Sequence Parser

Design and development - Updated periodically over the last week!

First thoughts:

The final product must be correct, runnable in an async environment and be scalable, along with being easily extensible for different formats of input, output, heuristics, et cetera.

Since the current problem is scaled down heavily and there’s a strict matching criterion (at least half of string matches), the implementation here is not very complex.

Project structure:

src

│   ├── main

│   │   ├── java

│   │   │   └── test

│   │   │   └── driver

│   │   │   ├── exceptions

│   │   │   │   └── IllegalInputFormatException.java

│   │   │   └── sequenceParser

│   │   │   ├── FastaReader.java (Reader for Fasta format files)

│   │   │   ├── Fragment.java (Data structure to store fragments)

│   │   │   ├── OverlappingFragments.java (Data structure to store output. Can be used to persist. Allows different format of inputs to be used without change in output)

│   │   │   ├── Reader.java (Interface which all types of readers can implement)

│   │   │   ├── SequenceParser.java (Main program, bulk of processing)

│   │   │   ├── SequencedFragments.java (Additional Data structure to handle processed data easily and combine results in O(n) time.

│   │   │   └── Sequencer.java (Thread-safe class to process data)

│   │   └── resources

│   │   ├── log4j.properties

│   │   └── project.properties

│   └── test

│   ├── java

│   │   └── test

│   │   └── driver

│   │   └── sequenceParser

│   │   ├── performanceTests

│   │   │   └── RunningTimeTest.java (Stress tests application)

│   │   └── unitTests

│   │   ├── FastReaderTest.java (Test the fasta reader)

│   │   └── SequenceParserTest.java (Mock reader and test processing)

│   └── resources

│   ├── Sample\_data.txt (Sample input from Annie’s email)

│   ├── coding\_challenge\_data\_set.txt (50 string input, provided as part of challenge)

│   └── test.properties

I have used the KMP (Knutt-Morris-Pratt) algorithm here to get the substring in O(n) time. This algorithm runs in O(n + m) time where n and m are the length of the 2 strings. Memoization with Dynamic programming can be applied too but that has O(n \* m) runtime complexity and O(n \* m) space complexity.

KMP algorithm works by using a backtracking matrix. I found the details on Wikipedia and implemented the algorithm in its original form. The algorithm can be modified to return the length of the matching substring.

Since the problem here has high degree of parallelization (Using Amdahl’s law), I chose to use multithreading to process each fragment and find out which fragment should follow it.

The output is stored in a Map whenever a match is found for a string. This is a concurrent map. After all threads have completed processing, the first fragment is taken and the reminder of the DNA sequence is stitched together.

To run this application, you can import it in IntelliJ or Eclipse and run the main function in SequenceParser.java class.

The location of the test file should be pasted in the project.properties file for the inputFilePath property.

This is a Maven project. Run the following command in the command line to run the tests.

$ mvn clean

$ mvn install

$ mvn test

I have used jUnit, log4j, Guice, Mockito to try to make the code production ready! While it may not be able to handle all the load and conditions it will be given in production, this is the baseline for release 1 and is ready to be shipped!

Also, I read further about DNA sequences and patching them together. Here’re some of my thoughts about real-world processing.

In this problem, we are considering at most 50 DNA sequences whose length doesn’t exceed 1000 characters. However, in a practical scenario when we are searching a database of size 109 – 1010 for the longest common overlapping subsequence along with missing terms in the sequences, we cannot use my solution for following reasons:

1. Implementing the dynamic programming algorithms in hardware is computation intensive. It improves the execution time but is still high cost for most researchers.

2. Distributing the sub-problems across multi-processors and integrating the results at the later step is also expensive.

To overcome the drawbacks, one of the approach I came across is FASTA algorithm, a heuristic method for string comparison that performs local alignment of sequences. Unlike dynamic programming, FASTA uses Smith-Waterman algorithm (D.P) over a narrow band of high scoring diagonals, which reduces the time to analyze and search entire database.

I wanted to implement on these heuristic algorithms too but ran out of time. However, I found those ideas very interesting.

Design patterns used: Chain of responsibility

I had marker interface in there for parsing too but decided to remove it as it wasn’t serving its purpose. If this was a webservice, having a marker interface would make more sense to allow only certain types of files as input.