Special course in Computer Science: Advanced Text Algorithms

Lecture 6: Alignments

Eugen Czeizler
Department of IT, Abo Akademi
http://combio.abo.fi/teaching/textalg/

(slides originally by I. Petre, E. Czeizler, V. Rogojin)

Approximate matching

- The exact matching is not always appropriate in pattern-matching applications.

- Sometimes, it is more important to find objects that are "close enough" to a given pattern, i.e., they match the pattern in a reasonably "approximate" way.

- Here, by "approximate" we mean that certain errors (that will be specified later on) are allowed.

Approximate matching and alignments

- Approximate matching and sequence comparison are central tools in computational molecular biology.

- This is due to, e.g.,:
  - The presence of errors in molecular data.
  - The existence of active mutational processes that can modify the sequences.

- Alignment = a lining up of the characters of two strings, that allows mismatches as well as matches, and allows characters of one string to be placed opposite spaces in the other string.
  - ACTCGCCTGATGGG
  - ACAC_CCACAT__G

How is sequence alignment useful?

- "Good" alignment between two sequences may imply that the sequences are similar.

- There are many sequences with unknown function or structure. At the same time there are also some sequences with known function and structure.

- If one sequence has known structure/function, then an alignment with another sequence may yield insights into the other's structure/function.

Sequence alignments

- Some typical questions answered by sequence alignment:
  - Is my newly sequenced gene reported somewhere?
  - Two sequence yield two proteins with similar functions – are there elements of their sequences shared?
  - If a protein A has known 3D structure and a protein B has unknown folding, could we align A with B to get a rough estimate of its structure?
  - If we have the sequences of the same protein from two organisms – how closely related are the organisms?
  - If a set of sequences (the same protein in several organisms), have good pairwise alignments, could we get an optimal alignment for ALL sequence (multiple alignment)?

Usefulness of sequence alignments

- Protein cytochromec has almost the same length in most organisms that produce it
  - one expects to see a relationship between their sequences in different organisms.
  - Same is true for proteins in the globulin family
  - Useful in deducing the evolutionary history

- Comparing two protein sequences, local alignment is useful in detecting structural or functional subunits such as motifs or domains

Usefulness of sequence alignments

- Comparing two DNA sequences and aiming to find similar segments is useful –they may yield insight into genetic parts.

- Example: the homeobox genes regulate high-level embryonic development in many organisms from fruit-flies to pigs to humans
  - A single mutation can transform one body part into another (a mutation experiment made fruit flies develop antennas as legs)
  - The protein sequences are of course very different with one exception: the homeodomain (about 60 aminoacids) are extremely similar in insects and mammalians –this is very odd because this is part of a crucial regulatory protein that binds to DNA

The Edit Distance Problem

- Sometimes one would like to measure the difference, or distance between two strings:
  - in evolutionary, structural or functional studies of biological molecules
  - in textual database retrievals
  - in spelling correction methods

- One way to formalize the distance between two strings S_1 and S_2, called the edit distance, focuses on transforming (or editing) one string into the other, by a series of edit operations on individual characters.

The Edit Distance Problem

- The permitted edit operations:
  - insertion of a character into the first string, denoted by I
  - deletion of a character from the first string, denoted by D
  - the substitution of a character from the first string with a character from the second string, denoted by R (replace)
  - the match between one character from the first string and one character from the second string, denoted by M.

- Observation: The roles of S_1 and S_2 are symmetric, since a deletion in one string corresponds to an insertion in the other.

The Edit Distance Problem

- An edit transcript is a sequence of operations I, D, R and M that transforms S_1 into S_2

- Example: An edit transcript that transforms the sequence “vintner” into “writers”:

```
RIMDMDMMI
V_intner
Wri_t_ers
```

String Alignment

- The edit transcript is a way of representing a particular transformation of a string S_1 into another string S_2

- Alternatively, we can use alignments.

- A (global) alignment *of* S_1 *and* S_2 *is* obtained by inserting spaces in the strings, and then placing them one above the other s.t. each char or space is opposite a unique char or space from the other string
  - *"global" ~ the entire strings participate in the alignment*
  - *local alignments ~ regions of high similarity*

- Example: A global alignment of "vintner" and "writers":

```
V_INTNER_
WRI_T_ERS
```

Definition of the Edit Distance

- The edit distance between strings S_1 and S_2 is the minimum number of operations I, D, and R in any transcript that transforms S_1 into S_2.
  - the operation M is not considered here.

- An edit transcript with the smallest number of operations I, D and R is an optimal transcript.

- Edit distance problem: Compute the edit distance and an optimal transcript for the given strings S_1 and S_2.

Computing the Edit Distance

- Given two strings $S_1[1..n]$ and $S_2[1..m]$ we can compute the edit distance (and the associated edit transcript) by using dynamic programming.

- Define D(i,j) to be the edit distance of prefixes $S_1[1..i]$ and $S_2[1..j]$.

- Thus, D(n,m) is the edit distance of S_1 and S_2

- We will compute D(n,m) by using the standard dynamic programming approach, i.e., by computing D(i,j) for all $0 \le i \le n$ and $0 \le j \le m$

Components of dynamic programming

- The dynamic programming approach has three essential components:

- Recurrence relation: How can we compute D(i,j) knowing only the values D(i',j') with $i' \leq i$ and $j' \leq j$?

- Tabular computation: How to store efficiently the computed values in order to avoid computing them over and over again?

- Traceback: How to find an optimal edit transcript after we have computed the edit distance?

The recurrence relation

- We establish a recursive relationship between the value D(i,j) with i,j≥0 and values of D with index pairs smaller than i,j.

- Base conditions for the edit distance problem:
  - D(i,0)=i since $S_1[1..i]$ can only be transformed into $S_2[1..0]=\varepsilon$ using i deletions.
  - D(0,j)=j since $S_1[1..0]=\varepsilon$ can only be transformed into $S_2[1..j]$ by inserting all j characters of $S_2[1..j]$.

- The recurrence relation for D(i,j) with i, j > 0:

$$D(i,j)=\min\begin{cases} D(i-1,j)+1 & \text{(D)} \\ D(i,j-1)+1 & \text{(I)} \\ D(i-1,j-1)+t(i,j) & \text{(MR)} \end{cases}$$

where $t(i,j)=\begin{cases} 0, & S_1[i]=S_2[j] \\ 1, & \text{otherwise} \end{cases}$

The correctness of the edit distance recurrence

- What is the meaning of the 3 cases (D), (I) and (MR) from the recurrence formula?

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 & \text{(D)} \\ D(i,j-1)+1 & \text{(I)} \\ D(i-1,j-1)+t(i,j) & \text{(MR)} \end{cases}$$

- A transcript that transforms $S_1[1..i]$ to $S_2[1..j]$ either
  - transforms $S_1[1..\ i\text{-}1]$ into $S_2[1..j]$ and deletes $S_1[i]$ (D)
  - transforms $S_1[1..\ i]$ into $S_2[1..j\text{-}1]$ and inserts $S_2[j]$, (I) or
  - transforms $S_1[1..\ i\text{-}1]$ into $S_2[1..j\text{-}1]$ and then matches or replaces $S_1[i]$ by $S_2[j]$ (MR).

The correctness of the edit distance recurrence

- An optimal transcript is one that minimizes over the three possibilities

- NB: More than one of the above cases may give the same minimum: there may be many co-optimal transcripts

Tabular computation

- It is easy to implement the recurrence formula D(i,j) as a recursive procedure.

- The problem is that a recursive procedure for D(i,j) computes the same values many times

- But there are only (n+1)(m+1) different D(i,j) values (since $0 \le i \le n$, $0 \le j \le m$)

- So, we could compute them in a suitable order, and store them in an array so that each value is computed only once.

Tabular computation

- We build the table D(i,j), with $0 \le i \le n$ and $0 \le j \le m$, in an increasing order of pairs (i,j).

- First, we initialize column 0 and row 0 according to the base cases of the recurrence relation:

```
for i := 0 to n do D(i,0) = i;
for j := 0 to m do D(0,j) = j;
```

Initialization of the dynamic programming table

- After initializing row 0 and column 0, we obtain the following table D(i,j):

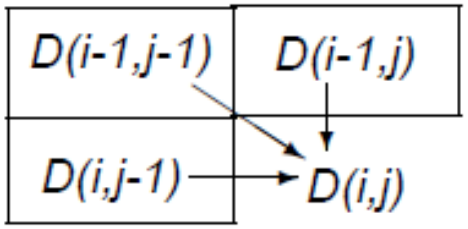
$D(i,j)$	S_2:		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
V	1	1							
i	2	2							
n	3	3							
t	4	4							
n	5	5							
e	6	6							
r	7	7							

- Then, we start computing the remaining cells of the table.

Computing the inner cells

- The values of the inner cells D(i,j) (i,j > 0) can be computed in any order as long as the three values required by the recurrence relation have been computed:

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 & \text{(D)} \\ D(i,j-1)+1 & \text{(I)} \\ D(i-1,j-1)+t(i,j) & \text{(MR)} \end{cases}$$



- For example, in a row-first order:

```
for i := 1 to n do // Compute row i
for j := 1 to m do // and column j of row i
    Compute D(i,j) according to the recurrence relation;
```

Example of row-first computation

- After computing the values in rows 1–3 we obtain the following D(i,j) table:

$D(i,j)$	S_2:		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
V	1	1	1	2	3	4	5	6	7
i	2	2	2	2	2	3	4	5	6
n	3	3	3	3	3	3	4	5	6
t	4	4							
n	5	5							
e	6	6							
r	7	7							

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 & \text{(D)} \\ D(i,j-1)+1 & \text{(I)} \\ D(i-1,j-1)+t(i,j) & \text{(MR)} \end{cases}$$

with

$$t(i,j) = \begin{cases} 0, & S_1(i) = S_2(j) \\ 1, & \text{otherwise} \end{cases}$$

- Let us compute now $D(4,1) = \min\{D(3,1)+1, D(4,0)+1, D(3,0)+1\}$

$$= \min\{4,5,4\} = 4$$

Example of row-first computation

- Let us compute next $D(4,2) = \min\{D(3,2)+1, D(4,1)+1, D(3,1)+1\}$
$= \min\{4,5,4\} = 4$

$D(i,j)$	S_2:		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
V	1	1	1	2	3	4	5	6	7
i	2	2	2	2	2	3	4	5	6
n	3	3	3	3	3	3	4	5	6
t	4	4	4						
n	5	5							
e	6	6							
r	7	7							

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 & \text{(D)} \\ D(i,j-1)+1 & \text{(I)} \\ D(i-1,j-1)+t(i,j) & \text{(MR)} \end{cases}$$

with

$$t(i,j) = \begin{cases} 0, & S_1(i) = S_2(j) \\ 1, & \text{otherwise} \end{cases}$$

Complexity of the tabulation

- When computing the value of a given cell (i,j) we:
  - look at the values in cells (i-1,j), (i,j-1) and (i-1,j-1)
  - look at the characters $S_1[i]$ and $S_2[j]$.

- Each of the (n+1)(m+1) cells is filled in constant time:

- Theorem: The edit distance D(n,m) of strings $S_1[1..n]$ and $S_2[1..m]$ can be computed in O(nm) time.

Traceback

- How can we compute the optimal edit transcripts?

- Establish some pointers in the table when we compute the values D(i,j):
  - Set a pointer from cell (i,j) to (i-1,j) if D(i,j)=D(i-1,j)+1
  - Set a pointer from cell (i,j) to (i,j-1) if D(i,j)=D(i,j-1)+1
  - Set a pointer from cell (i,j) to (i-1,j-1) if D(i,j)=D(i-1,j-1)+t(i,j)

- In particular:
  - pointers on row 0 point to the cell on the left, and
  - pointers on column 0 to the cell above

Example of traceback pointers

	S_2:		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	←1	←2	←3	←4	←5	←6	←7
V	1	↑1	↖1	↖←2	↖←3	↖←4	↖←5	↖←6	↖←7
i	2	↑2	↖↑2	↖2	↖2	←3	←4	←5	←6
n	3	↑3	↖↑3	↖↑3	↖↑3	↖3	↖←4	↖←5	↖←6
t	4	↑4	↖↑4	↖↑4	↖↑4	↖3	↖←4	↖←5	↖←6
n	5	↑5	↖↑5	↖↑5	↖↑5	↑4	↖4	↖←5	↖←6
e	6	↑6	↖↑6	↖↑6	↖↑6	↑5	↖4	↖←5	↖←6
r	7	↑7	↖↑7	↖6	↖←↑7	↑6	↑5	↖4	←5

Finding optimal transcripts

- An optimal transcript can be found by following the traceback pointers from cell (n,m) to (0,0):

1. pointer '←' from (i,j) to (i,j-1): insertion of $S_2[j]$

2. pointer '↑' from (i,j) to (i-1,j): deletion of $S_1[i]$

3. pointer '↖' from (i,j) to (i-1, j-1): match/replace, depending on whether $S_1[i]$ and $S_2[j]$ match or not

Example of optimal transcripts

	S_2:		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	←1	←2	←3	←4	←5	←6	←7
V	1	↑1	↖1	↖←2	↖←3	↖←4	↖←5	↖←6	↖←7
i	2	↑2	↖↑2	↖2	↖2	←3	←4	←5	←6
n	3	↑3	↖↑3	↖↑3	↖↑3	↖3	↖←4	↖←5	↖←6
t	4	↑4	↖↑4	↖↑4	↖↑4	↖3	↖←4	↖←5	↖←6
n	5	↑5	↖↑5	↖↑5	↖↑5	↑4	↖4	↖←5	↖←6
e	6	↑6	↖↑6	↖↑6	↖↑6	↑5	↖4	↖←5	↖←6
r	7	↑7	↖↑7	↖6	↖←↑7	↑6	↑5	↖4	←5

- We obtain the following optimal edit transcripts specified by pointers of the preceding table:
  1. R, I, M, D, M, D, M, M, I
  2. I, R, M, D, M, D, M, M, I
  3. R, R, R, M, D, M, M, I

Finding optimal alignments

- Alternatively, we can also find an optimal alignment by following the traceback pointers from cell (n,m) to (0,0):

1. pointer '←' from (i,j) to (i,j-1): space in S_1 opposite to $S_2[j]$, and
2. pointer '↑' from (i,j) to (i-1,j): space in S_2 opposite to $S_1[i]$
3. pointer '↖' from (i,j) to (i-1, j-1): $S_1[i]$ and $S_2[j]$ are aligned

Example of optimal alignments

	S_2:		w	r	i	t	e	r	s
S_1		0	1	2	3	4	5	6	7
	0	0	←1	←2	←3	←4	←5	←6	←7
V	1	↑1	↖1	↖←2	↖←3	↖←4	↖←5	↖←6	↖←7
i	2	↑2	↖↑2	↖2	↖2	←3	←4	←5	←6
n	3	↑3	↖↑3	↖↑3	↖↑3	↖3	↖←4	↖←5	↖←6
t	4	↑4	↖↑4	↖↑4	↖↑4	↖3	↖←4	↖←5	↖←6
n	5	↑5	↖↑5	↖↑5	↖↑5	↑4	↖4	↖←5	↖←6
e	6	↑6	↖↑6	↖↑6	↖↑6	↑5	↖4	↖←5	↖←6
r	7	↑7	↖↑7	↖6	↖←↑7	↑6	↑5	↖4	←5

1. _ V i n t n e r _
   w r i _ t _ e r s

2. V _ i n t n e r _
   w r i _ t _ e r s

3. V i n t n e r _
   w r i t _ e r s

Complexity of tracing an optimal transcript

- Theorem: After computing the dynamic programming table with pointers, an optimal transcript (or an optimal alignment, resp.) can be found in time O(n + m).

Proof.

- Starting from cell (n,m), a path leading to cell (0,0) can be followed by choosing any of the pointers.

- Each cell (i, j)≠(0,0) has at least one pointer to one of (i-1,j), (i,j-1) and (i-1,j-1).

- Thus, we need to follow at most n+m pointers.

Generalizations

- We can also consider edit operations with weights (or costs or scores): d for deletion/insertion, r for substitution, and e for match.

- The operation-weight edit distance problem is to find a transcript that transforms S_1 into S_2 with minimum total weight.

- Edit distance is a special case with d = r = 1 and e = 0

- Example: An alignment with r = 2, d = 4 and e = 1:

```
          v i n t n e r _
          w r i t _ e r s
weight = 2+2+2+1+4+1+1+4 = 17
```

Computing operation-weight edit distance

- Operation weights cause straight-forward modifications to the recurrences:

- Base cases with operation weights:
  - $D(i,0) = i \times d$
  - $D(0,j) = j \times d$

(representing the number of characters that are deleted/inserted)

Computing operation-weight edit distance (2)

- Inductive case for i,j > 0, with operation weights:

$$D(i,j) = \min \begin{cases} D(i-1,j) + d & \text{(D)} \\ D(i,j-1) + d & \text{(I)} \\ D(i-1,j-1) + t(i,j) & \text{(MR)} \end{cases}$$

where $t(i,j) = \begin{cases} e, & S_1[i] = S_2[j] \\ r, & \text{otherwise} \end{cases}$

- With these modifications, edit distance with operation weights and the corresponding transcript/alignment can be computed in time O(nm), exactly as before

Another generalization

- The cost of edit operations can also be allowed to depend on which characters of the alphabet are involved (alphabet-weight edit distance)

- Which are used, alphabet or operation weights?

- DNA strings more often compared applying operation-weight or unweighted costs.

- Proteins are most often compared using alphabet-weights over the amino-acid alphabet.

String Similarity

- An alternative formalization for the relatedness of strings is their similarity (rather than distance),
  - The similarity between strings is used in most bio-applications
- Let Σ′ be the alphabet Σ extended with the space '_'
- Denote the score of aligning chars x and y of Σ′ by s(x,y)
- Let $S'_1[1..l]$ and $S'_2[1..l]$ be the equal-length versions of strings S_1 and S_2, padded with spaces, for an alignment $\mathcal{A}$
- Example: Consider the following alignment for the sequences S_1=cacdbd and S_2=cabbdb:

  S'_1 : c a c _ d b d

  S'_2 : c a b b d b _

Value on an alignment

- The value (or total score) of alignment $\mathcal{A}$ is then

$$\sum_{i=1}^{l} s(S_1'[i], S_2'[i])$$

- Example: Consider the following score matrix for Σ′={a,b,c,d,_}:
  - Often s(x,y)≥0 iff x = y

s	a	b	c	d	_
a	1	-1	-2	0	-1
b		3	-2	-1	0
c			0	-4	-2
d				3	-1
–					0

The value of the previous alignment between S_1=cacdbd and S_2=cabbdb:

S'_1 : c a c _ d b d

S'_2 : c a b b d b _

0+1-2+0+3+3-1 = 4

Scoring an alignment

- The scoring matrix depends on the type of application one is interested in
  - there are no "universal" scoring schemes

- A popular scoring scheme when comparing DNA sequences is the following:
  - match of two nucleotides: score +1
  - mismatch of two nucleotides: score -1
  - gap (align a nucleotide with a space): score -2

	A	C	G	T	-
A	1	-1	-1	-1	-2
C		1	-1	-1	-2
G			1	-1	-2
T				1	-2
-					NAN

Defining string similarity

- The similarity of strings S_1 and S_2 (determined by a scoring matrix), is the value of an alignment of S_1 and S_2 that maximizes the total alignment score. This is also called the optimal alignment value.

- Similarity of strings S_1 and S_2 can be computed as a straight-forward modification of edit distance.

- For this, define V(i,j) to be the optimal alignment value of prefixes $S_1[1..i]$ and $S_2[1..j]$

Recurrences for String Similarity

- Base conditions are rather obvious (Ass. s(_,_)≤0):

$$V(0, j) = \sum_{k=1}^{j} s(\_, S_2(k))$$

$$V(i, 0) = \sum_{k=1}^{j} s(S_1(k), \_)$$

- They give the total score of aligning chars with spaces.

Computing String Similarity

- The general recurrence for i,j > 0 similarly takes the character-specific scores into account

$$V(i,j) = \max \begin{cases} V(i-1,j) + s(S_1(\mathrm{i}),\_) \\ V(i,j-1) + s(\_,S_2(\mathrm{j})) \\ V(i-1,j-1) + s(S_1(\mathrm{i}),S_2(\mathrm{j})) \end{cases}$$

- Applying these formulas, the similarity between two strings can be computed like the edit distance, with V(n,m) at the lower right corner of the table

- The similarity and an optimal alignment of $S_1[1..n]$ and $S_2[1..m]$ can be computed in time O(nm)

Special cases of similarity

- Depending on how we chose the scoring scheme, several problems can be modeled as special cases of optimal alignment or similarity, e.g., the longest common subsequence problem.

- A subsequence of a string S[1..n] of length n is specified by a list of indices $i_1<i_2<..<i_k$, for some $k \leq n$. The subsequence specified by this list of indices is $S[i_1]S[i_2]..S[i_k]$.
  - Example: abc is a subsequence of baccbac specified by the list of indices 2<5<7

- Problem: Find the longest common subsequence of two strings S_1 and S_2

The longest common subsequence problem

- Let us take the following scoring scheme
  - match of two characters: score +1
  - mismatch of two characters: score 0
  - gap (align a character with a space): score 0

- Then, it is quite immediate that with the above scoring scheme the matched characters in an alignment of maximum value form a longest common subsequence.

- A longest common subsequence for 2 given strings can be computed (by dynamic programming approach) in O(nm) time.

Substring vs subsequences

- The longest common substring of two sequences can be found in O(n+m) time, using suffix trees.

- The longest common subsequence for two given strings can be computed in O(nm) time.