



**BRNO UNIVERSITY OF TECHNOLOGY**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF INTELLIGENT SYSTEMS**

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

**AUTOMATA TECHNIQUES IN DNA ANALYSIS**

AUTOMATOVÉ TECHNIKY V ANALÝZE DNA

**BACHELOR'S THESIS**

BAKALÁŘSKÁ PRÁCE

**AUTHOR**

AUTOR PRÁCE

**LUCIE KLÍMOVÁ**

**SUPERVISOR**

VEDOUČÍ PRÁCE

**doc. Mgr. LUKÁŠ HOLÍK, Ph.D.**

**BRNO 2025**

# Bachelor's Thesis Assignment



164168

Institut: Department of Intelligent Systems (DITS)  
Student: **Klímová Lucie**  
Programme: Information Technology  
Title: **Automata techniques in DNA analysis**  
Category: Algorithms and Data Structures  
Academic year: 2024/25

## Assignment:

1. Familiarise yourself with the tool TE-greedy-nester for structure-based detection of LTR retrotransposons and their nesting, with the related problematics of searching for patterns in biological sequences.
2. Explore possibilities of speeding up the search procedure using technology of finite automata. Find inspiration in automata learning algorithms such as Alerghia, in automata based pattern matching algorithms, and of course in existing DNA analysis methods. Propose an algorithm that could potentially outperform tools such as BLAST or HMMER on an interesting class of the problem instances.
3. Implement the algorithm and evaluate its performance against the said tools.

## Literature:

1. LEXA, M. TE-greedy-nester: structure-based detection of LTR retrotransposons and their nesting. Bioinformatics, Volume 36, Issue 20. october 2020.
2. RAFAEL, C. and JOSE, O. Learning Stochastic Regular Grammars by Means of a State Merging Method. 2005
3. BLAST, <https://blast.ncbi.nlm.nih.gov/Blast.cgi>
4. HMMER, <http://hmmer.org/>

Requirements for the semestral defence:  
1,2

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Holík Lukáš, doc. Mgr., Ph.D.**  
Head of Department: Kočí Radek, Ing., Ph.D.  
Beginning of work: 1.11.2024  
Submission deadline: 14.5.2025  
Approval date: 4.11.2024

## Abstract

This thesis is focused on the possibilities of using finite automata to accelerate the detection of structural domains of transposons. The central part of the thesis introduces a method based on deterministic finite automata (DFA) as a faster alternative to the BLASTX tool. BLASTX is used by the TE-greedy-nester tool, which is designed to detect LTR retrotransposons. As a starting point, we used the HMMER tool, which uses a profile hidden Markov model (PHMM) to precisely describe the character of the searched sequence. Due to the significant nondeterminism of PHMMs, the determinization of the model of the entire domain proved unfeasible. Instead, a method to transform a PHMM into several smaller DFAs designed to detect domain subsequences was introduced. Closely located occurrences of these subsequences are subsequently interpreted as occurrences of the entire domain. The experimental evaluation demonstrated that the presented approach maintains high accuracy while achieving up to a tenfold search speedup compared to BLASTX.

## Abstrakt

Tato práce se zaměřuje na možnosti využití konečných automatů pro zrychlení detekce strukturních domén transpozonů. Hlavní část práce představuje metodu založenou na deterministických konečných automatech (DFA) jako rychlejší alternativu k nástroji BLASTX. Ten je využíván v rámci nástroje TE-greedy-nester, který slouží k detekci LTR retrotranspozonů. Jako výchozí bod byl využit nástroj HMMER, který velice přesně modeluje charakter hledané sekvence pomocí profilových skrytých Markovových modelů (PHMM). Vzhledem k vysoké míře nedeterminismu PHMM nebylo možné přímo vytvořit jeden deterministický model pro celou doménu. Místo toho byl navržen přístup, který PHMM transformuje na několik menších DFA navržených pro detekci podčástí domény. Blízké výskyty těchto podčástí pak indikují přítomnost celé domény. Výsledky testování ukázaly, že tento přístup zachovává vysokou přesnost a zároveň přináší až desetinásobné zrychlení vyhledávání oproti BLASTX.

## Keywords

profile hidden Markov models (PHMM), deterministic finite automata (DFA), LTR retrotransposons

## Klíčová slova

profilové skryté Markovovy modely (PHMM), deterministické konečné automaty (DFA), LTR retrotranspozony

## Reference

KLÍMOVÁ, Lucie. *Automata techniques in DNA analysis*. Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Lukáš Holík, Ph.D.

## Rozšířený abstrakt

Většina živých organismů obsahuje ve své DNA transpozony. Jedná se o úseky DNA, které jsou schopné se za určitých okolností přemístit do jiné části hostitelského genomu. Bylo prokázáno, že některé transpozony mohou mít regulační funkce, a ovlivňovat tak expresi okolních genů. Stáří transpozonů je odhadováno řádově na miliony let. Jejich anotace je proto klíčová pro porozumění dalším aspektům genové exprese a evoluce genomů.

K translokaci transpozonu nedochází příliš často, ale vzhledem k jejich četnému zastoupení, zejména v rostlinných genomech, běžně dochází k tomu, že je transpozon vložen na místo, kde se už nachází jiný, starší transpozon. To vede ke vzniku fragmentů transpozonů, což ztěžuje jejich detekci. Existuje několik nástrojů pro vyhledávání transpozonů a jedním z nich je TE-greedy-nester. Ten je schopen detekovat i hluboce zanořené transpozony tak, že opakovaně prohledává zadanou sekvenci a v každé iteraci z ní vyjme nejnovější, nefragmentované transpozony. Kvůli několika průchodům zadanou sekvencí je ale poměrně pomalý. Bylo experimentálně zjištěno, že přes 80 % jeho běhu zabírá nástroj BLASTX, který slouží k vyhledávání strukturních domén transpozonu. Cílem této práce je proto navrhnout alternativní metodu pro rychlejší vyhledávání těchto domén, založenou na konečných automatech. Ta by potenciálně mohla přispět ke zrychlení nástroje TE-greedy-nester a umožnila by tak vyhledávání transpozonů i v delších sekvencích.

První část práce se zaměřuje na možnosti využití algoritmu ALERGIA pro detekci strukturních domén transpozonů. ALERGIA vytváří pravděpodobnostní konečný automat (PFA) z množiny trénovacích sekvencí pomocí iterativního slučování podobných stavů. Tímto způsobem aproximuje distribuční vlastnosti vstupních sekvencí a umožňuje jejich efektivní modelování. Bylo prokázáno, že hledanou doménu lze pomocí PFA reprezentovat, ale vzhledem k nedostatečné sensitivitě bylo třeba automat rozšířit o další přechody, simulující mutace. Tyto nové přechody ale zavedly do automatu nedeterminismus, a proto bylo třeba použít složitější vyhledávací algoritmus. Tímto způsobem bylo možné strukturní domény detekovat, ale nevedlo to k urychlení vyhledávání v porovnání s dříve zmíněným nástrojem BLASTX.

Druhá, hlavní část práce je proto zaměřena na využití deterministických konečných automatů (DFA), které mají větší potenciál pro velmi rychlé vyhledávání. Jako výchozí bod byl využit nástroj HMMER, který je schopen velice přesně modelovat charakter hledané sekvence pomocí profilového skrytého Markovova modelu (PHMM). PHMM je nedeterministický pravděpodobnostní model se speciální strukturou navrženou pro reprezentaci biologických sekvencí včetně možných substitucí, inzercí a delecí. Zdeterminizovat model celé domény se ukázalo neproveditelné, zejména kvůli velkému počtu stavů automatu a značnému nedeterminismu, který je nutný pro modelování mutací. Práce proto představuje způsob, jak z PHMM vytvořit několik menších DFA, které mohou být využity pro vyhledání podčástí dané domény. Výskyty více takových podčástí ve správném pořadí a v dostatečné blízkosti jsou následně interpretovány jako výskyt celé domény.

Prezentovaný algoritmus byl otestován na uměle vygenerovaných sekvencích a byla změřena jeho rychlost a přesnost. Ukázalo se, že zjednodušení modelu, která byla nutná pro jeho determinizaci, nevedla k velkému poklesu sensitivity v porovnání s nástrojem BLASTX. Zároveň došlo až k desetinásobnému zrychlení vyhledávání, které bylo zejména patrné u velmi dlouhých sekvencí.

# Automata techniques in DNA analysis

## Declaration

I hereby declare that this thesis was prepared as an original author's work under the supervision of doc. Mgr. Lukáš Holík, Ph.D. All the relevant information sources which were used during the preparation of this thesis are properly cited and included in the list of references.

.....  
Lucie Klímová  
May 12, 2025

## Acknowledgements

I would like to thank my supervisor doc. Mgr. Lukáš Holík, Ph.D. for his guidance, help and patience during my work on this thesis. I would also like to thank doc. Ing. Tomáš Martínek, Ph.D. for valuable advice and consultations.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Biological Preliminaries</b>	<b>5</b>
2.1	DNA . . . . .	5
2.2	RNA . . . . .	6
2.3	Proteins . . . . .	6
2.4	Transposons . . . . .	8
<b>3</b>	<b>Automata Preliminaries</b>	<b>11</b>
3.1	Finite Automata . . . . .	11
3.2	Probabilistic Finite Automata . . . . .	13
3.3	Hidden Markov Models . . . . .	14
<b>4</b>	<b>TE-greedy-nester</b>	<b>16</b>
4.1	LTR Finder . . . . .	16
4.2	BLASTX . . . . .	16
4.3	TE-greedy-nester Algorithm . . . . .	17
<b>5</b>	<b>Existing Methods of Biological Sequence Alignment</b>	<b>19</b>
5.1	Scoring Systems . . . . .	19
5.2	Needleman–Wunsch Algorithm . . . . .	21
5.3	Smith-Waterman Algorithm . . . . .	22
5.4	BLAST . . . . .	23
5.5	HMMER . . . . .	25
<b>6</b>	<b>ALERGIA Algorithm in Biological Sequence Analysis</b>	<b>29</b>
6.1	ALERGIA . . . . .	29
6.2	Experimenting with ALERGIA . . . . .	33
6.3	Dealing with Nondeterminism . . . . .	35
6.4	Results . . . . .	37
<b>7</b>	<b>Proposed Gene Search Algorithm</b>	<b>39</b>
7.1	Process of Generating the Gene Model . . . . .	39
7.2	Proposed Search Algorithm . . . . .	46
<b>8</b>	<b>Performance Evaluation</b>	<b>48</b>
8.1	Evaluation Methodology . . . . .	48
8.2	Experimental Evaluation . . . . .	50
8.3	Summary of Results and Possible Improvements . . . . .	53

<b>9 Conclusion</b>	<b>54</b>
<b>Bibliography</b>	<b>55</b>

# Chapter 1

## Introduction

Most living organisms contain transposons in their DNA. These sequences are special since they are able to move along the host DNA. Due to their high numbers, especially in the plant genome, it is common that a transposon is inserted into an older transposon that is already present at that location. The nesting creates a sequence of transposon fragments, which makes them hard to detect. Some transposons have been found to play a crucial role in controlling gene expression. In addition, the age of transposons is estimated to be in the order of millions of years. Therefore, studying them can bring us new insights into genome evolution.

There are several tools capable of detecting transposons, but the one we found the most interesting is TE-greedy-nester [24]. It is able to detect even highly nested transposons by recursively removing the newest, unfragmented transposons. However, due to the recursion, it appears relatively slow. Approximately 80% of its runtime is taken up by a tool named BLASTX [6], which is used to find structural domains of the transposons. Thus, the main goal of this thesis is to introduce a new, faster method based on finite automata that could potentially replace the BLASTX tool and enable transposon detection in even longer sequences.

The first part of the thesis is focused on the possibilities of using the ALERGIA algorithm [7] to search for the desired domain. ALERGIA learns a probabilistic finite automaton (PFA) representing the domain using a state-merging technique. It has been shown that the domain can be represented using a PFA. However, due to the low sensitivity, the automaton had to be extended with additional transitions, simulating mutations. However, these new transitions introduced nondeterminism into the automaton, and therefore, a more sophisticated search algorithm had to be used. In this way, the structural domains could be detected, but it did not lead to a search speedup compared to BLASTX.

The second, central part of the thesis, therefore, focuses on the use of deterministic finite automata (DFAs), which have a greater potential for high-speed search. As a starting point, we used the HMMER tool [11], which uses a special type of probabilistic model called a profile hidden Markov model (PHMM) to precisely describe the character of the searched sequence, including substitutions, insertions and deletions. Determinizing the model of the entire domain proved unfeasible, primarily due to the large number of states and significant nondeterminism required to model the naturally occurring mutations. This work, therefore, presents a method to transform a PHMM into a set of smaller DFAs, which can be used to search for subsequences of the desired domain. Closely located occurrences of such subsequences in the correct order are interpreted as occurrences of the entire domain.

The presented algorithm was tested on artificially generated sequences, and its speed and accuracy were measured. It was shown that the simplification of the model, which was necessary for its determinization, did not lead to a significant decrease in sensitivity compared to the aforementioned BLASTX tool. In addition, the presented algorithm achieved a tenfold search speedup compared to BLASTX. The presented method shows a certain rate of false positives; however, it could be solved by introducing some metrics to evaluate the quality of the domain candidates found.

## **Thesis organisation**

Chapter 2 explains the basic biological principles necessary for understanding the problematics of LTR retrotransposon detection. Chapter 3 defines the necessary terms of automata theory related to this thesis, including a newly introduced extension of the classical nondeterministic finite automata called bounded counting automata (BCA). Chapter 4 describes the TE-greedy-nester tool used for LTR retrotransposon detection. The existing methods of biological sequence alignment are described in Chapter 5. Chapter 6 explores the possibilities and limitations of using the ALERGIA algorithm for detection of LTR retrotransposon domains. The central part of the thesis, which proposes the algorithm used to accelerate the domain detection, is located in Chapter 7. The proposed algorithm is evaluated in Chapter 8.

## Chapter 2

# Biological Preliminaries

This chapter briefly describes the biological preliminaries necessary for further understanding the methods used in biological sequence analysis and transposon detection. Sections 2.1, 2.2 and 2.3 are taken from [42].

### 2.1 DNA

Deoxyribonucleic acid (DNA) is a molecule that carries the genetic information of all eukaryotic organisms. It consists of two complementary antiparallel strands, which together form a structure called a double helix. The building blocks of these strands are called nucleotides, which consist of a pentose sugar, phosphate and nitrogenous base. DNA contains four types of bases: guanine (G), adenine (A), cytosine (C), and thymine (T). The order of the bases is crucial since it serves as a template for RNA and protein synthesis. The sugar-phosphate parts of the nucleotides are chained together and form the DNA backbone. The resulting strand has a pentose sugar at one end, known as the 3' end, and a phosphate at the other, the 5' end. These labels are often used to determine the direction of a strand.

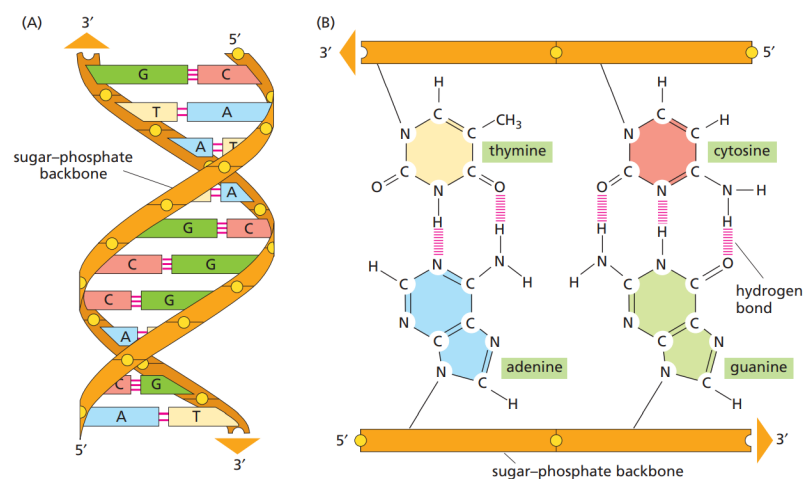


Figure 2.1: An illustration of DNA structure [1]. It consists of a sugar-phosphate backbone supporting each strand and the base sequence. The strands are complementary (thymine binds with adenine, and guanine binds with cytosine) and stick together using hydrogen bonds.

As already said, the two strands are complementary. Specifically, T binds with A and C with G, as shown in Figure 2.1. Thus, by knowing the sequence of one strand, we can determine the sequence of the other strand. This property is used in the process of DNA replication and repair.

## Genes

A gene is a relatively short part of the DNA sequence that serves as a template for functional RNA or protein synthesis. The genes are only a minor part of the DNA. For example, the protein-coding genes constitute approximately 1.5% of the human genetic information [5]. The rest is called non-coding DNA and was formerly considered non-functional. Today, however, it is known that these regions can have other non-structural functions, such as the regulation of gene expression.

## Mutations

Although DNA is a relatively stable molecule, many factors, including radiation, free radicals and DNA replication, can cause mutations. Substitutions are the most common type of mutation, occurring when a nucleotide is replaced by a different nucleotide that does not fit the position. Other mutation types are deletions and insertions, formed when a nucleotide or a whole sequence of nucleotides is removed or added to the original sequence. Deletions and insertions are not frequent in active genes because they can significantly alter the structure, thus often resulting in a malfunctioning protein.

## 2.2 RNA

Like DNA, ribonucleic acid (RNA) is a polymer consisting of four types of nucleotides: guanine (G), adenine (A), cytosine (C), and uracil (U). As can be noticed, the nucleotides are the same except for thymine (T), which is replaced by a similar nucleotide called uracil (U). Compared to DNA, RNA is primarily single-stranded and much smaller in length.

RNA can have various functions, including catalysing specific chemical reactions, but the most important is that it serves as a template for protein synthesis. Based on the gene sequence, a messenger RNA (mRNA) is formed during transcription. The mRNA strand is transferred out of the nucleus, where the final protein can be created during a process called translation. This information flow is called the central dogma of molecular biology, illustrated in Figure 2.2.

## 2.3 Proteins

Proteins are one of the most important molecules found in a cell. They catalyse most chemical reactions, have regulatory and signalling functions, and form a significant part of cellular structures.

The function of the protein is determined by its structure. The basic building blocks of the proteins are amino acids. The order of the amino acids determines the primary structure of the protein. Due to the interaction between amino acids, the chain folds into typical secondary structures, such as  $\alpha$ -helix and  $\beta$ -sheet. Further folding leads to the formation of the 3D tertiary structure, as shown in Figure 2.3. Some proteins require

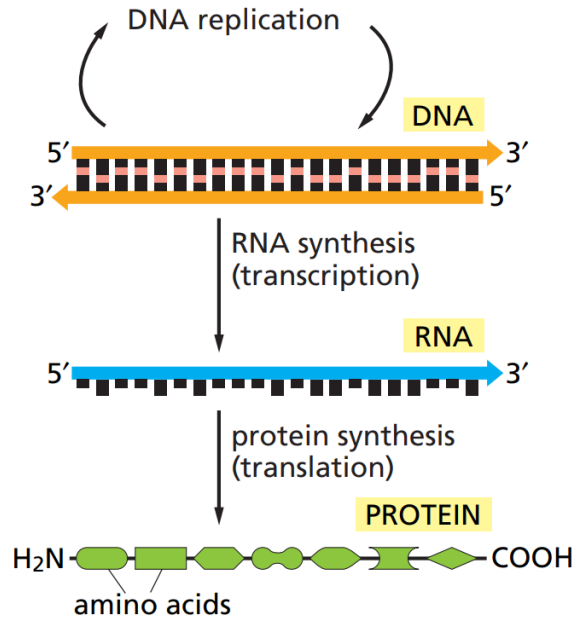


Figure 2.2: The central dogma of molecular biology [1].

multiple subunits to be fully functional. These subunits together form the quaternary structure of the protein.

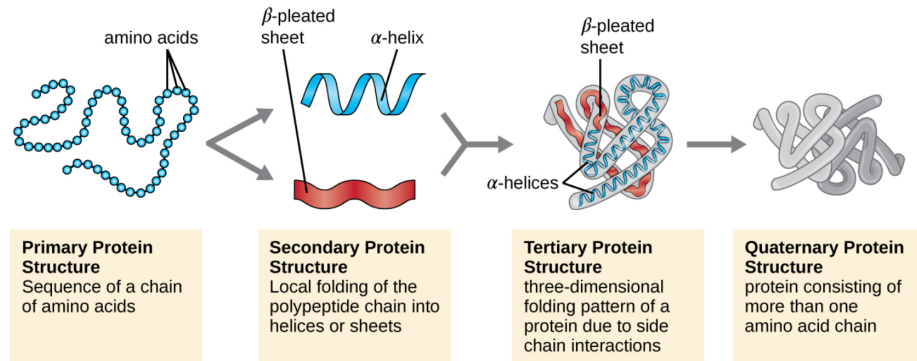


Figure 2.3: A schematic representation of the primary, secondary, tertiary and quaternary protein structure [25].

Amino acids can be characterised by many properties, such as polarity, acidity, and size, as shown in Figure 2.4. These properties are crucial in the formation of the secondary and tertiary protein structure. If the 3D protein structure is altered, the resulting protein can be dysfunctional. Thus, substituting an amino acid with a similar amino acid is more frequent because it preserves the physicochemical properties and typically does not change the 3D structure of the protein. This type of mutation is called a conservative replacement [27].

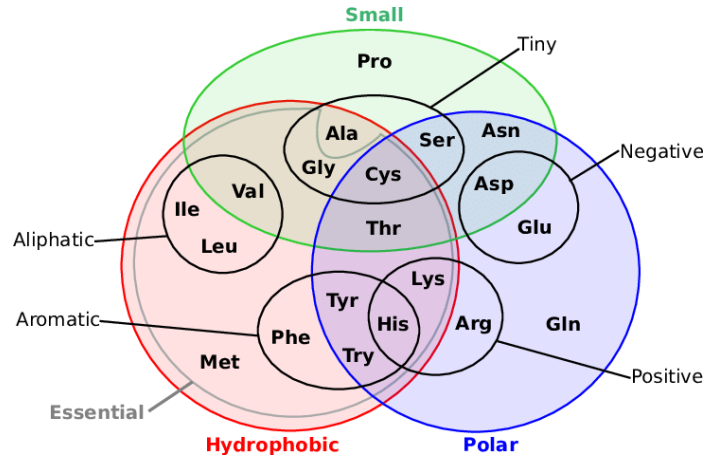


Figure 2.4: A Venn diagram of amino acid properties [23].

## Genetic code

There are 20 different amino acids. Each of them is encoded by a triplet of nucleotides called a codon. Given that there are four different nucleotides, there are 64 different codons; thus, some amino acids are coded by more than one codon, as shown in Figure 2.5. Three codons are unique since they do not encode any amino acid. Instead, they mark the position where the translation should stop. Therefore, they are called stop codons.

5' end	Second letter of the codon								3' end
	U	C	A	G	U	C	A	G	
U	UUU	Phe	UCU	Ser	UAU	Tyr	UGU	Cys	U
	UUC	Phe	UCC	Ser	UAC	Tyr	UGC	Cys	C
	UUA	Leu	UCA	Ser	UAA	Stop	UGA	Stop	A
	UUG	Leu	UCG	Ser	UAG	Stop	UGG	Trp	G
C	CUU	Leu	CCU	Pro	CAU	His	CGU	Arg	U
	CUC	Leu	CCC	Pro	CAC	His	CGC	Arg	C
	CUA	Leu	CCA	Pro	CAA	Gln	CGA	Arg	A
	CUG	Leu	CCG	Pro	CAG	Gln	CGG	Arg	G
A	AUU	Ile	ACU	Thr	AAU	Asn	AGU	Ser	U
	AUC	Ile	ACC	Thr	AAC	Asn	AGC	Ser	C
	AUA	Ile	ACA	Thr	AAA	Lys	AGA	Arg	A
	AUG	Met	ACG	Thr	AAG	Lys	AGG	Arg	G
G	GUU	Val	GCU	Ala	GAU	Asp	GGU	Gly	U
	GUC	Val	GCC	Ala	GAC	Asp	GGC	Gly	C
	GUA	Val	GCA	Ala	GAA	Glu	GGA	Gly	A
	GUG	Val	GCG	Ala	GAG	Glu	GGG	Gly	G

Figure 2.5: The genetic code [42]. A triplet of nucleotides called a codon encodes an amino acid. The code is degenerate since multiple codons can encode one amino acid.

## 2.4 Transposons

The genetic information of most eukaryotic organisms contains transposable elements (TEs) inserted into the DNA sequence throughout evolution [24]. These sequences can move along the host DNA. Therefore, they are called jumping genes. Generally, DNA is transcribed into

RNA, which serves as a template for protein synthesis, as described in previous sections. This phenomenon is called the central dogma of molecular genetics.

For a long time, it was thought that information was only transmitted in this direction. However, there are exceptions, such as retroviruses or TEs, for which the central dogma does not apply. Transposons, which are found in DNA, can either copy or excise themselves and can be inserted into another location in the host DNA, as shown in Figure 2.6.

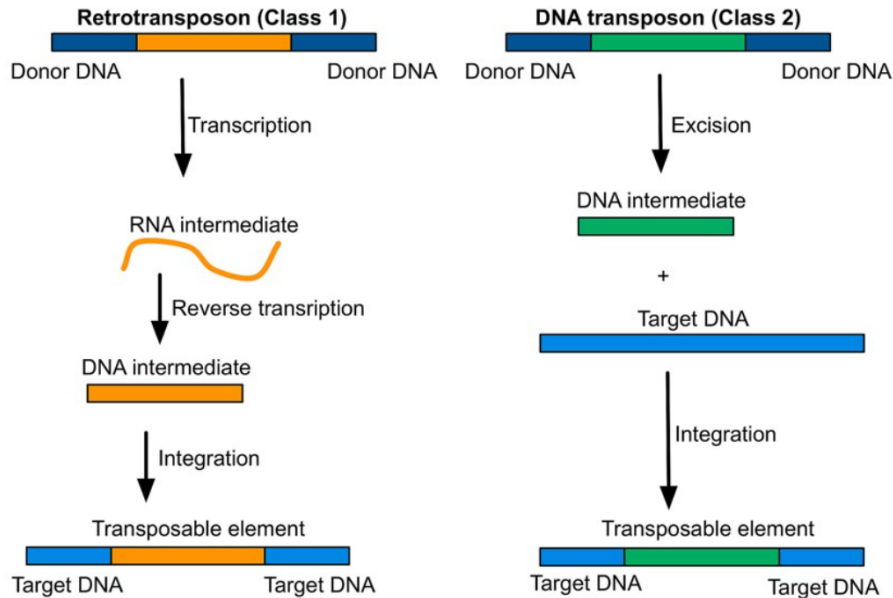


Figure 2.6: An illustration of transposon translocation [22]. Retrotransposon (Class 1) translocation can be described as a copy-and-paste process. The original TE remains in the donor DNA sequence. In contrast, a DNA transposon (Class 2) translocation is similar to a cut-and-paste operation since the original TE is excised from the donor DNA.

### LTR retrotransposons

LTR retrotransposons are the most common type of TEs in plant genomes, for example, constituting approximately 70% of maize genetic information [4]. However, they are also found in mammalian DNA. While some retrotransposons may be non-functional or have neutral effects, others have been found to play a crucial role in genome evolution and have regulatory functions, such as controlling gene expression [13]. It is, therefore, essential to localise them and determine the order in which they were nested.

### LTR retrotransposon structure

LTR retrotransposons consist of two long terminal repeats (LTRs), typically 250–600 bp in length, at both 5' and 3' ends, as shown in Figure 2.7. LTRs are relatively long, identical sequences that flank the LTR retrotransposon. Between these two LTRs is a coding region, approximately 5–7 kb long, that contains at least two domains: gag and pol genes, but the number can vary depending on the type of the transposon. These genes encode proteins such as protease (PR) and reverse transcriptase (RT), which are necessary for the transposon to replicate and move along the host DNA [41].

### Gypsy and BEL/Pao elements



### Copia elements



Figure 2.7: A schematic structure of LTR retrotransposons [34].

### Transposon nesting

Transposon translocation does not occur very often. However, due to their high numbers, especially in plant genomes, a transposon is often inserted into a part of DNA where another transposon is already present. The nesting leads to the formation of a sequence of transposon fragments and full-length transposons, which makes them hard to detect. An example of multilevel transposon nesting is shown in Figure 2.8.

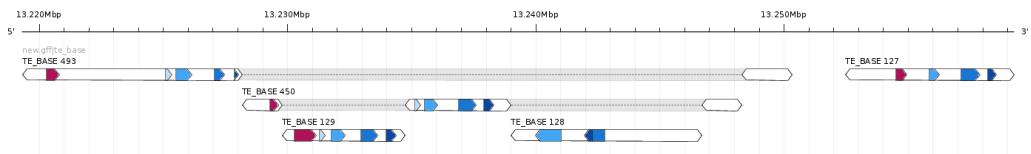


Figure 2.8: An example of multilevel transposon nesting [24].

# Chapter 3

## Automata Preliminaries

In order to fully understand the algorithms described in this thesis, the necessary terms must be defined formally.

### 3.1 Finite Automata

Finite automata are mathematical models widely used in pattern recognition and formal language theory. An automaton processes the input sequence, moving from state to state based on the current state and the symbol read. Once the entire string is processed, the automaton determines whether the string is accepted or rejected based on the automaton's final states. Finite automata can be defined in several ways. The following definitions are inspired by [18].

#### Nondeterministic finite automaton

A *nondeterministic finite automaton (NFA)* is defined as a five-tuple  $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ , where:

- $Q$  is a finite set of *states*,
- $\Sigma$  is an input *alphabet*,
- $\Delta \subseteq Q \times \Sigma \times Q$  is a set of tuples  $(q, a, r)$  called *transitions*, also denoted by  $q \xrightarrow{a} r$ , where  $q, r \in Q$  and  $a \in \Sigma$ ,
- $I \subseteq Q$  is the set of *initial states*,
- $F \subseteq Q$  is the set of *final states*.

A *run* of  $\mathcal{A}$  over a word  $w = a_1 \dots a_n \in \Sigma^*$  from state  $p_0$  to  $p_n, n \geq 0$  is a sequence of transitions  $p_0 \xrightarrow{a_1} p_1, p_1 \xrightarrow{a_2} p_2, \dots, p_{n-1} \xrightarrow{a_n} p_n$  from  $\Delta$ . The empty sequence is a run with  $p_0 = p_n$  over the empty word  $\epsilon$ . The run is *accepting* if  $p_0 \in I$  and  $p_n \in F$ . The *language* of  $\mathcal{A}$ ,  $L(\mathcal{A})$ , is a set of all words for which  $\mathcal{A}$  has an accepting run. A state  $q$  is *reachable* if there is a run from  $I$  to  $q$ .

An NFA  $\mathcal{A}$  is *deterministic (DFA)* if it has exactly one initial state  $I = \{q_0\}$  and for every state  $q \in Q$  and input symbol  $a \in \Sigma$ , there is at most one transition  $q \xrightarrow{a} r$  in  $\Delta$ . The DFA is *complete* if there is exactly one transition  $q \xrightarrow{a} r$  in  $\Delta$  for every  $q \in Q$  and  $a \in \Sigma$ . For a complete DFA, we can define a *transition function*  $\delta$ , such that  $\delta : Q \times \Sigma \rightarrow Q$ . Then by  $\delta(q, a)$ , we denote state  $r$ , such that  $q \xrightarrow{a} r \in \Delta$ .

## Determinization

This subsection is inspired by [19]. Finite automata  $\mathcal{A}$  and  $\mathcal{B}$  are *equivalent* if  $L(\mathcal{A}) = L(\mathcal{B})$ . For every NFA, there is an equivalent DFA. The process of converting NFA into an equivalent DFA is called determinization, and it is described in Algorithm 1. The states of the resulting DFA are denoted as the sets of states of the NFA that can be reached via a given symbol. A set is regarded as a final state of the DFA if at least one of the states in the set is a final state of the NFA.

---

### Algorithm 1: DETERMINIZE

---

**Input:**  $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \Delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$  – a NFA  
**Output:**  $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \Delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$  – an equivalent DFA

- 1  $Q_{\mathcal{B}} \leftarrow \emptyset;$
- 2  $\Delta_{\mathcal{B}} \leftarrow \emptyset;$
- 3  $I_{\mathcal{B}} \leftarrow \{I_{\mathcal{A}}\};$
- 4  $F_{\mathcal{B}} \leftarrow \emptyset;$
- 5 Queue *queue*;
- 6 *queue.enqueue*( $I_{\mathcal{A}}$ );
- 7 **while** *not queue.empty()* **do**
- 8      $S \leftarrow \text{queue.dequeue}();$
- 9      $Q_{\mathcal{B}} \leftarrow Q_{\mathcal{B}} \cup \{S\};$
- 10    **foreach** *symbol*  $a \in \Sigma$  **do**
- 11        $T \leftarrow \bigcup_{s \in S} \{t : s \xrightarrow{a} t \in \Delta_{\mathcal{A}}\};$
- 12       **if**  $T \notin Q_{\mathcal{B}}$  **then**
- 13             $\text{queue.enqueue}(T);$
- 14             $\Delta_{\mathcal{B}} \leftarrow \Delta_{\mathcal{B}} \cup \{S \xrightarrow{a} T\};$
- 15    **if**  $S \cap F_{\mathcal{A}} \neq \emptyset$  **then**
- 16         $F_{\mathcal{B}} \leftarrow F_{\mathcal{B}} \cup \{S\};$
- 17 **return**  $\mathcal{B};$

---

## Bounded counting automaton

For the purposes of the algorithms proposed in Chapter 7, we introduce a new, extended version of the classical NFA called a bounded counting automaton (BCA). It has a *counter*  $c$  that is updated when a transition is made. The counter is *bounded*, so transitions that would cause the counter to exceed the bound are not allowed. For convenience, the presented definition allows only a single counter and a single initial configuration and does not contain transition guards.

A *bounded counting automaton (BCA)* is defined as a six-tuple  $\mathcal{A} = (Q, \Sigma, \Delta, \text{conf}_0, F, b)$ , where:

- $Q$  is a finite set of *states*,
- $\Sigma$  is an input *alphabet*,
- $\Delta \subseteq Q \times \Sigma \times \mathbb{N} \times Q$  is a set of transitions, where a *transition* is a tuple  $(q, a, u, r)$ , denoted by  $q \xrightarrow{a, u} r$ , where  $q, r \in Q$ ,  $a \in \Sigma$  and  $u : \mathbb{N} \rightarrow \mathbb{N}$  is a counter *update*,
- $\text{conf}_0$  is the *initial configuration*, where a *configuration* is a pair of the form  $(q, k)$  where  $q \in Q$  and  $k \in \mathbb{N}$  is a counter value,

- $F \subseteq Q$  is the set of *final states*,
- $b \in \mathbb{N}$  is a counter upper *bound*.

The *update*  $u$  is a function that increments the counter by  $n \in \mathbb{N}$ , denoted by  $c := c + n$  or resets it to 0, denoted by  $c := 0$ . The *language* of  $\mathcal{A}$ ,  $L(\mathcal{A})$ , is defined as the language of its *configuration automaton*  $\text{Conf}(\mathcal{A})$ . The states of  $\text{Conf}(\mathcal{A})$  are configurations  $(q, k)$  of  $\mathcal{A}$  that are reachable and for which it applies that  $k \leq b$ . The set of initial states of  $\text{Conf}(\mathcal{A})$  contains only the initial configuration  $\text{conf}_0$ . The automaton  $\text{Conf}(\mathcal{A})$  has a transition  $(q, k) \xrightarrow{a} (r, u(k))$  if:

- $(q, k)$  is reachable,
- $\mathcal{A}$  has a transition  $q \xrightarrow{a, u} r \in \Delta$ ,
- $u(k) \leq b$ .

## 3.2 Probabilistic Finite Automata

Probabilistic finite automata (PFAs) are stochastic models that describe probability distributions over a set of strings by associating transitions with probabilities. PFAs can be particularly useful when modelling noisy data, such as biological sequences. In this thesis, they are used in Chapter 6, which describes the possibilities of using the ALERGIA algorithm [7] in biological sequence analysis. The following definitions are taken from [17].

### Probabilistic finite automaton

A *probabilistic finite automaton (PFA)* is defined as a five-tuple  $\mathcal{A} = (Q, \Sigma, \delta_p, I_p, F_p)$ , where:

- $Q$  is a finite set of *states*,
- $\Sigma$  is an input *alphabet*,
- $\delta_p : Q \times (\Sigma \cup \{\epsilon\}) \times Q \rightarrow [0, 1]$  is a *transition function*,
- $I_p : Q \rightarrow [0, 1]$  is a function defining *initial state probabilities*,
- $F_p : Q \rightarrow [0, 1]$  is a function defining *acceptance probabilities*.

The initial state probabilities must sum to 1:

$$\sum_{q \in Q} I_p(q) = 1.$$

Given that  $\delta_p(p, a, q)$ , where  $p, q \in Q$  and  $a \in \Sigma \cup \epsilon$ , denotes the probability of the automaton  $\mathcal{A}$  making a transition from  $p$  to  $q$  while reading  $a$  and  $F_p(p)$  represents the acceptance probability of state  $p$ . The transition and acceptance probabilities must satisfy the following equation:

$$\forall p \in Q : \sum_{q \in Q, a \in \Sigma \cup \epsilon} \delta_p(p, a, q) + F_p(p) = 1.$$

Let  $p \xrightarrow{a} q$  be a *transition* from  $p \in Q$  to  $q \in Q$  over  $a \in \Sigma \cup \{\epsilon\}$  such that  $\delta_p(p, a, q) > 0$ . A *run* of  $\mathcal{A}$  over a word  $w = a_1 \dots a_n \in \Sigma^*$ ,  $n \geq 0$  from state  $p_0$  to  $p_m$ ,  $m \geq 0$  is a sequence of transitions

$$\rho = p_0 \xrightarrow{a'_1} p_1, p_1 \xrightarrow{a'_2} p_2, \dots, p_{m-1} \xrightarrow{a'_m} p_m$$

such that:

- $a'_1 \dots a'_m \in \Sigma^*$ ,
- the subsequence of non- $\epsilon$  symbols of  $a'_1 \dots a'_m$  must be equal to the subsequence of non- $\epsilon$  symbols of  $w$ .

The probability of such run  $\rho$  is

$$P_{\mathcal{A}}(\rho) = I_p(p_0) \cdot \prod_{i=1}^m \delta_p(p_{i-1}, a'_i, p_i) \cdot F_p(p_m).$$

Let  $\mathcal{R}_{\mathcal{A}}(w)$  be the set of all possible runs of  $\mathcal{A}$  generating word  $w$ . The *probabilistic language* recognised by the PFA  $\mathcal{A}$  is a probability distribution over  $\Sigma^*$ . The probability of word  $w = a_1 \dots a_n \in \Sigma^*$ ,  $P_{\mathcal{A}}(w)$ , is then:

$$P_{\mathcal{A}}(w) = \sum_{\rho \in \mathcal{R}_{\mathcal{A}}(w)} P_{\mathcal{A}}(\rho),$$

and the distribution must verify:

$$\sum_{w \in \Sigma^*} P_{\mathcal{A}}(w) = 1.$$

A PFA  $\mathcal{A} = (Q, \Sigma, \delta_p, I_p, F_p)$  is a *deterministic (DPFA)* if:

- it has exactly one initial state  $q_0$  such that  $I_p(q_0) = 1$ ,
- $\delta_p(p, \epsilon, q) = 0$  for all  $p, q \in Q$ ,
- $\forall p \in Q \forall a \in \Sigma : |\{q : \delta_p(p, a, q) > 0\}| \leq 1$ .

### 3.3 Hidden Markov Models

Hidden Markov models (HMMs) are statistical models widely used in speech recognition and biological sequence analysis. Similarly to PFAs, they can be used to describe the character of a set of sequences. A tool named HMMER [11], described in Section 5.5, uses a special type of HMM called profile hidden Markov model (PHMM) to describe the character of the modelled gene precisely. This thesis introduces a method to convert a PHMM into a set of DFAs, described in Chapter 7. The following definition is inspired by [40].

#### Hidden Markov model

A *hidden Markov model (HMM)* is defined as a five-tuple  $\mathcal{A} = (S, O, \pi, t, E)$ , where:

- $S$  is a finite set of *states*,
- $O$  is a finite set of *observable symbols*,

- $\pi : S \rightarrow [0, 1]$  is a function assigning *initial state probabilities*,
- $t : S \times S \rightarrow [0, 1]$  is a *transition probability function*,
- $E \in \mathbb{R}^{|O| \times |S|}$  is a matrix of *emission probabilities*.

The initial state probabilities must sum to 1:

$$\sum_{s \in S} \pi(s) = 1.$$

An HMM is a model representing a distribution over a set of sequences. It generates an observable symbol sequence  $x = x_1 x_2 \dots x_L \in O^*$ , while the sequence of states  $y = y_1 y_2 \dots y_L \in S^*$ , the *path* that generated the sequence, remains hidden. It satisfies the *Markov condition*, so the probability of entering state  $j \in S$  depends only on the current state  $i \in S$  and does not change over time. Hence, the transition probability is:

$$P\{y_{n+1} = j \mid y_n = i, y_{n-1} = i_{n-1}, \dots, y_1 = i_1\} = P\{y_{n+1} = j \mid y_n = i\} = t(i, j).$$

The probability of emitting  $a \in O$  also depends only on the current state  $i$ , and it is denoted by  $E[a, i]$ . For all  $s \in S$ , the transition and emission probabilities must satisfy the following equations:

$$\sum_{r \in S} t(s, r) = 1,$$

$$\sum_{o \in O} E[o, s] = 1.$$

For simplicity, we denote the probabilistic measures that specify an HMM as  $\Theta$ . The probability of generating the observable symbol sequence  $x$  while taking the path  $y$  can then be computed as a joint probability:

$$P\{x, y \mid \Theta\} = P\{x \mid y, \Theta\} \cdot P\{y \mid \Theta\},$$

where:

$$P\{x \mid y, \Theta\} = \prod_{i=1}^L E[x_i, y_i],$$

$$P\{y \mid \Theta\} = \pi(y_1) \cdot \prod_{i=2}^L t(y_{i-1}, y_i).$$

## Chapter 4

# TE-greedy-nester

Existing tools capable of detecting LTR retrotransposons include, for example, the LTR finder [39], which is relatively fast but unable to identify nested TEs. Another tool named RepeatMasker [32] first locates fragments of structural elements that could be a part of a transposon and then tries to connect the closely located fragments to form a whole TE. Thus, it can detect some nesting.

The tool we found the most interesting is TE-greedy-nester [24]. It can detect even deep nesting by recursively searching for the newest, unfragmented transposons and removing them from the original sequence. It uses two main external tools to identify the structural components of LTR retrotransposons: BLASTX [6] and LTR finder [39].

Due to the recursive call of the algorithm on the entire input sequence, it appears relatively slow. Approximately 80% of its runtime is taken up by the BLASTX algorithm, which is used to make sequence alignments. Thus, the original motivation of this work was to introduce a new, faster tool based on finite automata that could possibly replace the BLASTX tool. However, the results show that the proposed method could be applicable even in other areas of biological data analysis. This chapter describes the TE-greedy-nester algorithm and the tools it uses in more detail.

### 4.1 LTR Finder

LTR finder [39] is a command-line tool designed for finding full-length LTR retrotransposons. First, it locates all the exact matching segments using a suffix-array-based algorithm and then tries to extend them. Then, if they occur, it detects sequences encoding RT and other enzymes between the pair of LTRs.

The user can adjust several parameters, including gap penalties, maximum and minimum LTR length, and maximum distance between LTRs, depending on their requirements and the query sequence characteristics. The program returns a list of candidate transposons, including their exact position and information about the detected domains.

### 4.2 BLASTX

BLASTX [6] is a variant of the BLAST tool [3] used when aligning a nucleotide sequence with the database of amino acid sequences. In order to make the alignment, the query sequence must first be converted into amino acid sequences according to the genetic code.

Because each amino acid is encoded by a codon, a triplet of nucleotides, the translation can start at three different positions. Each starting position gives a different translation result, creating three possible reading frames. Since DNA has two antiparallel strands, there are six different reading frames in total, as shown in Figure 4.1.

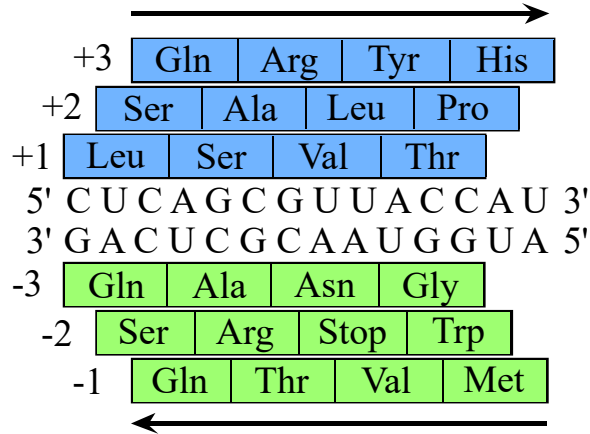


Figure 4.1: The six possible reading frames. The resulting amino acid sequence depends on the starting position of the translation. The arrows indicate the reading direction.

Apart from the fact that the translation must be done before making the alignment, BLASTX is based on the same principle as BLAST. The BLAST algorithm is described in more detail in Section 5.4.

### 4.3 TE-greedy-nester Algorithm

TE-greedy-nester [24] is a command-line tool that is able to detect even deeply nested LTR retrotransposons. Since older transposons are often fragmented by later inserted transposons, the program first locates the newest, unfragmented TEs. The TEs are then cut out of the original sequence, and the algorithm is repeated until no other transposon is found.

First, the TE-greedy-nester calls the LTR finder, which determines the position of LTRs and other noncoding regions, such as the primer binding site (PBS) and polypurine tract (PPT). Next, BLASTX is called to find the remaining domains of the transposon, including the gag gene, PR, and RT. The TE-greedy-nester algorithm is illustrated in Figure 4.2.

According to the location of the elements found by these tools, the score of each candidate is evaluated using a weighted graph algorithm. Each node represents a structural element of the LTR retrotransposon. The forward edges represent the correct order of the structural elements, and any deviation from the desired order or omission of any element is penalised using the dotted edges, as shown in Figure 4.2. The best candidate is found as a path from the left to the right LTR that visits the maximum of required nodes in the correct order.

Once all transposons are detected, a GFF file is generated to visualise the exact position of the TEs found and their structural elements.

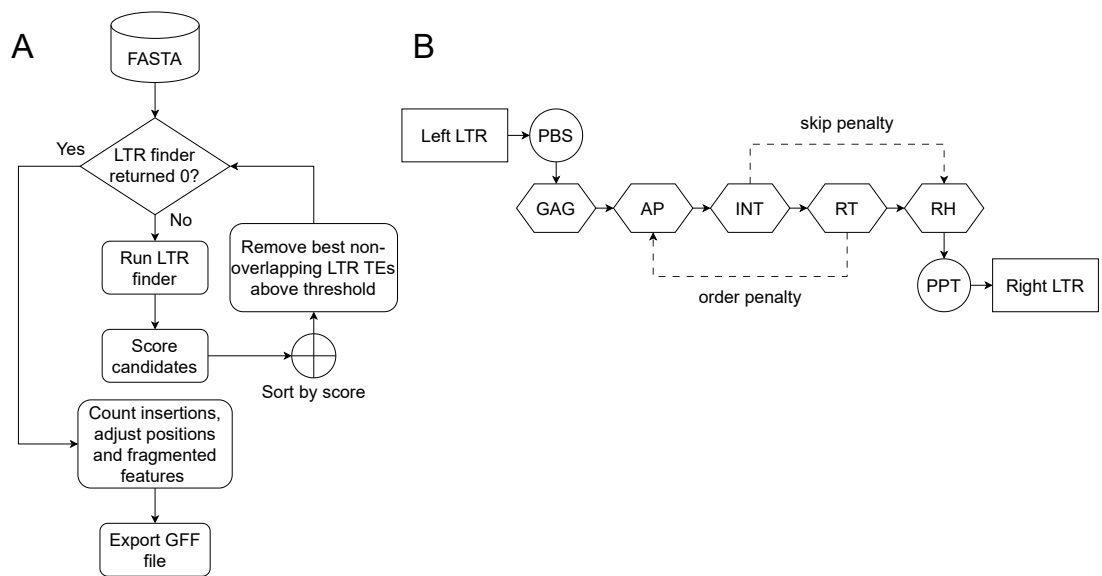


Figure 4.2: (A) An illustration of the TE-greedy-nester algorithm. It recursively searches for the newest, unfragmented transposons and removes them from the original sequence. (B) A scoring graph used to evaluate the TE candidates. Any structural deviation is penalized, denoted by dotted arrows. Modified from [24].



should be. A simple scoring system could be defined as follows:

$$S = \sum_{i=1}^n \begin{cases} 1 & \text{if } s_1[i] = s_2[i] & \text{(match)} \\ -1 & \text{if } s_1[i] = '-' \vee s_2[i] = '-' & \text{(gap)} \\ 0 & \text{otherwise} & \text{(mismatch),} \end{cases}$$

where:

- $S$  is the alignment score,
- $s_1$  and  $s_2$  are the sequences being aligned,
- $n$  is the alignment length (including gaps).

The alignment score should increase if the sequences match at a given location. If they do not match, the score should decrease or remain the same, depending on the scoring system. Insertions and deletions are not as common as substitutions. Thus, an additional gap penalty of  $w$  is usually applied. Inserting one longer segment is more probable than inserting multiple shorter segments. Thus, the gap penalty is often defined as  $w = a + bk$ . It consists of two components: the gap opening penalty  $a$  and the gap extension penalty  $bk$ , where  $k$  is the length of the gap and  $b$  is a suitable constant.

Due to the structural similarity of some amino acids, some substitutions occur more frequently than others in real biological sequences. Therefore, substitution matrices, such as BLOSUM [16] and PAM [10], are often used to calculate the score. These matrices were designed using real biological data, and they assign the scores based on the likelihood of amino acid substitutions.

	C	S	T	A	G	P	D	E	Q	N	H	R	K	M	I	L	V	W	Y	F	
C	9																				C
S	-1	4																			S
T	-1	1	5																		T
A	0	1	0	4																	A
G	-3	0	-2	0	6																G
P	-3	-1	-1	-1	-2	7															P
D	-3	0	-1	-2	-1	-1	6														D
E	-4	0	-1	-1	-2	-1	2	5													E
Q	-3	0	-1	-1	-2	-1	0	2	5												Q
N	-3	1	0	-2	0	-2	1	0	0	6											N
H	-3	-1	-2	-2	-2	-2	-1	0	0	1	8										H
R	-3	-1	-1	-1	-2	-2	-2	0	1	0	0	5									R
K	-3	0	-1	-1	-2	-1	-1	1	1	0	-1	2	5								K
M	-1	-1	-1	-1	-3	-2	-3	-2	0	-2	-2	-1	-1	5							M
I	-1	-2	-1	-1	-4	-3	-3	-3	-3	-3	-3	-3	-3	1	4						I
L	-1	-2	-1	-1	-4	-3	-4	-3	-2	-3	-3	-2	-2	2	2	4					L
V	-1	-2	0	0	-3	-2	-3	-2	-2	-3	-3	-3	-2	1	3	1	4				V
W	-2	-3	-2	-3	-2	-4	-4	-3	-2	-4	-2	-3	-3	-1	-3	-2	-3	11			W
Y	-2	-2	-2	-2	-3	-3	-3	-2	-1	-2	2	-2	-2	-1	-1	-1	-1	2	7		Y
F	-2	-2	-2	-2	-3	-4	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	1	3	6	F
	C	S	T	A	G	P	D	E	Q	N	H	R	K	M	I	L	V	W	Y	F	

Figure 5.1: The BLOSUM62 substitution matrix [38]. Groups of similar amino acids are marked in colour.

## 5.2 Needleman–Wunsch Algorithm

The Needleman–Wunsch algorithm [28] is a global alignment technique based on dynamic programming. By dividing the problem into a set of smaller problems, the optimal alignment can be found in quadratic time.

Figure 5.2 shows a part of the tree of all possible alignments of sequences GAATTC and GGATC. Each node represents the reminders of both sequences that have not been aligned yet. At each node of the tree, there are three options:

- align the characters at the given position, which leads to a match or a mismatch,
- insert a gap into the first sequence,
- insert a gap into the second sequence.

The alignment scores corresponding to each of the three options are marked in red.

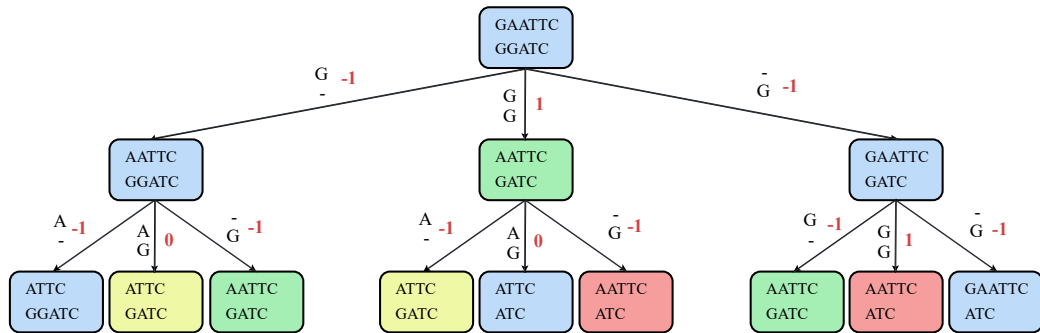


Figure 5.2: A part of the tree of all possible alignments of sequences GAATTC and GGATC. Inspired by [26]. The identical nodes are marked in colour.

As indicated, some nodes are the same; therefore, they do not need to be computed multiple times. After removing the duplicate nodes, the tree can be rearranged into an oriented graph, whose nodes are organised into an  $m \times n$  Needleman–Wunsch matrix, where  $m$  and  $n$  are the lengths of the sequences aligned. Each path in the graph represents one possible alignment. The diagonal edges represent symbol alignment, while horizontal and vertical edges represent gaps inserted into the top or left sequence, respectively. Thanks to the node elimination, the Needleman–Wunsch algorithm reduces the time complexity from exponential to  $O(mn)$ .

The first row and column of the matrix are initialised with a gradually increasing gap penalty score. The rest of the matrix is filled using the following rule:

$$M_{i,j} = \text{Max} \begin{cases} M_{i-1,j-1} + S_{i,j} \\ M_{i,j-1} - w \\ M_{i-1,j} - w, \end{cases}$$

where:

- $M$  is the Needleman–Wunsch matrix,
- $S_{i,j}$  is a function that assigns a score based on the similarity of nucleotides or amino acids at positions  $i$  and  $j$ ,

- $w$  is a function defining the gap penalty.

The optimal sequence alignment can be found by backtracking the graph path that produced the alignment score, located in the bottom right corner of the matrix. Figure 5.3 depicts the Needleman-Wunsch algorithm using the scoring system presented in Section 5.1.

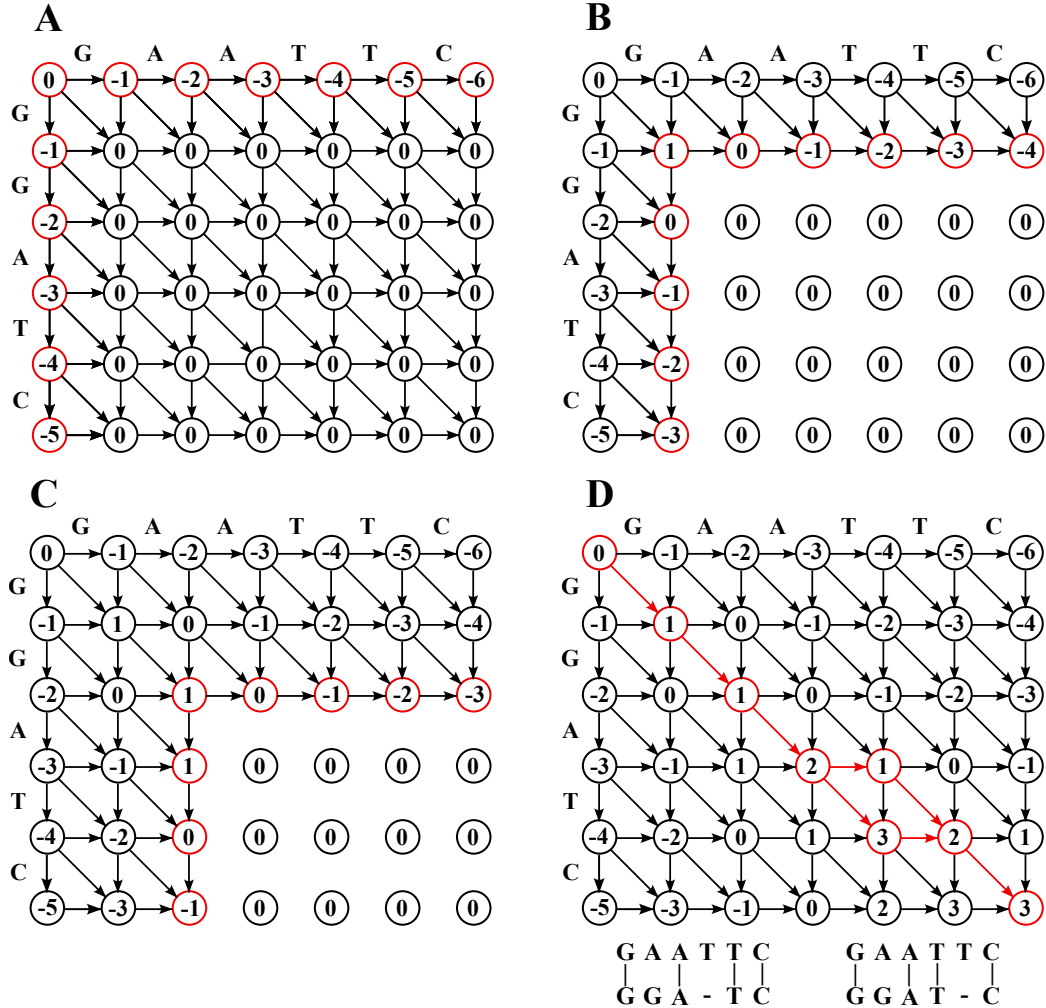


Figure 5.3: An illustration of the Needleman-Wunsch algorithm. Inspired by [14]. (A) The initialisation of the first row and column. (B, C) The process of filling the matrix. (D) The two resulting optimal alignments.

### 5.3 Smith-Waterman Algorithm

The Needleman-Wunsch algorithm produces a global alignment that indicates the similarity of sequences throughout their entire length. However, in many cases, it is necessary to identify all occurrences of a short sequence within a long sequence, a process referred to as a local alignment. The first widely used algorithm for local alignment was developed by Smith and Waterman [33]. They adjusted the Needleman-Wunsch algorithm with the following modifications:

- the scores in the matrix are not allowed to fall below zero to ensure that the alignment can start at any given position. Otherwise, the score would be reduced by leading and trailing gaps,
- the first row and column of the matrix are set to zero,
- the mismatches are also penalised.

The backtracking procedure, used to find the optimal local alignment, starts at the matrix's highest-scoring positions. The paths that produced these scores are followed until a zero score is reached.

## 5.4 BLAST

This section is inspired by [2]. Basic local alignment search tool (BLAST), developed by Altschul and collective [3], is a widely used bioinformatics tool for making a local sequence alignment. It receives a query sequence as input and searches a given database to find sequences that are similar to the query sequence. The base of the algorithm is similar to Smith-Waterman's, but it uses a few heuristics to reduce the execution time significantly.

First, a  $k$ -letter word list of the query sequence is created. Each word represents a short subsequence from the query, which is obtained using a  $k$ -wide sliding window. For amino acids, usually  $k = 3$ . The process of creating the word list is depicted in Figure 5.4. For each word, variant words (mutations) are generated based on a similarity threshold  $T$ . The score of the mutation is evaluated using a substitution matrix, such as the BLOSUM matrix [16]. Only words whose scores exceed  $T$  are retained. Then, the database is scanned for exact matches of these words. The matches found in the database are called seeds. The seeding process is illustrated in Figure 5.5.

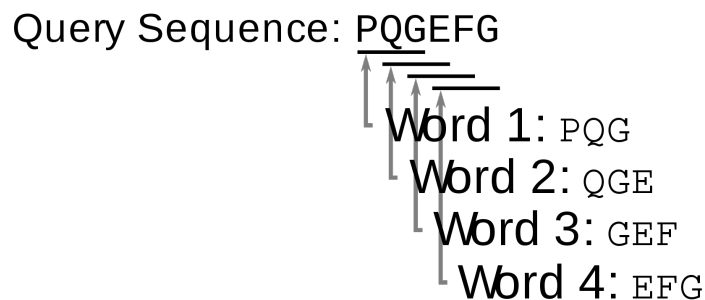


Figure 5.4: A process of generating the word list from a query sequence using a 3-wide sliding window [37].

The next step is called an extension. Closely located seeds are joined together, forming a longer, ungapped segment. The extension continues in both directions until the score drops more than a specified value  $X$  below the best score yet found. If the segment's score exceeds a specific value, it is considered a high-scoring segment pair (HSP) and will be further processed.



Closely located HSPs with sufficiently low E-value can be further merged. A modified version of Smith-Waterman’s algorithm is used to align the identified HSPs precisely.

## BLAST variants

There are several variants of the BLAST algorithm, each of which is used for different applications [6]. The following variants are relevant to this thesis:

- BLASTX is used to translate an amino acid query sequence into the six possible reading frames and compare them against a protein database,
- BLASTP performs a protein-to-protein comparison.

## 5.5 HMMER

HMMER [11] is a widely used tool designed to find sequence homologues and make sequence alignments. It is based on probabilistic models called profile hidden Markov models (PHMMs), which are used to model the character of the searched sequence precisely.

### Profile hidden Markov model

A profile HMM is a special type of HMM that is created based on an alignment of multiple variants of the searched sequence. The alignment consists of several columns, each containing the symbols and gaps observed at that position. The column is considered a *match column* if there are more matches and mismatches than gaps. Each match column is modelled by a triplet of a match, an insert, and a delete state. Therefore, a PHMM precisely models all types of mutations, including substitutions, insertions, and deletions.

A *profile hidden Markov model (PHMM)* is a five-tuple  $\mathcal{A} = (S, O, \pi, t, E)$ , where:

- $S = \{m_0, end, i_0\} \cup \bigcup_{k=1}^n \{m_k, i_k, d_k\}$  is a finite set of *states*, where:
  - *match states*  $m_k$  model the frequency of symbols observed in column  $k$ ,
  - *insert states*  $i_k$  represent possible insertions at column  $k$  and their symbol frequencies,
  - *delete states*  $d_k$  represents possible deletions in column  $k$ ,
  - $m_0$  is the *initial state*,
  - *end* denotes the end of the sequence,
  - $i_0$  models the insertions before the first match state,
  - $n$  is the number of match columns of the multiple sequence alignment,
- $O$  is a finite set of *observable symbols*,
- $\pi : S \rightarrow [0, 1]$  is a function that assigns *initial state probabilities* such that  $\pi(s) > 0$  only for  $s = m_0$ ,
- $t : S \times S \rightarrow [0, 1]$  is a *transition probability function*,
- $E \in \mathbb{R}^{|O| \times |S|}$  is a matrix of *emission probabilities*.

Additionally, let:

- $M = \{m_0, \dots, m_n\}$ ,
- $I = \{i_0, \dots, i_n\}$ ,
- $D = \{d_1, \dots, d_n\}$ .

The states are organised into a regular structure, where each match column  $1 \leq k \leq n$  of the multiple sequence alignment is modelled by a triplet of states  $(m_i, i_i, d_i)$ . The transition probabilities are non-zero only for the following transitions:

$$\begin{aligned} & t(i_n, end), t(m_n, end), t(d_n, end), \\ & t(d_k, d_{k+1}), t(d_k, m_{k+1}) \quad \text{for } 1 \leq k < n, \\ & t(i_k, i_k), t(m_k, i_k) \quad \text{for } 0 \leq k \leq n, \\ & t(i_k, m_{k+1}), t(m_k, m_{k+1}), t(m_k, d_{k+1}) \quad \text{for } 0 \leq k < n. \end{aligned}$$

These transitions define the regular structure of a PHMM. An example of a PHMM is shown in Figure 5.6. The

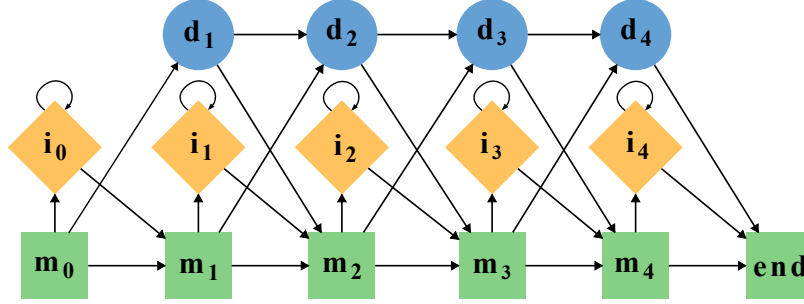


Figure 5.6: A repetitive structure of a PHMM. Inspired by [29].

The states  $m_0$ ,  $d_k$  for  $1 \leq k \leq n$ , and  $end$  are *silent*, meaning they do not emit any symbols. The emission probabilities of silent states are zero. Hence, the condition

$$\sum_{o \in O} E[o, s] = 1$$

does not hold for silent states. Furthermore, when computing the probability of generating sequence  $x$  while taking path  $y$ ,  $P\{x, y \mid \Theta\}$ , the length of the emitted symbol sequence  $x$  does not have to be equal to the length of the path  $y$ . Let  $y = y_1 y_2 \dots y_K \in S^*$ ,  $K \geq L$  be a state path and let  $y' = y'_1 y'_2 \dots y'_L$  be a sequence of non-silent states in  $y$ . Then, the probability of generating  $x = x_1 x_2 \dots x_L$  while taking path  $y$  is:

$$P\{x, y \mid \Theta\} = P\{x \mid y, \Theta\} \cdot P\{y \mid \Theta\},$$

where:

$$\begin{aligned} P\{x \mid y, \Theta\} &= \prod_{i=1}^L E[x_i, y'_i], \\ P\{y \mid \Theta\} &= \pi(y_1) \cdot \prod_{i=2}^K t(y_{i-1}, y_i). \end{aligned}$$

## Constructing a profile hidden Markov model

The PHMM is built based on a multiple sequence alignment, which can be obtained, for example, using tools such as Clustal Omega [31]. The alignment consists of several columns. The column is considered a match column if there are more matches and mismatches than gaps. Each match column is modelled by a triplet of a match, an insert, and a delete state.

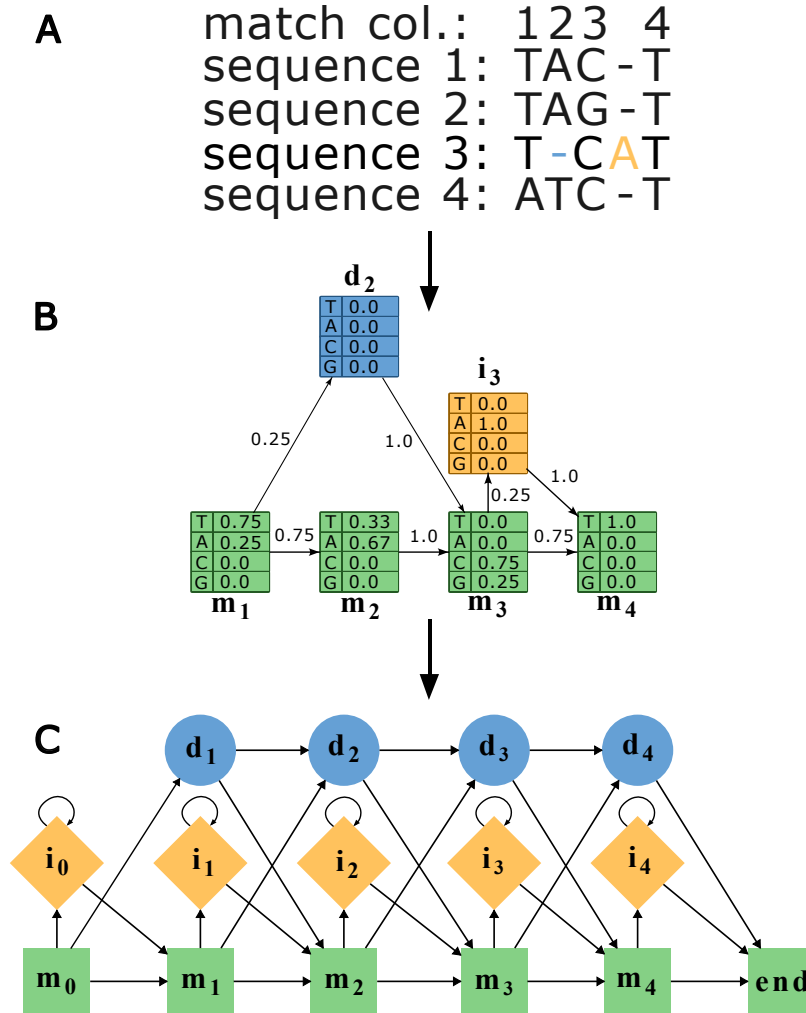


Figure 5.7: A schematic illustration of constructing a simple PHMM. Inspired by [29]. (A) A multiple sequence alignment. (B) The calculated emission and transition probabilities. (C) The resulting PHMM.

The emission probabilities of the match and insert states are determined based on the symbol frequencies at each alignment column. For example, in the model shown in Figure 5.7, the emission probability  $E[T, m_1] = 0.75$ , since in the first column, three out of four symbols are Ts. In the fourth column of the alignment, there is only one non-gap symbol, so the column is represented by state  $i_3$  with emission probability  $E[A, i_3] = 1.0$  because the only inserted symbol is A. The rest of the emission probabilities are calculated similarly.

The transition probabilities are calculated by counting the relative transition frequencies for each state. For example, there is one deletion in the second column, so  $t(m_1, d_2) = 0.25$  and  $t(m_1, m_2) = 0.75$ . In this state, the model would only generate the sequences that

occurred in the alignment. Thus, a small count is added to every transition and symbol frequency to allow further mutations.

### Finding the optimal path using the Viterbi algorithm

One of the essential tasks when using an HMM is to find the optimal path  $y = y_1 y_2 \dots y_L$  with the highest observation probability of a symbol sequence  $x = x_1 x_2 \dots x_L$ . For a classical HMM without silent states, each position in the state path can contain any of the HMM states. Thus, the total number of possible state sequences that must be compared is  $M^L$ , where  $M$  denotes the number of states and  $L$  is the length of the sequence  $x$ . Using a naive method is computationally unsuitable.

However, the Viterbi algorithm [36], based on dynamic programming, reduces the time complexity to  $O(LM^2)$  by recursively selecting only the most probable paths. It uses a variable  $\gamma(n, i)$ , representing the maximum probability of generating sequence  $x_1 x_2 \dots x_n$ , while ending in state  $i$  at position  $n$ . It is defined as:

$$\gamma(n, i) = \max_{y_1 \dots y_{n-1}} P(x_1 \dots x_n, y_1 \dots y_{n-1} y_n = i \mid \Theta).$$

The value of  $\gamma(n, i)$  is computed recursively as:

$$\gamma(n, i) = \max_k [\gamma(n-1, k) \cdot t(k, i) \cdot E[x_n, i]].$$

The maximum observation probability of the sequence  $x$  can be, at the end, computed as:

$$\max_y P\{x, y \mid \Theta\} = \max_k \gamma(L, k).$$

The desired optimal state sequence can be found by backtracking the transitions that led to the maximum observation probability. HMMER finds the optimal sequence alignment as the most probable path that generated the query sequence by using a slightly modified version of the Viterbi algorithm, adjusted to handle the silent states [12].

## Chapter 6

# ALERGIA Algorithm in Biological Sequence Analysis

The aim of this thesis is to explore the application of finite automata in biological sequence analysis, with a particular focus on their potential for detecting structural domains of LTR retrotransposons. This chapter briefly describes the possibilities of using the ALERGIA algorithm [7] in this area.

Let us have a database of amino acid sequences of one type of protein. Given that all these sequences are supposed to perform the same function, they should be similar in a certain way and thus contain some pattern. Although it is not possible to use exact matching algorithms to determine whether a particular sequence might be the desired protein due to naturally occurring mutations, we can use a PFA to describe the character of the entire set of sequences and then introduce some randomness by merging similar states. That can be achieved using the ALERGIA algorithm described in Section 6.1. Then, it could be decided, in linear time, whether the query sequence encodes the given protein, assuming that the learned DPFA is already available.

### 6.1 ALERGIA

ALERGIA [7] is an algorithm for learning a probabilistic finite automaton from a set of sample words using a state-merging technique. This section will describe its principles in more detail, but first, we must define the necessary terms. The following definitions and algorithms are taken from [17].

#### Deterministic frequency finite automaton

Given that *frequency* is the number of times an event occurs, a *deterministic frequency finite automaton (DFFA)* is a tuple  $\mathcal{A} = (Q, \Sigma, \delta, I_f, F_f, \delta_f)$ , where:

- $Q$  is a finite set of *states*,
- $\Sigma$  is an input *alphabet*,
- $\delta : Q \times \Sigma \rightarrow Q$  is a *transition function*,
- $I_f : Q \rightarrow \mathbb{N}$  is a function assigning *initial state frequencies*. Since the automaton is deterministic there is exactly one state  $q_\epsilon$  for which  $I_f(q_\epsilon) \neq 0$ ,

- $F_f : Q \rightarrow \mathbb{N}$  is a function assigning *final state frequencies*,
- $\delta_f : Q \times \Sigma \times Q \rightarrow \mathbb{N}$  is a *transition frequency function*.

The notation  $\delta_f(p, a, q)$ , where  $p, q \in Q$  and  $a \in \Sigma$ , can be interpreted as the number of times a transition  $p \xrightarrow{a} q$  is used. Because of the determinism, for every state  $p$  and symbol  $a$ , there is at most one state  $q$  such that  $\delta_f(p, a, q) > 0$ , and this state is such that  $\delta(p, a) = q$ .  $I_f(q)$  denotes the number of words starting in state  $q$ . By  $F_f(q)$ , we denote the number of words ending in state  $q$ . For a *well-defined* DFFA, the number of words entering and leaving a given state  $q$ ,  $FREQ(q)$ , must be identical:

$$\forall q \in Q : FREQ(q) = I_f(q) + \sum_{p \in Q, a \in \Sigma} \delta_f(p, a, q) = F_f(q) + \sum_{r \in Q, a \in \Sigma} \delta_f(q, a, r).$$

### Frequency prefix tree acceptor

Let  $S$  be a multiset of sample words from  $\Sigma^*$  and let  $PREF(S) = \{u \in \Sigma^* : uv \in S\}$  be a set of all prefixes of all words in  $S$ . Let  $|S|_r$  be the number of words in  $S$  matching the regular expression  $r$ . The *frequency prefix tree acceptor* of  $S$ ,  $FPTA(S)$ , is a DFFA defined as  $FPTA(S) = (Q, \Sigma, \delta, I_f, F_f, \delta_f)$ , where:

- $Q = \{q_u : u \in PREF(S)\}$ ,
- $I_f(q_\epsilon) = |S|$ ,
- $\forall ua \in PREF(S) : \delta(q_u, a) = q_{ua}$ ,
- $\forall ua \in PREF(S) : \delta_f(q_u, a, q_{ua}) = |S|_{ua\Sigma^*}$ ,
- $\forall u \in PREF(S) : F_f(q_u) = |S|_u$ .

### ALERGIA algorithm

The input of the ALERGIA algorithm is a multiset of sample words  $S$  and two parameters  $\alpha$  and  $t_0$ , which affect how much merging is done. Parameter  $\alpha$  specifies how similar the states must be to be merged, and  $t_0$  defines the minimum number of words passing through a state to consider a merge. First, an FPTA representing this set of words is created.

Two sets of states, *BLUE* and *RED*, are maintained throughout the process. The *RED* set contains states that have already been processed and thus are part of the final PFA. The states to be considered for merging are stored in the *BLUE* set.

The sets are initialised as follows: the *RED* set contains only the initial state, and the *BLUE* set contains all immediate successors of the initial state. Next, the lexicographically minimal *blue* state is chosen, and if there is a sufficiently similar *red* state, the states are merged. If no compatible *red* state is found, the *blue* state is added to the *RED* set. This procedure is repeated until no further state merging is possible. This process is described in Algorithm 2.

---

**Algorithm 2: ALERGIA**

---

**Input:**  $S$  – a multiset of sample words from  $\Sigma^*$ ,  $\alpha$  – a merging parameter,  
 $t_0$  – the minimal state frequency for the test to be statistically significant  
**Output:**  $\mathcal{A}$  – a DFFA

- 1  $\mathcal{A} \leftarrow FPTA(S)$ ;
- 2  $RED \leftarrow \{q_\epsilon\}$ ;
- 3  $BLUE \leftarrow \{q_a : a \in \Sigma \cap PREF(S)\}$ ;
- 4 **while** choose  $blue \in BLUE$  such that  $FREQ(blue) \geq t_0$  **do**
- 5     **if**  $\exists red \in RED : ALERGIA-COMPATIBLE(A, red, blue, \alpha)$  **then**
- 6          $\mathcal{A} = STOCHASTIC-MERGE(\mathcal{A}, red, blue)$ ;
- 7     **else**
- 8          $RED \leftarrow RED \cup \{blue\}$ ;
- 9          $BLUE \leftarrow \{q_{ua} : ua \in PREF(S) \wedge q_u \in RED\} \setminus RED$ ;
- 10 **return**  $\mathcal{A}$ ;

---

For the  $blue$  state, it must apply that  $FREQ(blue) \geq t_0$ . Thus, the first one that satisfies this condition is selected. The  $red$  state is chosen as the first one sufficiently similar to the  $blue$  state. That is achieved using Algorithm 4. Two states are compatible if the quotients:

$$\frac{F_f(red)}{FREQ(red)} \quad \text{and} \quad \frac{F_f(blue)}{FREQ(blue)}$$

and also for each symbol  $a \in \Sigma$ :

$$\frac{\delta_f(red, a)}{FREQ(red)} \quad \text{and} \quad \frac{\delta_f(blue, a)}{FREQ(blue)}$$

are all sufficiently similar. To check the similarity, Algorithm 3 is used.

---

**Algorithm 3: ALERGIA-TEST**

---

**Input:**  $f_1, n_1, f_2, n_2, \alpha > 0$   
**Output:** a boolean indicating if the frequencies  $\frac{f_1}{n_1}$  and  $\frac{f_2}{n_2}$  are sufficiently similar

- 1 **return**  $|\frac{f_1}{n_1} - \frac{f_2}{n_2}| < (\sqrt{\frac{1}{n_1}} + \sqrt{\frac{1}{n_2}}) \cdot \sqrt{\frac{1}{2}} \cdot \ln(\frac{2}{\alpha})$

---

---

**Algorithm 4: ALERGIA-COMPATIBLE**

---

**Input:**  $\mathcal{A} = (Q, \Sigma, \delta, I_f, F_f, \delta_f)$  – a DFFA, two states:  $p, q \in Q$ ,  $\alpha > 0$  – a merging parameter  
**Output:** a boolean indicating if the states  $p$  and  $q$  are compatible

- 1 **if** not  $ALERGIA-TEST(F_f(p), FREQ(p), F_f(q), FREQ(q), \alpha)$  **then**
- 2     **return** *False*
- 3 **foreach**  $a \in \Sigma$  **do**
- 4     **if** not  $ALERGIA-TEST(\delta_f(p, a), FREQ(p), \delta_f(q, a), FREQ(q), \alpha)$  **then**
- 5         **return** *False*
- 6 **return** *True*

---

The STOCHASTIC-MERGE algorithm works as follows. First,  $pred_{blue}$ , the immediate predecessor of the  $blue$  state and symbol  $a \in \Sigma$  such that  $\delta(pred_{blue}, a) = blue$ , are found. Then, the transition from  $pred_{blue}$  to  $blue$  is redirected to the  $red$  state. Hence, the divided automaton must now be reconnected, which is achieved using Algorithm 6.

---

**Algorithm 5: STOCHASTIC-MERGE**

---

**Input:**  $\mathcal{A}$  – a DFFA, two states:  $red \in RED$  and  $blue \in BLUE$

**Output:** updated  $\mathcal{A}$ , where the  $blue$  state was merged with the  $red$  state

```
1 Let  $(pred_{blue}, a)$  be such that  $\delta(pred_{blue}, a) = blue$ ;  
2  $n \leftarrow \delta_f(pred_{blue}, a, blue)$ ;  
3  $\delta(pred_{blue}, a) \leftarrow red$ ;  
4  $\delta_f(pred_{blue}, a, red) \leftarrow n$ ;  
5  $\delta_f(pred_{blue}, a, blue) \leftarrow 0$ ;  
6 return STOCHASTIC-FOLD( $\mathcal{A}, red, blue$ );
```

---

To fold state  $blue$  into  $red$ , we must adjust the acceptance frequency of  $red$  and the frequencies of all transitions from  $red$  that may be affected accordingly. Then, the successors of  $blue$  must be folded recursively.

---

**Algorithm 6: STOCHASTIC-FOLD**

---

**Input:**  $\mathcal{A} = (Q, \Sigma, \delta, I_f, F_f, \delta_f)$  – a DFFA, two states:  $red \in RED$  and  $blue \in BLUE$

**Output:** updated  $\mathcal{A}$ , where the subtree in  $blue$  is folded into  $red$

```
1  $F_f(red) \leftarrow F_f(red) + F_f(blue)$ ;  
2 foreach  $a \in \Sigma$  such that  $\delta(blue, a)$  is defined do  
3   if  $\delta(red, a)$  is defined then  
4      $\delta_f(red, a, \delta(red, a)) \leftarrow \delta_f(red, a, \delta(red, a)) + \delta_f(blue, a, \delta(blue, a))$ ;  
5      $\mathcal{A} \leftarrow$  STOCHASTIC-FOLD( $\mathcal{A}, \delta(red, a), \delta(blue, a)$ );  
6   else  
7      $\delta(red, a) \leftarrow \delta(blue, a)$ ;  
8      $\delta_f(red, a, \delta(red, a)) \leftarrow \delta_f(blue, a, \delta(blue, a))$ ;  
9 return  $\mathcal{A}$ ;
```

---

The DFFA generated by the ALERGIA algorithm can be easily converted to a DPFA by transforming the DFFA frequencies into probabilities using Algorithm 7.

---

**Algorithm 7: DFFA-TO-DPFA**

---

**Input:**  $\mathcal{A} = (Q, \Sigma, \delta, I_f, F_f, \delta_f)$  – a DFFA

**Output:**  $\mathcal{B} = (Q, \Sigma, \delta_p, I_p, F_p)$  – a corresponding DPFA

```
1 foreach  $q \in Q$  do  
2   if  $I_f(q) = 0$  then  
3      $I_p(q) \leftarrow 0$ ;  
4   else  
5      $I_p(q) \leftarrow 1$ ;  
6   if  $FREQ(q) > 0$  then  
7     foreach  $a \in \Sigma$  do  
8        $p \leftarrow \delta(q, a)$ ;  
9        $\delta_p(q, a, p) \leftarrow \frac{\delta_f(q, a, p)}{FREQ(q)}$ ;  
10       $F_p(q) \leftarrow \frac{F_f(q)}{FREQ(q)}$ ;  
11 return  $\mathcal{B}$ ;
```

---

## 6.2 Experimenting with ALERGIA

The ability of the ALERGIA algorithm to identify a given gene must be assessed before attempting to use it to search for that gene in a longer query sequence. Therefore, different settings of the parameters  $\alpha$  and  $t_0$  were tested. The Detano tool [15] implements the ALERGIA algorithm. Thus, it was used to perform the experiments. The training set contains sequences of Ty1-GAG\_Class\_I::LTR::Ty1/copia::Tork from the database used by the TE-greedy-nester tool [21]. Two sets of sequences were generated: variants of the gag gene that were 80% identical to the sequences from the training set and random sequences as a negative control. Each of the sets contained 1000 sequences of the length of 90 amino acids, corresponding to the length of the gag gene. The following metrics were used for the evaluation:

- $Sensitivity = \frac{TP}{TP+FN}$ ,
- $Specificity = \frac{TN}{TN+FP}$ ,

where:

- $TP$  is the number of sequences from the gene variants set correctly classified as positive,
- $TN$  is the number of sequences from the negative control set correctly classified as negative,
- $FP$  is the number of sequences from the negative control set incorrectly classified as positive,
- $FN$  is the number of sequences from the gene variants set incorrectly classified as negative.

The results are shown in Figure 6.1. As can be noticed, the sensitivity is insufficient, even though the sequences in the testing set were 80% identical to the sequences in the training set.

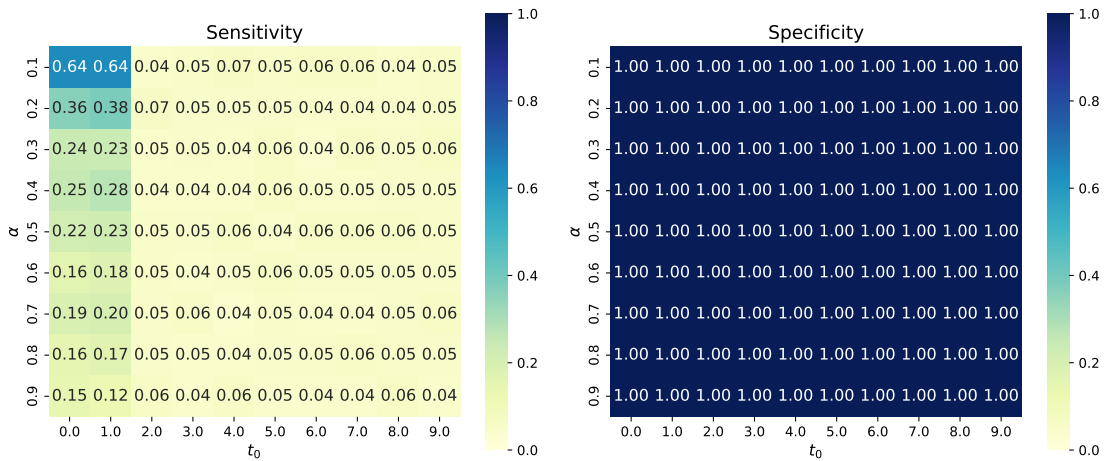


Figure 6.1: Sensitivity and specificity of the ALERGIA algorithm as functions of  $\alpha$  and  $t_0$ .

## Translation of amino acids into equivalence classes

Amino acids, the basic building blocks of proteins, can be divided into several groups according to their properties, such as polarity or acidity. Substituting an amino acid with another amino acid from the same group may lead to a conservative replacement. It has been proven that these mutations are much more frequent because they do not cause a significant change in the functionality of the protein [27]. Figure 6.3 shows two possible divisions into equivalence classes.

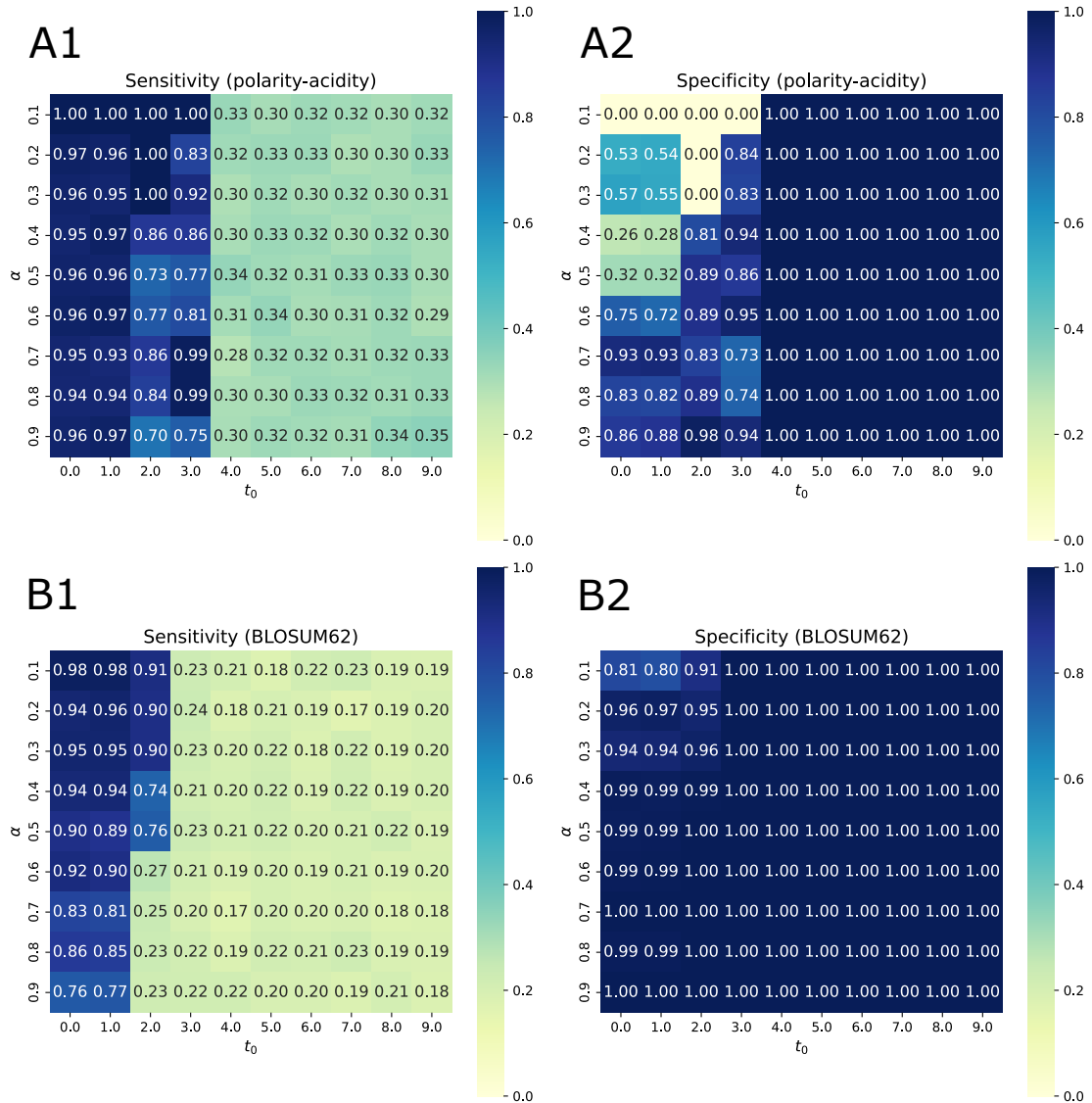


Figure 6.2: A comparison of different types of amino acid translation. (A1, A2) The results of using the translation according to polarity and acidity. (B1, B2) The results of using the translation according to the BLOSUM62 matrix.

Because the original ALERGIA algorithm did not prove to be very effective for this application, the same method of transcoding amino acids into these equivalence classes was used as in [30]. By reducing the size of the alphabet, the sensitivity was increased

significantly. The results are depicted in Figure 6.2. However, if we want to use this model for searching, the specificity needs to be as high as possible. Otherwise, we will get a large number of false positives. The  $\alpha$  and  $t_0$  settings that have sufficient specificity show low sensitivity, even though the testing sequences were 80% similar to the sequences in the training set. Therefore, it was necessary to find a way to improve the sensitivity further. The proposed method is described in Section 6.3

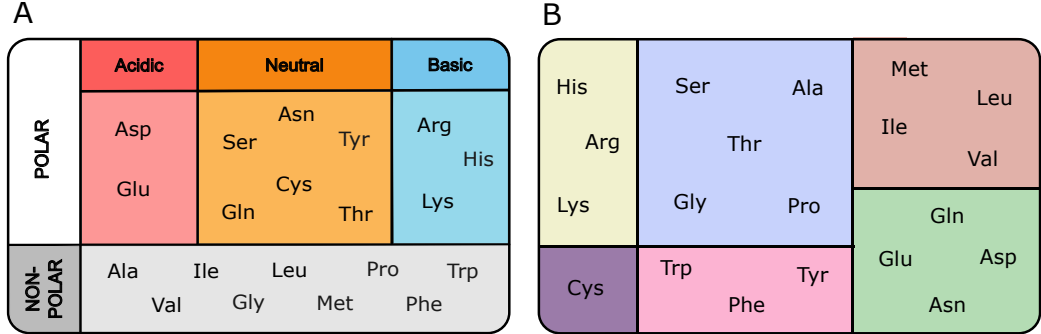


Figure 6.3: Two of the possible divisions into equivalence classes. (A) Division according to polarity and acidity. Inspired by [30]. (B) Division according to the BLOSUM62 matrix [38].

### 6.3 Dealing with Nondeterminism

The ALERGIA algorithm is relatively sensitive to the parameter  $\alpha$  and  $t_0$  settings. In our experiments, unfortunately, it either causes excessive state merging, resulting in many false positives or fails to simulate the frequent mutations. The insufficient sensitivity can be partially solved if we sometimes allow the automaton to take a zero-probability transition to one of the state’s direct successors. This effectively models substitutions that the ALERGIA algorithm did not learn. To define how often such transitions can be made, we introduce two bounds,  $b_t$  and  $b_c$ . The bound  $b_t$  defines the maximum number of zero-probability transitions taken in total, while  $b_c$  is the maximum number of zero-probability transitions taken consecutively. Unfortunately, allowing these transitions, modelling substitutions, introduces nondeterminism into the computation.

Furthermore, the DPFA generated by ALERGIA can only recognise sequences that exactly correspond to the given protein. In order to identify genes that are part of a longer sequence, the DPFA must be modified. Adding a self-loop in the initial state over the entire alphabet is necessary to read the symbols before the given gene. In addition, transitions from each final state to the initial state must be added to allow multiple occurrences of the given gene. These transitions introduce further nondeterminism into the automaton.

The automaton learned by ALERGIA consists of thousands of states and tens of thousands of transitions, making the determinization computationally unfeasible. Thus, a more complex search algorithm must be used. One possibility is to continuously compute the sets of states, in which the automaton can be after reading a prefix of the input sequence. This process is similar to Thompson’s algorithm [35], where the current set of possible states is updated at each iteration according to the current input symbol. It is analogous to the determinization algorithm described in Section 3.1. However, instead of generating the whole DFA, only the part that is needed during the matching algorithm is computed.

Nevertheless, this set would grow unacceptably due to the initial state self-loop, the tree-like structure of the NFA, and the introduced zero probability transitions. Therefore, its size needs to be reduced somehow. The runs that contain many zero-probability transitions do not need to be examined because they are not similar to the searched sequence. Therefore, exceeding one of the introduced bounds causes the corresponding state to be removed from the set.

To keep track of how many zero-probability transitions were made, instead of a set of possible states, a set of tuples of the form  $(q, pos, t, c)$  is maintained during the search algorithm, where:

- $q \in Q$  is the current state,
- $pos$  is the position in the query sequence where the corresponding run left the initial state (a candidate starting position of the searched gene),
- $t$  is the current number of zero-probability transitions taken in total,
- $c$  is the current number of zero-probability transitions taken consecutively.

The number of DPFA states limits the size of the set from above to prevent further set expansion. If a collision occurs, the tuple with the longer string read up to that point is selected. The proposed on-the-fly gene search is described in Algorithm 8.

The variable  $S$  is used to store the set of such tuples. By  $S'$  we denote the updated set that will be used in the next iteration. The variable  $i$  is used to store the index of the current input symbol. For each symbol of the query sequence, every tuple of the set has to be processed (Line 7). If the state  $q$  of the current tuple  $(q, pos, t, c)$  has a non-zero final state probability, the position  $pos$  is added to the set of found gene starting positions (Line 9).

By  $R$  we denote the set of direct successors of state  $q$  (Line 10). All the successors  $r \in R$  are processed (Line 11). If there exists a non-zero-probability transition from  $q$  to  $r$  over the current symbol  $a$  (Line 12),  $c$  is reset to zero and if there is no tuple with the same state  $r$ , the tuple  $(r, pos, t, 0)$  is added to  $S'$  (Line 17). If a collision occurs, the tuple with the longer string read up to that point is selected (Line 14). The same is done for the zero-probability transitions, except both  $t$  and  $c$  are incremented, and their values need to be tested against the bounds  $b_t$  and  $b_c$  before adding the new tuple to  $S'$  (Line 20 and 23). To ensure that the search sequence can start at any position, a tuple containing the *initial* state must be added to  $S$ , if not already present (Line 27).

---

**Algorithm 8: ON-THE-FLY-GENE-SEARCH**

---

**Input:**  $\mathcal{A} = (Q, \Sigma, \delta_p, I_p, F_p)$  – a DPFA generated by the ALERGIA algorithm, *sequence* – a query sequence,  $b_t$  – the maximal number of zero-probability transition taken in total,  $b_c$  – the maximal number of consecutive zero-probability transitions

**Output:** *positions* – a set of found gene positions

```
1 positions  $\leftarrow \emptyset$ ;
2 let initial  $\in Q$  be a state such that  $I_p(\textit{initial}) = 1$ ;
3  $S \leftarrow \{(\textit{initial}, 0, 0, 0)\}$ ;
4  $i \leftarrow 0$ ;
5 foreach symbol  $a$  in sequence do
6    $S' \leftarrow \emptyset$ ;
7   foreach  $(q, pos, t, c) \in S$  do
8     if  $F_p(q) > 0$  then
9        $positions \leftarrow positions \cup \{pos\}$ ;
10       $R \leftarrow \{r : \exists b \in \Sigma \text{ such that } \delta_p(q, b, r) > 0\}$ ;
11      foreach state  $r \in R$  do
12        if  $\delta_p(q, a, r) > 0$  then
13          if  $\exists (r', pos', t', c') \in S' : r' = r$  then
14            if  $pos < pos'$  then
15               $S' \leftarrow (S' \setminus (r', pos', t', c')) \cup (r, pos, t, 0)$ ;
16            else
17               $S' \leftarrow S' \cup (r, pos, t, 0)$ ;
18          else
19            if  $\exists (r', pos', t', c') \in S' : r' = r$  then
20              if  $pos < pos' \wedge t + 1 \leq b_t \wedge c + 1 \leq b_c$  then
21                 $S' \leftarrow (S' \setminus (r', pos', t', c')) \cup (r, pos, t + 1, c + 1)$ ;
22              else
23                if  $t + 1 \leq b_t \wedge c + 1 \leq b_c$  then
24                   $S' \leftarrow S' \cup (r, pos, t + 1, c + 1)$ ;
25       $i \leftarrow i + 1$ ;
26      if  $\neg \exists (initial, pos', t', c') \in S'$  then
27         $S \leftarrow S' \cup (initial, i, 0, 0)$ ;
28 return positions
```

---

## 6.4 Results

The introduced zero-probability transitions improved the sensitivity, but with most combinations of the  $\alpha$  and  $t_0$  parameters, they led to a large number of false positives. This did not occur only in cases where almost no state merging was performed.

At the same time, the higher the parameters  $b_c$  and  $b_t$  are, the larger the tuple sets are and the slower the algorithm is. The speed is most affected by the parameter  $b_c$ . The lowest setting that allowed finding the most occurrences of homologous sequences was  $b_c = 2$ . The speed of this setting was compared with the speed of the BLASTX tool. The results are shown in Figure 6.4.

Unfortunately, even this fastest setting did not lead to a speedup of the search. This was due to the fact that the average size of the set maintained throughout the process remained

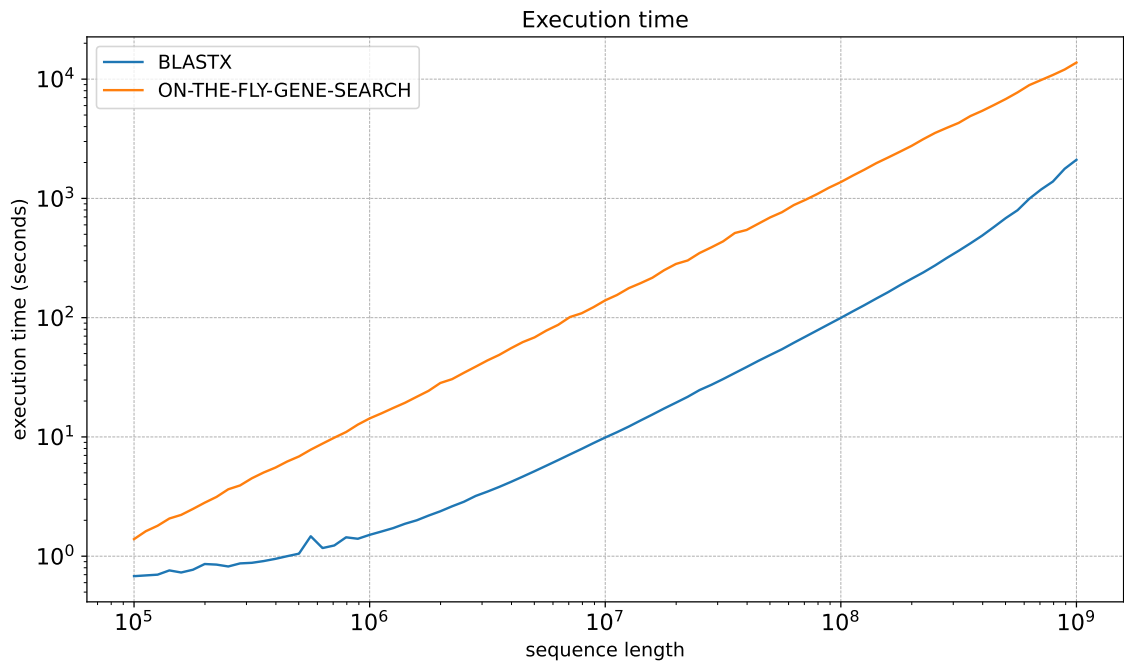


Figure 6.4: A comparison of execution times of the proposed ON-THE-FLY-GENE-SEARCH and BLASTX as functions of the query sequence length.

around ten elements. Although the current version is a prototype, it is unlikely that even a more optimised implementation would yield a significant improvement in performance compared to the BLASTX tool. Therefore, the second part of the thesis focuses on an alternative approach based on deterministic finite automata (DFAs), which has a greater potential to compete with the BLASTX tool.

## Chapter 7

# Proposed Gene Search Algorithm

This chapter introduces a method to convert a profile hidden Markov model (PHMM) into a set of relatively small deterministic finite automata (DFAs), which can be used for efficient LTR retrotransposon domain search. HMMER is very good at modelling the character of the searched sequence because it also considers that some positions are more prone to mutations than others. Furthermore, it precisely models all types of mutations, including insertions and deletions. Thanks to that, HMMER can detect even highly remote homologues [12].

However, the main disadvantage of PHMMs is that they are nondeterministic. Pattern matching using a nondeterministic automaton is possible, but it is computationally expensive due to the large number of different runs that must be evaluated. Dynamic programming algorithms, such as the Viterbi algorithm, reduce the time complexity; however, compared to the deterministic alternatives, the execution time is still relatively high.

### 7.1 Process of Generating the Gene Model

The main idea of this work is that if a PHMM could be transformed into a simplified DFA while minimizing the loss of accuracy, it would be possible to search for homologous sequences with a time complexity of  $O(L)$ . Unfortunately, direct determinization is unfeasible, as it would result in an unacceptably large automaton. An attempt was made to determinize the PHMM using the Mata library [9], but the process ran for nearly two weeks and consumed close to 30 GB of memory without completion. This explosion in the number of states of the resulting automaton occurred despite the original PHMM having only around 180 states. The explosion is caused by the highly nondeterministic structure of the PHMM, which is necessary for modelling mutations such as insertions, deletions, and substitutions.

However, there are several ways to significantly reduce the size of the resulting automaton while maintaining a reasonable accuracy. This section introduces a promising technique for transforming a PHMM into a set of relatively small, simplified DFAs. The key steps of this transformation are depicted in Figure 7.1, but they are explained in more detail in the following subsections.

#### Discretization of transition probabilities

Transition and emission probabilities of a PHMM describe the character of the searched gene very well. However, these probabilities must first be discretized to construct a simplified

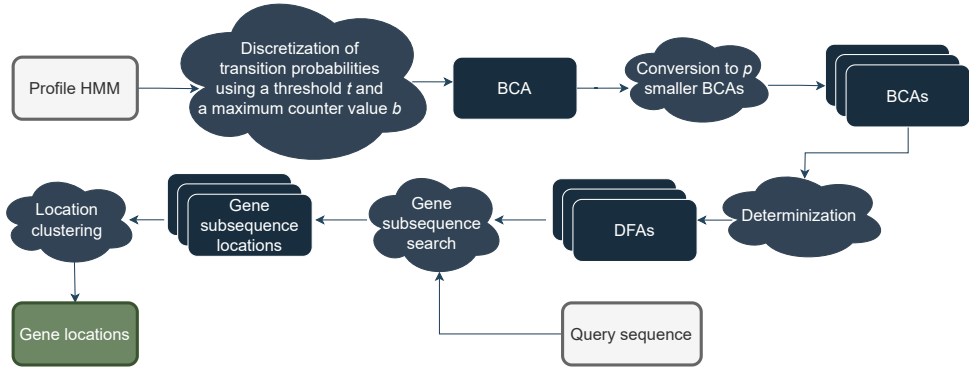


Figure 7.1: An illustration of the algorithm used for generating multiple DFAs from a single profile HMM. Each of the steps is described in more detail in the following subsections. Once the DFAs are created, we can use them to find the locations of corresponding subsections of the searched gene.

deterministic model. For this purpose, a threshold  $t$  is introduced. If the probability that a given symbol occurs at a given position is greater than or equal to  $t$ , we regard it as a *hit* and the transition that generated the symbol as a *hit transition*. Otherwise, we regard it as a *miss* or a *miss transition*.

However, a strict discretization may result in a loss of relevant information, leading to a significant model degradation. Thus, when searching for a gene, it is necessary to allow even the miss transitions to simulate naturally occurring mutations. At the same time, the hit transitions need to be prioritized. Therefore, we introduced a bounded counting automaton (BCA) described in Section 3.1. A BCA has one counter  $c$  updated when a transition is made. The counter determines how many consecutive transitions below the threshold  $t$  were taken. In addition, to prioritise the alignments that do not contain gaps, the transitions below  $t$ , that model insertions and deletions, *gap transitions*, were assigned an additional penalty based on the gap length  $g$ . Then, the counter update  $u$  of a transition  $p\{a,u\}\rightarrow q$  is defined as:

$$u = \begin{cases} c := c + 1 & \text{if } p\{a,u\}\rightarrow q \text{ is a miss transition} \\ c := 0 & \text{if } p\{a,u\}\rightarrow q \text{ is a hit transition} \\ c := c + 2 \cdot g & \text{if } p\{a,u\}\rightarrow q \text{ is a gap transition.} \end{cases}$$

To ensure that the automaton only accepts sequences sufficiently similar to the sequences in the database, we limited the maximum counter value to  $b$ .

The process of converting a PHMM, generated by HMMER, into a BCA is described in Algorithm 9. Only the match states are preserved (Line 4). They are converted into a linear sequence of states, which are connected by transitions for each symbol in the alphabet. The probabilities of these transitions are computed as a product of the relevant transition and emission probabilities. The transition resets the counter (Line 10) if the probability of emitting the symbol  $a$  at the given position  $k$ ,  $t(m_{k-1}, m_k) \cdot E[a, m_k]$ , is greater than or equal to  $t$ . Otherwise, the counter is incremented by 1 (Line 12). The insert states should be properly modelled using two states because, in PHMMs, the probability of gap opening differs from the probability of gap extension. However, the goal is to reduce the size of the model as much as possible so that it can be determinized later. Thus, the insertions

are modelled simply as a loop in the corresponding match state. The loop resets the counter if the probability of making the insertion is greater than or equal to  $t$  (Line 15) or otherwise increments the counter by 2 for each symbol inserted (Line 17).

---

**Algorithm 9:** PHMM-TO-BCA

---

**Input:**  $\mathcal{A} = (S, O, \pi, t, E)$  – a PHMM generated by HMMER,  $t$  – a threshold,  
 $b$  – a counter bound

**Output:**  $\mathcal{B} = (Q, \Sigma, \Delta, conf_0, F, b)$  – a corresponding BCA

```

1  $n \leftarrow |D|$ ;
2  $\Sigma \leftarrow O$ ;
3  $conf_0 \leftarrow (m_0, 0)$ ;
4  $Q \leftarrow M$ ;
5  $F \leftarrow \{m_n\}$ ;
6  $\Delta \leftarrow \emptyset$ ;
7 foreach symbol  $a \in O$  do
8   foreach match state  $m_k \in M, k \geq 1$  do
9     // converting match states
10    if  $t(m_{k-1}, m_k) \cdot E[a, m_k] \geq t$  then
11       $\Delta \leftarrow \Delta \cup \{m_{k-1} \{a, c:=0\} \rightarrow m_k\}$ ; // hit transition
12    else
13       $\Delta \leftarrow \Delta \cup \{m_{i-k} \{a, c:=c+1\} \rightarrow m_k\}$ ; // miss transition
14    // converting insert states
15    if  $k \neq n$  then
16      if  $t(m_k, i_k) \cdot E[a, i_k] \cdot t(i_k, m_{k+1}) \geq t$  then
17         $\Delta \leftarrow \Delta \cup \{m_k \{a, c:=0\} \rightarrow m_{k+1}\}$ ; // hit transition
18      else
19         $\Delta \leftarrow \Delta \cup \{m_k \{a, c:=c+2\} \rightarrow m_{k+1}\}$ ; // gap transition
20  foreach match state  $m_k \in M, k \geq 0$  do
21    // converting deletion states
22     $del\_start\_prob \leftarrow t(m_k, d_{k+1})$ ;
23     $del\_len \leftarrow 0$ ;
24    for  $l \leftarrow 2$  to  $n - k$  do
25      if  $del\_start\_prob \cdot t(d_{k+l-1}, m_{k+l}) \cdot E[a, m_{k+l}] \geq t$  then
26         $\Delta \leftarrow \Delta \cup \{m_k \{a, c:=0\} \rightarrow m_{k+l}\}$ ; // hit transition
27      else
28         $del\_len \leftarrow del\_len + 1$ ;
29         $\Delta \leftarrow \Delta \cup \{m_k \{a, c:=c+(2 \cdot del\_len)\} \rightarrow m_{k+l}\}$ ; // gap transition
30       $del\_start\_prob \leftarrow del\_start\_prob \cdot t(d_{k+l-1}, d_{k+l})$ ;
31 return  $\mathcal{B}$ ;

```

---

The deletion states are transformed into transitions leading to each subsequent state. This is achieved by the for loop starting on Line 21. The probability of deletion opening,  $del\_start\_prob$ , is set to the corresponding transition probability  $t(m_k, d_{k+1})$  (Line 19). The probability of making a deletion of length  $del\_len$  is computed as a product of the deletion opening probability, the corresponding deletion ending probability and emission probability of the corresponding symbol (Line 22). For the longer deletions, the  $del\_start\_prob$  is

iteratively multiplied by the transition probability leading to the next delete state (Line 27). The counter is again either reset (Line 23) or incremented by  $2 \cdot del\_len$  (Line 26) according to the probability of making the whole deletion and the threshold  $t$ .

An example of the resulting BCA is shown in Figure 7.2. As can be noticed, the insert states  $i_0$  and  $i_4$  were not used since the transitions modelling symbols before and after the searched sequence will be added later.

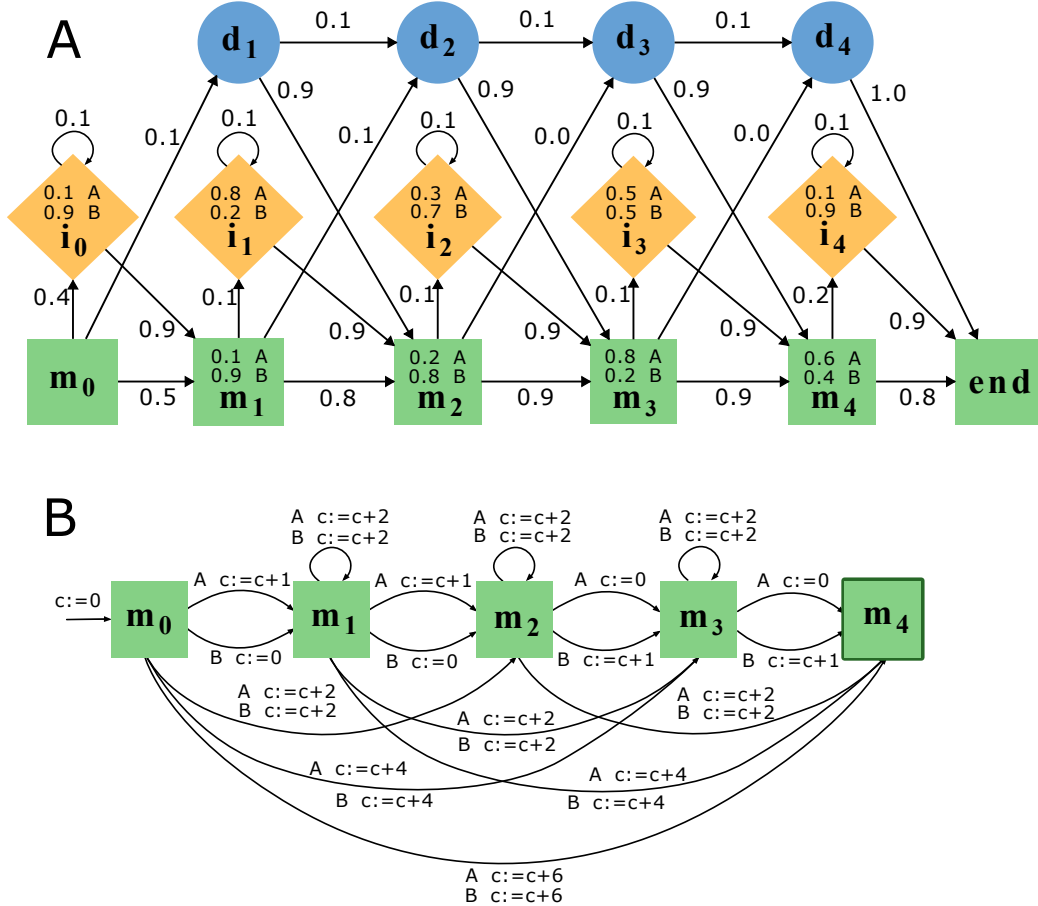


Figure 7.2: An illustration of PHMM to BCA conversion using  $t = 0.45$ . (A) A simple PHMM. (B) The resulting BCA.

### Bounded counting automaton partitioning

Since the determinization of the resulting BCA was still unfeasible, we took advantage of the linear structure of the PHMMs, which can easily be divided into  $p$  smaller parts. Each part represents a subsequence of the modelled gene and can be determinized separately. The process of generating  $p$  smaller BCAs is formalised in Algorithm 10. The starting and ending indices of the current part are stored in the *start* and *end* variables, respectively. Only the states and transitions that are within this computed range are preserved (Line 6 and 8). The last state of the part is used as the final state (Line 7). As already said, the leading and trailing loop transitions will be added later in a unified manner; thus, they are excluded from the set of transitions (Line 9).

---

**Algorithm 10: SPLIT-BCA**

---

**Input:**  $\mathcal{A} = (Q, \Sigma, \Delta, conf_0, F, b)$  – a BCA,  $p$  – the number of parts  
**Output:**  $BCAs$  – an array of  $p$  BCAs, each representing a subsequence of the searched gene

```
1  $start \leftarrow 0$ ;  
2  $end \leftarrow 0$ ;  
3 for  $i \leftarrow 0$  to  $p - 1$  do  
4    $part\_len \leftarrow \text{PART-LENGTH}(\mathcal{A}, p, i)$ ;  
5    $end \leftarrow end + part\_len$  ;  
6    $Q_i \leftarrow \{m_k : m_k \in Q \wedge start \leq k \leq end\}$ ;  
7    $F_i \leftarrow \{m_{end}\}$ ;  
8    $\Delta_i \leftarrow \{p \xrightarrow{a,u} q : p, q \in Q_i \wedge p \xrightarrow{a,u} q \in \Delta\}$ ;  
   // remove start and end self-loops  
9    $\Delta_i \leftarrow \Delta_i \setminus \{m_k \xrightarrow{a,u} m_k : k = start \vee k = end\}$ ;  
10   $BCAs[i] \leftarrow (Q_i, \Sigma, \Delta_i, (m_{start}, 0), F_i, b)$ ;  
11   $start \leftarrow end$ ;  
12 return  $BCAs$ 
```

---

The length of each subsequence is calculated using Algorithm 11. The parts should ideally be of the same length. Therefore, the remainder,  $n \bmod p$ , is evenly distributed among the first parts (Line 4).

---

**Algorithm 11: PART-LENGTH**

---

**Input:**  $\mathcal{A} = (Q, \Sigma, \Delta, conf_0, F, b)$  – a BCA,  $p$  – the number of parts,  
 $i \in \{0, 1, \dots, p - 1\}$  – a part index  
**Output:**  $part\_len$  – the length of the part number  $i$

```
1  $n \leftarrow |Q| - 1$ ;  
2  $part\_len \leftarrow n \text{ div } p$ ;  
3 if  $i < (n \bmod p)$  then  
4    $part\_len \leftarrow part\_len + 1$ ;  
5 return  $part\_len$ ;
```

---

Figure 7.3 shows an example of the resulting BCAs. Algorithm 12 is used to add the transitions necessary to use these small BCAs for searching the modelled sequence in a longer query. An initial state self-loop over the entire alphabet must be added (Line 3). In addition, transitions from the final to the initial state are added to allow multiple occurrences of the searched sequence (Line 4). Since these leading and trailing insertions should not influence the alignment score, the counter is reset when taking these transitions. The updated BCAs are shown in Figure 7.4.

## Determinization

The determinization of a BCA  $\mathcal{A}$  is relatively simple. As described in Section 3.1, the language of  $\mathcal{A}$ ,  $L(\mathcal{A})$ , is defined as the language of its configuration automaton  $Conf(\mathcal{A})$ , which

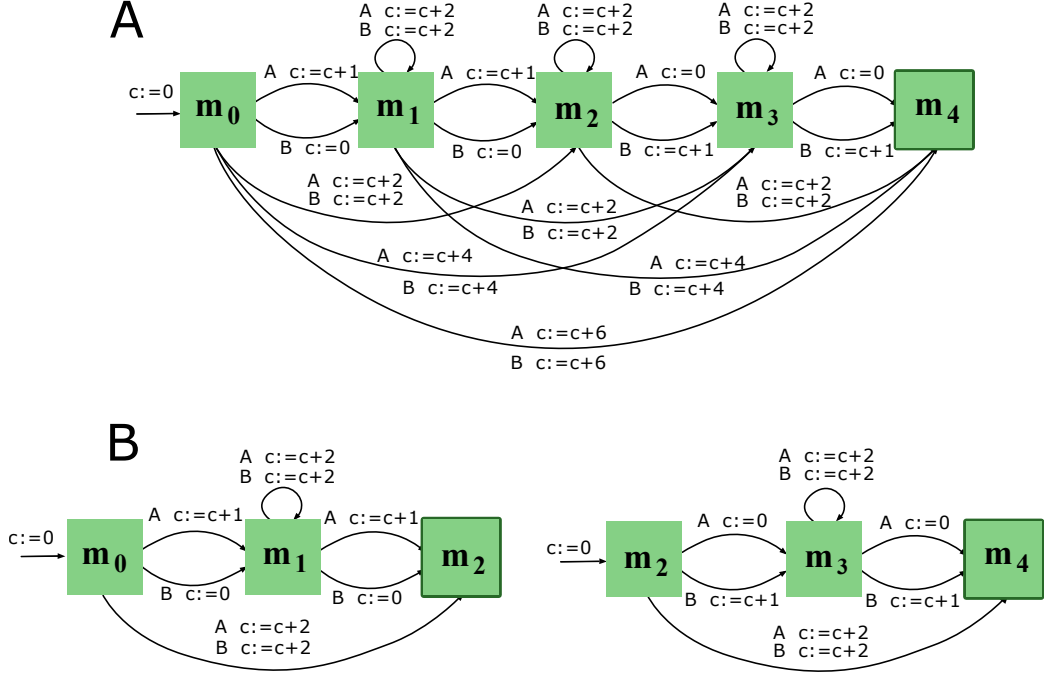


Figure 7.3: An illustration of BCA partitioning using  $p = 2$ . (A) A simple BCA. (B) The two resulting BCAs, each representing a subsequence of the modelled sequence.

---

**Algorithm 12: ADD-TRANSITIONS**

---

**Input:**  $\mathcal{A} = (Q, \Sigma, \Delta, conf_0, F, b)$  – a BCA  
**Output:** updated  $\mathcal{A}$

- 1  $(i, k) \leftarrow conf_0$ ;
- 2 **foreach** *symbol*  $a \in \Sigma$  **do**
- 3      $\Delta \leftarrow \Delta \cup \{i \xrightarrow{a, c:=0} i\}$ ;
- 4      $\Delta \leftarrow \Delta \cup \{f \xrightarrow{a, c:=0} i : f \in F\}$ ;
- 5 **return**  $\mathcal{A}$ ;

---

is an NFA whose states are configurations of  $\mathcal{A}$  of the form  $(q, k)$ , where  $q \in Q$  and  $k \in \mathbb{N}$  is a counter value. Thus, to determinize  $\mathcal{A}$ ,  $Conf(\mathcal{A})$  must be constructed using Algorithm 13. A data structure *queue* stores the reachable configurations of  $\mathcal{A}$ . The configurations from *queue* are regarded as the states of  $Conf(\mathcal{A})$  (Line 9). All transitions leading from the state of the current configuration are then processed. If the updated counter value does not exceed the bound  $b$ , the corresponding transition is added to  $\Delta_{Conf(\mathcal{A})}$  (Line 14) and the target configuration is enqueued, if not already processed (Line 16).

$Conf(\mathcal{A})$  is then determinized using Algorithm 1, described in Section 3.1. The language of the resulting DFA is equal to  $L(\mathcal{A})$ .

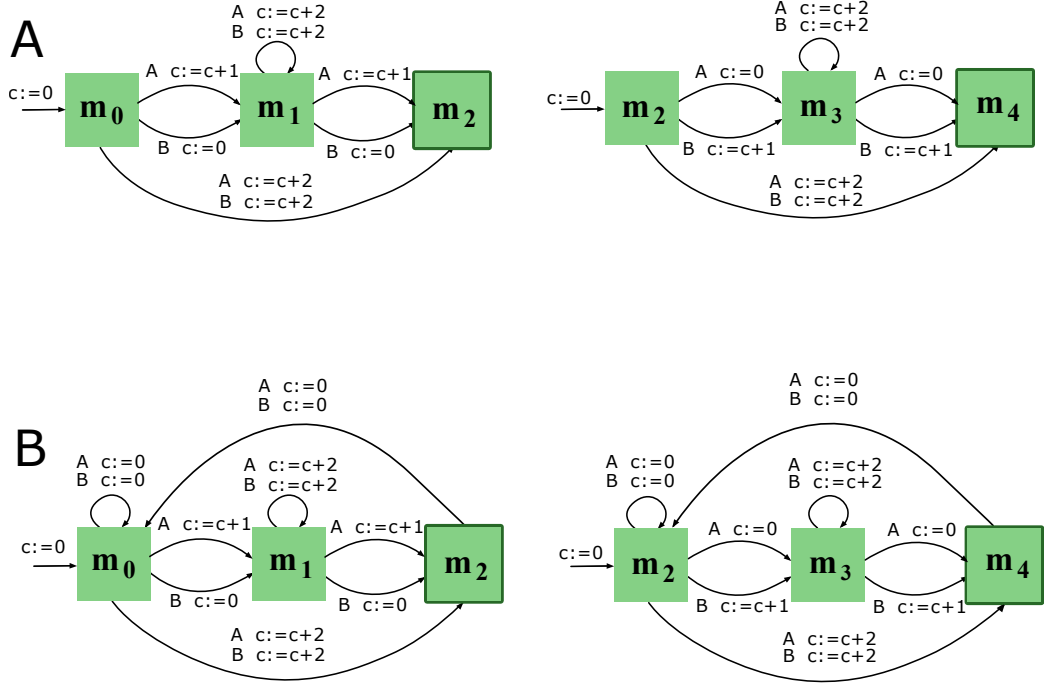


Figure 7.4: (A) BCAs representing subsequences of the modelled sequence. (B) The two resulting BCAs, that are able to search in a longer query.

---

**Algorithm 13: BCA-TO-CONF**

---

**Input:**  $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \Delta_{\mathcal{A}}, conf_{0_{\mathcal{A}}}, F_{\mathcal{A}}, b)$  – a BCA  
**Output:**  $Conf(\mathcal{A}) = (Q_{Conf(\mathcal{A})}, \Sigma, \Delta_{Conf(\mathcal{A})}, I_{Conf(\mathcal{A})}, F_{Conf(\mathcal{A})})$

- 1  $Q_{Conf(\mathcal{A})} \leftarrow \emptyset;$
- 2  $\Delta_{Conf(\mathcal{A})} \leftarrow \emptyset;$
- 3  $F_{Conf(\mathcal{A})} \leftarrow \emptyset;$
- 4  $I_{Conf(\mathcal{A})} \leftarrow \{conf_{0_{\mathcal{A}}}\};$
- 5 Queue *queue*;
- 6 *queue.enqueue* ( $conf_{0_{\mathcal{A}}}$ );
- 7 **while** *not queue.empty*() **do**
- 8      $(s, k) \leftarrow \text{queue.dequeue}();$
- 9      $Q_{Conf(\mathcal{A})} \leftarrow Q_{Conf(\mathcal{A})} \cup \{(s, k)\};$
- 10    **if**  $s \in F_{\mathcal{A}}$  **then**
- 11      $F_{Conf(\mathcal{A})} \leftarrow F_{Conf(\mathcal{A})} \cup \{(s, k)\};$
- 12    **foreach** transition  $s \xrightarrow{\{a, u\}} r \in \Delta_{\mathcal{A}}$  leading from  $s$  **do**
- 13     **if**  $u(k) \leq b$  **then**
- 14          $\Delta_{Conf(\mathcal{A})} \leftarrow \Delta_{Conf(\mathcal{A})} \cup \{(q, k) \xrightarrow{\{a\}} (r, u(k))\};$
- 15         **if**  $(r, u(k)) \notin Q_{Conf(\mathcal{A})}$  **then**
- 16              $\text{queue.enqueue}((r, u(k)));$
- 17 **return**  $Conf(\mathcal{A});$

---

## Gene model generation algorithm

The previous subsections explain the individual steps of gene model generation. Algorithm 14 integrates them into a single procedure for constructing the final gene model. First, the PHMM is converted into a BCA (Line 1). Then, the BCA is divided into  $p$  smaller BCAs (Line 2) and the leading and trailing transitions are added to each of the BCAs (Line 4). Next, the corresponding configuration automata are created (Line 5) and then determinized (Line 6). The model of the entire searched sequence is represented as a sorted array of the resulting DFAs.

---

### Algorithm 14: GENERATE-GENE-MODEL

---

**Input:**  $\mathcal{A} = (S, O, \pi, t, E)$  – a PHMM generated by HMMER,  $t$  – a threshold,  $b$  – a counter bound,  $p$  – the number of parts  
**Output:**  $DFAs$  – a sorted array of  $p$  DFAs modelling subsequences of the gene

- 1  $\mathcal{B} \leftarrow \text{PHMM-TO-BCA}(\mathcal{A}, t, b)$ ;
- 2  $BCAs \leftarrow \text{SPLIT-BCA}(\mathcal{B}, p)$ ;
- 3 **for**  $i \leftarrow 0$  **to**  $p - 1$  **do**
- 4      $BCAs[i] \leftarrow \text{ADD-TRANSITIONS}(BCAs[i])$ ;
- 5      $Conf(BCAs[i]) \leftarrow \text{BCA-TO-CONF}(BCAs[i])$ ;
- 6      $DFAs[i] \leftarrow \text{DETERMINIZE}(Conf(BCAs[i]))$ ;
- 7 **return**  $DFAs$ ;

---

## 7.2 Proposed Search Algorithm

Once the DFAs are created, they can be used to search for the corresponding subsequence of the modelled gene. We used a standard matching process, described in Algorithm 15. The DFA moves from state to state based on the current symbol read and the transition function (Line 5). The end of the searched subsequence is found if a final state is reached (Line 6).

---

### Algorithm 15: FIND-SUBSEQUENCE-POSITIONS

---

**Input:**  $\mathcal{A} = (Q, \Sigma, \Delta, I = \{q_0\}, F)$  – a DFA modelling a subsequence of the gene,  $sequence$  – a query sequence  
**Output:**  $positions$  – a set of found ending positions

- 1  $positions \leftarrow \emptyset$ ;
- 2  $state \leftarrow q_0$ ;
- 3  $pos \leftarrow 0$ ;
- 4 **foreach**  $symbol\ a\ in\ sequence$  **do**
- 5      $state \leftarrow \delta(state, a)$ ;
- 6     **if**  $state \in F$  **then**
- 7          $positions \leftarrow positions \cup \{pos\}$ ;
- 8      $pos \leftarrow pos + 1$ ;
- 9 **return**  $positions$ ;

---

## Location clustering

Algorithm 15 must localize most of the subsequence occurrences. Thus, it returns a lot of false positive results because the probability that a match for such a short sequence is

found in a random query is relatively high. This is similar to the seeding process used by the BLAST tool. However, if we say that for the result to be valid, at least  $m \leq p$  closely located subsequences in the correct order must be found, this number is significantly reduced. The process of clustering the closely located occurrences is described in Algorithm 16.

---

**Algorithm 16: FIND-GENE-POSITIONS**

---

**Input:** *DFAs* – a sorted array of  $p$  DFAs modelling subsequences of the gene,  
*sequence* – a query sequence,  $m$  – minimal number of parts found,  
*tolerance* – maximum distance from the expected position

**Output:** *positions* – a set of found starting positions

```

1  $p \leftarrow |DFAs|;$ 
2  $pos \leftarrow \bigcup_{i=0}^{p-1} \{(e, i) : e \in \text{FIND-SUBSEQUENCE-POSITIONS}(DFAs[i], \text{sequence})\};$ 
3  $matches \leftarrow \emptyset;$ 
4 foreach  $(e, i) \in pos$  do
5   if  $i \leq p - m$  then
6      $found\_parts \leftarrow \{i\};$ 
7     for  $j \leftarrow i + 1$  to  $p - 1$  do
8        $expected\_pos \leftarrow e + \sum_{k=i}^j \text{PART-LENGTH}(DFA[k], p, k);$ 
9       if  $\exists (e', j) \in pos$  where  $|e' - expected\_pos| \leq tolerance$  then
10         $found\_parts \leftarrow found\_parts \cup \{j\};$ 
11      if  $|found\_parts| \geq m$  then
12         $start \leftarrow e - \sum_{k=0}^{i-1} \text{PART-LENGTH}(DFA[k], k, p);$ 
13         $positions \leftarrow positions \cup \{start\};$ 
14 return  $positions;$ 

```

---

First, the ending positions of each of the parts are located and stored in the *pos* set in the form of  $(e, i)$ , where  $e$  is the ending position and  $i$  is the index of the part found (Line 2). For each part occurrence, it is determined whether a sufficient number of other parts occur in the expected location. The expected location is estimated by adding the sum of the relevant part lengths to the original ending position  $e$  (Line 8). Since the DFAs are not able to locate the ending position precisely, due to the gap transitions, the occurrence is considered *closely located* if its position is within *tolerance* from the expected position (Line 9). The *found\_parts* set is used to store the indices of closely located parts. If the size of *found\_parts* reaches  $m$  (Line 11), the entire gene is found, and its start position is estimated by subtracting the sum of the preceding parts from the original part position found (Line 12).

## Chapter 8

# Performance Evaluation

This chapter evaluates the performance of the search algorithm presented in Chapter 7. The goal is to demonstrate whether it has the potential to compete with commonly used tools, such as BLAST, and identify its strengths and limitations. The evaluation focuses on its ability to detect structural domains of LTR retrotransposons efficiently.

For evaluation purposes, a reference implementation was created, hereinafter referred to as DFA-based Uncovering of Close-homologues Kit or shortly DUCK. Since the Detano tool [15] was previously used for experimenting with the ALERGIA algorithm, an extended version of its weighted automata representation is used to store a PHMM and convert it to a set of BCA configuration automata. These are then determinized using the Mata library [9].

### 8.1 Evaluation Methodology

This section describes the datasets and evaluation metrics used to assess the performance of DUCK. The evaluation focuses on two key aspects: accuracy and execution time.

#### Test data generation

The sensitivity of tools designed to search for homologous sequences typically decreases with increasing mutation rates. Therefore, artificial sequences with predefined identity levels to the target sequence were generated to evaluate how DUCK's sensitivity is affected by different mutation rates. The substitution, insertion, and deletion rates in LTR retrotransposons widely vary across different species [20]. According to [8], the typical substitution to indel (insertions and deletions) ratio ranges from 4 to 60, with the most commonly observed value around 10. Thus, this ratio was used to approximate the mutation patterns of real sequences.

The mutated variants of the `Ty1-GAG_Class_I::LTR::Ty1/copia::Tork`, taken from the database used by the TE-greedy-nester tool [21], were then inserted into 1000-symbols-long random query sequences at specified locations. If DUCK correctly localised the inserted sequence (within a tolerance of 10% of the inserted sequence length), it is considered a *true positive (TP)* result. Missed occurrences were counted as *false negatives (FN)*. Results at positions other than where the searched sequence was inserted are called *false positives (FP)*. To ensure statistical robustness, 10,000 sequences were generated for each level of sequence identity.

## Used metrics

The following metrics were used to evaluate the reliability of DUCK:

- *Sensitivity* – the proportion of truly positive results identified as positive. The higher the sensitivity is, the fewer gene occurrences are missed. It can be computed using the following formula:

$$Sensitivity = \frac{TP}{TP + FN}.$$

- *FP/1000aa* – the number of false positives per 1000 amino acids. This normalised metric expresses how many FPs occur, on average, for every 1000 amino acids of the query sequence. It is defined as:

$$FP/1000aa = \frac{FP \cdot 1000}{sequence\_length}.$$

## Parameter optimization

DUCK’s behaviour is controlled by the four parameters described in Chapter 7:

- *p* – the number of parts into which the model is divided,
- *b* – the counter bound,
- *t* – the threshold,
- *m* – the minimum number of parts found for the result to be valid.

The optimal settings of *p*, *b*, *t* and *m* are crucial. The higher *p*, *b* and *t* are, the more sensitive DUCK is. However, such settings often lead to higher numbers of false positives, which would have to be further filtered. Therefore, it was necessary to find the best possible settings according to sensitivity and FP/1000aa metrics. The most interesting experimentally found DUCK settings are listed in Table 8.1. The threshold *t* is expressed as the negative natural logarithm of the probability in the same way as HMMER handles probabilities. There are many different DUCK settings, each suitable for a different use case. For clarity, only these six settings are further evaluated in Section 8.2, as they seemed most suitable for detecting the LTR retrotransposon domains.

<b>p</b>	<b>b</b>	<b>t</b>	<b>m</b>	<b>Sensitivity<sub>80</sub></b>	<b>Sensitivity<sub>60</sub></b>	<b>Sensitivity<sub>40</sub></b>	<b>FP/1000aa</b>
9	3	1.8	4	0.9976	0.9241	0.5043	0.3876
9	3	1.8	5	0.9964	0.8111	0.2612	0.0413
9	3	1.8	6	0.9735	0.5527	0.0843	0.0028
10	3	1.9	7	0.9982	0.9189	0.4551	0.1975
11	3	1.9	7	0.9993	0.9729	0.7327	1.2496
12	3	1.8	6	1.0	0.9978	0.9636	21.6064

Table 8.1: Several DUCK settings that showed a decent sensitivity, while maintaining low numbers of false positive results. The last row contains a setting which has a higher number of false positives but shows high sensitivity even for more remote homologues. The subscript indicates the percentage of the sequence identity for which the sensitivity was computed.

## 8.2 Experimental Evaluation

This section describes the experiments used to evaluate the performance of DUCK. All measurements were performed on a virtual Ubuntu 22.04.5 LTS machine with 32 GB of memory and two cores. For a meaningful comparison in the context of transposon detection, BLASTX was configured using the same settings as in the TE-greedy-nester tool.

### Execution time compared to BLASTX

The aim of the following measurement is to demonstrate whether using DUCK leads to a significant speedup of the search compared to the BLASTX tool. To allow comparison with the BLASTX tool, the input nucleotide sequence has to be translated into six possible reading frames. Therefore, a variant of DUCK, called DUCKX, was implemented. It uses the same search algorithm, except it translates the query nucleotide sequence into the six amino acid sequences before the search. The DUCK's speed strongly depends on the length of the query sequence. Therefore, several measurements were performed on input sequences of different lengths.

The results are shown in Figure 8.1. The number of DUCK's passes over the query sequence is determined by the parameter  $p$ . As expected, the best results were obtained with the variants of DUCKX that used  $p = 9$ . These settings resulted in a tenfold speedup of the search compared to BLASTX. The advantage of DUCK is that, except for execution times for very short sequences, which are affected by the time needed to load the prepared model, the execution time increases linearly with increasing query sequence length. This contrasts with BLASTX, whose runtime increases significantly faster. Therefore, DUCK appears particularly useful for processing very long sequences.

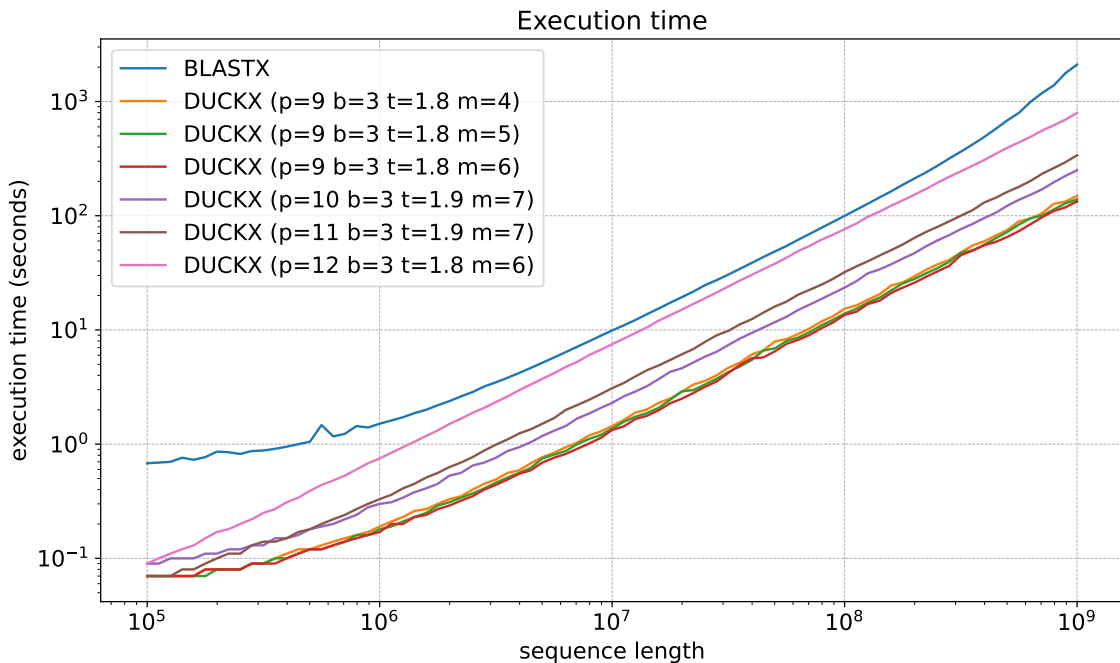


Figure 8.1: A comparison of the execution time of BLASTX and DUCKX as functions of the sequence length.

## Execution time compared to HMMER

There is currently no HMMER program available that is an alternative to the BLASTX tool [12]. HMMER is primarily designed to make protein-to-protein comparisons. Therefore, it is rather an alternative to BLASTP, as both tools are used to compare protein sequences. However, since DUCK is based on PHMMs generated by HMMER, it is reasonable to include their comparison in this evaluation.

HMMER only supports query sequences up to 100,000 amino acids in length. However, the execution time of DUCK is significantly impacted by the model parser, especially for such short sequences. The goal of this thesis is not to develop an optimised parser. For this reason, the DUCK's model parsing time is excluded from the performance measurements. This disadvantage of HMMER may seem unfair; however, HMMER's model loading time should be negligible, as it uses a much smaller model stored in an optimised binary format.

Several measurements using query sequences of different lengths were performed. A comparison of the execution times of HMMER and the fastest DUCK variant with  $p = 9$  is shown in Figure 8.2. The plot suggests that DUCK outperforms HMMER in terms of execution time. However, the results should be interpreted with caution. Since the measured execution times are very short, the results may be affected by measurement noise. Furthermore, HMMER is slightly disadvantaged in this comparison, as the DUCK's parsing time was excluded from the measurements.

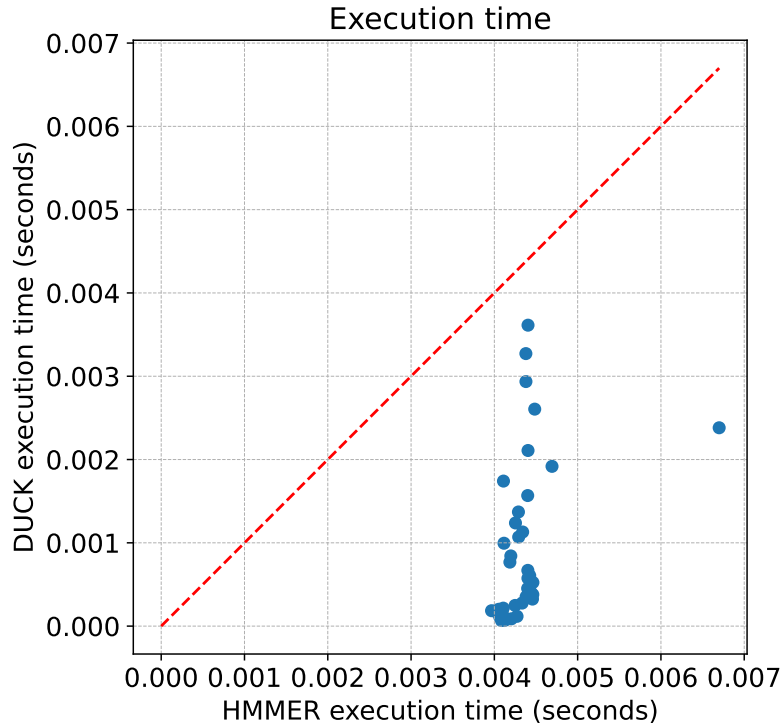


Figure 8.2: A comparison of execution times of HMMER and DUCK.

## Sensitivity compared to BLASTP

To assess sensitivity, it is more straightforward to compare the tool’s performance on a single amino acid query rather than on all six translated reading frames. Therefore, BLASTP was used as a reference method, as it performs the same type of protein-to-protein alignment as BLASTX but without the complexity introduced by the translation.

The results are shown in Figure 8.3. The sensitivity of certain configurations of DUCK is comparable to that of BLASTP. The best results are achieved when searching for sequences with low levels of mutation. This suggests that the method is well-suited, especially for identifying close homologues.

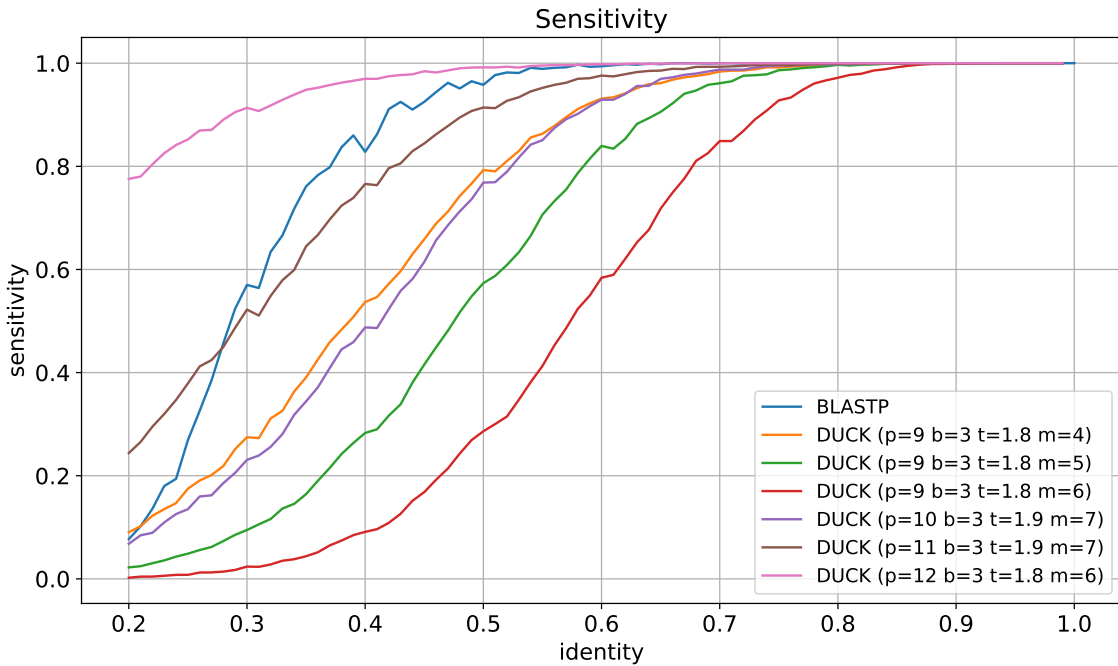


Figure 8.3: A comparison of the sensitivity of DUCK and BLASTP as functions of sequence identity.

## False positive rates compared to BLASTP

Sensitivity alone is not sufficient to evaluate the reliability of the search algorithm. It is also important to assess the frequency of false positive results it generates. For this purpose, we use the FP/1000aa metric, which indicates how many false positives we get per 1000 amino acids of the query sequence. A comparison of DUCK and BLASTP regarding the FP/1000aa metric is shown in Table 8.2.

The results show that the number of FP results is relatively low for most of the DUCK settings. The exception is the setting ( $p = 12, b = 3, t = 1.8, m = 6$ ), which has a significantly higher number of false positives than BLASTP. On the other hand, as shown in the previous subsection, the sensitivity of this DUCK setting remains high even for highly mutated sequences. Therefore, similar settings could be used as a sensitive pre-filter. However, this DUCK setting is only slightly faster than BLASTP. Thus, further speed improvements would be necessary to make the pre-filter more practical.

<b>p</b>	<b>b</b>	<b>t</b>	<b>m</b>	<b>FP/1000aa</b>
9	3	1.8	4	0.3876
9	3	1.8	5	0.0413
9	3	1.8	6	0.0028
10	3	1.9	7	0.1975
11	3	1.9	7	1.2496
12	3	1.8	6	21.6064
<b>BLASTP</b>				0.4451

Table 8.2: A comparison of different DUCK settings and the BLASTP tool in terms of false positive rates.

### 8.3 Summary of Results and Possible Improvements

The experimental evaluation showed that the model simplification needed to allow its determinization did not lead to a significant decrease in sensitivity compared to the BLASTP tool. Especially for close homologues, around 80% identity or higher, the results of DUCK and BLASTP were almost indistinguishable. Furthermore, it was demonstrated that several DUCK settings can achieve up to a tenfold acceleration in the search execution time.

Although the existing DUCK implementation is already faster than BLASTX, it might be possible to improve its speed even further. The proposed algorithm has the disadvantage that it requires processing the query sequence multiple times. Specifically, one pass is needed for each of the  $p$  DFAs. These passes are computationally independent, so it may be possible to use parallelisation techniques. The second option is designing a search algorithm that continuously switches between searching for individual parts during one pass, according to the last part found. That could potentially result in a further  $p$ -fold acceleration of the search procedure.

It has been shown that some DUCK settings are even more sensitive than BLAST. The disadvantage of these settings is that they produce a relatively large number of false positives. However, if we introduced some metric to determine the quality of the occurrences found, for example, using e-value or a similar metric, it would be possible to filter the results effectively. Using this technique, the number of false positives would be significantly reduced.

## Chapter 9

# Conclusion

This thesis showed that it may be possible to accelerate the detection of LTR retrotransposon domains using finite automata. The main idea of this work is that if it were possible to determinize a profile hidden Markov model (PHMM) generated by HMMER, it could accelerate the search for sequence homologues. However, the determinization of the model of the entire sequence proved unfeasible due to the significant nondeterminism needed to model frequent mutations precisely. Therefore, we introduced a method to transform a single PHMM, representing the searched domain, into several smaller deterministic finite automata (DFAs). These DFAs are then used to search for subsequences of the domain efficiently. Closely located subsequences in the correct order indicate the presence of the entire domain.

The presented algorithm was implemented and evaluated using artificial sequences. It was shown that the model simplification, which was necessary for its determinization, did not significantly decrease the sensitivity compared to BLAST. Additionally, the algorithm achieved up to a tenfold search speedup relative to BLAST.

Future work will focus on further accelerating the algorithm by reducing the number of passes through the query sequence. Additionally, a metric analogous to the e-value will be introduced to filter the results and minimise the number of false positives.

# Bibliography

- [1] ALBERTS, B.; JOHNSON, A.; LEWIS, J.; RAFF, M.; ROBERTS, K. et al. *Molecular Biology of the Cell*. 4th ed. New York: Garland Science, 2002. 1463 p. ISBN 0-8153-4072-9.
- [2] ALTSCHUL, S. BLAST Algorithm. In: *ELS*. John Wiley & Sons, Ltd, September 2005. ISBN 9780470015902.
- [3] ALTSCHUL, S. F.; GISH, W.; MILLER, W.; MYERS, E. W. and LIPMAN, D. J. Basic local alignment search tool. *Journal of Molecular Biology*. Elsevier, October 1990, vol. 215, no. 3, p. 403–410. ISSN 0022-2836.
- [4] BAUCOM, R.; ESTILL, J.; CHAPARRO, C.; UPSHAW, N.; JOGI, A. et al. Exceptional Diversity, Non-Random Distribution, and Rapid Evolution of Retroelements in the B73 Maize Genome. *PLoS genetics*. Public Library of Science, November 2009, vol. 5, no. 11, p. 1–13.
- [5] BOLAND, C. R. Non-coding RNA: It's Not Junk. *Digestive Diseases and Sciences*, November 2017, vol. 62, no. 5, p. 1107–1109. ISSN 1573-2568.
- [6] CAMACHO, C.; COULOURIS, G.; AVAGYAN, V.; MA, N.; PAPADOPOULOS, J. et al. BLAST+: architecture and applications. *BMC bioinformatics*. Springer, December 2009, vol. 10, p. 421.
- [7] CARRASCO, R. C. and ONCINA, J. Learning stochastic regular grammars by means of a state merging method. In: Springer. *International Colloquium on Grammatical Inference*. 1994, p. 139–152.
- [8] CHEN, J.-Q.; WU, Y.; YANG, H.; BERGELSON, J.; KREITMAN, M. et al. Variation in the Ratio of Nucleotide Substitution and Indel Rates across Genomes in Mammals and Bacteria. *Molecular Biology and Evolution*. Oxford University Press, March 2009, vol. 26, no. 7, p. 1523–1531. ISSN 0737-4038.
- [9] CHOCHOLATÝ, D.; FIEDOR, T.; HAVLENA, V.; HOLÍK, L.; HRUŠKA, M. et al. Mata: A Fast and Simple Finite Automata Library. In: FINKBEINER, B. and KOVÁCS, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer Nature Switzerland, 2024, p. 130–151. ISBN 978-3-031-57249-4.
- [10] DAYHOFF, M. O.; SCHWARTZ, R. M. and ORCUTT, B. C. A model of evolutionary change in proteins. In: *Atlas of Protein Sequence and Structure*. Silver Spring, MD: National Biomedical Research Foundation, 1978, 5, suppl. 3, p. 345–352.

- [11] EDDY, S. R. *HMMER: biosequence analysis using profile hidden Markov models*. 2020. Available at: <http://hmmerr.org>. Version 3.3.2. [Accessed: 2025-05-12].
- [12] EDDY, S. R. and WHEELER, T. J. *HMMER User's Guide*. 2013. Available at: <http://eddylib.org/software/hmmer3/3.1b1/Userguide.pdf>. [Accessed: 2025-04-01].
- [13] ELBARBARY, R. A.; LUCAS, B. A. and MAQUAT, L. E. Retrotransposons as regulators of gene expression. *Science*. American Association for the Advancement of Science, February 2016, vol. 351, no. 6274.
- [14] ENRIGHT, A. J. *Computational Analysis of Protein Function within Complete Genomes*. Hinxton, United Kingdom, 2002. Dissertation. University of Cambridge.
- [15] HAVLENA, V.; MATOUŠEK, P.; RYŠAVÝ, O. and HOLÍK, L. Accurate Automata-Based Detection of Cyber Threats in Smart Grid Communication. *IEEE Transactions on Smart Grid*. IEEE, 2023, vol. 14, no. 3, p. 2352–2366.
- [16] HENIKOFF, S. and HENIKOFF, J. G. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America*, November 1992, vol. 89, no. 22, p. 10915–10919.
- [17] HIGUERA, C. De la. *Grammatical Inference: Learning Automata and Grammars*. Cambridge, UK: Cambridge University Press, April 2010. 417 p. ISBN 978-0-521-76316-5.
- [18] HOLÍK, L.; SÍČ, J.; HOLÍKOVÁ, L. and VOJNAR, T. Fast Matching of Regular Patterns with Synchronizing Counting. In: *Foundations of Software Science and Computation Structures*. Heidelberg: Springer Verlag, April 2023, vol. 13992, no. 1, p. 392–412. ISSN 0302-9743.
- [19] HOPCROFT, J. E.; MOTWANI, R. and ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. USA: Addison-Wesley Longman Publishing Co., Inc., June 2006. 535 p. ISBN 0321455363.
- [20] JEDLIČKA, P.; LEXA, M. and KEJNOVSKY, E. What Can Long Terminal Repeats Tell Us About the Age of LTR Retrotransposons, Gene Conversion and Ectopic Recombination? *Frontiers in Plant Science*. Frontiers Media SA, May 2020, vol. 11, p. 644. ISSN 1664-462X.
- [21] JEDLIČKA, P. *Gydb\_arh\_chdcr.fa*. 2023. Available at: [https://gitlab.fi.muni.cz/lexa/nested/-/blob/master/dbs/gydb\\_arh\\_chdcr/gydb\\_arh\\_chdcr.fa](https://gitlab.fi.muni.cz/lexa/nested/-/blob/master/dbs/gydb_arh_chdcr/gydb_arh_chdcr.fa). [Accessed: 2025-05-01].
- [22] KARKI, G. *Transposable elements in eukaryotes*. 2020. Available at: <https://www.onlinebiologynotes.com/wp-content/uploads/2020/04/retrotransposons-and-DNA-transposons-1.jpg>. [Accessed: 2025-03-12].
- [23] LAMY, J.-B.; BERTHELOT, H. and FAVRE, M. Rainbow Boxes: A Technique for Visualizing Overlapping Sets and an Application to the Comparison of Drugs Properties. In: *2016 20th International Conference Information Visualisation (IV)*. IEEE, July 2016, p. 253–260. ISSN 2375-0138.

- [24] LEXA, M.; JEDLICKA, P.; VANAT, I.; CERVENANSKY, M. and KEJNOVSKY, E. TE-greedy-nester: structure-based detection of LTR retrotransposons and their nesting. *Bioinformatics*, July 2020, vol. 36, no. 20, p. 4991–4999. ISSN 1367-4803.
- [25] MAMINOV, A. D. Short-length peptides contact map prediction using Convolution Neural Networks. In: *Proceedings of The 6th International Workshop on Deep Learning in Computational Physics — PoS(DLCP2022)*. July 2022, vol. 429, p. 16.
- [26] MARTÍNEK, T. *BIF lecture*. Presented at FIT BUT. March 2022. Slides not publicly available.
- [27] MIYATA, T.; MIYAZAWA, S. and YASUNAGA, T. Two types of amino acid substitutions in protein evolution. *Journal of Molecular Evolution*. Springer, 1979, vol. 12, no. 3, p. 219–236. ISSN 1432-1432.
- [28] NEEDLEMAN, S. B. and WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*. Elsevier, March 1970, vol. 48, no. 3, p. 443–453. ISSN 0022-2836.
- [29] SATHYASEELAN, C.; PATRO, L. P. P. and RATHINAVELAN, T. Sequence patterns and HMM profiles to predict proteome wide zinc finger motifs. *Pattern Recognition*. Elsevier, March 2023, vol. 135, p. 109134. ISSN 0031-3203.
- [30] SHAH, A. *DNA Sequence Representation by Use of Statistical Finite Automata*. 2009. Master’s Project. San Jose State University.
- [31] SIEVERS, F.; WILM, A.; DINEEN, D.; GIBSON, T. J.; KARPLUS, K. et al. Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Molecular Systems Biology*. John Wiley & Sons, Ltd Chichester, UK, October 2011, vol. 7, no. 1, p. 539.
- [32] SMIT, A.; HUBLEY, R. and GREEN, P. *RepeatMasker Open-4.0*. 2013–2015. Available at: <http://www.repeatmasker.org>. [Accessed: 2025-05-12].
- [33] SMITH, T. and WATERMAN, M. Identification of common molecular subsequences. *Journal of Molecular Biology*. Elsevier Science, 1981, vol. 147, no. 1, p. 195–197. ISSN 0022-2836.
- [34] THOMAS BULLE, C.; PIEDNOËL, M.; DONNART, T.; FILEE, J.; JOLLIVET, D. et al. Mollusc genomes reveal variability in patterns of LTR-retrotransposons dynamics. *BMC Genomics*. Springer, November 2018, vol. 19, p. 1–18.
- [35] THOMPSON, K. Programming Techniques: Regular expression search algorithm. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery, June 1968, vol. 11, no. 6, p. 419–422. ISSN 0001-0782.
- [36] VITERBI, A. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*. IEEE, 1967, vol. 13, no. 2, p. 260–269.
- [37] WIKIPEDIA CONTRIBUTORS. *Query words*. 2009. Available at: [https://commons.wikimedia.org/wiki/File:Query\\_words.svg](https://commons.wikimedia.org/wiki/File:Query_words.svg). [Accessed: 2025-03-22].

- [38] WIKIPEDIA CONTRIBUTORS. *Blosum62 Dayhoff ordering*. 2022. Available at: <https://commons.wikimedia.org/wiki/File:Blosum62-dayhoff-ordering.svg>. [Accessed: 2025-03-30].
- [39] XU, Z. and WANG, H. LTR\_FINDER: an efficient tool for the prediction of full-length LTR retrotransposons. *Nucleic Acids Research*. Oxford University Press, July 2007, vol. 35, suppl\_2, p. W265–W268. ISSN 0305-1048.
- [40] YOON, B. J. Hidden Markov Models and their Applications in Biological Sequence Analysis. *Current Genomics*. Bentham Science Publishers, September 2009, vol. 10, no. 6, p. 402–415.
- [41] ZHANG, L.; YAN, L.; JIANG, J.; WANG, Y.; JIANG, Y. et al. The structure and retrotransposition mechanism of LTR-retrotransposons in the asexual yeast *Candida albicans*. *Virulence*. Taylor & Francis, August 2014, vol. 5, no. 6, p. 655–664.
- [42] ZVELEBIL, M. and BAUM, J. O. *Understanding Bioinformatics*. 1st ed. New York, NY, USA; Abingdon, UK: Garland Science, Taylor & Francis Group, LLC, May 2007. 5–42 p. ISBN 978-0-8153-4024-9.