Midterm Report

Project: DNA Sequence Analyzer and Visualizer

Yana Slavcheva, ID: 200231149

Student: _____ **, Date:** _____

signature

Supervisor: _____ **, Date:** _____

signature

Department of Computer Science, AUBG

Blagoevgrad, 2025

0. Table of contents

1. Introduction

1.1 Overview

This project focuses on the development of a C++ application for analyzing DNA sequences through an interactive graphical interface. The program allows users to load text-based FASTA files (see Section 2.4 for format specifications) with genetic information, search for specific nucleotide patterns, identify Open Reading Frames (ORFs, Section 2.3), and compare two sequences using an alignment algorithm.

1.2 Possible applications

This DNA analyzer and visualizer has direct applications in educational and research contexts. In a classroom setting, it can be used as an interactive teaching tool by allowing biology students to experiment with real DNA data and thus helping them understand concepts like gene structure and sequence alignment. For researchers, it can be a platform for preliminary genetic analysis, such as identifying potential genes in a newly sequenced genome or comparing genetic regions between different chromosomes or species.

This project could be seen as a basic, small, and simple desktop equivalent of other molecular biology tools with similar functionality. One such tool is SnapGene Viewer. Software like SnapGene Viewer provides biologists with an intuitive way to visualize plasmid and linear DNA sequences, identify key features like Open Reading Frames, and plan experiments. Commercial tools like SnapGene are very extensive, and this project aims to implement only a subset of their core features, specifically sequence visualization, ORF detection, and basic sequence analysis. In doing so, it demonstrates how computer science principles can be applied to address problems in bioinformatics.

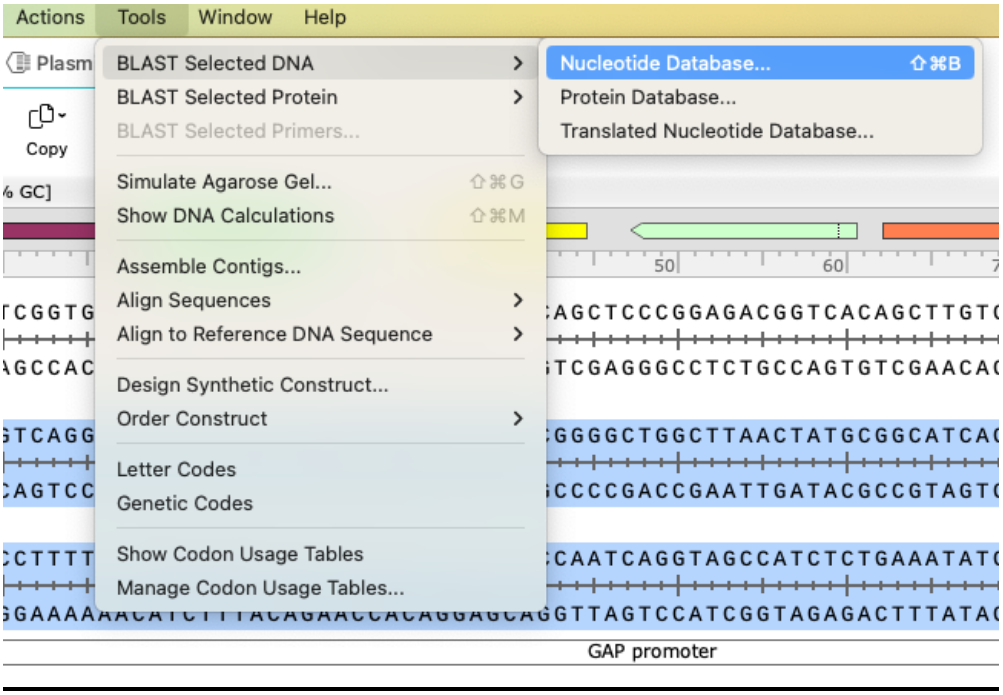

Figure 1. SnapGene Viewer interface showing multiple tools for sequence analysis.

1.3 Why this topic

Biology has been one of my main interests for about six years now. Before deciding to study Computer Science at AUBG, I had the opportunity to study Molecular Genetics at Sofia University. That showed me how much depth there is to this field and helped me develop a strong understanding of it. Even though I ultimately chose not to continue studying genetics, it has remained an area I'm very interested in and learn about in my spare time. That's why I thought there was no better topic for my senior project. It's a field I already know well, the project offers enough technical complexity, and most importantly, it can result in something useful. I hope that this project, as my introduction to bioinformatics, could perhaps guide me toward that direction in the future.

2. Domain Fundamentals

2.1 DNA structure

DNA (Deoxyribonucleic Acid) is the molecule that carries genetic instructions for the development and functioning of all organisms. Structurally, it is a double helix composed of two long strands. Each strand is made of simpler units called nucleotides. A nucleotide consists of one of four nitrogenous bases:

- Adenine (A),
- Thymine (T),
- Cytosine (C),
- and Guanine (G),

along with a sugar and a phosphate group. The two strands are connected by bonds between the complementary base pairs. A always pairs with T, and C always pairs with G. From a computational perspective, DNA can be represented as a single, very long string with the letters A, T, C, and G repeating in different order. However, not all parts of this long string are functionally equal.

2.2 Codons

The crucial function of DNA is to encode the instructions for building proteins but in fact, only about 1% of DNA codes for proteins. The code is read in units of three nucleotides called codons, where each codon specifies a single amino acid (a building block of a protein) or a “punctuation signal”. For example, the codon "ATG" codes for the amino acid methionine and also serves as the start codon, signaling the beginning of a protein-coding region. Conversely, the codons "TAA", "TAG", and "TGA" are stop codons that signal the end of that region. The sequence between a start codon and a stop codon forms a potential gene. See Figure 2 for visualization.

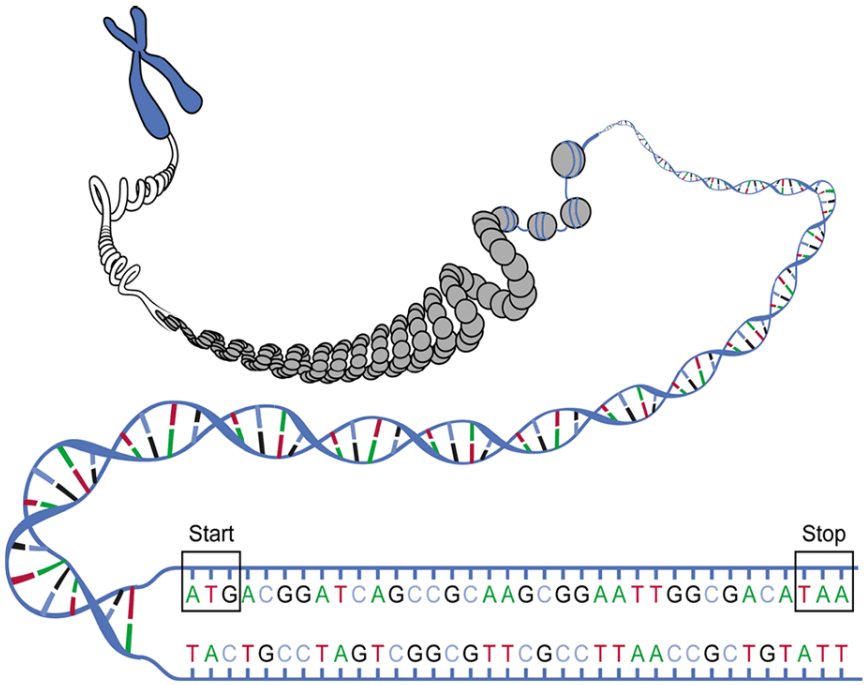

Figure 2. Schematic representation of DNA with a potential gene between a start and a stop codon.

2.3 Open Reading Frames (ORF)

An Open Reading Frame (ORF) is a part of DNA sequence that begins with a start codon and ends with a stop codon (TAA, TAG, or TGA), with a length that is a multiple of three. ORFs are considered potential genes, as they represent sequences that could be translated into a protein. Because DNA is double-stranded and each strand can be read in three different phases, there are six possible reading frames for any given DNA sequence, three on the forward strand and three on the reverse strand.

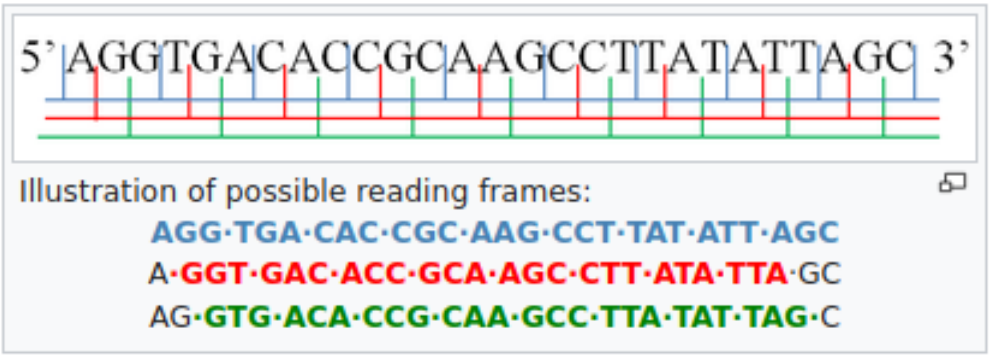

Figure 3. The three ways to read a single DNA strand.

Figure 3 illustrates how there are three possible reading frames on a single strand of DNA. By starting the grouping of nucleotides into triplets (codons) at different initial positions, three completely different sequences of amino-acid-coding units are generated. The reading frame is determined by the starting nucleotide:

- Frame 1 starts at nucleotide 1 (bases 1-2-3 form the first codon).
- Frame 2 starts at nucleotide 2 (bases 2-3-4 form the first codon).
- Frame 3 starts at nucleotide 3 (bases 3-4-5 form the first codon).

This process is repeated on the complementary (reverse) strand, yielding three more frames. Since only one of these six frames is typically used to encode a functional protein in a gene, a major goal in bioinformatics is to scan all six to identify all potential Open Reading Frames (ORFs). Identifying these ORFs is a core function of this project, as will be discussed later.

2.4 FASTA file format

The FASTA file format is the primary input for this project. The first step in processing involves parsing this file to extract the sequence data for analysis. FASTA is one of the most common text-based formats in bioinformatics because it presents nucleotide sequences in a simple and (as much as possible) human-readable way.

A FASTA file has a straightforward structure consisting of two parts. The first is the header line, which is the first line that always begins with a greater-than symbol (>). This line contains metadata about the sequence, such as its unique identifier, description, and source. The second part is the sequence data, which includes the actual biological sequence (A, T, C, and G). This sequence is usually long, often wrapped to 60 or 80 characters per line to improve readability.

You can see an example of a FASTA file in Figure 4:

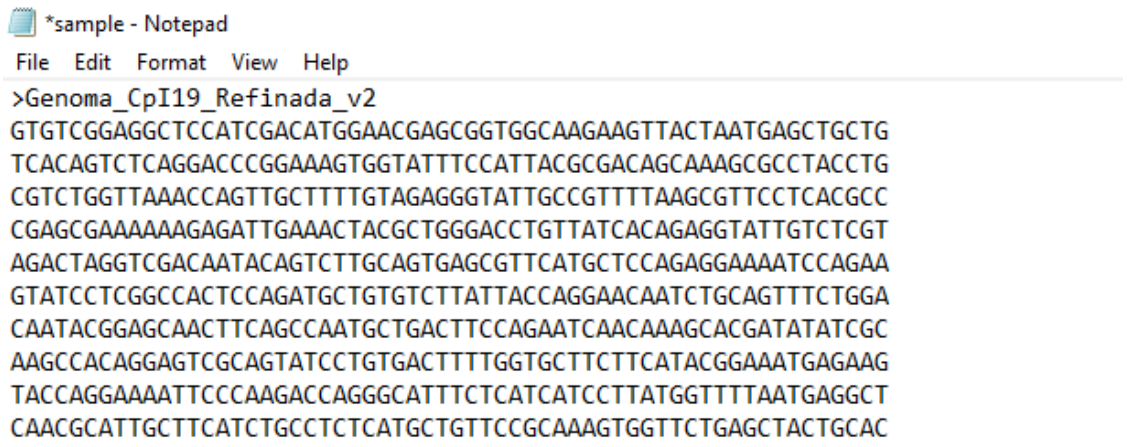

Figure 4. A FASTA file with a header and 10 lines of a DNA sequence.

3. Software Requirements

3.1 Functional requirements

The system should:

- Load and validate FASTA format files.
    - The program must check that there is a header line (first line starting with '>') and ensure the sequence string contains only valid nucleotide characters (A, T, C, G).

- Manage the sequence data in memory.
    - The loaded sequence(s) must be stored as a continuous string, and the user should be able to clear the current data or load a new file. This will enable analysis of different sequences.

- Display basic statistics about the loaded sequence.
    - After loading a file, the interface should show several key metrics like the length of the sequence and the nucleotide composition.

- Allow the user to search for custom nucleotide patterns.
    - The user can input a custom string (for example "ATGC") and the program will locate all instances and report their positions. This will be implemented using the Boyer-Moore string-searching algorithm which is efficient for searching on long sequences.

- Identify Open Reading Frames (ORFs).
    - The program must scan all six reading frames to find regions bracketed by start (ATG) and stop (TAA, TAG, TGA) codons. This is the primary method for predicting potential genes within the sequence.

- Perform pairwise sequence alignment.
    - The user can load two sequences and run a comparison. The program will show the aligned segments, display their similarities and differences, and calculate a similarity score using the Smith-Waterman algorithm for local alignment.

- Visualize the sequence and analysis results.
    - All results (ORFs, pattern matches, alignments) must be presented in an interactive graphical interface. A linear map will be used to represent the sequence, with visual annotations for different features, providing an intuitive overview of the analysis.

3.2 Non-functional requirements

- The application must be efficient.
    - For a standard chromosome-sized sequence (about 50 million base pairs), operations like ORF finding and pattern searching should complete within a few seconds.

- The program must have error handling.
    - The program must not crash in the case of invalid input. Instead, it should provide clear and user-friendly error messages for issues like invalid file paths, corrupt FASTA files, or invalid search patterns.

- The user interface must be intuitive.
    - The Qt-based GUI should be clear and easy to navigate. A user familiar with bioinformatics/genetics concepts should be able to perform all core tasks without prior instructions.

- The source code must be maintainable.
    - The code will be well-organized using the MVC pattern. The source code should be clearly written. Algorithms and other features need to be implemented in separate classes.

- Functionality must be verified through testing.
    - Unit tests will be implemented to verify critical components, such as FASTA file parsing and the output of the search and ORF-finding algorithms to ensure their correctness.

4. Program Architecture

4.1 MVC

The project follows the Model-View-Controller (MVC) architectural pattern. MVC is suitable because it isolates the analytical logic from the graphical interface, which makes the program easier to maintain and extend.

The Model component contains the core data and logic, including classes like FastaParser for file input, DNASequence for data storage, and SearchAlgorithm for analysis. It parses input files, manages the DNA sequence, and performs all the analytical operations, meaning it includes pattern searching, ORF finding, and sequence comparison.

The View forms the graphical interface, which will be developed using the Qt framework. It will include all visual elements and will be responsible for presenting data to the user and capturing their input.

The Controller processes user actions, invokes the appropriate methods within the Model, and updates the View with the resulting data. For example, when the user initiates a search, the Controller retrieves the query, executes the SearchAlgorithm in the Model, and sends the results to the View for display.

4.2 Class structure and polymorphism

At the center of the program is the DNASequence class. After a file is parsed, it stores the DNA sequence itself, the header information from the FASTA file, and a list of Feature objects that describe specific parts or findings within the sequence.

The Feature class is an abstract base class. It defines common properties for all sequence annotations, such as their start position, end position, and type. Because of this shared structure, different kinds of features can be processed in the same way and this is where polymorphism will be used. Classes such as Gene, SearchMatch, and RestrictionSite all inherit from Feature, but each represents a different kind of annotation. Gene objects mark predicted open reading frames (ORFs). SearchMatch objects record results from pattern searches. RestrictionSite objects mark enzyme cut locations.

File input is handled by the FastaParser class, which reads FASTA files, validates them, and then creates a DNASequence object with all its features.

Several other classes perform the main analysis tasks. SearchAlgorithm searches for specific nucleotide patterns using the Boyer–Moore method. TrieIndex manages k-mer indexing for faster lookup. OrfFinder identifies potential genes by scanning all reading frames. Aligner compares two sequences using the Smith–Waterman algorithm.

The graphical interface will consists of three main components: MainWindow managing the overall application, SequenceCanvas visually displaying the sequence and its features, and ResultsPanel, showing analysis results such as ORFs or search matches.

The project structure can be visualized in the following directory tree:

src/
- core/
  - DNASequence
  - Feature
- models/
  - Gene
  - SearchMatch
  - RestrictionSite
- io/
  - FastaParser
- algorithms/
  - SearchAlgorithm
  - TrieIndex
  - TrieNode
  - OrfFinder
  - Aligner
- gui/
  - *In progress*
- main.cpp
- tests/
- data/

5. Algorithms

5.1 String searching (Boyer-Moore)

Because DNA sequences can be very long (sometimes containing millions of base pairs) I needed an algorithm which will be able to handle searching through them quickly. That is why I chose the Boyer–Moore string-search algorithm, which is known for being highly efficient pattern-matching method that significantly outperforms naive search techniques. Its efficiency comes from the way it scans the text from right to left and uses preprocessed information about the pattern to skip sections of the sequence that cannot possibly contain a match. This skipping mechanism is based on two key rules - the “bad character” rule and the “good suffix” rule. In this implementation, the focus is on the bad character rule, because the nature of DNA's small alphabet makes the bad character rule particularly effective. Before the search begins, the algorithm preprocesses the pattern to create a bad character table, which stores the last occurrence of each character in the pattern. During the search, when a mismatch occurs, the algorithm checks this table to decide how far to shift the pattern. If the mismatched character does not appear in the pattern, the pattern can be shifted entirely past that position. If it does appear, the pattern is shifted so that the last occurrence of that character in the pattern aligns with its position in the text. This process is shown in the figure below:

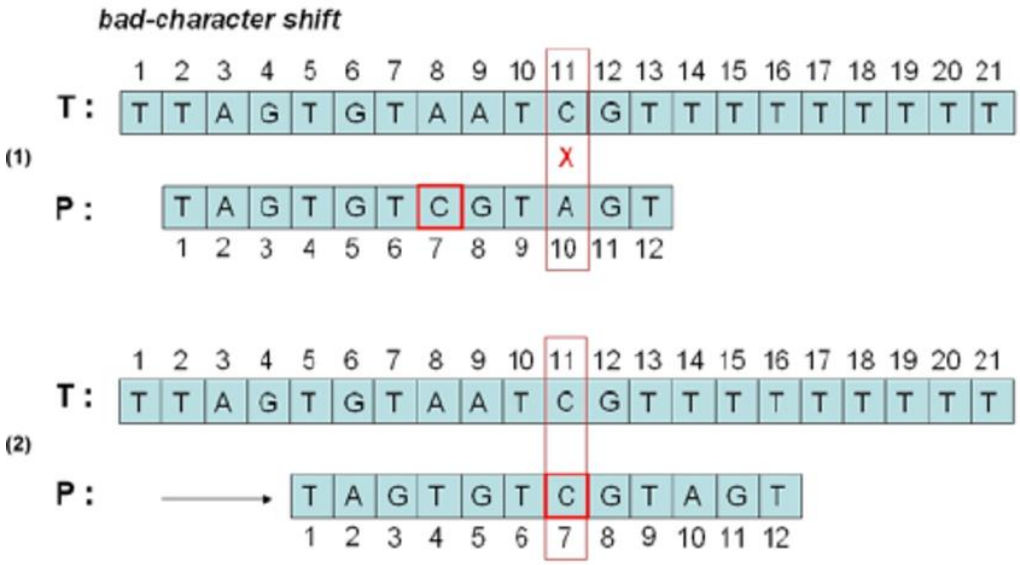

Figure 5. The bad character rule in the Boyer-Moore algorithm.

The algorithm can be explained by the following pseudo code:

```
BoyerMoore(text, pattern)
  build badCharTable(pattern)
  shift ← 0
  while shift ≤ (length(text) − length(pattern))
    j ← length(pattern) − 1
    while j ≥ 0 and pattern[j] = text[shift + j]
      j ← j − 1
    if j < 0 then
      record match at shift
      shift ← shift + pattern length
    else
      shift ← shift + max(1, j − badCharTable[text[shift + j]])
```

This technique allows the algorithm to skip many unnecessary comparisons. Because the pattern used in this project is typically a short codon (for example, “TAA”), the algorithm has limited opportunities to skip large portions of the sequence. As a

result, its performance in this context is O(n), but still highly efficient for scanning even very long DNA sequences.

5.2 Gene finding (Trie + ORF)

The gene-finding component of the application is a two-part system combining a trie data structure with a dedicated ORF finding algorithm. The first part, the trie, organizes all short substrings of the DNA sequence so that the program can instantly look up where any given codon or pattern occurs, making searches much faster than scanning the sequence base by base. A “k-mer” is a short substring of length k. During preprocessing, the algorithm inserts every possible k-mer from the DNA sequence (for example, every substring of length 3 to cover all codons) into the trie. Each node in the tree represents a nucleotide, and the paths from the root spell out the k-mers. Importantly, each node stores a list of all starting positions in the genome where its k-mer occurs. This will allow extremely fast lookups because once the trie is built, finding every instance of a specific codon like “ATG” becomes an O(m) operation, where m is the length of the pattern, since it only requires traversing the trie.

The second part, the ORF Finding Algorithm, uses this index to perform its core task. An Open Reading Frame (ORF) is a potential gene, defined as a sequence that starts with a start codon (“ATG”), ends with a stop codon (“TAA,” “TAG,” or “TGA”), and meets a minimum length requirement. The ORF finder operates as a state machine that scans the DNA across all six possible reading frames (three on the forward strand and three on the reverse). Instead of checking the sequence base by base, it uses the pre-built trie to instantly retrieve all start and stop codon positions. It then processes these lists, applying the biological rules like frame consistency and length thresholds to identify valid ORFs. The final output is a list of precise genomic coordinates for all predicted genes, which are then passed to the visualization module.

5.3 Sequence comparison (Smith-Waterman)

The Smith-Waterman algorithm is very commonly used in bioinformatics because it's an advanced method for finding the best local alignment between two DNA or protein sequences. Unlike simpler techniques that try to align entire sequences from start to finish, Smith-Waterman focuses on identifying the most similar regions within them, even if the rest of the sequences are unrelated. It uses a dynamic programming approach by building a scoring matrix in which each cell represents the best possible alignment score up to that point in the two sequences. The matrix is filled using straightforward rules:

- Matching nucleotides receive a positive score.
- Mismatches are given a penalty.
- Introducing a gap also incurs a penalty if it leads to a better score than continuing the alignment.
- If all possible scores at a given position are negative, the cell is reset to zero, effectively starting a new alignment.

These rules can be illustrated by the following pseudo code:

```
procedure SmithWaterman(A, B)
    create matrix H of size (len(A)+1) × (len(B)+1)
    initialize first row and column to 0
    for i from 1 to len(A)
        for j from 1 to len(B)
            match  ← H[i-1][j-1] + score(A[i], B[j])
            delete ← H[i-1][j] - gapPenalty
            insert ← H[i][j-1] - gapPenalty
            H[i][j] ← max(0, match, delete, insert)
    traceback from highest value in H to reconstruct alignment
```

Once the matrix is complete, the algorithm traces back from the highest-scoring cell to reconstruct the best local alignment, revealing the segments that are most alike. This is very valuable in bioinformatics because it can detect homologous regions - parts of DNA that come from a common evolutionary ancestor - even when they are surrounded by unrelated sequences.

s \ t	ϵ	C	T	C	A	A	T
ϵ	0	0	0	0	0	0	0
A	0	0	0	0	1	1	0
C	0	1	0	1	0	0	0
T	0	0	2	1	0	0	1
A	0	0	1	1	2	1	0
A	0	0	0	0	2	3	2
G	0	0	0	0	1	2	2

s \ t	ϵ	C	T	C	A	A	T
ϵ	·	·	·	·	·	·	·
A	·	·	·	·	↖	↖	←
C	·	↖	←	↖	↑	↑	↖
T	·	↑	↖	←	↖	·	↖
A	·	·	↑	↖	↖	↖	↑
A	·	·	↑	↑	↖	↖	←
G	·	·	·	·	↑	↑	↖

Figure 6. Smith-Waterman dynamic programming matrix.

For example, a researcher could use Smith-Waterman to compare a human gene with a mouse gene. While the surrounding DNA might differ greatly, the algorithm can locate and align a common gene which will provide clear evidence of shared function and evolutionary history. This ability to find similarities makes Smith-Waterman an extremely useful tool for studying genetic relationships.

The time complexity of this algorithm is O(m×n), where m and n are the lengths of the two sequences. Although this can be computationally demanding for very long sequences, the algorithm’s precision justifies the cost, as it guarantees an optimal local alignment rather than an approximate one.

6. Current progress

So far, I have successfully built the core of the program. The FASTA file parser is complete and can load real DNA sequences. It also includes error handling to manage problems like missing files or invalid data. Initially, I hadn't considered how much data a FASTA file could contain, but after discussions with my advisor and further research, I realized the scope needed adjustment. I found that a single human genome contains about 3.2 billion base pairs, with each base using approximately 1 byte of storage in FASTA file format. This means the raw sequence data for one full genome would be around 3.2 GB in size. I realized that such a large file would be too demanding for this project, as processing it would be very slow. Therefore, I decided to narrow the scope to focus on smaller sequences, such as individual genes or chromosomes. For example, chromosome 21 contains about 48 million base pairs (around 48 MB), which is far more manageable for the program's scale and performance goals.

After loading the FASTA file, the program uses the DNASequence class to store and verify the data. The method isValidDNA() checks that all characters in the sequence are valid nucleotides:

```
bool DNASequence::isValidDNA() const {
    return std::all_of(sequence.begin(), sequence.end(), [](char c) {
        return c == 'A' || c == 'T' || c == 'C' || c == 'G' || c == 'a'
|| c == 't' || c == 'c' || c == 'g';
            });
```

It ensures that only the four valid bases (A, T, C, and G) are present. Then the printSummary() function displays general information such as the header, sequence length, and validity:

```
void DNASequence::printSummary() const {
    cout << "DNA Sequence Summary" << endl;
    cout << "Header: " << header << endl;
    cout << "Length: " << length() << " bases" << endl;
    cout << "Valid DNA: " << (isValidDNA() ? "Yes" : "No") << endl;
```

At this stage, the application can take real genetic data and provide basic feedback about the sequence's composition and structure.

The first major algorithm, the Boyer-Moore search, is fully implemented and working. The algorithm allows the program to quickly find all occurrences of a given pattern in the loaded DNA sequence. It can automatically search for biological motifs like start and stop codons and display the results. It also allows the user to search for custom pattern of their choice. In case of invalid user input, the program displays an error message. For now, in the main() function pattern searches are tested through predefined common codons and biological motifs:

```
vector<string> testPatterns =
{"ATG","TAA","TAG","TGA","GGCC","ATAT","CG"};

  for (const auto& pattern : testPatterns) {
      if (pattern.length() <= seq.length()) {
          auto positions = SearchAlgorithm::boyerMooreSearch(seq,
pattern);
          SearchAlgorithm::printSearchResults(positions, pattern);
      }
```

The program can also display general information about the loaded DNA sequence like length, occurrences of each nitrogenous base, and more. Example code that counts nucleotide frequencies and calculates GC content in the main() function:

```
  int a_count = 0, t_count = 0, c_count = 0, g_count = 0;
  const string& seq = sequence->getSequence();

  for (char base : seq) {
      switch (toupper(base)) {
      case 'A': a_count++; break;
      case 'T': t_count++; break;
      case 'C': c_count++; break;
      case 'G': g_count++; break;
      }
  }

  double gc_content = (g_count + c_count) * 100.0 / seq.length();
```

I’m currently working on the gene finding feature. I expect it to be the most difficult to implement out of the three main functionalities in my program. This is because the program must scan all six possible reading frames of the DNA sequence (three on the forward strand and three on the reverse complement), correctly detect valid start and stop codons in each, and distinguish between overlapping ORFs in, potentially, a whole chromosome. The planned structure for trie-based indexing codons is similar to this:

```
struct TrieNode {
    bool isEnd;
    map<char, TrieNode*> children;
    vector<int> positions;
    TrieNode() : isEnd(false) {}
};
```

The next step is to implement the Smith-Waterman algorithm for sequence comparison. This will allow the program to align two sequences and identify regions of similarity, which is essential for comparing genes or chromosomes. Implementing this algorithm will require creating a dynamic programming matrix, managing its memory efficiently, and implementing a procedure to reconstruct the optimal alignment. The basic structure for computing the scoring matrix will be:

```
for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        int match = (A[i-1] == B[j-1]) ? score : -penalty;
        dp[i][j] = max({0, dp[i-1][j-1] + match, dp[i-1][j] - penalty, dp[i][j-1] - penalty});
    }
}
```

Once the algorithm is fully functional, I will integrate everything with the graphical interface using Qt. I chose Qt because it has its own dedicated IDE and Qt Creator, which will make it easier for me to build and design the user interface.

I also plan to develop a set of test cases to ensure that each feature, from parsing and pattern searching to ORF detection and alignment, works correctly.