Reichman University

Efi Arazi School of Computer Science

M.Sc. program

Developing a flexible and comprehensive software app for design of synthetic DNA sequences without unwanted patterns

by

Hadar Pur

Final project, submitted in partial fulfillment of the requirements
for the M.Sc. degree, School of Computer Science
Reichman University (The Interdisciplinary Center, Herzliya)

September 2025

This work was carried out under the supervision of Dr. Ilan Gronau from the Efi Arazi School of Computer Science, Reichman University.

Abstract

This project addresses the problem of eliminating patterns from DNA sequences, which may lead to harmful protein binding, by using a cost scheme that represents a biological function alongside efficient algorithmic methods. Based on the theoretical framework introduced in Zehavit Leibovich's thesis, I implemented a finite state machine and a dynamic programming algorithm to generate a minimum-cost sequence while preserving protein function and avoiding unwanted patterns. The cost model maintains the length of the encoded protein, assigns high penalties for non-synonymous substitutions that alter the encoded amino acid, and lower penalties for synonymous substitutions, which consider codon usage bias. Enhancements to the finite state machine proposed in Zehavit's thesis enable accurate codon tracking using two-base memory. The BioSynth tool was developed with both a graphical user interface (GUI) and a command-line interface (CLI), supporting codon optimization under biological constraints. Experimental results validate the accuracy of the adapted cost scheme compared to traditional cost tables and demonstrate the practical utility of the tool in synthetic DNA design.

Contents

1 Introduction and Background **5**

2 Accommodating Codon Usage **9**

2.1 A cost scheme for coding sequences assuming one substitution per codon 9

2.2 A general sequence scoring scheme that allows more than one substitution per codon 13

2.2.1 Overview of modified cost function 13

2.2.2 Examples of representative cases 15

2.2.3 Algorithm for computing the modified cost function 19

2.3 Modifying the dynamic programming algorithm and the FSM to accommodate the new scoring scheme 20

2.3.1 Example of representative cases 21

2.3.2 Modifying the KMP-based FSM to allow computation of last2(v) 22

2.3.3 Adapting the dynamic programming algorithm to the modified FSM 23

3 The BioSynth Application **26**

3.1 Implementation Details 26

3.1.1 Comparison between FSM Algorithm Design and Implementation 27

3.2 BioSynth App 30

3.2.1 Components Overview 32

3.2.2 Open Reading Frame (ORF) Considerations 34

3.2.3 Executing BioSynth 35

3.2.4 Graphical User Interface (GUI) Execution 36

3.2.5 CLI Execution 43

Bibliography **45**

A Amino Acids Scoring Scheme **46**

A.1 Symmetric amino acids scoring scheme . . . 46

A.1.1 Tryptophan (Trp) . . . 46

A.1.2 Methionine (Met) . . . 46

A.1.3 Phenylalanine (Phe) . . . 47

A.1.4 Histidine (His) . . . 47

A.1.5 Asparagine (Asn) . . . 48

A.1.6 Aspartic acid (Asp) . . . 48

A.1.7 Tyrosine (Tyr) . . . 49

A.1.8 Glutamine (Gln) . . . 49

A.1.9 Lysine (Lys) . . . 50

A.1.10 Glutamic acid (Glu) . . . 50

A.1.11 Cysteine (Cys) . . . 51

A.1.12 Isoleucine (Ile) . . . 51

A.1.13 Alanine (Ala) . . . 52

A.1.14 Proline (Pro) . . . 53

A.1.15 Threonine (Thr) . . . 53

A.1.16 Valine (val) . . . 54

A.2 Non-Symmetric amino acids scoring scheme with 4 codons . . . 55

A.2.1 Glycine (Gly) . . . 55

A.3 Non-Symmetric amino acids scoring scheme with 6 codons . . . 56

A.3.1 Arginine (Arg) . . . 56

A.3.2 Serine (Ser) . . . 56

A.3.3 Leucine (Leu) . . . 57

A.4 Three stop codons . . . 58

A.4.1 Stop . . . 58

B Generating a Codon Usage File using Kazusa Database **59**

C Application Execution Examples **61**

C.1 Non-coding Regions . . . 61

C.2 Single Coding Region . . . 64

C.3 Fully Overlapping ORFs . . . 70

C.4 Multiple Coding Regions (GUI) . . . 74

Chapter 1

Introduction and Background

Synthesis of DNA sequences has become a central tool in molecular biology, offering insights into the function and behavior of living organisms. This process involves physical synthesis of a specified DNA sequence—a process that has been made faster, cheaper, and more accurate by advancements in sequencing technologies.

Designing sequences for DNA synthesis involves considering several constraints, including the desired function or purpose of the sequence and the potential for unintended effects. Unwanted binding interactions between the DNA sequence and other molecules, like proteins, can lead to unintended consequences such as the activation of oncogenes or suppression of tumor suppressor genes. Thus, designing DNA sequences for synthesis requires careful consideration of these constraints to ensure that the synthesized sequence functions as intended and has minimal unwanted interactions with other molecules.

On the positive side, the design involves specifying an optimal target sequence and determining how local changes to this sequence may impact its functionality. On the negative side, it involves avoiding any unwanted binding of proteins to the DNA sequence by eliminating certain short sequence patterns that bind proteins with high likelihood. While several design tools exist to address this issue, there is no formal solution to the problem yet.

A recent study [1] [2] by Zehavit Leibovich and Ilan Gronau introduced a formal algorithmic framework for eliminating unwanted sequence patterns while minimizing interference with the desired biological function of synthesized DNA. Their framework includes two core algorithms. Algorithm 1 uses dynamic programming to compute the minimum-cost sequence that avoids unwanted patterns. The update steps of this dynamic programming algorithm rely on a finite state machine (FSM), which is efficiently generated by Algorithm 2, that generates all (and only) sequences that don't contain patterns in a given set P.

Algorithm 1 Computing a min-cost $\mathscr{P}$-clean sequence of length n

Compute an FSM(V, f) over alphabet Σ that generates all and only $\mathscr{P}$-clean sequences.
Initialize: $A[0,v] = \begin{cases} 0 & \text{if } v = v_{init} \\ \infty & \text{otherwise} \end{cases}$
for $i = 1...n$ and $v \in V$ **do**
    $A^*[i,v] = (u^*, \sigma^*) = \underset{u,\sigma: f(u,\sigma)=v}{\text{argmin}} \{A[i-1,u] + cost(i,\sigma)\}$
    $A[i,v] = A[i-1,u^*] + cost(i,\sigma^*)$
end for
$v_n = \underset{v \in V}{\text{argmin}} A[n,v]$
for $i = n...1$ **do**
    $(v_{i-1}, S_i) = A^*[i,v_i]$
end for

Zehavit's thesis presents a simple example to illustrate how a cost scheme can convey information about the functional role of different bases within a coding sequence. Zehavit's example considered the sequence $S = TACACA$, from which we wanted to eliminate the unwanted pattern $TACA$, which is associated with the binding of BlaI repressor. A functional cost function was defined for this sequence to reflect the fact that it represents a sequence of two codons: TAC (Tyrosine) and ACA (Threonine):

	Tyrosine (Tyr)			Threonine (Thr)		
S = ...	T	A	C	A	C	A
cost(i, A) = ...	w	0	∞	0	w	0
cost(i, T) = ...	0	w	x_1	w	w	x_2
cost(i, C) = ...	w	w	0	w	0	x_3
cost(i, G) = ...	w	w	∞	w	w	x_4

Table 1.1

The cost table, $cost(i,\sigma)$, is defined for every position $i = 1..6$ and every base $\sigma \in \{A, C, T, G\}$. The value of $cost(i,\sigma)$ is determined based on a few basic guidelines:

- If $\sigma = S_i$ (the i'th base in the target sequence S), then $cost(i,\sigma) = 0$.
- If substituting S_i with σ results in another codon of the same amino acid, then $cost(i,\sigma) = x \geq 0$. Example—replaces the first codon (TAC), which encodes Tyr with TAT, which also encodes Tyr. This is a synonymous change, so $cost(3,T) = x$.
- If substituting S_i with σ results in codon of different amino acid codon, then $cost(i,\sigma) = w \gg x$. Example—replaces the first codon (TAC), which encodes Tyr with TCC, which encodes Serine. This is a non-synonymous change, so $cost(2,C) = w \gg x$.
- If substituting S_i with b results in codon of a stop codon, then $cost(i,\sigma) \to \infty$. Example—replaces the first codon (TAC), which encodes Tyr with TAA, which encodes stop codon. This is a harmful change, so $cost(3,A) = \infty$.

Algorithm 2 Constructing a finite state machine (FSM) by computing the state space V and the transition function f

for $p \in P$ **do**
  for $j \in [1..|p|-1]$ **do**
    Set $f(p_{1..j-1}, p_j) \leftarrow p_{1..j}$
  end for
  Set $f(p_{1..|p|-1}, p_{|p|}) \leftarrow$ NULL
end for

InitEmptyQueue(stateQueue)
$V \leftarrow \{\varepsilon\}$
for $\sigma \in \Sigma$ **do**
  if $f(\varepsilon, \sigma)$ is not set yet **then**
    Set $f(\varepsilon, \sigma) \leftarrow \varepsilon$
  end if
  if $f(\varepsilon, \sigma) == \sigma$ **then**
    Set $g(\sigma) \leftarrow \varepsilon$
    stateQueue.push(σ)
  end if
end for

while stateQueue is not empty **do**
  $v \leftarrow$ stateQueue.pop()
  $V \leftarrow V \cup \{v\}$
  for $\sigma \in \Sigma$ **do**
    if $f(g(v), \sigma) ==$ NULL **then**
      Set $f(v, \sigma) \leftarrow$ NULL
    end if
    if $f(v, \sigma)$ is not set yet **then**
      Set $f(v, \sigma) \leftarrow f(g(v), \sigma)$
    end if
    if $f(v, \sigma) = v\sigma$ **then**
      Set $g(v\sigma) \leftarrow f(g(v), \sigma)$
      stateQueue.push($v\sigma$)
    end if
  end for
end while

The genetic code table (Figure 1.1) defines a universal mapping of all 64 codons (base triplets) to amino acids and applies to nearly all living organisms. This global mapping enables the generalization of the guidelines described above, allowing systematic computation of substitution costs for every position in any codon and extending the scoring scheme across all cases.

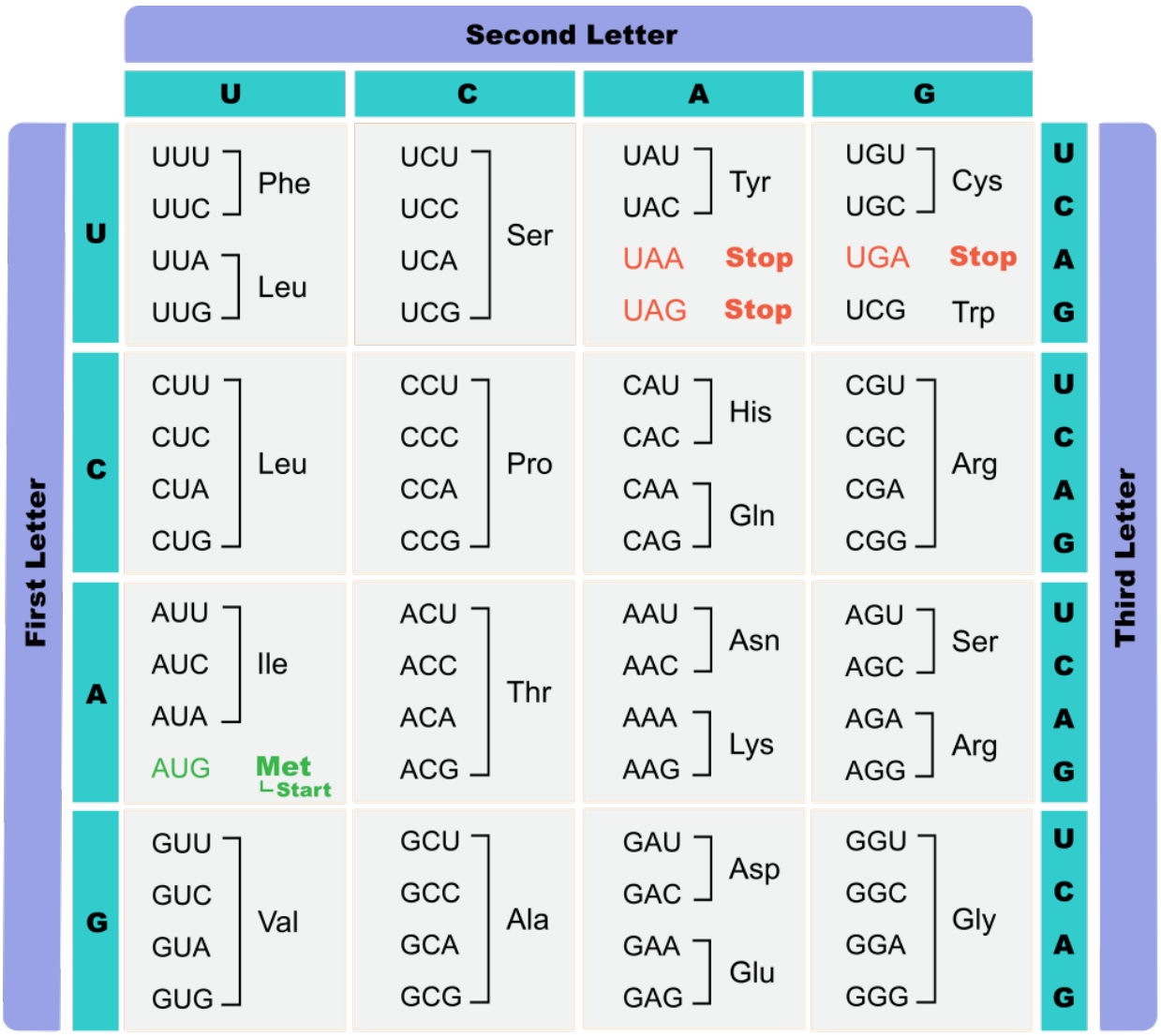


Figure 1.1: The Genetic code table.

The genetic code specifies a global mapping of all base triplets (codons) to amino acids. This mapping is universal and applies to all living organisms. Figure source: https://open.lib.umn.edu/evolutionbiology/chapter/5-8-using-the-genetic-code-2/

The main objective of this project is to suggest a formal cost scheme that allows to faithfully model coding sequences and also apply the necessary modifications to Zehavit's algorithms to accommodate this scheme. This work builds on the ideas outlined in Zehavit's framework and develops a formal approach to address key challenges in modeling function within coding sequences. The project focused on two main goals:

1. Designing a cost scheme that faithfully represents functional constraints in coding regions and adapting Zehavit's algorithms accordingly.
2. Implementing these improvements in a user-friendly software tool with both GUI and CLI support.

Chapter 2

Accommodating Codon Usage

2.1 A cost scheme for coding sequences assuming one substitution per codon

As an initial step of the project, we took Zehavit's example from Table 1.1 and used the basic ideas behind this cost table to define a cost table for each of the 64 possible codons. While here we focus on a few illustrative examples to highlight key patterns and biological considerations, the complete list of costs for all 64 codons is provided in the Appendix A. For instance, certain amino acids are uniquely encoded by a single codon—Tryptophan is one such example:

	Tryptophan (Trp)		
S =	T	G	G
cost(i, A) =	w	∞	∞
cost(i, T) =	0	w	w
cost(i, C) =	w	w	w
cost(i, G) =	w	0	0

Table 2.1.1: Cost table for Tryptophan

In this case, most single-nucleotide substitutions result in a different amino acid and are therefore assigned a cost of w, reflecting a non-synonymous change. Notably, two specific substitutions (TGG $\rightarrow$ TAG and TGG $\rightarrow$ TGA) result in a stop codon, which is biologically invalid and thus assigned an infinite cost.

Nine amino acids are each encoded by exactly two codons. In such cases, a single synonymous substitution is possible, and it is assigned a lower cost x_i. All other substitutions result in non-synonymous changes or stop codons. Phenylalanine serves as a representative example:

	Phenylalanine (Phe)		
S =	T	T	C
cost(i, A) =	w	w	w
cost(i, T) =	0	0	x_1
cost(i, C) =	w	w	0
cost(i, G) =	w	w	w

	Phenylalanine (Phe)		
S =	T	T	T
cost(i, A) =	w	w	w
cost(i, T) =	0	0	0
cost(i, C) =	w	w	x_2
cost(i, G) =	w	w	w

Table 2.1.2: Cost tables for Phenylalanine

We can combine these two tables because the first two positions in its codons have identical substitution costs. At the third position, there are two possible bases, and both scoring tables assign a cost of 0 to the original base. To unify these into one table, we define a cost x_i for the third position, independent of the original base, where $1 \leq i \leq 2$.

The combined table for these two codons is:

	Phenylalanine (Phe)		
S =	T	T	T/C
cost(i, A) =	w	w	w
cost(i, T) =	0	0	x_1
cost(i, C) =	w	w	x_2
cost(i, G) =	w	w	w

Table 2.1.3: Combined cost table for Phenylalanine

13 amino acids are encoded by more than two codons. In these cases, multiple synonymous substitutions are possible, often involving changes at the third codon position. For example, Alanine is encoded by four codons that differ only at the third base, allowing up to three synonymous substitutions. Each of these is assigned a lower cost x_i, while all other substitutions lead to non-synonymous changes.

	Alanine (Ala)		
S =	G	C	A
cost(i, A) =	w	w	0
cost(i, T) =	w	w	x_2
cost(i, C) =	w	0	x_3
cost(i, G) =	0	w	x_4

	Alanine (Ala)		
S =	G	C	T
cost(i, A) =	w	w	x_1
cost(i, T) =	w	w	0
cost(i, C) =	w	0	x_3
cost(i, G) =	0	w	x_4

	Alanine (Ala)		
S =	G	C	C
cost(i, A) =	w	w	x_1
cost(i, T) =	w	w	x_2
cost(i, C) =	w	0	0
cost(i, G) =	0	w	x_4

	Alanine (Ala)		
S =	G	C	G
cost(i, A) =	w	w	x_1
cost(i, T) =	w	w	x_2
cost(i, C) =	w	0	x_3
cost(i, G) =	0	w	0

Table 2.1.4: Cost tables for Alanine

Since Alanine's codons share the same first and second bases, substitution costs differ only at the third position. To unify the four codon tables, we assign costs x_i for third-position substitutions, where $1 \leq i \leq 4$. The combined table for these four codons is:

	Alanine (Ala)		
S =	G	C	*
cost(i, A) =	w	w	x_1
cost(i, T) =	w	w	x_2
cost(i, C) =	w	0	x_3
cost(i, G) =	0	w	x_4

Table 2.1.5: Combined cost table for Alanine

Note that for some amino acids, some of the substitutions result in a stop codon. For example, examine the two codons for Glutamic acid:

	Glutamic acid (Glu)		
S =	G	A	A
cost(i, A) =	w	0	0
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	0	w	x_2

	Glutamic acid (Glu)		
S =	G	A	G
cost(i, A) =	w	0	x_1
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	0	w	0

Table 2.1.6: Cost tables for Glutamic acid

The combined table for these two codons is:

	Glutamic acid (Glu)		
S =	G	A	A/G
cost(i, A) =	w	0	x_1
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	0	w	x_2

Table 2.1.7: Combined cost table for Glutamic acid

Certain codons, such as the start codon (ATG) and the three stop codons (TAA, TAG, TGA), play special roles in translation initiation and termination. Substitutions that create a stop codon can prematurely terminate translation and are therefore assigned an infinite cost. In contrast, substitutions that introduce a start codon (ATG) within an existing coding region change the amino acid and are assigned a cost of w.

While the basic cost table approach is quite flexible, there are some cases where it is not possible to merge the codon tables of all codons corresponding to a single amino acid into one combined table. This occurs in two main cases:

1. When some of the codons can mutate into stop codons through single-base substitutions.
2. When an amino acid is encoded by more than four codons, making it difficult to consolidate all synonymous substitutions into one table.

Consider Arginine, which is encoded by six different codons. These codons vary at both the first and third positions, and some substitutions can lead to stop codons—making a single combined scoring table inaccurate or incomplete.

	Arginine (Arg)		
S =	C	G	A
cost(i, A) =	x_5	w	0
cost(i, T) =	∞	w	x_2
cost(i, C) =	0	w	x_3
cost(i, G) =	w	0	x_4

	Arginine (Arg)		
S =	C	G	T
cost(i, A) =	w	w	x_1
cost(i, T) =	w	w	0
cost(i, C) =	0	w	x_3
cost(i, G) =	w	0	x_4

	Arginine (Arg)		
S =	C	G	C
cost(i, A) =	w	w	x_1
cost(i, T) =	w	w	x_2
cost(i, C) =	0	w	0
cost(i, G) =	w	0	x_4

	Arginine (Arg)		
S =	C	G	G
cost(i, A) =	x_5	w	x_1
cost(i, T) =	w	w	x_2
cost(i, C) =	0	w	x_3
cost(i, G) =	w	0	0

	Arginine (Arg)		
S =	A	G	A
cost(i, A) =	0	w	0
cost(i, T) =	∞	w	w
cost(i, C) =	x_6	w	w
cost(i, G) =	w	0	x_4

	Arginine (Arg)		
S =	A	G	G
cost(i, A) =	0	w	x_1
cost(i, T) =	w	w	w
cost(i, C) =	x_6	w	w
cost(i, G) =	w	0	0

Table 2.1.8: Cost tables for Arginine

A similar situation arises with Glycine, Serine, and Leucine, which also have six codons and exhibit asymmetries in their substitution patterns.

In the Appendix A I provide a complete list of cost tables for all 64 codons, including symmetry grouping when possible. This approach, suggested by Zehavit's example, is simple. However, it has one major limitation. It is valid only when at most one base substitution is allowed per codon. Let's consider a few examples that demonstrate the potential limitations of this assumption.

Let's consider a case where codon GAA for Glutamic acid is substituted to codon TTA for Leucine. This substitution involves two base substitutions (in positions 1, 2). According to the cost table for Glutamic acid (see below), this costs ∞. However, the true biological cost of this substitution should be *w* because we are changing the amino acid to Leucine.

	Glutamic acid (Glu)		
S =	G	A	A/G
cost(i, A) =	w	0	x_1
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	0	w	x_2

Table 2.1.9: Combined cost table for Glutamic acid

Also, let's consider a case where codon GCA for Alanine is substituted to codon TAA for Stop codon. This substitution involves two base substitutions (in positions 1, 2). According to the cost table for Alanine (see below), this costs $2w$. However, the true biological cost of this substitution should be ∞ because we are reaching a stop codon which will stop the DNA translation.

	Alanine (Ala)		
S =	G	C	*
cost(i, A) =	w	w	x_1
cost(i, T) =	w	w	x_2
cost(i, C) =	w	0	x_3
cost(i, G) =	0	w	x_4

Table 2.1.10: Combined cost table for Alanine

The challenges outlined above reveal that the problem is more complex than Zehavit's example suggests. To handle the cases we described, we need to develop a more flexible scoring scheme—one that allows for more than a single substitution per codon while still meeting our goals.

2.2 A general sequence scoring scheme that allows more than one substitution per codon

2.2.1 Overview of modified cost function

To address the limitations discussed in Section 2.1—namely, that the scoring tables only handle cases with a single substitution per codon—the scoring scheme within coding regions must be modified. We implement this by redefining the cost function as $cost(i, v, \sigma)$, where:

- i represents the position along the sequence.
- v denotes the state of the Finite State Machine (FSM).
- σ is the next character in the sequence.

Furthermore, we assume that each state v in the FSM determines the last two bases in the *modified* sequence, which we denote as $last2(v)$ (in Section 2.3.2 we modify the KMP-based FSM to satisfy this requirement). Similarly, $last3(i)$ returns the last three bases (in positions $i-2, i-1$ and i) in the *target* sequence. We also define an auxiliary function $codonPos(i)$, which indicates the coding role of position i:

- $codonPos(i) = 0$ if i is not a position within a coding sequence.
- $codonPos(i) = 1, 2, 3$ depending on the position of i within a codon in the coding sequence.

Note that when $codonPos(i) = 3$, $last3(i)$ gives the sequence of the codon in the target sequence. With these definitions, the cost function is formalized as follows:

- If $codonPos(i) = 0$, then the $cost(i, v, \sigma)$ depends on the nucleotide substitution. Transition substitutions ($A \leftrightarrow G, C \leftrightarrow T$) incur a smaller cost of α, while transversion substitutions ($A, G \leftrightarrow C, T$) are assigned a higher cost of β, reflecting the higher prevalence of transition mutations.
- If $codonPos(i) \in \{1, 2\}$, then $cost(i, v, \sigma) = 0$ for all values of v and σ.
- If $codonPos(i) = 3$, the cost is determined based on $last3(i)$, $last2(v)$ and σ.

The modified scoring function loads the cost of all substitutions within a given codon onto the third base (i.e., when $codonPos(i) = 3$). This allows the function to account for multiple substitutions within the same codon—rather than just one—by following these guidelines:

- If the codon remains unchanged, a zero cost is assigned $cost(i, v, \sigma) = 0$.
- If the codon is altered but still encodes the same amino acid (i.e., a synonymous substitution), a relatively small cost is assigned. This cost depends on the specific substitution and is modeled as $cost(i, v, \sigma) = \varepsilon(last2(v)\sigma)$, where ε assigns a predefined cost based on the resulting codon formed by the last two bases of v and the new base σ. These costs are defined as $-log$ of the observed frequency of the corresponding codon (the frequencies and how we receive them are explained in Chapter 3).
- If the codon is altered to encode a different amino acid, a uniformly high cost w is assigned, with an additional penalty $\delta_{edit} \in \{1, 2, 3\}$ that reflects the edit distance between the original and substituted codons: $cost(i, v, \sigma) = w + \delta_{edit}(last3(i), last2(v)\sigma)$. This cost prioritizes non-synonymous substitutions that require fewer nucleotide changes.
- If a non-stop codon is altered to a stop codon or if a stop codon is altered to a non-stop codon, an infinite cost is assigned, reflecting that these substitutions are not allowed: $cost(i, v, \sigma) = \infty$.

In addition, substitutions that disrupt the start codon (i.e., the first ATG in a coding region) are assigned an infinite cost, as they prevent the initiation of translation and render the entire protein untranslatable. By contrast, internal ATG codons simply encode Methionine like any other amino acid, and substitutions within them are treated according to the standard codon substitution tables. This distinction is reflected both in the scoring function and in the cost tables provided in the appendix.

2.2.2 Examples of representative cases

For example, assume that $last3(i) = \text{TTT}$, meaning the target sequence has a codon TTT that encodes for Phenylalanine.

- If $last2(v) = \text{TT}$ and $\sigma = \text{T}$, then $cost(i, v, \sigma) = 0$, as the optimized sequence in positions $i-2$, $i-1$, and i remains unchanged relative to the target sequence TTT.
- If $last2(v) = \text{TT}$ and $\sigma = \text{C}$, then $cost(i, v, \sigma) = \varepsilon(\text{TTC})$, as TTT is substituted with TTC, still encoding Phenylalanine.
- If $last2(v) = \text{TA}$ and $\sigma \in \{\text{A}, \text{G}\}$, or $last2(v) = \text{TG}$ and $\sigma = \text{A}$, then $cost(i, v, \sigma) = \infty$ to reflect the formation of a stop codon.
- Otherwise, $cost(i, v, \sigma) = w + \delta_{edit}(last3(i), last2(v)\sigma)$, reflecting substitution with a codon for a different amino acid. For example, TTT $\rightarrow$ TAT changes phenylalanine to tyrosine. This substitution incurs a base cost of w and requires one nucleotide change, so $\delta_{edit}(\text{TTT}, \text{TAT}) = 1$, and hence $cost(i, v, \sigma) = w + 1$. Similarly, TTT $\rightarrow$ CAT changes phenylalanine to histidine, which requires two nucleotide changes, giving $\delta_{edit}(\text{TTT}, \text{CAT}) = 2$ and $cost(i, v, \sigma) = w + 2$.

In Section 2.1, we presented the score tables corresponding to Zehavit's original solution. Recall that these tables were designed to produce correct scores when at most one position is substituted per codon. This is the cost table for the target sequence TTT:

	Phenylalanine (Phe)		
S =	T	T	T
cost(i, A) =	w	w	w
cost(i, T) =	0	0	0
cost(i, C) =	w	w	x_2
cost(i, G) =	w	w	w

Table 2.2.1: Cost table for Phenylalanine (Phe) with TTT target sequence

This cost table and our modified cost function described above it produce consistent scores when at most one position is substituted:

1. **No substitution** TTT $\rightarrow$ TTT: Both frameworks assign a cost of 0.
2. **Single synonymous substitution** TTT $\rightarrow$ TTC: The cost table assigns x_2, while the scoring scheme assigns $\varepsilon(\text{TTC})$, both reflecting a low cost as the amino acid remains Phenylalanine.
3. **Substitution to another amino acid**: All other single substitutions (other than TTT$\rightarrow$TTC) are non synonymous and thus result in a codon for an amino acid different than Phenylalanine. The cost table assigns a cost of w, while the scoring scheme assigns a cost of $w + \delta_{edit}(\text{TTT}, last2(v)\sigma) = w + 1$. These differ, but the discrepancy is simply due to a shift in the baseline definition of w.

For multiple substitutions, the two frameworks produce different scores, as demonstrated by the following specific cases:

1. **Double substitution** TTT $\to$ GTC: The cost table assigns $2w$, while the scoring scheme assigns $w + \delta_{edit}(\text{TTT}, \text{GTC}) = w + 2$, avoiding compounded penalties for a single amino acid change (assuming that $w \gg 2$).
2. **Stop codon formation** TTT $\to$ TGA: The cost table assigns $2w$, allowing this substitution at a high cost, whereas the proposed scoring scheme assigns ∞ to strictly prohibit it.

* * *

Now, consider another example, where $last3(i) = \text{GCN}$ (when $\text{N} \in \{\text{A}, \text{T}, \text{C}, \text{G}\}$) implying that the target sequence has a codon GCN that encodes for Alanine.

- If $last2(v) = \text{GC}$, then $cost(i, v, \sigma)$ is determined as follows: If $\sigma = S_i$, then $cost(i, v, \sigma) = 0$ (no substitution). Otherwise, $cost(i, v, \sigma) = \varepsilon(\text{GC}\sigma)$, determined by the relative prevalence of codons GCσ in the target species.
- If $last2(v) = \text{TA}$ and $\sigma \in \{\text{A}, \text{G}\}$, or $last2(v) = \text{TG}$ and $\sigma = \text{A}$, then $cost(i, v, \sigma) = \infty$.
- Otherwise, $cost(i, v, \sigma) = w + \delta_{edit}(last3(i), last2(v)\sigma)$. For example, GCA $\to$ TCA changes Alanine to Serine, giving $\delta_{edit}(\text{GCA}, \text{TCA}) = 1$ and $cost(i, v, \sigma) = w + 1$. Similarly, GCA $\to$ TTT changes Alanine to Phenylalanine with $\delta_{edit}(\text{TTT}, \text{CAT}) = 3$, so $cost(i, v, \sigma) = w + 3$.

The following table illustrates the cost table corresponding to Zehavit's original solution for codon GCT:

	Alanine (Ala)		
$S =$	G	C	T
cost(i, A) =	w	w	x_1
cost(i, T) =	w	w	0
cost(i, C) =	w	0	x_3
cost(i, G) =	0	w	x_4

Table 2.2.2: Cost table for Alanine (Ala) with GCT target sequence

For cases where at most one position per codon is substituted, both the cost table and the modified cost function yield consistent results, capturing the expected substitution costs:

- **No substitution** GCT $\to$ GCT: Both frameworks assign a cost of 0.
- **Synonymous substitution** GCT $\to$ GCσ (when $\sigma \in \{\text{A}, \text{C}, \text{G}\}$): The cost table assigns x_i, while the scoring scheme assigns $\varepsilon(\text{GC}\sigma)$, both indicating a minor impact as the amino acid remains Alanine.

- **Substitution to another amino acid**: In all other cases, a single base substitution produces a codon for an amino acid other than Alanine. The cost table assigns a cost of w, while the scoring scheme gives $w+1$, differing only by a shift in the baseline definition of w.

For cases involving multiple substitutions, the two frameworks results in different scores, as demonstrated for the codon TTT.

* * *

Another interesting example is the codon group GAR (GAA or GAG) encode for Glutamic Acid, a single substitution can lead to the formation of a stop codon.

- If $last2(v) = \text{GA}$, then $cost(i, v, \sigma)$ is determined as follows: If $\sigma = S_i$, then $cost(i, v, \sigma) = 0$ (no substitution). If $\sigma \in \{\text{T}, \text{C}\}$ then $cost(i, v, \sigma) = w + \delta_{edit}(last3(i), last2(v)\sigma)$ reflecting a non-synonymous substitution. Otherwise, $cost(i, v, \sigma) = \varepsilon(\text{GA}\sigma)$, determined by the relative prevalence of the resulting codon for Glutamic Acid.
- If $last2(v) = \text{TA}$ and $\sigma \in \{\text{A}, \text{G}\}$, or $last2(v) = \text{TG}$ and $\sigma = \text{A}$, then $cost(i, v, \sigma) = \infty$.
- Otherwise, $cost(i, v, \sigma) = w + \delta_{edit}(last3(i), last2(v)\sigma)$.

The following table illustrates the cost table corresponding to Zehavit's original solution for codon GAA:

	Glutamic Acid (Glu)		
$S =$	G	A	A
cost(i, A) =	w	0	0
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	0	w	x_2

Table 2.2.3: Cost table for Glutamic Acid (Glu) with GAA target sequence

For cases where at most one position per codon is substituted, both the cost table and the scoring scheme yield consistent results, capturing the expected substitution costs:

- **No substitution** GAA $\rightarrow$ GAA: Both frameworks assign a cost of 0.
- **Synonymous substitution** GAA $\rightarrow$ GAG: The cost table assigns x_2, while the scoring scheme assigns $\varepsilon(\text{GAG})$, both indicating a minor impact as the amino acid remains Glutamic Acid.
- **Stop codon formation** GAA $\rightarrow$ TAA: Both frameworks assign ∞ to prohibit this substitution, as it would result in a stop codon.

- **Substitution to another amino acid**: In all other cases, a single base substitution produces a codon for an amino acid other than Glutamic Acid. The cost table assigns a cost of w, while the scoring scheme assigns $w+1$.

When we allow more than one substitution per codon, in some cases, the two schemes produce the same score. For example, the substitution GAA $\rightarrow$ TGA, which forms a stop codon, results in a cost of ∞ in both the cost table and the scoring scheme, strictly prohibiting the formation of a stop codon. However, in other cases, the two schemes result in very different outcomes. For example, consider a substitution GAA $\rightarrow$ TAC. This substitution changes the amino acid from Glutamic Acid to Tyrosine. The cost table assigns a cost of ∞, reflecting the two substitutions G $\rightarrow$ T and A $\rightarrow$ C. In contrast, the scoring scheme assigns a cost of $w+2$, which captures only the amino acid change.

* * *

In the final example, we consider the case of a codon encoding for Arginine. This amino acid is one of the three that have six codons encoding it, which serves a unique challenge. Formally, we consider the case where $last3(i) \in \{\text{CGN}, \text{AGR}\}$ (where N can be any base and R can be A or G).

- If $last2(v)\sigma = last3(i)$ (i.e., the codon remains the same), then $cost(i,v,\sigma) = 0$.
- Otherwise, if $last2(v) \in \{\text{CG}, \text{AG}\}$ and $\sigma \in \{\text{A}, \text{G}\}$, then $cost(i,v,\sigma) = \varepsilon(last2(v)\sigma)$.
- Otherwise, if $last2(v) = \text{CG}$ and $\sigma \in \{\text{C}, \text{T}\}$, then $cost(i,v,\sigma) = \varepsilon(last2(v)\sigma)$.
- If $last2(v) = \text{TA}$ and $\sigma \in \{\text{A}, \text{G}\}$, or $last2(v) = \text{TG}$ and $\sigma = \text{A}$, then $cost(i,v,\sigma) = \infty$.
- Otherwise, $cost(i,v,\sigma) = w + \delta_{edit}(last3(i), last2(v)\sigma)$.

The following table illustrates the corrected cost table corresponding to Zehavit's original solution for the codon AGA:

	Arginine (Arg)		
S =	A	G	A
cost(i, A) =	0	w	0
cost(i, T) =	∞	w	w
cost(i, C) =	x_6	w	w
cost(i, G) =	w	0	x_4

Table 2.2.4: Corrected cost table for Arginine (Arg) with AGA target sequence

For cases where at most one position per codon is substituted, both the cost table and the scoring scheme yield consistent results, capturing the expected substitution costs:

- **No substitution** AGA $\rightarrow$ AGA: Both frameworks assign a cost of 0.

- **Synonymous substitution** AGA $\rightarrow$ CGA: This is a notable case where a substitution in the first base of the codon results in a synonymous change. The cost table assigns a value of x_6, and the scoring scheme assigns $\varepsilon(\text{CGA})$, both reflecting the minor impact of the substitution, as the amino acid remains Arginine.
- **Stop codon formation** AGA $\rightarrow$ TGA: Both frameworks assign ∞ to strictly prohibit this substitution, as it results in a premature stop codon.
- **Substitution to another amino acid**: In all other cases, a single base substitution produces a codon for an amino acid other than Arginine. The cost table assigns a cost of w, while the scoring scheme assigns $w+1$.

For cases involving multiple substitutions, the cost table and the scoring scheme can diverge. Consider the substitution AGA $\rightarrow$ CGT, which requires two changes: A $\rightarrow$ C (cost x_6) and A $\rightarrow$ T (cost w), giving a total of $w+x_6$ in the cost table. In contrast, the scoring scheme assigns $\varepsilon(\text{CGT})$ to this double substitution, reflecting its minor functional consequence and aligning better with the functional implications of the substitution than the cost table.

2.2.3 Algorithm for computing the modified cost function

The algorithm computes a cost function for nucleotide substitutions, considering codon positions, codon usage frequencies, and the effects of synonymous, non-synonymous, and stop codon substitutions. It takes a nucleotide sequence, codon usage data, FSM states, and cost parameters as input, returning a detailed substitution cost table. It takes the following parameters as input:

- **Global sequence information:** target_sequence, coding_positions.
  - **target_sequence**: The original sequence provided by the user.
  - **coding_positions**: An array mapping each position to its corresponding codon position (1, 2, or 3) or to 0 if the position is non-coding. This array is computed in a preprocessing step using the segmentation of the target sequence into coding and non-coding regions. The segmentation itself is determined by the tool with assistance from the user, as described in Chapter 3
- **Biological cost parameters:** codon_usage, α, β, w.
  - **codon_usage**: Codon frequency table for synonymous substitution costs.
  - $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{w}$: Cost parameters representing the penalties for transitions, transversions, and non-synonymous substitutions, respectively.
- **Runtime substitution parameters:** i, v, σ.
  Recall that i is the current sequence position, v the FSM state, and σ the next input character.

Algorithm 3 Calculate Cost Function

Function CalculateCost(*target_sequence*, *coding_positions*, *codon_usage*, α, β, w, i, v, σ):
codon_pos = coding_positions[i] # Non-coding: 0; Coding: $((i - \text{coding\_start}) \mod 3) + 1$.
if $codon\_pos == 0$ **then** # Non-coding region
  if $target\_sequence[i] == \sigma$ **then**
    return 0 # No substitution
  else if IsTransition($target\_sequence[i], \sigma$) **then**
    return α # Transition substitution
  else
    return β # Transversion substitution
  end if
else if $codon\_pos \in \{1,2\}$ **then** # Cost is always 0 for positions 1 and 2
  return 0
else if $codon\_pos \in \{-3,3\}$ **then** # At 3rd position of codon
  $target\_codon \leftarrow$ GetLast3($target\_sequence, i$)
  $last2\_bases \leftarrow$ GetLast2(v)
  $proposed\_codon \leftarrow$ Concatenate($last2\_bases, \sigma$)
  if $proposed\_codon == target\_codon$ **then**
    return 0 # No substitution
  else if EncodesSameAminoAcid($proposed\_codon, target\_codon$) **then**
    return $-\log(codon\_usage[proposed\_codon])$ # Synonymous substitution
  else if $codon\_pos == -3$ **or** EitherIsStopCodon($target\_codon, proposed\_codon$) **then**
    return ∞ # Start or stop codon elimination or stop codon formation
  else
    return w + EditDist($target\_codon, proposed\_codon$) # Non-synonymous substitution
  end if
end if

2.3 Modifying the dynamic programming algorithm and the FSM to accommodate the new scoring scheme

In its current implementation, the KMP-based FSM functions correctly. However, the new scoring scheme assumes that every state in the FSM retains information about the last two bases in the (modified) sequence, $last2(v)$. This causes a problem, because some states in the KMP-based FSM do not encode this information. As a result, the codon cannot always be derived correctly, which causes a problem in computing accurate scores. We illustrate the implications of this issue through several representative cases in the following subsection.

2.3.1 Example of representative cases

For instance, consider the KMP-based FSM that generates all sequences that do not contain unwanted patterns P = {'ATATCA', 'TAGTAC'}. This FSM consists of states corresponding to all strict prefixes of the unwanted patterns. The transition function $f(v, \sigma)$ is defined as the longest suffix of $v\sigma$ that is a prefix of any unwanted pattern. In this context, invalid transitions refer to transitions that would lead to a complete unwanted pattern. The table representing this FSM is shown below. For completeness, note that the FSM states are all strict prefixes of unwanted patterns, and transitions are defined as the longest suffix of $v\sigma$, which is a prefix of an unwanted pattern. The two transitions marked as invalid do not appear in the actual FSM. They are included here for completeness, as they appear in the standard KMP FSM where they serve as transitions into accepting states. However, they are excluded from the FSM generated by Algorithm 2 (and subsequently provided as input to Algorithm 1), as they correspond to matches of an unwanted pattern.

State(v)	**f(v,'A')**	**f(v,'C')**	**f(v,'G')**	**f(v,'T')**
ε-initial state	A	ε	ε	T
A	A	ε	ε	AT
T	TA	ε	ε	T
AT	ATA	ε	ε	T
TA	A	ε	TAG	AT
ATA	A	ε	TAG	ATAT
TAG	A	ε	ε	TAGT
ATAT	ATA	ATATC	ε	T
TAGT	TAGTA	ε	ε	T
ATATC	Invalid transition	ε	ε	T
TAGTA	A	Invalid transition	TAG	AT

Table 2.3.1: KMP-based FSM table for P = {'ATATCA', 'TAGTAC'}

For most states, $last2(v)$ can be easily determined by extracting the last two characters from the state ID. For example, $last2(\text{'TAGT'}) = \text{'GT'}$. However, in this example, there are three states—{ε, 'A', 'T'}—for which $last2(v)$ cannot be reconstructed. To address this, we replace these three states with 14 new states that represent all pairs of bases that are not prefixes of unwanted patterns in P. This expansion ensures that all possible bigram combinations of the four bases ('A', 'T', 'C', 'G') are accounted for. The modified FSM is described in the table below:

State(v)	f(v,'A')	f(v,'C')	f(v,'G')	f(v,'T')
AA	AA	AC	AG	AT
AC	CA	CC	CG	CT
AG	GA	GC	GG	GT
AT	ATA	TC	TG	TT
CA	AA	AC	AG	AT
CC	CA	CC	CG	CT
CG	GA	GC	GG	GT
CT	TA	TC	TG	TT
GA	AA	AC	AG	AT
GC	CA	CC	CG	CT
GG	GA	GC	GG	GT
GT	TA	TC	TG	TT
TA	AA	AC	TAG	AT
TC	CA	CC	CG	CT
TG	GA	GC	GG	GT
TT	TA	TC	TG	TT
ATA	AA	AC	TAG	ATAT
TAG	GA	GC	GG	TAGT
ATAT	ATA	ATATC	TG	TT
TAGT	TAGTA	TC	TG	TT
ATATC	Invalid transition	CC	CG	CT
TAGTA	AA	Invalid transition	TAG	AT

Table 2.3.2: KMP-based FSM table after inclusion of all bigram states for P = {'ATATCA', 'TAGTAC'}.

Note that in this modified FSM, we do not have an initial state (ε). So, the dynamic programming algorithm is modified to operate without an initial state (see Section 2.3.3 below). In this modified FSM, every state explicitly retains information about the last two bases in the sequence. Furthermore, the FSM defined in Table 2.3.2 is equivalent to the one in Table 2.3.1 in that it generates the same set of sequences—those that do not contain patterns from P. The following section describes a general procedure for modifying the KMP-based FSM.

2.3.2 Modifying the KMP-based FSM to allow computation of last2(v)

The following algorithm modifies the KMP-based FSM transitions by refining state definitions and updating transitions to bigram-based states. The process consists of the following steps:

1. **Remove 'problematic' states:** Eliminate the initial state (ε) and all single-character states from the state set.
2. **Add all bigram states:** Construct all possible bigram combinations from the alphabet and include them in the state set. Since this step retains the original FSM's bigram states, it adds at most 16 states.

3. **Update the transition function:** For each transition $f(v,\sigma)$ that is from a state added in step 2 above or to a state removed in step 1, assign it to the bigram state formed by the last character of v and σ.

Algorithm 4 Update States and Transitions

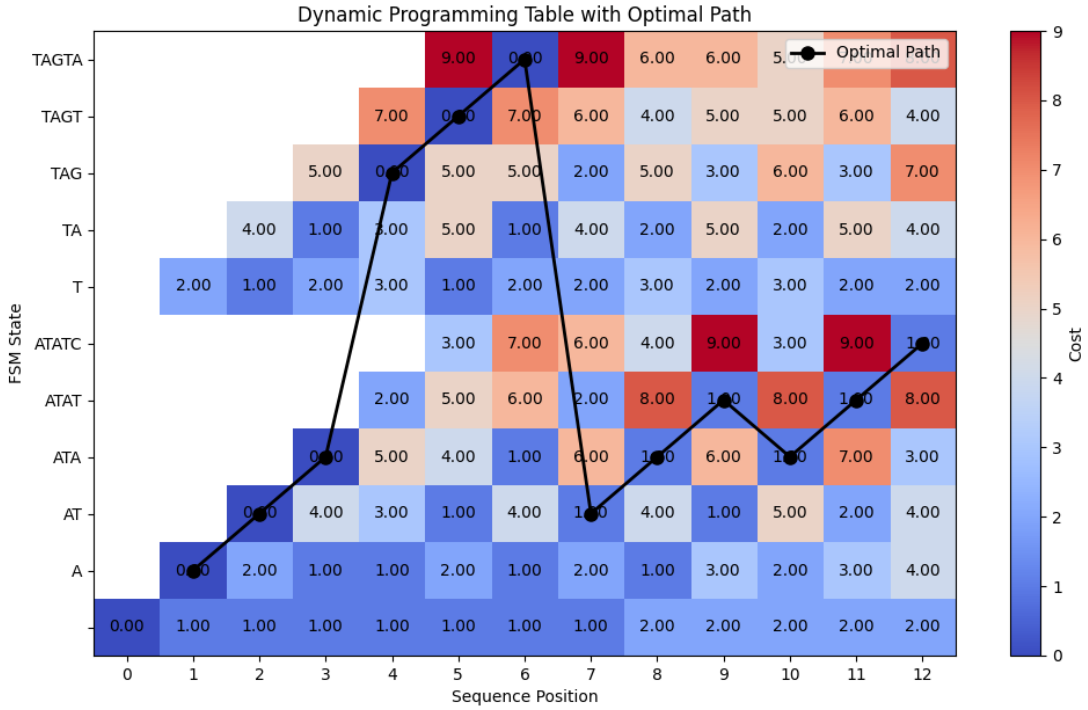
$V \leftarrow V \backslash (\Sigma \cup \{\varepsilon\})$ # Remove states that do not encode two characters
$V \leftarrow V \cup \{xy \mid x,y \in \Sigma\}$ # Expand V to include all bigrams over Σ

for each $v \in$ V **do**:
  for each $\sigma \in \Sigma$ **do**:
    if $f(v,\sigma)$ is not set yet **or** $f(v,\sigma) \in \Sigma \cup \{\varepsilon\}$ **then**
      $f(v,\sigma) \leftarrow v[-1] + \sigma$ # Update the transition function f
    end if
  end for
end for

Applying Algorithm 4 to a KMP-based FSM for a given set of unwanted patterns, P, produces an FSM that is equivalent to the original KMP-based FSM (as computed by Algorithm 2) in the sense that it generates all (and only) sequences that do not contain unwanted patterns in P. In addition, since the state labels correspond to a suffix of the generated string and all states of the modified FSM have labels of length at least 2, then $last2(v)$ can be computed for every state. Finally, this modification increases the FSM by at most 16 states, and typically at most 13 states, since the KMP-based FSM will have at least two 'problematic' states and at least one bigram state. Thus, the implication on the the running time of the dynamic programming algorithm (which is linear in $|V|$) is relatively minor.

2.3.3 Adapting the dynamic programming algorithm to the modified FSM

The dynamic programming (DP) algorithm (see Algorithm 1), which computes the optimal sequence, needs to be updated to reflect the modified FSM described above. This is because the new FSM does not include an initial state (ε), unlike the original version. In addition, the cost function now depends on the state v, which was not the case in the original algorithm. Consider an application of the original DP algorithm on the target sequence $S =$ 'ATAGTACATATC' with the KMP-based FSM for $P = \{$'ATATCA','TAGTAC'$\}$. The algorithm generates the following DP matrix:



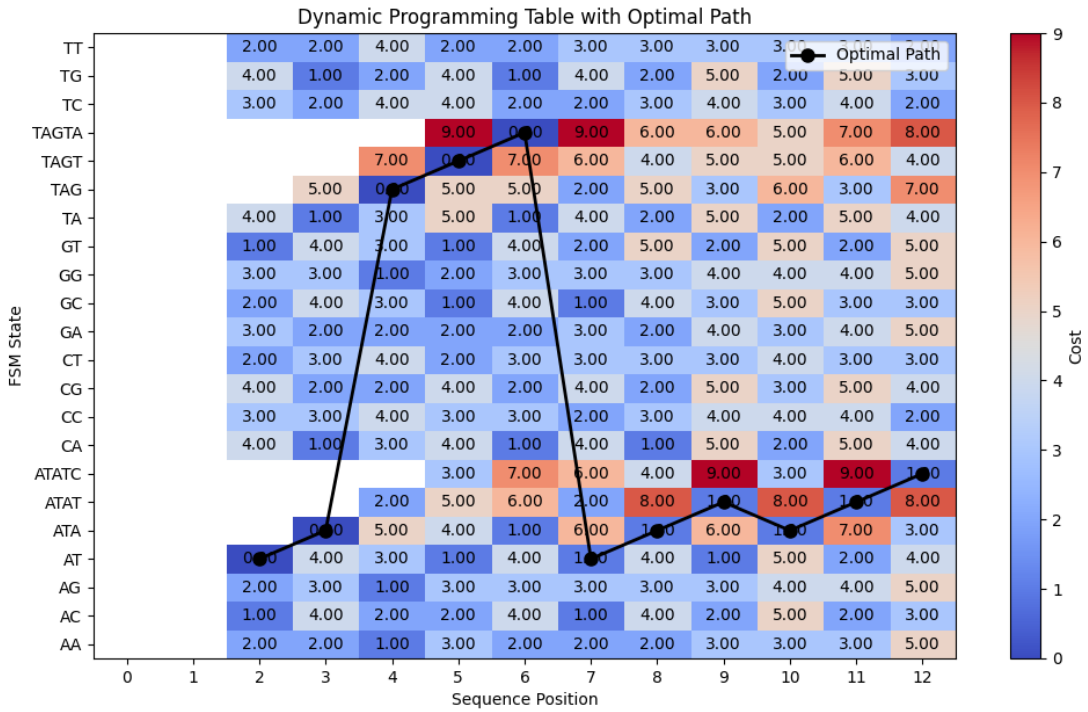
In this case, the minimal cost is 1.0, as shown in column 12, with the optimal path highlighted in black. To adapt the DP algorithm for the new FSM, we initialize the DP matrix in column 2, effectively discarding columns 0 and 1. The key modification involves initializing bigram states in column 2 of the DP table. The initial substitution costs for the first two positions are computed using initial_cost function below for each bigram state v, with the cost added to $A[(2,v)]$. This function uses the cost function associated with non-coding sequences. The DP updates remain the same, and the backtracking is slightly modified by stopping at column 2 and then using the state reached in column 2 to reconstruct the first two characters in the output sequence. The function initial_cost and the modified DP algorithm are described below:

Function initial_cost(i, σ):
if $target\_sequence[i] == \sigma$ **then**
    return 0 # No substitution
else if IsTransition($target\_sequence[i], \sigma$) **then**
    return α # Transition substitution
else
    return β # Transversion substitution
end if

Algorithm 5 Modified algorithm for Computing a min-cost $\mathscr{P}$-clean sequence of length n

Compute the **modified version** of the KMP-based FSM (V, f) for the set of unwanted patterns P by first applying Algorithm 2 (Zehavit's algorithm for computing the KMP-based FSM) and then modifying it according to Algorithm 4.
for $v \in V$ **do**:
  if $|v| == 2$ **then**:
    $A[2,v] = initial\_cost(1, v[0]) + initial\_cost(2, v[1])$
  else:
    $A[2,v] = \infty$
  end if
end for
for $i = 3...n$ and $v \in V$ **do**:
  $A^*[i,v] = (u^*, \sigma^*) = \underset{u,\sigma: f(u,\sigma)=v}{\operatorname{argmin}} \{A[i-1,u] + cost(i,u,\sigma)\}$
  $A[i,v] = A[i-1,u^*] + cost(i,u^*,\sigma^*)$
end for
$v_n = \underset{v \in V}{\operatorname{argmin}}\{A[n,v]\}$
for $i = n...3$ **do**:
  $(v_{i-1}, S_i) = A^*[i, v_i]$
end for
Return Concatenate(v_2, S)

The DP matrix produced by this algorithm on the target sequence $S =$ 'ATAGTACATATC' with the KMP-based FSM for $P = \{$'ATATCA','TAGTAC'$\}$ is given below:



Chapter 3

The BioSynth Application

3.1 Implementation Details

This section presents the construction of a Finite State Machine (FSM) that adheres to Zehavit's algorithm, emphasizing efficient computation of state transitions through table management for functions f and g, with the Python code in Algorithm 6 closely follows the pseudocode presented in Algorithm 2, demonstrating that it constitutes an accurate implementation of the algorithm. To accommodate the new cost function, after constructing the KMP-based FSM as described section 2.3, the algorithm proceeds by incorporating bigrams into the FSM, following the steps of Algorithm 4. This additional step ensures that the FSM accurately represents the structure required for computing the minimum-cost $\mathscr{P}$-clean sequence of length n with bigram states, effectively bridging the gap between the FSM construction and the elimination algorithm.

Algorithm 6 Python code for implementing Algorithms 2 and 4

```
1:  def kmp_based_fsm(unwanted_patterns, sigma) :
2:      f = {}
3:      g = {}
4:      states = set()
5:      epsilon = ''
6:
7:      for p in unwanted_patterns:
8:          for j in range(1, len(p) + 1):
9:              f[(p[: j − 1], p[j − 1])] = p[: j]
10:         f[(p[: −1], p[−1])] = None
```

```
11:     state_queue = deque()
12:     states.add(epsilon)
13:     for s in sigma:
14:         if (epsilon, s) not in f:
15:             f[(epsilon, s)] = epsilon
16:         if f[(epsilon, s)] == s:
17:             g[s] = epsilon
18:             state_queue.append(s)
19:
20:     while state_queue :
21:         v = state_queue.popleft()
22:         states.add(v)
23:
24:         for s in sigma:
25:             if f[g[v], s] is None:
26:                 f[(v, s)] = None
27:             if (v, s) not in f:
28:                 f[(v, s)] = f[(g[v], s)]
29:             if f[(v, s)] == v + s:
30:                 g[v + s] = f[(g[v], s)]
31:                 state_queue.append(v + s)
32:
33:     states.remove(epsilon)
34:     for s in sigma:
35:         if s in states:
36:             states.remove(s)
37:
38:     for (v, s) in f.copy():
39:         if v in sigma | {epsilon}:
40:             del (f[(v, s)])
41:
42:     for x in sigma:
43:         for y in sigma:
44:             states.add(x + y)
45:
46:     for v in states:
47:         for s in sigma:
48:             if (v, s) not in f or f[(v, s)] in sigma | {epsilon}:
49:                 f[(v, s)] = v[-1] + s
```

3.1.1 Comparison between FSM Algorithm Design and Implementation

The initial section of the algorithm focuses on initializing crucial variables essential for subsequent operations. These variables are defined as follows:

unwanted_patterns: Represented as a set denoted by P, capturing the collection of undesired patterns.

sigma: Denoted by Σ, signifying the alphabet of permissible characters.

states: Represented as a set denoted by V, embodying the states within the Finite State Machine (FSM).

f: A dictionary denoting the transition function of the FSM, mapping (state, character) pairs to new states, and denoted by f.

g: Another dictionary functioning as a helper to compute the transition function, denoted by g.

epsilon: A string represented by ε, symbolizing an empty string.

Implementation:

```
2:  f = {}
3:  g = {}
4:  states = set()
5:  epsilon = ''
```

Following the initialization, the algorithm progresses through distinct phases. The first phase focuses on computing transitions for pattern prefixes. Specifically, it iterates through each prefix of each pattern in *unwanted_patterns*, considering all possible extensions denoted as $f[(v,s)]$, where $vs \in \text{pref}(unwanted\_patterns) \setminus unwanted\_patterns$. It identifies elongations leading to complete patterns as invalid transitions, where $vs \in unwanted\_patterns$. Notably, certain prefix extensions may subsequently be recognized as invalid transitions when *vs* possesses a proper suffix within *unwanted_patterns*.

Implementation:

```
7:  for p in unwanted_patterns:
8:      for j in range(1, len(p) + 1):
9:          f[(p[: j − 1], p[j − 1])] = p[: j]
10:     f[(p[: −1], p[−1])] = None
11:
12:
```

Pseudo-code:

```
1: for p ∈ P do
2:     for j ∈ [1..|p| − 1] do
3:         Set f(p_{1..j−1}, p_j) ← p_{1..j}
4:     end for
5:     Set f(p_{1..|p|−1}, p_{|p|}) ← NULL
6: end for
```

In the second phase, the state space *states* is initialized with the initial state *epsilon*, and all transitions $f[(epsilon,s)]$ are examined. If $f[(epsilon,s)]$ hasn't been set in the previous phase, indicating that no pattern in *unwanted_patterns* starts with s, $f[(epsilon,s)]$ is set to *epsilon*. However, if $f[(epsilon,s)]$ was set in the first phase to s, it implies the existence of a pattern in P that starts with s, prompting the addition of s to the processing queue of states, along with the computation of $g[s] = epsilon$.

Implementation:

```
state_queue = deque()
states.append(epsilon)
for s in sigma:
    if (epsilon, s) not in f:
        f[(epsilon, s)] = epsilon
    if f[(epsilon, s)] == s:
        g[s] = epsilon
        state_queue.append(s)
```

Pseudo-code:

```
InitEmptyQueue(stateQueue)
V ← {ε}
for σ ∈ Σ do
    if f(ε, σ) is not set yet then
        Set f(ε, σ) ← ε
    end if
    if f(ε, σ) == σ then
        Set g(σ) ← ε
        stateQueue.push(σ)
    end if
end for
```

In the third phase, a breadth-first search approach is employed on the FSM graph, starting from the initial state *epsilon*. This phase, depicted in both pseudo-code and implementation code, iterates through each state in *states*, ensuring that all transitions $f[(g[v],s)]$ are appropriately set. Additionally, it handles cases where transitions are undefined, implying suffix presence in *unwanted_patterns*, and updates the processing queue accordingly. Through these phases, the algorithm systematically constructs the FSM and computes the transition function, facilitating subsequent pattern recognition tasks effectively.

Implementation:

```
while state_queue :
    v = state_queue.popleft()
    states.append(v)

    for s in sigma:
        if f[g[v], s] is None:
            f[(v, s)] = None
        if (v, s) not in f:
            f[(v, s)] = f[(g[v], s)]
        if f[(v, s)] == v + s:
            g[v + s] = f[(g[v], s)]
            state_queue.append(v + s)
```

Pseudo-code:

```
while stateQueue is not empty do
    v ← stateQueue.pop()
    V ← V ∪ {v}
    for σ ∈ Σ do
        if f(g(v), σ) == NULL then
            Set f(v, σ) ← NULL
        end if
        if f(v, σ) is not set yet then
            Set f(v, σ) ← f(g(v), σ)
        end if
        if f(v, σ) = vσ then
            Set g(vσ) ← f(g(v), σ)
            stateQueue.push(vσ)
        end if
    end for
end while
```

In the final phase, Algorithm 4 is implemented by adding bigram states to the FSM. First, all single-character states and the empty state (ε) are removed from the state set V and from the transition function f (line 1 in Algorithm 4; lines 33–40 in the code). Then, all possible bigram combinations xy for $x, y \in \Sigma$ are added to V (line 2 in Algorithm 4; lines 42–44 in the code). These additions extend the FSM to support transitions between bigram states, rather than relying on single-character states. Finally, for each state $v \in V$ and each symbol $\sigma \in \Sigma$, if $f(v,\sigma)$ is undefined or leads to a removed single-character state, the transition is redirected to $v[-1]+\sigma$, ensuring it points to a valid bigram state (lines 4–10 in Algorithm 4; lines 46–50 in the code).

Implementation:

```
33: states.remove(epsilon)
34: for s in sigma:
35:     if s in states:
36:         states.remove(s)
37:
38: for (v,s) in f.copy():
39:     if v in sigma | {epsilon}:
40:         del (f[(v,s)])
41:
42: for x in sigma:
43:     for y in sigma:
44:         states.add(x+y)
45:
46: for v in states:
47:     for s in sigma:
48:         if (v,s) not in f or
49:           f[(v,s)] in sigma | {epsilon}:
50:             f[(v,s)] = v[-1]+s
```

Pseudo-code:

```
1:  V ← V\(Σ ∪ {ε})
2:  V ← V ∪ {xy | x,y ∈ Σ}
3:
4:  for each v ∈ V do:
5:      for each σ ∈ Σ do:
6:          if f(v,σ) is not set yet or
              f(v,σ) ∈ Σ ∪ {ε} then
7:              f(v,σ) ← v[-1]+σ
8:          end if
9:      end for
10: end for
11:
12:
13:
14:
15:
16:
17:
```

3.2 BioSynth App

The BioSynth app was developed using a combination of Python, HTML, and CSS, employing various techniques to create a versatile tool. Developed on Apple Silicon Mac (macOS) using PyCharm, the app supports both Graphical User Interface (GUI) and Command Line Interface (CLI) execution, allowing for a comprehensive exploration of its functionalities. This section will detail the purpose of each module, outline user and system interactions, and discuss the components of the generated reports along with their significance.

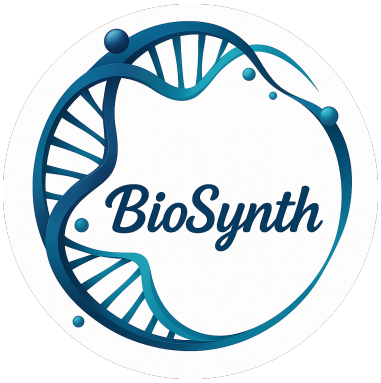


Figure 3.1: BioSynth logo.

To ensure code correctness and maintainability, development strictly adhered to a **Test-Driven Development (TDD)** workflow. Unit tests were authored prior to the implementation of each feature, allowing early identification of bugs, ensuring modular design, and delivering a highly reliable software product. The tests are executed daily through GitHub Actions, providing continuous validation of the codebase and ensuring long-term stability as the project evolves.

The app uses the following libraries:

- **Biopython/Bio**: Essential for biological data handling and analysis.
- **NumPy**: Enables fast numerical operations and array handling.
- **Jinja2**: Dynamically generates HTML templates for UI and reporting.
- **PyQt5**: Develops the GUI for an interactive user experience.
- **pywebview**: Integrates web-based content for a modern interface.

These libraries enable BioSynth to process biological data, present a user-friendly interface, and generate detailed reports effectively. The source code can be found here.

To operate the application, the user must provide the following **three input text files**:

1. **Target sequence file** – a plain text file containing a DNA sequence composed exclusively of the characters A, T, G, and C. The sequence should be provided in a **single continuous line**. For example:

   ```
   ATAGTACATATC
   ```

2. **Unwanted pattern list** – a plain text file containing DNA patterns (substrings) that should be eliminated from the target sequence. Each pattern should appear **in a separate line**, separated by whitespace. For example:

   ```
   TAGTAC
   ATATCA
   ```

3. **Codon usage file** – a plain-text file defining the relative codon usage frequencies for a specific organism. The file must consist of two columns separated by a tab: the codon (a three-letter DNA triplet) and its corresponding frequency. Each codon–frequency pair must appear **in a separate line**, separated by whitespace. Frequencies should be normalized such that, for each amino acid, the sum of the frequencies of all its synonymous codons equals 1. For example:

   ```
   TAT 0.49
   TAC 0.51
   ...
   ```

After validating the format, the program calculates substitution costs applying the negative base-10 logarithm of the frequency of each codon. For example, for codon TAC with frequency 0.56:

$$\varepsilon(\text{TAC}) = -\log_{10}(0.56) \approx 0.25$$

Codon usage information can be obtained from a variety of public databases. A detailed description of how to extract a codon usage table from the Kazusa database is given in Appendix B.

3.2.1 Components Overview

BioSynth is composed of several main components, each responsible for a specific function:

- **BioSynth:** The central class that coordinates interactions among the different modules.
- **UIController:** Handles the logic and flow of the user interface, ensuring smooth navigation between windows.
- **BaseWindow, UploadWindow, SettingsWindow, EliminationWindow, ResultsWindow:** These windows handle different stages of the program, from file uploads to presenting results.
- **CLIController, CommandController:** Manage CLI-based operations and command executions.
- **InputData, CostData, EliminationData, OutputData:** Singleton-style classes that hold the application's input, configuration parameters, intermediate state, and output results. They serve as centralized data holders accessed and modified by various controllers and views.
- **EliminationController, EliminationScorerConfig:** Manage the elimination of unwanted patterns and the associated scoring configurations
- **FSM (Finite State Machine):** Manages the states and transitions during the elimination process.
- **ReportController:** Compiles and organizes data into structured reports.

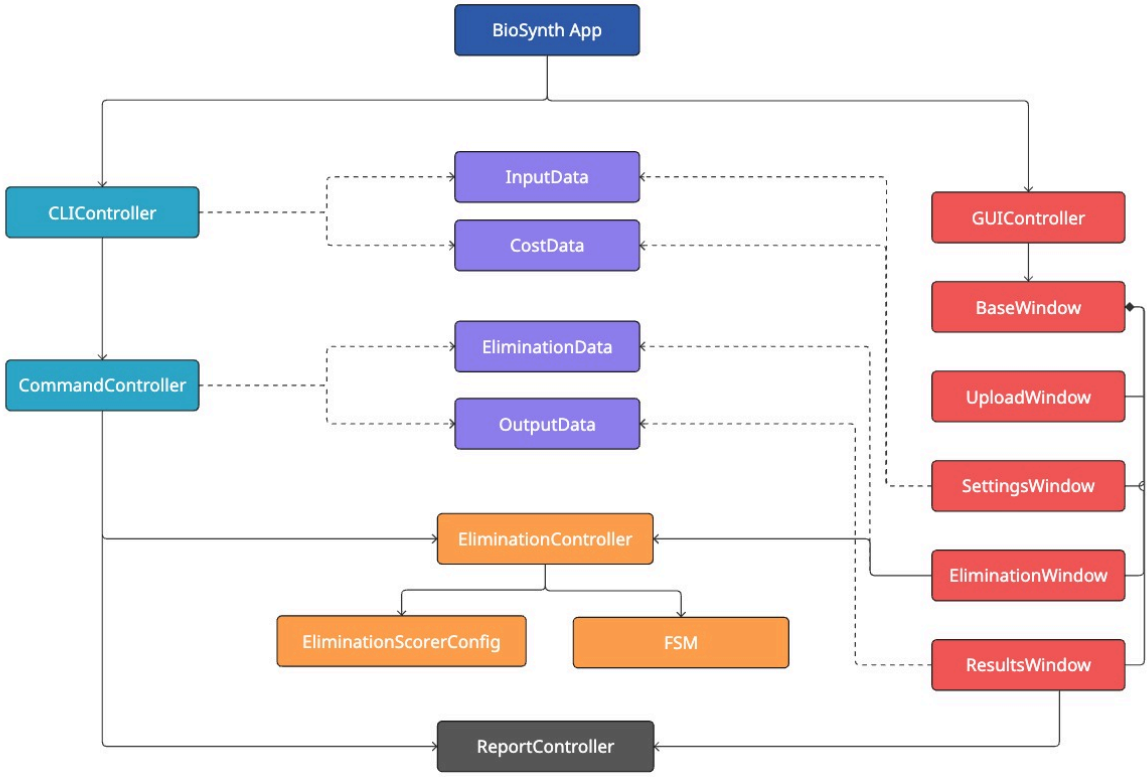


Figure 3.2: BioSynth main components class diagram.

Figure 3.2 shows the BioSynth class diagram, illustrating the main components and their interactions. Each node in the diagram represents a class, showing how different parts of the app interact with each other. The arrows and colors have specific meanings, as outlined below:

- **Nodes:** Each node represents a class within the application.
- **Colors:** Categorize classes by function:
  - **Blue:** Command-line interface (CLI) components.
  - **Red:** Graphical user interface (GUI) components.
  - **Orange:** Internal logic related to data processing, scoring, and elimination.
  - **Purple:** Global data holders accessed throughout the application.
- **Relationships:**
  - **Solid arrows:** Indicate direct relationships between classes (e.g., method calls or inheritance). An arrow ($\rightarrow$) represents a direct association — class A uses or communicates with class B, while a filled diamond ($\blacklozenge$) marks aggregation — class A contains class B as part of its structure (i.e., B does not exist independently of A).
  - **Dashed arrows:** Indicate indirect or weaker relationships, such as accessing global data (e.g., singleton), event-driven callbacks, or conceptual associations (e.g., controller $\rightarrow$ data model).

Design Principles

The architecture of the BioSynth application follows a modular and layered design, promoting the separation of concerns and maintainability. The graphical user interface (GUI) and command-line interface (CLI) components are fully decoupled, yet both interact with a common data layer and share the same elimination processing pipeline. Core data structures are implemented as singleton-style modules, enabling a shared-memory model in which centralized data objects serve as global state holders accessible to all controllers and views.

3.2.2 Open Reading Frame (ORF) Considerations

The program scans the target DNA sequence to identify all open reading frames (ORFs). An ORF is formally defined as a sequence that begins with a start codon (ATG) and ends with an in-frame stop codon (TAA, TAG, TGA). All reading frames are considered, and the program detects every such region, including ORFs that partially or fully overlap each other. While overlapping ORFs are detected, the program does not support their simultaneous use; instead, it applies distinct handling rules depending on the nature of the overlap, as detailed later in this section.

Let's consider the sequence s = ATGCAATGAGTTAA. The DNA sequence shown below contains two distinct ORFs that share part of their sequence but do not fully contain one another. ORF 1 starts at position 1 and ends at position 9, while ORF 2 starts at position 6 and ends at position 14.

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Sequence:	A	T	G	C	A	A	T	G	A	G	T	T	A	A
ORF 1:	A	T	G	C	A	A	T	G	A					
ORF 2:						A	T	G	A	G	T	T	A	A

Now consider the sequence s = ATGCATATGGTTTGA. In this sequence, ORF 1 spans the entire region from position 1 to position 15, while ORF 2 begins at position 7 and continues to position 15. This configuration results in a contained overlap, where ORF 2 shares its entire sequence with the latter portion of ORF 1.

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sequence:	A	T	G	C	A	T	A	T	G	G	T	T	T	G	A
ORF 1:	A	T	G	C	A	T	A	T	G	G	T	T	T	G	A
ORF 2:							A	T	G	G	T	T	T	G	A

BioSynth provides a set of internal rules that govern its behavior when identifying and processing ORFs. These **ORF-handling settings are fixed**, as they ensure biological reproducibility and consistency between runs, and currently cannot be adjusted by the user.

- **Minimum Codon Length:** Each coding region must contain at least 5 internal codons (excluding start and stop codons). This constraint ensures only substantial ORFs are considered for analysis.

- **Overlap Resolution Strategy:**
  - **Fully overlapping ORFs** – when one ORF contains another, then the contained ORF is not considered a potential coding sequence.
  - **Partially overlapping ORFs** – considered ambiguous; program execution is halted with an error message.

In cases where the target sequence contains multiple non-overlapping ORFs, some of these may not represent biologically coding sequences. To account for such situations, BioSynth allows users to exclude specific ORFs from the set of potential coding sequences:

- **GUI** - after ORF identification, users can manually deselect individual ORFs. This provides interactive control, enabling exploratory use in which only biologically relevant ORFs are retained.
- **CLI** - all detected non-overlapping ORFs are automatically treated as coding sequences, and manual exclusion is not supported. This design ensures reproducibility and aligns with the expected use of the CLI in automated pipelines.

3.2.3 Executing BioSynth

BioSynth can be executed through either the CLI or the GUI. Below are the steps for both methods.

Initial Setup

Before executing the application, ensure that BioSynth is correctly installed. To avoid potential dependency conflicts, it is recommended to create a dedicated Python virtual environment:

```
python -m venv biosynth_venv
source biosynth_venv/bin/activate
```

Subsequently, install the BioSynth package using pip:

```
pip install biosynth-tool
```

Launching the Application

Once installed via PyPI, BioSynth can be executed from any directory by entering the following command in the terminal:

```
biosynth
```

Data Input

Prior to executing BioSynth, users should prepare three input text files as outlined in Section 3.2. In addition, the application requires the specification of several operational parameters:

- **-g --gui**: Activates the graphical user interface (GUI) mode. When this option is selected, other parameters are ignored.
- **-s --target_sequence**: Specifies the path to the target sequence file; this option is mandatory when running without a GUI.
- **-p --unwanted_patterns**: Specifies the path to the unwanted patterns file; this option is mandatory when running without a GUI.
- **-c --codon_usage**: Specifies the path to the codon usage file; this option is mandatory when running without a GUI.
- **-o --out_dir**: Defines the output directory path; Optional parameter, default value is the downloads directory.
- **-a --alpha**: Specifies the value for the transition substitution cost; optional parameter, default value is 1.0.
- **-b --beta**: Specifies the value for the transversion substitution cost; optional parameter, default value is 2.0.
- **-w --non_synonymous_w**: Specifies the value for non-synonymous substitution cost; optional parameter, default value is 100.0.

Execution and Monitoring

Once the input files and settings are configured, the program will display real-time logs, providing feedback on the program's progress. Users can monitor these logs to gain insights into the operations being performed and to identify any potential issues.

3.2.4 Graphical User Interface (GUI) Execution

Navigating the Interface

Upon launching the application, users will be greeted by the main dashboard.

The UI is organized into several sections to facilitate easy navigation:

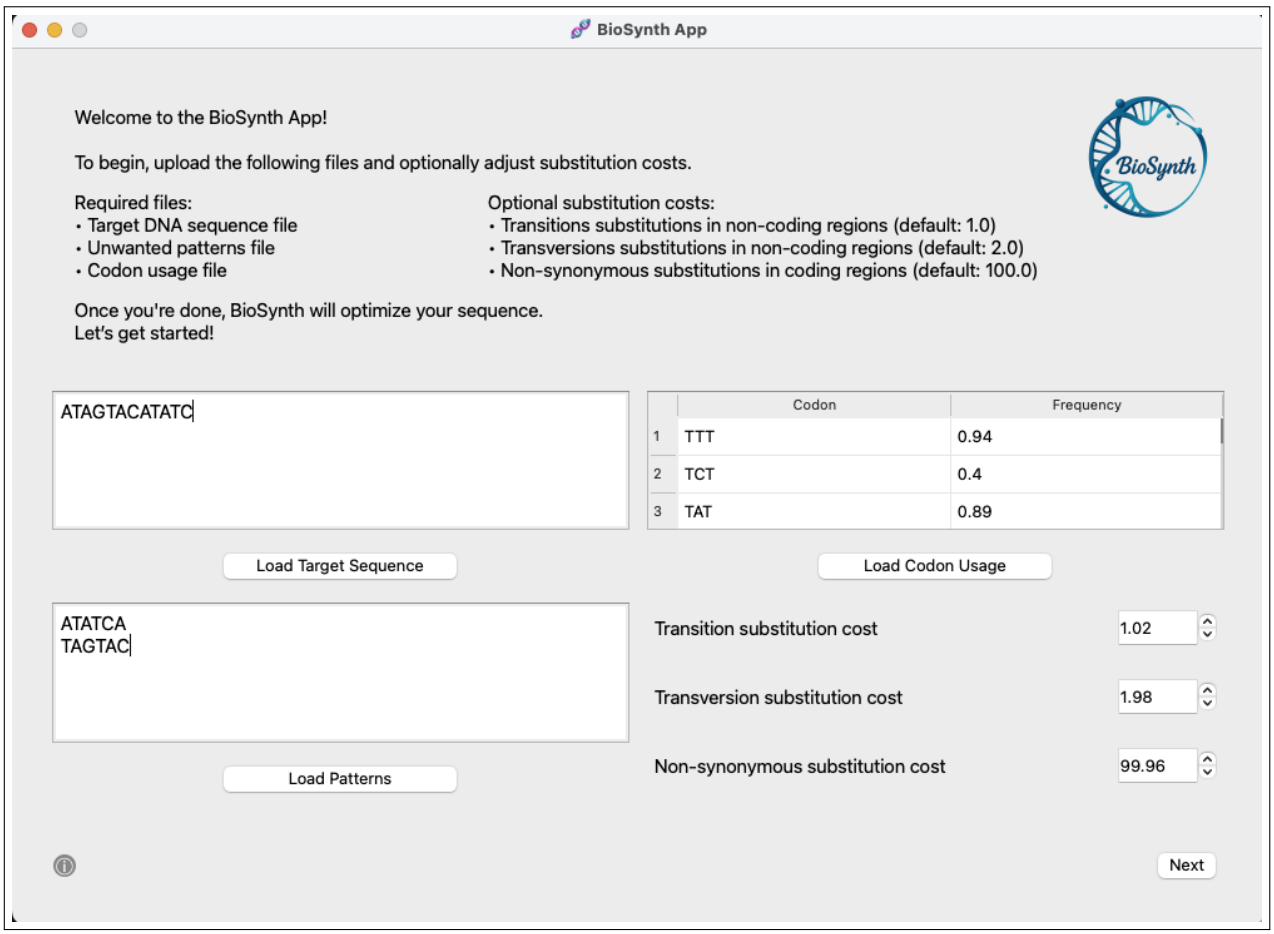
- **Upload Window – File Selection**: Enables users to select the target sequence file, the unwanted patterns file, and the codon usage file. It also allows configuration of the substitution costs: α, β, and w.

- **Settings Window**: Provides options to configure program settings, such as the exclusion of certain ORFs from the set of coding regions.
- **Elimination Window**: Displays a detailed overview of the elimination process, including the costs associated with coding and non-coding regions, as well as the total cost post-elimination, similar to the CLI output.
- **Results Window**: Enables users to interact with the results, view the cost associated with each change, download the DNA optimized sequence as a text file, and review the elimination report for the specific run.

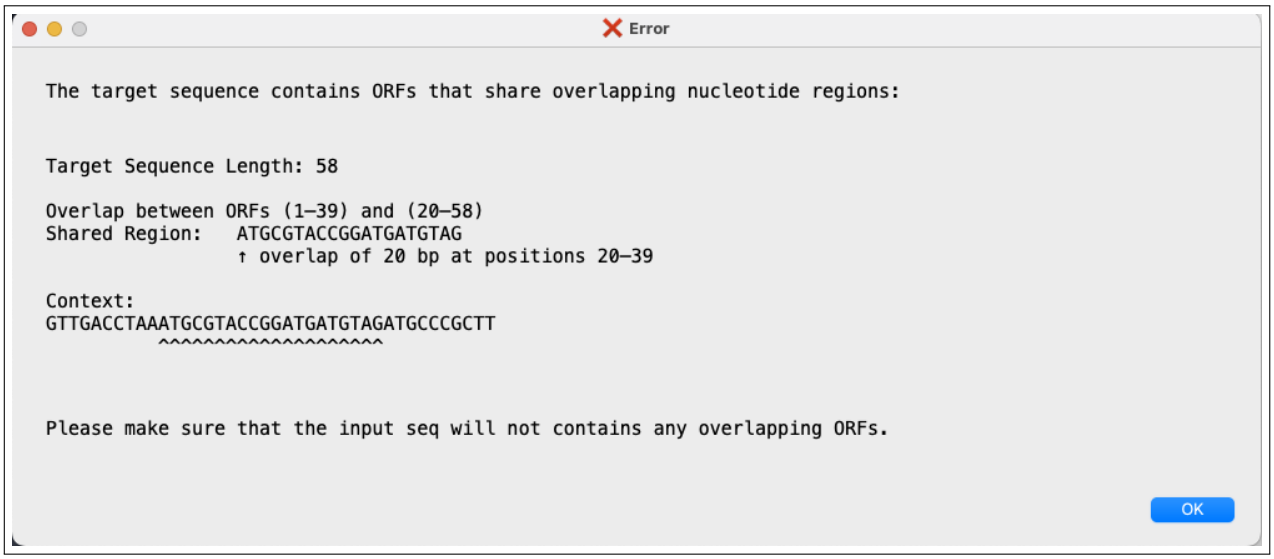
Upon completion, the program provides a summary of the results in the Results window and saves the processed data to the specified output directory. Users can view detailed logs and download the output files directly from the GUI.

Execution Example

The following walkthrough demonstrates how the program operates using the GUI with a simple example. The process begins by uploading the target sequence file, patterns file, and the codon usage file in the upload window, where users also have the option to configure cost function parameters. For more information about the program and its limitations, the user can click on the 'i' button.

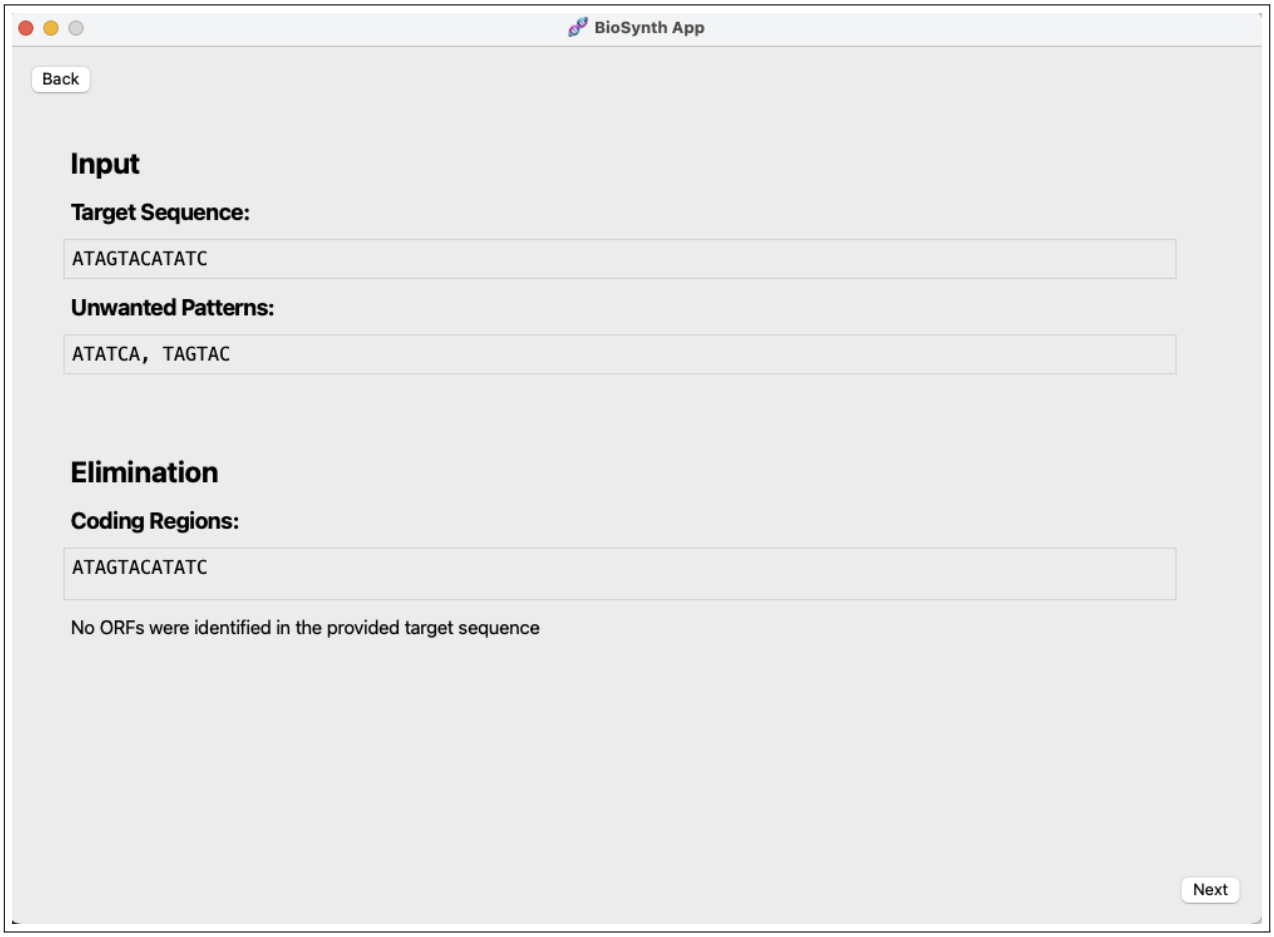


If the program detects partial overlap between ORFs, an error message is displayed to prevent ambiguous processing:

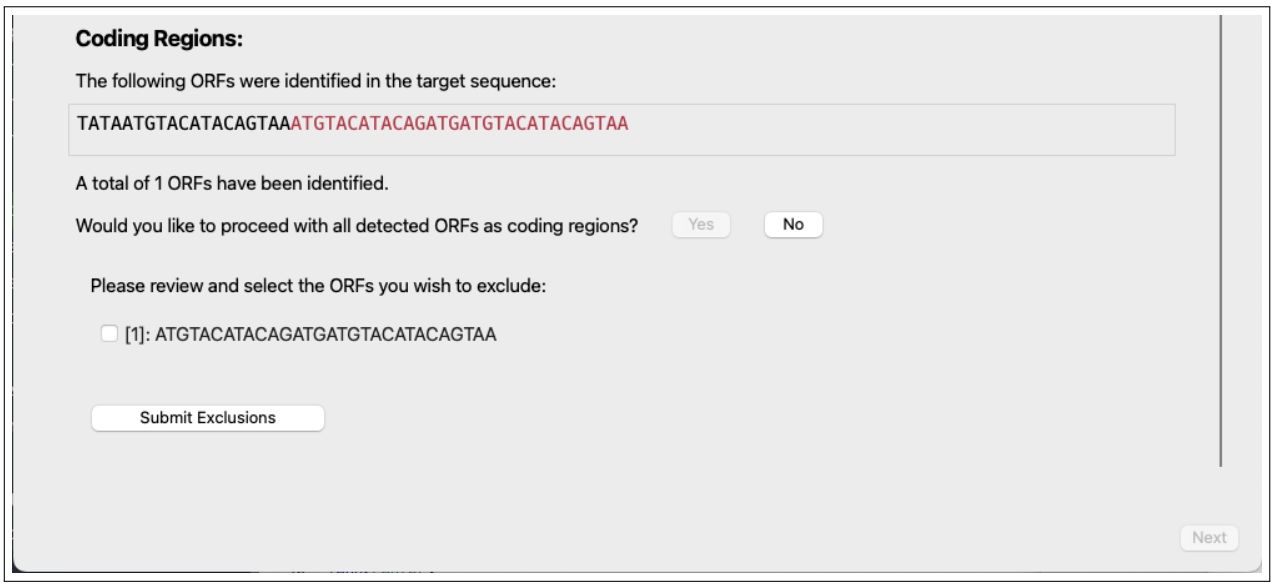


After the input was processed and checked for no partial overlaps, the user can click 'Next' to transition into the Settings window and see target sequence, the unwanted patterns, and the partitioned

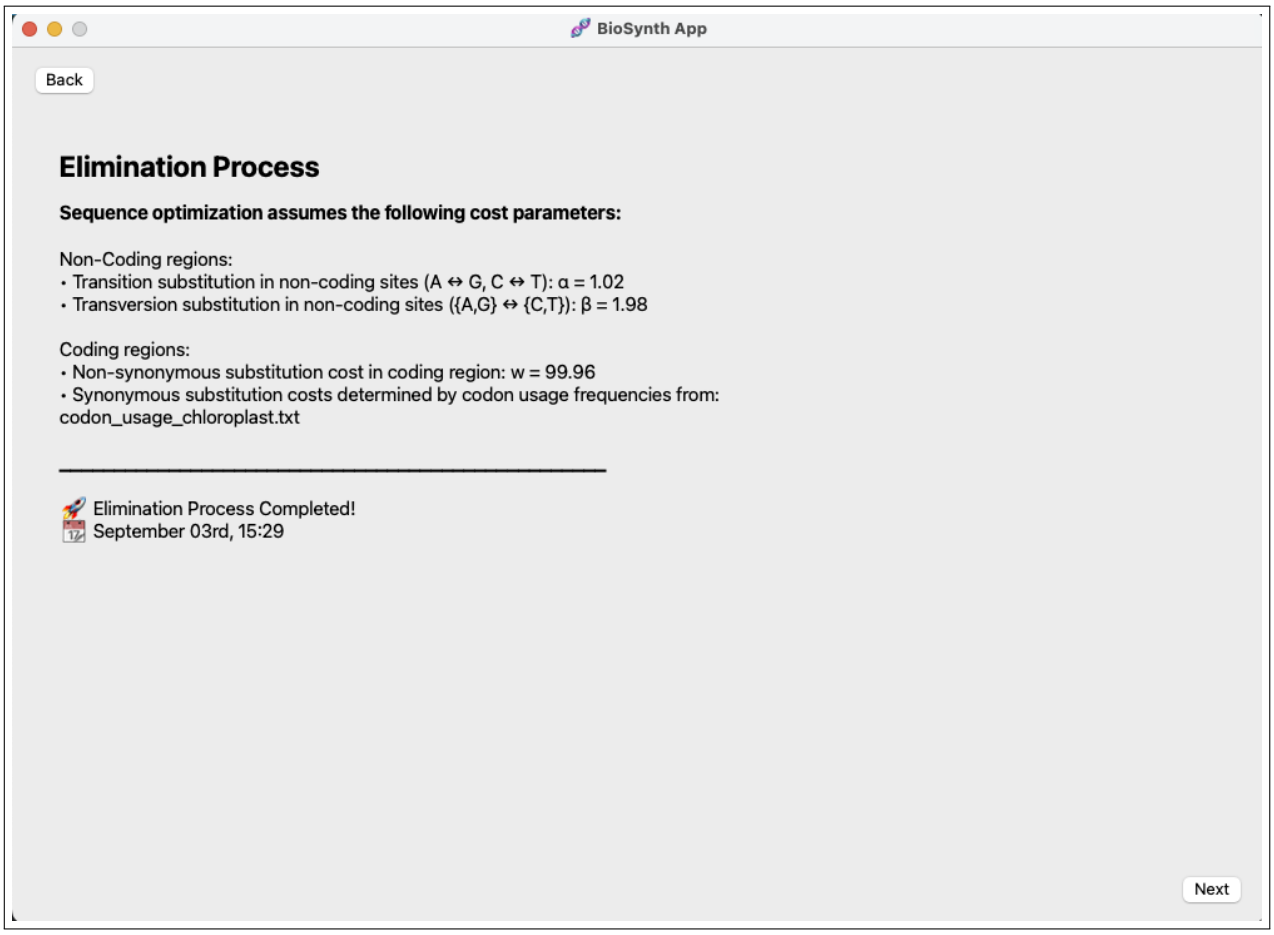
coding and non-coding regions. At this stage, the user can choose to exclude some of the ORFs from the set of coding regions.



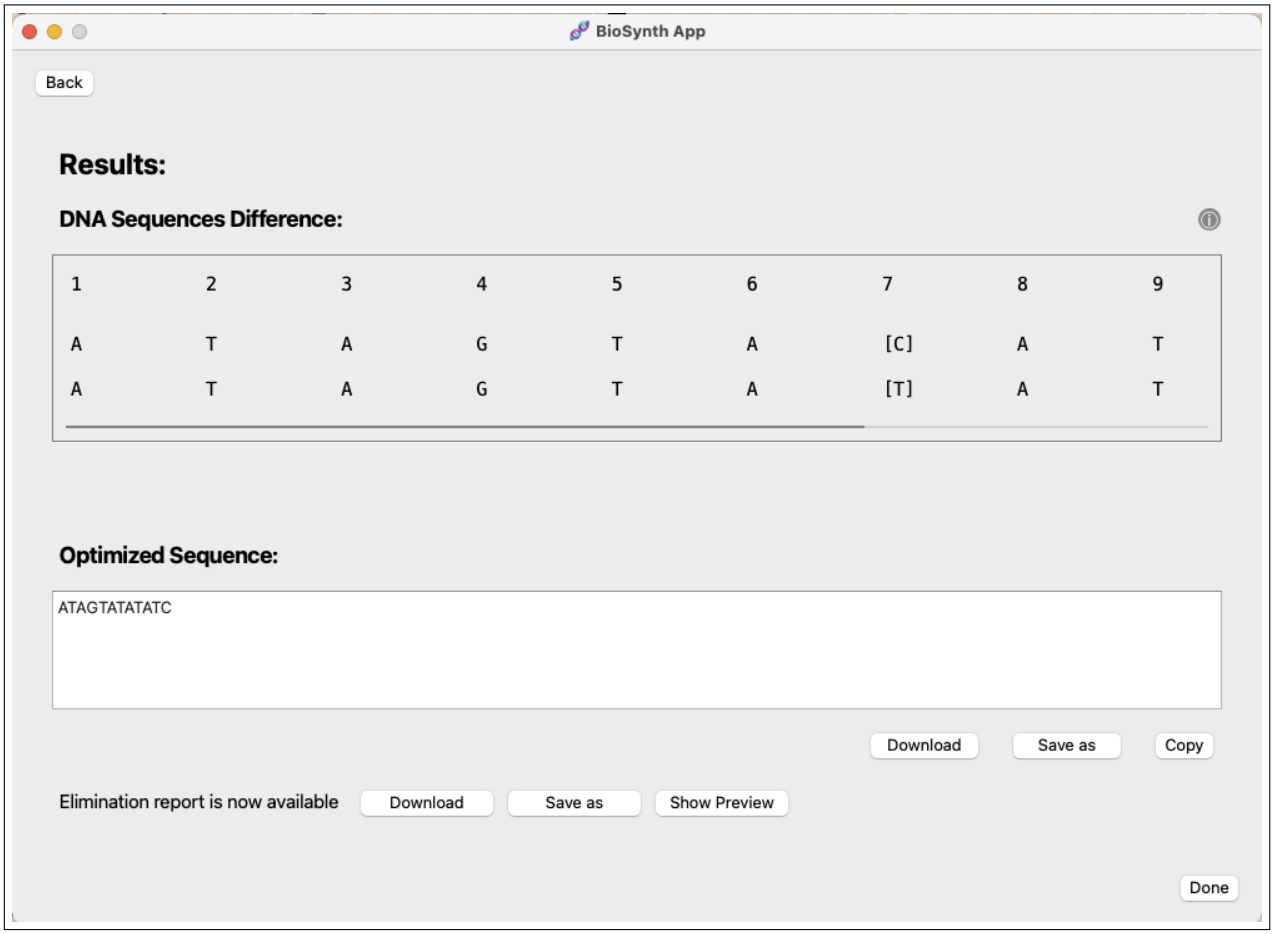
In case there is a coding region in the sequence, there is an option to exclude some of them. The user is prompted by an option to proceed ('Yes') or to exclude some ORFs ('No'). If the user selects 'No', they will be prompted with a list of coding regions to exclude from further processing:



Clicking 'Next' again takes the user to the Elimination window, which shows the progress of the elimination algorithm. This screen presents logs identical to those shown during CLI execution, including the input data, a step-by-step account of the elimination process, and the final results:



Once the elimination process is completed, the user can review the modifications made to the target sequence in the Results window:



Each modification is associated with a location-based cost, accessible via the 'i' button. Users can also preview and download a full elimination report. Additional examples covering various biological scenarios can be found in Appendix C.

The following shows the final report generated in HTML format:

runi.ac.il hadar.pur@post.runi.ac.il 02Aug2025,14:30:50

Input
Target Sequence
Unwanted Patterns

Elimination
Elimination Process

Results
DNA Sequences Difference
Detailed substitutions relative to the target sequence
Optimized Sequence

Input

Target Sequence:

ATAGTACATATC

Unwanted Patterns:

ATATCA, TAGTAC

Elimination

No ORFs were identified in the provided target sequence.

Elimination Process

Sequence optimization assumes the following cost parameters:

Non-Coding regions:

• Transition substitution in non-coding sites (A ↔ G, C ↔ T): α = 1.02

• Transversion substitution in non-coding sites ({A,G} ↔ {C,T}): β = 1.98

Coding regions:

• Non-synonymous substitution cost in coding region: w = 99.96

• Synonymous substitution costs determined by codon usage frequencies from: biosynth_codon_usage.txt

Results

DNA Sequences Difference:

1	2	3	4	5	6	7
A	T	A	G	T	A	[C]
A	T	A	G	T	A	[T]

Detailed substitutions relative to the target sequence:

Position 7 C -> T Cost: 1.02

Optimized Sequence:

ATAGTATATC

3.2.5 CLI Execution

The CLI execution method allows users to run the program in an automated non-interactive mode using the default or specified settings. The required inputs are the target sequence file, unwanted patterns file, and codon usage file, which can be provided using either short or long-form options (see Subsection 3.2.3 for the full list of available flags).

Command Syntax

```
biosynth -s <target_sequence_file_path> -p <unwanter_pattern_file_path> -c <
   codon_usage_file_path>
```

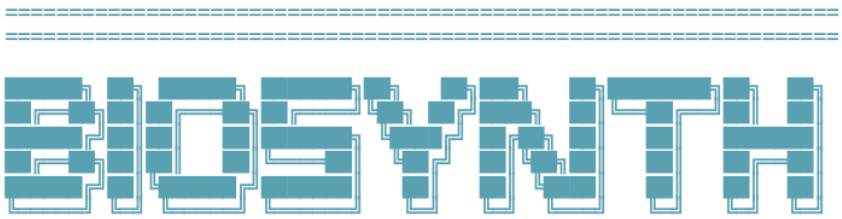
Results and Output

Upon completion, the program will display the input data, elimination process and optimized sequence on the terminal (similar to the Elimination window in 3.2.4) and save the optimized sequence to the output file along with the elimination report. These files will contain the necessary information for further analysis or review.

Execution Example

```
biosynth -s s_file_no_coding.txt -p p_file_no_coding.txt -c
   biosynth_codon_usage.txt -a 1.02 -b 1.98 -w 99.96
```

The corresponding textual output generated from this execution is presented on the subsequent page.



Target sequence:
ATAGTACATATC

Unwanted patterns:
ATATCA, TAGTAC

No ORFs were identified in the provided target sequence.

Sequence optimization assumes the following cost parameters:

Non-Coding regions:
• Transition substitution in non-coding sites (A ↔ G, C ↔ T): α = 1.02
• Transversion substitution in non-coding sites ({A,G} ↔ {C,T}): β = 1.98

Coding regions:
• Non-synonymous substitution cost in coding region: w = 99.96
• Synonymous substitution costs determined by codon usage frequencies from:
codon_usage_chloroplast.txt

🚀 Elimination Process Completed!
📆 September 02nd, 19:37

Optimized Sequence:
ATAGTATATATC

Detailed substitutions relative to the target sequence:
Position 7 C -> T Cost: 1.02

The final report and optimized sequence can be found in the following paths:
/Users/hadarpur/Downloads/BioSynth-Outputs/BioSynth-Report_02-Sep-2025_19-37-54.html
/Users/hadarpur/Downloads/BioSynth-Outputs/Optimized-Sequence_02-Sep-2025_19-37-54.txt

Bibliography

[1] Zehavit Leibovich. “Eliminating unwanted patterns with minimal interferences”. In: (June 2021). URL: https://www.runi.ac.il/media/iawjvg5e/zehavit-thesis-final.pdf.

[2] Zehavit Leibovich and Ilan Gronau. “Optimal Design of Synthetic DNA Sequences Without Unwanted Binding Sites”. In: *Journal of Computational Biology* 29.9 (Sept. 2022), pp. 663–670. URL: https://www.liebertpub.com/doi/10.1089/cmb.2021.0417.

Appendix A

Amino Acids Scoring Scheme

A.1 Symmetric amino acids scoring scheme

There are 16 amino acids with symmetric cost tables, allowing them to be summarized using a single cost table.

Two amino acids (Tryptophan and Methionine) are coded by only 1 codon:

A.1.1 Tryptophan (Trp)

	Tryptophan (Trp)		
S =	T	G	G
cost(i, A) =	w	∞	∞
cost(i, T) =	0	w	w
cost(i, C) =	w	w	w
cost(i, G) =	w	0	0

A.1.2 Methionine (Met)

The following tables present the cost values for Methionine (Met) depending on the context in which the ATG codon appears. When ATG functions as a start codon, substitutions are not allowed to preserve the functionality of translation initiation. However, when ATG appears internally within a protein sequence, substitutions are allowed; we treat it like any other codon, and the costs reflect non-synonymous substitutions, reflecting the greater flexibility in that context.

	Methionine (Met) — Start		
S =	A	T	G
cost(i, A) =	0	∞	∞
cost(i, T) =	∞	0	∞
cost(i, C) =	∞	∞	∞
cost(i, G) =	∞	∞	0

	Methionine (Met) — Internal		
S =	A	T	G
cost(i, A) =	0	*w*	*w*
cost(i, T) =	*w*	0	*w*
cost(i, C) =	*w*	*w*	*w*
cost(i, G) =	*w*	*w*	0

The other 14 amino acids coded by multiple codons that differ in their 3rd base.

A.1.3 Phenylalanine (Phe)

	Phenylalanine (Phe)		
S =	T	T	T
cost(i, A) =	w	w	w
cost(i, T) =	0	0	0
cost(i, C) =	w	w	x
cost(i, G) =	w	w	w

	Phenylalanine (Phe)		
S =	T	T	C
cost(i, A) =	w	w	w
cost(i, T) =	0	0	x
cost(i, C) =	w	w	0
cost(i, G) =	w	w	w

These two tables can be combined into one as follows:

	Phenylalanine (Phe)		
S =	T	T	T/C
cost(i, A) =	w	w	w
cost(i, T) =	0	0	x
cost(i, C) =	w	w	x
cost(i, G) =	w	w	w

A.1.4 Histidine (His)

	Histidine (His)		
S =	C	A	T
cost(i, A) =	w	0	w
cost(i, T) =	w	w	0
cost(i, C) =	0	w	x
cost(i, G) =	w	w	w

	Histidine (His)		
S =	C	A	C
cost(i, A) =	w	0	w
cost(i, T) =	w	w	x
cost(i, C) =	0	w	0
cost(i, G) =	w	w	w

These two tables can be combined into one as follows:

	Histidine (His)		
S =	C	A	T/C
cost(i, A) =	w	0	w
cost(i, T) =	w	w	x
cost(i, C) =	0	w	x
cost(i, G) =	w	w	w

A.1.5 Asparagine (Asn)

	Asparagine (Asn)		
S =	A	A	T
cost(i, A) =	0	0	w
cost(i, T) =	w	w	0
cost(i, C) =	w	w	x
cost(i, G) =	w	w	w

	Asparagine (Asn)		
S =	A	A	C
cost(i, A) =	0	0	w
cost(i, T) =	w	w	x
cost(i, C) =	w	w	0
cost(i, G) =	w	w	w

These two tables can be combined into one as follows:

	Asparagine (Asn)		
S =	A	A	T/C
cost(i, A) =	0	0	w
cost(i, T) =	w	w	x
cost(i, C) =	w	w	x
cost(i, G) =	w	w	w

A.1.6 Aspartic acid (Asp)

	Aspartic acid (Asp)		
S =	G	A	T
cost(i, A) =	w	0	w
cost(i, T) =	w	w	0
cost(i, C) =	w	w	x
cost(i, G) =	0	w	w

	Aspartic acid (Asp)		
S =	G	A	C
cost(i, A) =	w	0	w
cost(i, T) =	w	w	x
cost(i, C) =	w	w	0
cost(i, G) =	0	w	w

These two tables can be combined into one as follows:

	Aspartic acid (Asp)		
S =	G	A	T/C
cost(i, A) =	w	0	w
cost(i, T) =	w	w	x
cost(i, C) =	w	w	x
cost(i, G) =	0	w	w

A.1.7 Tyrosine (Tyr)

	Tyrosine (Tyr)		
S =	T	A	T
cost(i, A) =	w	0	∞
cost(i, T) =	0	w	0
cost(i, C) =	w	w	x
cost(i, G) =	w	w	∞

	Tyrosine (Tyr)		
S =	T	A	C
cost(i, A) =	w	0	∞
cost(i, T) =	0	w	x
cost(i, C) =	w	w	0
cost(i, G) =	w	w	∞

In this case, if we replace the 3rd base with C or G it will produce a stop codon, so we must take it under consideration. Therefore, these two tables can be combined into one as follows:

	Tyrosine (Tyr)		
S =	T	A	T/C
cost(i, A) =	w	0	∞
cost(i, T) =	0	w	x
cost(i, C) =	w	w	x
cost(i, G) =	w	w	∞

A.1.8 Glutamine (Gln)

	Glutamine (Gln)		
S =	C	A	A
cost(i, A) =	w	0	0
cost(i, T) =	∞	w	w
cost(i, C) =	0	w	w
cost(i, G) =	w	w	x

	Glutamine (Gln)		
S =	C	A	G
cost(i, A) =	w	0	x
cost(i, T) =	∞	w	w
cost(i, C) =	0	w	w
cost(i, G) =	w	w	0

In this case, if we replace 1st base C with T it will produce a stop codon, so we must take it under consideration. Therefore, these two tables can be combined into one as follows:

	Glutamine (Gln)		
S =	C	A	A/G
cost(i, A) =	w	0	x
cost(i, T) =	∞	w	w
cost(i, C) =	0	w	w
cost(i, G) =	w	w	x

A.1.9 Lysine (Lys)

	Lysine (Lys)		
S =	A	A	A
cost(i, A) =	0	0	0
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	w	w	x

	Lysine (Lys)		
S =	A	A	G
cost(i, A) =	0	0	x
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	w	w	0

In this case, if we replace the 1st base A with T it will produce a stop codon, so we must take it under consideration. Therefore, these two tables can be combined into one as follows:

	Lysine (Lys)		
S =	A	A	A/G
cost(i, A) =	0	0	x
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	w	w	x

A.1.10 Glutamic acid (Glu)

	Glutamic acid (Glu)		
S =	G	A	A
cost(i, A) =	w	0	0
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	0	w	x

	Glutamic acid (Glu)		
S =	G	A	G
cost(i, A) =	w	0	x
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	0	w	0

In this case, if we replace the 1st base G with T it will produce a stop codon, so we must take it under consideration. Therefore, these two tables can be combined into one as follows::

	Glutamic acid (Glu)		
S =	G	A	A/G
cost(i, A) =	w	0	x
cost(i, T) =	∞	w	w
cost(i, C) =	w	w	w
cost(i, G) =	0	w	x

A.1.11 Cysteine (Cys)

	Cysteine (Cys)		
S =	T	G	T
cost(i, A) =	w	w	∞
cost(i, T) =	0	w	0
cost(i, C) =	w	w	x
cost(i, G) =	w	0	w

	Cysteine (Cys)		
S =	T	G	C
cost(i, A) =	w	w	∞
cost(i, T) =	0	w	x
cost(i, C) =	w	w	0
cost(i, G) =	w	0	w

In this case, if we replace the 3rd base with A it will produce a start codon, so we must take it under consideration. Therefore, these two tables can be combined into one as follows:

	Cysteine (Cys)		
S =	T	G	T/C
cost(i, A) =	w	w	∞
cost(i, T) =	0	w	x
cost(i, C) =	w	w	x
cost(i, G) =	w	0	w

A.1.12 Isoleucine (Ile)

	Isoleucine (Ile)		
S =	A	T	T
cost(i, A) =	0	w	x
cost(i, T) =	w	w	0
cost(i, C) =	w	w	x
cost(i, G) =	w	0	w

	Isoleucine (Ile)		
S =	A	T	C
cost(i, A) =	0	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	w	0
cost(i, G) =	w	0	w

	Isoleucine (Ile)		
S =	A	T	A
cost(i, A) =	0	w	0
cost(i, T) =	w	w	x
cost(i, C) =	w	w	x
cost(i, G) =	w	0	w

In this case, if we replace the 3rd base with G it will produce a start codon, so we must take it under consideration. Therefore, These three tables can be combined into one as follows:

	Isoleucine (Ile)		
S =	A	T	T/C/A
cost(i, A) =	0	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	w	x
cost(i, G) =	w	0	w

A.1.13 Alanine (Ala)

	Alanine (Ala)		
S =	G	C	T
cost(i, A) =	w	w	x
cost(i, T) =	w	w	0
cost(i, C) =	w	0	x
cost(i, G) =	0	w	x

	Alanine (Ala)		
S =	G	C	C
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	0	0
cost(i, G) =	0	w	x

	Alanine (Ala)		
S =	G	C	A
cost(i, A) =	w	w	0
cost(i, T) =	w	w	x
cost(i, C) =	w	0	x
cost(i, G) =	0	w	x

	Alanine (Ala)		
S =	G	C	G
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	0	x
cost(i, G) =	0	w	0

These four tables can be combined into one as follows:

	Alanine (Ala)		
S =	G	C	*
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	0	x
cost(i, G) =	0	w	x

A.1.14 Proline (Pro)

	Proline (Pro)		
S =	C	C	T
cost(i, A) =	w	w	x
cost(i, T) =	w	w	0
cost(i, C) =	0	0	x
cost(i, G) =	w	w	x

	Proline (Pro)		
S =	C	C	C
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	0	0	0
cost(i, G) =	w	w	x

	Proline (Pro)		
S =	C	C	A
cost(i, A) =	w	w	0
cost(i, T) =	w	w	x
cost(i, C) =	0	0	x
cost(i, G) =	w	w	x

	Proline (Pro)		
S =	C	C	G
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	0	0	x
cost(i, G) =	w	w	0

These four tables can be combined into one as follows:

	Proline (Pro)		
S =	C	C	*
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	0	0	x
cost(i, G) =	w	w	x

A.1.15 Threonine (Thr)

	Threonine (Thr)		
S =	A	C	T
cost(i, A) =	0	w	x
cost(i, T) =	w	w	0
cost(i, C) =	w	0	x
cost(i, G) =	w	w	x

	Threonine (Thr)		
S =	A	C	C
cost(i, A) =	0	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	0	0
cost(i, G) =	w	w	x

	Threonine (Thr)		
S =	A	C	A
cost(i, A) =	0	w	0
cost(i, T) =	w	w	x
cost(i, C) =	w	0	x
cost(i, G) =	w	w	x

	Threonine (Thr)		
S =	A	C	G
cost(i, A) =	0	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	0	x
cost(i, G) =	w	w	0

These four tables can be combined into one as follows:

	Threonine (Thr)		
S =	A	C	*
cost(i, A) =	0	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	0	x
cost(i, G) =	w	w	x

A.1.16 Valine (val)

	Valine (val)		
S =	G	T	T
cost(i, A) =	w	w	x
cost(i, T) =	w	0	0
cost(i, C) =	w	w	x
cost(i, G) =	0	w	x

	Valine (val)		
S =	G	T	C
cost(i, A) =	w	w	x
cost(i, T) =	w	0	x
cost(i, C) =	w	w	0
cost(i, G) =	0	w	x

	Valine (val)		
S =	G	T	A
cost(i, A) =	w	w	0
cost(i, T) =	w	0	x
cost(i, C) =	w	w	x
cost(i, G) =	0	w	x

	Valine (val)		
S =	G	T	G
cost(i, A) =	w	w	x
cost(i, T) =	w	0	x
cost(i, C) =	w	w	x
cost(i, G) =	0	w	0

These four tables can be combined into one as follows:

	Valine (val)		
S =	G	T	*
cost(i, A) =	w	w	x
cost(i, T) =	w	0	x
cost(i, C) =	w	w	x
cost(i, G) =	0	w	x

A.2 Non-Symmetric amino acids scoring scheme with 4 codons

There is one amino acid that is coded by 4 codons, but the symmetry is violated by 1st base with a case that produces a stop codon.

A.2.1 Glycine (Gly)

	Glycine (Gly)		
S =	G	G	T
cost(i, A) =	w	w	x
cost(i, T) =	w	w	0
cost(i, C) =	w	w	x
cost(i, G) =	0	0	x

	Glycine (Gly)		
S =	G	G	C
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	w	0
cost(i, G) =	0	0	x

	Glycine (Gly)		
S =	G	G	G
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	w	w	x
cost(i, G) =	0	0	0

	Glycine (Gly)		
S =	G	G	A
cost(i, A) =	w	w	0
cost(i, T) =	∞	w	x
cost(i, C) =	w	w	x
cost(i, G) =	0	0	x

In this case, if we replace the 1st base G with T it will produce a stop codon if the 3rd base is A, so we must take it under consideration.

A.3 Non-Symmetric amino acids scoring scheme with 6 codons

There are 3 amino acids coded by 6 codons, which cannot be summarized to a single table.

A.3.1 Arginine (Arg)

	Arginine (Arg)		
S =	C	G	T
cost(i, A) =	w	w	x
cost(i, T) =	w	w	0
cost(i, C) =	0	w	x
cost(i, G) =	w	0	x

	Arginine (Arg)		
S =	C	G	A
cost(i, A) =	x	w	0
cost(i, T) =	∞	w	x
cost(i, C) =	0	w	x
cost(i, G) =	w	0	x

	Arginine (Arg)		
S =	C	G	C
cost(i, A) =	w	w	x
cost(i, T) =	w	w	x
cost(i, C) =	0	w	0
cost(i, G) =	w	0	x

	Arginine (Arg)		
S =	C	G	G
cost(i, A) =	x	w	x
cost(i, T) =	w	w	x
cost(i, C) =	0	w	x
cost(i, G) =	w	0	0

	Arginine (Arg)		
S =	A	G	A
cost(i, A) =	0	w	0
cost(i, T) =	∞	w	w
cost(i, C) =	x	w	w
cost(i, G) =	w	0	x

	Arginine (Arg)		
S =	A	G	G
cost(i, A) =	0	w	x
cost(i, T) =	w	w	w
cost(i, C) =	x	w	w
cost(i, G) =	w	0	0

A.3.2 Serine (Ser)

	Serine (Ser)		
S =	T	C	T
cost(i, A) =	w	w	x
cost(i, T) =	0	w	0
cost(i, C) =	w	0	x
cost(i, G) =	w	w	x

	Serine (Ser)		
S =	T	C	C
cost(i, A) =	w	w	x
cost(i, T) =	0	w	x
cost(i, C) =	w	0	0
cost(i, G) =	w	w	x

	Serine (Ser)		
S =	T	C	A
cost(i, A) =	w	w	0
cost(i, T) =	0	w	x
cost(i, C) =	w	0	x
cost(i, G) =	w	∞	x

	Serine (Ser)		
S =	T	C	G
cost(i, A) =	w	w	x
cost(i, T) =	0	w	x
cost(i, C) =	w	0	x
cost(i, G) =	w	w	0

	Serine (Ser)		
S =	A	G	T
cost(i, A) =	0	w	w
cost(i, T) =	w	w	0
cost(i, C) =	w	w	x
cost(i, G) =	w	0	w

	Serine (Ser)		
S =	A	G	C
cost(i, A) =	0	w	w
cost(i, T) =	w	w	x
cost(i, C) =	w	w	0
cost(i, G) =	w	0	w

A.3.3 Leucine (Leu)

	Leucine (Leu)		
S =	C	T	T
cost(i, A) =	w	w	x
cost(i, T) =	w	0	0
cost(i, C) =	0	w	x
cost(i, G) =	w	w	x

	Leucine (Leu)		
S =	C	T	C
cost(i, A) =	w	w	x
cost(i, T) =	w	0	x
cost(i, C) =	0	w	0
cost(i, G) =	w	w	x

	Leucine (Leu)		
S =	C	T	A
cost(i, A) =	w	w	0
cost(i, T) =	x	0	x
cost(i, C) =	0	w	x
cost(i, G) =	w	w	x

	Leucine (Leu)		
S =	C	T	G
cost(i, A) =	w	w	x
cost(i, T) =	x	0	x
cost(i, C) =	0	w	x
cost(i, G) =	w	w	0

	Leucine (Leu)		
S =	T	T	A
cost(i, A) =	w	∞	w
cost(i, T) =	0	0	0
cost(i, C) =	x	w	x
cost(i, G) =	w	∞	w

	Leucine (Leu)		
S =	T	T	G
cost(i, A) =	w	∞	w
cost(i, T) =	0	0	x
cost(i, C) =	x	w	0
cost(i, G) =	w	w	w

A.4 Three stop codons

There are 3 stop codons which are not symmetric and cannot be summarized to any common table.

A.4.1 Stop

	Stop		
S =	T	A	A
cost(i, A) =	∞	0	0
cost(i, T) =	0	∞	∞
cost(i, C) =	∞	∞	∞
cost(i, G) =	∞	x	x

	Stop		
S =	T	A	G
cost(i, A) =	∞	0	x
cost(i, T) =	0	∞	∞
cost(i, C) =	∞	∞	∞
cost(i, G) =	∞	∞	0

	Stop		
S =	T	G	A
cost(i, A) =	∞	x	0
cost(i, T) =	0	∞	∞
cost(i, C) =	∞	∞	∞
cost(i, G) =	∞	0	∞

The issue is that these three codons cannot be combined into one table. For example, TAG can only have its third base replaced, while TGA allows substitutions only at the second base. In contrast, TAA permits changes at both the second and third positions because it shares features with both TAG and TGA.

Appendix B

Generating a Codon Usage File using Kazusa Database

To generate a codon usage file for the BioSynth app, users can extract codon usage frequencies from the Kazusa Codon Usage Database and convert the output using a provided local script.

Step 1: Extract Codon Usage Data

1. Visit the Kazusa Codon Usage Database: `https://www.kazusa.or.jp/codon/`
2. Under the **QUERY Box for search with Latin name of organism**, enter the name of the organism. For example: 'Saccharomyces cerevisiae'
3. Click the **Submit** button.
4. In the search results, locate the desired genome type (e.g., Saccharomyces cerevisiae) and click the corresponding link under the **Link** column.
5. The codon usage table will be displayed. Select a format (e.g., Standard), choose 'Codon Usage Table with Amino Acids', and click **Submit**. You should then see the following header:

   ```
   fields: [triplet] [amino acid] [fraction] [frequency: per thousand] ([number])
   ```

6. Select the entire codon usage table (not including the header), for example:

   ```
   UUU F 0.59 26.1 (170666)  UCU S 0.26 23.5 (153557)  UAU Y 0.56 18.8 (122728)  UGU C 0.63  8.1 ( 52903)
   UUC F 0.41 18.4 (120510)  UCC S 0.16 14.2 ( 92923)  UAC Y 0.44 14.8 ( 96596)  UGC C 0.37  4.8 ( 31095)
   UUA L 0.28 26.2 (170884)  UCA S 0.21 18.7 (122028)  UAA * 0.47  1.1 (  6913)  UGA * 0.30  0.7 (  4447)
   UUG L 0.29 27.2 (177573)  UCG S 0.10  8.6 ( 55951)  UAG * 0.23  0.5 (  3312)  UGG W 1.00 10.4 ( 67789)

   CUU L 0.13 12.3 ( 80076)  CCU P 0.31 13.5 ( 88263)  CAU H 0.64 13.6 ( 89007)  CGU R 0.14  6.4 ( 41791)
   CUC L 0.06  5.4 ( 35545)  CCC P 0.15  6.8 ( 44309)  CAC H 0.36  7.8 ( 50785)  CGC R 0.06  2.6 ( 16993)
   CUA L 0.14 13.4 ( 87619)  CCA P 0.42 18.3 (119641)  CAA Q 0.69 27.3 (178251)  CGA R 0.07  3.0 ( 19562)
   CUG L 0.11 10.5 ( 68494)  CCG P 0.12  5.3 ( 34597)  CAG Q 0.31 12.1 ( 79121)  CGG R 0.04  1.7 ( 11351)
   ```

```
AUU I 0.46 30.1 (196893)  ACU T 0.35 20.3 (132522)  AAU N 0.59 35.7 (233124)  AGU S 0.16 14.2 ( 92466)
AUC I 0.26 17.2 (112176)  ACC T 0.22 12.7 ( 83207)  AAC N 0.41 24.8 (162199)  AGC S 0.11  9.8 ( 63726)
AUA I 0.27 17.8 (116254)  ACA T 0.30 17.8 (116084)  AAA K 0.58 41.9 (273618)  AGA R 0.48 21.3 (139081)
AUG M 1.00 20.9 (136805)  ACG T 0.14  8.0 ( 52045)  AAG K 0.42 30.8 (201361)  AGG R 0.21  9.2 ( 60289)

GUU V 0.39 22.1 (144243)  GCU A 0.38 21.2 (138358)  GAU D 0.65 37.6 (245641)  GGU G 0.47 23.9 (156109)
GUC V 0.21 11.8 ( 76947)  GCC A 0.22 12.6 ( 82357)  GAC D 0.35 20.2 (132048)  GGC G 0.19  9.8 ( 63903)
GUA V 0.21 11.8 ( 76927)  GCA A 0.29 16.2 (105910)  GAA E 0.70 45.6 (297944)  GGA G 0.22 10.9 ( 71216)
GUG V 0.19 10.8 ( 70337)  GCG A 0.11  6.2 ( 40358)  GAG E 0.30 19.2 (125717)  GGG G 0.12  6.0 ( 39359)
```

7. Copy and paste it into a plain text file.
8. Save the file.

Step 2: Convert to BioSynth Format

1. Download the convert_kazusa_to_biosynth.py script from the BioSynth repository.
2. This script reads the codon usage file the you just created and outputs a two-column text file in the format required by the BioSynth app: each line should contain a codon followed by its usage frequency, separated by whitespace.
3. Run the script from the CLI:

```
cd /path/to/script_directory
python ./convert_kazusa_to_biosynth.py <codon_usage_file_path> -o <
    output_file>
```

4. The output file will consist of lines in the following form:

```
TAC 0.51
GCT 0.37
...
```

Note: The script convert_kazusa_to_biosynth.py is designed to convert codon usage tables provided by the Kazusa database. If you wish to get the table from another resource, please make sure to write your own converter script to ensure that you are in the right format.

Appendix C

Application Execution Examples

This appendix presents a series of case studies that demonstrate BioSynth's behavior under varying biological constraints. Three of the scenarios—no ORFs, a single ORF, and fully overlapping ORFs—were executed via the command line interface (CLI). The fourth scenario, involving multiple coding regions, was processed through the graphical user interface (GUI) to demonstrate user-controlled exclusion of specific ORFs. The accompanying reports illustrate how BioSynth removes unwanted patterns while preserving biological function where applicable.

C.1 Non-coding Regions

This case examines BioSynth's behavior when the input sequence contains an ORF that is too short to be considered a coding sequence. The target sequence is:

ACGTTGCCAGTCATGCGTACGCTTTAACCGAGTCCGTCGACGGTACCTCGGCTTCCA
GGTCG

An ORF exists between positions 10 and 24; however, because it is shorter than the required threshold of five internal codons, it is not considered as a coding region.

Position:	1	2	…	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	…
Sequence:	A	C	…	C	A	T	G	C	G	T	A	G	C	C	T	T	T	A	A	C	…
ORF:			…		A	T	G	C	G	T	A	G	C	C	T	T	T	A	A		…

In this example, we consider two unwanted patterns: ACG and GGT. The positional occurrence of these patterns within the target sequence is illustrated in Table C.1.1.

Position:	1	2	3	4	…	18	19	20	21	22	…	39	40	41	42	43	44	45	…	47	58	59	60
Sequence:	A	C	G	T	…	T	A	G	C	C	…	G	A	C	G	G	T	A	…	A	G	G	T
Pattern 1:	A	C	G				A	C	G				A	C	G								
Pattern 2:															G	G	T				G	G	T

Table C.1.1: The target sequence and associated unwanted pattern matches.

The ACG pattern occurs at positions 1–3, 19–21, and 40–42, while the GGT pattern occurs at positions 42–44 and 58–60. Notably, the third occurrence of ACG (positions 40–42) overlaps with the first occurrence of GGT (positions 42-44), which shares nucleotide at position 42. Because the sequence does not contain a valid coding sequence, the entire target sequence is treated as non-coding, and only the non-coding substitution rules apply (with costs α and β). BioSynth eliminates unwanted patterns using four substitutions, all of which are G→A transitions. Since transitions have lower costs, the algorithm selects transition substitution (G→A) instead of transversions. As the solution includes four transitions, the total cost is 4α. Notably, the substitution at position 42 simultaneously removes both the third occurrence of ACG and the first occurrence of GGT. The overlap of these patterns can be observed in Table C.1.1. The complete elimination report is presented on the following page.

Input

Target Sequence:

ACGTTGCCAGTCATGCGTACGCTTTAACCGAGTCCGTCGACGGTACCTCGGCTTCCAGGTCG

Unwanted Patterns:

GGT, ACG

Elimination

No ORFs were identified in the provided target sequence.

Elimination Process

Sequence optimization assumes the following cost parameters:

Non-Coding regions:

• Transition substitution in non-coding sites (A ↔ G, C ↔ T): α = 1.02

• Transversion substitution in non-coding sites ({A,G} ↔ {C,T}): β = 1.98

Coding regions:

• Non-synonymous substitution cost in coding region: w = 99.96

• Synonymous substitution costs determined by codon usage frequencies from: codon_usage_chloroplast.txt

Results

DNA Sequences Difference:

1	2	3	4	5	6	7	8	9
A	C	[G]	T	T	G	C	C	A
A	C	[A]	T	T	G	C	C	A

Detailed substitutions relative to the target sequence:

```
Position 3      G     ->   A       Cost: 1.02
Position 21     G     ->   A       Cost: 1.02
Position 42     G     ->   A       Cost: 1.02
Position 59     G     ->   A       Cost: 1.02
```

Optimized Sequence:

ACATTGCCAGTCATGCGTACACTTTAACCGAGTCCGTCGACAGTACCTCGGCTTCCAGATCG

C.2 Single Coding Region

In this scenario, the sequence contains one ORF which satisfies the minimum length threshold and is therefore treated as an active coding region. The target sequence is:

CGTACGTGACTACGATGAAACCCGTTGACCGCGTTCGCTGACCGGCTTGCCC

The active coding region extends from position 15 to 41, covering the sequence detailed below. Nucleotide triplets within this region are governed by coding-specific substitution rules, whereas substitutions outside this region adhere to non-coding cost parameters.

Position:	...	13	14	15	16	17	18	19	20	...	36	37	38	39	40	41	42	43	...
Sequence:	...	C	G	A	T	G	A	A	A	...	C	G	C	T	G	A	C	C	...
ORF:				A	T	G	A	A	A	...	C	G	C	T	G	A			

In this example, we consider five unwanted patterns: TGAC, CGAC, TAAC, TGGC and TGAT. The positional occurrence of these patterns within the target sequence is illustrated in Table C.2.1.

Position:	...	6	7	8	9	10	11	...	25	26	27	28	29	30	...	38	39	40	41	42	43	...
Sequence:	...	G	T	G	A	C	T	...	T	T	G	A	C	C	...	C	T	G	A	C	C	...
Pattern 1:			T	G	A	C		...		T	G	A	C		...		T	G	A	C		...

Table C.2.1: The target sequence and associated unwanted pattern matches.

The TGAC pattern appears at positions 7–10, 26–29, and 39-42, and it is the only unwanted pattern detected in the target sequence. Notably, the third instance (positions 39–42) contains the stop codon TGA of the coding region (positions 39–41), followed by an additional C base. Preserving this stop codon imposes strict constraints on the allowed substitutions for eliminating the pattern. In addition, although the patterns CGAC, TAAC, TGGC, and TGAT are not present in the current sequence, the algorithm must ensure that no substitutions generate new instances of these patterns in the optimized sequence. BioSynth eliminates unwanted patterns using four substitutions:

1. A transversion substitution G→T at position 8 - this eliminates the first occurrence of TGAC (positions 7–10). Trying a transition in any of the four positions would create one of the unwanted patterns (TGAT, TTAC, TGGC, or TGCC). Therefore, the minimal-cost solution is the higher-cost transversion substitution β.

2. A synonymous substitution GTT→GTA, both encoding the amino acid Valine, at positions 24-26 - this eliminates the second occurrence of TGAC (positions 26–29) while maintaining the coding constraints, without introducing new unwanted patterns. Therefore, the minimal-cost solution is synonymous substitution cost $\varepsilon(GTA)$.

3. A stop codon substitution TGA→TAA at positions 39-41 - this substitution preserves the stop codon function while eliminating TGAC at positions 39–42. Since this substitution converted one stop codon (TGA) to another stop codon (TAA), BioSynth treats this substitution as a synonymous substitution. Therefore, the minimal-cost solution is synonymous substitution cost $\varepsilon(TAA)$.

4. A transition substitution C→T at position 42 - this is necessary because substitution (3) removes the TGAC instance but creates a new TAAC instance. The C→T substitution eliminates TAAC without introducing a new unwanted pattern. Therefore, the minimal-cost solution is the low-cost transition substitution α.

Note that the third occurrence of TGAC requires two substitutions (substitutions 3+4 above) for complete elimination. A single transition at position 42 (C→T) would introduce another unwanted pattern TGAT. In addition, performing a transversion substitution at position 42 (e.g., C→A or C→G) would indeed remove any occurrence of an unwanted pattern but incurs a higher substitution cost than the two-step alternative. One possible option is the single substitution TGA→TAG. Using the codon_usage_chloroplast.txt codon usage table for "chloroplast Marchantia polymorpha", the frequency of TAG (0.06) corresponds to a cost of $-\log(0.06) = 1.22$. This is higher than the combined cost of the two-step substitution, TGA→TAA ($-\log(0.91) = 0.04$) followed by C→T (transition cost 1.02), which totals 1.06. BioSynth therefore selects the more economical two-step strategy, as it both guarantees elimination and avoids the higher cost transversion at position 42.

To verify this outcome, we tested with an alternative codon_usage_s_cerevisiae.txt codon usage table for "Saccharomyces cerevisiae". Under the revised frequencies, TGA→TAG became more favorable ($-\log(0.23) = 0.64$), making it cheaper than the two-step TGA→TAA + C→T pathway. In this case, BioSynth directly selected the single substitution.

This comparison highlights how codon usage preferences can directly influence the substitution selected for the optimal solution. The complete elimination reports are presented on the following pages, first using codon usage table for "chloroplast Marchantia polymorpha" and then using codon usage table for "Saccharomyces cerevisiae".

Input

Target Sequence:

CGTACGTGACTACGATGAAACCCGTTGACCGCGTTCGCTGACCGGCTTGCCC

Unwanted Patterns:

CGAC, TAAC, TGGC, TGAC, TGAT

Elimination

The following ORFs were identified in the target sequence:

CGTACGTGACTACGATGAAACCCGTTGACCGCGTTCGCTGACCGGCTTGCCC

A total of 1 ORFs have been identified:

1. ATGAAACCCGTTGACCGCGTTCGCTGA

All ORFs are assumed to be coding regions.

Elimination Process

Sequence optimization assumes the following cost parameters:

Non-Coding regions:

• Transition substitution in non-coding sites (A ↔ G, C ↔ T): α = 1.02

• Transversion substitution in non-coding sites ({A,G} ↔ {C,T}): β = 1.98

Coding regions:

• Non-synonymous substitution cost in coding region: w = 99.96

• Synonymous substitution costs determined by codon usage frequencies from: codon_usage_chloroplast.txt

Results

DNA Sequences Difference:

1	2	3	4	5	6	7	8
C	G	T	A	C	G	T	[G]
C	G	T	A	C	G	T	[T]

Detailed substitutions relative to the target sequence:

```
Position 8       G      ->   T        Cost: 1.98
Position 26      GTT    ->   GTA      Cost: 0.42
Position 41      TGA    ->   TAA      Cost: 0.04
Position 42      C      ->   T        Cost: 1.02
```

Optimized Sequence:

```
CGTACGTTACTACGATGAAACCCGTAGACCGCGTTCGCTAATCGGCTTGCCC
```

Input

Target Sequence:

CGTACGTGACTACGATGAAACCCGTTGACCGCGTTCGCTGACCGGCTTGCCC

Unwanted Patterns:

TAAC, TGAT, CGAC, TGGC, TGAC

Elimination

The following ORFs were identified in the target sequence:

CGTACGTGACTACGATGAAACCCGTTGACCGCGTTCGCTGACCGGCTTGCCC

A total of 1 ORFs have been identified:

1. ATGAAACCCGTTGACCGCGTTCGCTGA

All ORFs are assumed to be coding regions.

Elimination Process

Sequence optimization assumes the following cost parameters:

Non-Coding regions:

• Transition substitution in non-coding sites (A ↔ G, C ↔ T): α = 1.02

• Transversion substitution in non-coding sites ({A,G} ↔ {C,T}): β = 1.98

Coding regions:

• Non-synonymous substitution cost in coding region: w = 99.96

• Synonymous substitution costs determined by codon usage frequencies from: codon_usage_s_cerevisiae.txt

Results

DNA Sequences Difference:

1	2	3	4	5	6	7	8
C	G	T	A	C	G	T	G
C	G	T	A	C	G	T	G

Detailed substitutions relative to the target sequence:

```
Position 10      C      ->   A        Cost: 1.98
Position 26      GTT    ->   GTA      Cost: 0.68
Position 41      TGA    ->   TAG      Cost: 0.64
```

Optimized Sequence:

```
CGTACGTGAATACGATGAAACCCGTAGACCGCGTTCGCTAGCCGGCTTGCCC
```

C.3 Fully Overlapping ORFs

In this scenario, the input sequence contains fully overlapping ORFs. The target sequence is:

TAAACATACAGTAGATGAGTTACATACAGATGATTTACATACAGATGTAGTACA

Within this sequence, two ORFs are considered, as follows:

Position:	...	14	15	16	17	18	19	20	...	29	30	31	32	33	...	47	48	49	50	51	52	...
Sequence:	...	G	A	T	G	A	G	T	...	G	A	T	G	A	...	G	T	A	G	T	A	...
ORF 1:			A	T	G	A	G	T	...	G	A	T	G	A	...	G	T	A	G			...
ORF 2:											A	T	G	A	...	G	T	A	G			...

In this case, BioSynth considers only the longest ORF that fully contains the other ORFs, located at positions 15–50, as coding regions. Outside these coding regions, substitutions follow non-coding cost parameters.

In this example, we consider four unwanted patterns: CGA, TGA, TAA and TGG. The positional occurrence of these patterns within the target sequence is illustrated in Table C.3.1

Position:	1	2	3	4	...	14	15	16	17	18	19	20	...	29	30	31	32	33	34	35	...
Sequence:	T	A	A	A	...	G	A	T	G	A	G	T	...	G	A	T	G	A	T	T	...
Pattern 1:	T	A	A																		
Pattern 2:								T	G	A						T	G	A			

Table C.3.1: The target sequence and associated unwanted pattern matches.

The TAA pattern occurs at positions 1–3, while the TGA pattern occurs at 16-18 and 31-33. Although CGA and TGG patterns are not present in the current sequence, the algorithm must ensure that no substitutions generate new instances of these patterns in the optimized sequence. BioSynth eliminates unwanted patterns using three substitutions:

1. A transition substitution T→C at position 1 - this eliminates the only occurrence of TAA (positions 1–3). Therefore, the minimal-cost solution is the low-cost transition substitution α.

2. A synonymous substitution AGT→TCT, both encoding Serine, at positions 18-20 - this eliminates the first occurrence of TGA (positions 16-18) while maintaining the coding constraints, without introducing new unwanted patterns. Therefore, the minimal-cost solution is a synonymous substitution cost $\varepsilon(TCT)$.

3. A non-synonymous substitution ATG (Methionine)→ATC (Isoleucine) at positions 30-32 - this eliminates the second occurrence of TGA (positions 31-33). This instance of TGA overlaps two codons within the ORF (ATG and ATT). ATT encodes isoleucine, whereas ATG, although typically serves as a start codon, in this context it appears in the middle of the ORF and therefore functions solely as a standard codon encoding methionine. As a result, any substitution in ATG leads to a non-synonymous change, while in ATT, removing the

TGA pattern requires changing the first base, which likewise produces a non-synonymous substitution. Thus, the minimal-cost substitution that eliminates this unwanted pattern is a non-synonymous change requiring a single base change. Several solutions exist; in this case, the algorithm chose ATG $\rightarrow$ ATC (another possible option would be ATT $\rightarrow$ TTT), corresponding to a cost of $w + \delta_{edit}(\text{ATG}, \text{ATC}) = w + 1$.

The complete elimination report is presented on the following page.

Input

Target Sequence:

TAAACATACAGTAGATGAGTTACATACAGATGATTTACATACAGATGTAGTACA

Unwanted Patterns:

TGA, TGG, TAA, CGA

Elimination

The following ORFs were identified in the target sequence:

TAAACATACAGTAGATGAGTTACATACAGATGATTTACATACAGATGTAGTACA

A total of 1 ORFs have been identified:

1. ATGAGTTACATACAGATGATTTACATACAGATGTAG

All ORFs are assumed to be coding regions because BioSynth was executed using CLI.

Elimination Process

Sequence optimization assumes the following cost parameters:

Non-Coding regions:

• Transition substitution in non-coding sites (A ↔ G, C ↔ T): α = 1.02

• Transversion substitution in non-coding sites ({A,G} ↔ {C,T}): β = 1.98

Coding regions:

• Non-synonymous substitution cost in coding region: w = 99.96

• Synonymous substitution costs determined by codon usage frequencies from: codon_usage_chloroplast.txt

Results

DNA Sequences Difference:

1	2	3	4	5	6	7	8
[T]	A	A	A	C	A	T	A
[C]	A	A	A	C	A	T	A

Detailed substitutions relative to the target sequence:

```
Position 1       T      ->   C        Cost: 1.02
Position 20      AGT    ->   TCT      Cost: 0.40
Position 32      ATG    ->   ATC      Cost: 100.96
```

Optimized Sequence:

```
CAAACATACAGTAGATGTCTTACATACAGATCATTTACATACAGATGTAGTACA
```

C.4 Multiple Coding Regions (GUI)

This example illustrates an input sequence that contains multiple coding regions, some of which are open reading frames (ORFs) that are not active coding regions. Unlike the previous example—where all coding regions were valid—this case requires the user to exclude specific ORFs that are non-coding. Using the GUI, these ORFs can be deselected, ensuring that only the true coding regions are considered during the elimination process. The target sequence is

AAACCCATGAAACCCGTTGACTTAAGATAAAGGGATATATGCCCGGGTGTCTTCGAC
CCTGAATATGCGGACCTTCAAATATTGAAGCGTAACTCGGCCATGAAAGGGTTCGAT
CATTGACGTAG

Four non-overlapping ORFs are identified within the sequence. ORF 1 spans positions 7 to 30, ORF 2 spans positions 39 to 62:

Position:	…	6	7	8	9	10	11	12	…	26	27	28	29	30	31	…	38	39	40	41	42	43	44	…	59	60	61	62	63	…
Sequence:	…	C	A	T	G	A	A	A	…	G	A	T	A	A	A	…	T	A	T	G	C	C	C	…	C	T	G	A	A	…
ORF 1:			A	T	G	A	A	A	…	G	A	T	A	A																
ORF 2:																		A	T	G	C	C	C	…	C	T	G	A		

ORF 3 spans positions 65 to 85, and ORF 4 spans positions 100 to 120:

Position:	…	64	65	66	67	68	…	82	83	84	85	86	…	99	100	…	…	…		120		…
Sequence:	…	T	A	T	G	C	…	T	T	G	A	A	…	C	A	T	G	A	…	T	T	G	A	C	…
ORF 3:			A	T	G	C	…	T	T	G	A													
ORF 4:															A	T	G	A	…	T	T	G	A	

In this case, the user chooses to exclude ORFs 1 and 4, reclassifying them as non-coding regions. BioSynth then applies codon-preserving substitutions within the designated coding regions, while treating the excluded ORFs according to non-coding cost parameters.

In this example, we consider one unwanted pattern: CGA. The positions within the target sequence are summarized in Table C.4.1.

Position:	…	53	54	55	56	57	…	78	79	80	81	82	83	…	110	111	112	113	114	…
Sequence:	…	T	C	G	A	C	…	A	A	T	A	T	T	…	T	C	G	A	T	…
Pattern:			C	G	A											C	G	A		

Table C.4.1: The target sequence and associated unwanted pattern matches.

The CGA pattern appears at positions 54–56 and 111–113. BioSynth eliminates unwanted pattern using two substitutions:

1. A synonymous substitution CGA→CGT, both encoding Arginine, at positions 54-56 - this eliminates the first occurrence of CGA (positions 54–56) while maintaining the coding constraints, without introducing new unwanted patterns. Therefore, the minimal-cost solution is a synonymous substitution cost $\varepsilon(CGT)$.

2. A transition substitution G→A at position 112 - this eliminates the second occurrence CGA (positions 111–113). Therefore, the minimal-cost solution is the low-cost transition substitution α.

To demonstrate the impact of excluding non-coding ORFs, we compare this case to a CLI execution that considers the full list of ORFs. In the CLI scenario, BioSynth considers all valid ORFs in the sequence as coding regions and applies codon-preserving substitutions within these ORFs. As a result, to eliminate the unwanted pattern CGA at positions 111–113, a synonymous substitution TTC→TTT is applied to positions 109-111, instead of the G→A transition applied to position 112 when ORF 4 was excluded (see above). The complete elimination reports for both executions are provided on the following pages, first for the GUI execution and then for the CLI execution.

Input

Target Sequence:

AAACCCATGAAACCCGTTGACTTAAGATAAAGGGATATATGCCCGGGTGTCTTCGACCCTGAATATGCGGACCTTCAAATATTGAAGCGTAACTCGGCCATGA
AAGGGTTCGATCATTGACGTAG

Unwanted Patterns:

CGA

Elimination

The following ORFs were identified in the target sequence:

AAACCCATGAAACCCGTTGACTTAAGATAAAGGGATATATGCCCGGGTGTCTTCGACCCTGAATATGCGGACCTTCAAATATTGAAGCGTAACTCGGCCATGA
AAGGGTTCGATCATTGACGTAG

A total of 4 ORFs have been identified:

1. ATGAAACCCGTTGACTTAAGATAA
2. ATGCCCGGGTGTCTTCGACCCTGA
3. ATGCGGACCTTCAAATATTGA
4. ATGAAAGGGTTCGATCATTGA

The specific ORFs that the user designated as non-coding are as follows:

1. ATGAAACCCGTTGACTTAAGATAA
4. ATGAAAGGGTTCGATCATTGA

These coding regions will be reclassified as non-coding areas.

The target sequence has been updated based on the selected coding regions:

AAACCCATGAAACCCGTTGACTTAAGATAAAGGGATATATGCCCGGGTGTCTTCGACCCTGAATATGCGGACCTTCAAATATTGAAGCGTAACTCGGCCATGA
AAGGGTTCGATCATTGACGTAG

Elimination Process

Sequence optimization assumes the following cost parameters:

Non-Coding regions:

• Transition substitution in non-coding sites (A ↔ G, C ↔ T): α = 1.02

• Transversion substitution in non-coding sites ({A,G} ↔ {C,T}): β = 1.98

Coding regions:

• Non-synonymous substitution cost in coding region: w = 99.96

• Synonymous substitution costs determined by codon usage frequencies from: codon_usage_chloroplast.txt

Results

DNA Sequences Difference:

1	2	3	4	5	6	7	8	9
A	A	A	C	C	C	A	T	G
A	A	A	C	C	C	A	T	G

Detailed substitutions relative to the target sequence:

```
Position 56      CGA    ->   CGT      Cost: 0.47
Position 112     G      ->   A        Cost: 1.02
```

Optimized Sequence:

AAACCCATGAAACCCGTTGACTTAAGATAAAGGGATATATGCCCGGGTGTCTTCGTCCCTGAATATGCGGACCTTCAAATATTGAAGCGTAACTCGGCCATGA
AAGGGTTCAATCATTGACGTAG

Input

Target Sequence:

AAACCCATGAAACCCGTTGACTTAAGATAAAGGGATATATGCCCGGGTGTCTTCGACCCTGAATATGCGGACCTTCAAATATTGAAGCGTAACTCGGC
CATGAAAGGGTTCGATCATTGACGTAG

Unwanted Patterns:

CGA

Elimination

The following ORFs were identified in the target sequence:

AAACCCATGAAACCCGTTGACTTAAGATAAAGGGATATATGCCCGGGTGTCTTCGACCCTGAATATGCGGACCTTCAAATATTGAAGCGTAACTCGGC
CATGAAAGGGTTCGATCATTGACGTAG

A total of 4 ORFs have been identified:

1. ATGAAACCCGTTGACTTAAGATAA
2. ATGCCCGGGTGTCTTCGACCCTGA
3. ATGCGGACCTTCAAATATTGA
4. ATGAAAGGGTTCGATCATTGA

All ORFs are assumed to be coding regions because BioSynth was executed using CLI.

Elimination Process

Sequence optimization assumes the following cost parameters:

Non-Coding regions:

• Transition substitution in non-coding sites (A ↔ G, C ↔ T): α = 1.02

• Transversion substitution in non-coding sites ({A,G} ↔ {C,T}): β = 1.98

Coding regions:

• Non-synonymous substitution cost in coding region: w = 99.96

• Synonymous substitution costs determined by codon usage frequencies from: codon_usage_chloroplast.txt

Results

DNA Sequences Difference:

1	2	3	4	5	6	7–9	10–12	13–15
A	A	A	C	C	C	ATG	AAA	CCC
A	A	A	C	C	78&	ATG	AAA	CCC

Detailed substitutions relative to the target sequence:

```
Position 56      CGA    ->   CGT      Cost: 0.47
Position 111     TTC    ->   TTT      Cost: 0.03
```

Optimized Sequence:

AAACCCATGAAACCCGTTGACTTAAGATAAAGGGATATATGCCCGGGTGTCTTCGTCCCTGAATATGCGGACCTTCAAATATTGAAGCGTAACTCGGC
CATGAAAGGGTTTGATCATTGACGTAG

תקציר

הפרויקט עוסק בבעיה של סילוק תבניות לא רצויות מרצף דנ"א, העלולות להוביל לקשירת חלבונים מזיקה, על ידי שימוש בסכמת עלות המייצגת פונקציה ביולוגית ושיטות אלגוריתמיות יעילות. בהסתמך על המסגרת התיאורטית שהוצגה בתזה של זהבית ליבוביץ', יישמתי מכונת מצבים סופית ואלגוריתם תכנות דינמי ליצירת רצף בעלות מינימלית, תוך שמירה על תפקוד החלבון והימנעות מתבניות לא רצויות. מודל העלות משמר את אורך החלבון המקודד, מטיל קנס גבוה על החלפות בסיסים שמשנות את חומצת האמינו המקודדת, וקנסות נמוכים יותר על החלפות שמשמרות את חומצת האמינו המקודדת - תוך לקיחה בחשבון את ההטיות בשימוש קודונים. שיפורים במכונת המצבים הסופית שהוצעה בתזה של זהבית מאפשרים מעקב מדויק אחר קודונים באמצעות זיכרון בעל שני בסיסים. הכלי BioSynth פותח עם ממשק משתמש גרפי וממשק שורת פקודה, ותומך באופטימיזציית קודונים תחת אילוצים ביולוגיים. תוצאות הניסויים מאמתות את דיוק סכמת העלות המותאמת לעומת טבלאות העלות המסורתיות, וממחישות את השימוש המעשי בכלי בתכנון דנ"א סינתטי.

עבודה זו בוצעה בהדרכתו של ד"ר אילן גרונאו מבי"ס אפי ארזי למדעי המחשב, אוניברסיטת רייכמן.

אוניברסיטת רייכמן
בית־ספר אפי ארזי למדעי המחשב
התכנית לתואר שני (M.Sc.)

פיתוח אפליקציית תוכנה גמישה ומקיפה לתכנון של רצפי דנ"א סינתטיים ללא תבניות בלתי רצויות

מאת
הדר פור

פרויקט גמר, מוגש כחלק מהדרישות לשם קבלת תואר מוסמך M.Sc., בית ספר אפי ארזי למדעי המחשב, אוניברסיטת רייכמן

ספטמבר 2025