Bioinformatics Algorithms

More Inexact Matching

Recap

- No class Tuesday
- HWs back - next week!

More Variations on Inexact Matching:
Bounding the number of differences.

Last time: k-mismatch.
(allow no insertions or deletions)
direct dynamic programming:

$O(mn)$

suffix tree approach:

$O(km)$

Useful because instead of maximizing a score, many applications want only exact (or nearly exact) copies of P in T.

High level: dyn programming vs. suffix tree

Next: extend to support both mismatches and spaces

But first - why??

Well, boils down to speed.

Most DNA comparisons don't have bounded differences.

But some do:

- searching for sequence tagged sites (STSs) & expressed sequence tags (ESTs) in newly sequenced DNA
- searching families for genetic diseases
- Molecular epidemiology: tracing transmission of a virus with a mutating genome

Key: nearly the same: bounding # of changes makes sense

2 variants:

① k-difference global alignment:

Input: S_1 + S_2, k

Goal: Find best global alignment of S_1 + S_2 with at most k mismatches or spaces (if one exists).

Really a special case of edit distance. (but smaller space of solutions)

② k-difference inexact matching:

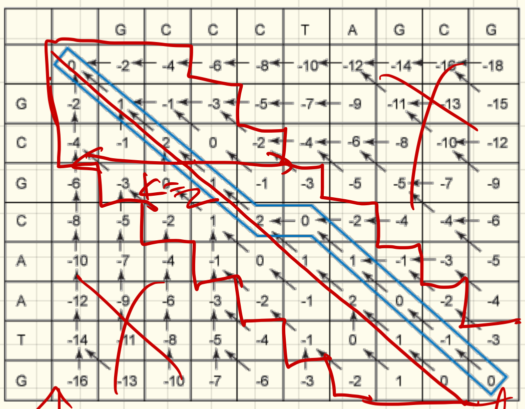
Given P + T, find all copies of P ~~as~~ in T which differ by at most k substitutions, insertions, or deletions.

Approach: Hybrid

① k-difference global alignment

Same as global alignment, but ensure $\leq k$ changes

Could do dynamic programming:



		G	C	C	C	T	A	G	C	G
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
G	-2	1	-1	-3	-5	-7	-9	-11	-13	-15
C	-4	-1	2	0	-2	-4	-6	-8	-10	-12
G	-6	-3	0	1	-1	-3	-5	-5	-7	-9
C	-8	-5	-2	1	2	0	-2	-4	-4	-6
A	-10	-7	-4	-1	0	1	1	-1	-3	-5
A	-12	-9	-6	-3	-2	-1	2	0	-2	-4
T	-14	-11	-8	-5	-4	-1	0	1	-1	-3
G	-16	-13	-10	-7	-6	-3	-2	1	0	0

↑ Store in each cell the # of insertion, deletions, & subs so far on path.

If $> k$, reject & try others

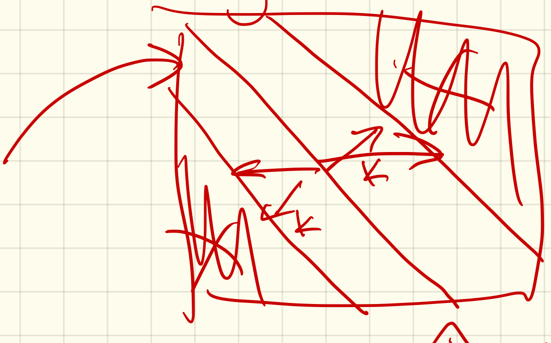
How to improve?!

Side note: What if k is unknown?

Run for $k=0$: (no)
Try $k=1$: (still no)
keep doubling until get a yes

$O(km)$

note diagonal is important:



$O(km)$ size

do dyn. programming here

② k-difference inexact matching:

(Essentially, like k-mismatch from Tuesday, but now allow spaces.)

Suffix tree issue:

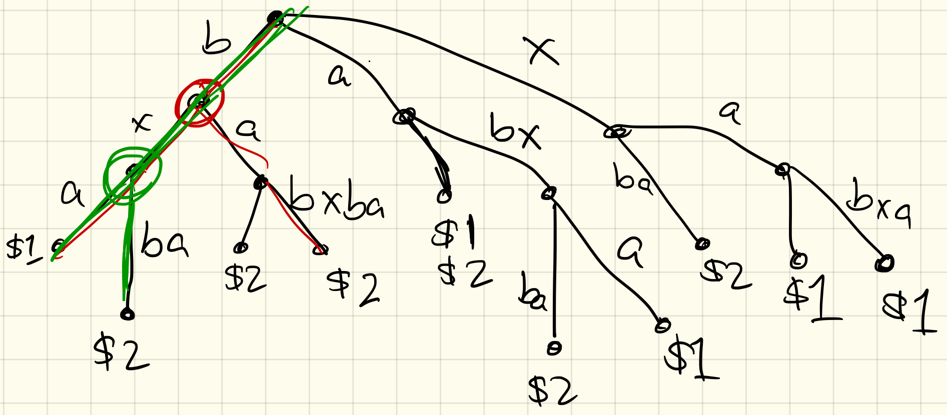
doesn't play well with insert/delete

Recall: We used longest extensions to "slide" over common substrings.

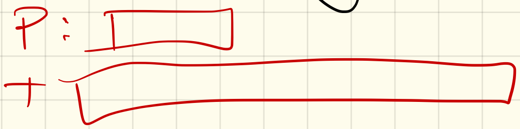
S_1 = xabxa (\$1)

S_2 = babxba (\$2)

Tree:



Also more difficult than global alignment:

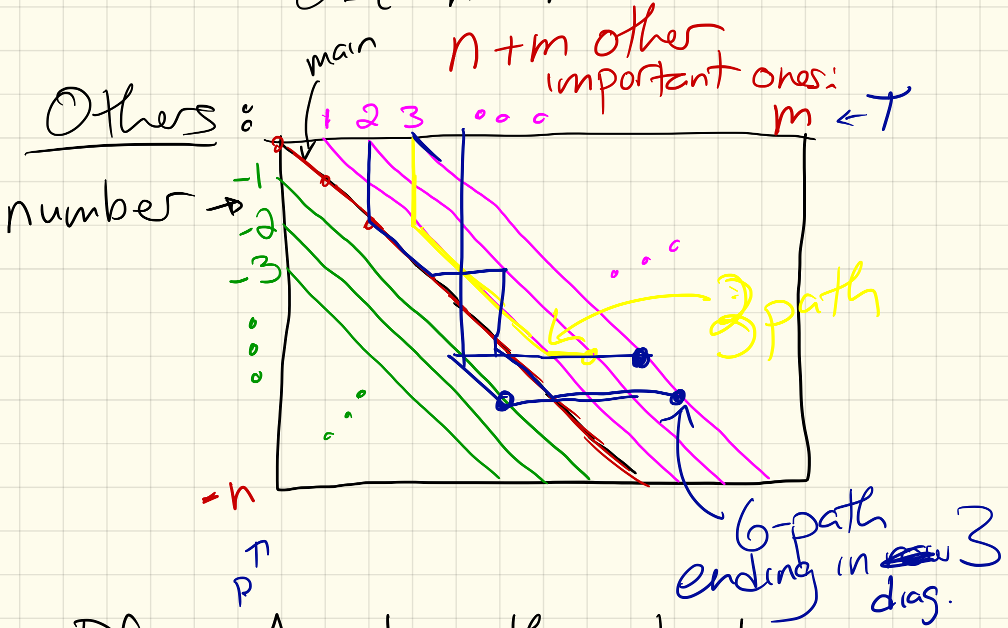


Since P + T are different lengths, the "diagonal" is not helpful.

<u>Solution</u>: Hybrid approach!
(first due to [Landau-Vishkin] + [Myers])

Dfn: Main diagonal is again all cells (i,i) ~~[illegible]~~ for $0 \le i \le n \le m$.

Others:



Dfn: A d-path starts in row 0 & specifies exactly d mismatches/spaces.

A d-path is farthest reaching in diagonal i if it:

- ends in diagonal i
- & ending column (in diagonal i) is ≥ any d-path ending in i

Now: Hybrid approach

- Will have k iterations, each in $O(m)$ time.

- in iteration $d \le k$, find farthest d-path on diagonal i (for all $-n \le i \le m$).

How? use the $(d-1)$-paths from the last iteration

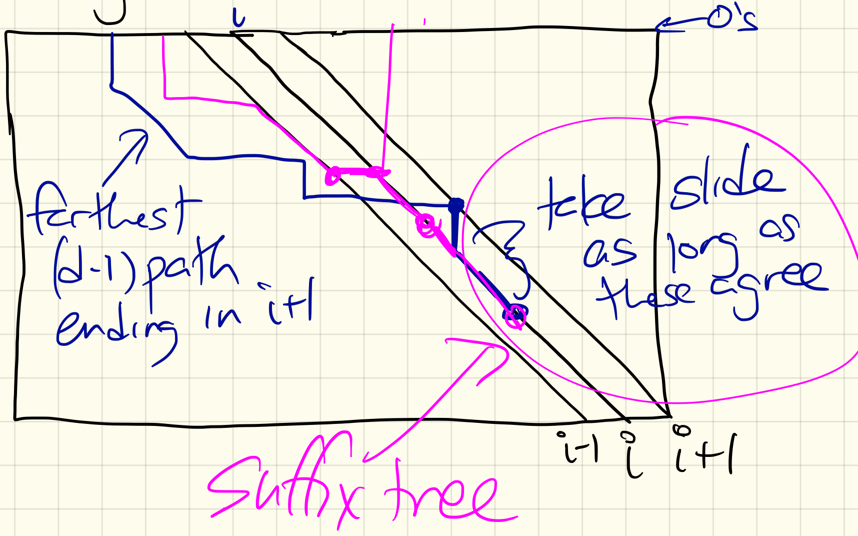
<u>Details</u>:

<u>For $d=0$</u>: this is just the longest common extention:

$O(i \cdot m)$ time using suffix trees

Next: For $d > 0$ + diagonal i, ← $n+m$
3 paths to consider:
(to keep farthest reaching d-path)

① R_1: the farthest reaching $(d-1)$-path on diagonal $i+1$, then a space (so a vertical edge in table), then longest extension along diagonal i:



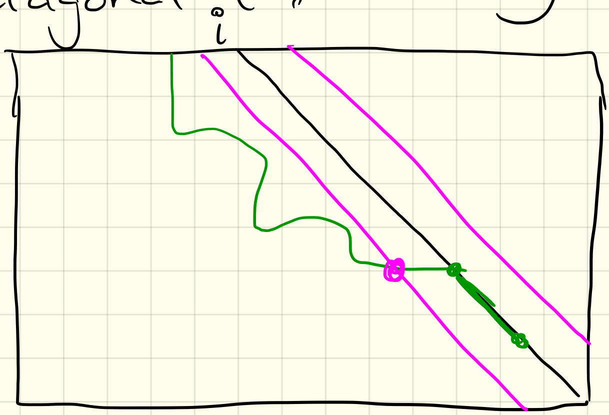
② R_2: farthest reaching $(d-1)$ path on $i-1$, then horizontal edge, then longest extension on i

③ R_3: fartest-reaching $(d-1)$ path on i, then diagonal mismatch, then longest extension.

The cool part:

These are the only choices!

If there is some better farthest reaching path w/ d errors ending in diagonal i:



- find last entry point for i
- Claim: when it crossed $i-1$ or $i+1$, would have had farther reaching $(d-1)$-path

Runtime & Space:

- d ranges from 0 to k
- $O(n+m)$ diagonals

$\Rightarrow O(km)$ space

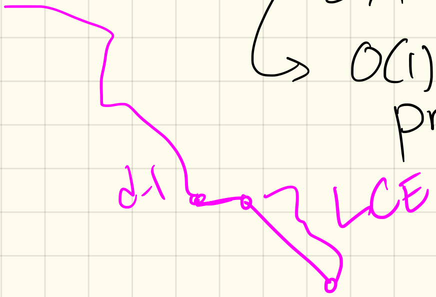
For time:

Loop from 0 to k.

- Inside, retrieve $O(m+n)$ past solutions

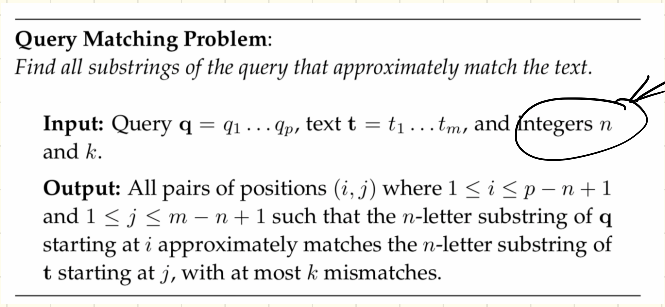
+ do longest common extension queries

$\hookrightarrow O(1)$ after linear preprocessing



$\Rightarrow O(km)$

Another variant:



Query Matching Problem:
Find all substrings of the query that approximately match the text.

Input: Query $\mathbf{q} = q_1 \ldots q_p$, text $\mathbf{t} = t_1 \ldots t_m$, and integers n and k.

sometimes

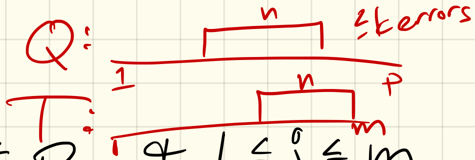
Output: All pairs of positions (i, j) where $1 \le i \le p - n + 1$ and $1 \le j \le m - n + 1$ such that the n-letter substring of $\mathbf{q}$ starting at i approximately matches the n-letter substring of $\mathbf{t}$ starting at j, with at most k mismatches.

The other book calls this the "threshhold all-against-all" problem.

Key difference:

- alignment of all pairs of substrings
  (not all pairs of strings)

Brute force:



For each $1 \le i \le p$ & $1 \le j \le m$,

do dynamic programing <u>table</u> for $P[i..p]$ & $T[j..m]$

- where edit distance can't be more than k

(If n - the length of substring - is specified, then it's D.P. for $P[i..(i+n-1)]$ & $T[j...(j+n-1)]$)

<u>Runtime</u>: $O(p^2 m^2)$

For each (i,j), quadratic DP table

$\sim O(n^4)$

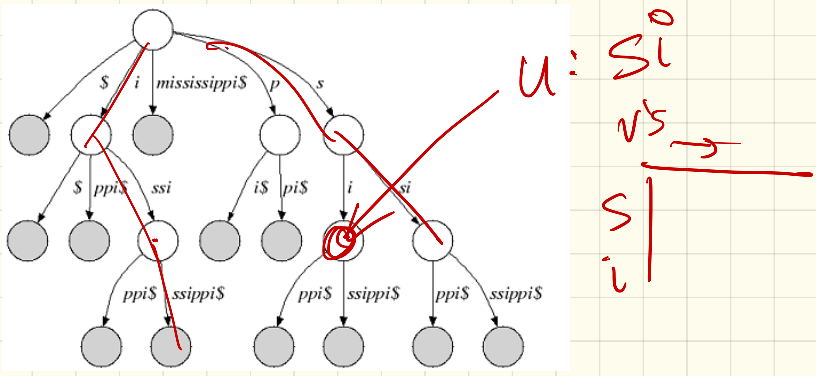
Another cool hybrid approach:
Build suffix trees for both P + T:

$\mathcal{T}_P$ and $\mathcal{T}_T$

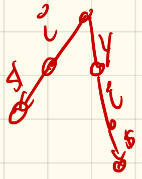
- each node represents a substring of P (or T)
- each substring in P is a prefix of some node

Ex: P = mississippi

$\mathcal{T}_P$:



T = marymayi



So: do dynamic programming

↳ but over all pairs of nodes from the trees.

More carefully:

prefix of some suffix

For $u \in \mathcal{T}_P$ & $v \in \mathcal{T}_T$,

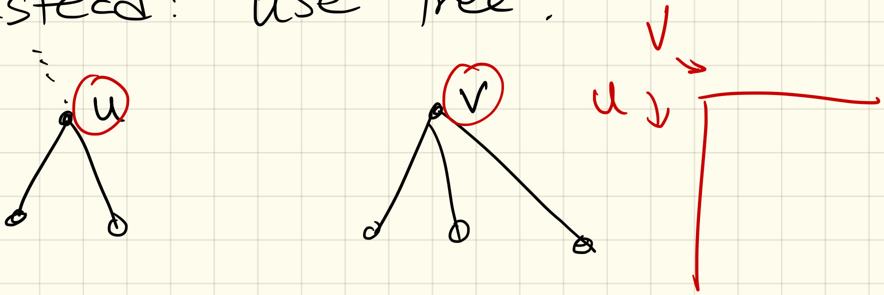
cell (u,v) is the DP table for edit distance from u's substring in P & v's substring in T.

Could solve by taking each pair of leaves & doing full DP.

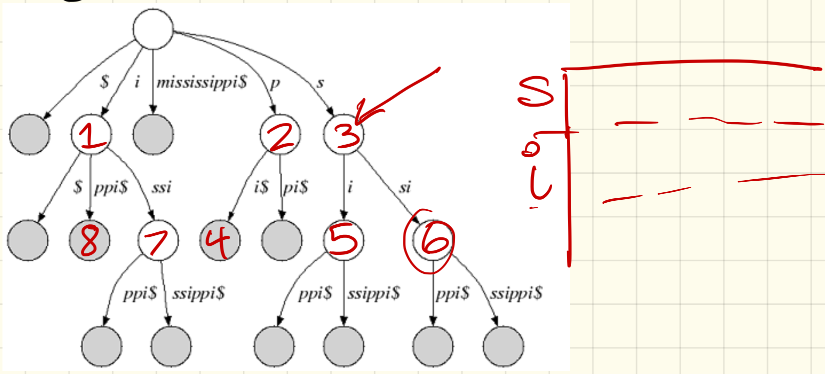
downside:

no better than normal DP

Instead: use tree!



Label nodes by string length in each tree



use parent's answer

Runtime: Well...

Worst case, no better.

But! In practice:

$$O(|T_p| \cdot |T_T| + \underset{\nwarrow \text{ output size}}{R})$$

(So: if tree compresses well, this is faster.)

Ex: In a few tests, seemed ~100 times faster for DNA.

(Amino acid test claimed even better.)

Another (heuristic) approach:

ℓ-mer filtration: If an n-letter P substring matches an n-letter substring of T, then <u>some</u> ℓ-mer is identical.

<u>Note</u>: ℓ-mers in common can be found by hashing:

If there aren't many - use these to isolate likely matches.

Pinning this down:

Theorem 9.1 *If the strings $x_1 \ldots x_n$ and $y_1 \ldots y_n$ match with at most k mismatches, then they share an l-mer for $l = \lfloor \frac{n}{k+1} \rfloor$, that is, $x_{i+1} \ldots x_{i+l} = y_{i+1} \ldots y_{i+l}$ for some $1 \leq i \leq n - l + 1$.*

<u>Pf</u>:

Algorithm:

- Find all matches of ℓ-mers, for $\ell = \left\lfloor \frac{n}{k+1} \right\rfloor$.
- For each potential match, do expand to left + right until $k+1$ mismatches are found.

$\downarrow$

(Use suffix trees!)

Note: