Insights to FastA

Aly Shmahell
"aly.shmahell@gmail.com"
Department of Computer Science, University of L'Aquila

May 15, 2017

Abstract

Sequence Alignment is a sub-problem in bioinformatics that takes root in sequence processing from computer science, the basic idea from computer science is finding matches or mismatches or positions or patterns in any number of given sequences.
What is added in Sequence Alignment is the value derived from a certain result of sequence processing and its meaning in biology.
A certain match or difference in alignment between two characters in a sequence could have a different implication from another when it comes to sequences that represent DNA or Proteins.
What is important to know is that alignments have certain scores for each match or mismatch between two characters from each two sequences being aligned, when added up they compose the ultimate score for a certain alignment between the two sequences being aligned.
The first method used is the exhaustive method in which we perform a shift series pass on each of the two sequence being aligned then we calculate the score of that resulted alignment.
suffice to say the shift process alone is of $O(n^2)$ complexity for two sequences of the same length.
The second approach taken is using dot-plot and dynamic programming, that is dynamic programming over a matrix of size n^2 for two sequences of the same size. in this case we could easily end up with $O(n^2)$ complexity for two sequences that have no alignment at all.
The third approach is Heuristic Search, introducing probabilistic facts and devising methods to navigate quickly through our search by eliminating some search paths that would probably lead to no good alignment.

This is where FastA comes.

0.1 Introduction

FastA stands for Fast All, a software package of tools that can perform quick or optimized search on different databases, be it Nucleotides or Proteins.
The programs in the FASTA (pronounced FAST-Ay, for FAST-All) package have evolved from the original FASTP program, described by Lipman and Pearson (1985). FASTP combined a look-up table-based rapid sequence comparison program, SRCHN (Wilbur and Lipman, 1983), a rescanning phase that used the PAM250 similarity scoring matrix to identify ungapped alignments with high local similarity and a rapid Smith–Waterman alignment in a band around the best local alignment (Smith and Waterman, 1981); the band alignment allowed a limited number of gaps (16–32) to be inserted, and substantially improved alignment scores for distantly related sequences. [1]

0.2 Importance of Locality and Similarity

There are two points derived from biology that can help us speed up the alignments depending on locality/similarity:

- Two genes in different species may be similar over short conserved regions and dissimilar over remaining regions. [2] Mainly due to some regions having crucial manifestation for the survival of the organism, be it functional or structural.

- If your sequences are more than 100 amino acids long (or 100 nucleotides long) you can considered them as homologues if 25% of the aa are identical (70% of nucleotide for DNA). Below this value you enter the twilight zone. [3]
  Twilight zone = protein sequence similarity between 0-20% identity: is not statistically significant, i.e. could have arisen by chance. [3]

0.3 Mathematical Heuristics[2]

We know from biology why local similar regions are important in finding a good alignment, what we need to know if it's computationally feasible to develop approximate methods for alignment based on similarity, for that we need to get mathematical.
Paul Erdös-Alfréd Rényi developed a rule to find how much likely it is to get an element of a set, randomly over a series of fair tries:
First we examine the classical coin flip example, given a coin with two sides ("Head", "Tail"), it is 0.5 likely to get either in one flip, but for consecutive flips, it is likely to get a series of either as the following:

$$R = log_{1/p}(n)$$

where:

- R is the expected length of recurrence of either probability.
- p is the probability in a single hit.
- n the number of tries. (the length of the series of flips).

If we test this on a series of a 100 flips per coin we come to the probability of a recurrent side as follows:

$$R = log_{1/0.5}(100) = log_2(100) = 6.64$$

now for a 10000 flips:

$$R = log_2(10000) = 13.29$$

Now if we apply this on an alignment result between two sequences, and we exchange ("Head","Tail") for ("Match", "Mismatch") as the following:

AATCAT
ATTCAG
becomes:
HTHHHT

Then keeping in mind that for nucleotides we have 4 bases, that means the probability of occurrence for each in any one time is: $p = \frac{1}{4}$, therefor : $\frac{1}{p} = 4$, we arrive to the following likely length of matches for an alignment over a 10000 bases:

$$R = log_4(10000) = 6.64$$

But the thing is, in alignment we really have no idea if we are aligning with the correct shift, that means we can shift both sequences left or right, that extends the try space from n to $n * m$ considering n as the length of the first sequence and m the length of the second, which means for an alignment of two sequences each of which have 10000 bases, the length of recurrent matches is likely to be:

$$R = log_4(10000 * 10000) = 13.29$$

The implications of our little experiment with numbers are the following:

- For a sequence of 10000 bases, we can safely assume that for a good alignment to happen, it is likely to include words of length 13.
- In such alignment, it is much easier to do a search over a word size of 13 rather than 1.
- In such alignment, even if these words were not continuous, it is still helpful to build upon that number rather than go the exhaustive or dynamic route.

0.4 The Algorithm

The FastA Algorithm is composed of a few steps, while a bit tedious to code, are time saving for a researcher.
For the purpose of clear presentation, small python segments will accompany the steps.
First we need to understand the file structure that a FastA tool would have to deal with on a local machine:

- metadata : a file that contains the bio-alphabet the researcher is using, since a developer doesn't really know what type of sequence is being aligned, it is helpful to leave that to the bioengineer.
- sequences : the file containing the sequences to be aligned.
- database : the file containing already known sequences that comprise the database of previous experiments.
- scoreMatrix: a 2-D matrix that defines how biologically significant every match or mismatch is, and assigns a value to each and every one.

It goes without saying that step 0 is loading these files to memory:

```
scoreMatrix = []
database = []
sequences = []
alphabet = []
# using list mapping to reduce a file to a 1 dimentional array of alphabet
with open('metadata') as metadataFile:
  alphabet =[item for sublist in map(list,(value.rstrip('\n') for value in
    metadataFile)) for item in sublist]
# using line splitting and integer conversion to input the score matrix
with open('scoreMatrix') as scoreMatrixFile:
  scoreMatrix = [[int(value) for value in line.split()] for line in
    scoreMatrixFile]
# inputting the database as a terminal parameter with paying attention to
    new lines
with open(sys.argv[1],'r') as databaseFile:
  database = [[value for value in line.rstrip('\n')] for line in
    databaseFile]
# inputting the sequencesFile as a terminal parameter with paying attention
    to new lines
with open(sys.argv[2],'r') as sequencesFile:
  sequences = [[value for value in line.rstrip('\n')] for line in
    sequencesFile]
```

The first step is creating a hashtable that includes the position of every letter of our bio-alphabet for each of the two sequences being aligned at the moment.
it is important to note a hashtable is a lookup table that contains "keys to values", each key here is our nucleotide base letter, the values for each key are the positions where that letter occurs.

```
def fasta():
  for inputSequence in sequences:
    # populating seqWordTable
    seqWordTable = [[] for i in range(len(alphabet))]
    for nucSeqPos,nucSeq in enumerate(inputSequence):
      seqWordTable[alphabet.index(nucSeq)].append(nucSeqPos)
    # testing each database sequence against the input sequence at hand
    for databaseSequence in database:
      # populating dbWordTable
      dbWordTable = [[] for i in range(len(alphabet))]
      for nucDbPos,nucDb in enumerate(databaseSequence):
        dbWordTable[alphabet.index(nucDb)].append(nucDbPos)
```

The second step is calculating the difference in position between every two matching letters so that we can recreate our ("Match","Mismatch") experiment in code, this step is programatically defined as creating an OffSet Table.

```
# populating offset table
      offsetTable = [[] for i in range(len(alphabet))]
      for nucCount in range(4):
        for seqWordTableVal in seqWordTable[nucCount]:
          for dbWordTableVal in dbWordTable[nucCount]:
            if seqWordTableVal-dbWordTableVal >= 0:
              offsetTable[nucCount].append([seqWordTableVal-dbWordTableVal,[
    seqWordTableVal,dbWordTableVal]])
```

The third step is finding best regions of alignment, as in finding the offsets that are most reoccuring for each base letter, that means the longest continuous occurance of matches, and the longer the match the most significant the local alignment is, therefor it is both biologically significant and compute-time reducing in the next steps.

```
# finding the best regions of alignment
# counting the occurance of matches for each letter
offsetTableCount = count(offsetTable)
# taking only the longest streaks
offsetTableCount = offsetTableCount[:max(len(offsetTableCount), 10)]
# cleaning the hashtable from matches that were not significant
offsetTable = cleanList(offsetTable, [x[0] for x in offsetTableCount])
```

The fourth step is cleaning overlapping regions, taking into consideration that the longer ones are more important than the shorter.
Sometimes we find two shifts that can produce a good alignment, but one shift overlaps the other, in this case we only consider shifting according to the shift that gives us best score

```
# TODO : cleaning least scoring overlapping regions
```

The fifth step is connecting the regions (letter streaks resulted from non-overlapping shifts), by considering insertion, deletion or gaps with penalties, and scoring to see if this alignment is above a threshold.
Connecting the regions is most of the times done using the Smith–Waterman algorithm if precision is required, and it is important to note we already did the heavy lifting in the previous steps, therefor the Smith–Waterman algorithm should not be as time consuming as when it is done without FastA.

```
# TODO : connecting the regions & rescoring
```

0.5 Code Repository

A Github repository exists where I upload the full source code to my projects, it includes my (not as of this day ready) FastA implementation:

`https://github.com/AlyShmahell/AlyShmahell-BioInformatics`

Bibliography

[1] William R Pearson, http://onlinelibrary.wiley.com/doi/10.1038/npg.els.0005255/abstract , 2005.

[2] Dan Graur Ph.D, http://nsmn1.uh.edu/dgraur/BioInfo2011/Presentations/Class%204/8Bioinformaticsblast_fasta.ppt , 2011.

[3] Lorenza Bordoli, http://www.ch.embnet.org/CoursEMBnet/Basel03/slides/BLAST_FASTA.pdf , 2003.