BC205: Algorithms for Bioinformatics. VI. NGS Mapping

Christoforos Nikolaou

April 25, 2018

In previous chapters

- Sequence alignment
- Fast string matching

The Problem: Even rapid searches are not rapid enough

- A typical NGS experiment yields 10^6 reads of $>$100bp in length
- Most reference genomes lie in the order of $10^9 bp$
- It is highly demanding to perform a rapid search (BLAT-like) with the reference genome

NGS Basics

- Input: A sample that can be converted to DNA through sequencing
- Applications:
  - De-multiplexing (if the sample is multiplexed)
  - Quality Control
  - Mapping
  - Downstream Analyses
- Output: Millions of reads in the form of fastq files

The “online” approach

- Take the search pattern -> transform it -> search for it
- Transforming means breaking it into k-mers, creating an index etc
- The problem when the text is very long and the patterns too many is similar to having cars travel a highway, only you ask that *only one car* uses the highway from the beginning to the end of the road.
- We need a strategy that will allow multiple cars/patterns to traverse the highway/text simultaneously

The offline approach

- Index the **text**
- We can try to store all the text in a (transformed) object that will be parsed more easily
- (Make the highway accessible to everyone)

Suffix Tries

- Suffix Tries are an efficient way to create a rapidly "searchable" representation of a long sequence
- Imagine the sequence $S = GAGTAAGTCA$. In order to create its suffixes (as dictated by the "suffix") we simply append a \$ sign at the end of each possible suffix of S

Suffixes
GAGTAAGTCA$
AGTAAGTCA$
GTAAGTCA$
TAAGTCA$
AAGTCA$
AGTCA$
GTCA$
TCA$
CA$
A$

Suffix Tries

- Suffixes ordered alphabetically

Suffixes
A\$
AAGTCA\$
AGTAAGTCA\$
AGTCA\$
CA\$
GAGTAAGTCA\$
GTAAGTCA\$
GTCA\$
TAAGTCA\$
TCA\$

Suffix Tries

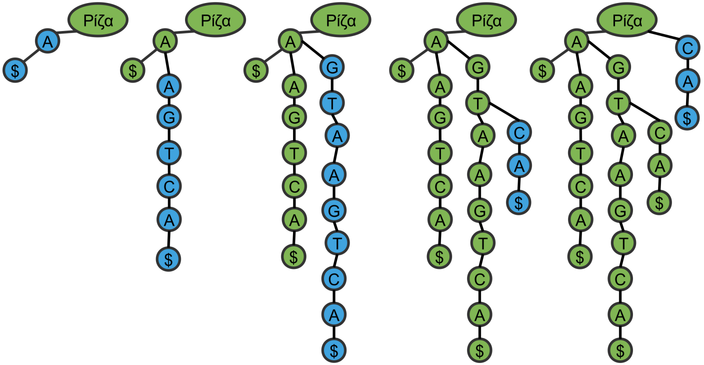
- Lexicographically ordered list keeping the position in the original sequence

Suffixes
A$ [9]
AAGTCA$ [4]
AGTAAGTCA$ [1]
AGTCA$ [5]
CA$ [8]
GAGTAAGTCA$ [0]
GTAAGTCA$ [2]
GTCA$ [6]
TAAGTCA$ [3]
TCA$ [7]

Building a Suffix Trie

- A Suffix Trie looks like a tree.
- You start with a root
- Add each suffix to the root given that its prefix doesn't already exist or
- Add each suffix to the already existing prefix if it does

Building a Suffix Trie



Building a Suffix Trie: Pseudocode

First create the suffixes

```
seq="GAGTAAGTCA"
suffixes=[]
for i in range(len(seq)):
  suffixes[i]=seq[:i]+"$"+i
suffixes.sort
```

Suffixes
A$ [9]
AAGTCA$ [4]
AGTAAGTCA$ [1]
AGTCA$ [5]
CA$ [8]
...

Building a Suffix Trie: Pseudocode II

- And now build the Trie

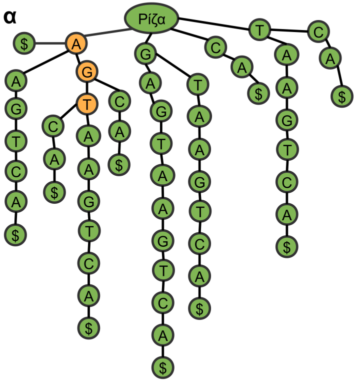
```
trie["R"]=()
# loop over suffixes
for pattern in suffixes:
  innertrie=trie
  # loop over prefixes of each suffix
  for j in range(len(pattern)):
    if pattern[:j] in innertrie.keys():
      # trie becomes sub-trie of the
      # "daughters" of the prefix
      innertrie=innertrie.values()
      break
    else:
      trie[pattern[:j]].append(pattern[j+1:])
return trie
```

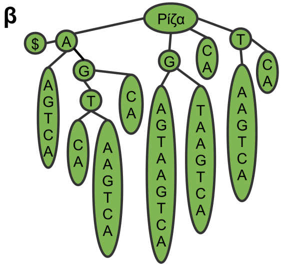
Problems of Suffix Tries

- Basically: Memory
- How many nodes will a suffix trie of a sequence with length l have?
  - in the simplest case of a two-alphabet ordered sequence ("XXXYYY"=$X(n)Y(n)) the trie will constitute of:
    1. a series of n Y nodes
    2. n branches of $n+1$ nodes
       thus a total of $n(n+1) = n^2$ nodes $O(n^2)$

Suffix Trees

- Building a Suffix Trie requires $O(l^2)$ for a sequence of length l and l^2 memory.
- We can reduce memory requirements by constructing a slightly different object called a **Suffix Tree**.
- Suffix Trees are like Tries by combining nodes of non-branching paths into. The number of nodes in a Suffix Tree for a sequence with length s is $2s$ (Can you prove this?)





Using a Suffix Tree/Trie

Basis: Every substring of S is a prefix of some suffix of S

- Decide if a pattern exists in a sequence
- Find a pattern in a sequence
- Find how many times a pattern occurs in a sequence
- Find the position(s) of a patten in a sequence
- Find the **longest repeat** in a sequence
- Find the **longest common pattern** between Try this: `https://visualgo.net/en/suffixtree`

Search a Suffix Tree: Pseudocode

```
consider current node
for (i in current node child)
  if (i not match)
    continue
  if (i is full match) return all results
  else if (i is partial match) go deeper
return no match
```

How much time is needed to search a pattern with length p in a Suffix Tree?

Suffix Arrays

- Even Suffix Trees have huge memory requirements as the size of sequence gets bigger (remember the size of the human genome is ∼3Gbp)
- A memory-efficient alternative was given with the introduction of **Suffix Arrays**
- **Suffix Arrays** are arrays holding the *starting positions* of the lexicographically ordered list of all suffixes of a sequence S

Constructing a Suffix Array

- For our sequence $S = GAGTAAGTCA$ and its lexicographically ordered suffixes (adding one more suffix for the empty $ sign)

Suffixes
A$ [9]
AAGTCA$ [4]
AGTAAGTCA$ [1]
AGTCA$ [5]
CA$ [8]
...

- The **Suffix Array** is:
  $SuffixArray(GAGTAAGTCA) = [9, 4, 1, 5, 8, 0, 2, 6, 3, 7]$

Suffix Array from a Suffix Trie

- Use recursion to go through each node of the tree, order its daughter nodes lexicographically and extract the values (offsets)

```
def preorder(tree,node)
for node in tree:
  subtree=preorder(tree[node])
```

and backwards (Suffix Trie from a Suffix Array)

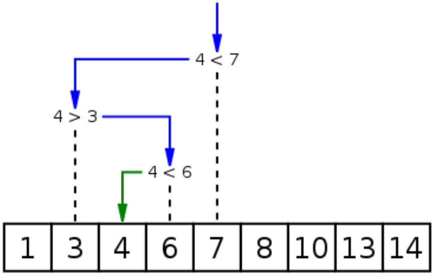
- This is tricky. Can you think about it? (at home)

Searching through a Suffix Array

- Searching through a Suffix Array is even faster than with a Suffix Tree.
- Since all suffixes are now ordered lexicographically, any Suffix that matches the pattern will be "clustered" together with similar other suffixes.
- The question thus becomes "how can we find the position in the **Suffix Array**?"
- Easy: Use *Binary Search*

  1. Take the pattern P and the **Suffix Array** of length S
  2. Take the length of S and divide by 2 ($mid = s/2$)
  3. Compare the pattern P with $S[mid]$:
  4. Ask if P is lexicographically $> mid$: make new $mid = s - mid/2$ or lexicographically $< mid$: make new $mid = s - mid/2$;
  5. Repeat until you *hit* it.

Binary Search through a Suffix Array



- You actually need to implement this process twice to find the lower-most and the higher-most index
- All indices inbetween are matches of your pattern in sequences S, plus they are ordered lexicographically
- Think of the implementation as an exercise

Suffix Array: Memory and Search Complexity

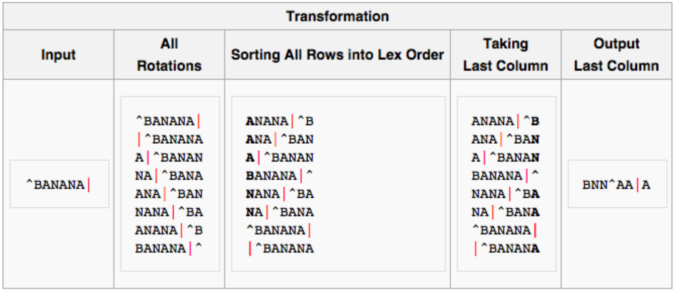
- What is the size of a Suffix Array for a sequence S?
  - In memory it is similar to a Suffix Tree ($2s$)
- What is the time needed for search of a pattern with length p?
- What is the search time dependent on in Suffix Trees and in Suffix Arrays?

Genome Compression

- Longer sequences (e.g. complete chromosomes) require an even more memory-efficient approach. We need ways to create a searchable object in linear time and with linear memory requirement.
- How can we compress the sequence in length without losing the information about the order of its residues.

Burrows-Wheeler Transform BWT

- A memory and time efficient transformation of strings that allows:
  - Compression
  - Rapid Searches



Transformation				
Input	All Rotations	Sorting All Rows into Lex Order	Taking Last Column	Output Last Column
^BANANA|	^BANANA| |^BANANA A|^BANAN NA|^BANA ANA|^BAN NANA|^BA ANANA|^B BANANA|^	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA	BNN^AA|A

Construction of the BWT

- Given a Sequence S
  1. Take all cyclic rotations of S
  2. Order them lexicographically
  3. Take the last column of the ordered Cyclic Rotation Table
- The BWT is at the same time a means of effective compression of the genome and a searchable object. It can therefore be:
  - used to compress the genome
  - uncompressed in the full original sequence
  - used for rapid searches

Burrows-Wheeler Transform

- For our sequence $S = GAGTAAGTCA$ we can construct the Cyclic Rotations Table (CRT) like this:

Cyclic Rotations
GAGTAAGTCA$
$GAGTAAGTCA
A$GAGTAAGTC
CA$GAGTAAGT
TCA$GAGTAAG
GTCA$GAGTAA
AGTCA$GAGTA
AAGTCA$GAGT
TAAGTCA$GAG
GTAAGTCA$GA
AGTAAGTCA$G

Burrows-Wheeler Transform

- The BWT is the last column in a lexicographically ordered list of Cyclic Rotations

Ordered Cyclic Rotations	BWT
$GAGTAAGTCA	A
A$GAGTAAGTC	C
AAGTCA$GAGT	T
AGTAAGTCA$G	G
AGTCA$GAGTA	A
CA$GAGTAAGT	T
GAGTAAGTCA$	$
GTAAGTCA$GA	A
GTCA$GAGTAA	A
TAAGTCA$GAG	G
TCA$GAGTAAG	G

BWT Properties

- Each column of the CR-Table contains some permutation of the sequence S.
- The first column is just the residues of S ordered lexicographically
- What is particular about the last one?
  - Given that CRT is ordered then repeated substrings in S such as “AGT” will cluster together in the first columns.
  - For the same reason some parts of it like “GT” will also cluster together in the first and leave “A” in the last
  - The last column, the BWT, is an efficient way to convert the sequence’s repeats to “runs”. It can be thus seen as an efficient way to compress the sequence with run-length encoding. Our sequence becomes $ACTGA\$(AG)_2$, whic is two residues less than the original.

Inverting the BWT

- If the BWT is an efficient sequence compression scheme we should be able to reconstitute the original sequence from its BWT. We call this “BWT inversion”. It is done in the following way.

Inverting the BWT #1

Apart from the last column of the CRT (the BWT) we also know that the first is ordered lexicographically. We also know that the CRT has *length*$(S) = s$ columns

First		BWT
$	?????????	A
A	?????????	C
A	?????????	T
A	?????????	G
A	?????????	A
C	?????????	T
G	?????????	**$**
G	?????????	A
G	?????????	A
T	?????????	G
T	?????????	G

Inverting the BWT #2

How can we start filling the "?" columns? We know that the $ sign occurs only once at the end of the sequence, so the first residue in the line that ends with $ should be the first in the sequence (and thus follow "$" in the CRT)

First		BWT
$	**G**????????	A
A	?????????	C
A	?????????	T
A	?????????	G
A	?????????	A
C	?????????	T
G	?????????	$
G	?????????	A
G	?????????	A
T	?????????	G
T	?????????	G

Inverting the BWT #3

In the same way we know that the residue following G should be first in a line that ends with G. The problem is there are three such lines and the possibilities are: A, T or T(again). Which one should we choose?

First		BWT
$	**G**????????	A
A	?????????	C
A	?????????	T
A	?????????	**G**
A	?????????	A
C	?????????	T
G	?????????	$
G	?????????	A
G	?????????	A
T	?????????	**G**
T	?????????	**G**

Inverting the BWT #3

We will take advantage of a basic principle of the BWT, that the repeated elements in the BWT (last column) appear ***in the order they are found in the sequence***. We thus choose "A" to place next to G as this is the first G in our sequence

First		BWT
$	GA???????	A
A	?????????	C
A	?????????	T
A	?????????	**G1**
A	?????????	A
C	?????????	T
G	?????????	$
G	?????????	A
G	?????????	A
T	?????????	**G2**
T	?????????	**G3**

Inverting the BWT #4

Next, we will take advantage of another basic principle of the BWT, that the repeated elements in the first and the BWT (last column) will ***always appear in the same order***. Our G was actually the first G (G1) and so we choose “A” to place next to our G. We also know that this is the 3rd A in the first column.

First		BWT
$	GA???????	A1
A	?????????	C1
A	?????????	T1
A3	?????????	G1
A	?????????	A2
C	?????????	T2
G	?????????	$
G	?????????	A3
G	?????????	A4
T	?????????	G2
T	?????????	G3

Inverting the BWT #5

Carrying on in the same way, we look for the residue in the first column in the row that ends with the 3rd A. That is G again and it is the 2nd G. G2 is in its turn the last in a row that starts with T and so on . . .

First		BWT
$	GAGT?????	A1
A	?????????	C1
A	?????????	T1
A3	?????????	G1
A	?????????	A2
C	?????????	T2
G1	?????????	$
G2	?????????	A3
G	?????????	A4
T1	?????????	G2
T	?????????	G3

Inverting the BWT #6

Following the same way we reconstruct the first row of the CRT. But that is all we need as it contains the sequence (if we append the BWT[1] which is always the last residue of the sequence by default)

First		BWT
$	GAGTAAGTC	**A1**
A1	?????????	C1
A2	?????????	T1
A3	?????????	G1
A4	?????????	A2
C1	?????????	T2
G1	?????????	$
G2	?????????	A3
G3	?????????	A4
T1	?????????	G2
T2	?????????	G3

Try this at home!

- Write your own code to create the BWT of a given message and have a friend decode the message by writing the code of the BWT inversion

Searching through the BWT

- So how can we use the BWT to search for a pattern? What we need is the BWT, from which we can directly extract the first column and the **Suffix Array**
- How we get the Suffix Array from the BWT? (This is left as an exercise for you).

No	First		BWT	SuffixArray
1	$	GAGTAAGTC	A1	10
2	A1	$GAGTAAGT	C1	9
3	A2	AGTCA$GAG	T1	4
4	A3	GTAAGTCA$	G1	1
5	A4	GTCA$GAGT	A2	5
6	C1	A$GAGTAAG	T2	8
7	G1	AGTAAGTCA	$	0
...	...	....	...	....

Searching through the BWT

- Lets assume we want to locate the pattern AGT in the sequence. We can do this by applying a strategy that reminds us of the inversion of the BWT.
  1. Start with the last (i=3) element of *pattern* and locate *pattern*[*i*] **in the first column**. The result is No=10,11 (T1,T2). (Keep in mind that residues are clustered in consecutive rows).
  2. Decrement i (i=3-1=2). In the located rows 10,11 look for *pattern*[*i*] **in the BWT column**. For rows 10,11 "G" is found in both.
  3. Locate the rows in which the BWT residues (G2,G3) appear in the **first column**. These are 8,9.
  4. Decrement i (i=2-1=1) and repeat steps 1-2 until the pattern is complete.

Searching through the BWT

- We finally end up in rows 4,5 which contain A (A3,A4) in the first row. This means AGT is found twice in our sequence.
- But where? We can get this from the suffix array. The positions are 1 and 5. Indeed:

0	**1**	2	3	4	**5**	6	7	8	9	10
G	**A**	**G**	**T**	A	**A**	**G**	**T**	C	A	$

History

- Boyer Moore was used for the very first searches in sequences (mostly protein)
- As sequences accumulated heuristic approaches such as BLAST became prominent but the advent of fully sequenced genomes created bottlenecks even for BLAST
- Modified BLAST approaches such as BLAT are still widely used for remapping a limited amount of sequences (e.g. ESTs) to a known (reference) genome and to compare sequences from closely related species (e.g. to find orthologs between human and mouse)
- NGS rendered even BLAT unusable. The first NGS mapping approaches combined a k-mer hashing approach with BLAST-like seeding (e.g. ELAND)
- The next-generation of aligners (Bowtie, BWA etc) use suffix trees and BWT

Questions/Exercises

- How many leaves will a suffix tree have for a given sequence S?
- Think of a way to speed up the construction of a **Suffix Trie** by reordering the suffixes in another way. What would that way be?
- Write the code to construct a **Suffix Tree**
- Use a **Suffix Trie** (or **Tree**) to find the longest palindrome (e.g. "AATTAA") in a sequence
- Write the code to create a BWT from a sequence S.
- Can you construct the BWT of a sequence S given its Suffix Array?
- There is a direct connection between the BWT and the Suffix Array (can you spot it?)