Next Generation Sequencing

E.M. Bakker

Several slides are based on/taken from [7].

Overview

- Introduction
- Next Generation Technologies
- The Mapping Problem
  - The MAQ Algorithm
  - The Bowtie Algorithm
  - Burrows-Wheeler Transform
- Sequence Assembly Problem
  - De Bruijn Graphs
  - Other Assembly Algorithms

Introduction

1953 Watson and Crick: the structure of the DNA molecule

⇒ DNA carrier of the genetic information, the challenge of reading the DNA sequence became central to biological research.

⇒ methods for DNA sequencing were extremely inefficient, laborious and costly.

1965 Holley: reliably sequencing the yeast gene for $tRNA^{Ala}$ required the equivalent of a full year's work per person per base pair (bp) sequenced (1bp/person year).

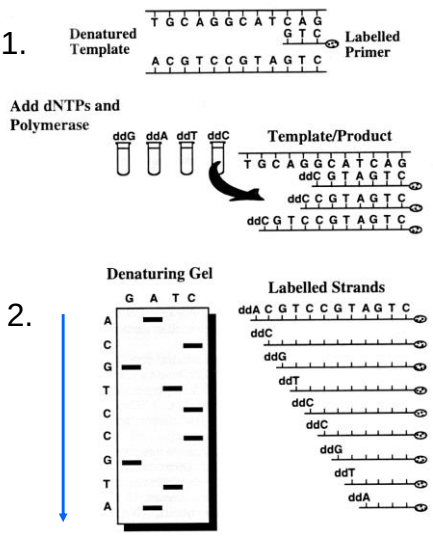
1970 Two classical methods for sequencing DNA fragments by Sanger and Gilbert.

Sanger sequencing (sketch):

1. The polymerase extends the labeled primer, randomly by either:
   - a normal dCTP base, or
   - a modified ddCTP base

   At every position where a ddCTP is inserted, polymerization terminates ! => a mix of fragments, where the length of each fragment is a function of the relative distance from the modified base to the primer.

2. In 4 lanes electrophoretic separation of the mix of fragments in each of the four tubes (ddG, ddA, ddT, and ddC) is established. Now the bands on the gel accumulate equal length fragments representing the Labelled Strands shown to the right. => This allows us to read the complement of the original template from bottom to top.



Introduction

1980 In the 1980s methods were augmented by
- partial automation
- the cloning method, which allowed fast and exponential replication of a DNA fragment.

1990 Start of the human genome project: sequencing efficiency reached 200,000 bp/person-year.

2002 End of the human genome project: 50,000,000 bp/person-year. (Total cost summed to $3 billion.)

Note: Moore's Law doubling transistor count every 2 years
Here: doubling of #base pairs/person-year every 1.5 years

Introduction

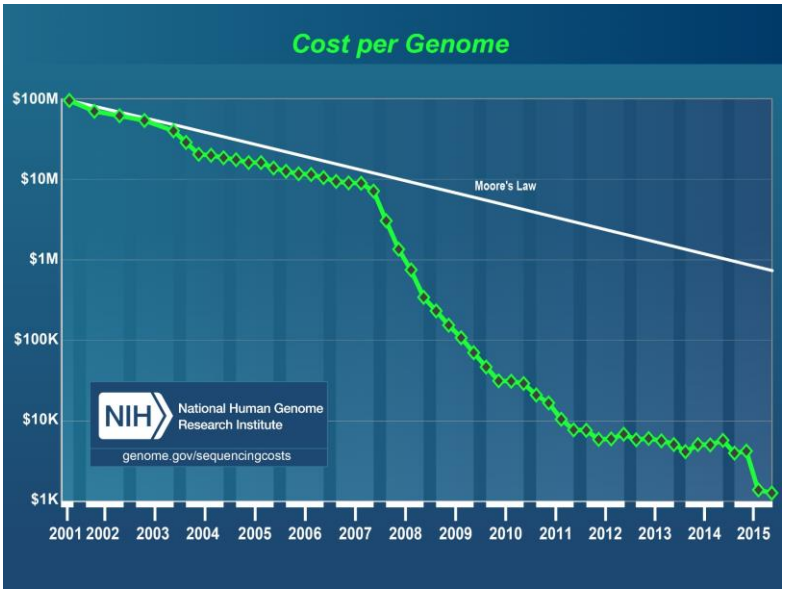
Recently (2007 – 2011/2012):
- new sequencing technologies next generation sequencing (NGS) or deep sequencing
  - reliable sequencing of $100x10^9$ bp/person-year.
  - NGS now allows compiling the full DNA sequence of a person for ~ $10,000
  - within 3-5 years cost is expected to drop to ~$1000.

2017: Veritas Genetics uses a Illumina HiSeq X Ten system with an average coverage depth of 30X. Whole genome sequencing in 8-10 weeks at a price of $999.
(See http://www.nanalyze.com/2016/03/does-full-genome-sequencing-really-cost-1000-now/)

2018: HiSeq's successor NovaSeq 6000 6Tb <2 days (150b reads)

2019: NextSeq 500 Whole sequence run in 26h, 2 days shorter than NovaSeq 6000

100$ Genome scan soon!



From: https://www.genome.gov/sequencingcosts/

Next Generation Sequencing Technologies

Next Generation Sequencing technology of **Illumina**, one of the leading companies in the field:

- DNA is replicated => millions of copies.
- All DNA copies are randomly shredded into various fragments using restriction enzymes or mechanical means => fragments form the input for the sequencing phase.
- Hundreds of copies of each fragment are generated/assembled at one spot (cluster) on the surface of a huge matrix.
- Initially it is not known which fragment sits where.

Note: There are quite a few other technologies available and/or currently under development.

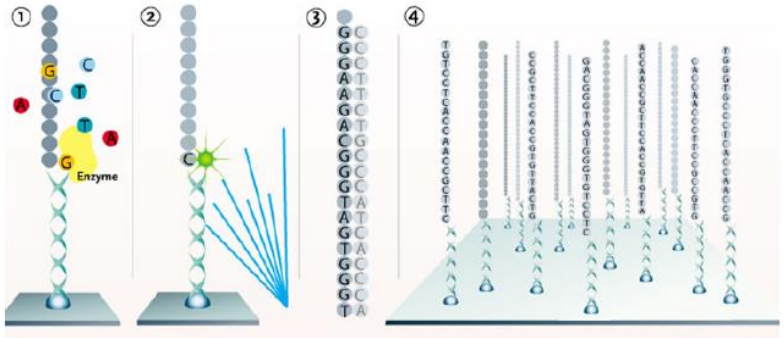
Next Generation Sequencing technology of Illumina (continued):

- A DNA replication enzyme is added to the matrix along with 4 slightly modified nucleotides.
- The 4 nucleotides are slightly modified chemically so that:
  - each would emit a unique color when excited by laser,
  - each one would terminate the replication.
  - hence, the growing complementary DNA strand on each fragment in each cluster is extended by one nucleotide at a time.

Next Generation Sequencing technology of Illumina (continued):

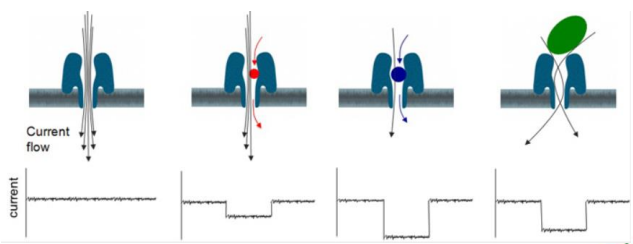
- A laser is used to obtain a read of the actual nucleotide that is added in each cluster. (The multiple copies in a cluster provide an amplified signal.)
- The solution is washed away together with the chemical modification on the last nucleotide which prevented further elongation and emitted the unique signal.
- Millions of fragments are sequenced efficiently and in parallel.
- The resulting fragment sequences are called reads.
- **Reads form the input for further sequence mapping and assembly.**

Next Generation Sequencing Technologies

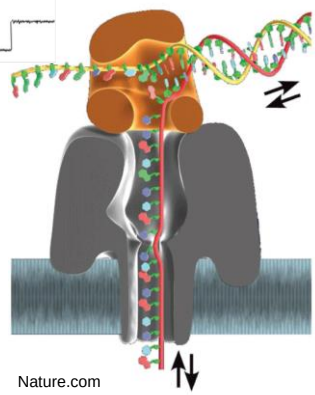


(1) A modified nucleotide is added to the complementary DNA strand by a DNA polymerase enzyme.
(2) A laser is used to obtain a read of the nucleotide just added.
(3) The full sequence of a fragment thus determined through successive iterations of the process.
(4) A visualization of the matrix where fragment clusters are attached to flow cells.

MinION: nanopores



Phys.org



Nature.com

- Read length ~10kb
- Longest reported230 – 300 kb
- 500 bps per pore

The Mapping Problem and the Assembly Problem

The Mapping Problem

INPUT: m reads $S_1,\ldots, S_m$ of length l and an approximate reference genome R.

QUESTION: What are the positions $x_1,\ldots, x_m$ along R where each read $S_1,\ldots, S_m$ matches, respectively?

The Assembly Problem

INPUT: m reads $S_1,\ldots, S_m$ of length l.

QUESTION: What is the sequence of the full genome?

The crucial difference between the problems of mapping and assembly is that in case of assembly **we do not have a reference genome**, and we must assemble the full sequence directly from the reads.

The Mapping Problem

The Short Read Mapping Problem on reference genome R

INPUT: m reads $S_1,\ldots, S_m$ of length l and an approximate reference genome R.

QUESTION: What are the positions $x_1,\ldots, x_m$ along R where each read $S_1,\ldots, S_m$ matches, respectively?

For example: after sequencing the genome of a person we want to map it to an **existing sequence of the human genome.**

- The new sample will not be 100% identical to the reference genome:
  - natural variation in the population
  - mismatches or gaps
  - sequencing errors
  - repetitive regions

  Humans are diploid organisms
  - different alleles on the maternal and paternal chromosomes
  - two slightly different reads mapping to the same location (some with mismatches)

Solutions for the Mapping Problem

Naive algorithm
- for each S_i scan the reference string R
- match the read at each position p and
- pick the best match

Time complexity: $O(m \cdot l \cdot |R|)$ for exact or inexact matching.
m number of reads, l read length

Considering the parameters for our problem ($m \sim 10^8$, $l \sim 10^2$, $|R| \sim 3 \cdot 10^9$) , this is impractical.

Less naive solution
- use the Knuth-Morris-Pratt algorithm to match each S_i to R

Time complexity: $O(m \cdot (l+|R|)) = O(ml + m|R|)$ for **exact matching**.

A substantial improvement but still not enough…

KMP Preprocessed P example

Pattern P ϵ {a,b,…}*:

Automaton:

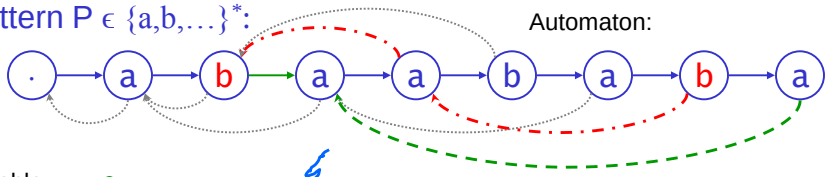


Table:

1	2	3	4	5	6	7	8
a	b	a	a	b	a	b	a
0	1	1	2	2	3	4	3

failure links (suffix = prefix):

top: as 'automaton'

bottom: as table

- how to determine?
- how to use?

```
                  6
P: a b a a b a b a
P:       a b a a b a b a
                  3
                    7
P: a b a a b a b a
P:       a b a a b a b a
                  4
                    8
P: a b a a b a b a
P:             a b a a b ...
                  3
```

NB See slide 67 and further (is part of the exam materials).

The Mapping Problem: Suffix Trees

1. Build a suffix tree for R.
2. For each S_i *(i in {1,…,m})*find matches by traversing the tree from the root.

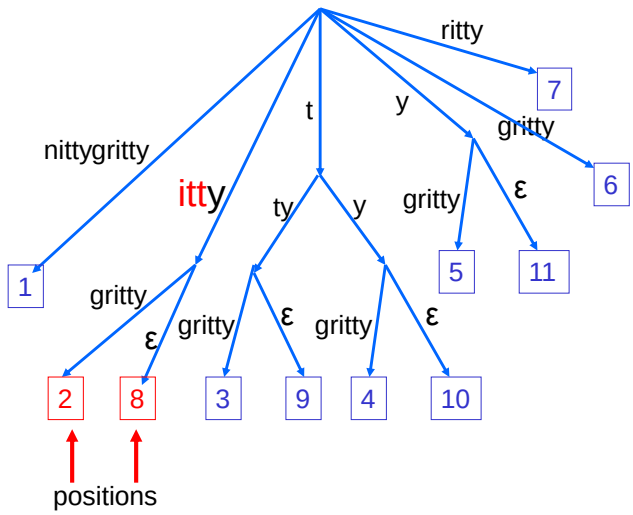
Time complexity: O(ml+|R|), assuming the tree is built using Ukkonen's linear-time algorithm.

Notes:

- Time complexity is practical.
- Only build the tree for the reference genome once.
- The tree can be saved and used repeatedly.

Suffix Tree example: find ‘itt’ in ‘nittygritty’

nittygritty	1
ittygritty	2
ttygritty	3
tygritty	4
ygritty	5
gritty	6
ritty	7
itty	8
tty	9
ty	10
y	11



The Mapping Problem: Suffix Trees

Space complexity may be an obstacle:

- The leafs of the suffix tree also hold the indices where the suffixes begin
  => the tree requires $O(|R| \cdot \log|R|)$ bits just for coding of the indices.
- The constants are large due to the additional layers of information required for the tree (such as suffix links, etc.).
- The original human genome reference sequence demands just $|R|\log(\#symbols)$ bits
  => we can store the human genome using ~750MB
  But! ~64GB for the Suffix Tree (2016: very doable)!

- Final problem: suffix trees allow only for **exact matching.**

The Mapping Problem: Hashing

- Preprocess the reference genome R into a hash table H.
- The keys of the hash are all the substrings of length l in R:
  - Table H contains: the position p in R where the substring ends.
  - Matching: given S_i the algorithm returns $H(S_i)$.

Time complexity: $O(m \cdot l + l \cdot |R|)$

Note:

- The space complexity is $O(l \cdot |R| + |R| \cdot \log|R|)$ since we must also hold the binary representation of each substring's position. (Still quite high.)
- Represent each nucleotide as a 2-bit code: factor of 4 reduction.
- Practical solution: Partition the genome into several chunks.

- Again, only exact matching allowed.

The Mapping Problem: The MAQ Algorithm

The MAQ (Mapping and Alignment with Qualities) algorithm by Li, Ruan and Durbin, 2008 [5]

- An efficient solution to the short read mapping problem
- Allows for inexact matching, up to a predetermined maximum number of mismatches.
- Memory efficient.
- Measures for base and read quality, and for identifying sequence variants.

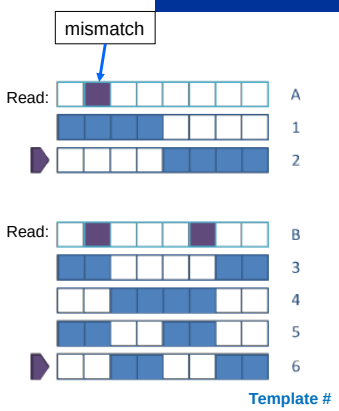
The MAQ Algorithm

A key insight:

- If a read in its correct position to the reference genome has one mismatch, then partitioning that read into two makes sure one part is still exactly matched.

- Hence, if we perform an exact match twice using only a subset of the read bases (a template), one of the subsets is enough to find the match.
- For 2 mismatches, 6 templates:
  - some templates are noncontiguous
  - each template covers half the read
  - Each template has a partner template that complements it to form the full read
  - If the read has no more than two mismatches, at least one template will not be affected by any mismatch.
- For 3-mismatches, 20 templates guarantee at least one fully matched template.
- Etc.

The MAQ Algorithm: Templates



A and B are reads, where purple boxes indicate mismatches with respect to the reference.

The numbered templates have blue boxes in the positions they cover.

1 mismatch:

- comparing read A through template 1 => no full match
- template 2 fully matched

2 mismatches:

- read B is fully matched with template 6
- Any combination of up to two mismatched positions will be avoided by at least one of the 6 templates.

Note: 6 templates guarantee 57% of all 3 mismatches will have at least one fully matching template.

The MAQ Algorithm

The algorithm:

- Generate the number of templates required to guarantee at least one full match for the desired maximal number of mismatches, and
- For each read:
  - match the read against the templates
  - use the exact matching template as a seed for extending across the full read.

- Identifying the exact matches is done by
  - hashing the **read templates** into template-specific hash tables, and
  - scanning the reference genome R against each table.

Note:

- It is not necessary to generate templates and hash tables for the full read, since the initial seed will undergo extension, e.g. using the Smith-Waterman algorithm.
- Therefore the algorithm initially processes only **the first 28 bases of each read.** (The first bases are the most accurate bases of the reads.)

The MAQ Algorithm

Algorithm (for the case of up to two mismatches):

1. Index the first 28 bases of each read with complementary templates 1, and 2, thereby generating hash tables H1 and H2, respectively.
2. Scan the reference genome R:
   - for each position and each orientation, query a 28-bp window through templates 1, 2 against the appropriate tables H1, H2, respectively.
   - If hit, extend and score the complete read based on mismatches.
   - For each read, keep the two best scoring hits and the number of mismatches therein.
3. Repeat steps 1+2 with complementary templates 3, 4, then 5, 6.

Remark:

The reason for indexing against a pair of complementary templates each time has to do with a feature of the algorithm regarding paired-end reads (see original paper for details).

The MAQ Algorithm Complexity

Time Complexity:

- $O(m \cdot l)$ for generating the hash tables in Step 1, and
- $O(l \cdot |R|)$ for scanning the genome in Step 2.
- Repeating Steps 1+2 three times in this implementation has no effect on the asymptotic complexity.

Space Complexity:

- $O(m \cdot l)$ for holding the hash tables in Step 1, and
- $O(m \cdot l + |R|)$ total space for Step 2, but only $O(m \cdot l)$ space in main memory at any one time.

The MAQ Algorithm Complexity

- Note that, not the full read length l is used, but a window of length $l' <= l$ ($l' = 28$-bp in the implementation presented above).
- The size of each key in a template hash table is only $l'/2$, since each template covers half the window.

=> Time and space complexity of step 1 is: $O(2\, m\, l'/2) = O(ml')$.

This space reduction makes running the algorithm on a PC very feasible.

The time and space required for extending a match in Step 2 using the Smith-Waterman algorithm, where p *the probability of a hit* (using the l' length window):

- The time complexity : $O(l' |R| + p|R|l^2)$.
- The space complexity: $O(ml' + l^2)$.

Note: The value of p is small, and decreases drastically the longer l' is, since most (other) l' -long windows in the reference genome will likely not capture the exact coordinates of a true read.

Read Mapping Qualities

The MAQ algorithm provides a measure of the confidence level for the mapping of each read, denoted by QS and defined as:

$$QS = -10 \log_{10}(Pr\{\text{read S is wrongly mapped}\})$$

For example:

QS = 30 if the probability of incorrect mapping of read S is 1/1000.

This confidence measure is called phred-scaled quality, similar to the scaling scheme originally introduced by Phil Green et al. [3] for the human genome project.

Example: Base calling with Phred Quality Score = 30

=> Probability of incorrect base call is 1/1000.

The Bowtie Algorithm

Another way to **map reads to a reference genome** is given by the Bowtie algorithm, presented in 2009 by Langmead et al.[1].

It solves the mapping problem using a space-efficient indexing scheme.

The indexing scheme used is called the Burrows-Wheeler Transform [2] and was originally developed for data compression purposes.

The Burrows-Wheeler Transform

Applying the Burrows-Wheeler transform BW(T) to the text
T = "the next text that i index.":

1. First, we generate all cyclic shifts of T.
2. Next, we sort these shifts lexicographically.
   - define the character '.' as the minimum and we assume that it appears exactly once, as the last symbol in the text.
   - followed lexicographically by ' ‘ (space)
   - followed by the English letters according to their natural ordering.
   - Call the resulting matrix M.

The transform BW(T) is defined as the sequence of the last characters in the rows of M.

Note that, the last column is a permutation of all characters in the text since each character appears in the last position in exactly one cyclic shift.

Burrows-Wheeler Transform

.the_next_text_that_i_index
_i_index.the_next_text_that
_index.the_next_text_that_i
_next_text_that_i_index.the
_text_that_i_index.the_next
_that_i_index.the_next_text
at_i_index.the_next_text_th
dex.the_next_text_that_i_in
e_next_text_that_i_index.th
ex.the_next_text_that_i_ind
ext_text_that_i_index.the_n
ext_that_i_index.the_next_t
hat_i_index.the_next_text_t
he_next_text_that_i_index.t

Some of the cyclic shifts of T sorted lexicographically and indexed by the last character.

Burrows-Wheeler Transform

- Storing BW(T) requires the same space as the size of the text T since it is a permutation of T.

BW(the human genome)

Each base {A,C,T,G} represented by 2 bits => storing the permutation requires 2 times $3x10^9$ bits (*instead of ~30 times $3x10^9$ for storing all indices of T*).

Burrows-Wheeler Transform

The following holds for BW(T):

1. # occurrences of char c in T = # occurrences of char c in BW(T) (BW(T) permutation of the T).

2. The first column of the matrix M can be obtained by sorting BW(T) lexicographically.

3. Determine the number of occurrences of the substring 'xt' in T:
   - BW(T) is the last column of the lexicographical sorting of the shifts.
   - **The character at the last position of a row appears in the text T immediately prior to the first character in the same row (each row is a cyclical shift).**
   - => consider the interval of 't' in the first column, and check how many of these rows have an 'x' at the last position.

Burrows-Wheeler Transform

.the_next_text_that_i_index
_i_index.the_next_text_that
_index.the_next_text_that_i
_next_text_that_i_index.the
_text_that_i_index.the_next
_that_i_index.the_next_text
at_i_index.the_next_text_th
dex.the_next_text_that_i_in
e_next_text_that_i_index.th
ex.the_next_text_that_i_ind
ext_text_that_i_index.the_n
ext_that_i_index.the_next_t
hat_i_index.the_next_text_t
he_next_text_that_i_index.t

1. Some of the cyclic shifts of T sorted lexicographically and indexed by the last character.

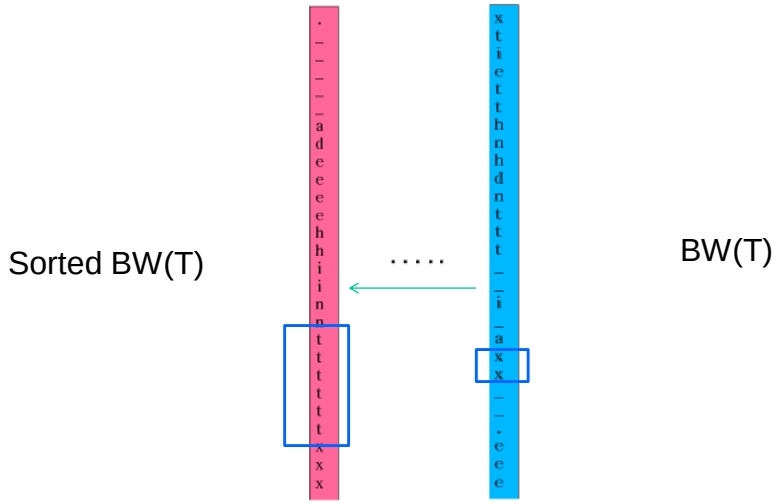
Burrows-Wheeler Transform

The following holds for BW(T):

1. # occurrences of char c in T = # occurrences of char c in BW(T) (BW(T) permutation of the T).

2. The first column of the matrix M can be obtained by sorting BW(T) lexicographically.

3. Determine the number of occurrences of the substring 'xt' in T:
   - BW(T) is the last column of the lexicographical sorting of the shifts.
   - **The character at the last position of a row appears in the text T immediately prior to the first character in the same row (each row is a cyclical shift).**
   - => consider the interval of 't' in the first column, and check how many of these rows have an 'x' at the last position.



2. Recovering the first column (left) by sorting the last column.
3. Determine the number of occurrences of the substring 'xt' in T

Burrows-Wheeler Transform

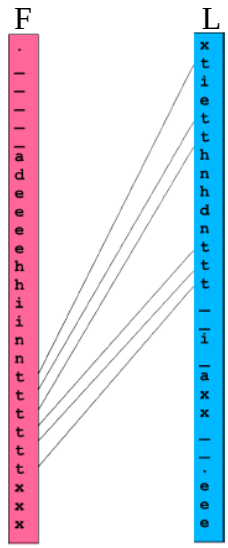
Given BW(T) also the second column can be derived:

- 'xt' appears *twice* in the text, and *three rows* start with an 'x'.
- *Two of the three* must be followed by a 't', where the lexicographical sorting determines which 'x'.
- The third 'x' is followed by a '.' (see first row) => '.' must follow the first 'x' in the first column since '.' is smaller lexicographically than 't'.
- The second and third occurrences of 'x' in the first column are therefore followed by 't'.

Note: We can use the same process to recover the characters at the second column for each interval, and then the third, etc.

t text that i index. the ne**x**
t that i index. the next te**x**
text that i index.the next
that i index. the next text
the next text that i index.

We have actually:



Last-first mapping: Each 't' character in L is linked to its position in F and no crossed links are possible.

The j-th occurrence of character X in L corresponds to the same text character as the j-th occurrence of X in F.

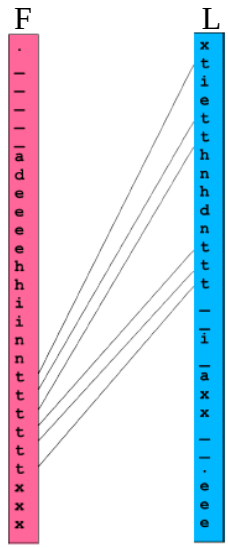
Burrows-Wheeler Transform

The previous two central properties of the BW-transform are captured in the Lemma by Ferragina and Manzini[4]:

Lemma 12.1 (Last-First Mapping):

Let M be the matrix whose rows are all cyclical shifts of T sorted lexicographically, and let L(i) be the character at the last column of row i and F(i) be the first character in that row. Then:

1. In row i of M, L(i) precedes F(i) in the original text: T =…L(i) F(i)…
2. The j-th occurrence of character X in L corresponds to the same text character as the j-th occurrence of X in F.



Last-first mapping: j-th occurrence of character 't' character in L is linked to its j-th position in F and no crossed links are possible.

Burrows-Wheeler Transform

Proof:

1. Follows directly from the fact that each row in M is a cyclical shift.

2. Let X_j denote the j-th occurrence of character X in L, and let α be the character following X_j in the text and β the character following X_{j+1}.

   Then, since X_j appears above X_{j+1} in L, α appears at the beginning of a row above the row that starts with β.

   The rows are lexicographically ordered, hence α must be equal or lexicographically smaller than β.

   Now clearly $X\ \alpha \leq_{\text{lexicographically}} X\ \beta$ holds.

   Hence, as the rows are lexicographically ordered, if character X_j appears in F it is followed by α, and thus will be above X_{j+1} which is followed by β.

   Thus proofing the Lemma.

Reconstructing the Text

Algorithm UNPERMUTE for reconstructing a text T from its Burrows-Wheeler transform BW(T) utilizing Lemma 12.1 [4]:

- assume the actual text T is of length u
- append a unique $ character (='.') at the end, which is the smallest lexicographically

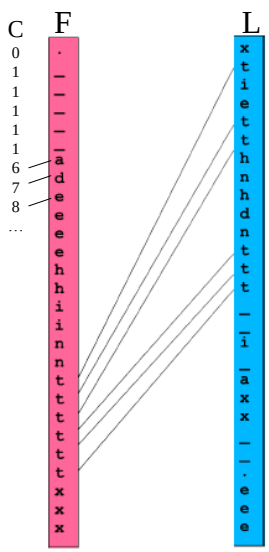
UNPERMUTE[BW(T)]

1. Compute the array C[1,…,|Σ|] : where C(**c**) is the number of characters {$, 1,…,**c**-1} in T, i.e., the number of characters that are lexicographically smaller than **c**
2. Construct the last-first mapping LF, tracing every character in L to its corresponding position in F:
   LF[i] = C(L[i]) + r(L[i], i) + 1, where r(c, i) is the number of occurrences of character **c** in the prefix L[1, i -1]
3. Reconstruct T backwards:

```
s = 1, T(u) = L[1];
for i = u – 1,…, 1
do
  s = LF[s];
  T[i] = L[s];
od;
```

Notice, that C(e) + 1 = 9 is the position of the first occurrence of 'e' in F.

C(e) = 8, as there are 8 characters in T that are smaller than 'e'.



Last-first mapping: Each 't' character in L is linked to its position in F and no crossed links are possible.

Example

T = acaacg$ (u = 6) is transformed to BW(T) = gc$aaac, and we now wish to reconstruct T from BW(T) using UNPERMUTE:

1. Compute array C. For example: C(c) = 4 since there are 4 occurrences of characters smaller than 'c' in T ('$' and 3 occurrences of 'a'). Now C(c) + 1 = 5 is the position of the first occurrence of 'c' in F.
2. Perform the LF mapping. For example, $LF[c_2] = C(c) + r(c,7) + 1 = 6$, and indeed the second occurrence of 'c' in F is at F[6].
3. Determine the last character in T: T(6) = L(1) = 'g'.
4. Iterate backwards over all positions using the LF mapping. For example, to recover the character T(5), we use the LF mapping to trace L(1) to F(7), and then T(5) = L(7) = 'c'.

Remark We do not actually hold F in memory, we only keep the array C defined above, of size | Σ |, which we can easily obtained from L.

r(c,7) = # of occurences of 'c' in length 7-1 prefix, used as an offset to obtain the right 'c'.

Determine its location in F and find its corresponding predecessor in L using C(.) and r(.,.)
i.e., find corresponding 'g' using the last-first mapping

Note: '$' = '.'

Example of running UNPERMUTE to recover the original text. Source: [1].

Exact Matching

EXACTMATCH exact matching of a query string P to T, given BW(T)[4] is similar to UNPERMUTE, we use

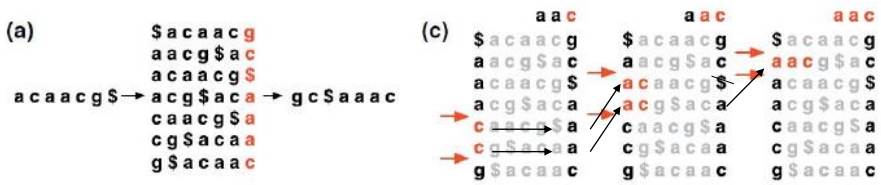
- the same C and r(c, i)
- denote by sp the position of the first row in the interval of rows in M we are currently considering
- denote by ep the position of the first row beyond this interval of rows

=> the interval of rows is defined by the rows sp,…, ep - 1.

EXACTMATCH[P[1,…,p], BW(T)]

1. **c** = P[p]; sp = C[**c**] + 1; ep = C[**c**+1] + 1; i = p - 1;
2. **while** sp < ep **and** i >= 1
   c = P[i];
   sp = C[**c**] + r(**c**, sp) + 1;
   ep = C[**c**] + r(**c**, ep) + 1;
   i = i - 1;
3. **if** (sp == ep) **return** "no match"; else return sp, ep;

P = ‘aac’ ←



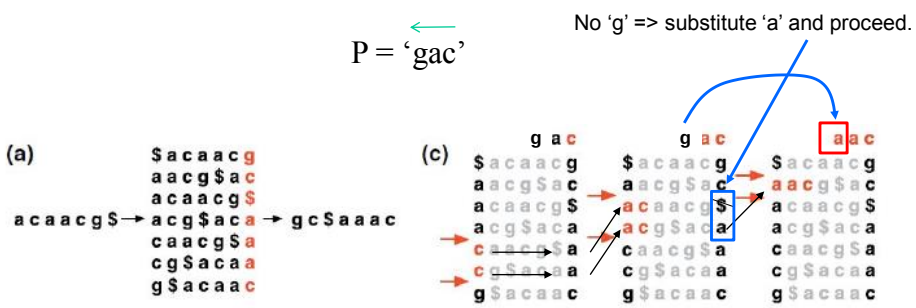
Example of running EXACTMATCH to find a query string in the text [1].

Inexact Matching

Sketch

- Each character in a read has a **numeric quality value**, with lower values indicating a higher likelihood of a sequencing error.
- Similar to EXACTMATCH, calculating matrix intervals for successively longer query suffixes.

- If the range becomes empty (a suffix does not occur in the text), then the algorithm selects an already-matched query position and substitute a different base there, introducing a mismatch into the alignment.
- The EXACTMATCH search resumes from just after the substituted position.
- The algorithm selects only those substitutions that
  - are consistent with the alignment policy and
  - yield a modified suffix that occurs at least once in the text.
  - If there are multiple candidate substitution positions, then the **algorithm greedily selects a position with a maximal quality value.**



Example of running INEXACTMATCH to find a query string in the text [1].

The Assembly Problem

How to assemble an unknown genome based on many highly overlapping short reads from it?

Problem Sequence assembly

INPUT: m l-long reads $S_1,\ldots, S_m$.

QUESTION: What is the sequence of the full genome?

The crucial difference between the problems of mapping and assembly is that now **we do not have a reference genome**, and we must assemble the full sequence directly from the reads.

De Bruijn Graphs

Definition

A k-dimensional de Bruijn graph of n symbols is a directed graph representing overlaps between sequences of symbols.

It has n^k vertices, consisting of all possible k-tuples of the given symbols. Note: the same symbol may appear multiple times in a tuple.

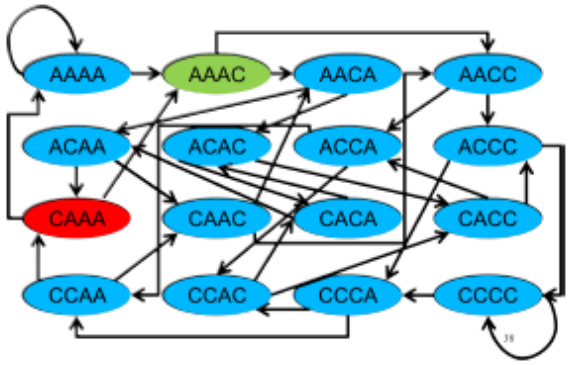
If we have the set of symbols $A = \{a_1,\ldots,a_n\}$ then the set of vertices is:

$V = \{ (a_1,\ldots, a_1, a_1), (a_1,\ldots, a_1, a_2),\ldots, (a_1,\ldots, a_1, a_n),$
$(a_1,\ldots, a_2, a_1), \ldots \quad \ldots, (a_n,\ldots, a_n, a_n)\}$

If a vertice w can be expressed by shifting all symbols of another vertex v by one place to the left and adding a new symbol at the end, then v has a directed edge to w.

Thus the set of directed edges E is:

$E = \{((v_1, v_2,\ldots, v_k), (w_1, w_2,\ldots, w_k)) |\ v_2 = w_1, v_3 = w_2,\ldots, v_k = w_{k-1},$ and w_k new$\}$



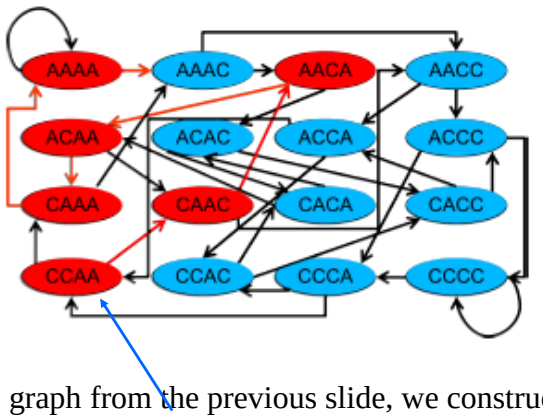
A portion of a 4-dimensional de Bruijn graph.

The vertex CAAA has a directed edge to vertex AAAC, because if we shift the label CAAA to the left and add the symbol C we get the label AAAC.

In this way the word CAAAC defines a directed edge

De Bruijn Graphs

- Given a read, every contiguous (k+1)-long word in it corresponds to an edge in the k-dimensional de Bruijn graph of the symbols {A,C,T,G}.
- Form a subgraph G of the full de Bruijn graph by introducing only the edges that correspond to (k+1)-long words in some read.
- A path in this graph defines a potential subsequence in the genome.
- Hence, we can convert a read to its corresponding path in the constructed subgraph G



Using the graph from the previous slide, we construct the path corresponding to CCAACAAAAC:

- shift the sequence one position to the left each time, and marking the vertex with the label of the 4 first nucleotides.

De Bruijn Graphs

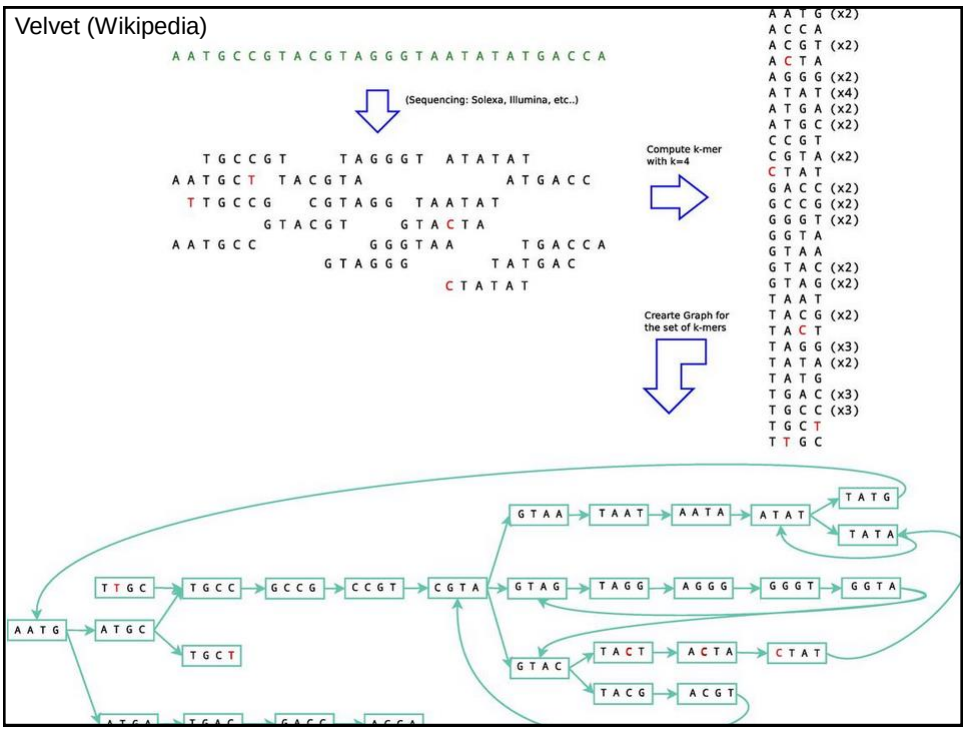
- Form paths for all the reads
- Identify common vertices on different paths
- Merge different read-paths through these common vertices of the paths into one long sequence

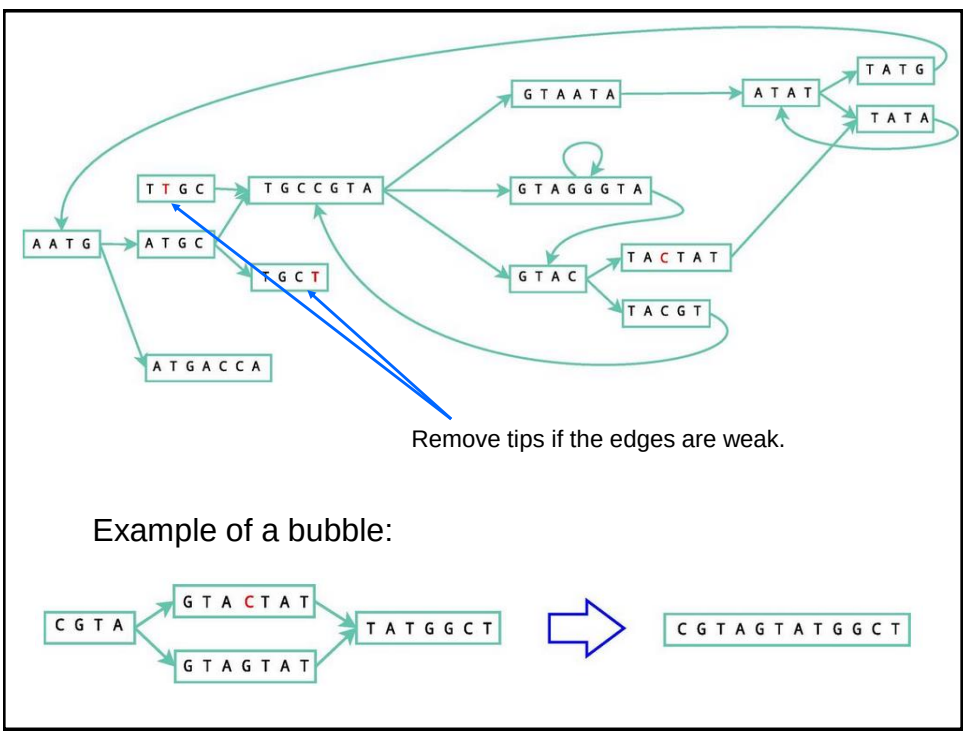
- The two reads in (a) are converted to paths in the graph
- The common vertex TGAG is identified
- Combine these two paths into (b).

De Bruijn Graphs

Velvet (2008), by Zerbino and Birney[6], Velvet GUI (2012) , is an algorithm which uses de Bruijn graphs to assemble reads; with the following difficulties:

- repeats will show up as cycles in the merged path that we form. We do not know how many times each cycle must be traversed in order to form the full sequence of the genome.
- If we have two cycles starting at the same vertex, we cannot tell which one to traverse first.
- Velvet attempts to address this issue by utilizing the extra information we have in the case of paired-ends sequencing (both ends of a DNA fragment are sequenced in Read 1 and Read 2, respectively. The distance between each paired read is known).


Velvet (Wikipedia)
AATGCCGTACGTAGGGTAATATATGACCA
(Sequencing: Solexa, Illumina, etc..)
TGCCGT TAGGGT ATATAT
AATGCT TACGTA ATGACC
TTGCCG CGTAGG TAATAT
GTACGT GTACTA
AATGCC GGGTAA TGACCA
GTAGGG TATGAC
CTATAT
Compute k-mer with k=4
Crearte Graph for the set of k-mers
AATG (x2)
ACCA
ACGT (x2)
ACTA
AGGG (x2)
ATAT (x4)
ATGA (x2)
ATGC (x2)
CCGT
CGTA (x2)
CTAT
GACC (x2)
GCCG (x2)
GGGT (x2)
GGTA
GTAA
GTAC (x2)
GTAG (x2)
TAAT
TACG (x2)
TACT
TAGG (x3)
TATA (x2)
TATG
TGAC (x3)
TGCC (x3)
TGCT
TTGC
AATG
ATGC
TGCT
TTGC
TGCC
GCCG
CCGT
CGTA
GTAA
TAAT
AATA
ATAT
TATG
TATA
GTAG
TAGG
AGGG
GGGT
GGTA
GTAC
TACT
ACTA
CTAT
TACG
ACGT


AATG
ATGC
TGCT
TTGC
TGCCGTA
ATGACCA
GTAATA
ATAT
TATG
TATA
GTAGGGTA
GTAC
TACTAT
TACGT
Remove tips if the edges are weak.
Example of a bubble:
CGTA
GTACTAT
GTAGTAT
TATGGCT
CGTAGTATGGCT

Other Assembly Algorithms

- HMM based
- Majority based
- Etc.

- Long reads: string graphs

Other Assembly Algorithms

K.R. Bradnan et al. Assemblathon 2: evaluating *de novo* methods of genome assembly in three vertebrate species
(http://gigascience.biomedcentral.com/articles/10.1186/2047-217X-2-10, 2013)

"Many current genome assemblers produced useful assemblies, containing a significant representation of their genes and overall genome structure.

However, the high degree of variability between the entries suggests that there is still much room for improvement in the field of genome assembly and that approaches which work well in assembling the genome of one species may not necessarily work well for another."

Other Assembly Algorithms

Salzberg SL, Phillippy AM, Zimin A, Puiu D, Magoc T, Koren S, Treangen TJ, Schatz MC, Delcher AL, Roberts M, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. Genome Res. 2012; 22(3):557–67.

Three conclusions:

1.Quality: data quality, rather than the assembler itself, has a dramatic effect on the quality of an assembled genome

2.Variability: the degree of contiguity of an assembly varies enormously among different assemblers and different genomes

3.Correctness: the correctness of an assembly also varies widely and is not well correlated with statistics on contiguity.

Other Assembly Algorithms

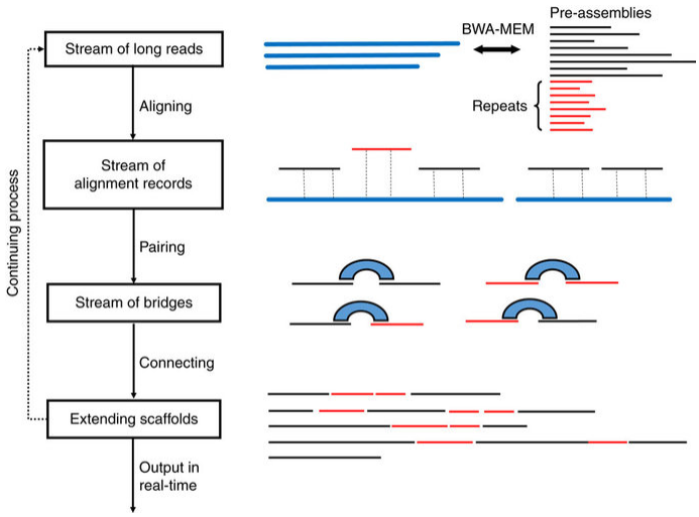
Scaffolding and completing genome assemblies in real-time with nanopore sequencing

By Minh Duc Cao, Son Hoang Nguyen, Devika Ganesamoorthy, Alysha G. Elliott, Matthew A. Cooper & Lachlan J. M. Coin

Nature Communications 8, Article number: 14515 (2017)

"Long read sequencing technologies, for example Pacific Biosciences' (PacBio) and Oxford Nanopore MinION sequencing, allow users to generate reads spanning most repetitive sequences, which can be used to close gaps in fragmented assemblies."

Figure 1: Workflow of the real-time algorithm.



Stream of long reads are aligned to the existing contigs to create alignment records. Bridges connecting contigs are formed, and are used for extending scaffolds. These steps are performed in a streaming fashion.

Image from reference [8].

Bibliography

[1] Ben Langmead, Cole Trapnell, Mihai Pop and Steven L. Salzberg. Ultrafast and memory-effcient alignment of short DNA sequences to the human genome. Genome Biology, 2009.

[2] Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[3] Brent Ewing and Phil Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. Genome Research, 1998.

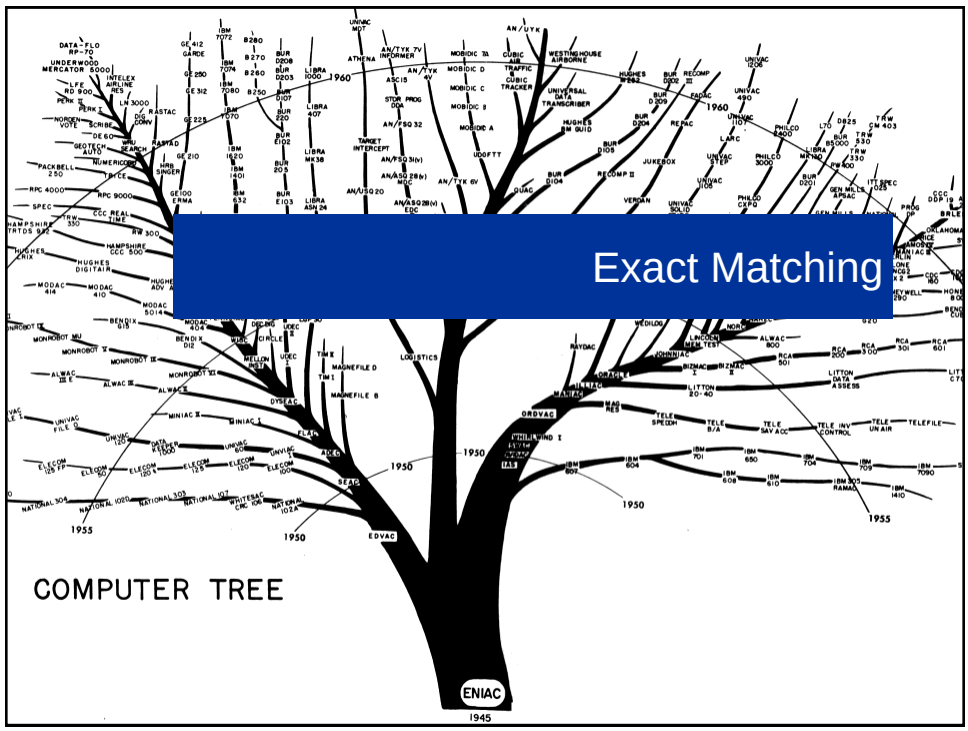
[4] Paulo Ferragina and Giovani Manzini. Opportunistic data structures with applications. FOCS '00 Proceedings of the 41st Annual Symposium on Foundations of Computer Science, 2000.

[5] Heng Li, Jue Ruan and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Research, 2008.

[6] Daniel R. Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Research, 2008.

[7] Ron Shamir, Computational Genomics Fall Semester, 2010 Lecture 12: Algorithms for Next Generation Sequencing Data, January 6, 2011 Scribe: Anat Gluzman and Eran Mick

[8] Minh Duc Cao, Son Hoang Nguyen, Devika Ganesamoorthy, Alysha G. Elliott, Matthew A. Cooper & Lachlan J. M. Coin. Scaffolding and completing genome assemblies in real-time with nanopore sequencing. Nature Communications 8, Article number: 14515, 2017.



Exact Matching Algorithms

Exact Matching Problem:

search pattern P in text T (P,T are strings)

- **Knuth Morris Pratt**
  preprocess pattern P
- **Aho Corasick**
  preprocess a pattern P
  of several strings $P = \{ P_1, \ldots, P_r \}$
- **Suffix Trees**
  preprocess text T
  or several texts, a 'database' of texts

Exact Matching: Preprocessing Pattern(s)

Knuth-Morris-Pratt (KMP)

Aho-Corasick

Searching Pattern P in Text T

Naive:

```
T: neeneeneeneeeeneneeeneeneneeneneen
P: neeneeen
    neeneeen
     neeneeen
      neeneeen
```

Better:

```
T: neeneeneeneeeeneneeeneeneneeneneen
P: neeneeen
       neeneeen
          neeneeen
```

KMP Preprocessed P example

Pattern P ∈ {a,b,…}*:

Automaton:

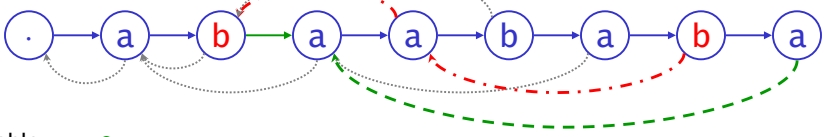


Table:

1	2	3	4	5	6	7	8
a	b	a	a	b	a	b	a
0	1	1	2	2	3	4	3

failure links (suffix = prefix):

top: as 'automaton'

bottom: as table

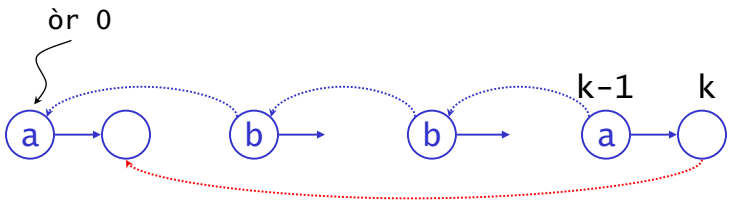
- how to determine?
- how to use?

```
                  6
P: a b a a b a b a
P:       a b a a b a b a
                  3
                    7
P: a b a a b a b a
P:       a b a a b a b a
                  4
                    8
P: a b a a b a b a
P:             a b a a b ...
                  3
```

KMP computing failure links

failure link ~ new 'best' match (after mismatch)

Pattern P:




```
Flink[1] = 0;
for k from 2 to PatLen
do  fail = Flink[k-1]
    while ( fail>0 and P[fail]≠P[k-1] )
    do  fail = Flink[fail];
    od
    Flink[k] = fail+1;
od
```

KMP computing failure links

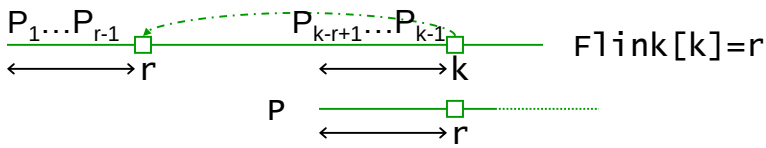
```
Flink[1] = 0;
for k from 2 to PatLen
do  fail = Flink[k-1]
    while ( fail>0 and P[fail]≠P[k-1] )
    do  fail = Flink[fail];
    od
    Flink[k] = fail+1;
od
```

Table:

1	2	3	4	5	6	7	8	
a	b	a	a	b	a	b	a	fail:
0								0
0	1							0->F->Flink[2]=0+1
0	1	1						1->T->0->F->Flink[3]=1
0	1	1	2					1->F->Flink[4]=1+1
…								…
0	1	1	2	2	3	4	3	0 1 1 2 2 3 4 3

prefixes via failure links

Pattern P:

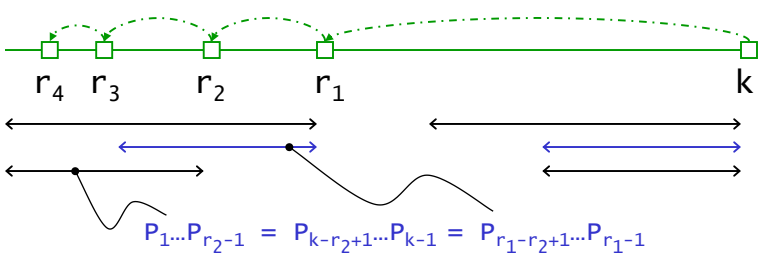


Flink[k]=r

$P_1 \ldots P_{r-1} = P_{k-r+1} \ldots P_{k-1}$ maximal r<k

prefix suffix

all such values r:



$P_1 \ldots P_{r_2-1} = P_{k-r_2+1} \ldots P_{k-1} = P_{r_1-r_2+1} \ldots P_{r_1-1}$

$\Rightarrow$ Flink[r_1]=r_2

other methods

- **Boyer-Moore**

```
T = marktkoopman
P = schoenveter
        schoe…
```

work backwards

- **Karp-Rabin 'fingerprint'**

a_{i-1} a_i … a_{i+n-1} a_{i+n}
p_1 … p_n

hash-value

$$a_i B^{n-1} + a_{i+1} B^{n-2} + \dots + a_{i+n-1} B^0$$
$$a_{i+1} B^{n-1} + \dots + a_{i+n-1} B^1 + a_{i+n} B^0$$

exact matching with a set of patterns

$P = \{ P_1, \dots, P_r \}$ total length m
all occurrences in text T length n

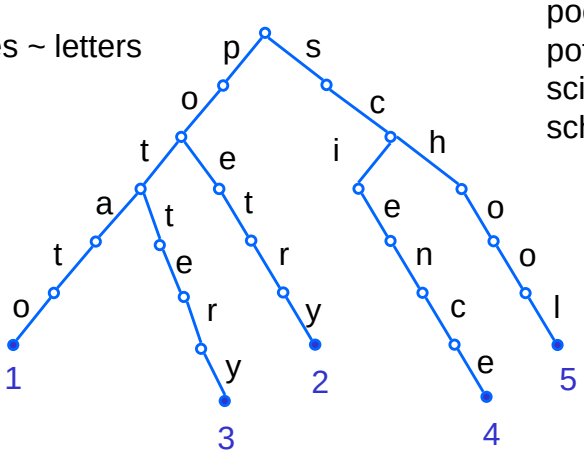
- **AHO CORASICK**
  generalizes KMP *failure links:*
  longest suffix that is prefix
  (perhaps in another string)

> assume no subwords within P

keyword tree - *trie*

{ potato, 1
poetry, 2
pottery, 3
science, 4
school } 5

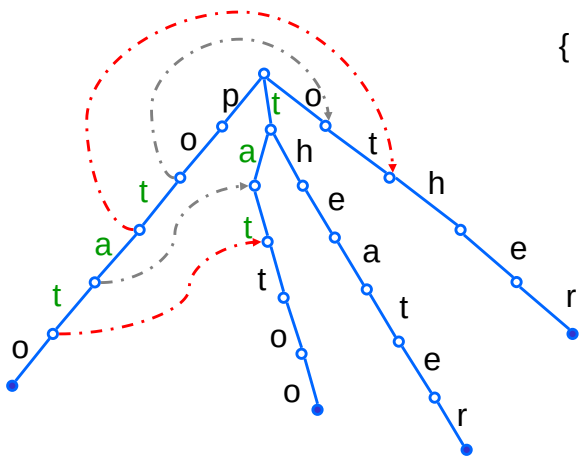
edges ~ letters



leaves ~ keywords

failure links

{ potato,
tattoo,
theater,
other }

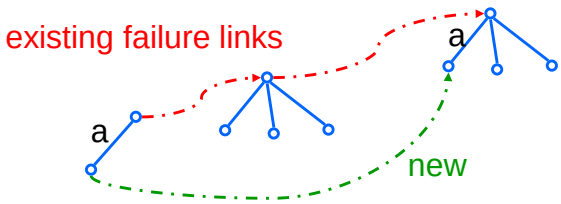


potato
other

potato
tattoo

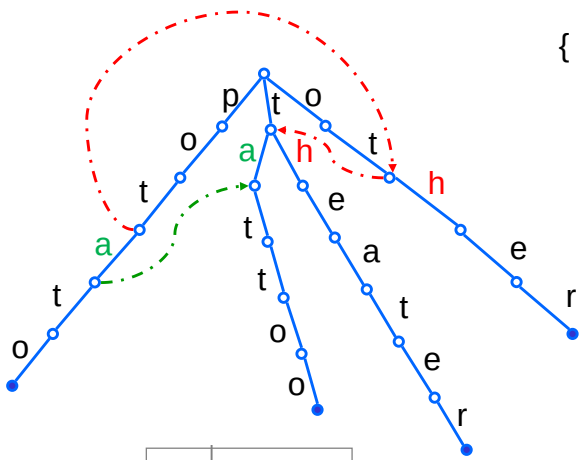
failure links into other branches!
=> into other patterns

Algorithm for adding the failure links: follow the links



edge with label a: follow existing failure links starting at the parent node of the edge until an outgoing edge with label a is found

Adding failure links

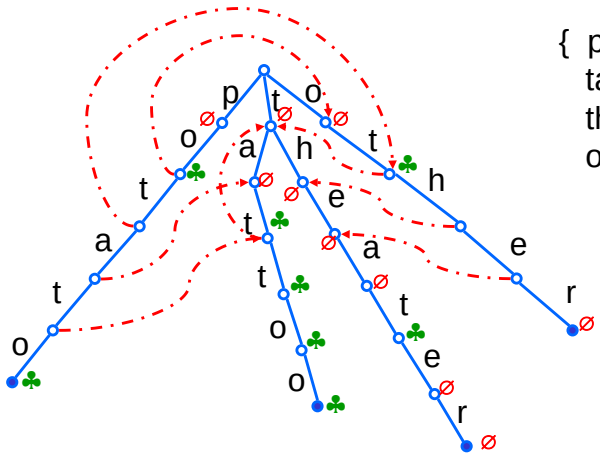


{ potato, tattoo, theater, other }

potato
other ☹
theater
tattoo

breadth first
(level-by-level)

failure links



{ potato,
tattoo,
theater,
other }

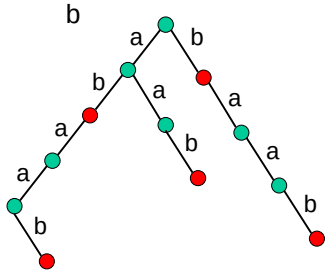
'shortcuts' ∅ root ♣ to child of root (see for example 'po') [single letter]

Preprocess text T: Suffix Trees

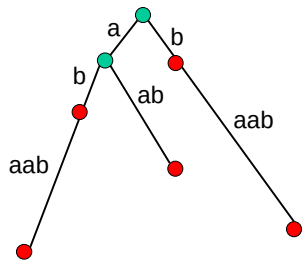
trie vs. suffix tree

Text T: abaab
Suffixes: baab
aab
ab
b

Text string T (= ‘abaab’) + all its suffixes



trie

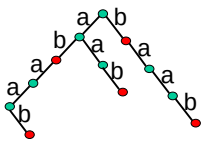


suffix tree

From: www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

Trie vs. Suffix Tree

Trie:

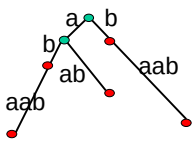


Given text T:

- |Trie(T)| = $O(|T|)^2$ quadratic
- bad example: T = $a^n b^n$
- Trie(T) like DFA for the suffixes of T
- minimize DFA → directed acyclic word graph

DFA = Deterministic Finite Automaton
(also used by Ukkonen)

Suffix Tree:

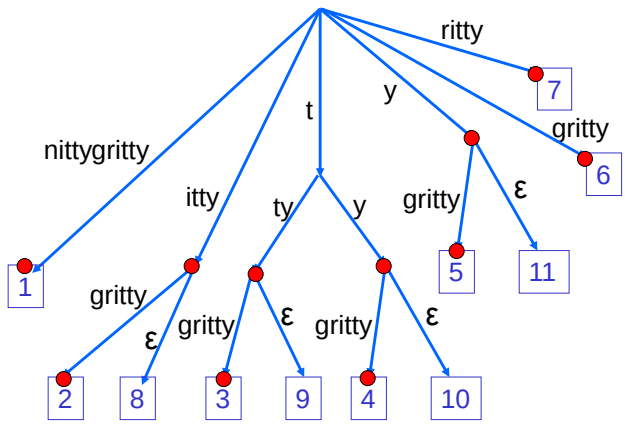


- only branching nodes and leaves represented
- edges labeled by substrings of T
- correspondence of leaves and suffixes
- |T| leaves, hence < |T| internal nodes
- |Tree(T)| = O(|T| + size(edge labels)) linear

‘nittygritty’

Order of insertion is position of suffix in Suffix Tree of text T:

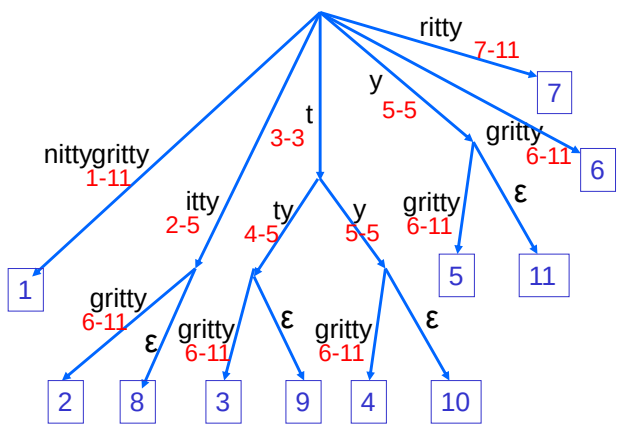
nittygritty	1
ittygritty	2
ttygritty	3
tygritty	4
ygritty	5
gritty	6
ritty	7
itty	8
tty	9
ty	10
y	11



‘nittygritty’

Text T and its suffixes:

nittygritty	1
ittygritty	2
ttygritty	3
tygritty	4
ygritty	5
gritty	6
ritty	7
itty	8
tty	9
ty	10
y	11



implementation: refer to positions of the substrings in T

linear time construction

Text T and its suffixes:

nittygritty
ittygritty
ttygritty
tygritty
ygritty
gritty
ritty
itty
tty
ty
y

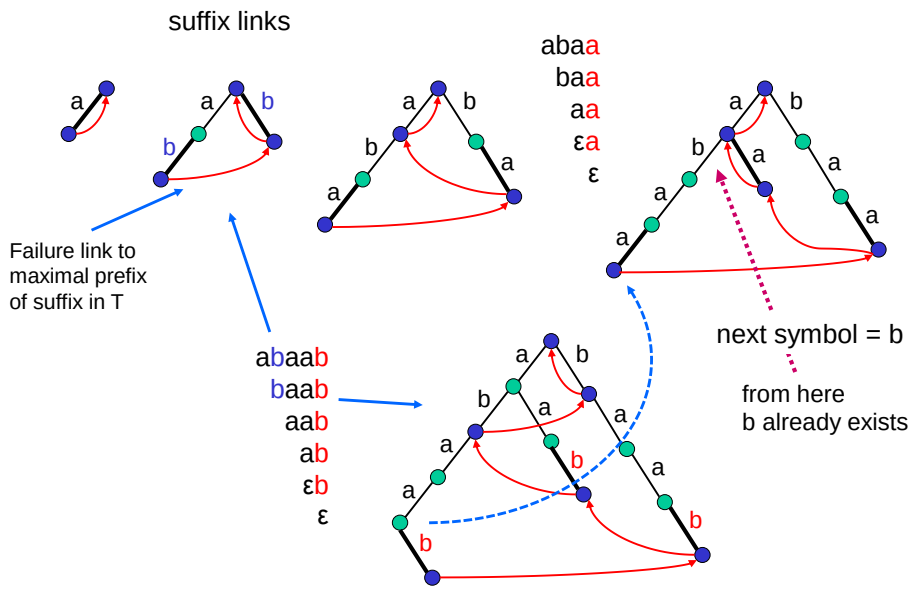
Weiner (1973)

'algorithm of the year'

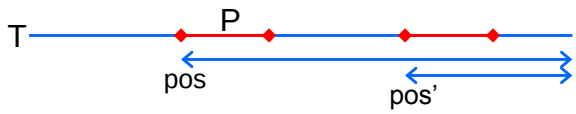
McCreight (1976)

'on-line' algorithm (Ukkonen 1992)

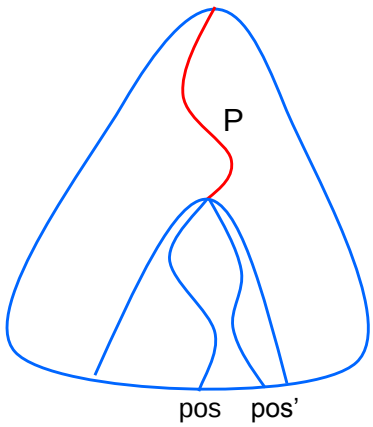
online construction of suffix trie for T = 'abaab'



Suffix Tree application: full text index



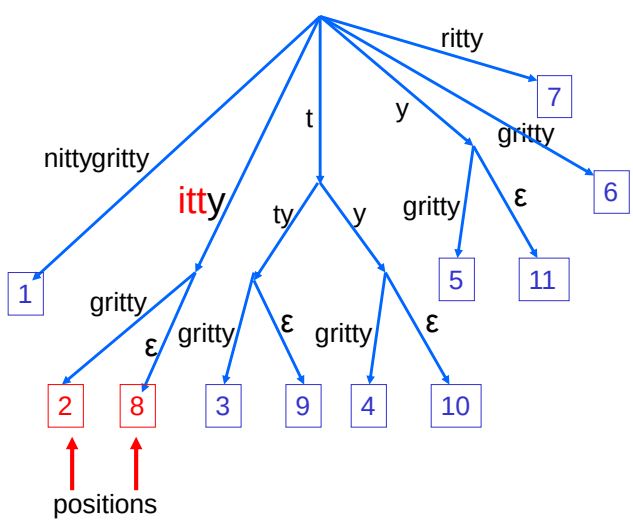
P in T ⇔ P is prefix of suffix of T



subtree under P
~ locations of P

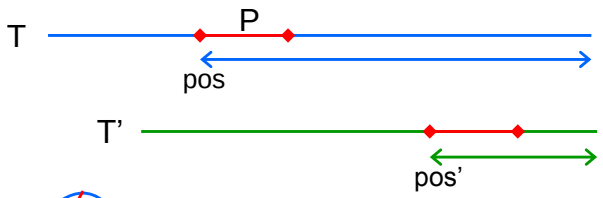
example: find 'itt' in 'nittygritty'

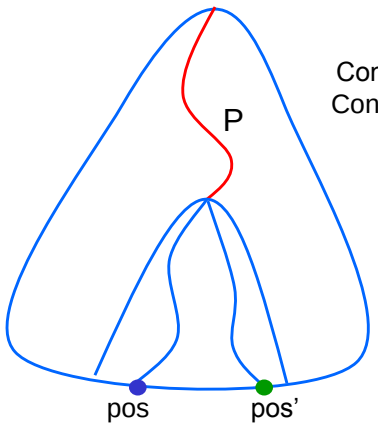
nittygritty	1
ittygritty	2
ttygritty	3
tygritty	4
ygritty	5
gritty	6
ritty	7
itty	8
tty	9
ty	10
y	11



Suffix Tree application: longest common substring

apples
plate





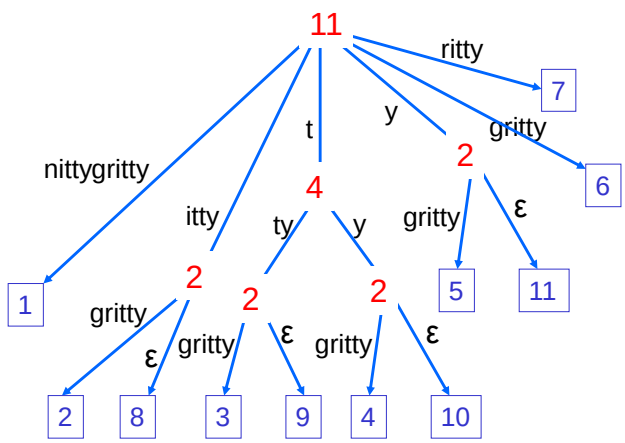
Construct a 'generalized' suffix tree
Containing suffixes of both T and T'.

Mark T and T' suffixes:

And find the longest common prefixes of suffixes from different texts stored in the Suffix Tree.

application: counting 'motifs'

nittygritty	1
ittygritty	2
ttygritty	3
tygritty	4
ygritty	5
gritty	6
ritty	7
itty	8
tty	9
ty	10
y	11



The suffix tree now contains counts for the number of suffixes in a sub-tree

Suffix Trees Experiments: ‘motifs’, repeats in DNA

as reported by Ukkonen

- human chromosome 3
- the first 48 999 930 bases
- 31 min cpu time (8 processors, 4 GB)

- human genome: 3×10^9 bases
- suffix tree for Human Genome is feasible

longest repeat?

Occurrences at: 28395980, 28401554r **Length: 2559**

ttagggtacatgtgcacaacgtgcaggtttgttacatatgtatacacgtgccatgatggtgtgctgcacccattaactcgtcatttagcgtta
ggtatatctccgaatgctatccctccccctcccccacccacaacagtccccggtgtgtgatgttccccttcctgtgtccatgtgttctca
ttgttcaattcccacctatgagtgagaacatgcggtgtttggttttttgtccttgcgaaagtttgctgagaatgatggtttccagcttcatccata
tccctacaaaggacatgaactcatcatttttttatggctgcatagtattccatggtgtatatgtgccacattttcttaacccagtctacccttgttg
gacatctgggttggttccaagtctttgctattgtgaatagtgccgcaataaacatacgtgtgcatgtgtctttatagcagcatgatttataatcc
tttgggtatatacccagtaatgggatggctgggtcaaatggtatttctagttctagatccctgaggaatcaccacactgacttccacaatggt
tgaactagtttacagtcccagcaacagttcctatttctccacatcctctccagcacctgttgtttcctgacttttaatgatcgccattctaactg
gtgtgagatggtatctcattgtggttttgatttgcatttctctgatggccagtgatgatgagcatttttcatgtgttttttggctgcataaatgtctt
cttttgagaagtgtctgttcatatccttcgcccacttttgatgggggttgtttgttttttttcttgtaaatttgttggagttcattgtagattctgggtatta
gccctttgtcagatgagtaggttgcaaaaattttctcccattctgtaggttgcctgttcactctgatggtggtttcttctgctgtgcagaagctct
ttagtttaattagatcccatttgtcaattttggctttgttgccatagcttttggtgttttagacatgaagtccttgcccatgcctatgtcctgaatg
gtattgcctaggttttcttctagggtttttatggttttaggtctaacatgtaagtctttaatccatcttgaattaattataaggtgtatattataaggtg
taattataaggtgtataattatatattaattataaggtgtatattaattataaggtgtaaggaagggatccagtttcagctttctacatatggctag
ccagttttccctgcaccatttattaaatagggaatcctttccccattgcttgtttttgtcaggtttgtcaaagatcagatagttgtagatatgcgg
cattatttctgagggctctgttctgttccattggtctatatctctgttttggtaccagtaccatgctgttttggttactgtagccttgtagtatagttt
gaagtcaggtagcgtgatggttccagctttgttcttttggcttaggattgacttggcaatgtgggctctttttggttccatatgaacttttaaagt
agtttttttccaattctgtgaagaaattcattggtagcttgatggggatggcattgaatctataaattaccctgggcagtatggccattttcacaa
tattgaatcttcctacccatgagcgtgtactgttcttccatttgtttgtatcctctttatttcattgagcagtggtttgtagttctccttgaagaggt
ccttcacatcccttgtaagttggattcctaggtattttattctctttgaagcaattgtgaatgggagttcactcatgatttgactctctgtttgtctg
ttattggtgtataagaatgcttgtgatttttgcacattgattttgtatcctgagactttgctgaagttgcttatcagcttaaggagattttgggctga
gacgatggggttttctagatatacaatcatgtcatctgcaaacagggacaatttgacttcctctttttcctaattgaatacccgttatttccctctc
ctgcctgattgccctggccagaacttccaacactatgttgaataggagtggtgagagagggcatccctgtcttgtgccagttttcaaaggg
aatgcttccagttttgtccattcagtatgatattggctgtgggtttgtcatagatagctcttattattttgagatacatcccatcaatacctaattt
attgagagtttttagcatgaagagttcttgaattttgtcaaaggccttttctgcatcttttgagataatcatgtggtttctgtctttggttctgtttata
tgctggagtacgtttattgattttcgtatgttgaaccagccttgcatcccagggatgaagcccacttgatcatggtggataagcttttgatgt
gctgctggattcggtttgccagtattttattgaggatttctgcatcgatgttcatcaaggatattggtctaaaattctctttttttgttgtgtctctgt
caggctttggtatcaggatgatgctggcctcataaaatgagttagg

ten occurrences?

**ttttttttttttttgagacggagtctcgctctgtcgcccaggctggagtgcagtg
gcgggatctcggctcactgcaagctccgcctcccgggttcacgccattct
cctgcctcagcctcccaagtagctgggactacaggcgcccgccactacg
cccggctaatttttttgtatttttagtagagacggggtttcaccgttttagccgg
gatggtctcgatctcctgacctcgtgatccgcccgcctcggcctcccaaag
tgctgggattacaggcgt**

Length: 277

Occurrences at: 10130003, 11421803, 18695837, 26652515, 42971130, 47398125
In the reversed complement at: 17858493, 41463059, 42431718, 42580925

Suffix Tree Remarks

suffix tree

efficient (linear) storage,
but constant ±40, *large, 'overhead'*

suffix *array*

has constant ±5 'overhead'
hence more practical
but has its own complications

naïve n log(n) algorithm is in some cases not too bad… (next slide)

Suffix Array

nittygritty	1
ittygritty	2
ttygritty	3
tygritty	4
ygritty	5
gritty	6
ritty	7
itty	8
tty	9
ty	10
y	11

⇒

gritty	6
itty	8
ittygritty	2
nittygritty	1
ritty	7
tty	9
ttygritty	3
ty	10
tygritty	4
y	11
ygritty	5

lexicographic order of the suffixes

References

Dan **Gusfield**
Algorithms on Strings, Trees, and Sequences
Computer Science and Computational Biology

This book lists *many* applications for suffix trees
and has extended implementation details.

Several slides on suffix-trees are based on and/or copied from
Esko **Ukkonen**, Univ Helsinki (Erice School, 30 Oct 2005)