1 Introduction to BLAST

1.1 Introduction

Basic **L**ocal **A**lignment **S**earch **T**ool (**BLAST**) is a powerful tool for comparing and identifying sequences which share similarity. This can be useful for several reasons:

- Identifying an unknown sequence by finding annotated (or known) sequences which are similar
- Finding similar sequences in other species (e.g. orthologs)
- Predicting function by identifying similar regions in other sequences which already have a known function

In this tutorial, we are going to use a version of BLAST called **BLAST+**.

BLAST+ is split into different applications which are based on the type of sequence provided by you, the user, as well as the type of sequences in the database being searched. There are three things you will need each time you want to run a BLAST search:

- **A query sequence** (*can be nucleotide or protein*)
- **A sequence database** (*can be nucleotide or protein*)
- **A BLAST application** (*this will depend on your query sequence and database - more on this later!*)

Why do I need this tutorial you may say! Well, running BLAST+ is like running a lab experiment. To get meaningful results, you must first optimize the conditions you are using. After this tutorial you will not only be able to run BLAST, but be able to tailor your search to your specific biological question.

1.2 Learning outcomes

By the end of this tutorial you can expect to be able to:

- Create a BLAST database from your own sequences
- Describe the difference between BLAST programs and when to use them
- Run BLAST locally
- Generate tailored BLAST output files

1.3 Tutorial sections

This tutorial is split into two sections:

- Part 1: Creating a BLAST database
- Part 2: Running a local BLAST+ search

1.4 Authors

This tutorial was created by Victoria Offord.

1.5 Running the commands from this tutorial

You can run the commands in this tutorial either directly from the Jupyter notebook (if using Jupyter), or by typing the commands in your terminal window.

1.5.1 Running commands on Jupyter

If you are using Jupyter, command cells (like the one below) can be run by selecting the cell and clicking *Cell -> Run* from the menu above or using *ctrl Enter* to run the command. Let's give this a try by printing our working directory using the *pwd* command and listing the files within it. Run the commands in the two cells below.

```
pwd
```

```
ls -l
```

1.5.2 Running commands in the terminal

You can also follow this tutorial by typing all the commands you see into a terminal window. This is similar to the "Command Prompt" window on MS Windows systems, which allows the user to type DOS commands to manage files.

To get started, select the cell below with the mouse and then either press control and enter or choose Cell -> Run in the menu at the top of the page.

```
echo cd $PWD
```

Now open a new terminal on your computer and type the command that was output by the previous cell followed by the enter key. The command will look similar to this:

```
cd /home/manager/pathogen-informatics-training/Notebooks/BLAST/
```

Now you can follow the instructions in the tutorial from here.

1.6 Let's get started!

This tutorial assumes that you have BLAST+ installed on your computer. For download and installation instructions, please see ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/.

To check that you have installed the software correctly, you can run the following command:

```
blastn -h
```

This should return the following help message:

```
USAGE
  blastn [-h] [-help] [-import_search_strategy filename]
    [-export_search_strategy filename] [-task task_name] [-
db database_name]
    [-dbsize num_letters] [-gilist filename] [-seqidlist filename]
    [-negative_gilist filename] [-negative_seqidlist filename]
    [-entrez_query entrez_query] [-db_soft_mask filtering_algorithm]
    [-db_hard_mask filtering_algorithm] [-subject subject_input_file]
    [-subject_loc range] [-query input_file] [-out output_file]
    [-evalue evalue] [-word_size int_value] [-gapopen open_penalty]
    [-gapextend extend_penalty] [-perc_identity float_value]
    [-qcov_hsp_perc float_value] [-max_hsps int_value]
    [-xdrop_ungap float_value] [-xdrop_gap float_value]
    [-xdrop_gap_final float_value] [-searchsp int_value]
    [-sum_stats bool_value] [-penalty penalty] [-reward reward] [-
no_greedy]
    [-min_raw_gapped_score int_value] [-template_type type]
    [-template_length int_value] [-dust DUST_options]
    [-filtering_db filtering_database]
    [-window_masker_taxid window_masker_taxid]
    [-window_masker_db window_masker_db] [-soft_masking soft_masking]
    [-ungapped] [-culling_limit int_value] [-best_hit_overhang float_value]
    [-best_hit_score_edge float_value] [-window_size int_value]
    [-off_diagonal_range int_value] [-use_index boolean] [-
index_name string]
    [-lcase_masking] [-query_loc range] [-strand strand] [-
parse_deflines]
    [-outfmt format] [-show_gis] [-num_descriptions int_value]
    [-num_alignments int_value] [-line_length line_length] [-html]
    [-max_target_seqs num_sequences] [-num_threads int_value] [-remote]
    [-version]

DESCRIPTION
   Nucleotide-Nucleotide BLAST 2.7.0+

Use '-help' to print detailed descriptions of command line arguments
```

For the first part of this tutorial, we are going to look at how to create a BLAST database from a file containing your own sequences. Answers to all of the questions can be found here. To get started with the tutorial, head to the first section: Part 1: Creating a BLAST database

2 Part 1: Creating a BLAST database

2.1 Introduction

As mentioned before, we need two things to run a local BLAST search: * *Your query sequence* * *A database to search*

When you run BLAST online, you are offered a series of pre-formatted databases (e.g. nr/nt, refseq_rna...). You can download these databases from `ftp://ftp.ncbi.nlm.nih.gov/blast/db/`. This is great, but what if we want to search our query against our own set of sequences?

Your sequences will typically be in FASTA format, but BLAST cannot use this. So, this part of tutorial will show you how to use **makeblastdb** to convert your FASTA sequences into a format which BLAST can use.

2.2 Storing database files

Before we get started, let's consider a bit of housekeeping. BLAST databases are typically kept in a folder called **db**. Within this, it is good practice to give each of your databases *their own folder*. This is so that you don't accidentally overwrite the original files when you download newer versions of the same database or accidentally replace an old database by giving your new database the same name.

To have a look at what we mean, let's take a look at the **db** folder for this tutorial.

```
ls db
```

In this part of the tutorial, we are going to create a BLAST database from a set of FASTA-formatted *bacteria* sequences which can be found in the bacteria folder (*db/bacteria/bacteria.fa*). Let's take a closer look.

```
cd db/bacteria
```

```
ls
```

```
head bacteria.fa
```

What is the name of the file containing our FASTA sequences?
hint: it will have the file extension .fa or .fasta

What type of sequences do we have in our bacteria file?
hint: are they nucleotide or protein?

2.3 Creating a BLAST database

To create a BLAST database from our FASTA sequences we use the **makeblastdb** application. Information about the different parameters we can give to *makeblastdb* can be found by typing **makeblastdb –help**.

However, there are two parameters we must *always* give to **makeblastdb**: the location of our input file and the type of sequences it contains.

Parameter	Meaning
-in	The location of the file containing your FASTA sequences.
-dbtype	The type of sequences in your database (*e.g. nucleotide=nucl or protein=prot*)

Using these parameters, the command we need will take the format:

makeblastdb -in [*input file*] **-db_type** [*nucl or prot*]

Using the answers from the previous section and the information above, let's try creating our BLAST database.

```
makeblastdb -in bacteria.fa -dbtype nucl
```

Using the output generated from our command, try and answer the following:

What is our new BLAST database (DB) called?

How many sequences were added to our new database?

If you want to check that the number of sequences added to the new database match the number of sequences in our FASTA file we can use *grep*.

```
grep -c '>' bacteria.fa
```

** Was the number of sequences added to our database the same as the number of sequences in our FASTA file?**

Now let's take a look at the files we have created.

```
ls -l
```

You will notice that three new files have been created with new file extensions: .nhr, .nin and .nsq. You don't need to worry what these files are but in general: .nhr file are the headers, .nin the index and .nsq the sequences.

2.4 Naming databases and creating logfiles

In the previous section we created a database using *only* the required parameters. However, there are several other parameters which can be very useful.

Parameter	Meaning
-title	The name of the database (*e.g. how it will be referenced by BLAST*)
-out	The prefix for your output database files (*e.g. database.nin,database.nhr...*)
-logfile	The file in which to write all command output and errors

Let's take a look at what these parameters actually do. The following command will generate a BLAST database called **bacteria_nucl** from our FASTA sequences stored in **bacteria.fa** which can be recalled by BLAST using the reference **bacteria_nucl** and writes all command line output to **bacteria_nucl.log**

```
makeblastdb -in bacteria.fa -dbtype nucl -title bacteria_nucl \
            -out bacteria_nucl -logfile bacteria_nucl.log
```

Did you notice that this time there was no output (*e.g. Building a new DB,....*)? This has all been written to **bacteria_nucl.log**. Let's take a look.

```
head bacteria_nucl.log
```

Let's also take a look at the database files generated.

```
ls -l
```

Here you will see the files created by our first command, which used only the required parameters, have the prefix *bacteria.fa*. This is because by default -out is the same as -in (*see makeblastdb –help*). We changed this by giving a simpler prefix e.g. -out bacteria_nucl. This can be very useful when you have complex file names but want a simpler or more descriptive database name.

3 Exercise 1

You will have noticed that there is also a file in the /bacteria folder called **bacteria_tr.fa** which also contains FASTA sequences which need to be converted into a BLAST database. Create a BLAST database from this file which has the output prefix **bacteria_prot** and can be referenced using the title **bacteria_prot**.

It is up to you whether you create a logfile but it is worth using *head* to check the type of sequences. (*hint: they might not be nucleotide*).

What do you notice about the file extensions for the bacteria_prot database?
(*hint: use ls -l*)

Why do you think they are different from the previous files?
(*hint: sequence type*)

3.1 Summary

We have created two BLAST databases, one nucleotide (bacteria_nucl) and one protein (bacteria_prot), each containing 75 bacterial sequences which we will now use in the next part of the tutorial. Click here for how to run a BLAST search, or return to the index.

The answers to the questions in this tutoial can be found here.

4 Part 2: Running a local BLAST+ search

4.1 Introduction

BLAST+ is divided into different applications based on three broad categories:

1. Search tools
2. Database tools
3. Sequence filtering tools

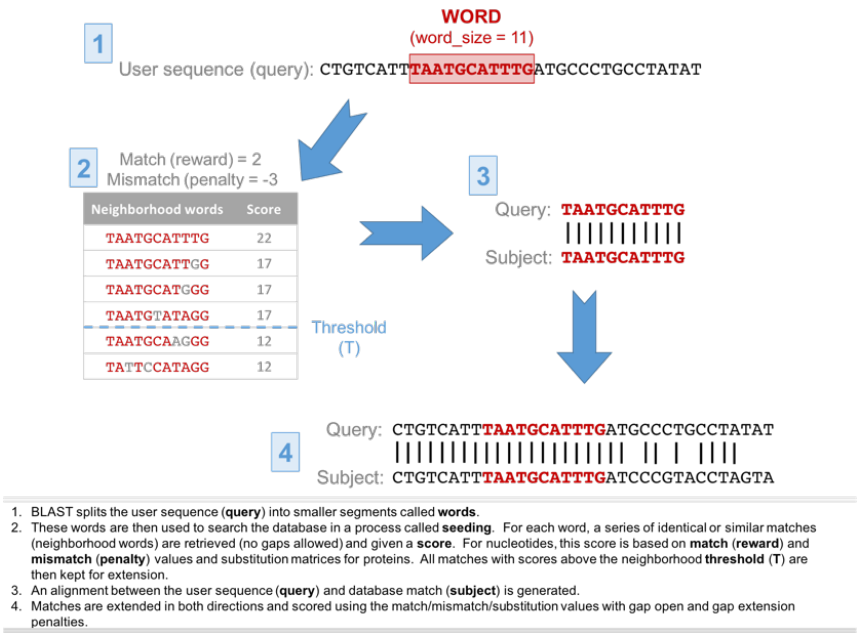
We have already seen a *database* tool when we used *makeblastdb* to generate our BLAST databases. However, we have not yet looked at the different search tools. These search tools are divided based on functionality, query type and database type. Below is a summary of the five main search tools.

Program	Input format	Database
blastn	nucleotide	nucleotide
blastp	protein	protein
blastx	translated nucleotide	protein
tblastn	protein	translated nucleotide
tblastx	translated nucleotide	translated nucleotide

The first two search tools, **blastp** and **blastn**, have an additional **-task** option which will optimize the parameters such as word size or gap cost. This can be useful if you are searching short, similar or dissimilar sequences. Below is a summary of the available tasks and their uses.

Program	Task name	Description
blastn	blastn	Traditional blastn search
blastn	blastn-short	Optimized for queries shorter than 50 bases
blastn	megablast	Optimized to search for sequences with high similarity (*e.g. intraspecies*)
blastn	dc-megablast	Discontiguous megablast. Optimised to search for more distant (*e.g. interspecies*) sequences
blastp	blastp	Traditional blastp search
blastp	blastp-short	Optimized for queries shorter than 30 residues

4.2 How do these BLAST applications work?



How BLAST works

Every BLAST search starts with a query sequence provided by you, the user. BLAST takes that query sequence and splits it into smaller fragments called *words*. It then uses these **words** to search the database for other sequences that contain identical or similar, words. This process is called **seeding**.

Each match or **hit** is checked to make sure that it meets a certain threshold or **score**. The score for each alignment is calculated using a **substitution matrix** and, if the score meets the threshold, the alignment moves forward for **extension**. During the extension phase, BLAST will try to increase the length of the alignment, extending out from either side. Extension will continue until the **score** for that alignment falls below a pre-defined threshold or **score**. The extended alignment is known as a **High-scoring Segment Pair (HSP)**.

So, to recap. The BLAST output for a query sequence is known as the **result**. When a sequence that is similar to the query is found in the database, this is known as a **hit**. Within the hit there may be multiple regions of similarity which can be aligned and are known as a **HSPs**. Each hit can have multiple **HSPs** which must each meet a minimum threshold or **score**.

4.3 Running a simple BLAST search

All of this sounds great, but it would be easier to understand if we could see an example, right? Then let's try running one of the simpler BLAST applications, **blastn**.

First we need to remember the three things we need for any BLAST search:

- A query sequence
- A sequence database
- A BLAST application

Now, take another look at the table above, what do we need for a **blastn** search?

So, we already have our application, **blastn** and we have already generated our nucleotide database **bacteria_nucl** in the first part of the tutorial here. All we need now is our nucleotide query sequence which can be found in **example/unknown.fa**.

Let's take a look and check that our query is a nucleotide sequence!


```
ls example
```



```
cat example/unknown.fa
```

So, now we have our three essentials let's run our **blastn** search. To look at the parameters available type **blastn -help**.

Parameter	Meaning
-task	Only for blastn and blastp. Defaults to megablast for blastn.
-query	The location of the file containing your query sequence.
-db	Location and reference (e.g. bacteria_nucl) of your BLAST database
-out	Location and name of the output file

The format of the command will be:

blastn -task [*task*] **-query** [*input file*] **-db** [*database reference*] **-out** [*output file*]

As we are not in the same directory as our database, we will need to tell the program to look in the **db** folder. We can do this by putting the location (relative to where we are now) before the reference e.g. **db/bacteria/bacteria_nucl**. You can do the same for the output file which we want to write to the **example** folder. It is normally a good idea to give your output file a descriptive name. Here we use the program and a generic description of the database being queried e.g. **blastn_bacteria.out**.

Now, let's try and identify our unknown sequence using **blastn**!


```
blastn -task blastn -query example/unknown.fa \
           -db db/bacteria/bacteria_nucl \
           -out example/blastn_bacteria.out
```

If this has worked, you should be able to see a new file in the **example** directory called **blastn_bacteria.out**. To look at your file, you can open it in a text editor or look at it in the terminal using the command:

less example/blastn_bacteria.out

The output can be split into two sections: a summary **hit** table and the corresponding **alignments**. If we were to have multiple queries, these sections would then be replicated for each query.

Let's take a look at the hit summary...

```
grep -A 10 'Query=' example/blastn_bacteria.out
```

The hits from the **bacteria_nucl** database which match our unknown sequence are ordered by their **bit score** from highest to lowest, the highest representing the *best* hit. This is not the same as the alignment score that we were discussing earlier. The **bit score** is derived from those alignment scores, but is normalised so that it is possible to compare the alignment scores from different searches.

In addition to a bit score, each hit is also given an **E Value** which represents the number of different alignments that have scores which are the same or better than a score which is expected to occur by chance in a database search. Broadly speaking, it is a measure of confidence. The lower the value the more confident you can be that this score would not occur by chance.

Let's take a look at the lower end of the table and see how the bit score and e value differ.

```
grep -A 5 '  AY461808.1 ' example/blastn_bacteria.out
```

Now let's look at the alignment which corresponds to the *best* hit (GQ903013.1).

```
grep -A 18 '> GQ903013.1' example/blastn_bacteria.out
```

The alignment gives more detail about this match than the summary table. Here, the hit is referred to as the **subject** (or Sbjct). The alignment gives us details about the orientation of our hit and query sequences. In this case, the sequences align in the same, forward direction (Plus/Plus) but this may not always be the case (e.g. if the hit was in the reverse orientation Plus/Minus). It also provides information on the number of exact matches (Identities) and gaps within our alignment, both of which contribute to the alignment score.

Based on the output of our blastn search, which species do you think our unknown sequence comes from? What gene might it be?

4.4 Output formats

BLAST results can be written in different formats. If we don't specify an output format the default is **pairwise** which contains a summary hit table and the corresponding alignments as we saw above. There are several other useful formats which are available using the **-outfmt** parameter.

-outfmt value	** Description **
0	pairwise
1	query-anchored showing identities
2	query-anchored no identities
3	flat query-anchored, show identities
4	flat query-anchored, no identities
5	XML Blast output
6	tabular
7	tabular with comment lines
8	Text ASN.1
9	Binary ASN.1
10	Comma-separated values
11	BLAST archive format (ASN.1)

If you don't need to see the alignments, a tabular output is often the most simple to work with. Let's try adding **-outfmt 6** to our command (don't forget to change the output file name!!).

```
blastn -task blastn -query example/unknown.fa \
           -db db/bacteria/bacteria_nucl \
           -out example/blastn_bacteria_outfmt6.out -outfmt 6
```

And take a look at what we've got...

```
head -10 example/blastn_bacteria_outfmt6.out
```

Our output is now in a tab-delimited format but we have no column names. By default, these are:

Heading tags	Meaning
qseqid	Query identifier/accession
sseqid	Subject (hit) identifier/accession
pident	Percentage of identical positions in alignment
length	Alignment length
mismatch	Number of mismatches in alignment
gapopen	Number of gaps in alignment
qstart	Start position of alignment in query
qend	End position of alignment in query
sstart	Start position of alignment in query
send	End position of alignment in query
evalue	E Value
bitscore	Bit Score

One useful statistic that we are given when we use the online version of BLAST is the percentage of our query that has been aligned, also known as our query coverage. Not to worry, we don't have to manually calculate this as BLAST has some extra parameters we can use. For more information try **blastn -help** and look at the **Formatting options** section.

Let's say we don't want to know all of the alignment statistics, how can we generate a summary which tells us: query id, subject id, query length, subject length, alignment length, percentage identity, query coverage, bit score and evalue? Well, we need to specify the corresponding tags. To do this, we still need to use the **-outfmt** parameter but now we put our format identifier (0-11) followed by the tags for the columns we want to include. The tags should be separated by a single space with the format identifier and tags all enclosed in double quotes. Let's give it a try, it will make much more sense once you see it written out below!

```
blastn -task blastn -query example/unknown.fa \
          -db db/bacteria/bacteria_nucl \
          -out example/blastn_bacteria_final.out \
          -outfmt "6 qseqid sseqid qlen slen length \
          pident qcovs bitscore evalue"
```

Let's take a look at our output. Remember, the columns are in the same order as you specified in the command.

```
head -10 example/blastn_bacteria_final.out
```

From this output, you should now be able answer the following questions.

What percentage of our query aligns with our top hit?

Is our query sequence the same length as our top hit?

4.5 Using different tasks to optimise parameters?

In the last section of the tutorial, you will have noticed that we used the **-task** parameter to tell **blastn** that we want to use the **blastn** parameters. But what are these parameters?

If you remember back at the start, we described how BLAST splits your query sequence into smaller segements called **words**. The length of the word is defined by a parameter called **-word_size** which has a default value of 11. Broadly speaking, we can think of this as the minimum length of the initial alignment which can be found and extended by BLAST. So,if you have a large database, you can increase the speed of your search just by increasing word size.

The word size is just one of the parameters which is automatically changed when you use tasks such as **megablast**. For **megablast**, the word size is increase to a default of 28 and the cost of opening and extending gaps in the alignment is optimised to find long, highly similar alignments. This is why **megablast** is very efficient and particularly suited to interspecies comparisons.

Let's see if using **-task megablast** parameters instead of blastn changes our results. We are going to use the default **-outfmt 6** columns this time.

```
blastn -task megablast -query example/unknown.fa \
           -db db/bacteria/bacteria_nucl \
           -out example/megablast_bacteria_outfmt6.out -outfmt 6
```

Let's take a look at our output. Looks pretty similar to the blastn results, right?

```
head -10 example/megablast_bacteria_outfmt6.out
```

Well, that's not quite true. Let's see how many results we have in both our blastn and megablast searches.

```
wc -l example/blastn_bacteria_outfmt6.out
```

```
wc -l example/megablast_bacteria_outfmt6.out
```

Did the blastn and megablast searches produce the same nummber of hits? Why do you think this is?
(*hint: have a look at the end of the pairwise alignment file and think about the default megablast word size*)

4.6 Searches using translated nucleotide sequences

Sometimes, depending on the biological question (e.g. don't do this with primers!), it can be better to perform a BLAST search using a translated nucleotide query. The simplest explanation is that

several codons may encode the same amino acid (redundancy). So, while there may be differences between two nucleotide sequences, they may in fact encode the same amino acid sequence.

So far, we have created our own BLAST database of bacteria sequences and identified our unknown sequence as TcpC from *Escherichia coli*. But, is it only found in bacteria?

5 Exercise 2

We can't use our bacteria database for this search but we can use what you learnt in the first part of the tutorial, here, to generate a new database. There are some sequences provided for you in the db/mammalian folder. Let's take a look.


```
ls db/mammalian
```

```
head db/mammalian/mammalian.fa
```

Using mammalian.fa create a new database which has the output prefix *mammalian* and can be referenced as *mammalian*. (_hint: you don't need to be in the same folder as your FASTA file to write your database files there, just prefix the output prefix with the relative location - e.g. db/mammalian/mammalian)

If our query sequence is nucleotide and we want to search a protein database, what BLAST application do we need to use?
(*hint: look at the BLAST application table above*)

** With example/unknown.fa, run a BLAST search using the application in your answer above and search the database you have just created. We want a standard tabulated output file with the following additional columns** * Full subject title * Query length * Subject length * Percentage query coverage

Notice in the previous tabulated output there is only the subject accession, not the full title/description. For the answer to this exercise you should look at stitle on the application help page. Also, you don't need to specify all of the standard output columns, just use std (e.g. -outfmt "6 std extra1 extra2…". Remember, as we are not using either blastn or blastp, we do not need the -task parameter.

** What is our top hit? How much of our query sequence is covered by this alignment? What is the length of our top hit and where does the alignment start and finish?**

So, our original question was whether TcpC is only found in bacteria. The answer is both yes and no. Looking at our answers, we could not find the whole TcpC protein in the mammalian database (see your answer to for query coverage). However, we did find a region of similarity in mammalian toll-like receptors. Biologically, this makes sense as TcpC and other bacterial protiens contain a region called a TIR domain. These domains are also found in mammalian innate immune receptors which include toll-like receptors. There is evidence to suggest that the similarity between the bacterial TcpC and mammalian immune receptor TIR domains allow the bacteria to interfere with the host immune system. And we can see much of this with a simple BLAST!

Well done, you have finished this tutorial! You can return to the index or revisit the previous section.

The answers to the questions in this tutoial can be found here.