Special course in Computer Science: Advanced Text Algorithms

Lecture 7: Approximate pattern matching, local alignments and alignments with gaps

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

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

TUCS

Approximate Pattern Matching

- Important generalization of exact matching: locate *similar* occurrences of a pattern instead of exact copies.
- Given a parameter k, a substring T' of T is an approximate occurrence of P iff the optimal alignment between P and T' is at least k
- Approximate occurrences of a pattern P within a text T can be computed as a slight variation of (global) alignment.
    - we will use a recurrence formula very similar with the one for computing a global alignment between two strings
    - dynamic programming approach.

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. Moreover, a space in one string cannot be aligned with a space in 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
```

Recurrence relation

- Let Σ′ be the alphabet Σ extended with the space '_'
- Denote by s(x,y) the score of aligning chars x and y of Σ′
- Base conditions give the total score of aligning chars with spaces:

$$V(0, j) = 0$$

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

- The base condition for row 0 implies that T[1]; T[2];...; T[j] are aligned with spaces "for free", i.e., "it doesn't cost (or pay) to slide P along T"

Recurrence relation

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

$$V(i,j) = \max \begin{cases} V(i-1,j) + s(P[\mathrm{i}],\_) \\ V(i,j-1) + s(\_,T[\mathrm{j}]) \\ V(i-1,j-1) + s(P[\mathrm{i}],T[\mathrm{j}]) \end{cases}$$

TUCS

Recurrence relation

2. Align P[i] and T[j]
    - Find the best alignment of P[1…i-1] and T[1…j-1]
    - The score of aligning P[1…i] and T[1…j] would then be V(i-1,j-1) + s(P[i],T[j])

3. Align P[i] with a gap
    - Find the best alignment of P[1…i-1] and T[1…j]
    - The score would then be V(i-1,j)+s(P[i],_)

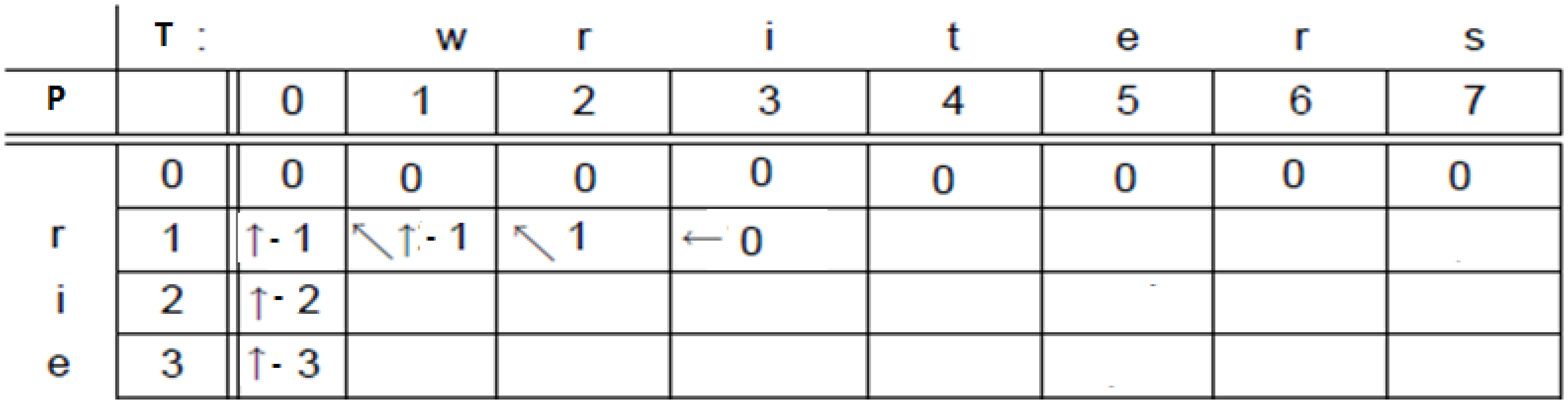
4. Align T[j] with a gap
    - Find the best alignment of P[1…i] and T[1…j-1]
    - The score would then be V(i,j-1)+s(_,T[j])

- Table V (i,j) can be filled, as before, in time $\Theta(nm)$

Example

- Take the following score scheme:
  - any match: score 1
  - any mismatch: score -1
  - any gap: score -1

And the two strings: P=rie and T=writer



	T :		w	r	i	t	e	r	s
P		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
r	1	↑-1	↖↑-1	↖1	←0				
i	2	↑-2							
e	3	↑-3							

Finding Approximate Occurrences

- An approximate occurrence of P[1..n] ends at position j within T if and only if V(n,j)≥k

- T[l..j] is an approximate occurrence of P in T if and only if V(n,j)≥k and there is a path of backpointers from cell (n,j) to cell (0,l)

Finding Approximate Occurrences

- There can be multiple approximate occurrences of P (of different length) ending at the same position j of T.

- The shortest approximate occurrences of a pattern P in the text T can be located as follows:

1. Find each column j on row n with $V(n,j) \geq k$

2. For each such j, trace pointers from the cell (n,j) to row 0, preferring pointers '↑' over '↖', and '↖' over '←'
   - This way we explicitly output only the shortest approximate occurrences of P within the text T.

Significance of backpointers

1. pointer '↑' from (i,j) to (i-1,j): space in T opposite to P[i]

2. pointer '↖' from (i,j) to (i-1, j-1): P[i] and T[j] are aligned

3. pointer '←' from (i,j) to (i,j-1): space in P opposite to T[j], and

Local alignments

- Sometimes although 2 strings are not highly similar, they may contain regions that are highly similar.
    - Thus, we are interested in finding a pair of substrings, one from each of the 2 strings that exhibit high similarity.

- Local alignment (or local similarity) problem: Given strings S_1 and S_2, find substrings α and β of S_1 and S_2 of maximal similarity among all pairs of substrings from S_1 and S_2. Let v^* denote the value of the optimal solution.

- In Lecture 6 we discussed two measures for the relatedness of two strings:
    - the edit distance
    - the similarity

Local alignments

- Question: Why is the local alignment defined in terms of similarity instead of edit distance?

- If we would search for 2 substrings minimizing the edit distance, then, under most natural scoring schemes, the optimal pair would be exactly matching substrings.
  - The matching substrings could be 1 character long
  - They would not identify a region of high similarity.

- Maximizing similarity is thus more useful for finding longer areas of high similarity.
  - Matches contribute positively
  - Mismatches contribute negatively

Local alignments

- Example: Let S_1=pqraxabcstvq and S_2=xyaxbacsll and we take the following scoring scheme:
    - each match: score +2
    - each mismatch: score -2
    - each gap, i.e. aligning a space with a character: score -1.

Then, the substrings α=axabcs of S_1 and β=axbacs of S_2 have the following optimal alignment of value 8.

```
a x a b _ c s
a x _ b a c s
```

Over all choices of pairs of substrings, one from S_1 and one from S_2, the 2 substrings α and β have maximum similarity.

Hence, for this example v*=8 and is defined by α=axabcs and β=axbacs

Local vs. Global Alignment?

- Global alignment is often meaningful when comparing members of the same protein family
  - Protein cytochrome c has almost the same length in most organisms that produce it, so one expects to see a relationship between their sequences in different organisms.
  - Same is true for proteins in the globulin family

- When trying to infer evolutionary history by examining protein sequence similarities and differences, one usually compares proteins in the same familly.

Local vs. Global Alignment?

- Local alignment considered more useful for comparing anonymous DNA sequences (where only some internal sections may be related)

- When comparing two protein sequences, local alignment is useful in detecting structural or functional subunits such as motifs or domains
  - the homeobox genes regulate high-level embryonic development in many organisms from fruit-flies to pigs to humans
  - The protein sequences are of course very different with one exception: the homeodomain (about 60 aminoacids) is extremely similar in insects and mammalians –this is very odd because this is part of a crucial regulatory protein that binds to DNA

Computing Local Alignment

- The local alignment problem btw strings $S_1[1..n]$ and $S_2[1..m]$ can be solved in $O(nm)$ time
    - even though there are $\Theta(n^2m^2)$ possible pairs of substrings!

- In the definition of local alignments (given earlier) any scoring scheme was allowed for the global alignment of the two chosen substrings.
    - The following restriction will be useful for computing the local alignment.

- Assume first that the similarity of two empty strings is 0
    - This allows the local alignment algorithm to chose the substrings α and β to be empty.

TUCS

Computing Local Alignment

- Consider the following restricted version of the problem:

Given indices $i \leq n$ and $j \leq m$, the *local suffix alignment problem* consists of finding a (possibly empty) suffix α of $S_1[1..i]$ and a (possibly empty) suffix β of $S_2[1..j]$ of maximal similarity.

We denote by $v(i,j)$ the value of the optimal local suffix alignment for the index pair (i,j)

Example of Local Suffix Alignments

- Example: Take the following score scheme:
  - s(x,y) = 2 when x=y≠_,
  - s(x,y) = -1 when x≠y, for any x,y$\in\Sigma\cup\{_\}$

Consider strings S_1=abcfdef and S_2=fffcde.

Then:

- v(3,4)=2, since $\alpha=\beta$=c
- v(4,5)=1, since α=cf and β=cd
- v(5,5)=3, since α=f_d and β=fcd

	1	2	3	4	5	6	7
S_1 :	a	b	c	f	d	e	f
S_2 :	f	f	f	c	d	e	

- Since the definition allows either or both suffixes to be empty, v(i,j)≥0

Computing Local Alignment

- For each common substring u of sequences S_1 and S_2, there are i and j such that u is at the same time a suffix of $S_1[1…i]$ and of $S_2[1…j]$

- For each i=0,1,…,n and j=0,1,…,m, v(i,j) is the best score between a suffix of $S_1[1…i]$ and a suffix of $S_2[1…j]$.

- Then the highest value in matrix v will give the most similar substrings of sequences S_1 and S_2
  - That is $v^*=\max\{v(i,j) \mid i \leq n, j \leq m\}$

Local suffix alignment problem

- How does one align a suffix $S_1[1…i]$ with a suffix $S_2[1…j]$ in the best way?

- If i=0 or j=0 then the best is to align two empty strings with score 0.

- The base conditions: $v(i,0)=0$
  $v(0,j)=0$

- Otherwise, there are four options when computing the best alignment for a suffix of $S_1[1…i]$ and a suffix of $S_2[1…j]$, depending on how the endings are aligned

Local suffix alignment problem

1. Take the two suffixes to be the empty strings
   - this gives score 0

2. Align $S_1[i]$ and $S_2[j]$
   - Find the best alignment of $S_1[1\ldots i-1]$ and $S_2[1\ldots j-1]$
   - The score of aligning $S_1[1\ldots i]$ and $S_2[1\ldots j]$ would then be $v(i-1,j-1) + s(S_1[i], S_2[j])$

3. Align $S_1[i]$ with a gap
   - Find the best alignment of $S_1[1\ldots i-1]$ and $S_2[1\ldots j]$
   - The score would then be $v(i-1,j)+s(S_1[i],\_)$

4. Align $S_2[j]$ with a gap
   - Find the best alignment of $S_1[1\ldots i]$ and $S_2[1\ldots j-1]$
   - The score would then be $v(i,j-1)+s(\_,S_2[j])$

Local suffix alignment problem

- Choose that option which maximizes the alignment score:

$$v(\mathrm{i}, \mathrm{j}) = \max \begin{cases} 0 \\ v(\mathrm{i}-1, \mathrm{j}-1) + s(S_1[\mathrm{i}], S_2[\mathrm{j}]) \\ v(\mathrm{i}-1, \mathrm{j}) + s(S_1[\mathrm{i}], \_) \\ v(\mathrm{i}, \mathrm{j}-1) + s(\_, S_2[\mathrm{j}]) \end{cases}$$

- A table storing the v(i,j) values, including the backpointers, can be computed applying the recurrences, in a similar way as before

TUCS

Local suffix alignment problem

- The value v(i,j) stores the highest score between a suffix of $S_1[1\ldots i]$ and a suffix of $S_2[1\ldots j]$
  - There is always the alignment between the empty suffixes of $S_1[1\ldots i]$ and $S_2[1\ldots j]$ with cost 0
  - We introduce 0 in the maximum formula in the previous slide: alignment of score 0 is always guaranteed (align empty suffixes) –we look for anything better
  - The matrix will only have nonnegative values

Local alignment

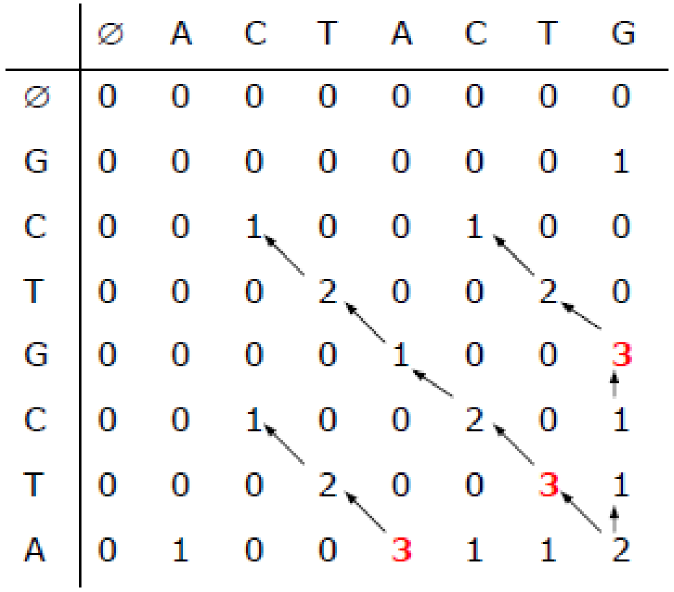
- Solving the local suffix alignment gives also the score of the best local alignment: the largest value in the matrix v.

- Question: How does one find that best local alignment?

- Answer: “walk” from the highest value in the matrix following the arrows until the first zero is reached

Local alignment –example

- Locally compare sequences ACTACTG and GCTGCTA
- Scoring scheme:
  - Match: score +1
  - Mismatch: score -1
  - Gap: score -1



	∅	A	C	T	A	C	T	G
∅	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	1
C	0	0	1	0	0	1	0	0
T	0	0	0	2	0	0	2	0
G	0	0	0	0	1	0	0	3
C	0	0	1	0	0	2	0	1
T	0	0	0	2	0	0	3	1
A	0	1	0	0	3	1	1	2

```
   ACTACTG        ACTACTG        ACTACTG
GCTGCTA           GCTGCTA           GCTGCTA
```

Complexity of Local Alignment

- Maximum value v* is found say, in cell v(i*,j*), by going through *all cells of the* table. Substrings α and β with similarity v* are then found by tracing backpointers from cell (i*,j*) along a path (i*,j*);... ;(i',j'); (i_0,j_0), where $v(i_0,j_0)=0$

- Then $\alpha=S_1[i'..\ i^*]$ and $\beta=S_2[j'..j^*]$

- Theorem: Local alignment between strings $S_1[1..n]$ and $S_2[1..m]$ can be computed in time O(nm)

Proof.

- Table v(i,j) is filled in constant time per cell
- The cell (i*,j*) with an optimal score is found in time O(nm), and the traceback for (i',j') requires at most n + m steps

Remarks

- Instead of a single highest-scoring pair (α,β) of substrings, a number of similar substrings, say with similarity above a given threshold, can be found in a similar manner.

- Suitable scoring schemes are needed for meaningful local alignments:
  - scoring matches with 1 and mismatches/spaces with 0 locates *longest common subsequences*
  - penalizing mismatches/spaces with large negative values yields *longest common substrings*
  - scoring matrices with a positive average score tend to prefer long alignments, which approach *global alignments*

Alignments with Gaps

- A gap *is a maximal consecutive run of spaces in a single string participating in an alignment*

- In some cases alignments with gaps correspond better to the biological phenomena that we try to model, e.g., the likelihood of mutational events needed to transform one sequence into the other
    - a deletion or an insertion of an entire (DNA) substring (i.e., a gap) often occurs as a single mutational event
    - gaps are sometimes key features for inferring evolutionary history of a set of strings

Alignments with Gaps

- Example of an alignment with gaps:

```
c t t t a a c _ _ a _ a c
c _ _ _ c a c c c a t _ c
```

This alignment includes 5 matches, 1 mismatch, 4 gaps, and 7 spaces.

- By including a term in the objective function to reflect the gaps in the alignment, we can influence the distribution of spaces in the alignment.

How to Score the Gaps?

- Different possibilities to score the gaps of an alignment:
    - constant, affine, convex, and arbitrary

- A constant gap weight is the simplest:

- Set s(_,x) = s(x,_) = 0 for every char x, and score each gap by constant W_g (independent of gap length)

- Then the score of an alignment is:

$$\sum_{i=1}^{l} s(S'_1[i], S'_2[i]) - kW_g$$

where S'_1 and S'_2 are the strings padded with spaces for the alignment, and k is the total number of gaps

How to Score the Gaps?

- Changing the value W_g relative to the other weights can change how spaces are distributed in the optimal alignment.

- For instance, large W_g encourages the alignment to have just a few gaps, and the aligned portions of the 2 strings will fall into a few substrings.

Affine Gap Weights

- Generalization of constant gap weight: Treat W_g as a *gap initiation weight, and* add a *gap extension weight W_s for each space*
    - a gap of length k adds cost W_g+k W_s to the score (which is an "affine" function)

- Affine gap weights are probably the most commonly used ones in molecular biology
    - Default weights of FASTA are W_g= 10 and W_s= 2

- Optimal alignments under this model maximize

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

with scores s(_,x) = s(x,_) =-W_s for each x, and k is the number of gaps in the alignment

Convex Gap Weights

- It seems that some biological phenomena are better modeled by a gap weight function where each additional space in a gap contributes less to the gap weight than the preceding space.

- An example of such a convex *gap weight* where additional spaces cost less than earlier ones is W_g+log l for gaps of length l

- Finally, we may have also arbitrary gap weights, where the weight of a gap is an arbitrary function w(l) of its length.

Time Bounds for Different Gap Weights

- Optimal alignments can be found in the following times:

1. $O(nm^2+n^2m)$ for arbitrary gap weights
2. $O(nm\log m)$ for convex gap weights
3. $O(nm)$ for affine and constant gap weights

- We will discuss the first and the third case in details (The algorithm for convex gap weights is more complicated)

Computing Arbitrary Gap-Weight Alignments

- Consider an optimal alignment between the prefixes $S_1[1..i]$ of S_1 and $S_2[1..j]$ of S_2;

- It can either
  - align $S_1[i]$ to the left of $S_2[j]$ (case E)
  - align $S_2[j]$ to the left of $S_1[i]$, or (case F)
  - align $S_1[i]$ against $S_2[j]$ (case G)

- Let E(i,j) be the maximum value of alignments of type E, and respectively F(i,j) and G(i,j) the maximum values of alignments of type F and G.

- The maximum value V(i,j) of any alignment between $S_1[1..i]$ and $S_2[1..j]$ is then max{E(i,j),F(i,j),G(i,j)}

Recurrences for Arbitrary Gap Weights

- Let w(l) be the weight of a gap of length l
- Base cases:

$$V(i,0) = -w(i), \quad V(0,j) = -w(j)$$
$$E(i,0) = -w(i), \quad F(0,j) = -w(j)$$
$$G(0,0) = 0, \text{ and G is undefined if only one of i or j is 0}$$

(as the cost of aligning a non-empty string with a gap)

- Recurrence formulas for the different cases for i; j > 0 are as follows:

$$E(i,j) = \max\{V(i,k) - w(j-k) \mid 0 \le k \le j-1\}$$
$$F(i,j) = \max\{V(l,j) - w(i-l) \mid 0 \le l \le i-1\}$$
$$G(i,j) = V(i-1,j-1) + s(S_1[i], S_2[j])$$

Complexity

- The optimal alignment value V(n,m) can be computed by filling an (n + 1)(m + 1) table V(i,j) according to the recurrences

- Theorem: The similarity of $S_1[1..n]$ and $S_2[1..m]$ under arbitrary gap weights can be computed in time $O(nm^2+n^2m)$

Proof.

- Each E(i,j) is computed by examining j cells of table V, leading to $\Sigma_{1\leq j\leq m} j = O(m^2)$ for computing a single row and $O(nm^2)$ for all E(i,j)
- Similarly, each F(i,j) is computed from i cells of table V, leading to $O(mn^2)$ time to compute all values F(i,j)
- In addition to that, each of V(i,j) and G(i,j) are assigned in constant time

Affine Gap Weights

- Optimal alignments with *affine gap weights can be* computed more efficiently

- The reason is that the cost of extending a gap of length l by one space is now predictable:

$$w(l+1) = W_g + W_s \times (l+1) = w(l) + W_s$$

- All that matters is whether a new gap is started (with initiation weight W_g) or whether it has already begun

- This insight is formalized in the recurrences for cases E and F (using variables V(i,j) , E(i,j) , F(i,j) and G(i,j) in similar roles as before)

Recurrences for Affine Gap Weights

- Base cases:

$$V(i,0) = E(i,0) = -W_g - iW_s$$
$$V(0,j) = F(0,j) = -W_g - jW_s$$

(start a gap and make it i or j spaces long)

- For i; j > 0, V(i,j) = max{E(i,j) ; F(i,j) ;G(i,j)}, as above

- Case G of aligning S_1[i] with S_2[j] also remains the same:

$$G(i,j) = V(i-1, j-1) + s(S_1[i], S_2[j])$$

- What about cases E and F (either string ends with a gap)?

Affine Gap Weight Recurrences

- Consider the recurrence for E(i,j), where, by definition, $S_1[i]$ will be aligned with a character to the left of $S_2[j]$.

(a) If $S_1[i]$ is exactly one place to the left of $S_2[j]$, i.e., a gap begins in S_1 opposite character $S_2[j]$

$$E(i,j) = V(i,j-1) - W_g - W_s$$

(b) If $S_1[i]$ is to the left of $S_2[j-1]$, i.e., the same gap in S_1 is opposite both characters $S_2[j-1]$ and $S_2[j]$

$$E(i,j) = E(i,j-1) - W_s$$

Whichever the case, E(i,j) is by definition the maximum:

$$E(i,j) = \max\{E(i,j-1), V(i,j-1) - W_g\} - W_s$$

Affine Gap Weight Recurrences

- The explanations for F(i,j) go in a similar way and we obtain the following recurrence formula

$$F(i,j) = \max\{F(i-1,j), V(i-1,j) - W_g\} - W_s$$

- As before, the optimal alignment value is found in cell V(n,m)

Time Analysis

- Theorem: The similarity of strings $S_1[1..n]$ and $S_2[1..m]$ with affine gap weights can be computed in time $O(nm)$

Proof.

- The number of values V(i,j) , E(i,j) , F(i,j) , and G(i,j) is O(nm), and each of them is computed from a constant number of previously computed values

- NB: The above method computes also similarity with *constant gap weights, as a special case* $W_s = 0$