13 The DIAMOND algorithm

This chapter is based on the following paper, which is recommended reading:

- Buchfink, B, Xie, C, & Huson, DH. Fast and Sensitive Protein Alignment using DIAMOND, Nature Methods, 12:59-60 (2015)

Functional analysis is an important computational problem in microbiome analysis.

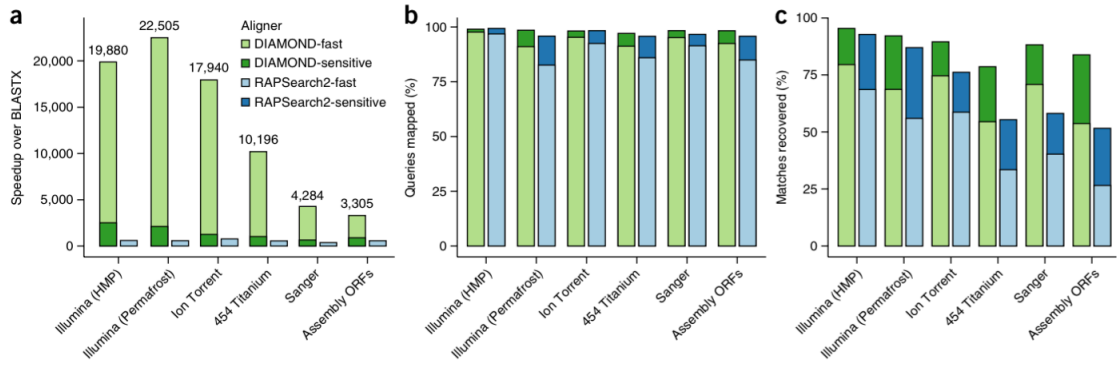
Typical problem:

- You have ≈ 3 billion DNA reads of length 100 –150 bp
- You want to compare them against a reference database of proteins, e.g. NCBI-nr, currently ≈ 54 million entries

Need to perform protein-protein local alignment on all 6-frame translations of the reads, that is, on 18 billion queries.

To solve a problem of this size using BLASTX takes over 30 years on a single server (using 48 cores). DIAMOND solves this problem in about one day. DIAMOND is approximately 20,000 times faster than BLASTX on the typical problem, without much loss of sensitivity.

Comparison of BLASTX, RAPSearch2[1] and DIAMOND:



The goal of this chapter is to discuss some of the concepts on which the program DIAMOND is based.

13.1 BLAST

The tool of choice for this problem was BLASTX:

- It translates each read in all six frames
- It extracts a set of "seeds" for each given query
- Seeds are compared against the reference sequences
- Seed matches are "extended" so as to produce local alignments

BLASTX is too slow to be applied to high-throughput sequencing data. How to do this type of calculation significantly faster?

[1] Zhao, Y., Tang, H. & Ye, Y. Bioinformatics 28, 125-126 (2012)

More detailed outline of the algorithm employed by BLAST:

Consider the following query: `P Q G E F G` (this is a translated read).

Extract all seeds of fixed length (here: 3):

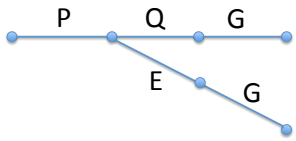
`PQG`, `QGE`, `GEF`, `EFG`,

Find all similar words with a BLOSUM62 (or other) score $\geq T$, where T is a threshold, 15, say.

Example: For `PQG` we also use `PEG`, but not `PQA`, because:

`P Q G`	`P Q G`
`P E G`	`P Q A`
7+2+6 = 15	7+5+0 = 12

Once all seeds have been produced, they are put into a "keyword tree":



This is used to scan the reference sequences for seed matches:

`A L V L P E G V L V`

Another example:

Query:	`...L A R R G G V K`	`R I S ...`
Seeds:	`G V K`	18
	`G A K`	16
	`G I K`	16
	`G G K`	14
	`G L K`	13
	`G N K`	12
	...	

Possible alignment at seed match:

```
Query:      ...L A R R G G V K R I S ...
            ...L         G + G   I + ...
Reference:  ...L N Y G I G A K V I A ...
```

Original BLAST uses ungapped extension to full alignment:

```
...R G G V K R ...
 ←             →
...G I G L K V ...
```

Extension is stopped when the running score drops X below the best score seen so far, with $X = 15$, say.

The challenge: How to increase speed without losing sensitivity?

13.2 Reduced alphabet

For protein sequences, BLAST generates neighboring words to increase the sensitivity of seeds.

An alternative approach to increase sensitivity is to use a reduced alphabet for seeds, treating letters that often replace each other in alignments as identical.

Here is a reduction to four letters:

```
[LVIMC] [FYW] [AGSTP] [EDNQKRH]
```

Here is a reduction to 11 letters:

```
[KREDQN] C G H [ILV] M F Y W P [STA]
```

13.3 Representing seeds and their locations

Consider an alphabet using k letters. Then we can intepret a seed of length k as a base-k number.

If using an alphabet with $k = 11$, then we can represent any seed with ≤ 18 letters by a single 64-bit number, because $11^{18} < 2^{64} < 11^{19}$.

Let S be a seed that occurs in reference number r at position p.

We describe S and its occurrence using two 64-bit numbers, namely a, the encoding of S, and b, the *location*. The location is a 64-bit number obtained by packing the two 32-bit numbers r and p together.

13.4 Double indexing

"Double indexing" is an alternative to the keyword approach of BLAST and also to the use of a hash table.

Outline:

1. Extract all reduced-alphabet seeds from references
2. Sort seeds lexicographically
3. Extract all reduced-alphabet seeds from queries
4. Sort seeds lexicographically
5. Traverse both sorted lists simultaneously to find common seeds, as discussed later.

13.5 Sorting seeds

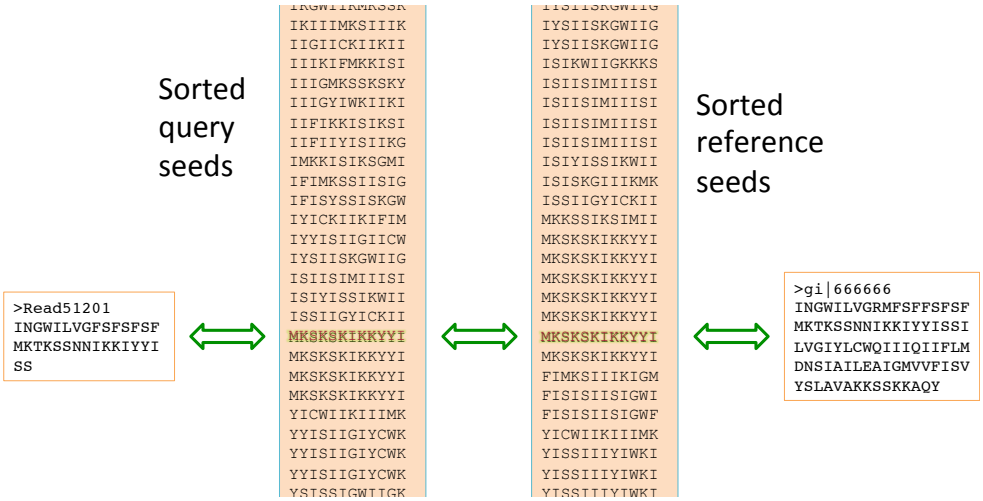
Want to sort seeds, together with their locations, quickly. The data is pairs of numbers (a, b), where a is the encoding of a seed and b its location. Want to sort by the first component.

Idea: use *radix* sort ("counting sort").

Using radix sort, DIAMOND can sort all seeds extracted from NCBI-nr (tens of billions of seeds) in a few minutes on a server.

13.6 Seed matching

Once all seeds in the query file and in the references file have been sorted, then both lists can be traversed simultaneously to find all seed matches:



13.7 Partitioning

Assume the reference database has 35 million sequences with 500 seeds each.

Then there are 7 billion seeds in total and each requires 16 bytes for storage of the encoding and the location, thus the memory usage is:

$$\approx 112\text{GB}.$$

We can easily partition the problem into 2^h parts using the lowest h bits of the seeds.

Choose h so as to get the desired number of parts 2^h. For example, using $h = 10$ will partition the problem into 1024 parts.

Each of the parts can be processed independently.

For example, we can create a queue of all parts and dispact jobs to a set of worker threads.

And/or we can parse through the input files multiple times, each time reading different parts of the list of seeds into memory for processing.

(Exercise: Show that the following is true: If we use an alphabet of size 11, seed length 11, and $h = 10$, then any seed can be represented by a 32-bit integer.)

13.8 Spaced seeds

The seeds used by BLASTX, and most other sequence database search tools, are *contiguous* seeds that consist of a set of consecutive letters.

A spaced seed consists of a segment of *length* n in which only a fixed subset of positions is used, and the number of used positions is called the *weight* k. The exact layout of the positions to be used is determined by the *seed shape* Z, which is specified by a bit vector, e.g.:

$$Z = 111101101110111,$$

where 1's indicate positions to use and 0's indicate positions to ignore.

Let Z be be spaced seed shape of weight k and length n. (Note that Z is a contiguous seed in the special case of $k = n$.)

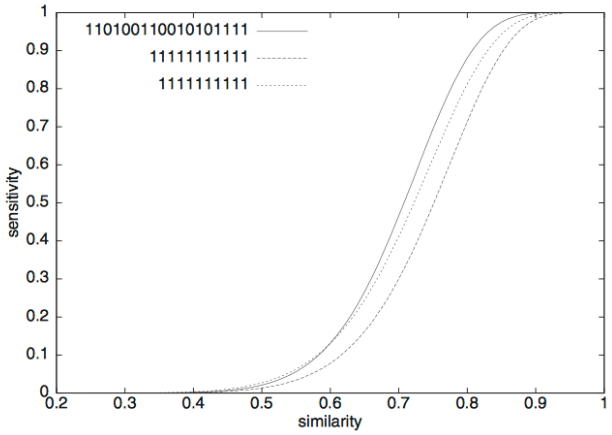
Let q be a query string and r a reference string.

We say that there is a seed *match* of shape Z between q and r if there exists a position i_0 in q and j_0 in r such that $q[i_0 + k] = r[j_0 + k]$ for all k for which the k-th bit in Z is 1 (with all indices starting at 0).

Here is an example with $i_0 = 1$ and $j_0 = 2$:

```
Reference     SLWAKKRTVDGQPKWLPLVAHLVDASNVSRMLFNQWLSD
Spaced seed     1111011011101 11
Query          FWAKKRTVNGQPLWLPLTQHLEDASNVSR
```

For seeds used in DNA alignment, a study was undertaken to compare the sensitivity of spaced seeds with that of contiguous seeds[2]. Dynamic programming was used to compute the sensitivity when comparing a query and reference DNA sequence, both of length 64, for different levels of similiary, comparing a spaced seed of weight 11 with a contiguous seed of length 10 and 11:



(Source: Ma et al, 2002)

This plot shows that a spaced seed of weight 11 is more sensitive than a continguous seed of the same weight, or even of smaller weight 10.

Note that increasing the seed weight by 1 reduces the number of random hits by a factor of K, where K is the alphabet size.

In consequence, a spaced seed of weight 11 is more sensitive and produces less random hits than a contiguous seed of weight 10.

13.9 Multiple spaced seeds

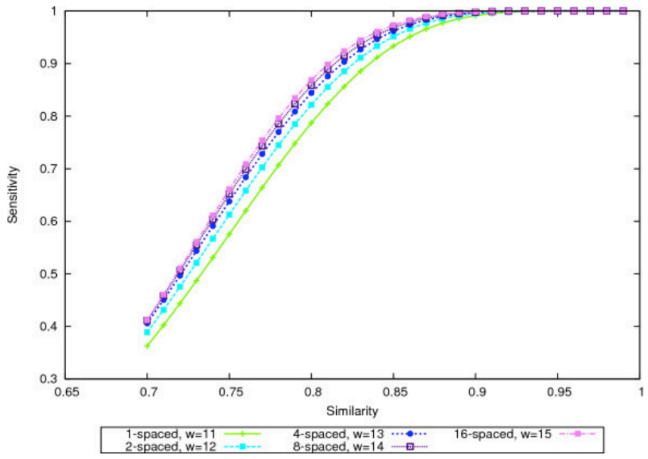
One way to improve the sensitivity is to decrease the seed weight, at the cost of increasing the number of random hits by a factor of K.

An alternative is to add another seed shape. This can significantly increase the sensitivity while only doubling the number of random hits.

This figure shows that doubling the number of seeds provides an increase in sensitivity even when simultaneously increasing the weight[3]:

[2] Ma *et al* (2002) PatternHunter: faster and more sensitive homology search. Bioinformatics 18, 440–445

[3] Ilie *et al* (2011) Seeds for effective oligonucleotide design, BMC Genomics 120



(Source: Ilie et al, 2011)

Hence, sensitivity is best increased by using multiple different seed shapes. For example, by default, the program DIAMOND uses the following four seed shapes of weight 12 and length $15 - 26$:

111101101110111
1111000101011001111
11101001001000100101111
111010010000101000010100111

13.10 Left-most seeds

Let Z be a seed shape that we have used to compute a double index for a set of query sequences Q and reference sequences R.

Let q and r be a query and reference sequence, respectively, and assume that two sequences are very similar.

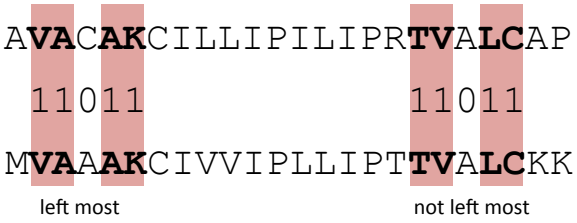
It may occur that there are more than one seeds for which q and r match, call them S_1 and S_2, say, both being part of the same larger alignment of q and r.

In the doubling index approach, S_1 and S_2 will usually be encountered in quite different positions in the sorted lists of seeds for Q and R, respectively.

How to avoid extending and then reporting the same alignment twice?

Definition 13.10.1 (Left-most seed) *Let Z be a seed shape and q and r two sequences. We say a seed match at positions i in q and j in r is* left-most*, if there is no other seed match with positions i' in q and j' in r that is* further left*, that is, with $i' < i$ and $j' < j$, and is on the* same diagonal*, that is, with $i' - j' = i - j$.*

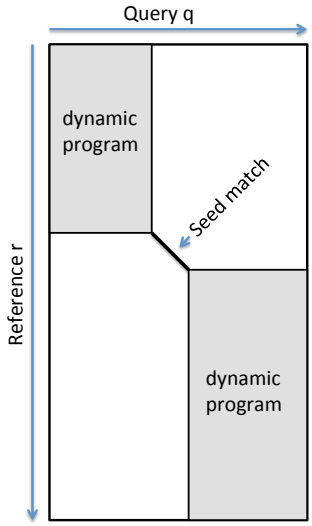
Being left-most is a property that can be easily checked by simply sliding the seed shape Z to the left in both sequences, position-by-position, and checking whether a match can be found:



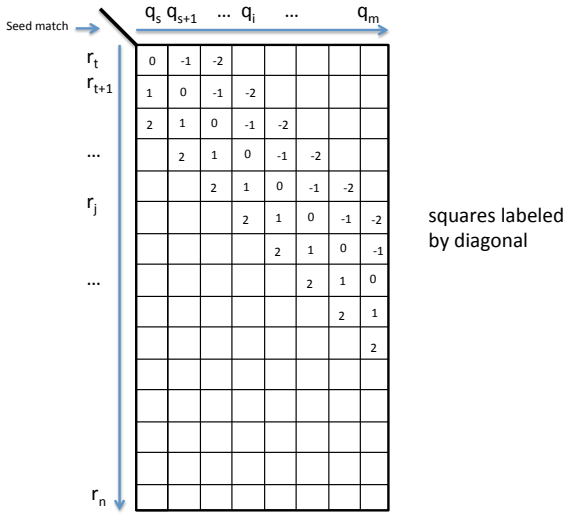
13.11 Extension

Once a seed match has been identified, the computational task is to extend the seed match to a full local alignment.

This requires two dynamic programs, as indicated here:

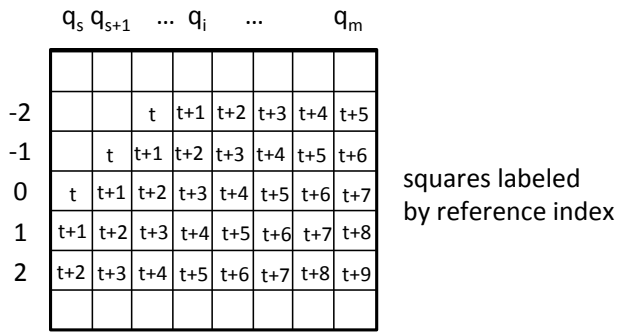


Let us look closer at the bottom right alignment problem:



For the sake of speed, we intend to perform a banded dynamic program that only takes the labeled cells into account, that is, only those cells that are within a distance of $w = 2$ to the main diagonal.

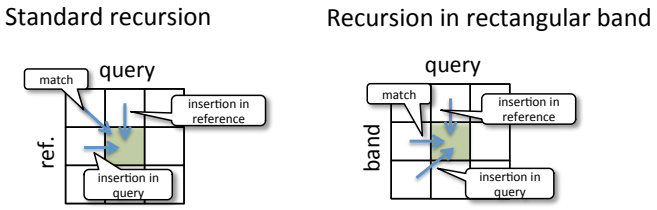
We can transform the matrix into a matrix that is centered along the band:



13.12 Banded alignment

We perform banded alignment in the rectangular matrix shown above. The first and last rows are initialized to the value $-\infty$.

We assign the substitution score $S(q_s, r_t)$ to the cell $F(s, 0)$ and then define the recursion based on the following figure:



Note that we compute the index $j(i, k)$ of the reference sequence as a function of the indices of a given cell in the banded alignment matrix as follows:

$$j(i, k) = t + (i - s) + k.$$

The recursion is then as follows, for all $i = s \dots m$ and $k = -w \dots w$ with $t \leq j(i, k) \leq n$:

$$F(i, k) = \max \begin{cases} F(i-1, k) + S(q_i, r_{j(i,k)}) \\ F(i, k-1) - d \\ F(i-1, k+1) - d \end{cases}$$

where d is the gap penalty and w the band width.

When filling the matrix, we keep track of the cell z that contains the best value seen so far. It's value contributes to the total alignment score, which consists of the sum of the best score for the left and right alignments, plus the seed score.

Traceback is performed from z to the cell with coordinates $(s, 0)$.

To speed up the calculation of local alignments, one can use the following so-called *X-drop heuristic*:

Let Z be the best score seen in any of the previous columns. If the best score in the current column is less than $Z - X$, then do not consider any further columns. Here, X is a value, of 30, say, for protein sequences.

13.13 The DIAMOND algorithm

In summary, here is an outline of the DIAMOND algorithm.

Input the list of query sequences Q.

Translate all queries, extract all spaced seeds and their locations, using a reduced alphabet. Sort them. Call this $S(Q)$.

Input the list of reference sequences R.

Extract all spaced seeds and their locations, using a reduced alphabet. Sort them. Call this $S(R)$.

Traverse $S(Q)$ and $S(R)$ simuntaneously. For each seed S that occurs both in $S(Q)$ and $S(R)$, consider all pairs of its locations x and y in Q and R, respectively: check whether there is a left-most seed match. If this is the case, then attempt to extend the seed match to a significant banded alignment.

13.14 Summary

- Alignment of queries against a set of reference sequences is usually done in a seed-and-extend fashion
- One approach is to build an index for the seeds of the references and then to search for each query seed in the index
- An alternative to this is to sort all seeds occurring in the references, and in the queries, respectively, and then to traverse the lists together
- Spaced seeds offer more sensitivity and specificity than continguous seeds

Based on these ideas, the program DIAMOND can align short sequencing reads against the NCBI-nr database 20,000 times as fast as BLASTX.