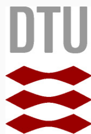
Aligning DNA sequences on compressed collections of genomes

Part 2. Alignment

The CODATA-RDA Research Data Science Applied workshop on Bioinformatics
ICTP, Trieste - Italy
July 24-28, 2017

Nicola Prezza

Technical University of Denmark
DTU Compute
DK-2800 Kgs. Lyngby
Denmark



Problem

Now we have a draft G_H of the Human genome. How do we obtain the genome sequence G_x of a specific individual x? (x=your name)

Solution 1: de novo assembly

We can try sequencing+assembling from scratch (*de novo assembly*). Too expensive (we need a lot of sequencing) and time-consuming (assembling is slow).

Solution 2: alignment and calling

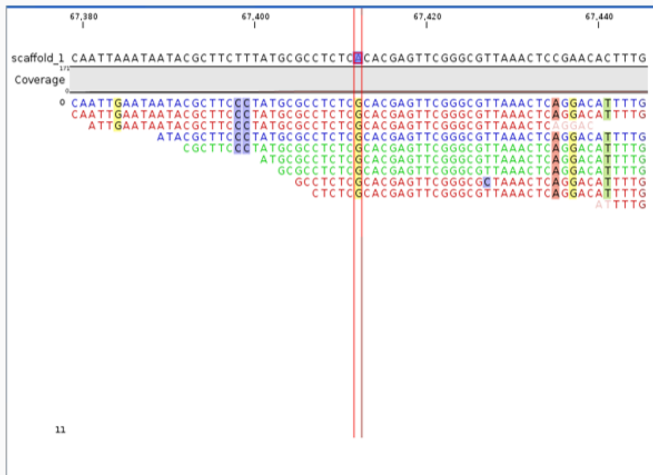
Can we exploit G_H? idea:

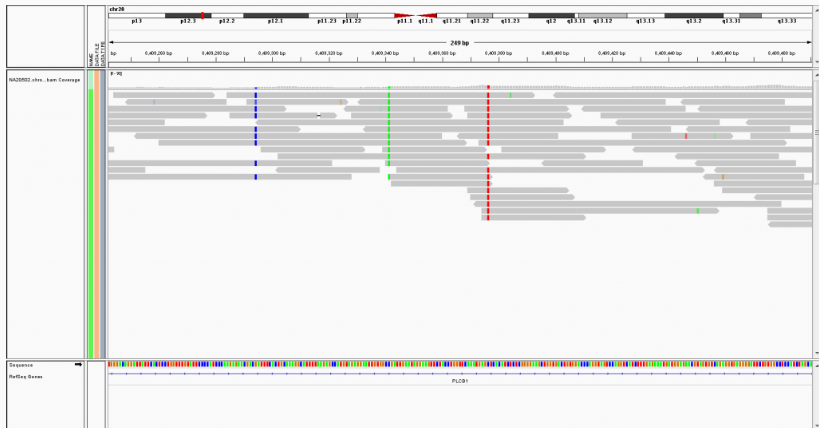
1. Sequence G_x and obtain a set of **reads** $r_1, r_2, \ldots$ (read = DNA string of length ≈ 100)
2. **Alignment**: search each r_i inside G_H. Maybe r_i occurs with some errors: find the best match.
3. **Calling**: using G_H and the differences between $r_1, r_2, \ldots$ and G_H, reconstruct G_x.

Solution 2 (alignment) is much more convenient w.r.t. solution 1 (assembly) because:

- We need to sequence less. In technical terms, we need less **coverage**
- Alignment is much faster than assembling

Alignment and calling




chr20
p13
p12.3
p12.2
p12.1
p11.23
p11.22
p11.1
q11.1
q11.21
q11.22
q11.23
q12
q13.11
q13.12
q13.13
q13.2
q13.31
q13.33
249 bp
Sequence
RefSeq Genes
PLCB1

The pattern matching problem

Technical problems that we will ignore

1. r_i can occur inside G_H with (a lot of) differences: mismatches, indels, gaps, inversions ...
2. The best match of r_i can occur in multiple places: which one do we choose?

In order to study the problem from a clean computational standpoint, we consider the simpler case of exact pattern matching:

Problem definition: exact pattern matching

- **Input**: a set of strings (or reads) $r_1, r_2, \ldots, r_t$, all of length m, and a text (genome) G of length $n \gg m$.
- **Output**: for each string r_i, the set of positions $i_1, \ldots, i_{occ}$ where r_i appears inside G

Example

G = AGCATCAGCCTCGGCAGGATGCATTTCACATTTGTGATCTCATTTAACCTCCACAAAGACC

r_1 = CCT

r_2 = TTT

read r_1

G = AGCATCAGCCTCGGCAGGATGCATTTCACATTTGTGATCTCATTTAACCTCCACAAAGACC

output: 9, 48

read r_2

G = AGCATCAGCCTCGGCAGGATGCATTTCACATTTGTGATCTCATTTAACCTCCACAAAGACC

output: 24, 31, 43

How do we solve the pattern matching problem?

First solution

A first idea: for each read r_i, we scan the whole text G to search matches

Time cost

We have t reads of length m and G's length is n. In total, we need to perform $t \cdot m \cdot n$ steps. This can be lowered to $t \cdot n$ steps with specialized algorithms.

In practice ...

t is usually in the order of 10^8. For the Human genome, $n \approx 3 \cdot 10^9$. Assuming that each step takes $\approx 1\ ns = 10^{-9}\ s$ on a computer, then $t \cdot n \approx 3 \cdot 10^9\ s \approx 9.5$ years.

Clearly, scanning G for each read is not a good idea ...

Analogy

Suppose someone gives you a book, and asks you to find a particular word in it. Without further information, the best thing you can do is to read the whole book in order to find that word.

After you read the book, this person asks you to find another word in it. Well, you need to read the book again ...

... unless the first time you read the book you stored some information. Example: for every word in the book, the line/position where it appears.

Inverted and q-gram indexes

Example

```
G  = "When on board H.M.S. Beagle, as naturalist, I was much struck with certain facts in the
distribution of the organic beings inhabiting South America, and in the geological relations
of the present to the past inhabitants of that continent. These facts, as will be seen in the
latter chapters of this volume, seemed to throw some light on the origin of species--that mystery
of mysteries, as it has been called by one of our greatest philosophers. On my return home, it
occurred to me, in 1837, that something might perhaps be made out on this question by patiently
accumulating and reflecting on all sorts of facts which could possibly have any bearing on it.
After five years’ work I allowed myself to speculate on the subject, and drew up some short notes;
these I enlarged in 1844 into a sketch of the conclusions, which then seemed to me probable: from
that period to the present day I have steadily pursued the same object. I hope that I may be excused
for entering on these personal details, as I give them to show that I have not been hasty in coming
to a decision. "

from "The Origin of Species", Charles Darwin
```

Additional information (words in alphabetic order)

```
"as": line 1, word 6; line 3, word 13; line 5, word 3;
...
"years": line 8, word 3.
...
"After": line 8, word 1
...
"Beagle": line 1, word 5
...
```

Indexing

The additional information we store to speed-up subsequent searches is called **index**

Definition

An **index** $\mathcal{I}(G)$ of some text G is a collection of information—we call it a **data structure**—that speeds up searches of words inside G

Inverted indexes

The index of our example takes the name inverted index (can you tell why "inverted"?)

Indexing

First complication

DNA is not organized in words (i.e. no spaces). How can we build an inverted index?

Possible solution

Put in the inverted index all sub-strings of length q, for some $q > 0$. This index takes the name of q-gram index.

This solution is valid and is used in practice. However, how do we find reads longer than q? solution used in practice: seed and extend

Example: seed and extend

Example

`G = AGCATCAGCCTCGGCAGGATGCATTTCACATTTGTGATCTCATTTAACCTCCACAAAGACC`

Our q-gram index, with q=3

`AAA: 55`

`AAC: 46`

`ACA: 28, 53`

...

`TTT: 24, 31, 43`

Find r_1 = ACATTTGT

`G = AGCATCAGCCTCGGCAGGATGCATTTCACATTTGTGATCTCATTTAACCTCCACAAAGACC`

1. **Seed**: using the inverted index, we find occurrences of `ACA`: 28, 53
2. **Extend**: check if also the remaining characters `TTTGT` match. We discard occurrence 53. Output = 28.

Exercise

Build the 3-gram index (i.e. q=3) of the following genome:

`AGCATCAGCCACGCCATCATC`

q-gram indexes are a simple solution that is used also in practice. However, they do not offer guarantees in the worst case:

What happens if `ACATTTGT` appears just 1 time inside the genome, but `ACA` appears 1.000.000 times?

In this case, our seed-and-extend strategy needs to check all 1.000.000 occurrences of `ACA` before finding the occurrence of `ACATTTGT`

Full-text index

We therefore re-formulate our definition with stronger requirements:

Definition: full text index

A **full-text index** $\mathcal{I}(G)$ of some text G is a data structure that permits to find all *occ* occurrences of a string of length m inside G in time proportional to $m + occ$.[1]

Note: in practice, m and occ are very small when compared to n. On average, reads coming from a sequencing experiment satisfy $m \approx 100$ and $occ \ll 100$.

$m + occ \leq 200$ steps are therefore much better than $n \approx 3 \cdot 10^9$ steps (approximately 10.000.000 times faster) required if we do not build any index!

[1] To be precise: we are satisfied even with a time proportional to $(m^c + occ) \cdot \log^d(n)$, where c and d are constants

A first (space-inefficient) solution

There is a first, simple solution that solves the pattern matching problem in just $m + occ$ steps: we index all strings of length q, *for every* $q = 1, 2, \ldots, n$

`G = AGCATCAGCCTCGGCAGGATGCATTTCACATTTGTGATCTCATTTAACCTCCACAAAGACC`

Our full-text index

`A`: 1, 4, 7, 16, ...

`C`: 3, 6, 9, 10, ...

...

`AA`: 46, 55 ...

...

`AAA: 55`

`AAC: 46`

`ACA: 28, 53`

...

`AGCA...AAGACC`: 1

What is the problem of this index?

A first (space-inefficient) solution

Problem

This index is too BIG. How many strings are we putting inside the index?

A first (space-inefficient) solution

There are n strings of length 1, $n-1$ strings of length 2, $n-2$ strings of length 3, ..., 1 string of length n

In total: $n + (n-1) + (n-2) + ... + 1 = \frac{(n+1)n}{2} \approx n^2$ strings

Even if we spend only 1 Byte per string in our index, on the Human genome this space is $\frac{(n+1)n}{2} \approx 10^{18} \approx 1$ million Terabytes of data.

One more requirement: space

We therefore ask one last requirement:

The full-text index should not take too much space. Ideally, we want to use a space proportional to n Bytes.

Example

For example, $4 \cdot n$ Bytes could be a reasonable size. On the Human genome, $4 \cdot n$ Bytes ≈ 12 GigaBytes

Suffix arrays

Idea: suffix sorting

Definition

A **suffix** of a string w is a string that begins in the middle of w and ends at the end of w. Similar for **prefix** (i.e. string that begins at the beginning of w and ends in the middle of w).

Example

All suffixes of `ATTTGTG` are:

- `ATTTGTG`
- `TTTGTG`
- `TTGTG`
- `TGTG`
- `GTG`
- `TG`
- `G`

Idea: suffix sorting

First idea: build a (sorted) dictionary with just the suffixes of our genome and their starting positions.

Example

All alphabetically-sorted suffixes of `ATTTGTG` and their starting positions are:

1. `ATTTGTG`: 1
2. `G`: 7
3. `GTG`: 5
4. `TG`: 6
5. `TGTG`: 4
6. `TTGTG`: 3
7. `TTTGTG`: 2

Idea: suffix sorting

Now, note that any string that appears in the text is a prefix of a contiguous range of suffixes in our list.

Example

Suppose we are searching `TT`. Then two suffixes are prefixed by `TT`:

1. `ATTTGTG`: 1
2. `G`: 7
3. `GTG`: 5
4. `TG`: 6
5. `TGTG`: 4
6. `TTGTG`: 3
7. `TTTGTG`: 2

Looking at their starting positions, we can tell that `TT` occurs at positions 3 and 2 inside `ATTTGTG`

Idea: suffix sorting

Pros

- This index works for every string length (i.e. q is not fixed)
- All occurrences of the string are in a contiguous range in our list
- Easy to find the range quickly: binary search (the same you use to search a word in a dictionary)
- Only n strings in our index!

Cons

Unfortunately, there is still a problem... our index contains a string of length 1, one string of length 2, ..., one string of length n. How many characters in total?

Every character takes one Byte, so again we have
$1 + 2 + 3 + ... + n = \frac{(n+1)n}{2} \approx n^2$ Bytes

suffix sorting

We are however close to a solution. We only need to take one little step further: do not store the suffixes' characters, just their starting positions in the text.

- `ATTTGTG`: 1
- `G`: 7
- `GTG`: 5
- `TG`: 6
- `TGTG`: 4
- TTGTG: 3
- TTTGTG: 2

$\Rightarrow$

- 1
- 7
- 5
- 6
- 4
- 3
- 2

suffix sorting

Text:

```
1234567
ATTTGTG
```

Index:

1,7,5,6,4,3,2

To search a pattern: binary search in the list of positions, and jump in the text to look up characters.

suffix arrays

This list of text positions takes the name of **suffix array**.

Manber, Udi, and Gene Myers. "Suffix arrays: a new method for on-line string searches." siam Journal on Computing 22.5 (1993): 935-948.

suffix arrays: space

How much space?

We store n numbers in the range $1, ..., n$. For genomes of length $n < 4 \cdot 10^9 =$ 4 Billions of nucleotides, each such number requires $\log_2(4 \cdot 10^9) \approx 32$ bits $= 4$ Bytes.

Space

The suffix array + the original genome take only $5 \cdot n$ Bytes!

On the Human genome, this is approximately 15 GB.

suffix arrays: time

Search procedure

For every read $r_1, r_2, ..., r_t$:

- We do a binary search to find the range: $\log n$ steps
- For each binary search step, we read at most m characters from the text
- After we find the range in the suffix array, we spend time *occ* to read all occurrences

Search time

Finding the *occ* occurrences of t reads of length m in a genome of length n takes approximately $t \cdot m \cdot \log n + occ$ steps.

Example

Suppose we want to search $t = 10^8$ reads of length $m = 100$ in the Human genome ($n \approx 3 \cdot 10^9$), and assume each step takes 1 ns on your computer. Then, $t \cdot m \cdot \log n \approx 320$ seconds. Much better than 9.5 years :)

Exercise

Build the suffix array of the genome `CCTCCTAGAG`

Is this all?

It seems that suffix arrays definitely solve our problem!

This is true until we need to index just one genome ...

Population genomics

Sequencing is becoming cheaper and faster

This means that it is relatively cheap to get the genomic sequence of a single person

The interest now is moving towards **population genomics**: get the genomes of as many individuals as possible, and use them to build a database of known genetic mutations (and their evolution in space/time)

Example: the 1000 genomes project

`http://www.internationalgenome.org/`

Possibly, we would like to **index all these genomes**. Why?

By aligning your DNA sequences on such an index, it would be possible to quickly spot mutations in your DNA that could be linked, e.g. to particular known diseases.

Population genomics

Now the problem should be clear: the suffix array of 1000 Human genomes takes $5 \cdot (3 \cdot 10^9) \cdot 1000 = 1.5 \cdot 10^{13} \approx 15$ TeraBytes!

... and this is just for 1000 people. The world population was estimated to have reached 7.500.000.000 on April 24, 2017. The United Nations estimates it will further increase to 11.2 billion in the year 2100

How can we possibly index all this stuff?

Population genomics

Key observation: any two human genomes are $> 99.9\%$ similar. On average, the number of differences between two random people is just 3 million of DNA bases, and even less within groups of genetically-related people (e.g. populations, families ...)

This suggests that, after storing just one Human genome (3 GB), each new genome only adds 3MB of information.

$\Rightarrow$ 1000 Human genomes could be stored in just 6 GB ...

Next lecture: data compression and compressed text indexes