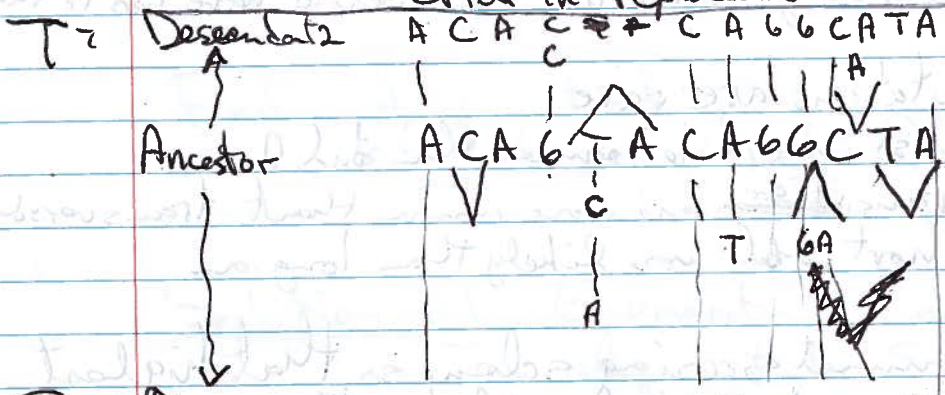
Sept 12 2016

Sequence alignment

Mutation: Change in sequence due to error in replication or unrepaired DNA damage



Substitution: Change one nuc for another

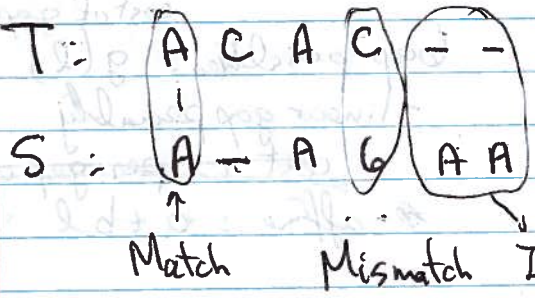
A ↔ 6 } transition
C ↔ T } common

A ↔ C, 6, T transversion (~3 times less common)

Insertion: Insert 1 or more nuc.

Deletion: Delete one or more

Alignment of S, T is obtained by inserting gaps in S and T to obtain S' and T', so that in S' and T', nucleotides derived from same ancestor occur at same position. Gaps reflect nucleotides missing due to del. or insertion



Problem: Ancestor and mutation scenarios are generally not known.

Idealized Pairwise Global Alignment Problem

Given: S, T
Find: Alignment of S and T that is most likely to reflect the evolutionary scenario that would have led to them

Key idea:
1. Mutations are rare
2. Subst. are more common than indel
3. Transitions are more common than transversions
4. Short indel more likely than long ones

⇒ Define alignment scoring scheme so that highest scoring alignment is the one that is the most likely to be correct.

Scoring a given alignment

```
S: A C A G T C - - T A
T: A T A - T A A A T A
   1 -1 1 -2 1 -1 -2 -2 1 1 : -3
```

Score: Subst. score + indel score

Subst. fct. or cost matrix

$$M = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{pmatrix} +1 & -2 & -1 & -2 \\ -2 & +1 & -2 & -1 \\ & & \ddots & \\ & & & \ddots \end{pmatrix} \end{array}$$

Gap penalties: $g(\ell)$ = cost of gap of length ℓ

- linear gap penalty: $g\ell = -2\ell$ ~~cost c per gap character~~
- affine: $a + b\ell = -5 - 1\ell$

Pairwise seq. Aln. Problem (linear gap penalty)

Given: $S = S_1 \ldots S_m$, $T = T_1 \ldots T_n$ sequences
M subst cost matrix
c

Find: Alignment of S, T
s.t that score(Aln(S,T)) is maximized

Solution #1: Enumerate and evaluate all possible alignments for S and T, report best

Problem: Way too slow $\Omega(2^{m+n})$

Solution #2: Needleman-Wunsch algo. (1970)

S = AGCT
T = ACA

Dynamic Prog algo

$s_1\ s_2\ s_3\ s_4$
S = ACAT
T = AGT
$t_1\ t_2\ t_3$

X =

	·	A	G	T
·				
A				
C				
A				
T			○	

$X_{i,j}$ = Score of the best alignment for $s_1 \dots s_i$ against $t_1 \dots t_j$

Note: We want $X_{m,n}$, and we want the alignment that achieves this score.

How to calculate $X_{i,j}$?

Alignment	Score
Best Aln $\begin{pmatrix} s_1 \dots s_i \\ t_1 \dots t_j \end{pmatrix}$ either looks like Best Aln $\begin{pmatrix} s_1 \dots s_{i-1} & s_i \\ t_1 \dots t_{j-1} & t_j \end{pmatrix}$	$\rightarrow X_{i-1,j-1} + M($
OR Best Aln $\begin{pmatrix} s_1 \dots s_{i-1} & s_i \\ t_1 \dots t_j & - \end{pmatrix}$	$\rightarrow X_{i-1,j} + c$
OR Best Aln $\begin{pmatrix} s_1 \dots s_i & - \\ t_1 \dots t_{j-1} & t_j \end{pmatrix}$	$\rightarrow X_{i,j-1} + c$

So: $X_{i,j} = \max \begin{cases} X_{i-1,j-1} + M(s_i, t_j) \\ X_{i-1,j} + c \\ X_{i,j-1} + c \end{cases}$ (when $i>0, j>0$)

$X_{0,j} = j \cdot c$
$X_{i,0} = i \cdot c$
$X_{0,0} = 0$

	·	A	G	T
·	0	-2	-4	-6
A	-2	+1	0	-2
C	-4	-1	0	-1
A	-6	-3	-2	-3
T	-8	-5	-4	-1

$c = -2$

Calculate X entri
to right, top

For each entry, keep
to entry that achieved

$X_{4,3} = -1$: Score of best alignment

We recover best alignment following backpointers from (m,n)
↖ : Match
← : Gap in S
↑ : Gap in T

Best = A C A T
Alm = A - G T

Running time: $O(m \cdot n)$
Space: $O(m \cdot n)$

Note: NW algo can be modi
with affine gap pen
→ Why does it not al
→ What is the modific

To recover all optimal alignments, do Depht-First-Search
backpointers starting from (m,n).

Pairwise local alignment problem

Idea: Often, two sequences only share significant sim
over a particular region.
Examples: Two

Given: Sequences $S = s_1 \ldots s_m$ and $T = t_1 \ldots t_n$
Subst. matrix M
gap penalty c

Find: Indices i, j, k, ℓ, where $1 \le i \le j \le m$, $1 \le k \le \ell \le n$

such that $\text{Score}\left(\text{BestAln}\left(\begin{matrix} s_i \ldots s_j \\ t_k \ldots t_\ell \end{matrix}\right)\right)$ is maximized

Can we do better than this?

```
for i:1...m
    for j:i...m
        for k:1...n
            for l:k...n
                NW(s_i...s_j, t_k...t_l)
```

$O(m^3 n^3)$

Smith Waterman algo.

$X_{i,j}$ = Score of best local alignment for $s_1 \ldots s_i, t_1 \ldots t_j$ where s_i and t_j are included in alignment

$$X_{i,j} = \max \begin{cases} X_{i-1,j-1} + M(s_i, t_j) \\ X_{i-1,j} + c \\ X_{i,j-1} + c \\ 0 \end{cases}$$

$X_{i,0} = 0 \quad \forall i = 1 \ldots m$
$X_{0,j} = 0 \quad \forall j = 1 \ldots n$

Traceback from $\underset{i \in \{1 \ldots m\}, j \in \{1 \ldots n\}}{\text{argmax}} \{X_{i,j}\}$ until hitting zero

Fast Alignment Heuristics

COMP462/561: Computational Biology Methods

*Based on Course Notes by Dr. Mathieu Blanchette

1

Smith-Waterman: Local Alignment (1981)

Problem:

Given two sequences, *S* and *T*, of lengths *m* and *n*, find the substring *s* of *S* and the substring *t* of *T* such that the alignment score of *s* against *t* is maximized

Algorithm: Dynamic programming algo (very similar to NW)

1. Initialization matrix
2. Fill matrix with appropriate alignment scores
3. Trace back from highest scoring cell(s) to find best alignment(s)

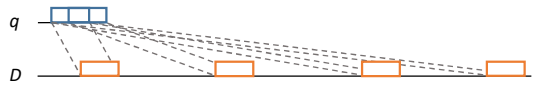
2

Motivation

Problem:

Given a query sequence, *q*, of length *m* (small, ~1000 nucleotides) and a large database (target), *D*, of size *n* (billions of nucleotides)

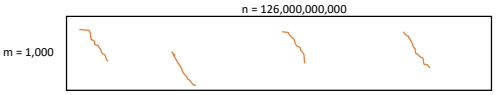
Find all local alignments of *q* within *D* that have a score above threshold, *T*



3

Smith-Waterman?

- SW is **too slow**…would take O(mn + m*hits)
- How?
  - Trace back all entries of a dynamic programming matrix with a score>*T*



- ***Too slow, too much memory!***

4

SW Initialization

A = CGTGAATTCAT

B = GCTTAC

$H(i,0) = 0, 0 \leq i \leq m$

$H(0,j) = 0, 0 \leq j \leq n$

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0											
C	0											
T	0											
T	0											
A	0											
C	0											

5

SW Matrix Filling

Similar to **Needleman-Wunsch (NW)**, fill in the matrix such that:

$$H(i,j) = max^1 \begin{cases} H(i-1,j-1) + s(a_i, b_j) & Match/Mismatch \\ max_{k \geq 1}\{H(i-k,j) + g\} & Deletion \\ max_{l \geq 1}\{H(i,j-l) + g\} & Insertion \end{cases}, 1 \leq i \leq m, 1 \leq j \leq n$$

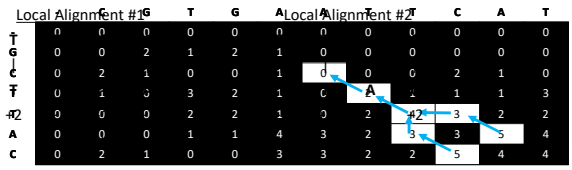
	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	2	1	2	1	0	0	0	0	0	0
C	0	2	1	0	1	1	0	0	0	2	1	0
T	0	0	0	3	2	1	0	0	1	1	2	1
T	0	0	0	2	2	1	0	0	2	1	1	3
A	0	0	0	1	1	4	3	2	1	3	2	2
C	0	2	1	0	0	3	3	2	2	5	4	4

With a match score of +2 and a mismatch & indel score equal to -1.

6

SW Trace Back

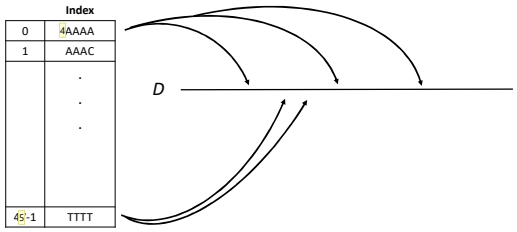
- With NW, we trace back from the bottom right-most cell of the matrix
  - Slightly different with SW. **How?**



7

7

Preprocess Database to Build Indices



10

10

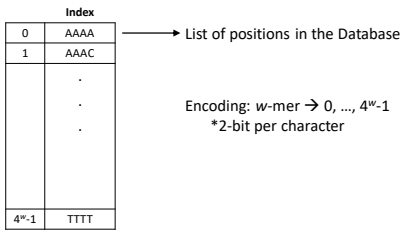
Basic Local Alignment Search Tool Idea

- Give up on (guaranteed) optimality
  - *Heuristic* approach
  - Search only for local-alignments with **high-scoring gapless alignments (HSPs)**
- Pre-process the database, *D*, so that queries can be answered in constant time with respect to *n*
- **BLAST** was published in 1990 by Altschul, Lipman, Miller,
  - cited by more than 10^5 papers

8

8

Indexing the Database



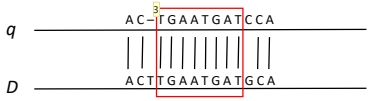
Encoding: *w*-mer → 0, …, 4^w-1
*2-bit per character

11

11

Gapless Alignments

- If *q* has a good alignment, *X*, somewhere in *D*



- Then *X* is likely to contain a HSP

9

9

Scanning for Hits in *D*

- Given a query, *q*

For each *w*-mer in q — O(|q|)
  For each matches c: q in *D* — O(w)
    Investigate match further…

How many hits do we expect for a *w*-mer of size 11?

$$\frac{3 \times 10^9}{4^{11}} = 1000$$

12

12

Slide 9

2 Assumption:
Lê Nhật Hưng, 23-Sep-19

1 "High scoring pair"
Lê Nhật Hưng, 23-Sep-19

3 Portion in q 100% identical to portion in D
Lê Nhật Hưng, 23-Sep-19

Slide 10

4 Size of portion expect to have perfect match is 4.

Usually, it's 11
Lê Nhật Hưng, 23-Sep-19

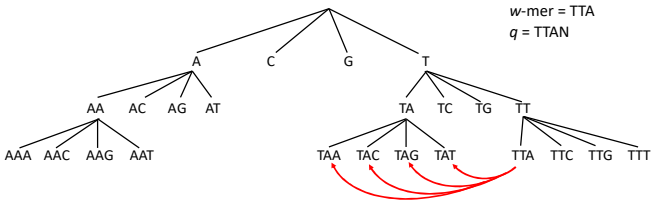
5 w=4 here
Lê Nhật Hưng, 23-Sep-19

Slide 12

6 TYPO: must be w
Lê Nhật Hưng, 23-Sep-19

Improving Scan Times

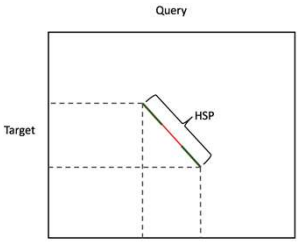
- Encode database indices in a **trie**



13

13

BLAST HSP



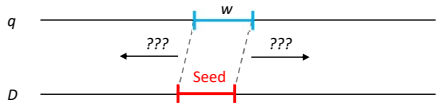
How can we improve the hit further?

16

16

Extending hits to find High Scoring Pair (HSP)

Goal: Quickly determine if a match is promising or not



14

14

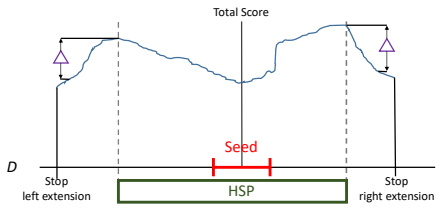
Statistics of Local Alignments

- Even if *D* was completely random, we would expect to observe some pretty high scoring HSPs
  - How do we know when we should get excited?
- **E-value** (score(HSP))
  - The expected number of local alignments with a score greater or equal to HSP's that would be found in a random *D*

17

17

Ungapped Extension Phase



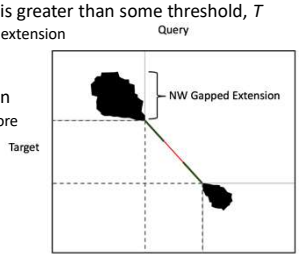
Time? Linear in size of extension

15

15

Gapped Extension

- If the HSP's alignment score is greater than some threshold, *T*
  - Do a more expensive gapped extension
    - Using a variant NW
- Perform NW in each direction
  - Consider only entries with score greater than "best so far"



18

18

Choosing the size of *w*

Small (<=11)
- High probability of finding exact *w*-mer in HSP
- Lots of false positive seeds
- High sensitivity
- Slow

Large (>12)
- Miss many HSPs
- Few false positives
- Low sensitivity
- Fast

19

19

Karlin-Altschul (1990)

$$E(S) = Kmn$$

- $E(S) = K\ m\ n\ e$
- S is the score of the ungapped HSP alignment
- K and depend on the scoring scheme and background probabilities
  - scales scores scheme
- A low E-value (10^{-1} - 10^{-100}) is a good match
  - Low chance of observing HSP given random chance alone

20

20

Variants

For proteins: inexact matches are considered
- Based on a **point accepted matrix (PAM)**

Query	Target	BLAST variant
DNA	DNA	blastn
Protein	Protein	blastp
Protein	DNA	tblastn
DNA	protein	blastx

21

21

Optimizations

- **Dealing with repeats in q or D**
- **Two-Hit method**
  - Lower *T* to allow more hits, but only extend if two hits fall on the same diagonal
    - Within a window of fixed length
    - *Increases* hits and *lowers* extensions
- **Gapped seeds**

22

22

Upcoming Topics

- Wednesday – **multiple sequence alignment (MSA)**
  - Dr. Blanchette will return!
- End of the semester – **Burrows-Wheeler Transform (BWT)**
  - https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform
  - In pattern matching: https://www.youtube.com/watch?v=z5EDLODQPtg

23

23

Burrows-Wheeler Transform

CMSC 423

Motivation - Short Read Mapping

If we already know the genome of one cow, we can get reads from a 2nd cow and map them onto the known cow genome.

Need to do millions of string searches in a long string.

Bowtie

Software Highly accessed Open access

Ultrafast and memory-efficient alignment of short DNA sequences to the human genome

Ben Langmead*, **Cole Trapnell**, **Mihai Pop** and **Steven L Salzberg**

* Corresponding author: Ben Langmead langmead@cs.umd.edu

▼ Author Affiliations

Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

For all author emails, please log on.

Genome Biology 2009, **10**:R25 doi:10.1186/gb-2009-10-3-r25

BWA

Fast and accurate short read alignment with Burrows–Wheeler transform

Heng Li and Richard Durbin*

+ Author Affiliations

* To whom correspondence should be addressed.

Received February 20, 2009.
Revision received May 6, 2009.
Accepted May 12, 2009.

Bioinformatics (2009) 25(14):1754-1760.

Bowtie Performance

Varying read length using Bowtie, Maq and SOAP

Length	Program	CPU time	Wall clock time	Peak virtual memory footprint (megabytes)	Bowtie speed-up	Reads aligned (%)
36 bp	Bowtie	6 m 15 s	6 m 21 s	1,305	-	62.2
	Maq	3 h 52 m 26 s	3 h 52 m 54 s	804	36.7×	65.0
	Bowtie -v 2	4 m 55 s	5 m 00 s	1,138	-	55.0
	SOAP	16 h 44 m 3 s	18 h 1 m 38 s	13,619	216×	55.1
50 bp	Bowtie	7 m 11 s	7 m 20 s	1,310	-	67.5
	Maq	2 h 39 m 56 s	2 h 40 m 9 s	804	21.8×	67.9
	Bowtie -v 2	5 m 32 s	5 m 46 s	1,138	-	56.2
	SOAP	48 h 42 m 4 s	66 h 26 m 53 s	13,619	691×	56.2
76 bp	Bowtie	18 m 58 s	19 m 6 s	1,323	-	44.5
	Maq 0.7.1	4 h 45 m 7 s	4 h 45 m 17 s	1,155	14.9×	44.9
	Bowtie -v 2	7 m 35 s	7 m 40 s	1,138	-	31.7

The performance of Bowtie v0.9.6, SOAP v1.10, and Maq versions v0.6.6 and v0.7.1 on the server platform when aligning 2 M untrimmed reads from the 1,000 Genome project (National Center for Biotechnology Information Short Read Archive: SRR003084 for 36 base pairs [bp], SRR003092 for 50 bp, and SRR003196 for 76 bp). For each read length, the 2 M reads were randomly sampled from the FASTQ file downloaded from the Archive such that the average per-base error rate as measured by quality values was uniform across the three sets. All reads pass through Maq's "catfilter". Maq v0.7.1 was used for the 76-bp reads because v0.6.6 does not support reads longer than 63 bp. SOAP is excluded from the 76-bp experiment because it does not support reads longer than 60 bp. Other experimental parameters are identical to those of the experiments in Table 1. CPU, central processing unit.

Maq & SOAP build hash table of locations of k-mers

Langmead et al. (2008)

Burrows-Wheeler Transform

Text transform that is useful for compression & search.

banana

banana$	$banana	BWT(banana) =
anana$b	a$banan	annb$aa
nana$ba	ana$ban	
ana$ban	anana$b	
na$bana	banana$	
a$banan	nana$ba	
$banana	na$bana	

sort →

Tends to put runs of the same character together.

Makes compression work well.

“bzip” is based on this.

Another Example

appellee$

appellee$		$appellee
ppellee$a		appellee$
pellee$ap		e$appelle
ellee$app	sort →	ee$appell
llee$appe		ellee$app
lee$appel		lee$appel
ee$appell		llee$appe
e$appelle		pellee$ap
$appellee		ppellee$a

BWT(appellee$) =
e$elplepa

Doesn't always improve the compressibility...

Recovering the string

$appellee
appellee$
e$appelle
ee$appell
ellee$app
lee$appel
llee$appe
pellee$ap
ppellee$a

BWT → sort → BWT

e	$
$	a
e	e
l	e
p	e
l	l
e	l
p	p
a	p

sort these 2 columns → first column

$a
ap
e$
ee
el
le
ll
pe
pp

prepend BWT column → first 2 columns

e	$a
$	ap
e	e$
l	ee
p	el
l	le
e	ll
p	pe
a	pp

Sort these 3 columns → first 3 columns

$ap
app
e$a
ee$
ell
lee
lle
pel
ppe

Inverse BWT

```
def inverseBWT(s):
    B = [s1,s2,s3,...,sn]
    for i = 1..n:
        sort B
        prepend si to B[i]
    return row of B that ends with $
```

Another BWT Example

dogwood$
ogwood$d
gwood$do
wood$dog
ood$dogw
od$dogwo
d$dogwoo
$dogwood

sort →

$dogwood
d$dogwoo
dogwood$
gwood$do
od$dogwo
ogwood$d
ood$dogw
wood$dog

last column →

BWT(dogwood$) =
do$oodwg

do$oodwg Another BWT Example

d	$	$d	d	$d	$do	d	$do	$dog	d	$dog	$dogw
o	d	d$	o	d$	d$d	o	d$d	d$do	o	d$do	d$dog
$	d	do	$	do	dog	$	dog	dogw	$	dogw	dogwo
o	g	gw	o	gw	gwo	o	gwo	gwoo	o	gwoo	gwood
o	o	od	o	od	od$	o	od$	od$d	o	od$d	od$do
d	o	og	d	og	ogw	d	ogw	ogwo	d	ogwo	ogwoo
w	o	oo	w	oo	ood	w	ood	ood$	w	ood$	ood$d
g	w	wo	g	wo	woo	g	woo	wood	g	wood	wood$
		Prepend			Sort			Prepend			Sort

d	$dogw	$dogwo	d	$dogwo	$dogwoo	d	$dogwoo	$dogwood			
o	d$dog	d$dogw	o	d$dogw	d$dogwo	o	d$dogwo	d$dogwoo			
$	dogwo	dogwoo	$	dogwoo	dogwood	$	dogwood	dogwood$			
o	gwood	gwood$	o	gwood$	gwood$d	o	gwood$d	gwood$do			
o	od$do	od$dog	o	od$dog	od$dogw	o	od$dogw	od$dogwo			
d	ogwoo	ogwood	d	ogwood	ogwood$	d	ogwood$	ogwood$d			
w	ood$d	ood$do	w	ood$do	ood$dog	w	ood$dog	ood$dogw			
g	wood$	wood$d	g	wood$d	wood$do	g	wood$do	wood$dog			
		Prepend			Sort			Prepend			Sort

Searching with BWT: LF Mapping

LF Mapping

BWT(unabashable)	Σ								
	$	a	b	e	h	l	n	s	u
$unabashable	0	0	0	0	0	0	0	0	0
abashable$un	0	0	0	1	0	0	0	0	0
able$unabash	0	0	0	1	0	0	1	0	0
ashable$unab	0	0	0	1	1	0	1	0	0
bashable$una	0	0	1	1	1	0	1	0	0
ble$unabasha	0	1	1	1	1	0	1	0	0
e$unabashabl	0	2	1	1	1	0	1	0	0
hable$unabas	0	2	1	1	1	1	1	0	0
le$unabashab	0	2	1	1	1	1	1	1	0
nabashable$u	0	2	2	1	1	1	1	1	0
shable$unaba	0	2	2	1	1	1	1	1	1
unabashable$	0	3	2	1	1	1	1	1	1
	1	3	2	1	1	1	1	1	1

# of times letter appears before this position in the last column.

LF Property: The i^{th} occurrence of a letter X in the last column corresponds to the i^{th} occurrence of X in the first column.

BWT Search

BWTSearch(aba) Start from the **end** of the pattern

Step 1: Find the range of "a"s in the first column

Step 2: Look at the same range in the last column.

Step 3: "b" is the next pattern character. Set B = the LF mapping entry for b in the first row of the range.
Set E = the LF mapping entry for b in the last + 1 row of the range.

Step 4: Find the range for "b" in the first row, and use B and E to find the right subrange within the "b" range.

BWT(unabashable)

	LF Mapping Σ								
	$	a	b	e	h	l	n	s	u
$unabashable	0	0	0	0	0	0	0	0	0
abashable$un	0	0	0	1	0	0	0	0	0
able$unabash	0	0	0	1	0	0	1	0	0
ashable$unab	0	0	0	1	1	0	1	0	0
bashable$una	0	0	1	1	1	0	1	0	0
ble$unabasha	0	1	1	1	1	0	1	0	0
e$unabashabl	0	2	1	1	1	0	1	0	0
hable$unabas	0	2	1	1	1	1	1	0	0
le$unabashab	0	2	1	1	1	1	1	1	0
nabashable$u	0	2	2	1	1	1	1	1	0
shable$unaba	0	2	2	1	1	1	1	1	1
unabashable$	0	3	2	1	1	1	1	1	1
	1	3	2	1	1	1	1	1	1

BWT Searching Example 2

pattern = "bana"

a	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

n	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

(B,E) = 0, 2

n	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

a	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

(B,E) = 1, 2

a	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

(B,E) = 0, 1

b	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

BWT Searching Notes

- Don't have to store the LF mapping. A more complex algorithm (later slides) lets you compute it in O(1) time in ***compressed*** data on the fly with some extra storage.

- To find the range in the first column corresponding to a character:
  - Pre-compute array C[c] = # of occurrences in the string of characters lexicographically < c.
  - Then start of the "a" range, for example, is: C["a"] + 1.

- Running time: O(|pattern|)
  - Finding the range in the first column takes O(1) time using the C array.
  - Updating the range takes O(1) time using the LF mapping.

Relationship Between BWT and Suffix Arrays

```
s = appellee$
    123456789
```

BWT matrix	The suffixes are obtained by deleting everything after the $		Suffix array (start position for the suffixes)		Suffix position - 1 = the position of the last character of the BWT matrix
$appellee	$	These are still in sorted order because "$" comes before everything else	9		s[9-1] = e
appellee$	appellee$		1		s[1-1] = $
e$appelle	e$		8		s[8-1] = e
ee$appell	ee$		7		s[7-1] = l
ellee$app	ellee$		4	– subtract 1 →	s[4-1] = p
lee$appel	lee$		6		s[6-1] = l
llee$appe	llee$		5		s[5-1] = e
pellee$ap	pellee$		3		s[3-1] = p
ppellee$a	ppellee$		2		s[2-1] = a

($ is a special case)

Relationship Between BWT and Suffix Trees

- Remember: Suffix Array = suffix numbers obtained by traversing the leaf nodes of the (ordered) Suffix Tree from left to right.

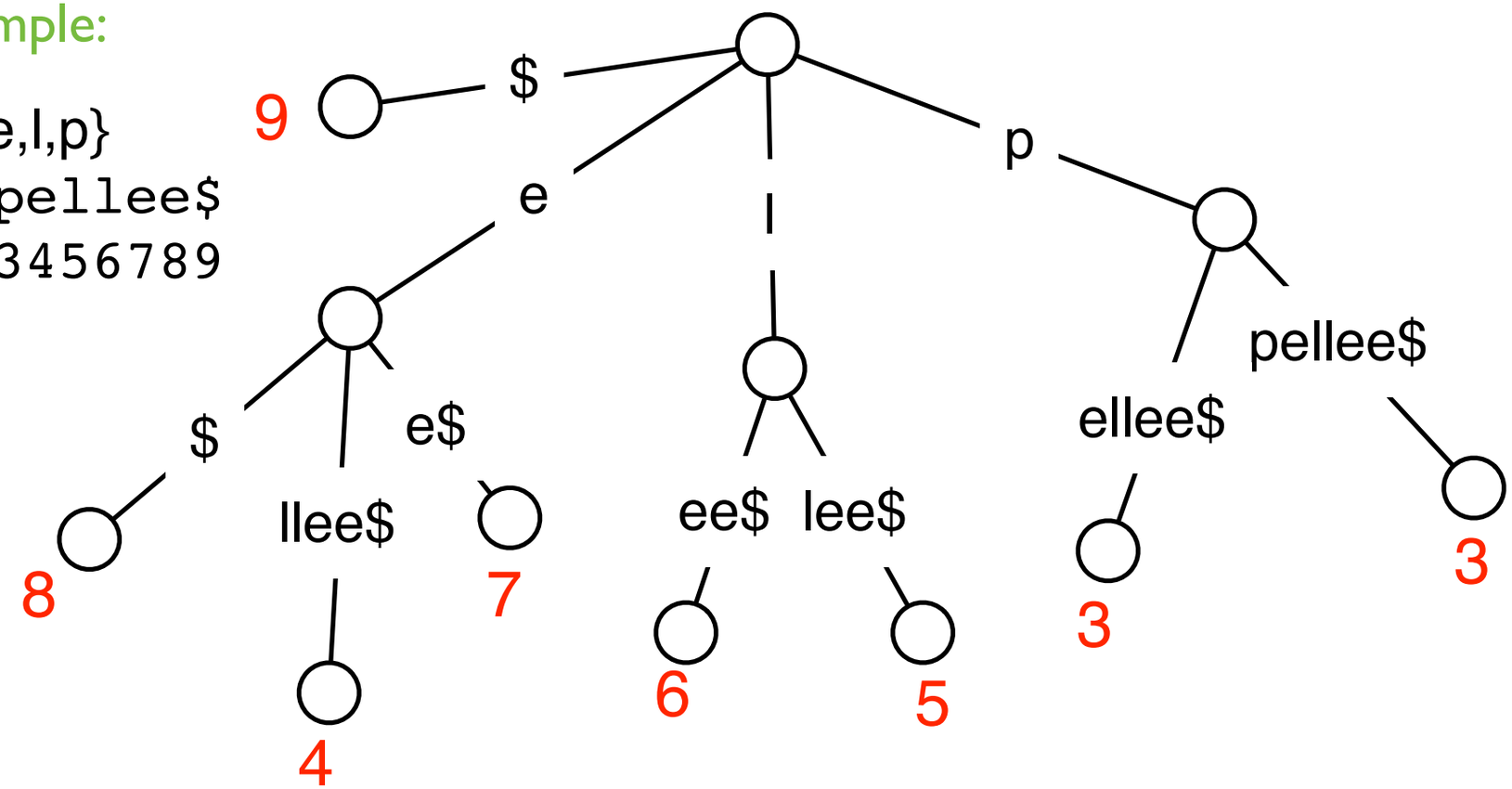
- Suffix Tree ⇒ Suffix Array ⇒ BWT.

Ordered suffix tree for previous example:

$\Sigma = \{\$,e,l,p\}$

s = appellee$

123456789



Computing BWT in O(n) time

- Easy $O(n^2 \log n)$-time algorithm to compute the BWT (create and sort the BWT matrix explicitly).

- Several direct O(n)-time algorithms for BWT.
  These are space efficient.

- Also can use suffix arrays or trees:

  Compute the suffix array, use correspondence between suffix array and BWT to output the BWT.

  O(n)-time and O(n)-space, but the constants are large.

Move-To-Front Coding

To encode a letter, use its index in the current list, and then move it to the front of the list.

List with all letters from the allowed alphabet →

Σ	do$oodwg
$dgow	1
d$gow	13
od$gw	132
$odgw	1322
o$dgw	13220
o$dgw	132202
do$gw	1322024
wdo$g	13220244 = MTF(do$oodwg)

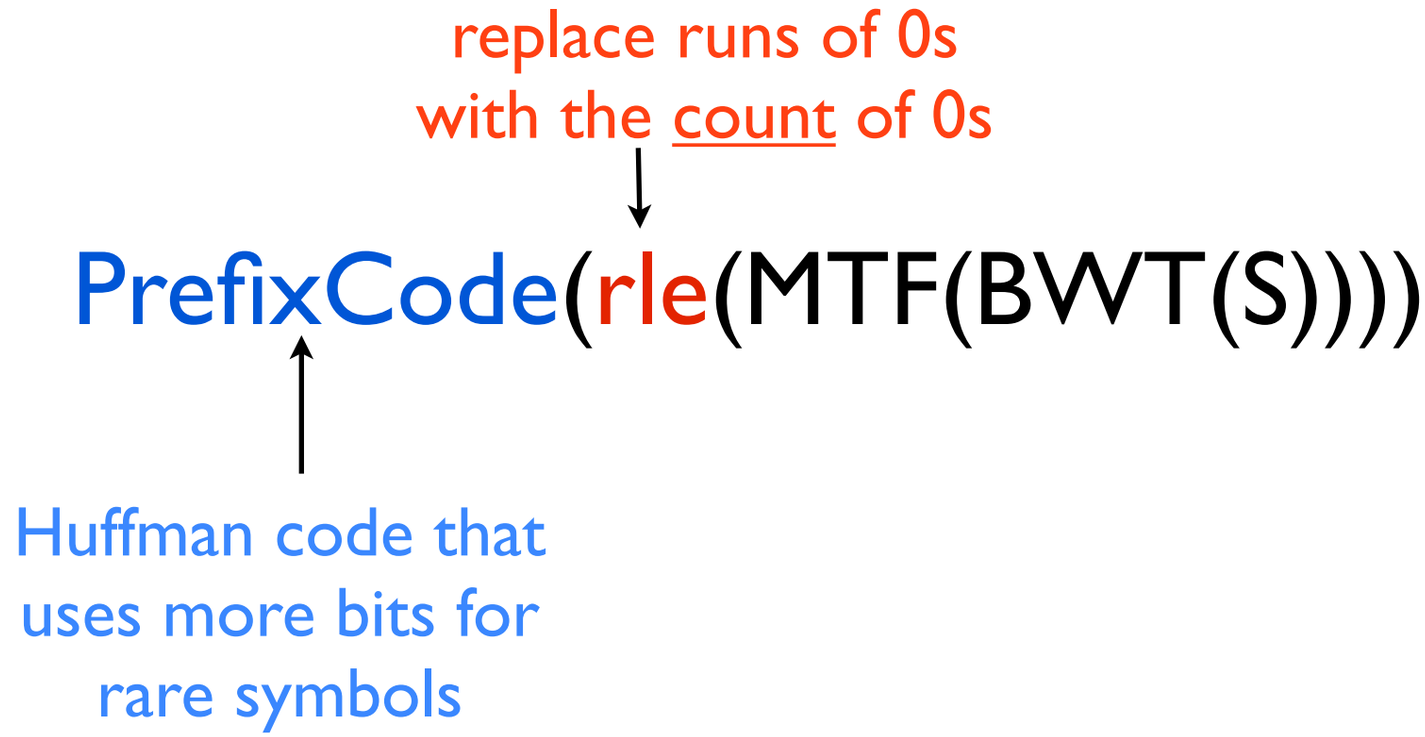
Benefits:

- Runs of the same letter will lead to runs of 0s.
- Common letters get small numbers, while rare letters get big numbers.

Compressing BWT Strings

Lots of possible compression schemes will benefit from preprocessing with BWT (since it tends to group runs of the same letters together).

One good scheme proposed by Ferragina & Manzini:



Pseudocode for CountingOccurrences in BWT w/o stored LF mapping

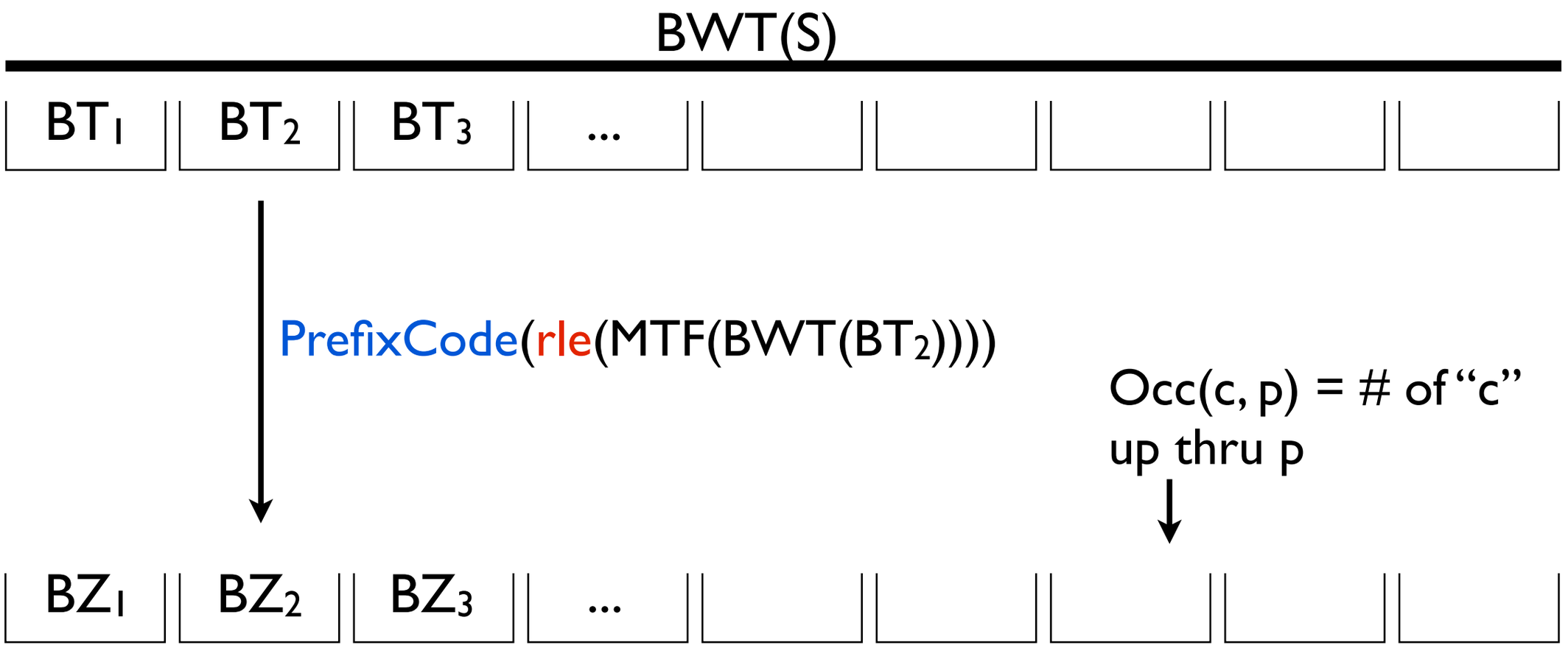
```
function Count(S_bwt, P):
    c = P[p], i = p
    sp = C[c] + 1; ep = C[c+1]

    while (sp ≤ ep) and (i ≥ 2) do
      c = P[i-1]
      sp = C[c] + Occ(c, sp-1) + 1
      ep = C[c] + Occ(c, ep)
      i = i - 1

  if ep < sp then
    return "not found"
  else
    return ep - sp + 1
```

C[c] = index into first column where the "c"s begin.

Occ(*c*, *p*) = # of of *c* in the first *p* characters of BWT(S), aka the LF mapping.

Computing Occ in Compressed String

Break BWT(S) into blocks of length L (we will decide on a value for L later):

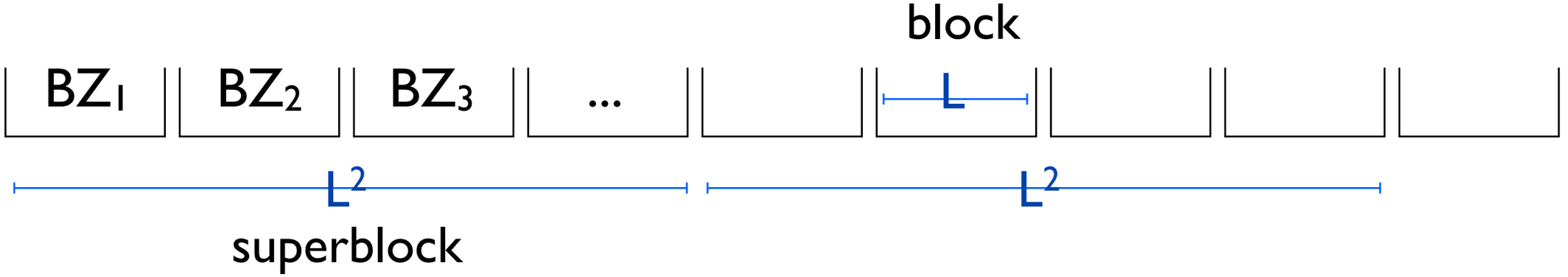


Assumes every run of 0s is contained in a block [just for ease of explanation].

We will store some extra info for each block (and some groups of blocks) to compute Occ(c, p) quickly.

Extra Info to Compute Occ

block: store |∑|-long array giving # of occurrences of each character up thru and including this block *since the end of the last super block*.



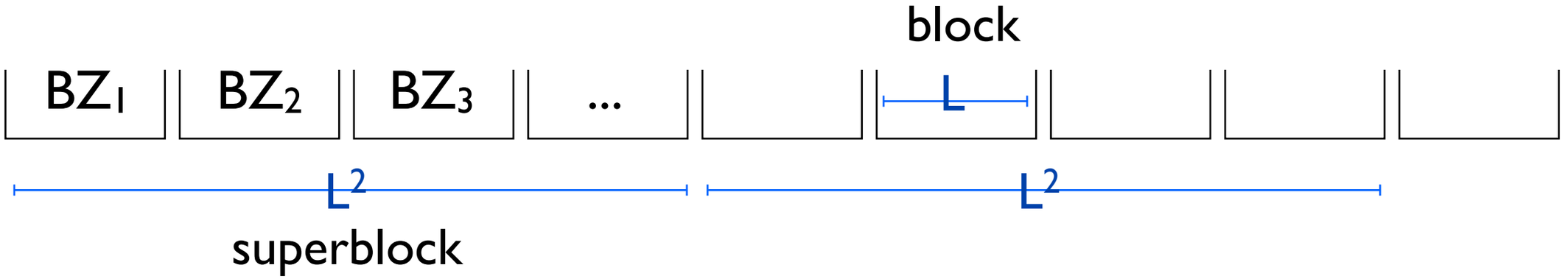
superblock: store |∑|-long array giving # of occurrences of each character up thru *and including* this superblock

Extra Info to Compute Occ

u = compressed length
Choose $L = O(\log u)$

u/L blocks, each array is $|\Sigma| \log L$ long $\Rightarrow$
$\frac{u}{L} \log L = \frac{u}{\log u} \log \log u$ total space.

block: store $|\Sigma|$-long array giving # of occurrences of each character up thru and including this block *since the end of the last super block.*



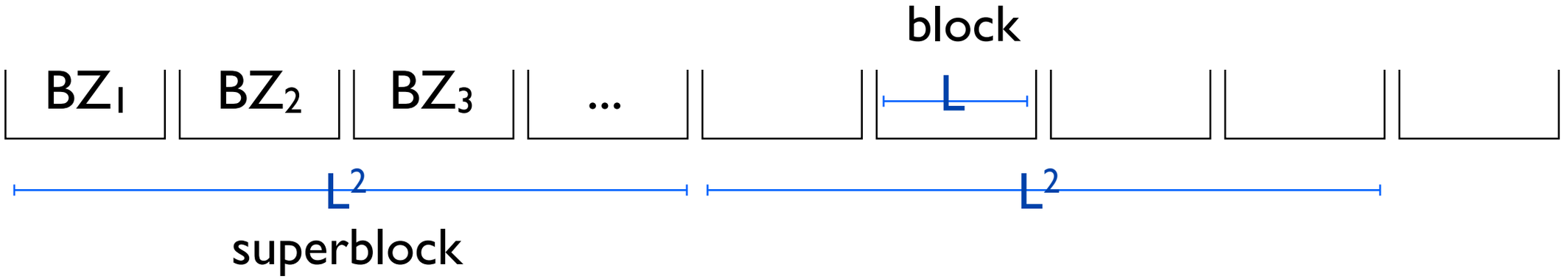
superblock: store $|\Sigma|$-long array giving # of occurrences of each character up thru *and including* this superblock

Extra Info to Compute Occ

u = compressed length
Choose $L = O(\log u)$

u/L blocks, each array is $|\Sigma| \log L$ long $\Rightarrow$
$\frac{u}{L} \log L = \frac{u}{\log u} \log \log u$ total space.

block: store $|\Sigma|$-long array giving # of occurrences of each character up thru and including this block *since the end of the last super block.*

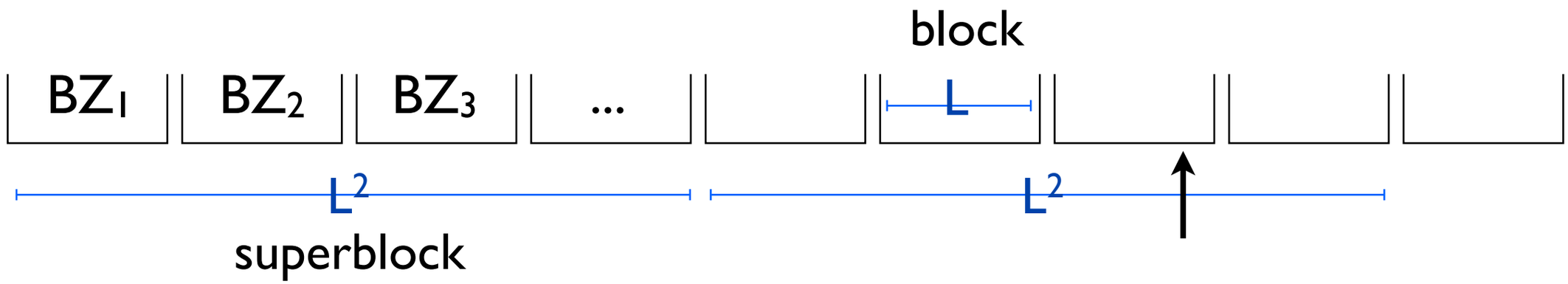


superblock: store $|\Sigma|$-long array giving # of occurrences of each character up thru *and including* this superblock

u/L^2 superblocks, each array is $|\Sigma| \log u$ long
$\Rightarrow \quad \frac{u}{(\log u)^2} \log u = \frac{u}{\log u}$ total space.

Extra Info to Compute Occ

u = compressed length
Choose $L = O(\log u)$



Occ(c, p) = # of "c" up thru p:
sum value at last superblock, value at end of previous block, but then need to handle *this block.*

Store an array: $M[c, k, BZ_i, MTF_i]$ = # of occurrences of c through the kth letter of a block of <u>type</u> (BZ_i, MTF_i).

Size: $O(|\Sigma|L2^L|\Sigma|) = O(L2^{L'}) = O(u^c \log u)$ for $c < 1$ (since the string is compressed)

Recap

BWT useful for searching and compression.

BWT is *invertible*: given the BWT of a string, the string can be reconstructed!

BWT is computable in O(n) time.

Close relationships between Suffix Trees, Suffix Arrays, and BWT:

- Suffix array = order of the suffix numbers of the suffix tree, traversed left to right
- BWT = letters at positions given by the suffix array entries - 1

Even after compression, can search string quickly.

Clustering

COMP462/561: Computational Biology Methods

Fall 2016

M & W: 10:00 am – 11:30 am

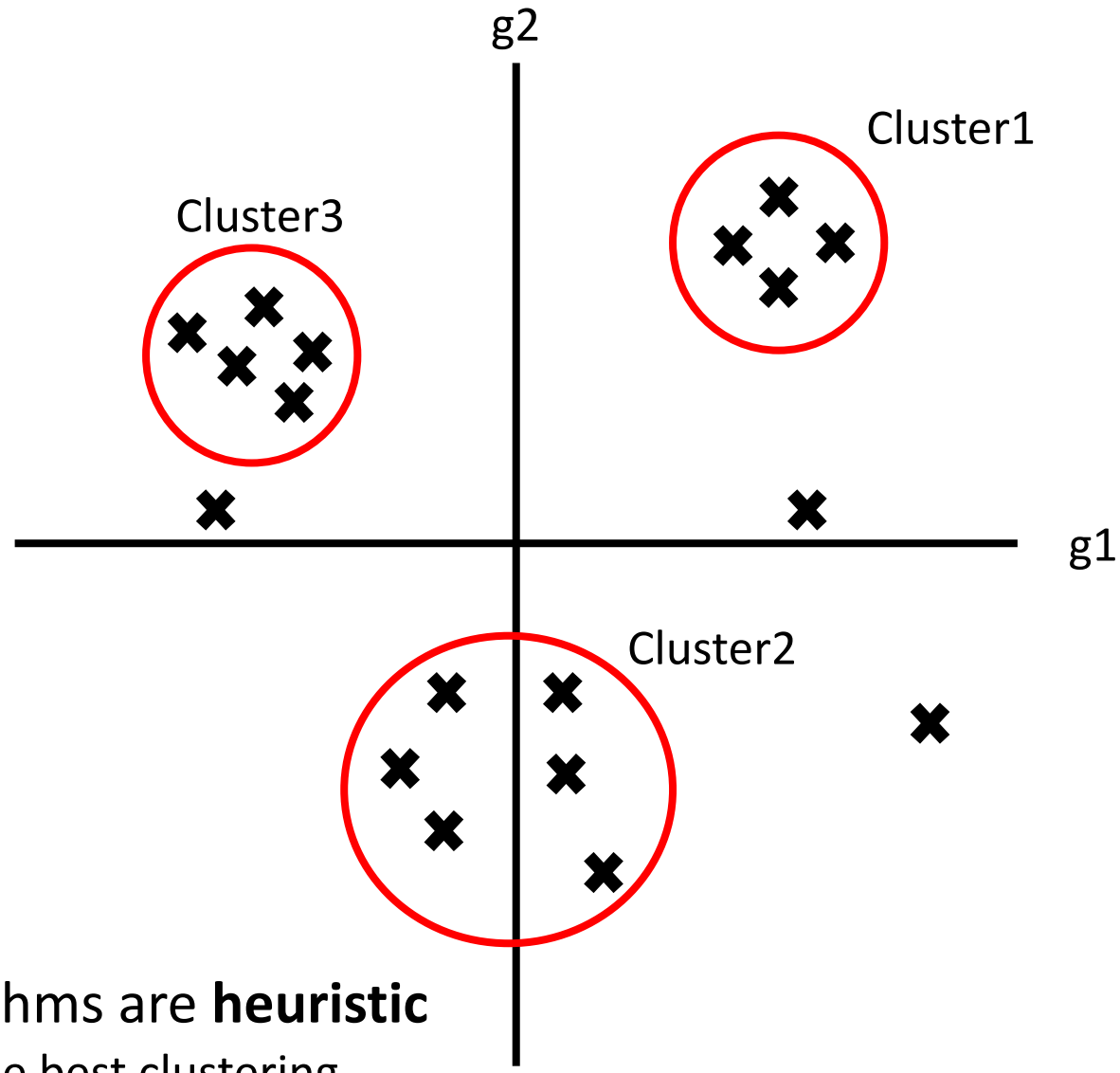
*Based on Course Notes by Dr. Mathieu Blanchette

Motivation

Given: A collection of unlabeled samples $X_i \ldots X_n$, where X_i represents the data for sample i

Goal: Partition samples into groups that are similar within themselves but dissimilar between

	X_1	…	X_n
gene1			
gene2			
gene3			
…			
$gene_{k-1}$			
$gene_k$			

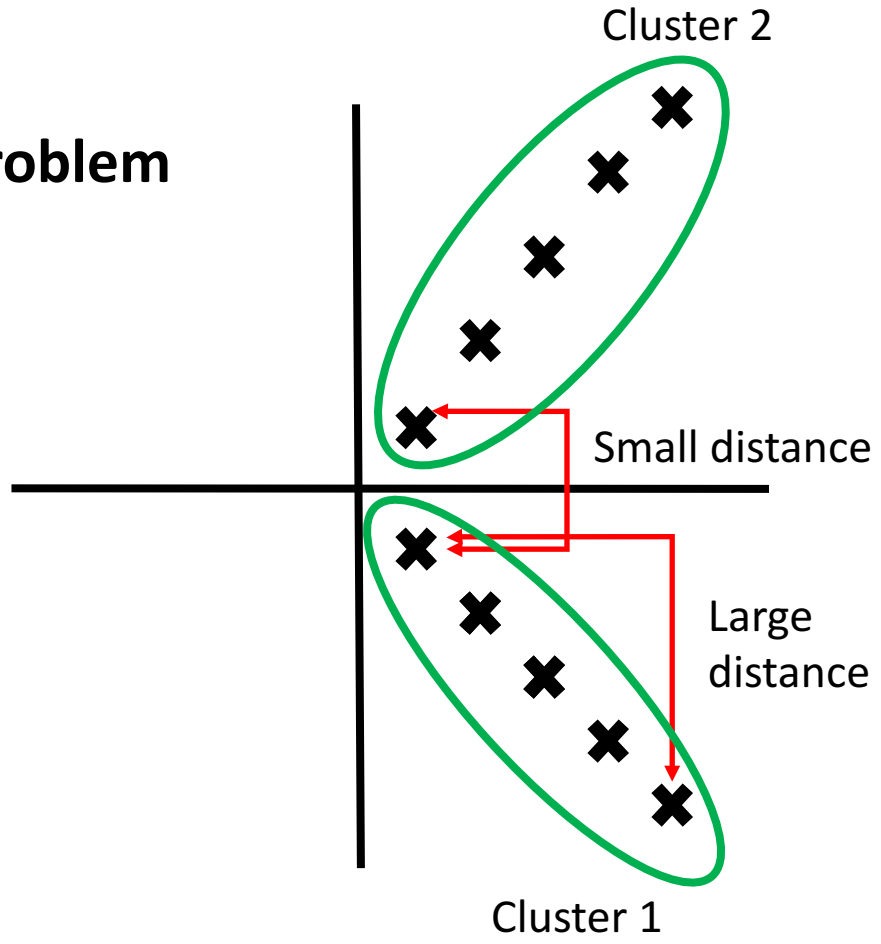


- All the clustering algorithms are **heuristic**
  - They don’t guarantee the best clustering

Similarity (or Distance) Measures

Given: Two expression profiles, X_i and X_j

Problem



Euclidean Distance

$$d_E(X_i, X_j) = \sqrt{\sum_{g=1\ldots k} (X_{i,g} - X_{j,g})^2}$$

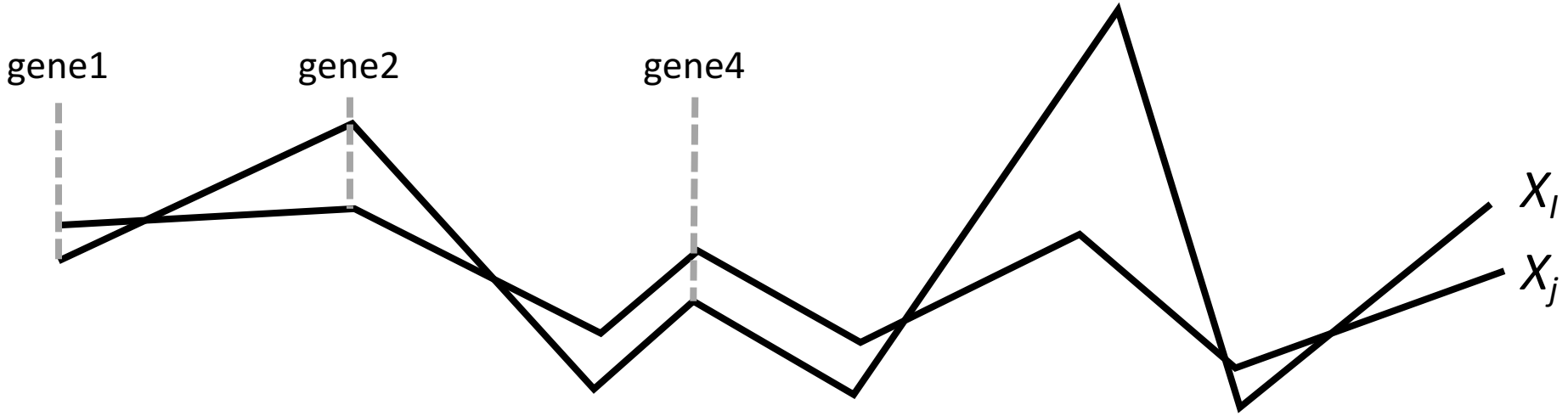
Pearson Correlation Coefficient

Similarity Measure

$$Sim(X_i, X_j) = \frac{Cov(X_i, X_j)}{\sqrt{Var(X_i) \times Var(X_j)}}$$

$$= \frac{\sum (X_i(g) - \bar{X}_i)(X_j(g) - \bar{X}_j)}{\sqrt{(\sum (X_i(g) - \bar{X}_i)^2) \times (\sum (X_j(g) - \bar{X}_j)^2)}}$$

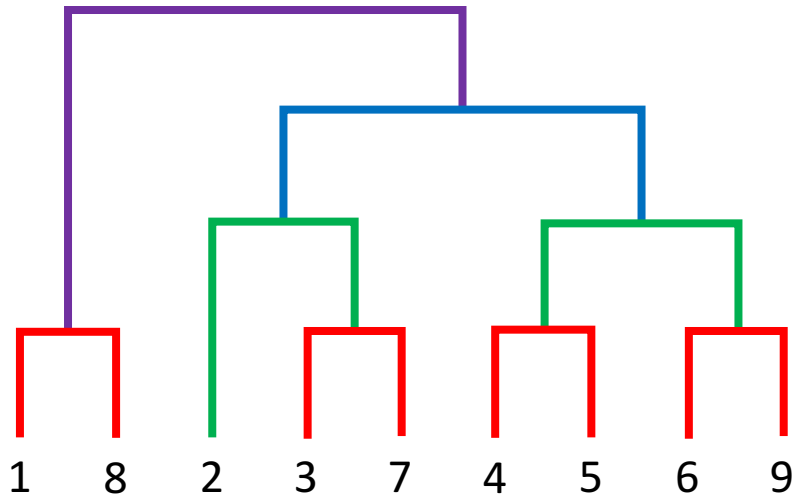
Pearson Correlation Coefficient Cont'd



- Different expression level
  - But always goes in the same direction

Hierarchical Clustering

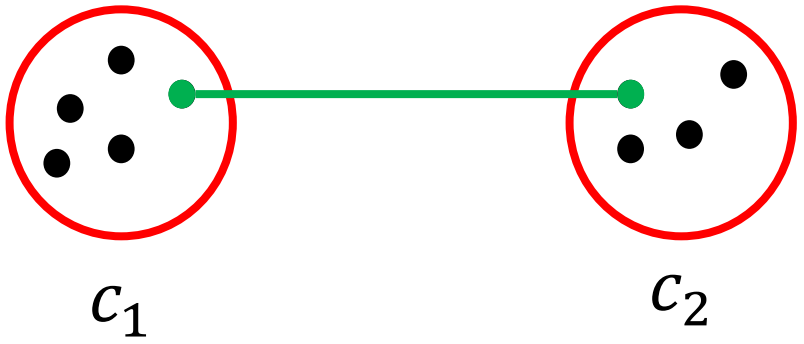
1. Start with each data point in its own cluster
2. Find the two clusters that are the closest and merge them
3. Repeat step two until all data points belong to a single cluster



Measuring Similarity Between Clusters

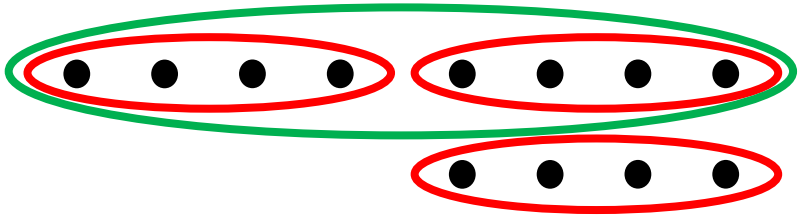
1) Single Linkage approach

$$Sim(c_1, c_2) = max_{x \in c_1, y \in c_2} \{sim(x, y)\}$$



Problem

- Given the following data points:

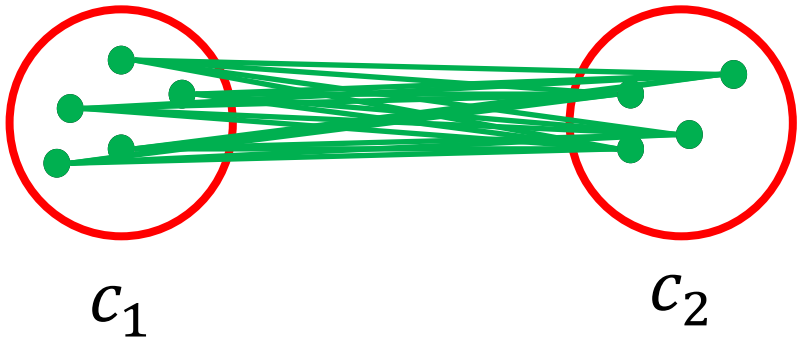


- Apply single linkage approach to clustering
- Get long and skinny clusters by having one point near the others
  - Shouldn't the two clusters on the right pair better together?

Measuring Similarity Between Clusters

2) Average linkage

$$Sim(c_1, c_2) = \frac{1}{|c_1| \cdot |c_2|} \sum_{x \in c_1, y \in c_2} Sim(x, y)$$



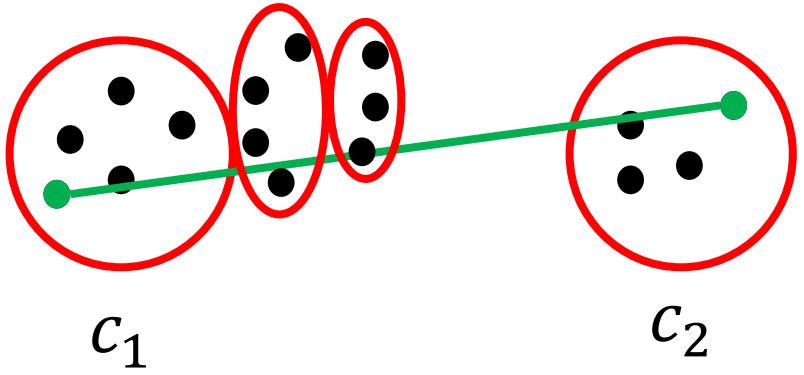
Take all pairs!

Measuring Similarity Between Clusters

3) Complete linkage

Makes very compact clusters

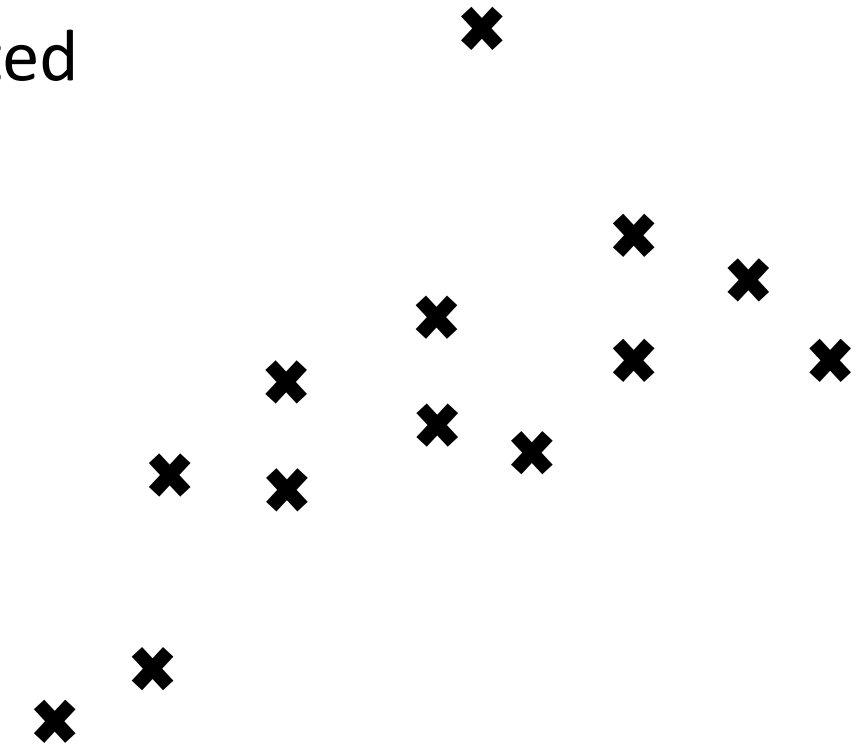
$$Sim(c_1, c_2) = min_{x \in c_1, y \in c_2}\ sim(x, y)$$



K-Means Algorithm

- '*k*' is the number of clusters desired / expected
- Each cluster has a centroid

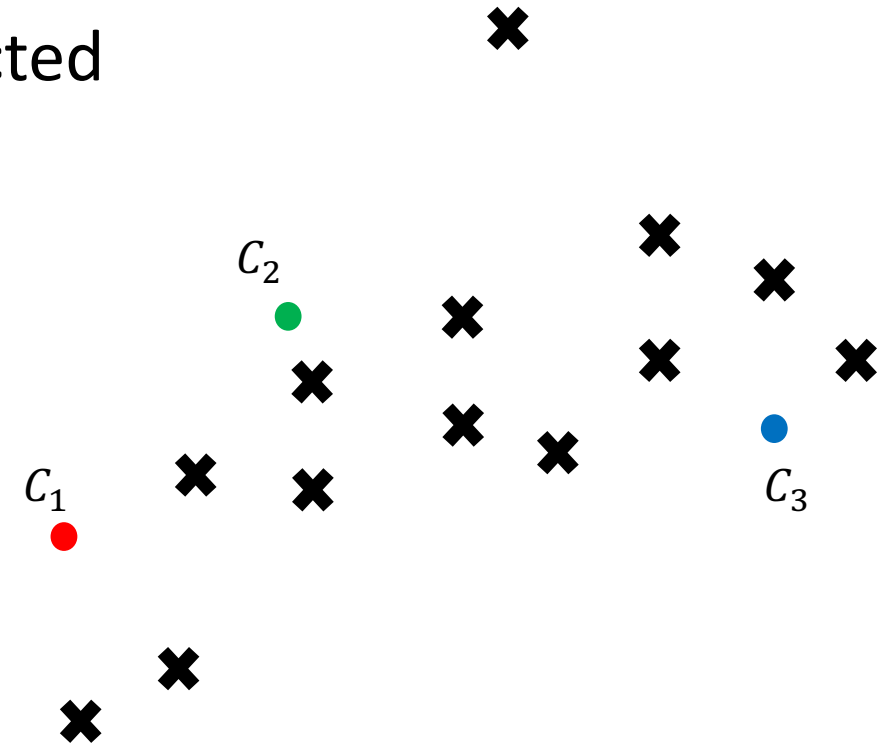
1. Randomly choose k centroids
2. Assign data points to nearest centroid
3. Move centroid to center of cluster
4. Repeat 2-4. Stop when no change to data point assignment



K-Means Algorithm

- '*k*' is the number of clusters desired / expected
- Each cluster has a centroid

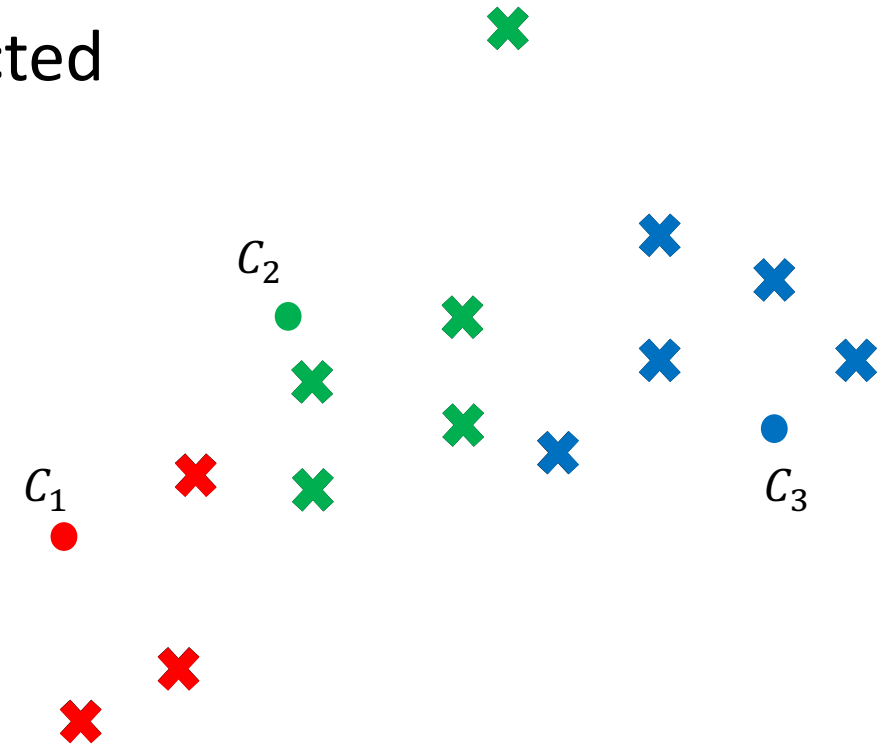
1. Randomly choose k centroids
2. Assign data points to nearest centroid
3. Move centroid to center of cluster
4. Repeat 2-4. Stop when no change to data point assignment



K-Means Algorithm

- '*k*' is the number of clusters desired / expected
- Each cluster has a centroid

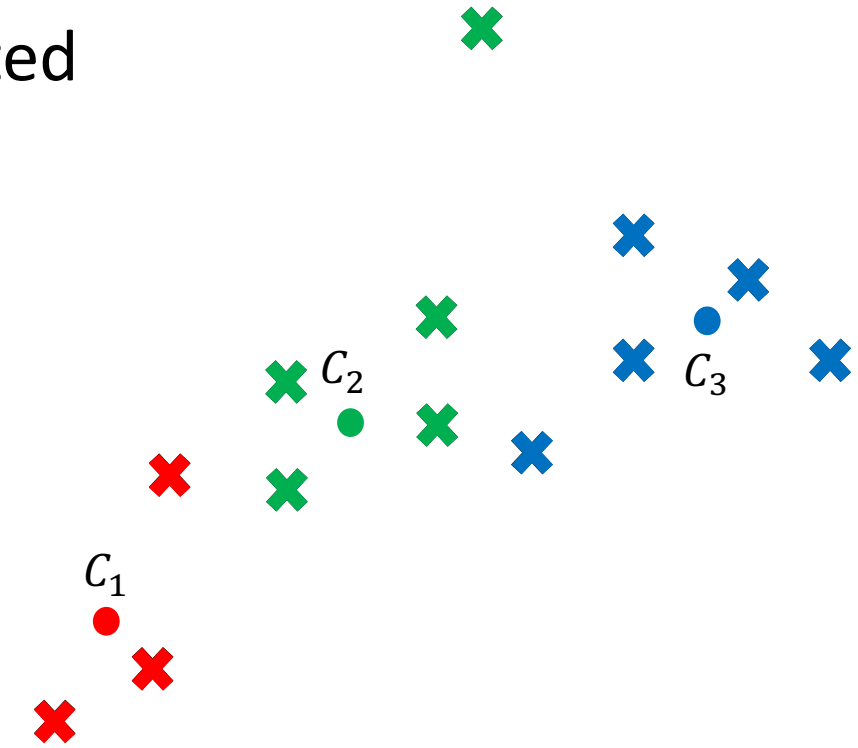
1. Randomly choose k centroids
2. Assign data points to nearest centroid
3. Move centroid to center of cluster
4. Repeat 2-4. Stop when no change to data point assignment



K-Means Algorithm

- '*k*' is the number of clusters desired / expected
- Each cluster has a centroid

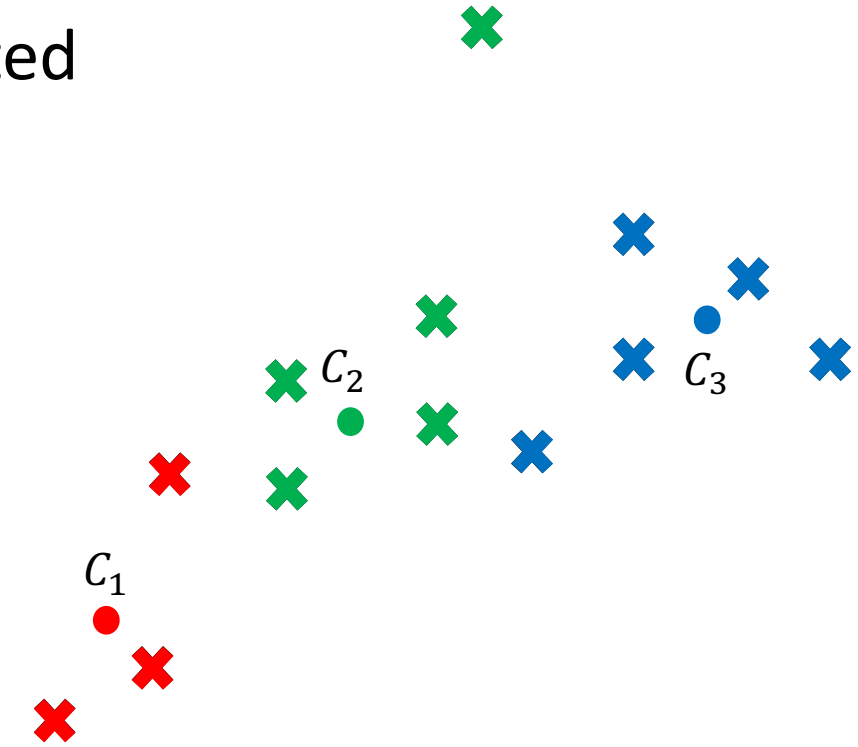
1. Randomly choose k centroids
2. Assign data points to nearest centroid
3. Move centroid to center of cluster
4. Repeat 2-4. Stop when no change to data point assignment



K-Means Algorithm

- '*k*' is the number of clusters desired / expected
- Each cluster has a centroid

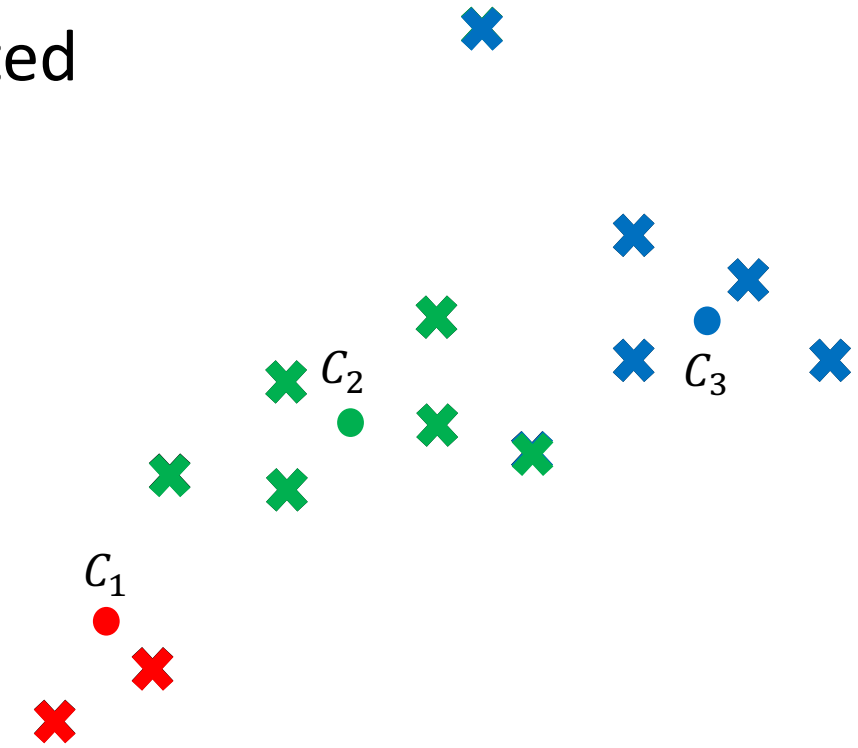
1. Randomly choose k centroids
2. Assign data points to nearest centroid
3. Move centroid to center of cluster
4. Repeat 2-4. Stop when no change to data point assignment



K-Means Algorithm

- '*k*' is the number of clusters desired / expected
- Each cluster has a centroid

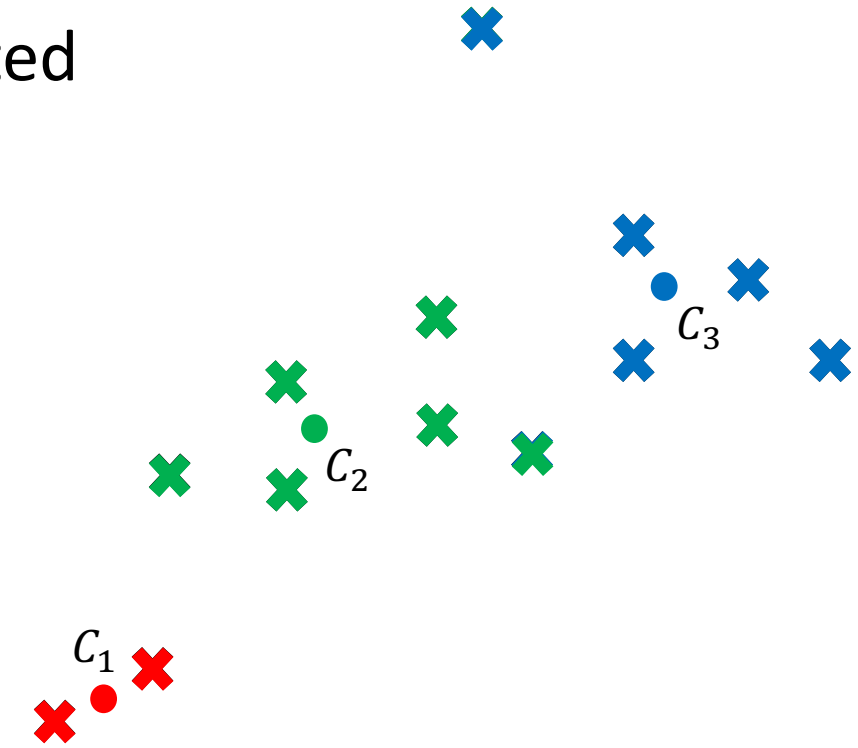
1. Randomly choose k centroids
2. Assign data points to nearest centroid
3. Move centroid to center of cluster
4. Repeat 2-4. Stop when no change to data point assignment



K-Means Algorithm

- '*k*' is the number of clusters desired / expected
- Each cluster has a centroid

1. Randomly choose k centroids
2. Assign data points to nearest centroid
3. Move centroid to center of cluster
4. Repeat 2-4. Stop when no change to data point assignment



Cluster Validation

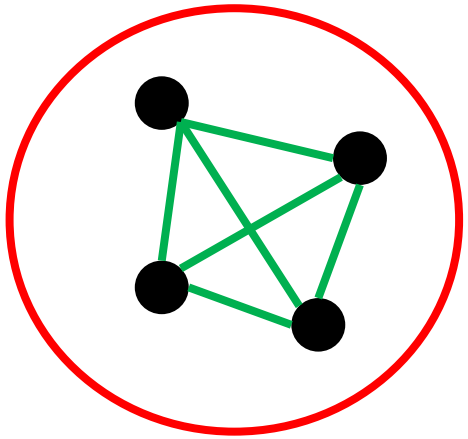
- **Cohesion:** measures how closely related data points in a cluster are (i.e., within cluster Sum of Squares [WSS])

$$WSS = \sum_{i} \sum_{x \in c_i} \|x - m_i\|^2$$

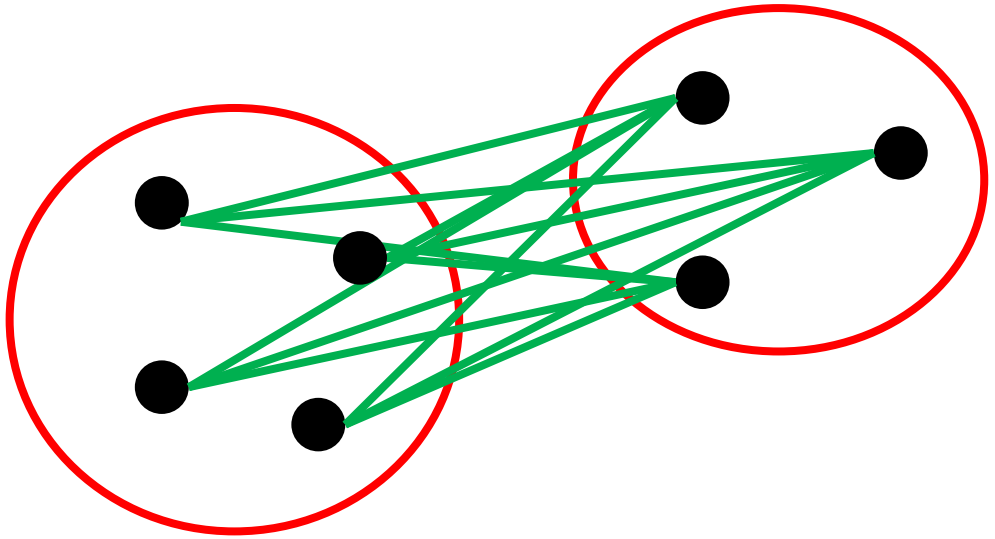
- **Separation:** measures how distinct or well-separated a cluster is from others (i.e., between cluster Sum of Squares [BSS])

$$BSS = \sum_{i} \sum_{j} |c_i| \cdot |c_j| \cdot \|m_i - m_j\|^2$$

Cohesion and Separation



Cohesion



Separation

Corrected UPGMA algorithm
Given: a distance matrix D with n species:

1) Initialize n clusters, $C_1,\ldots,C_n$, each with a single species in it. Create a leaf node for each of the clusters.

2) Define the distance between two clusters as the average pairwise distance between members of the two clusters:

$$d(C_i, C_j) = \frac{\sum_{a \in C_i} \sum_{b \in C_j} D(a,b)}{|C_i| * |C_j|}$$

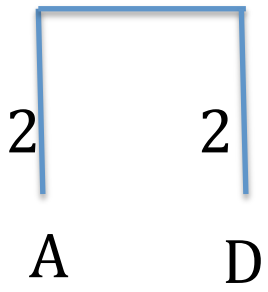
3) Repeat:

3.1 Pick the two clusters C_i and C_j such that d(C_i,C_j) is minimized.
3.2 Create a new cluster $C_k = C_i \cup C_j$
3.3 Create a new node in the tree, make it the parent of nodes *i* and *j*, at height d(C_i,C_j)/2.
3.4 Add cluster C_k to the list of clusters, and remove clusters C_i and C_j.

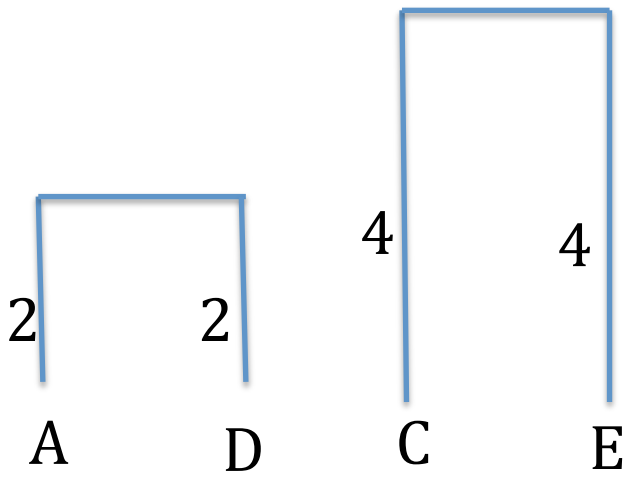
Example: Consider the following distance matrix D:

	A	B	C	D	E
A	-	16	16	4	16
B		-	10	16	10
C			-	16	8
D				-	16
E					-

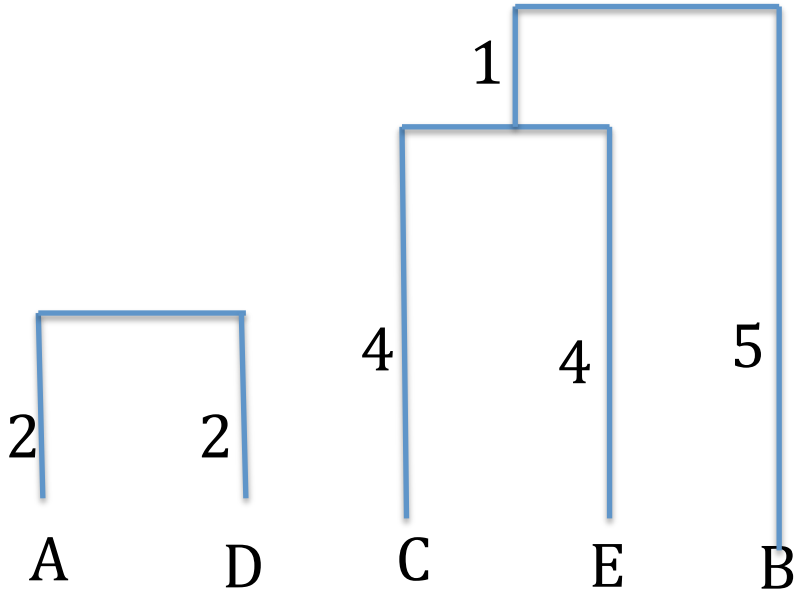
First set C1={A}, C2={B}, C3={C}, C4={D}, C5={E}.
The pair with the smallest distance is (C1,C4). Merge them to obtain C6={A,D} and create their parent node at distance d(C1,C4)/2 = 4/2 = 2 from each, to obtain:



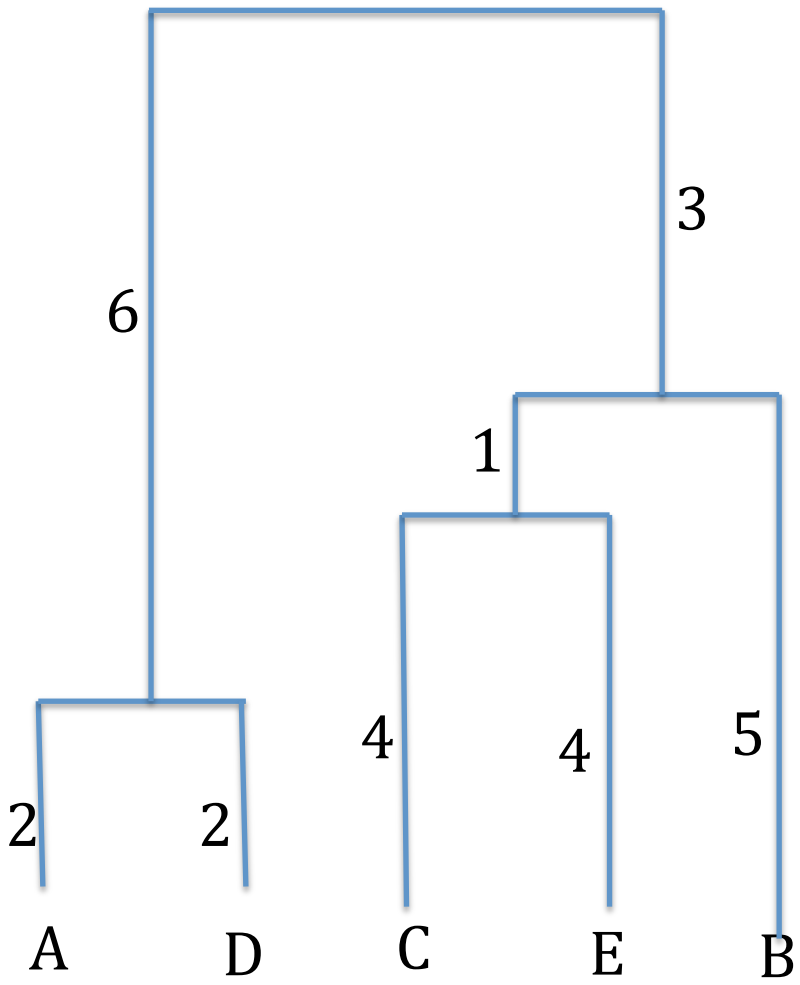
The next pair of clusters that is the closest is C3 and C5, with d(C3,C5)=8. Merge them to obtain C7={C,E} and create their parent node at distance d(C3,C5)/2 = 8/2 = 4 from each, to obtain:



The next pair of clusters that is the closest is C7={C,E} and C2={B}, with d(C7,C2)=(10+10)/2=10. Merge them to obtain C8={C,E,B} and create their parent node at height d(C7,C2)/2 = 10/2 = 5, to obtain:



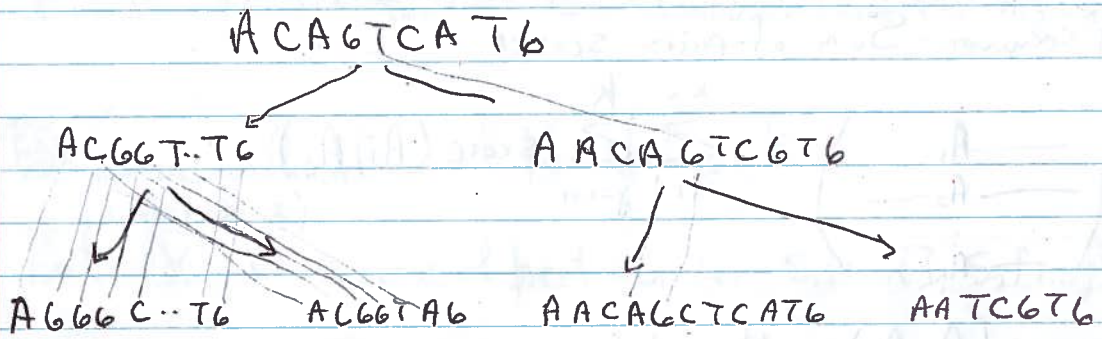
There are only two clusters C6 and C8. Merge them and place their parent node at height d(C6,C8)/2 = ((16+16+16+16+16+16+16)/6) /2 = 8, to obtain:

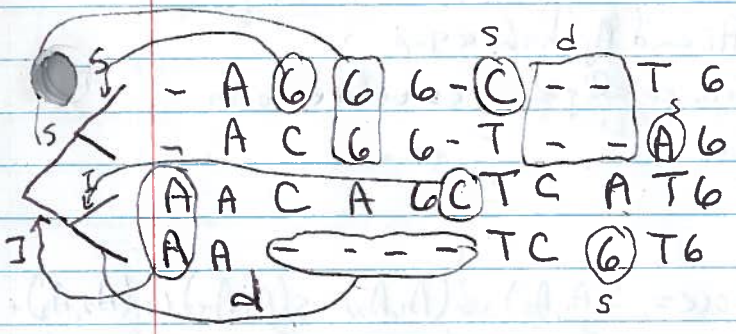

6
3
1
4
4
5
2
2
A
D
C
E
B

4522 Pontiac
→ 3452

Notes: Local alignment problem only makes sense when $\sum_{a \in \Sigma, b \in \Sigma} M(a,b) < 0$

Multiple seq. alignment.





Biological problem: Given | n sequences
Phylogenetic tree T, where each sequence is assigned to a leaf

Find: Multiple alignment where all nucleotides derived from the same common ancestor are aligned to each other.

Scoring a given MSA

Goal: Find a scoring function $score: MSA \to \mathbb{R}$
such that $score(A) > score(A') \Rightarrow A$ is more likely to be biologically correct than A'

Many choices of scoring functions are possible
Most common: Sum-of-pairs score

$$Score\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_k \end{pmatrix} = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} score(A_i, A_j)$$

where $score(A_i, A_j)$ is obtained by
1. Removing positions where both A_i and A_j have a gap
2. Scoring the remaining pairwise alignments as before, with subst. Matrix M, gap penalty c.

Example:

1. A C T
2. A 6 –
3. 6 T
4. – 6 T

$Score = s(A_1, A_2) + s(A_1, A_3) + s(A_1, A_4) + s(A_2, A_3) + s(A_2, A_4) + s(A_3, A_4)$

$= -3 - 6 - 6 - 3 - 3 + 2$

$= -19$

Optimal MSA problem

Given: Sequences $S_1 \ldots S_n$
Subst. matrix M
Gap penalty c

Find: MSA for $S_1 \ldots S_n$ that achieves the highest possible

Generalization of N-W algorithm
(For n=3)

Let $X_{i,j,k}$ = score of best aln for $S_1[1 \ldots i]$, $S_2[1 \ldots j]$, $S_3[1 \ldots$

$$X_{i,j,k} = \max \begin{cases} X_{i-1,j-1,k-1} + M(S_1[i], S_2[j]) + M(S_1[i], S_3[k]) + M(S_2[j], S \\ X_{i-1,j-1,k} + M[S_1[i], S_2[j]] + 2 \cdot c \\ \vdots \\ X_{i,j,k-1} + 2c \end{cases}$$

For 3 seq. of length l
Dyn Prog table has $l \times l \times l$ entries

For n seq of length l : $O(l^n \cdot 2^n)$

↓

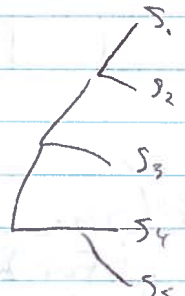
Exponential time algorithm

↗ Heuristic, with no guarantee of optimality

Progressive alignment Algo.

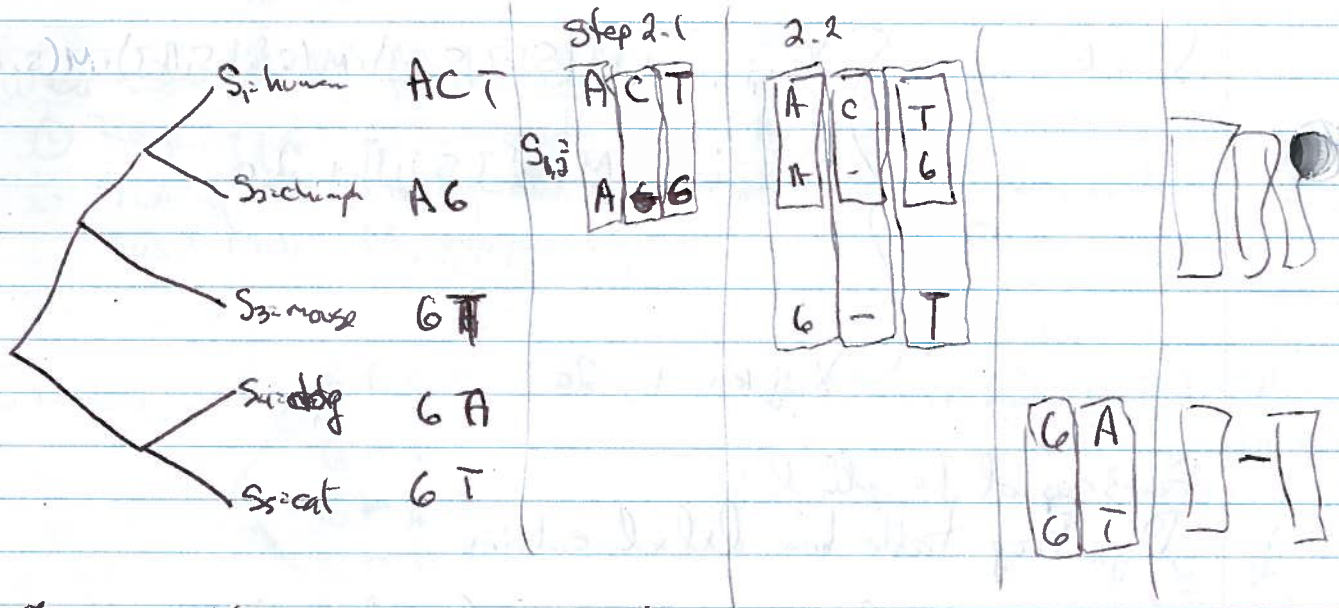
Input: $S_1 \dots S_n$, M, c

① Guess phylogenetic tree for $S_1 \dots S_n$



② For each internal node in ~~post~~ order traversal (from leaves to root)

- Find optimal alignment btw pairs of seq or alnt at two children

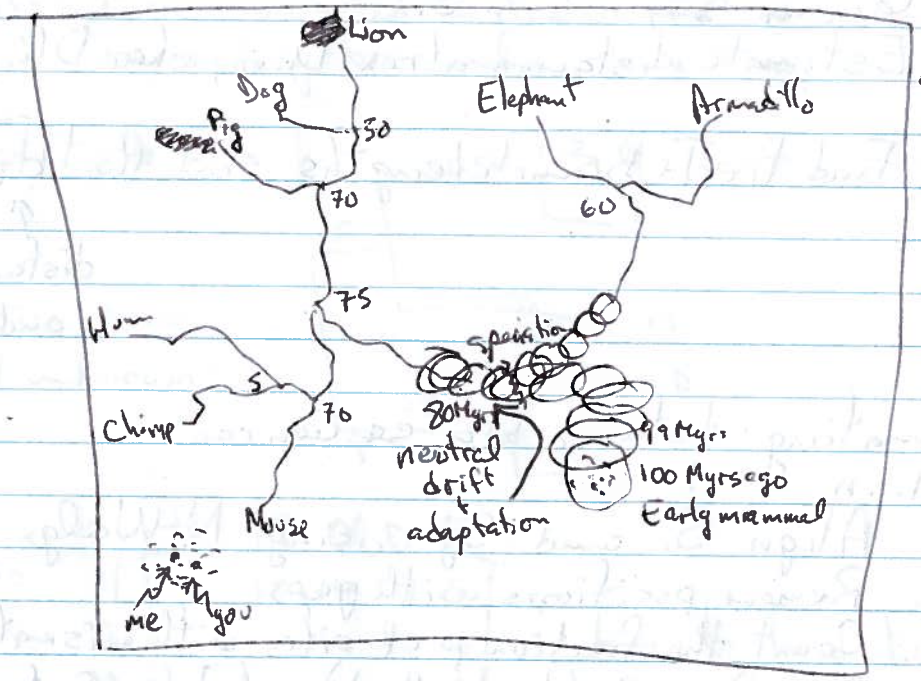


2.1. Align S_1 vs S_2 to obtain $S_{1,2}$

2.2 Align $S_{1,2}$ vs S_3 to obtain $S_{1,2,3}$

2.3 Align S_4 vs S_5 to obtain $S_{4,5}$

Sequence Evolution + Phylogenetics



$\{A, C, G, T\}^*$ = Space of all DNA sequences

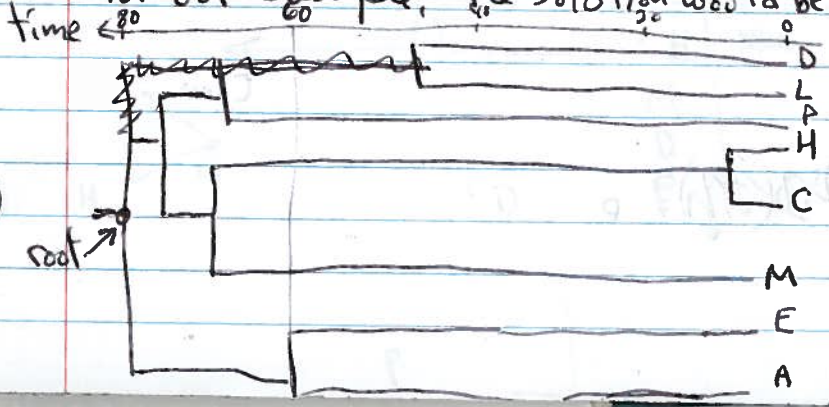
Phylogenetic Inference Problem (Biological version)

Given: DNA sequences from multiple species

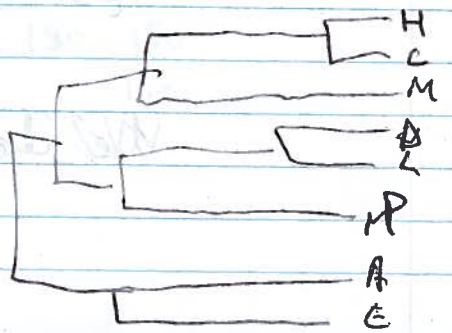
Find: Evolutionary tree that describes their evolution

→ Topology, Dating of speciation events

For our example, the solution would be:



Note: this is the same topology as



Distance-based Phylo. Inference methods

Idea: Given seq. $S_1 \ldots S_n$

(A) Estimate distance matrix $D_{n \times n}$, where $D(i,j)$ = distance btw S_i, S_j

(B) Find tree + Branch lengths such that $d_T(i,j) \approx D(i,j)$

($d_T(i,j)$: distance btw nodes i and j in tree T)

(A) Estimating distance btw sequences

For $i,j = 1 \ldots n$

1. Align S_i and S_j using N-W alg.
2. Remove positions with gaps
3. Count the fraction p of sites with mismatches
4. $D(i,j) = -\frac{3}{4}\left(\log\left(1-\frac{4}{3}p\right)\right)$ (Jukes-Cantor model)

Example

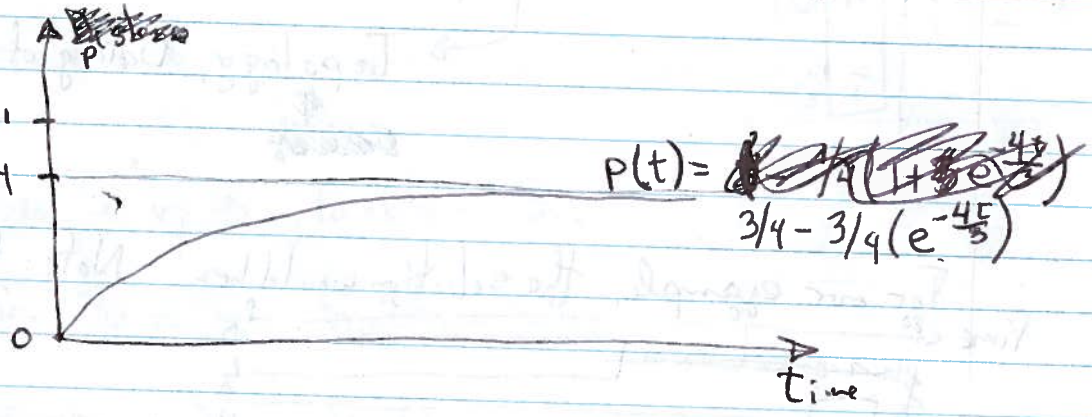
S_i: A C - - T G A C T G

S_j: A G T T T A - C A G

$\Rightarrow p = 3/7$

$D(i,j) = 0.63$

Note: If the two seqs. are very long



$p(t) = 3/4 - 3/4(e^{-\frac{4t}{3}})$

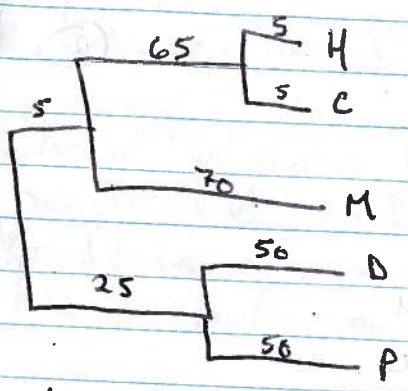
We choose $D(i,j) = p$

Under Kimura 2-parameter model with rates of transition v. transversion

Notes: More advanced distance measures consider separately transitions and transversions

Example:

True tree (unknown)



$D =$

	H	C	M	D	P
H	-	10	140	150	150
C		-	140	150	150
M			-	150	150
D				-	100
P					

How to find tree?

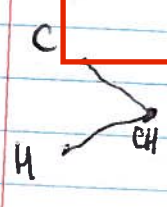
Incorrect
See appendix for corrected version and example

UPGMA

Repeat

1. Choose two closest nodes according to $D_{u,v}$
2. Merge two nodes into a new node w
3. Remove row/col u, v from matrix
4. Insert new row/col for w : $D(w,i) = D(u,i) - \frac{1}{2} D(u,v)$

	CH	M	D	P
CH	~~135~~	135	145	145
M		-	150	150
D				100
P				



M

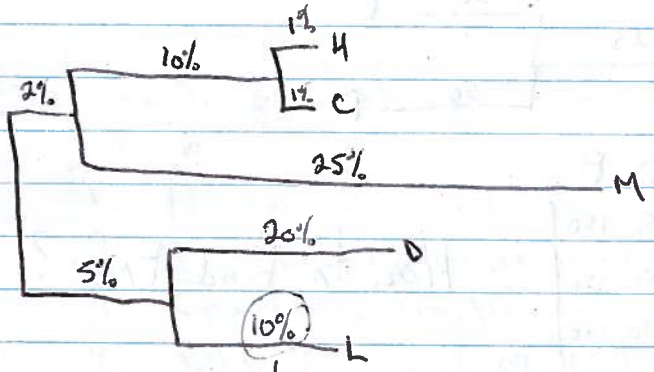
D

P

Running time: ?

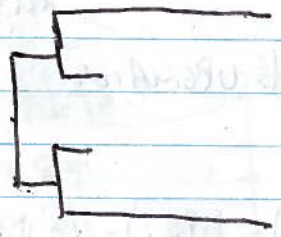
Problem: Only works if ① Distances are estimated perfectly accurately
② Mutation rate is constant

More realistic tree:



Mutation rate varies
⇒ Distances btw seq cannot be calculated in Mys but instead in Expected substitution per site

This means that 10% positions would have gotten mutated. the expected # of sub/s. per site is 0.1



⇒ UPGMA will produce the wrong tree

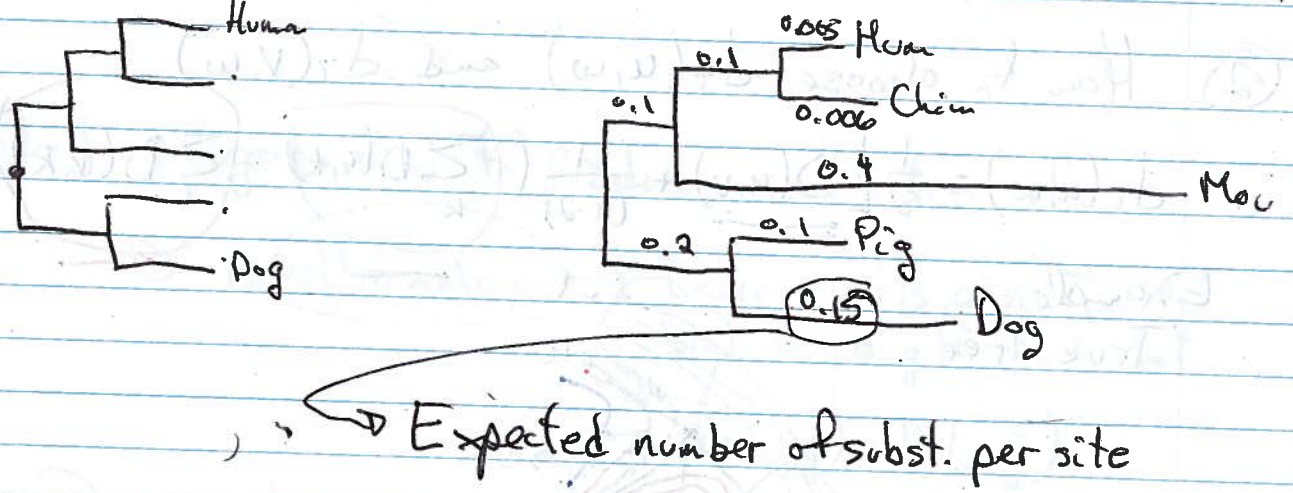
Unweighted Pair-Group Method with Arithmetic Mean

Reminder: UPGMA Algo

Input: $S_1 \ldots S_n$
Output: Tree T with branch lengths

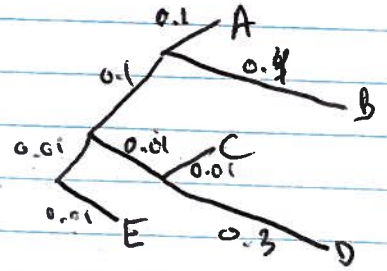
1. Estimate distance matrix D from pairwise alignments
2. Repeat
   - 2.1. Choose two nodes (u,v) to pair up
   - 2.2. Remove (u,v) from D
   - 2.3. Create new node w as ancestor of u,v
   - 2.4. Add new row/column for w in D.

UPGMA produces correct output if
1. Distances are estimated perfectly accurately
2. Mutation rate is constant

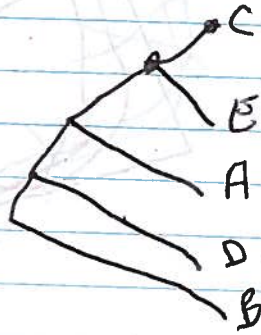


→ Expected number of subst. per site

Suppose true tree



UPGMA



Wrong Tree

Neighbor-joining Algo (Nei, Saitou)

Same as UPGMA, but different node pair selection rule

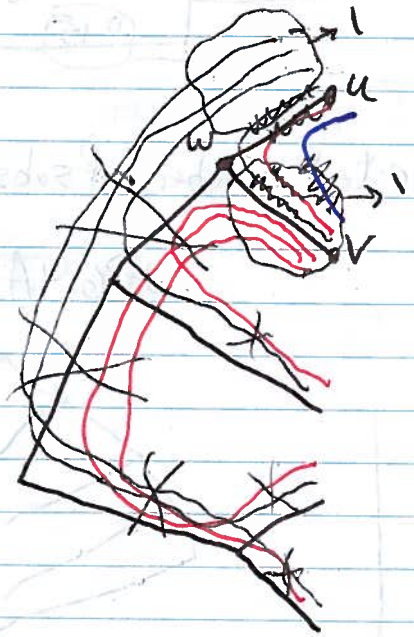
① Calculate $Q_{n\times n}$, where $Q(i,j) = \sum_{k=1}^{n} D(i,k) + \sum_{k=1}^{n} D(j,k) - (n-2)D($

Theorem: $u, v \leftarrow \underset{i,j}{\arg\max}\{Q(i,j)\}$, then nodes u and v must be a cherry

② How to choose $d_T(u,w)$ and $d_T(v,w)$

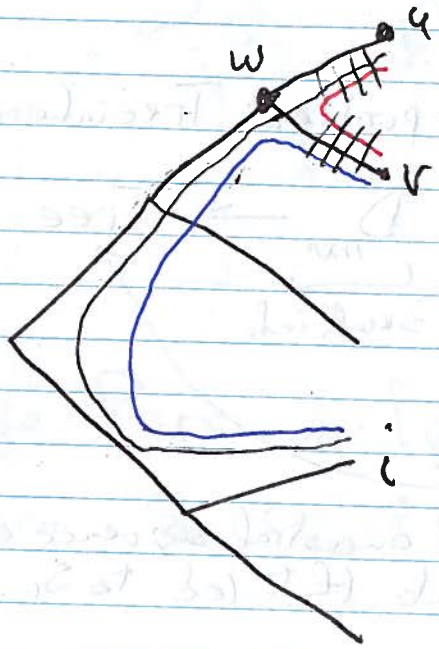
$$d_T(u,w) = \frac{1}{2}\left[D(u,v) + \frac{1}{(n-2)}\left(\sum_k D(u,k) - \sum_k D(v,k)\right)\right]$$

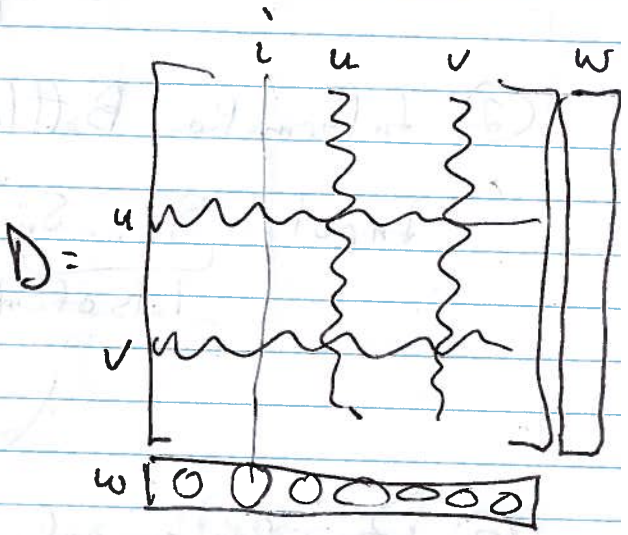
Example:
True Tree



Adding row/col for w in D

$$D(w,i) = \frac{1}{2}\left(D(u,i) + D(v,i) - D(u,v)\right)$$





N-J is guaranteed to produce correct tree if

D is ultrametric → There exists a tree with branch length such that

$$d_T(i,j) = D(i,j)$$

Running time: $O(n^4)$, but can be reduced to $O(n^3)$

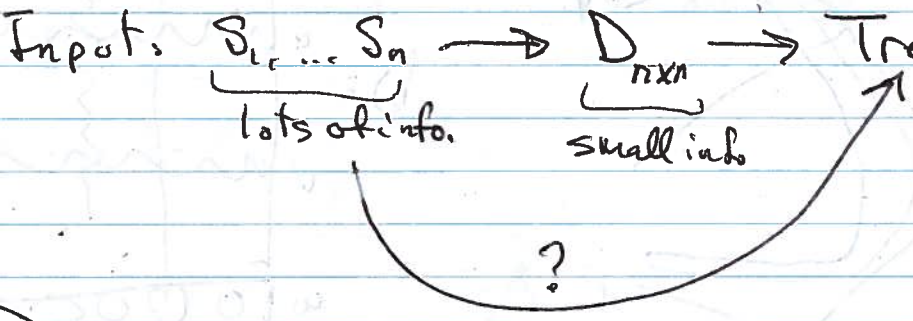
Problems with N-J algo

① If D is not ultrametric (D is not estimated ~~with~~ perfectly)

$\Rightarrow$ N-J comes with guarantees about optimality

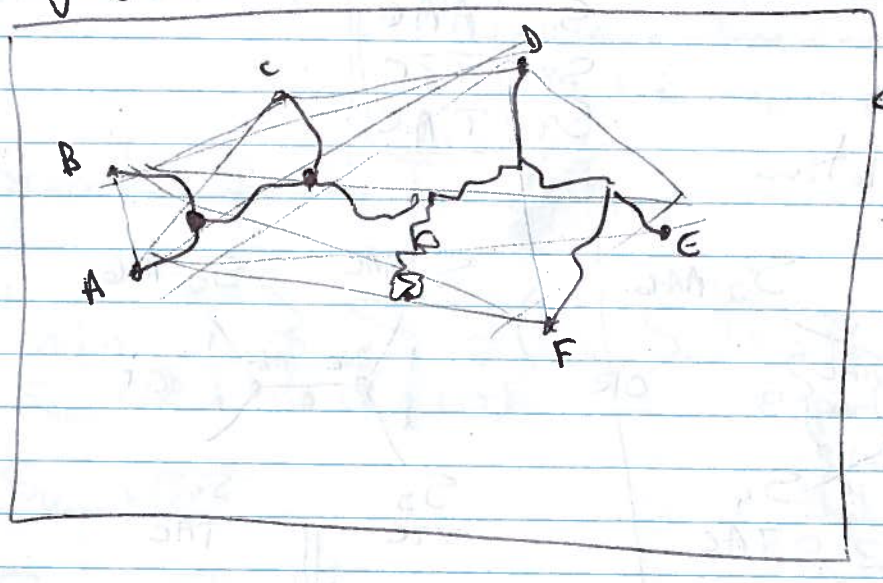
NO

② Information Bottleneck problem: Tree inferred may be inaccurate



③ Provide no info about ancestral sequence or about evolutionary events that led to $S_1 \ldots S_n$

Phylogenetic Inference - Maximum Parsimony



Large Parsimony Problem:

Given: $S_1 \ldots S_n$ of length L, in multiple sequence aln, with columns containing gaps removed

Find:
- Tree T with leaves labeled with $S_1 \ldots S_n$
- Ancestral sequence S_u for each internal node of T

such that $\sum_{(u,v) \in E(T)} d(S_u, S_v)$ is minimized

$d(S_u, S_v)$ = # of substitutions btw S_u, S_v

Example: Input:

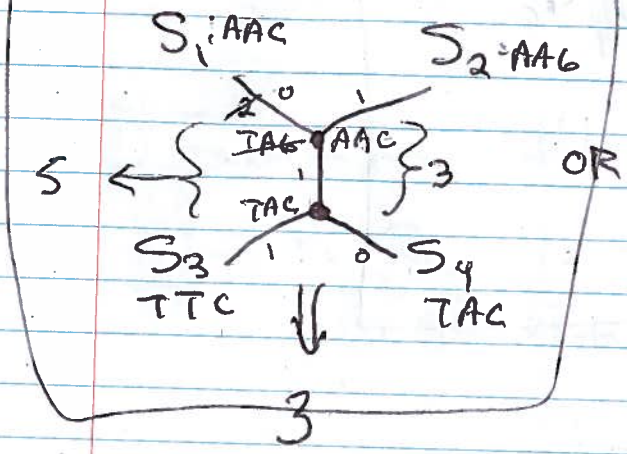
MSA

S_1: AAC
S_2: AA6
S_3: TTC
S_4: TAC

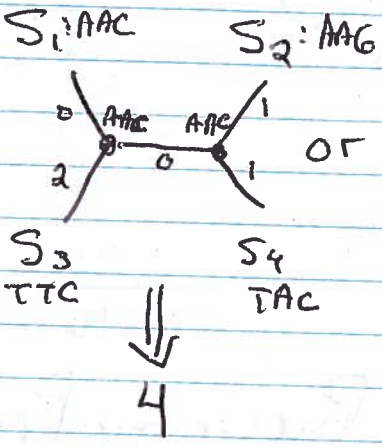
Unrooted tree

Rooted tree

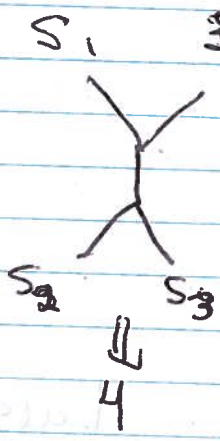
Opt-Solution



3

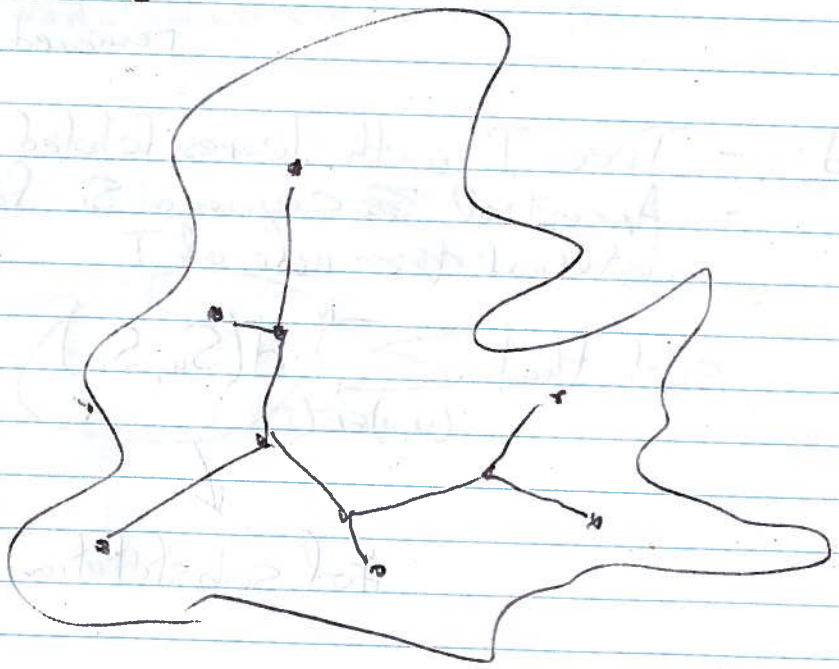


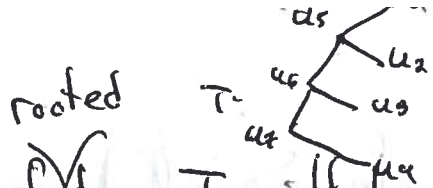
4



4

Steiner tree





Define Parsimony Score of tree T with leaves $u_1, u_2 \dots u_n$ and internal nodes $u_{n+1}, u_{n+2} \dots, u_{2n-1}$

$$\text{ParsScore}(S_1, S_2 \dots S_n, T) = \min_{\substack{S_{u_{n+1}} \in (\Sigma)^n \\ S_{u_{n+2}} \in (\Sigma)^n \\ \vdots \\ S_{u_{2n-1}} \in \Sigma^n}} \left\{ \sum_{(u,v) \in E(T)} d(S_u, S_v) \right\}$$

$\equiv$ The minimum # of substitutions performed along branches of T that can explain $S_1, S_2 \dots S_n$

$\left(4^L\right)^{n-1}$ cases $= 4^{L(n-1)}$

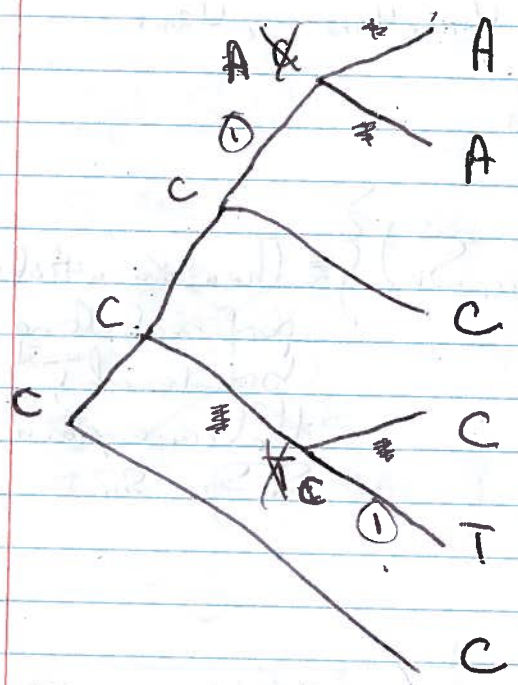
Observation: $\text{ParsScore}(S_1 \dots S_n, T) = \sum_{i=1}^{L} \text{ParsScore}(S_1[i], \dots, S_n[i], T)$

→ Small Parsimony Problem

Given: - Aligned $S_1, \dots, S_n$ of length L
- Phylogenetic tree T

Goal: Calculate $\text{ParsScore}(S_1 \dots S_n, T)$

Example: Small Parsimony Problem

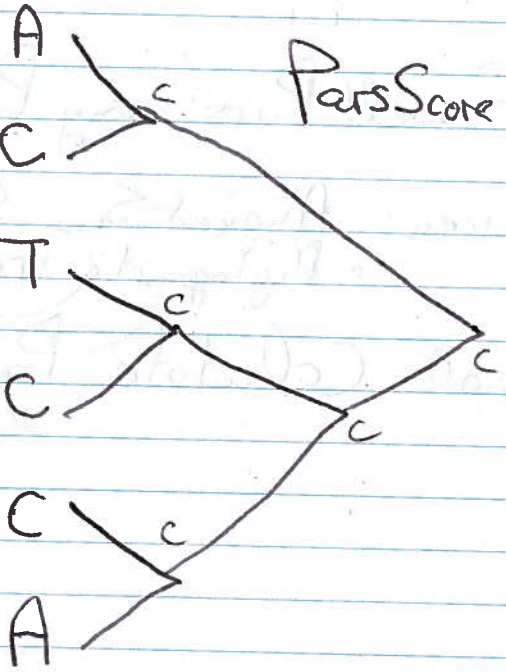


Score = 4

Score = 2 Optimal sol.

$\Rightarrow$ ParsScore(A, A, C, C, C, T, C, Tree) = 2

How to calculate $ParsScore[S_1[i], S_2[i], \ldots S_n[i], T]$?



ParsScore = 3

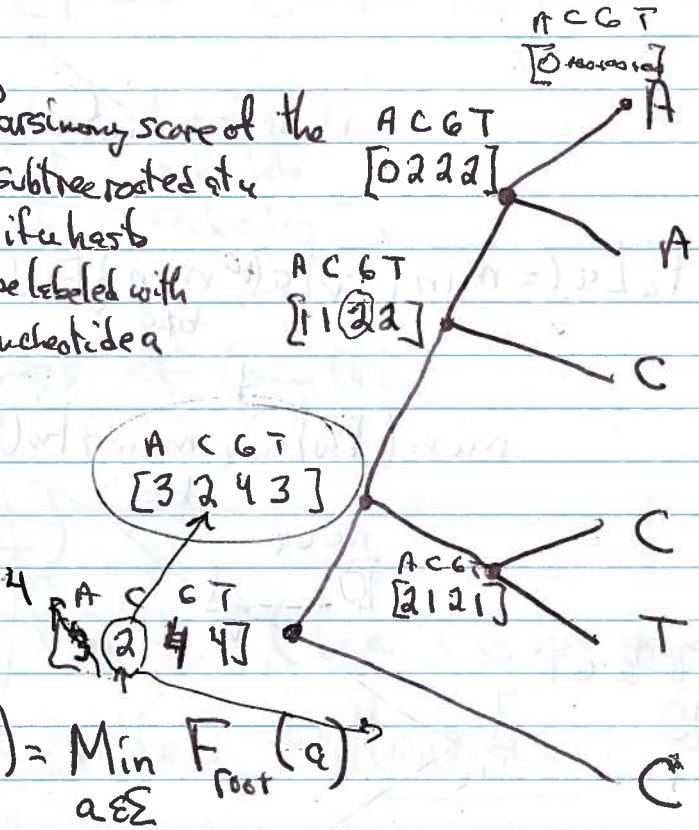
Sankoff Algorithm (1975)

Input: $\{S_1, S_2 \ldots S_n$ each of length 1
Tree T

Goal: Calculate ParsScore$(S_1 \ldots S_n, T)$

Define $F_u[a]$ = Parsimony score of the subtree rooted at u if u has b be labeled with nucleotide a

(u = node, $a \in \{A, C, G, T\}$)



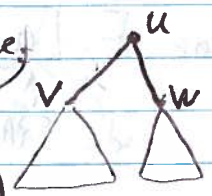
$$\text{ParsScore}(\text{Tree}) = \min_{a \in \Sigma} F_{\text{root}}(a)$$

Algo: For each node u in tree (from leaves back to the root)

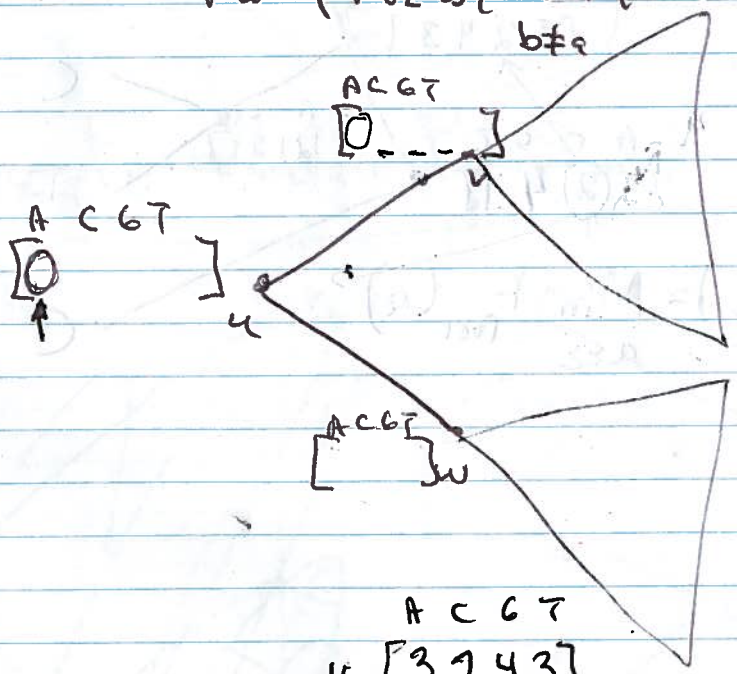
For each $a \in \{A, C, G, T\}$

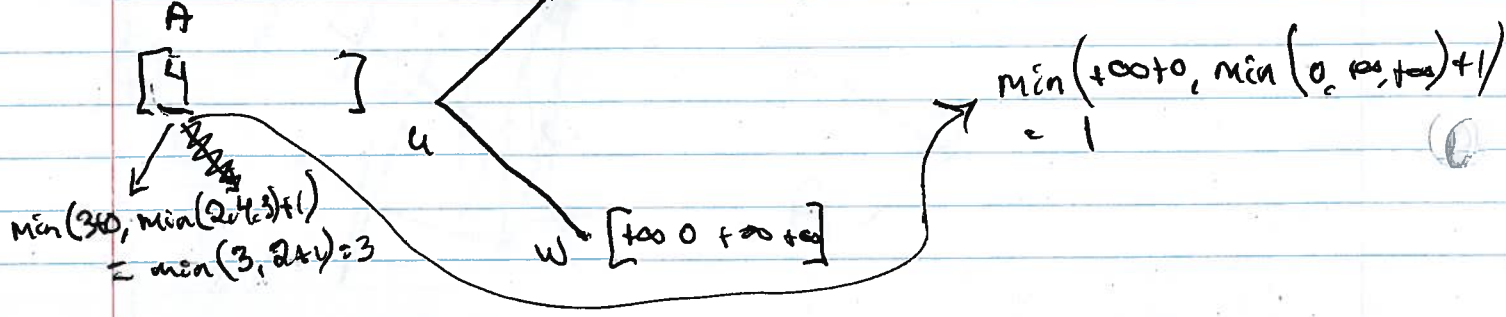
if u is a leaf $F_u[a] = \begin{cases} 0 \text{ if } s_u = a \\ +\infty \text{ if } s_u \neq a \end{cases}$

if v is an internal node,



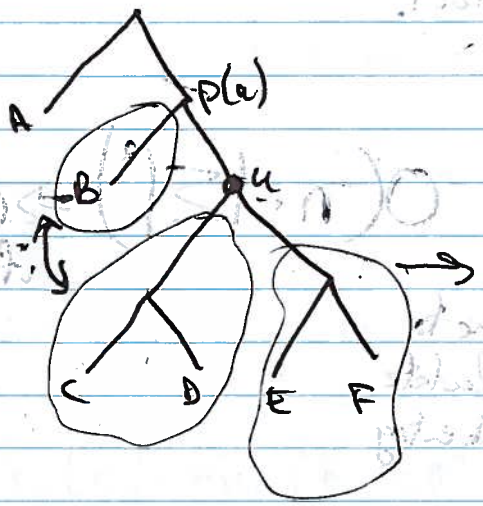
$$F_u[a] = \min\left(F_v[a], \min_{b \neq a}\{F_v(b)\} + 1\right) + \min\left(F_w[a], \min_{b \neq a}\{F_w(b)\} + 1\right)$$





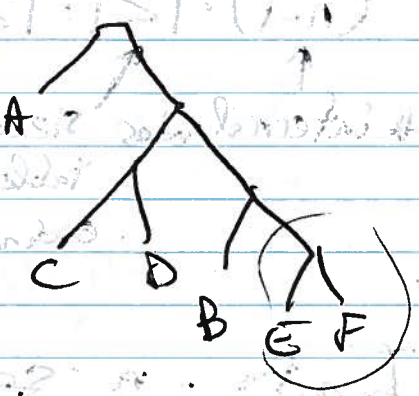
Tree neighborhood definitions

T:



Nearest-neighbor interchange (NNI)

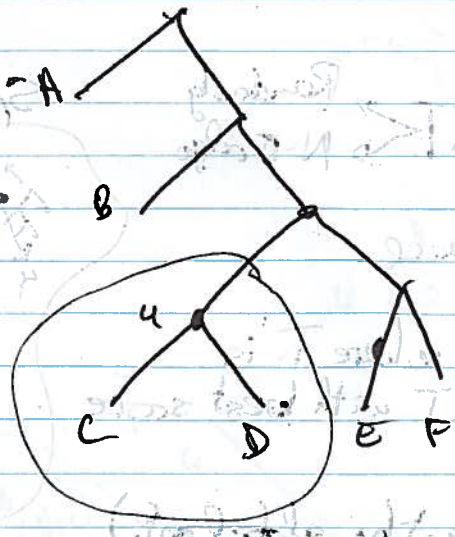
Pick some internal node u



$$Neighbors(T) = \{ T' \text{ s.t. } T' \text{ can be obtained from } T \text{ with } \underline{\text{one}} \text{ NNI} \}$$

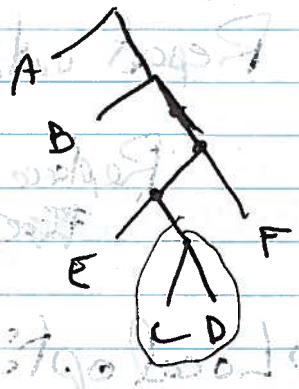
$$|Neighbors(T)| = (n-2) \cdot 2$$

Subtree Prune and Regraft



Choose any node in T
- cut in off from T
- reinsert along other edge of T

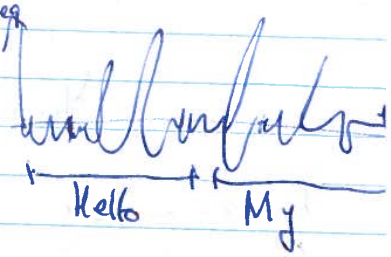
⇒



Hidden Markov Models (HMM)

Goal: "Decode" a sequence of observations

Speech recognition : Observation:



Biological sequence annotation : Observation: ACTACTAGGTAC

Genes

Protein: VLIYVRRCY

α-helix β-sh

Toy example: You visit a city with different neighborhoods.

set of states = $S = \{F, E, C\} = S = \{S_1, S_2, \ldots S_n\}$

F: French, E: English, C: Chinese

You walk randomly in city.

Every minute, someone greets you.

set of symbols = $\Sigma = \{b, h, n, a\} = \{\sigma_1, \sigma_2, \ldots \sigma_k\}$

b: bonjour, h: hello, n: chinese, a: namaste

You record seq. of greetings: $X = x_1, x_2, \ldots x_L$

where $x_i \in \Sigma$ $\quad X = b\,b\,a\,a\,h\,h\,h\,a\,h\,b \ldots$

Problem: Given: $X = x_1 \ldots x_n$

Find: Path $P = p_1 \ldots p_n$ that is most likely given X

We need to know:

① Emission Probabilities: $Pr_e[x_i = \alpha \mid p_i = \beta]$ ($\alpha \in \Sigma$, $\beta \in S$)

States \ Symbols	b	h	n	a
F	0.9	0.05	0.05	0
E	0.1	0.5	0.1	0.3
C	0.3	0.3	0.3	0.1

→ sum to 1

② Transition probabilities

$Pr_\varepsilon[P_{i+1} = \gamma \mid P_i = \beta]$

ε S

Destination

Source	E	C	F
E	0.9	0	0.1
C	0.1	0.4	0.5
F	0.3	0.1	0.6

Path: E EEEEEEEE FFFFEE FFFCCC

③ Initial State probability

$Pr[P_1 = \gamma]$

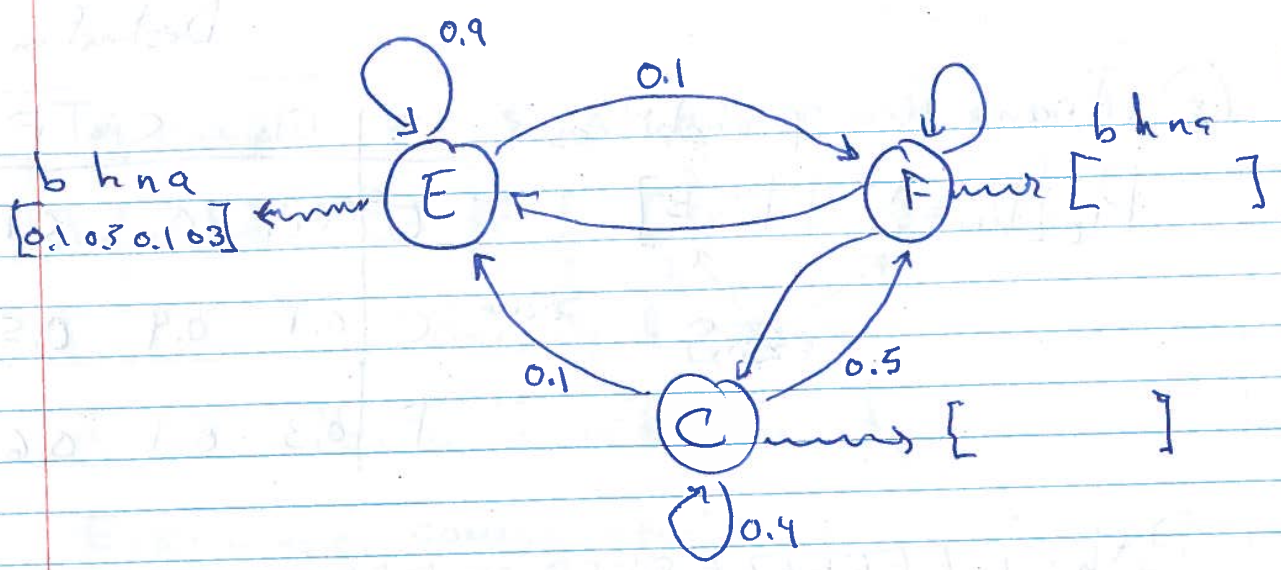
ε S

E	C	F
0.5	0	0.5

Assumptions: - Markovian assumption (1st order)

Prob of going to state γ only depends on where you are now, not on how you got there

- Observations are independent from each other (given the states)



HMM as generative model: Generate a random of sequence

① Pick randomly P_1 from the initial state prob. dist

Repeat: 2.1 Emit an observation from current state

2.2 Transition to new state acc. to transition prob. matrix

Questions (Assume we know S, Σ, emission prob, transition prob, initial prob)

① Maximum likelihood path:

Given: $X = x_1 \dots x_L$

Find: Path $P = P_1 \dots P_L$ that is most likely to have generated X, ie. $Pr[P_1 \dots P_L \mid x_1 \dots x_L]$ is maximized

Answer: Viterbi algo.

② Posterior decoding

Given: $X = x_1 \dots x_L$, time i, state β

Calculate: $Pr[P_i = \beta \mid X]$

Answer: Forward-Backward algo

③ Estimation Problem: Assume we know S, Σ but not emission prob, not transition prob, not initial

Given: $X = x_1 \ldots x_k$

Find: Emission prob, Transit prob, Initial prob } such that $Pr[X \mid E, T, I]$ is max

Answer: Baum-Welch algorithm

Viterbi Algorithm:

Given: $X = x_1 \ldots x_k$ (seq. of k observations)

(S, Σ, E, T, I) = HMM

- S: states
- Σ: alphabet
- E: emission prob matrix
- T: transition prob matrix
- I: initial state prob. vector

Find: $P = P_1 \ldots P_k$, where $P_i \in S$

such that $Pr[P_1 \ldots P_k \mid X = x_1 \ldots x_n]$ is maximized

Question 1: How to calculate $Pr[P_1 \dots P_L \mid X = x_1 \dots x_L]$?
for a given path $P = P_1 \dots P_L$

$$Pr[P_1 \dots P_L \mid X = x_1 \dots x_L] = \frac{Pr[P_1 \dots P_L \wedge X = x_1 \dots x_L]}{Pr[X = x_1 \dots x_L]}$$

Independent of Path P

$$Pr[P_1 \dots P_L \wedge X = x_1 \dots x_L] = Pr[X = x_1 \dots x_L \mid P_1 \dots P_L] \cdot Pr[P_1 \dots P_L]$$

$$= \left(\prod_{i=1}^{L} Pr_E[x_i \mid P_i] \right) \cdot Pr_I(P_1) \cdot \prod_{i=1}^{L-1} Pr_T[P_{i+1} \mid P_i]$$

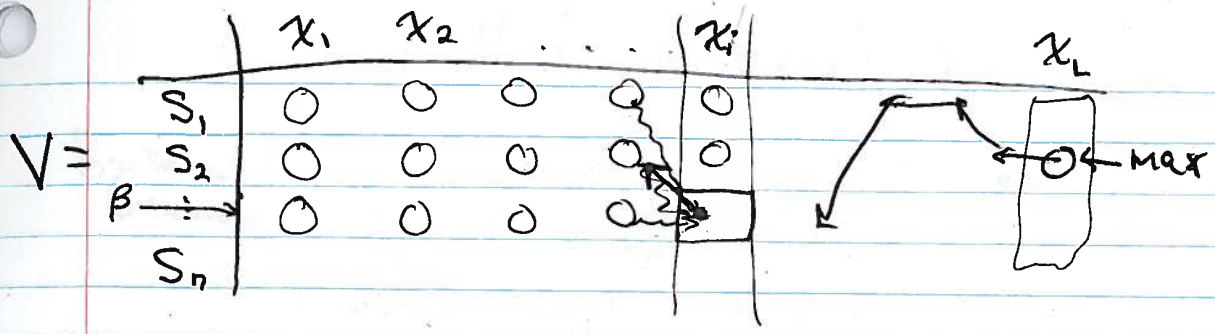
Question 2: How to find P s.t. $Pr[P_1 \dots P_L \mid x_1 \dots x_L]$ is max?

$\Rightarrow$ Viterbi algorithm

Define $V(\beta, i)$ = Prob. of the most likely path of length i, given observation $x_1 \dots x_i$, assuming path ends in state β

$\beta \in S$, $i \in \{1 \dots L\}$

$$= \max_{P_1 \dots P_i \text{ where } P_i = \beta} \{ Pr[P_1 \dots P_i, x_1 \dots x_i] \}$$



$$V(\beta, i) = \max_{\gamma \in S} \{ V(\gamma, i-1) \cdot Pr_t[\beta | \gamma] \} \cdot Pr_e[x_i | \beta]$$

Fill V table column by column, left-to-right

$V(\beta, 1) = Pr_I(\beta) \cdot Pr_e[x_1 | \beta]$ (Initialization)

$$\max_{P_1 \ldots P_L} \{ Pr[P_1 \ldots P_L, x_1 \ldots x_L] \} = \max_{\beta \in S} \{ V(\beta, L) \}$$

To recover path P that maximized $Pr[P_1 \ldots P_L, x_1 \ldots x_L]$

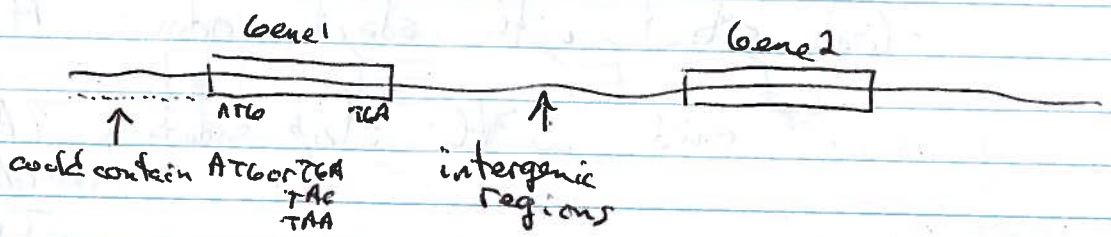
- Trace back dyn. prog algo from $\max \{ V(\beta, L) \}$

Complexity Time = $O(n^2 \cdot L)$

Space = $O(n \cdot L)$

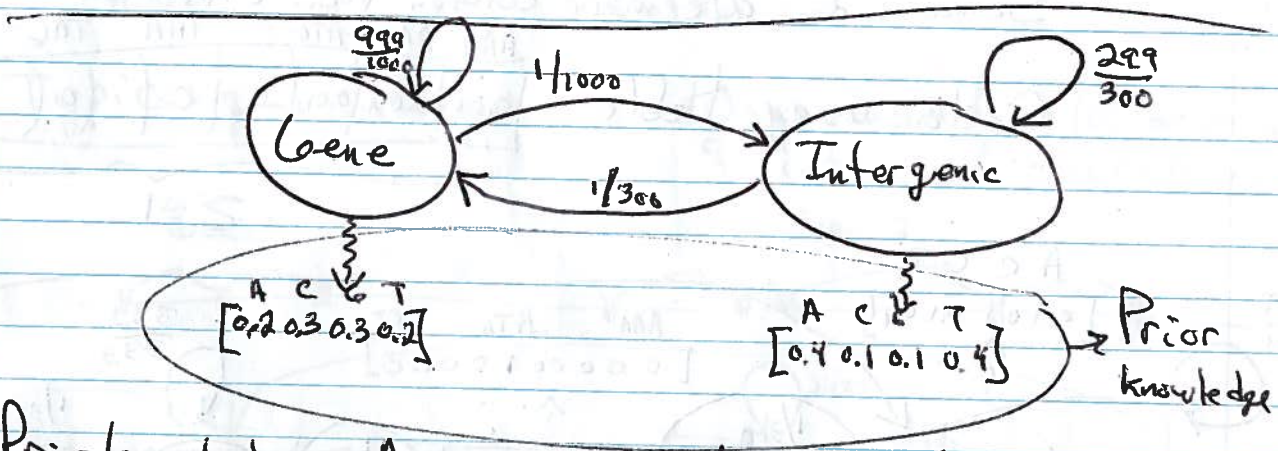
HMMs for Gene Finding

Bacterial genome:



Problem: Given: Genome sequence X

Find: Start/End position of each gene in X



Prior knowledge: Avg. gene length ~ 1000bp

Avg. intergenic length ~ 300bp

Genome X = ATAGATAACAGGCTCGTGGTC | ATAT

Viterbi Path = IIIIIIII | 66666666666666 | IIII

Integ | Gene | Integ

Gene Properties:

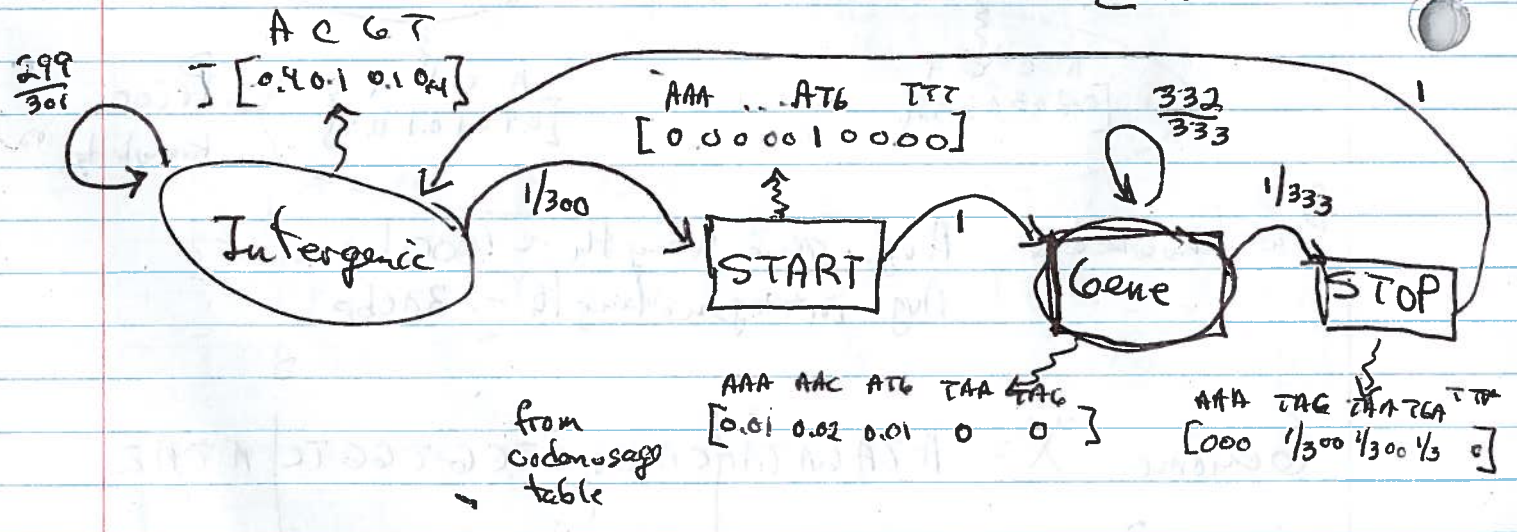
- Gene start with start codon: ATG
- . end with stop codons: TAG
  TAA
  TGA

- Gene is made of codons (triplets of nuc.)
- Some codons are more common than others

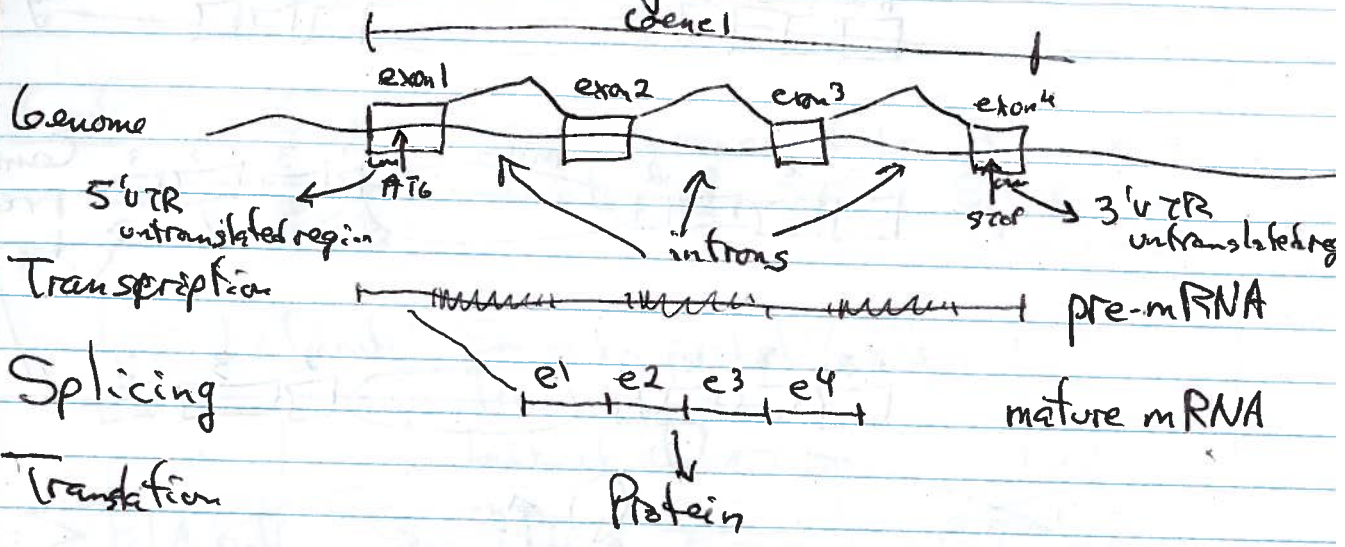
Codon usage table

AAA	AAC	ATG	...	TAA	TAG	TGA
0.01	0.02	0.01		0	0	0

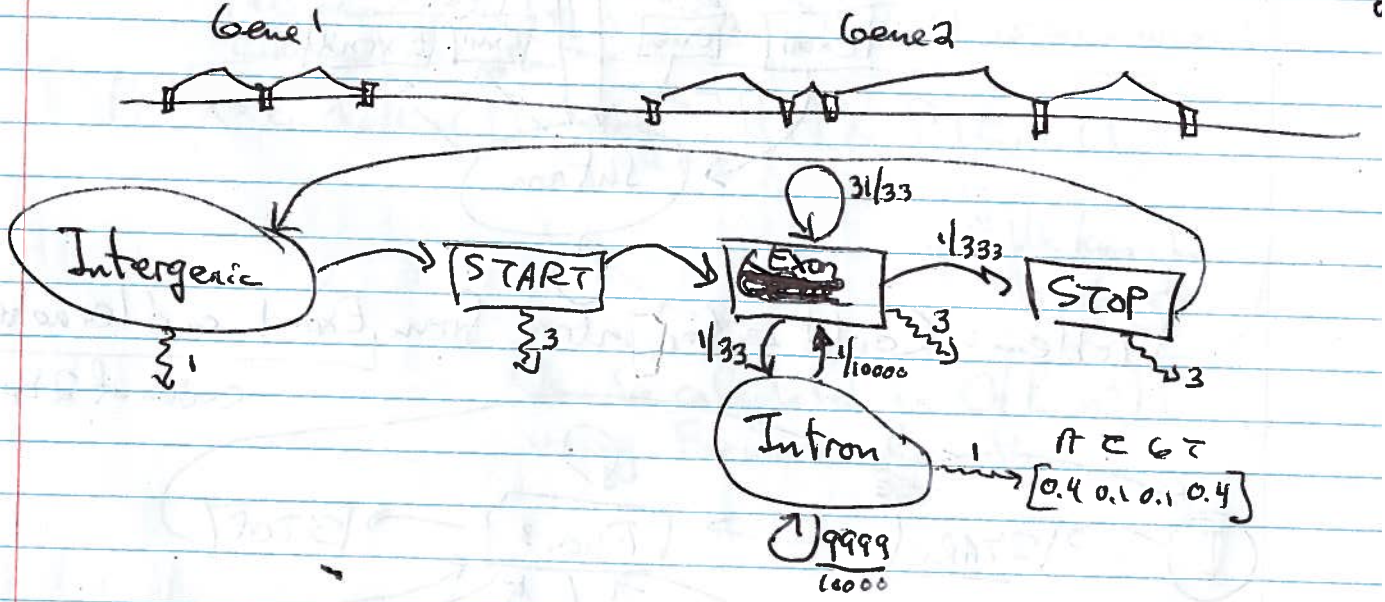
$\Sigma = 1$

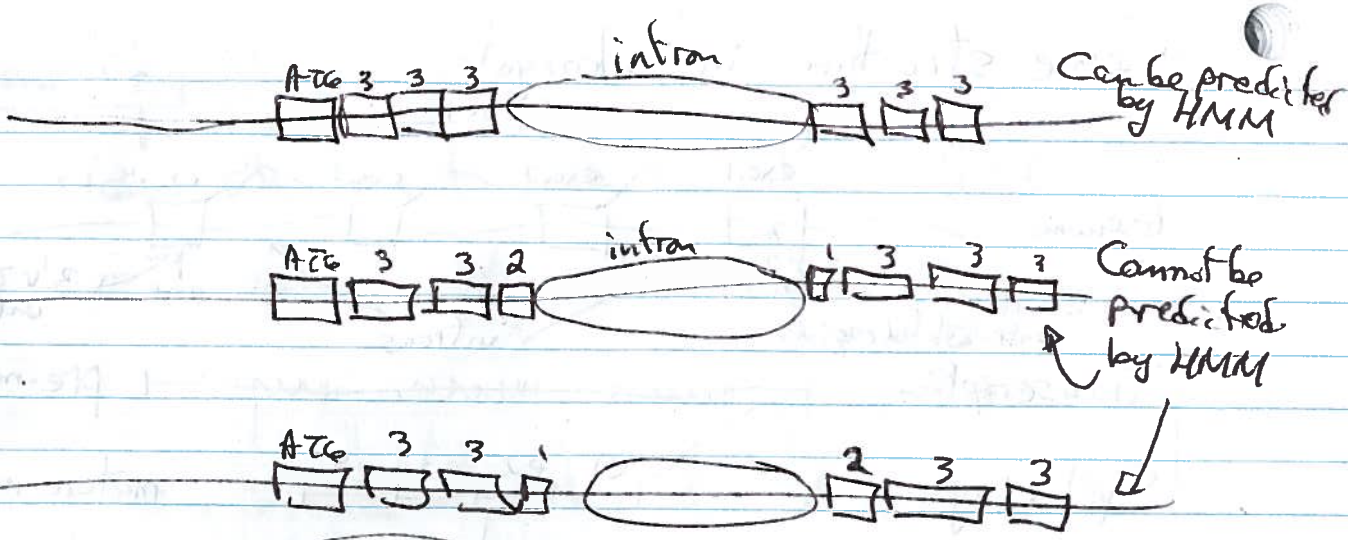


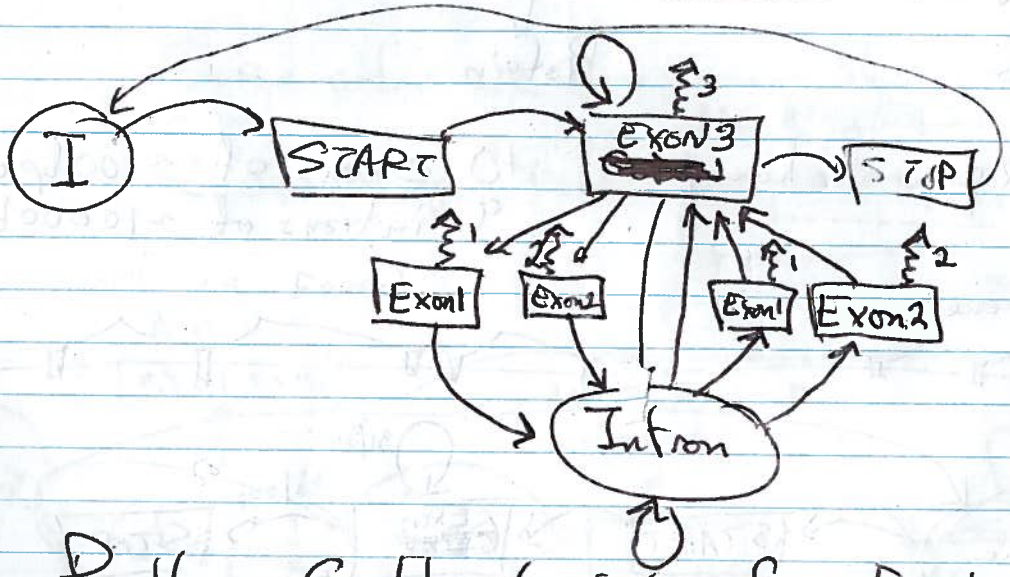
Gene structure in eukaryote



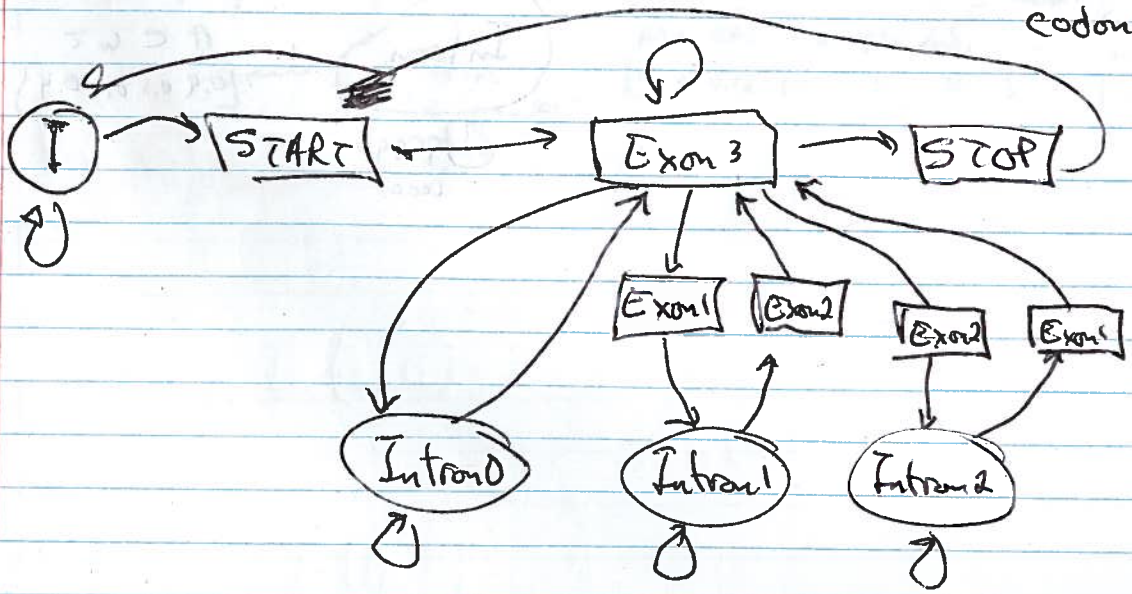
Typical gene in human : 10 exons of ~100 bp on aver
9 introns of ~10000 bp on av







Problem: Could enter intron from Exon1 and leave from Exon1 codon of 2 bp



Training HMMs

Goal: Given: • Long sequence of observations $X = x_1 \dots x_L$
• Structure of an HMM: Set of states S
Set of possible trans. T

Find: • Emission probabilities : E
• Transition prob. : T
• Initial state prob : I

such that E, T, I best represent the observed seq. X

Maximum Likelihood estimation ← $\Pr[X = x_1 \dots x_L \mid E, T, I]$ is maximized

$$\Pr[A] = \sum_b \Pr[A, B = b]$$

$$\Pr[X = x_1 \dots x_L \mid E, T, I] = \sum_{\text{path } P = p_1 \dots p_L} \Pr[X, P \mid E, T, I]$$

We know how to calculate

Can be calculated in $O(L \cdot n^2)$ using Forward algorithm

Simplified version:

Given: $X = x_1 \dots x_L$
$P = P_1 \dots P_L$ ← annotation of X

Find: E, T, I s.t. $Pr[X, P \mid E, T, I]$ is max

Let $N_t(s, s')$ = # of times transition happened btw s and s'
= # of positions i s.t. $P_i = s \wedge P_{i+1} = s'$

Then, choose $T(s, s') = \dfrac{N_t(s, s')}{\sum_{x \in S} N_t(s, x)}$ → total # of times in state s

Example: X: A C A ' I A C I A T G A C T G
P: G G G I I I I I I G G G G I I

$T(G, I) = \frac{2}{7}$ $\quad T(G, G) = \frac{5}{7}$

For emissions: $N_e(s, a)$ = # times a was emitted from s
= # positions i s.t. $P_i = s \wedge X_i = a$

Choose $E(s, a) = \dfrac{N_e(s, a)}{\sum_b N_e(s, b)}$

Example: $E(G, A) = \frac{4}{7}$

Back to problem where P is not given

① Viterbi training

Ⓐ Choose "reasonable" E, T, I (either from prior knowledge or randomly)

Repeat until convergence

Ⓑ Use Viterbi algo to find best path for $X = x_1 \ldots x_k$ assuming E, T, I ⇒ Produces path P

Ⓒ Re-estimate E, T, I from P, X as done previously

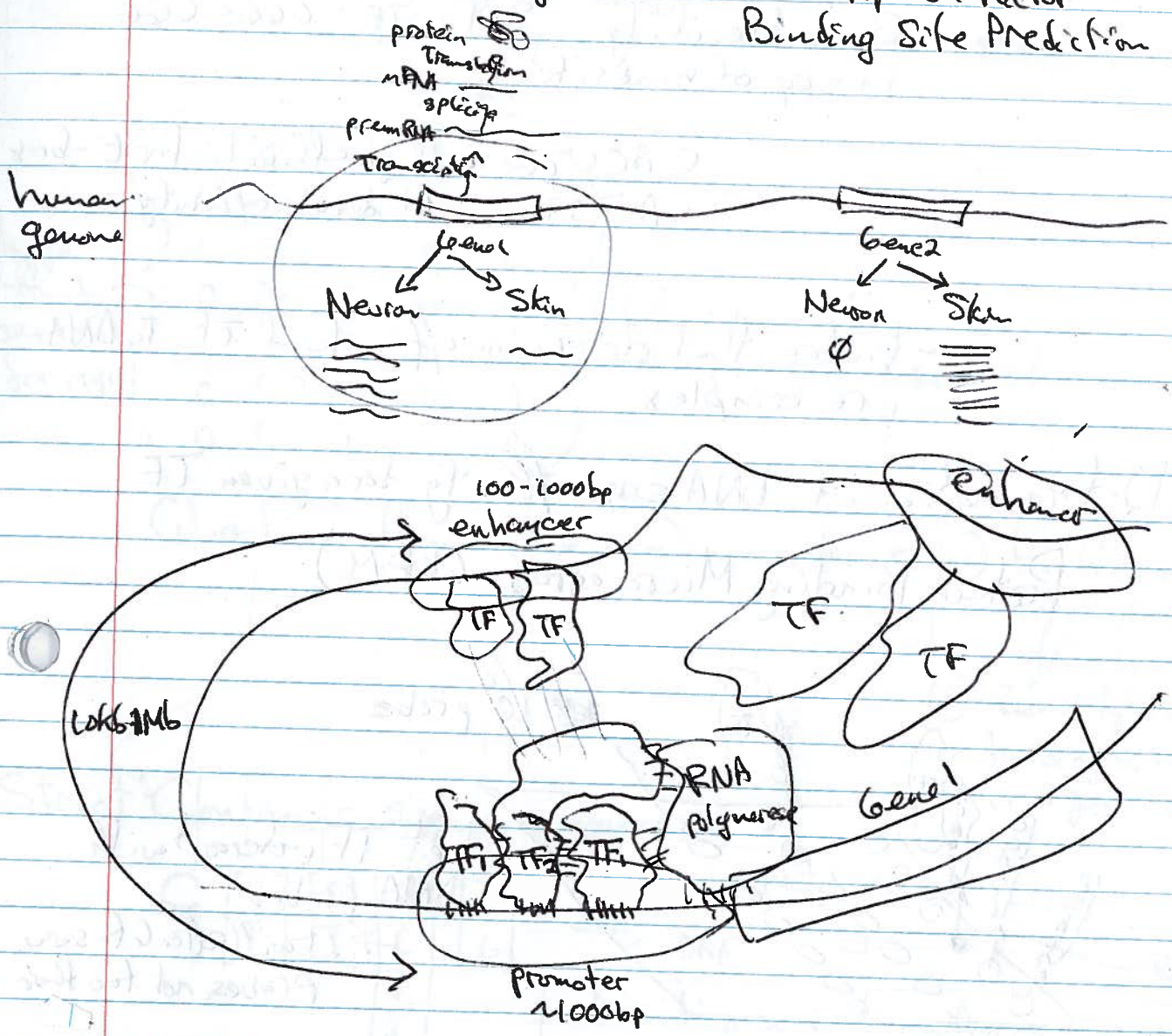
Problem: Algo. frequently gets stuck in local optima

② Baum-Welsch algorithm

Bases re-estimation of E, T, I on all paths (weighted by their probability)

⇒ Reduces the prob of getting stuck in local optima

Gene Regulation + Transcription Factor Binding Site Prediction



Transcription factors: - Proteins that - bind specific DNA seq.
- Alter expression of nearby gene(s)
- activation
- repression

- In humans: 2000 different TFs
↳ each binds to different DNA sequences
Transcription Factor Binding sites

TFBS: - 6-15 bp long
- Some flexibility in seq. of bind site

E-box TF bind: CACGTG
SP1 TF: GGGCGGG

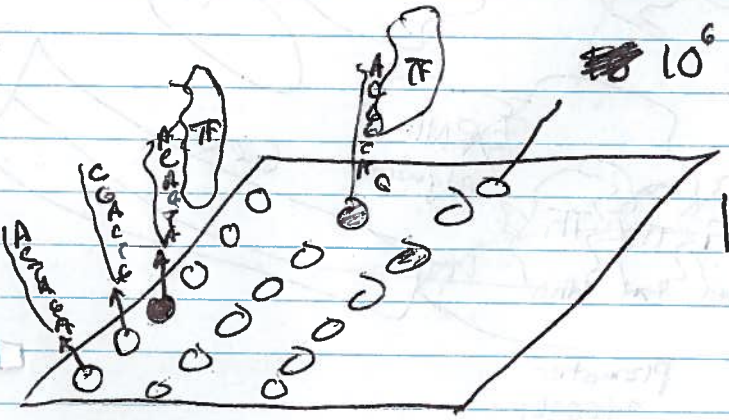
CACGTG: High affinity for E-box
CATGTG: Moderate affinity

⋮

- Rules that determine affinity of TF to DNA are complex

Determination of DNA seq. affinity for a given TF

Protein Binding Micro-array (PBM)



10^6 probes

Let TF interact with DNA probes.
→ It will attach to some probes, not to others

Take picture of PBM ⇒ infer affinity of TF to each probe

⇒

	intensity
ACAGTA	1000
ACGGTA	250
⋮	

Example: [redacted]

DNA sequences bound by TF [redacted] myc

not independent

C	C	T	A	A
C	T	A	C	G
C	T	T	A	G
C	T	T	G	G
C	C	T	T	A
C	C	T	C	A
C	C	T	T	A

high affinity sequences for myc

Representative sample of all possible sites TF

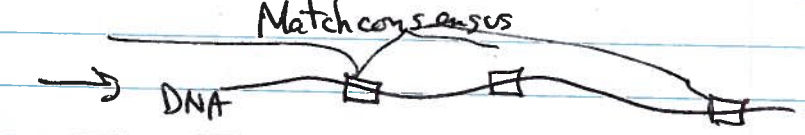
Question: How can we use in order to
① Build model for that TF's affinity
② Identify new binding sites in a genome

Strict Consensus sequence approach

$$C\begin{bmatrix}C\\T\end{bmatrix}\begin{bmatrix}A\\T\end{bmatrix}\begin{bmatrix}A\\C\\G\\T\end{bmatrix}\begin{bmatrix}A\\G\end{bmatrix}$$

or

$$C\begin{bmatrix}C\\T\end{bmatrix}[T]\begin{bmatrix}A\\C\\G\\T\end{bmatrix}\begin{bmatrix}A\\G\end{bmatrix}$$



Position weight matrices

	1	2	3	4	5
A	0	0	4/7	2/7	4/7
C	1	4/7	0	2/7	0
G	0	0	0	4/7	3/7
T	0	3/7	6/7	2/7	0

Candidate sequence to score: C T A G G

$1 \times 3/7 \times 4/7 \times 1/7 \times 3/7 = 9/7^4 = 0.00\ldots$

How to identify matches for a given PWM in a given sequence

S= A C A G T G A C T T

For each starting position i in S

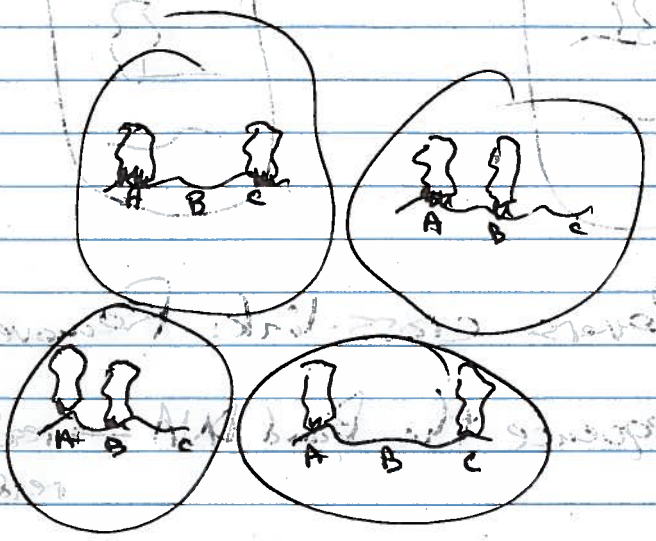
Calculate score of S[i, i+1, ... i+5-1] on PWM

If score > Threshold, then predict Binding
other no binding

ChIP-Seq + Motif discovery

Goal of ChIP-Seq: Identify regions of genome bound by a given TF in a given type of cells.

ChIP-Seq: Chromatin Immunoprecipitation followed by Seq.



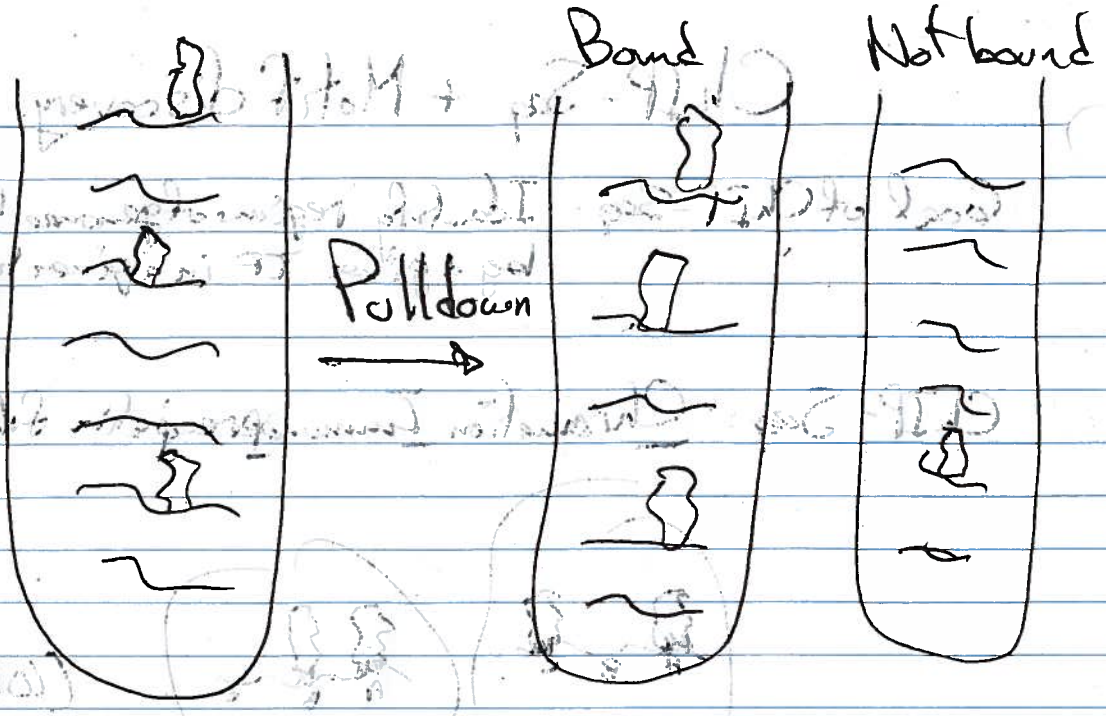
(10 Million cells)

(Typically: 100 - 100,000 sites occupied by TF)

① Cross-linking proteins to DNA: Strengthen bonds btw prot. and DNA

② Extract DNA with proteins attached to it

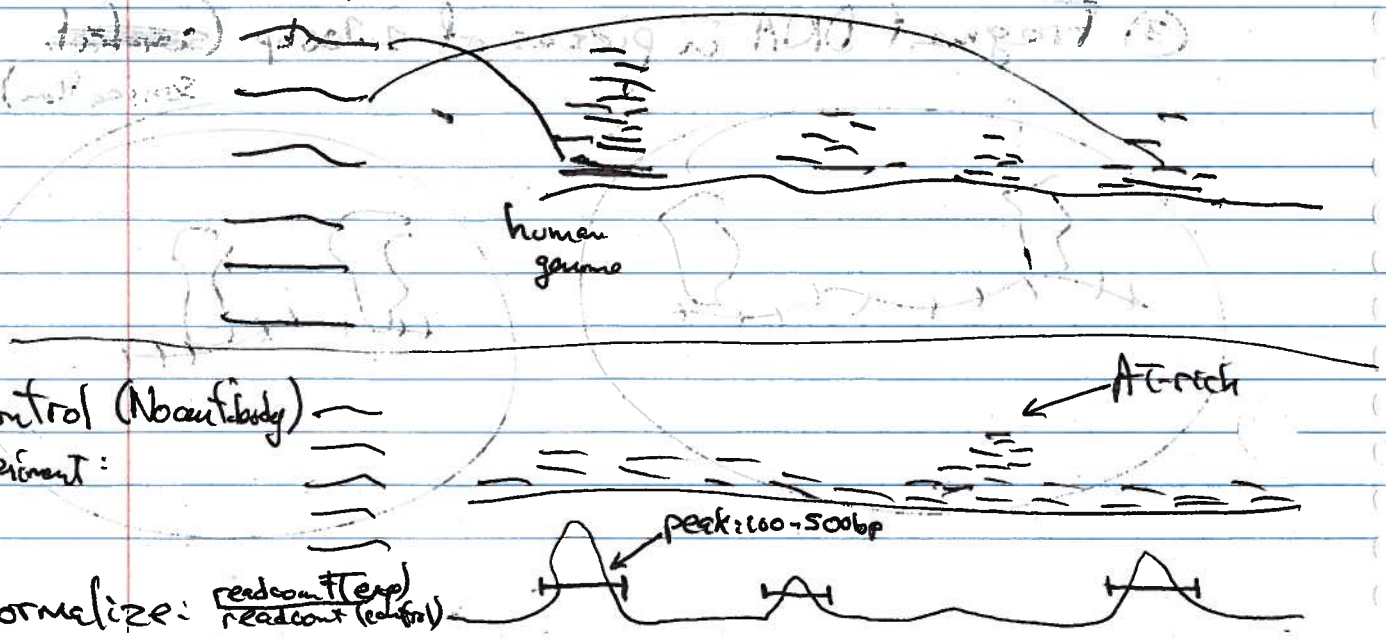
③ Fragment DNA in pieces of ~200bp (~~sonicate~~ sonication)



⑤: Reverse cross-link: Remove the TF from DNA

⑥: Sequence the Bound DNA → read1: ATTAGGTC (200bp)
read2: TGTGCTA...
⋮
read 10^7: TAATGAT...

⑦ ~~Map~~ Read Mapping
Local alignment btw reads and the human genome



100 - 100,000

ChIP-seq outcome: List of genomic regions (100-500bp that are believed to be bound by TF

Problem: Doesn't tell us which 6-15bp region is bound

⇒ Can't build consensus or PWM for TF

Motif Discovery Problem

Given: Set of sequences $S_1, S_2, \ldots S_n$ (of length 100-500) each believed to contain binding site for TF

Find: Consensus sequence for TF

OR

PWM for TF

Consensus sequence: Regular expression

$w = \{A\}\{C\}\{A\}\{T\}\{A\}$
$\quad\quad\{T\}\quad\{C\}$
$\quad\quad\{G\}$

How should we score a candidate cons. seq $w = w_1 \ldots w_k$

Criteria:

- w should occur in each of $S_1 \ldots S_n$
  - → too strict: Some S_i might be false-positives
  - → $\{A\}\{A\}\ldots\{A\}$ matches everywhere

Motif enrichment approach → consensus sequence

Let N_w = # of matches for w in $S_1, S_2, \ldots S_n$

E_w = Expected # of matches of w in a set of random sequences $R_1, R_2, \ldots R_n$, where R_i has same length S_i

→ each nucleotide is chosen independently with $P_A: 0.3$, $P_C: 0.2$, $P_G: 0.2$, $P_T: 0.3$

How to compute E_w? $w = w_1 w_2 \ldots w_k$ where $w_i \in \{A, C, G, T\}$

Ex: $w = \{A\}\{C\}\{A, T\}\{G\}\{T\}$

$\Pr[w$ has a match starting at position p in random seq$] = ?$

R: . . . . . p ... p+k-1

$$= \prod_{i=1}^{k} \Pr[w_i \text{ has match at position } p+i-1]$$

$$P_{match}(w) = \prod_{i=1}^{k} \left(\sum_{a \in w_i} P_a \right)$$

$$E_w = (\#\text{positions eligible for match}) \cdot P_{match}(w)$$

$$= \sum_{i=1}^{n} (\text{length of } S_i - k + 1) \cdot P_{match}(w)$$

Finally, from N_w, E_w

We want to find w where N_w and E_w are the most different, i.e. $N_w >> E_w$

Z-score approach

$$Z_w = \frac{N_w - E_w}{\sqrt{E_w}}$$

Complete algo.

For each possible consensus seq. w

Calculate N_w, E_w, Z_w

Report word w with highest z_w

DNA sequencing + Genome sequencing

Goal :

→ File : >seq1
ACTAGCTA] read1
>seq
TGACTAG..] read2
⋮

Illumina Sequencing : See video.

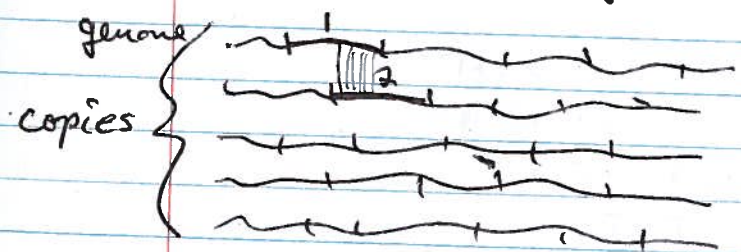
Limitation : - Length of a read is limited ($\lesssim$ 300bp) (Illumina)

Genome sequencing + Assembly

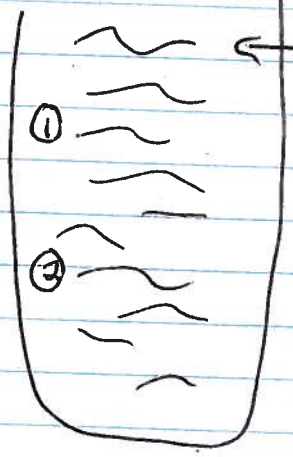
Goal: Get entire DNA seq. of a long genome

Problem: Seq. machines produce reads that short

Shotgun sequencing



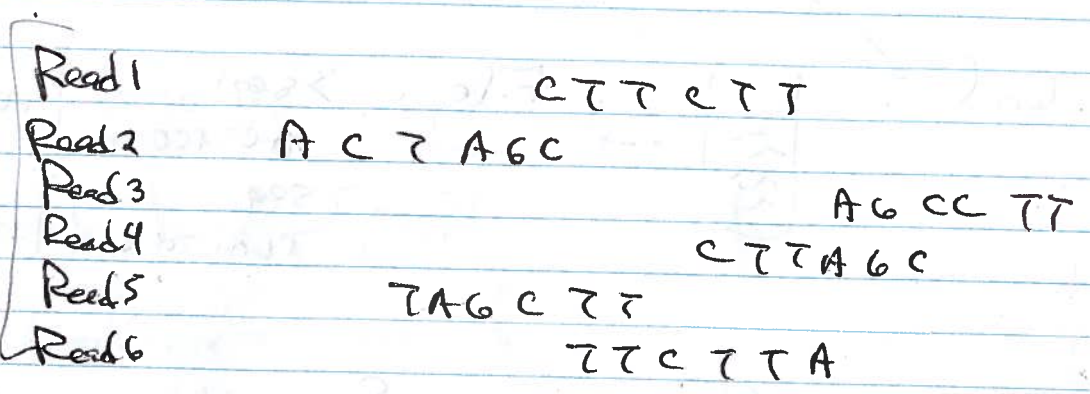
① Generate many copies of genome
② Cut seq. into small pieces
→ sonication, ~~restriction enzyme~~
randomly



③ Sequence DNA fragments
ACTAGC
TAGCC...
⋮

④ Assemble reads into genome

Truegenome : A C T A G C T T C T T A G C C T T
(unknown)



Problem: Shortest Superstring Problem

Given: Set of reads $R_1 \ldots R_n$, of length L = six

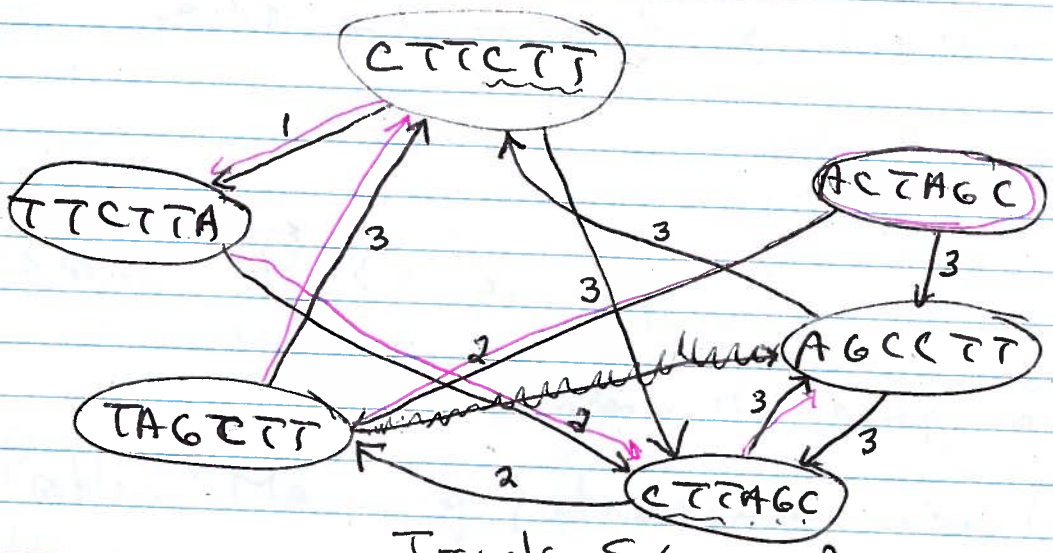
Find: Sequence G such that

1) Each R_i is a substring of G
2) G is a short as possible

Assumption: No sequencing error.

Overlap-Layout-Consensus Approach

Build Graph: V: set of reads
E: overlaps btw reads of at least $k = 3$ bases



Traveling Salesperson Problem → NP-Complete Problems

Goal: Find shortest Hamiltonian Path in G

Smallest total weight

Path that visits each vertex exactly once

ACTAGC
TAGCTT
CTTCTT
TTCTTA
CTTAGC
AGCCTT

weight: 2+3+1+2+3
= 11

Predicted Genome ACTAGCTTCTTAGCCTT

Gene Expression + Class comparison

Final Exam: Dec 14th (6pm) : open book, covers all topics

Goal: Capture and compare "state" of different cells

- cells with positive outcome upon treatment
- negative outcome

→ state $\vec{P} = (p_1, p_2 \ldots, p_{20000})$

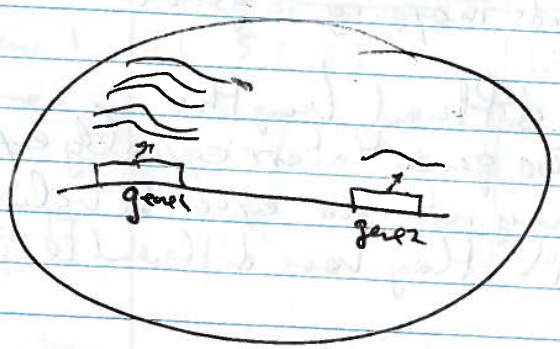
where p_i = abundance of protein p_i in cells.

Problem: Measuring protein abundance is hard (mass spectrometry)

Alternative: Measure mRNA abundance

$\vec{G} = (g_1, g_2 \ldots, g_{20000})$

where g_i = abundance of mRNA from gene i in cells



→ $g_1 = 5$
$g_2 = 1$

Note: $g_i \neq p_i$ because: mRNA degration
translation is regulated

RNA-sequencing (RNA-seq)

Goal: Measure gene expression levels $\vec{G} = (g_1 \cdots g_{20000})$

① Extract RNA from cells
② Reverse transcribed RNA to cDNA (↑ complementary)
③ Fragment cDNA in ~200bp pieces
④ Sequence each cDNA fragment

Read1 : ACTAG...
Read2 : ...
⋮
Read 10^7 : TACTA

Human Genes	R_i
gene1	1
gene2	3
gene3	1
⋮	
gene 20000	0

⑤ Map (↑ align) each read to its gene
Find the gene to which the read best aligns
⑥ Count R_i = #of reads mapping to gene i
Problem: Genes have different lengths
⇒ Two genes that are equally expressed may not have equal R values if they have different length
⑦ Normalize:

$$FPKM(g_i) = \frac{R_i}{(\text{length of } g_i \text{ (in kb)}) \cdot (\text{Total \# reads in Millions})}$$

FPKM: Fragment per kilobase per million reads

Class Comparison Problem

Given: Normalized gene expression data from two set of samples

A: (control) with $N_A = 20$ samples

$\vec{A_1} = (A_1(1), A_1(2) \ldots A_1(20000))$

$A_1(i)$ ↖ FPKM of gene i in sample 1

$\vec{A_2} = ($ $)$

$\vec{A_{N_A}} = ($ $)$

B = (treatment) with $N_B = 25$ samples

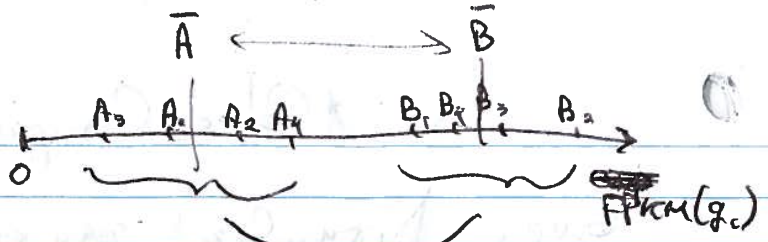
$\vec{B_1} = ($ $)$

$\vec{B_{N_B}} = ($ $)$

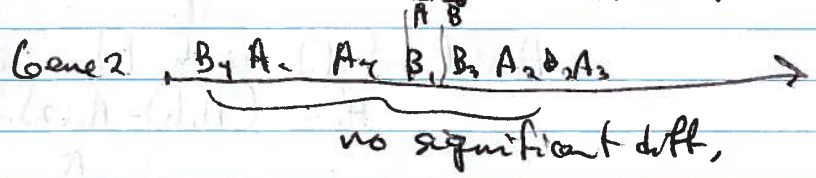
	20			25				t-stat	p-value
	A_1	A_2 ...	A_{N_A}	B_1			B_{N_B}		
Gene 1	5.1	5.7	3.7	1.7	1.3	1.9	2.1	7.1	0.001
Gene 2								1.3	0.25
⋮									⋮
Gene 17								best genes	0.0001 = 1/10^4
Gene 20000									0.35

Goal: Find genes that are "differentially expressed" b/w A, B

Example

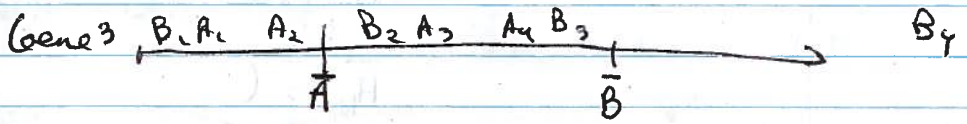


Significant diff



no significant diff.

$d(g_i)$ = diff. of means: $\bar{A}(g_i) - \bar{B}(g_i)$



Student t-test (performed separately for each gene)

H_0: Expression values from samples A and B come from the same normal distribution → Assumption

$\mu_A = \mu_B$

$\sigma_A = \sigma_B$

H_1: $\mu_A \neq \mu_B$

① Calculate $t(g_i) = \dfrac{\bar{A}(g_i) - \bar{B}(g_i)}{\sqrt{\dfrac{S_A^2}{N_A} + \dfrac{S_B^2}{N_B}}}$

S_A^2 = Variance in A

② Calculate [p-value for $t(g_i)$] = Prob that two random samples drawn from same distrib. would have t-statistics $t(g_i)$

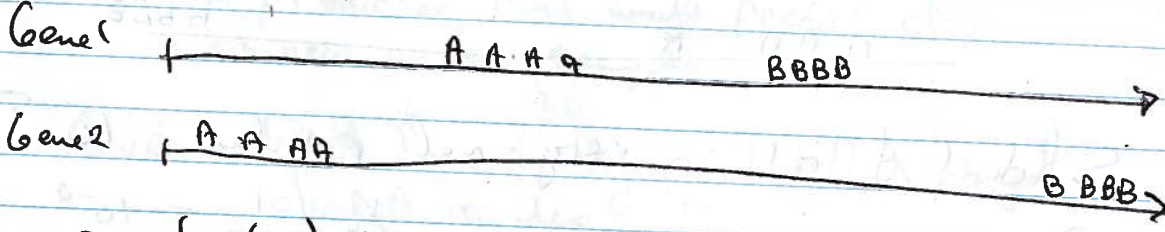
Under H_0: t follow a Student(m) ← degrees of freedom

$$m = \frac{\left(\frac{S_A^2}{N_A} + \frac{S_B^2}{N_B}\right)^2}{\frac{\left(\frac{S_A^2}{N_A}\right)^2}{N_A - 1} + \frac{\left(\frac{S_B^2}{N_B}\right)^2}{N_B - 1}}$$

If p-value(g_i) ≤ 0.05, then all g_i "diff. expressed"

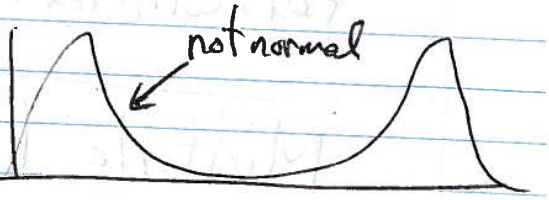
> 0.05, then g_i is not diff. exp.

Issue: Violation of assumptions: - Normality of data



p-value(g_2) < p-value(g_1)

Suppose distribution under H_0



Permutation test: Estimate p-value without assumption about underlying distrib. of data

Real $A_1\ A_2\ A_4\ A_5$

Shuffled $B_4\ A_1\ A_2\ B_3$

For each gene i

① Calculate $t(g_i)$

② Repeat $K=1000$ times

2.1 Randomly reshuffle class labels to obtain $\tilde{A}, \tilde{B}$

2.2 Calculate $\tilde{t}(g_i)$ from

2.3 If $\tilde{t}(g_i) \geq t(g_i)$ the

③ Report p-value$(g_i) = \frac{\text{success}}{K}$

A A A A B B B B

Student t-test: very small p-value: 10^{-}

Permutation: p-value: $\frac{1}{\binom{8}{4}} = 10^{-}$

Multiple hypothesis testing

We've done 20,000 tests

If all genes come from H_0, then best value we would expect to observe would be: $1/20{,}000$

Bonferroni correction: corrected Pvalue(g_i) = p-value(g_i) · N=

Class Prediction Problem

	A_1 A_2 ... A_{N_A}	B_1 ... B_{N_B}	X
Gene 1			
2	RNA-seq data		
3			
Gene 20000			

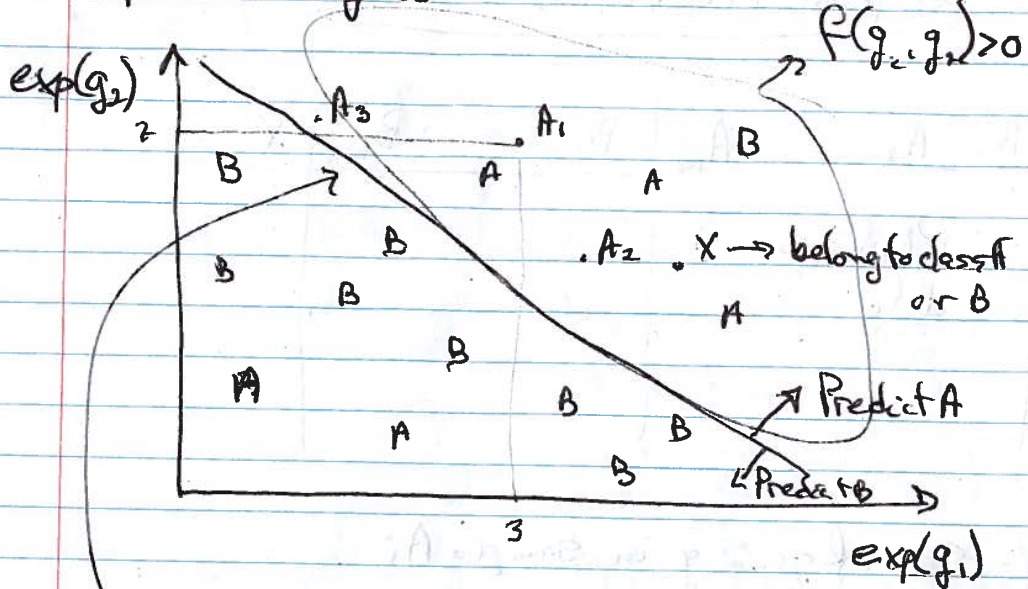
$A_i[g]$ = exp. of gene g in sample A_i

$B_j[g]$ = ...

Goal: From RNA-seq data from A, B, train a predictor that would predict class of new, unseen samples

Given: RNA-seq data X, predict if X belongs to class A or class B.

Assume # genes = 2



Linear classifier: linear function of $exp(g_1)$, $exp(g_2)$

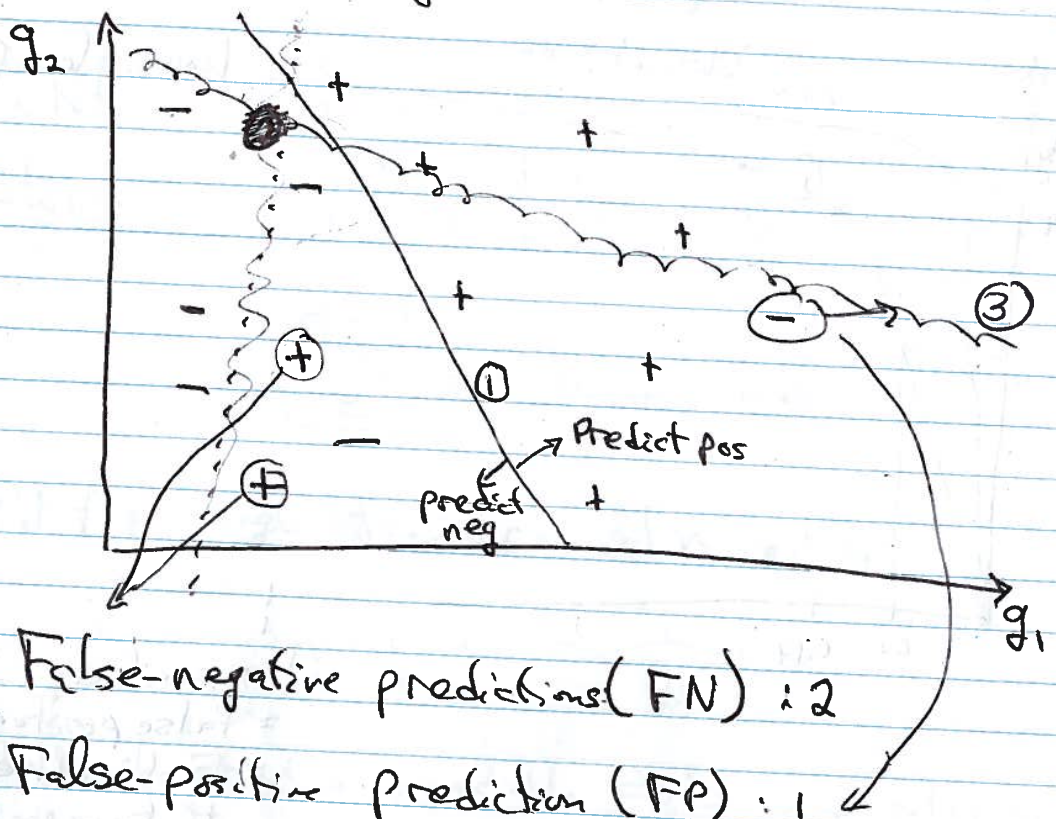
$g_2 = 10 - 2g_1$

$f(g_1, g_2) > 0 \rightarrow$ predict A

$< 0 \rightarrow$ predict B

$f(g_1, g_2) = 2g_1 + g_2 - 10$

Assessing a classifier's accuracy



False-negative predictions (FN) : 2

False-positive prediction (FP) : 1

True-positive (TP) : 7

True-negative (TN) : 5

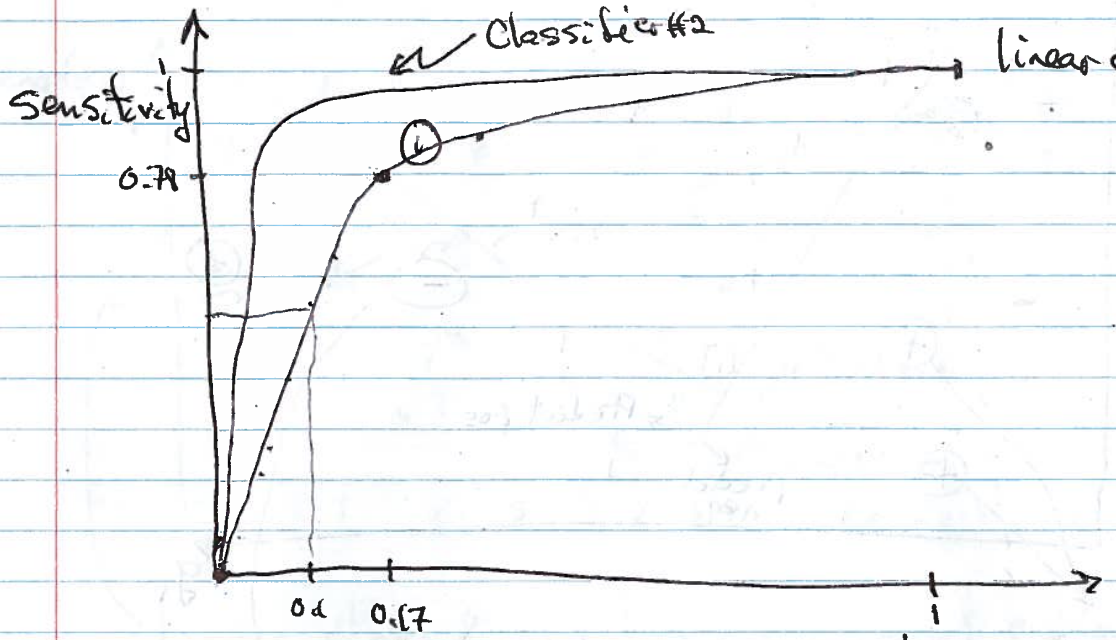
① Sensitivity : $\frac{TP}{TP+FN} = \frac{7}{7+2} = \frac{7}{9} \approx 78\%$

Specificity : $\frac{TN}{TN+FP} = \frac{5}{5+1} = \frac{5}{6} = 83\%$

② Sensitivity : 100%
Specificity : 50%

③ Sensitivity = 4/9 ≈ 55%
Specificity : 100%

Receiving-Operating Curve



1-specificity
= false positive rate
= Fraction neg. example that are predicted pos.

RNA secondary structure prediction

DNA

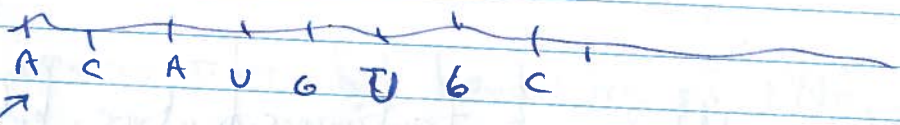
Stable:

A = T

C ≡ G

hydrogen bonds

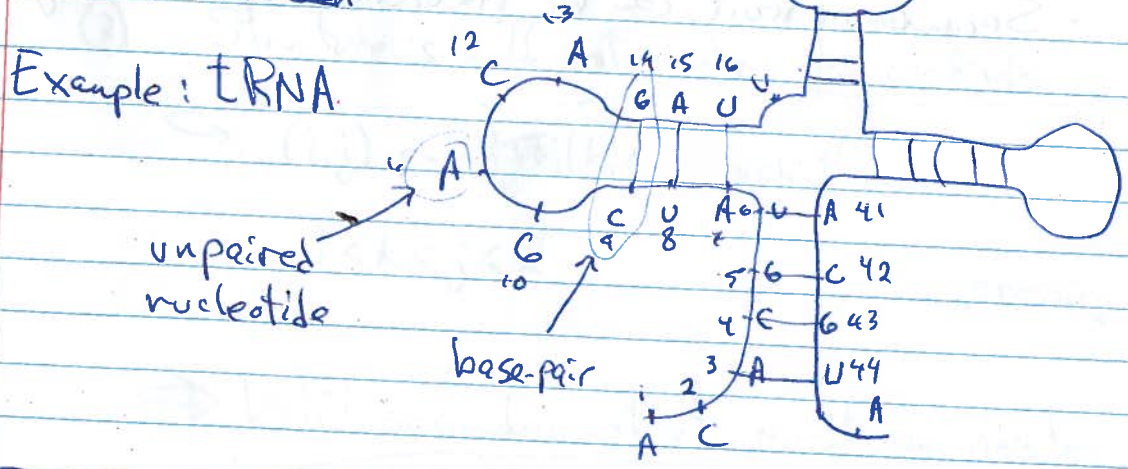
RNA : single-stranded chain of nucleotide



Unfolded structure is unstable ⇒ Sequence folds to maximize its stability

Folded ~~sequence~~:

Example: tRNA



Function of RNA molecules depends on their structure

↓

depends on sequence

Key ideas: A sequence typically folds in its most stable structure

Secondary structure of sequence $S = S_1 S_2 \ldots S_L$ is defined as list of pairs of positions that form basepai

Sec. struct. $\{(3,44), (4,43), (5,42), (6,41), (7,16), (8,15) \ldots\}$

Tertiary structure ≡ 3D structure: (x, y, z) coordinates of each atom in the sequence

Idea: · We want tertiary structure, but it is hard to predict

- Secondary struct. is sufficient to lets us predict function, and also useful for predicting tertiary struct
- Secondary struct. can be predicted computationally

Secondary Structure prediction problem

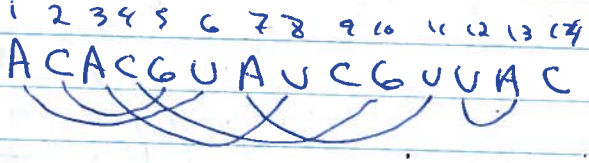
Version 1: Given: Sequence $S_1 \ldots S_L$

Find: Secondary structure for $S_1 \ldots S_L$ that is the most stable

↓

# of base pairings present in the sec. struct.

Example:



$\{(1,6), (2,5), (3,8) \ldots\}$

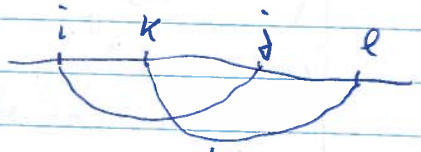
Problem: Ignores rules about bendability of RNA

Rules

① if $(i,j) \in$ Struct, then $|i-j| \geq 3$

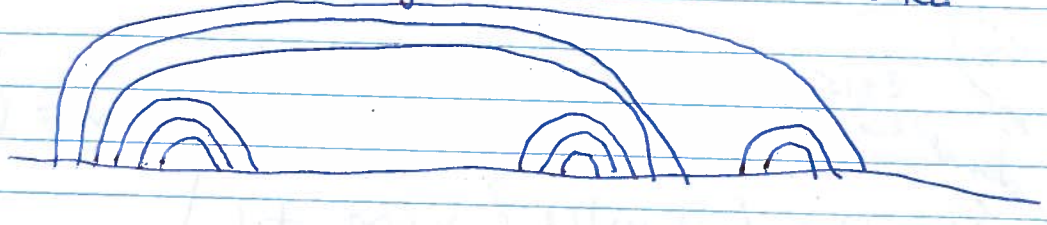
② Structure should not contain pseudoknots

↳ (i,j) and (k,ℓ) such that $i < k < j < \ell$



crossing base pairs

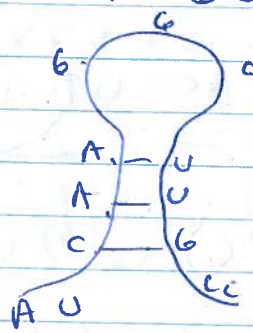
⇒ Valid secondary struct must be nested



Version 2: Given: RNA seq $S_1 \ldots S_L$

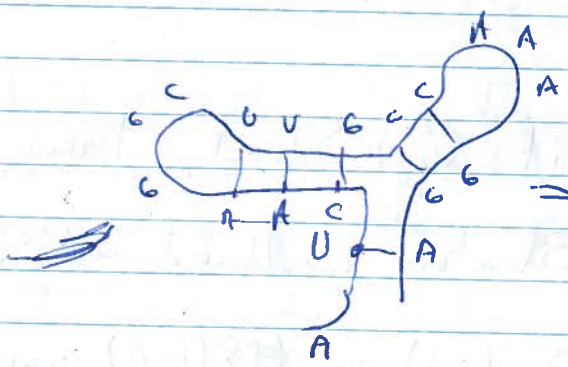
Find: Sec. struct. that maximizes total # of base pairs, subject to rules ① and ②

Example: A U C A A G G C U U G C C A A A G G A



Solution: (short) ⇒ 3 base pairs

longer:

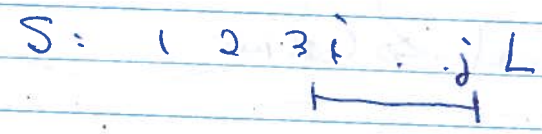


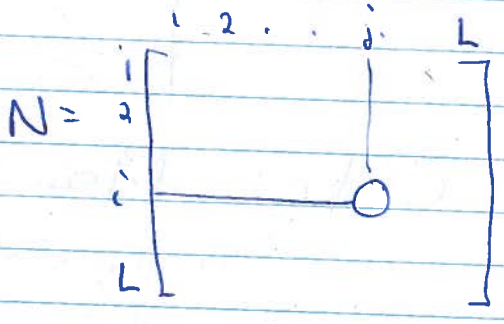
⇒ 6 base pairs

Nussinov Algorithm: Dynamic prog. algo.

Define $N(i,j)$ = Maximum # of base pairs that can be formed for $S_i \ldots S_j$

We want $N(1,L)$

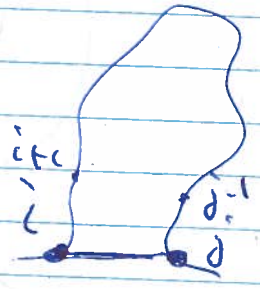




How to calculate $N(i,j)$?

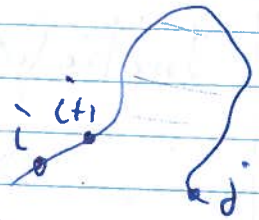
Case 1: S_i is paired with S_j

$$1 + N(i+1, j-1)$$



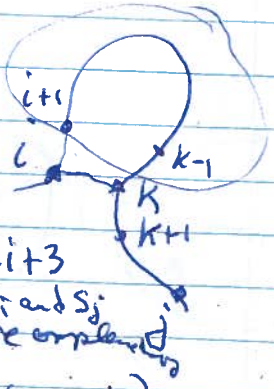
Case 2: S_i is not paired with anything

$$0 + N(i+1, j)$$



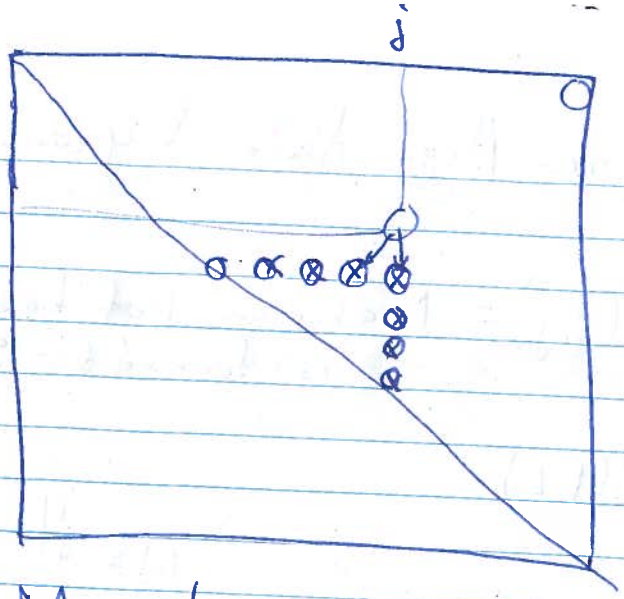
Case 3: S_i is paired with some nucleotide S_k, where $k < j$

$$1 + N(i+1, k-1) + N(k+1, j)$$

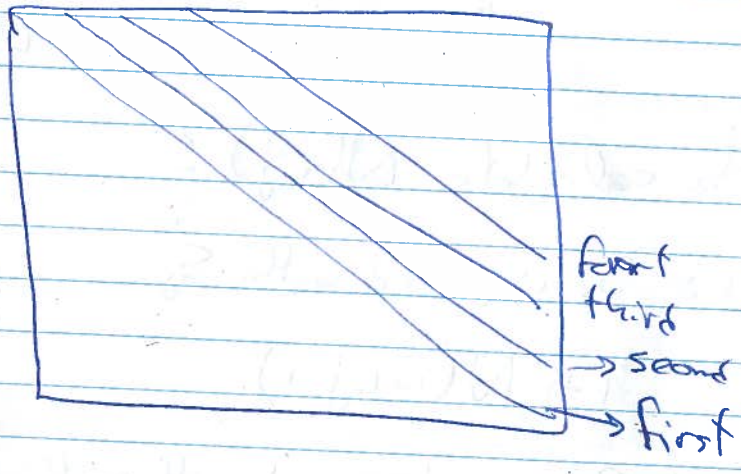


$$N(i,j) = \max \begin{cases} 1 + N(i+1, j-1) & \leftarrow \text{if } j \geq i+3 \text{ and } S_i \text{ and } S_j \text{ are complementary} \\ 0 + N(i+1, j) \\ 1 + \max_{i+3 \leq k < j,\ S_i \text{ is complementary to } S_k} \{ N(i+1, k-1) + N(k+1, j) \} \end{cases}$$

N:



Order: Main diagonal → Corner



Initialization: $N(i,i) = 0 \quad \forall 1 \le i \le L$

$N(i,i+1) = 0$

$N(i,i+2) = 0$

From there, use recurrence

COMP561: Computational Biology Methods & Research

RNA minimum free energy secondary structures

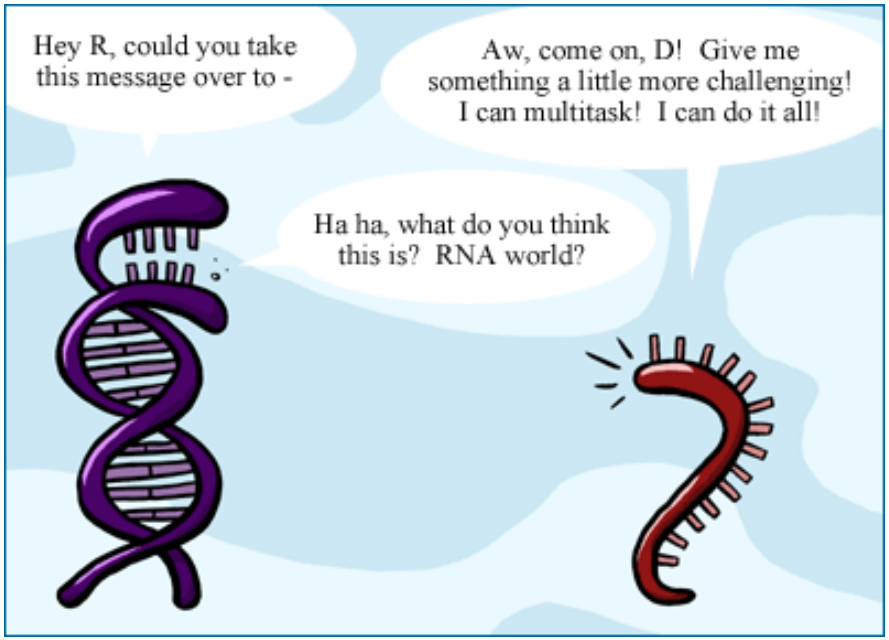
Jérôme Waldispühl
School of Computer Science, McGill

RNA world

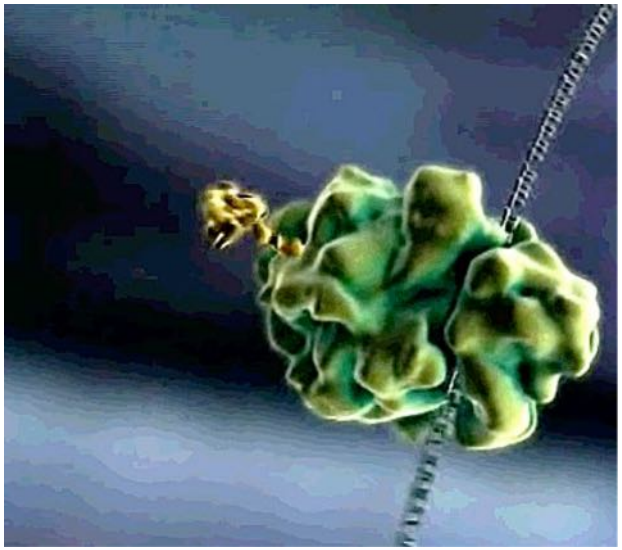
In prebiotic world, RNA thought to have filled two distinct roles:

1. an information carrying role because of RNA's ability (in principle) to self-replicate,
2. a catalytic role, because of RNA's ability to form complicated 3D shapes.

Over time, DNA replaced RNA in
Its first role, while proteins replaced
RNA in its second role.



RNA classification

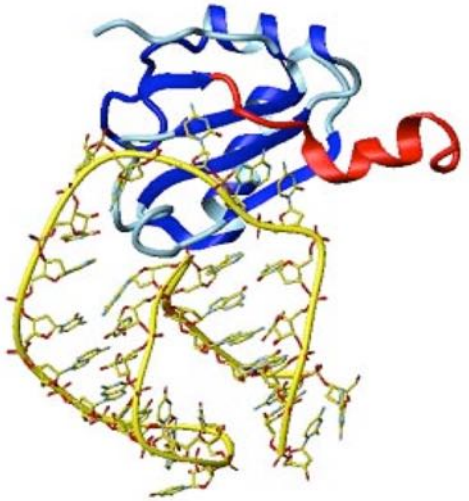


Messenger RNA:

- Carry genetic information,
- Structure less important.

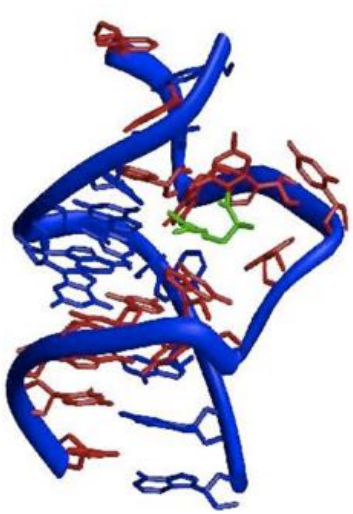
Non-coding RNA:

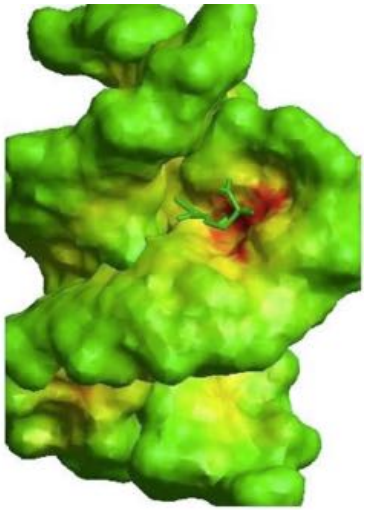
- Functional,
- Structure is important.

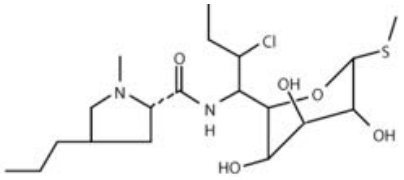


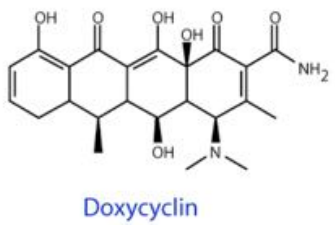
RNA structure and function

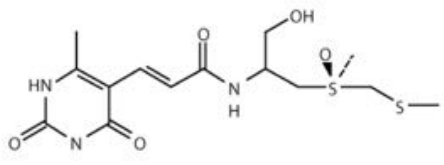
- RNAs have a 3D structure,
- This 3D structure allow complex functions,
- The variety of RNA structures allow the specific recognition of a wide range of ligands,
- Some molecules target these RNA structures (antibiotics, antiviruses):

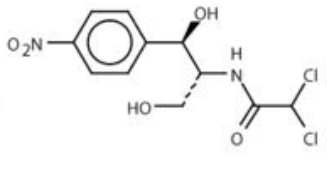




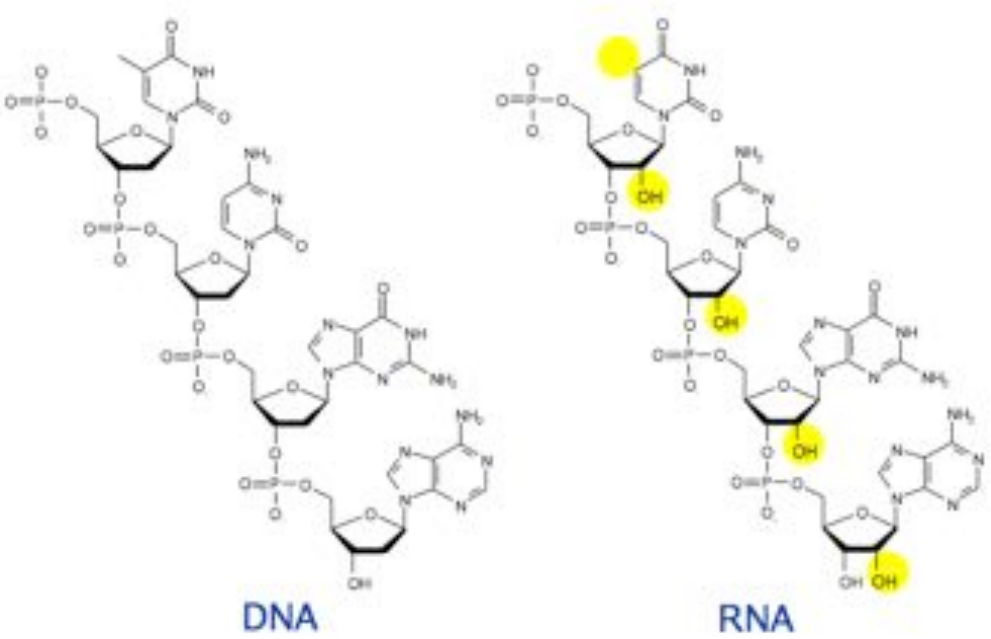








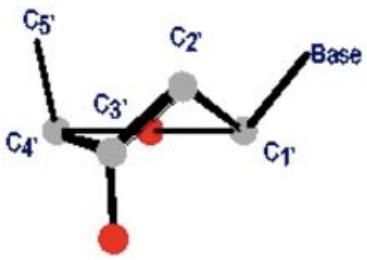
RNA vs DNA: Chemical nature



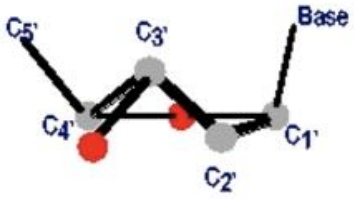
- 2’-OH group attached to sugar (instead of 2’-H): *more polar*
- Substitution of thymine by uracile = suppression of group 5-CH3

Small modifications => big effects

RNA vs DNA: Modification of the local and global geometry



C2’ endo

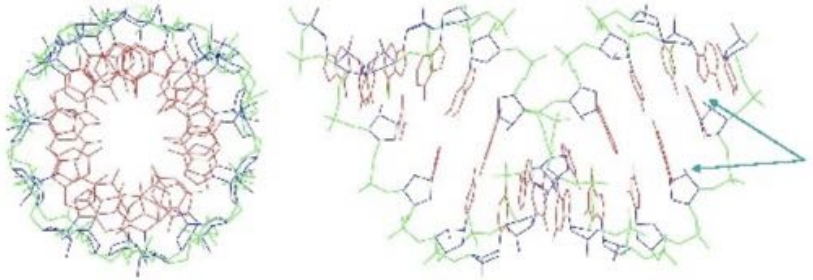


C3’ endo

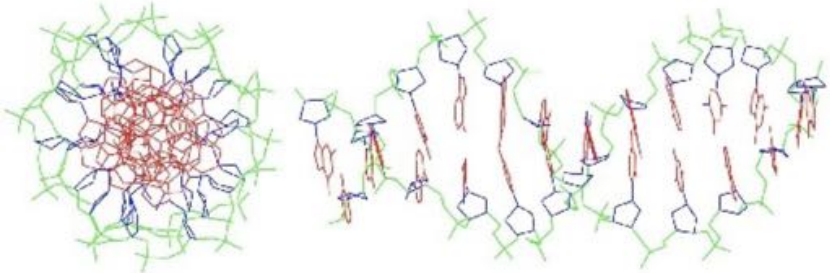
:Local conformation

Global conformation:

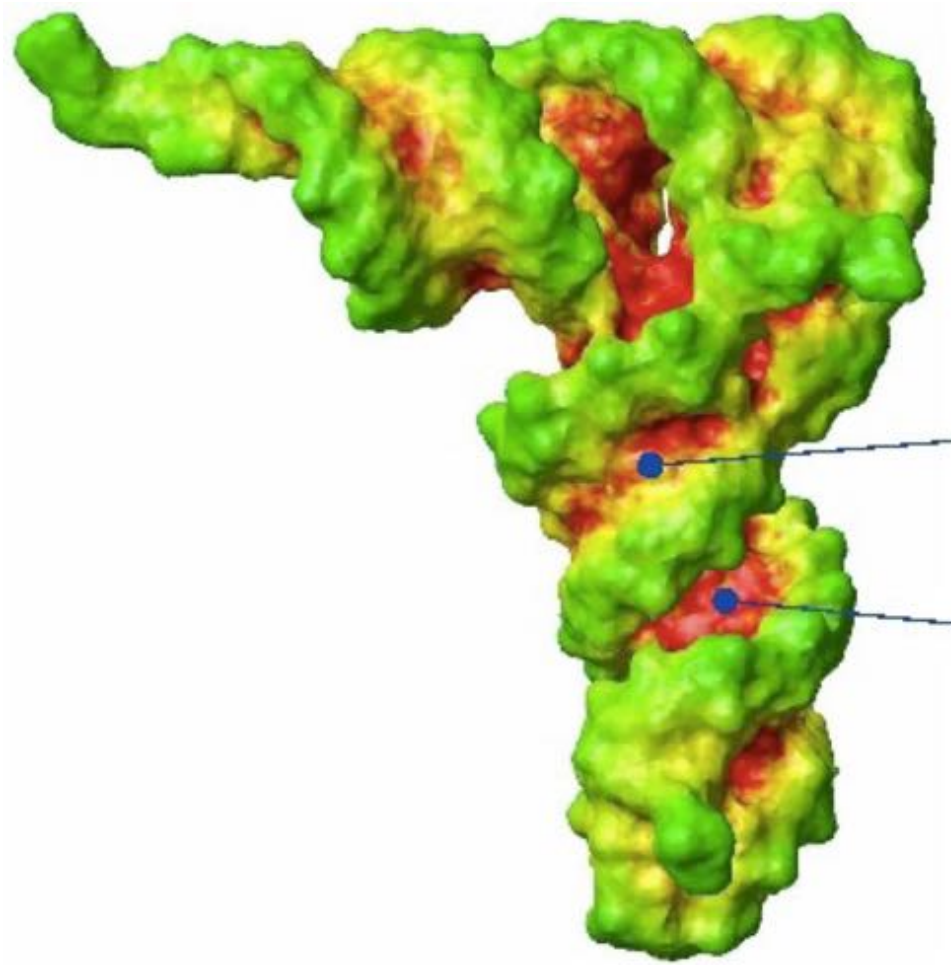
RNA favorite:



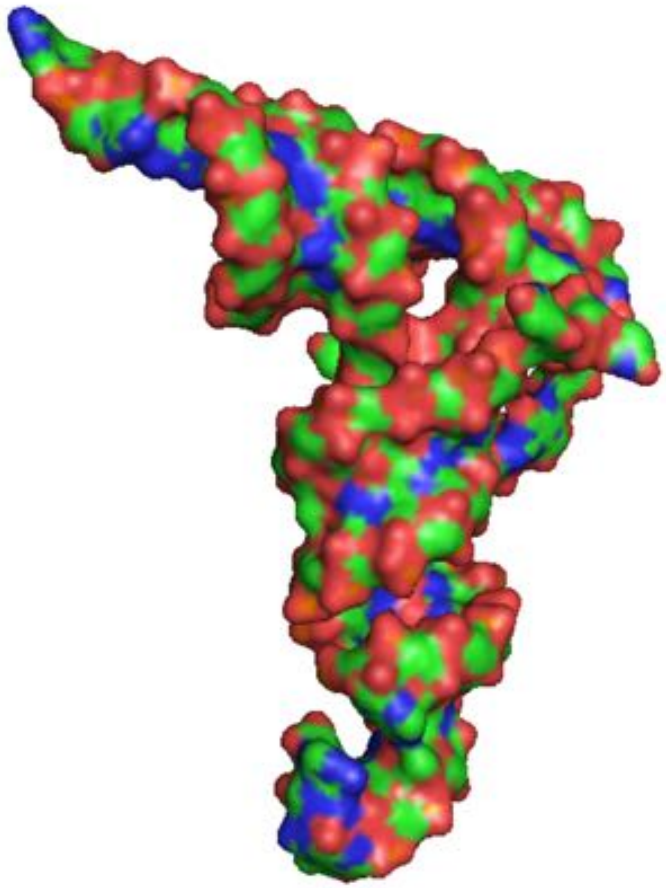
DNA favorite:



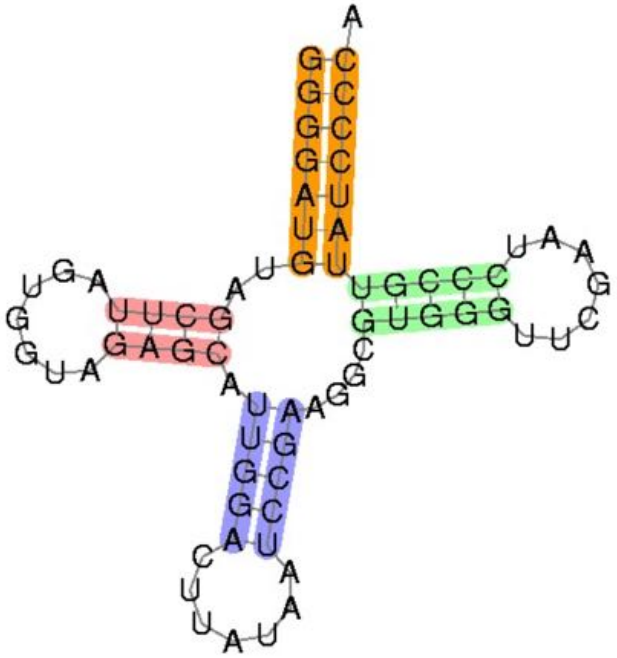
RNA vs DNA: Consequence of the modification of the geometry



RNA secondary structure



The **secondary structure** is the set of canonical base-pairs forming the scaffold of the 3D structure.



RNA secondary structure

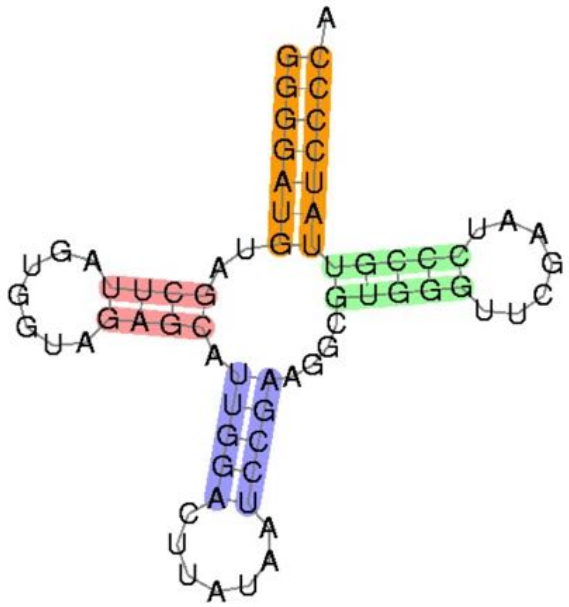
Central assumption: RNA secondary structure forms before the tertiary structure.

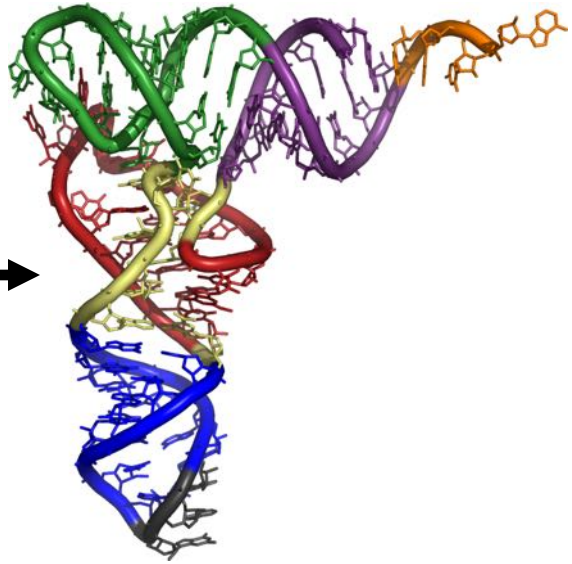
Primary structure

Secondary structure

Tertiary structure

cgcgggggttgatataatataaaaaataat
aaataataataataataattatcatcattt
ccgacccatattataataatacgggttgg
aaatatagatataatatttattatattgat
ataatacatatatataagttagaggaaat
gttgtttaaaggttaaactgttagattgca
aatctacacatttagagttcgattctcttc
atttcttatatatatactacccacgcg





This class: Secondary structure prediction using energy minimization principles

Principle of minimum energy

For a closed system with fixed entropy, the total energy is minimized at equilibrium.

Application to RNA folding:

- Closed system: Isolated RNA molecule
- Energy of system: Folding energy of the RNA
- State of the system: An RNA (secondary) structure

Definition

- Let $\omega \in \{A,C,G,U\}^*$ be a RNA sequence
- Let Δ be the ensemble of all secondary structures S compatible with ω.
- Let $E(S, \omega)$ be the free energy on ω folded in S.

Then, the minimum free energy (MFE) of ω is:

$$MFE(\omega) = \min_{S \in \Delta}(E(S,\omega))$$

And the minimum free energy secondary structure of ω is the structure S such that $E(S,\omega) = MFE(\omega)$.

Note: Here, we assume it exists an unique structure S satisfying the equation.

Free energy of a RNA secondary structure

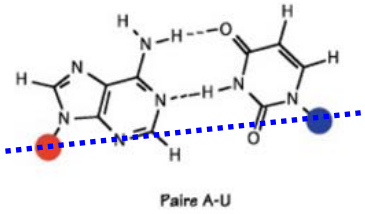
First approximation:

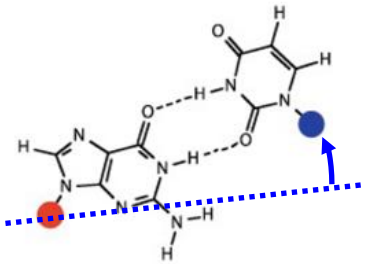
- Base pairs stabilize the RNA secondary structure
- Free energy ≡ number of base pairs

Second approximation:

- Base pairs have different energies
- Free energy: sum of all base pair energies

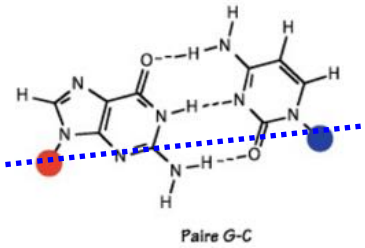
A-U or U-A : 2





G-U or U-G : 1

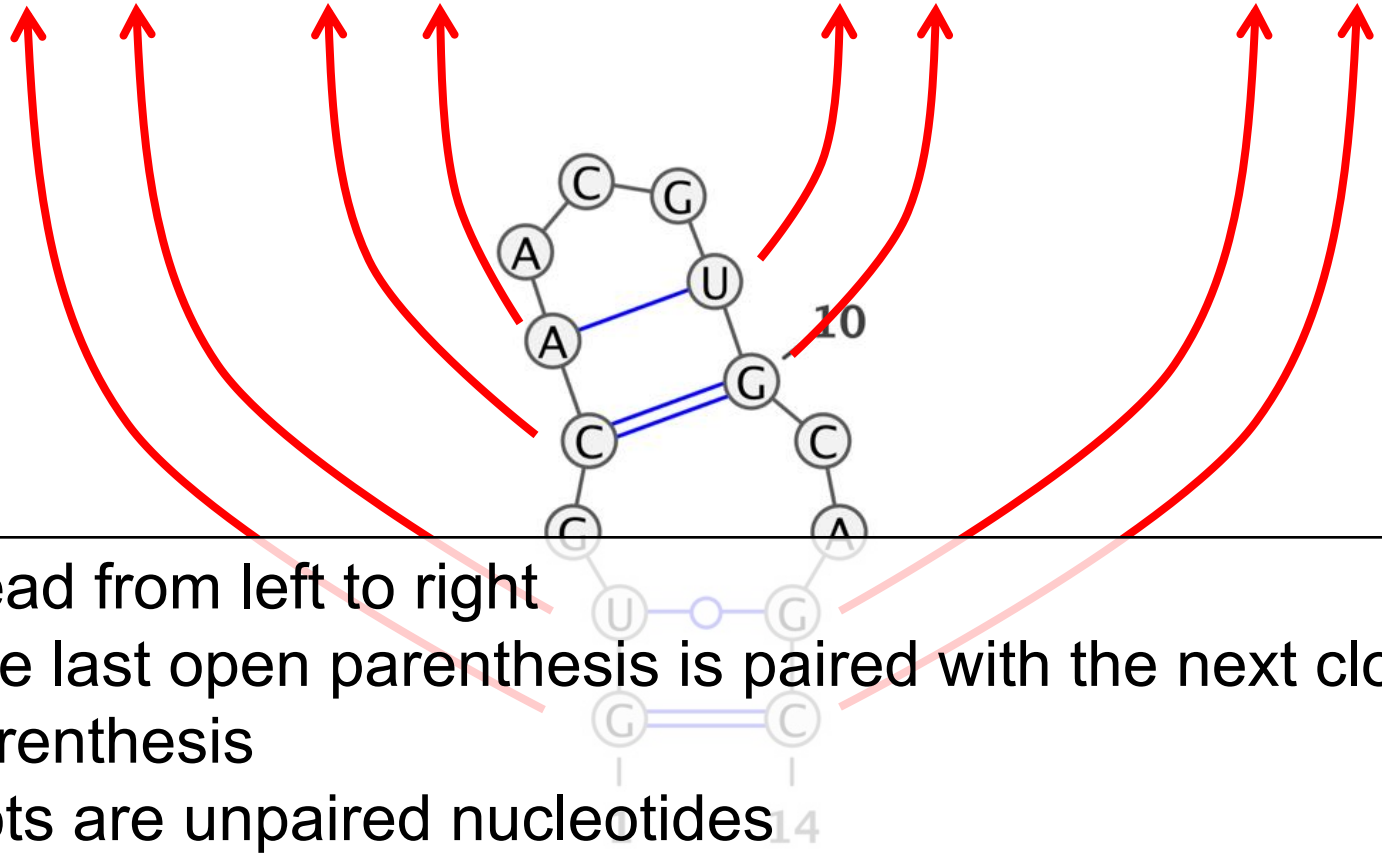
C-G or G-C : 3



Modeling RNA secondary structure

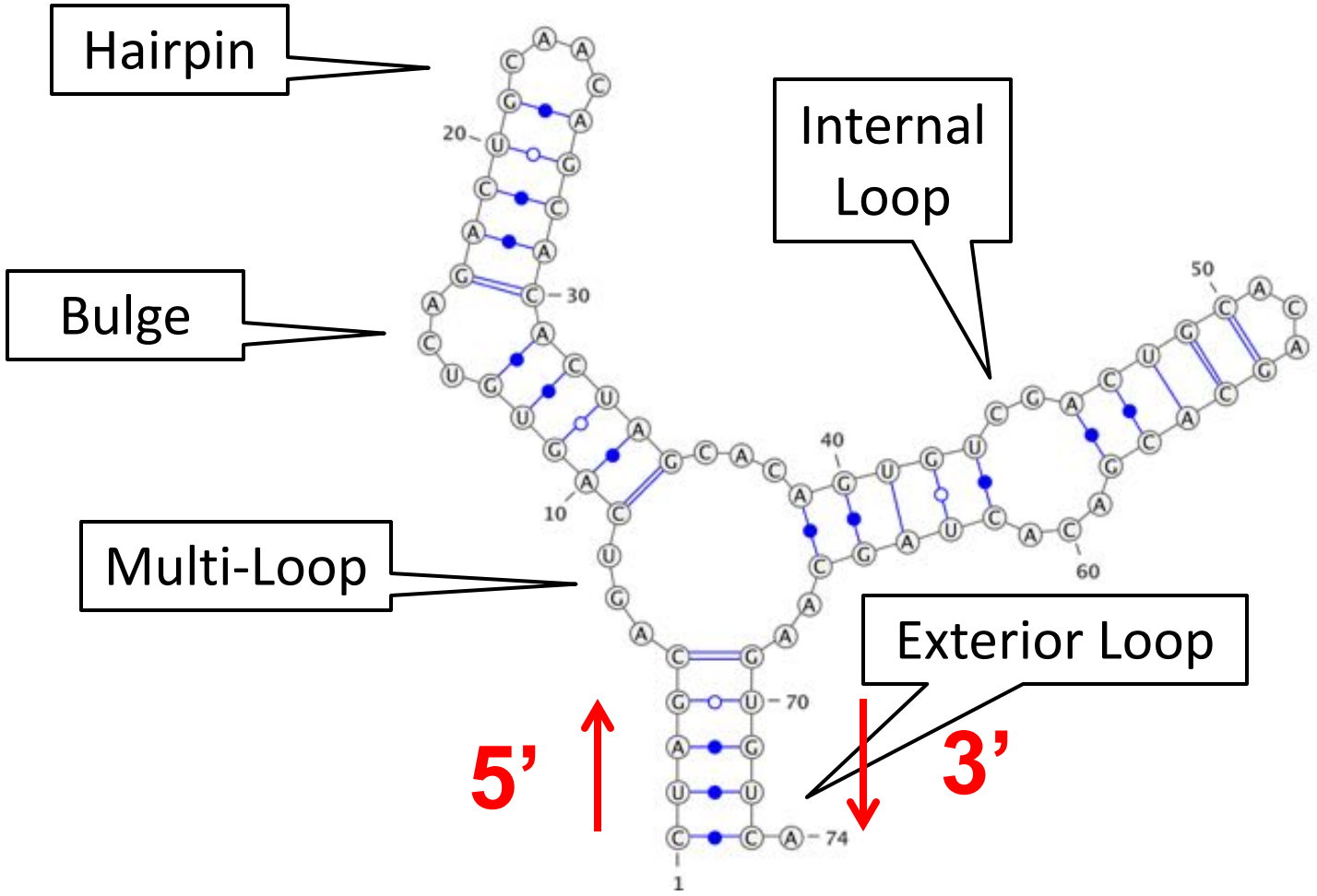
G U G C A A C G U G C A G C

((. ((. . .)) . .))



- Read from left to right
- The last open parenthesis is paired with the next closed parenthesis
- Dots are unpaired nucleotides

RNA Nomenclature

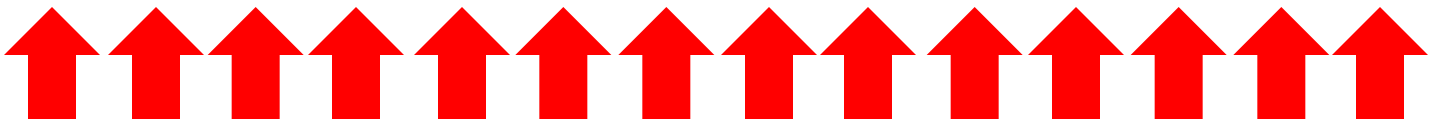


CUAGCAGUCAGUGUCAGACUGCAACAGCACACUAGCACAGUGUCGACUGCACAGCACGACACUAGCAAGUGUCA

((((((...((((((...((((((....))))))))))))...((((((..((((((...))))))...))))))..))))))

String to contacts

```
1 2 3 4 5 6 7 8 9 0 1 2 3 4
G U G C A A C G U G C A G C
( ( . ( ( . . . ) ) . . ) )
```



1	2	4	5		

Contacts:
(5,9)
(4,10)
(2,13)
(1,14)

Notes:

- Assume no crossing interactions.
- Each it exist a closing parenthesis for each opening one.
- Base pair opens before closing.

Contacts to String

Principle: Look up at the rightmost position.

- If there is a base pair, print it and print recursively before and between this base pair.
- Otherwise, print unpaired and move one position left.

Contacts: (4,10), (5,8), (1,3).

```
1234567890
xxx(xxxxx)    f(1,10)
xxx(xxxx.)    f(1,3)+f(5,9)
xxx((xx).)    f(1,3)+f(5,8)
xxx((x.).)    f(1,3)+f(6,7)
xxx((..))     f(1,3)+f(6,6)
(x)(.(..))    f(1,3)
(.)(.(..))    f(2,2)
```

RNA secondary structure prediction using dynamic programming

Compute the secondary structure with the maximal number of canonical base pairs (Nussinov-Jacobson, 1980).

$$\delta(i,j) = \begin{cases} 1 & (i,j)\ is\ a\ valid\ base\ pair \\ -\infty & Otherwise \end{cases}$$

Algorithm (Nussinov-Jacobson):

$$M(i,j) = \begin{cases} 0 & if\ i \geq j - \theta \\ M(i,j-1) & No\ base\ pair\ at\ j \\ \max\limits_{i \leq k < j-\theta} \left(\delta(k,j) + M(i,k-1) + M(k+1,j-1) \right) & (k,j)\ is\ a\ base\ pair \end{cases}$$

Example

Ω=GCCAGU, θ=1

	0	1	2	3	4	5	6
0	M	G	C	C	A	G	U
1	G	0	0	1	1	1	2
2	C	–	0	0	0	1	1
3	C	–	–	0	0	1	1
4	A	–	–	–	0	0	1
5	G	–	–	–	–	0	0
6	U	–	–	–	–	–	0

M(1,6)

M(1,3) = max(M(1,2),δ(1,3)+M(2,2)) = max(0,1+0) = 1

M(1,4) = max(M(1,3), δ(1,4)+M(2,3), δ(2,4)+M(1,1)+M(3,3))

= max(1,0+0,0+0+0) = 1

Backtracking

M(1,|ω|) returns the maximal number of base pairs but not the structure.

How do we retrieve the secondary structure?

Backtracking!

Idea: Once we know the value of M(1,|ω|), we can trace the base pairs that were used to obtain it.

Example

Ω=GCCAGU, θ=1

M	G	C	C	A	G	U
G	0	0	1	1	1	2
C	–	0	0	0	1	1
C	–	–	0	0	1	1
A	–	–	–	0	0	1
G	–	–	–	–	0	0
U	–	–	–	–	–	0

$$M(1,6)=2=\begin{cases} M(1,5)=1 \\ \delta(1,6)+M(2,5)=1+1=2 \\ M(1,1)+\delta(2,6)+M(3,5)=0+0+1=1 \\ M(1,2)+\delta(3,6)+M(4,5)=0+0+0=0 \\ M(1,3)+\delta(4,6)+M(5,5)=1+1+0=2 \end{cases}$$

Example (option 1)

Ω=GCCAGU, θ=1

M	G	C	C	A	G	U
G	0	0	1	1	1	2
C	–	0	0	0	1	1
C	–	–	0	0	1	1
A	–	–	–	0	0	1
G	–	–	–	–	0	0
U	–	–	–	–	–	0

(? ? ? ?) Base pairs={(1,6)}

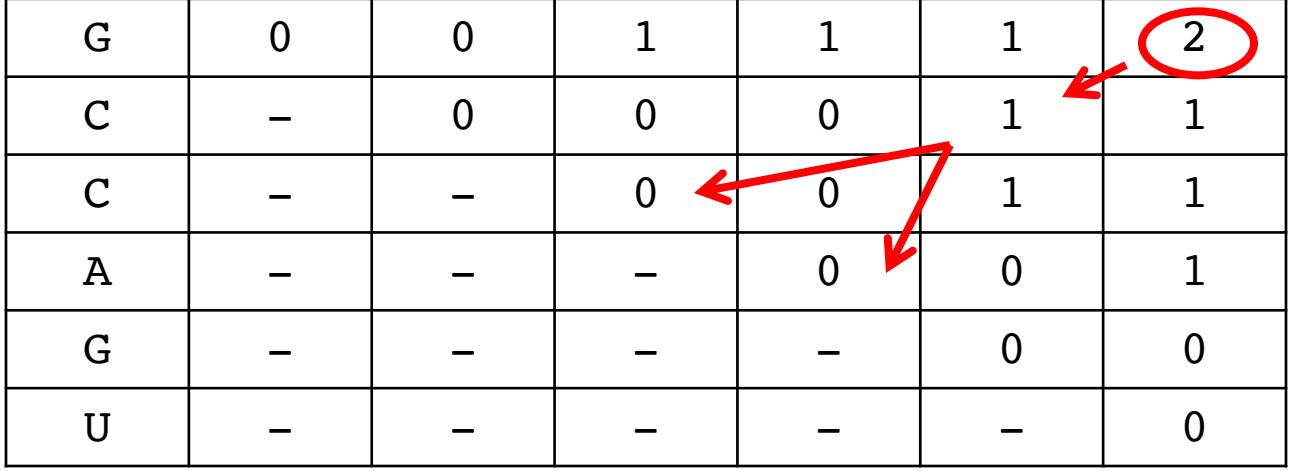
((? ?)) Base pairs={(1,6),(2,5)}

((. .)) Base pairs={(1,6),(2,5)}

Example (option 2)

Ω=GCCAGU, θ=1

M	G	C	C	A	G	U
G	0	0	1	1	1	2
C	–	0	0	0	1	1
C	–	–	0	0	1	1
A	–	–	–	0	0	1
G	–	–	–	–	0	0
U	–	–	–	–	–	0



(? ? ? ?) Base pairs={(1,6)}

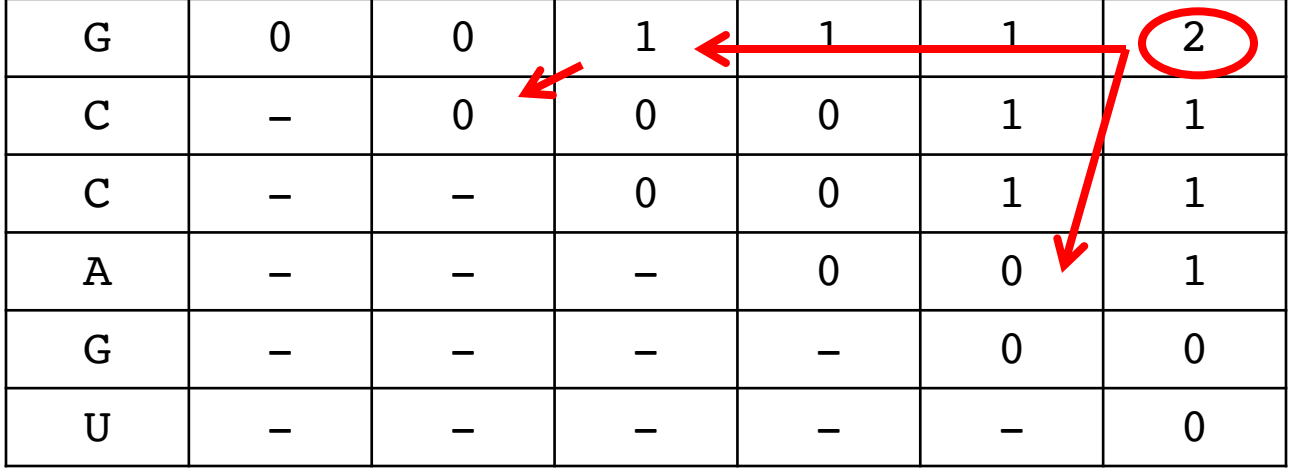
(? (?)) Base pairs={(1,6),(3,5)}

(. (.)) Base pairs={(1,6),(3,5)}

Example (option 3)

Ω=GCCAGU, θ=1

M	G	C	C	A	G	U
G	0	0	1	1	1	2
C	–	0	0	0	1	1
C	–	–	0	0	1	1
A	–	–	–	0	0	1
G	–	–	–	–	0	0
U	–	–	–	–	–	0



`???(?)` Base pairs={(4,6)}

`(?)(?)` Base pairs={(1,3),(4,6)}

`(.)(.)` Base pairs={(1,3),(4,6)}

RNA nearest neighbor energy model

Accuracy of the Nussinov-Jacobson model is moderate. We need a better model to weight the structures.

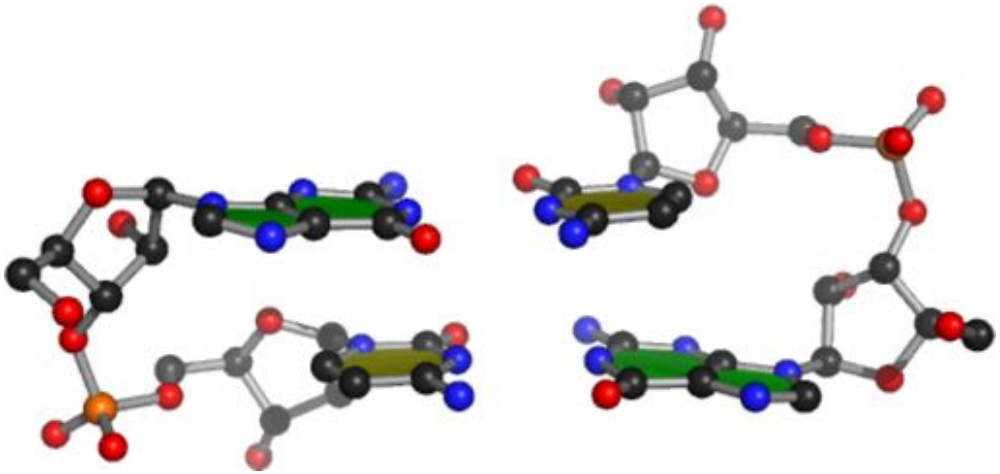
How?: Build an energy model from experimental measures (D. Turner).

But we need:

- *to define what are the important structural features that has to be evaluated.*
- *to keep the energy contribution local in order to allow a divide-and-conquer aproach (fast).*

Stacking base pairs

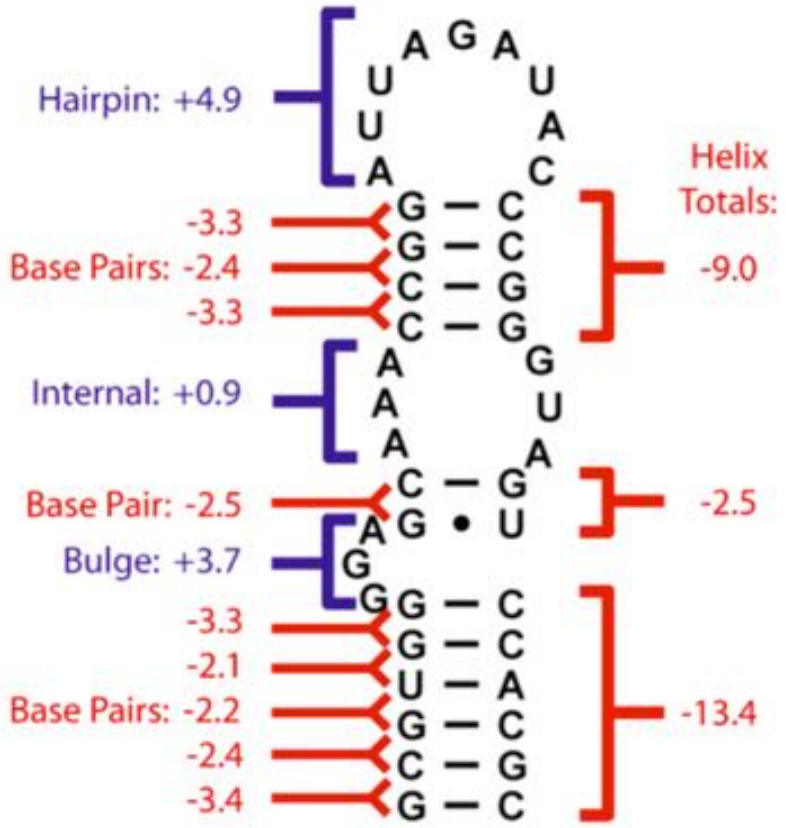
- Base stacking interactions between the pi orbitals of the bases' aromatic rings contribute to stability.
- GC stacking interactions with adjacent bases tend to be more favorable.



Note: Stacking energy are orientated.

$$\begin{matrix} 5' - CG - 3' \\ 3' - GC - 5' \end{matrix} \neq \begin{matrix} 5' - GC - 3' \\ 3' - CG - 5' \end{matrix}$$

Nearest Neighbor Energy Model



http://biomol.bme.utexas.edu

Zuker Algorithm

- Introduced by M. Zuker and P. Stiegler in 1981.
- Calculate the secondary structure with the MFE.
- Adaption of the Nussinov-Jacobson model to the thermodynamical nearest energy model.
- Algorithm originally implemented in the *mfold* software.
- Other popular implementation include:
  - *RNAfold* in the Vienna RNA Package
  - *RNAstructure*
  - *UNAfold* (*mfold* successor)

Want to know more?

Enroll COMP564 “Advanced Computational Biology Methods & Research” !!!

When? Winter 2019

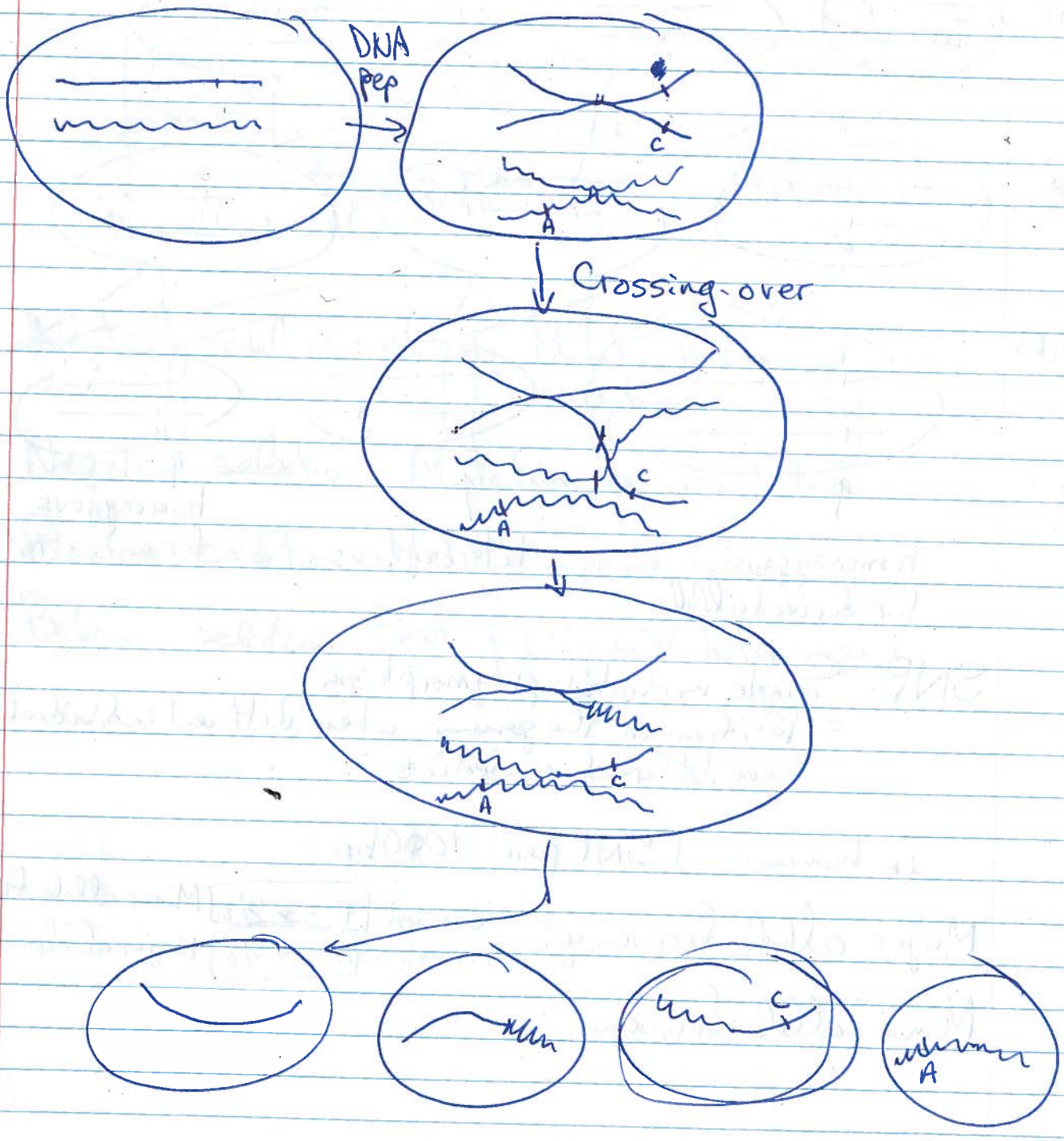
Why? You liked COMP561 and want to know about bioinformatics.

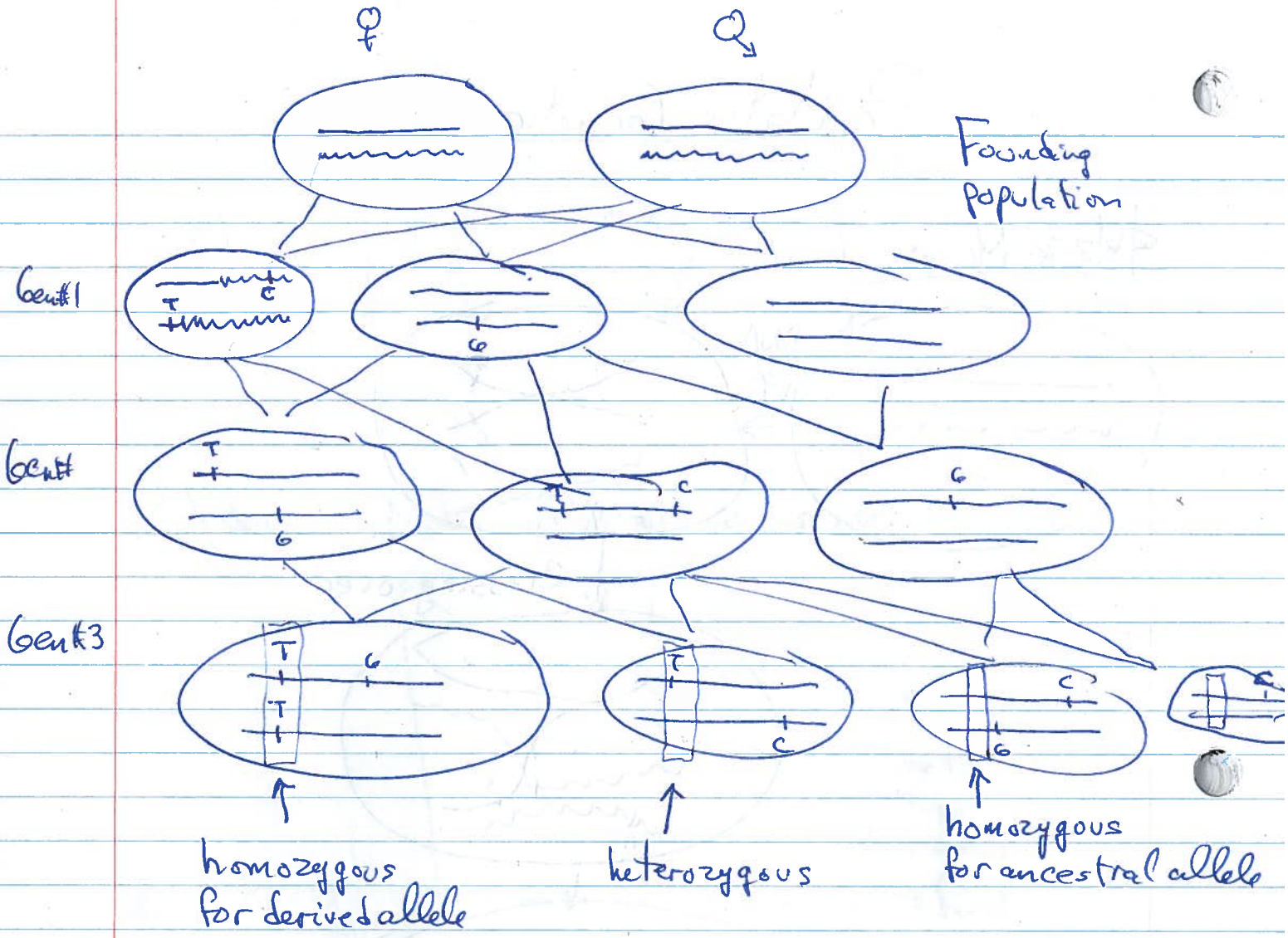
What? We cover fundamental algorithms in computational structural & system biology.

jeromew@cs.mcgill.ca

Population Genetics

~~Meio~~ Meiosis





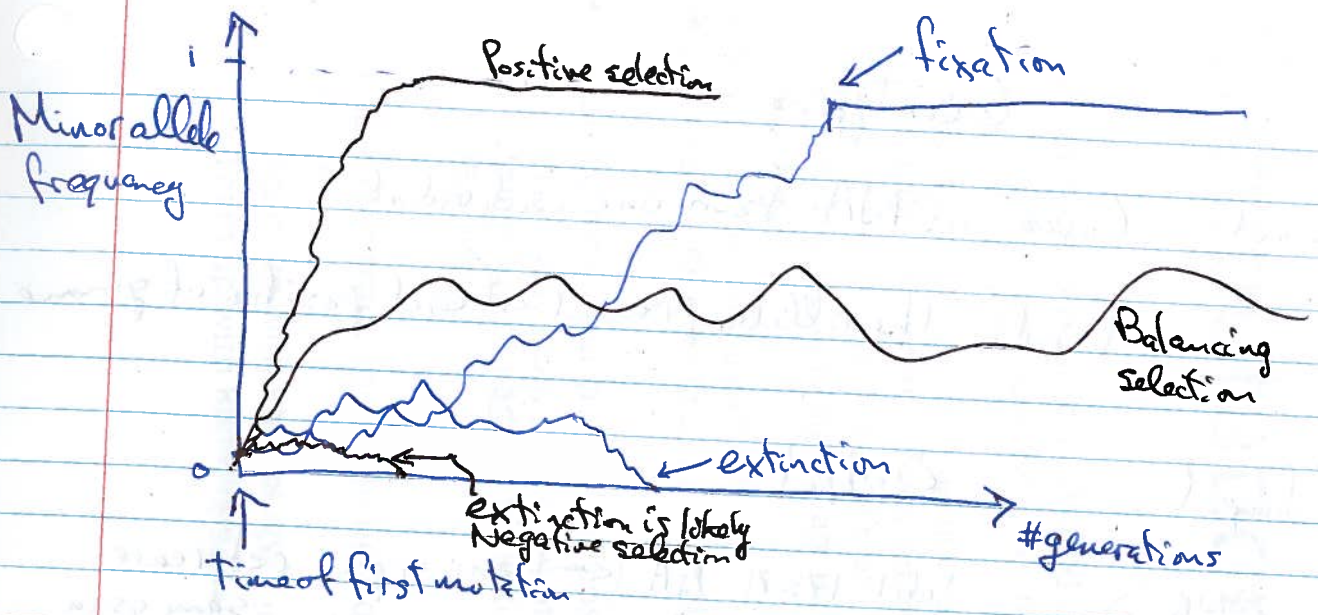
SNP: Single nucleotide polymorphism
= Position in the genome where different individuals have different nucleotides

In human: ~1 SNP per 1000bp

Major allele frequency:

Minor allele frequency:

derived: T → 3/8 Minor allele freq: 0.375
ancestral: A → 5/8 Major allele: 0.625



— Neutral model of selection: Mutation has no consequence on fitness

— Negative selection: Mutation reduces fitness

Positive selection: Mutation improves fitness

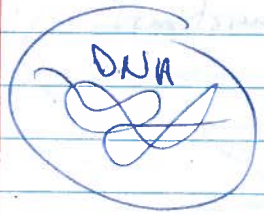
Balancing selection: Best fit is with heterozygous

Genotyping

Goal: Given: DNA from one individual

Find: The alleles present at each position of genome

Input → Output



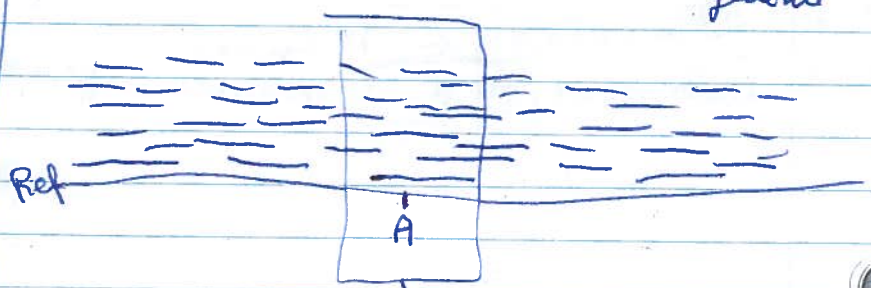
chr1	173271	AA	← homozygous reference = same as in reference human genome
chr2	173471	AC	← heterozygous
chr7	...	TT	← homozygous non-reference

- ~~Gen~~ Genotyping arrays
- Genotyping by sequencing

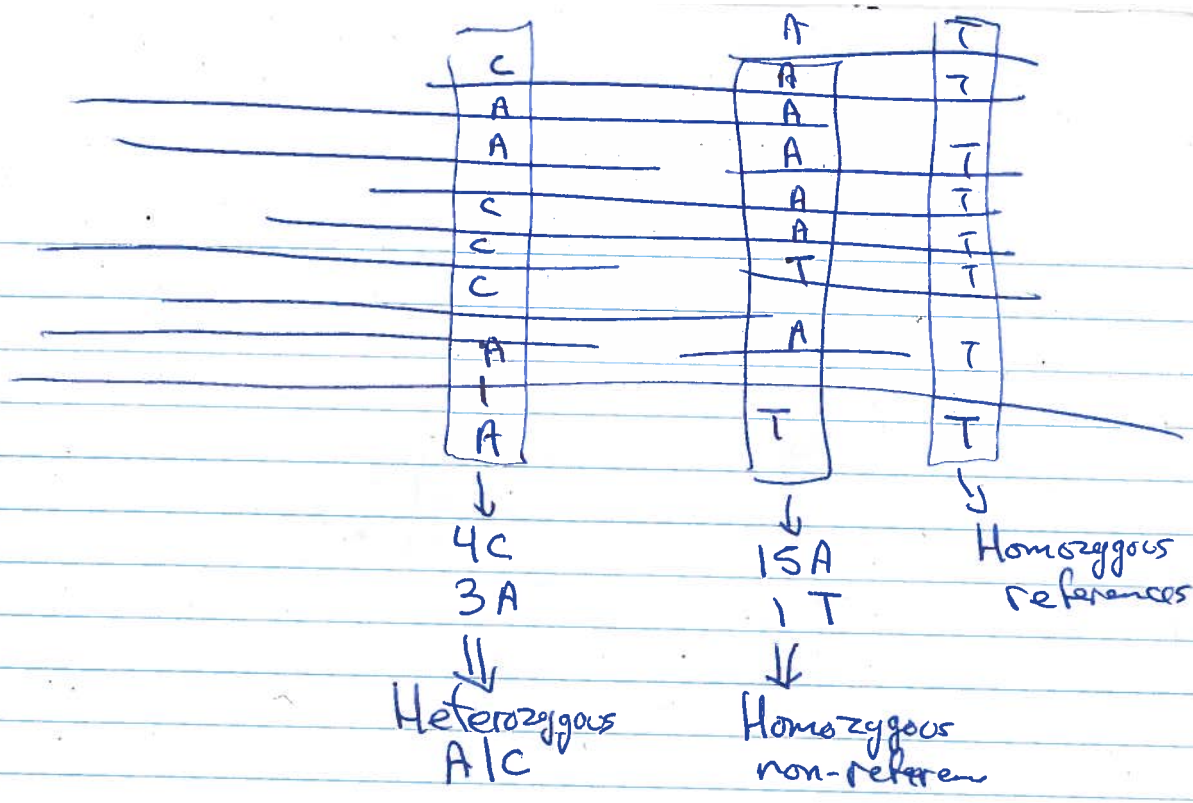
① Extract + fragment + sequence DNA

Fastq file | ACTAAAC..
ATTACACCA

② Align (map) each read to reference human genome



Next page


C
A
A
C
C
C
A
A
A
A
A
A
A
T
T
T
T
T
T
T
T
4C
3A
15A
1T
Homozygous
references
Heterozygous
A/C
Homozygous
non-referen

Suffix Arrays

CMSC 858S

Suffix Arrays

- Even though Suffix Trees are O(n) space, the constant hidden by the big-Oh notation is somewhat "big": ≈ 20 bytes / character in good implementations.

- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. "Linear" but large.

- Suffix arrays are a more efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

- Slight space vs. time tradeoff.

Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

index of suffix suffix of s

sort the suffixes alphabetically

the indices just "come along for the ride"

8	$
5	atg$
1	attcatg$
4	catg$
7	g$
3	tcatg$
6	tg$
2	ttcatg$

Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

index of suffix suffix of s

sort the suffixes alphabetically

the indices just "come along for the ride"

8
5
1
4
7
3
6
2

Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

1	cattcat$
2	attcat$
3	ttcat$
4	tcat$
5	cat$
6	at$
7	t$
8	$

index of suffix suffix of s

sort the suffixes alphabetically

the indices just "come along for the ride"

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

1	cattcat$
2	attcat$
3	ttcat$
4	tcat$
5	cat$
6	at$
7	t$
8	$

index of suffix suffix of s

sort the suffixes alphabetically

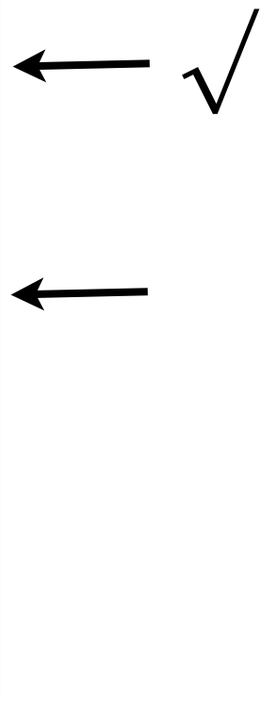
the indices just "come along for the ride"

8
6
2
5
1
7
4
3

Search via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$



- Does string "at" occur in s?
- Binary search to find "at".

- What about "tt"?

Counting via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

- How many times does “at” occur in the string?
- All the suffixes that start with “at” will be next to each other in the array.
- Find one suffix that starts with “at” (using binary search).
- Then count the neighboring sequences that start with at.

K-mer counting

Problem: Given a string s, an integer k, output all pairs (b, i) such that b is a length-k substring of s that occurs exactly i times.

k = 2

		CurrentCount	
8	$	1	
6	at$	1	
2	attcat$	2	
5	cat$	1	(at,2)
1	cattcat$	2	
7	t$	1	(ca,2)
4	tcat$	1	(t$,1)
3	ttcat$	1	(tc,1)
		1	(tt,1)

1. Build a suffix array.

2. Walk down the suffix array, keeping a CurrentCount count
   If the current suffix has length < k, skip it

   If the current suffix starts with the same length-k string as the previous suffix:
      increment CurrentCount
   else
      output CurrentCount and previous length-k suffix
      CurrentCount := 1
Output CurrentCount & length-k suffix.

Constructing Suffix Arrays

- Easy $O(n^2 \log n)$ algorithm:

  sort the n suffixes, which takes O(n log n) comparisons, where each comparison takes O(n).

- There are several direct O(n) algorithms for constructing suffix arrays that use very little space.
- The Skew Algorithm is one that is based on divide-and-conquer.

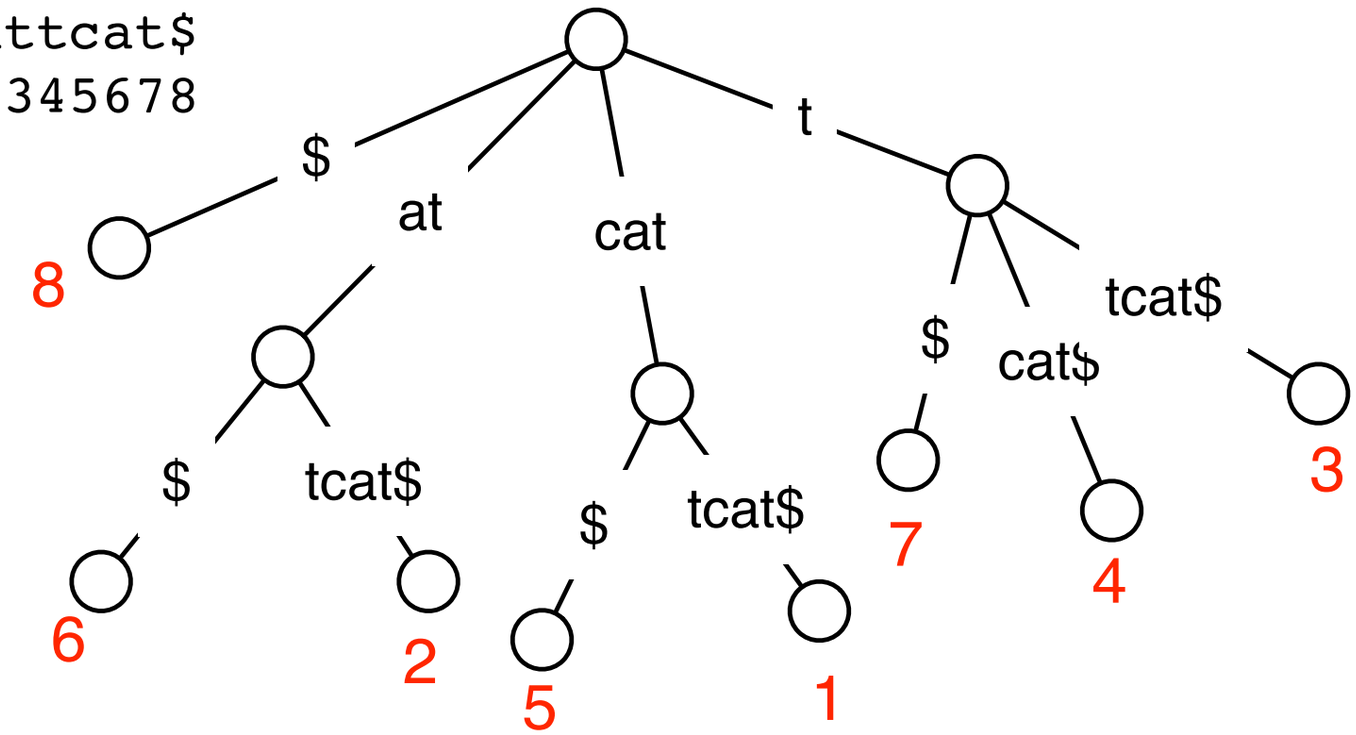
- An simple O(n) algorithm: build the suffix tree, and exploit the relationship between suffix trees and suffix arrays (next slide)

Relationship Between Suffix Trees & Suffix Arrays

$\Sigma = \{\$, a, c, t\}$

s = cattcat$

12345678



Red #s = starting position of the suffix ending at that leaf

Leaf labels left to right: 86251743

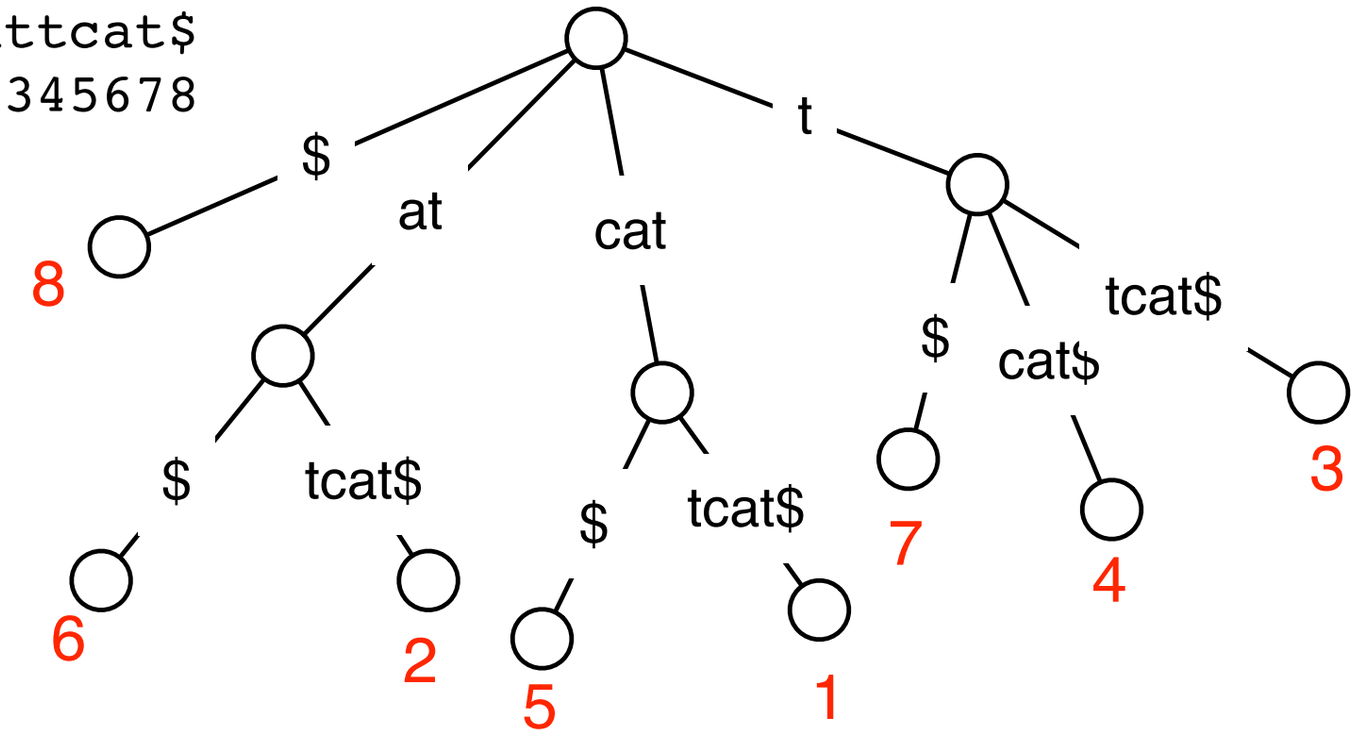
Edges leaving each node are sorted by label (left-to-right).

Relationship Between Suffix Trees & Suffix Arrays

$\Sigma = \{\$, a, c, t\}$

s = cattcat$

12345678



Red #s = starting position of the suffix ending at that leaf

Edges leaving each node are sorted by label (left-to-right).

Leaf labels left to right: 86251743

s = cattcat$

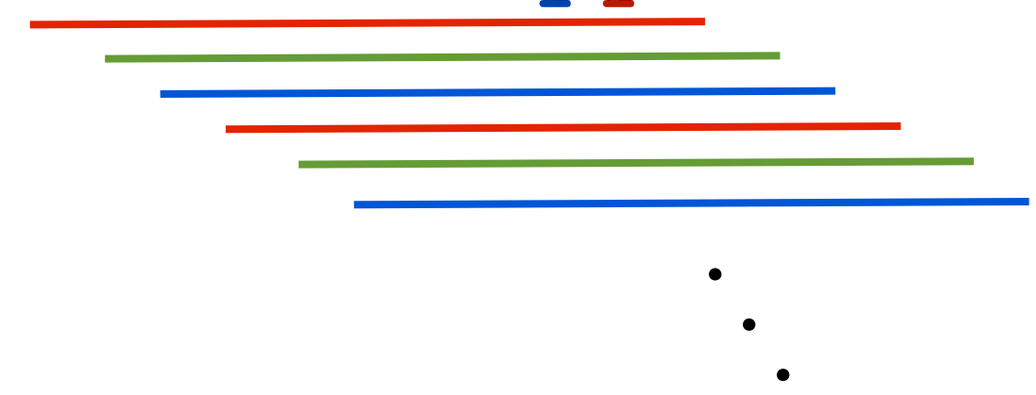
8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

The Skew Algorithm

Kärkkäinen & Sanders, 2003

- **Main idea: Divide suffixes into 3 groups:**
  - Those starting at positions i=0,3,6,9,.... (i mod 3 = 0)
  - Those starting at positions 1,4,7,10,... (i mod 3 = 1)
  - Those starting at positions 2,5,8,11,... (i mod 3 = 2)
- For simplicity, assume text length is a multiple of 3 after padding with a special character.

mississippi$$



Basic Outline:

- Recursively handle suffixes from the i mod 3 = 1 and i mod 3 = 2 groups.
- Merge the i mod 3 = 0 group at the end.

Handing the 1 and 2 groups

s = mississippi$$

iss	iss	ipp	i$$	ssi	ssi	ppi

triples for groups 1 and 2 groups

t = C C B A E E D

assign each triple a token in lexicographical order

AEED	4
BAEED	3
CBAEED	2
CCBAEED	1
D	7
ED	6
EED	5

recursively compute the suffix array for tokenized string

4321765

Every suffix of t corresponds to a suffix of s.

Relationship Between t and s

s = mississippi$$

iss	iss	ipp	i$$	ssi	ssi	ppi

C C B A E E D

t = CCBAEED

t_4

4321765

Key Point #1: The order of the suffixes of t is the same as the order of the group 1 & 2 suffixes of s.

Why?

Every suffix of t corresponds to some suffix of s (perhaps with some extra letters at the end of it --- in this case EED)

Because the tokens are sorted in the same order as the triples, the sort order of the suffix of t matches that of s.

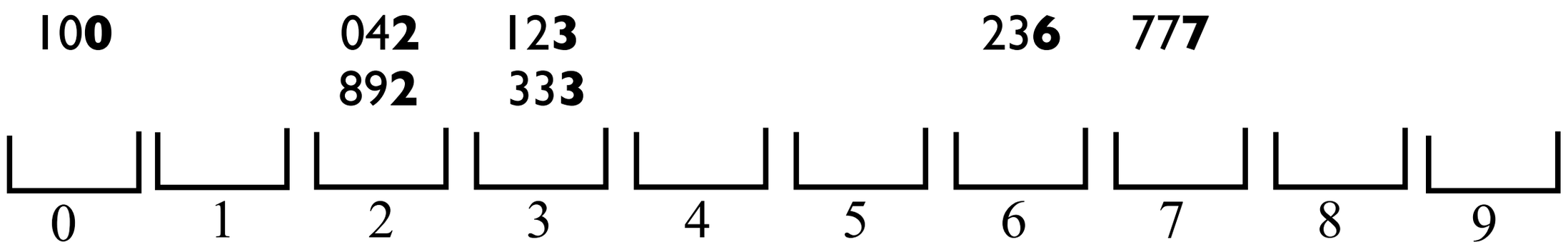
So: The recursive computational of the suffix array for t gives you the ordering of the group 1 and group 2 suffixes.

Radix Sort

- O(n)-time sort for n items when items can be divided into constant # of digits.

- Put into buckets based on least-significant digit, flatten, repeat with next-most significant digit, etc.

- Example items: 10**0** 12**3** 04**2** 33**3** 77**7** 89**2** 23**6**



- # of passes = # of digits
- Each pass goes through the numbers once.

Handling 0 Suffixes

- First: sort the group 0 suffixes, using the representation ($s[i]$, S_{i+1})
  - Since the S_{i+1} suffixes are already in the array sorted, we can just *stably* sort them with respect to $s[i]$, again using radix sort.

1,2-array:

ipp	iss	iss	i$$	ppi	ssi	ssi

0-array:

mis	pi$	sip	sis

- We have to merge the group 0 suffixes into the suffix array for group 1 and 2.
- Given suffix S_i and S_j, need to decide which should come first.
  - If S_i and S_j are both either group 1 or group 2, then the recursively computed suffix array gives the order.
  - If one of i or j is 0 (mod 3), see next slide.

Comparing 0 suffix S_j with 1 or 2 suffix S_i

Represent S_i and S_j using subsequent suffixes:

$i \pmod 3 = 1$:

$(s[i], S_{i+1}) \overset{?}{<} (s[j], S_{j+1})$

$\equiv 2 \pmod 3$ $\equiv 1 \pmod 3$

$i \pmod 3 = 2$:

$(s[i], s[i+1], S_{i+2}) \overset{?}{<} (s[j], s[j+1], S_{j+2})$

$\equiv 1 \pmod 3$ $\equiv 2 \pmod 3$

⇒ the suffixes can be compared quickly because the relative order of S_{i+1}, S_{j+1} or S_{i+2}, S_{j+2} is known from the 1,2-array we already computed.

Running Time

$$T(n) = O(n) + T(2n/3)$$

time to sort and merge

array in recursive calls is 2/3rds the size of starting array

Solves to $T(n) = O(n)$:

- Expand big-O notation: $T(n) \leq cn + T(2n/3)$ for some c.
- Guess: $T(n) \leq 3cn$
- Induction step: assume that is true for all $i < n$.
- $T(n) \leq cn + 3c(2n/3) = cn + 2cn = 3cn$ □

Recap

- Suffix arrays can be used to search and count substrings.

- Construction:
  - Easily constructed in $O(n^2 \log n)$
  - Simple algorithms to construct them in O(n) time.
  - More complicated algorithms to construct them in O(n) time using even less space.

- More space efficient than suffix trees: just storing the original string + a list of integers.

Suffix Trees

CMSC 423

Preprocessing Strings

- Over the next few lectures, we'll see several methods for preprocessing string data into data structures that make many questions (like searching) easy to answer:
  - Suffix Tries
  - Suffix Trees
  - Suffix Arrays
  - Borrows-Wheeler transform

- Typical setting: A long, known, and fixed text string (like a genome) and many unknown, changing query strings.
  - Allowed to preprocess the text string once in anticipation of the future unknown queries.
- Data structures will be useful in other settings as well.

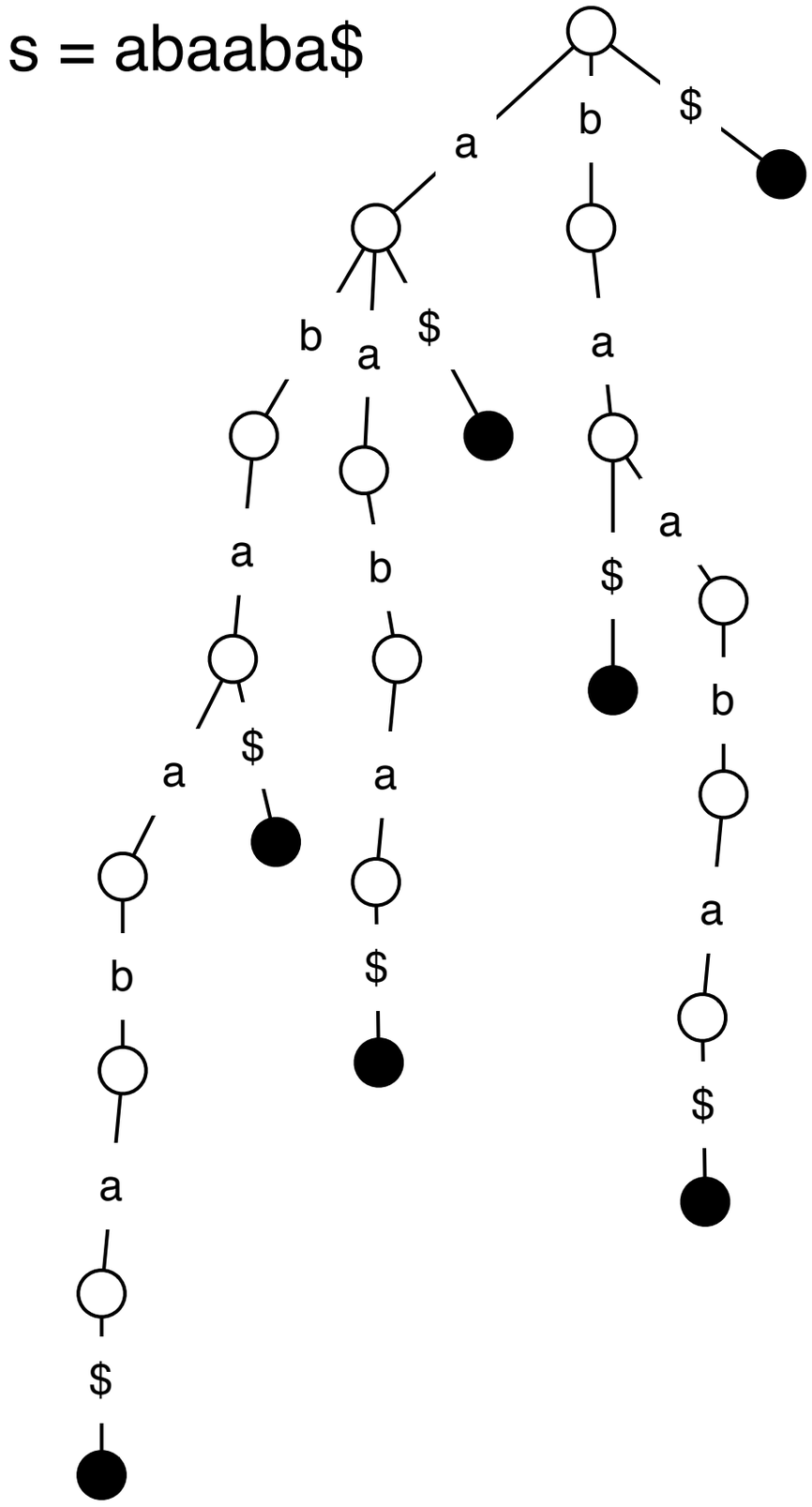
Suffix Tries

- A trie, pronounced "try", is a tree that exploits some structure in the keys
  - e.g. if the keys are strings, a binary search tree would compare the entire strings, but a trie would look at their individual characters

- Suffix trie are a space-efficient data structure to store a string that allows many kinds of queries to be answered quickly.

- Suffix trees are hugely important for searching large sequences like genomes. The basis for a tool called "MUMMer" (developed by UMD faculty).

Suffix Tries



SufTrie(s) = suffix trie representing string s.

Edges of the suffix trie are labeled with letters from the alphabet ∑ (say {A,C,G,T}).

Every path from the root to a solid node represents a suffix of s.

Every suffix of s is represented by some path from the root to a solid node.

Why are all the solid nodes leaves?
How many leaves will there be?

Processing Strings Using Suffix Tries

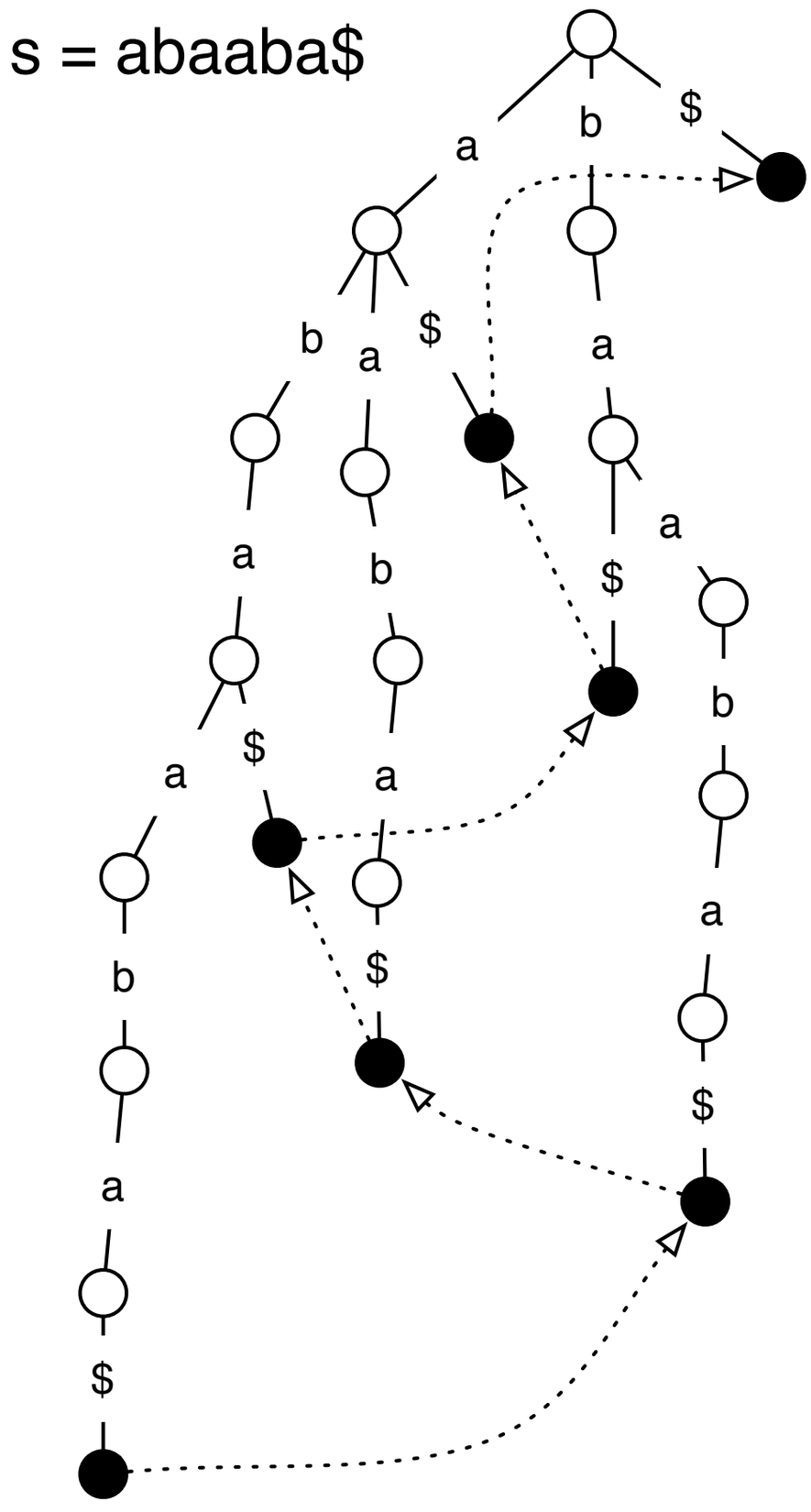
Given a suffix trie T, and a string q, how can we:

- determine whether q is a substring of T?
- check whether q is a suffix of T?
- count how many times q appears in T?
- find the longest repeat in T?
- find the longest common substring of T and q?

Main idea:

every substring of s is a prefix of some suffix of s.

Searching Suffix Tries



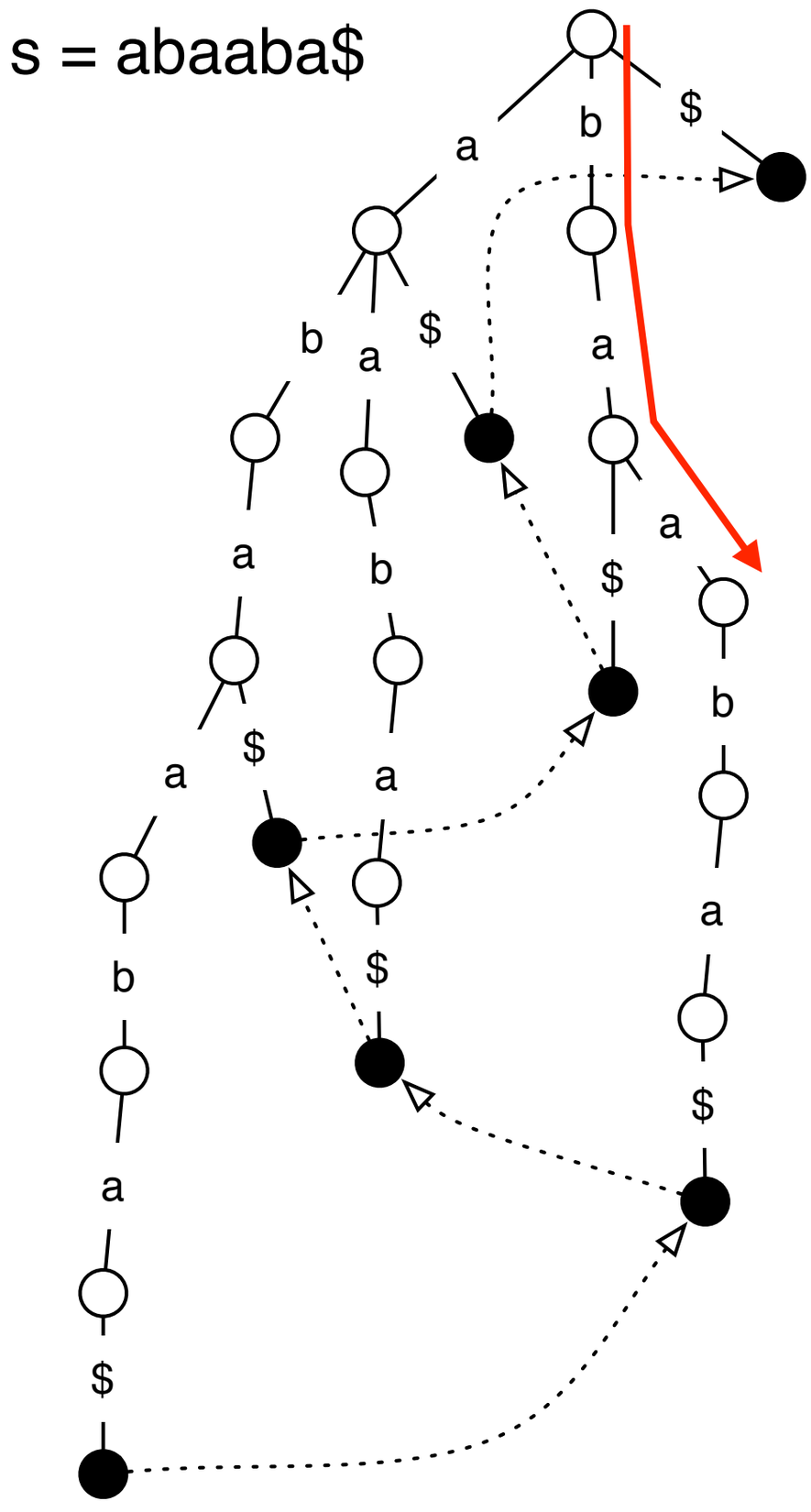
Is “baa” a substring of s?

Follow the path given by the query string.

After we’ve built the suffix trees, queries can be answered in time:

O(|query|)

regardless of the text size.



Searching Suffix Tries

Is “baa” a substring of s?

Follow the path given by the query string.

After we’ve built the suffix trees, queries can be answered in time:

O(|query|)

regardless of the text size.

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Check whether q is a **suffix** of T:

Count # of occurrences of q in T:

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Count # of occurrences of q in T:

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the number of occurrences of q.

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the number of occurrences of q.

Find the longest repeat in T:

Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the number of occurrences of q.

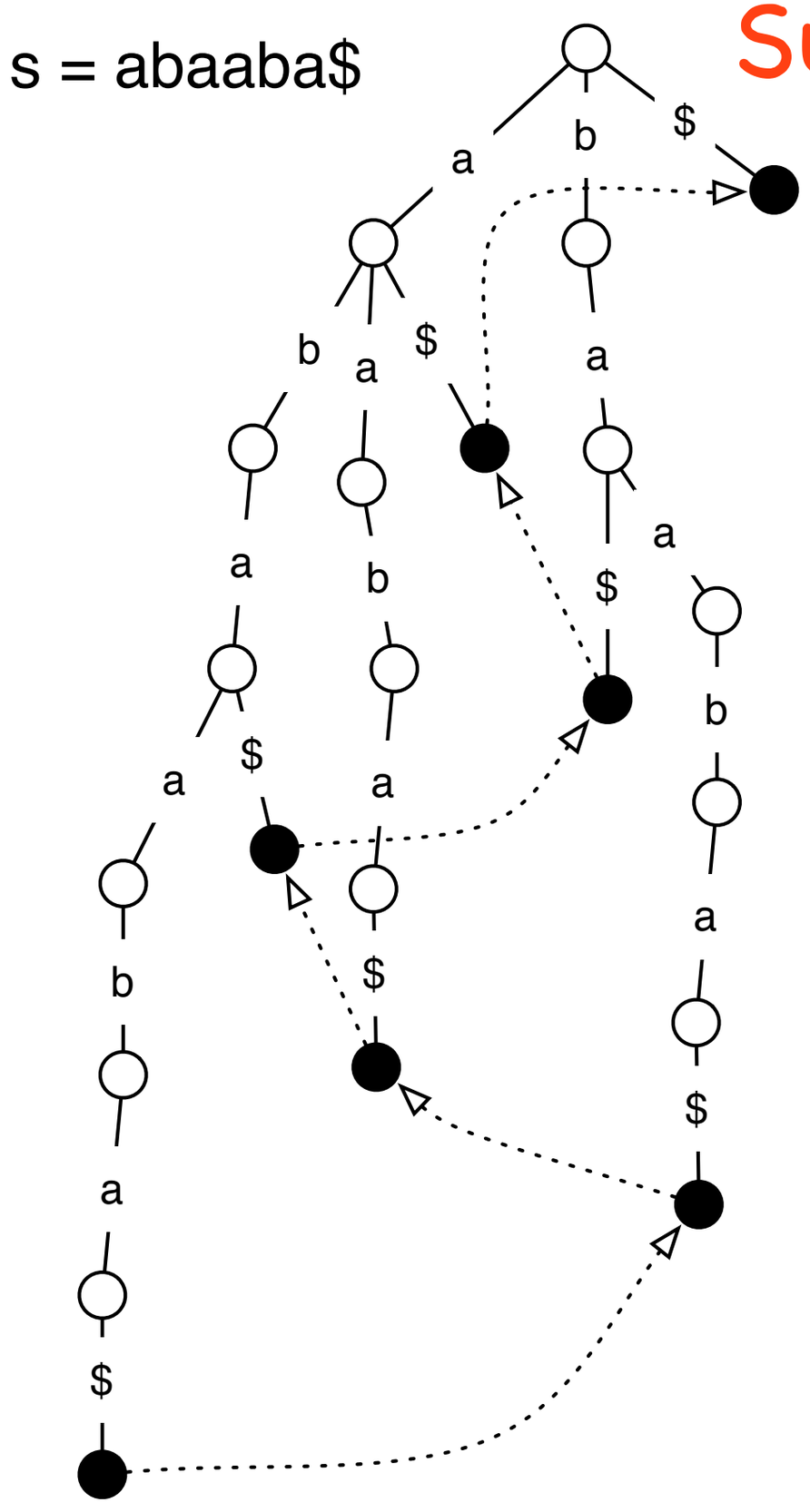
Find the longest repeat in T:

Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:

Start at the root, and follow the edge labeled with the lexicographically (alphabetically) smallest letter.

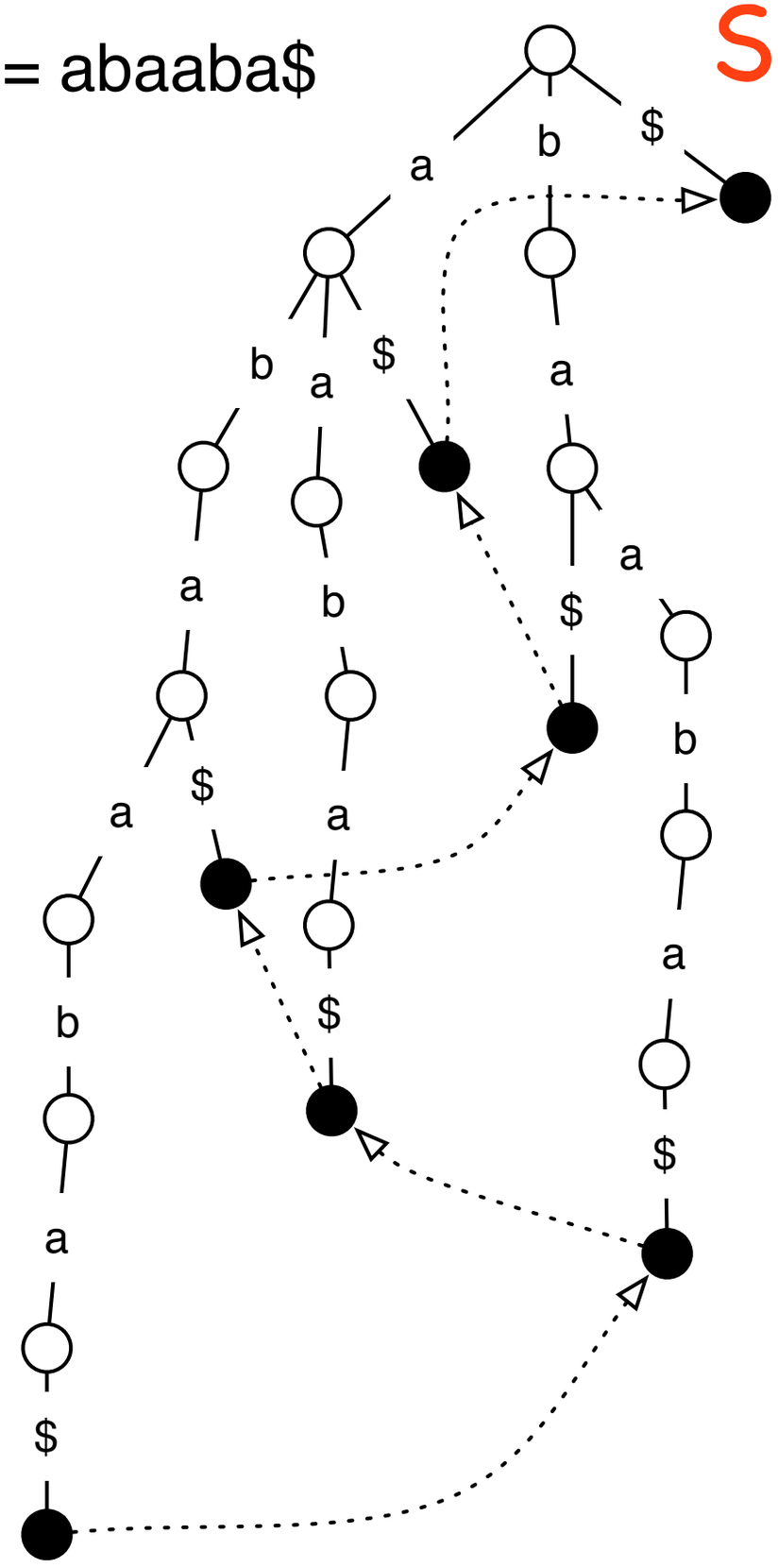
Suffix Links



- Suffix links connect node representing "xα" to a node representing "α".
- Most important suffix links are the ones connecting suffixes of the full string (shown at right).
- But every node has a suffix link.
  - Why?
  - How do we know a node representing α exists for every node representing xα?

Suffix Tries

s = abaaba$



A node represents the prefix of some suffix:

a b a a b a $

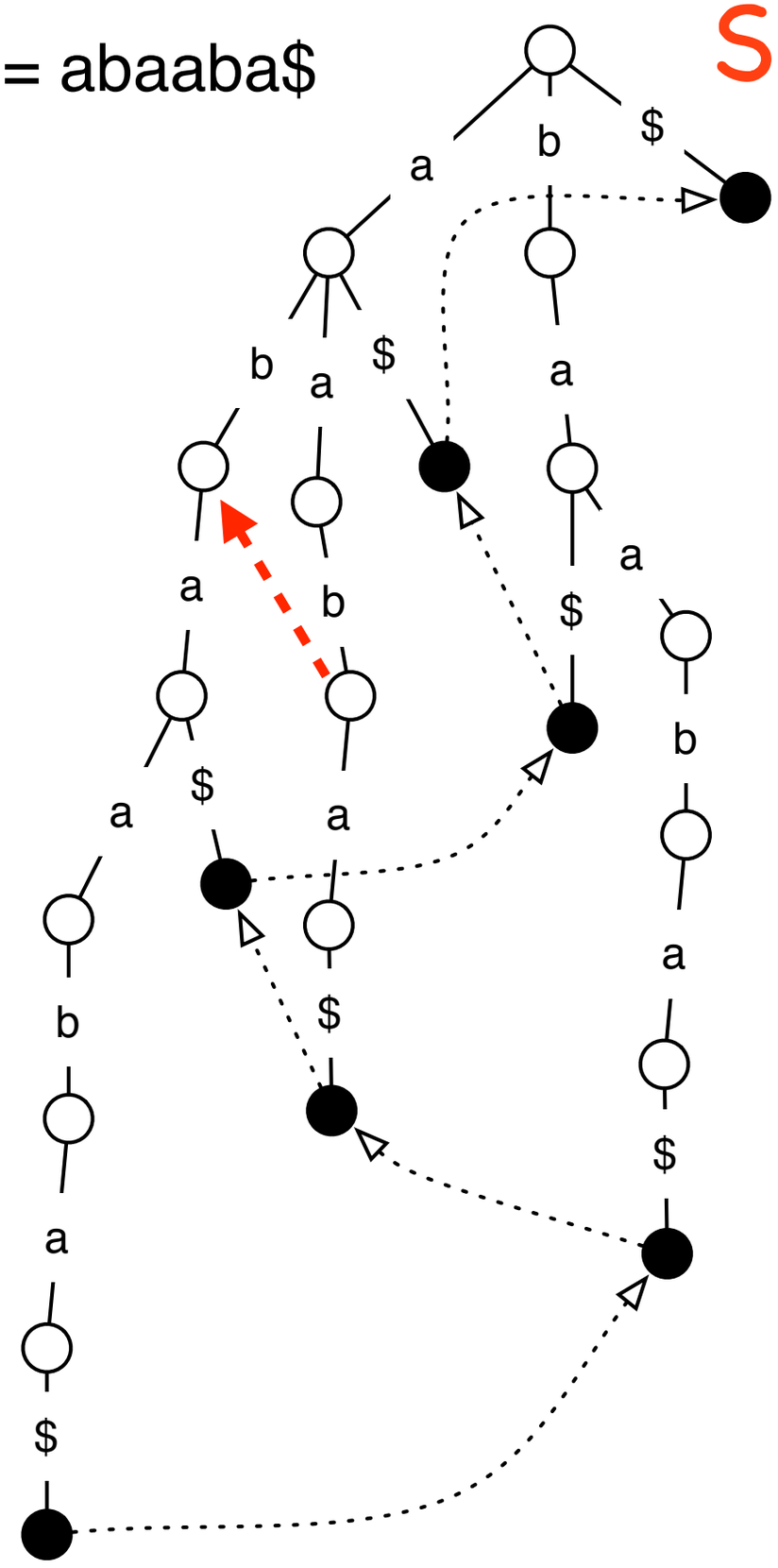
s

The node's suffix link should link to the prefix of the suffix s that is 1 character shorter.

Since the suffix trie contains all suffixes, it contains a path representing s, and therefore contains a node representing every prefix of s.

Suffix Tries

s = abaaba$



A node represents the prefix of some suffix:

a b a a b a $

s

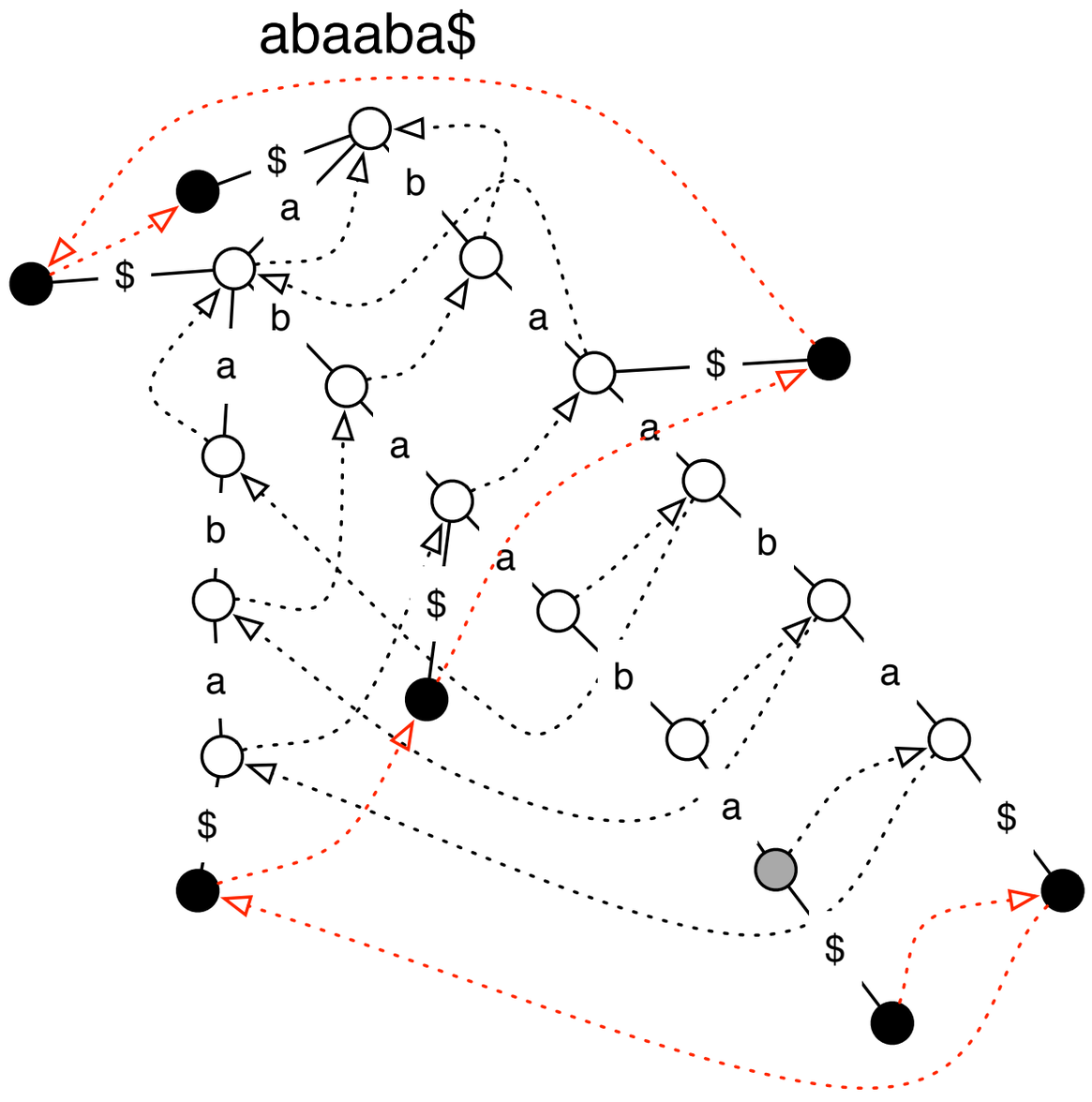
The node's suffix link should link to the prefix of the suffix s that is 1 character shorter.

Since the suffix trie contains all suffixes, it contains a path representing s, and therefore contains a node representing every prefix of s.

Applications of Suffix Tries (II)

Find the longest common substring of T and q:

T = abaaba$
q = bbaa

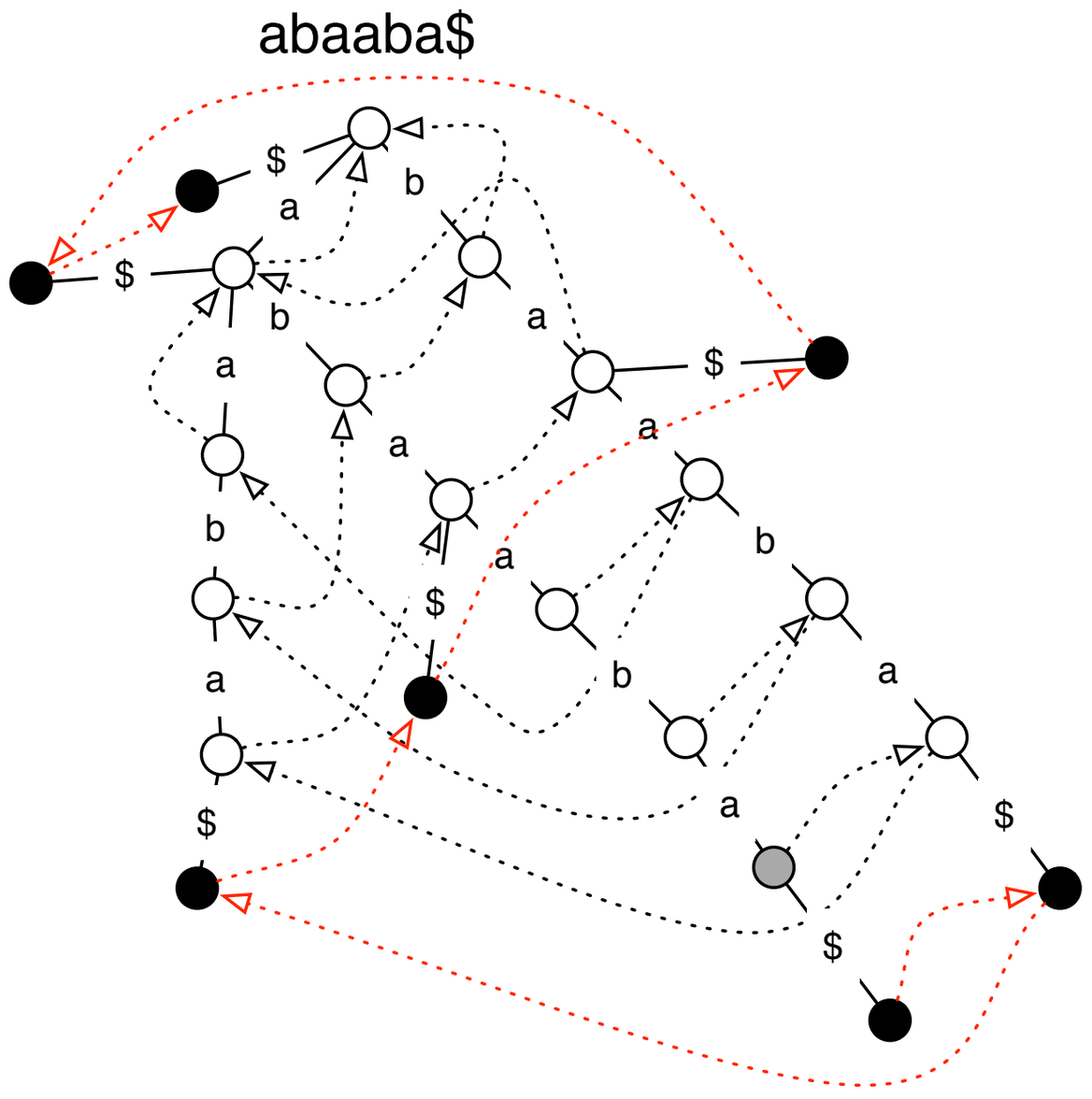


Applications of Suffix Tries (II)

Find the longest common substring of T and q:

Walk down the tree following q.
If you hit a dead end, save the current depth, and follow the suffix link from the current node.
When you exhaust q, return the longest substring found.

T = abaaba$
q = bbaa



Constructing Suffix Tries

Suppose we want to build suffix trie for string:

s = abbacabaa

We will walk down the string from left to right:

abbacabaa

building suffix tries for s[0], s[0..1], s[0..2], ..., s[0..n]

To build suffix trie for s[0..i], we will use the suffix trie for s[0..i-1] built in previous step

To convert SufTrie(S[0..i-1]) → SufTrie(s[0..i]), add character s[i] to all the suffixes:

abbacabaa
i=4

Need to add nodes for the suffixes:

abbac
bbac
bac
ac
c

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?

Suppose we want to build suffix trie for string:

s = abbacabaa

We will walk down the string from left to right:

abbacabaa

building suffix tries for s[0], s[0..1], s[0..2], ..., s[0..n]

To build suffix trie for s[0..i], we will use the suffix trie for s[0..i-1] built in previous step

To convert SufTrie(S[0..i-1]) → SufTrie(s[0..i]), add character s[i] to all the suffixes:

abbacabaa
i=4

Need to add nodes for the suffixes:

abbac
bbac
bac
ac
c

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?

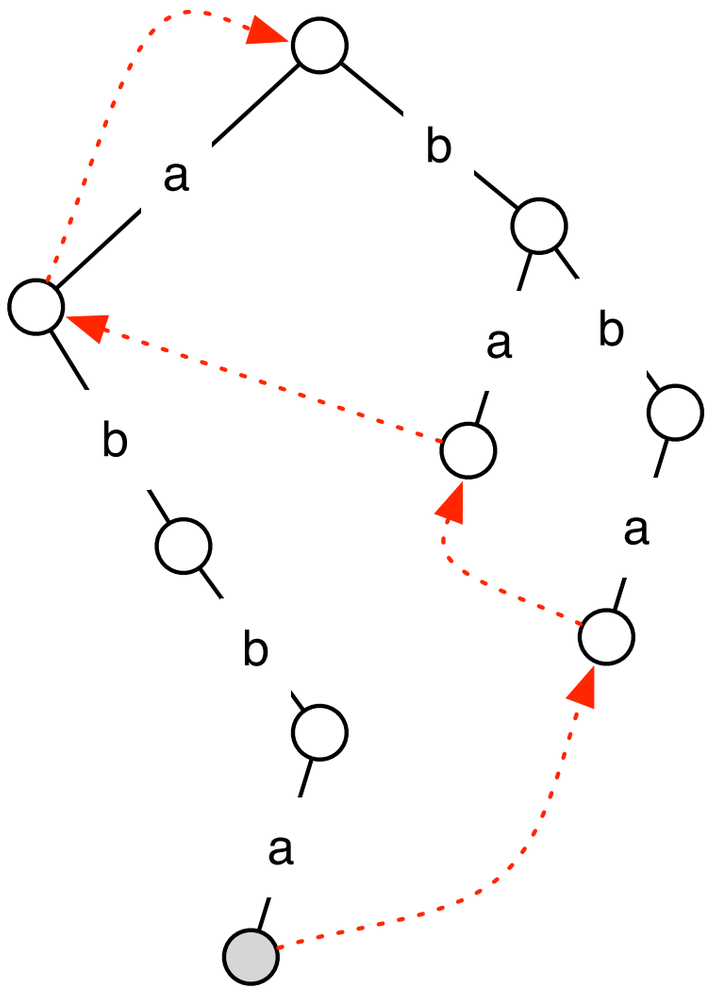
abbacabaa
i=4

Need to add nodes for the suffixes:

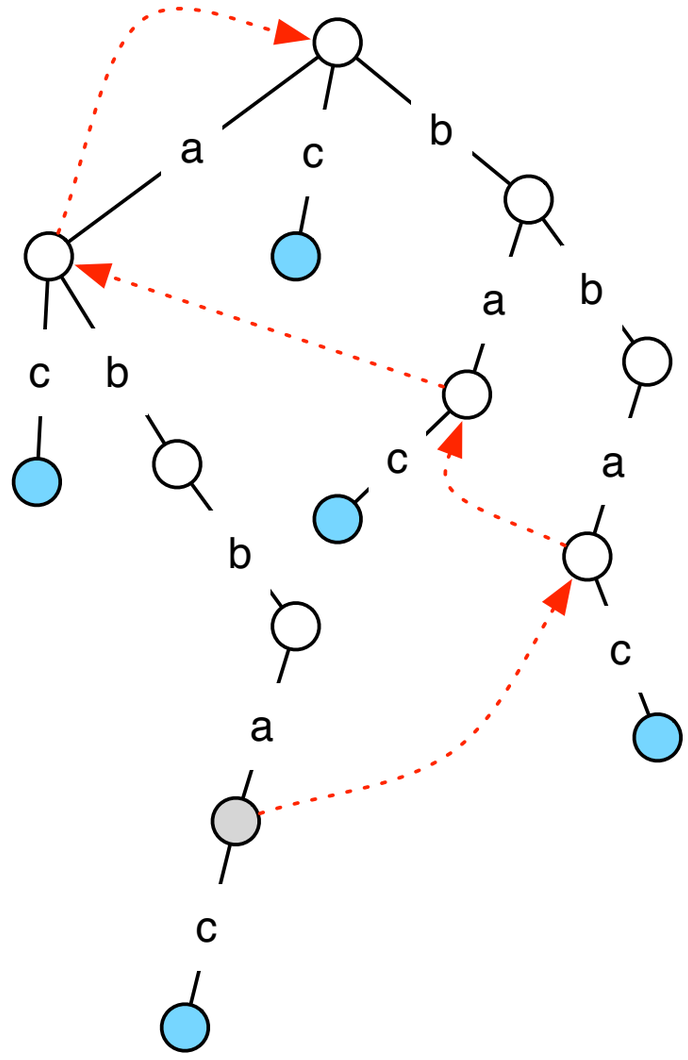
abbac
bbac
bac
ac
c

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?



SufTrie(abba)



SufTrie(abbac)

Where is the new deepest node? (aka longest suffix)

How do we add the suffix links for the new nodes?

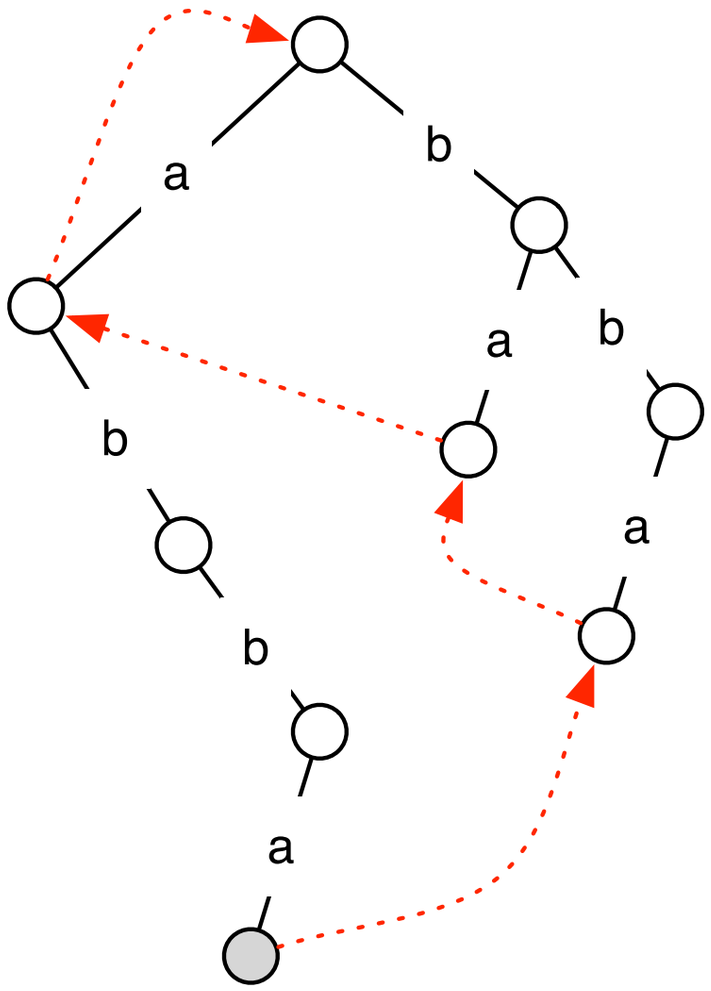
abbacabaa
i=4

Need to add nodes for the suffixes:

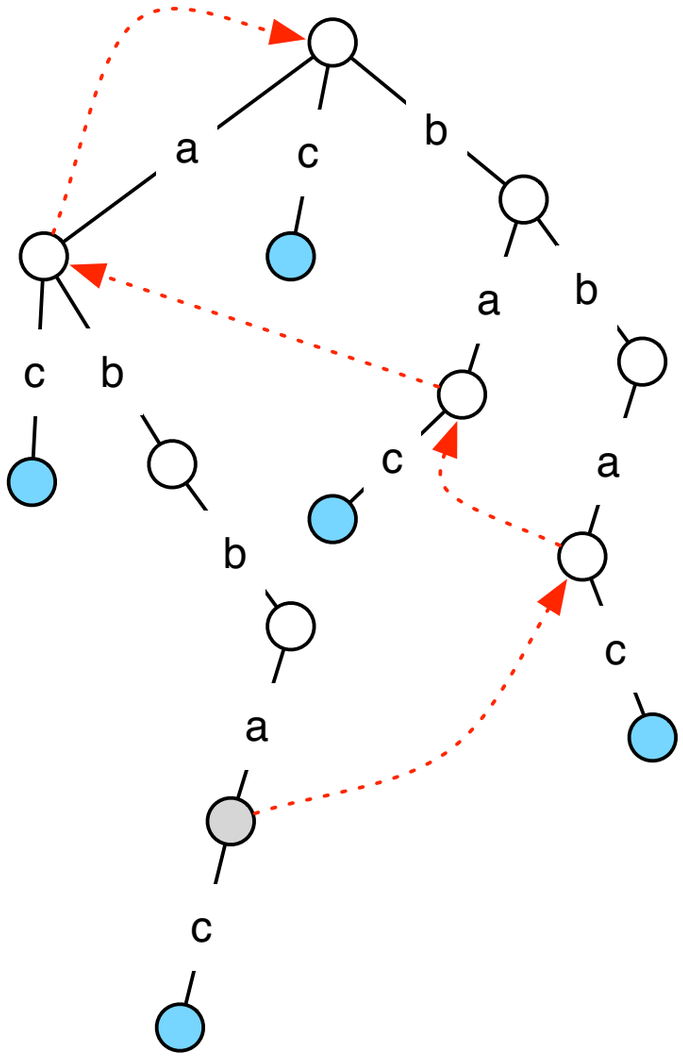
abbac
bbac
bac
ac
c

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?



SufTrie(abba)



SufTrie(abbac)

Where is the new deepest node? (aka longest suffix)

How do we add the suffix links for the new nodes?

To build SufTrie(s[0..i]) from SufTrie(s[0..i-1]):

CurrentSuffix = longest (aka deepest suffix)

Repeat:
Add child labeled s[i] to CurrentSuffix.
Follow suffix link to set CurrentSuffix to next shortest suffix.

until you reach the root or the current node already has an edge labeled s[i] leaving it.

Because if you already have a node for suffix αs[i] then you have a node for every smaller suffix.

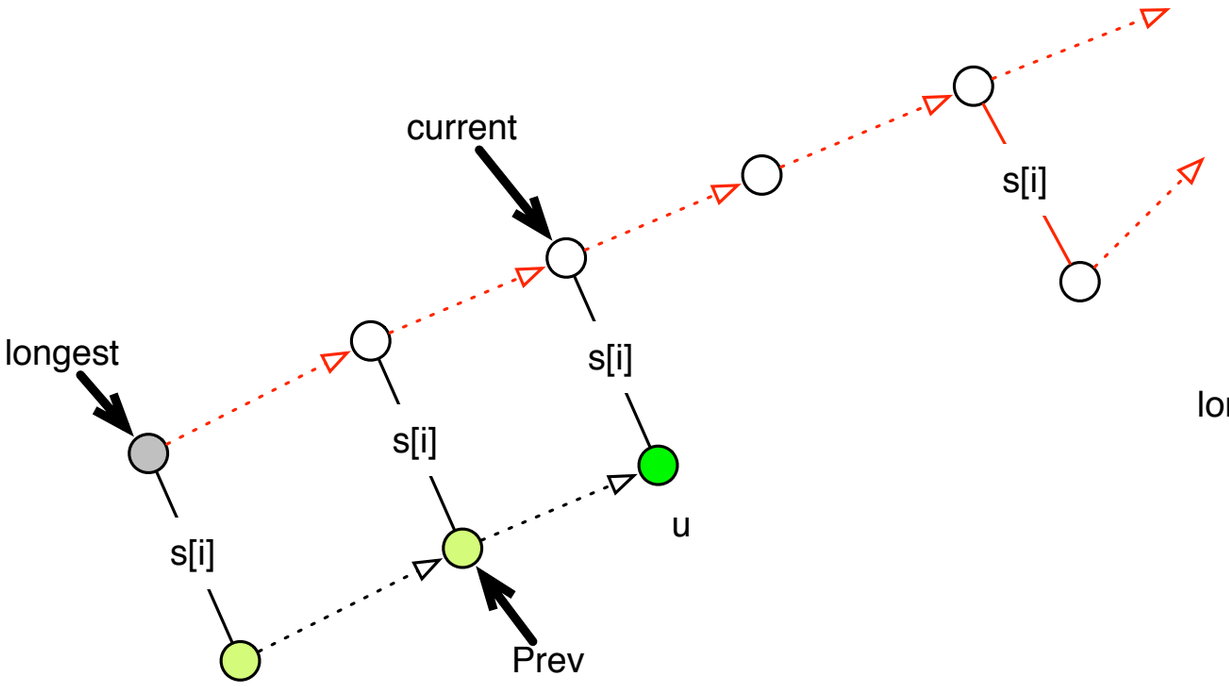
Add suffix links connecting nodes you just added in the order in which you added them.

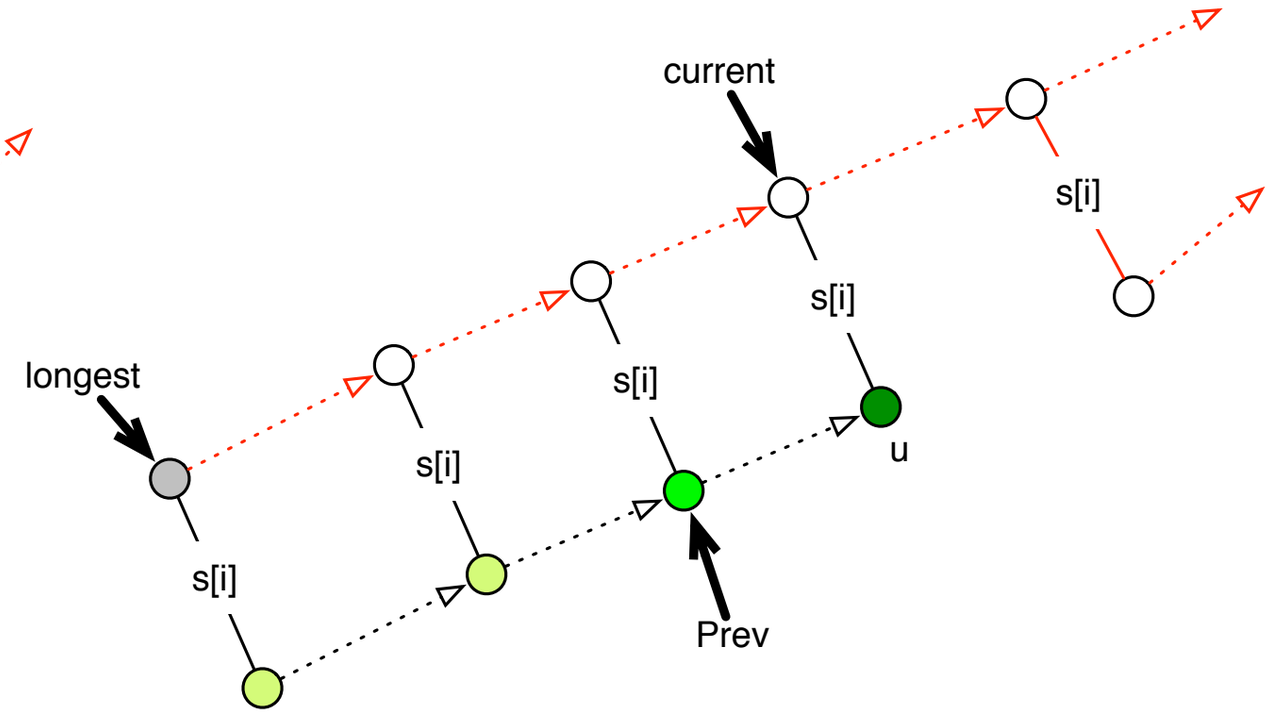
In practice, you add these links as you go along, rather than at the end.

Python Code to Build a Suffix Trie

```
class SuffixNode:
    def __init__(self, suffix_link = None):
        self.children = {}
        if suffix_link is not None:
            self.suffix_link = suffix_link
        else:
            self.suffix_link = self

    def add_link(self, c, v):
        """link this node to node v via string c"""
        self.children[c] = v
```

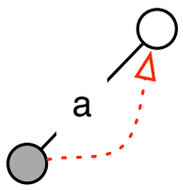
```
def build_suffix_trie(s):
    """Construct a suffix trie."""
    assert len(s) > 0

    # explicitly build the two-node suffix tree
    Root = SuffixNode()      # the root node
    Longest = SuffixNode(suffix_link = Root)
    Root.add_link(s[0], Longest)

    # for every character left in the string
    for c in s[1:]:
        Current = Longest; Previous = None
        while c not in Current.children:

            # create new node r1 with transition Current -c->r1
            r1 = SuffixNode()
            Current.add_link(c, r1)

            # if we came from some previous node, make that
            # node's suffix link point here
            if Previous is not None:
                Previous.suffix_link = r1

            # walk down the suffix links
            Previous = r1
            Current = Current.suffix_link

        # make the last suffix link
        if Current is Root:
            Previous.suffix_link = Root
        else:
            Previous.suffix_link = Current.children[c]

        # move to the newly added child of the longest path
        # (which is the new longest path)
        Longest = Longest.children[c]
    return Root
```

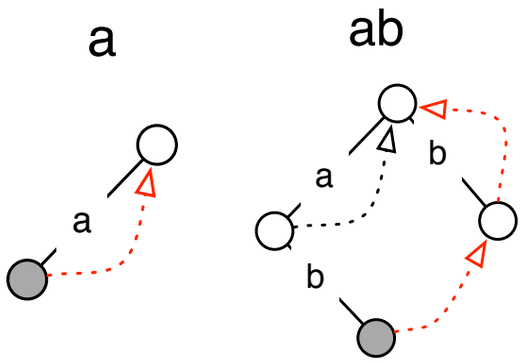

current
longest
s[i]
s[i]
s[i]
s[i]
u
Prev

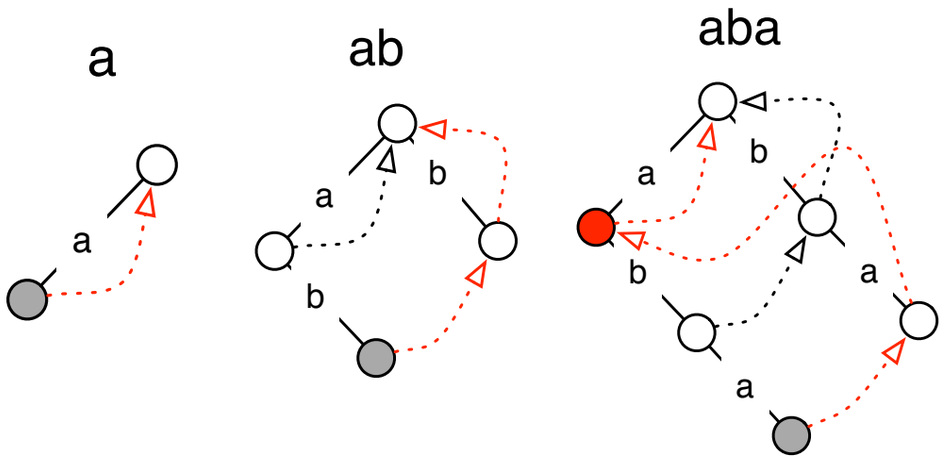

current
longest
s[i]
s[i]
s[i]
s[i]
u
Prev

current
boundary path
s[i]
s[i]
s[i]
s[i]
s[i]
Prev
longest

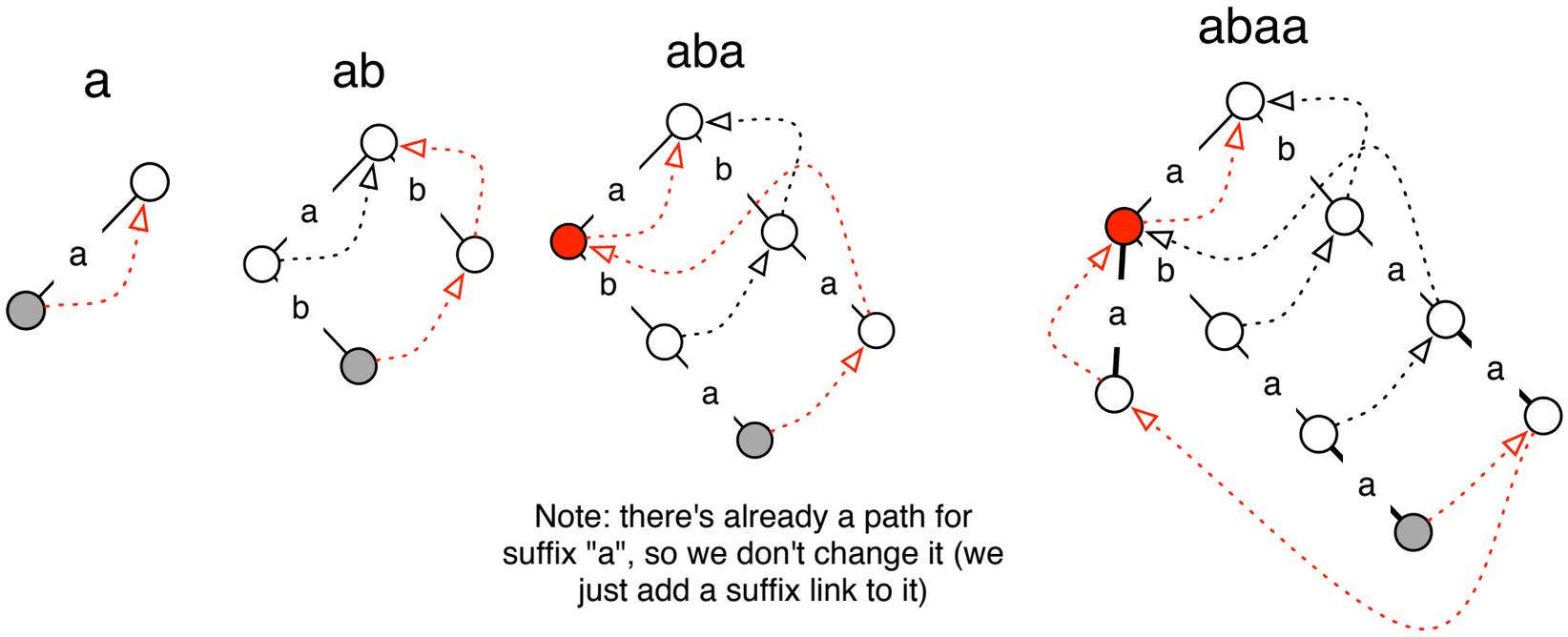
a



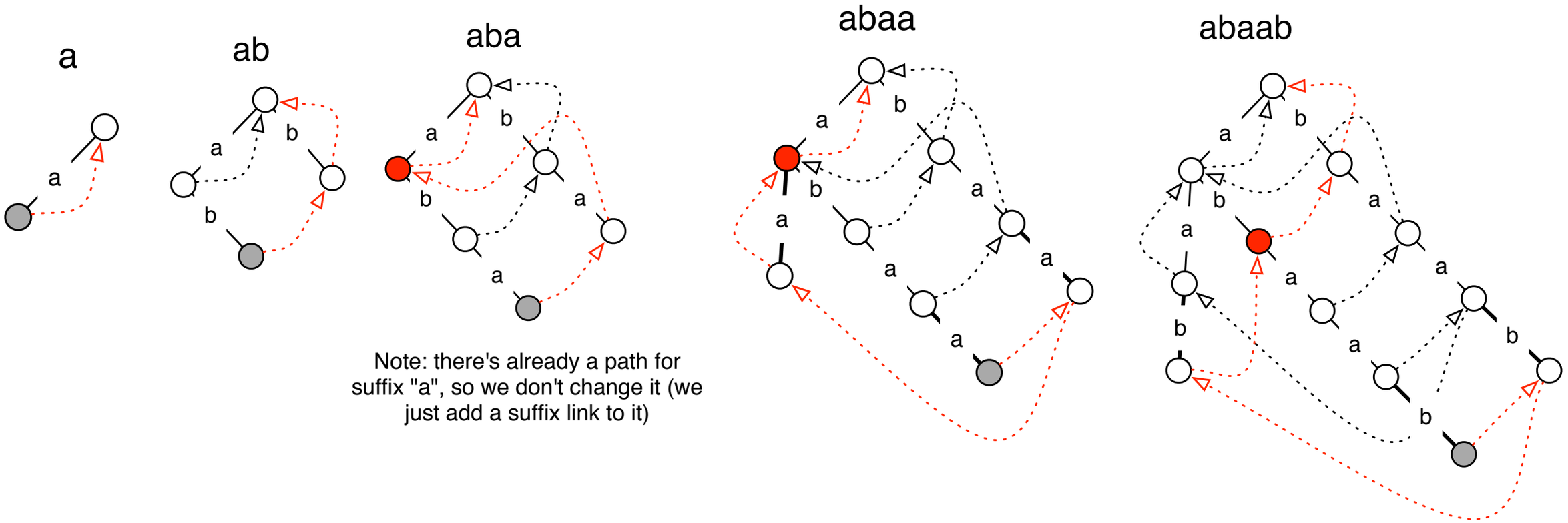

a
a
ab
a
b
b



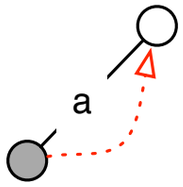
Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)



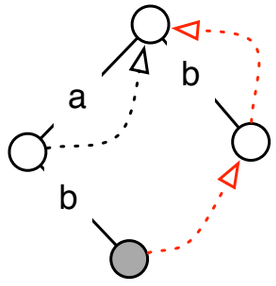
Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)


a
ab
aba
abaa
abaab
Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)

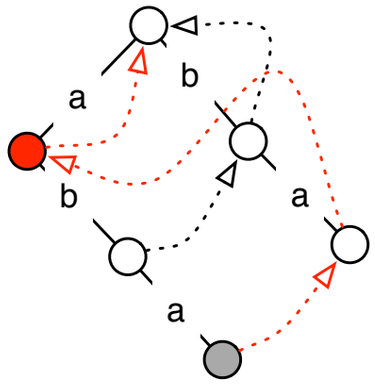
a



ab

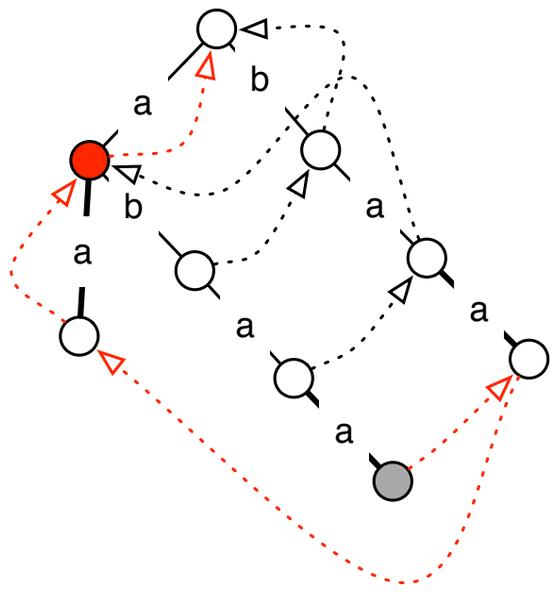


aba

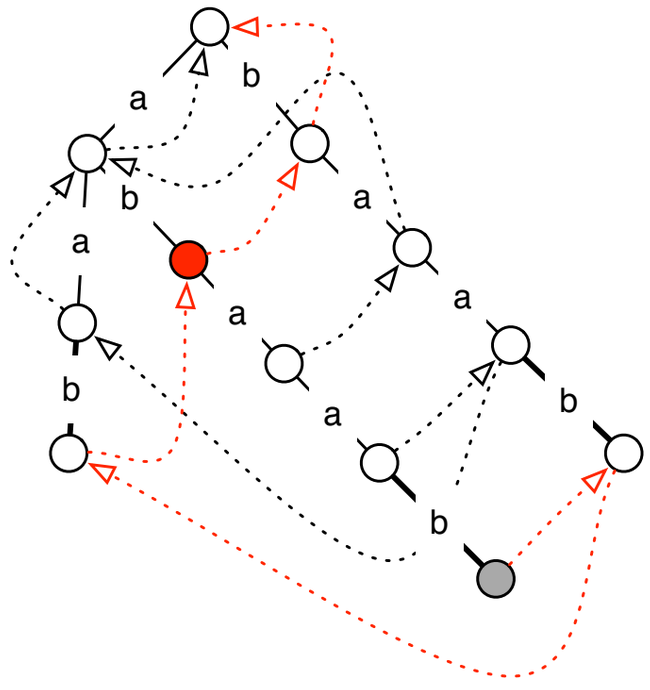


Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)

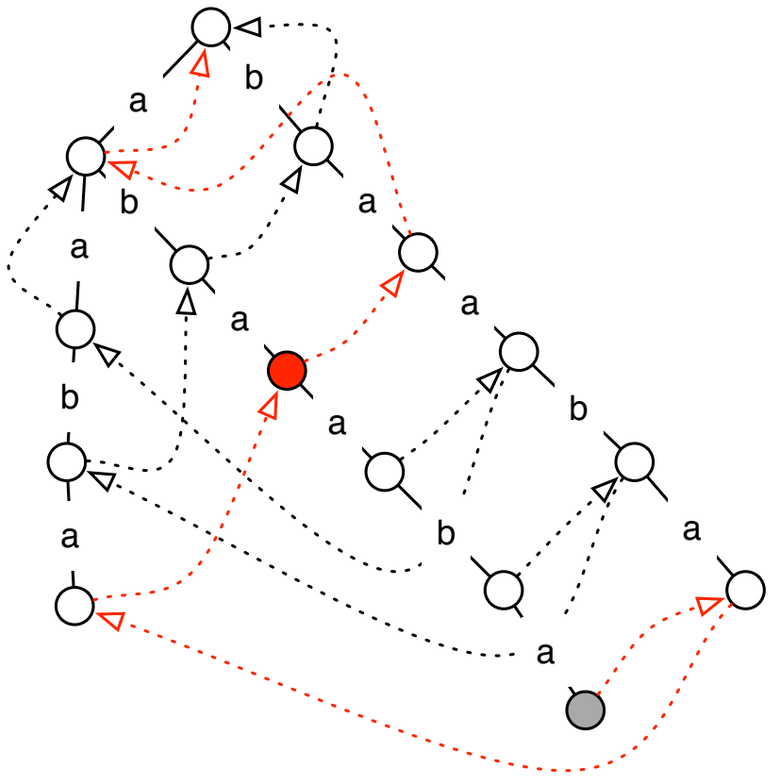
abaa



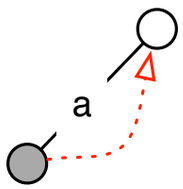
abaab



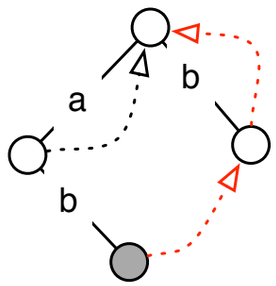
abaaba



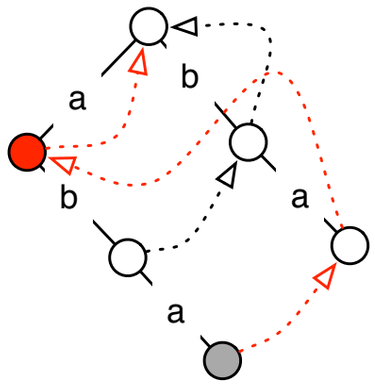
a



ab

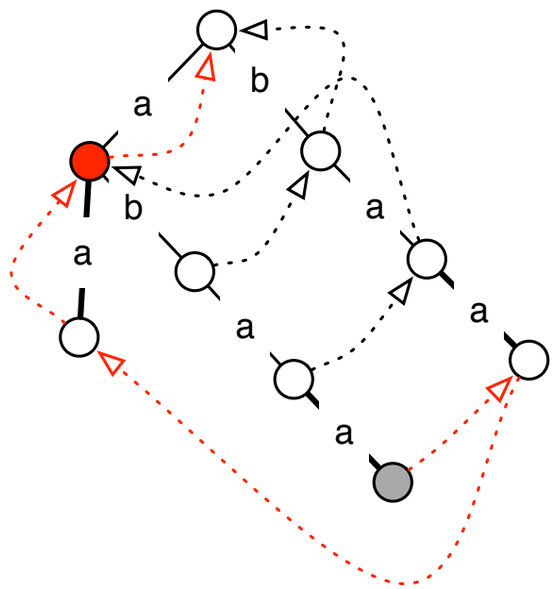


aba

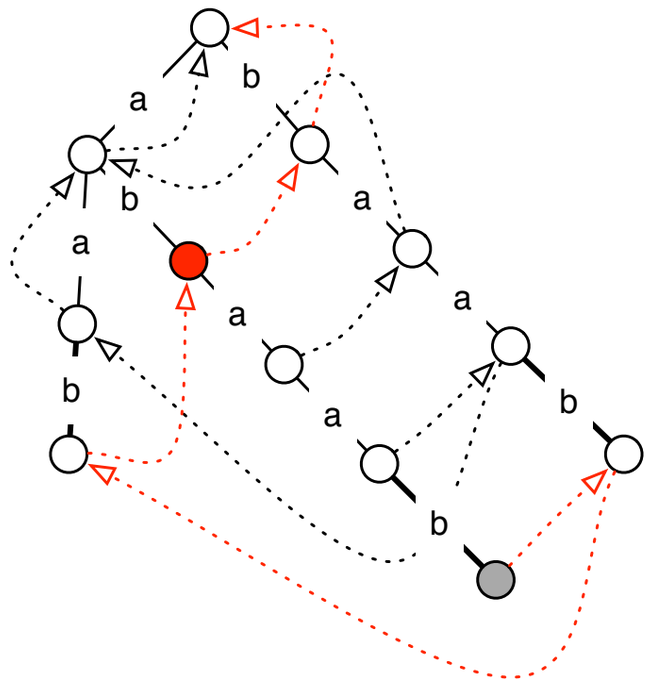


Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)

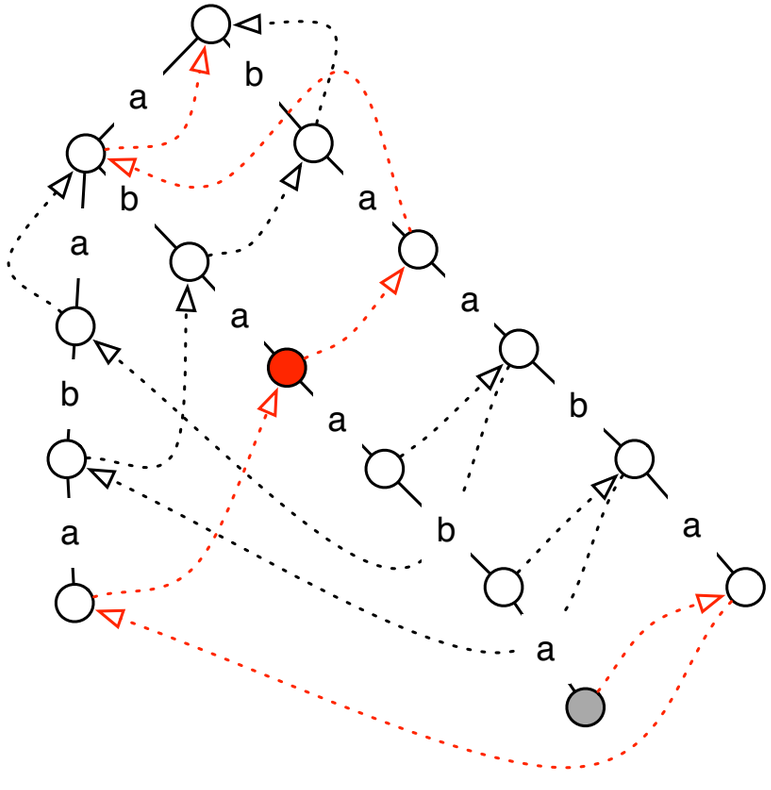
abaa



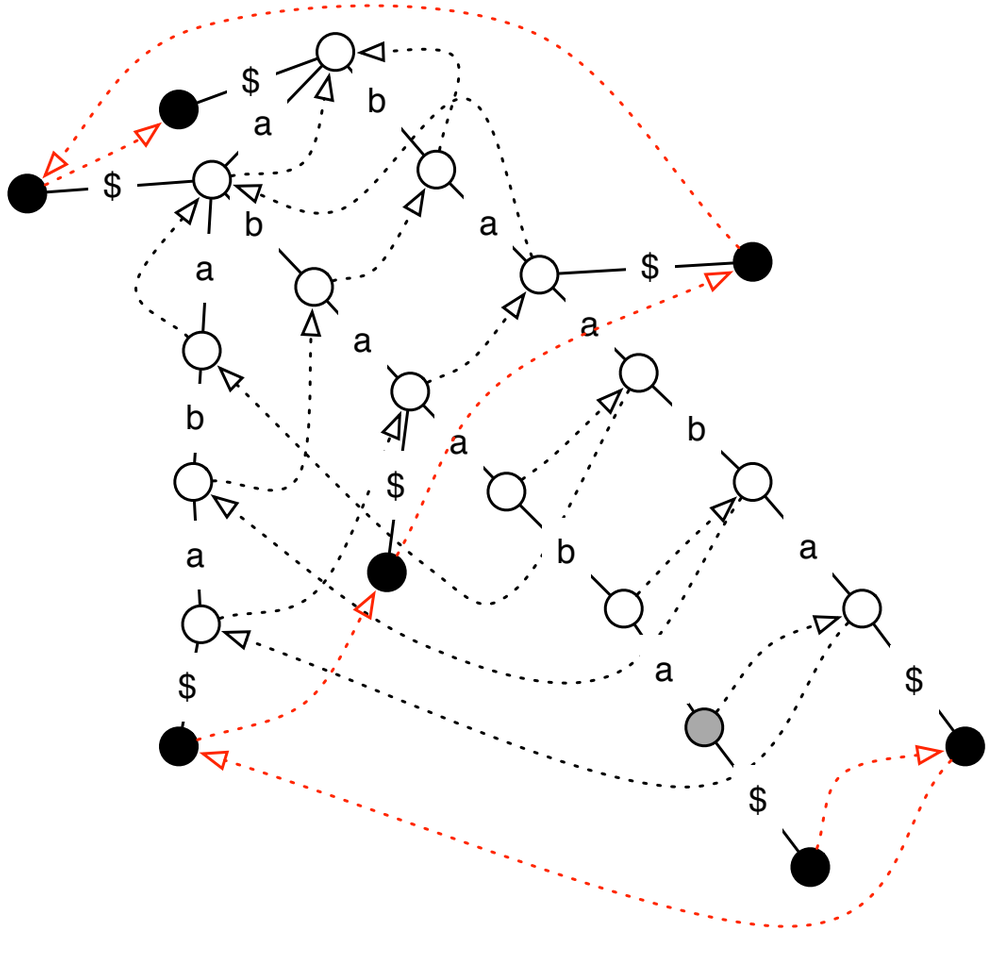
abaab



abaaba

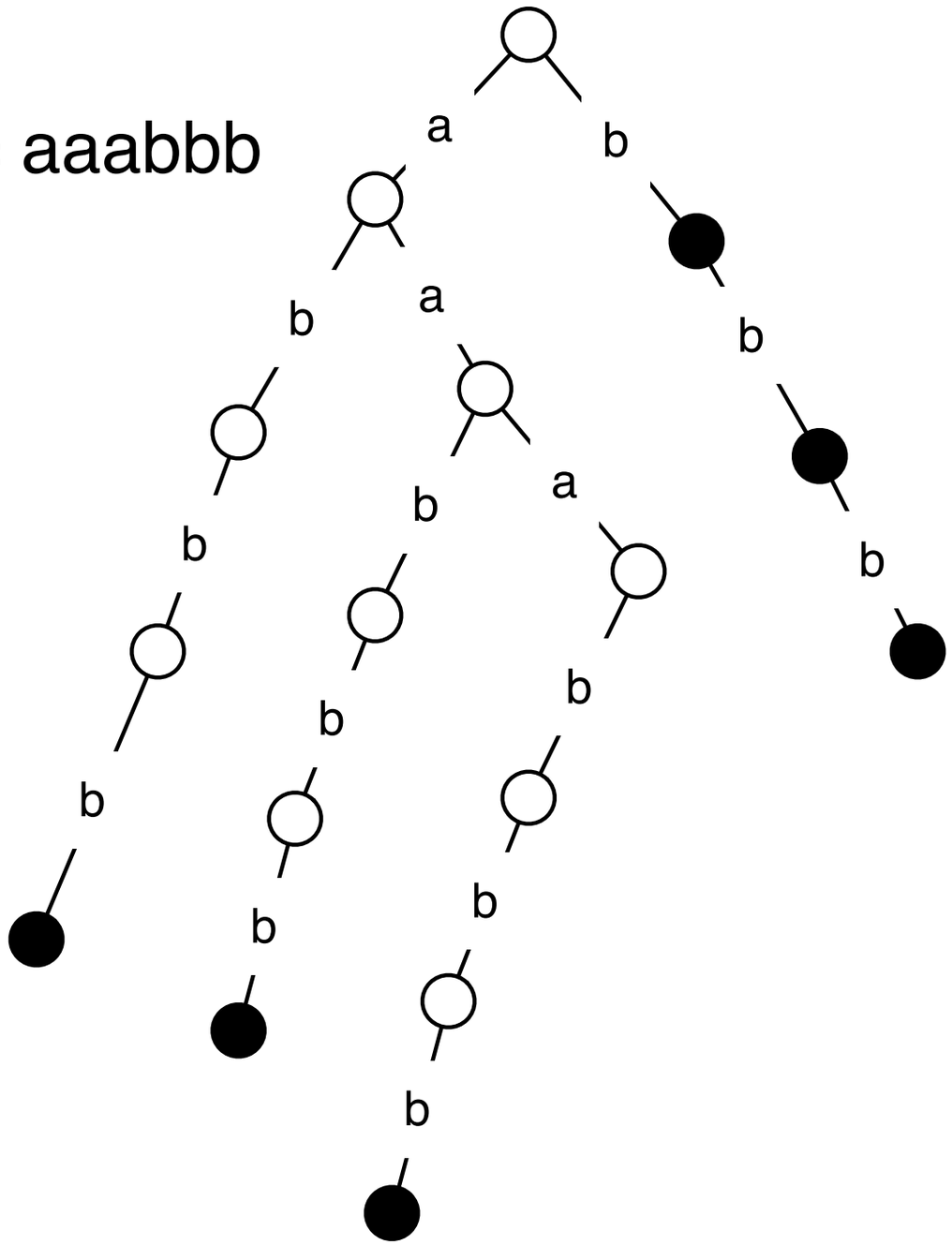


abaaba$



How many nodes can a suffix trie have?

s = aaabbb



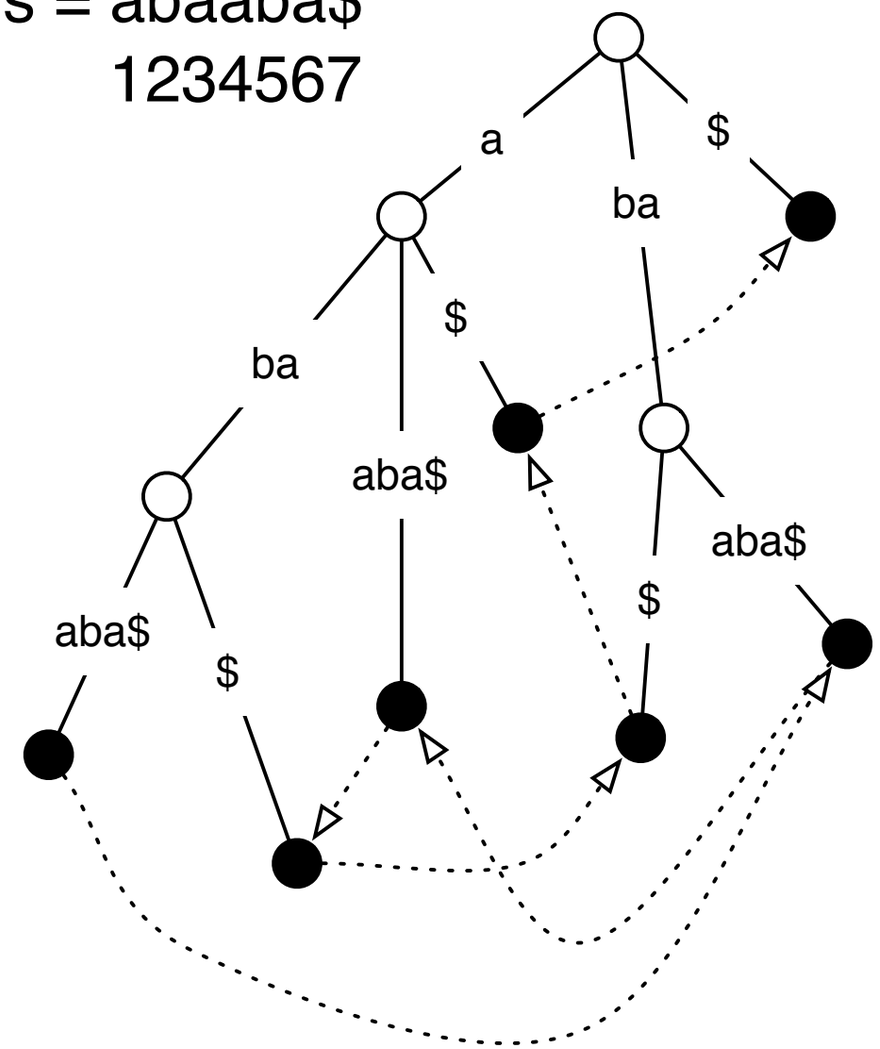
- $s = a^n b^n$ will have
  - 1 root node
  - n nodes in a path of "b"s
  - n paths of n+1 "b" nodes
- Total = $n(n+1)+n+1 = O(n^2)$ nodes.
- This is not very efficient.
- How could you make it smaller?

So... we have to “trie” again...

Space-Efficient Suffix Trees

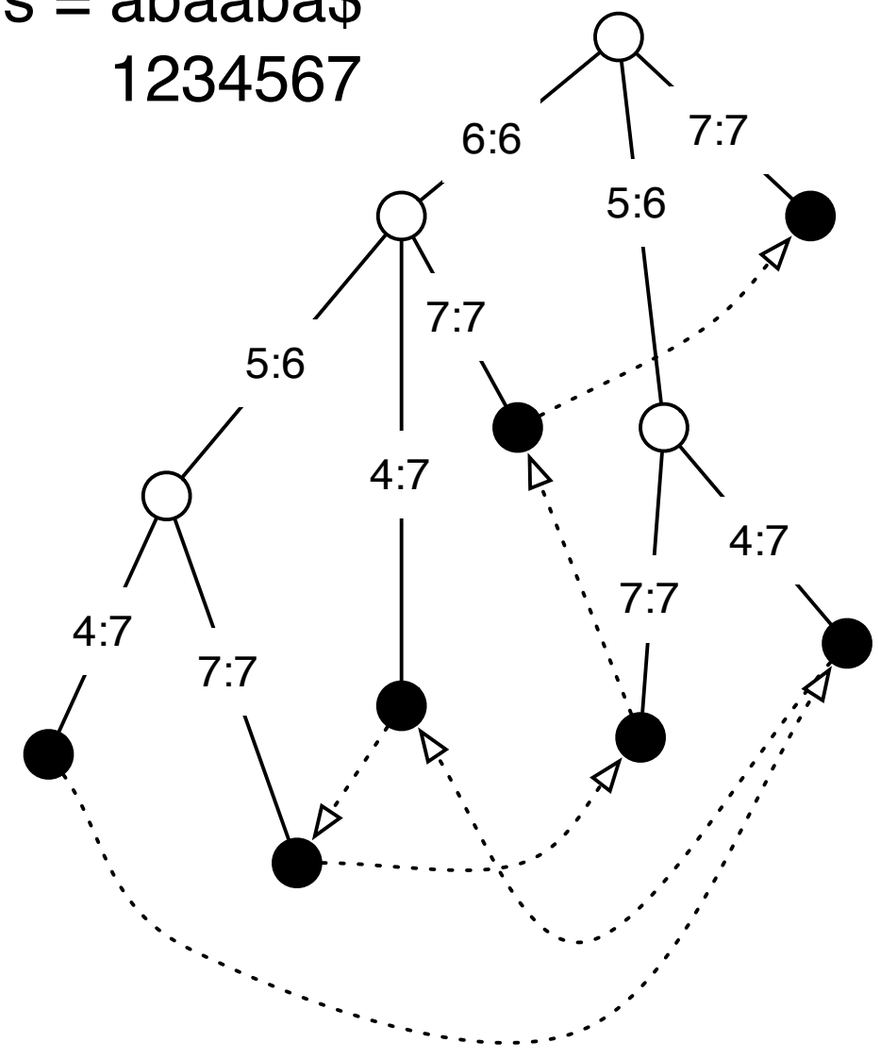
A More Compact Representation

s = abaaba$
1234567



- Compress paths where there are no choices.

s = abaaba$
1234567



- Represent sequence along the path using a range [i,j] that refers to the input string s.

Space usage:

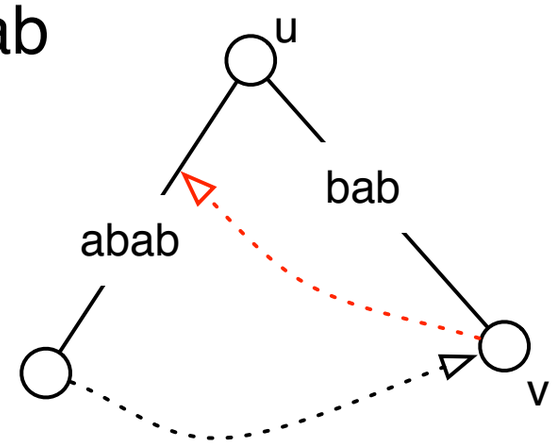
- In the compressed representation:
  - # leaves = O(n) [one leaf for each position in the string]
  - Every internal node is at least a binary split.
  - Each edge uses O(1) space.

- Therefore, # number of internal nodes is about equal to the number of leaves.
- And # of edges ≈ number of leaves, and space per edge is O(1).
- Hence, linear space.

Constructing Suffix Trees - Ukkonen's Algorithm

- The same idea as with the suffix trie algorithm.
- Main difference: not every trie node is explicitly represented in the tree.
- Solution: represent trie nodes as pairs (u, α), where u is a real node in the tree and α is some string leaving it.
- Some additional tricks to get to O(n) time.

s = abab



suffix_link[v] = (u, ab)

Storing more than one string with Generalized Suffix Trees

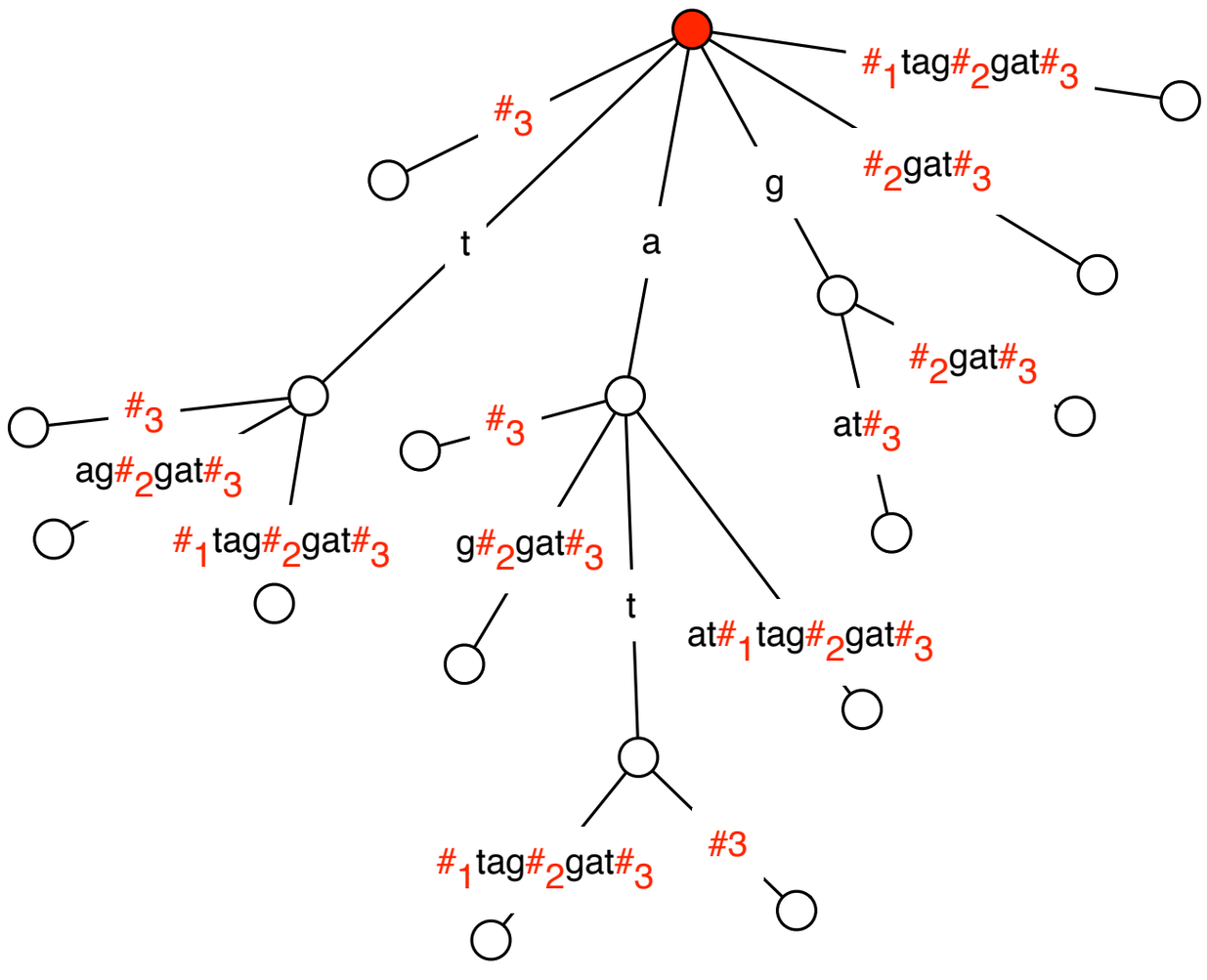
Constructing Generalized Suffix Trees

Goal. Represent a set of strings P = {s_1, s_2, s_3, ..., s_m}.

Example. att, tag, gat

Simple solution:

(1) build suffix tree for string aat#$_1$tag#$_2$gat#$_3$



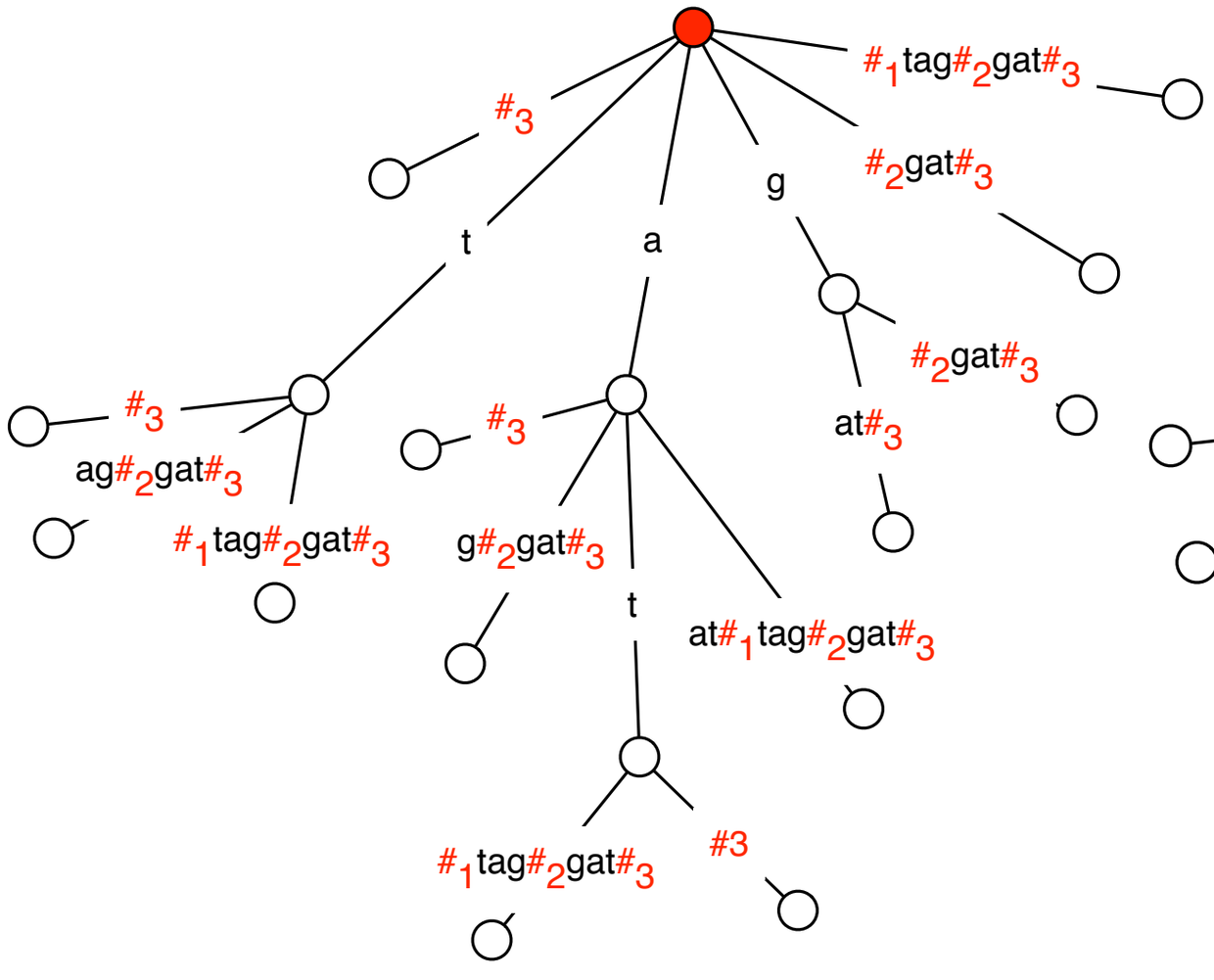
Constructing Generalized Suffix Trees

Goal. Represent a set of strings P = {s_1, s_2, s_3, ..., s_m}.

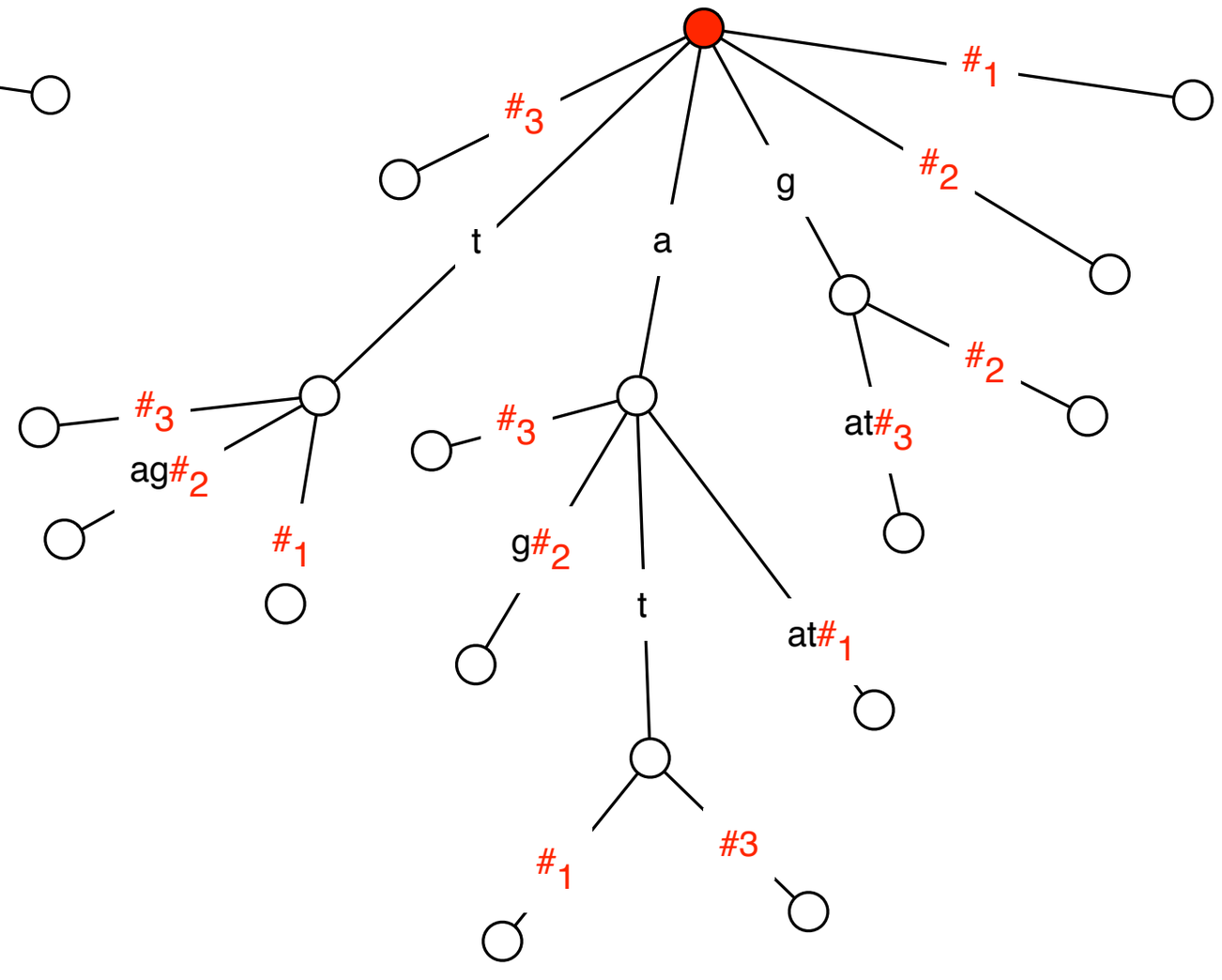
Example. att, tag, gat

Simple solution:

(1) build suffix tree for string aat#$_1$tag#$_2$gat#$_3$



(2) For every leaf node, remove any text after the first # symbol.



Applications of Generalized Suffix Trees

Longest common substring of S and T:

Determine the strings in a database $\{S_1, S_2, S_3, ..., S_m\}$ that contain query string q:

Applications of Generalized Suffix Trees

Longest common substring of S and T:

Build generalized suffix tree for {S, T}
Find the deepest node that has has descendants from both strings (containing both $\#_1$ and $\#_2$)

Determine the strings in a database $\{S_1, S_2, S_3, ..., S_m\}$ that contain query string q:

Applications of Generalized Suffix Trees

Longest common substring of S and T:

Build generalized suffix tree for {S, T}
Find the deepest node that has has descendants from both strings (containing both $\#_1$ and $\#_2$)

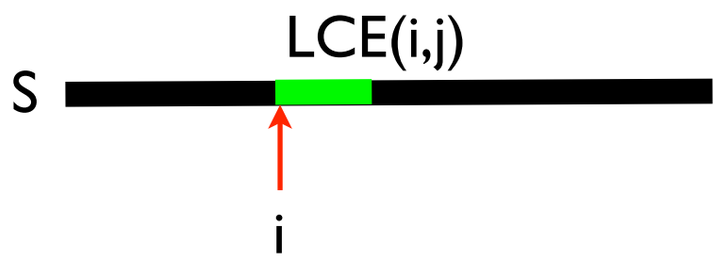
Determine the strings in a database $\{S_1, S_2, S_3, ..., S_m\}$ that contain query string q:

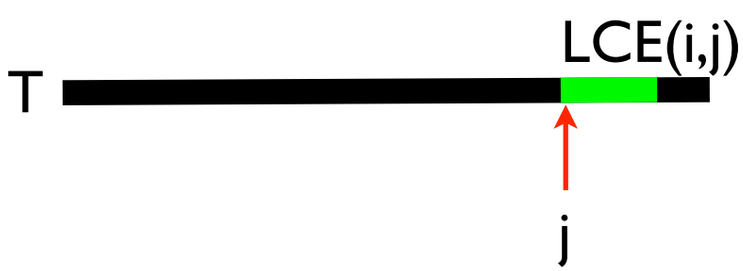
Build generalized suffix tree for $\{S_1, S_2, S_3, ..., S_m\}$
Follow the path for q in the suffix tree.
Suppose you end at node u: traverse the tree below u, and output i if you find a string containing $\#_i$.

Longest Common Extension

Longest common extension: We are given strings S and T. In the future, many pairs (i,j) will be provided as queries, and we want to quickly find:

the longest substring of S starting at i that matches a substring of T starting at j.





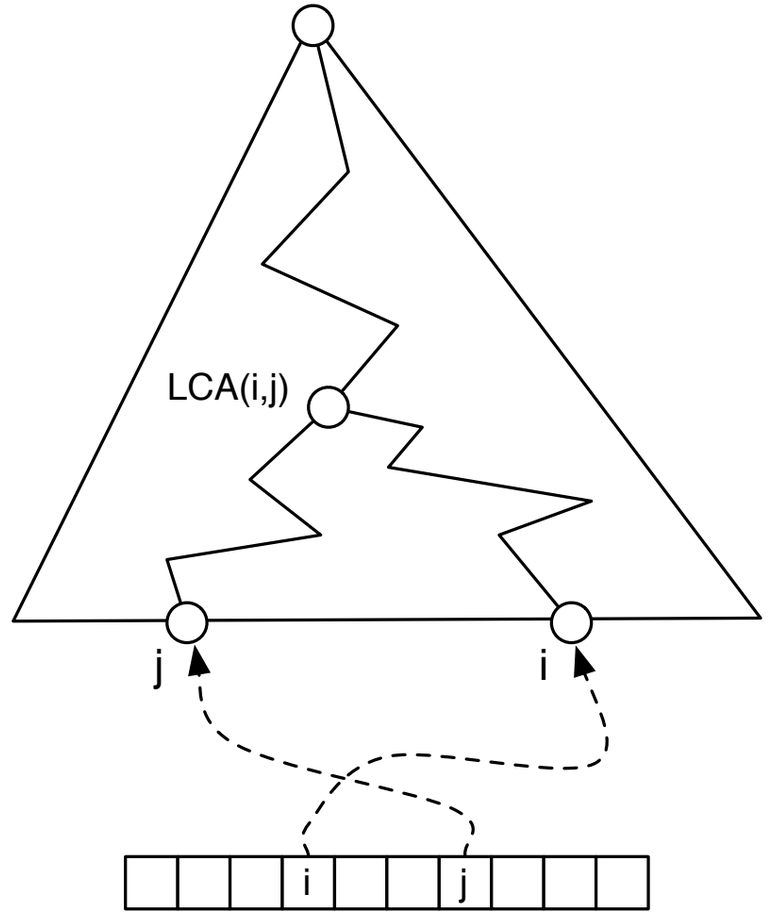
Build generalized suffix tree for S and T.

Preprocess tree so that lowest common ancestors (LCA) can be found in constant time.

Create an array mapping suffix numbers to leaf nodes.

Given query (i,j):

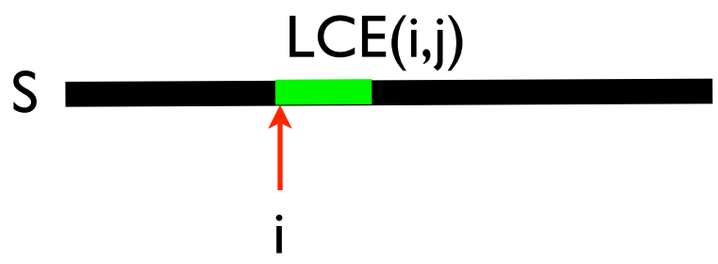
- Find the leaf nodes for i and j
- Return string of LCA for i and j

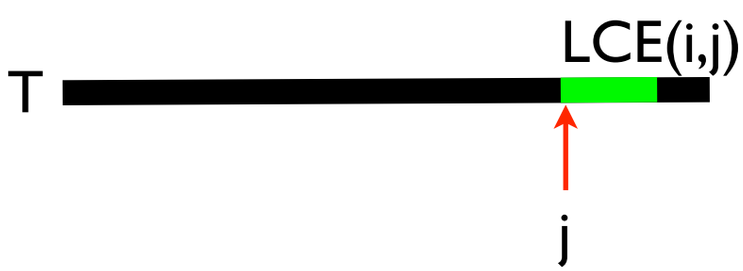


Longest Common Extension

Longest common extension: We are given strings S and T. In the future, many pairs (i,j) will be provided as queries, and we want to quickly find:

the longest substring of S starting at i that matches a substring of T starting at j.





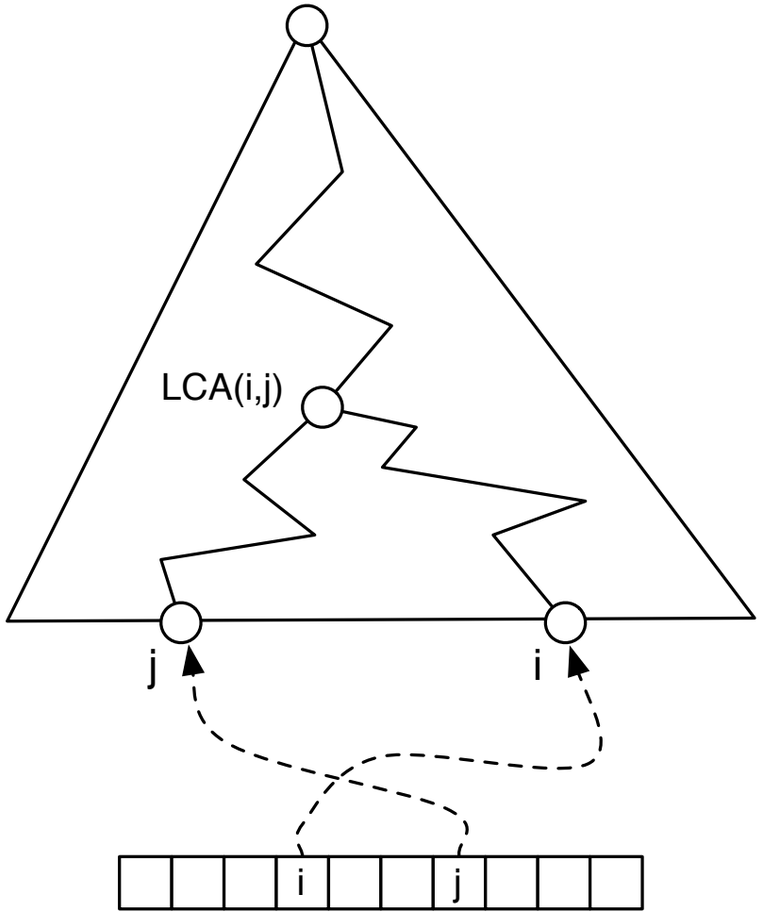
Build generalized suffix tree for S and T. O(|S| + |T|)

Preprocess tree so that lowest common ancestors (LCA) can be found in constant time. O(|S| + |T|)

Create an array mapping suffix numbers to leaf nodes. O(|S| + |T|)

Given query (i,j):

- Find the leaf nodes for i and j O(1)
- Return string of LCA for i and j O(1)



Using LCE to Find Palindromes

Maximal even palindrome at position i: the longest string to the left and right so that the left half is equal to the reverse of the right half.

Goal: find all maximal palindromes in S.

Construct S^r, the reverse of S.

Preprocess S and S^r so that LCE queries can be solved in constant time (previous slide).

LCE(i, n-i) is the length of the longest palindrome centered at i.

For every position i:
    Compute LCE(i, n-i)

Using LCE to Find Palindromes

Maximal even palindrome at position i: the longest string to the left and right so that the left half is equal to the reverse of the right half.

Goal: find all maximal palindromes in S.

Construct S^r, the reverse of S. O(|S|)

Preprocess S and S^r so that LCE queries can be solved in constant time (previous slide). O(|S|)

LCE(i, n-i) is the length of the longest palindrome centered at i.

For every position i: O(|S|)
    Compute LCE(i, n-i) O(1)

Total time = O(|S|)

Recap

- Suffix tries natural way to store a string -- search, count occurrences, and many other queries answerable easily.
- But they are not space efficient: $O(n^2)$ space.
- Suffix trees are space optimal: O(n), but require a little more subtle algorithm to construct.
- Suffix trees can be constructed in O(n) time using Ukkonen's algorithm.
- Similar ideas can be used to store sets of strings.

Algorithms and data structures for read mapping

Based on
Lecture notes from Ron Shamir (Tel aviv Univ.)
And
Carl Kingsford (CMU)

Read mapping problem

- Input:
  - Reference genome R (human: $3*10^9$ bp)
  - Set of reads $S_1, S_2, \ldots, S_m$ ($m = 10^9$, $|S_i| = 100$)
- Output
  - For each read S_i, the position in R that matches S_i (possibly allowing for a small number of mismatches (SNPs, errors))

Solutions

- Naïve :
  - For each S_i
    - For each position $p=l,\ldots,|R|$
      - Try matching S_i to the substring $R[p-l+1,\ldots,p]$

- Complexity:
  $O(lm|R|)$ exact or inexact matching

Solutions (2)

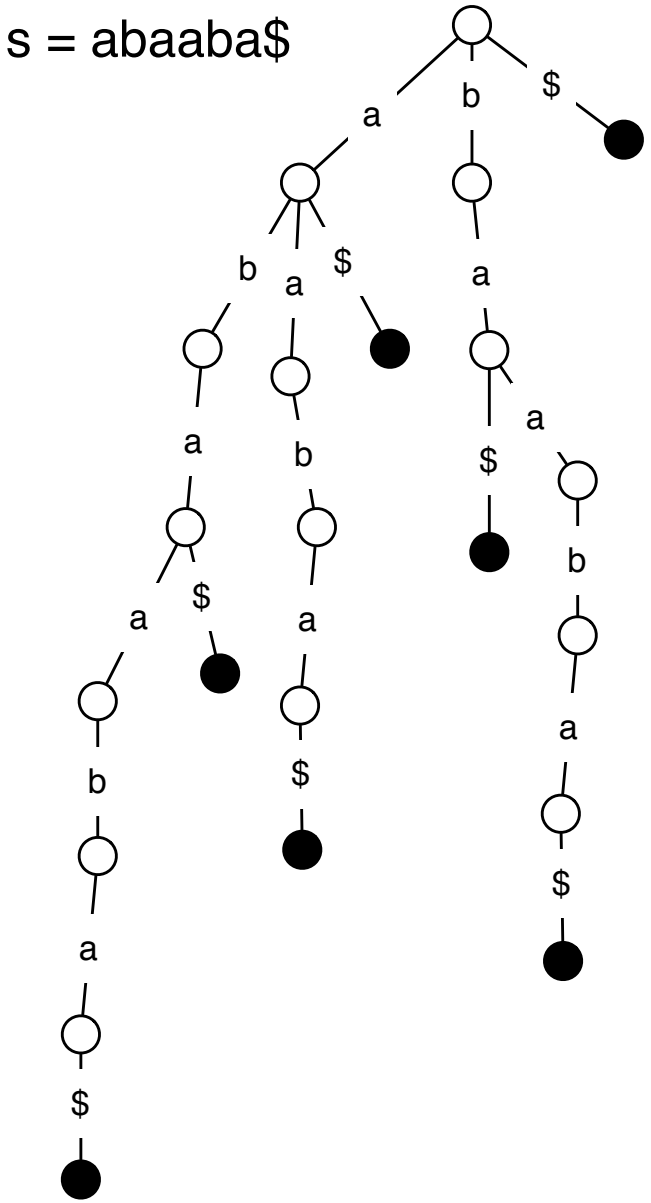
- Less Naïve:
  - For each S_i
    - Match S_i to R using KMP [Knuth-Morris-Pratt]
- Complexity:

  $O(m(l+|R|))= O(ml+m|R|)$ exact matching

Solutions (3)

- Suffix tree approach:
  - Build suffix tree for R
  - For each S_i
    - Find matches of S_i to R by tree traversal from the root
- Time complexity: $O(lm+|R|)$ exact matching
- Space Complexity: $O(|R|\log|R|)$ vs $|R|\log|\Sigma|$ for the text
- Can store Human Genome text in 750M bytes (6G bits) but, need ~64G bytes for the tree
  - large constants, hard to implement

Suffix Tries



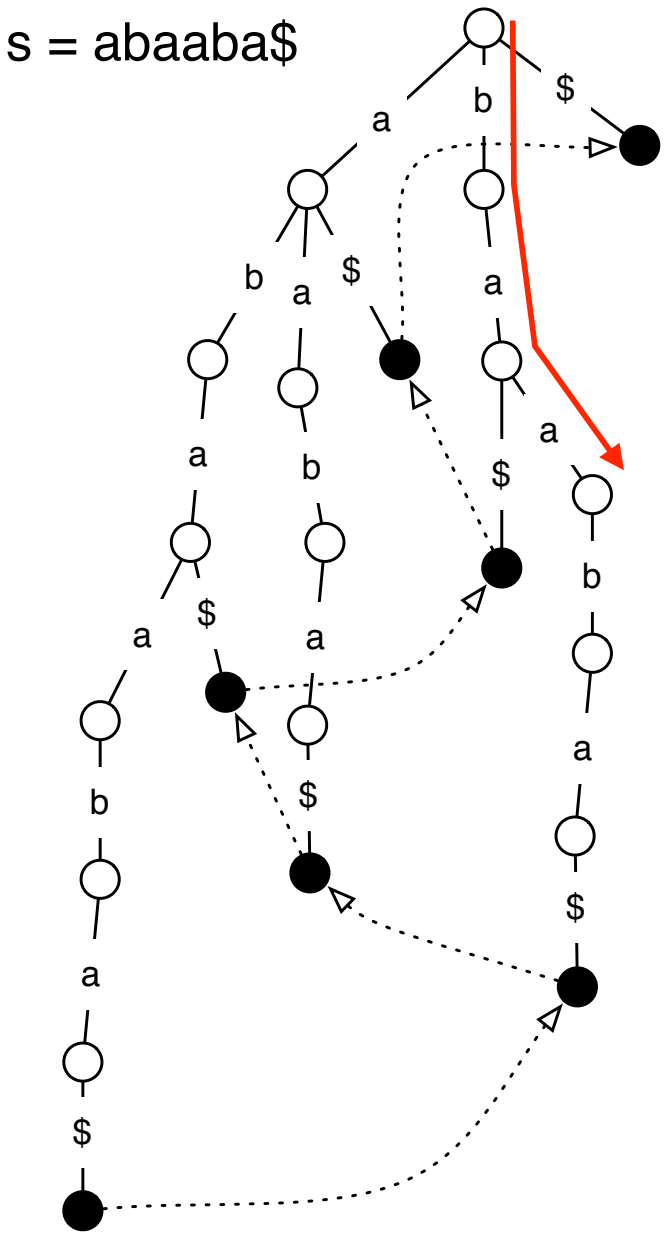
SufTrie(s) = suffix trie representing string s.

Edges of the suffix trie are labeled with letters from the alphabet ∑ (say {A,C,G,T}).

Every path from the root to a solid node represents a suffix of s.

Every suffix of s is represented by some path from the root to a solid node.

How many leaves will there be?



Searching Suffix Tries

Is "baa" a substring of s?

Follow the path given by the query string.

After we've built the suffix trees, queries can be answered in time:

O(|query|)

regardless of the text size.

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the number of occurrences of q.

Find the longest repeat in T:

Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:

Start at the root, and follow the edge labeled with the lexicographically (alphabetically) smallest letter.

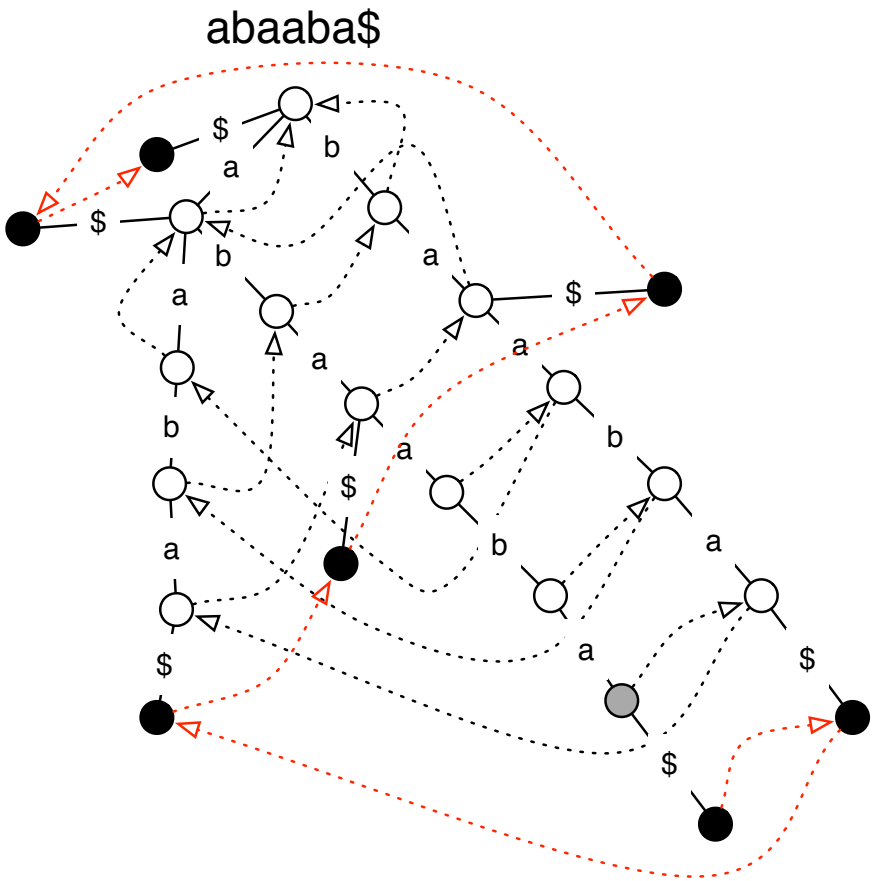
Applications of Suffix Tries (II)

Find the longest common substring of T and q:

Walk down the tree following q.
If you hit a dead end, save the current depth, and follow the suffix link from the current node.
When you exhaust q, return the longest substring found.

T = abaaba$
q = bbaa

q = abbaaa



Building a suffix trie

Suppose we want to build suffix trie for string:

s = abbacabaa

We will walk down the string from left to right:

abbacabaa

building suffix tries for s[0], s[0..1], s[0..2], ..., s[0..n]

To build suffix trie for s[0..i], we will use the suffix trie for s[0..i-1] built in previous step

To convert SufTrie(S[0..i-1]) → SufTrie(s[0..i]), add character s[i] to all the suffixes:

abbacabaa
i=4

Need to add nodes for the suffixes:

abbac
bbac
bac
ac
c

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?

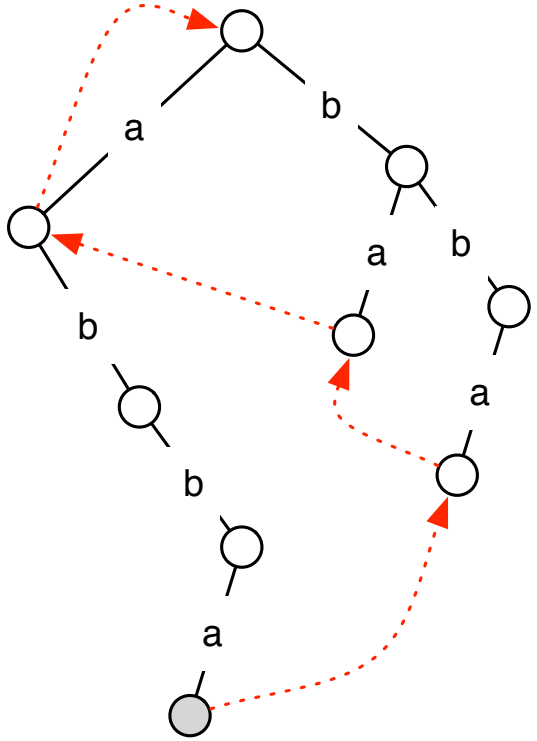
abbacabaa
i=4

Need to add nodes for the suffixes:

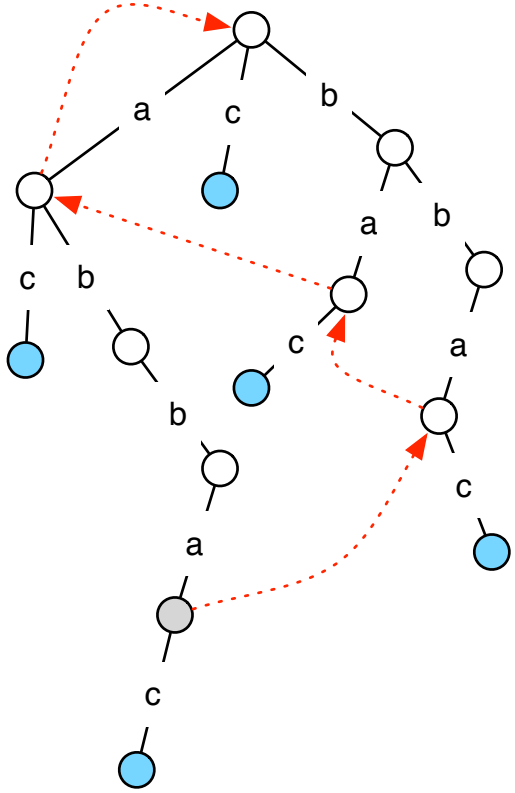
abbac
bbac
bac
ac
c

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?



SufTrie(abba)

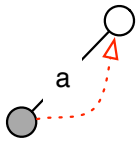


SufTrie(abbac)

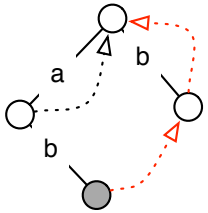
Where is the new deepest node? (aka longest suffix)

How do we add the suffix links for the new nodes?

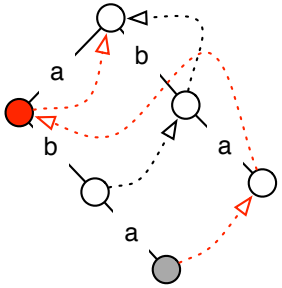
a



ab

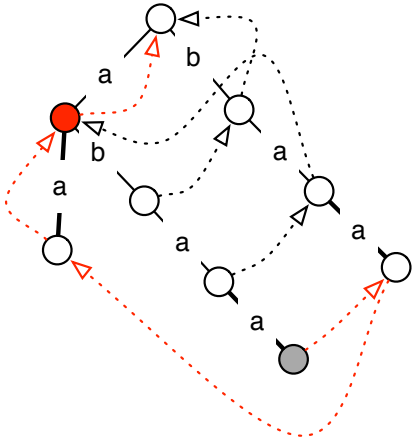


aba

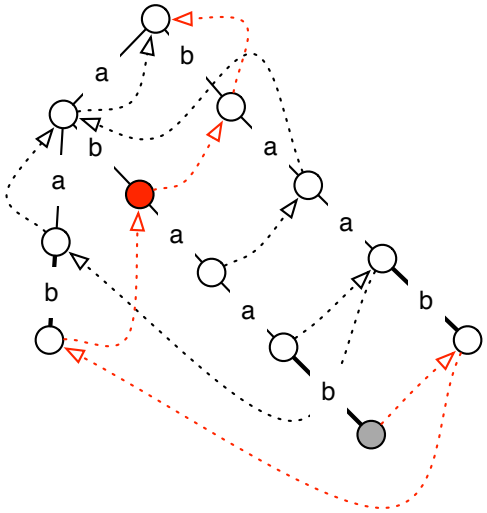


Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)

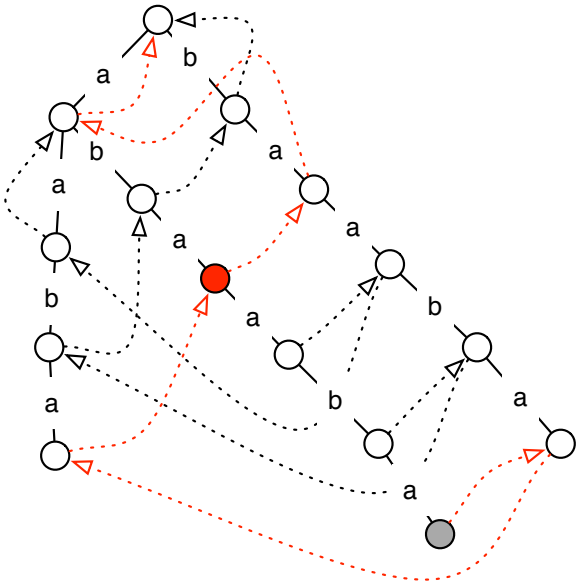
abaa



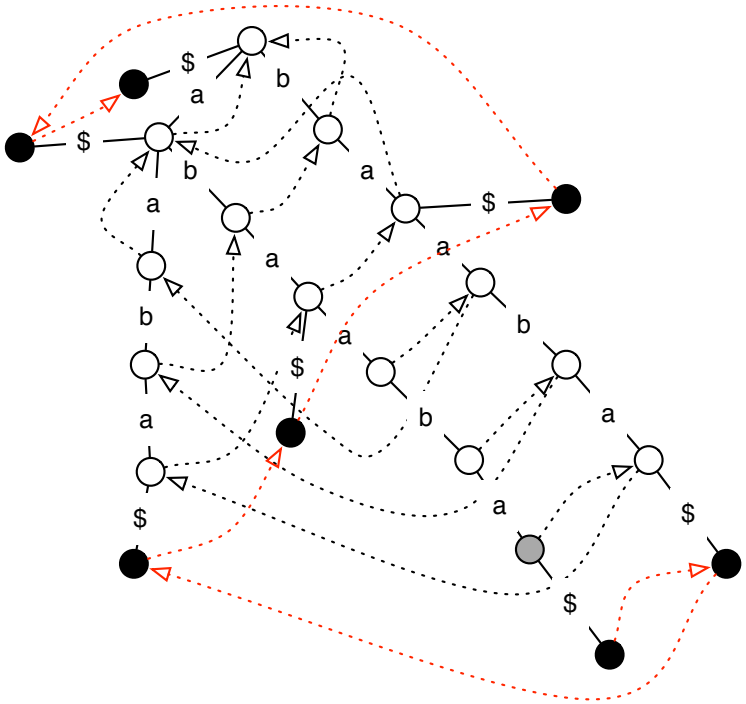
abaab



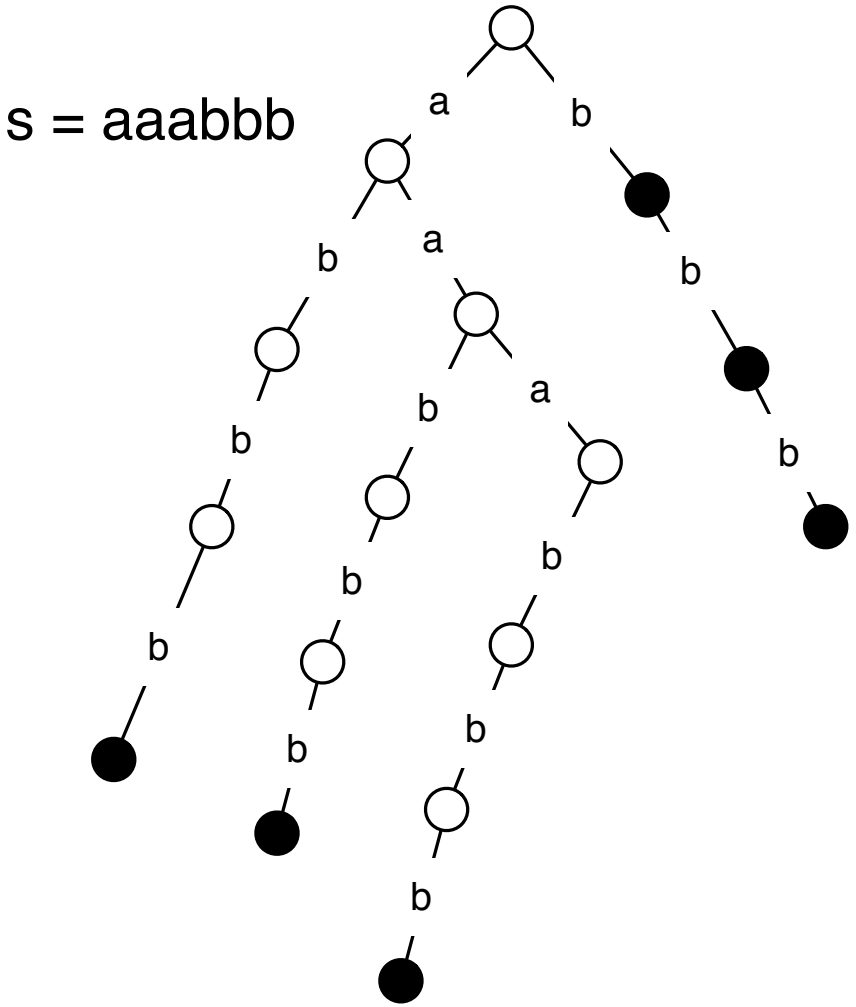
abaaba



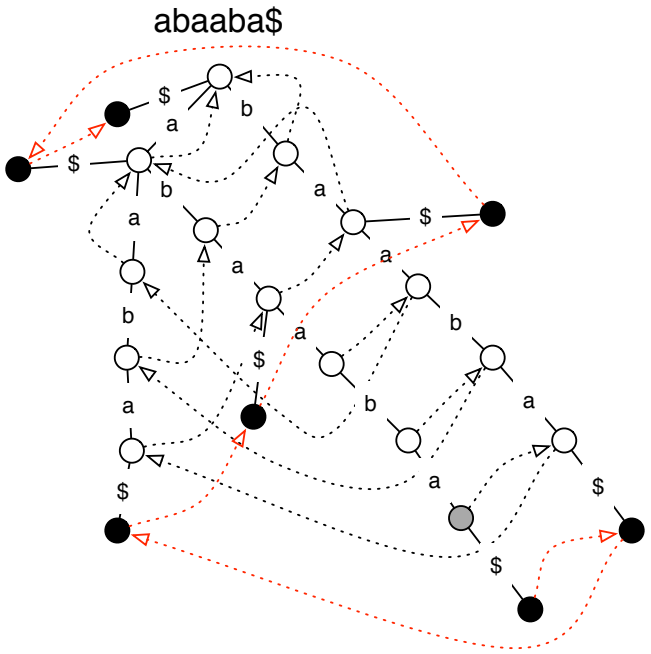
abaaba$



How many nodes can a suffix trie have?

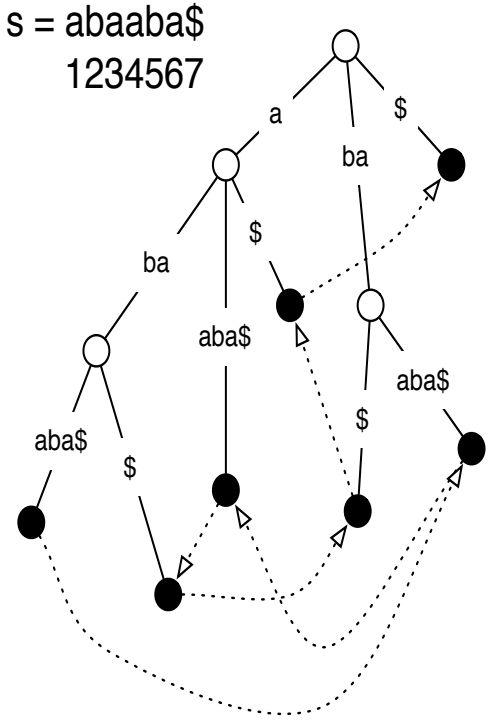


- $s = a^n b^n$ will have
  - 1 root node
  - n nodes in a path of "b"s
  - n paths of n+1 "b" nodes
- Total = $n(n+1)+n+1 = O(n^2)$ nodes.
- This is not very efficient.
- How could you make it smaller?

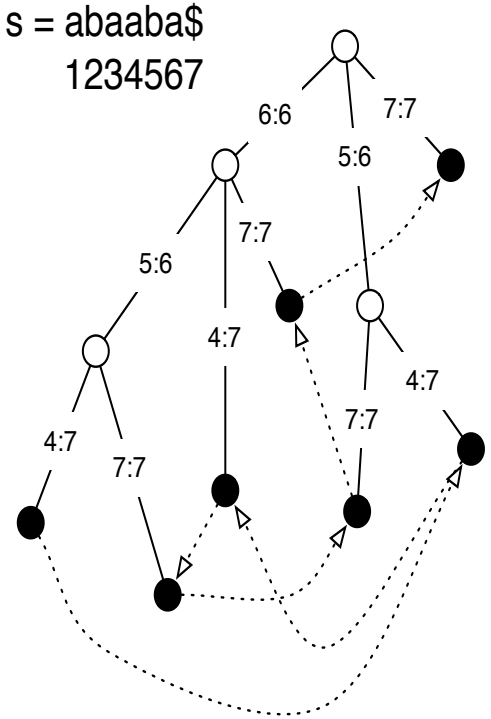


Suffix trees

A More Compact Representation



- Compress paths where there are no choices.



- Represent sequence along the path using a range [i,j] that refers to the input string s.

Space usage:

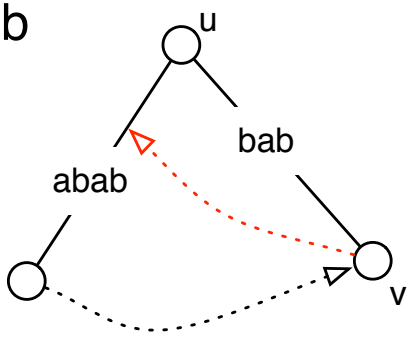
- In the compressed representation:
  - # leaves = O(n) [one leaf for each position in the string]
  - Every internal node is at least a binary split.
  - Each edge uses O(1) space.

- Therefore, # number of internal nodes is about equal to the number of leaves.
- And # of edges ≈ number of leaves, and space per edge is O(1).
- Hence, linear space.

Constructing Suffix Trees - Ukkonen's Algorithm

- The same idea as with the suffix trie algorithm.
- Main difference: not every trie node is explicitly represented in the tree.
- Solution: represent trie nodes as pairs (u, α), where u is a real node in the tree and α is some string leaving it.
- Some additional tricks to get to O(n) time.

s = abab



suffix_link[v] = (u, ab)

Suffix Arrays

- Even though Suffix Trees are O(n) space, the constant hidden by the big-Oh notation is somewhat "big": ≈ 20 bytes / character in good implementations.

- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. "Linear" but large.

- Suffix arrays are a more efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

- Slight space vs. time tradeoff.

Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

index of suffix	suffix of s
1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

sort the suffixes alphabetically

the indices just "come along for the ride"

8	$
5	atg$
1	attcatg$
4	catg$
7	g$
3	tcatg$
6	tg$
2	ttcatg$

Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

1	cattcat$
2	attcat$
3	ttcat$
4	tcat$
5	cat$
6	at$
7	t$
8	$

index of suffix suffix of s

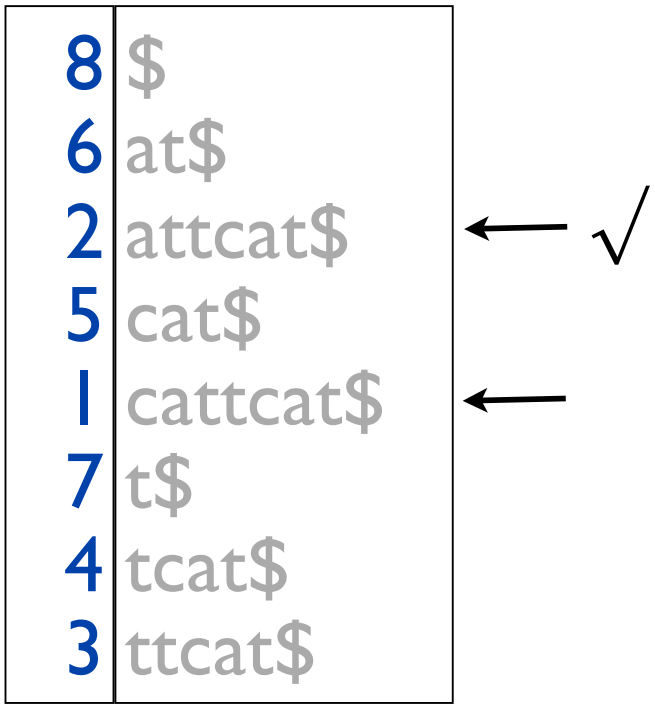
sort the suffixes alphabetically

the indices just "come along for the ride"

8
6
2
5
1
7
4
3

Search via Suffix Arrays

s = cattcat$



- Does string "at" occur in s?
- Binary search to find "at".
- What about "tt"?

Counting via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

- How many times does "at" occur in the string?
- All the suffixes that start with "at" will be next to each other in the array.
- Find one suffix that starts with "at" (using binary search).
- Then count the neighboring sequences that start with at.

Constructing Suffix Arrays

- Easy $O(n^2 \log n)$ algorithm:

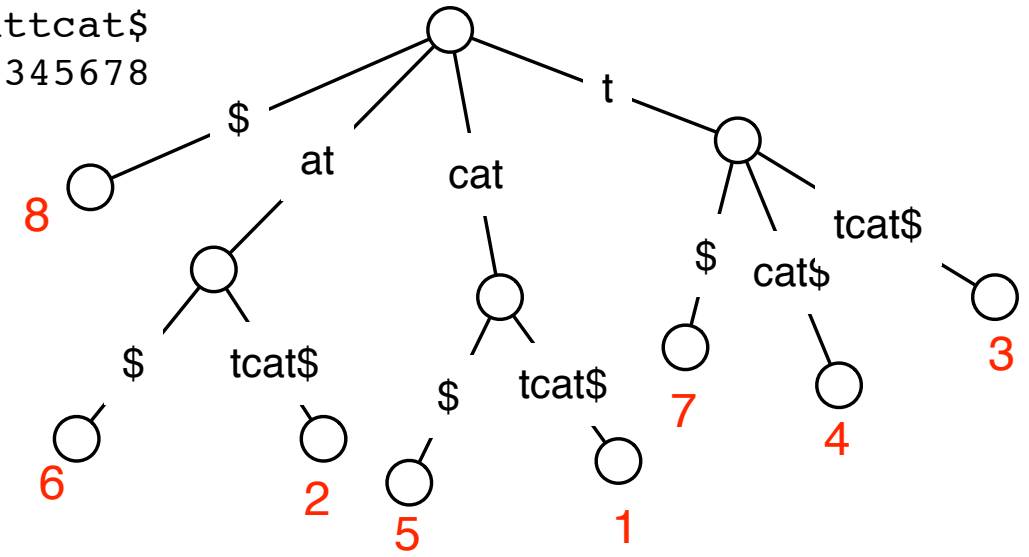
  sort the n suffixes, which takes O(n log n) comparisons, where each comparison takes O(n).

- There are several direct O(n) algorithms for constructing suffix arrays that use very little space.
- The Skew Algorithm is one that is based on divide-and-conquer.

- An simple O(n) algorithm: build the suffix tree, and exploit the relationship between suffix trees and suffix arrays (next slide)

Relationship Between Suffix Trees & Suffix Arrays

∑ = {$,a,c,t}
s = cattcat$
12345678



s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

Red #s = starting position of the suffix ending at that leaf

Leaf labels left to right: 86251743

Edges leaving each node are sorted by label (left-to-right).