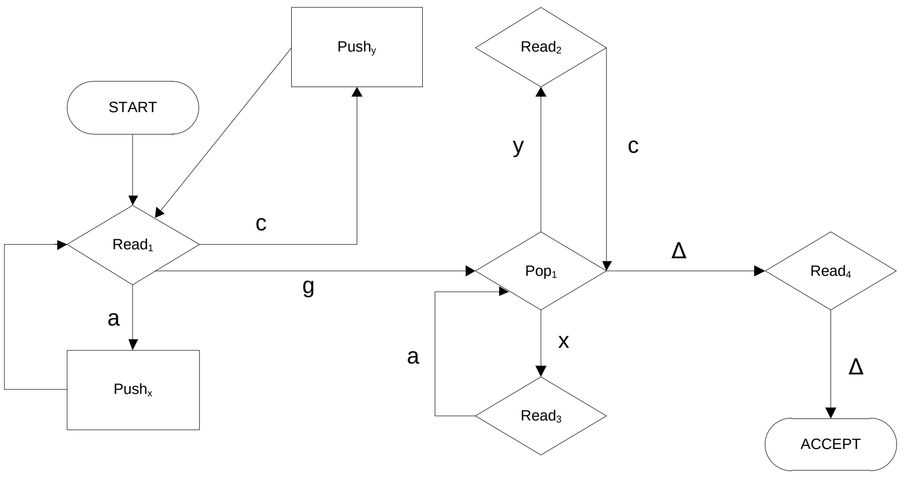


Figure 4.16: An example of a DPDA that accepts an oddpalindrome language with a separator, $L = \{w\texttt{g}(\texttt{reverse(}\ w\ \texttt{)}) \mid w \in \{\texttt{a}, \texttt{c}\}^*\}$.

It is consequently clear that CA$_{T2}$s can only accept a subset of the DCFLs.

A CA$_{T2}$ accepting L' where $L = \{\texttt{a}^\texttt{n}\texttt{c}^\texttt{n} \mid \texttt{n} \geq \texttt{0}\}$ is presented in Figure 4.17. L' is not a DCFL but a CFL that can be accepted by a non-deterministic PDA (Cohen [1997]). Although CA$_{T2}$s do not accept all the DCFLs they do accept a subset of CFLs that are not DCFLs too. Provision is made for Δ to be read from any state and all Δ edges lead to the final state. However, state q_5 can only be reached if the condition that $d_0 \neq d_1$ is met. Thus words belonging to L will not be accepted but all other words will be — i.e. L' will be accepted.

4.4.5.4 Regulating determinism in CA$_{T2}$s by Boolean expressions

Definition 4.4.6 defines a CA$_{T2}$ in a deterministic manner. The determinism is regulated by alphabet characters in Σ in that there is at most one outgoing edge for each character of the alphabet at each state. Another possibility that is not extensively studied here is to extend the current definition so that determinism is regulated by simple relational expressions. This will imply that there can be more than one outgoing edge on $a_i \in \Sigma$ from the same state, each annotated by its own relational expression. Consequently the same two counters (d_j and d_k) may occur on transition edges, in relational expressions as operands, more than once.

if $(d_j < d_k)$ then $\delta(a, q_j) = q_r$

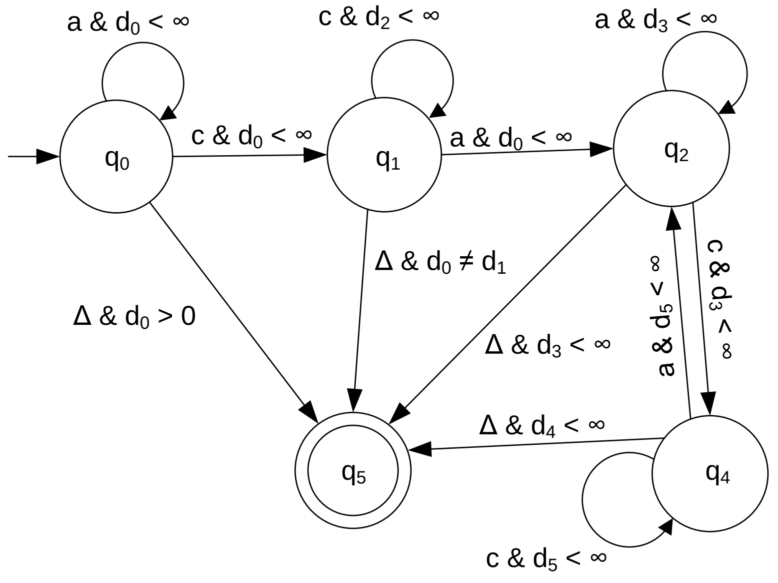


Figure 4.17: A CA_{T2} accepting L' where $L = \{\mathtt{a^n c^n} \mid \mathtt{n} \geq \mathtt{0}\}$.

if $(d_j > d_k + 5)$ then $\delta(a, q_j) = q_k$
else $\delta(a, q_j) = \perp$

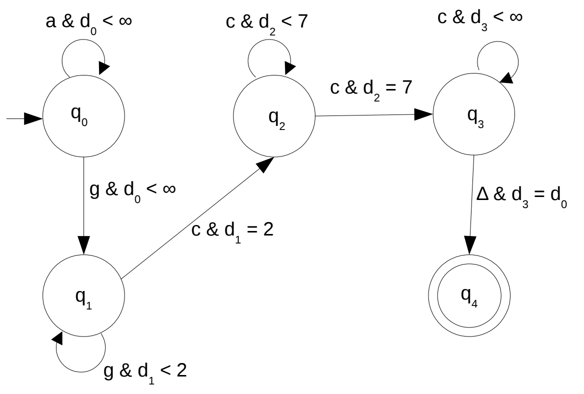
Thus, the Boolean expressions should be mutually exclusive (to prevent non-determinism). Such a machine will be referred to as a CA'_{T2}.

Such a CA'_{T2} machine accepting $L = \{\mathtt{a^{n-2}ggc^{n+5}} \mid \mathtt{n} \geq 2\}$ is depicted in Figure 4.18(a). Note that state q_2 has two transitions on character `c` but the non-determinism is removed because of the associated relational expressions. This machine has less states than the corresponding CA_{T2} that conforms to Definition 4.4.6. Such a CA_{T2} is depicted in Figure 4.18(b).

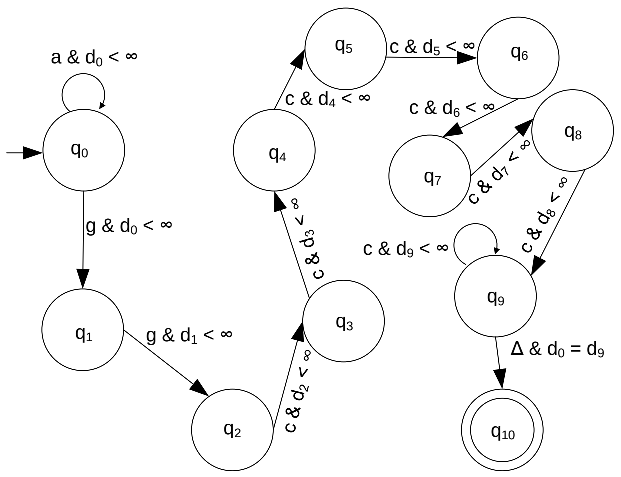
4.4.6 Counting automata type 1: $\text{CA}_{T1}s$

CA_{T1}s are defined almost as are CA_{T2}s except for a change in the definition of the threshold function φ. Formally,

Definition 4.4.7. *A CA_{T1} is represented by the same 7-tuple as a CA_{T2}, but the threshold function, φ is defined as follows:*
Let $D = \{d_{j1}, d_{j2}, \ldots d_{jk}\}$ be a set of counters of some set of states, $\{q_{j1}, q_{j2}, \ldots q_{jk}\}$.



(a) $L = \{\mathtt{a}^{\mathtt{n}-2}\mathtt{ggc}^{\mathtt{n}+5} \mid \mathtt{n} \geq 2\}$ being accepted by a $\mathrm{CA}_{\mathtt{T2}}^{'}$



(b) $L = \{\mathtt{a}^{\mathtt{n}-2}\mathtt{ggc}^{\mathtt{n}+5} \mid \mathtt{n} \geq 2\}$ being accepted by a $\mathrm{CA}_{\mathtt{T2}}$

Figure 4.18: A $\mathrm{CA}_{T2}^{'}$ and a CA_{T2} catering for the same language

Let $P1, P2$ be a partition of $Q \times \Sigma$ such that $P1 \cup P2 = Q \times \Sigma$, $P1 \cap P2 = \emptyset$ and $|P2| > 0$. Then

$$\begin{aligned} \varphi : Q \times \Sigma &\rightarrow \mathbb{N} \cup D \\ \textit{where } \varphi(q_i, a) &= \begin{cases} n & (q_i, a) \in P1 \textit{ and } n \in \mathbb{N} \\ d_j & (q_i, a) \in P2 \textit{ and } d_j \in D \end{cases} \end{aligned}$$

□

Thus, CA_{T1}, CA_{T2} and CA_{T3} differ only in respect of the number of counter-to-counter comparisons allowed, i.e. the size of $P2$. Whereas the definition of a CA_{T2} specifies at most one counter-to-counter comparison, the definition of CA_{T1} allows for more than one and the definition of CA_{T3} does not allow for any. Put differently:

- $|P2| = 0$ for CA_{T3}
- $0 \leq |P2| \leq 1$ for CA_{T2}
- $0 \leq |P2|$ for CA_{T1}

By a counter-to-counter comparison is meant precisely the same as before. To determine the destination state when a specific character of the the alphabet occurs in state q_i, the counter (d_i) of q_i may be compared against the counter, d_j, of some other state, q_j. If the relevant relational expression evaluates to true then a transition is made to that new state (not necessarily to state q_j), i.e. $\delta(q_i, a) = q_k$; and if the relational expression evaluates to false then $\delta(q_i, a) = \perp$. The definition of CA_{T1} still only permits one comparison per state (either counter-to-counter or counter-to-constant), but counter-to-counter comparisons may be specified at multiple states.

The abstract algorithm for testing whether a word is a member of the language defined by a CA_{T1}, is the same as the algorithm for CA_{T3}s — see Algorithm 4.4.2. CA_{T1}s can accept regular languages. They can also accept a subset of CFLs as well as a subset of CSLs, as will be illustrated below.

As mentioned in Chapter 2, CSLs are accepted by both 2PDAs and LBAs. To illustrate how 2PDAs, LBAs and CA_{T1}s process strings from a CSL, the example language $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}}\mathtt{g}^{\mathtt{n}} \mid \mathtt{n} \geq \mathtt{1}\}$ will be investigated.

4.4.6.1 The 2PDA case

A 2PDA accepting $\mathrm{L} = \{\mathtt{a}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}}\mathtt{g}^{\mathtt{n}} \mid \mathtt{n} \geq \mathtt{1}\}$ is presented in Figure 4.19.

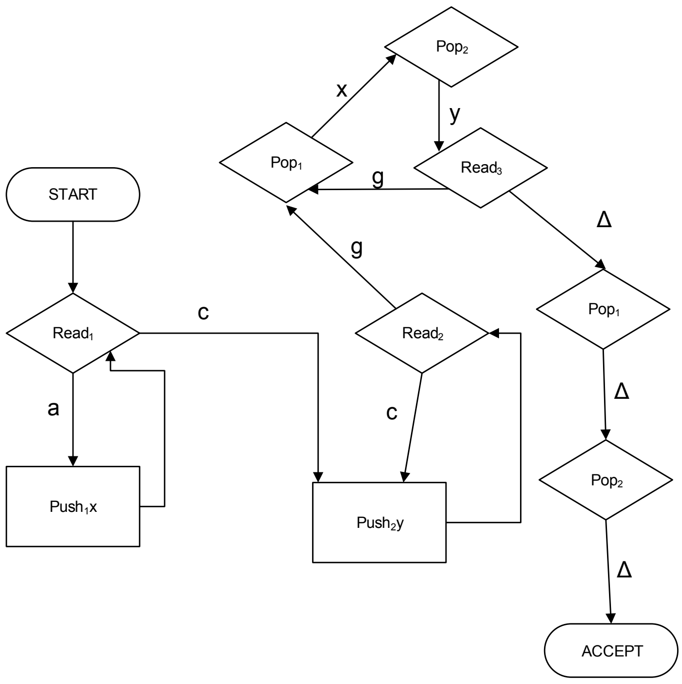


Figure 4.19: A 2PDA accepting $L = \{\mathtt{a}^\mathtt{n}\mathtt{c}^\mathtt{n}\mathtt{g}^\mathtt{n} \mid \mathtt{n} \geq \mathtt{1}\}$.

The 2PDA reads the first group of as in state *Read*$_1$. For each a read, an x is pushed onto stack$_1$. Thus a loop is introduced: *Read*$_1$ – a– *Push*$_1$x. Whenever a c is read in *Read*$_1$ we branch to *Push*$_2$y where we push a y for the first c read onto stack$_2$. A new loop is formed *Read*$_2$ - c - *Push*$_2$y. Thus for each c read in *Read*$_2$ a y is pushed onto stack$_2$. If we are in *Read*$_2$ and we read a g we branch to state *Pop*$_1$ popping an x which leads us to state *Pop*$_2$ where a y is popped. From *Pop*$_2$ we branch to state *Read*$_3$ where the remainder of the group of gs is read. A new loop is introduced: *Read*$_3$ - g - *Pop*$_1$x - *Pop*$_2$y. The number of xs on stack$_1$ is compared to the number of ys on stack$_2$ and all of these are compared to the number of gs on the input tape. Thereby the number of as, cs and gs are compared. Whenever Δ is encountered on the tape the end of the tape is reached, and both stacks should be empty too. Thus all words in L are accepted; all words in L' are rejected. In order to accept $L = \{\mathtt{a}^\mathtt{n}\mathtt{c}^\mathtt{n}\mathtt{g}^\mathtt{n} \mid \mathtt{n} \geq \mathtt{1}\}$ with a 2PDA we need 2 stacks each of size n and we have to execute 7n actions. Thus we have a run time complexity of O(n).

4.4.6.2 The LBA case

Figure 4.20 displays an LBA accepting $L = \{\mathtt{a}^\mathtt{n}\mathtt{c}^\mathtt{n}\mathtt{g}^\mathtt{n} \mid \mathtt{n} \geq \mathtt{1}\}$.

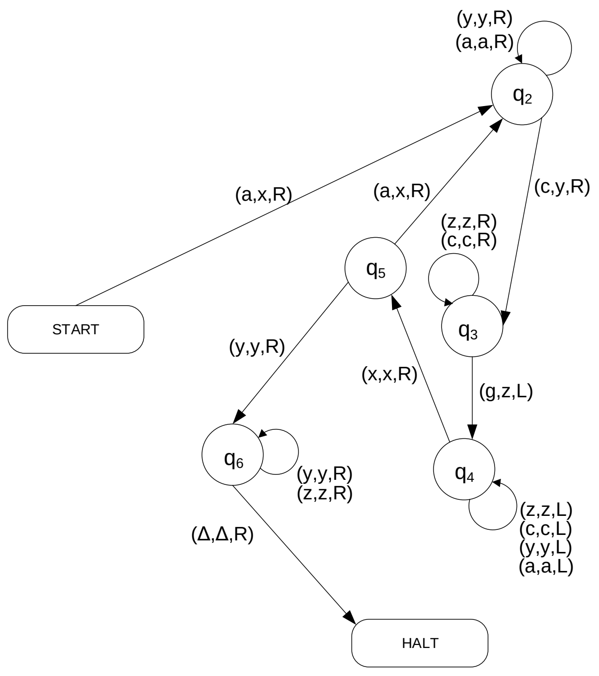


Figure 4.20: A TM accepting $L = \{\mathtt{a^n c^n g^n} \mid \mathtt{n} \geq \mathtt{1}\}$.

The processing of L by this LBA can be described as follows:

- An `a` is read in the *START* state in the first cell and an `x` is printed in the cell being read replacing the `a`. In q_2 the TM traverses through the remainder of the `a`s on the tape. Whenever the first `c` is encountered a `y` is printed on the tape and we move to state q_3.
- In q_3 we traverse through the remainder of the `c`s until a `g` is encountered. When, in state q_3 and a `g` is encountered, a `z` is printed on the tape. We then proceed to state q_4.
- In state q_4 we move to the left, iterating through all the `z`s, `c`s, `a`s and `y`s on the tape to date until the rightmost `x` is reached. We then move right and proceed to state q_5.
- If, in state q_5 an `a` is encountered, the LBA prints an `x`, moves right and proceeds to state q_2 where it iterates through the remainder of the `a`s as well as the `y`s printed on the tape and repeats the described process. Thus, states q_5, q_2, q_3 and q_4 are traversed until we encounter a `y` instead of an `a` in q_5.

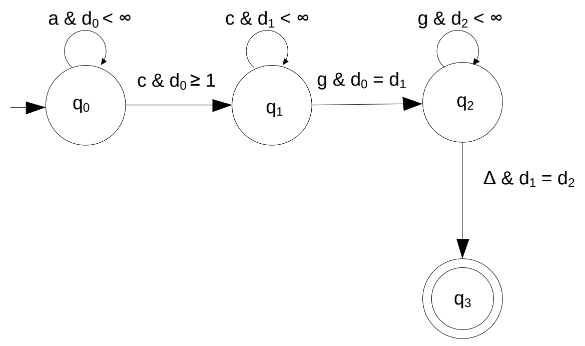


Figure 4.21: A CA$_{T1}$ accepting $L = \{\mathtt{a}^\mathtt{n}\mathtt{c}^\mathtt{n}\mathtt{g}^\mathtt{n} \mid \mathtt{n} \geq \mathtt{1}\}$.

- When this is the case, the number of as should be equal to the number of cs which should in turn be equal to the number of gs on the original input tape. The LBA traverses through the remainder of the tape reading only ys and zs.
- When Δ is encountered the end of the input string is reached and the LBA moves to the *HALT* state. It can easily be verified that this can only happen if the number of as, cs and gs match.

Note that the length of the input string is 3n. The length of the tape needed to determine if a word belongs to L is $3n + 1$ which is smaller than $(2(3n) + 2)$. Thus the length of the input tape needed for the machine depicted in Figure 4.20 is within the bounds of an LBA (Definition 2.3.21). Note also that the TM in Figure 4.20 has to traverse the input string more than once. In fact, the LBA traverses the input string $4n^2 + 6n + 2$ times. Thus the runtime complexity of the TM is $O(n^2)$.

4.4.6.3 The CA$_{T1}$ case

A CA$_{T1}$ accepting $L = \{\mathtt{a}^\mathtt{n}\mathtt{c}^\mathtt{n}\mathtt{g}^\mathtt{n} \mid \mathtt{n} \geq \mathtt{1}\}$ is depicted in Figure 4.21.

The CA$_{T1}$ has 4 states. In state $q_0 = \langle q_0, d_0 \rangle$ all the as of the input string are read and the counter d_0 is incremented for each a read until we read a c in state q_0. We then move to state $q_1 = \langle p_1, d_1 \rangle$ and counter d_1 is incremented for each c read until we read a g. At this stage, in the interest of minimising runtime, we need to compare the content of d_0 and d_1, which should be equal. In this case the presented CA$_{T1}$ proceeds. In state q_2 we read the remainder of the gs and the counter, d_2, is incremented each time that q_2 is entered until Δ is reached. The contents of d_1 and d_2 should be equal to reach the final state q_3. We traverse the

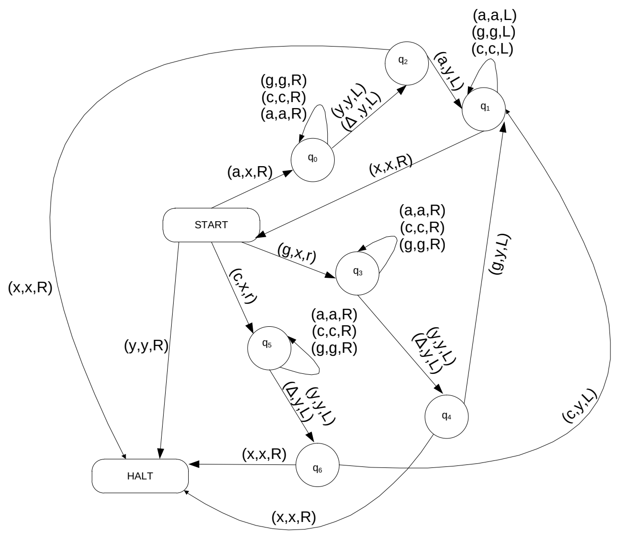


Figure 4.22: An LBA accepting palindromes over $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}\}$ with word lengths greater than 2. For an input string of length n a tape length of $n + 1$ is needed during processing.

input string once, to reach our final state. No additional stacks are used — only the respective counters.

4.4.6.4 CA$_{T1}$s cannot recognise all CSLs

In Subsection 4.4.5, a PDA accepting the language palindrome *with* a separator was presented. This example highlighted the fact that not all deterministic CFLs can be represented by a CA$_{T2}$.

The language consisting of all palindromes *without* a separator is a CSL. An LBA accepting this palindrome language is presented in Figure 4.22. Note, however, that a CA$_{T1}$ cannot be constructed to recognise palindromes (including palindrome with a separator, a CFL). The underlying reason why this is not possible is the same as the reason given in the previous section for CA$_{T2}$: counters do not store information about the alphabet but only about the number of occurrences.

4.4.6.5 Regulating determinism in CA$_{T1}$s by Boolean expressions

Just as CA$_{T2}$ machines can be extended to CA$'_{T2}$ machines, so can CA$_{T1}$s be extended to CA$'_{T1}$ machines. Again, this extension would allow for multiple transitions on a given character from a given state, the transitions being governed by

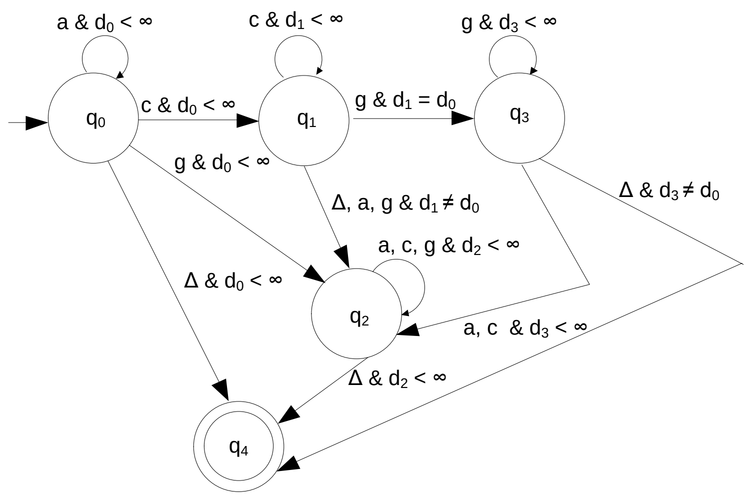


Figure 4.23: A CA'_{T1} machine accepting L' if $L = \{\mathtt{a^n c^n g^n} \mid \mathtt{n} \geq \mathtt{1}\}$.

simple mutually exclusive counter-to-counter based Boolean expressions. However, in this case, these expressions are allowed to involve comparisons between the current state's counter and the counters of any other states.

To briefly illustrate the idea, consider the complement, L', of the above CSL $L = \{\mathtt{a^n c^n g^n} \mid \mathtt{n} \geq \mathtt{1}\}$. Since it is known that the complement of a CSL is itself a CSL, L' is a CSL.

It is interesting to observe that a CA'_{T1} machine can be constructed to accept the complement, L'. Figure 4.23 illustrates such a CA'_{T1} machine. Of course, since L' is a CSL, it is also possible to construct both an LBA as well as a 2PDA to recognise L'. However, it is not possible to construct a CA_{T1} machine that recognises L'. Thus, CA'_{T1}s recognise a larger set of CSLs than CA_{T1}s.

4.5 Conclusion

CA_{T3}, CA_{T2} and CA_{T1} have been represented by the same 7-tuple. However, the threshold functions of these CAs differ. The definition of a CA_{T2} allows for at most one pair of counter-to-counter comparisons. The definition of CA_{T1} specifies any number of counter-to-counter comparisons. CA_{T3} does not allow for counter-to-counter comparisons at all.

CA'_{T2} and CA'_{T3} were introduced. It has been illustrated that CA'_{T2}s need, in some cases, less states than CA'_{T2} to accept the same CFL. However, CA'_{T1} can accept CSLs that cannot be accepted by CA$_{T1}$.

In Chapters 7 and 8 it is illustrated *how* CA$_{T3}$s can be employed to gather valuable statistical information within the context of TR detection.

Notions on introducing a CA (CA$_{T0}$) that has the ability to carry out the evaluation of some of the simple relational expressions have been explored. Clearly such a CA should have the ability to decrement counters too. No attempt is made to classify the languages defined by CA$_{T0}$ in terms of the Chomsky hierarchy. Ideas related to CA$_{T0}$ are included in Appendix A.

Chapter References

A. Arnove. Chomsky.Info. Online: http://www.chomsky.info/books.htm, 2018.

M. Cadilhac, A. Finkel, and P. McKenzie. Bounded Parikh automata . *International Journal of Foundations of Computer Science*, 23(8), 2012.

S. Chakraborty, L. T. X. Phan, and P. S. Thiagarajan. Event count automata: A state-based model for stream processing systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2490-7. doi: 10.1109/RTSS.2005.21. URL http://dx.doi.org/10.1109/RTSS.2005.21.

N. Chomsky. Three models for the description of language. *I.R.E. Transactions of Information Theory*, 2:113–124, September 1956.

N. Chomsky. *Aspects of the Theory of Syntax*. Cambridge, Massachusetts: MIT Press, 1965.

N. Chomsky. *Cartesian Linguistics: A Chapter in the History of Rationalist Thought. Third edition*. Cambridge University Press, 1966.

C. E. Cleland. The concept of computability. *Theoretical Computer Science*, 317:209–225, 2004.

D. I. A. Cohen. *Introduction to Computer Theory*. Wiley: New York, 2nd ed. edition, 1997.

M. D. Davis, R. Sigal, and E. J. Weyuker. *Computability, Complexity, and Languages*. Fundamentals of Theoretical Computer Science. Academic Press Professional, Inc., San Diego, CA, USA, 2nd edition, 1994. ISBN 0-12-206382-1.

D. Harel and Y. Feldman. *Algorithmics: The Spirit of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2004. ISBN 0-321-11784-0.

J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory languages and computation*. Addison-Wesley publishing company, 1979.

D. Hovland. Regular expressions with numerical constraints and automata with counters. *Theoretical Aspects of Computing - ICTAC 2009*, 5684:231–245, 2009a.

D. Hovland. Regular expressions with unordered concatenation and numerical constraints. *Theoretical Aspects of Computing - ICTAC 2009*, 5684:231–245, 2009b.

S. Hughes. Speech. The Pennsylvania Gazette, 2001.

W. Karianto. Adding monotonic counters to automata and transition graphs. *Developments in Language Theory*, 3572:308–319, 2005.

L. Krauss and S.M. Carroll. Science in the Dock, Discussion with Noam Chomsky. Chomsky.info. 2006-03-01, 03 2006.

K. Mahesh. *Theory of computation: a problem-solving approach.* John Wiley and Sons, Ltd, Publication, 1st edition, 2013. ISBN 978-1-118-54679-6(P/B).

M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, IInc. , 1 edition, 1967.

D. Pesetsky. Linguistics universals and universal grammar. Online: http://web.mit.edu/linguistics/people/faculty/pesetsky/Pesetsky_MITECS_Universals_UG.pdf, 2009.

P. Robinson. The Chomsky Problem. The New York Times, 1979.

A. Salomaa. *Computation and Automata* , volume 25 of *Encyclopedia of mathematics and its applications.* Cambridge University Press, 1985. ISBN 0 521 30245 5.

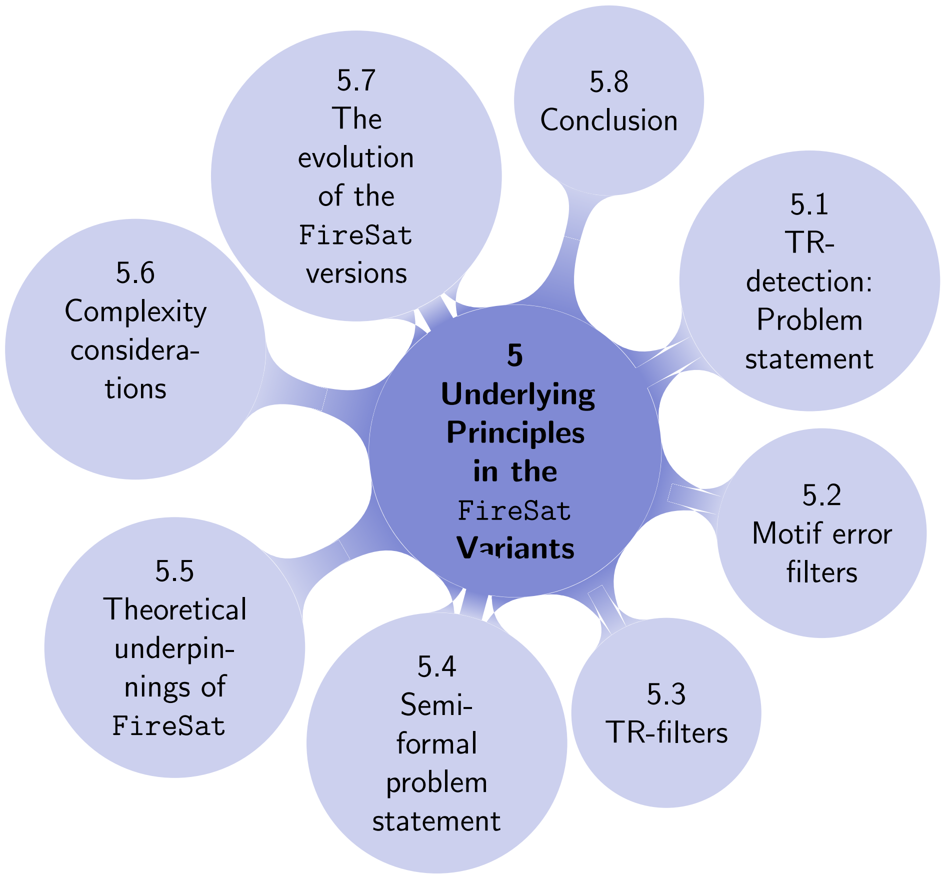
M.P. Schutzenberger. Finite counting automata. *Information and control*, 5(2), 1962.

M. Sipser. *Introduction to the Theory of Computation.* COURSE TECHNOLOGY, CENGAGE Learning, 2 edition, 2006. ISBN 978-0-619-21764-8.

C.M. Sperberg-McQueen. Notes on finite state automata with counters. Online: https://www.w3.org/XML/2004/05/msm-cfa.html, 2004.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. URL http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf.

H. Wilf, D. Berlenski, D. Perrin, R. Askey, and D. Foata. In Memoriam: Marcel-Paul Schützenberger, 1920-1996. *Electronic Journal of Combinatorics*, October 1996.


5.7 The evolution of the FireSat versions
5.8 Conclusion
5.6 Complexity considera-tions
5.1 TR-detection: Problem statement
5 Underlying Principles in the FireSat Variants
5.2 Motif error filters
5.5 Theoretical underpin-nings of FireSat
5.4 Semi-formal problem statement
5.3 TR-filters

5 Underlying Principles in the `FireSat` Variants

"When people see some things as beautiful, other things become ugly. When people see some things as good, other things become bad." ... Lao Tzu

---

This chapter introduces some of the theoretical underpinnings of `FireSat`. `FireSat` algorithms are designed to detect tandem repeats on DNA, specifically minisatellites and satellites. `FireSat`$_1$, presented in Chapter 6, is a concatenated pDFA-based[1] algorithm.

`FireSat`$_2$ and `FireSat`$_{2'}$ will be introduced in Chapter 7. `FireSat`$_2$ entails the cascading of pCA$_{T3}$s whereas `FireSat`$_{2'}$ cascades pNCA$_{T3}$s to detect TRs.[2]

A description of `FireSat`$_3$ then follows in Chapter 8. It relies on the composition of CA$_{T3}$s.

These algorithms extend the flexibility and accuracy benefits that were previously experienced with FA-based microsatellite detection algorithms, into the

[1] pDFAs refer to prototype DFAs, defined in Chapter 2, Definition 2.4.2.

[2] pCA$_{T3}$s refer to counting prototype DFAs, defined in Chapter 4, Definition 4.4.4. *Prototype non-deterministic counting automata type 3* (pNCA$_{T3}$) are almost the same as pCA$_{T3}$s, except that they are non-deterministic.

domains of minisatellite and satellite detection. To prevent a search space explosion implied by dealing with larger motifs, several techniques are employed during pattern recognition.

The remainder of this chapter is laid out as follows: The TR detection problem statement is given in Section 5.1. Motif error filters and TR filters are dealt with in Sections 5.2 and 5.3 respectively. Key aspects of the theoretical underpinnings for `FireSat` are discussed in Section 5.5. The evolution of `FireSat` is discussed in Section 5.7. This chapter is concluded in Section 5.8.

5.1 TR-Detection: problem statement

ATRs in genetic sequences were informally described in Definition 2.1.4. They are defined here in terms of the conventions that follow. Note that the definition of a TR, as found below, is similar to that of De Ridder [2010]. Here examples have been extended to illustrate mutations where $|\rho| = 6$ in Section 5.1.1.

A PTR with motif ρ that is repeated p times (where $p > 1$) is denoted by ρ^p. An ATR u that is derived from ρ^p, must also have the motif (ρ) as its prefix. Therefore it has the form $\rho u_2 \cdots u_p$ where each ATRE, u_k $(k = 2 \cdots p)$, is the result of having at most ε_{max} mutations on ρ. ε keeps track of the *actual* number of motif errors in the TRE. In theory, ε_{max} could be anywhere in the range $0 \leq \varepsilon_{max} \leq |\rho|$. The user is, however, given the option to limit the number of motif errors by entering a value for ε_{max}.[3]

In principle, then, an algorithm seeking TRs could rely on ε_{max} alone to determine when the end of a candidate string has been found. However, in practice, it is useful to rely on metrics that serve as additional filters. To improve the detection of possible TRs, the `FireSat` algorithms implement additional filters. These are discussed in Sections 5.2 and 5.3.

In `FireSat`, it is optional to use these filters. End-users may use them to enable `FireSat` to determine whether a string that has been found to be a possible TR at some point in the algorithm, should be output as such, or whether additional processing should occur to see if the string can be extended further to report a longer TR. Options for manipulating TR filters are discussed further in Section 5.3.

The next section, Section 5.1.1, discusses the types of mutations (motif errors) that are tolerated.

[3]If TRs, stretching over a range of motif lengths, are to be detected, $\varepsilon_{\%}$ is calculated and compared against $\varepsilon_{max_{\%}}$.

5.1.1 Motif errors: types of mutations tolerated

ε_{max} for an ATRE depends on $|\rho|$. Within the constraints specified by ε_{max}, `FireSat` tolerates the three conventional error-types in correspondence to the LD: deletions, insertions and mismatches.

To briefly illustrate types of motif errors per ATRE, consider an example based on the six character PTRE $\rho = \texttt{acgtac}$, where $\varepsilon_{max} = 2$. The authorised forms of each ATRE, u_k, where mutations have occurred will then be as follows:

1. The word ρ itself: $u_k = \texttt{acgtac}$ and $|u_k| = 6$.
2. The word ρ with the deletion of one nitrogenous base: $u_k \in \{\texttt{cgtac}, \texttt{agtac}, \texttt{actac}, \texttt{acgac}, \texttt{acgtc}, \texttt{acgta}\}$. Thus, in all these cases $|u_k| = 5$.
3. The word ρ with the mismatch of one base: $u_k \in \{\texttt{xcgtac|x:\{c,g,t\}}\} \cup \{\texttt{axgtac}|\texttt{x} : \{\texttt{a,g,t}\}\} \cup \{\texttt{acxtac}|\texttt{x} : \{\texttt{a,c,t}\}\} \cup \{\texttt{acgxac}|\texttt{x} : \{\texttt{a,c,g}\} \cup \{\texttt{acgtxc}|\texttt{x} : \{\texttt{c,g,t}\}\} \cup \{\texttt{acgtax}|\texttt{x} : \{\texttt{a,g,t}\}\}$. In all these cases $|u_k| = 6$.
4. The word ρ with the insertion of one base in front of any position: $u_k \in \{\texttt{xacgtac}|\texttt{x} : \{\texttt{a,c,g,t}\}\} \cup \{\texttt{axcgtac}|\texttt{x} : \{\texttt{a,c,g,t}\}\} \cup \{\texttt{acxgtac}|\texttt{x} : \{\texttt{a,c,g,t}\}\} \cup \{\texttt{acgxtac}|\texttt{x} : \{\texttt{a,c,g,t}\}\} \cup \{\texttt{acgtxac}|\texttt{x} : \{\texttt{a,c,g,t}\}\} \cup \{\texttt{acgtaxc}|\texttt{x} : \{\texttt{a,c,g,t}\}\}$. In all these cases $|u_k| = 7$.
5. The word ρ with the deletion of two nitrogenous bases: $u_k \in \{\texttt{gtac}, \texttt{ctac}, \texttt{cgtc}, \texttt{cgta}, \texttt{atac}, \texttt{agtc}, \texttt{agta},$ $\texttt{acac}, \texttt{actc}, \texttt{acta}, \texttt{cgac}, \texttt{agac}, \texttt{acgc}, \texttt{acga}, \texttt{acgt}\}$. Thus, in all these cases $|u_k| = 4$.
6. The word ρ containing two mismatches. $u_k \in \{\{\texttt{xygtac}|\texttt{x} : \{\texttt{c,g,t}\} \wedge \texttt{y} : \{\texttt{a,g,t}\}\} \cup \{\texttt{xcytac}|\texttt{x} : \{\texttt{c,g,t}\} \wedge \texttt{y} : \{\texttt{a,c,t}\}\} \cup \{\texttt{xcgyac}|\texttt{x} : \{\texttt{c,g,t}\} \wedge \texttt{y} : \{\texttt{a,c,g}\}\} \cup \{\texttt{xcgtyc}|\texttt{x} : \{\texttt{c,g,t}\} \wedge \texttt{y} : \{\texttt{c,g,t}\}\} \cup \{\texttt{xcgtay}|\texttt{x} : \{\texttt{c,g,t}\} \wedge \texttt{y} : \{\texttt{a,g,t}\}\} \cup \{\texttt{axytac}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{a,c,t}\}\} \cup \{\texttt{axgyac}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{a,c,g}\}\} \cup \{\texttt{axgtyc}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{c,g,t}\}\} \cup \{\texttt{axgtay}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{a,g,t}\}\} \cup \{\texttt{acxyac}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{a,c,g}\}\} \cup \{\texttt{acxtyc}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{c,g,t}\}\} \cup \{\texttt{acxtay}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{a,g,t}\}\} \cup \{\texttt{acgxyc}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{c,g,t}\}\} \cup \{\texttt{acgxay}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{a,g,t}\}\} \cup \{\texttt{acgtxy}|\texttt{x} : \{\texttt{a,g,t}\} \wedge \texttt{y} : \{\texttt{a,g,t}\}\}$. Whenever ATREs contain only mismatches it is clear that the length of the derived word will not deviate from the length of the original ρ. Clearly $|u_k| = 6$.
7. The word ρ with the insertion of two bases in front of any position: $u_k \in \{$ $\{\texttt{xyacgtac—x,y:\{a,c,g,t\}}\ \}\cup\{\texttt{xaycgtac—x,y:\{a,c,g,t\}}\ \}\cup\{\texttt{xacygtac—x,y:\{a,c,g,t\}}\ \}\cup$ $\{\texttt{xacgytac—x,y:\{a,c,g,t\}}\ \}\cup\{\texttt{xacgtyac—x,y:\{a,c,g,t\}}\ \}\cup\{\texttt{xacgtayc—x,y:\{a,c,g,t\}}\ \}\cup$ $\{\texttt{axycgtac—x,y:\{a,c,g,t\}}\ \}\cup\{\texttt{axcygtac—x,y:\{a,c,g,t\}}\ \}\cup\{\texttt{axcgytac—x,y:\{a,c,g,t\}}\ \}\cup$

{axcgtyac—x,y:{a,c,g,t} }∪{axcgtayc—x,y:{a,c,g,t} }∪{acxygtac—x,y:{a,c,g,t} }∪ {acxgytac—x,y:{a,c,g,t} }∪{acxgtyac—x,y:{a,c,g,t} }∪{acxgtayc—x,y:{a,c,g,t} }∪ {acgxytac—x,y:{a,c,g,t} }∪{acgxtyac—x,y:{a,c,g,t} }∪{acgxtayc—x,y:{a,c,g,t} }∪ {acgtxyac—x,y:{a,c,g,t} }∪{acgtxayc—x,y:{a,c,g,t} }∪{acgtaxyc—x,y:{a,c,g,t} }}.
In all cases, the length of the resulting string, $|u_k| = 8$.

It should be noted that all these words keep at least 4 bases from the original word ρ. The aforementioned manner of defining authorised forms of mismatches and deletions of u_k is derived from experimental observations cited by Rivals et al. [1995][4]. It has been endorsed by Benson [1999] as providing statistically relevant information. Clearly, a combination of mismatches, insertions and deletions may occur within an ATRE.

5.1.2 Dealing with ambiguities

As explained in Chapter 2, Section 2.5.4, ambiguities may arise when specifying the mutations that explain the difference between a potential TRE and a given motif. Note that the respective chapters that present the different versions of FireSat have sections explaining *how* the different FireSat versions deal with such ambiguities.

5.2 Motif error filters

During its searches, FireSat uses as the motif, ρ, a substring of the input string. It then checks progressively whether subsequent parts of the input string could be seen as contiguous TREs relative to this ρ.

1. $\varepsilon_{max\%}$
   FireSat allows the user to pre-specify $\varepsilon_{max\%}$, the maximum percentage of motif errors to be tolerated in a TRE. A counter of motif errors, ε, is maintained. If $\frac{\varepsilon}{|\rho|} \times 100 > \varepsilon_{max\%}$ then the string currently under consideration, say u_i, will be rejected as a TRE.

2. $\kappa_{max\%}$
   FireSat also allows the user to constrain the maximum percentage of motif errors that may occur *adjacently* within a TRE to be less than a prespecified percentage value, $\kappa_{max\%}$. It maintains a counter, κ, of such adja-

[4]However, Rival's approach allows insertions behind ρ. This convention is not used here. Note that FireSat$_1$ and FireSat$_2$ do not allow insertions behind ρ, whereas FireSat$_{2'}$ and FireSat$_3$ do.

cent motif errors. Again, if at any stage it is found that $\frac{\kappa}{|\rho|} \times 100 > \kappa_{max\%}$ then the string currently under consideration, u_i, will be rejected as a TRE.

3. $\sigma_{max\%}$
   The substring error is a percentage value measuring the number of errors, weighted according to error type. The measure is computed at appropriate points by `FireSat` and then compared against a user-specified threshold value $\sigma_{max\%}$. During processing $\sigma\% \leq \sigma_{max\%}$ should always hold.

   The value of $\sigma\%$ depends, *inter alia*, on penalties (or weights) allocated by the user to mismatches (p_m), deletions (p_d) and insertions (p_i). For a given motif, ρ, and a given substring that has been partitioned into the form $u = \rho u_2 \cdots u_p$, $\sigma\%$ on u is computed as:

$$\sigma\% = \frac{(n_d \times p_d) + (n_i \times p_i) + (n_m \times p_m)}{|\rho|} \times 100$$

   where n_d is the total number of deletions in u; n_i is the total number of insertions in u; n_m is the total number of mismatches in u.

   The *substring error filter* provides the user with the opportunity to fine-tune a search. For example, by allocating a high enough penalty value to insertions and deletions, the user can manipulate the `FireSat` search to report only PTRs and ATRs with mismatches.

4. *Match score*
   The *match score* contributes to finding the best alignment position of TREs. Details explaining *how* the *match score* (LC$_n$) is used within the `FireSat` context are postponed until Chapter 9.

5.3 TR-filters

`FireSat` computes two additional metrics: α, the number of ATREs that occur consecutively; and β, the total number of TREs. For the metric α a maximum value can be specified. For the metric β a minimum value can be indicated. These user-specified values for the relevant metrics are used by `FireSat` as threshold values in determining when a given substring can be regarded as a TR. Each of these metrics will now be considered in turn.

1. *Consecutive ATREs filter*: α_{max}
   The user has the option of entering a value denoted by α_{max}. This value indicates the maximum number of ATREs that are allowed to occur next to each other.

The counter α is maintained to record the total number of consecutive ATREs since the last PTRE. The counter is incremented whenever an ATRE has been determined, irrespective of the type of errors (or single error) that makes it an ATRE and not a PTRE. However, when a PTRE is read, then the value of α is again set to zero. The processing of a string will only proceed if $\alpha \leq \alpha_{max}$.

2. *The TR-length filter:* β_{min}
   To avoid the output of unwanted data, the user may indicate the minimum number of TREs that has to occur before a TR is output, denoted by β_{min}.

5.4 Semi-formal problem statement

A general semi-formal problem statement for the `FireSat` algorithms describing the detection of TRs introduced by a motif ρ of length $|\ell|$ allowing for at most ε motif errors, can be provided as follows:

Input: A genetic string, $gSeq$, a motif ρ of length ℓ, the maximum number of motif errors, ε, where $|gSeq| \geq 2 \times |\ell| - \varepsilon$.
Output: All positions $i : gSeq[i \cdots i + m - 1] = \rho.u_2.u_3 \ldots u_n$ where $u_i \approx \rho$, $\approx$ denotes approximate matching as governed by the user-specified parameters that were discussed above and $m = |\rho.u_2.u_3 \cdots u_n|$.

5.5 Theoretical underpinnings of `FireSat`

Recall from Chapters 2 and 4 that all types of FAs, whether DFAs, NFAs, CA$_{T3}$s or NCA$_{T3}$s, are regular language acceptors. Also recall that regular languages can, in turn, be defined by regexs. It can easily be demonstrated that a set of TRs is in fact a regular language. Consequently such a TR set can be described by a regex. An FA can, in principle, be defined to recognise this TR set.

This section illustrates how minisatellites and satellites can be defined by regexs.

Bio-informatics relies on a four-character alphabet, $\Sigma = \{$`a,c,g,t`$\}$. From the definition of a regex:

- `a`, `c`, `g` and `t` are regexs
- Any concatenation of one or more regex is a regex. Thus, any PTRE is a regex.
- For the PTRE, ρ, the regex representing the corresponding set of possible PTRs can be expressed as $\rho\rho(\rho)^*$.

To illustrate how regexs can describe sets of ATRs consider, for example, the ATR set that has ρ as a motif, where $|\rho| = 50$, and allow for at most one deletion per ATRE as a motif error. Let ρ_i denote the string resulting from a deletion in position i. A regex defining all the possible TREs is $(\rho + \rho_1 + \rho_2 ... + \rho_{50})$ and the set of resulting TRs can be described by the regex $\rho(\rho + \rho_1 + \rho_2 ... + \rho_{50})(\rho + \rho_1 + \rho_2 ... + \rho_{50})^*$.

Although it would be tedious to do, this example can clearly be generalised to cater for various combinations of mutation types as well as for an arbitrary number of motif errors. This would result in large, complex regexs and correspondingly large and complex FAs. The problem of defining an FA that recognises a specified set of TRs can therefore, in principle, always be solved.

However, to avoid the inevitable complexity and to keep the solution to such a problem as simple as possible, pDFAs were relied upon by `FireSat`$_1$; pCA$_{T3}$s by `FireSat`$_2$; pNCA$_{T3}$s by `FireSat`$_{2'}$ and a composition of CA$_{T3}$s by `FireSat`$_3$.

5.5.1 Defining regex templates

`FireSat`$_1$, `FireSat`$_2$ and `FireSat`$_{2'}$ divide ρ into contiguous substrings, denoted by υ_i, where $i = 1, \dots k$. The way in which this subdivision occurs differs in each of the respective cases. In the case of `FireSat`$_1$ and `FireSat`$_2$, $1 \leq |\upsilon_i| \leq 4$, for $i = 1, \dots k$, whereas for `FireSat`$_{2'}$, $|\upsilon_i|$ is always exactly 1.

Before considering the algorithmic details of these `FireSat` variants, it will be convenient to consider four types of repeats: mono-repeats, di-repeats, tri-repeats and quad-repeats. A specific υ_i could serve as the motif for one of these types of repeats. If υ_i is repeated two or more times, the result is a *perfect* mono-, di-, tri- or quad-repeat, depending on $|\upsilon_i|$. Alternatively, if we let e_{max} refer to the maximum allowable number of motif errors in υ_i, this could serve as the basis for defining *approximate* mono-, di-, tri- or quad-repeats, again depending on $|\upsilon_i|$. Some examples follow:

- *Mono-repeats*: TR_a = {`aa, aaa, aaaa, ...`} is an example of a perfect mono-repeat language consisting of repetitions of one single character. A regex that generates this language is: `aa(a)`*. A regex *template* `WW(W)`* uses the variable `W` as a place holder[5] for an arbitrary element of Σ. This regex template may be said to *induce* a specific regex when `W` is replaced with an element of Σ.

- *Di-repeats:* TR_{ac} = {`acac, acacac, acacacac...`}, is a perfect di-repeat language generated by the regex `acac(ac)`*. The regex template,

[5]In the current context, capital `W` is used to conform with the convention used for non-terminals in CFGs. These non-terminals may be considered to be variables.

`WXWX(WX)`*, induces any language of perfect di-repeats. The regex template

    WX(WX + W + X + W′X + WX′ + ZWX + WZX)
    (WX + W + X + W′X + WX′ + ZWX + WZX)*,

where `W,W′,X,X′,Z` $\in$ {`a,c,g,t`} and `W′` $\neq$ `W`, `X′` $\neq$ `X`, could be used to induce any di-repeats where e_{max} is 2 and $e_{max\%}$ is 50.

- *Tri-repeats:* Figure 5.1 shows a DFA, $DFA_P(\mathtt{acg}, \mathtt{0})$, that accepts the perfect tri-repeat `acgacg(acg)`*. Figure 5.2 shows a DFA, $DFA_M(\mathtt{acg}, \mathtt{1})$, that accepts all words defined by $v_i(v_i + u_i)^*$ where $v_i = \mathtt{acg}$ and where u_i is a string derived from v_i that contains at most 1 mismatch. Note that sink states are introduced to cater for words not defined by the respective FAs. (See x_9 in Figure 5.1 and q_7 in Figure 5.2.) Note further that the presented FAs are cyclic. Fire$_\mu$Sat$_1$ relies on such cyclic FAs for microsatellite detection.

  Perfect tri-repeats can be induced from the regex template `WXYWXY(WXY)`*. If a 33,3% motif error is allowed, then the associated tri-repeats are induced by the regex template:

      WXY(WXY + WX + XY + WY + W′XY + WX′Y + WXY′ +
                    ZWXY + WZXY + WXZY)
      (WXY + WX + XY + WY + W′XY + WX′Y + WXY′ +
                    ZWXY + WZXY + WXZY)*

  where `W,W′,X,X′,Y,Y′,Z` $\in$ {`a,c,g,t`} and `W′` $\neq$ `W`, `X′` $\neq$ `X` and `Y′`, $\neq$ `Y`.

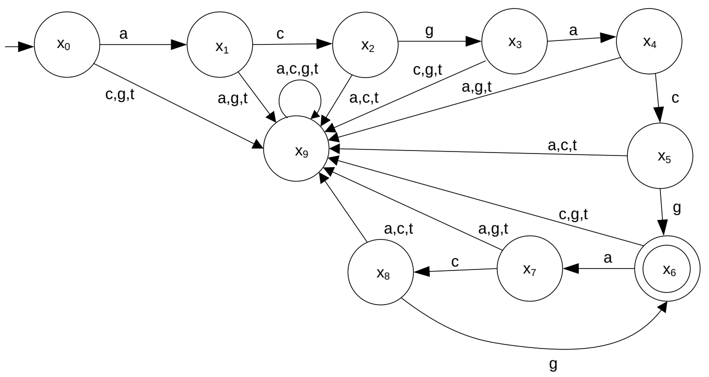


Figure 5.1: A cyclic FA accepting the perfect tri-repeat nucleotide sequences defined by `acgacg(acg)`*.

- *Quad-repeats:* The regex template `WXYZWXYZ(WXYZ)`* where `W,X,Y,Z` $\in$ {`a,c,g,t`} generates all perfect quad-repeats. The regex template generating general quad-repeats where $\varepsilon_{max\%}$ is 25, is:

      (WXYZ)(WXYZ + W′XYZ + WX′YZ + WXY′Z + WXYZ′ + XYZ + WYZ +

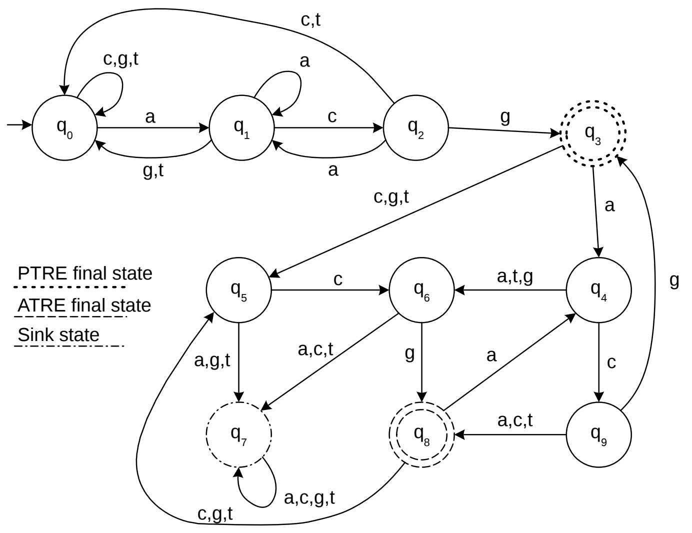


Figure 5.2: A cyclic FA that accepts $v =$ `acg` and $v(v + u_i)^*$, where u_i is a string derived from v that contains at most 1 mismatch error.

```
          WXZ + WXY +QWXYZ + WQXYZ + WXQYZ + WXYQZ )
(WXYZ + W′XYZ + WX′YZ + WXY′Z + WXYZ′ + XYZ + WYZ +
          WXZ + WXY + QWXYZ + WQXYZ + WXQYZ + WXYQZ)*
```

where W,W′,X,X′,Y,Y′,Z,Z′,Q $\in$ {a,c,g,t} and W′ $\neq$ W, X′ $\neq$ X, Y′ $\neq$ Y and Z′ $\neq$ Z.

In principle, other general expressions can be constructed reflecting error rates such as 50%, 75%, etc.

Note that the respective languages of the above regexs define *microsatellites*. Also note that, in each case, a family of these regexs can be represented by a regex template.

By applying the rules and theory provided in the previous sections of this chapter and by using the principles as described for defining mono-, di-, tri- and quad-repeats, regexs (or, indeed, regex templates) generating TRs of *any* length can be defined. Further discussion about how to do this will be given below. Once such regexs are available, however, the corresponding FAs need to be constructed.

In principle, software tool-kits such as the Fire Engine (Watson [1994]) could be used for this purpose. Such tool-kits typically take a regex based on an arbitrary alphabet as input and construct an FA that accepts the corresponding regex generated language.

However, since the alphabet size of regexs describing genomic strings is exactly four, specially tailored techniques were explored that aim to construct language-equivalent automata more effectively.

Before indicating how the different versions of `FireSat` evolved, Section 5.5.2 presents the parts of Kleene's theorem relevant to $\texttt{FireSat}_2$. Kleene proved these theorems in 1956 (Cohen [1997]).

5.5.2 Kleene's Theorem underpins FireSat$_2$

Kleene's theorem (Cohen [1997]) provides the theoretical underpinning of $\texttt{FireSat}_2$. The theorem is stated in various parts. Only *Part 3* itself and *Rule 2* stated in *Part 3* are relevant to $\texttt{FireSat}_2$. These are as follows:

Part 3:

> Every language that can be defined by a regex can also be defined by an FA.

Rule 2 of Part 3:

> If there is an FA called FA_1 that accepts the language defined by the regex r_1 and there is an FA called FA_2 that accepts the language defined by the regex r_2, then there is an FA, say FA_3, that accepts the language defined by the regex $r_1 + r_2$.

The proof of Rule 2 was presented in Definition 2.3.10. In addition, Example 2.3.11 illustrated the application of this rule. The proof and example make it clear that in general, the final states of FA_1 and FA_2 become final states of the resulting FA, FA_3.

In principle it is therefore possible to partition the final states of an FA into the final states of the two summed FAs. This principle holds for any states (not only final states), i.e. any state labelled in a specific way in an operand FA of the sum operation will map to a state in the summed FA.

$\texttt{FireSat}_2$ applies this principle to various pCA_{T3}s where states in the operand pCA_{T3}s may be labelled as mismatch, insertion or deletion states, and will retain those labels in the summed pCA_{T3}.

Cascading FAs
The way in which the TR-detection problem is addressed in this thesis enabled me to simplify the product rule inside Kleene's theorem to a cascading operation given in Definition 2.4.4. Note that $\texttt{FireSat}_1$, $\texttt{FireSat}_2$ as well as $\texttt{FireSat}_{2'}$ rely on the cascade operation. Recall that the cascade operation as given in Definition 2.4.4 relates only to ADFAs. Cascading say pDFA_s and pDFA_t representing string

sets s and t respectively, results in the set of strings such that each element has a prefix of a string in s and a suffix of a string in t. Using this insight, prototype automata can be cascaded to construct ADFAs and *non-deterministic counting automata* (NCA_{T3}s) that accept TREs of any motif length.

5.6 Complexity considerations

Conventionally, algorithmic studies try to trace the time and space complexity both theoretically and empirically. In this study such considerations were under-emphasized. This is because the primary concern of the research presented in this thesis was to determine whether automata can be employed to manifest a certain accuracy against other existing algorithms. Thus time and space efficiency was not of primary interest. Deeper investigations into time and space complexity is left for future research. Chapter 9 takes up the theme of comparing accuracy.

5.7 The evolution of the `FireSat` versions

In the cause of this study various versions of `FireSat` were evolved and incrementally tested for accuracy. The evolution of the four `FireSat` versions is outlined below:

$\texttt{FireSat}_1$
For $\texttt{FireSat}_1$ it is shown how mono-, di-, tri- and quad-pDFAs can be concatenated in different ways to construct ADFAs defining various genomic strings. In particular, this approach can be used to construct an ADFA that recognises the language describing TREs associated with motifs whose lengths lie in a pre-specified range. In this context, pDFAs are investigated as building blocks to construct acyclic FAs detecting sets of minisatellite and satellite TREs. It will be seen that $\texttt{FireSat}_1$ is a relatively fast, brute force algorithm, detecting TRs — but not very accurately. The idea of cascading mono-, di-, tri- and quad-automata is refined for $\texttt{FireSat}_2$.

$\texttt{FireSat}_2$
In contrast with $\texttt{FireSat}_1$, $\texttt{FireSat}_2$ cascades mono-, di-, tri- and quad-pCA_{T3}s to construct a NCA_{T3}. Although $\texttt{FireSat}_2$ is very accurate, it will be seen that in practice the implementation of $\texttt{FireSat}_2$ may be impractical. The details of *why* this is the case will become apparent in Chapter 7. When $\texttt{FireSat}_2$ is seeking the next TRE during TR detection, and if it happens to deem the next nucleotide in the source string to be an insertion, then it requires that all subsequent nucleotides in the source string must also be construed as insertions

until the next perfect match within the destination string is found. Subject to this constraint, the selection of the next TRE is based on the LD between the source (input) and destination (motif) string. Because of this constraint, $\texttt{FireSat}_2$ is said to rely on a Levenshtein-based distance (LBD) rather than on the LD as such.

$\texttt{FireSat}_{2'}$
In contrast to $\texttt{FireSat}_1$ and $\texttt{FireSat}_2$, $\texttt{FireSat}_{2'}$ cascades mono-pNCA$_{T3}$s *only*. $\texttt{FireSat}_{2'}$ calculates the LD between a source and a destination string and can achieve a high degree of TR-detection accuracy. Note that $\texttt{FireSat}_{2'}$ has not been fully implemented.

$\texttt{FireSat}_3$
$\texttt{FireSat}_3$ utilizes a composition of CA$_{T3}$s to investigate a TR-detection approach that differs extensively from that of its predecessors. The detection results obtained by $\texttt{FireSat}_3$ corresponds to the detection results of $\texttt{FireSat}_{2'}$.

5.8 Conclusion

In De Ridder [2010], I reported on my research into three FA-based *microsatellite* detection algorithms called $\texttt{Fire}_\mu\texttt{Sat}_1$, $\texttt{Fire}_\mu\texttt{Sat}_2$ and $\texttt{Fire}_\mu\texttt{Sat}_3$ respectively[6]. These algorithms differ fundamentally from FireSat. $\texttt{Fire}_\mu\texttt{Sat}_1$ and $\texttt{Fire}_\mu\texttt{Sat}_3$ implement cyclic deterministic CA$_{T3}$s. In contrast, $\texttt{FireSat}_1$ relies on cascaded, acyclic pDFAs. Even though $\texttt{Fire}_\mu\texttt{Sat}_2$ is more like FireSat in that it relies on acyclic FAs, it also differs fundamentally from the FireSat algorithms.

To detect microsatellites $\texttt{Fire}_\mu\texttt{Sat}_2$ iterates, for a certain length, through different independent ADFAs. $\texttt{FireSat}_1$ utilises some of the principles introduced by $\texttt{Fire}_\mu\texttt{Sat}_2$. Specifically, $\texttt{FireSat}_1$ iterates for a certain substring of a potential TRE, through different pDFAs when selecting the next pDFA to be cascaded.

Note that $\texttt{Fire}\mu\texttt{Sat}_2$ does not utilise the cascade operation. Similar to $\texttt{Fire}_\mu\texttt{Sat}_1$ and $\texttt{Fire}_\mu\texttt{Sat}_3$, $\texttt{FireSat}_2$ and $\texttt{FireSat}_{2'}$ utilise CA$_{T3}$s. However, deterministic cyclic CA$_{T3}$s were used for microsatellite detection in the case of the two previously mentioned $\texttt{Fire}_\mu\texttt{Sat}$ versions. It will be seen that $\texttt{FireSat}_2$ and $\texttt{FireSat}_{2'}$ construct acyclic, non-deterministic counting automata to detect TRs on DNA. The approach of $\texttt{FireSat}_3$, where a composition of CA$_{T3}$s are used to detect TRs, differs significantly from all its predecessors.

Next, Chapter 6 provides an in-depth discussion of $\texttt{FireSat}_1$ that includes the selection of pDFAs within the $\texttt{FireSat}_1$ context.

[6] An implementation of $\texttt{Fire}_\mu\texttt{Sat}_2$ is available at www.dna-algo.co.za.

Chapter References

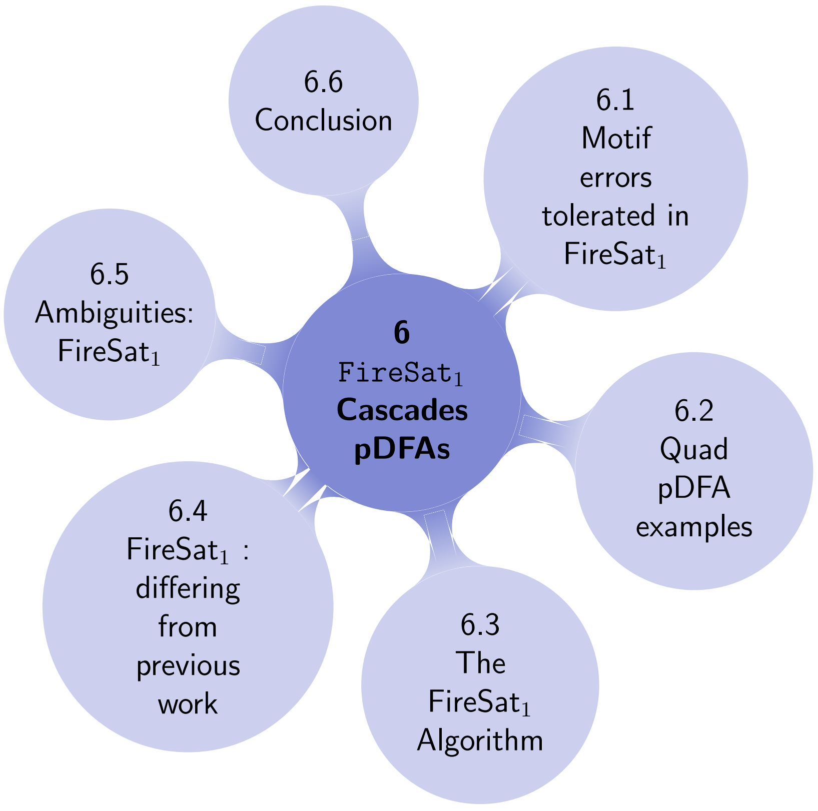
G. Benson. Tandem Repeats Finder. *Nucleic Acids Research*, 27(2):573 – 580, November 1999.

D. I. A. Cohen. *Introduction to Computer Theory.* Wiley: New York, 2nd ed. edition, 1997.

C. De Ridder. Flexible Finite automata-based algorithms for detecting microsatellites in DNA. Master's thesis, Department of Computer Science, University of Pretoria, July 2010.

E. Rivals, J. P. Delahaye, O. Delgrange, and M. Dauchet. A first step toward chromosome analysis by compression algorithms. In *Proceedings of the First International IEEE Symposium on Intelligence in Neural and Biological Systems*, pages 233–239, 1995.

B. W. Watson. The design and implementation of the FIRE Engine: A C++ toolkit for Finite Automata and Regular Expressions. Online: http://alexandra.tue.nl/extra1/wskrap/public html/9411065.pdf, 1994.


6.6 Conclusion
6.1 Motif errors tolerated in FireSat$_1$
6.5 Ambiguities: FireSat$_1$
6 FireSat$_1$ Cascades pDFAs
6.2 Quad pDFA examples
6.4 FireSat$_1$: differing from previous work
6.3 The FireSat$_1$ Algorithm

6

FireSat$_1$ Cascades pDFAs

“Life will give you whatever experience is most helpful for the evolution of your consciousness. How do you know this is the experience you need? Because this is the experience you are having at this moment .”... Eckhart Tolle

---

This chapter explains various features of FireSat$_1$. Firstly, Section 6.1 reviews the constraints imposed on the kinds of motif errors tolerated by FireSat$_1$. Thereafter Section 6.2 illustrates how pDFAs can be associated with different mutations. Finally, Section 6.3 outlines the overall algorithmic logic followed in FireSat$_1$, including the high-level pseudo-code. Empirical results with respect to the detection ability of FireSat$_1$ are deferred to Chapter 9. Section 6.4 indicates how FireSat$_1$ differs from previous work. Thereafter Section 6.5 explains how FireSat$_1$ deals with ambiguities. Section 6.6 concludes this chapter.

6.1 Motif errors tolerated in FireSat$_1$

The FireSat$_1$ user has to select several parameters. The first of these is the maximum percentage of motif errors ($\varepsilon_{max\%}$) to be tolerated per TRE.

j	Description	*pDFA*
1	perfect	$pDFA_P(v,0)$
2	1 mismatch	$pDFA_M(v,1)$
3	1 deletion	$pDFA_D(v,1)$
4	1 insertion	$pDFA_I(v,1)$
5	2 mismatches	$pDFA_M(v,2)$
6	3 mismatches	$pDFA_M(v,3)$
7	4 mismatches	$pDFA_M(v,4)$

Table 6.1: Quad-pDFAs permitted in FireSat$_1$.

Recall from Chapter 5 that ρ, the motif being used to identify TREs, is partitioned into the substrings $v_1v_2\dots v_k$, where $|v_i|$ is between 1 and 4. FireSat$_1$ uses a pDFA to represent each v_i as well as several more pDFAs, one to represent each *allowable* variant of v_i.

Practical constraints limit the number of errors and type of errors that can be allowed per pDFA. What these constraints are and how they are met depends on whether the pDFA is a quad-, tri-, di- or mono-pDFA. The matter is summarised in Tables 6.1, 6.2, 6.3 and 6.4. Each row of each table is associated with a pDFA. The first column of a table (with heading j) gives an index into a row of the table. Entries in the second column briefly describe the pDFA associated with the respective row. The pDFA is named in the third column[1].

The ordering of these pDFAs is important because it indicates the order in which FireSat$_1$ selects the different pDFAs indicated in each row when seeking TREs.

Consider Table 6.1. Suppose that FireSat$_1$ needs to test whether some input string, say u, can be regarded as a match or an approximate match of some string v, where $|v| = 4$. Then FireSat$_1$ first uses $pDFA_P(v,0)$ to check whether a perfect match is found. If not, it checks $pDFA_M(v,1)$ to see if u differs from v by 1 mismatch; if not, it checks $pDFA_D(v,1)$ to see if u differs from v by 1 deletion, then $pDFA_I(v,1)$ to see if u differs from v by 1 insertion (but not at the end of u), etc. Note that the order of checking is therefore firstly for 1 mismatch, then for 1 deletion , then for 1 insertion, and *thereafter* for 2, 3 or 4 mismatches.

Tables 6.2, 6.3 and 6.4 are used in a similar fashion if $|v| = 3$, $|v| = 2$ or $|v| = 1$.

[1] Recall names assigned to pDFAs in the text just after Definition 2.4.2. In general, the pDFA named as $pDFA_X(v,e)$ accepts v as well as exactly e mutations of type X in v, where $X \in \{P,M,D,I\}$ represents the type of mutation (Perfect, Mismatch, Deletion or Insertion). Also note that for FireSat$_1$ for each pDFA e represents *exactly* the number of motif errors catered for. This is in contrast with e of FireSat$_2$ where pCA$_{T3}$s are concatenated, each catering for v. Within the context of FireSat$_2$, e represents the maximum number and not the exact number of motif errors tolerated per v.

j	Description	*pDFA*
1	perfect	$pDFA_P(\upsilon, 0)$
2	1 mismatch	$pDFA_M(\upsilon, 1)$
3	1 deletion	$pDFA_D(\upsilon, 1)$
4	1 insertion	$pDFA_I(\upsilon, 1)$
5	2 mismatches	$pDFA_M(\upsilon, 2)$
6	3 mismatches	$pDFA_M(\upsilon, 3)$

Table 6.2: Tri-pDFAs permitted in FireSat$_1$.

j	Description	*pDFA*
1	perfect	$pDFA_P(\upsilon, 0)$
2	1 mismatch	$pDFA_M(\upsilon, 1)$
3	1 deletion	$pDFA_D(\upsilon, 1)$
4	1 insertion	$pDFA_I(\upsilon, 1)$
5	2 mismatches	$pDFA_M(\upsilon, 2)$

Table 6.3: Di-pDFAs permitted in FireSat$_1$.

j	Description	*pDFA*
1	perfect	$pDFA_P(\upsilon, 0)$
2	1 mismatch	$pDFA_M(\upsilon, 1)$

Table 6.4: Mono-pDFAs permitted in FireSat$_1$.

The way in which the partitioning $\rho = \upsilon_1, \upsilon_2 \cdots \upsilon_{k+1}$ takes place in `FireSat`$_1$ is very specific: $|\upsilon_i| = 4$, for $i = 1, \ldots k$ and $|\upsilon_{k+1}|$ may be 0, 1, 2 or 3.

The next section illustrates the quad-pDFAs in the last column of Table 6.1 when $\upsilon =$ `acgt`.

6.2 Quad pDFA examples

To illustrate how pDFAs can recognise different mutation types, we consider now the case of quad-pDFAs.

- Figure 6.1 shows four different quad pDFAs, each accepting $\upsilon_i =$ `acgt`, but also catering for exactly 1, 2, 3 or 4 mismatches, respectively.
- Figure 6.2 shows two different quad pDFAs. Again, each accepts υ_i, but also caters for exactly 1 or exactly 2 insertions, respectively.
- Finally, Figure 6.3 shows a quad pDFA accepting υ_i, but also catering for exactly 1 deletion with respect to υ_i.

Each of the figures is now discussed in a little more detail. The pDFAs in the figures can generally be thought of as *pDFA$_X$(*`acgt`*,e)* for some mutation type X and some number of errors e. However, in certain instances, the pDFA recognises only a string with e mutations on `acgt` and not `acgt` itself.

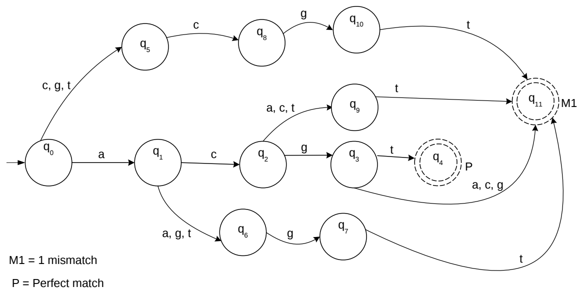
6.2.1 Mismatches

Figure 6.1(a) is a pDFA accepting $\upsilon =$ `acgt` allowing for at most one mismatch—i.e. $e = 0$ or $e = 1$. If e is anything else, the pDFA will clearly crash. Two types of final states can be distinguished. State q_4 (labelled with an additional P next to the state) accepts exactly $\upsilon =$ `acgt` and q_{11} (labelled additionally with M1) accepts all mismatches derived from $\upsilon =$ `acgt` where $e = 1$.

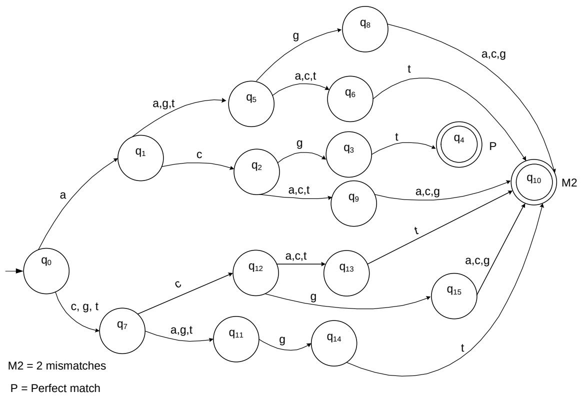
Figures 6.1(b), 6.1(c) and 6.1(d) cater for zero or two, zero or three and zero or four mismatches, respectively, and similar final state labels are used.

6.2.2 Insertions

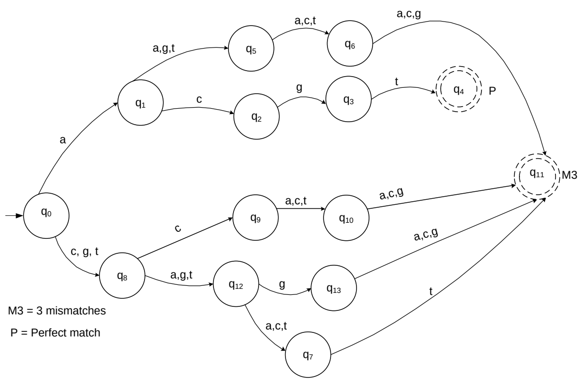
Table 6.1 indicates that `FireSat`$_1$ allows one insertion error only for $|\upsilon_i| = 4$. Figure 6.2(a) illustrates that a pDFA accepting $\upsilon_i =$ `acgt` and u_i deduced from υ_i where $e = 1$, and the motif error (e) is an insertion, requires at least 15 states.



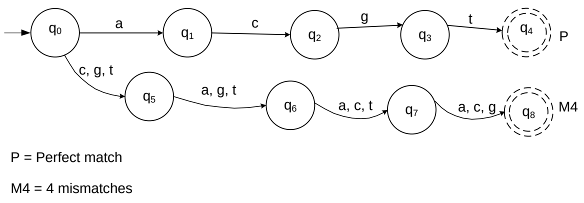
(a) $pDFA_M(acgt, 1)$.



(b) $pDFA_M(acgt, 2)$.



(c) $pDFA_M(acgt, 3)$.



(d) $pDFA_M(acgt, 4)$.

Figure 6.1: Quad-pDFA mismatch acceptors, where v_i = `acgt` and $e = 0, 1, 2, 3$ or 4.

Whenever there is a perfect match, state q_4 will be reached. All insertions, u_i, derivable from v_i when $e = 1$ will reach state q_{11}. Note that provision is made for a single insertion in front of v_i but not behind v_i.

It is interesting to note that insertions derived from the same v_i, but with $e = 2$ requires a pDFA with 29 states. Figure 6.2(b) depicts such a pDFA. Perfect matching words (v_i) will reach state q_4 whereas words containing two insertion errors end up at state q_9.

6.2.3 Deletions

Again, referring to Table 6.1, recall that FireSat$_1$ allows for a maximum of one deletion when $|v_i| = 4$. Figure 6.3 depicts a pDFA making provision for exactly one deletion in $v_i =$ acgt. It is clear that the cascading state, q_3, of Figure 6.3 is reached for an input string u_i derived from $v_i =$ acgt where $e = 1$. Note that this pDFA does *not* accept $v_i =$ acgt itself.

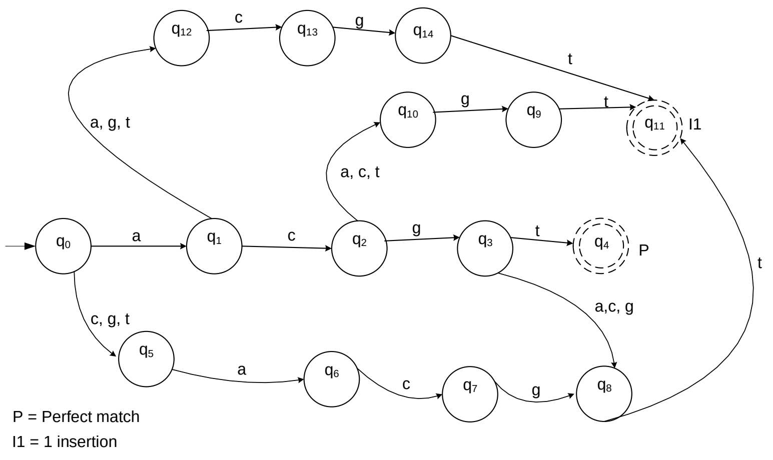
6.2.4 Concluding remarks

The quad-pDFAs, exemplified for a given string, serve to illustrate the following points:

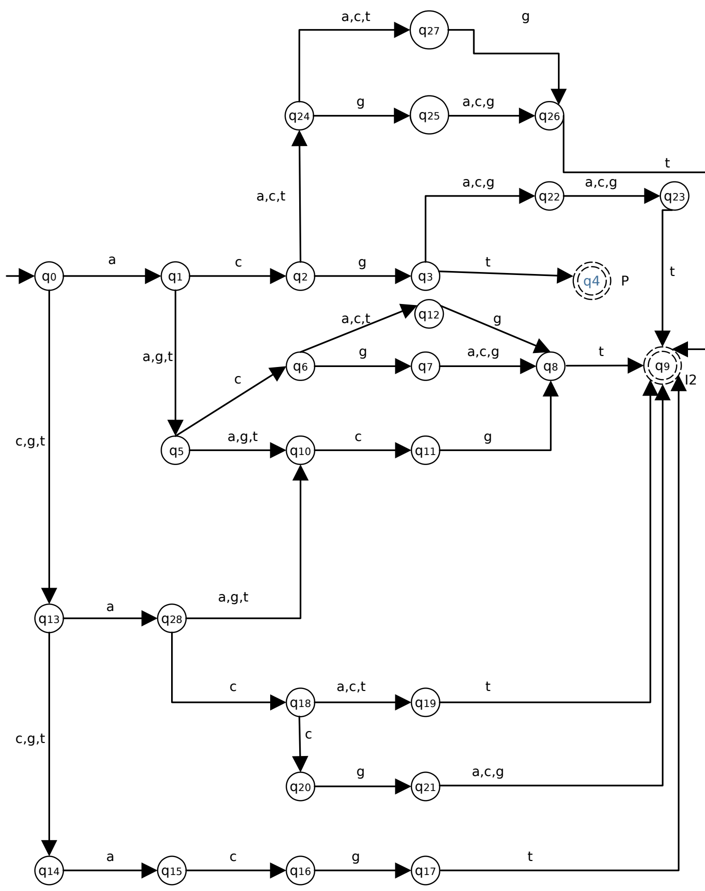
1. The shape of the various pDFAs remain exactly the same for *any* string of length 4 that is drawn from a 4-character alphabet. Thus, a quad-pDFA *template* for each row in Table 6.1 can be constructed as a once-off exercise and then subsequently decorated as needed for each new string v of length 4.

2. Clearly, given a string of length 1, 2 or 3 from a 4-character alphabet, it would be relatively easy to devise mono-, di- and tri-pDFAs that conform to the requirements of Tables 6.2, 6.3 and 6.4 respectively. Again, in these cases *templates* can be constructed as a once-off exercise and then subsequently decorated as needed.

3. As previously indicated, a motif ρ of arbitrary length (in principle) can be partitioned into contiguous substrings of length 4 followed by a suffix of length 0, 1, 2 or 3. As will be explained in greater detail below, FireSat$_1$ searches for TRs in an input string by matching suitable portions of the input against quad-pDFAs, decorated according to the substring of ρ under investigation at a given moment. If processing cannot proceed any further along a given decorated quad-pDFA, FireSat$_1$ may backup (four or less places, as appropriate) in the input string and select the next pDFA lower down in Table 6.1 for decorating and processing against the input.



(a) $e = 0$ or 1 insertion.



(b) $e = 0, 1$ or 2 insertions.

Figure 6.2: Quad-pDFA insertion acceptors, where $\upsilon_i =$ `acgt` and $e = 0, 1$ or 2.

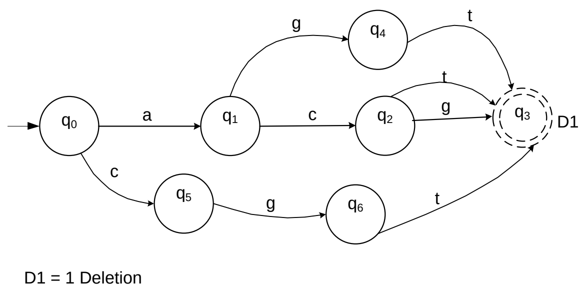


Figure 6.3: A pDFA that accepts substrings derived from v_i = `acgt` with 1 deletion error.

4. Note that in the worst case, backing up may result in the input being checked against $pDFA_M(v, 4)$ — the last pDFA in Table 6.1. Since any string of length 4 will be accepted by this pDFA, it will always be possible to move ahead in ρ to the next substring v_i, of length 4. Of course, if processing has reached the tail end of ρ it may be necessary to use a mono-, di- or tri-pDFA; otherwise the quad-pDFAs of Table 6.1 are used again, appropriately redecorated.

5. In the absence of additional filtering, once the tail end of ρ has been processed, the corresponding matched substring of input could theoretically be considered an ATRE. If it is indeed considered to be an ATRE, then the next stretch of input is again examined against ρ for the next ATRE, etc.

As suggested above, without any further filtering, the foregoing process could declare an arbitrary string of length $|\rho|$ in the input to be an ATRE, even if every nucleotide was considered to be a mismatch mutation. Clearly additional filters are needed in `FireSat`$_1$ to pose restrictions on deciding whether a potential TRE should be classified as such or not. These filters were explained in Chapter 5.

6.3 The `FireSat`$_1$ Algorithm

Algorithm 6.3.1 provides a high level specification of the `FireSat`$_1$ algorithm. The algorithm is invoked by calling:

$$FireSat(l_{min}, l_{max}, \varepsilon_{max\%}, \kappa_{max\%}, \sigma_{max\%}, \alpha_{max}, \beta_{min}, p_d, p_i, p_m, s)$$

where s represents the genetic sequence to be searched and where l_{min} and l_{max} specify the minimum and maximum lengths of motifs for which searches should be carried out. The remaining parameters are the filters and penalty weights already described in Chapter 5. It is assumed that all these parameters are globally available. The algorithm returns a set of tuples, each tuple providing information about a TR identified in s.

Algorithm 6.3.1.

```
func FireSat($l_{min}, l_{max}, \varepsilon_{max\%}, \kappa_{max\%}, \sigma_{max}, \alpha_{max}, \beta_{min}, p_m, p_d, p_i, s$) : $tuples$
    $tuples := \emptyset$
    ; for $l \in [l_{min}, l_{max}] \rightarrow$
        $pos := 0$
        ; $k, m := \lfloor \frac{l}{4} \rfloor$,  mod $(l, 4)$ { Number of quad-pDFAs & length of tail }
        ; do $(pos \leq |s| - l) \rightarrow$
            $\rho := s[pos, pos + l - 1]$
            ; $\langle \rho, pos, len, n_{tre}, n_m, n_d, n_i \rangle := computeTR(\rho, k, m)$
            ; if $(len > |\rho|) \rightarrow$
                $tuples := tuples \cup \{\langle \rho, pos, len, n_{ptre}, n_m, n_d, n_i \rangle\}$
                ; $pos := pos + len$
            [] $(len \leq |\rho|) \rightarrow pos := pos + 1$
            fi
        od
    rof
    ; return $tuples$
cnuf
```

□

In overview, FireSat$_1$ processing proceeds as follows.

An outer loop of the algorithm iterates over motif lengths in a given range, i.e. over $l \in [l_{min}, l_{max}]$. It determines m and k where m is the number of quad-pDFAs needed for a motif of this length and where k indicates whether a tri-, di- or mono-pDFA is needed to represent the motif's suffix.

An inner loop then scans through the input string, s, for TRs whose motif length is l. Suppose that, at a given point, processing is at a position pos of s. FireSat$_1$ takes the first l nucleotides starting at pos as a motif.

The algorithm then scans further into the input string from $pos + l$, seeking a TR. To do so, it uses the quad-pDFAs (and possibly a tri-, di- or mono-pDFA) decorated as prescribed by the current motif.

If a TR is found, the TR is reported and further TR searching commences from where the TR terminated. Should no TR be found at all (typically because one of the threshold values has been exceeded), then the search is repeated from index position $pos + 1$ and a new motif is established at that position.

The foregoing is repeated until the entire string s has been processed for motif length l. Thereafter, the entire search is repeated until each motif length in the range l_{min} to l_{max} has been examined.

computeTR is called by the algorithm. Its purpose will be explained below. The function returns a tuple containing the following information about a possible TR that might be located at *pos*:

1. ρ the PTRE of the detected TR;
2. *pos* the start position of the detected TR;
3. *len* the length of the detected TR;
4. n_{tre} the number of TREs occurring in the detected TR; and
5. n_m; n_d; n_i the number of mismatches, deletions and insertions that occurred throughout the detected TR.

If *len* is greater than the motif length, then a TR is deemed to have been found at position *pos* and the tuple of TR-related information is added to a tuple set. The algorithm returns this tuple set. If *pos* is deemed *not* to be the starting index of a TR, the next TR search starts from *pos* incremented by *len* (if a TR was found) or by 1 (if a TR was not found).

As indicated above, using $\rho = s[pos, pos + l - 1]$, `FireSat`$_1$ determines k and m such that

$$\rho = \upsilon_1\upsilon_2 \ldots \upsilon_k\upsilon_{k+1}$$

where $|\upsilon_i| = 4$ for $i = 1, \ldots k$, and $|\upsilon_{k+1}| = m \in [0, 3]$. Each υ_i is treated separately by a suitable pDFA. The cascading states in the pDFA referencing υ_{k+1} (or υ_k if $|\upsilon_{k+1}| = 0$) are regarded as *final* states.

For $i = 1, \ldots k + 1$, *computeTR* relies on subfunction *pDFASelector* to find a pDFA to represent υ_i and carry out the next matching stage. (For ease of reference, assume below that a quad-pDFA is used. The narrative is trivially similar when a tri-, di- or mono-pDFA needs to be used to represent υ_{k+1} .)

For each υ_i, *pDFASelector* selects a pDFA template from Table 6.1. For convenience, call it *pDFA*$_j$, where j indexes the row number in Table 6.1 to indicate the type of the pDFA. Initially $j = 1$, but it is subsequently incremented as necessary, under circumstances described below. *pDFASelector* then decorates *pDFA*$_j$ with the characters in υ_i.

Assuming *next* to be the index of the next element of s to be examined, a substring in s starting at *next* (call it u_i) is checked for membership of the language of *pDFA*$_j$. The following outcomes are possible:

1. A cascading state of *pDFA*$_j$ is reached. The index j is set to 1, the index *next* is appropriately updated, and υ_{i+1} is used to redecorate *pDFA*$_1$.
2. A cascading state of *pDFA*$_j$ is not reached. In this case, *pDFA*$_{j+1}$ is selected as the next pDFA template to be decorated with the elements of υ_i. Substring u_i is now tested for membership of the language of *pDFA*$_{j+1}$.
3. A final state of *pDFA*$_{k+1}$ (or *pDFA*$_k$) is reached. In this case, a possible TRE has been identified (subject, of course, to compliance of all the other filter values). One of two possibilities may occur: A search for the next possible TRE commences as described above. Alternatively there has not been yielded to all the other threshold values and the current TRE is discarded.

For reasons already noted, in the absence of additional filtering, once the tail end of ρ has been processed, the corresponding matched substring of input could always be considered an ATRE. However, apart from carrying out the actions just mentioned, *pDFASelector* also has to keep track continuously of the value of the various metrics against which filtering occurs. Substrings of s are filtered out as potential TRs when they exceed the filtering bounds. *computeTR* checks whether the restrictions on α_{max} and β_{max} are being met or not before a TR is being output as such.

The function *pDFASelector* thus keeps track of the number of mismatches, insertions and deletions states that are encountered. It relates these values to the input parameters, and reports back when a TRE has been found, or when one is no longer available.

6.4 `FireSat`$_1$: differing from previous work

It should be noted that the foregoing approach generalises the use of template ADFAs as implemented for `Fire`μ`Sat`$_2$ and described in De Ridder [2010] and De Ridder et al. [2013]. However, the template ADFAs of `Fire`μ`Sat`$_2$ differed significantly from those used here. For `Fire`μ`Sat`$_2$ different ADFAs are used to cater for $|\rho| = 2 \cdots 5$. For each of these motif lengths, ADFAs were constructed to cater for different motif error types (mismatch, insertion, deletion or a combination of the pre-mentioned) and number of motif errors. A cascade operation was therefore not needed for `Fire`μ`Sat`.

It should be noted that in theory, a simple cascading operation on two pDFAs results in a slightly larger ADFA. However, `FireSat`$_1$ does not simply cascade one pDFA onto another. Rather, as described above, it sometimes *backtracks* and replaces one attempted cascading operation with another using a different pDFA as the second operand of the cascading operation. In effect this means that `FireSat`$_1$ seeks a path through an acyclic NFA (ANFA) in trying to find TRs, rather than through an ADFA as was the case with `Fire`μ`Sat`. However, cascading only two pDFAs at a time means that `FireSat`$_1$ does not have to store the entire underlying ADFA in memory.

The function *computeTR* in `FireSat`$_1$ is broadly similar to its counterpart used in all the versions of `Fire`$_\mu$`Sat` described in De Ridder [2010]. However, keeping track of the number of mutations, and dealing with ambiguities in mutation types was considerably simpler in the `Fire`$_\mu$`Sat` algorithms.

Here, the resolution of ambiguities always takes place in the context of the v substrings of length 4 (or less). The way in which ambiguities are resolved is reflected in the ordering of pDFAs in Tables 6.1 to 6.4. To keep tabs of the number of errors encountered, global counters are used as part of the algorithm. These will be more explicitly discussed in subsequent versions of `FireSat`.

On the one hand, then, the cascading-based generalisation of `Fire`$_\mu$`Sat` that resulted in `FireSat`$_1$ means that it is now possible, at least in principle, to use FAs to detect TRs whose motif is of arbitrary length. The price to be paid for this generalisation is that the way in which mutation errors may be interpreted has to be constrained significantly.

6.5 Ambiguities: `FireSat`$_1$

Tables 6.1 to 6.4 indicate the order of preference in dealing with ambiguities. They prioritise mismatches over deletions. Deletions are in turn prioritised over insertions.

In this way, they avoid ambiguities with respect to mutations *within* $v_i, i = 1, \dots k + 1$ that might otherwise have arisen.

This order of prioritisation is in line with the general approach followed by other researchers, as can be inferred from Figure 3.4. The figure depicts a concept lattice whose objects are TR-detection algorithms and whose attributes are properties of these algorithms. The figure makes it clear that all TR-detection algorithms that have the property of detecting ATRs, also have the property of detecting mismatches. This is however not the case for insertions or deletions, i.e. some algorithms that detect ATRs do not allow for interpreting mutations either as insertions and/or as deletions.

FireSat$_1$ partially overcomes this limitation in that it allows for mutations to be interpreted *sometimes* as deletions or insertions. However, there may never be more than 1 insertion or deletion within an v_i substring of ρ.

Of course, these rules on the mutations allowed in v_i for $i = 1, \ldots n$, correspondingly constrain how mutations may be interpreted in ρ itself. The total number of mutations counted in an incoming string being adjudged as a possible TRE in relation to ρ cannot therefore be guaranteed to correspond with the LD between these two strings. Nevertheless, in FireSat$_1$ the mutation counts are tested against various user-specified filter values to decide whether or not the input string under consideration should be accepted as a valid TRE.

6.6 Conclusion

In principle, the pDFAs implemented for FireSat$_1$ can be augmented with additional pDFAs that allow for more deletions and/or insertions, and the ordering in which the various motif errors are prioritised can be modified.

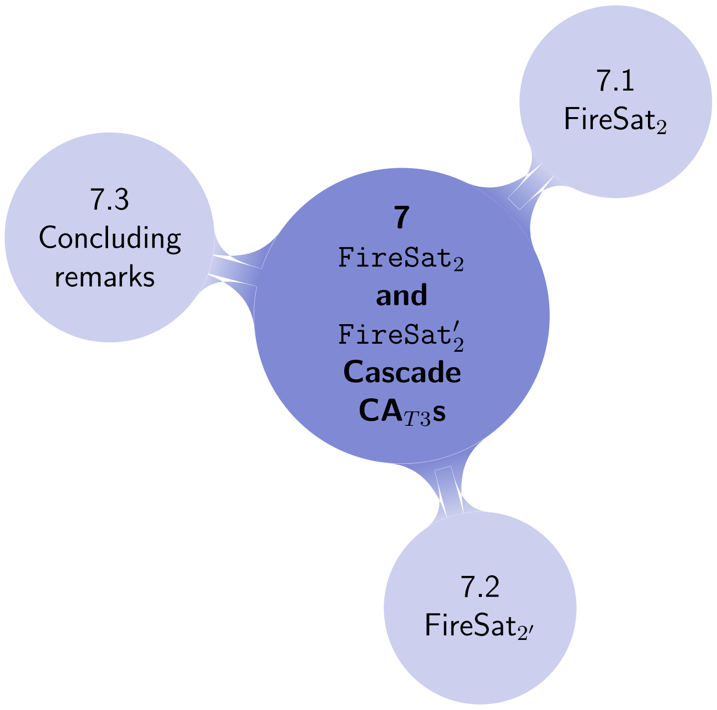
A prototype version of FireSat$_1$ as described above was implemented. Results obtained are reported in Chapter 9. One of the issues noted was that FireSat$_1$ loses accuracy because the number of insertions and deletions have been artificially limited to one per quad-, tri-, di- or mono-pDFA. This experience stimulated new ideas that led to the formulation of FireSat$_2$. It is described in the next chapter.

FireSat$_1$ was my first attempt at developing an FA-based algorithm for detecting TRs. It does not have the ability to detect TRs as accurately as FireSat$_2$, FireSat$_{2'}$ and FireSat$_3$. During the theoretical development of FireSat$_1$ it was realised that there could be a different and more effective way of using cascaded pCA$_{T3}$s instead of pDFAs during TR-detection. Nevertheless, FireSat$_1$ is included in this thesis because it provided the theoretical background that instigated the development of FireSat$_2$ and subsequently of FireSat$_{2'}$. FireSat$_1$ has been implemented — Chapter 9 reports on its accuracy.

Chapter References

C. De Ridder. Flexible Finite automata-based algorithms for detecting microsatellites in DNA. Master's thesis, Department of Computer Science, University of Pretoria, July 2010.

C. De Ridder, D.G. Kourie, B.W. Watson, T.R. Fourie, and P.V. Reyneke. Fine-tuning the search for microsatellites. *Journal of Discrete algorithms*, 20:21–37, 2013.



7

`FireSat`$_2$ **and** `FireSat`$_{2'}$ **Cascade CA$_{T3}$s**

"The intuitive mind is a sacred gift and the rational mind is a faithful servant. We have created a society that honours the servant and has forgotten the gift." ... Albert Einstein

---

This chapter introduces both `FireSat`$_2$ and `FireSat`$_{2'}$. `FireSat`$_2$ relies on pCA$_{T3}$s that are concatenated and together are used to calculate the number of mutations in an input string, allocated according to a certain priority scheme. To address the shortcomings of `FireSat`$_2$, `FireSat`$_{2'}$ is presented in Section 7.2. `FireSat`$_{2'}$ cascades pNCA$_{T3}$s to detect TRs.

7.1 `FireSat`$_2$

The overall flow of logic of `FireSat`$_2$ is almost the same as that of `FireSat`$_1$ as shown in Algorithm 6.3.1. (The number of quad-pDFAs, k, and length of the tail, m is not computed.) In both cases, NCA$_{T3}$s that recognise ρ and a subset of its mutational variants underlie the respective algorithms.

However, details differ considerably in regard to how these respective NCA$_{T3}$s are built and stored. Notionally, this may be regarded as taking place within the call to *computeTR*.

As was seen in the previous chapter, `FireSat`$_1$ does not pre-assemble and store underlying pDFAs, but rather, in a just-in-time fashion, it assembles and stores the immediately needed parts as the string to be recognised is being scanned. In contrast, `FireSat`$_2$ builds and stores the entire NCA$_{T3}$ before scanning for TREs. The FA that is evolved in `FireSat`$_2$, call it FA$_\rho$, is different from the underlying one in `FireSat`$_1$. Its language is a larger subset of mutational variants of ρ.

In this case, FA_ρ is composed of various pCA$_{T3}$s. It also has so-called pseudo-states that are used to keep global counts of the various mutation types. It will be seen that this FA$_\rho$ together with the global counts provide information from which a *distance* can be computed between ρ and the subsequent substring of s that is to be examined as a possible TRE. Since this distance resembles the LD, but is not identical to the LD, it is referred to as a Levenshtein Based Distance (LBD). This LBD is used as the basis for identifying the next TRE in s.

In contrast to `FireSat`$_1$, `FireSat`$_2$ does not constrain the TR search to the artificial limit of one insertion and one deletion for every quad-, tri- or di-pCA$_{T3}$. Instead, an arbitrary number of mismatches, deletions and insertions are allowed, provided that their sum remains within the specified thresholds.

In `FireSat`$_1$, quad-pDFA templates were cascaded together with a single quad-, tri- or di-pDFA template tagged on at the end of the resulting structure as needed. This was done solely on the basis of the length of ρ, without reference to ρ's actual characters. Only later was the resulting template structure decorated to reflect the characters of the motif.

In `FireSat`$_2$ the pCA$_{T3}$s to be cascaded are determined according to an entirely different heuristic. Now, the actual character content of the motif, ρ, is considered — not merely its length. A heuristic is followed whereby ρ is decomposed into substrings, each of which is *the longest possible substring of unique characters.* By *unique characters* is meant that no character appears more than once in the substring. Because genetic strings only have four characters, this means that substrings, resulting from this decomposition, will have a maximum length of 4.

Example 7.1.1.
Suppose $\rho =$ `acacggggacgt`. Then its decomposition into the longest possible substrings of unique characters results in $\rho =$ `ac.ac.g.g.g.g.acgt`. □

For each substring, `FireSat`$_2$ constructs a *decorated* pCA$_{T3}$ that recognises the substring and all its mutational variants, together with counters that track the number of mutational variants. These pCA$_{T3}$s are then cascaded together as FA_ρ is evolved.

Example 7.1.2.
The pCA_{T3}s for the substrings in Example 7.1.1 will be:

- *two di-pCA_{T3}s, each representing* `ac` *that will be cascaded to cater for* `acac`*;*
- *four mono-pCA_{T3}s, each representing* `g` *to cater for* `gggg` *that will be cascaded onto the pCA_{T3}s for* `acac`*; and*
- *one quad-pCA_{T3}, representing* `acgt`*, is cascaded onto the result.*

□

The material in the subsections that follow illustrates how the required pCA_{T3}s are constructed.

- Subsection 7.1.1 considers quad-pCA_{T3}s, illustrating how one can be constructed to deal with mismatches, a separate one to deal with insertions and yet another separate one to deal with deletions.
- Subsection 7.1.2 does the same with respect to mono-, di- and tri-pCA_{T3}s.
- Subsection 7.1.3 shows how to construct a single di-pCA_{T3} that caters for *all* mutation types.
- Subsection 7.1.4 highlights the three kinds of counters needed when using a NCA_{T3}.
- Next, Subsection 7.1.5 gives an analysis of the summed di-pCA_{T3}.
- Thereafter, Subsection 7.1.6 illustrates *how* a $\texttt{FireSat}_2$ NCA_{T3} is built and traversed.
- The next section, Section 7.1.7, suggests an alternative way of doing motif error management.
- Section 7.1.8 discusses how ambiguities are handled.
- Section 7.1.9 gives some concluding remarks pertaining to $\texttt{FireSat}_2$.

Thereafter, Section 7.2 gives details of $\texttt{FireSat}_{2'}$.

7.1.1 Quad-pCA$_{T3}$s

In this subsection, mismatch quad-pCA$_{T3}$s (7.1.1.1), deletion quad-pCA$_{T3}$s (7.1.1.2) and insertion quad-pCA$_{T3}$s (7.1.1.3) are presented.

In these and all similar pCA$_{T3}$s, the cascading states will be called a mismatch (deletion, insertion or perfect, respectively) cascading state, if it occurs in a mismatch (deletion, insertion or perfect, respectively) pCA$_{T3}$.

It will be seen that these pCA$_{T3}$s are given additional *pseudo-states*. These are linked to conventional states by dashed lines. Each pseudo-state is labelled by a character P, M or I, indicating that the state is associated with a counter for perfect matches, mismatches or insertions, respectively.

Pseudo-states are assumed to have the following functionality:

> Suppose conventional state C is linked to pseudo-state S and suppose that S is labelled X, where X is one of the counter labels P, M or I. Then whenever state C is entered (via any of its conventional inbound transition arcs), then control is passed to pseudo-state S, its counter for X is appropriately updated as described below. Control is then returned to state C.

Note that there is no pseudo-state counter associated with deletions. Section 7.1.1.2 explains how *deletions* are calculated by using the perfect match counter value.

7.1.1.1 Quad-pCA$_{T3}$s: mismatches

Consider again the four quad-pDFAs in Figure 6.1 accepting various numbers of mismatches on the string `acgt`. Note that the pDFA in Figure 6.1(d) (providing for 4 mismatch errors) has 9 states. These are fewer states than the pDFAs in Figures 6.1(a) (providing for 1 mismatch error and having 12 states); 6.1(b) (providing for 2 mismatch errors and having 16 states) and 6.1(c) (providing for 3 mismatch errors and having 14 states). This observation was the starting point for the pCA$_{T3}$s proposed here.

Instead of constructing 4 different pCA$_{T3}$s accepting v of length 4, one for each $e_{max} = 1, 2, 3$ or 4, the idea emerged of constructing a single machine that is used to increment a counter for each mismatch state reached. Thus the pDFA presented in Figure 6.1(d) has been adapted to the pCA$_{T3}$ in Figure 7.1.

Note that the pCA$_{T3}$ in Figure 7.1 is indeed a counting automaton of type 3. Whenever an arbitrary state, q_i, is entered, the corresponding counter d_i is incremented. The dashed lines, connecting states to the so called pseudo-state M indicate that the total number of mismatches read by the pCA$_{T3}$ presented in Figure 7.1 can be obtained by adding the counter values of states q_5, q_6, q_7 and q_8. The count can be compared against an initialised threshold value. Similarly, the sum of the counter values of states q_1, q_2, q_3 and q_4, all connected to pseudo-state P, indicate the number of perfect matches read for an input string.

This contrasts with the machines presented in Figure 6.1. In those cases, the different final states indicate the number of mismatches. For example, if state q_{11} in Figure 6.1(c) is reached, then the number of mismatches that occurred is 3. Note that if there is a perfect match, then states q_1, q_2, q_3 and q_4 in Figure 7.1 will be traversed and the mismatch counter of the pseudo-state, M, will be at its initialised value, namely 0.

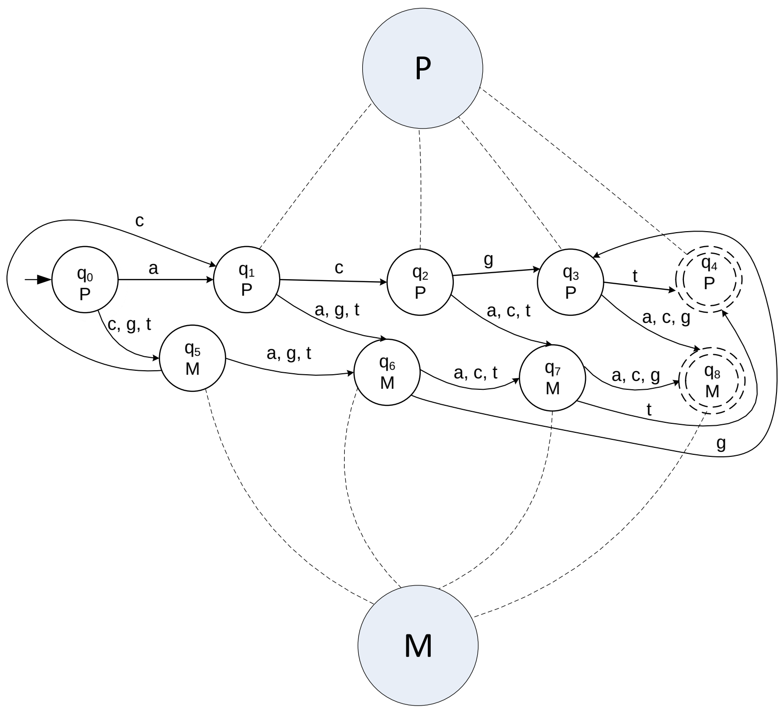


Figure 7.1: A quad-pCA$_{T3}$ accepting $\upsilon =$ `acgt` or any substring generated from υ with 4 or less mismatch errors ($e \leq 4$).

7.1.1.2 Quad-pCA$_{T3}$s: deletions

The deletion pCA$_{T3}$ in Figure 7.2 counts the number of *perfect* matches in a string that has deletions. Each state of the deletion machine is a cascading state, including the start state q_0.

As illustrated in Figure 7.2, dashed lines connect all the deletion pCA$_{T3}$ states to a large pseudo-state labelled P. P represents a *perfect match pCA$_{T3}$ counter.* Each time a deletion state is entered a perfect match has occurred and P is updated accordingly.

The value of P at any stage stands in a dual relationship to the number of deletions, D, encountered at that stage, i.e. in general $D = |v| - P$ and in the present case, $D = 4 - P$.

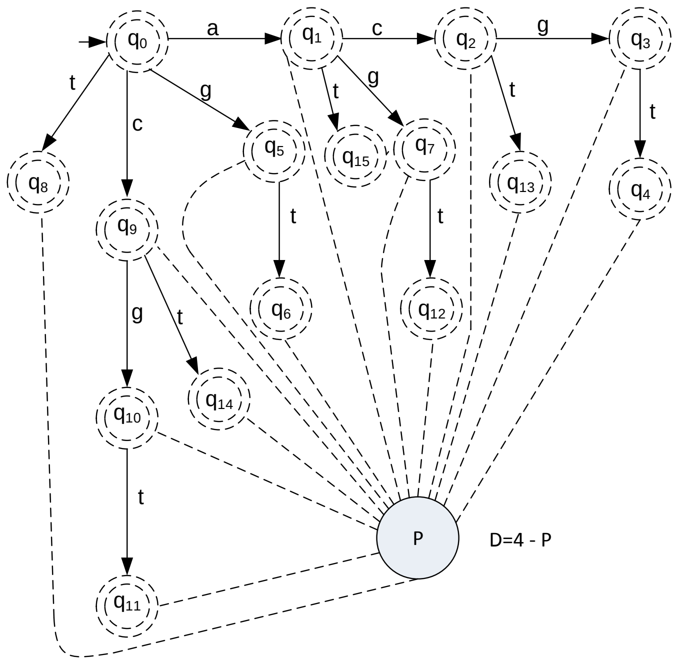


Figure 7.2: A pCA$_{T3}$ accepting $v =$ `acgt` and substrings derived from v with up to 4 deletion errors.

7.1.1.3 Quad-pCA$_{T3}$s: insertions

There is an increase in the number of states of quad-pDFAs that cater for insertions as the number of insertions rises. Thus, for example, the quad-pDFA providing for 1 insertion (Figure 6.2(a)) has 15 states, whereas the quad-pDFA catering for 2 insertions (Figure 6.2(b)) has 29 states.

If quad-pCA$_{T3}$s are used instead of quad-pDFAs, the number of states also increases as the number of insertions to be catered for rises. However, the insertion pCA$_{T3}$s have significantly fewer states than their pDFA counterparts. It can be shown that a quad-pCA$_{T3}$ that caters for maximally 2 insertions would need 13 states — less than half those required by a quad-pDFA.

Figure 7.3 shows a pCA$_{T3}$ that can cater for up to $4 \times 4 = 16$ insertions if the predicate $I < 4$ is removed from the pseudo state. It has a mere 21 states including its one cascading state, q_4. Note that a perfect match has to follow after insertions have occurred. As before, the convention followed corresponds to that of Holub [2010] and Melichar [1996] in the sense that insertions after the fourth (i.e. last) perfect match are not allowed. The pseudo-state, I, represents the total *insertion* counter.

Figure 7.4 shows a pCA$_{T3}$ that is language-equivalent to the pCA$_{T3}$ in Figure 7.3 but has fewer states. Note that although this pCA$_{T3}$ contains cycles it can still be regarded an ADFA, since the cycles are limited by finite threshold values. Clearly, one could cater for any number of insertions occurring consecutively by adjusting the conditions on respective counters of the pCA$_{T3}$ in Figure 7.4.

Insertion pCA$_{T3}$s: limitations
These differences mean that pCA$_{T3}$s, similar to the one in Figure 7.3, cannot be cascaded to recognise strings that have more than 4 insertions before the next perfect match.[1] Neither can it be cascaded in such a manner that a mismatch can occur directly after insertions. This is in contrast to the mismatch-pCA$_{T3}$ and deletion-pCA$_{T3}$. When two mismatch pCA$_{T3}$s are cascaded together, the resulting pCA$_{T3}$ can recognise strings with more than 4 consecutive mismatch errors. Furthermore perfect matches, deletions and/or insertions may follow these mismatches. The same applies in the case of deletion pCA$_{T3}$s.

Sub-sections 7.1.1.1, 7.1.1.2 and 7.1.1.3 illustrated quad-pCA$_{T3}$s that recognise mutational variants of $\upsilon =$ `acgt`. In each case, the relevant quad-pCA$_{T3}$ can recognise υ, as well as strings with $e \leq |\upsilon|$. In all cases, the template quad-pCA$_{T3}$s associated with an arbitrary string of length 4 can easily be inferred.

For completeness, the next subsection briefly illustrates similar scenarios for $|\upsilon| = 3, 2$ and 1 respectively.

7.1.2 Tri-, di- and mono-pCA$_{T3}$s for mutational errors

This subsection provides tri-, di-, and mono-pCA$_{T3}$s for various mutation scenarios. Again, each of the pCA$_{T3}$s will be able to recognise υ itself as well as strings with motif errors $e \leq |\upsilon|$. And, again, template tri-, di- and mono-pCA$_{T3}$s can easily be inferred from these examples.

For a given υ, three separate machines are constructed. They recognise υ with mismatches, υ with deletions and υ with insertions, respectively. These three machines also recognise υ itself. A *perfect* machine is included for completeness — i.e. one that recognises $|\upsilon|$ itself and nothing else.

[1] Note that the assigned threshold values, against which the respective counts of states q_5, q_6, q_7 and q_8 are compared, can be adjusted as required.

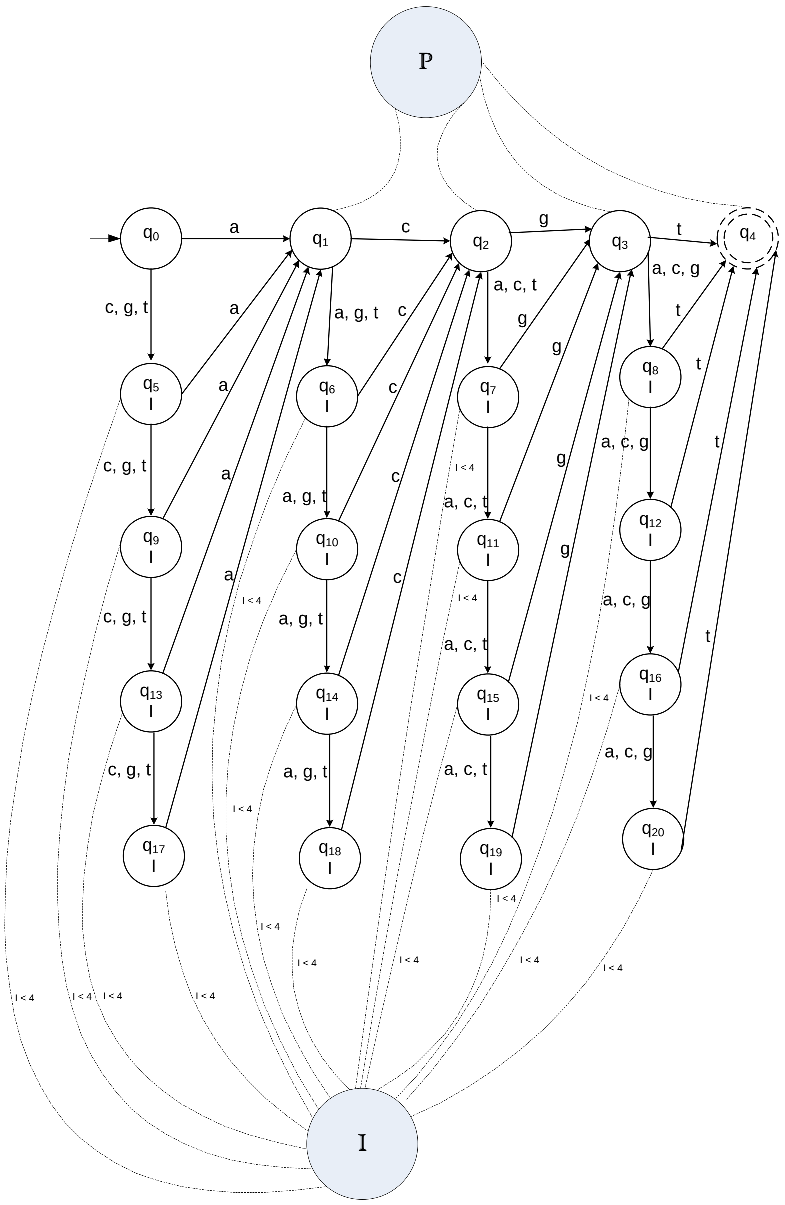


Figure 7.3: A pCA_{T3} accepting $\upsilon =$ `acgt` and any substring derived from υ that has at most 4 insertion errors.

7.1.2.1 Tri-pCA_{T3}s

The pCA_{T3}s in Figure 7.5 assume that $\upsilon =$ `acg`. The pCA_{T3}s in Figures 7.5(a), 7.5(b), 7.5(c) and 7.5(d) recognise respectively υ only; υ and up to 3 mismatches; υ and up to 3 deletions; and υ and up to 3 insertions. Note that a perfect match has to occur after at most 3 consecutive insertions.

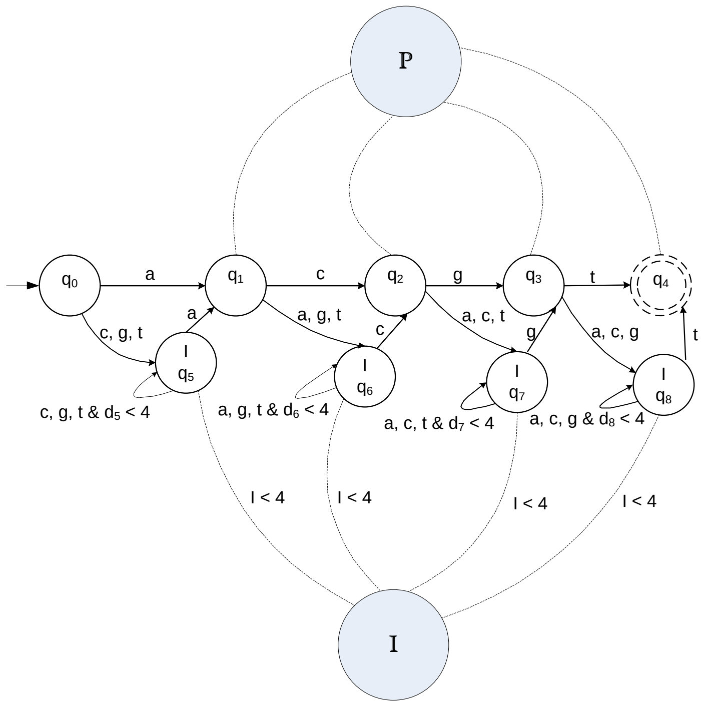
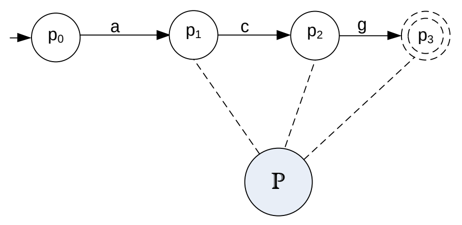


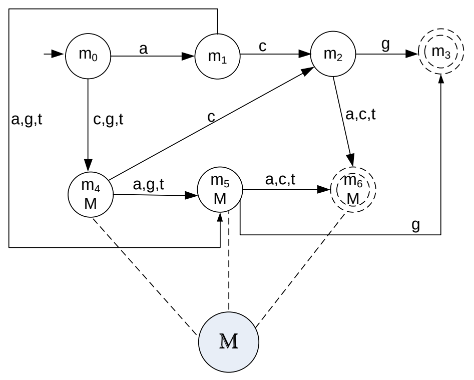
Figure 7.4: A pCA$_{T3}$ accepting υ = `acgt` and any substring derived from υ, containing insertions, where $0 \leq e \leq 4$.

7.1.2.2 Di-pCA$_{T3}$s

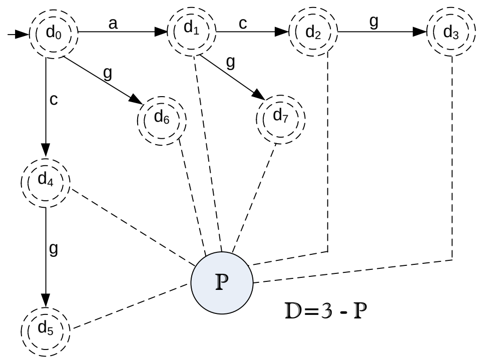
The di-pCA$_{T3}$s in Figure 7.6 cater for υ = `ac`. The di-pCA$_{T3}$ in Figure 7.6(a) is for a perfect match only. The di-pCA$_{T3}$ in Figure 7.6(b) allows for two mismatches to occur. The di-pCA$_{T3}$ in Figure 7.6(c) allows for a maximum of 2 deletions. The counter associated with pseudo-state I in Figure 7.6(d) is used to build a recogniser that recognises υ and up to 2 insertions.



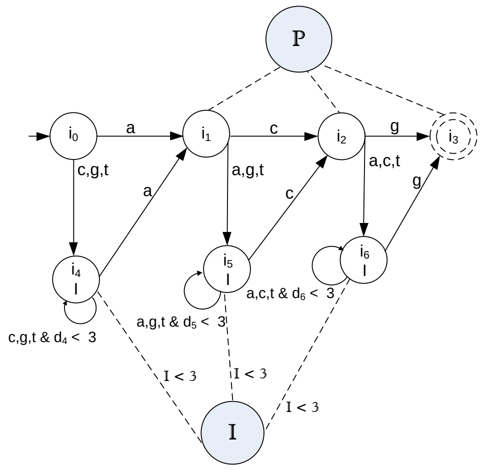
(a) Perfect tri-pCA_{T3}.



(b) Mismatch tri-pCA_{T3}.

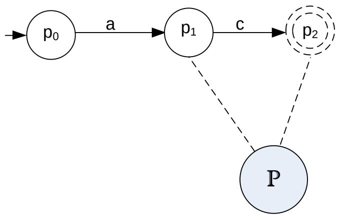


(c) Deletion tri-pCA_{T3}.

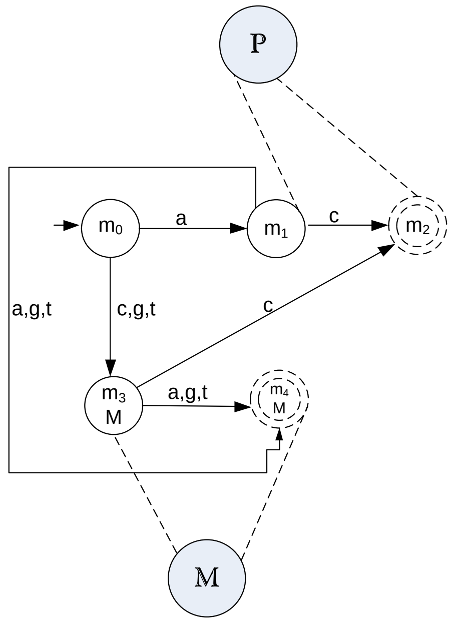


(d) Insertion tri-pCA_{T3}.

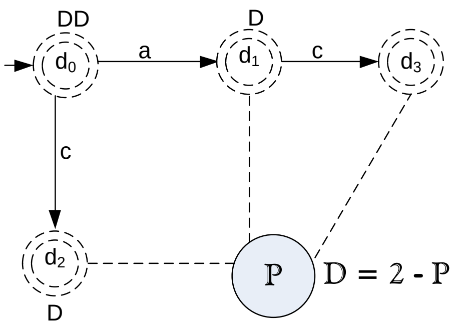
Figure 7.5: Tri-pCA_{T3}s acceptors for $v = \texttt{acg}$ and $0 \leq e \leq 3$.



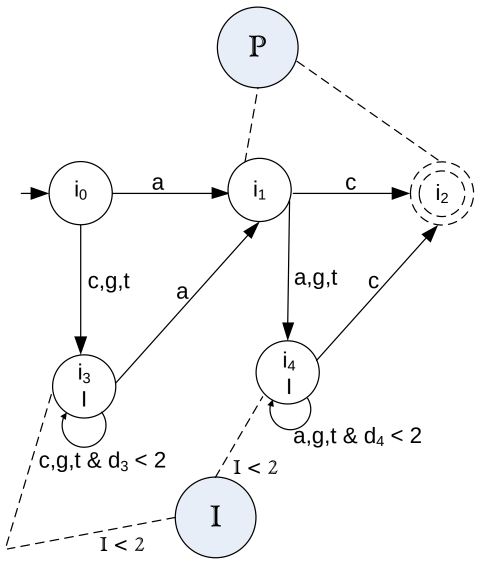
(a) Perfect di-pCA$_{T3}$.



(b) Mismatch di-pCA$_{T3}$.



(c) Deletion di-pCA$_{T3}$.

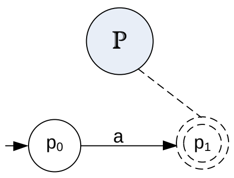


(d) Insertion di-pCA$_{T3}$.

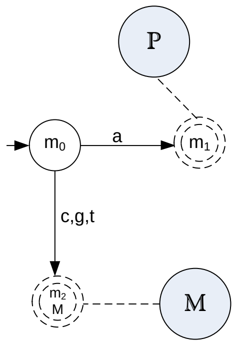
Figure 7.6: Di-pCA$_{T3}$s acceptors for $v = \texttt{ac}$ and $0 \leq e \leq 2$.

7.1.2.3 Mono-pCA$_{T3}$s

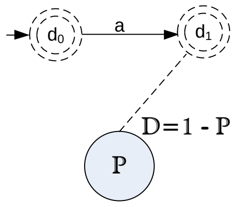
Figure 7.7 contains the various types of mono-pCA$_{T3}$s for $v = \texttt{a}$. These are the perfect, mismatch, deletion and insertion mono-pCA$_{T3}$s. These are shown in



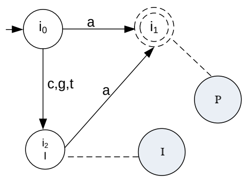
(a) Perfect mono-pCA$_{T3}$.



(b) Mismatch mono-pCA$_{T3}$.



(c) Deletion mono-pCA$_{T3}$.



(d) Insertion mono-pCA$_{T3}$.

Figure 7.7: Mono-pCA$_{T3}$s for $v = \mathtt{a}$ and $0 \leq e \leq 1$.

Subfigures 7.7(a), 7.7(b), 7.7(c) and 7.7(d) respectively.

7.1.3 Summing pCA$_{T3}$s over all mutation types

Note that the component pCA$_{T3}$s dealing with mismatches, deletions and insertions shown in the respective figures of Subsections 7.1.1 and 7.1.2 can be summed together to produce corresponding quad-, tri-, di- or mono-pCA$_{T3}$s.

In each case, the P, M, D and I pseudo-state counters are carried over from the component pCA$_{T3}$s to the resulting pCA$_{T3}$ as part of this summing operation. It will be seen that in addition to the pseudo-state counters, *global* counters are also required for computing the LBD. They are updated by referring to these pseudo-state counters. Global counters will be discussed in more detail in Section 7.1.4.

This section uses the di-pCA$_{T3}$s in Figure 7.6 to illustrate how this summation over all mutation types is carried out. The quad-, tri- and mono-pCA$_{T3}$s summations take place analogously to the illustrated di-pCA$_{T3}$ case.

Recall that the summation of two FAs, F_1 and F_2 with languages $L(F_1)$ and $L(F_2)$ yields an FA recognising language $L(F_1) \cup L(F_2)$. Because set union is both commutative and associative, the order in which the summation of more than one FA takes place does not matter. The same applies to pCA$_{T3}$s. In the present example, the mismatch di-pCA$_{T3}$ is first added to the deletion di-pCA$_{T3}$. The resulting machine is then added to the insertion di-pCA$_{T3}$. Since the mismatch pCA$_{T3}$ (as well as the insertion and deletion pCA$_{T3}$s) recognise a perfect match, the resulting di-CA$_{T3}$ automatically caters for perfect matches.

Note that cascading states from two component pCA$_{T3}$ machines that are summed together will never be combined together in the summed machine. To verify this assertion, consider a pCA$_{T3}$ that is the sum of a mismatch, deletion and insertion pCA$_{T3}$ respectively. Suppose the string v traces a path through the pCA$_{T3}$ to a cascading state, q, making t_{no} transitions. Clearly, if $t_{no} < |v|$ then q has to be a deletion cascading state. If $t_{no} > |v|$ then q has to be an insertion cascading state. If $t_{no} = |v|$ then q has to be either a mismatch cascading state or a perfect cascading state.

In the latter case, the cascading state cannot be both a mismatch and perfect cascading state. This can be seen by inspecting the mismatch / perfect pCA$_{T3}$s given above for quad-, tri-, di- and mono-pCA$_{T3}$s (See Figures 7.1, 7.5(b), 7.6(b) and 7.7(b)). It will be seen that in the pCA$_{T3}$, a perfect match of v is dealt with at a state being treated as a *deletion cascading state*, whereas mismatch cascading states deal with any variant of v containing mismatches. Details will follow.

A transition table is used to specify the transition function of the pCA$_{T3}$ that results from a summation step. Each row of the table represents a state and each column represents an alphabet character. The cell defined by the row for state z and the column for character α contains the destination state specified by $\delta(z, \alpha)$.

The following conventions are used in transition tables and diagrams representing the resulting pCA_{T3}s:

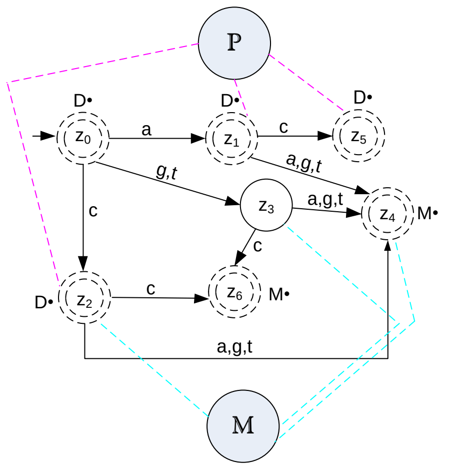
- Each state of a summed machine combines two states of the component machines. Each table row therefore not only indicates a newly assigned state name of the summed machine, but also indicates the names of the states used in the component machines to compose that new state. The diagrams show only the new state names.
- If $\delta(q, \alpha)$ is an undefined destination state, then $\perp$ is entered into the corresponding table cell. In the diagram, state q will not have an outbound edge, labelled by α.
- In the diagrams of summed pCA_{T3}s, the following rule is applied to determine to which states the M, P and I pseudo-states should be connected:
  If a component state, say p, is connected to an M (and/or P and/or I, respectively) pseudo-state, and if p is combined with some other state to make up a summed state, say q, then q should also be connected to the M (and/or P and/or I, respectively) pseudo-state.
- A state of the summed machine is regarded as a cascading state if and only if it is made up of a cascading state from at least one of its component machines. In the tables, the $\bullet$ character is used as a superscript to the new state name to indicate that it is a cascading state. In the diagrams, cascading states are drawn, as before, as two dashed-lined concentric circles.
- In the diagrams, mismatch, deletion and insertion cascading states are labelled by $M\bullet$, $D\bullet$ and $I\bullet$ respectively. The labels are inherited from the component machines. $P\bullet$ is omitted from diagrams. The reason for the omission of $P\bullet$ is that the number of perfect matches differs for each type ($M\bullet$, $D\bullet$, and $I\bullet$) of cascading state. The text below distinguishes between the different calculations of the number of perfect matches.

Subsections 7.1.4 and 7.1.5 will indicate the circumstances under which global counters are to be updated when one of these labels is encountered during processing.

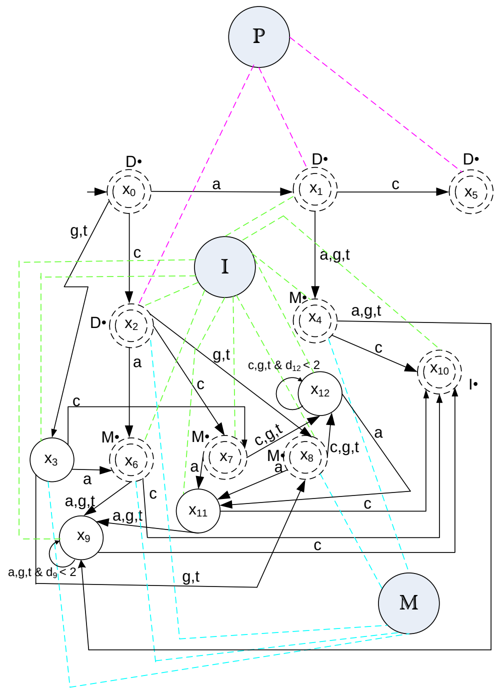
Table 7.1 is the transition table resulting from summing the mismatch and deletion di-pCA_{T3}s of Figures 7.6(b) and 7.6(c) respectively. Notice that all states except z_3 are cascading, because it is the only state that does not involve a cascading state of at least one component pCA_{T3}. The graphical depiction is given in Figure 7.8(a).[2]

States \ Alpha	a	c	g	t
$z_0^\bullet = m_0$ or d_0	m_1 or $d_1 = z_1$	m_3 or $d_2 = z_2$	m_3 or $\perp = z_3$	m_3 or $\perp = z_3$
$z_1^\bullet = m_1$ or d_1	m_4 or $\perp = z_4$	m_2 or $d_3 = z_5$	m_4 or $\perp = z_4$	m_4 or $\perp = z_4$
$z_2^\bullet = m_3$ or d_2	m_4 or $\perp = z_4$	m_2 or $\perp = z_6$	m_4 or $\perp = z_4$	m_4 or $\perp = z_4$
$z_3 = m_3$ or $\perp$	m_4 or $\perp = z_4$	m_2 or $\perp = z_4$	m_4 or $\perp = z_4$	m_4 or $\perp = z_4$
$z_4^\bullet = m_4$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$z_5^\bullet = m_2$ or d_3	$\perp$	$\perp$	$\perp$	$\perp$
$z_6^\bullet = m_2$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table 7.1: Transition table resulting from adding the mismatch and deletion di-pCA$_{T3}$s.



(a) Mismatches & deletions di-pCA$_{T3}$.



(b) Complete di-pCA$_{T3}$.

Figure 7.8: di-pCA$_{T3}$s for $\upsilon =$ ac.

Table 7.2 is the transition table of the di-pCA$_{T3}$ that is the sum of the deletion-mismatch di-pCA$_{T3}$ in Figure 7.8(a) and the insertion di-pCA$_{T3}$ in Figure 7.6(d). The graphical depiction of this final pCA$_{T3}$ is given in Figure 7.8(b).

[2]Note that tables illustrating how to add mono-pCA$_{T3}$s, tri-pCA$_{T3}$s and quad-pCA$_{T3}$s are provided in Appendix B.

States \ Alpha	a	c	g	t
$x_0^\bullet = z_0$ or i_0	z_1 or $i_1 = x_1$	z_2 or $i_3 = x_2$	z_3 or $i_3 = x_3$	z_3 or $i_3 = x_3$
$x_1^\bullet = z_1$ or i_1	z_4 or $i_4 = x_4$	z_5 or $i_2 = x_5$	z_4 or $i_4 = x_4$	z_4 or $i_4 = x_4$
$x_2^\bullet = z_2$ or i_3	z_4 or $i_1 = x_6$	z_6 or $i_3 = x_7$	z_4 or $i_3 = x_8$	z_4 or $i_3 = x_8$
$x_3 = z_3$ or i_3	z_4 or $i_1 = x_6$	z_6 or $i_3 = x_7$	z_4 or $i_3 = x_8$	z_4 or $i_3 = x_8$
$x_4^\bullet = z_4$ or i_4	i_4 or $\perp = x_9$	i_2 or $\perp = x_{10}$	i_4 or $\perp = x_9$	i_4 or $\perp = x_9$
$x_5^\bullet = z_5$ or i_2	$\perp$	$\perp$	$\perp$	$\perp$
$x_6^\bullet = z_4$ or i_1	i_4 or $\perp = x_9$	i_2 or $\perp = x_{10}$	i_4 or $\perp = x_9$	i_4 or $\perp = x_9$
$x_7^\bullet = z_6$ or i_3	i_1 or $\perp = x_{11}$	i_3 or $\perp = x_{12}$	i_3 or $\perp = x_{12}$	i_3 or $\perp = x_{12}$
$x_8^\bullet = z_4$ or i_3	i_1 or $\perp = x_{11}$	i_3 or $\perp = x_{12}$	i_3 or $\perp = x_{12}$	i_3 or $\perp = x_{12}$
$x_9 = \perp$ or i_4	i_4 or $\perp = x_9$	i_2 or $\perp = x_{10}$	i_4 or $\perp = x_9$	i_4 or $\perp = x_9$
$x_{10}^\bullet = \perp$ or i_2	$\perp$	$\perp$	$\perp$	$\perp$
$x_{11} = \perp$ or i_1	i_4 or $\perp = x_9$	i_2 or $\perp = x_{10}$	i_4 or $\perp = x_9$	i_4 or $\perp = x_9$
$x_{12} = \perp$ or i_3	i_1 or $\perp = x_{11}$	i_3 or $\perp = x_{12}$	i_3 or $\perp = x_{12}$	i_3 or $\perp = x_{12}$

Table 7.2: Transition table resulting from the addition of the deletion, mismatch and insertion di-pCA$_{T3}$s.

States	Pseudo Counter Update	Cascading Type
$x_0 = m_0$ or d_0 or i_0	-	D•
$x_1 = m_1$ or d_1 or i_1	inc(P)	D•
$x_2 = m_3$ or d_2 or i_3	inc(M); inc(P) ; inc(I)	D•
$x_3 = m_3$ or $\perp$ or i_3	inc(M); inc(I)	Not cascading
$x_4 = m_4$ or $\perp$ or i_4	inc(M); inc(I)	M•
$x_5 = m_2$ or d_3 or i_4	inc(P)	D•
$x_6 = m_4$ or $\perp$ or i_4	inc(M); inc(I)	M•
$x_7 = m_2$ or $\perp$ or i_3	inc(I)	M•
$x_8 = m_4$ or $\perp$ or i_3	inc(M); inc(I)	M•
$x_9 = i_4$ or $\perp$	inc(I)	Not cascading
$x_{10} = i_2$ or $\perp$	-	I•
$x_{11} = i_1$ or $\perp$	-	Not cascading
$x_{12} = i_3$ or $\perp$	inc(I)	Not cascading

Table 7.3: Attributes of the x-states.

Table 7.3 summarises information about the states of the final di-pCA$_{T3}$. The first column of the table indicates the states from which the respective x-states were originally derived.

Column 2 shows the *pseudo-counters* that are updated whenever an x-state is entered that is connected to an associated pseudo-state. The updating corresponds exactly to the updating done by the original machines. In some instances, more than one pseudo-counter has to be updated.

The final column indicates the *cascading type* of cascading states — i.e. whether it is a M•, D• or I• cascading state, or not a cascading state at all. The type can be inferred directly from the component di-pCA$_{T3}$s given in Figure 7.6. This information is available since it is always apparent which component states are used to derive a summed state. In principle it is therefore possible to carry over all the state information of the component pCA$_{T3}$s to the summed pCA$_{T3}$. This is exactly the information that is displayed in Table 7.3.

7.1.4 Three levels of counters

To determine whether or not a substring of s is to be considered a TRE, `FireSat`$_2$ systematically traverses various complete pCA$_{T3}$s (such as the one shown in Figure 7.8(b)) that are used to construct FA$_\rho$ – the underlying NCA$_{T3}$ whose language includes TREs of $\rho = \upsilon_1\upsilon_2 \ldots \upsilon_n$ where $|\upsilon| \leq 4$. Recall that the language of each complete pCA$_{T3}$ corresponds to some υ_i, as well as all permitted mutational variants of υ_i. Three type counters are updated during such traversals. The value of the fourth counter is calculated.

1. The counting automata state counters are relevant at some insertion states of the pCA$_{T3}$s. They determine the number of insertions that may occur consecutively. Figure 7.8(b) shows such counters in use at states x_9 and x_{12}.

2. Take note of the three pseudo-counters denoted by P, M and I that are also local to each specific pCA$_{T3}$. They keep count of the number of perfect matches, mismatches and insertions, respectively that are encountered during the traversal of the specific pCA$_{T3}$.

3. Finally, global counters denoted by GP, GM, GI and GD are used to count the overall number of perfect matches and of errors by mutational type that have occurred to date with reference to υ, in the substring of s, that is being evaluated as a possible TRE. The updating of these global counters occurs whenever a transition is made from a cascading state of a pCA$_{T3}$ that represents υ_i to a state of another pCA$_{T3}$ that represents υ_{i+1}. Updating also takes place when a final state is reached of the pCA$_{T3}$ that represents υ_n. Table 7.4 indicates how global counters are updated, depending on the cascading type.

7.1.5 An analysis of the summed di-pCA$_{T3}$

This subsection elaborates further on the di-pCA$_{T3}$ in Figure 7.8(b). The di-pCA$_{T3}$ recognises $\upsilon =$ `ac` (a perfect match) and all strings that carry mutations

Cascading Type	Global Counter Updating

Mismatch	GM = GM + M;
	GP = GP + ($\|v\| - M$);
	GI and GD unchanged.

Insertion	GI = GI + I;
	GP = GP + $\|v\|$;
	GM and GD unchanged.

Deletion	GD = GD + ($\|v\| - P$);
	GP = GP + P;
	GM and GI unchanged.

Table 7.4: The updating of global counters.

of this substring, whether mismatches, insertions or deletions, complying with the constraint discussed earlier.

Various states of the di-pCA$_{T3}$ are linked to the pseudo-counter states with pseudo-counters P, M and I. Note that there is no pseudo-state for a deletion pseudo-counter, D — i.e. the number of deletions encountered on a path from the start state to the current state. Recall that its value can be inferred by referencing pseudo-counter P.

The pseudo-counters are used to update the global counters GP, GM, GI and GD as already indicated in Table 7.4. Recall that these global state updates are only made if a transition is to be made from the cascading state of the di-pCA$_{T3}$ to a state in a pCA$_{T3}$ outside of the di-pCA$_{T3}$.

Deletion cascading states and counter updating
The deletion cascading states x_0, x_1, x_2 and x_5 are labelled by $D\bullet$. Table 7.4 requires that GD be increased at these states by an amount $|v| - P$. For these states one of the following holds:

- *No match* has occurred and there were two deletions (x_0). P will have its initialised value of 0 and therefore $|v| - P = 2 - P = 2$. Thus in this case GD is increased by 2.
- *One perfect match and one deletion* occurred — states x_1 and x_2. Thus $|v| - 1 = 2 - 1 = 1$. Consequently GD is incremented by 1 and GP by 1 too.
- *Two perfect matches and no deletions* occurred — state x_5. In this case, two perfect matches have occurred, and $P = 2$. Thus $|v| - 2 = 2 - 2 = 0$. Consequently GD is unchanged and GP is incremented by 2.

Table 7.4 also requires that GP be updated at these states by the current value of P, while GM and GI remain unchanged.

Mismatch cascading states and counter updating
The mismatch cascading states are labelled by the character $M\bullet$. These are, namely, x_4, x_6, x_7 and x_8. These mismatch cascading states can originate as follows:

- *One perfect match and one mismatch* — states x_4 and x_7.
- *Two mismatches* — state x_6 and state x_8.

A dashed blue line from the relevant x-state to the M pseudo-state indicates that the M counter will be incremented upon reaching the x-state. Table 7.4 indicates how the various global counters are updated upon reaching one of these states, namely GM is updated by M, whereas GD and GI remain unchanged. Note that GP is updated by $|v| - M$. This is because the length of a path from the source state to a mismatch state is designed to be exactly $|v|$, and perfect matches along this path are not necessarily counted by the the pseudo-counter P. Instead, the number of perfect matches is given by $|v| - M$.

Insertion cascading states and counter updating
The dashed green line of Figure 7.8(b) from a state to the I pseudo-state indicates that counter I should be incremented whenever the state is reached.

There is only one insertion cascading state namely state x_{10}. It is marked by the character $I\bullet$. There are various paths from x_0 to x_{10}. They differ in the number of states they cross that cause I to increment. A path may indicate one, two, three or four insertions respectively. A selection of possible paths from x_0 to x_{10} are listed below, together with an indication of the resulting insertion count:

- *One insertion*: $\langle x_0, x_1, x_4, x_{10}\rangle$
- *Two insertions*: $\langle x_0, x_3, x_6, x_9, x_{10}\rangle$
- *Three insertions*: $\langle x_0, x_2, x_7, x_{11}, x_9, x_{10}\rangle$
- *Four insertions*: $\langle x_0, x_3, x_8, x_{11}, x_9, x_9, x_{10}\rangle$

Table 7.4 indicates that GI is increased by the number of insertions counted in I en route to state x_{10}. It also indicates that along all paths leading to the insertion cascading state, exactly $|v|$ perfect matches occur, and hence GP is updated by $|v|$. Furthermore, since no mismatches or deletions occur along these paths, GM and GD are unchanged.

7.1.6 Building and traversing an NCA$_{T3}$

Although FireSat$_2$ is intended to detect TRs with fairly large motifs (i.e. mini satellites), this subsection illustrates how the NCA$_{T3}$ is built by considering a very short motif, namely $\rho = \texttt{aac}$. Extending the principles illustrated here to longer motifs is straightforward.

The substrings with unique characters that make up $\rho = \texttt{aac}$ are $\upsilon_1 = \texttt{a}$ and $\upsilon_2 = \texttt{ac}$. FireSat$_2$ therefore builds an NCA$_{T3}$ that recognises υ_1 and all its mutational variants, followed by υ_2 and all its mutational variants. This is illustrated in Figure 7.9.

The centre of Figure 7.9(a) shows Mono$_1$, a mono-CA$_{T3}$ recognising a and all its mutational variants. Mono$_1$ has four cascading states: q_0, q_1, q_2 and q_3. The figure symbolically indicates that a di-CA$_{T3}$ is to be concatenated onto the mono-CA$_{T3}$ *at each of these four cascading states.* Four squares, connected to the cascading states with dotted lines, show where the cascade operation is executed. The four di-CA$_{T3}$s, named Di$_1$, Di$_2$, Di$_3$ and Di$_4$ are instances of the same complete di-CA$_{T3}$ already illustrated in Figure 7.8(b).

Note that the states of the di-CA$_{T3}$ in the latter figure were named $x_0 \ldots x_{11}$. To differentiate the states of Di$_1$, Di$_2$, Di$_3$ and Di$_4$ from one another, they will be named $p_0 \ldots p_{11}$; $r_0 \ldots r_{11}$, $s_0 \ldots s_{11}$ and $t_0 \ldots t_{11}$ respectively[3]. This is indicated in their respective squares.

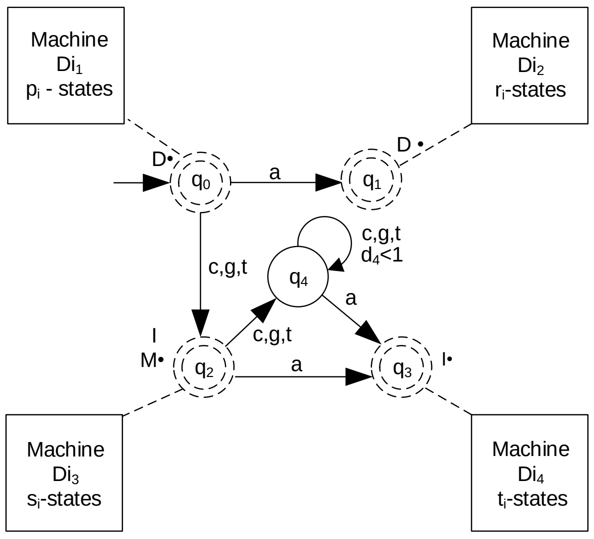
The four cascading operations attaching Di$_1$, Di$_2$, Di$_3$ and Di$_4$ onto the Mono$_1$ result in an NCA$_{T3}$ that has previously been called FA$_\rho$ or, in this case FA$_{\texttt{aac}}$. That FA$_{\texttt{aac}}$ is indeed *non-deterministic* can be seen by noting, for example, that the cascaded start state (resulting from merging states q_0 and p_0) has two transitions on a — one to q_1 and another to state p_1.

Because of this non-determinism, a given input string may trace more than one path through FA$_{\texttt{aac}}$ before ending in a final state. Different paths might result in different values for the global counters. Consider, for example, ac as input to FA$_{\texttt{aac}}$. Tables 7.5 and 7.6 illustrate two different paths through FA$_{\texttt{aac}}$ for this input.

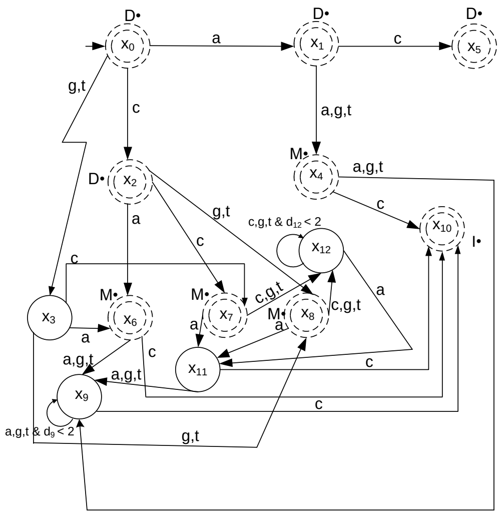
A row in the tables represents either a transition from one state to another, or a point where control is at a cascading state and global counters are to be updated, because the next transition is out of the Mono$_i$ into one of the concatenated Di$_j$'s. The tables also provide the associated counter updates that are associated with each row.

The column headings have the following meaning:

[3] Note that these names have been chosen for ease of reference, even though strictly speaking, in terms of the definition of cascading given in Definition 2.4.4, the new states resulting from the cascading operation are a subset of $\{q_0 \ldots q_3\} \times \{x_0 \ldots x_{11}\}$.


Machine Di_1 p_i - states
Machine Di_2 r_i-states
Machine Di_3 s_i-states
Machine Di_4 t_i-states
D•
D •
q_0
q_1
q_2
q_3
q_4
a
c,g,t
c,g,t d_4<1
I M•
I•

(a) Machine Mono_1.



(b) Machine Di_x.

Figure 7.9: A cascaded NCA_{T3} constructed from pCA_{T3}s catering for $\rho = aac$ where $\varepsilon_{max} = |\rho|$.

- Source input
  Refers to the character read from the source string. *None* indicates that the row provides information about an update to the global counters.
- Current machines

Source Input	Current Machines	States Traversed	Cascading Type	Pseudo Counters	Global Counters	Global Update
None	Mono_1 Di_1	$\langle q_0, p_0 \rangle$ *cascade*	Deletion	P = 0; M = 0; I = 0	GP = 0; GD = 1; GM = 0; GI = 0;	Yes
a	Di_1	$\langle p_0, p_1 \rangle$	Deletion	P = 1; M = 0; I = 0	GP = 0; GD = 1; GM = 0; GI =0;	No
c	Di_1	$\langle p_1, p_5 \rangle$ Final	Deletion	P = 2; M = 0; I = 0	GP = 2; GD = 1; GM = 0; GI = 0;	Yes

Table 7.5: First Tracing of `ac` on $\text{FA}_{\texttt{aac}}$ presented in Figure 7.9.

Source Input	Current Machines	States Traversed	Cascading Type	Pseudo Counters	Global Counters	Global Update
a	Mono_1	$\langle q_0, q_1 \rangle$	Deletion	P = 1; M = 0; I = 0;	GP = 0; GD = 0; GM = 0; GI = 0;	No
None	Mono_1 Di_2	$\langle q_1, r_0 \rangle$ *cascade*	Deletion	P = 1; M = 0; I = 0	GP = 1; GD = 0; GM = 0; GI = 0;	Yes
c	Di_2	$\langle r_0, r_2 \rangle$ Final	Deletion	P = 1; M = 0; I = 0	GP = 2; GD = 1; GM = 0; GI = 0;	Yes

Table 7.6: Second Tracing of `ac` on $\text{FA}_{\texttt{aac}}$ presented in Figure 7.9.

This column indicates the pCA_{T3} at which the next transition is to be made. If a global counter update is to be made because the next transition will be to another pCA_{T3}, then the current and next pCA_{T3} are mentioned.

- States traversed
  If a transition is to be made on the current input, then the column indicates the states involved. It also indicates if the destination state is a `final` state. On the other hand, if a transition will be made in the next row to a new pCA_{T3}, then the column gives the states of the pCA_{T3}s that were cascaded.

- Cascading type
  If the current state is a cascading state, then the cascading type of the cascading state is indicated.

- Pseudo counters
  Pseudo counters refer to the values of pCA_{T3} pseudo counters.

- Global counters
  This column shows the updated global counters. Updating takes place before a transition is made to a state of a new pCA_{T3}, *or* when a final state is reached.

- Global update
  A *Yes* entry indicates the updating of global counters, a *No* entry the contrary.

Table 7.5 shows the transitions through $\texttt{Mono}_1$ and $\texttt{Di}_1$ if `ac` is interpreted as a mutation of `aac` that has a deletion of the prefix `a` of `aac`, and thereafter matches the suffix `ac` of `aac`. In the first row, the deletion is reflected in the updated global counter GD. The formula given in Table 7.4 is used, namely $GD = GD + (|v| - P)$. Here P starts off as 0, and v refers to `a`, the string upon which $\texttt{Mono}_1$ is based. In the second and third rows, P is updated. Because a final state is reached in the third row, the global counters are also updated. Thus GP becomes 2, while GD remains unaltered at 1, and GM and GI unaltered at 0.

Table 7.6 shows the transitions through $\texttt{Mono}_1$ and $\texttt{Di}_2$ if `ac` is interpreted as a mutation of `aac` that has a match at prefix `a` of `aac`, and then a deletion at the second `a` of `aac`, and thereafter a match, the suffix `c`, of `aac`. In the first row, the match is reflected in the updated pseudo- counter P. Note that this is the pseudo-counter of $\texttt{Mono}_1$. In the second row, the formula given in Table 7.4 is used to update GP, namely $GP = GP + P$. In the third row, the pseudo-counter P of $\texttt{Di}_2$ is updated to 1. Because a final state is reached in the third row, the global counters are also updated. Thus GP becomes 2 and $GD = GD + (|v| - P) = 0 + (|\texttt{ac}| - 1) = 1$. Note that in this case, v refers to the string upon which $\texttt{Di}_2$ was built, namely `ac`.

There are no other paths tracing `ac` through $FA_{\texttt{aac}}$. In both of the above cases, the total number of mutational errors is given by the sum of the global counters, namely $GM + GI + GD = 1$. This provides enough information to yield the LD between `aac` and `ac`, namely $min(\{1, 1\}) = 1$.

It is co-incidental that in this example, the global counters following the two different paths ended up with the same values. This need not always be the case. In general there could be multiple paths, and the total number of mutational errors may differ. Finding the minimum would yield an estimate of the LD between ρ and the input string. However, since the approach here relies on pCA_{T3}s that allow for at most 4 insertions before a match occurs, it is not certain that the minimum of $GM + GI + GD$ over all possible paths will *always* yield the LD. To allow for unusual cases where the LD is based on five or more successive insertions, we speak here of the LBD — Levenshtein-based distance.

$\texttt{FireSat}_2$ broadly follows the logic illustrated above. It selects a candidate TRE that starts at the current position in the input string s. For that candidate TRE, all possible paths through FA_ρ are traced. The total number of errors recorded by the global counters is computed for each path and used to compute the LBD between ρ and the selected TRE. There could be several candidate TRE strings — some longer than ρ because mutations are interpreted as insertions; and some shorter than ρ, because mutations are interpreted as deletions. In the end, the candidate TRE is selected that has the minimum LBD between ρ and itself from amongst all other candidate TREs.

In the foregoing discussion, the LBD was based on $GM + GI + GD$. Clearly, this sum should not exceed the user-specified value for the maximum motif error,

ε_{max}. Candidate TREs that do not comply with this requirement are eliminated from consideration.

Additionally, recall that `FireSat`$_2$ allows the user to specify a weight for each mutational error type. If these are specified, then the LBD will be based on the appropriately weighted sum of global mutational errors.

Once the best candidate TRE has been identified, `FireSat`$_2$ has to apply the other appropriate filter tests to verify whether or not to search for a next TRE of the TR found to date.

7.1.7 Managing ε_{max}

Assume that $\rho = \upsilon_1\upsilon_2 \cdots \upsilon_n$ where each υ_i consists of unique elements. This section has shown how to construct FA$_\rho$ that consists of n mono-, di-, tri- and/or quad-pCA$_{T3}$s. Each pCA$_{T3}$ recognises an υ_i as well as mutations on υ_i The maximum number of mutations allowed on each pCA$_{T3}$ is $|\upsilon_i|$.

As indicated above, the sum of global counters is compared against the user-specified value of ε_{max}. Suppose, for example, that $\varepsilon_{max\%}$ was specified to be 50%. This allows for a scenario where the first 50% of a source string fully matches the first 50% of the destination string, whereas the remainder of the source and destination string do not match at all.

An alternative way to ensure that motif errors are more evenly spread is to regulate the number of motif errors for each consecutive υ_i too. This can be accomplished by introducing further restrictions on the mutational errors. The precise details of how to do this, and the possible benefits are, however, beyond the scope of this study.

7.1.8 Ambiguities: `FireSat`$_2$

As previously mentioned, `FireSat`$_2$ calculates an LBD between two strings. The proposed LBD is limited in the sense that a perfect match always has to follow after one or more insertions. Thus a mismatch can, for example, not follow an insertion. Section 7.1.6 illustrated *how* a `FireSat`$_2$ NCA$_{T3}$ is built and traversed. From that discussion it is clear that all paths, leading to the final states of an NCA$_{T3}$, within filter bounds, are traversed. If more than one *shortest* LBD is calculated between a source and a destination string — i.e. if there is more than one way of assigning mutation errors to yield the same LBD between the two strings — then `FireSat`$_2$ gives priority to mismatches above deletions and deletions above insertions.

7.1.9 Conclusion: $\texttt{FireSat}_2$

It is challenging to implement $\texttt{FireSat}_2$. The main reason is that for each nucleotide a *new* automaton should be built[4] until a TR is detected. After a TR (or the absence of a TR) is reported, the construction of a new instance of FA_ρ proceeds, based on a new value of ρ, and this is continued until the entire input string, s has been traversed. The principles described above are applied when automata are constructed. It is computationally inefficient to build these automata, in terms of both time- and space efficiency. $\texttt{FireSat}_{2'}$ was designed to address the shortcomings of $\texttt{FireSat}_2$.

Chronologically, $\texttt{FireSat}_3$ was developed before $\texttt{FireSat}_{2'}$. However, $\texttt{FireSat}_3$ is based on somewhat different principles to both $\texttt{FireSat}_2$ and $\texttt{FireSat}_{2'}$ and for that reason, it is addressed in the next chapter, while $\texttt{FireSat}_{2'}$ is described in the section that now follows.

7.2 $\texttt{FireSat}_{2'}$

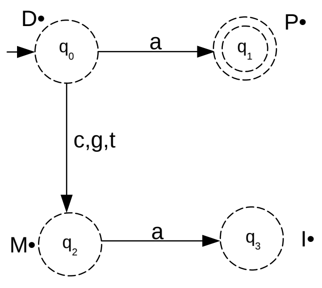
Each of the algorithms $\texttt{FireSat}_1$, $\texttt{FireSat}_2$ and $\texttt{FireSat}_{2'}$ have the same broad structure shown in Algorithm 6.3.1 in the sense that the length of the motif, l, is held constant in the body of the outer for-loop while the motif itself changes for each iteration of the inner do-loop.

In the case of $\texttt{FireSat}_1$, $|\upsilon_i| = 4$ for all i (except possibly υ_n). This means that for each instance of ρ within the inner do-loop, the *same number* of quad-pDFAs (plus a tail of a given length pDFA) are used to build FA_ρ. As a result, the structure of FA_ρ remains the same for each of these instances. Only the labels on the transitions need to be changed to match each new selection of ρ within the inner do-loop.

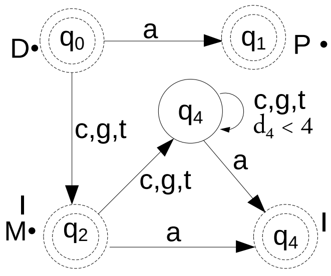
In the case of $\texttt{FireSat}_2$, υ_i is selected to be as long as possible, subject to the constraint that all of its elements differ from one another. This means that the pCA_{T3}s that are used to build FA_ρ change with each new selection of ρ. It is no longer possible to use the same structure for FA_ρ from one iteration of the inner do-loop to the next and merely relabel the transitions. This fact significantly slows down $\texttt{FireSat}_2$. Although $\texttt{FireSat}_2$ incurs this efficiency penalty, it is able to handle mutational errors in the TRs that are sought more flexibly than $\texttt{FireSat}_1$.

$\texttt{FireSat}_{2'}$ seeks to have the efficiency advantage of $\texttt{FireSat}_1$ while retaining $\texttt{FireSat}_2$'s flexibility in handling mutational errors. In order to do this, $\texttt{FireSat}_{2'}$

[4]If a TRE (starting at $|\rho| + 1$) relative to the current FA_ρ is not found, a next nucleotide start position, thus a *new* motif is being considered as ρ.



(a) A mono-pCA$_{T3}$ where at most 1 insertion is allowed before a perfect match.



(b) A mono-pCA$_{T3}$ where $0 \leq e \leq 4$ insertions are allowed before a perfect match.

Figure 7.10: Mono-pCA$_{T3}$s where $\upsilon =$ a.

selects υ_i to be as short as possible, namely $|\upsilon_i| = 1$. This means that only mono-prototype finite automata are used to build FA$_\rho$. However, as will be seen below, these are not conventional mono-pCA$_{T3}$s as described above, but slightly adapted mono-pNCA$_{T3}$s. The details of these mono-pNCA$_{T3}$s are shown in Section 7.2.1.

This adaptation means that the overall structure of FA$_\rho$ may be retained from one iteration of the inner do-loop to the next and only labels need to be changed to match each new instance of ρ.

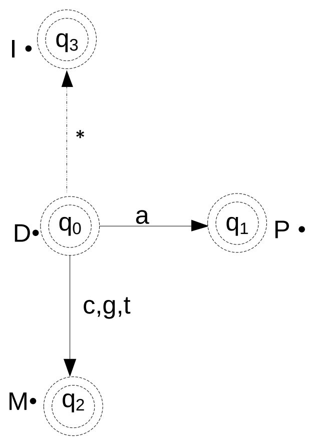
It also means that the flexibility in handling mutational errors is retained. Just as in `FireSat`$_2$, for each type of mutational error, tabs are kept of error counts local to each mono-pNCA$_{T3}$, as well as of global error counts at each part of the resulting NCA$_{T3}$. Issues relating to counter updating are discussed in Sections 7.2.2 and 7.2.3.

7.2.1 The pNCA$_{T3}$s used in `FireSat`$_{2'}$

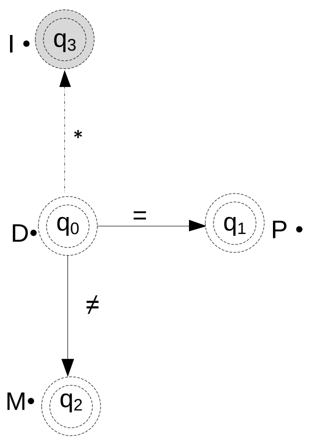
Both the mono-pCA$_{T3}$s depicted in Figure 7.10 were considered for constructing a FA$_\rho$. However, an NCA$_{T3}$ composed from either of these mono-pCA$_{T3}$s is not sufficiently flexible to deal with the various possible mutational errors. Specifically, these structures do not allow for a mismatch or deletion to occur after an insertion.

These deterministic pCA$_{T3}$s were therefore revised to the *non-deterministic* pNCA$_{T3}$ presented in Figure 7.11(a). (Figure 7.11(b) shows the structure of this pNCA$_{T3}$ in an abstract form.) pNCA$_{T3}$s such as this one are used in `FireSat`$_{2'}$.

All four the states of the pNCA$_{T3}$ in Figure 7.11(a) are cascading states. State q_0



(a) A $\texttt{FireSat}_{2'}$ mono-pNCA$_{T3}$ catering for $v = \texttt{a}$ with $0 \leq e \leq 1$.



(b) A mono-pNCA$_{T3}$ with a higher level of abstraction.

Figure 7.11: $\texttt{FireSat}_{2'}$: mono-pNCA$_{T3}$s.

is a deletion cascading state; state q_1 is a perfect match cascading state (labelled $P\bullet$); state q_2 is a mismatch cascading state and state q_3 an insertion cascading state. Note that the $\langle q_0, q_3\rangle$ transition is available on *any* character as a non-deterministic possibility.[5] If cascading states are also regarded as final states, then the language of the pNCA$_{T3}$ in Figure 7.11(a) is $\{\Lambda, \texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$. This would be the case for a pNCA$_{T3}$, representing any target string `a`, `c`, `g` or `t`.

However, the cascading type of the (final) state at which each string in the language of a pNCA$_{T3}$ for v terminates, also determines the mutational error type of that string relative to v. pNCA$_{T3}$s such as the one in Figure 7.11(a) have been designed to tolerate at most 1 mutational error. Moreover, because of the non-determinism, the error type may sometimes be ambiguous.

Thus, referring the pNCA$_{T3}$ in Figure 7.11(a), the source string `a` may be interpreted as a perfect match (if cascading state q_1 is reached) or as an insertion (if cascading state q_3 is reached); the source string `c` may be interpreted as a mismatch (if cascading state q_2 is reached) or as an insertion (if cascading state q_3 is reached); etc.

The next section illustrates how pNCA$_{T3}$s are cascaded and how counter updating is done.

[5] *Any* character is represented by " * ". Within the current context " * " implies that *no* character from the *destination string* is used for decoration of the insertion transition. Consequently the insertion cascading state *cascades* to the current NCA$_{T3}$. This is illustrated in Table 7.8.

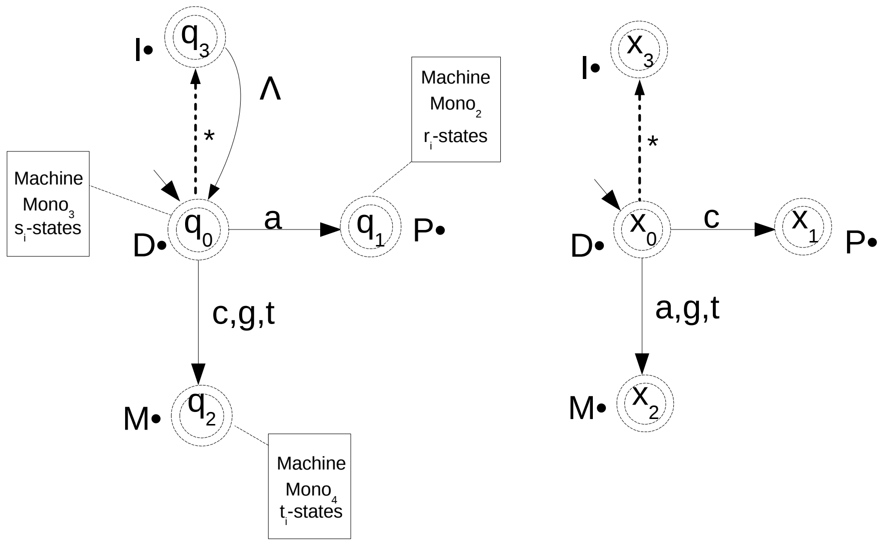


Figure 7.12: An NCA_{T3} accepting $\rho =$ ac and any substring derived from ρ where $\varepsilon \leq 2$.

7.2.2 Cascading $pNCA_{T3}$s and doing counter updating

Consider FA_ρ when $\rho =$ ac and where $|\varepsilon \leq 2|$ — i.e. up to 2 errors of any mutational type are to be tolerated in the string ac.

The right hand side of Figure 7.12 shows a $pNCA_{T3}$ for c and all its mutational variants. Here the states are generically labelled as $X_i, i = 0, \ldots 3$.

The left hand side of Figure 7.12 shows a $pNCA_{T3}$ for a and all *its* mutational variants. Here the states are labelled $q_i, i = 0, \ldots 3$. Each of the cascading states q_0, q_1 and q_2 is connected to a rectangle. Each rectangle represents an instance of the $pNCA_3$ for c that has been cascaded onto the respective cascading state of the $pNCA_{T3}$ for a. In carrying out the cascading operation the states generically labelled as X_i are to be relabelled s_i, t_i and r_i, respectively, $i = 0, \ldots 3$. Note that the insertion machine returns to state q_0. Thus the machine remains the one where the insertion has been read. The state traversal $\langle q_0, s_0 \rangle$ takes place when a deletion is deemed to have occurred, the state traversal $\langle q_1, r_0 \rangle$ takes place when a match is deemed to have occurred, the state traversal $\langle q_2, t_0 \rangle$ takes place when a mismatch is deemed to have occurred, the state traversal $\langle q_3, q_0 \rangle$ takes place when an insertion is deemed to have occurred.

Using the same conventions as in Section 7.1.6, Table 7.7 illustrates how `ac` as *source* string may traverse the resulting $FA_{\texttt{ac}}$ and be interpreted as a perfect match against the target string `ac`.

Another way of tracing source string `ac` on $FA_{\texttt{ac}}$ is to consider `a` as an insertion. This is displayed in Table 7.8. Since the total error count in this case, and indeed, in all other possible traces of source string `ac` through $FA_{\texttt{ac}}$, is more than when tracing `ac` as a perfect match, the LD between `ac` and `ac` is 0, and the source string is seen as a perfect match against the target string.

Table 7.9 illustrates one way that source string `ag` will traverse the same machine, $FA_{\texttt{ac}}$. In this case, `g` is interpreted as a mismatch and the distance between source and target strings `ag` and `ac` is 1. All other interpretations of `ag` will also result in at least a distance of 1, so that the LD between these two strings is 1.

Table 7.7: Tracing `ac` as a perfect match on $FA_{\texttt{ac}}$.

Source Input	Current Machines	States Traversed	Cascading Type	Counters	Global Counters	Cascading Here
a	$Mono_1$	$\langle q_0, q_1\rangle$	Perfect	P = 1; D = 0 ; M = 0; I = 0	GP = 0; GD = 0; GM = 0; GI = 0;	No
None cascade	$Mono_1$ $Mono_2$	$\langle q_1, r_0\rangle$ *cascade*	Perfect	P = 1; D = 0; M = 0; I = 0	GP = 1; GD = 0; GM = 0; GI =0;	Yes
c	$Mono_2$	$\langle r_0, r_1\rangle$ Perfect Final	Perfect	P = 1; D = 0 M = 0; I = 0	GP = 2; GD = 0; GM = 0; GI = 0;	No

Table 7.8: Tracing `ac` through $FA_{\texttt{ac}}$ where `a` is regarded as an insertion.

Source Input	Current Machines	States Traversed	Cascading Type	Counters	Global Counters	Cascading Here
a	$Mono_1$	$\langle q_0, q_3\rangle$	Insertion	P = 0; D = 0 ; M = 0; I = 1	GP = 0; GD = 0; GM = 0; GI = 0;	No
None return	$Mono_1$ $Mono_1$	$\langle q_3, q_0\rangle$ *return*	Insertion	P = 0; D = 0; M = 0; I = 1	GP = 0; GD = 0; GM = 0; GI = 1;	No
c	$Mono_1$	$\langle q_0, q_2\rangle$ Mismatch Final	Mismatch	P = 0; D = 0 M = 1; I = 0	GP = 0; GD = 0; GM = 1; GI = 1;	No

Table 7.9: Tracing `ag` on $FA_{\texttt{ac}}$ where `g` is regarded as a mismatch.

Source Input	Current Machines	States Traversed	Cascading Type	Counters	global Counters	Cascading Here
a	$Mono_1$	$\langle q_0, q_1\rangle$	Perfect	P = 1; D = 0 ; M = 0; I = 0	GP = 0; GD = 0; GM = 0; GI = 0;	No
None cascade	$Mono_1$ $Mono_4$	$\langle q_1, r_0\rangle$ *cascade*	Perfect	P = 1; D = 0; M = 0; I = 0	GP = 1; GD = 0; GM = 0; GI =0;	Yes
g	$Mono_4$	$\langle r_0, r_2\rangle$ Mismatch Final	Mismatch	P = 0; D = 0 M = 1; I = 0	GP = 1; GD = 0; GM = 1; GI = 0;	No

Cascading Type	Global Counter Updating

Mismatch	GM = GM + 1;
	GP = GP;
	GI = GI;
	GD = GD;

Insertion	GM = GM;
	GP = GP;
	GI = GI + 1;
	GD = GD;

Deletion	GM = GM;
	GP = GP;
	GI = GI;
	GD = GD + 1;

Perfect	GM = GM;
	GP = GP + 1;
	GI = GI;
	GD = GD;

Table 7.10: `FireSat`$_{2'}$: the updating of global counters.

Table 7.10 is analogous to Table 7.4, showing how global counters are to be updated in the case of `FireSat`$_{2'}$. Since we are dealing with mono-NCA$_{T3}$s only, the updating of the global counters entails simple incrementation by 1, in contrast to the more elaborate incrementation needed in Table 7.4.

7.2.3 The number of states: pruning

The principles illustrated in the previous section can be used to build an NCA$_{T3}$ that recognises ρ of arbitrary length. The number of states depends both on $|\rho|$ and on $\varepsilon_{max\%}$. The exercise described below illustrates this point.

The number of states in an NCA$_{T3}$ that caters for a string of length $|\rho|$ where $\varepsilon_{max\%} = 100\%$, is given by Equation 7.1.

$$\text{State}_{count}(n) = 1 + \sum_{n=1}^{|\rho|} (3)^n \tag{7.1}$$

Figure 7.13 illustrates the exponential growth in the number of states represented by this equation. The y-axis is on a logarithmic scale and shows the number of states in an NCA$_{T3}$ that recognises a string of length $|\rho|$, where $|\rho|$ is given on the x-axis. The blue line shows the number of states as determined by Equation 7.1, i.e. when $\varepsilon_{max\%} = 100\%$. In this case, an NCA$_{T3}$ catering for a string of length $|\rho| = 20$ will have more than 500 million states.

However, the number of states is significantly reduced by pruning. The cyan curve in Figure 7.13 represents the number of states for $|\rho|$ when $\varepsilon_{max\%} = 20\%$. Note that this curve was determined empirically by building the actual NCA$_{T3}$s. The number of states when $|\rho| = 20$ reduces to approximately 79 000.

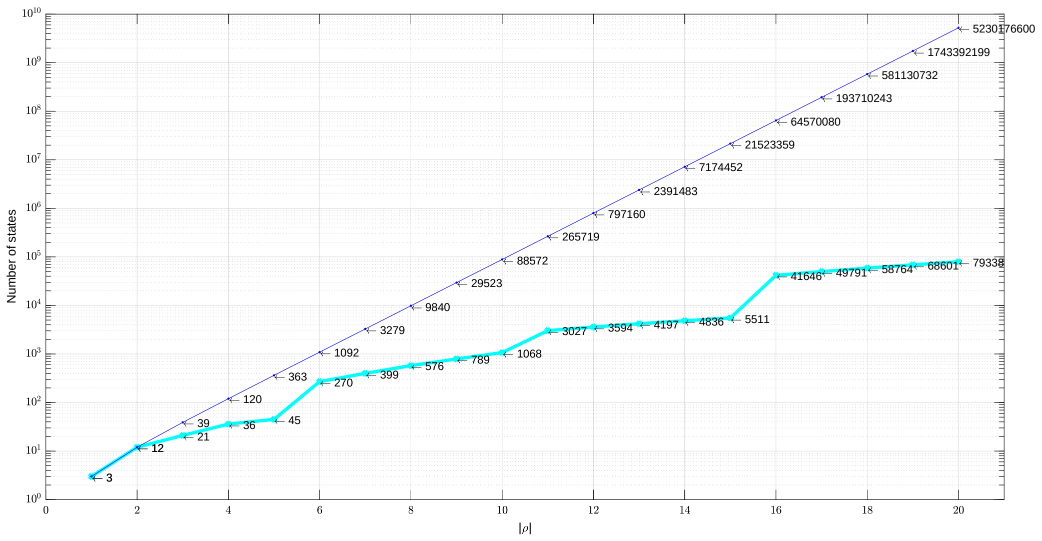


Figure 7.13: FireSat$_{2'}$: the number of states represented by the blue line when $\varepsilon_{max\%} = 100\%$. The curve in cyan illustrates how the number of states is reduced when $\varepsilon_{max\%} = 20\%$.

7.3 Concluding remarks

The foregoing subsections illustrated extensively how quad-, tri-, di- and mono-pCA$_{T3}$s can be used to recognise a *specified* string and a *specific* mutation type of that string. The original idea was to use pCA$_{T3}$ counters to keep count of the number of mutations of a given type.

It was also shown how larger quad-, tri-, di- and mono-pCA$_{T3}$s can be composed from the various smaller pCA$_{T3}$s so as to recognise *all* mutation types of a given string. The sum operation on DFAs was used for this purpose. Unfortunately, adding these pCA$_{T3}$s means that it is no longer possible to cater for mismatches, insertions and deletions separately at the same state, since there is a single counter at each state. Instead, mismatch, insertion and perfect *pseudo-states* were introduced whose counters — M, I and P — establish the total number of occurrences of mismatches, insertions and perfect matches within the larger pCA$_{T3}$.

These larger pCA$_{T3}$s were then cascaded together to build FA$_\rho$ — an NCA$_{T3}$ that recognises a motif, ρ, and acceptable variations of that motif. To keep track of the overall count of motif errors and perfect matches, global counters (GP, GM, GI and GD) were introduced. These are used to check whether or not the substring currently under consideration constitutes an acceptable TRE.

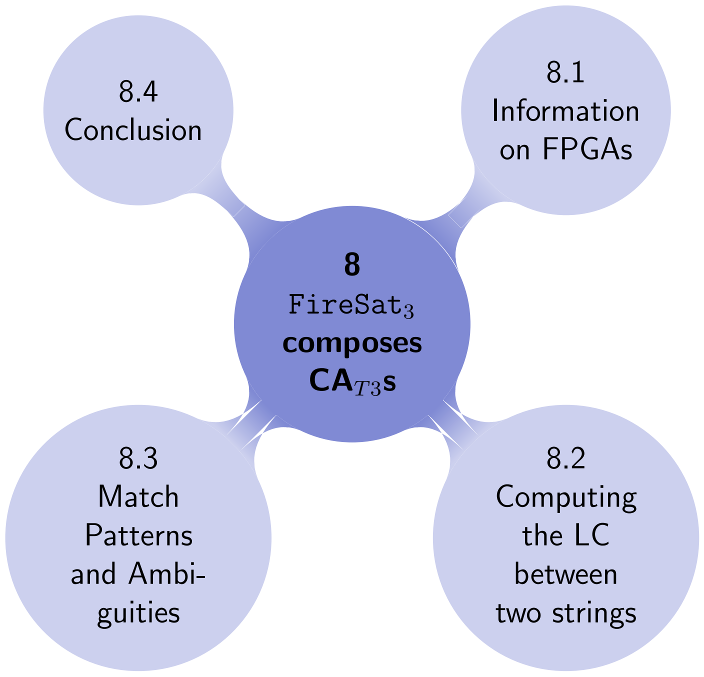
`FireSat`$_2$ is however hampered by two limitations. Firstly, the entire FA$_\rho$ has to be re-built for each new value of ρ. This is computationally expensive. Consequently `FireSat`$_2$ is impractical to implement, especially as $|\rho|$ becomes larger. Secondly, as explained earlier, `FireSat`$_2$ can only calculate an LBD and not the LD itself.

To improve on these shortcomings `FireSat`$_{2'}$ was introduced. In this case, mono-pNCA$_{T3}$s are cascaded to construct an NCA$_{T3}$. This NCA$_{T3}$ has the ability to calculate the LD between a source and a destination string. Chapter 9 reports on the empirical results of both `FireSat`$_2$ and `FireSat`$_{2'}$.

Chapter References

J. Holub. *Finite Automata in pattern matching.* John Wiley & Sons inc., 1st edition, 2010. ISBN 978-0-470-50519-9.

B. Melichar. Space complexity of linear approximate string matching. In *Proceedings of the Prague Stringology Club Workshop '96*, pages 28—36, 1996. URL http://www.stringology.org/event/1996/p4.html.


8.4
Conclusion
8.1
Information on FPGAs
8
FireSat$_3$ composes CA$_{T3}$s
8.3
Match Patterns and Ambiguities
8.2
Computing the LC between two strings

8

`FireSat`$_3$ **composes CA$_{T3}$s**

" A man who has depths in his shame meets his destiny and his delicate decisions upon paths which few ever reach... "... Friedrich Nietsche

---

In seeking the next TRE to be added to a TR found to date, `FireSat`$_3$ computes the LC between the current motif (the destination string) and an appropriate set of source strings. From this set, a string with the largest LC is selected as the next TRE.

`FireSat`$_3$ uses a slightly modified form of the dynamic programming algorithm discussed in Chapter 2 to compute the LC. The modification entails a first phase during which all perfect matches between relevant substrings of the destination and source strings are recorded. To do this, `FireSat`$_3$ makes use of a set of CA$_{T3}$-based machines. I call this set a Levenshtein correspondence automaton (LCA).

Note that an LCA is not strictly necessary for computing an LC. It was conceived in line with the central theme of this thesis: to investigate the use of FA-technology for TR detection. However, the LCA's architecture also suggests how a version of `FireSat`$_3$ could be built into a field-programmable gate array (FPGA) or implemented on a graphical processing unit (GPU). These hardware platforms allow for a high degree of parallelism. Section 8.1 gives a brief introduction to FPGAs. Subsequent sections explain an extended version of the LCA that would enable an FPGA implementation. However, it is beyond the scope of this thesis to implement and test parallelised FPGA- or GPU-based versions of `FireSat`$_3$.

In Section 8.2, therefore, five steps are given explaining *how* to construct the LCA and how to calculate the LC from it. The extension needed for FPGA implementations will be highlighted. Section 8.3 explains how `FireSat`$_3$ deals with ambiguities. Section 8.4 concludes this chapter.

8.1 Information on FPGAs

Xilinx Inc introduced commercial FPGAs in 1985. Broadly speaking, FPGAs may be thought of as reprogrammable silicon chips (Guerra [2016]). More specifically an FPGA is a general-purpose, multi-level programmable logic device. When an FPGA is configured (using a hardware description language), the internal circuitry is connected in a way that creates a hardware implementation of some software application. FPGAs are thus configured by end users to accomplish a specific task. This is possible as FPGAs are composed of blocks of logic connected with so-called programmable interconnects[1].

FPGAs provide hardware-timed speed and reliability. A reprogrammable FPGA has the same flexibility as software running on a processor-based system, but it is not as limited by the number of processing cores available. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not necessarily affected when one adds more processing (Guerra [2016]). In general, to justify the relatively large upfront expense of embedding custom-designed application specific integrated circuits (ASICs), a high intensity of usage of the FPGA application should be anticipated.

Numerous books have been written on FPGAs and how to configure them. Trimberger [1994] and Koch et al. [2016] are two such examples.

Systolic array-based[2] FPGAs have been implemented for DNA comparison as well as for protein sequence comparison. See, for example, Guo et al. [2012]; Hoang

[1]Programmable interconnect resources are electrically programmable interconnections (pre-laid vertically and horizontally) that provide the so called routing path for the programmable logic blocks. Routing paths contain wire segments of varying lengths that can be interconnected via electrically programmable switches (Guerra [2016]).

[2]A systolic array is a grid-like structure consisting of special processing elements that pro-

and Lopresti [1993] and Becker et al. [2004]. It will be seen that `FireSat`$_3$ is particularly suitable for systolic array FPGA implementation.

8.2 Computing the LC between two strings

Suppose that the LC is required between an arbitrary source and destination string, denoted by $S = s_0 s_1 \ldots s_{|S|-1}$ and $R = r_0 r_1 \ldots r_{|R|-1}$ respectively[3].

This section explains how an LCA can be built to assist in finding the LC between S and R. An LCA consists of a number of CA$_{T3}$s, concatenated together in a specific way. When relevant substrings of S and R are run on these CA$_{T3}$s, their state counters store information about perfect matches that occur between these substrings. The dynamic programming algorithm described in Chapter 2 can then be used to compute the desired LC by accessing this counter information in the LCA.

The process of constructing the LCA and then determining an LC can be divided into five steps that will be fully discussed below. In overview, the 5 steps are as follows:

- *Step 1*
  Construct an LCA template based on the respective lengths of the source and destination strings, $|S|$ and $|R|$. This is accomplished by determining both the number of CA$_{T3}$s in the LCA as well as the number of states in each of the CA$_{T3}$s. The states of the CA$_{T3}$s are topologically arranged in a matrix that corresponds to the matrix in Figure 2.26. Details of this step are in Subsection 8.2.1.

- *Step 2*
  Use appropriate prefixes and suffixes of S and R to determine the transition labels in the various CA$_{T3}$s in the LCA template. This process is called *decorating* the LCA. Subsection 8.2.2 explains this step.

- *Step 3*
  Use S and R to construct an appropriate string for each CA$_{T3}$ in the LCA.

cesses data in a manner similar to an n-dimensional pipeline. However, unlike a pipeline, the input data as well as partial results flow through the array. Additionally, data can flow in a systolic organisation at multiple speeds in multiple directions. Systolic arrays have a high rate of input/output. These arrays are well-suited for intensive parallel operations (Johnson et al. [1993]).

[3] $R = r_0 r_1 \ldots r_{|R|-1}$ is used to refer to the destination (or reference) string instead of $D = d_0 d_1 \ldots d_{|D|-1}$. This is to avoid notational ambiguity, since d_i has been used throughout to refer to counters of CA$_{T3}$s.

Run these strings on their respective CA$_{T3}$s. The resulting counter values in the various CA$_{T3}$ states will then be either 0 or 1. Recall that when the dynamic programming algorithm fills in the cell of a matrix, it has to investigate whether or not the cell represents source and destination characters that match. This step ensures that a state counter value of 1 indicates that the corresponding characters do indeed match, and a counter value of 0 indicates the contrary. This step is elaborated in Subsection 8.2.3.

- *Step 4*
  This step shows how to extend certain CA$_{T3}$s in the LCA so that an *extended* matrix of states results. This extended matrix of states can be mapped to an extended matrix used for dynamic programming computations. The extension allows for the computations to be carried out in a *mirrored diagonally*[4] based fashion, such that the computations along each diagonal can be carried out in parallel. Note that the version of `FireSat`$_3$ implemented in this thesis *did not* carry out this step. This step is elaborated on in Subsection 8.2.4.

- *Step 5*
  This step entails the calculation of the LC. It is explained in Subsection 8.2.5 in terms of the original (i.e. unextended) matrix of states.

  If the extended matrix mentioned in Step 4 is used in the context of an FPGA or GPU implementation, then calculation of the LC is somewhat different. Details are given in Subsection 8.2.6.

8.2.1 Step 1: Constructing an LCA

`FireSat`$_3$ is based on the construction of an LCA. It will be seen that an LCA consists of several component machines, each of which is a slightly modified version of a CA$_{T3}$. I have called such a component machine an LC-CA$_{T3}$. These machines will be used in an LCA in an interrelated fashion to determine the LC between two strings, as outlined below.

Note that CA$_{T3}$s are the building blocks of an LCA. To explain what an LC-CA$_{T3}$ is and *how* it works, this section starts off with an example of an LC-CA$_{T3}$.

An example LC-CA$_{T3}$

Figure 8.1 displays an LC-CA$_{T3}$. As seen in the figure, an LC-CA$_{T3}$ allows for both conventional transitions as well as for *neutral* transitions. A neutral transition does not increment the counter of its destination state. Graphically such a transition is depicted by a two-headed arrow ($\twoheadrightarrow$). Note that the introduction of

[4]The term *mirrored diagonally* is explained in Section 8.2.4.

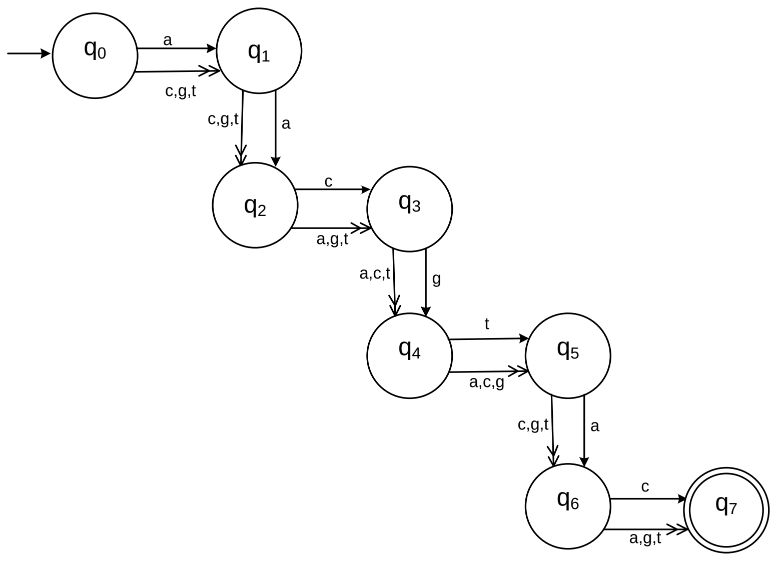


Figure 8.1: An LC-CA$_{T3}$.

neutral transitions is merely a notational convenience to represent a CA$_{T3}$ more compactly and may be used in the following circumstances. Suppose that a CA$_{T3}$ has the following characteristics:

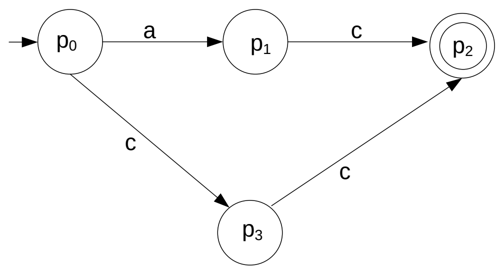
- states p_0, p_1, p_2 and p_3 and alphabet $\{\mathtt{a,c}\}$;
- transitions $\delta(p_0, \mathtt{a}) = p_1$, $\delta(p_1, \mathtt{c}) = p_2$, $\delta(p_0, \mathtt{c}) = p_3$ and $\delta(p_3, \mathtt{c}) = p_2$.
- Suppose that in the context of the application at hand, the counter of p_3 is entirely irrelevant.

In this case, without loss of generality, p_3 and the transition represented by $\delta(p_0, \mathtt{c}) = p_3$ may be replaced by a neutral transition from p_0 to p_1 on `c`. This is permissible because the counter of p_1 is not affected by the change, and the transitions from p_1 correspond identically to the transitions from the removed p_3. These principles are graphically illustrated in Figure 8.2.

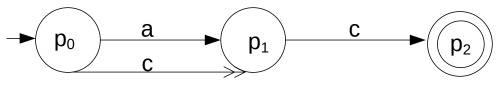
The LC-CA$_{T3}$ in Figure 8.1 accepts any string of length 7 that is built up from the alphabet $\{\mathtt{a,c,g,t}\}$. However, only `aacgtac` results in each of the counters, $d_1, \ldots d_7$, being incremented, each from 0 to 1.

As a result, if the string `accgtac` is run on the machine in Figure 8.1, then all the counters will have the value of 1, apart from d_2, which will still have the value 0.

The steps outlined next, in Subsection 8.2.1.1, provide general formulae for calculating the number of CA$_{T3}$s in terms of the lengths of the source and destination



(a) A CA$_{T3}$ without neutral (double) arrow heads.



(b) A CA$_{T3}$ with neutral (double) arrow heads.

Figure 8.2: Two CA$_{T3}$s where the counter values of d_1 and d_2, for `ac` and `cc` being input, respectively, will be the same.

strings, i.e. $|S|$ and $|R|$. It also shows how to calculate the number of states in each such CA$_{T3}$. In doing so, information is provided about the LCA template (that will subsequently be decorated) in terms of its component CA$_{T3}$s and about the terminological conventions that will be used to reference these components and their states. All of this is concretely illustrated in an example that follows in Subsection 8.2.1.2.

8.2.1.1 Dimensions of an LCA

1. The LC-CA$_{T3}$s needed are classed as source and destination LC-CA$_{T3}$s respectively. Exactly $\left\lceil \frac{|S|}{2} \right\rceil$ and $\left\lfloor \frac{|R|}{2} \right\rfloor$ source and destination LC-CA$_{T3}$s are needed, respectively.

2. It is convenient to arrange the states of the various LC-CA$_{T3}$s in a two-dimensional matrix. The columns are labelled $S[-1], S[0], \ldots S[|S|-1]$ and the rows are labelled $R[-1], R[0], \ldots R[|R|-1]$.

   The start states of the *source* LC-CA$_{T3}$s are positioned in row $R[-1]$. The start state of the first *source* LC-CA$_{T3}$ is at column $S[0]$ and thereafter in evenly marked columns, $S[2], S[4], \ldots$. The start states of the *destination* LC-CA$_{T3}$s are positioned in column $S[-1]$. The start state of the first *destination* LC-CA$_{T3}$ is at $R[1]$ and thereafter in odd numbered rows $R[3], R[5], \ldots$.

3. A source (destination) LC-CA$_{T3}$ will be referenced by the column (row, respectively) of its start state. Thus $S[i]$ (or $R[j]$) refers to the source (or destination) LC-CA$_{T3}$ whose start state is at column $S[i]$ (or row $R[j]$ respectively).

4. The number of states needed for LC-CA$_{T3}$ $S[i]$ (where $i = 0, 2, \ldots$) depends on $|S|$, on $|R|$ and on i. It also depends on whether or not $|S| \geq |R|$ holds. Formulae to compute these state numbers are given below. Dual formulae apply for the number of states needed for LC-CA$_{T3}$ $R[j]$. These formulae, too, are given below.

Let $|S[i]|$ and $|R[j]|$ denote the number of states in LC-CA$_{T3}$ $S[i]$ and LC-CA$_{T3}$ $R[j]$ respectively. If $|S| \geq |R|$ then:

$$|S[i]| = \left\{ \begin{array}{ll} 2|R|+1 & \text{if } i < (|S|-|R|) \\ 2|R| & \text{if } i = (|S|-|R|) \\ 2(|S|-i) & \text{if } i > (|S|-|R|) \end{array} \right\} \text{and } |R[j]| = 2(|R|-j) \tag{8.1}$$

If $|S| < |R|$ then:

$$|R[j]| = \left\{ \begin{array}{ll} 2|S|+1 & \text{if } j < (|R|-|S|) \\ 2|S| & \text{if } j = (|R|-|S|) \\ 2(|R|-j) & \text{if } j > (|R|-|S|) \end{array} \right\} \text{and } |S[i]| = 2(|S|-i) \tag{8.2}$$

It will be convenient to assign names to states in an LCA according to the following scheme: states in an LC-CA$_{T3}$ are numbered from 0 onward. Names are assigned to these states, using their number as a subscript to the name of their associated LC-CA$_{T3}$. Thus the start state of machine $S[0]$ is referenced by the name $S[0]_0$; the next state is referenced as $S[0]_1$; etc.

8.2.1.2 Example of an LCA Template

To illustrate the defined formulae and the principles employed, consider the LCA template in Figure 8.3, where $|R| = 6$ and $|S| = 7$; $r_i, s_i \in \{\texttt{a,c,g,t}\}$.

Figure 8.3 shows the following:

- The LCA consists of seven separate LC-CA$_{T3}$ machines. For reference purposes, their individual states are arranged to form a matrix collectively, as shown in the figure. The rows of the matrix are denoted by $R[-1], R[0], \ldots R[5]$ and the columns by $S[-1], S[0], \ldots S[6]$.

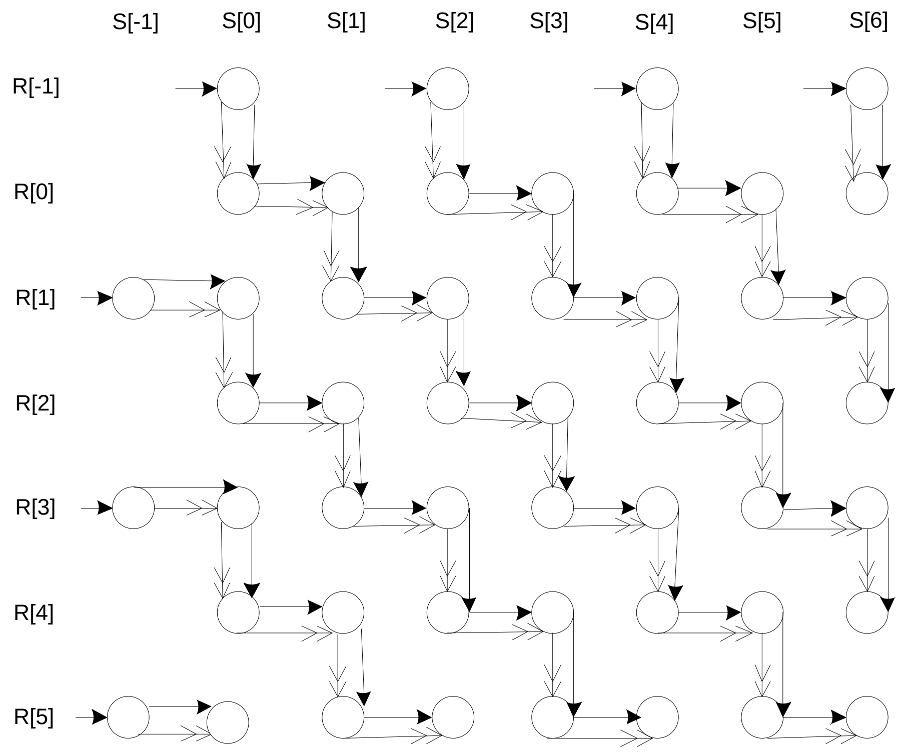


Figure 8.3: An LCA template assembled from LC-CA$_{T3}$ templates.

- The seven LC-CA$_{T3}$s divide into two classes: those whose start states are in the first row of the matrix (row $R[-1]$), and those whose start states are in the first column of the matrix (column $S[-1]$). Those whose start states are in $R[-1]$ are the *source* LC-CA$_{T3}$s and those whose start states are in $S[-1]$ are the *destination* LC-CA$_{T3}$s.

- There are four source LC-CA$_{T3}$s, each starting in row $R[-1]$. Their start states are, respectively, at columns $S[0], S[2], S[4]$ and $S[6]$. They are named $S[0], S[2], S[4]$ and $S[6]$ respectively.

- There are three destination LC-CA$_{T3}$s, each starting in column $S[-1]$. Their start states are, respectively, at rows $R[1], R[3]$ and $R[5]$. They are named $R[1], R[3]$ and $R[5]$ respectively.

- Notice the zig-zag pattern in which the states of the various LC-CA$_{T3}$s are arranged in the matrix. Notice, too, that each LC-CA$_{T3}$ has exactly the number of states specified by Equations 8.1 and 8.2. For example, considering $|S[4]|$ and noting that $|S| > |R|$, and $4 > |S| - |R|$, the third entry of Equation 8.1 correctly specifies that $|S[4]| = 2(|S|-4) = 2(7-4) = 6$.

Figure 8.3 can serve as a template for an arbitrary source string of size 7 and arbitrary destination string of size 6. In a later elaboration of this example, columns $S[0], S[1], \ldots S[6]$ will be labelled with a specific source string's elements, $s_0 s_1 \ldots s_6$. Rows $R[0], R[1], \ldots R[5]$ will be labelled with a specific destination string's elements, $r_0 r_1 \ldots r_5$. These will also be used to decorate (i.e. label) the various transitions between states in the LCA.

The foregoing therefore indicates the general approach for constructing an LCA template for arbitrary source and destination strings, provided that their lengths have been given.

Figure 8.4 illustrates various alternative LCA templates, each with different combinations of source and destination lengths.

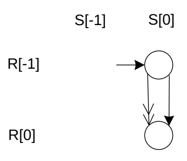
It will be seen that in all cases, Equations 8.1 and 8.2 correctly specify the number of states in each respective LC-CA$_{T3}$.

- In Figure 8.4(a), $|S| = |R| = 1$. This template is for the shortest allowable strings.
- In Figure 8.4(b) $|S| = |R| = 2$. This template provides another example of where the source string is equal to the destination string.
- In Figure 8.4(c), $|S| = 8$ and $|R| = 5$. Comparing this figure with Figure 8.3 highlights how the LCA changes with an additional source string element and one less destination string element.
- In Figure 8.4(d), $|S| = 4$ and $|R| = 7$. This figure illustrates an LCA template where the destination string is longer than the source string.

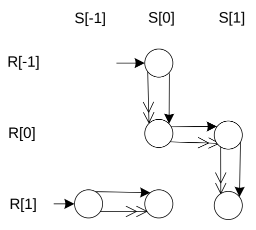
8.2.2 Step 2: Decorating an LCA

The LCA consists of all the *destination machines* and all the *source machines* as described in the previous section and illustrated in Figure 8.3. I next describe how to decorate the single arrowhead transitions of the LC-CA$_{T3}$s in a template LCA, when specific source and destination strings of appropriate lengths are available.

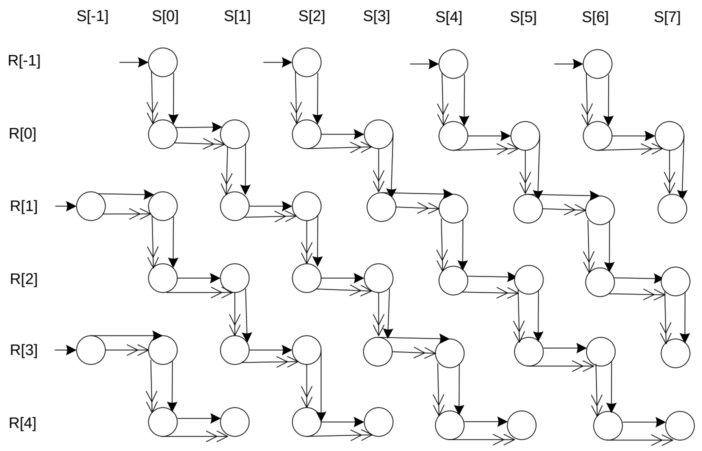
Consider a function *merge(U,V)* where $U = u_0 u_1 \ldots u_{m-1}$ and $V = v_0 v_1 \ldots v_{n-1}$ are two string parameters. Suppose that *merge(U,V)* returns W defined as follows:



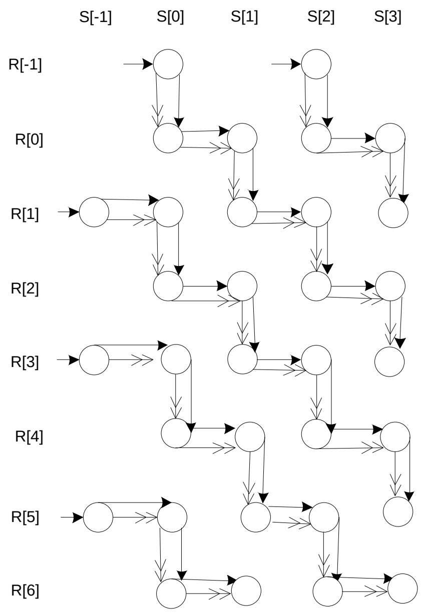
(a) $|S| = |R| = 1$.



(b) $|S| = |R| = 2$.



(c) $|S| = 8$ and $|R| = 5$.



(d) $|S| = 4$ and $|R| = 7$.

Figure 8.4: An illustration of LCAs catering for source and destination strings of varying lengths.

$$W = \begin{cases} u_0v_0u_1v_1\dots u_{n-1}v_{n-1}u_n & \text{if } m > n \\ u_0v_0u_1v_1\dots u_{m-1}v_{m-1} & \text{if } m \leq n \end{cases}$$

Let $U_{[i,j)}$ denote the substring $u_iu_{i+1}\dots u_{j-1}$ in string $U = u_0u_1\dots u_{m-1}$. To compute the LC between S and R, the following calls to *merge(U,V)* will be required:

- $merge(S_{[i,|S|)}, R)$ for $i = 0, 2, 4, \dots (|S| - 1)$ if $|S|$ is odd
- $merge(S_{[i,|S|)}, R)$ for $i = 0, 2, 4, \dots |S|$ if $|S|$ is even
- $merge(R_{[i,|R|)}, S)$ for $i = 1, 3, 5, \dots (|R| - 1)$ if $|R|$ is even
- $merge(R_{[i,|R|)}, S)$ for $i = 1, 3, 5, \dots |R|$ if $|R|$ is odd

Assume that a function, $makestrings(S, R)$, exists that ensures all the required calls to *merge* are made and that it returns all the resulting strings.

As an illustration, consider the source and destination strings $S =$ `acctacc` and $R =$ `acgtac` respectively. The resulting merged strings delivered by the function call $makestrings(S, R)$ are indicated in the table below. (Note that upper case characters originate from the source string whereas lower case characters originate from the destination string. This convention is used to clarify the working of the algorithm. The LCA is, however, decorated with lower case characters only.)

Function call	String Name	Merged String	Length
$merge(S_{[0,\|S\|)}, R)$	$S[0]$	`AaCcCgTtAaCcC`	13
$merge(S_{[2,\|S\|)}, R)$	$S[2]$	`CaTcAgCtCa`	10
$merge(S_{[4,\|S\|)}, R)$	$S[4]$	`AaCcCg`	6
$merge(S_{[6,\|S\|)}, R)$	$S[6]$	`Ca`	2
$merge(R_{[1,\|R\|)}, S)$	$R[1]$	`cAgCtCaTcA`	10
$merge(R_{[3,\|R\|)}, S)$	$R[3]$	`tAaCcC`	6
$merge(R_{[5,\|R\|)}, S)$	$R[5]$	`cA`	2

The second column of the table associates each string with an LC-CA$_{T3}$ name (i.e. $S[0] \cdots S[6]$; $R[1] \cdots R[5]$ respectively). This is because each string is used to determine the labels for conventional transitions (single-headed arrows) in its associated LC-CA$_{T3}$. That same string is also used to derive a string to run on its associated LC-CA$_{T3}$. In both these cases, however, the string in the table has to be slightly tweaked.

- To obtain the string with which to *decorate* conventional transitions of an LC-CA$_{T3}$, discard the last character of the LC-CA$_{T3}$'s string. Thus, for the current example, use `AaCcCgTtAaCc` to decorate the single headed arrows of $S[0]$, use `CaTcAgCtC` to decorate the single headed arrows of $S[2]$ … and use `c` to decorate $R[5]$.

- To obtain the string to be *run* on an LC-CA$_{T3}$, discard the *first* character of its associated string. Thus, for the current example, run `aCcCgTtAaCcC` on $S[0]$, run `aTcAgCtCa` on $S[2]$ … and run `A` on $R[5]$.

Let $S[i]_j$ denote the j^{th} state in LC-CA$_{T3}$ $S[i]$. Then $\delta(S[i]_j, \texttt{a}) = S[i]_{j+1}$, where `a` is the label of the conventional transition (denoted by the single-headed arrow). In this case, `c, g, t` are labels of the neutral transition (the double-headed arrow) from $S[i]_j$ to $S[i]_{j+1}$.

Figure 8.5 shows how the LCA template in Figure 8.3 is to be decorated for the strings in the current example. Consider, for example, the LC-CA$_{T3}$, $S[4]$ of Figure 8.5. Note that $\delta(S[4]_0, \texttt{a}) = S[4]_1$. Thus, if an `a` is read in state $S[4]_0$, then the single-headed arrow to state $S[4]_1$ is followed and $S[4]_1$'s counter is incremented. If a `c` is read in state $S[4]_0$, then the double-headed arrow is followed to $S[4]_1$ but $S[4]_1$'s counter is *not* incremented.

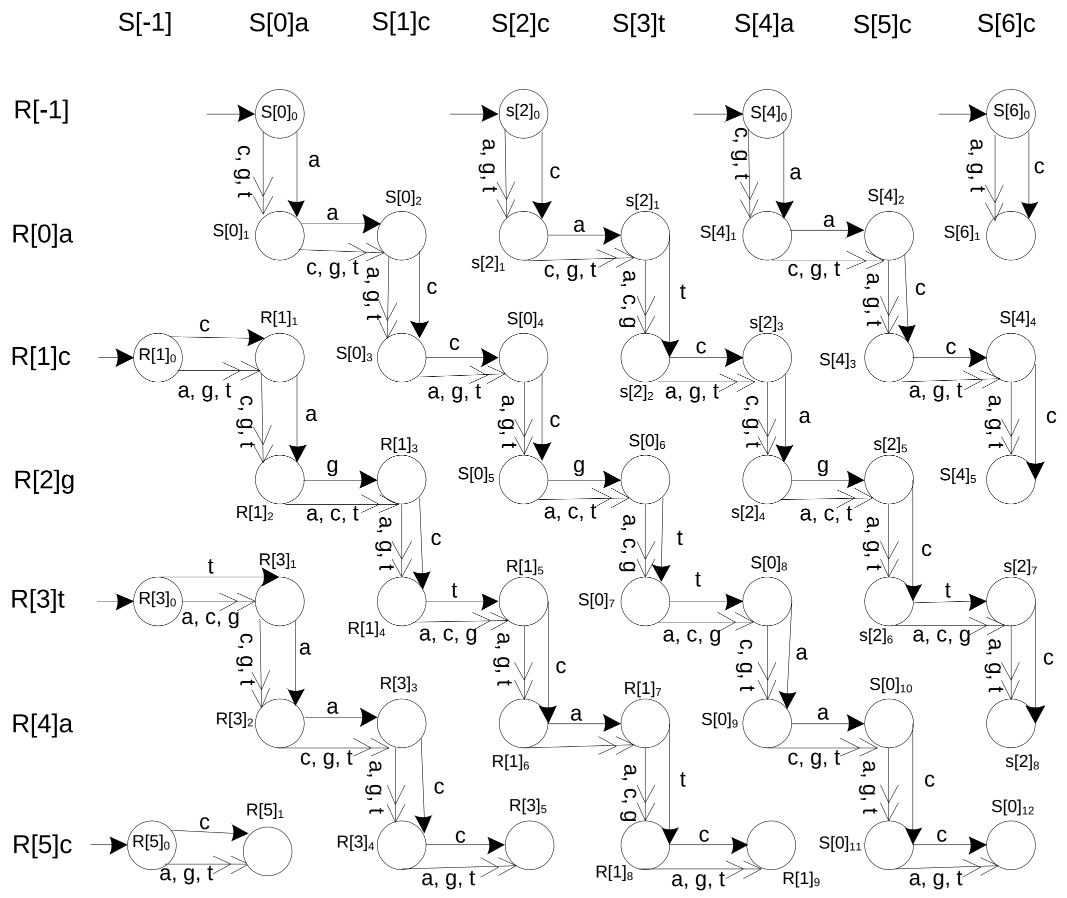


Figure 8.5: An LCA whose LC-CA_{T3}s are decorated for $S = \texttt{acctacc}$ and $R = \texttt{acgtac}$.

8.2.3 Step 3: Running strings on the LCA

This subsection assumes that the LCA in Figure 8.5 that has been decorated using the characters of the given S and R strings, as was described in the previous step. In this step, specific strings now have to be run[5] on each of the respective LC-CA$_{T3}$s of this LCA.

The table below shows these strings and the LC-CA$_{T3}$s on which they are to be run.

LC-CA$_{T3}$	String	Length
$S[0]$	`aCcCgTtAaCcC`	12
$S[2]$	`aTcAgCtCa`	9
$S[4]$	`aCcCg`	5
$S[6]$	`a`	1
$R[1]$	`AgCtCaTcA`	9
$R[3]$	`AaCcC`	5
$R[5]$	`A`	1

As an illustration, consider what happens when the string `aCcCgTtAaCcC` is run on $S[0]$. A conventional transition is made from state $S[0]_0$ to state $S[0]_1$ on character `a`, so the counter of $S[0]_1$ becomes 1; a neutral transition is made from state $S[0]_1$ to state $S[0]_2$ on character `c`, so the counter of $S[0]_2$ stays at 0; etc.

All the counter values resulting from running the strings in the table above on their indicated LC-CA$_{T3}$s can be derived in a similar way. These values are shown in the respective states of Figure 8.6.

Step 2 and *Step 3* have been formulated as two separate and sequential steps mainly for explanatory purposes. These steps could be combined when implementing them in code. The labelling of transition(s) at each state can be delayed until that state is visited. At that point, the label of a conventional downward transition can be inferred from the relevant column heading, and the label of a conventional horizontal transition can be inferred from the relevant row heading. Indeed, since each state's counter value is all that is required in subsequent steps, it is not strictly necessary to explicitly label the transitions. Implementation details such as this, as well as the data structure chosen to represent the LCA, are considered beyond the scope of the present discussion.

[5]To "run" a string (say $W = w_0 \ldots w_{\ell-1}$) on a given LC-CA$_{T3}$ (say L with states $L_0 \ldots L_{\ell-1}$) means to verify whether a conventional or neutral transition should be made at state L_j on character w_j for $j = 0 \ldots \ell - 1$, and to increment the counter of L_j accordingly.

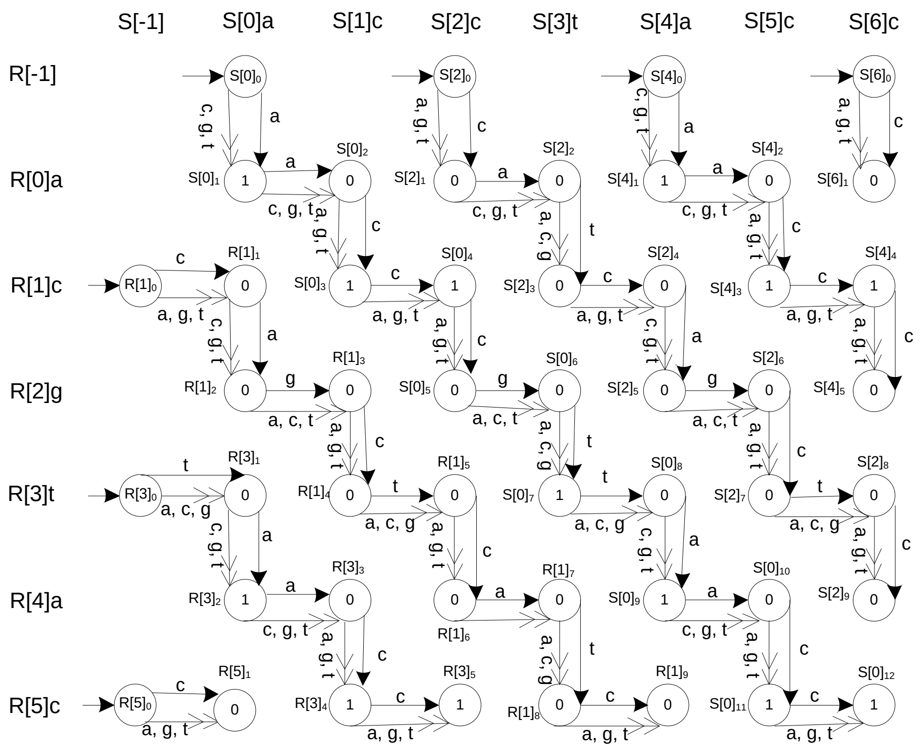


Figure 8.6: The LCA's counter values after the constructed input strings have been run.

8.2.4 Step 4: Extending the LCA

The state counter information contained in an LCA as depicted in Figure 8.6 can be used directly to derive the LC between the strings S and R that were used to build and label the LCA. Relying on the dynamic programming principles explained in Chapter 2, *sequential* computations can be carried out in either row- or column-based fashion. This is described in *Step 5*, as further discussed in Subsection 8.2.5. It is, in fact, how `FireSat`$_3$ is implemented to derive the empirical results described in the next chapter.

The explanation to follow shows how the LCA can be extended to support an FPGA implementation. Further details about the FPGA underlying algorithm are given in Section 8.2.6. Such an FPGA implementation would carry out the dynamic programming computations concurrently (in parallel)[6].

Note however, that the explanations in this subsection as well as in Subsection 8.2.6 are given for completeness and are not considered in the main line of research of this thesis.

The FPGA would carry out the concurrent operations on the so-called *mirrored diagonals* of the matrix. Figure 8.7 distinguishes between *diagonals* (the solid lines) and *mirrored diagonals* (the dashed lines). The processing required at all states on a mirrored diagonal is carried out in one clock cycle of the FPGA, starting at the top left and progressing one (mirrored diagonal) at a time towards the bottom right.

In order to facilitate such a traversal certain LC-CA$_{T3}$s in the LCA have to be extended by providing them with additional states. In addition, to simplify, stating the underlying algorithmic logic, the LC-CA$_{T3}$s are renamed and the labelling of their states and columns are changed accordingly.

The relationship between the matrix of counter values of states for the LCA derived up to this point (such as the one depicted in Figure 8.6) and the matrix of values, M, derived from the dynamic programming computations described in Chapter 2, Subsection 2.5.2, deserves some attention. There is an obvious one-to-one mapping between the state counter values and the cells of M. Exactly how the LC is computed in the discussion below—whether in the space defined by the matrix of states and their associated counter values or a separate matrix of values, M—is an implementation issue. For the present purposes, the focus will be on the matrix of states such as the one in Figure 8.6, and it will be convenient to call it the *SR* matrix.

To facilitate the explanation and provide a generalised formula that relies on the concurrent processing of CA$_{T3}$s in the LCA, the LCA are extended as follows:

[6]In fact, a variety of multi-processor hardware platforms could be used to support the concurrent implementation based on the proposed extension.

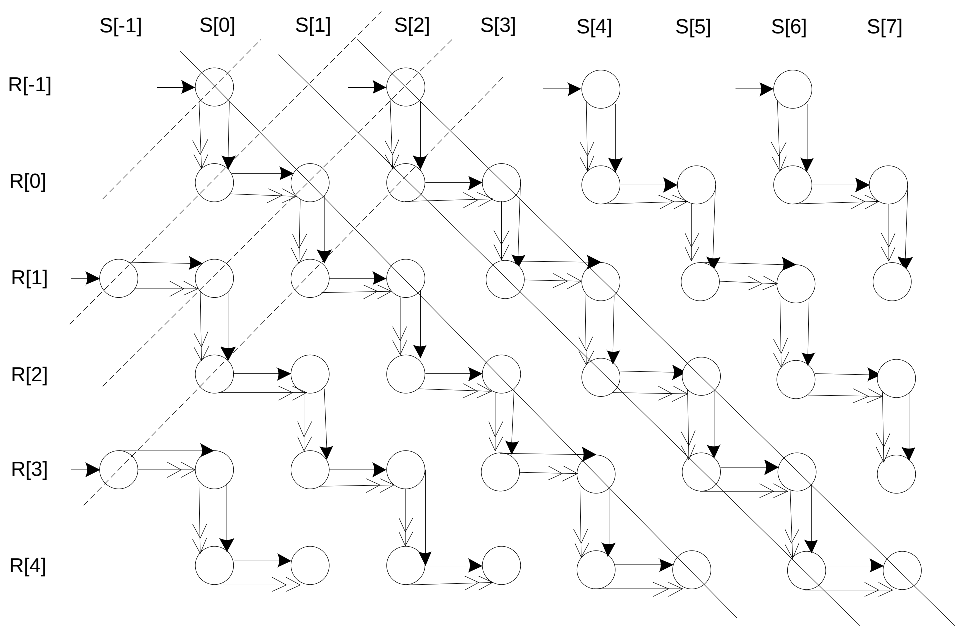


Figure 8.7: The difference between a *diagonal traversal* (solid lines) and a mirrored *diagonal traversal* (dashed lines) of an LCA.

1. The machines not starting at the top row should be extended to do so. Note that no input is run on the extended parts of the CA$_{T3}$s. Thus their transitions are not labelled. The counters of the states that comprise the extensions are set to 0.
2. Renumber the extended CA$_{T3}$s and unextended CA$_{T3}$ machines as follows:
   - Machine $R[1]$ becomes the extended machine $M[-1]$;
   - Machine $R[3]$ becomes the extended machine $M[-2]$;
   - Machine $R[5]$ becomes the extended machine $M[-3]$;
   - Machine $S[0]$ becomes the extended machine $M[0]$;
   - Machine $S[2]$ becomes the extended machine $M[1]$;
   - Machine $S[4]$ becomes the extended machine $M[2]$; and
   - Machine $S[6]$ becomes the extended machine $M[3]$.
3. Renumber the states according to the numbers of the new machines. The conventions explained in Section 8.2.2 are applied to number the states of the respective machines. Note that the counter values $d[i]_j$ are numbered in

correspondence to the states. Thus, for example, the counter of state $m[0]_0$ is $d[0]_0$. Figure 8.8 shows the values of state counters obtained in Section 8.2.3 in the respective states. Note that the counter values remain 0 for those states that are added when the CA$_{T3}$s are extended.

Recall that the *SR* matrix has the following features:

- It has $|S|+1$ columns labelled $S[-1], S[0], \ldots S[|S|-1]$.
- It has $|R|+1$ rows labelled $R[-1], R[0], \ldots R[|R|-1]$.
- It embeds the states of various LC-CA$_{T3}$s.
- Whenever there is a conventional transition between two states vertically below each other in column $S[i]$, say in rows $R[j]$ and $R[j+1]$, it is on the $(i+1)^{st}$ character of the source string S, denoted by $S[i]$. This transition is then accompanied by a neutral transition on all other characters.
- Dually, whenever there is a conventional transition between two states horizontally alongside each other in row $R[j]$, say in columns $S[i]$ and $S[i+1]$, it is on the $(j+1)^{st}$ character of the target string R, denoted by $R[j]$. This transition is then accompanied by a neutral transition on all other characters.
- For $i = 0, 2, \ldots 2\left\lfloor \frac{|S|-1}{2} \right\rfloor$, $S[i]$ names an LC-CA$_{T3}$ whose start state is in row $R[-1]$ and column $S[i]$ of *SR*.
- For $j = 1, 3, \ldots 2\left\lceil \frac{|R|-1}{2} \right\rceil - 1$, $R[j]$ names an LC-CA$_{T3}$ whose start state is in column $S[-1]$ and row $R[j]$ of *SR*.

The matrix of counter values of states resulting from transforming the *SR* matrix will be referred to as the *MR* matrix. Figure 8.8 is an example of an *MR* matrix resulting from extending the *SR* matrix of Figure 8.6. The following transformations on *SR* are required to produce *MR*.

1. Rename the LC-CA$_{T3}$ called $S[2i]$ in *SR* to $M[i]$ in *MR*, for $i = 0, 1, \ldots, \left\lfloor \frac{|S|-1}{2} \right\rfloor$.
2. Relabel column $S[-1]$ in *SR* as $R[0]$ in *MR*.
3. Expand *MR* by $|R|-1$ columns, inserted to the left of $R[0]$. Label these columns (from right to left) $R[1], R[2], \ldots R[|R|-1]$.

4. For $j = 0, 1, \ldots, \left\lceil \frac{|R|-1}{2} \right\rceil$], *rename* the LC-CA$_{T3}$s in *SR* called $R[2j+1]$ to $M[-j]$ in *MR* and then *extend* $M[-j]$ to the left. The next step below will describe how to carry out this extension to the left. The result will be that row $R[-1]$ and column $R[j]$ of *MR* will contain the start state of LC-CA$_{T3}$ $M[-j]$, for $j = 1, \ldots, \left\lceil \frac{|R|-1}{2} \right\rceil$.

5. Extending LC-CA$_{T3}$ $R[2j+1]$ in *SR* to the left to obtain LC-CA$_{T3}$ $M[-j]$ in *MR* entails the following:
   Replace $R[2j+1]$'s start state (in row $R[2j+1]$ column $S[-1]$ of *SR*) with *all* of the following new states and neutral transitions in *MR*:

   - In column $R[0]$ of *MR*, provide states in rows $R[j-1]$ and $R[j]$ and a neutral transition between them.
   - In column $R[1]$ of *MR*, provide states in rows $R[j-2]$ and $R[j-1]$ and a neutral transition between them.
     Provide a neutral transition from the new state in column $R[1]$, row $R[j-1]$ to the new state in column $R[0]$, row $R[j-1]$.
   - In column $R[2]$ of *MR*, provide states in rows $R[j-3]$ and $R[j-2]$ and a neutral transition between them.
     Provide a neutral transition from the new state in column $R[2]$, row $R[j-2]$ to the new state in column $R[1]$, row $R[j-2]$.
   - $\ldots$
   - In column $R[j]$ of *MR*, provide states in rows $R[-1]$ and $R[0]$ and a neutral transition between them.
     Provide a neutral transition from the new state in column $R[j]$, row $R[0]$ to the new state in column $R[j-1]$, row $R[0]$.

6. Rename all the states and their associated counters in *MR* using the conventions of Section 8.2.2. Thus the j^{th} state of LC-CA$_{T3}$ $M[i]$ is referenced as $m[i]_j$ and its counter is referenced as $d[i]_j$ respectively.

7. The values of state counters obtained for *SR* in Section 8.2.3 are retained in their corresponding states in *MR*. However, the value of the counters of all new states in *MR* are set to 0.

The resulting renamed LCA is presented in Figure 8.8. Note that the counter values have been populated in correspondence to decoration and running as explained earlier.

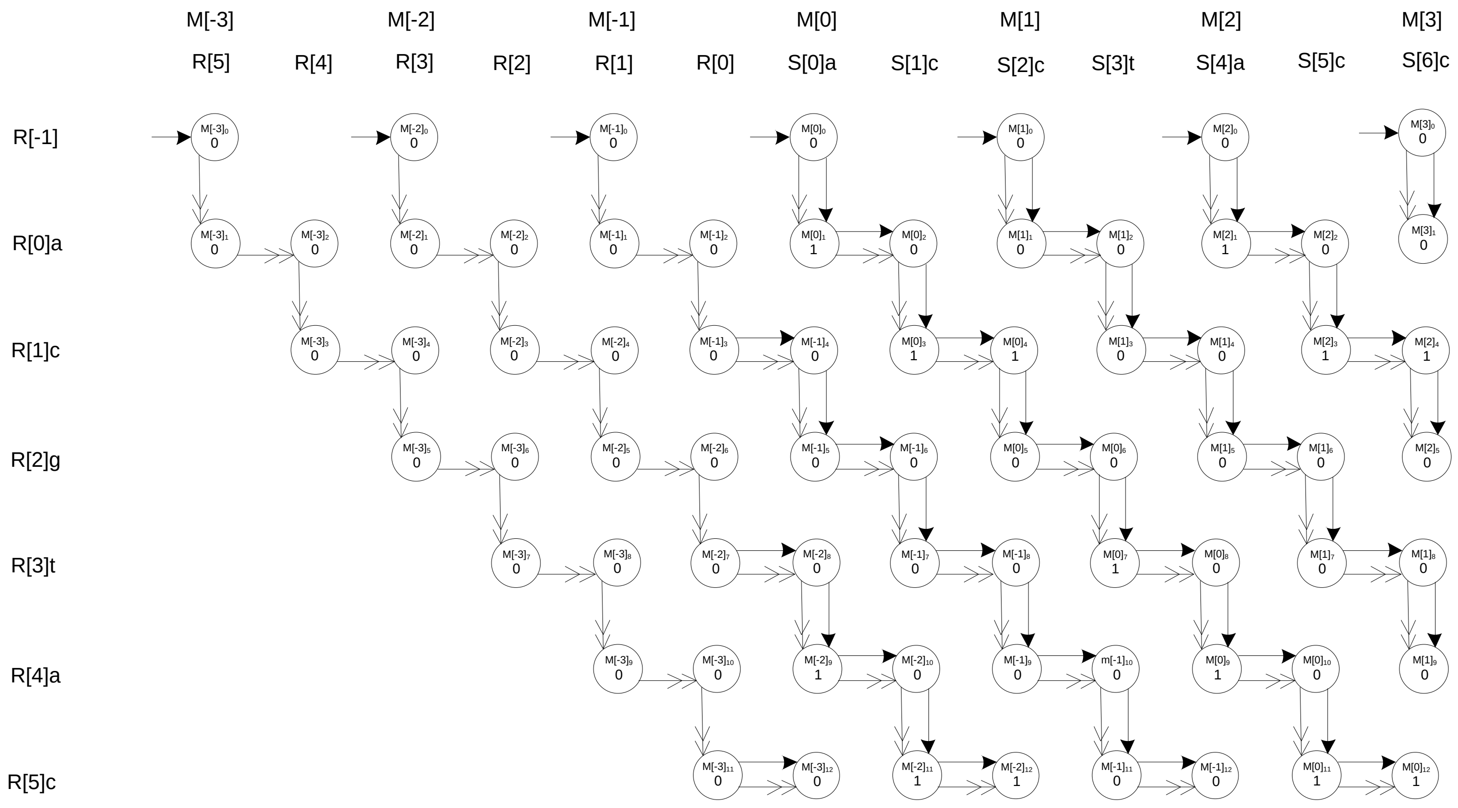


Figure 8.8: Counter values after the input strings have been run on the decorated LC-CA$_{T3}$s.

8.2.5 Step 5: Calculate the LC

This subsection describes how the LC is computed and used in the implemented version of `FireSat`$_3$. The next subsection indicates how it could be calculated and used in an FPGA version.

The LC calculation carried out by `FireSat`$_3$ relies directly on the dynamic programming computations explained in Section 2.5.3. R is the motif, ρ, currently under consideration. The source string, S, corresponds to the longest substring of the genetic string under investigation that could constitute the next potential TRE of a potential TR that is being sought. Thus $|S| = \lceil (1 + \frac{\varepsilon_{max\%}}{100}) \times |\rho| \rceil$.

An LCA similar to the one in Figure 8.6 is constructed as described in Step 8.2.1, decorated as described in Step 8.2.2 and initialised by running appropriate strings through it as described in Step 8.2.3. This delivers an SR matrix of initialised state counter values. A state counter value of 1 indicates a perfect match in the relevant underlying strings. These values may be directly referenced in carrying out the dynamic programming computations.

Note that the matrix used for the dynamic programming computations in Section 2.5.3 indicated the destination string as column elements and the source string as row elements. (See, for example, Figure 2.26.) Clearly, since the LC operation carried out on two strings is commutative, reversing this convention makes no difference. This is the case in the present description where columns represent source string elements, and rows represent destination string elements.[7]

Also note that whether the dynamic programming computations are recorded in the SR matrix itself, or written into a separate matrix, such as the matrix called M in Section 2.5.3, is entirely a matter of implementation. For convenience, the narrative below refers to the M matrix.

`FireSat`$_3$ determines the next TRE to be added to the currently found TR in four phases:

1. The M matrix is completed in a column-wise fashion. This means that when the computations in the column corresponding to $S[i]$ have been completed, then the entry in the last row of that column gives $LC(S_{[0,i+1)}, \rho)$.

   A so-called match pattern may be associated with each state in the SR matrix. It indicates the sequence of mutations that are assumed to have been incurred in arriving at that state. More will be said below about the match pattern in Section 8.3. At this point, note that the state containing $LC(S_{[0,i+1)}, \rho)$ will contain a match pattern associated with strings $S_{[0,i+1)}$ and ρ.

[7]This was done to facilitate the drawing of diagrams where the source string is longer than the destination string.

2. A prefix of S that delivers the maximum LC value is selected as the next possible TRE in the TR found to date. Note that, by the very definition of the LC metric, as i increases, the value of $LC(S_{[0,i+1)}, \rho)$ will be monotonically non-decreasing until it reaches a certain maximum level, after which its value will start to be monotonically non-increasing.

3. This prefix substring still needs further verification before being accepted as a TRE. The verification implies that its normalised LC value, LC_n, has to be higher than a `FireSat`$_3$ calculated match score threshold value. The next chapter will discuss in detail how threshold values were empirically determined for this study.

4. Finally, the substring also has to comply with the various other user-specified filter values that were mentioned in previous chapters, such as the maximum motif error percentage ($\varepsilon_{max\%}$), the maximum percentage of motif errors that may occur adjacently ($\kappa_{max\%}$), the maximum substring error allowed as weighted by error type ($\sigma_{max\%}$), etc.

Only once the substring has passed these tests is it appended to the TR found to date, and `FireSat`$_3$ then seeks the next TRE.

The fact that more than one prefix of S may have the same LC with respect to R gives rise to ambiguities about which prefix to use as the next TRE. These matters are discussed further in Section 8.3. Before doing so, attention is briefly turned to the logic needed to calculate the LC from an LCA in *extended* form.

8.2.6 Step 5′: Calculate the LC — Concurrent version

Subsection 8.2.4 described how to construct and initialise *MR* — the extended matrix of state counters. *MR* can be used to compute the LC between the source and destination strings, S and R, respectively. Recall that if $M[i]$ references the i^{th} LC-CA$_{T3}$ in *MR*, then $d[i]_j$ references the counter value in the j^{th} state of $M[i]$.

The following GCL pseudocode describes how these counter variables may be updated in a dynamic programming fashion to compute the LC between the source and destination strings:

```
if  $(d[i]_j = 1) \rightarrow d[i]_j := d[i]_{j-2} + 1$
[]  $(d[i]_j = 0 \wedge odd(j)) \rightarrow d[i]_j := \max(d[i]_{j-2}, d[i-1]_{j+1}, d[i]_{j-1})$
[]  $(d[i]_j = 0 \wedge even(j)) \rightarrow d[i]_j := \max(d[i]_{j-2}, d[i]_{j-1}, d[i+1]_{j-3})$
fi
```

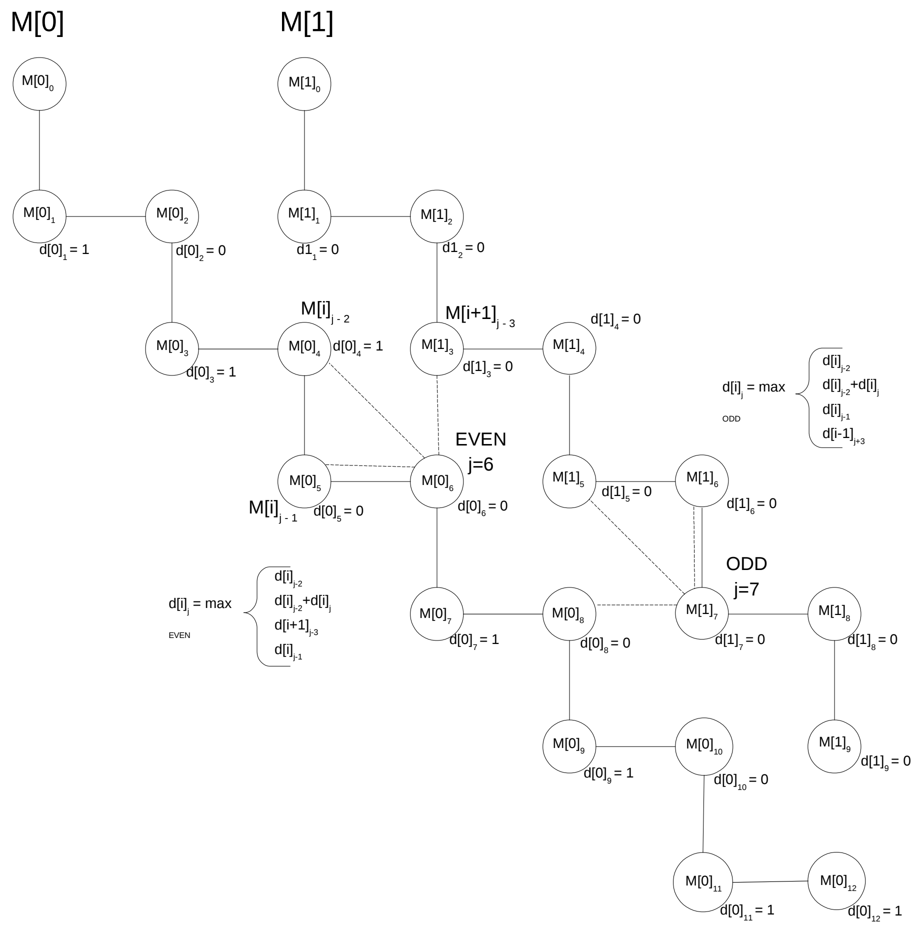


Figure 8.9: Machines $M[0]$ and $M[1]$ illustrating the calculation of the value of the even numbered state $m[0]_6$ and the odd numbered state $m[1]_7$ respectively.

Figure 8.9 graphically exemplifies the application of this pseudocode. In the example, the states whose counter values that might be needed to determine an updated value of $d[0]_6$ are connected to state $M[0]_6$ by dotted lines. Moreover, when updating $d[0]_6$, information is also available to infer whether or not the update involves a mutation error at that point and what type of motif error should be assumed.

Thus, if counter $d[0]_6 = 1$ then a perfect match has occurred and $d[0]_6$ is updated to $d[0]_4 + 1$.

If $d[0]_6 = 0$ then a mutation error is assumed to have occurred. Since 6 is even, $d[0]_6$ is updated by assigning to it the maximum of the $d[0]_5, d[0]_4, d[1]_3$. The type of mutation error that is assumed to have occurred can be inferred from the element that has the maximum value:

1. If $d[0]_5$ ($d[i]_{j-1}$) is selected as the maximum then an insertion is indicated as motif error type.
2. If $d[0]_4$ ($d[i]_{j-2}$) is selected as the maximum then a mismatch is indicated as motif error type.
3. If $d[1]_3$ ($d[i+1]_{j-3}$) is selected as the maximum, then a deletion is indicated as motif error type.

To calculate the value for $d[1]_7$ a similar process is followed. If $d[1]_7 = 1$ then a perfect match has occurred and $d[1]_7$ is updated to $d[1]_5 + 1$. If $d[1]_7 = 0$ then a mutation error is assumed to have occurred. Since 7 is odd, $d[1]_7$ is updated by assigning to it the maximum of the $d[1]_6, d[1]_5, d[0]_8$. Again, the type of mutation error that is assumed to have occurred can be inferred from the element that has the maximum value:

1. If $d[1]_6$ ($d[i]_{j-1}$) is selected as the maximum then a deletion is indicated as motif error type.
2. If $d[1]_5$ ($d[i]_{j-2}$) is selected as the maximum then a mismatch is indicated as motif error type.
3. If $d[0]_8$ ($d[i-1]_{j+3}$) is selected as the maximum, then an insertion is indicated as motif error type.

The dotted lines in Figure 8.10 indicate mirrored diagonals of the *MR*. Note that updates to state counters on a given mirrored diagonal are independent of one another. Their values depend only on the updated values of state counters on the mirrored diagonal to the left. Updates on a given mirrored diagonal can therefore be carried out in parallel.

In principle, therefore, the *MR* as derived from a given S and R could be mapped onto an FPGA that executes the given pseudocode in a step-wise fashion, sweeping through mirrored diagonals from left to right. Each step involves the concurrent updating of *all* state counters on a mirrored diagonal.

Note that, as in Step 5 above, the state counter values in the final row give the LCs of the prefixes of S (as determined by the relevant column) with respect to R.

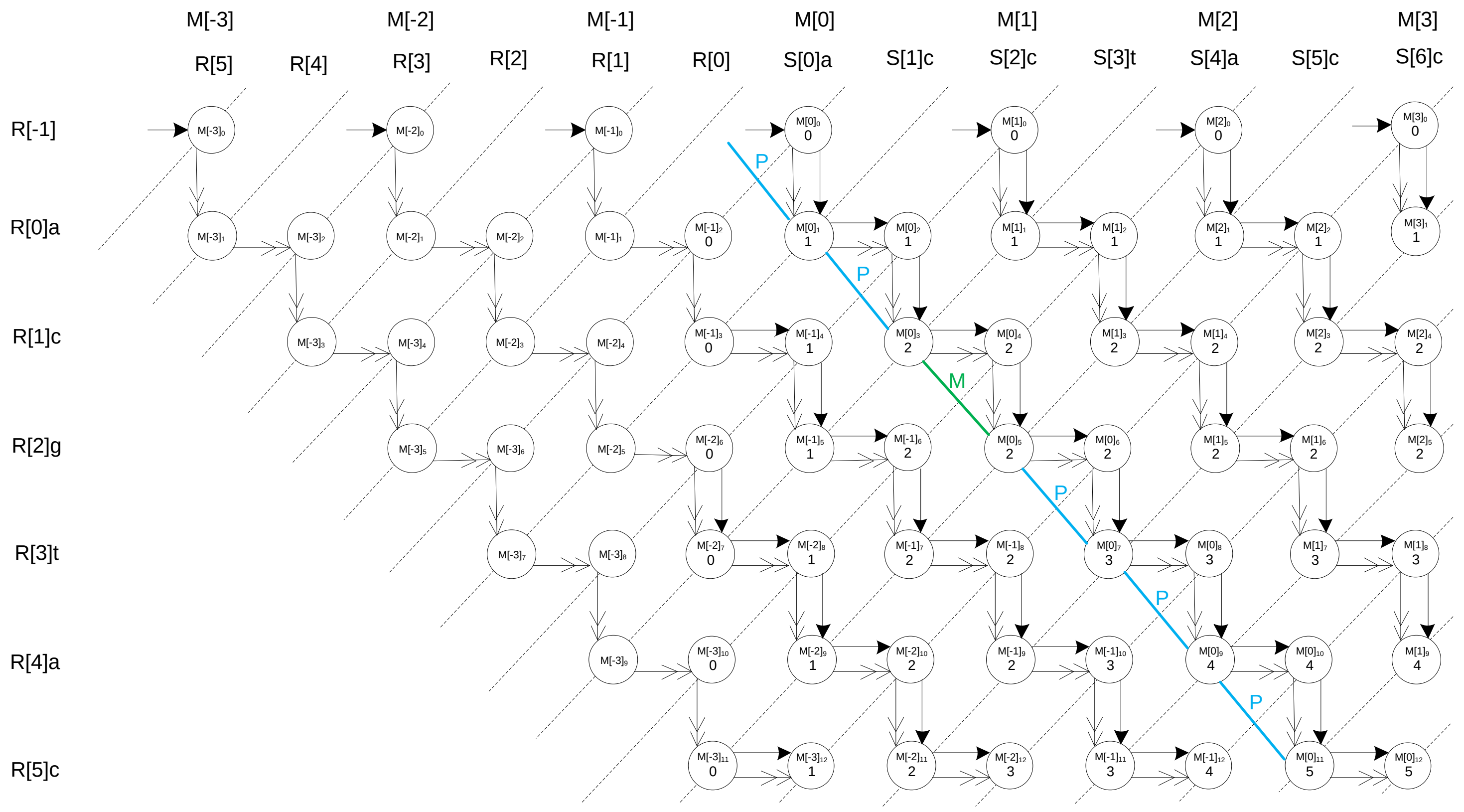


Figure 8.10: The resultant LCA CA_{T3} displaying the LC in the right corner.

8.3 Match Patterns and Ambiguities

When comparing source string S and destination string R, a *match pattern* is used to indicate the positions and types of mutations that are deemed to have occurred. The match pattern is a string of length $n = max(|S|, |R|)$ whose i^{th} element is from the alphabet {P,M,D,I}, indicating whether a perfect match, mismatch, deletion or insertion is deemed to have occurred in the i^{th} position, for $i = 1, \dots n$.

`FireSat`$_3$ ensures that when $LC(S, R)$ is being computed, an associated match pattern is also available at each state in the LCA. The principles for deriving a match pattern from an ordinary (unextended) LCA are the same as for the extended LCA case. How the elements of a match pattern can be inferred while traversing an extended LCA has already been covered in Subsection 8.2.6 above.

It was seen in that subsection that, under certain circumstances, $d[i]_j$ is to be updated by selecting the maximum of $d[i]_{j-2}$, $d[i-1]_{j+1}$ or $d[i]_{j-1}$ (or by selecting the maximum of $d[i]_{j-2}$, $d[i]_{j-1}$ or $d[i+1]_{j-3}$ under other circumstances). It was also seen that the element to be selected as maximum, (whether, say, $d[i]_{j-2}$, $d[i-1]_{j+1}$ or $d[i]_{j-1}$) indicates that a mismatch, deletion or insertion is deemed to have occurred.

Using this information to record the match pattern that evolves at each state while traversing the extended LCA is a straightforward implementation issue. As an example, the coloured lines in Figure 8.10 show that the match pattern between `acgtac` and `acctac` evolves as "PMPPP". (A blue line is used to indicate a perfect match ("P") and a green line used to indicate a mismatch. ("M").)

Note that when determining a match pattern, a pre-specified mutation priority ordering should be used to adjudicate between choices to be made when ties occur in the maximum. `FireSat`$_3$ relies on the same default priority ordering as that of `FireSat`$_2$ when such ambiguities need to be resolved. In terms of this priority ordering, mismatches rank above deletions and deletions above insertions. This ordering is denoted by MDI. The ordering where insertions rank above deletions is denoted by MID. `FireSat`$_3$ allows the user to specify a MID ordering instead of MDI if desired.

One of the uses of this match pattern is to determine the total number of insertions deemed to have been incurred when computing $LC(S, R)$. This count is needed in order to compute the normalised LC value, LC_n, as will be discussed in the next chapter.

The match pattern is also needed to compute the weighted substring error. Recall that the weighted substring error, $\sigma\%$, is the weighted sum of mutations in a candidate TRE, expressed as a percentage of the motif length. The candidate

TRE is filtered out if its weighted substring error exceeds a specified maximum, $\sigma_{max\%}$.

In deciding on a substring to serve as the next possible TRE in the TR found to date, ambiguity issues similar to those of `FireSat`$_2$ may arise. Again, this is true, whether or not the LC computation is based on Step 5 or Step 5′ described above. In both cases, more than one substring may yield the same LC with respect to the motif. In such situations, there will be ambiguity as to which substring to select as the next TRE.

Figure 8.10 shows an example of this source of ambiguity. Notice that $\mathrm{M}[0]_{12}$ has value 5, meaning that the LC between `acgtac` and `acctacc` is 5. But the LC between `acgtac` and `acctac`, given in $\mathrm{M}[0]_{11}$, *also* has value 5. In `FireSat`$_3$, where such ties arise, the *shortest* source string with the maximal LC value is selected as the next candidate TRE.

8.4 Conclusion

The foregoing thus indicates how, given R and S where $R = \rho$ and $|S| = \lceil(1 + \frac{\varepsilon_{max\%}}{100}) \times |\rho|\rceil$, a prefix of S might be determined that has the best LC value over all other prefixes. In the context of TR detection, such a prefix may or may not be a suitable candidate as a TRE to be added on to the TR detected to date.

The next chapter will show how a threshold function for the LC_n was found from empirical synthetic data. The threshold function together with a threshold factor were used to decide whether or not the prefix substring should be considered as TRE.

This chapter also provides the broad outline of a version of `FireSat`$_3$ that could potentially be implemented on either an FPGA or a GPU by developing the LCA in an extended format. Such an implementation could be designed to execute computations in parallel along the mirrored diagonal of the extended LCA. Further exploration of such FPGA/GPU implementations is beyond the scope of this thesis.

Chapter References

J. Becker, M. Platzner, and S. Vernalde. Large Scale Protein Sequence Alignment using FPGA Reprogrammable Logic Devices. *Field Programmable Logic and Application*, 3203:23—32, 2004.

M. Guerra. The Principles of FPGAs. http://www.electronicdesign.com/fpgas/principles-fpgas, 2016.

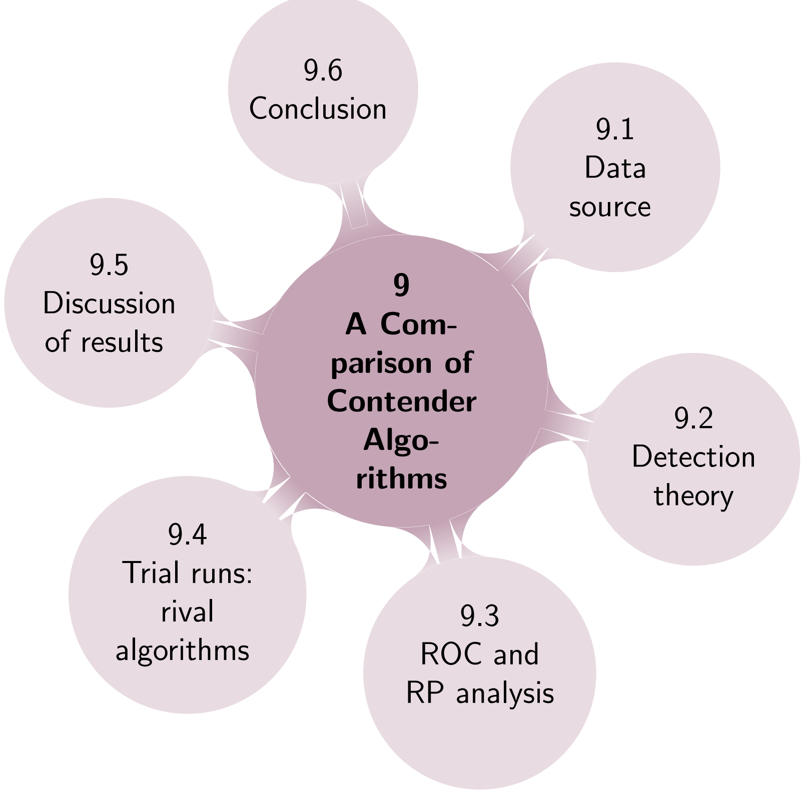
X. Guo, H. Wang, and V. Devabhaktuni. A Systolic Array-Based FPGA Parallel Architecture for the BLAST Algorithm. *ISRN Bioinformatics*, 2012:11, 2012. doi: 10.5402/2012/195658.

D. Hoang and D. Lopresti. FPGA implementation of systolic sequence alignment. *Field-Programmable Gate Arrays: Architecture and Tools for Rapid Prototyping*, pages 183–191, 1993.

K. T. Johnson, A. R. Hurson, and B. Shirazi. General-purpose systolic arrays. *Computer*, 26(11):20–31, November 1993. ISSN 0018-9162. doi: 10.1109/2.241423. URL http://dx.doi.org/10.1109/2.241423.

D. Koch, F. Hannig, and D. Ziener. *FPGAs for Software Programmers.* Springer Verlag, 1 edition, 2016.

S. Trimberger. *Field-Programmable Gate Array Technology.* Springer Science and Business, 1 edition, 1994.



9 A Comparison of Contender Algorithms

"The wonder of it all is that we are not in control. Control. Intention. All are but illusions. Who decides? Memory and Inspiration, one a thorn, the other a rose. Either leads the Soul by the proverbial nose. Really." ... Ihaleakela Hew Len

---

The previous chapters explained the theoretical underpinnings of the various versions of `FireSat`. Algorithm 6.3.1 gave an outline of the flow of logic with specific reference to $\texttt{FireSat}_1$, but the basic structure applies to other versions of `FireSat` as well.

Various versions of `FireSat` were implemented to examine their accuracy in detecting TRs. This has been done in `Matlab`. Different versions of `FireSat` were used during trial runs. Although $\texttt{FireSat}_1$ is not as accurate as $\texttt{FireSat}_2$ or $\texttt{FireSat}_3$ it has been used for some trial runs to illustrate *the extent to which* $\texttt{FireSat}_2$ and $\texttt{FireSat}_3$ improved accuracy.

TR detection algorithms may be viewed as *binary detectors*: i.e. each algorithm outputs indices that, in effect, partition the input file into substrings of nucleotides that are deemed to constitute TRs (subject to given threshold specifications) and substrings that are deemed *not* to constitute TRs. Thus every nucleotide in the input is classified as either part of a TR or not. A body of theory for analysing the outputs of such binary detectors has been developed over the years. This includes so-called *receiver operating characteristic* (ROC) analysis, and notions such as the *precision* and *recall* of a detector. This chapter will briefly review the theory. The review includes an explanation of a so-called *recall-precision* (RP) curve, that is used on 20 different data sets, to establish an effective *match score threshold function*[1] for `FireSat`.

The chapter is organised as follows: Section 9.1 explains how synthetic data was generated to be used as input for trial runs with the various algorithms. Then Section 9.2 introduces statistical terminology from detection theory. ROC curves and RP curves are discussed in Section 9.3. As part of this section, RP curves are derived from the synthetic data and used to estimate an effective *match score threshold function* to be used when running the various `FireSat` algorithms.

Section 9.4 reports on trial runs in respect of `FireSat`. The section also reports on the binary detection performance on the same data of `Tandem Repeats Finder (TRF)`, `mreps` and `Phobos` — i.e. on the performance of algorithms identified in Chapter 3 as being the most likely rivals to `FireSat`. The recall, precision and related performance measures of various algorithms are graphically compared in Section 9.5. Section 9.6 concludes this chapter.

9.1 Data source

Synthetic data was required so as to have a basis for determining a good *match score threshold function* for various versions of `FireSat`. The synthetic data was used to examine the accuracy of `FireSat` and to compare it against other rival algorithms. Trial runs of these algorithms were carried out on this synthetic data. The details of how the synthetic data was generated is discussed next.

`Matlab` was used to generate twenty sets of data in FASTA format to be used during trial runs[2].

Data generation is aimed at embedding known TRs at known positions in each data set. Each data set is effectively an input string, referred to here as *gSeq*. To generate *gSeq*, firstly generate a random sequence of nucleotides of length 11 000 to serve temporarily as *gSeq*. An additional random nucleotide sequence of length 200, called the *supermotif*, is also generated. The purpose of the supermotif

[1]The threshold related concepts established by the method described here can only loosely be referred to as "optimal" since optimality is not guaranteed but approximated. For this reason the threshold related concepts will be described as "effective" rather than as "optimal".

[2]The synthetic data used for the trial runs as well as a `Matlab` program generating random synthetic data can be found at `http://www.dna-algo.co.za`.

is to serve as a template from which motifs are to be derived. Ideally, both *gSeq* and the supermotif should be TR-free—i.e. it should not contain any TR such that $10 \leq |\rho| \leq 200$ and $2\% \leq \varepsilon_{max\%} \leq 20\%$. Of course, this condition cannot be guaranteed to hold, *a priori*. Furthermore, if it does not hold, it might potentially (but not necessarily) give rise to small errors in the precision or recall measurements subsequently made. Such errors are, however, not critical as their resulting inaccuracies in the measurements would apply equally to all algorithms. Nevertheless, in an effort to attain the ideal, *gSeq* as well as the supermotif were inspected and any TRs that were found were removed.

It will be convenient to use ρ throughout to refer to a PTRE. Prefixes of the supermotif of varying lengths are used as ρ for different data sets. For example, if a TR with $|\rho| = 25$ is to be generated, then ρ is taken as the first 25 nucleotides of the supermotif. On this basis, data sets were generated for which $|\rho| = 10, 25, 50, 100$ and 200, respectively.

Each data set contains exactly 10 TRs. All the TREs in a given data set are derived from the same ρ in the manner described below. In each data set, these 10 TRs replace nucleotide entries already in *gSeq* at the exact same pre specified index positions, namely 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000 and 10 000 respectively. As shown in Table 9.1, in general, the TR at index 1000 consists of ρ followed by 1 TRE, that at index 2000 consists of ρ followed by 2 TREs, and so on, so that the TR at index 10 000 consists of ρ followed by 10 TREs.

TR number	Start index	Structure
1.	1000	ρ followed by 1 TRE
2.	2000	ρ followed by 2 TREs
3.	3000	ρ followed by 3 TREs
4.	4000	ρ followed by 4 TREs
5.	5000	ρ followed by 5 TREs
6.	6000	ρ followed by 6 TREs
7.	7000	ρ followed by 7 TREs
8.	8000	ρ followed by 8 TREs
9.	9000	ρ followed by 9 TREs
10.	10000	ρ followed by 10 TREs

Table 9.1: Structure of a synthetic data file.

Each of the *five* different values of ρ is used to generate *four* different data sets. These four different data sets differ with respect to a parameter denoted by P_{err}. The parameter represents the maximum percentage mutation error allowed on a TRE. The values used for P_{err} are 2%, 5%, 10% and 20%. Thus, in total $5 \times 4 = 20$ different versions of the *gSeq* data set are generated.

Note that P_{err} should be distinguished from $\varepsilon_{max\%}$. Recall that $\varepsilon_{max\%}$ is a `FireSat` threshold value that indicates the maximum percentage of motif errors, $\varepsilon_{max\%}$, allowed per TRE. In contrast to $\varepsilon_{max\%}$, P_{err} is a percentage value used when synthetic data is generated. This is clarified in the next bulleted list.

Further note that data sets based on $|\rho| = 200$ are exceptions to the scheme illustrated in Table 9.1. In these data sets, TRs do not occur from position 6000 onwards. Consequently, the four data sets for $|\rho| = 200$ contain 5, not 10, TRs.

Generating a random TRE from a given ρ that complies with a given value for P_{err} proceeds according to the following scheme:

- Each nucleotide in ρ is considered in turn as a candidate for a possible mutation until one of two conditions holds: either all nucleotides in ρ have been considered, or the number of mutations to date on ρ has reached the threshold value of $\lceil |\rho| \times P_{err} \rceil$.
- With probability P_{err}, a nucleotide under consideration is mutated.
- If a nucleotide is mutated, then one of the three *types* of mutations is randomly selected with equal probability. Recall that the three mutation possibilities are deletion, insertion, or mismatch.
- If an insertion is to be generated, a random nucleotide for insertion is selected with equal probability from the four-character alphabet.
- If a mismatch is to be generated, a random nucleotide, other than the current nucleotide under consideration, is selected with equal probability from the remaining three characters of the nucleotide alphabet.
- If a deletion is to be generated the current nucleotide is deleted.

9.2 Detection theory

As previously pointed out, detecting TRs in DNA is a binary classification problem. In TR detection, the data set is a sequence of nucleotides. Each nucleotide in the data set either forms part of a TR or not. Whether a given nucleotide truly forms part of a TR can, in principle, be determined by some objective criterion. If the nucleotide is truly part of a TR, we say that the nucleotide is a *real* exemplar (or positive exemplar). If this is not the case, then we say that the nucleotide is a *real negative* exemplar (or simply a negative exemplar). Detection theory in relation to TR detection is concerned with the extent to which TR detection algorithms correctly classify real exemplars and negative exemplars in a given data set.

Statistical terms relevant to TR detection in the context of this thesis are explained next. These terms correspond to the definitions presented by Zou et al. [2007], Fawket [2003] and Zhu et al. [2010]. The terms refer to nucleotides in a genetic string that either do or do not occur in a TR, where the tolerances regarding what is to constitute a TR is assumed to have been pre-specified by some set of criteria.

1. P (Condition positives)
   P is the number of real exemplars (positive objects) in the data set.

2. N (Condition negatives)
   N is the number of negative exemplars in the data set. In practice, N is relatively high — 90% or more of the nucleotides in a typical DNA sequence will be negative exemplars.

3. TP (True positives)
   TP refers to the number of real exemplars *identified as such* by the TR detector — i.e. to the number of so-called *hits*.

4. TN (True negatives)
   TN refers to the number of real negatives *identified as such* by the TR detector — i.e. the number of nucleotides to be reckoned as negative exemplars because they were *not reported* as real exemplars by the TR detector.

5. FP (False positives)
   FP refers to the number of real negatives, incorrectly identified as real exemplars by the TR detector. Such an error is also referred to as a false alarm or a *Type 1* error.

6. FN (False negatives)
   FN refers to the number of real exemplars that are *not detected* as such by the TR detector. The nucleotides are therefore considered as negative exemplars when, in reality, they are not. Such an error is also called a miss or a *Type 2* error.

7. PP (Positive predictions)
   PP = FP + TP — i.e. all the nucleotides in the data set that the TR detector deems to be real exemplars.

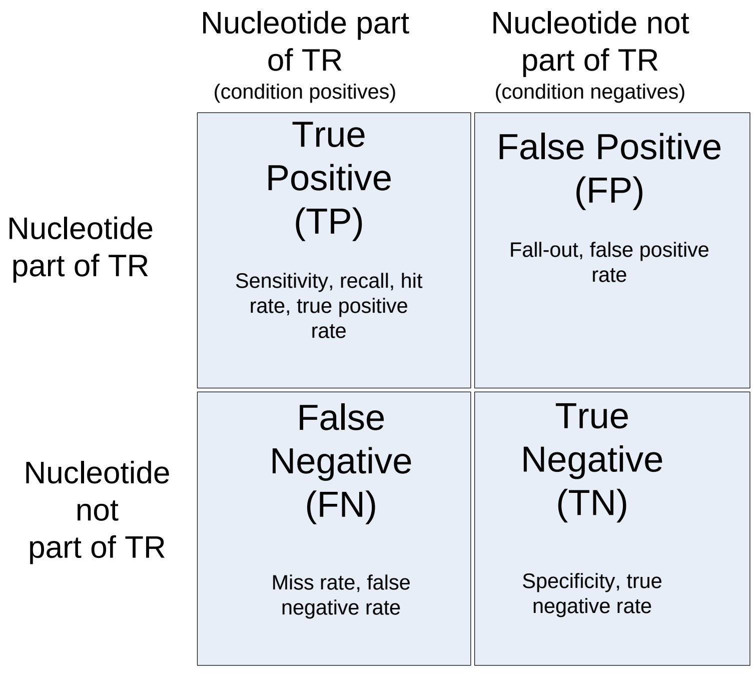

Nucleotide part of TR
(condition positives)
Nucleotide not part of TR
(condition negatives)
Nucleotide part of TR
Nucleotide not part of TR
True Positive (TP)
Sensitivity, recall, hit rate, true positive rate
False Positive (FP)
Fall-out, false positive rate
False Negative (FN)
Miss rate, false negative rate
True Negative (TN)
Specificity, true negative rate

Figure 9.1: A TR-detector confusion matrix.

A so-called confusion matrix (see Figure 9.1) further illuminates the above terminology. The confusion matrix shows that, for nucleotides reported as being part of TRs by a TR detector, two possibilities exist. Such a reported nucleotide can either be a *true* detection. Then it is a *true positive.* Alternatively the detection can be an incorrect report. A nucleotide that is falsely reported as a true exemplar is referred to as a *false positive.* Similarly, there are two possibilities relating to nucleotides that are not reported as part of a TR. Firstly these nucleotides may constitute real exemplars that have been missed. These are referred to as *false negatives.* Alternatively the nucleotides not reported as real exemplars may indeed be negative exemplars. These are referred to as *true negatives* — in other words, the TR detector has correctly rejected these as exemplars.

In the confusion matrix of Figure 9.1, the *rates* associated with the occurrences of instances of *true positives*, *false positives*, *false negatives* and *true negatives* are included in a *smaller* font. These rates are defined in the next enumerated list.

1. True Positive Rate (TPR)
   TPR is the proportion of real exemplars in the data that are TR detector hits. TPR is therefore calculated as follows:

   $$\text{TPR} = \frac{\text{TP}}{\text{TP+FN}} = \frac{\text{TP}}{\text{P}}$$

   Other words that are interchangeably used in the literature to refer to TPR are *recall; sensitivity* and *probability of detection.*

2. False Negative Rate (FNR)
   FNR is defined as the proportion of real exemplars in the data that escape detection by the TR detector. FNR is calculated as follows:

   $$\text{FNR} = \frac{\text{FN}}{\text{TP+FN}} = \frac{\text{FN}}{\text{P}} = 1 - \text{TPR}$$

   FNR is also referred to as *miss ratio* in the literature.

3. Positive Predictive Value (PPV)
   PPV, also referred to as *precision*, is the proportion of nucleotides reported as real exemplars by the TR detector that are indeed hits. PPV is calculated as follows:

   $$\text{PPV} = \frac{\text{TP}}{\text{TP+FP}}$$

4. False Discovery Rate (FDR)
   FDR is the proportion of nucleotides reported as real exemplars by the TR detector that are in fact negative exemplars.

   $$\text{FDR} = \frac{\text{FP}}{\text{TP+FP}} = 1 - \text{PPV}$$

   In this thesis the term *imprecision* is used to refer to FDR.

5. True Negative Rate (TNR)
   TNR is the proportion of all real negative exemplars in the data that are deemed to be real negative exemplars by the TR detector. It is calculated as:

   $$\text{TNR} = \frac{\text{TN}}{\text{FP+TN}} = \frac{\text{TN}}{\text{N}}$$

Note that the denominator *FP+TN* constitutes the total of real negative nucleotides. In the literature the word *specificity* is often used to refer to TNR. TNR can also be referred to as *correct rejection rate.*

6. False Positive Rate (FPR)
   FPR is calculated as follows:

$$\text{FPR} = \frac{\text{FP}}{\text{FP+TN}} = \frac{\text{FP}}{\text{N}} = 1 - \text{TNR}$$

   FPR is also referred to as *probability of false alarm*, *false alarm rate* or *fall-out.* This implies that FPR is the proportion of real negative nucleotides in the data that are falsely reported as real exemplars by the TR detector.

This thesis focusses on the recall (TPR) and precision (PPV) of various TR detectors. The miss ratio (FNR) and imprecision (FDR) are merely the respective complimentary values of the former two rates. The last two rates mentioned above (TNR and FPR) are provided for completion. They are of historical significance but, as will be pointed out, they are not suitable for the analysis of TR detector performance.

The harmonic mean is considered a good way of summarising the values of several rates, say $x_1 \dots x_n$, obtained from experiments. It is defined as the reciprocal of the mean of reciprocals of a set of rates, i.e.

$$\left(\frac{1}{n}\left(\frac{1}{x_1} + \frac{1}{x_2} + \cdots \frac{1}{x_n}\right)\right)^{-1} = \frac{n(x_1 x_2 \cdots x_n)}{x_1 + x_2 \cdots + x_n}$$

The *F-score* or *F-measure* is a harmonic mean that is commonly used in binary classification systems. It is defined as the harmonic mean of precision and recall. It can be used to measure how accurately a binary classification algorithm has performed (Van Rijsbergen [1979]). The F score is denoted by F_1 score:

$$F_1 \text{ score} = \frac{2(\text{precision.recall})}{\text{precision} + \text{recall}} \tag{9.1}$$

or, using the above notation:

$$F_1 \text{ score} = \frac{2(\text{PPV.TPR})}{\text{PPV} + \text{TPR}} \tag{9.2}$$

When a *match score threshold* function is proposed in Section 9.3.2, reference is made to the *standard deviation* (σ). The definition of σ is as follows:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{9.3}$$

where N is the number of exemplars in the data; x_i is the different exemplar values and μ is the mean — the simple average of the objects.

9.3 ROC and RP analysis

Receiver operating characteristic (ROC) analysis was developed during World War 2 in the context of radar detection. The objective was to analyse the accuracy of a classifying system that is intended to differentiate a signal from noise. Generally, such systems classify with respect to some pre specified *threshold* value. If the threshold value is too low, practically everything will be classified as a signal and nothing as noise, and *vice-versa* when the threshold is too high. ROC analysis is concerned with trying to discover an appropriate threshold such that the number of false positives and false negatives remain within acceptable bounds.

To this end, an ROC *curve*, drawn in so-called ROC *space*, characterises the classifier's performance as the threshold varies. The x-axis of the ROC space represents *1 - specificity* (1 - TNR) — i.e. the number of negative exemplars present in the data *wrongly* identified as positive exemplars by the classifier under study, expressed as a proportion of the total number of negative identifiers. The y-axis represents *recall* (or *sensitivity*, TPR) — i.e. the positive exemplars identified by the classifier under study as a proportion of the total number of positive exemplars present in the data.

When a classifier is run on a given set of data using a given threshold value, the outcome has a certain specificity and recall. This pair of values provides a single point in the ROC space. The ROC curve for a given classifier and a given set of data, is the set of points traced in ROC space for varying threshold values.

In subsequent years, ROC analysis was applied to several areas beyond the domain of radar detection. These areas of application include clinical classifiers that are dependant on screening, diagnostic tests, laboratory testing, epidemiology, radiology and bioinformatics (Zou et al. [2007]).

Another metric pair, often used in the same manner as ROC, is *Recall-Precision* (RP)[3]. The objective of both of these metric pairs (ROC and RP) is to determine

[3]Note that reference is made throughout to Recall-Precision even though Recall-*Imprecision* is actually used, where Imprecision simply refers to (*1 - Precision*).

a threshold value that is likely to increase the number of true positives while minimizing the number of false positives.

Ke and Sukthankar [2004] point to the subtle differences between ROC and RP analysis that dictate the use of one rather than the other in specific scenarios. ROCs are well-suited for classifier evaluation when the total negative population (N) is comparable to the total positive population (P). This is not the case in TR detection, where N is typically much larger than P. Consequently the *specificity* in TR detection is usually close to 0. As a result, the value of (1-specificity) is close to 1, and the consequent ROC curve for a TR-detector will typically not provide clear information about threshold suitability.

Thus, based on the insights of both Ke and Sukthankar [2004] and Mikolajczyk and Schmid [2003], an RP analysis rather than an ROC analysis was undertaken, as described below. Subsection 9.3.1 explains the notion of RP space and an RP curve drawn in RP space to characterise the behaviour of a TR detector with respect to some data set. Subsection 9.3.2 then describes the specific RP analysis carried out on the synthesised data sets using $\texttt{FireSat}_3$ as the TR detector.

9.3.1 RP curves

This subsection explains RP curves in reference to some abstract TR detector working on some abstract dataset that contains known TRs. In order to compute the recall and precision, the following convention is used:

- Each nucleotide in the data set that forms part of a known TR is deemed to be a *real exemplar*.
- Each of the remaining nucleotides is deemed to be a *negative exemplar*.

A run of the TR detector takes as input the data set and threshold value. Such a run then provides as output an estimate of which nucleotides in the data set are exemplars and which are not. In other words, each TR detector run delivers a certain number of *true positives*, and of *false positives*.

Using the x-axis to represent imprecision (i.e. (1 - precision)) and the y-axis to represent recall, all possible combinations of recall and imprecision constitute the RP-space. Each run of a TR detector delivers a certain value for precision and a certain value for recall, and thus a certain point in RP-space. Figure 9.2 shows several points of potential interest in RP space.

The ideal point in RP space is (0,1) — the upper left corner of the RP-space. This coordinate would indicate that the detector in question had achieved 100% precision (0% imprecision) and 100% recall on a given dataset. Clearly the closer

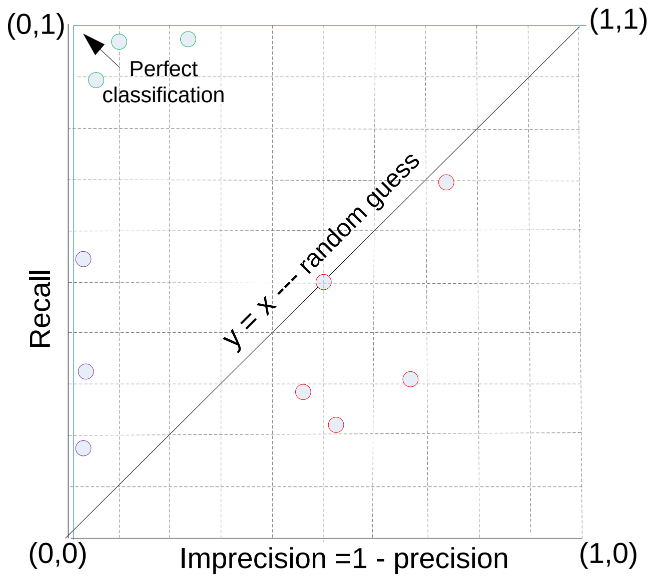


Figure 9.2: *Recall-Imprecision* space — trade-off between recall and imprecision.

a coordinate is to this *ideal point* the higher the overall accuracy of the detector under consideration for the given data set. Conversely, the worst extreme in RP space is at point (1,0), indicating total imprecision and no recall — the bottom right corner of RP-space.

The diagonal in Figure 9.2 represents detections with equal recall and imprecision, say (x, x) for $0 \leq x \leq 1$. The region around the lower extreme of the diagonal (i.e. near (0,0)) indicates low recall and high precision (low imprecision). This occurs when most of the exemplars declared to be true by the TR detector are indeed so, but where large numbers of true exemplars have not been identified. Conversely, the region around the upper extreme of the diagonal (i.e. (1,1)) indicates high recall and low precision (high imprecision). This occurs when the detector declares practically everything to be a true exemplar, thus almost never failing to identify true exemplars while also having an inordinately high number of false positives.

Broadly speaking, a random guess is likely to give a point close to $(0.5, 0.5)$ — the mid-point of the diagonal running from $(0, 0)$ to $(1, 1)$. In general, a point close to the diagonal suggests a relatively high level of inaccuracy either in terms of recall, or imprecision, or both.

By running a TR detector on the same data set for varying threshold values, an RP curve can be generated in RP space. Such a curve will characterise the behaviour of the TR detector with respect to the data set.

An RP curve consisting of the horizontal line connecting (0,1) and (1,1) represents the behaviour of an abstract detector that always has 100% recall, but displays increasing imprecision as the threshold is lowered. Note that the *area under the curve* (AUC) of such an abstract detector is 1.

The AUC will always be between 0 and 1 for any detector. It is sometimes used as an overall summary of a detector's accuracy over all threshold values. The larger the AUC, the more accurate the detector. By this measure, the abstract detector just mentioned represents an ideal. As already mentioned, points near the diagonal line from (0,0) to (1,1) suggest poor performance either in terms of recall, precision or both. Since the area under this diagonal is 0.5, a detector whose AUC is about 0.5 or less would normally be regarded as a poor one (Fawket [2003]).

The next subsection reports on investigations of various RP curves characterising `FireSat`$_3$ when using the synthetic data sets previously described.

9.3.2 Determining an effective *match score threshold function* for `FireSat`

The logic flow of Algorithm 6.3.1 is not only a broad outline of `FireSat`$_1$, but also of the other `FireSat` versions.

In each case, the algorithm is invoked with parameters specifying various threshold filter values and penalty weights, denoted by $\varepsilon_{max\%}$, $\kappa_{max\%}$, $\sigma_{max\%}$, α_{max}, , β_{min}, p_d, p_i and p_m respectively. Although default values have been assigned to these parameters, the user may optionally alter the defaults. In each case these values are available globally, and therefore also to the relevant version of the function called *computeTR* in Algorithm 6.3.1. This function determines whether a TR is present at a given position in the input string, s.

However, there are no clear guidelines on how to optimally select these weights and parameters. The RP analysis described in this subsection is intended to mitigate this challenge.

An RP analysis on an input string, s, for a given motif length, $|\rho|$, entails calculating an LC$_n$ for each position in s. Let i be any position in s and position

$j = i + |\rho|$. An LC$_n$ is calculated between substrings of s, starting at i and j respectively, each of length $|\rho|$.

Since the substrings are of equal length, the computation of the LC (and the LC$_n$) is straightforward: simply compare s_{i+k} against s_{j+k} for $k = 0, \cdots |\rho| - 1$ and add the number of matches. Let $\ell_{i,j}$ be the LC$_n$ associated with strings:

$$s_{[i,i+|\rho|)}, s_{[j,j+|\rho|)}$$

Noting that if i lies within a TR in s, then ℓ_i will tend to be relatively high, but will tend to decline towards the end of the TR. Given some threshold value, t, the following heuristic is applied in order to compute an R and P value, once ℓ_i is available for $i = 0, \cdots |\rho|$.

If $\ell_i \geq t$ then position i is considered to be an exemplar of a TR element else position i is not considered an exemplar of a TR element.

By comparing exemplars against positive exemplars in the synthetic data set, true and false positives can be identified, and thus also R and P values.

For each data set and for each threshold value, $t \in \{0.01, 0.02, \cdots 1.0\}$ the recall and 1 - precision is calculated and plotted in the relevant RP space, forming an RP curve.

RP curves were obtained for each of the 20 synthetic datasets. Recall that a data set is characterised by one of five motif lengths ($|\rho|$) and one of four maximum percentage mutation error rates (P_{err}). The resulting RP curves are displayed in Figure 9.3.

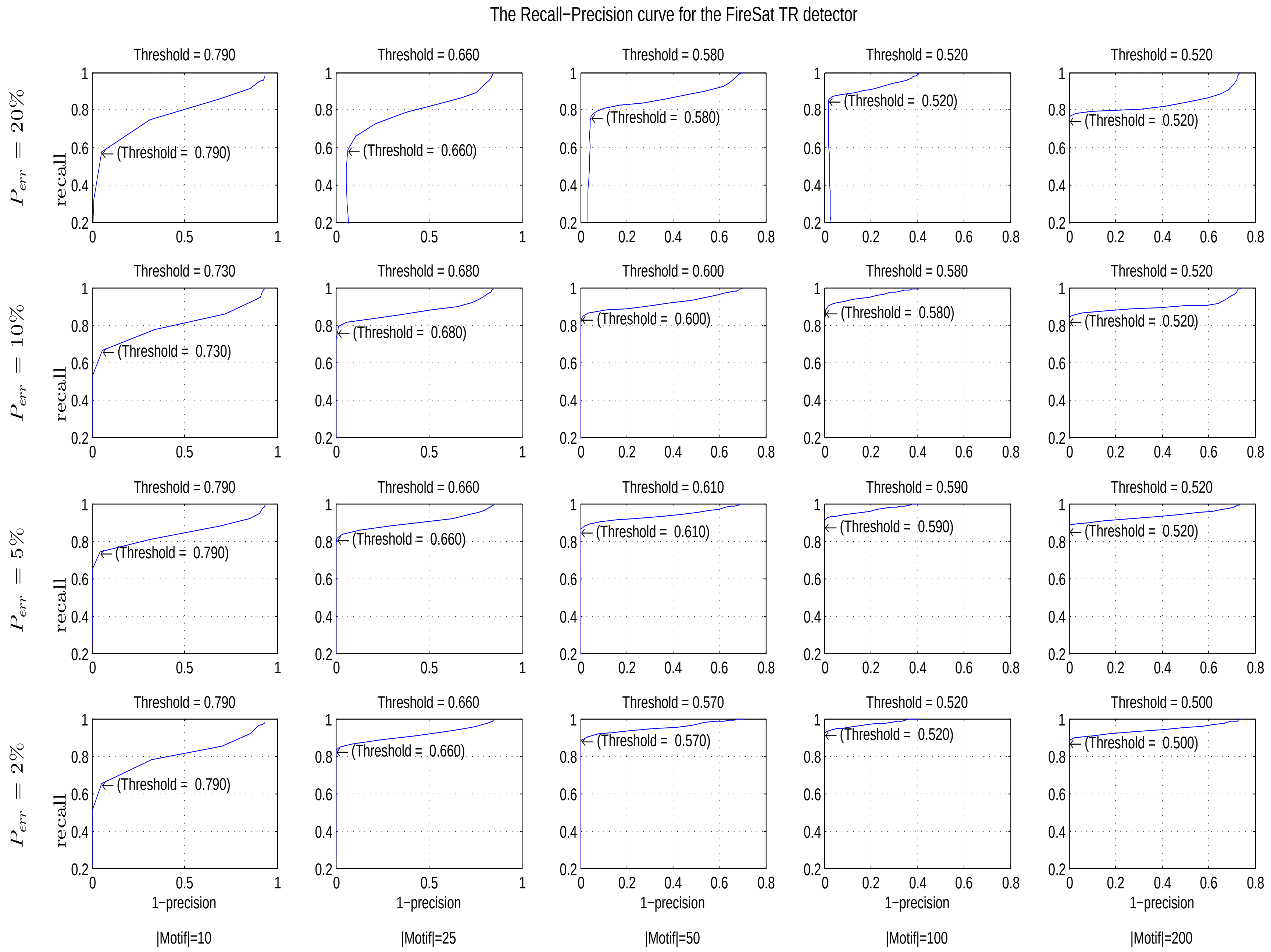


Figure 9.3: Recall:Precision.

In general, the upper right part of a curve results from low threshold values. These low threshold values result in high recall but also high imprecision. As the threshold values increase from a low of 0.01 towards 1.0, the curve dips down to the bottom left of the diagram where there is very low recall and also low imprecision.

In each of the curves, there is an inflection point (in some cases more pronounced than in others) such that a small *increase* in threshold value (change to the left) rapidly degrades the recall performance while only marginally improving precision, whereas a small *decrease* in threshold value (change to the right) only marginally improves the recall performance while fairly rapidly degrading precision (i.e. increasing imprecision). Inspection of the sub-figures in Figure 9.3 suggests an inflection point occurs at a threshold value of about:

- 0.73 to 0.79 in data sets based on a motif length of 10;
- 0.66 to 0.68 in data sets based on a motif length of 25;
- 0.57 to 0.61 in data sets based on a motif length of 50;
- 0.52 to 0.59 in data sets based on a motif length of 100; and
- 0.50 to 0.52 in data sets based on a motif length of 200.

For each P_{err} value, these inflection point threshold values as noted in the respective sub-figures of Figure 9.3 were plotted against their associated motif lengths to obtain the piece-wise linear graphs in Figure 9.4. These graphs clearly show how these threshold values decrease with length.

A *match score threshold* function (in $|\rho|$) was sought that could serve as an approximate upper bound on these piece-wise linear graphs. The function $tf(|\rho|)$ as defined in Equation (9.4) was found by a trial and error process to serve this purpose.

$$tf(|\rho|) = \frac{1}{2} + \frac{1}{\sqrt{|\rho|}} \tag{9.4}$$

It will be seen in Section 9.4.4 that a *match score threshold*, t, is calculated as $t = f \times tf(|\rho|)$ where f is some pre specified *match score factor* and $|\rho|$ is the motif length of TRs associated with the dataset under consideration. Clearly, if f is chosen less than 1, the calculated threshold will be below the curve for $tf(|\rho|)$ shown in Figure 9.4.

As reported below, trial runs were done where $f = 1$, and $f = 0.8$. The match score threshold obtained is compared against the match score — the LC_n. Thus, for a candidate TRE to be accepted as a TRE, the match score (LC_n) computed for that TRE with respect to ρ must be $\geq f \times tf(|\rho|)$.

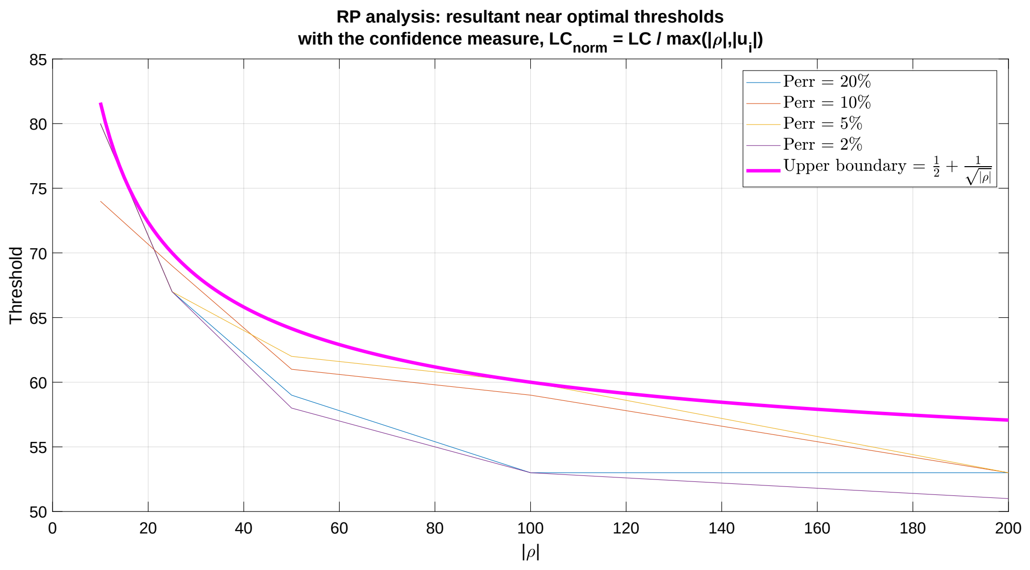


Figure 9.4: *RP* analysis — effective thresholds.

9.4 Trial runs: rival algorithms

Chapter 3 provided a concise literature overview of the most prominent TR-detectors. Formal concept lattices were used to classify these TR-detectors. In Section 3.2, `Phobos`, `Inverter`, `mreps` and `TRF` are mentioned as algorithms of potential interest to rival `FireSat`. However, since `Inverter` only has the ability to detect PTRs and not ATRs, it is not discussed further here. This section therefore reports on trial runs carried out using the remainder of these algorithms, giving an overview of the input and output in each case. Subsection 9.4.1 deals with `Phobos`; Subsection 9.4.2 presents `mreps` and Subsection 9.4.3 deals with `TRF`. Subsection 9.4.4 reports on `FireSat` runs.

The synthetic data, generated as described in the previous section, was used in the trial runs. The trial runs were conducted under the Windows 64 bit operating system. A `Matlab` script was written to manage the executions of the different algorithms and the input files.

For the runs with `Phobos`, `mreps` and `FireSat`, the same `Matlab` script parameter was used to specify both the maximum and the minimum motif lengths of TRs for which to search. However, `TRF` does not provide switches to limit searches to specific values of $|\rho|$. As a result, `TRF` was run without constraints on $|\rho|$ and the output was post-processed to extract detected TRs having the relevant motif lengths.

Since precision and recall are computed with respect to nucleotide classification rather than TR classification, P is taken as the total number of nucleotides that are present in TRs in the original data set. TP is the number of nucleotides correctly classified by the package under consideration as being part of a TR; and FP is the number of nucleotides *incorrectly* classified by the package under consideration as being part of a TR. As pointed out above, precision is then computed as $\frac{TP}{TP+FP}$ and recall as $\frac{TP}{P}$.

9.4.1 The `Phobos` runs

The `Phobos` software can be accessed at http://www.ruhr-uni-bochum.de/ecoevo/cm/cm_phobos.htm[4].

When doing a search, a number of different parameters can be set. Input parameters are divided into different related groups as follows:

- search modes;
- general options;
- output options;
- options for imperfect search; and
- requirements on satellites to be reported.

Search modes
`Phobos` provides three search modes from which the user can choose:

- Imperfect search — TRs that consist of ATREs and PTREs are detected. This is the mode that has been selected for trial runs of this study.
- Perfect search — Only PTRs are reported.
- Extend exact search — TR detection firstly identifies PTRs that are referred to as seeds. These are then extended to both sides to allow for ATRs too. The interested reader can consult the `Phobos` web page to obtain more details of this option.

General options
The three so-called *general options* included in `Phobos` are listed below:

[4]The following citation is available for `Phobos`: *Mayer, Christoph, Phobos 3.3.11, 2006-2010, http://www.rub.de/ecoevo/cm/cm_phobos.htm.* Note, however, that Mayer has not published the full details of the Phobos algorithm.

- *Processing a subsequence of an input string* — the `Phobos` GUI version provides text boxes to specify the integer positions from which detection should start (*From*) and end (*To*). Consequently `Phobos` always searches for TRs with a pre-specified $|\rho|$-length contained in the current synthetic data file being run. For the trial runs both the *From* and *To* parameters were set equal to the length of the current PTRE ($|\rho|$) of which TRs were to be detected.

- *Repeat unit range* options enable the user to indicate the range of motif lengths to be detected. Trial run searches, for the purposes of this study, searched for one motif length at a time as explained in the previous item.

- *Treating N*. When a string is sequenced and it is not clear which nucleotide belongs in a certain position, the character N is printed in the corresponding FASTA format file. `Phobos` provides the user with various options to deal with N. A detailed discussion of how N can be processed is beyond the scope of this thesis. The `Phobos` web page can, however, be consulted in this regard by the interested reader. Note that the synthetic data, used for trial runs, does not contain any Ns. Consequently this option was not utilised when trial runs were executed.

Output options
`Phobos` has five modes selectable for output format. The main features of two of these output modes are presented in this discussion. The three other formats that may be selected are:
-*general feature format (GFF)*;
-*one-per-line format* and;
-*FASTA format.*
A discussion of these formats are beyond the scope of this research. The `Phobos` users manual, available from the web, provides detailed discussions of all the output options. For the purposes of trial runs *extended* mode was selected.

Besides the previous mentioned output format options, `Phobos` output can either be in so called *standard* format or in *extended* format. For both standard and extended output the following is included:

- the genetic sequence's name;
- the length of the detected TR;
- the number of `ACGT` and `U` characters;
- the normalised repeat length;
- the score;

- the percentage of perfection; and
- the motif.

Besides the output indicated in the itemised list, *Extended Phobos Output* includes additional columns where the number of mismatches, insertions, deletions and *N*s (if *N*s are not treated as motif errors) are output. A next line outputs the type of motif errors and where they occur.

When using extended `Phobos` format there are three modes for printing a repeat sequence:

- do not print sequence;
- print repeat sequence; and
- print repeat alignment.

The meaning of the first two items is clear. The *print repeat alignment* option entails that an alignment of the repeat sequence and its perfect counterpart is printed. For the trial runs the extended `Phobos` option was used.

Other switches that can be set when an *imperfect* TR detection is executed are:

- *Mismatch score*
  During the alignment of a putative TR with its exact counterpart each match is allocated a score of 1. The mismatch score can be allocated by the user. This switch was not set for trial runs.
- *Gap score*
  `Phobos` refers to indels as gaps. Gap scores are set by the user. This switch was not set for trial runs.
- *Recursion depth*
  `Phobos` uses a recursive alignment algorithm. A higher recursion depth leads to a higher alignment quality. A recursion depth of 3 has a low alignment quality. A recursion depth of 5 has a high alignment quality. A very high alignment quality is achieved if the recursion depth is set to 7. For the purposes of the executed trial runs, the recursion depth was set to 7.

Requirements on satellites to be reported
Options are provided that allow the user to set threshold switches to prevent the output of unwanted data. For trial runs these switches were not manipulated.

Default values were used. A detailed discussion of these switches is available from the `Phobos` web page.

Note that default values were used for the remainder of the parameters. Results obtained from running `Phobos` on the synthetic data files described in Section 9.1 are displayed in Table C.1 in Appendix C.

9.4.2 The `mreps` runs

`mreps` can be accessed at: http://mreps.univ-mlv.fr/. The input parameters for `mreps` are less than that of `Phobos` and are as follows:

- `-from n` where `n` indicates the position of the nucleotide from which processing should start, in case the complete file should not be processed.
- `-to n` where `n` indicates the position of the nucleotide up to where the input string should be scanned for TRs.
- `-step n` this switch controls how the input string is read and buffered in memory. The default value of -1 was allocated for the trial runs. This means that the complete source string (s) was uploaded to memory and processed at once.
- `-minsize n` `n` indicates the minimum length a detected TR should have in order to be reported. This switch was not set for trial runs.
- `-maxsize n` here `n` is the maximum length a TR should have to be reported. This switch was not set for the trial runs.
- `-minp n` `n` indicates here the minimum $|\rho|$ of TRs to be detected. This parameter was set to 10, 25, 50, 100 or 200, depending on $|\rho|$ of the dataset being run.
- `-maxp n` where `n` indicates the maximum $|\rho|$ of TRs to be detected. Again, this parameter was set to 10, 25, 50, 100 or 200, in line with $|\rho|$ of the current dataset.
- `-exp x` here `x` represents the minimum number of TREs to occur for a string to be reported as a detected TR. For the trial runs `x` was set to two.
- `-res n` the term `-res n` refers to the so called *resolution* parameter. The resolution parameter was introduced by Kolpakov et al. [2003] to manage error toleration within a complete detected TR. The value of the resolution parameter depends on the value of a so called error-rate, calculated when the *best* period for a TR is determined and when merging is done. The larger

`n` is, the more motif errors are tolerated. A high resolution value does, however, not prevent more *perfect* TRs from being detected and reported. A higher resolution value implies a slower run time. The resolution value is related to the motif length. In practice, a *good* resolution value depends on $|\rho|$. Full details of the resolution parameter are beyond the scope of the present discussion. These details can be found in Kolpakov et al. [2003]. According to the authors of `mreps`, a resolution value equal to 5 is usually sufficient to detect all meaningful TRs whose motif lengths $10 \leq |\rho| \leq 15$. However, if $|\rho| \geq 15$ the resolution should be adjusted. Depending on $|\rho|$ the value of the resolution parameter can be increased to 50. For the trial runs the resolution parameter was set to 50.

- `-allowsmall` If the `-allowsmall` switch is off, then small TRs are filtered out. It was observed that the difference between data detected when the switch is *on* and when the switch is *off* is insignificantly small. A few more nucleotides are detected correctly if $|\rho| = 10$ and a few more nucleotides are detected incorrectly for $|\rho| = 25$ if the switch is off. The `-allowsmall` switch does not have any impact on the outcome of detected data if $|\rho| = 50$; $|\rho| = 100$ or $|\rho| = 200$. For the trial runs the `-allowsmall` switch was *off.*

- `-xmloutput file` This switch indicates that the output file should be in .xml format.

- `-fasta` This switch was on for the trial runs, indicating that the format of the DNA sequence input file is FASTA.

- `-noprint` If the `-noprint` switch is set then the attributes of detected TRs are output only, not the TRs themselves. This switch was not set for the trial runs.

The `mreps` trial run detection results are included in Table C.2, in Appendix C.

9.4.3 The `TRF` runs

In spite of the fact that `Tandem Repeats Finder (TRF)` is fairly old (published by Benson [1999]), it is still used for bench-marking purposes. For example, Lim et al. [2012] bench-marked against `TRF`. Schaper et al. [2015] included `TRF` in their studies. During my previous studies, the different versions of FireμSat were also bench-marked against `TRF`. I provided an extensive discussion on the input and output parameters of `TRF` in De Ridder [2010]. This text can be consulted for additional details about any of the parameters of `TRF` that are briefly discussed next.

1. `File`: *The input DNA sequence file in FASTA format.*

2. `Match`, `Mismatch`, `Delta`: *Alignment parameters that represent the weights for matches, mismatches and indels respectively.*

   Lower weights entered as the alignment parameters of `TRF` allow alignments with more mismatches and indels. `Match` = 2 has proven effective with `Mismatch` and `Delta`[5] ranging between 3 and 7. Mismatch and indel weights are interpreted as negative numbers. The values allowed are 3, 5 and 7. Benson [2003a] points out that in these types of alignment options, 3 is more permissive and 7 is less permissive (i.e. $-3 > -7$). Benson [2003b] recommends the values 2, 7 and 7 for `Match`, `Mismatch` and `Delta`, respectively. These recommendations were used in the trial runs.

3. `PM` *and* `PI`: *Detection parameters*
   Detection parameters consist of a matching probability `PM` and an indel probability `PI`. By default, `PM` = 0.80 (80) and `PI` = 0.10 (10). The default values were used for the trial runs.

4. `Minscore`: *Minimum alignment score*
   The minimum alignment score indicates the alignment score that must be met or that must be exceeded for a tandem repeat to be reported [Benson [2003a]].

   The alignment of two or more possibly approximate tandem repeats of a motif (referred to as a pattern by Benson) of which the length of the repeated motif is n, is modelled by a sequence of n-independent Bernoulli trials[6]. If two potential TREs are aligned then the alignment score is calculated using the weight penalties indicated in item 2 of this enumerated list. For the purposes of the trial runs presented here `minscore` was set to 20,

[5] "Delta" represents weights for indels.

[6] Bernoulli trials are associated with a succession of coin tosses and is a fixed value over the tosses or trials. In this sense, the tosses / trials are independent of each other.

the default value. Several trial runs were conducted and the most accurate detection outcome was achieved with `minscore` set to 20.

5. `Maxperiod`: *Maximum period size*
   In the present context, period size may be considered to be the same as $|\rho|$. As a default, `TRF` will find all TRs with a period size between 1 and 2000. However, the period size can be made smaller in length by setting `Maxperiod` [Benson [2003a]]. For different trial runs this parameter was set, each time, equal to the *motif length* ($|\rho|$) holding for the current data set. Note that `TRF` does not provide switches for searching for TRs of a specific motif length, e.g $|\rho| = 25$. Instead, if `Maxperiod` is set to 25 and run on a data set with TRs whose motif length is 25, then `TRF` will attempt to detect TRs of motif length 25 or less in that data set. As a result, `TRF` could possibly miss a TR in a data set with motif ρ but classify a substring of that TR as a TR with motif length $< |\rho|$.

`TRF` provides three additional optional switches that the user may specify as part of the command line input, namely:
`-f`: *flanking sequence*;
`-m`: *masked sequence file* and;
`-d`: *data file*.
These switches were not set for trial runs.

`TRF` generates a summary table of repeats as well as an alignment explanation as output. Detailed discussions of these are beyond the scope of this thesis and can be found in De Ridder [2010].

Note that `TRF` detects repeats of motif lengths ranging from 1 to 2000 nucleotides. Whenever a TR contains numerous repeats, the same repeat will be detected at various period sizes (motif lengths). This does, however, not influence the results of the trial runs.

Information about TRs detected by `TRF` with the indicated parameter settings is included in Table C.3 of Appendix C.

9.4.4 FireSat runs

To empirically examine the detection ability of the different underlying algorithmic principles proposed in this thesis, three versions of FireSat, namely FireSat$_1$, FireSat$_2$ and FireSat$_3$ were implemented and run on the synthesised data.

Recall that, as described earlier, FireSat$_1$, FireSat$_2$ and FireSat$_3$ each establish a candidate string as a possible TRE in their own way. FireSat$_1$ determines the candidate TRE without computing an LD at all, FireSat$_2$ relies on computing an LBD, and FireSat$_3$ computes an LC that is compared against a threshold value derived from a factor setting that is provided as input, as earlier described in Subsection 9.3.2.

Each FireSat version then goes through the same steps of comparing the candidate TRE against the various filter settings (substring error, etc.), only accepting the candidate TRE into the TR to date if it passes these various filter tests.

However, FireSat$_2$, as implemented for the empirical tests described below, has been slightly tweaked from its original description. In determining the candidate TRE, the computed LBD is converted into an LC value[7] and the resulting LC$_n$ is computed. As for FireSat$_3$, the LC$_n$ is then compared against a calculated match score threshold value .[8]

Note that FireSat$_{2'}$ was not fully implemented. Instead, it was empirically verified that the approach followed in FireSat$_{2'}$ to determine the LC between two strings indeed yields exactly the same LC as the approach of FireSat$_3$. As a result, TRs determined by a full implementation of FireSat$_{2'}$ would correspond identically to those determined by FireSat$_3$, if all other parameters were set at the same respective levels.

All FireSat versions were implemented as Matlab scripts and run against the synthetic data. Details, describing the different software implementations of FireSat, are given in Appendix C.

For the trial runs the 20 synthetic data sets were input to `try_do_all_trs_11.m` and `try_do_all_trs_11_0p8.m`. The results obtained are reported in Section 9.5. The same results can be obtained by manipulating the parameters of the functional FireSat$_3$ Matlab implementation.

The command `help fsat` gives a listing of the parameters that can be used as input to FireSat$_3$. The following parameter settings can be used for the runs on the 20 different data sets:

[7]Of course, this is only an estimated value, since it is based on the LBD rather than the LD *per se*.

[8]Recall from Section 9.3.2 that the parameters input to the the match score threshold function are $|\rho|$ and a match score threshold factor, f.

- `sfname` - the name of the source file.
  Any source string in FASTA format can be used. For the trial runs the 20 synthetic data sets were used.

- `mlen` - the *motif length* to use when searching for TRs in the file.
  Depending on the data file, one of the following should be provided as the motif length: [10,25,50,100,200].

- `tfact` - the *match score threshold factor* Separate runs should be made on each data set, first using a match score threshold factor of 0.8 and then using a factor of 1.0.

- `m_err_max_perc` - *the maximum motif error percentage* This refers to the filter limit denoted by $\varepsilon_{max\%}$[9] in the text above. It limits the maximum number of mutations allowed in a TRE. To simulate the trial runs set it to $2 \times P_{err}$, where P_{err} is the value associated with the data set used.

- `mis_max_perc`, `del_max_perc` and `ins_max_perc` - *the maximum mismatch, deletion and insertion error percentages* respectively.
  These can be used for fine tuning the number of mutational errors in a TRE by type. No such fine tuning was attempted for the trial runs. To simulate the trial runs each of these switches should be set to the P_{err} value of the current data set.

- `sigma_max_perc` - The *maximum weighted substring error* ($\sigma_{max\%}$)
  The default value of ∞ should be used to simulate the trial runs.

- `p_mis`, `p_del` and `p_ins` - The *mutation penalties weights*
  These should be left at their default values of 1. Their values have no effect on the outcome of the trial runs.

- `beta_min` - The *minimum number of TREs* in a TR (β_{min}).
  The default value, namely 2, should be used to simulate the trial runs.

Each run delivered the following output for each detected TR:

- `mlen` - The motif length used in this run.

- `tr_pos` - The start position of the detected TR.

- `motif_errperc_max` - The value of $\varepsilon_{max\%}$.

- `tr_len` - The length of the detected TR.

[9]`FireSat` will detect PTRs if $\varepsilon_{max\%}$ is set equal to 0.

- `ntres` - The number of TREs in the TR.
- `conf` - The mean value of the LC$_n$s of the TREs (excluding the motif) in the TR.

The *Precision*, *Recall* and F_1 *score* values resulting from the `FireSat`$_1$, `FireSat`$_2$ and `FireSat`$_3$ implementations are displayed in Appendix C, Tables C.4, C.5, C.6, C.7 and C.8. In Section C.1 of Appendix C it is explained *how* to run `FireSat`.

9.5 Discussion of results

For each of the aforementioned TR-detecting algorithms the *recall*, *precision* and F_1 *scores* were calculated with respect to the various synthetic data sets. These values for the various data sets run, using the various algorithms, can be seen in the following tables in Appendix C:

- `Phobos` results are displayed in Table C.1;
- `mreps` results are displayed in Table C.2;
- `TRF` results are displayed in Table C.3;
- `FireSat`$_1$ results are displayed in Table C.4;
- `FireSat`$_2$ results are displayed in Table C.5 run with a *match score threshold factor* of 0.8; and
- `FireSat`$_2$ results run with a *match score threshold factor* of 1.0 is presented in Table C.6;
- `FireSat`$_3$ results run with a *match score threshold factor* of 0.8, are displayed in Table C.7; and
- `FireSat`$_3$ results run with a *match score threshold factor* of 1, are displayed in Table C.8.

Inspection of this data shows that `FireSat`$_1$ does not perform as well as `FireSat`$_2$ and `FireSat`$_3$. However, for $|\rho| = 200$, the TRs detected by `FireSat`$_1$ compares well against those detected by `TRF` and `mreps`.

It will also be seen that results for `FireSat`$_2$ are not given for all data files. Recall that `FireSat`$_2$ composes a new FA for each nucleotide in the input string until a TR that starts at that nucleotide is detected. Consequently the runtime of `FireSat`$_2$ increases dramatically as $|\rho|$ increases. Furthermore the memory requirements of `FireSat`$_2$ grows significantly as $|\rho|$ increases. For these reasons, `FireSat`$_2$ was only run on datasets where $|\rho| = 10$.

For these data sets, `FireSat`$_2$ run with a *match score threshold factor* of 0.8, outperforms `TRF`, `mreps` and `FireSat`$_3$ in terms of the F$_1$ scores. Its accuracy is on a par with that of `Phobos`.

Figure 9.5 visualises the relative performance of `mreps`, `TRF`, `Phobos` and `FireSat`$_3$, both in the case of where the match score threshold factor = 0.8 throughout (denoted in the figure by `FireSat`$_{3(0.8)}$, and in the case where it is 1.0 (denoted in the figure by `FireSat`$_{3(1.0)}$. The figure summarises relative algorithm performance of *all* motif lengths at each of the four P_{err} values used. The recall performance is shown in green and the precision in blue.

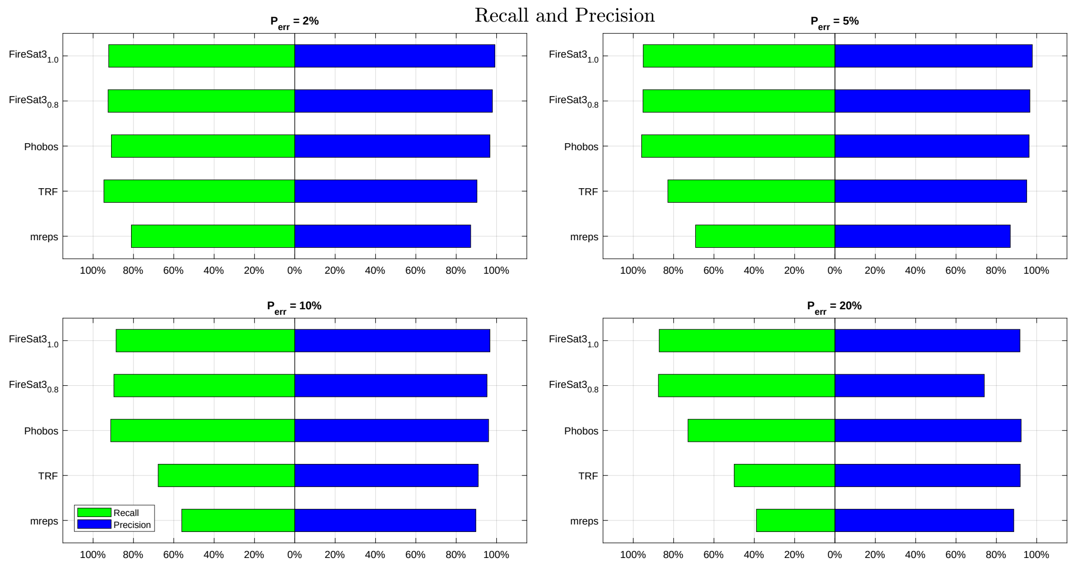


Figure 9.5: Recall:Precision.

This figure suggests that `FireSat`$_{3(0.8)}$, `FireSat`$_{3(1.0)}$ and `Phobos` are clearly superior to `mreps` and `TRF`. Inspection of the data relating to the detailed performance of the latter two algorithms will show, however, that while they perform relatively well for motif lengths of 10 their performance degrades when the motif length increases. Note also that their performance improves somewhat as P_{err} becomes smaller.

Figure 9.6 displays the F_1 score for these algorithms and confirms these findings more concisely. Note that by this measure, `FireSat`$_{3(1.0)}$ slightly outperforms `FireSat`$_{3(0.8)}$ and `Phobos` in all cases except where $P_{err} = 10\%$. In that case,

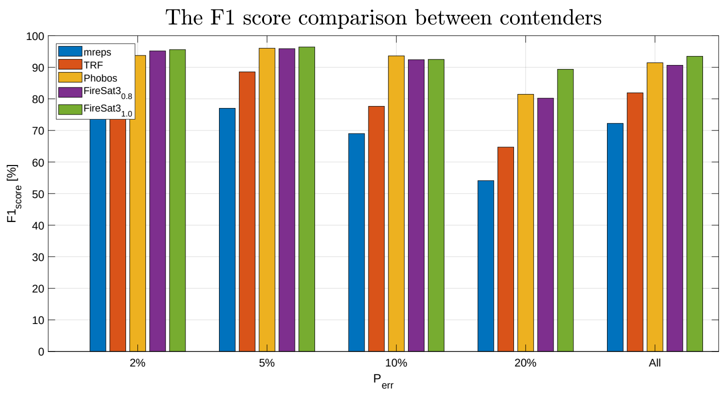


Figure 9.6: F_1 score.

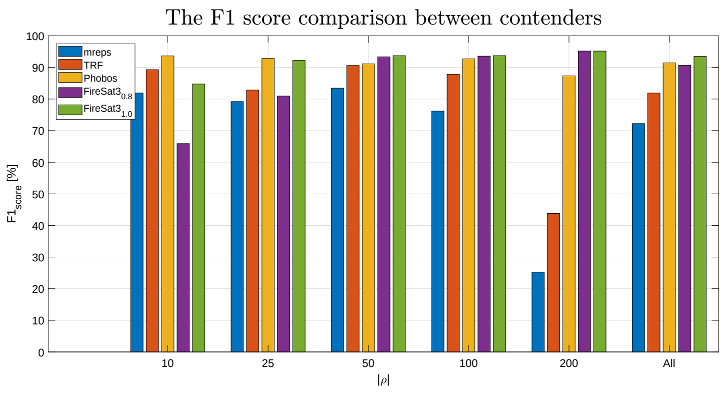


Figure 9.7: F_1 score — length compare.

`Phobos` just takes the lead. It is also relevant to note that $\texttt{FireSat}_{3(1.0)}$ significantly outperforms its contenders when $P_{err} = 20\%$.

Figure 9.7 presents F_1 scores of the respective $|\rho|$-lengths for the contenders. $\texttt{FireSat}_{3(0.8)}$ and $\texttt{FireSat}_{3(1.0)}$ outperforms their contenders where $|\rho| = 200$. The same two algorithms slightly outperforms `Phobos` if $|\rho| = 100$ and $|\rho| = 50$. `Phobos`, however, outperforms $\texttt{FireSat}_{3(0.8)}$ and $\texttt{FireSat}_{3(1.0)}$ for shorter motif lengths i.e. $|\rho| = 25$ and $|\rho| = 10$. Note that the performance of `TRF` and `mreps` is slightly more competitive for shorter lengths too.

$\texttt{FireSat}_3$ used the MDI mutational precedence order for these trial runs. Table C.9 reports on results achieved by $\texttt{FireSat}_3$ where MID was used instead. Comparing Tables C.9 and C.8 shows that the mutational priority order did not have a noteworthy impact on $\texttt{FireSat}_3$ results.

9.6 Conclusion

The foregoing has demonstrated that in most circumstances, $\texttt{FireSat}_{3(1.0)}$ is a somewhat more accurate TR-detector than its contenders. Its superiority was particularly pronounced for data sets where $P_{err} = 20\%$. One could make the same claims for $\texttt{FireSat}_{2'}$, since the TRs it detects would be exactly the same as those of $\texttt{FireSat}_3$.

$\texttt{FireSat}_1$ was not as effective at TR detection, but still compared favourably against other existing TR detectors whenever TRs with longer motif lengths should be detected.

Although $\texttt{FireSat}_2$ appears to be a highly accurate TR-detector alternative, its implementation presents practical challenges.

The next chapter concludes this thesis and points to future research initiatives.

Chapter References

G. Benson. Tandem Repeats Finder. *Nucleic Acids Research*, 27(2):573 – 580, November 1999.

G. Benson. Tandem Repeats Finder: Definitions: FASTA Format. Online: http://tandem.bu.edu/trf/trf.definitions.html, Sept 2003a.

G. Benson. Tandem Repeats Finder:Unix Version help: using Tandem Repeats Finder for Unix. Online: file://D:/Tandem%20Repeats%20Finder%20%20Unix%20Version%20Help.htm, Sept 2003b.

C. De Ridder. Flexible Finite automata-based algorithms for detecting microsatellites in DNA. Master's thesis, Department of Computer Science, University of Pretoria, July 2010.

T. Fawket. ROC Graphs: Notes and Practical Considerations for Data Mining Researchers. Online: http://www.hpl.hp.com/techreports/2003/HPL-2003-4.pdf, 2003.

Y. Ke and R. Sukthankar. PCA-SIFT: A More Distinctive Representation for Local Image Descriptors. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'04, pages 506–513, Washington, DC, USA, 2004. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=1896300.1896374`.

R. Kolpakov, G. Bana, and G. Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *PubMed Central Nucleic Acid Research* , 31(13):3672–3678, 2003.

K. G. Lim, C. K. Kwoh, L. Y. Hsu, and A. Wirawan. Review of tandem repeat search tools: a systematic approach to evaluating algorithmic performance. *Briefings in Bioinformatics*, May 2012.

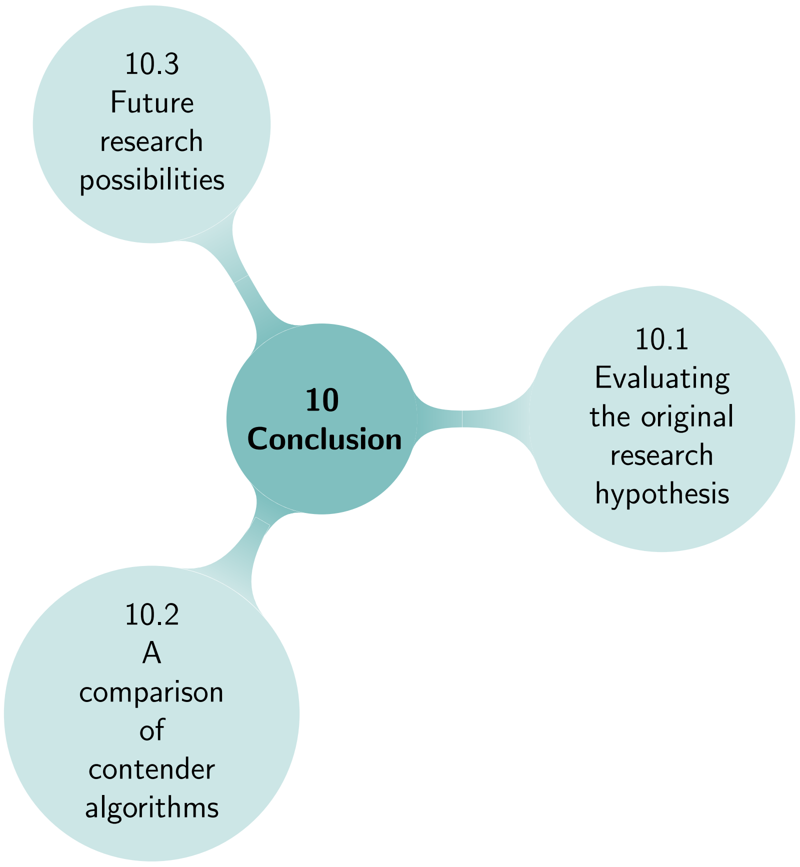
K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 2, pages 257–263. IEEE, 2003. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8603`.

E. Schaper, A. Korsunsky, A. Messina, R. Murri, J. Pečerska, H. Stockinger, S. Zoller, I. Xenarios, and M. Anisimova. TRAL: Tandem repeat annotation library. *Bioinformtics*, 31(18):3051–3053, 2015.

C. J. Van Rijsbergen. *Information Retrieval*. Butterworths, 2nd edition, 1979. URL `http://www.dcs.gla.ac.uk/Keith/Preface.html`.

W. Zhu, N. Zeng, and N. Wang. Sensitivity, specificity, accuracy, associated confidence interval and ROC analysis with practical SAS implementations. In *NESUG proceedings: Health Care and Life Sciences*, STOC '78, 2010. URL `http://www.lexjansen.com/cgi-bin/xsl_transform.php?x=shl&c=nesug`.

K.H. Zou, J. O'Malley, and L. Mauri. Receiver-operating characteristic analysis for evaluating diagnostic tests and predictive models. *Circulation*, 115(5):654–657, 2007.



10 Conclusion

"We cannot do great things on this earth. We can only do small things with great love." ... Mother Teresa

---

This concluding chapter has three sections. Section 10.1 evaluates the hypothesis that was presented in Chapter 1 in terms of the research findings described above. Section 10.2 summarises the outcome of the trial runs reported on in Chapter 9. Section 10.3 points to future research initiatives suggested by this study.

10.1 Evaluating the original research hypothesis

Recall from Chapter 1 that this research project was directed at investigating the following hypothesis:

> *Finite Automata can be used to accurately detect minisatellite and satellite TRs in DNA.*

The investigation led to four different `FireSat` algorithms. Each relied on a newly defined type of automaton, specifically tailored for the task. The harmonic mean of recall and precision measurements was used to compare the performance of these algorithms against one another as well as against the performance of relevant other algorithms in the domain (as identified in Chapter 3). The use of the harmonic mean as a suitable metric was motivated in Chapter 9. Detailed results of these comparisons are reported on in tabular format in Appendix C. Chapter 9 graphically represents some of the main results.

The four algorithms and the different finite automata upon which they rely can be considered under four sub-hypotheses, each of which are now mentioned and briefly discussed below with respect to their overall performance as measured by the harmonic mean:

- *The cascaded PFAs as used in **FireSat**$_1$ result in accurate detection of minisatellite and satellite TRs in DNA*

  Recall that in `FireSat`$_1$, quad-PFAs were cascaded together and possibly also cascaded with a mono-, di- or tri-PFA. This arrangement was indeed able to detect a subset of the TRs present in the generated synthetic data. However, the F$_1$ scores of `FireSat`$_2$ and `FireSat`$_3$ are higher than those of `FireSat`$_1$ in all but one case, namely when $P_{err} = 20\%$ and $|\rho| = 10$, where `FireSat`$_1$ marginally outperforms the others. Consequently it can be concluded that the cascading and decoration of quad-PFAs in a brute force manner as was done for `FireSat`$_1$ does not deliver a TR detecting algorithm that is as accurate as the other proposed algorithms.

- *The cascaded pCA$_{T3}$s as used in **FireSat**$_2$ result in accurate detection of minisatellite and satellite TRs in DNA*

  For each potential first TRE `FireSat`$_2$ builds a *new* LBD NCA$_{T3}$ by cascading pCA$_{T3}$s corresponding to unique substrings into which ρ can be divided. It is time consuming to construct a *new* LBD NCA$_{T3}$ for each potential first TRE of a TR. Furthermore the resulting NCA$_{T3}$s are memory intensive since *all* the potential LBD paths are included. The path with the smallest distance is selected, while also giving preference to specific mutation types, as discussed earlier. Consequently `FireSat`$_2$ was only implemented for a $|\rho| = 10$. For $|\rho| = 10$ `FireSat`$_2$ outperforms `FireSat`$_1$ and `FireSat`$_3$. `FireSat`$_2$ also performs better than `mreps` and `TRF`. Its accuracy performance is on a par with that of `Phobos`.

  `FireSat`$_2$ is however hampered by runtime and memory difficulties. It became clear that if, for a given motif length, an automaton could be constructed once only and then subsequently decorated with the next potential first element (ρ) of a TR these limitations could be addressed. This insight led to the proposal of `FireSat`$_{2'}$.

- *The cascaded nPCA$_{T3}$s as used in **FireSat**$_{2'}$ result in accurate detection of minisatellite and satellite TRs in DNA*

Given each motif length, FireSat$_{2'}$ entails the cascading of mono-pNCA$_{T3}$s in a fixed way, such that transitions are merely relabelled whenever the next motif of the same length is to be considered. To further prevent a search space explosion, paths that allow for more than the allowed $\varepsilon_{max\%}$ errors are eliminated from the search space. Even though FireSat$_{2'}$ was not fully implemented, FireSat$_{2'}$ and FireSat$_3$ are deemed to have identical performance. This is because the two algorithms select TREs according to the same logic, relying on the computation of the LC$_n$ between two strings.[1] The algorithms differ in the way in which the LC$_n$ between these strings is computed, but empirical tests verified that the same LC$_n$ value is indeed always obtained. The same claims that are made for FireSat$_3$ below hold consequently for FireSat$_{2'}$ too.

- *The composition of CA$_{T3}$s as described in **FireSat$_3$** results in accurate detection of minisatellite and satellite TRs in DNA*

  FireSat$_3$ is a very accurate, memory efficient TR detector. In particular FireSat$_3$ detects TRs more accurately than contender algorithms for longer repetitive motifs, for example if $|\rho| \geq 50$. Furthermore FireSat$_3$ is also more accurate whenever TRs should be detected that contain a high motif error percentage, for example, if $\varepsilon_{max\%} = 20\%$.

  The approach of FireSat$_3$ differs significantly from that of FireSat$_1$, FireSat$_2$ and FireSat$_{2'}$. If fully implemented, FireSat$_{2'}$ would, however, be as accurate and generic as FireSat$_3$. The accuracy performance is on a par with the best in the world.

10.2 A comparison of contender algorithms

From Chapter 9 it is clear that, in most cases, FireSat$_3$ detects TRs more accurately than mreps and TRF. Of the contender algorithms, Phobos[2] has been identified as the most accurate. In summary it is noted that:

- Phobos is more accurate than FireSat$_3$ when detecting tandem repeats in synthetic data whenever the motif error percentage is 10%. However, FireSat$_3$ detects tandem repeats more accurately in the generated synthetic data when motif error percentages of 2%, 5% and 20% are present.

[1] The FireSat$_{2'}$ implementation calculates the LD as well as the LC.

[2] Note that the algorithmic details of Phobos have not been published. The author of Phobos requests that the Phobos website: http://www.mybiosoftware.com/sequence-analysis/4834 should be referenced if results generated by Phobos are published.

- An analysis of detected tandem repeats, processed per motif length, shows that `FireSat`$_3$ performs better than `Phobos` for the data sets where the motif length is longer than 25. (See Figure 9.7.)
- `FireSat`$_3$ consistently performs well for all motif errors and all motif lengths. There are, however, certain special combinations of P_{err} and motif lengths where some of the contender algorithms outperform `FireSat`$_3$. To some extent, the performance of the contender algorithms seems data dependant.

Of course all these claims also hold for `FireSat`$_{2'}$. While exploring the main hypothesis and subsequent sub-hypothesis of the research, the notion of a counting automaton — later called a CA_{T3} — suggested itself. It was realised that by simply adjusting the number of *counter-against-counter* comparisons, additional types of automata could be defined that accept languages belonging to different levels within the Chomsky language hierarchy. Chapter 4 records these findings as a by-product of this thesis, even though the material is not concerned with TR detection *per se*.

10.3 Future research possibilities

There are various possibilities for extending, refining or varying some of the search strategies in each of the four algorithms developed in this thesis. Some of these possibilities are listed below.

`FireSat`$_1$

A DFA constructed in `FireSat`$_1$ is made up of cascaded quad-pDFAs plus a tail pDFA that may be shorter. The decision to base `FireSat`$_1$ on concatenated quad-pDFAs was a pragmatic one, derived in part from experience in using ADFAs for microsatellite detection in `Fire`μ`Sat`$_2$. The fact that $|\Sigma| = 4$ influenced the decision to use quad-pDFAs. It held up the prospect of efficiently recognising minisatellites or satellites based on a DFA that is built from a limited number of unlabelled "off-the-shelf" skeleton pDFAs. Had larger pDFAs been used (e.g. a pDFA recognising a string of five nucleotides and its mutations), the number of off-the-shelf skeleton pDFAs would have had to increase. More compromises would also have to be made in terms of dealing with mutations.

However, there might be applications where a version of `FireSat`$_1$ based on tri-pDFAs instead of quad-pDFAs would be of particular value. This is because protein synthesis relies on sequences of amino acids, each of which is encoded by exactly *three* adjacent nucleotides (Crick et al. [1961]). The precise possibilities in this regard is a matter for future research.

Another line of possible future investigation is to determine *how* $\texttt{FireSat}_1$ will perform if ρ is divided into substrings with unique characters. The principles outlined for $\texttt{FireSat}_2$ in Section 7.1 could be employed. Instead of the mono-, di-, tri- and quad-pCA$_{T3}$s that $\texttt{FireSat}_2$ cascades, $\texttt{FireSat}_1$ could cascade mono-, di-, tri- and quad-pDFAs.

$\texttt{FireSat}_2$

The NCA$_{T3}$ built by $\texttt{FireSat}_2$ results in inefficiencies in terms of run time and memory management. Decisions about the next TRE are based on an LBD rather than an actual LD computation. The precise relationship between the LBD calculated by the $\texttt{FireSat}_2$ NCA$_{T3}$ and the actual LD has not been explored in this thesis. While it is clear that many times, the LD and LBD will be exactly the same, it is left for future research to spell out the precise characteristics of source and destination strings that generate differences between them.

It may be meaningful to investigate alternative ways of implementing $\texttt{FireSat}_2$ to improve memory and run time effectiveness. Note that the run time and the number of states will be reduced drastically if $\texttt{FireSat}_2$ is pruned in line with $\varepsilon_{max\%}$.

There are other problems within the context of computational biology where an NCA$_{T3}$, similar to the NCA$_{T3}$ constructed for $\texttt{FireSat}_2$, can be built once and then re-used. This can be done, for example, in the context of a global search where a genetic query sequence is given and a database should be searched for related sequences.

$\texttt{FireSat}_{2'}$ and $\texttt{FireSat}_3$

$\texttt{FireSat}_3$ proved to be very accurate for TR detection. Since $\texttt{FireSat}_{2'}$ would detect exactly the same TRs it would be as accurate as $\texttt{FireSat}_3$ if fully implemented.

A threshold function together with a threshold factor have been established, that are used in conjunction with one another to increase the detection accuracy of $\texttt{FireSat}_3$.

This algorithm could, however, be implemented in a more runtime effective way, as well as a more human friendly manner. To improve the runtime significantly, an FPGA implementation of $\texttt{FireSat}_3$ is a possibility for the future. Future attention should also be given to improving the user interface.

There are a variety of other pattern matching problems where the $\texttt{FireSat}_{2'}$ and $\texttt{FireSat}_3$ approaches may be effective. Here are a few examples.

- Given two strings, find their common approximate substrings. Of course, in this problem, the extent of approximation would need to be specified.
- Search for the best approximate match of a small string in a larger string.
- Solve the *shortest superstring problem.* This problem takes as input a large string, referred to as the superstring, and a set of short strings. The objective is then to find, in the superstring, the smallest substring that contains all the short strings.

Finally, ways of extending, refining and applying the the counting automata described in Chapter 4 deserve further investigation.

Chapter References

F.H. Crick, L. Barnett, and R.J. Watts-Tobin. General nature of the genetic code for proteins. *Nature*, 192:1227—1237, 1961.

A

Counting Automata Type 0

Counting Automata Type 0 (CA$_{T0}$s) and TMs are considered here as evaluators of simple boolean expressions. Note that these are preliminary ideas.

A TM always has natural output in the sense that when it stops processing, then whatever is left on the tape can be considered to be its output. This is irrespective of whether or not the tape has been solely used as a scratch pad. In a very natural manner, then, it follows that we can use a TM to do calculations by using, for example, binary encoding.

In contrast to TMs, CAs do not have *natural output.* It was noted in previous sections that output capability can be added to DFAs resulting, for example, in Moore and Mealy machines. Such DFAs with output capability are sometimes referred to as transducers. By taking into account what is put into, left in or popped from the *stack*, PDAs can also serve the roll of transducers—especially DPDAs for doing parsing.

CAs can similarly be modified to output their counters in a meaningful manner and serve the role of transducers. For the purposes of this thesis, however, CAs with output capability will not be considered.

Of interest is the evaluation of a simple boolean expression with one of the operators $>, \geq, <, \leq, =$ or $\neq$. Such a boolean expression either yields a *true* value (an outcome of accept in our context) or a *false* value (a *reject* outcome that is equivalent to *crash* in our context). CA$_{T0}$s are defined in Definition A.0.1.

Definition A.0.1. ***Computing Automata Type 0***
A Computing Automata Type 0 (CA_{T0}) is a 6-tuple $(Q, \Sigma, \delta^+, \delta^-, q_0, \digamma)$ where:

- $Q = \{q_0, q_1, q_2, \cdots q_n\}$ *is a non-empty finite set of states. Every* q_i *is a tuple — $\langle p_i, d_i\rangle$ where d_i is an integer counter and p_i is a state identifier. The initial value of d_i is zero. d_i is incremented or decremented every time state $q_i = \langle p_i, d_i\rangle$ is entered.*
- $q_0 \in Q$ *is designated to be the start state.*
- *A subset of Q represents the final states ($\digamma$).*
- $\Sigma = \{a_1, a_2, a_3 ... a_s\}$ *is a finite alphabet.*
- $\delta^+ : Q \times \Sigma \to Q$ *is a possibly partial incrementation transition function. For $\delta^+(q_m, a_r) \to q_i$, every time state $q_i = \langle p_i, d_i\rangle$ is entered d_i is incremented. δ^+ is graphically depicted with $\blacktriangleright$.*
- $\delta^- : Q \times \Sigma \to Q$ *is a possibly partial decrementation transition function. For $\delta^-(q_m, a_r) \to q_i$, every time state $q_i = \langle p_i, d_i\rangle$ is entered d_i is decremented if $d_i > 0$. In case $d_i = 0$ a zero signal, α, is triggered. δ^- is graphically depicted with $\triangleright$.*

States that are not final states may be cloned. *By this is meant that the original state, say q_i, is* replaced *by two or more clones, say $q_{iA} \cdots q_{iZ}$. Cloned states share the value of their counter, d_i. To avoid redundancy, the incoming and/or outgoing edges of clone states must differ. Note that although a start state may be cloned, only one start state may have the functionality of a start state, indicated by an incoming edge out of no where.* □

In what follows, illustrations are provided of how some of the simple relational expressions that have been assumed possible to evaluate can be defined by TMs and CA_{T0}s (see Figures A.1, A.2, A.4(a), A.4(b) and A.5).

All input is assumed to be of the form $a^x ca^y \Delta\Delta\Delta \cdots$, where $x, y \geq 0$. The tape is assumed to be of infinite length. An example of input is `aaaaacaaaΔΔ`$\cdots$. The five `a`'s on the left of the `c` represent the left operand. The three `a`'s on the right of the `c` represent the right operand.

A.1 A TM halting on $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ when $x = y$

Figure A.1 presents a TM halting on $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ where $x = y$.

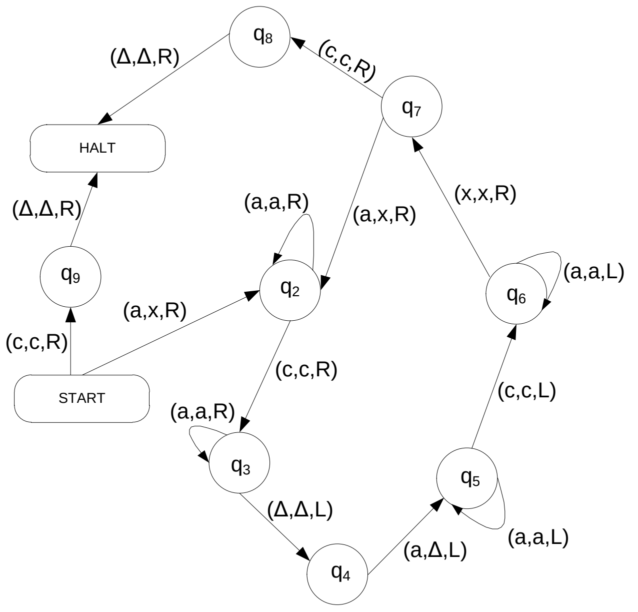


Figure A.1: A TM halting on $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ where $x = y$.

The TM presented in Figure A.1 determines whether it should halt (indicating the first group of a's is equal to the second group of a's) or not by processing the described input string as follows. In the start state a c or an a is read.

If a c is read in the start state, it means that $x = 0$. Consequently y should also be equal to 0. In state q_9 a Δ is read to proceed to the halt state — $x = y = 0$.

If an a is read in the start state the machine prints a x, the tape head moves right and the TM proceeds to state q_2. In state q_2 the TM iterates, moving the tape head right through the remainder of as on the tape to the left of the separator c until the separator, c, is encountered.

Reading a c, the TM proceeds to state q_3 where it iterates (moving the tape head right) through all the as to the right of the separator until the tape is empty and thus Δ is encountered. The machines then proceeds to state q_4, moving the tape head to the left to read the most right a (state q_4) on the tape, printing a Δ, proceeding to state q_5 where it iterates, moving the tape head left, through the remainder of as to the right of the separator c on the tape.

When the separator c is read on the tape the machine proceeds to state q_6 where it iterates through the first group of as moving the tape head to the left until an x is read.

In that case the tape head moves right to read either an a or a c (state q_7).

If a `c` is read in state q_7, then all the `as` to the left of the separator have been changed to `xs`. The TM proceeds to state q_8 where it checks if Δs have been printed for all `as` to the right of the separator `c` too. If that is indeed the case a Δ is read and the TM proceeds to the halt state indicating that the number of `as` to the left of the separator is equal to the number of `as` to the right of the separator.

If an `a` is read in state q_7 the machine prints an `x` and the described loop ($q_2 - q_3 - q_4 - q_5 - q_6 - q_7$) is executed until a `c` is read in state q_7 or the machine crashes.

A.2 A CA$_{T0}$ halting on $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ when $x = y$

Figure A.2 presents a CA$_{T0}$ that also halts on input $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ if $x = y$. Clone states q_{1A} and q_{1B} replace state q_1. Each refer to the counter d_1 of state q_1. Note that the outgoing edges of these cloned states differ.

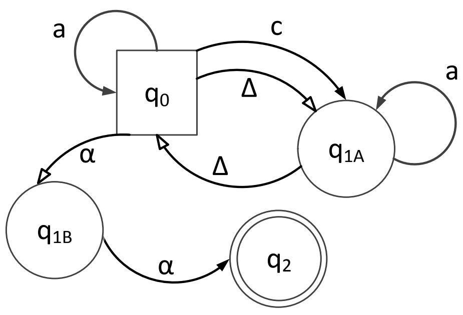


Figure A.2: A CA$_{T0}$ halting in a final state on $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ where $x = y$.

In state q_0 the number of `a`'s belonging to the first group of `a`'s is counted. When the `c` (separator) is read, we move to state q_{1A} and the second group of `a`'s is counted.

Once all the `a`'s have been read, the first Δ is read while we are still in state q_{1A}. At that point, we move from state q_{1A} to state q_0. We note that the transition arrow head of the transition labelled Δ is clear thus d_0 (the counter associated with q_0) is decremented on entrance.

The next Δ is read while we are in state q_0. The machine moves again to state q_{1A} decrementing d_1. These transitions continue, traversing through Δ's looping from q_{1A} to q_0, until $d_0 = 0$.

Because d_0 can no longer be decremented, a zero signal (α) is triggered. The machine branches to state q_{1B}, decrementing counter d_1 which triggers another zero signal α, and the machine reaches its final state q_2.

Thus the final state, q_2 is only reached when number of a's in the first group is equal to the number of a's in the second group.

A.3 A TM halting on $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ when $x < y$

The TM presented in Figure A.3 halts on the input $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ where $x < y$.

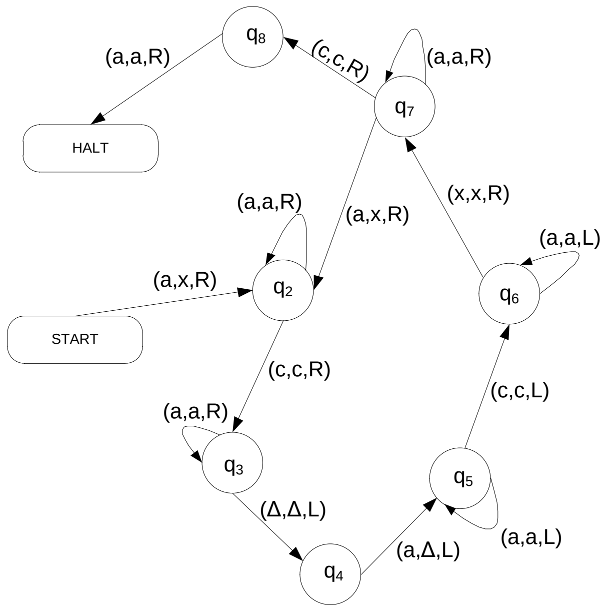
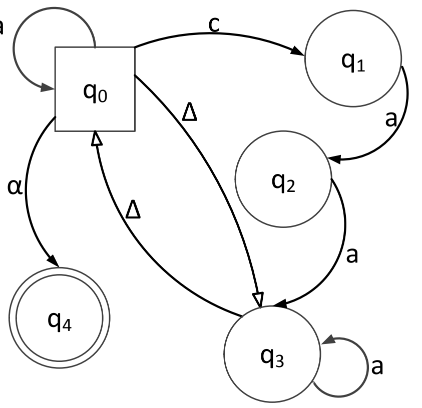


Figure A.3: A TM halting on $\mathtt{a}^{\mathtt{x}}\mathtt{c}\mathtt{a}^{\mathtt{y}}$ where $x < y$.

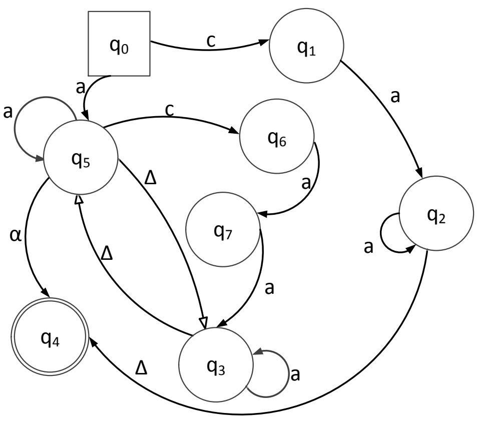
The same principles used during the development of the TM depicted in Figure A.1 were used. In state q_8 however an a should be read to reach the HALT state instead of a Δ. Thereby the TM verifies that there is at least one more a in the second group of as than in the first. The TM presented in Figure A.3 will not be discussed in detail.

A.4 A CA$_{T0}$ halting on $\mathtt{a^x c a^y}$ when $x < y$

Figure A.4(a) shows a CA$_{T0}$ that halts on $\mathtt{a^x c a^y}$ as input, provided that $0 < x < y$. Note that this automaton does not handle the case when $x = 0$. Figure A.4(a) extends this to the case where $x \geq 0$.



(a) Requires that $x > 0$



(b) Requires that $x \geq 0$

Figure A.4: CA$_{T0}$s that halting in a final state on input $\mathtt{a^x c a^y}$ and $x < y$

In Figure A.4(a), the first group of a's is counted by incrementing d_0, the counter of q_0. (Note the coloured arrow head entering q_0.) After the first group of a's have been read, the c that separates the two groups of a's is read. We then progress to state q_1 where we read the first a of the second group of a's to progress to state q_2.

To ensure that there is at least one a more in the second group than there is in the first group we read the second a of the second group of a's and move to state q_3. In state q_3 counter d_3 is incremented for each remaining a read (the loop-transition with the coloured arrow head labelled with a) until Δ is read. Δ indicates the end of the input. Next we traverse between q_0 and q_3 by reading Δ from the infinite tape. If we are in state q_3 and we read a Δ we progress to state q_1 along a clear transition arrow head. Thus the counter d_0 is decremented. From q_0 we read another Δ and move to state q_3 where counter d_3 is decremented. Note that the transition labelled with Δ also has an clear transition arrow head. This process continues until $d_0 = 0$. If $d_0 = 0$ and d_0 is entered again α is triggered which leads to the accept state, state q_4.

This automaton can be extended to cater for $x = 0$ too (i.e. the first sequence of a's is empty) as shown in Figure A.4(b). Two additional states, labelled q_1 and

q_2 in Figure A.4(b) are required. At state q_0 an edge labelled with c should be added. This edge leads to q_1. In state q_1 at least one a should be read. Thus we have an outgoing edge from q_1 labelled with an a and leading to state q_2 where any number of a's can be read. Eventually Δ is reached and the final state, q_4, is entered. The decrementing functionality is not used if $x = 0$.

A.5 A CA$_{T0}$ halting on a$^{\mathrm{x}}$ca$^{\mathrm{y}}$ when $x > y$

Figure A.5 depicts a CA$_{T0}$ that accepts binary words of the form $a^x ca^y$ if $x > y$.

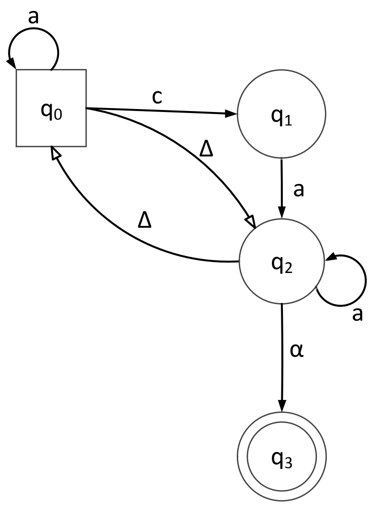


Figure A.5: A CA$_{T0}$ halting in a final state on input a$^{\mathrm{x}}$ca$^{\mathrm{y}}$ and $x > y$

By reading the first group of a's and incrementing d_0 every time an a is read state q_0 keeps track of the number of a's stored in the left operand in Figure A.5. When the separator, c is read the machine proceeds to state q_1 where the first a of the second group of a's are read and there is proceeded to state q_2. State q_2 keeps track of the number of a's in the second group of a's. When Δ is encountered the machine starts to loop between state q_0 and state q_2 where d_0 and d_2 are respectively decremented upon transition entrance. The pre mentioned loop is executed until d_2 is entered while it is zero. If that happens the zero indicator α is triggered. The machine proceeds to the halt state. Counter d_0 is decremented first thus the machine presented in Figure A.5 will only halt if there is at least 1 a more in the first group of a's than in the second group of a's.

A.6 Conclusion: remarks about CA$_{T0}$s

CA$_{T0}$s as defined in Definition A.0.1 and presented in Figures A.2, A.4(a), A.4(b) and A.5 have Moore machine characteristics in the sense that incrementation and decrementation are done in the states. Mealy machine characteristics can also be ascribed to CA$_{T0}$s in the sense that the transitions joining the different states carry information indicating whether addition or subtraction should be done in a particular state.

Note that counters in the shown examples are often incremented redundantly in that their incrementation does not contribute towards the decision about whether a specific string should be accepted or not. These transitions can probably be replaced by neutral transitions where neither incrementing nor decrementing take place. In the interest of simplicity and consistency, neutral transitions will not be considered.

B

pCA$_{T3}$ Transition Tables

This appendix supports Chapter 7 where both $\texttt{FireSat}_2$ and $\texttt{FireSat}_{2'}$ are presented. $\texttt{FireSat}_2$ implements cascaded mono-, di-, tri- and quad-pCA$_{T3}$s to construct an NCA$_{T3}(\rho)$ where $\rho = \upsilon_1, \upsilon_2 \cdots \upsilon_n$ and $1 \leq |\upsilon| \leq 4$.

The different pCA$_{T3}$s are constructed by adding for a specific $|\upsilon_i|$ the mismatch pCA$_{T3}(\upsilon_i)$ to the deletion pCA$_{T3}(\upsilon_i)$. The resultant pCA$_{T3}$ is thereafter added to the insertion pCA$_{T3}(\upsilon_i)$. The di-pCA$_{T3}$ ($|\upsilon| = 2$) is for example constructed by adding the mismatch di-pCA$_{T3}$ to the deletion di-pCA$_{T3}$. The resultant di-pCA$_{T3}$ is thereafter added to the insertion di-pCA$_{T3}$. The construction of the di-pCA$_{T3}$, catering for all the mutation types, is illustrated in Subsection 7.1.3. Note that cascading states are indicated with a $^\bullet$ *only* in the first column of the corresponding transition table.

Before pCA$_{T3}(\upsilon_i)$, catering for all the mutation types can be computed, the *mismatch* pCA$_{T3}(\upsilon_i)$, *deletion* pCA$_{T3}(\upsilon_i)$ and *insertion* pCA$_{T3}(\upsilon_i)$ should be available. Graphical depictions of these pCA$_{T3}$s are given in Subsections 7.1.1 (quad-pCA$_{T3}$s) and 7.1.2 (tri-, di- and mono-pCA$_{T3}$s). By referring to these machines and applying the algorithm inside *Rule 2 of Part 3* of Kleene's Theorem[1] the tables included below were compiled. Graphical illustrations of computed transition graphs are in some cases also provided. Cascading states are indicated

[1] *Rule 2 of Part 3* of Kleene's theorem states that if there is an FA called FA_1 that accepts the language defined by the regex r_1 and there is an FA called FA_2 that accepts the language defined by the regex r_2, then there is an FA, say FA_3, that accepts the language defined by the regex $r_1 + r_2$.

with dotted circles. Note that the counter updating and associated pseudo states, indicated in Chapter 7, are not included here.

B.1 Summing mono pCA$_{T3}$s over all mutation types

Note that the same steps outlined in Section 7.1.3 have been followed. The steps are not repeated in this Appendix in detail. A $^\bullet$ indicates a cascading state.

States \ Alpha	a	c	g	t
$R_0 = P_0$ or $D_0^\bullet$	P_1 or $D_1 = R_1$	$\perp$	$\perp$	$\perp$
$R_1^\bullet = P_1^\bullet$ or $D_1^\bullet$	$\perp$	$\perp$	$\perp$	$\perp$

Table B.1: A transition table of a mono-pCA$_{T3}$ catering for a perfect match and a deletion.

States \ Alpha	a	c	g	t
$Z_0 = M_0$ or I_0	I_1 or $M_1 = Z_1$	I_2 or $M_2 = Z_2$	I_2 or $M_2 = Z_2$	I_2 or $M_2 = Z_2$
$Z_1^\bullet = I_1^\bullet$ or $M_1^\bullet$	$\perp$	$\perp$	$\perp$	$\perp$
$Z_2 = I_2$ or M_2	I_1 or $\perp = Z_3$	$\perp$	$\perp$	$\perp$
$Z_3^\bullet = I_1^\bullet$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table B.2: A transition table, displaying a mono-pCA$_{T3}$ catering for a mismatch and an insertion.

Finally, the combined mono perfect / mono deletion pCA$_{T3}$ presented in Table B.1 could be added to the sum of the mismatch and insertion pCA$_{T3}$ presented in Table B.2. The resulting transition table is shown in Table B.3. The corresponding transition graph is shown in Figure B.1.

States \ Alpha	a	c	g	t
$Q_0^\bullet = Z_0$ or $R_0^\bullet$	Z_1 or $R_1 = Q_1$	Z_2 or $\perp = Q_2$	Z_2 or $\perp = Q_2$	Z_2 or $\perp = Q_2$
$Q_1^\bullet = Z_1^\bullet$ or $R_1^\bullet$	$\perp$	$\perp$	$\perp$	$\perp$
$Q_2^\bullet = Z_2^\bullet$ or $\perp$	Z_3 or $\perp = Q_3$	$\perp$	$\perp$	$\perp$
$Q_3^\bullet = Z_3^\bullet$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table B.3: Transition table for a mono-pCA$_{T3}$ where $0 \leq e \leq 1$. .

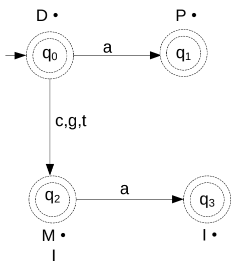


Figure B.1: A mono-pCA$_{T3}$ for $\rho = \mathtt{a}$

B.2 Summing tri-pCA$_{T3}$s over all mutation types

The same principles described for mono-pCA$_{T3}$s and di-pCA$_{T3}$s are employed to construct a tri-pCA$_{T3}$ that caters for all mutation types. To obtain Table B.4 the mismatch machine and deletions machine (both presented in Figure 7.5) are added together. The resultant machine, is added to the insertion tri-pCA$_{T3}$, provided in Figure B.2. Table B.5 is obtained. Table B.5 defines a tri-pCA$_{T3}$ catering for perfect matches, mismatches, deletions and insertions. Note that Table B.5 has a total number of 30 states.

States \ Alpha	`a`	`c`	`g`	`t`
$Z_0^\bullet = M_0$ or $D_0^\bullet$	M_1 or $D_1 = Z_1$	M_4 or $D_4 = Z_2$	M_4 or $D_5 = Z_3$	M_4 or $\perp = Z_4$
$Z_1^\bullet = M_1$ or $D_1^\bullet$	M_5 or $\perp = Z_5$	M_2 or $D_2 = Z_6$	M_5 or $D_6 = Z_7$	M_5 or $\perp = Z_5$
$Z_2^\bullet = M_4$ or $D_4^\bullet$	M_5 or $\perp = Z_5$	M_2 or $\perp = Z_8$	M_5 or $D_7 = Z_9$	M_5 or $\perp = Z_5$
$Z_3^\bullet = M_4$ or $D_5^\bullet$	M_5 or $\perp = Z_5$	M_2 or $\perp = Z_8$	M_5 or $\perp = Z_5$	M_5 or $\perp = Z_5$
$Z_4 = M_4$ or $\perp$	M_5 or $\perp$	M_2 or $\perp = Z_8$	M_5 or $\perp = Z_5$	M_5 or $\perp = Z_5$
$Z_5 = M_5$ or $\perp$	M_6 or $\perp = Z_{10}$	M_6 or $\perp = Z_{10}$	M_3 or $\perp = Z_{11}$	M_6 or $\perp = Z_{10}$
$Z_6^\bullet = M_2$ or $D_2^\bullet$	M_6 or $\perp = Z_{10}$	M_6 or $\perp = Z_{10}$	M_3 or $D_3 = Z_{12}$	M_6 or $\perp = Z_{10}$
$Z_7^\bullet = M_5$ or $D_6^\bullet$	M_6 or $\perp = Z_{10}$	M_6 or $\perp = Z_{10}$	M_3 or $\perp = Z_{11}$	M_6 or $\perp = Z_{10}$
$Z_8 = M_2$ or $\perp$	M_6 or $\perp = Z_{10}$	M_6 or $\perp = Z_{10}$	M_3 or $\perp = Z_{11}$	M_6 or $\perp = Z_{10}$
$Z_9^\bullet = M_5$ or $D_7^\bullet$	M_6 or $\perp = Z_{10}$	M_6 or $\perp = Z_{10}$	M_3 or $\perp = Z_{11}$	M_6 or $\perp = Z_{10}$
$Z_{10}^\bullet = M_6^\bullet$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$Z_{11}^\bullet = M_3^\bullet$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$Z_{12}^\bullet = M_3^\bullet$ or $D_3^\bullet$	$\perp$	$\perp$	$\perp$	$\perp$

Table B.4: Transition table: addition of deletion and mismatch tri-pCA$_{T3}$s.

States \ Alpha	a	c	g	t
$T_0^\bullet = I_0$ or $Z_0^\bullet$	I_1 or $Z_1 = T_1$	I_4 or $Z_2 = T_2$	I_4 or $Z_3 = T_3$	I_4 or $Z_4 = T_4$
$T_1^\bullet = I_1$ or $Z_1^\bullet$	I_7 or $Z_5 = T_5$	I_2 or $Z_6 = T_6$	I_7 or $Z_7 = T_7$	I_7 or $Z_5 = T_5$
$T_2^\bullet = I_4$ or $Z_2^\bullet$	I_1 or $Z_5 = T_8$	I_5 or $Z_8 = T_9$	I_5 or $Z_9 = T_{10}$	I_5 or $Z_5 = T_{11}$
$T_3^\bullet = I_4$ or $Z_3^\bullet$	I_1 or $Z_5 = T_8$	I_5 or $Z_8 = T_9$	I_5 or $Z_5 = T_{11}$	I_5 or $Z_5 = T_{11}$
$T_4 = I_4$ or Z_4	I_1 or $Z_5 = T_8$	I_5 or $Z_8 = T_9$	I_5 or $Z_5 = T_{11}$	I_5 or $Z_5 = T_{11}$
$T_5 = I_7$ or Z_5	I_8 or $Z_{10} = T_{12}$	I_2 or $Z_{10} = T_{13}$	I_8 or $Z_{11} = T_{14}$	I_8 or $Z_{10} = T_{12}$
$T_6^\bullet = I_2$ or $Z_6^\bullet$	I_{10} or $Z_{10}5 = T_{15}$	I_{10} or $Z_{10} = T_{15}$	I_3 or $Z_{12} = T_{16}$	I_{10} or $Z_{10} = T_{15}$
$T_7^\bullet = I_7$ or $Z_7^\bullet$	I_8 or $Z_{10} = T_{12}$	I_2 or $Z_{10} = T_{13}$	I_8 or $Z_{11} = T_{14}$	I_8 or $Z_{10} = T_{12}$
$T_8 = I_1$ or Z_5	I_7 or $Z_{10} = T_{17}$	I_2 or $Z_{10} = T_{13}$	I_7 or $Z_{11} = T_{18}$	I_7 or $Z_{10} = T_{17}$
$T_9 = I_5$ or Z_8	I_1 or $Z_{10} = T_{19}$	I_6 or $Z_{10} = T_{20}$	I_6 or $Z_{11} = T_{21}$	I_6 or $Z_{10} = T_{20}$
$T_{10}^\bullet = I_5$ or $Z_9^\bullet$	I_1 or $Z_{10} = T_{19}$	I_6 or $Z_{10} = T_{20}$	I_6 or $Z_{11} = T_{21}$	I_6 or $Z_{10} = T_{20}$
$T_{11} = I_5$ or Z_5	I_1 or $Z_{10} = T_{19}$	I_6 or $Z_{10} = T_{20}$	I_6 or $Z_{11} = T_{21}$	I_6 or $Z_{10} = T_{20}$
$T_{12}^\bullet = I_8$ or $Z_{10}^\bullet$	I_9 or $\perp = T_{22}$	I_2 or $\perp = T_{23}$	I_9 or $\perp = T_{22}$	I_9 or $\perp = T_{22}$
$T_{13}^\bullet = I_2$ or $Z_{10}^\bullet$	I_{10} or $\perp = T_{24}$	I_{10} or $\perp = T_{24}$	I_3 or $\perp = T_{25}$	I_{10} or $\perp = T_{24}$
$T_{14}^\bullet = I_8$ or $Z_{11}^\bullet$	I_9 or $\perp = T_{22}$	I_2 or $\perp = T_{23}$	I_9 or $\perp = T_{22}$	I_9 or $\perp = T_{22}$
$T_{15}^\bullet = I_{10}$ or $Z_{10}^\bullet$	I_{11} or $\perp = T_{26}$	I_{11} or $\perp = T_{26}$	I_3 or $\perp = T_{25}$	I_{11} or $\perp = T_{26}$
$T_{16}^\bullet = I_3^\bullet$ or $Z_{12}^\bullet$	$\perp$	$\perp$	$\perp$	$\perp$
$T_{17}^\bullet = I_7$ or $Z_{10}^\bullet$	I_8 or $\perp = T_{27}$	I_2 or $\perp = T_{23}$	I_8 or $\perp = T_{27}$	I_8 or $\perp = T_{27}$
$T_{18}^\bullet = I_7$ or $Z_{11}^\bullet$	I_8 or $\perp = T_{27}$	I_2 or $\perp = T_{23}$	I_8 or $\perp = T_{27}$	I_8 or $\perp = T_{27}$
$T_{19}^\bullet = I_1$ or $Z_{10}^\bullet$	I_7 or $\perp = T_{28}$	I_2 or $\perp = T_{23}$	I_7 or $\perp = T_{28}$	I_7 or $\perp = T_{28}$
$T_{20}^\bullet = I_6$ or $Z_{10}^\bullet$	I_1 or $\perp = T_{29}$	$\perp$	$\perp$	$\perp$
$T_{21}^\bullet = I_6$ or $Z_{11}^\bullet$	I_1 or $\perp = T_{29}$	$\perp$	$\perp$	$\perp$
$T_{22} = I_9$ or $\perp$	$\perp$	I_2 or $\perp = T_{23}$	$\perp$	$\perp$
$T_{23} = I_2$ or $\perp$	I_{10} or $\perp = T_{24}$	I_{10} or $\perp = T_{24}$	I_3 or $\perp = T_{25}$	I_{10} or $\perp = T_{24}$
$T_{24} = I_{10}$ or $\perp$	I_{11} or $\perp = T_{26}$	I_{11} or $\perp = T_{26}$	I_3 or $\perp = T_{25}$	I_{11} or $\perp = T_{26}$
$T_{25}^\bullet = I_3^\bullet$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$T_{26} = I_{11}$ or $\perp$	I_{12} or $\perp = T_{30}$	I_{12} or $\perp = T_{30}$	I_3 or $\perp = T_{25}$	I_{12} or $\perp = T_{30}$
$T_{27} = I_8$ or $\perp$	I_9 or $\perp = T_{22}$	I_2 or $\perp = T_{23}$	I_9 or $\perp = T_{22}$	I_9 or $\perp = T_{22}$
$T_{28} = I_7$ or $\perp$	I_8 or $\perp = T_{27}$	I_2 or $\perp = T_{23}$	I_8 or $\perp = T_{27}$	I_8 or $\perp = T_{27}$
$T_{29} = I_1$ or $\perp$	I_7 or $\perp = T_{28}$	I_2 or $\perp = T_{23}$	I_7 or $\perp = T_{28}$	I_7 or $\perp = T_{28}$
$T_{30} = I_{12}$ or $\perp$	$\perp$	$\perp$	I_3 or $\perp = T_{25}$	$\perp$

Table B.5: Transition table representing a tri-pCA$_{T3}$, catering for insertions, perfect matches, deletions and mismatches. This machine caters for at most 9 insertions, 3 mismatches and 3 deletions.

B.3 Summing quad-pCA$_{T3}$s over all mutation types

Transition Table B.6 presents the automaton (in tabular form) that results from the summation of the *mismatch* and *deletion* quad-pCA$_{T3}$s. The pre mentioned automata are presented in Figures 7.1 and 7.2. The insertion quad-pCA$_{T3}$ is presented in Figure 7.4. This automaton is added to the automaton presented in Table B.6 to obtain Table B.7. Note that the final table, Table B.7 defines a quad-pCA$_{T3}$ with 43 states that caters for perfect matches, mismatches, deletions and

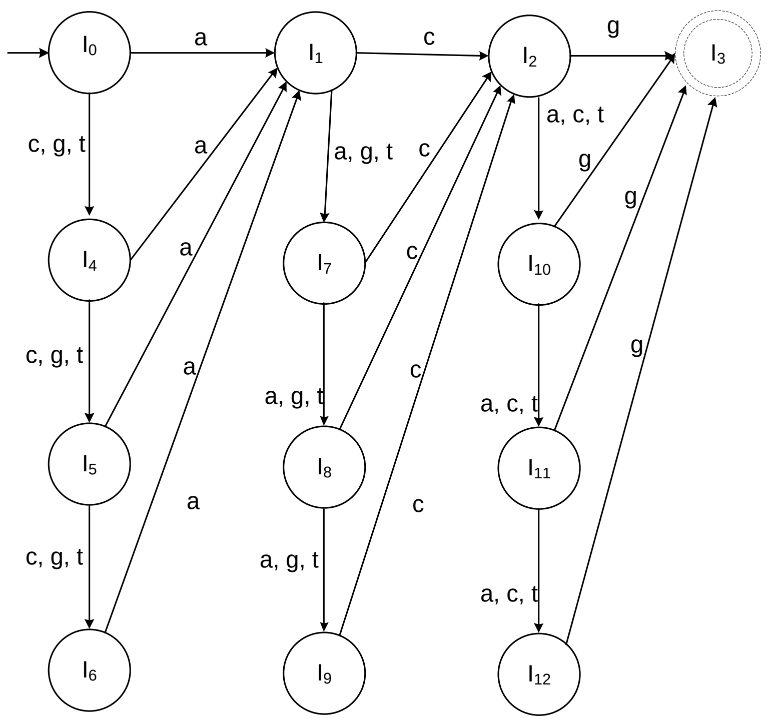


Figure B.2: An insertion tri-pCA$_{T3}$ catering for $\rho = \texttt{acg}$, allowing for at most 3 consecutive insertions and for a total of 9 insertions.

insertions. For any state ${}^{i}X^{j}$ if i $= \Delta$ then the predicate on the edge (transition) entering X must be $d_x < 4$; if $j = \bullet$ then X is a cascading state. Both of these conditions can hold at the same time.

States \ Alpha	`a`	`c`	`g`	`t`
$Z_0^\bullet = M_0$ or $D_0\bullet$	M_1 or $D_1 = Z_1$	M_5 or $D_9 = Z_2$	M_5 or $D_5 = Z_3$	M_5 or $D_4 = Z_4$
$Z_1^\bullet = M_1$ or $D_1^\bullet$	M_6 or $\perp = Z_5$	M_2 or $D_2 = Z_7$	M_6 or $D_6 = Z_8$	M_6 or $D_{14} = Z_9$
$Z_2^\bullet = M_5$ or $D_9^\bullet$	M_6 or $\perp = Z_5$	M_2 or $\perp = Z_{10}$	M_6 or $D_{12} = Z_{11}$	M_6 or $D_{15} = Z_{12}$
$Z_3^\bullet = M_5$ or $D_5^\bullet$	M_6 or $\perp = Z_5$	M_2 or $\perp = Z_{10}$	M_6 or $\perp = Z_5$	M_6 or $D_{10} = Z_{13}$
$Z_4^\bullet = M_5$ or $D_4^\bullet$	M_6 or $\perp = Z_5$	M_2 or $\perp = Z_{10}$	M_6 or $\perp = Z_5$	M_6 or $\perp = Z_5$
$Z_5 = M_6$ or $\perp$	M_7 or $\perp = Z_{14}$	M_7 or $\perp = Z_{14}$	M_3 or $\perp = Z_{17}$	M_7 or $\perp = Z_{14}$
$Z_7^\bullet = M_2$ or $D_2^\bullet$	M_7 or $\perp = Z_{14}$	M_7 or $\perp = Z_{14}$	M_3 or $D_3 = Z_{15}$	M_7 or $D_7 = Z_{16}$
$Z_8^\bullet = M_6$ or $D_6^\bullet$	M_7 or $\perp = Z_{14}$	M_7 or $\perp = Z_{14}$	M_3 or $\perp = Z_{17}$	M_7 or $D_{11} = Z_{18}$
$Z_9^\bullet = M_6$ or $D_{14}^\bullet$	M_7 or $\perp = Z_{14}$	M_7 or $\perp = Z_{14}$	M_3 or $\perp = Z_{17}$	M_7 or $\perp = Z_{14}$
$Z_{10} = M_2$ or $\perp$	M_7 or $\perp = Z_{14}$	M_7 or $\perp = Z_{14}$	M_3 or $\perp = Z_{17}$	M_7 or $\perp = Z_{14}$
$Z_{11}^\bullet = M_6$ or $D_{12}^\bullet$	M_7 or $\perp = Z_{14}$	M_7 or $\perp = Z_{14}$	M_3 or $\perp = Z_{17}$	M_7 or $D_{13} = Z_{19}$
$Z_{12}^\bullet = M_6$ or $D_{15}^\bullet$	M_7 or $\perp = Z_{14}$	M_7 or $\perp = Z_{14}$	M_3 or $\perp = Z_{17}$	M_7 or $\perp = Z_{14}$
$Z_{13}^\bullet = M_6$ or $D_{10}^\bullet$	M_7 or $\perp = Z_{14}$	M_7 or $\perp = Z_{14}$	M_3 or $\perp = Z_{17}$	M_7 or $\perp = Z_{14}$
$Z_{14} = M_7$ or $\perp$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_4 or $\perp = Z_{21}$
$Z_{15}^\bullet = M_3$ or $D_3^\bullet$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_4 or $D_8 = Z_{22}$
$Z_{16}^\bullet = M_7$ or $D_7^\bullet$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_4 or $\perp = Z_{22}$
$Z_{17} = M_3$ or $\perp$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_4 or $\perp = Z_{21}$
$Z_{18}^\bullet = M_7$ or $D_{11}^\bullet$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_4 or $\perp = Z_{21}$
$Z_{19}^\bullet = M_7$ or $D_3^\bullet$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_8 or $\perp = Z_{20}$	M_4 or $\perp = Z_{21}$
$Z_{20}^\bullet = M_8^\bullet$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$Z_{21}^\bullet = M_4^\bullet$ or $\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$Z_{22} = M_4^\bullet$ or $D_8^\bullet$	$\perp$	$\perp$	$\perp$	$\perp$

Table B.6: Quad-pCA$_{T3}$ transition table catering for υ = `acgt` with up to 4 deletions and up to 4 mismatches.

States \ Alpha	`a`	`c`	`g`	`t`
$X_0^\bullet = Z_0^\bullet$ or I_0	X_1	X_2	X_3	X_4
$X_1^\bullet = Z_1^\bullet$ or Q_1	X_5	X_6	X_7	X_8
${}^\triangle X_2^\bullet = Z_2^\bullet$ or Q_5	X_9	X_{10}	X_{11}	X_{12}
${}^\triangle X_3^\bullet = Z_3^\bullet$ or Q_5	X_9	X_{10}	X_{13}	X_{14}
${}^\triangle X_4^\bullet = Z_4^\bullet$ or Q_5	X_9	X_{10}	X_{13}	X_{13}
${}^\triangle X_5^\bullet = Z_5$ or Q_6	X_{14}	X_{15}	X_{16}	X_{14}
$X_6^\bullet = Z_7^\bullet$ or Q_2	X_{17}	X_{17}	X_{18}	X_{17}
${}^\triangle X_7^\bullet = Z_8^\bullet$ or Q_6	X_{14}	X_{15}	X_{16}	X_{18}
${}^\triangle X_8^\bullet = Z_9^\bullet$ or Q_6	X_{14}	X_{15}	X_{16}	X_{14}
$X_9^\bullet = Z_5$ or Q_1	X_{14}	X_{15}	X_{16}	X_{14}
${}^\triangle X_{10} = Z_{10}$ or Q_5	X_{19}	X_{20}	X_{21}	X_{20}
${}^\triangle X_{11}^\bullet = Z_{11}^\bullet$ or Q_5	X_{19}	X_{20}	X_{21}	X_{22}
${}^\triangle X_{12}^\bullet = Z_{12}^\bullet$ or Q_5	X_{19}	X_{20}	X_{21}	X_{20}
${}^\triangle X_{13}^\bullet = Z_5$ or Q_5	X_{19}	X_{20}	X_{21}	X_{20}
${}^\triangle X_{14}^\bullet = Z_{14}$ or Q_6	X_{23}	X_{24}	X_{23}	X_{25}
$X_{15}^\bullet = Z_{14}$ or Q_2	X_{26}	X_{26}	X_{27}	X_{28}
${}^\triangle X_{16}^\bullet = Z_{17}$ or Q_6	X_{23}	X_{24}	X_{23}	X_{29}
${}^\triangle X_{17} = Z_{14}$ or Q_7	X_{26}	X_{26}	X_{27}	X_{30}
${}^\triangle X_{18}^\bullet = Z_{18}^\bullet$ or Q_6	X_{26}	X_{26}	X_{27}	X_{30}
$X_{19} = Z_{14}$ or Q_1	X_{23}	X_{24}	X_{23}	X_{32}
${}^\triangle X_{20} = Z_{14}$ or Q_5	X_{30}	X_{31}	X_{31}	X_{33}
${}^\triangle X_{21} = Z_{17}$ or Q_5	X_{30}	X_{31}	X_{31}	X_{25}
${}^\triangle X_{22}^\bullet = Z_{19}^\bullet$ or Q_5	X_{30}	X_{31}	X_{31}	X_{25}
${}^\triangle X_{23}^\bullet = Z_{20}^\bullet$ or Q_6	X_{34}	X_{35}	X_{34}	X_{34}
$X_{24}^\bullet = Z_{20}^\bullet$ or Q_2	X_{36}	X_{36}	X_{39}	X_{36}
${}^\triangle X_{25}^\bullet = Z_{21}^\bullet$ or Q_5	X_{40}	X_{41}	X_{41}	X_{41}
${}^\triangle X_{26}^\bullet = Z_{20}^\bullet$ or Q_7	X_{37}	X_{37}	X_{39}	X_{37}
$X_{27}^\bullet = Z_{20}^\bullet$ or Q_3	X_{38}	X_{38}	X_{38}	X_{42}
${}^\triangle X_{28}^\bullet = Z_{22}^\bullet$ or Q_7	X_{37}	X_{37}	X_{39}	X_{37}
${}^\triangle X_{29}^\bullet = Z_{21}^\bullet$ or Q_6	X_{34}	X_{35}	X_{34}	X_{34}
$X_{30}^\bullet = Z_{20}^\bullet$ or Q_1	X_{34}	X_{35}	X_{34}	X_{34}
${}^\triangle X_{31}^\bullet = Z_{20}^\bullet$ or Q_5	X_{40}	X_{41}	X_{41}	X_{41}
${}^\triangle X_{32}^\bullet = Z_{22}^\bullet$ or Q_6	X_{34}	X_{35}	X_{34}	X_{34}
${}^\triangle X_{33}^\bullet = Z_{22}^\bullet$ or Q_5	X_{40}	X_{41}	X_{41}	X_{41}
${}^\triangle X_{34} = \perp$ or Q_6	X_{34}	X_{35}	X_{34}	X_{34}
$X_{35} = \perp$ or Q_2	X_{37}	X_{37}	X_{39}	X_{37}
${}^\triangle X_{37} = \perp$ or Q_7	X_{37}	X_{37}	X_{39}	X_{37}
${}^\triangle X_{38} = \perp$ or Q_8	X_{38}	X_{38}	X_{38}	X_{42}
$X_{39} = \perp$ or Q_3	X_{38}	X_{38}	X_{38}	X_{42}
$X_{40} = \perp$ or Q_1	X_{34}	X_{35}	X_{34}	X_{34}
${}^\triangle X_{41} = \perp$ or Q_5	X_{40}	X_{41}	X_{41}	X_{41}
$X_{42}^\bullet = \perp$ or $Q_4^\bullet$	$\perp$	$\perp$	$\perp$	$\perp$

Table B.7: Transition table of a quad-pCA$_{T3}$ catering for insertions, perfect matches, deletions and mismatches where $0 \leq e \leq 4$.

B.4 Conclusion

This appendix presented mono-, tri- and quad- pCA$_{T3}$s catering for perfect matches, mismatches, insertions and deletions such that $0 \leq e \leq |v|$.

C

Detection Results

This appendix explains in Section C.1 how to run the different `FireSat` versions. The different versions can be downloaded from `www.dna-algo.co.za`. In Section C.2 the detection statistics, referred to in Chapter 9, are presented. The next section, Section C.3 gives results obtained by running different versions of `FireSat`. Additional graphs, generated from the trial run results, are presented in Section C.4. Finally this appendix is concluded in Section C.5.

C.1 FireSat software

Each of the `FireSat` versions consists of some `Matlab` files.[1] In the case of `FireSat`$_3$ four `mex` files are also part of the implementation.[2] The functionality of the respective files are explained in the corresponding sections: Section C.1.2 deals with `FireSat`$_1$; Section C.1.3 with `FireSat`$_2$; Section C.1.4 with `FireSat`$_{2'}$ and Section C.1.5 with `FireSat`$_3$.

For each of the available `FireSat` folders (`data`, `firesat_1`, `firesat_2`, `firesat_2+` and `firesat_3`.) the following are included:

[1]Recall that `Octave` is the freeware counter part of `Matlab`. Note that `FireSat` can run on `Octave` too. Subsection C.1.6 gives some guidelines in this regard.

[2]The `C++` source code is available upon request.

- The files they consist of together with a brief description of the functionality of these respective files.
- Guidelines that can be followed to run the different FireSat implementations.
- An indication of the output that the various FireSat implementations generate.

Only FireSat$_3$ has been implemented in a way that a user can relatively easily manipulate its parameters. The parameters, that can be manipulated via the functional interface of FireSat$_3$, are provided in a listing in Section C.1.5.4. There is also a version of FireSat$_3$ available that prompts the user for input from the Matlab command window.

Note that FireSat$_{2'}$ has not been implemented to run as a FireSat algorithm. However, the underlying principles demonstrating how FireSat$_{2'}$ can calculate an LD or an LC have been implemented.

C.1.1 Downloading the FireSat software

To download FireSat go to www.dna-algo.co.za and click on the link Downloads. Thereafter click on the FireSat 1, 2 and 3 link.
From there click on phd_cdr_implement.zip to download FireSat. Inside the zipped folder reside five folders: data, firesat_1, firesat_2, firesat_2+ and firesat_3. The FireSat versions are discussed in different subsections below. Before FireSat is discussed a brief description of the data folder follows next.

The data folder
In the data folder reside 2 folders:

- data_mini, the folder with the data used for the trial runs and the RP analysis.
- data_pseudo_random_fasta contains additional pseudo random generated data.

Note that these two folders have a folder gen that generates random data.

C.1.2 FireSat$_1$

FireSat$_1$ resides in the firesat_1 folder.

C.1.2.1 Files to be found in the `firesat_1` folder

A list of the different files as well as a brief description of the functionality of each of the files to be found in the `firesat_1` folder follows below:

- `try_do_all_trs_11_default.m`: the implementation of $\texttt{FireSat}_1$.
- `isoctave.m` verifies whether `Octave` is being used.
- `pfa_det.m`: an implementation of the quad-PFAs introduced in Chapter 6.
- `pfa_det_inside.m`: implementations of the mono-, di- and tri-PFAs presented in Chapter 6.
- `cascaded_fa_len.m`: concatenates the PFAs that constitute ρ.

Note that the latter three items are building blocks of $\texttt{FireSat}_1$.

C.1.2.2 Running $\texttt{FireSat}_1$ in `Matlab`

To run $\texttt{FireSat}_1$ on the 20 synthetic data sets, used for all the trial runs, the steps outlined below should be followed:

- *Step 1:* open `try_do_all_trs_11_default.m` in the `Matlab` IDE.
- *Step 2:* run $\texttt{FireSat}_1$ by clicking the run button under the editor tab. Alternatively, type `try_do_all_trs_11_default` in the command window[3] of the `Matlab` IDE followed by `enter`. This should be done while the current folder is `firesat_1`. Regardless of which instructions are followed to run `try_do_all_trs_11_default.m`, `try_do_all_trs_11_default.m` should be run twice. Only after the second run, results will be displayed in the command window.
- *Step 3:* the results will now be displayed in the command window. A `mat` file is also created after the first run, with the output. However, if `try_do_all_trs_11_default.m` is being run while the `mat` file is available, `try_do_all_trs_11_default.m` will read the data from the `mat` file and will not compute results.

[3]Note that if there is referred to *command window* in this section, the command window of the `Matlab` IDE is implied.

C.1.2.3 `FireSat`$_1$: output

Output of `FireSat`$_1$ includes for each detected TR the following:

- `Motif length`;
- `Perr`;
- `TR start position` and
- `TR length`.

Furthermore, `FireSat`$_1$ statistical data, which is generated to compare all the `FireSat` versions against one another and against the identified contender algorithms, is also displayed. Percentage values of the following statistical metrics are calculated and displayed:

- `F1_score`;
- `Precision`;
- `Imprecision` and
- `Recall`.

C.1.3 `FireSat`$_2$

`FireSat`$_2$ is available from the `firesat_2` folder.

C.1.3.1 Files to be found in the `firesat_2` folder

A description of the files enclosed in the `firesat_2` folder is included below:

- `try_do_all_trs_11_default_0p8_disable_penalties.m`: is the implementation of `FireSat`$_2$.
- `build_cstr.m` and `make_state.m`: these two files together build and decorate the respective NCA$_{T3}$s.
- `nxt_plen.m`: this file determines whether the next pCA$_{T3}$, to be appended, should be a mono-pCA$_{T3}$, di-pCA$_{T3}$, tri-pCA$_{T3}$ or quad-pCA$_{T3}$.

- `tt.m`: this file enumerates the columns in the output matrix and is used for counter updating too.[4]
- `clevenN.m`: generates a matrix, using the numbers assigned in `tt.m` to keep track of the items listed below.
  - The number of mutation types (perfects, mismatches, deletions and insertions) occurring on the different paths of the $\texttt{FireSat}_2$ $\mathrm{NCA}_{T3}(\rho)$.
  - The respective current positions up to where $\texttt{FireSat}_2$ has processed the source and the destination strings for the different enumerated paths.
- `isoctave.m:` verifies whether $\texttt{FireSat}_2$ is being run from `Octave`.

C.1.3.2 Running $\texttt{FireSat}_2$

To run $\texttt{FireSat}_2$ on the relevant synthetic data[5] the same steps as for $\texttt{FireSat}_1$ should be followed. The steps are repeated here:

- *Step 1:* open `try_do_all_trs_11_default_0p8_disable_penalties.m` in the `Matlab` IDE.
- *Step 2:* run $\texttt{FireSat}_2$ by clicking on the run button under the editor tab. Alternatively while the current folder in the `Matlab` IDE is `firesat_2` type: `try_do_all_trs_11_default_0p8_disable_penalties`, followed by `enter` in the command window. Results will only be displayed in the command window after $\texttt{FireSat}_2$ has been run for a second time.
- *Step 3:* the output will be displayed in the command window. Furthermore, a `mat` file is created (after the first run) where the output is stored. Note that whenever `try_do_all_trs_11_default.m` is run in the presence of the `mat` file the output contained in the `mat` file will be displayed. Thus new computations will, in such a case, not take place.

C.1.3.3 $\texttt{FireSat}_2$: output

The output for $\texttt{FireSat}_2$ for each TR includes:

- `Motif length`;

[4]In `Octave` the described enumeration is included in `try_do_all_trs_11_default_0p8_disable_penalties.m`.

[5]Recall that for the trial runs $\texttt{FireSat}_2$ was run only on the 4 synthetic data sets where $|\rho| = 10$.

- `Perr`;
- `TR start position` and
- `TR length`.

Values of the following statistical metrics are also calculated and displayed:

- `F1-score`;
- `Precision`;
- `Imprecision` and
- `Recall`.

Recall that trial run data for $\texttt{FireSat}_2$ has only been generated for a motif length of 10.

C.1.4 $\texttt{FireSat}_{2'}$

Note that on the website $\texttt{FireSat}_{2'}$ is referred to as `firesat_2+`.

C.1.4.1 Files residing in the `firesat_2+` folder

Three folders and a file reside in the `firesat_2+` folder:

- `+plot`: provides functionality to plot graphs into .pdf files.
- `+original`: an implementation to illustrate the underlying principles of $\texttt{FireSat}_{2'}$. Note that `pleven3.m` resides in this folder too.
- `+errmax`: has the same functionality as `+original` and caters additionally for state-counting and calculating ε_{max}. Within `+errmax` resides:
  - `ntry_states.m`: Note that `ntry_states.m` calculates the *number of states* generated by a $\texttt{FireSat}_{2'}$ NCA_{T3}. It can be recalled from Chapter 7 that the number of states generated depends both on the number of motif errors to be tolerated, ε_{max}, and the length of the destination string, $|\rho|$.
  - `pleven3.m` the file that implements the theoretical underpinnings of $\texttt{FireSat}'_2$.
- `isoctave.m` verifies whether `firesat+` is being run from `Octave`.

C.1.4.2 Running `FireSat`$_{2'}$

To run the `FireSat`$_{2'}$ implementations residing in either `+original` or `+errmax` `pleven3.m` should be run. Note that `pleven3.m` resides in both +folders.

To execute `FireSat`$_{2'}$ navigate to the `firesat_2+` folder. Two options for running `FireSat`$_{2'}$ are provided. These options are discussed next:

Option 1: run `FireSat`$_{2'}$ *on the default data*

- To run `FireSat`$_{2'}$ residing in the `+original` folder. Type: `original.pleven3` in the command window, followed by `enter`.
- To run `FireSat`$_{2'}$ residing in `+errmax` type: `errmax.pleven3` in the command window, followed by `enter`.

Option 2: run `FireSat`$_{2'}$ *on user given data*

- *Run* `FireSat`$_{2'}$ *residing in* `+original`
  Navigate to the `firesat_2+` folder. To determine the LC$_n$ between `acgttgagt` and `cgtatat` type:
  `[source,destination,pmdi]=original.pleven3('acgttgagt', 'cgtatat')`
  in the command window, followed by `enter`.
- *Run* `FireSat2'` *residing in* `+errmax`
  Navigate to the `firesat_2+` folder. To determine the LC$_n$ between `acgttgagt` and `cgtatat` type:
  `[source,destination,pmdi]=errmax.pleven3('acgttgagt', 'cgtatat',10)`
  in the command window, followed by `enter`. Here 10 is the value allocated to ε_{max}. Any value can be entered for ε_{max}. However, if the number of differences between the two input strings is greater than ε_{max} an error message is displayed.

Note that `+errmax` outputs the number of states that constitutes the `FireSat`$_{2'}$ NCA$_{T3}(\rho)$, where ρ is the destination string.

C.1.4.3 FireSat$_{2'}$: output

The output of both `+original` and `+errmax` includes the following information:

- `LCnorm`: the $\mathtt{LC}_n$ is being output.
- `LD`: the LD is being output.
- `destination`: the destination string entered is being output.
- `source`: the source string entered is being output.
- `|source|`: length of the source string.
- `|destination|`: length of the destination string.
- `p`: number of perfect matches.
- `m`: number of mismatches.
- `d`: number of deletions.
- `i`: number of insertions.

C.1.5 $\mathtt{FireSat}_3$

In the `firesat_3` folder, downloadable from `www.dna-algo.co.za.`, three folders, namely `15_fsat`, `18_excel` and `+plot` reside. The latter two folders were used to generate graphs displaying trial run results. Some of these graphs are included in Chapter 9 and others in this appendix below.

To run $\mathtt{FireSat}_3$ the user should open `15_fsat`. Several methods to invoke $\mathtt{FireSat}_3$ reside in the `15_fsat` folder. These can be divided into three groups, in line with the easiness of parameter manipulation. In Section C.1.5.1 guidelines to five versions of $\mathtt{FireSat}_3$, with input parameters that should not be changed are presented. Two versions of $\mathtt{FireSat}_3$, where parameters can easily be manipulated in the preamble, are discussed in Section C.1.5.2. Section C.1.5.3 presents details of a $\mathtt{FireSat}_3$ version with a functional interface as well as a $\mathtt{FireSat}_3$ version that prompts in the command window for input.

C.1.5.1 $\mathtt{FireSat}_3$ versions to determine the best parameter combination where parameters cannot easily be manipulated

The 20 synthetic data sets, used for the trial runs against contender algorithms, were all input to the five versions of $\mathtt{FireSat}_3$ below. It was noticed that, of the enumerated implementations below, `try_do_all_trs_11.m` outperformed the other $\mathtt{FireSat}_3$ implementations.
Consequently `try_do_all_trs_11.m` was used for the trial runs against `TRF`, `mreps`

and `Phobos`. Chapter 9 reports in detail on trial run results. The parameter settings of `try_do_all_trs_11.m` are considered to be the default parameter settings of `FireSat`$_3$. The five `FireSat`$_3$ versions are:

1. `try_do_all_trs_00.m`;
2. `try_do_all_trs_01.m`;
3. `try_do_all_trs_10.m`;
4. `try_do_all_trs_11.m` and;
5. `try_do_all_trs_11_0p8.m`.

Each of the enumerated versions is catering for a specific parameter set. Apart from `try_do_all_trs_11_0p8.m` the names of the different versions of `FireSat`$_3$ correspond largely to one another. The two final characters, before the `.m` extension, differ however for each version. These two characters can be seen as a code `xy` where:

- `x` indicates the mutation precedence order. If `x = 1`, mismatches are given priority over deletions, which in turn are given priority over insertions. This is referred to as `mdi`. Otherwise `x = 0`, `mid` holds — mismatches are given priority over insertions which are in turn given priority over deletions.
- `y` indicates insertions are added in the first column in an increasing manner. Deletions are added to the first row in an increasing manner. This implies that additional insertions and deletions are added to the match pattern, described in Chapter 8. Note that this is considered to be implementation detail.

Consider `try_do_all_trs_11.m`, here `x = 1` and `y = 1`. Thus the `mdi` precedence order holds and insertions or deletions are added when calculating the LC$_n$ as explained above. The pre mentioned settings are also considered to be the default settings.

`try_do_all_trs_11_0p8`
For `try_do_all_trs_11_0p8` the `xy` parameters settings are exactly the same as that of `try_do_all_trs_11.m`. However, the `0p8` is used to indicate that the *match score threshold factor* is set equal to 0.8. Recall from Chapter 9 that the default *match score threshold factor* is 1.0. The default *match score threshold factor* was used for the 4 remaining enumerated versions above.

Running a `try_do_all_trs` `Matlab` *script*
To run the above versions the current folder should be `15_fsat`.

Ensure that the current folder is `15_fsat` then first run `setup_environ.m`. This is done by typing `setup_environ` in the command window.

Open the `Matlab` script, e.g. `try_do_all_trs_10.m`, that should be run. The steps outlined below can be followed:

- *Step 1:* open `try_do_all_trs_10.m` in the `Matlab` IDE.
- *Step 2:* run it by clicking the run button under the editor tab. After `try_do_all_trs_10.m` has been run, click the run button for a second time.
- *Step 3:* the output will now be displayed in the command window. Furthermore a `mat` file has been created, after the first run, where the results are stored. Note that running `try_do_all_trs_10.m` in the presence of the `mat` file will result in displaying the results contained in the `mat` file — new computations will in such a case not take place.

Output of `try_do_all_trs_10.m`
The output format is exactly the same for all the `try_do_all_trs_xy.m` scripts. Output is displayed as an Nx5 array where each row (N) denotes a detected TR. The information displayed for each detected TR in the respective columns is as follows:

- `Motif length;`
- `Perr;`
- `TR start position` and
- `TR length`.

Percentage values of statistical data similar to that which was generated for the trial runs, is also being output underneath the following headings:

- `F1_score`;
- `Precision` and
- `Recall`.

C.1.5.2 One implemented FireSat$_3$ version where a data set can easily be selected in the preamble

The version, `find_trs_user_input_11.m`, simplifies the selection of a data set as input for `FireSat`$_3$. The data sets to select from are the 20 data sets that were used for the trial runs against the contender algorithms reported on in Chapter 9. There are referred to these data sets as `original`. A dataset can also be selected from 20 additional pseudo random data sets, referred to as `pseudo random`.

Running `find_trs_user_input_11.m`
The `FireSat`$_3$ `Matlab` script, `find_trs_user_input_11.m` is written in a way that most of the documented parameters can easily be manipulated in the preamble.

To run `find_trs_user_input_11.m`, type `find_trs_user_input_11` in the command window and press `enter`, while `15_fsat` is the current `Matlab` folder. Follow the command prompts displayed in the command window.

Output of `find_trs_user_input_11.m`
Output is displayed as an Nx7 array where each row denotes a detected TR. The information displayed for each detected TR in the respective columns is as follows:

- `Motif length;`
- `Maximum motif error percentage;`
- `TR starting position;`
- `TR length`;
- `Number of TREs`;
- `Confidence` and;
- `Detected motif`.

C.1.5.3 FireSat$_3$: Three versions of FireSat where parameters can easily be manipulated

Three versions of `FireSat`$_3$ where parameters can relatively easily be manipulated are presented in the enumerated list below. Note that only a limited number of parameters can be manipulated when running the `Matlab` script, indicated in the first item. All the parameters of `fsat`, described in Subsection C.1.5.4, can however be manipulated by the `Matlab` scripts discussed in the final two items of the enumerated list. A description of the parameters are included. The same parameter listing can furthermore be obtained by typing `help fsat` in the command window while the current folder is `15_fsat`.

Running `fsat` *and manipulating input parameters*

Three options are available for running `fsat`. These are discussed below.

1. *Run* `fsat` on *a user given input file using the default parameter settings.*
   For example, if the user given input file is `'sss.fasta'` and TRs where $|\rho| = 50$ should be detected. In the command window, while the current folder is `15_fsat`, type `fsat('sss.fasta',50)` followed by `enter`.

2. *Run* `fsat` on *a user given input file where the user manipulates the parameters.*
   To manipulate a number of parameters when detecting TRs, values should be keyed in for all the parameters. The different parameters are separated by commas. Entries are associated with parameters in a pre-determined order as follows: `fsat('sss.fasta',`$|\rho|$`,` $\varepsilon_{max\%}$`,` $\texttt{mis}_{max\%}$`,` $\texttt{del}_{max\%}$`,` $\texttt{ins}_{max\%}$`,` $\texttt{m}_p$`,` $\texttt{d}_p$`,` $\texttt{i}_p$`,` $\sigma_{max\%}$`,` β_{min}`,`
   `match score threshold factor`).

   For example if TRs should be detected in `'sss.fasta'` where $|\rho| = 50$, $\varepsilon_{max\%} = 20$, $\text{mis}_{max\%} = 20$, $\text{del}_{max\%} = 20$, $\text{ins}_{max\%} = 20$, $\text{m}_p = 1$, $\text{d}_p = 1$, $\text{i}_p = 1$, $\sigma_{max\%} = 40$, $\beta = 2$, *match score threshold factor* = 1. In the command window type:
   `fsat(sss.fasta, 50, 20, 20, 20, 20, 1, 1, 1, 40, 2, 1)` followed by `enter`, while the current folder is `15_fsat`.

3. *Run* `fsat` on *a user given input file where the user manipulates the parameters while being prompted by* `fsat`.
   Type `prompt_user_input_fsat` in the `Matlab` command window, while `15_fsat` is the current folder. `fsat` will now start to prompt the user for input.

Output of `fsat`
Note that the output is exactly the same as discussed for `find_trs_user_input_11.m` listed above.

C.1.5.4 Input parameters of `FireSat`$_3$: `fsat`

A listing of the input parameters is included below.

```
fsat − functional interface

function [trs, mots] = fsat(sfname,mlen,motif_errperc_max,mis_max_perc,...

del_max_perc,ins_max_perc, p_mis, p_del, p_ins,...

sigma_max, beta, match_score_tfactor )

Parameters:

'sfname'− FASTA format genetic sequence to be searched for TRs.

'mlen'  − |rho| the motif length (PTRE−length) range [10−250].

'motif_errperc_max' − var−epsilon max percentage, maximum motif error percentage.

'mis_max_perc, del_max_perc, ins_max_perc' − the maximum error percentage allowed

for m,d and i respectively. These are optional parameters.

(default = 20%).

'p_mis, p_del, p_ins' − are the penalties for mismatches, deletions and

insertions. The default penalties are [1 1 1].

'sigma_max_perc' − the maximum substring error percentage allowed. sigma_max is

calulated as follows:

'(p_mis*nm + p_del*nd + p_ins*ni)*100/mlen = sigma_max_perc'

where nd, nm and ni are the number of deletions, mismatches and

insertions respectively.

sigma_max_perc is calculated over a TRE.

The range of sigma_max_perc is [0−100] (default = 40%).

'beta'  − beta−min, the minimum TREs that should  occur before a TR is valid

(default = 2).

'match_score_tfactor' − the threshold factor, range [0,1] (default 1.0) is

multiplied with a motif length dependant match score threshold function.

The value obtained is compared with the LC−norm (Levenshtein
```

`correspondence based match score) to validate a TRE detection.`

C.1.5.5 Additional files used by all the FireSat$_3$ versions

Below follows a brief description of the functionality added by the remaining files that are used by all the available FireSat$_3$ versions.

- leven_corr.mex, leven_corr.mexw64, and leven_corr.mexa64 are Matlab executables written in C++.

  These functions calculate both the LC$_n$ and the match pattern.

- setup_environ.m: has been written to copy the applicable mex files to the correct file system location.

- isoctave.m: verifies whether FireSat is being run from Octave.

- pager.m: has been added for Octave to enable a continuous output display.

C.1.6 Notes on how to run FireSat in Octave

To run FireSat from Octave, Octave and Octave-statistics should be available. Note that the Matlab and Octave input and output for FireSat is exactly the same.

- FireSat$_1$: The same guidelines as given for Matlab in Subsection C.1.2.2 should be followed to run FireSat$_1$ in Octave. An Octave IDE and an Octave command window should be used instead of a Matlab IDE and a Matlab command window.

- FireSat$_2$: Follow the Matlab guidelines in Subsection C.1.3.2 to run FireSat$_2$ in Octave. An Octave IDE and an Octave command window should be used instead of a Matlab IDE and a Matlab command window.

- FireSat$_{2'}$: +original and +errmax reside in the firesat_2+ folder. Recall that pleven3 (the file that illustrates the underlying principles of FireSat$_{2'}$) is available from both +original and +errmax. To run pleven3 from:

  - `+original`, navigate to the `+original` folder. To run `pleven3` on the default data and parameters type: `pleven3` at the command prompt. The user can enter her chosen parameters by typing:
    `[source,destination]=pleven3('acgttgagt', 'cgtatat')` at the command prompt.

  - `+errmax`, navigate to the `+errmax` folder. To run `pleven3` on the default data and parameters type: `pleven3` at the command prompt. To determine the LC$_n$ between `acgttgagt` and `cgtatat` type:
    `[source,destination,pmdi]=pleven3('acgttgagt', 'cgtatat',10)` at the command prompt.

  Subsection C.1.4.2 includes all the `firesat_2+` parameter details.

- `FireSat`$_3$: The same instructions as provided for `Matlab` in Subsection C.1.5 should be followed. An `Octave` IDE and an `Octave` command window should be used instead of a `Matlab` IDE and a `Matlab` command window.

C.2 Tables displaying detection statistics of contender algorithms

Chapter 9 described the process of generating synthetic data that was used for the trial runs reported on in this section. Recall that 20 synthetic data sets were generated where the repetitive motif lengths are 10, 25, 50, 100 and 200. For each length different data sets were generated with motif errors of 2%, 5%, 10% and 20% respectively. As the exact nucleotide positions of TRs are known it is possible to calculate the statistical results displayed below. The different subsections of this appendix display, in tabular form, the outcomes of the trial runs for the respective algorithms as follows:

- Subsection C.2.1 `Phobos`;
- Subsection C.2.2 `mreps`;
- Subsection C.2.3 `TRF` and
- Subsection C.3.3 `FireSat`.

C.2.1 Phobos

The parameter settings of `Phobos` were set as described in Subsection 9.4.1. Table C.1 presents the corresponding trial run statistics.

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	95.5	98.0	99.1	97.5	92.0
$P_{err} = 5\ \%$	95.7	96.8	97.3	97.0	93.2
$P_{err} = 10\ \%$	93.6	93.8	95.6	95.9	99.3
$P_{err} = 20\ \%$	94.2	89.3	89.9	93.4	95.0
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	95.1	95.9	82.4	95.8	85.7
$P_{err} = 5\ \%$	93.5	96.6	94.5	95.1	99.3
$P_{err} = 10\ \%$	92.0	91.1	91.8	93.7	84.9
$P_{err} = 20\ \%$	89.5	81.0	80.1	73.6	54.1
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	95.3	96.9	90.0	96.6	88.8
$P_{err} = 5\ \%$	94.6	96.7	95.9	96.1	96.1
$P_{err} = 10\ \%$	92.8	92.4	93.7	94.8	91.5
$P_{err} = 20\ \%$	91.8	85.0	84.7	82.4	69.0

Table C.1: Detection Statistics: `Phobos`

C.2.2 mreps

Subsection 9.4.2 discussed the available parameter switches for `mreps`. The settings of these switches for the trial runs were also given in the same section. Table C.2 displays the `mreps` detection statistics.

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	69.0	71.8	85.8	95.0	97.5
$P_{err} = 5\ \%$	67.5	75.4	84.4	96.5	96.2
$P_{err} = 10\ \%$	72.9	83.9	88.4	96.4	96.4
$P_{err} = 20\ \%$	77.9	87.6	86.6	94.1	undefined
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	100.0	96.2	99.8	88.2	29.3
$P_{err} = 5\ \%$	95.4	96.4	91.0	72.4	14.3
$P_{err} = 10\ \%$	94.8	87.2	70.2	55.3	14.4
$P_{err} = 20\ \%$	93.1	42.6	62.8	37.5	0
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	81.7	82.2	92.3	91.5	45.1
$P_{err} = 5\ \%$	79.0	84.7	87.6	82.7	24.9
$P_{err} = 10\ \%$	82.4	85.5	78.3	70.2	25.0
$P_{err} = 20\ \%$	84.8	57.4	72.8	53.6	undefined

Table C.2: Detection Statistics: `mreps`

C.2.3 TRF

Information about TRs detected by `TRF` with the parameter settings, indicated in Subsection 9.4.3 is included in Table C.3 below.

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	95.2	67.4	99.0	93.4	91.2
$P_{err} = 5\ \%$	94.8	96.8	97.4	92.7	99.8
$P_{err} = 10\ \%$	93.4	80.8	95.6	91.9	undefined
$P_{err} = 20\ \%$	93.3	88.2	90.2	94.0	undefined
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	86.0	99.1	99.2	99.0	78.6
$P_{err} = 5\ \%$	92.6	97.5	97.7	91.1	35.7
$P_{err} = 10\ \%$	68.0	88.4	65.8	92.7	0
$P_{err} = 20\ \%$	92.8	53.9	81.4	50.5	0
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	90.4	80.2	99.1	96.1	84.4
$P_{err} = 5\ \%$	93.7	97.1	97.6	91.9	52.6
$P_{err} = 10\ \%$	78.7	84.5	78.0	92.3	undefined
$P_{err} = 20\ \%$	93.1	66.9	85.5	65.7	undefined

Table C.3: Detection Statistics: `TRF`

C.3 FireSat

TR detections achieved by the different `FireSat` versions are reported below.

C.3.1 $\texttt{FireSat}_1$

Table C.4 reports on the detection ability of $\texttt{FireSat}_1$. Note that the *match score threshold function* and *match score threshold factor* were used. TREs were validated by $\varepsilon_{max\%}$ only. As indicated in Tables 6.1, 6.2, 6.3 and 6.4, mismatches were given priority over deletions that were given priority over insertions.

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	11.6	53.0	94.8	98.0	99.3
$P_{err} = 5\ \%$	23.3	91.9	95.6	43.6	24.6
$P_{err} = 10\ \%$	42.0	91.4	30.6	96.9	22.0
$P_{err} = 20\ \%$	78.4	15.3	53.1	96.0	18.6
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	76.6	82.7	29.1	5.28	85.3
$P_{err} = 5\ \%$	96.0	60.5	10.6	14.8	46.4
$P_{err} = 10\ \%$	84.9	34.2	18.9	74.6	21.3
$P_{err} = 20\ \%$	83.4	16.9	78.2	31.0	5.3
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	20.2	64.6	44.5	10.0	91.8
$P_{err} = 5\ \%$	37.5	72.9	19.1	22.1	32.1
$P_{err} = 10\ \%$	56.2	49.7	23.4	84.3	21.7
$P_{err} = 20\ \%$	80.8	16.0	63.3	46.9	8.28

Table C.4: Detection Statistics: $\texttt{FireSat}_1$

C.3.2 $\texttt{FireSat}_2$

Tables C.5 and C.6 report on results achieved by $\texttt{FireSat}_2$. The *match score threshold function* and the *match score threshold factor* were used. The *match* *score factor* was set to 0.8 (Table C.5) and 1.0 (Table C.6) respectively. For `FireSat`$_2$ the LC$_n$ was calculated, using the LBD, as follows:

$$LC_n(s,d) = 1 - \frac{|\rho| - LBD - Ins}{|\rho|} \tag{C.1}$$

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	97.7	undefined	undefined	undefined	undefined
$P_{err} = 5\ \%$	97.8	undefined	undefined	undefined	undefined
$P_{err} = 10\ \%$	97.6	undefined	undefined	undefined	undefined
$P_{err} = 20\ \%$	96.8	undefined	undefined	undefined	undefined
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	93.2	0	0	0	0
$P_{err} = 5\ \%$	90.3	0	0	0	0
$P_{err} = 10\ \%$	73.5	0	0	0	0
$P_{err} = 20\ \%$	87.8	0	0	0	0
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	95.4	undefined	undefined	undefined	undefined
$P_{err} = 5\ \%$	93.9	undefined	undefined	undefined	undefined
$P_{err} = 10\ \%$	83.9	undefined	undefined	undefined	undefined
$P_{err} = 20\ \%$	92.1	undefined	undefined	undefined	undefined

Table C.5: Detection Statistics: `FireSat`$_2$ — *match score threshold factor*: 0.8

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	97.3	undefined	undefined	undefined	undefined
$P_{err} = 5$ %	98.1	undefined	undefined	undefined	undefined
$P_{err} = 10$ %	97.6	undefined	undefined	undefined	undefined
$P_{err} = 20$ %	97.1	undefined	undefined	undefined	undefined
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	65.8	0	0	0	0
$P_{err} = 5$ %	72.5	0	0	0	0
$P_{err} = 10$ %	61.5	0	0	0	0
$P_{err} = 20$ %	52.3	0	0	0	0
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	78.5	undefined	undefined	undefined	undefined
$P_{err} = 5$ %	83.4	undefined	undefined	undefined	undefined
$P_{err} = 10$ %	75.5	undefined	undefined	undefined	undefined
$P_{err} = 20$ %	68.0	undefined	undefined	undefined	undefined

Table C.6: Detection Statistics: `FireSat`$_2$ — *match score threshold factor*: 1.0

C.3.3 `FireSat`$_3$

In this subsection Table C.7 displays detection statistics when `FireSat`$_3$ is run with a *match score threshold factor* set to 0.8. Thereafter Table C.8 presents detection statistics with a *match score threshold factor* setting of 1.0. Note that in both pre mentioned cases mismatches are given priority over deletions, which are in turn given priority over insertions. Finally in Table C.9 detection statistics achieved by `FireSat`$_3$ where mismatches are given priority over insertions and insertions are given priority over deletions are displayed. Note that the *match score threshold factor* setting was 1.0.

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	74.1	99.5	99.6	99.0	99.3
$P_{err} = 5\ \%$	76.1	96.8	97.5	98.1	98.0
$P_{err} = 10\ \%$	71.8	95.5	95.8	97.3	97.3
$P_{err} = 20\ \%$	28.2	42.2	87.6	93.2	93.3
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	85.4	81.2	93.4	95.8	92.2
$P_{err} = 5\ \%$	92.5	96.6	89.9	96.5	98.0
$P_{err} = 10\ \%$	88.8	93.5	93.9	86.4	90.4
$P_{err} = 20\ \%$	95.2	89.3	89.9	82.7	93.2
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	79.3	89.4	96.4	97.4	95.6
$P_{err} = 5\ \%$	83.5	96.7	93.6	97.3	98.0
$P_{err} = 10\ \%$	79.4	94.5	94.8	91.5	93.7
$P_{err} = 20\ \%$	43.5	57.3	88.7	87.6	93.3

Table C.7: `FireSat`$_3$ detection statistics when the match score threshold factor setting is 0.8.

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	96.8	99.5	99.6	99.0	99.3
$P_{err} = 5$ %	98.6	96.8	97.5	98.1	98.0
$P_{err} = 10$ %	96.7	95.5	95.8	97.3	97.3
$P_{err} = 20$ %	96.3	88.3	90.3	92.2	93.3
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	78.8	81.2	93.4	95.8	92.2
$P_{err} = 5$ %	89.5	96.6	89.9	96.5	98.0
$P_{err} = 10$ %	64.0	93.5	93.9	86.4	90.4
$P_{err} = 20$ %	68.2	87.7	89.9	84.9	93.2
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	86.9	89.4	96.4	97.4	95.6
$P_{err} = 5$ %	93.9	96.7	93.6	97.3	98.0
$P_{err} = 10$ %	77.0	94.5	94.8	91.5	93.7
$P_{err} = 20$ %	79.8	88.0	90.1	88.4	93.3

Table C.8: `FireSat`$_3$ detection statistics when the *match score threshold factor* is set to 1.0.

Precision	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	96.8	99.5	99.6	99.0	99.3
$P_{err} = 5\ \%$	98.6	97.3	98.2	98.4	97.3
$P_{err} = 10\ \%$	96.7	95.4	96.1	97.0	97.6
$P_{err} = 20\ \%$	96.3	93.3	90.6	92.4	93.1
Recall	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	78.8	81.2	93.4	95.9	92.2
$P_{err} = 5\ \%$	88.0	96.9	90.5	96.7	97.3
$P_{err} = 10\ \%$	64.0	93.4	94.2	86.1	90.6
$P_{err} = 20\ \%$	68.2	82.7	90.2	85.0	93.0
$F1_{score}$	$mlen = 10$	$mlen = 25$	$mlen = 50$	$mlen = 100$	$mlen = 200$
$P_{err} = 2\%$	86.9	89.4	96.4	97.4	95.6
$P_{err} = 5\ \%$	93.0	97.1	94.2	97.5	96.5
$P_{err} = 10\ \%$	77.0	94.4	95.1	91.2	94.0
$P_{err} = 20\ \%$	79.8	87.7	90.4	88.5	93.0

Table C.9: `FireSat`$_3$ detection statistics where mismatches have priority over insertions and insertions have priority over deletions. The *match score threshold factor* is set to 1.0.

C.4 Additional Graphs

Additional graphs, not included in Chapter 9 are presented in this section. The results reported here correspond to the findings reported in Section 9.5.

Figures C.1, C.2, C.3 and C.4 visualise the relative performance of `mreps`, `TRF`, `Phobos` and `FireSat`$_3$, both in the case of where the match score threshold factor $= 0.8$ throughout (denoted in the figure by `FireSat`$_{3(0.8)}$), and in the case where it is 1.0 (denoted in the figure by `FireSat`$_{3(1.0)}$). These figures summarise relative algorithm performance over *all* motif lengths for each of the four P_{err} values used. Figure C.1 visualise precision and recall where $P_{err} = 2\%$. Figures C.2, C.3 and C.4 visualise the same for $Perr = 5\%$, $Perr = 10\%$ and $Perr = 20\%$ respectively. The recall performance is shown on the left of these figures and the precision on the right.

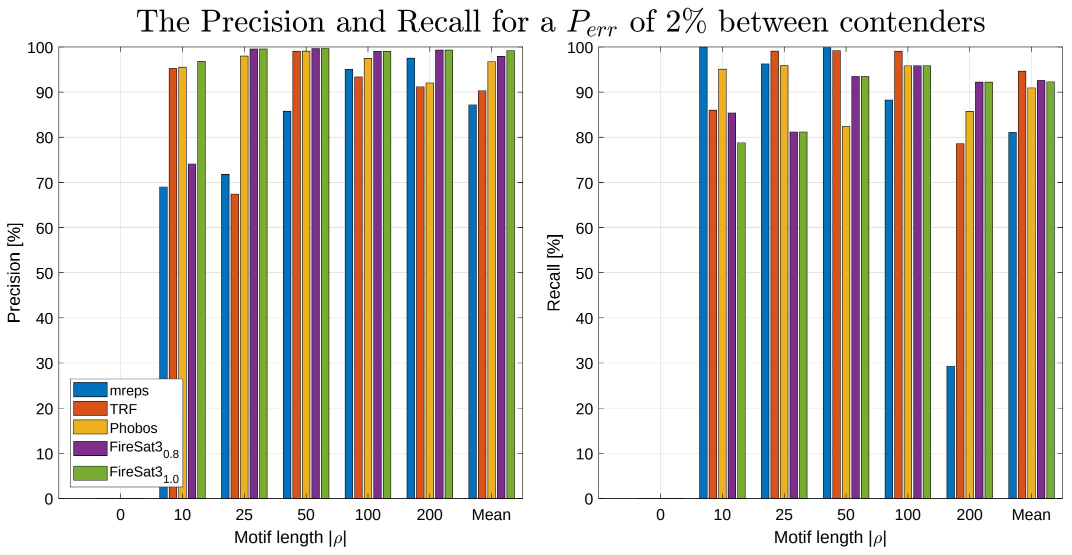


Figure C.1: Precision (on the left) and recall (on the right) where $Perr = 2\%$.

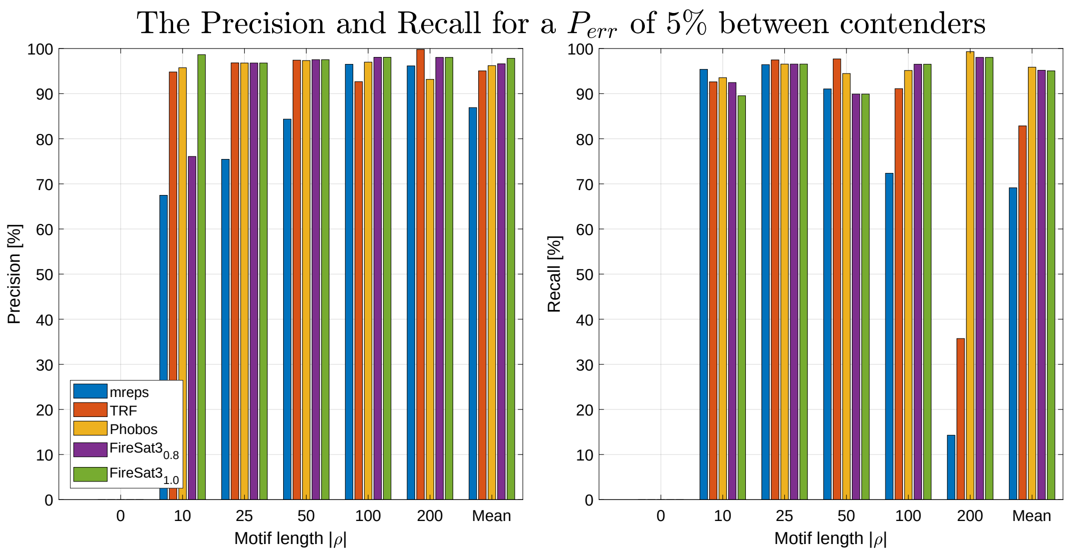


Figure C.2: Precision (on the left) and recall (on the right) where $Perr = 5\%$.

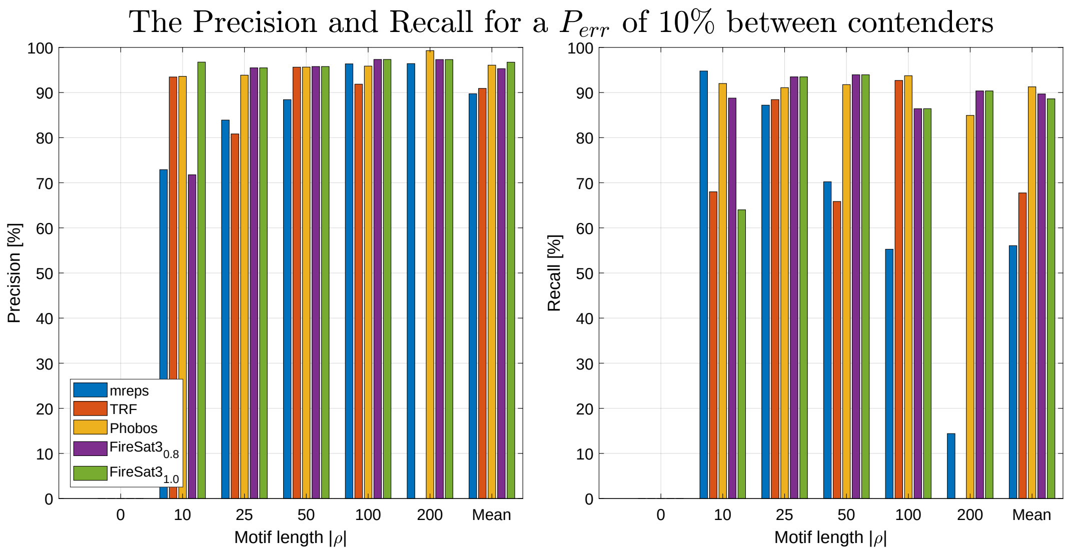


Figure C.3: Precision and (on the left) and recall (on the right) where $Perr = 10\%$.

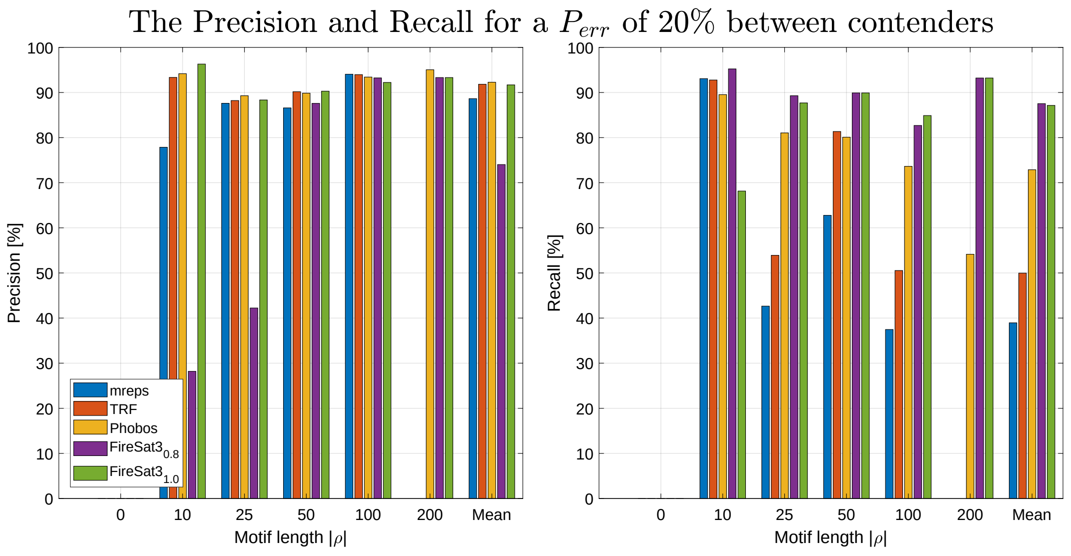


Figure C.4: Precision (on the left) and recall (on the right) where $Perr = 20\%$.

C.5 Conclusion

This appendix provided guidelines explaining how to run `FireSat` in its first section. The remainder of this appendix elaborated on the outcome of trial run results. In Section C.2 detection statistics achieved by the identified contender algorithms were presented. The detection statistics achieved by `FireSat` were included in Section C.3. Finally Section C.4 provided graphs not included in Chapter 9.

References

C. Abajian. Sputnik: Source code published online. Online: http://espressosoftware.com/sputnik/sputnik_source.html, 1994.

C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: A new structure for pattern matching. *SOFSEM, Theory and Practice of Informatics, Lecture Notes in Computer Science*, 1725:295 – 310, 1999.

American Heritage Dictionary. The American Heritage Science Dictionary. Houghton Mifflin Harcourt Publishing Company, 2010.

T. Andrews. Computation Time Comparison Between Matlab and C++ Using Launch Windows. 2012.

M. Anisimova, J. Pečerska, and E. Schaper. Statistical approaches to detecting and analyzing tandem repeats in genomic sequences. *Frontiers in Bioengineering and Biotechnology*, 3(2015):31, 2015. URL http://doi.org/10.3389/fbioe.2015.00031.

T. Anwar and A. U. Khan. SSRscanner: a program for reporting distribution and exact location of simple sequence repeats. *Bioinformation*, 1(3):89–91, 2006.

A. Archambault. Mining genomic data for tandem repeats. Online: http://qcbs.ca/wiki/bioinformatic_tools_to_detect_microsatellites_loci_from_genomic_data, 2012.

A. Arnove. Chomsky.Info. Online: http://www.chomsky.info/books.htm, 2018.

R. Bailey. The cell-cell structure. Online: http://biology.about.com/od/cellanatomy/a/eukaryprokarycells.htm, 2012.

J. Becker, M. Platzner, and S. Vernalde. Large Scale Protein Sequence Alignment using FPGA Reprogrammable Logic Devices. *Field Programmable Logic and Application*, 3203:23—32, 2004.

G. Benson. Tandem Repeats Finder. *Nucleic Acids Research*, 27(2):573 – 580, November 1999.

G. Benson. Tandem Repeats Finder: Definitions: FASTA Format. Online: http://tandem.bu.edu/trf/trf.definitions.html, Sept 2003a.

G. Benson. Tandem Repeats Finder:Unix Version help: using Tandem Repeats Finder for Unix. Online: file://D:/Tandem%20Repeats%20Finder%20%20Unix%20Version%20Help.htm, Sept 2003b.

B. P. Bergeron. *Bioinformatics Computing*. Prentice-Hall/ Professional Technical Reference: Upper Saddle River, NJ, 2003.

A. Biegert and J. Schoöding. De novo identification of highly diverged protein repeats by probabilistic consistency. *Bioinformatics*, 24(6), 2008.

V. Boeva, M. Regnier, D. Papatsenko, and V. Makeev. Short fuzzy tandem repeats in genomic sequences, identification, and possible role in regulation of gene expression. *Bioinformatics*, 22(6):676–684, 2006.

M. Cadilhac, A. Finkel, and P. McKenzie. Bounded Parikh automata . *International Journal of Foundations of Computer Science*, 23(8), 2012.

A. T. Castelo, W. Martins, and G. R. Gao. TROLL: Tandem Repeat Occurrence Locator. *Bioinformatics Applications Note*, 18(4):634–636, 2002.

S. Chakraborty, L. T. X. Phan, and P. S. Thiagarajan. Event count automata: A state-based model for stream processing systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2490-7. doi: 10.1109/RTSS.2005.21. URL http://dx.doi.org/10.1109/RTSS.2005.21.

N. Chomsky. Three models for the description of language. *I.R.E. Transactions of Information Theory*, 2:113–124, September 1956.

N. Chomsky. *Aspects of the Theory of Syntax*. Cambridge, Massachusetts: MIT Press, 1965.

N. Chomsky. *Cartesian Linguistics: A Chapter in the History of Rationalist Thought. Third edition.* Cambridge University Press, 1966.

C. E. Cleland. The concept of computability. *Theoretical Computer Science*, 317:209–225, 2004.

D. I. A. Cohen. *Introduction to Computer Theory*. Wiley: New York, 2nd ed. edition, 1997.

G. M. Cooper. *The Cell: A Molecular Approach*. Sinauer Associates, Inc., 2nd ed. edition, 2000.

F.H. Crick, L. Barnett, and R.J. Watts-Tobin. General nature of the genetic code for proteins. *Nature*, 192:1227—1237, 1961.

O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, volume 8. Academic Press, 1972. ISBN 0122005503.

M. D. Davis, R. Sigal, and E. J. Weyuker. *Computability, Complexity, and Languages*. Fundamentals of Theoretical Computer Science. Academic Press Professional, Inc., San Diego, CA, USA, 2nd edition, 1994. ISBN 0-12-206382-1.

C. De Ridder. Flexible Finite automata-based algorithms for detecting microsatellites in DNA. Master's thesis, Department of Computer Science, University of Pretoria, July 2010.

C. De Ridder, D. G. Kourie, and B. W. Watson. FireμSat: an algorithm to detect microsatellites in DNA. In *Proceedings of the Prague Stringology Conference*, pages 137–150, August 2006a. ISBN 80-01-03533-6.

C. De Ridder, D. G. Kourie, and B. W. Watson. FireμSat: meeting the challenge of detecting microsatellites in DNA. In *Proceedings of the 2006 annual research conference of SAICSIT*, pages 247–256, 2006b. ISBN 1-59593-567-3.

C. De Ridder, P. V. Reyneke, B. W. Watson, O. Reva, and D. G. Kourie. Cascading Finite Automata for minisatellite detection. In *The 22nd Annual Symposium of the Pattern Recognition Association of South Africa*, pages 31–36, November 2011.

C. De Ridder, D.G. Kourie, B.W. Watson, T.R. Fourie, and P.V. Reyneke. Fine-tuning the search for microsatellites. *Journal of Discrete algorithms*, 20:21–37, 2013.

O. Delgrange and E. Rivals. STAR: an algorithm to search for tandem approximate repeats. *Bioinformatics*, 20(16): 2812–2820, June 2004.

H.C. Doran and P.B. Van Wamelen. Application of the Levenshtein Distance Metric for the Construction of Longitudinal Data Files. *Educational Measurements: Issues and Practice*, 29(2):13—23, 2010.

L. Du, H. Zhou, and H. Yan. OMWSA:detection of DNA repeats using moving window spectral analysis. *Bioinformatics Applications Note*, 23(5):631 – 633, 2007.

S. Easterbrook, J. Singer, M-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. *Guide to Advanced Empirical Software Engineering*, 2007.

B.C. Faircloth. msatcommander: detection of microsatellite repeat arrays and automated, locus-specific primer design. *Molecular Ecology Resources*, 8(1):92–94, 2008.

H. Fan and J-Y. Chu. A brief review of short tandem repeat mutation. *Genomic Proteomics Bioinformatics*, 5(1): 7—14, 2007.

T. Fawket. ROC Graphs: Notes and Practical Considerations for Data Mining Researchers. Online: http://www.hpl.hp.com/techreports/2003/HPL-2003-4.pdf, 2003.

P. Fearnhead. SequenceLDhot: detecting recombination hotspots. *Bioinformatics*, 22(24):3061–3066, 2006.

R. L. Glas, V. Ramesh, and I. Vessey. An analysis of research in Computing disciplines. *Communications of the ACM*, 47(6), June 2004.

P. Goubil-Gambrell. What do Practitioners need to know about Research Methodology? *Directions in Technical communications research*, 1991.

M. Guerra. The Principles of FPGAs. http://www.electronicdesign.com/fpgas/principles-fpgas, 2016.

X. Guo, H. Wang, and V. Devabhaktuni. A Systolic Array-Based FPGA Parallel Architecture for the BLAST Algorithm. *ISRN Bioinformatics*, 2012:11, 2012. doi: 10.5402/2012/195658.

D. Harel and Y. Feldman. *Algorithmics: The Spirit of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2004. ISBN 0-321-11784-0.

A.M. Hauth and D.A. Joseph. Beyond tandem repeats: Complex pattern structures and distant regions of similarity. *Bioinformatics*, 18(1):S31 – S37, July 2002.

S. Hjelmqvist. Fast, memory efficient Levenshtein algorithm. Online: http://www.codeproject.com/Articles/13525/Fast-memory-efficient-Levenshtein-algorithm, 2012.

D. Hoang and D. Lopresti. FPGA implementation of systolic sequence alignment. *Field-Programmable Gate Arrays: Architecture and Tools for Rapid Prototyping*, pages 183–191, 1993.

J. Holub. *Finite Automata in pattern matching*. John Wiley & Sons inc., 1st edition, 2010. ISBN 978-0-470-50519-9.

J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory languages and computation*. Addison-Wesley publishing company, 1979.

D. Hovland. Regular expressions with numerical constraints and automata with counters. *Theoretical Aspects of Computing - ICTAC 2009*, 5684:231–245, 2009a.

D. Hovland. Regular expressions with unordered concatenation and numerical constraints. *Theoretical Aspects of Computing - ICTAC 2009*, 5684:231–245, 2009b.

S. Hughes. Speech. The Pennsylvania Gazette, 2001.

K. T. Johnson, A. R. Hurson, and B. Shirazi. General-purpose systolic arrays. *Computer*, 26(11):20–31, November 1993. ISSN 0018-9162. doi: 10.1109/2.241423. URL http://dx.doi.org/10.1109/2.241423.

J. Jorda and A.V. Kajava. T-REKS: identification of Tandem Repeats in sequences with a K-means based algorithm. *Bioinformatics*, 25:2632–2638, 2009.

W. Karianto. Adding monotonic counters to automata and transition graphs. *Developments in Language Theory*, 3572:308–319, 2005.

Y. Ke and R. Sukthankar. PCA-SIFT: A More Distinctive Representation for Local Image Descriptors. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'04, pages 506–513, Washington, DC, USA, 2004. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=1896300.1896374.

A. Kempe. Reduction of intermediate alphabets in finite-state transducer cascades. In *TALN 2000 : 7e confrence annuelle sur le Traitement Automatique des Langues Naturelles*, pages 207—215, 2000. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.2744.

A. Kent, J.G. Williams, C. M. Hall, and R. Kent. *Encyclopedia of Computer Science and Technology*, volume 25 no 10, page 91. Marcel Dekker, Inc; New York and Basel, 1992.

M.W. King. The medical biochemistry page. Online: http://themedicalbiochemistrypage.org, February 2012.

D. Koch, F. Hannig, and D. Ziener. *FPGAs for Software Programmers*. Springer Verlag, 1 edition, 2016.

R. Kofler and C. Sclötterer. Sciroko applications note. *Bioinformatics*, 23(13):1683–1685, 2007.

R. Kolpakov, G. Bana, and G. Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *PubMed Central Nucleic Acid Research* , 31(13):3672–3678, 2003.

D.G. Kourie and B.W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, Berlin Heidelberg, first edition, 2012.

L. Krauss and S.M. Carroll. Science in the Dock, Discussion with Noam Chomsky. Chomsky.info. 2006-03-01, 03 2006.

K. Kressel. Practice-Relevant Research in Mediation: Toward a Reflective Research Paradigm. *Negotiation Journal*, 13(2):143–160, 1997.

A. Krishan and F. Tang. Exhaustive whole-genome tandem repeats search. *Bioinformatics*, 20(16):2702–2710, 2004.

S. Kurtz. Approximate string searching under weighted edit distance. In *Proceedings of the Third South American Workshop on String Processing*, Ottawa, Canada, 1996. Carlton University Press. URL http://www.zbh.uni-hamburg.de/pubs/pdf/Kur1996.pdf.

S. Kurtz, J. Choudhuri, E. Ohlebush, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acid Research*, 29:4633–4642, 2001.

G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.

G.M. Landau and U. Vishkin. Fast String Matching with k Differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi: 10.1016/0022-0000(88)90045-1.

F. Lee. *Interspersed repeats and Tandem Repeats in the Molecular Biology Web Book*, chapter 3. www.web-books.com, 1996.

A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–326, 2003.

V.I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–708, 1966.

K. G. Lim, C. K. Kwoh, L. Y. Hsu, and A. Wirawan. Review of tandem repeat search tools: a systematic approach to evaluating algorithmic performance. *Briefings in Bioinformatics*, May 2012.

N. M. Luscombe, D. Greenbaum, and M. Gerstein. What is Bioinformatics? A proposed definition and overview of the field. *Methods of Information in Medicine*, 40:346–358, 2001.

K. Mahesh. *Theory of computation: a problem-solving approach*. John Wiley and Sons, Ltd, Publication, 1st edition, 2013. ISBN 978-1-118-54679-6(P/B).

K. T. Masombuka, C. de Ridder, and D. G. Kourie. An investigation of software for minisatellite detection. In Fred Nicolls, editor, *Proceedings on the twenty-first PRASA symposium*, pages 171–176, November 2010. ISBN 978-0-7992-2470-2.

D. Matuszek. Linear-bounded automata. Online: http://www.seas.upenn.edu/ cit596/notes/dave/chomsky2.html, 1996.

C. Mayer. Phobos: a tandem repeat search tool 2012, Phobos 3.3.11, 2006-2010. URL http://www.rub.de/ecoevo/cm/cm_phobos.htm.

McGraw-Hill and S.P. Parker. *McGraw-Hill Dictionary of Scientific and Technical Terms*. McGraw-Hill Companies, Inc, 6 edition, 2003.

E. Meglécz, C. Costedoat, V. Dubut, T. Malausa, and J.F. Martin. QDD: a user-friendly program to select microsatellite markers and design primers. *Bioinformatics*, 26(3):403 – 404, 2010.

B. Melichar. Space complexity of linear approximate string matching. In *Proceedings of the Prague Stringology Club Workshop '96*, pages 28—36, 1996. URL http://www.stringology.org/event/1996/p4.html.

K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 2, pages 257–263. IEEE, 2003. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8603.

M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, IInc. , 1 edition, 1967.

S. B. Mudunuri and H. A. Nagarajaram. IMEx: Imperfect microsatellite extractor. *Bioinformatics*, 23(10):1181–1187, 2007.

National Institute of Health of the United States of America. BISTIC Definition Committee. NIH working definition of Bioinformatics and Computational Biology. Online: http://www.bisti.nih.gov/CompBioDef.pdf, July 2000.

G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31—88, 2001.

S. B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443—453, 1970.

A. M. Newman and J.B. Cooper. Xstream: A practical algorithm for identification and architecture modeling of tandem repeats in protein sequences. *BMC Bioinformatics*, 8, 2007. doi: 10.1186/1471-2105-8-382.

J. Paces. Bioinformatics: Tools for analysis of Biological sequences. *In: Proceedings of the Prague Stringology Conference 2001*, pages 50–58, 2001.

V. Parisi, V. De Fonzo, and F. Aluffi-Pontini. STRING. *Bioinformatics* , 19(14):1733–1738, 2003.

J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1984. ISBN 0-201-05594-5.

M. Pellegrini, M.E. Renda, and A. Vecchio. TRStalker: an efficient heuristic for finding fuzzy tandem repeats. *Bioinformatics*, 26:i358–i366, 2010.

M. Pellegrini, M.E. Renda, and A. Vecchio. Tandem repeats discovery service (TReaDS) applied to finding novel cis-acting factors in repeat expansion diseases. *BMC Bioinformatics*, 13(4):S3, 2012.

D. Pesetsky. Linguistics universals and universal grammar. Online: http://web.mit.edu/linguistics/people/faculty/pesetsky/Pesetsky_MITECS_Universals_UG.pdf, 2009.

A. Pestronk. DNA repeat sequences and diseases. Online: http://www.neuro.wustl.edu/neuromuscular /mother/dnarep.htm, Sept 2005.

R. Pokrzywa and A. Polanski. BWTrs: A tool searching for tandem repeats in DNA sequences based on the Burrows-Wheeler transform . *Genomics*, 96:316 – 321, 2010.

E. Rivals. Eric Rivals's homepage. Online: http://www.lirmm.fr/ rivals/tete-en.html, 2004.

E. Rivals, J. P. Delahaye, O. Delgrange, and M. Dauchet. A first step toward chromosome analysis by compression algorithms. In *Proceedings of the First International IEEE Symposium on Intelligence in Neural and Biological Systems*, pages 233–239, 1995.

P. Robinson. The Chomsky Problem. The New York Times, 1979.

M. Safran, I. Solomon, O. Shmueli, M. Lapidot, S. Shen-Orr, A. Adato, U. Ben-Dor, N. Esterman, N. Rosen, I. Peter, T. Olender, V. Chalifa-Caspi, and D. Lancet. GeneCardsTM 2002: Towards a complete, object-oriented, human gene compendium. *Bioinformatics Applications Note*, 18(11):1542–1543, 2002.

A. Salomaa. *Computation and Automata* , volume 25 of *Encyclopedia of mathematics and its applications.* Cambridge University Press, 1985. ISBN 0 521 30245 5.

W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

E. Schaper, A. V. Kajava, A. Hauser, and M. Anisimova. Repeat or not repeat? statistical validation of tandem repeat prediction in genomic sequences. *Nucleic Acids Research*, 2012. doi: 10.1093/nar/gks726. URL http://nar.oxfordjournals.org/content/early/2012/08/24/nar.gks726.abstract.

E. Schaper, A. Korsunsky, A. Messina, R. Murri, J. Pečerska, H. Stockinger, S. Zoller, I. Xenarios, and M. Anisimova. TRAL: Tandem repeat annotation library. *Bioinformtics*, 31(18):3051–3053, 2015.

M.P. Schutzenberger. Finite counting automata. *Information and control*, 5(2), 1962.

D. Sharma, B. Issac, G.P. Raghava, and R. Ramaswamy. Spectral Repeat Finder (SRF): identification of repetitive sequences using Fourier transformation. *Bioinformatics*, 20(9):1405–1412, 2004.

M. Sipser. *Introduction to the Theory of Computation*. COURSE TECHNOLOGY, CENGAGE Learning, 2 edition, 2006. ISBN 978-0-619-21764-8.

T.F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

D. Sokol, G. Benson, and J. Tojeira. Tandem repeats over the edit distance. *Bioinformatics*, 23(2):e30–e35, 2007. doi: 10.1093/bioinformatics/btl309. URL http://bioinformatics.oxfordjournals.org/content/23/2/e30.abstract.

C.M. Sperberg-McQueen. Notes on finite state automata with counters. Online: https://www.w3.org/XML/2004/05/msm-cfa.html, 2004.

V.B. Sreenu, G. Ranjitkumar, S. Swaminathan, S. Priya, B. Bose, M.N. Pavan, G. Thanu, J. Nagaraju, and H.A. Nagarajaram. MICAS: a fully automated web server for microsatellite extraction and analysis from prokaryote and viral genomic sequences. *Applied Bioinformatics Application Note*, 2(3), 2003.

K. Sutner. Context sensitive grammars, 2011a. URL https://www.cs.cmu.edu/~./FLAC/pdf/ContSens-6up.pdf.

K. Sutner. Context free grammars, 2011b. URL http://www.cs.cmu.edu/~./FLAC/pdf/CFG.pdf.

R. Szklarczyk and J. Heringa. Tracking repeats using significance and transitivity. *Bioinformatics*, 20(suppl 1): i311–i317, 2004. doi: 10.1093/bioinformatics/bth911. URL http://bioinformatics.oxfordjournals.org/content/20/suppl_1/i311.abstract.

S. Temnykh, Lukashova A., S. Cartinhour, G. de Clerck, L. Lipovich, and S. Mccouch. Computational and experimental analysis of microsatellites in rice (oryza sativa l.): frequency, length variation, transposon associations, and genetic marker potential. *Genome research*, 11:1441 – 1452, 2001.

W. Thiel, R. Michalek, and A. Graner. Exploiting EST databases for the development and characterization of gene-derived SSR-markers in barley (Hordeum vulgare L.) 20. *Theoretical and applied genetics*, 106(3):411–422, 2003.

M. I. Thurston and D. Field. MsatFinder: Detection and characterisation of microsatellites. Online: http://www.genomics .ceh.ac.uk/msatfinder/, 2005.

S. Trimberger. *Field-Programmable Gate Array Technology*. Springer Science and Business, 1 edition, 1994.

M. K. Trochim. Deduction and Induction. Online: http://www.socialresearchmethods.net, 2006.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. URL http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf.

C. J. Van Rijsbergen. *Information Retrieval*. Butterworths, 2nd edition, 1979. URL http://www.dcs.gla.ac.uk/Keith/Preface.html.

C. M. Vide, V. Mitrana, and Gheorghe Paun. *Grammars and Automata for String Processing*. CRC Press, 2003.

R. A. Wagner and M. J. Fischer. The String-to-String Correction Problem. *J. ACM*, 21(1):168–173, Jan 1974. ISSN 0004-5411. doi: 10.1145/321796.321811. URL http://doi.acm.org/10.1145/321796.321811.

R. A. Wagner and R. Lowrance. An Extension of the String-to-String Correction Problem. *J. ACM*, 22(2):177–183, Apr 1975. ISSN 0004-5411. doi: 10.1145/321879.321880. URL http://doi.acm.org/10.1145/321879.321880.

P.E. Warburton, J. Giordano, Y. Cheung, Y. Gelfand, and G. Benson. Inverted Repeat Structure of the Human Genome:The X-Chromosome Contains a Preponderance of Large, Highly Homologous Inverted Repeats That Contain Testes Genes. *Genome research*, 14:1861–1869, 2004.

B. W. Watson. The design and implementation of the FIRE Engine: A C++ toolkit for Finite Automata and Regular Expressions. Online: http://alexandra.tue.nl/extra1/wskrap/public html/9411065.pdf, 1994.

B.W. Watson. *Constructing minimal acyclic deterministic finite automata*. PhD thesis, Faculty of Engineering, Built Environment and Information Technology, University of Pretoria, 2010.

Y. Wexler, Z. Yakhini, Y. Kashi, and D. Geiger. Finding approximate tandem repeats in Genomic sequences. *Proceedings of the Eighth Annual International Conference on Computational Molecular Biology*, pages 223–232, 2004.

H. Wilf, D. Berlenski, D. Perrin, R. Askey, and D. Foata. In Memoriam: Marcel-Paul Schützenberger, 1920-1996. *Electronic Journal of Combinatorics*, October 1996.

S.A. Yevtushenko. The correctness-by-construction approach to programming. In *Proceedings of the 7th national conference on Artificial Intelligence KII-2000. System of data analysis "concept explorer" (in Russian)*, pages 127–134, 2000.

H. Zhou, L. Du, and H. Yan. Detection of tandem repeats in DNA sequences based on parametric spectral estimation. *Trans. Info. Tech. Biomed.*, 13(5):747–755, September 2009. ISSN 1089-7771.

W. Zhu, N. Zeng, and N. Wang. Sensitivity, specificity, accuracy, associated confidence interval and ROC analysis with practical SAS implementations. In *NESUG proceedings: Health Care and Life Sciences*, STOC '78, 2010. URL `http://www.lexjansen.com/cgi-bin/xsl_transform.php?x=shl&c=nesug`.

K.H. Zou, J. O'Malley, and L. Mauri. Receiver-operating characteristic analysis for evaluating diagnostic tests and predictive models. *Circulation*, 115(5):654–657, 2007.