BCH2203 Python - 8. Automating Alignment with BLAST

Ramses van Zon

March 6, 2024

What is alignment?

Alignment is a matching technique to find similar sequences. It allows that allows for imperfections like substitutions omissions and gaps.

Typically, we match shorter sequences ("queries") to a longer, target sequence, or to a search set or database of target sequences.

We can do this for nucleotide as well as protein sequences.

One distinguishes global alignment: matching everything, and local alignment: matching substrings.

```
5' ACTACTAGATTACTTACGGATCAGGTACTTTAGAGGCTTGCAACCA 3'
             |||| |||||| ||||||||||||||
          5' TACTCACGGATGAGGTACTTTAGAGGC 3'
```

What is alignment used for?

Alignment can, for instance, be useful for:

- Identifying sequences from your lab with known reference genomes.
- Finding genes from one organism in the DNA of other species.
- Finding conserved sequences and motifs across species.
- Finding evolutionary relationships.

Why not use regexes?

In a previous lecture, we saw how to use regexes for pattern matching.

These do not work well for local alignment, because:

- Expressing that gaps are allowed anywhere will match almost anything.
- Regexes do not have any notion of how well the pattern matches.
- Which makes it hard to put in biological aspects (e.g., substituting T for C is not as bad as A for C, longer gaps are more likely than a lot of small gaps, etc.)
- Regexes look at all possible matches, which is inefficient.

Alignment in a Nutshell

Given some target sequence, such as `CTGA...AGTTAGTGG...`

And some query sequence, such as `AGTTCCCTG`

One possible alignment is:

```
CTGA...AGTTAG--TGG...
       ||||    ||
       AGTT*CCCTG
```

Note that gaps and substitutions are allowed.

The alignment depends on the scoring criteria.

Scoring criteria can be rather sophisticated.

Alignment in a Nutshell, continued

- The so-called **Smith-Waterman** algorithm lies at the heart of most local alignment algorithms, and is very efficient (much better than brute force trying).
- The algorithm assigns a score.

  Roughly speaking, the fewer matches, the lower the score.
- Instead of raw scores, alignment tools report the likelihood that the match is due to chance, the e-value.
- We are almost always interested in the Highest Scoring Segment pairs (HSPs), i.e., the ones with the lower e-values.

Optimized but Heuristic Alignment

- Although gaps are allowed, HSPs will have substantial stretches of exact matches.
- This realization is the basis of **more efficient** algorithms.
- Such algorithms first search for short exact matches of length k, then do alignment around those spots.
- The search for short exact matches is done using a hash storing locations of all short sequences of length k in the query (Some aligners also hash k-mers in the target sequence).
- This is heuristic approach; it cannot guarantee that the best match is found.
- The size of the exact match is a parameter.
  ($k \approx 11$ for DNA, $k \approx 4$ for proteins)

Basic Local Alignment Search Tool

A suite of programs that allow finding alignments in a series of sequences using this heuristic approach.

Advantages:

- No need to code your own aligner.
- Fast algorithm.
- Implemented in C++: fast!
- Can run over the web.
- Can also run on own computer or a supercomputer.
- Evolving versions:
  BLAST+ > BLAST2 > BLAST1

BLAST®

https://blast.ncbi.nlm.nih.gov/Blast.cgi

Ways to perform BLAST queries

Run online though browser

blast.ncbi.nlm.nih.gov/Blast.cgi

Pros:

- Convenient, point and click

Cons:

- Not scriptable, cannot be automated, and so not easily reproducible.
- Subject to internet connectivity
- Speed varies
- Cannot use your own sequence database

Run online though Biopython

Pros:

- Using the `Bio.Blast.NCBIWWW` submodule, we can do the same queries from within Python
- Scriptable, automatable.

Cons:

- Subject to internet connectivity
- Speed varies
- Cannot use your own sequence database

Run locally using local targets

Pros:

- Using the `Bio.Blast.Applications` submodule, we can do queries locally from within Python
- Scriptable, automatable.
- Run without internet
- Less speed variation
- Use your own target sequences

Cons:

- Must install BLAST
- Need to download database

What you need to run locally or on a supercomputer

1. Have BLAST+ installed
2. Download reference databases to use as target
   (unless you have your own target database, of course)
3. Use Biopython to use it from within Python.

1. Getting BLAST+

You need to have BLAST+ installed even if you are going to work with BLAST through Biopython.

On a cluster

If you're on a supercomputer like the Teach cluster or Niagara, there's likely a module for it, which you need to load on the shell command line, before launching Python:

```
$ module load  gcc gmp lmdb boost blast+
```

On your own computer

For local installations, go to *ftp.ncbi.nlm.nih.gov*

Select the version appropriate for your OS and install it.

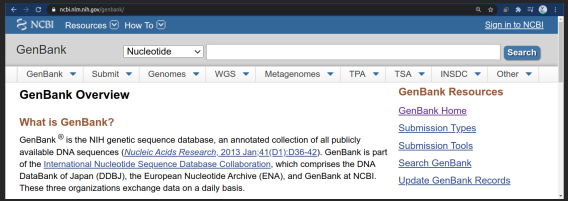
or your OS's package manager may have it (e.g. `sudo apt install ncbi-blast+` on Ubuntu).

2. Getting the Reference Database

You can align any sequence to any other, but mostly, people align against one or more known reference genomes.

We already saw that Genbank collects, stores and distributes such data.

We can use Biopython to download sequences.



Access Entrez from Python

- Use Bio.Entrez

```
>>> from Bio import Entrez
```

- Let Genbank know your email (yes, they insist)

```
>>> Entrez.email = "rzon@scinet.utoronto.ca"
```

- Start a search, e.g. for *E. coli* DNA sequences, with at most 13 results:

```
>>> handle = Entrez.esearch(db="nucleotide", term="E. coli", retmax=13)
```

- Read the result:

```
>>> output = handle.read()
>>> print(output)
b'<?xml version="1.0" encoding="UTF-8" ?>\n<!DOCTYPE eSearchResult PUBLIC "-//NLM//DTD esearch 2006 0628//EN"
"https://eutils.ncbi.nlm.nih.gov/eutils/dtd/20060628/esearch.dtd">\n<eSearchResult><Count>17572373</Count>
<RetMax>13</RetMax><RetStart>0</RetStart><IdList>\n<Id>1906573107</Id>\n<Id>1906520706</Id>\n<Id>1906485224
</Id>\n<Id>1906425573</Id>\n<Id>1906410586</Id>\n<Id>1906400389</Id>\n<Id>1906377346</Id>\n<Id>1906368402</Id>
\n<Id>1906361046</Id>\n<Id>1906353597</Id>\n<Id>1906347957</Id>\n<Id>1906327159</Id>\n<Id>1906325706</Id>\n
</IdList><TranslationSet><Translation>     <From>E. coli</From>     <To>"Escherichia coli"[Organism] OR E.
coli[All Fields]</To>    </Translation></TranslationSet><TranslationStack>   <TermSet>    <Term>"Escherichia
coli"[Organism]</Term>    <Field>Organism</Field>    <Count>8450531</Count>    <Explode>Y</Explode>
</TermSet>   <TermSet>    <Term>E. coli[All Fields]</Term>    <Field>All Fields</Field>    <Count>10324226
</Count>    <Explode>N</Explode>   </TermSet>   <OP>OR</OP>   <OP>GROUP</OP>  </TranslationStack>
<QueryTranslation>"Escherichia coli"[Organism] OR E. coli[All Fields]</QueryTranslation></eSearchResult>\n'
```

Let's unpack this "mess"

- We get a string back from Entrez, but it's prepended with "b".
- That indicates that this is a bytes string, not a character string.
- Bytes strings can be converted to character strings with .decode()

```
>>> output = output.decode()
>>> print(output)
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE eSearchResult PUBLIC "-//NLM//DTD esearch 20060628//EN" "https://eutils.ncbi.nlm.nih.gov/eutils/dtd/20060
<eSearchResult><Count>17572373</Count><RetMax>13</RetMax><RetStart>0</RetStart><IdList>
<Id>1906573107</Id>
<Id>1906520706</Id>
<Id>1906485224</Id>
<Id>1906425573</Id>
<Id>1906410586</Id>
<Id>1906400389</Id>
<Id>1906377346</Id>
<Id>1906368402</Id>
<Id>1906361046</Id>
<Id>1906353597</Id>
<Id>1906347957</Id>
<Id>1906327159</Id>
<Id>1906325706</Id>
</IdList><TranslationSet><Translation>     <From>E. coli</From>     <To>"Escherichia coli"[Organism] OR E. coli[All
```

That's XML!

- Entrez returns search results as XML.
- Python has a built-in module called `xml` to deal with that.
- `xml` can either read xml from a file or from a string.
- It has a few sub modules:

  xml.etree.ElementTree, xml.dom, xml.dom.minidom, xml.dom.pulldom, xml.sax, xml.parsers.expat
- We really only need the basic submodule, etree

```
>>> import xml.etree.ElementTree as ET
>>> searchtree = ET.ElementTree(ET.fromstring(output))
>>> print(searchtree)
<xml.etree.ElementTree.ElementTree object at 0x2ac8d2ae4d68>
```

- This has read it as a tree for use by Python, it's no longer plain text.

Or: Write a separate script to get reference sequences

- The data is large (hundreds of GB), and is better downloaded in compressed form, which additional, possibly *external tools* to uncompress. You only want to do this once.
- The data needs to be pre-processed to create the index. This is done by one of the BLAST utilities. You only want to do this once.
- But you do want to know what you did to download it: a script.

Getting Reference Nucleotide Database (Linux/Unix/MacOS)

`nt` is the "Non-redundant nucleoTide" database, which you'd likely use from DNA alignment.

(for protein alignment, use the `nr` database).

Get it as one big fasta file (~300GB)

```
$ curl ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/nt.gz -O
$ gunzip nt.gz
```

For BLAST to work with the database, it needs to create an index to the reference sequence.

```
$ module load gcc gmp lmdb boost blast+            # on SciNet clusters
$ makeblastdb -in nt -out nt -dbtype nucl
#and just wait a few days...
```

Using pre-processed data that comes with its index

Instead, download the pre-indexed database in 28 parts:

```
$ cd $SCRATCH/db  # or some other folder
$ wget ftp://ftp.ncbi.nlm.nih.gov/blast/db/nt.??.tar.gz
$ gunzip nt.??.tar.gz
$ for n in nt.??.tar.gz ; do tar xvf nt.$n.tar ; done
```

Commands of BLAST+

BLAST+ has a number of commands.

Biopython will call these for you, but it's good to know what they are and what they do.

Program	Query type	Database/reference type
blastn	nucleotides	nucleotides
blastp	protein	protein
blastx	nucleotides	protein
tblastn	protein	nucleotides
tblastx	nucleotide	nucleotides
psiblast	protein	protein
deltablast	protein	protein

BLAST needs to know where to look for the database. Specify this with the BLASTDB shell environment variable.

```
$ export BLASTDB=$SCRATCH/db  # the folder with the database
```

Example of running BLAST locally

Suppose we have a dna fragment, `dna.fasta`, that we want to align it to all nucleic sequences in the `nt` database.

On the teach cluster:

```
$ module load gcc gmp lmdb boost blast+
$ export BLASTDB="$SCRATCH/db/nt"
$ blastn -query dna.fasta -db nt -out dna.xml -evalue 0.001 -outfmt 5
BLASTN 2.10.1+, build Sep 28 2020 11:01:31

Reference: Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb
Miller (2000), "A greedy algorithm for aligning DNA sequences", J
Comput Biol 2000; 7(1-2):203-14.
...
```

This `blastn` command takes a bit over a minute and writes text to screen.

That is hard to parse and use in a pipeline.

Instead, use the following `blastn` command to produce an **XML file** with the matches.

```
$ blastn -query dna.fasta -db nt -out dna.xml -evalue 0.001 -outfmt 5
```

Other output formats are possible as well.

Enter Biopython

We could read now in and process the XML in Python.

But we want to avoid making our pipeline too complex, with shell scripts for blast and python scripts for processing . . .

For simplicity and sanity, try to stay within the same programming environment as much as possible.

Although BLAST must run as an external program, Biopython contains a wrapper for it, so we can do it in Python:

```
>>> from Bio.Blast.Applications import NcbiblastnCommandline
>>> blastn_cmdline = NcbiblastnCommandline(query="dna.fasta", db="nt", evalue=0.001, outfmt=5, out="dna.xml")
>>> print(blastn_cmdline)
blastn -out dna.xml -outfmt 5 -query dna.fasta -db nt -evalue 0.001
>>> stdout, stderr = blastn_cmdline()
```

This produces the same XML file with the matches.

This really is just a wrapper around BLAST, remember you must have BLAST+ installed.

Biopython offers a parser specific for the BLAST output which reads an output file into a data structure.

Parsing the BLAST output with Biopython

```
>>> from Bio.Blast import NCBIXML
>>> filehandle = open("dna.xml")
>>> blast_record = NCBIXML.read(filehandle)
>>> filehandle.close()
```

or, if you have lots of results
(i.e., multiple query sequences):

```
>>> blast_records = NCBIXML.parse(result_handle)
```

```
<?xml version="1.0"?>
<!DOCTYPE BlastOutput PUBLIC "-//NCBI//NCBI BlastOutput/EN"
<BlastOutput>
  <BlastOutput_program>blastn</BlastOutput_program>
  <BlastOutput_version>BLASTN 2.9.0+</BlastOutput_version>
  <BlastOutput_reference>Zheng Zhang, Scott Schwartz, Lukas
  <BlastOutput_db>nt</BlastOutput_db>
  <BlastOutput_query-ID>Query_1</BlastOutput_query-ID>
  <BlastOutput_query-def>NC_000019.10|55232502-55232646 NC_0
  <BlastOutput_query-len>144</BlastOutput_query-len>
  <BlastOutput_param>
    <Parameters>
      <Parameters_expect>0.001</Parameters_expect>
      <Parameters_sc-match>1</Parameters_sc-match>
      <Parameters_sc-mismatch>-2</Parameters_sc-mismatch>
      <Parameters_gap-open>0</Parameters_gap-open>
      <Parameters_gap-extend>0</Parameters_gap-extend>
      <Parameters_filter>L;m;</Parameters_filter>
    </Parameters>
  </BlastOutput_param>
<BlastOutput_iterations>
<Iteration>
  <Iteration_iter-num>1</Iteration_iter-num>
  <Iteration_query-ID>Query_1</Iteration_query-ID>
  <Iteration_query-def>NC_000019.10|55232502-55232646 NC_000
  <Iteration_query-len>144</Iteration_query-len>
  <Iteration_hits>
```

Getting a summary

```
>>> def blast_summary(blastrecordalignment, evaluethresh):
...     for alignment in blastrecordalignment:
...       for hsp in alignment.hsps:
...           if hsp.expect < evaluethresh:
...               print("Alignment")
...               print("sequence:", alignment.title)
...               print("length:", alignment.length)
...               print("e value:", hsp.expect)
...               print(hsp.query[0:75] + "")
...               print(hsp.match[0:75] + "")
...               print(hsp.sbjct[0:75] + "")
...
>>> blast_summary(blast_record.alignments, 0.04)
```

```
Alignment
sequence: gi|1549097453|gb|CP034522.1| Eukaryotic synthetic
length: 64242768
e value: 1.47947e-67
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACC
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACC
Alignment
sequence: gi|1548994293|gb|CP034497.1| Eukaryotic synthetic
length: 64242768
e value: 1.47947e-67
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACC
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACC
Alignment
sequence: gi|22038498|gb|AC010327.8| Homo sapiens chromosome
length: 146664
e value: 1.47947e-67
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACC
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACC
Alignment
sequence: gi|20087139|gb|AC116342.2| Homo sapiens chromosome
length: 39450
e value: 1.47947e-67
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACC
```

For completeness: Web-based BLAST Example using Biopython

```
>>> from Bio import SeqIO
>>> from Bio.Blast import NCBIWWW, NCBIXML
>>> seqrec = SeqIO.read("dna.fasta", "fasta")
>>> result_handle = NCBIWWW.qblast("blastn", "nt", seqrec.seq)
>>> blast_record = NCBIXML.read(result_handle)
>>> blast_summary(blast_record.alignments, evaluethresh=0.04)
Alignment
sequence: gi|1549097453|gb|CP034522.1| Eukaryotic synthetic construct chromosome 19
length: 64242768
e value: 1.47947e-67
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACCTCACACGGTCACCAC...
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||...
ATGGCAACCAGAGCAGACAGACCCACTCACCAGCTCTCAAGCCACCTCCCAGCCGTCACCTCACACGGTCACCAC...
Alignment
sequence: gi|1548994293|gb|CP034497.1| Eukaryotic synthetic construct chromosome 19
...
```

Conclusions

- For alignment, use existing, fast tools like BLAST
- Biopython can interface with BLAST, making it easier to build a pipeline.
- Reference data can be big: download once and reuse.