Algorithms in Bioinformatics

Final bits of dynamic programming

Recap

For HW - careful about shuffle!

A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

$$\mathtt{BANANA}_{\mathtt{ANANAS}} \qquad \mathtt{BAN}_{\mathtt{ANA}}{}^{\mathtt{ANA}}{}_{\mathtt{NAS}} \qquad \mathtt{B}_{\mathtt{AN}}{}^{\mathtt{AN}}{}_{\mathtt{A}}{}^{\mathtt{A}}{}_{\mathtt{NA}}{}^{\mathtt{NA}}{}_{\mathtt{S}}$$

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

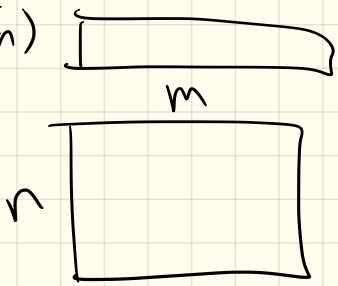
$$\mathtt{PRO}^{\mathtt{D}}{}_{\mathtt{G}}{}^{\mathtt{Y}}{}_{\mathtt{R}}{}^{\mathtt{NAM}}{}_{\mathtt{AMMI}}{}^{\mathtt{I}}{}_{\mathtt{N}}{}^{\mathtt{C}}{}_{\mathtt{G}} \qquad \mathtt{DY}_{\mathtt{PRO}}{}^{\mathtt{N}}{}_{\mathtt{G}}{}^{\mathtt{A}}{}_{\mathtt{R}}{}^{\mathtt{M}}{}_{\mathtt{AMM}}{}^{\mathtt{IC}}{}_{\mathtt{ING}}$$

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

each string (A + B) must keep same order in C

space + time

$O(n)$



Dynamic programming

- Utilize "optimal substructure"

  ie find a recursive formulation that tries all possibilities

- Then memoize:
  instead of doing all sorts of recursive calls, store in a data structure

- Often, will simply re-formulate to fill in the data structure iteratively

  Why? Languages (most) do better at iteration.

  ie. C-based (Java, Python, ...)

  Not true in functional languages:

  ie. Mathematica, Haskell

Ex: up through Local Alignment

Section 6.9:

penalty for indel is ρ
→ pays $\rho \cdot x$

"Gaps" — common in alignment, since DNA errors usually drop more than a single nucleotide.

So: want to modify penalty so that x spaces is not the same as x individual indels.

Defn: Affine gap penalty:

$$-(\rho + \sigma x)$$

Explanation:

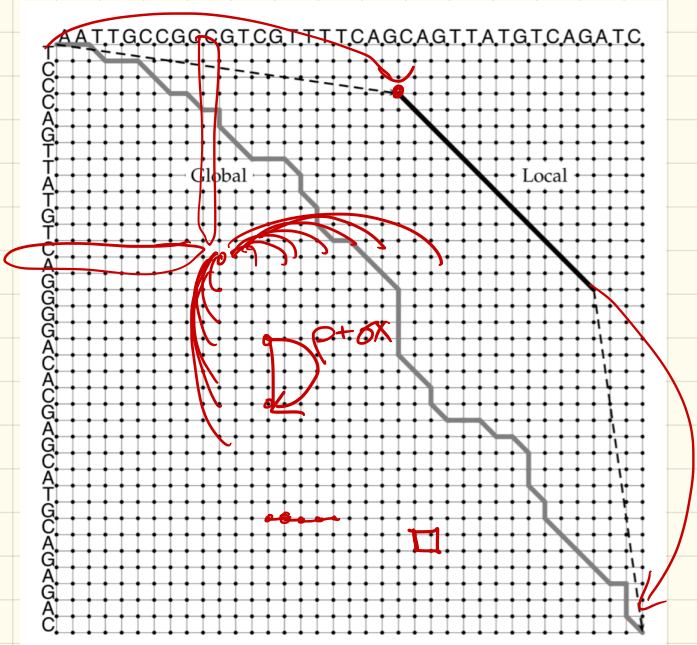
ρ is cost for even a single space gap (typically big(er))

σ - fairly small

Cool part: Algorithm is basically unchanged!

Back to the edit graph:

Need to modify, just like w/ local alignment.



do this for each x (s.t. $x \leq n$ or m)

Solution:

Add edge from (i,j) to $\left\{ \begin{matrix} (i, j+x) \\ (i+x, j) \end{matrix} \right\}$

of weight $-(\rho + \sigma x)$

The problem: runtime!

This changes 3 lookups to fill in a cell!

<u>Now</u>: check all of row i & column j

$\Rightarrow O(i+j)$

$= O(m+n)$

Total: $O(m \cdot n) \times O(m+n)$

$= O(n^3)$ if $n \gg m$

However, we can fix this:

Instead, track best path ending at (i,j)

- with no gap (diagonal edges - pay δ)
- with vertical gap
- with horizontal gap

So - 3 recurrences!

vertical gap

$$\overset{\downarrow}{s}_{i,j} = \max \begin{cases} \overset{\downarrow}{s}_{i-1,j} - \sigma \\ \vec{s}_{i-1,j} - (\rho + \sigma) \end{cases}$$

had $\rho + (x-1)\sigma$

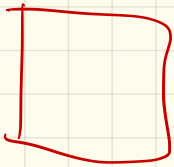
add σ, so now $\rho + x \cdot \sigma$

top: in middle of gap

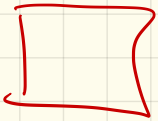
$$\vec{s}_{i,j} = \max \begin{cases} \vec{s}_{i,j-1} - \sigma \\ s_{i,j-1} - (\rho + \sigma) \end{cases}$$

horizontal gap

$\overset{\downarrow}{s}$



$\vec{s}$

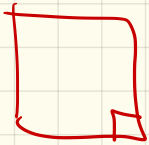


And then:

diagonal

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \overset{\downarrow}{s}_{i,j} \leftarrow \text{vertical gap} \\ \vec{s}_{i,j} \leftarrow \text{horizontal gap} \end{cases}$$

s:



End result:

pseudcode

for $i \leftarrow 1$ to n

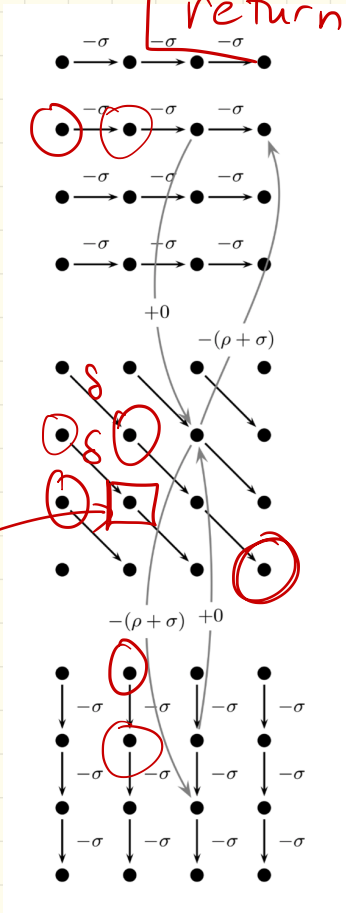
for $j \leftarrow 1$ to m

7 lookups { fill in $\vec{S}_{i,j}$

fill in $\vec{S}_{i,j}$

fill in $\overset{\downarrow}{S}_{i,j}$

return $S_{m,n}$



gaps in v

$\vec{S}_{i,j}$

$S_{i,j}$ normal alignment

$\overset{\downarrow}{S}_{i,j}$ gaps in w

to fill in i,j need ~7 table lookups

Next: Multiple Alignment (6.10)

Some similarities may not show up strongly with pairs.

But in larger group, obvious!

$k=3$

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C
 ||||| |   X|||| |          || XXX|||  | ||||  |
-ATTGC-G--ATTCGTAT------GGGACA-TGGATGCATGCAG-TGAC
```

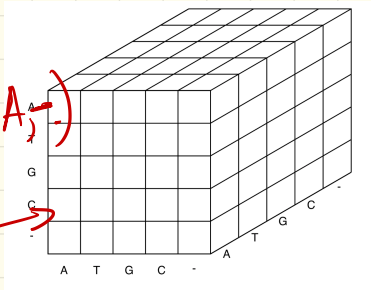
Assumptions / notation:

- inputs are $\vec{v}_1, \ldots, \vec{v}_k$ each a string of length n_i
- let $n \geq \max_i n_i$
- each row of alignment contains v_i plus $(n - n_i)$ -'s.
- Scoring function δ:

all A: 1 $\quad \delta(A, T, A, -)$

all different: bad

$k=3$



Example: $k=3$

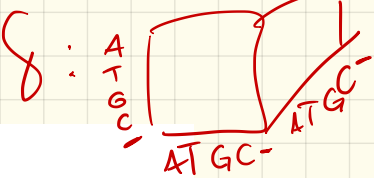
$S_{i,j,k}$ will be best alignment of:

$v_1[1..i]$
$v_2[1..j]$
$v_3[1..k]$

What will possibilities be? (think recursively!)

- could match $v_1[i]$ to $v_2[j]$ to $v_3[k]$
  pay $\delta(v_1[i], v_2[j], v_3[k])$
- indel in any of 3:
  $S_{i-1,j,k} + \delta(-, v_2[j], v_3[k])$
  $S_{i,j-1,k}$ δ :
  $S_{i,j,k-1}$ δ
- 2 indels

Recurrence:



$$s_{i,j,k} = \max \begin{cases} s_{i-1,j,k} & +\delta(v_i, -, -) \\ s_{i,j-1,k} & +\delta(-, w_j, -) \\ s_{i,j,k-1} & +\delta(-, -, u_k) \\ s_{i-1,j-1,k} & +\delta(v_i, w_j, -) \\ s_{i-1,j-1,k-1} & +\delta(v_i, -, u_k) \\ s_{i,j-1,k-1} & +\delta(-, w_j, u_k) \\ s_{i-1,j-1,k-1} & +\delta(v_i, w_j, u_k) \end{cases}$$

Picture:

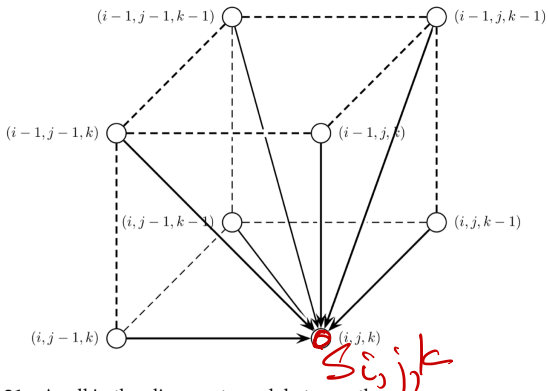
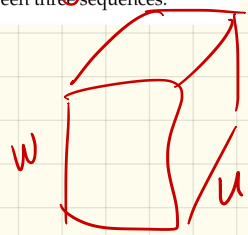


Figure 6.21 A cell in the alignment graph between three sequences.

Runtime (for $k=3$):



$O(n^3)$ entries

7 lookups each $\Rightarrow O(n^3)$

The bad news:

Consider more general k.

Size of matrix:

$$O(n^k)$$

Time to fill each cell:

$$\overline{1}\ \overline{2}\ \overline{3}\ \cdots\ \overline{k}$$

$$O(2^k - 1)$$

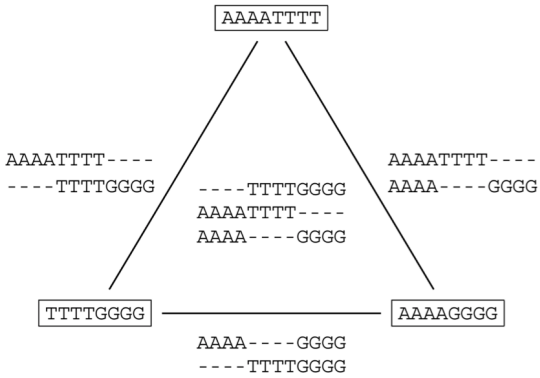
Runtime:

$$O(2^k n^k)$$

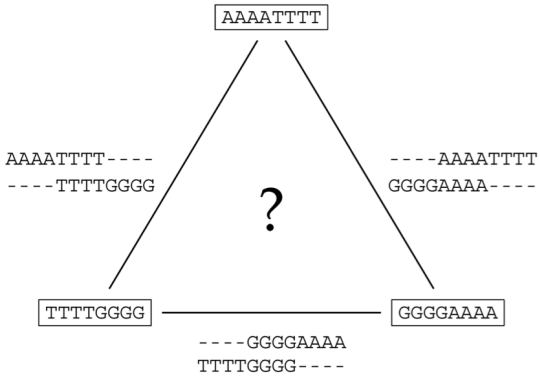
exponential in # of strings

So how to do better?

Pairwise alignments might not be possible to combine:



(a) Compatible pairwise alignments



(b) Incompatible pairwise alignments

People still do this though!

<u>CLUSTAL</u>:

- Computes all pairwise alignments + chooses the strongest.
- Merge these two (so stuck with their gaps)
- Reduce to k-1 strings (+ repeat).

<u>Note</u>: What kind of strategy?

Greedy!

Why? Assumption that a high score is a good indication they are close.

More on scoring function:

- k-dimensional matrix δ isn't really practical (especially if >4 letters!)

Many other variations — the choice of δ can drastically affect quality.

Note: none of this changes the recurrence!

(Often, which δ you use depends on your goal.)

2 commonly used examples:

1. Entropy approach:

let p_x = frequency of letter x in a column

for each column:

$$\text{entropy} = \sum_x p_x \log p_x$$

Ex: each nucleotide $\frac{n}{4}$ times
$\Rightarrow$ entropy $= -2$

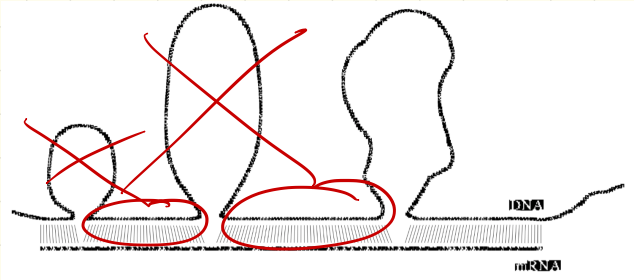
only one nucleotide
$\Rightarrow$ entropy $= 0$

2. Sum-of-pairs score for δ:

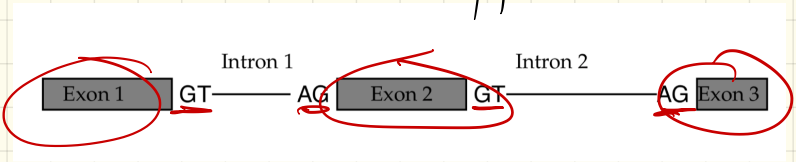
- Compute all pairwise scores for v_i & v_j
- add them up

Gene Prediction (6.11)

- intron/exon model of a gene
  (especially in eukaryotic organisms)



One approach: splicing signals
(statistical approach, 6.12)



Problem: these profiles are pretty weak
(we'll discuss a bit anyway, since still used)

Codons: groups of 3

Stop codons: TAA, TAG, TGA

Example: ATGCTTAGTCTG

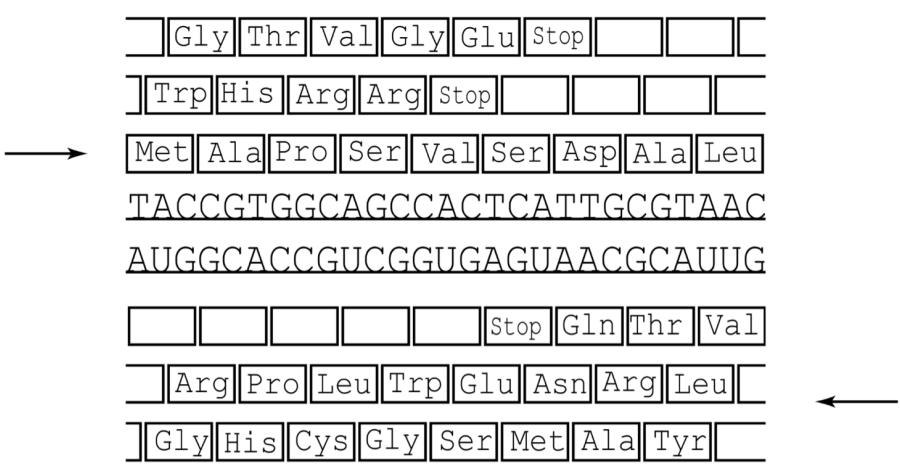


Figure 6.25 The six reading frames for the sequence ATGCTTAGTCTG. The string may be read forward or backward, and there are three frame shifts in each direction.

64 possible 3-letter groups

Avg # of codons b/t 2 stops is $\frac{64}{3} \approx 21$.

(Roughly 300 codons in avg protein)

Often, write down codon usage:

frequency of each occurence of each codon within a given sequence

	U			C			A			G		
U	UUU	Phe	57	UCU	Ser	16	UAU	Tyr	58	UGU	Cys	45
	UUC	Phe	43	UCC	Ser	15	UAC	Tyr	42	UGC	Cys	55
	UUA	Leu	13	UCA	Ser	13	UAA	Stp	62	UGA	Stp	30
	UUG	Leu	13	UCG	Ser	15	UAG	Stp	8	UGG	Trp	100
C	CUU	Leu	11	CCU	Pro	17	CAU	His	57	CGU	Arg	37
	CUC	Leu	10	CCC	Pro	17	CAC	His	43	CGC	Arg	38
	CUA	Leu	4	CCA	Pro	20	CAA	Gln	45	CGA	Arg	7
	CUG	Leu	49	CCG	Pro	51	CAG	Gln	66	CGG	Arg	10
A	AUU	Ile	50	ACU	Thr	18	AAU	Asn	46	AGU	Ser	15
	AUC	Ile	41	ACC	Thr	42	AAC	Asn	54	AGC	Ser	26
	AUA	Ile	9	ACA	Thr	15	AAA	Lys	75	AGA	Arg	5
	AUG	Met	100	ACG	Thr	26	AAG	Lys	25	AGG	Arg	3
G	GUU	Val	27	GCU	Ala	17	GAU	Asp	63	GGU	Gly	34
	GUC	Val	21	GCC	Ala	27	GAC	Asp	37	GGC	Gly	39
	GUA	Val	16	GCA	Ala	22	GAA	Glu	68	GGA	Gly	12
	GUG	Val	36	GCG	Ala	34	GAG	Glu	32	GGG	Gly	15

Note: species specific!

Slide & use these likelihoods -

higher scores in a run (higher likelihood) show up as peaks

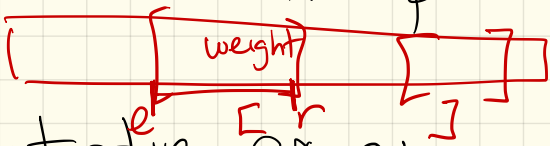
(Maybe more later in course...)

Second (better) approach:

Similarity-based (6.13)

Relies on previously sequenced genes & their proteins

So: Given a known target protein and a genomic sequence, find substrings (exons) of the sequence that match the protein.



Defns: putative exon:
possible exon

Each gets a triple (l, r, w):
Start point end likelihood

Maximum chain:
maximum weight subset of exons s.t. no 2 overlap

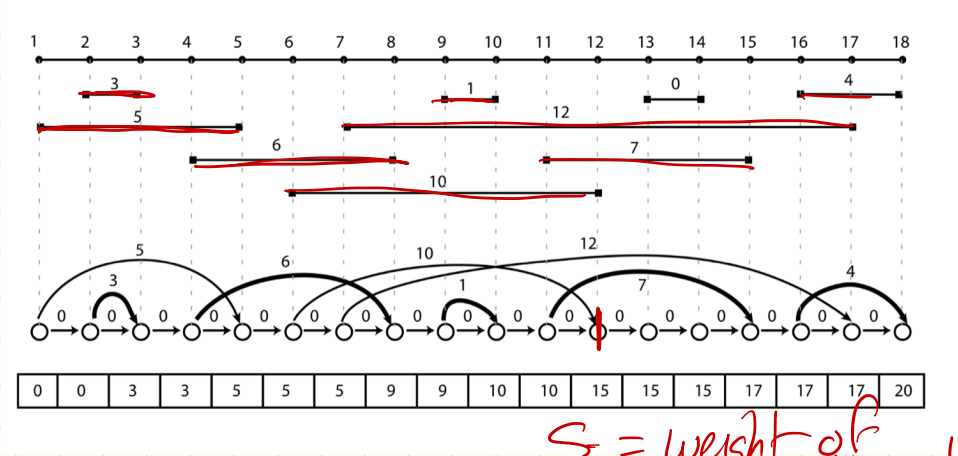
Problem:

Exon Chaining Problem:
Given a set of putative exons, find a maximum set of nonoverlapping putative exons.

Input: A set of weighted intervals (putative exons).

Output: A maximum chain of intervals from this set.

Turn into a graph:



S_i = weight of best path ending at v_i

putative exons: weighted edges

Goal: maximum length path from left vertex to right

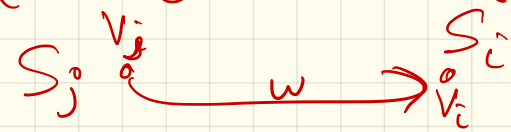
Let s_i = longest path ending at vertex i

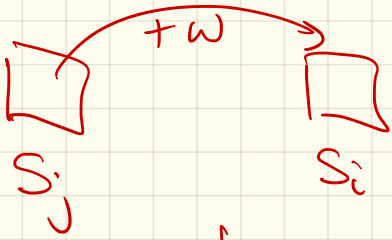
(Solution: read s_n)

So: fill in s_i

How?

Look at all edges that end at s_i





Find their best path + add weight

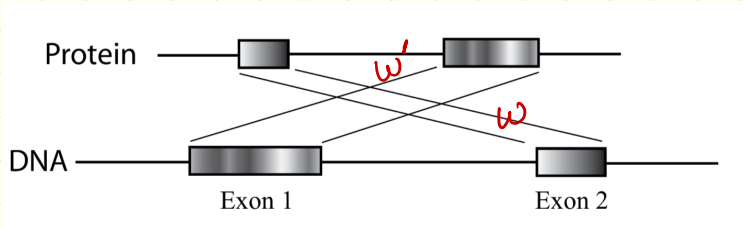
Store maximum over all of these

Final pseudocode:

```
EXONCHAINING(G, n)
for i ← 1 to 2n
    s_i ← 0
for i ← 1 to 2n
    if vertex v_i in G corresponds to the right end of an interval I
        j ← index of vertex for left end of the interval I
        w ← weight of the interval I
        s_i ← max {s_j + w, s_{i-1}}
    else
        s_i ← s_{i-1}
return s_2n
```

lck

Problem: the scoring (again)



Reminder: No class Thursday.

You have homework.

Please also read 6.14

Next HW - up next week, over more dynamic programming.

Next week: Divide & Conquer