CSCI X490 Algorithms for Computational Biology

Lecture Note 2 (by Liming Cai)

February 23, 2016

Structure of the Course

Structure of the Course

- Part I. Introduction to Algorithms (Chapter 2)
- Part II. Fundamental Techniques (Chapters 4 - 6)
- Part III Advanced Algorithms (Chapters 7 - 10)
- Part IV Probabilistic Methods (Chapters 11 - 12)

Chapter 6. Dynamic Programming

Chapter 6. Dynamic Programming

Chapter 6. Dynamic Programming

Chapter 6. Dynamic Programming

Chapter 6. Dynamic Programming

Chapter 6. Dynamic Programming

Chapter 6. Dynamic Programming

6.1 The power of DNA sequence comparison

cancer-causing oncogene

cystic fibrosis

Chapter 6. Dynamic Programming

6.2 The change problem

Given amount of money M, find a way to change M into the smallest number of coins from denominations $c = \{c_1, c_2, \ldots, c_d\}$.

Chapter 6. Dynamic Programming

6.2 The change problem

Given amount of money M, find a way to change M into the smallest number of coins from denominations $c = \{c_1, c_2, \ldots, c_d\}$.

For example, $c = \{1, 5, 10, 25\}$ for the US money.

Chapter 6. Dynamic Programming

Dynamic programming seeks solutions that can be computed from solutions to subproblems;

Chapter 6. Dynamic Programming

Dynamic programming seeks solutions that can be computed from solutions to subproblems;

Step 1: analysis of problem (*top-down*)

Assume S contains minimum number of coins for $M = 77$ cents. Then *there are at most 4 following scenarios to consider*:

Chapter 6. Dynamic Programming

Dynamic programming seeks solutions that can be computed from solutions to subproblems;

Step 1: analysis of problem (*top-down*)

Assume S contains minimum number of coins for $M = 77$ cents. Then *there are at most 4 following scenarios to consider*:

(1) At least one penny coin is included in S; then S should also contain minimum number of coins for $77 - 1 = 76$ cents;

Chapter 6. Dynamic Programming

Dynamic programming seeks solutions that can be computed from solutions to subproblems;

Step 1: analysis of problem (*top-down*)

Assume S contains minimum number of coins for $M = 77$ cents. Then *there are at most 4 following scenarios to consider*:

(1) At least one penny coin is included in S; then S should also contain minimum number of coins for $77 - 1 = 76$ cents;

(2) At least one nickel coin is included in S; then S should also contain minimum number of coins for $77 - 5 = 72$ cents;

Chapter 6. Dynamic Programming

Dynamic programming seeks solutions that can be computed from solutions to subproblems;

Step 1: analysis of problem (*top-down*)

Assume S contains minimum number of coins for $M = 77$ cents. Then *there are at most 4 following scenarios to consider*:

(1) At least one penny coin is included in S; then S should also contain minimum number of coins for $77 - 1 = 76$ cents;

(2) At least one nickel coin is included in S; then S should also contain minimum number of coins for $77 - 5 = 72$ cents;

(3) At least one dime coin is included in S;....

(4) If one quarter coin is included in ....

Chapter 6. Dynamic Programming

But we do not known which one is the best option.

Chapter 6. Dynamic Programming

But we do not known which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

Chapter 6. Dynamic Programming

But we do not known which one is the best option.

> But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

But it is difficult to represent 'solutions'.

Chapter 6. Dynamic Programming

But we do not known which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

But it is difficult to represent 'solutions'.

Step 2: define objective function
and formulate recurrences (*top-down*)

Chapter 6. Dynamic Programming

But we do not known which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

But it is difficult to represent 'solutions'.

Step 2: define objective function
and formulate recurrences (*top-down*)

Instead, we define a single numerical value on solution:
the smallest number of coins used:

Chapter 6. Dynamic Programming

But we do not known which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

But it is difficult to represent 'solutions'.

Step 2: define objective function
and formulate recurrences (*top-down*)

Instead, we define a single numerical value on solution:
the smallest number of coins used:

$$smallestNumCoins(M) = \min \begin{cases} smallestNumCoins(M-1)+1, \\ smallestNumCoins(M-5)+1, \\ smallestNumCoins(M-10)+1, \\ smallestNumCoins(M-25)+1 \end{cases}$$

a recurrence to the numerical answer.

Chapter 6. Dynamic Programming

Step 3: computing $smallestNumCoins$ with an algorithm

Chapter 6. Dynamic Programming

Step 3: computing $smallestNumCoins$ with an algorithm

A **straightforward top-down recursive** algorithm:

```
StraightForwardRecursiveChanges($M, c$)
1.  if $M = 0$
2.      return (0)
3.  else
4.      $v_{min} \leftarrow M$
5.      for $i \leftarrow 1$ to $|c|$
6.          if $M - c_i \geq 0$
7.          $v_i \leftarrow$ StraightForwardRecursiveChanges($M - c_i, c$) $+1$
8.              if $v_i < v_{min}$
9.                  $v_{min} \leftarrow v_i$
10.      return ($v_{min}$)
```

Chapter 6. Dynamic Programming

Step 3: computing $smallestNumCoins$ with an algorithm

A **straightforward top-down recursive** algorithm:

```
StraightForwardRecursiveChanges($M, c$)
1.    if $M = 0$
2.       return (0)
3.    else
4.       $v_{min} \leftarrow M$
5.       for $i \leftarrow 1$ to $|c|$
6.          if $M - c_i \geq 0$
7.          $v_i \leftarrow$ StraightForwardRecursiveChanges($M - c_i, c$) $+1$
8.             if $v_i < v_{min}$
9.                $v_{min} \leftarrow v_i$
10.       return ($v_{min}$)
```

Apparently there are a lot of re-computations.

Chapter 6. Dynamic Programming

A **less naive, top-down recursive** algorithm by keeping a table $T_{1,\ldots,M}$

Chapter 6. Dynamic Programming

A **less naive, top-down recursive** algorithm by keeping a table $T_{1,\dots,M}$

Initially, $T_k = -1$, for all $k = 1, 2, \dots, M$.

```
LessNaiveRecursiveChanges($M, c$)
1.  if $M = 0$
2.      return $0$
3.  else
4.      $v_{min} \leftarrow M$
5.      for $i \leftarrow 1$ to $|c|$
6.          if $M - c_i \geq 0$
7.              if $T_{M-c_i} = -1$
8                   $T_{M-c_i} \leftarrow$ LessNaiveRecursiveChanges($M - c_i, c$)
9.              if $T_{M-c_i} + 1 < v_{min}$
10.                 $v_{min} \leftarrow T_{M-c_i} + 1$
11.     $T_M \leftarrow v_{min}$
12.     return
```

Note: T is global, as a "communication media".

Chapter 6. Dynamic Programming

A **bottom-up, iterative** algorithm

Chapter 6. Dynamic Programming

A **bottom-up, iterative** algorithm

```
DPChanges$(n, c, T)$
1.  $T_0 = 0$
2.  **for** $n \leftarrow 1$ **to** $M$
3.      $v_{min} \leftarrow n$
4.      **for** $i \leftarrow 1$ **to** $|c|$
5.          **if** $n - c_i \geq 0$
6.              **if** $T_{n-c_i} + 1 \leq v_{min}$
7.                  $v_{min} \leftarrow T_{n-c_i} + 1$
8.      $T_n \leftarrow v_{min}$
9   **return**
```

Chapter 6. Dynamic Programming

Step 4: Compute a solution, not just the numerical solution

Chapter 6. Dynamic Programming

Step 4: Compute a solution, not just the numerical solution

We keep another array $coinFrom$ to record how and from which amount of money a coin was generated (the underlined parts).

Chapter 6. Dynamic Programming

Step 4: Compute a solution, not just the numerical solution

We keep another array $coinFrom$ to record how and from which amount of money a coin was generated (the underlined parts).

```
DPChanges($M, c, T, coinFrom$)
1.  $T_0 = 0$;  $\underline{coinFrom_0 = 0}$;
2.  for $n \leftarrow 1$ to $M$
3.      $v_{min} \leftarrow n$
4.      for $i \leftarrow 1$ to $|c|$
5.          if $n - c_i \geq 0$
6.              if $T_{n-c_i} + 1 \leq v_{min}$
7.                  $v_{min} \leftarrow T_{n-c_i} + 1$;  $\underline{from \leftarrow n - c_i}$
8.      $T_n \leftarrow v_{min}$; $\underline{coinFrom_n \leftarrow from}$
9   return
```

Chapter 6. Dynamic Programming

Example:

$M = 7$, $c = \{1, 2, 4\}$, $\quad c_1 = 1, c_2 = 2, c_3 = 4$

the result of running algorithm $\textsc{DPChanges}$:

Chapter 6. Dynamic Programming

Example:

$M = 7$, $c = \{1, 2, 4\}$, $\quad c_1 = 1, c_2 = 2, c_3 = 4$

the result of running algorithm $\text{DPC}\textsc{hanges}$:

n (cents)	0	1	2	3	4	5	6	7
T (minimum number of coins)	0	1	1	2	1	2	2	3
$coinFrom$ (other than the last coin)	0	0	0	2	0	1	2	3

Chapter 6. Dynamic Programming

6.4 Edit distance and alignments

edit distance: allowing the alignment of two sequences of different lengths

Chapter 6. Dynamic Programming

6.4 Edit distance and alignments

edit distance: allowing the alignment of two sequences of different lengths

different from Hamming distance

Chapter 6. Dynamic Programming

6.4 Edit distance and alignments

edit distance: allowing the alignment of two sequences of different lengths

different from Hamming distance

edit operations: *substitution*, *insertion*, and *deletion*

Chapter 6. Dynamic Programming

example:

Chapter 6. Dynamic Programming

example:

`TGCATAT` by deleting last `T`
`TGCATA` by deleting last `A`
`TGCAT` by insering `A` in the front
`ATGCAT` by substituting `C` for `G` in the third position
`ATCCAT` by inserting G before the last `A`
`ATCCGAT`

Chapter 6. Dynamic Programming

example:

`TGCATAT` by deleting last `T`
`TGCATA` by deleting last `A`
`TGCAT` by insering `A` in the front
`ATGCAT` by substituting `C` for `G` in the third position
`ATCCAT` by inserting G before the last `A`
`ATCCGAT`

and another series of operations:

`TGCATAT` by inserting `A` at the front
`ATGCATAT` by deleting the second `A`
`ATGCTAT` by substituting `C` for `G`
`ATCCTAT` by substituting `G` for the second `T`
`ATCCGAT`

Chapter 6. Dynamic Programming

example:

`TGCATAT` by deleting last `T`
`TGCATA` by deleting last `A`
`TGCAT` by insering `A` in the front
`ATGCAT` by substituting `C` for `G` in the third position
`ATCCAT` by inserting G before the last `A`
`ATCCGAT`

and another series of operations:

`TGCATAT` by inserting `A` at the front
`ATGCATAT` by deleting the second `A`
`ATGCTAT` by substituting `C` for `G`
`ATCCTAT` by substituting `G` for the second `T`
`ATCCGAT`

These two series of operations correspond to the alignments:

```
-TGC-ATAT          -TGCATAT
ATCCGAT--          ATCC-GAT
```

Chapter 6. Dynamic Programming

Given an alignment, we can present it as a path in a grid:

```
AT-GTTAT-
ATCGT-A-C
```

Chapter 6. Dynamic Programming

Given an alignment, we can present it as a path in a grid:

```
AT-GTTAT-
ATCGT-A-C
```

$$(0,0) \to (1,1) \to (2,2) \to (2,3) \to (3,4) \to (4,5) \to (5,5) \to (6,6) \to (7,6) \to (7,7)$$

Chapter 6. Dynamic Programming

Given an alignment, we can present it as a path in a grid:

```
AT-GTTAT-
ATCGT-A-C
```

$$(0,0) \rightarrow (1,1) \rightarrow (2,2) \rightarrow (2,3) \rightarrow (3,4) \rightarrow (4,5) \rightarrow (5,5) \rightarrow (6,6) \rightarrow (7,6) \rightarrow (7,7)$$

Given two sequences,

(1) there are more than one possible alignments;
(2) each alignment has a score (to be defined);
(3) each alignment corresponds to a path on a grid;
(4) the goal is to find a path (i.e., an alignment) with a highest score.

Chapter 6. Dynamic Programming

6.5 Longest common subsequences

A simplified scenario:

Chapter 6. Dynamic Programming

6.5 Longest common subsequences

A simplified scenario:

subsequence:
if $s =$`ATTGCTA`, the both `AGC` and `ATTA` are subsequences of s.

Chapter 6. Dynamic Programming

6.5 Longest common subsequences

A simplified scenario:

subsequence:
if $s =$`ATTGCTA`, the both `AGC` and `ATTA` are subsequences of s.

common subsequence:
`TCTA` is a *common* subsequence of two sequences $\texttt{A}\underline{\texttt{TCT}}\texttt{G}\underline{\texttt{A}}\texttt{T}$ and $\underline{\texttt{T}}\texttt{G}\underline{\texttt{C}}\texttt{A}\underline{\texttt{TA}}$

Chapter 6. Dynamic Programming

6.5 Longest common subsequences

A simplified scenario:

subsequence:
if s =ATTGCTA, the both AGC and ATTA are subsequences of s.

common subsequence:
TCTA is a *common* subsequence of two sequences $\texttt{A\underline{TCT}G\underline{A}T}$ and $\texttt{\underline{T}G\underline{C}A\underline{TA}}$

Finding a common subsequence is a simple case for alignment:

```
AT-C-TGAT
-TGCAT-A-
```

which only count the number of matches and not penalizing insertions or deletions or mismatches.

Chapter 6. Dynamic Programming

LONGEST COMMON SUBSEQUENCE PROBLEM:

Find the longest subsequence common to two strings.

Input: two strings, $v = v_1 \ldots v_n$ and $w = w_1 \ldots w_m$;
Output: The longest common subsequence (LCS) of v and w.

Chapter 6. Dynamic Programming

LONGEST COMMON SUBSEQUENCE PROBLEM:

Find the longest subsequence common to two strings.

Input: two strings, $v = v_1 \dots v_n$ and $w = w_1 \dots w_m$;
Output: The longest common subsequence (LCS) of v and w.

In the coin changing problem, we analyzed one coin at time. We looked at the last coin added.

Chapter 6. Dynamic Programming

LONGEST COMMON SUBSEQUENCE PROBLEM:

Find the longest subsequence common to two strings.

> **Input:** two strings, $v = v_1 \dots v_n$ and $w = w_1 \dots w_m$;
> **Output:** The longest common subsequence (LCS) of v and w.

In the coin changing problem, we analyzed one coin at time. We looked at the last coin added.

Here we analyze one character at a time. But since this problem concerns two sequences, we may analyze two characters (one on each sequence) at a time. We will look at the last characters on the sequences.

Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

Define: $s_{i,j}$ be the length of an LCS between prefix string $v_1 v_2 \dots v_i$ and prefix string $w_1 w_2 \dots w_j$.

Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

Define: $s_{i,j}$ be the length of an LCS between prefix string $v_1 v_2 \dots v_i$ and prefix string $w_1 w_2 \dots w_j$.

There are only three possibilities that would happen to v_i and w_j during an alignement, we have the recurrence for $s_{i,j}$:

Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

Define: $s_{i,j}$ be the length of an LCS between prefix string $v_1 v_2 \ldots v_i$ and prefix string $w_1 w_2 \ldots w_j$.

There are only three possibilities that would happen to v_i and w_j during an alignement, we have the recurrence for $s_{i,j}$:

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1, & \text{if } v_i = w_j \\ s_{i-1,j} + 0, & \\ s_{i,j-1} + 0, & \end{cases}$$

Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

Define: $s_{i,j}$ be the length of an LCS between prefix string $v_1 v_2 \dots v_i$ and prefix string $w_1 w_2 \dots w_j$.

There are only three possibilities that would happen to v_i and w_j during an alignement, we have the recurrence for $s_{i,j}$:

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1, & \text{if } v_i = w_j \\ s_{i-1,j} + 0, & \\ s_{i,j-1} + 0, & \end{cases}$$

$s_{0,j} = s_{i,0} = 0$ for all $0 \leq i \leq n$ and $0 \leq j \leq m$.

Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

Define: $s_{i,j}$ be the length of an LCS between prefix string $v_1 v_2 \dots v_i$ and prefix string $w_1 w_2 \dots w_j$.

There are only three possibilities that would happen to v_i and w_j during an alignement, we have the recurrence for $s_{i,j}$:

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1, & \text{if } v_i = w_j \\ s_{i-1,j} + 0, & \\ s_{i,j-1} + 0, & \end{cases}$$

$s_{0,j} = s_{i,0} = 0$ for all $0 \le i \le n$ and $0 \le j \le m$.

Again, we do not want to directly implement the recurrence using the top-down recursive approaches.

Chapter 6. Dynamic Programming

```
LCS($(v, w)$)
1.   for $i \leftarrow 0$ to $n$
2.     $s_{i,0} \leftarrow 0$
3.   for $j \leftarrow 0$ to $m$
4.     $s_{0,j} \leftarrow 0$
5.   for $i \leftarrow 1$ to $n$
6.     for $j \leftarrow 1$ to $m$
7.       if $v_i = w_j$
8.         $a \leftarrow 1$
9.       else
10.          $a \leftarrow -\infty$
11.      if $s_{i-1,j-1} + a > \max\{s_{i,j-1}, s_{i-1,j}\}$
12.          $s_{i,j} \leftarrow s_{i-1,j-1} + 1;\quad b_{i,j} \leftarrow ' \nwarrow '$
14.        else
15.          if $s_{i,j-1} > \max\{s_{i-1,j-1} + a, s_{i-1,j}\}$
16.            $s_{i,j} \leftarrow s_{i,j-1};\quad b_{i,j} \leftarrow ' \leftarrow '$
18.          else
19.            $s_{i,j} \leftarrow s_{i-1,j};\quad b_{i,j} \leftarrow ' \uparrow '$
10.    return
```

Chapter 6. Dynamic Programming

Figure 6.14 on page 173 (left table for LCS).

table T	0	1 T	2 G	3 C	4 A	5 T	6 A
0	0	0	0	0	0	0	0
1 A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ←
2 T	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
3 C	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
4 T	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
5 G	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
6 A	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
7 T	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

Chapter 6. Dynamic Programming

Figure 6.14 on page 173 (left table for LCS).

table	0	1	2	3	4	5	6
T		T	G	C	A	T	A
0	0	0	0	0	0	0	0
1 A	0	0 $\uparrow$	0 $\uparrow$	0 $\uparrow$	1 $\nwarrow$	1 $\leftarrow$	1 $\leftarrow$
2 T	0	1 $\nwarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\uparrow$	2 $\nwarrow$	2 $\leftarrow$
3 C	0	1 $\uparrow$	1 $\uparrow$	2 $\nwarrow$	2 $\leftarrow$	2 $\uparrow$	2 $\uparrow$
4 T	0	1 $\nwarrow$	1 $\uparrow$	2 $\uparrow$	2 $\uparrow$	3 $\nwarrow$	3 $\leftarrow$
5 G	0	1 $\uparrow$	2 $\nwarrow$	2 $\uparrow$	2 $\uparrow$	3 $\uparrow$	3 $\uparrow$
6 A	0	1 $\uparrow$	2 $\uparrow$	2 $\uparrow$	3 $\nwarrow$	3 $\uparrow$	4 $\nwarrow$
7 T	0	1 $\nwarrow$	2 $\uparrow$	2 $\uparrow$	3 $\uparrow$	4 $\nwarrow$	4 $\uparrow$

In notation of alignment:

```
A T - C - T G A T
- T G C A T - A -
↑ ↖ ← ↖ ← ↖ ↑ ↖ ↑
```

The LCS is between the two sequences is `TCTA`

Chapter 6. Dynamic Programming

Retrieve the corresponding LCS from table $b_{i,j}$.

Retrieve the corresponding LCS from table $b_{i,j}$.

The following recursive function prints the found LCS between $v_1 v_2 \dots v_i$ and string $w_1 w_2 \dots w_j$:

```
PrintLCS($b, v, i, j$)
1.    if $i = 0$ or $j = 0$
2.        return
3.    if $b_{i,j} = '\nwarrow'$
4.        PrintLCS($b, v, i-1, j-1$)
5.        print $v_i$
6.    else
7.        if $b_{i,j} = '\uparrow'$
8.            PrintLCS($b, v, i-1, j$)
9.        else
10.           PrintLCS($b, v, i, j-1$)
```

Chapter 6. Dynamic Programming

The LCS computes the similarity between two sequences, thus to maximize the length.

On the other hand, edit distance is to measure the similarity between the two using distance, thus to minimize the score.

$$d_{i,j} = \min \begin{cases} d_{i-1,j-1}, & \text{if } v_i = w_j \\ d_{i-1,j} + 1, & \\ d_{i,j-1} + 1, & \end{cases}$$

For alignment, usually we are looking for higher scores.

Chapter 6. Dynamic Programming

6.6 Global pairwise sequence alignment

Chapter 6. Dynamic Programming

6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

(1) Scoring matrices δ, 4×4 for nucleic acids and 20×20 for proteins, which include scores for matches and substitutions.

Chapter 6. Dynamic Programming

6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

(1) Scoring matrices δ, 4×4 for nucleic acids and 20×20 for proteins, which include scores for matches and substitutions.

(2) For insertion and deletion (*indel*, $'-'$), a penalty is applied.

Chapter 6. Dynamic Programming

6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

(1) Scoring matrices δ, 4×4 for nucleic acids and 20×20 for proteins, which include scores for matches and substitutions.

(2) For insertion and deletion (*indel*, $'-'$), a penalty is applied.

(3) If the penalty is uniform for every gap, thus linear, then it can be built into the scoring matrices, resulting in 5×5 and 21×21 matrices.

Chapter 6. Dynamic Programming

6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

(1) Scoring matrices δ, 4×4 for nucleic acids and 20×20 for proteins, which include scores for matches and substitutions.

(2) For insertion and deletion (*indel*, $'-'$), a penalty is applied.

(3) If the penalty is uniform for every gap, thus linear, then it can be built into the scoring matrices, resulting in 5×5 and 21×21 matrices.

(4) But often the gap penalty is not uniform. For example, **affine** gap penalty is defined as $o + e(l-1)$ for l consecutive gaps, where o is the *gap opening* penalty and e is the *gap extension* penalty.

Chapter 6. Dynamic Programming

Global Alignment Problem

Chapter 6. Dynamic Programming

Global Alignment Problem

Find the best alignment between two strings under a give scoring matrix.

Input: String v, w and a scoring matrix δ.
Output: An alignment of v and w whose score (as defined by the matrix δ) is the maximum among possible alignments of v and w.

Chapter 6. Dynamic Programming

GLOBAL ALIGNMENT PROBLEM

Find the best alignment between two strings under a give scoring matrix.

Input: String v, w and a scoring matrix δ.
Output: An alignment of v and w whose score (as defined by the matrix δ) is the maximum among possible alignments of v and w.

Define: $s_{i,j}$ to be the maximum score to align $v_1 \ldots v_i$ and $w_1 \ldots w_j$, given scoring matrix δ.

Chapter 6. Dynamic Programming

Global Alignment Problem

Find the best alignment between two strings under a give scoring matrix.

Input: String v, w and a scoring matrix δ.
Output: An alignment of v and w whose score (as defined by the matrix δ) is the maximum among possible alignments of v and w.

Define: $s_{i,j}$ to be the maximum score to align $v_1 \dots v_i$ and $w_1 \dots w_j$, given scoring matrix δ.

Like LCS, we do not know which of the three scenarios is the best, so

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

where $\delta(v_i, -)$ is the gap penalty for aligning v_i to a gap, etc..

Chapter 6. Dynamic Programming

```
GlobalPairwiseAlignment$(v, w)$
0.   $s_{0,0} = 0$
1.   for $i \leftarrow 1$ to $n$                initializing the first column
2.       $s_{i,0} \leftarrow \sum_{k=1}^{i} \delta(v_k, -)$
3.   for $j \leftarrow 1$ to $m$                initializing the first row
4.       $s_{0,j} \leftarrow \sum_{k=1}^{j} \delta(-, w_k)$
5.   for $i \leftarrow 1$ to $n$
6.      for $j \leftarrow 1$ to $m$             filling entries for the matrix
7.       if $s_{i-1,j-1} + \delta(v_i, w_j) > \max\{s_{i,j-1} + \delta(-, w_j), s_{i-1,j} + \delta(v_i, -)\}$
8.          $s_{i,j} \leftarrow s_{i-1,j-1} + \delta(v_i, w_j)$;   $b_{i,j} \leftarrow{}'\nwarrow{}'$
9.       else
10.        if $s_{i,j-1} + \delta(-, w_j) > \max\{s_{i-1,j-1} + \delta(v_i, w_j), s_{i-1,j} + \delta(v_i, -)\}$
11.              $s_{i,j} \leftarrow s_{i,j-1} + \delta(-, w_j)$ ;   $b_{i,j} \leftarrow{}'\leftarrow{}'$
12.        else
13.              $s_{i,j} \leftarrow s_{i-1,j} + \delta(v_i, -)$;   $b_{i,j} \leftarrow{}'\uparrow{}'$
14.     return
```

Chapter 6. Dynamic Programming

6.9 Alignment with affine gap penalty

So far, we have adopted the gap penalties that are "column-independent", such as $\delta(x, -)$.

Chapter 6. Dynamic Programming

6.9 Alignment with affine gap penalty

So far, we have adopted the gap penalties that are "column-independent", such as $\delta(x,-)$.

When $\delta(x,-)$ is a constant $-\gamma$, where $\gamma > 0$ fixed regardless of x, we can replace the recurrence for global alignment

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

Chapter 6. Dynamic Programming

6.9 Alignment with affine gap penalty

So far, we have adopted the gap penalties that are "column-independent", such as $\delta(x, -)$.

When $\delta(x, -)$ is a constant $-\gamma$, where $\gamma > 0$ fixed regardless of x, we can replace the recurrence for global alignment

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

with

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

Chapter 6. Dynamic Programming

Now if gap penalty for a gap of length l (i.e., number of single gaps) is defined as

$$\rho + (l-1)\sigma$$

where $\rho > 0$ is a gap opening penalty and $\sigma > 0$ is gap extension penalty.

Chapter 6. Dynamic Programming

Now if gap penalty for a gap of length l (i.e., number of single gaps) is defined as

$$\rho + (l-1)\sigma$$

where $\rho > 0$ is a gap opening penalty and $\sigma > 0$ is gap extension penalty.

We cannot simply replace $-\gamma$ with $-\rho - (l-1)\sigma$ because γ is for just one single gap.

Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \max\limits_{1 \le l \le j} \{s_{i,j-l} - \rho - (l-1)\sigma\} \\ \max\limits_{1 \le l \le i} \{s_{i-l,j} - \rho - (l-1)\sigma\} \end{cases}$$

Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \max\limits_{1 \le l \le j} \{s_{i,j-l} - \rho - (l-1)\sigma\} \\ \max\limits_{1 \le l \le i} \{s_{i-l,j} - \rho - (l-1)\sigma\} \end{cases}$$

Does it work?

Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \max\limits_{1 \le l \le j} \{s_{i,j-l} - \rho - (l-1)\sigma\} \\ \max\limits_{1 \le l \le i} \{s_{i-l,j} - \rho - (l-1)\sigma\} \end{cases}$$

Does it work?

Yes.

Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \max\limits_{1 \le l \le j} \{s_{i,j-l} - \rho - (l-1)\sigma\} \\ \max\limits_{1 \le l \le i} \{s_{i-l,j} - \rho - (l-1)\sigma\} \end{cases}$$

Does it work?

Yes.

But how much time would it take to build the DP table?

Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \max\limits_{1 \le l \le j} \{s_{i,j-l} - \rho - (l-1)\sigma\} \\ \max\limits_{1 \le l \le i} \{s_{i-l,j} - \rho - (l-1)\sigma\} \end{cases}$$

Does it work?

Yes.

But how much time would it take to build the DP table?

$O(n^2)$ time is needed for GlobalPairwiseAlignment.

Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \max\limits_{1 \le l \le j} \{ s_{i,j-l} - \rho - (l-1)\sigma \} \\ \max\limits_{1 \le l \le i} \{ s_{i-l,j} - \rho - (l-1)\sigma \} \end{cases}$$

Does it work?

Yes.

But how much time would it take to build the DP table?

$O(n^2)$ time is needed for GlobalPairwiseAlignment.

The above recurrence would required $O(n^3)$ time.

Chapter 6. Dynamic Programming

Solution-2: actually gap opening and gap extension penalties can be handled separately. For this, we need multiple recurrence-based DP techniques.

Chapter 6. Dynamic Programming

Solution-2: actually gap opening and gap extension penalties can be handled separately. For this, we need multiple recurrence-based DP techniques.

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

Chapter 6. Dynamic Programming

Solution-2: actually gap opening and gap extension penalties can be handled separately. For this, we need multiple recurrence-based DP techniques.

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

where $I_{i,j}$ is the score of optimal alignment between $v_1 \ldots v_i$ and $w_1 \ldots w_j$ for which w_j is inserted, and

Chapter 6. Dynamic Programming

Solution-2: actually gap opening and gap extension penalties can be handled separately. For this, we need multiple recurrence-based DP techniques.

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

where $I_{i,j}$ is the score of optimal alignment between $v_1 \ldots v_i$ and $w_1 \ldots w_j$ for which w_j is inserted, and

$D_{i,j}$ the score of optimal alignment between $v_1 \ldots v_i$ and $w_1 \ldots w_j$ for which v_i is for which v_i is deleted.

Chapter 6. Dynamic Programming

Thus,

$$I_{i,j} = \max \begin{cases} I_{i,j-1} - \sigma & \text{extending the gap} \\ s_{i,j-1} - \rho & \text{closing the gap} \end{cases}$$

Chapter 6. Dynamic Programming

Thus,

$$I_{i,j} = \max \begin{cases} I_{i,j-1} - \sigma & \text{extending the gap} \\ s_{i,j-1} - \rho & \text{closing the gap} \end{cases}$$

$$D_{i,j} = \max \begin{cases} D_{i-1,j} - \sigma & \text{extending the gap} \\ s_{i-1,j} - \rho & \text{closing the gap} \end{cases}$$

Chapter 6. Dynamic Programming

Thus,

$$I_{i,j} = \max \begin{cases} I_{i,j-1} - \sigma & \text{extending the gap} \\ s_{i,j-1} - \rho & \text{closing the gap} \end{cases}$$

$$D_{i,j} = \max \begin{cases} D_{i-1,j} - \sigma & \text{extending the gap} \\ s_{i-1,j} - \rho & \text{closing the gap} \end{cases}$$

We would need to computed three tables, one for each of $s_{i,j}, I_{i,j}$ and $D_{i,j}$.

Chapter 6. Dynamic Programming

Thus,

$$I_{i,j} = \max \begin{cases} I_{i,j-1} - \sigma & \text{extending the gap} \\ s_{i,j-1} - \rho & \text{closing the gap} \end{cases}$$

$$D_{i,j} = \max \begin{cases} D_{i-1,j} - \sigma & \text{extending the gap} \\ s_{i-1,j} - \rho & \text{closing the gap} \end{cases}$$

We would need to computed three tables, one for each of $s_{i,j}, I_{i,j}$ and $D_{i,j}$.

Each table can be computed in time $O(n^2)$.

Chapter 6. Dynamic Programming

6.7 Scoring alignment

Chapter 6. Dynamic Programming

6.7 Scoring alignment

Consider alignment

```
ATTGTTAT-
ATCGT-A-C
```

with a simple model. Assume $p(\mathtt{T}, \mathtt{C})$ to be probability that T aligns to C,

Chapter 6. Dynamic Programming

6.7 Scoring alignment

Consider alignment

```
ATTGTTAT-
ATCGT-A-C
```

with a simple model. Assume $p(\mathtt{T}, \mathtt{C})$ to be probability that T aligns to C,

then the score of (column 3) aligning T with C can be defined as the ratio:

$$\frac{p(\mathtt{T}, \mathtt{C})}{q(\mathtt{T})q(\mathtt{C})}$$

the denominator is the probability for T and C to occur independently.

Chapter 6. Dynamic Programming

6.7 Scoring alignment

Consider alignment

```
ATTGTTAT-
ATCGT-A-C
```

with a simple model. Assume $p(\mathtt{T}, \mathtt{C})$ to be probability that T aligns to C,

then the score of (column 3) aligning T with C can be defined as the ratio:

$$\frac{p(\mathtt{T}, \mathtt{C})}{q(\mathtt{T})q(\mathtt{C})}$$

the denominator is the probability for T and C to occur independently.

If the score for column 3 is greater than 1, it means T and C are evolutionarily related; otherwise unrelated.

Chapter 6. Dynamic Programming

Alignment probability is $\prod_{k=1}^{r} \frac{p(\bar{v}_k,\bar{w}_k)}{q(\bar{v}_k)q(\bar{w}_k)}$, i.e., product over r columns.

Chapter 6. Dynamic Programming

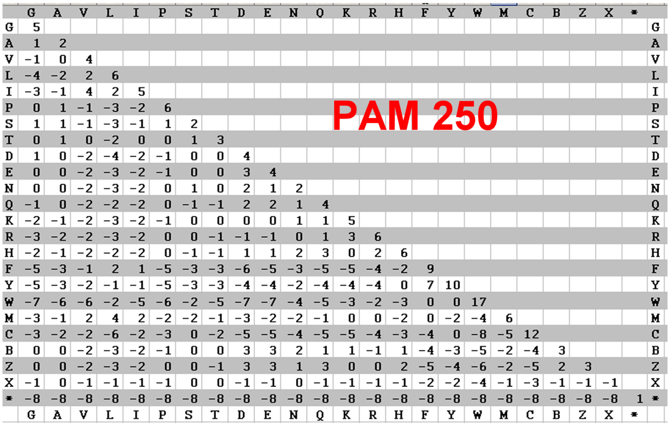
Alignment probability is $\prod_{k=1}^{r} \frac{p(\bar{v}_k, \bar{w}_k)}{q(\bar{v}_k)q(\bar{w}_k)}$, i.e., product over r columns.

Taken the logarithm, the score becomes either positive or negative, and the product becomes summation,

which the sum of column scores!

Chapter 6. Dynamic Programming

PAM (*point accepted mutations*) matrices



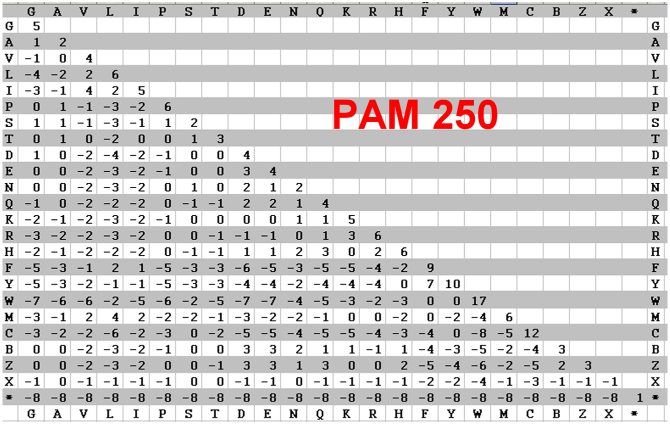
PAM 250

	G	A	V	L	I	P	S	T	D	E	N	Q	K	R	H	F	Y	W	M	C	B	Z	X	*	
G	5																								G
A	1	2																							A
V	-1	0	4																						V
L	-4	-2	2	6																					L
I	-3	-1	4	2	5																				I
P	0	1	-1	-3	-2	6																			P
S	1	1	-1	-3	-1	1	2																		S
T	0	1	0	-2	0	0	1	3																	T
D	1	0	-2	-4	-2	-1	0	0	4																D
E	0	0	-2	-3	-2	-1	0	0	3	4															E
N	0	0	-2	-3	-2	0	1	0	2	1	2														N
Q	-1	0	-2	-2	-2	0	-1	-1	2	2	1	4													Q
K	-2	-1	-2	-3	-2	-1	0	0	0	0	1	1	5												K
R	-3	-2	-2	-3	-2	0	0	-1	-1	-1	0	1	3	6											R
H	-2	-1	-2	-2	-2	0	-1	-1	1	1	2	3	0	2	6										H
F	-5	-3	-1	2	1	-5	-3	-3	-6	-5	-3	-5	-5	-4	-2	9									F
Y	-5	-3	-2	-1	-1	-5	-3	-3	-4	-4	-2	-4	-4	-4	0	7	10								Y
W	-7	-6	-6	-2	-5	-6	-2	-5	-7	-7	-4	-5	-3	-2	-3	0	0	17							W
M	-3	-1	2	4	2	-2	-2	-1	-3	-2	-2	-1	0	0	-2	0	-2	-4	6						M
C	-3	-2	-2	-6	-2	-3	0	-2	-5	-5	-4	-5	-5	-4	-3	-4	0	-8	-5	12					C
B	0	0	-2	-3	-2	-1	0	0	3	3	2	1	1	-1	1	-4	-3	-5	-2	-4	3				B
Z	0	0	-2	-3	-2	0	0	-1	3	3	1	3	0	0	2	-5	-4	-6	-2	-5	2	3			Z
X	-1	0	-1	-1	-1	-1	0	0	-1	-1	0	-1	-1	-1	-1	-2	-2	-4	-1	-3	-1	-1	-1		X
*	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	1	*
	G	A	V	L	I	P	S	T	D	E	N	Q	K	R	H	F	Y	W	M	C	B	Z	X	*	

Examine closely related protein sequences for mutation rates.

Chapter 6. Dynamic Programming

PAM (*point accepted mutations*) matrices



PAM 250

	G	A	V	L	I	P	S	T	D	E	N	Q	K	R	H	F	Y	W	M	C	B	Z	X	*	
G	5																								G
A	1	2																							A
V	-1	0	4																						V
L	-4	-2	2	6																					L
I	-3	-1	4	2	5																				I
P	0	1	-1	-3	-2	6																			P
S	1	1	-1	-3	-1	1	2																		S
T	0	1	0	-2	0	0	1	3																	T
D	1	0	-2	-4	-2	-1	0	0	4																D
E	0	0	-2	-3	-2	-1	0	0	3	4															E
N	0	0	-2	-3	-2	0	1	0	2	1	2														N
Q	-1	0	-2	-2	-2	0	-1	-1	2	2	1	4													Q
K	-2	-1	-2	-3	-2	-1	0	0	0	0	1	1	5												K
R	-3	-2	-2	-3	-2	0	0	-1	-1	-1	0	1	3	6											R
H	-2	-1	-2	-2	-2	0	-1	-1	1	1	2	3	0	2	6										H
F	-5	-3	-1	2	1	-5	-3	-3	-6	-5	-3	-5	-5	-4	-2	9									F
Y	-5	-3	-2	-1	-1	-5	-3	-3	-4	-4	-2	-4	-4	-4	0	7	10								Y
W	-7	-6	-6	-2	-5	-6	-2	-5	-7	-7	-4	-5	-3	-2	-3	0	0	17							W
M	-3	-1	2	4	2	-2	-2	-1	-3	-2	-2	-1	0	0	-2	0	-2	-4	6						M
C	-3	-2	-2	-6	-2	-3	0	-2	-5	-5	-4	-5	-5	-4	-3	-4	0	-8	-5	12					C
B	0	0	-2	-3	-2	-1	0	0	3	3	2	1	1	-1	1	-4	-3	-5	-2	-4	3				B
Z	0	0	-2	-3	-2	0	0	-1	3	3	1	3	0	0	2	-5	-4	-6	-2	-5	2	3			Z
X	-1	0	-1	-1	-1	-1	0	0	-1	-1	0	-1	-1	-1	-1	-2	-2	-4	-1	-3	-1	-1	-1		X
*	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	1	*
	G	A	V	L	I	P	S	T	D	E	N	Q	K	R	H	F	Y	W	M	C	B	Z	X	*	

Examine closely related protein sequences for mutation rates.

Chapter 6. Dynamic Programming

One PAM: the amount of time in which an “average” protein mutates 1% of its amino acids.

Let $f(i, j)$ be the frequency that amino acids i and j are aligned.

Chapter 6. Dynamic Programming

One PAM: the amount of time in which an “average” protein mutates 1% of its amino acids.

Let $f(i, j)$ be the frequency that amino acids i and j are aligned.

Let $f(i)$ be the frequency of amino acid i. Then $\frac{f(i,j)}{f(i)f(j)}$ is to measure i and j are aligned as oppose to they occur independently.

Chapter 6. Dynamic Programming

One PAM: the amount of time in which an "average" protein mutates 1% of its amino acids.

Let $f(i,j)$ be the frequency that amino acids i and j are aligned.

Let $f(i)$ be the frequency of amino acid i. Then $\frac{f(i,j)}{f(i)f(j)}$ is to measure i and j are aligned as oppose to they occur independently.

Let $f(i,j) = f(i)f(j|i)$, by considering f as a probability.

Chapter 6. Dynamic Programming

One PAM: the amount of time in which an "average" protein mutates 1% of its amino acids.

Let $f(i,j)$ be the frequency that amino acids i and j are aligned.

Let $f(i)$ be the frequency of amino acid i. Then $\frac{f(i,j)}{f(i)f(j)}$ is to measure i and j are aligned as oppose to they occur independently.

Let $f(i,j) = f(i)f(j|i)$, by considering f as a probability.

Let $g(i,j) = f(j|i)$ is the probability of mutating to j given amino acid i.

So the measure is $\frac{f(i,j)}{f(i)f(j)} = \frac{f(j|i)}{f(j)} = \frac{g(i,j)}{f(j)}$

Chapter 6. Dynamic Programming

One PAM: the amount of time in which an "average" protein mutates 1% of its amino acids.

Let $f(i,j)$ be the frequency that amino acids i and j are aligned.

Let $f(i)$ be the frequency of amino acid i. Then $\frac{f(i,j)}{f(i)f(j)}$ is to measure i and j are aligned as oppose to they occur independently.

Let $f(i,j) = f(i)f(j|i)$, by considering f as a probability.

Let $g(i,j) = f(j|i)$ is the probability of mutating to j given amino acid i.

So the measure is $\frac{f(i,j)}{f(i)f(j)} = \frac{f(j|i)}{f(j)} = \frac{g(i,j)}{f(j)}$

Taken logarithm, the measure becomes

$$\log \frac{g(i,j)}{f(j)} = \log \frac{observed\, frequency}{expected\, frequency}$$

Chapter 6. Dynamic Programming

PAM 1 matrix

The (i, j) entry in the PAM 1 matrix is $\log \frac{g(i,j)}{f(j)}$.

Chapter 6. Dynamic Programming

PAM 1 matrix

The (i, j) entry in the PAM 1 matrix is $\log \frac{g(i,j)}{f(j)}$.

Define G to be the matrix containing entries $g(i, j)$. Then

Chapter 6. Dynamic Programming

PAM 1 matrix

The (i, j) entry in the PAM 1 matrix is $\log \frac{g(i,j)}{f(j)}$.

Define G to be the matrix containing entries $g(i, j)$. Then

PAM 1 matrix $= \log(G/f(j))$, where $f(j)$ is applied to the jth column.

Chapter 6. Dynamic Programming

PAM 1 matrix

The (i, j) entry in the PAM 1 matrix is $\log \frac{g(i,j)}{f(j)}$.

Define G to be the matrix containing entries $g(i, j)$. Then

PAM 1 matrix $= \log(G/f(j))$, where $f(j)$ is applied to the jth column.

Define G^n to be the matrix obtained from G by multiplying itself n times.

Chapter 6. Dynamic Programming

PAM 1 matrix

The (i, j) entry in the PAM 1 matrix is $\log \frac{g(i,j)}{f(j)}$.

Define G to be the matrix containing entries $g(i, j)$. Then

PAM 1 matrix $= \log(G/f(j))$, where $f(j)$ is applied to the jth column.

Define G^n to be the matrix obtained from G by multiplying itself n times.

PAM n matrix is $\log(G^n/f(j))$, where $f(j)$ is applied to jth column.

Chapter 6. Dynamic Programming

PAM 1 matrix

The (i, j) entry in the PAM 1 matrix is $\log \frac{g(i,j)}{f(j)}$.

Define G to be the matrix containing entries $g(i, j)$. Then

PAM 1 matrix $= \log(G/f(j))$, where $f(j)$ is applied to the jth column.

Define G^n to be the matrix obtained from G by multiplying itself n times.

PAM n matrix is $\log(G^n/f(j))$, where $f(j)$ is applied to jth column.

But what does G^n mean and what does PAM n mean?

Chapter 6. Dynamic Programming

Assume G

G	X	Y	Z
X	a	**b**	c
Y	d	e	f
Z	g	h	i

$g(X, Y) = \mathbf{b}$

Chapter 6. Dynamic Programming

Assume G

G	X	Y	Z
X	a	**b**	c
Y	d	e	f
Z	g	h	i

$g(X, Y) = \mathbf{b}$

Consider G^2

G^2	X	Y	Z
X	...	$(ab + be + ch)$	...
Y	...	...	...
Z	...	...	...

Chapter 6. Dynamic Programming

Assume G

G	X	Y	Z
X	a	**b**	c
Y	d	e	f
Z	g	h	i

$g(X,Y) = \mathbf{b}$

Consider G^2

G^2	X	Y	Z
X	...	$(ab+be+ch)$	...
Y	...	...	...
Z	...	...	...

where

$$ab = g(X,X)g(X,Y)$$
$$be = g(X,Y)g(Y,Y)$$
$$ch = g(X,Z)g(Z,Y)$$

Chapter 6. Dynamic Programming

Assume G

G	X	Y	Z
X	a	**b**	c
Y	d	e	f
Z	g	h	i

$g(X, Y) = \mathbf{b}$

Consider G^2

G^2	X	Y	Z
X	...	$(ab + be + ch)$	...
Y	...	...	...
Z	...	...	...

where

$$ab = g(X, X)g(X, Y)$$
$$be = g(X, Y)g(Y, Y)$$
$$ch = g(X, Z)g(Z, Y)$$

These are probabilities of two step mutations from X to Y. So n is a multiple of the time unit.

Chapter 6. Dynamic Programming

Assume G

G	X	Y	Z
X	a	**b**	c
Y	d	e	f
Z	g	h	i

$g(X, Y) = \mathbf{b}$

Consider G^2

G^2	X	Y	Z
X	...	$(ab + be + ch)$	...
Y	...	...	...
Z	...	...	...

where

$$\begin{aligned} ab &= g(X, X)g(X, Y) \\ be &= g(X, Y)g(Y, Y) \\ ch &= g(X, Z)g(Z, Y) \end{aligned}$$

These are probabilities of two step mutations from X to Y. So n is a multiple of the time unit.

http://www.bioinformatics.nl/tools/pam.html

Chapter 6. Dynamic Programming

BLOSUM (*blocks of amino acid substitution*) matrices

Chapter 6. Dynamic Programming

BLOSUM (*blocks of amino acid substitution*) matrices

Scores within a BLOSUM are log-odds scores that measure, in an alignment, the logarithm for the ratio of the likelihood of two amino acids appearing with a biological sense and the likelihood of the same amino acids appearing by chance.

Chapter 6. Dynamic Programming

BLOSUM (*blocks of amino acid substitution*) matrices

Scores within a BLOSUM are log-odds scores that measure, in an alignment, the logarithm for the ratio of the likelihood of two amino acids appearing with a biological sense and the likelihood of the same amino acids appearing by chance.

The matrices are based on the minimum percentage identity of the aligned protein sequence used in calculating them.

Chapter 6. Dynamic Programming

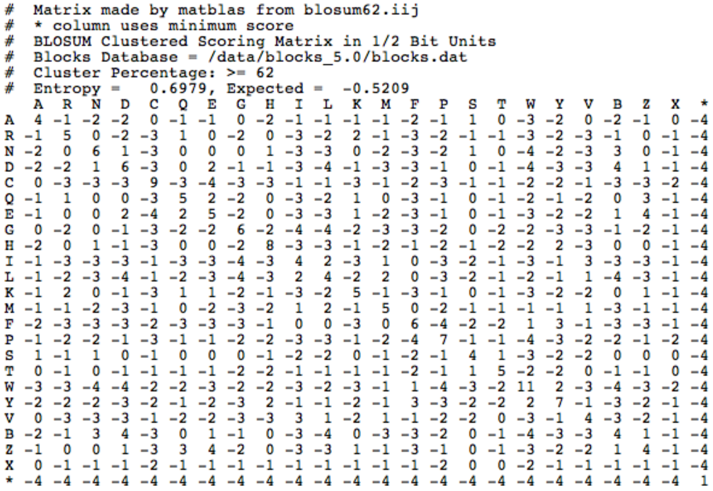
BLOSUM (*blocks of amino acid substitution*) matrices

Scores within a BLOSUM are log-odds scores that measure, in an alignment, the logarithm for the ratio of the likelihood of two amino acids appearing with a biological sense and the likelihood of the same amino acids appearing by chance.

The matrices are based on the minimum percentage identity of the aligned protein sequence used in calculating them.

Every possible identity or substitution is assigned a score based on its observed frequences in the alignment of related proteins

Chapter 6. Dynamic Programming



```
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1  0 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -3 -2 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0  3 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0  0 -1 -4
I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3 -3 -1 -4
L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4 -3 -1 -4
K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0  1 -1 -4
M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3 -1 -1 -4
F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3 -3 -1 -4
P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -1 -2 -4
S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0  0  0 -4
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1  0 -4
W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -3 -2 -4
Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -2 -1 -4
V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3 -2 -1 -4
B -2 -1  3  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4  1 -1 -4
Z -1  0  0  1 -3  3  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
X  0 -1 -1 -1 -2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2  0  0 -2 -1 -1 -1 -1 -1 -4
* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1
```

Chapter 6. Dynamic Programming

6.8 Local sequence alignment

To find conserved regions between two sequences that not necessarily similar overall.

Chapter 6. Dynamic Programming

6.8 Local sequence alignment

To find conserved regions between two sequences that not necessarily similar overall.

A situation that global alignment is not appropriate.

```
...........xxxxxxxxxx
xxxxxxxxxx...........
```

where `xxxxxxxxxx` is a conserved motif.

Also see Figure 6.16

Chapter 6. Dynamic Programming

A *local alignment* between two sequences

$v = v_1 \dots v_n$ and $w = w_1 \dots w_m$

Chapter 6. Dynamic Programming

A *local alignment* between two sequences

$v = v_1 \dots v_n$ and $w = w_1 \dots w_m$

is a global alignment between two substrings

$v_a \dots v_b$ and $w_c \dots w_d$, of v and w respectively,

that achieves the best alignment score among all such indexes a, b, c, d, $1 \leq a \leq b \leq n$ and $1 \leq c \leq d \leq m$.

Chapter 6. Dynamic Programming

Local sequence alignment

Find the best local alignment between two strings.

Chapter 6. Dynamic Programming

Local sequence alignment

Find the best local alignment between two strings.

Input: Strings v and w and a scoring matrix δ,
Output: Substrings of v and w whose global alignment, as defined by δ, is the maximum among all global alignments of all substrings of v and w.

Chapter 6. Dynamic Programming

How to solve this 'seemingly the same problem' as global alignment?

Chapter 6. Dynamic Programming

How to solve this 'seemingly the same problem' as global alignment?

(1) The global alignment algorithm actually computes all "semi-global alignments".

Chapter 6. Dynamic Programming

How to solve this 'seemingly the same problem' as global alignment?

(1) The global alignment algorithm actually computes all "semi-global alignments".

That is, it computes all best alignment scores for prefix substrings $v_1 \ldots v_i$ and $w_1 \ldots w_j$ for all indexes i, j.

Chapter 6. Dynamic Programming

(2) For local alignment, we would also like to drop any alignment 'head'
    that has been penalized.

```
    v1.....vh.........vk.....vi
    w1.....wp.........wq.....wj
   ->| neg |<--
      score
```

Chapter 6. Dynamic Programming

(2) For local alignment, we would also like to drop any alignment 'head'

that has been penalized.

```
 v1.....vh.........vk.....vi
 w1.....wp.........wq.....wj
->| neg |<--
   score
```

which can be achieved by following recurrence:

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ 0 \end{cases}$$

Chapter 6. Dynamic Programming

(3) For local alignment, we would like to drop any alignment 'tail' that has a negative score.

```
v1............vk....vi
w1............wq....wj
              ->| neg |<--
                score
```

Chapter 6. Dynamic Programming

(3) For local alignment, we would like to drop any alignment 'tail' that has a negative score.

```
v1.............vk....vi
w1.............wq....wj
               ->| neg |<--
                  score
```

This can be accomplished by tracing the DP table from the highest value cell.

Chapter 6. Dynamic Programming

6.10 Multiple alignment

Chapter 6. Dynamic Programming

6.10 Multiple alignment

To compare more than two sequences.

How to score an alignment (a column) involving k sequences ?

Chapter 6. Dynamic Programming

6.10 Multiple alignment

To compare more than two sequences.

How to score an alignment (a column) involving k sequences ?

Option-1: assume a k-dimensional scoring matrix. (unrealistic)

Chapter 6. Dynamic Programming

6.10 Multiple alignment

To compare more than two sequences.

How to score an alignment (a column) involving k sequences ?

Option-1: assume a k-dimensional scoring matrix. (unrealistic)

Option-2: *Sum-of-Pair* (SP) scoring scheme:
the score of a multiple alignment is computed as the
sum of the scores between every pair of the aligned sequences.

Chapter 6. Dynamic Programming

6.10 Multiple alignment

To compare more than two sequences.

How to score an alignment (a column) involving k sequences ?

Option-1: assume a k-dimensional scoring matrix. (unrealistic)

Option-2: *Sum-of-Pair* (SP) scoring scheme:
the score of a multiple alignment is computed as the
sum of the scores between every pair of the aligned sequences.

Option-3: entropy approach:
the score of a multiple alignment is computed as the
sum of entropies of all aligned columns.

The entropy for a column i is computed as

$$\sum_{x \in \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}} f_x^i \log f_x^i$$

where f_x^i is the frequency of residue x in column i.

Chapter 6. Dynamic Programming

A dynamic programming solution for multiple alignment.

Chapter 6. Dynamic Programming

A dynamic programming solution for multiple alignment.

Assume we have 3 sequences u, v, w to align.
Consider aligning prefixes:

$u_1 \dots u_i$
$v_1 \dots v_j$
$w_1 \dots w_k$

Chapter 6. Dynamic Programming

A dynamic programming solution for multiple alignment.

Assume we have 3 sequences u, v, w to align.
Consider aligning prefixes:

$u_1 \dots u_i$
$v_1 \dots v_j$
$w_1 \dots w_k$

$$s_{i,j,k} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(u_i, v_j, w_k) \\ s_{i,j-1,k-1} + \delta(-, v_j, w_k) \\ s_{i-1,j,k-1} + \delta(u_i, -, w_k) \\ s_{i-1,j-1,k} + \delta(u_i, v_j, -) \\ s_{i,j,k-1} + \delta(-, -, w_k) \\ s_{i,j-1,k} + \delta(-, v_j, -) \\ s_{i-1,j,k} + \delta(u_i, -, -) \end{cases}$$

Chapter 6. Dynamic Programming

A dynamic programming solution for multiple alignment.

Assume we have 3 sequences u, v, w to align.
Consider aligning prefixes:

$u_1 \ldots u_i$
$v_1 \ldots v_j$
$w_1 \ldots w_k$

$$s_{i,j,k} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(u_i, v_j, w_k) \\ s_{i,j-1,k-1} + \delta(-, v_j, w_k) \\ s_{i-1,j,k-1} + \delta(u_i, -, w_k) \\ s_{i-1,j-1,k} + \delta(u_i, v_j, -) \\ s_{i,j,k-1} + \delta(-, -, w_k) \\ s_{i,j-1,k} + \delta(-, v_j, -) \\ s_{i-1,j,k} + \delta(u_i, -, -) \end{cases}$$

Inefficient: $7 \times n^3$ time. In general,

Chapter 6. Dynamic Programming

A dynamic programming solution for multiple alignment.

Assume we have 3 sequences u, v, w to align.
Consider aligning prefixes:

$u_1 \dots u_i$
$v_1 \dots v_j$
$w_1 \dots w_k$

$$s_{i,j,k} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(u_i, v_j, w_k) \\ s_{i,j-1,k-1} + \delta(-, v_j, w_k) \\ s_{i-1,j,k-1} + \delta(u_i, -, w_k) \\ s_{i-1,j-1,k} + \delta(u_i, v_j, -) \\ s_{i,j,k-1} + \delta(-, -, w_k) \\ s_{i,j-1,k} + \delta(-, v_j, -) \\ s_{i-1,j,k} + \delta(u_i, -, -) \end{cases}$$

Inefficient: $7 \times n^3$ time. In general,

$O(n^m 2^{m-1})$ time and $O(n^m)$ size table for m sequences of length n.

Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

Typically **progressive** approaches:

Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

Typically **progressive** approaches:

- Note that use a collection of pairwise alignments may not work

Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

Typically **progressive** approaches:

- Note that use a collection of pairwise alignments may not work

- Find a small set of ‘core’ sequences and align other to them

Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

Typically **progressive** approaches:

- Note that use a collection of pairwise alignments may not work

- Find a small set of ‘core’ sequences and align other to them

  E.g., *center-star* algorithm
  E.g., CLUSTAL

the “once gap, forever gap” strategy

Chapter 6. Dynamic Programming

$6.10\frac{1}{2}$ HMM and Dynamic Programming Solutions

A Markov Model characterizes stochastic processes that assume following Markov property.

> The “oblivious” property, i.e, the conditional probability distribution of future states of a stochastic process depends only upon the present state, not on the sequence of events that preceded it.

Chapter 6. Dynamic Programming

A Markov Model over alphabet Σ consists of a set S of states and transitions T between states, such that

Chapter 6. Dynamic Programming

A Markov Model over alphabet Σ consists of a set S of states and transitions T between states, such that

(1) With a probability distribution, each state can emit symbols in Σ;

$$\text{for every } s \in S, \sum_{x \in \Sigma} p(s, x) = 1$$

Chapter 6. Dynamic Programming

A Markov Model over alphabet Σ consists of a set S of states and transitions T between states, such that

(1) With a probability distribution, each state can emit symbols in Σ;

$$\text{for every } s \in S, \sum_{x \in \Sigma} p(s,x) = 1$$

(2) With a probability distribution, there are transitions from each state to all other states in the model;

$$\text{for every } s \in S, \sum_{t \in S} q(s,t) = 1$$

Chapter 6. Dynamic Programming

Definitions:

Chapter 6. Dynamic Programming

Definitions:

Let M be a Markov model over alphabet Σ.
Let $X = x_1 \ldots x_n$ be a sequence over Σ, i.e., $x_i \in \Sigma$.

Chapter 6. Dynamic Programming

Definitions:

Let M be a Markov model over alphabet Σ.
Let $X = x_1 \dots x_n$ be a sequence over Σ, i.e., $x_i \in \Sigma$.

Let $\pi = s_1 \dots s_n$ be a sequence of states taken from S, i.e., $s_i \in S$ (π is called a path).

Chapter 6. Dynamic Programming

Definitions:

Let M be a Markov model over alphabet Σ.
Let $X = x_1 \dots x_n$ be a sequence over Σ, i.e., $x_i \in \Sigma$.

Let $\pi = s_1 \dots s_n$ be a sequence of states taken from S, i.e., $s_i \in S$ (π is called a path).

Then the probability for M to generate symbol sequence X with the path π is

$$p(X, \pi | M)$$

$$= p(s_1, x_1)q(s_1, s_2)p(s_2, x_2) \dots p(s_{n-1}, x_{n-1})q(s_{n-1}, s_n)p(s_n, x_n)$$

$$= \prod_{k=1}^{n-1} p(s_k, x_k)q(s_k, s_{k+1}) \times p(s_n, x_n)$$

Chapter 6. Dynamic Programming

Uses of HMMs

1. Modeling specific classes of sequences

   e.g., profile HMM for motifs

Chapter 6. Dynamic Programming

Uses of HMMs

1. Modeling specific classes of sequences

   e.g., profile HMM for motifs

2. Modeling general classes of sequences

   typically for prediction

Chapter 6. Dynamic Programming

Fundamental algorithms with HMMs

1. decoding (for prediction, discrimination)

Chapter 6. Dynamic Programming

Fundamental algorithms with HMMs

1. decoding (for prediction, discrimination)
2. computing likelihood (for model fitness)

Chapter 6. Dynamic Programming

Fundamental algorithms with HMMs

1. decoding (for prediction, discrimination)
2. computing likelihood (for model fitness)
3. learning (for building models)

Chapter 6. Dynamic Programming

HMM Decoding Problem:

Chapter 6. Dynamic Programming

HMM DECODING PROBLEM:

Given a Markov model M and a sequence X over the alpha Σ, find the path π^* such that

$$p(X, \pi^*|M) \text{ achieves the maximum.}$$

Chapter 6. Dynamic Programming

HMM DECODING PROBLEM:

Given a Markov model M and a sequence X over the alpha Σ, find the path π^* such that

$$p(X, \pi^* | M) \text{ achieves the maximum.}$$

That is to find an optimal path π^* such that

$$\pi^* = \arg\max_{\pi} \{p(X, \pi | M)\}$$

Chapter 6. Dynamic Programming

Dynamic programming to compute π^* such that

$p(X, \pi^*|M)$ achieves the maximum.

Chapter 6. Dynamic Programming

Dynamic programming to compute π^* such that

$p(X, \pi^*|M)$ achieves the maximum.

If we consider the model M to be a “generic” sequence
then the task is to find the “best alignment” between X and M

Chapter 6. Dynamic Programming

Dynamic programming to compute π^* such that

$p(X, \pi^* | M)$ achieves the maximum.

If we consider the model M to be a "generic" sequence
then the task is to find the "best alignment" between X and M

That is: to "align" x_i on X with state s_j

Chapter 6. Dynamic Programming

Dynamic programming to compute π^* such that

$p(X, \pi^* | M)$ achieves the maximum.

If we consider the model M to be a "generic" sequence
then the task is to find the "best alignment" between X and M

That is: to "align" x_i on X with state s_j

The "score" is $p(s_j, x_i)$ for this "column".

Chapter 6. Dynamic Programming

Consider a more general algorithm VITERBI ALGORITHM:

Computing the maximum probability for M to generate prefix $x_1 x_2 \dots x_i$

such that symbol x_i is emitted by state s_j.

Chapter 6. Dynamic Programming

Consider a more general algorithm Viterbi Algorithm:

Computing the maximum probability for M to generate prefix $x_1 x_2 \dots x_i$

such that symbol x_i is emitted by state s_j.

We define such probability to be $V_{i,j}$

Chapter 6. Dynamic Programming

Consider a more general algorithm VITERBI ALGORITHM:

Computing the maximum probability for M to generate prefix $x_1x_2 \dots x_i$

such that symbol x_i is emitted by state s_j.

We define such probability to be $V_{i,j}$

Then we have the following recurrence:

$$V_{i,j} = \max_{s_k \in S}\{V_{i-1,k} \times q(s_k, s_j) \times p(s_j, x_i)\}$$

Chapter 6. Dynamic Programming

Consider a more general algorithm VITERBI ALGORITHM:

Computing the maximum probability for M to generate prefix $x_1 x_2 \dots x_i$

such that symbol x_i is emitted by state s_j.

We define such probability to be $V_{i,j}$

Then we have the following recurrence:

$$V_{i,j} = \max_{s_k \in S} \{V_{i-1,k} \times q(s_k, s_j) \times p(s_j, x_i)\}$$

If stochastic processes always begin from state s_b, then:

$$V_{1,b} = 1 \times p(s_b, x_1)$$

$$V_{1,a} = 0 \text{ for every } a \neq b$$

Chapter 6. Dynamic Programming

dynamic programming to compute $G_{i,j}$
consisting 4 steps:

Chapter 6. Dynamic Programming

dynamic programming to compute $G_{i,j}$
consisting 4 steps:

1. problem analysis
2. objective function and recurrence formulation
3. iterative algorithm implementation
4. tracing back the solution (path)

Chapter 6. Dynamic Programming

Use an HMM to construct a sequence profile

called a *profile-HMM*

Chapter 6. Dynamic Programming

Use an HMM to construct a sequence profile

called a *profile-HMM*

example:

```
                CGGGGCTatccagctGGGTCGTCACATTCCCCTTTCGATA
     TTTGAGGGTGCCCAATAAgggcaactCCAAAGCGGACAAA
                     GGatggatctGATGCCGTTTGACGACCTAAATCAACGGCC
                   AAGGaagcaaccCCAGGAGCGCCTTTGCTGGTTCTACCTG
CTAAAAGATTATAATGTCGGTCCttggaactTC
       CTGTACATCATGCTGCatgccattTTCAAC
         TACATGATCTTTTGatggcactTGGATGATGAGGGAATGC
                       ________
```

Chapter 6. Dynamic Programming

8 columns for match

Chapter 6. Dynamic Programming

8 columns for match

What about insertions and deletions?

Chapter 6. Dynamic Programming

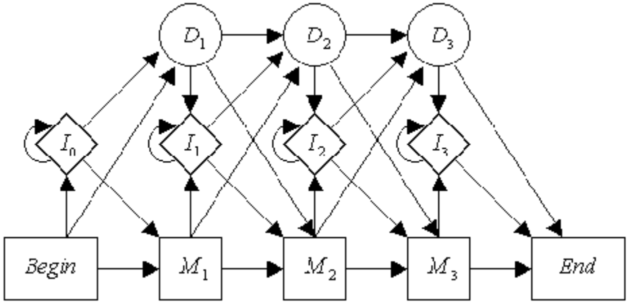
8 columns for match

What about insertions and deletions?

Profile-HMM definition:

A profile-HMM consists of the following states:
- begin and end states
- match states M_j, $j = 1, 2, \ldots, m$
- insert states I_j, $j = 1, 2, \ldots, m$
- delete states D_j, $j = 1, 2, \ldots, m$

Chapter 6. Dynamic Programming



Chapter 6. Dynamic Programming

Viterbi algorithm is still usable for computing π^* but needs to be revised a little.

Chapter 6. Dynamic Programming

Viterbi algorithm is still usable for computing π^*
but needs to be revised a little.

Define:

$V_{i,j}^{M}$ is the optimal probability the HMM produces
prefix $x_1 \dots x_i$ ending at state M_j.

Chapter 6. Dynamic Programming

Viterbi algorithm is still usable for computing π^* but needs to be revised a little.

Define:

$V^M_{i,j}$ is the optimal probability the HMM produces prefix $x_1 \ldots x_i$ ending at state M_j.

$V^I_{i,j}$ is the optimal probability the HMM produces prefix $x_1 \ldots x_i$ ending at state I_j.

Chapter 6. Dynamic Programming

Viterbi algorithm is still usable for computing π^* but needs to be revised a little.

Define:

$V^M_{i,j}$ is the optimal probability the HMM produces prefix $x_1 \dots x_i$ ending at state M_j.

$V^I_{i,j}$ is the optimal probability the HMM produces prefix $x_1 \dots x_i$ ending at state I_j.

$D^I_{i,j}$ is the optimal probability the HMM produces prefix $x_1 \dots x_i$ ending at state D_j.

Chapter 6. Dynamic Programming

Recurrences for $V^M_{i,j}$:

Chapter 6. Dynamic Programming

Recurrences for $V_{i,j}^M$:

$$V_{i,j}^M = \max\{$$
$$V_{i-1,j-1}^M q(M_{j-1}, M_j) p(M_j, x_i),$$
$$V_{i-1,j-1}^I q(I_{j-1}, M_j) p(M_j, x_i),$$
$$V_{i-1,j-1}^D q(D_{j-1}, M_j) p(M_j, x_i)$$
$$\}$$

Chapter 6. Dynamic Programming

Recurrences for $V_{i,j}^M$:

$$
\begin{aligned}
V_{i,j}^M = \max\{ & \\
& V_{i-1,j-1}^M q(M_{j-1}, M_j) p(M_j, x_i), \\
& V_{i-1,j-1}^I q(I_{j-1}, M_j) p(M_j, x_i), \\
& V_{i-1,j-1}^D q(D_{j-1}, M_j) p(M_j, x_i) \\
\} &
\end{aligned}
$$

Base cases?

Chapter 6. Dynamic Programming

Recurrences for $V_{i,j}^M$:

$$V_{i,j}^M = \max\{ \quad V_{i-1,j-1}^M q(M_{j-1}, M_j) p(M_j, x_i), \quad V_{i-1,j-1}^I q(I_{j-1}, M_j) p(M_j, x_i), \quad V_{i-1,j-1}^D q(D_{j-1}, M_j) p(M_j, x_i) \quad \}$$

Base cases?

And recurrences for V^I, V^D?

Chapter 6. Dynamic Programming

To set up a profile-HMM:

- obtain a multiple alignment of training data;
- determine the number of match states;
- compute emission prob distribution for every match state;
- determine the number insert states;
- compute emission prob distribution for every insert state;
- determine the number of delete states;
- determine the transition probability distributions;

Chapter 6. Dynamic Programming

To set up a profile-HMM:

- obtain a multiple alignment of training data;
- determine the number of match states;
- compute emission prob distribution for every match state;
- determine the number insert states;
- compute emission prob distribution for every insert state;
- determine the number of delete states;
- determine the transition probability distributions;

Resolve the over fitting issue with pseudo-counts

Chapter 6. Dynamic Programming

Example of profile-HMM:

```
atccag-ct
gggcaa-ct
atggat-ct
a-gcaatcc
ttggaa-ct
atgcca-tt
atggca-ct
```

Chapter 6. Dynamic Programming

Example of profile-HMM:

```
atccag-ct
gggcaa-ct
atggat-ct
a-gcaatcc
ttggaa-ct
atgcca-tt
atggca-ct
```

1. how to determine match columns and insert columns (for consensus)
2. deletion is w.r.t. match
3. when there is no pseudo counts
4. to avoid over fiting

Chapter 6. Dynamic Programming

Using a profile HMM (of a family of sequences) to search for new members on genomes/databases:

Chapter 6. Dynamic Programming

Using a profile HMM (of a family of sequences) to search for new members on genomes/databases:

- construct a profile-HMM
- develop Viterbi algorithm
- choose a scanning window size
- post-process results

Chapter 6. Dynamic Programming

6.11 Gene Prediction

Identification of protein coding genes in genome sequences.

Statistical approaches

based on statistical features surrounding genes

Chapter 6. Dynamic Programming

6.11 Gene Prediction

Identification of protein coding genes in genome sequences.

- Statistical approaches
  - based on statistical features surrounding genes
- Similarity-based approaches
  - based on similarities of genes across different species