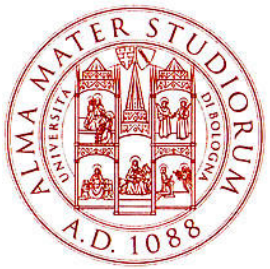
Algorithms and Data Structures 2012-2013

Lesson 9: some examples of algorithms for bioinformatics



Luciano Bononi

<bononi@cs.unibo.it>

http://www.cs.unibo.it/~bononi/

(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)

International Bologna Master in Bioinformatics

University of Bologna

29/05/2013, Bologna

Outline of the lesson

- **Graphs**
  - Dynamic programming techniques
  - Definition and concepts: ***dynamic programming****: It basically consists* of solving an instance of a problem by taking advantage of already computed solutions for smaller instances of the same problem.
  - Case study:
    - Sequence comparison

Comparing Sequences

- **EXERCISE:** similarity of sequences (two subproblems)
  - Provide a way to measure the similarity of sequences
    - Same characters in the same place
  - Aligment of sequences:
    - Move subsequences inserting blank spaces (to improve the similarity)

Comparing Sequences

Example of problems in bioinformatics:

- **1)** Given 2 sequences of the same alphabet, both about the same length (tens of thousands of characters). We know that the sequences are almost equal, with only a few isolated differences such as insertions, deletions, and substitutions of characters. The average frequency of these differences is low, say, one each hundred characters. We want to find the places where the differences occur.
- Problems like this appear when, for instance, the same gene is sequenced by two different labs and they want to compare the results; or even when the same long sequence is typed twice into the computer and we are looking for typing errors.

Comparing Sequences

- **Example of problems in bioinformatics:**

- **2) Given 2 sequences of the same alphabet,** with a few hundred characters each. We want to know whether there is a prefix of one which is similar to a suffix of the other. If the answer is yes, the prefix and the suffix involved must be produced.

Comparing Sequences

- **Example of problems in bioinformatics:**

- **3)** We have the same problem as in (2), but now we have several hundred sequences that must be compared (each one against all). In addition, we know that the great majority of sequence pairs are unrelated, that is, they will not have the required degree of similarity.
- Problems like (2) and (3) appear in the context of fragment assembly in programs to help large-scale DNA sequencing.

Comparing Sequences

- **Example of problems in bioinformatics:**

- **4)** We have two sequences over the same alphabet with a few hundred characters each. We want to know whether there are two substrings, one from each sequence, that are similar.

Comparing Sequences

- **Example of problems in bioinformatics:**

- **5)** We have the same problem as in (4), but instead of two sequences we have one sequence that must be compared to thousands of others.

- Problems like (4) and (5) occur in the context of searches for local similarities using large biosequence databases.

Result: a single basic algorithmic idea and a dynamic programming technique can be used to solve all of the above problems (1-5).

Global Comparison: the basic algorithm:

Consider the following DNA sequences:

GACGGATTAG and GATCGGAATAG.

Notice that they actually look very much alike, a fact that becomes more obvious when we align them one above the other as follows.

GA-CGGATTAG

GATCGGAATAG

Global Comparison: the basic algorithm:

Our goal in this section is to present an efficient algorithm that takes two sequences and determines the best alignment between them as we did above. Of course, we must define what the "best" alignment is before approaching the problem. To simplify the discussion, we will adopt a simple formalism; later, we will see possible generalizations.

Global Comparison: the basic algorithm:

- Aligment concept: Given 2 sequences:
  - they may have different size
    - We insert blank spaces (-) to make them same size
    - …and to improve their similarity (blanks can be included in both sequences, but not in the same position on both sequences)
  - If they have same size
    - Align them one above the other and see their similarity

Similarity score: providing a score function to the similarity concept

Global Comparison: the basic algorithm:

Similarity score function (example):

Given Ai and Bi the symbols in the sequences A, B, in position i,

Score(Ai,Bi) =+1 (match) if Ai=Bi

-1 (character mismatch) if Ai is not equal to Bi

-2 (blank mismatch) if Ai OR Bi is the blank char.

Choice ot the score values is important! More on this later.

BEST ALIGNMENT between A and B: the one providing the max total score.

= SIMILARITY sim (A, B) = max [Summatory_i (score(Ai,Bi)]

Global Comparison: the basic algorithm:

Example: given A and B as follows

A = GA-CGGATTAG

B = GATCGGAATAG

Sum[Score (Ai, Bi)] = [1+1-2+1+1+1+1-1+1+1+1] = +6

Similarity (A,B) = +6 (with this aligment)

… but how many possible alignment can be considered?

An exponential number! Which means that we have to enumerate (brute force) all of them before deciding which alignemnt provides the better similarity! (we need a better solution)

Global Comparison: the basic algorithm:

…we need a better solution: **Dynamic Programming!**

Main idea: Given two sequences *s and t, instead of* determining the similarity between *s and t as whole sequences only, we build up the solution* by determining all similarities between arbitrary *prefixes of the two sequences. We* start with the shorter prefixes and use previously computed results to solve the problem for larger prefixes.

Let's see an example to build the concept…

Global Comparison: the basic algorithm:

Dynamic Programming technique:

Given two strings s and t, let *m be the size of s and n the size of t. There are m+1 possible prefixes of s and n+1 prefixes of f, including the empty string. Thus, we can arrange our calculations in* an *(m + 1) x (n + 1) array where entry (z, j) contains the similarity between s[1..i] and t[1..jl.*

Example: s = AAAC and t = AGC.

Global Comparison: the basic algorithm:

Example: s = AAAC and t = AGC.

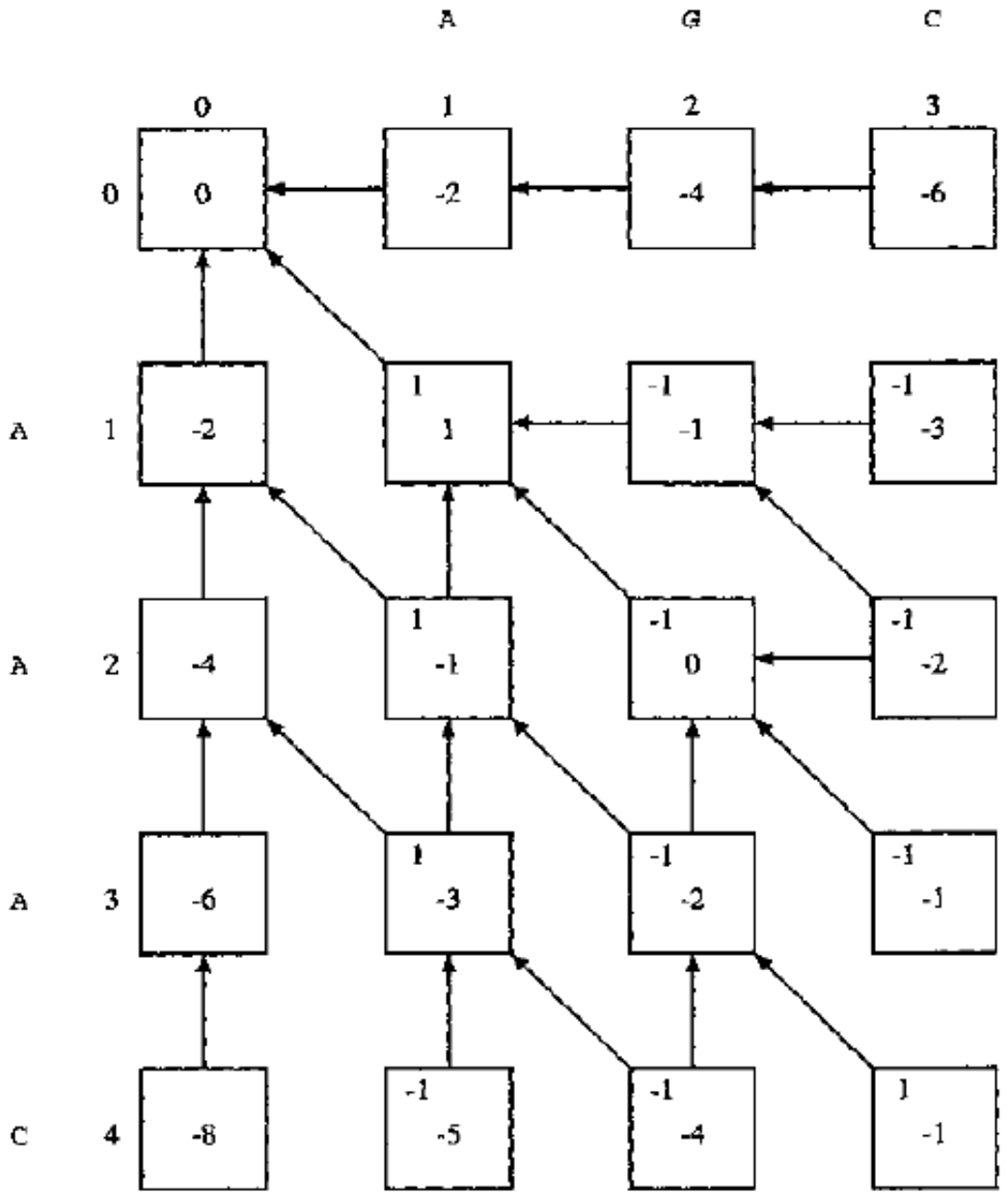
We create an array corresponding to *s = AAAC and t = AGC. We place s* along the left margin and *t along the top to indicate the prefixes more easily. Notice that* the first row and the first column are initialized with multiples of the space penalty (—2 in our case). This is because there is only one alignment possible if one of the sequences is empty: Just add as many spaces as there are characters in the other sequence. The score of this alignment is — *2k, where k is the length of the nonempty sequence. Hence, filling* the first row and column is easy.

Global Comparison

Example: s = AAAC t = AGC.

Now let us concentrate on the other entries. The key observation here is that we can compute the value for entry (i, *j) looking at just three previous entries: those for* (i - 1, j), (i - 1,j - 1) and (i, j - 1).

The reason is that there are just three ways of obtaining an alignment between s[1..i] and *t [1..j], and each one uses one of these* previous values

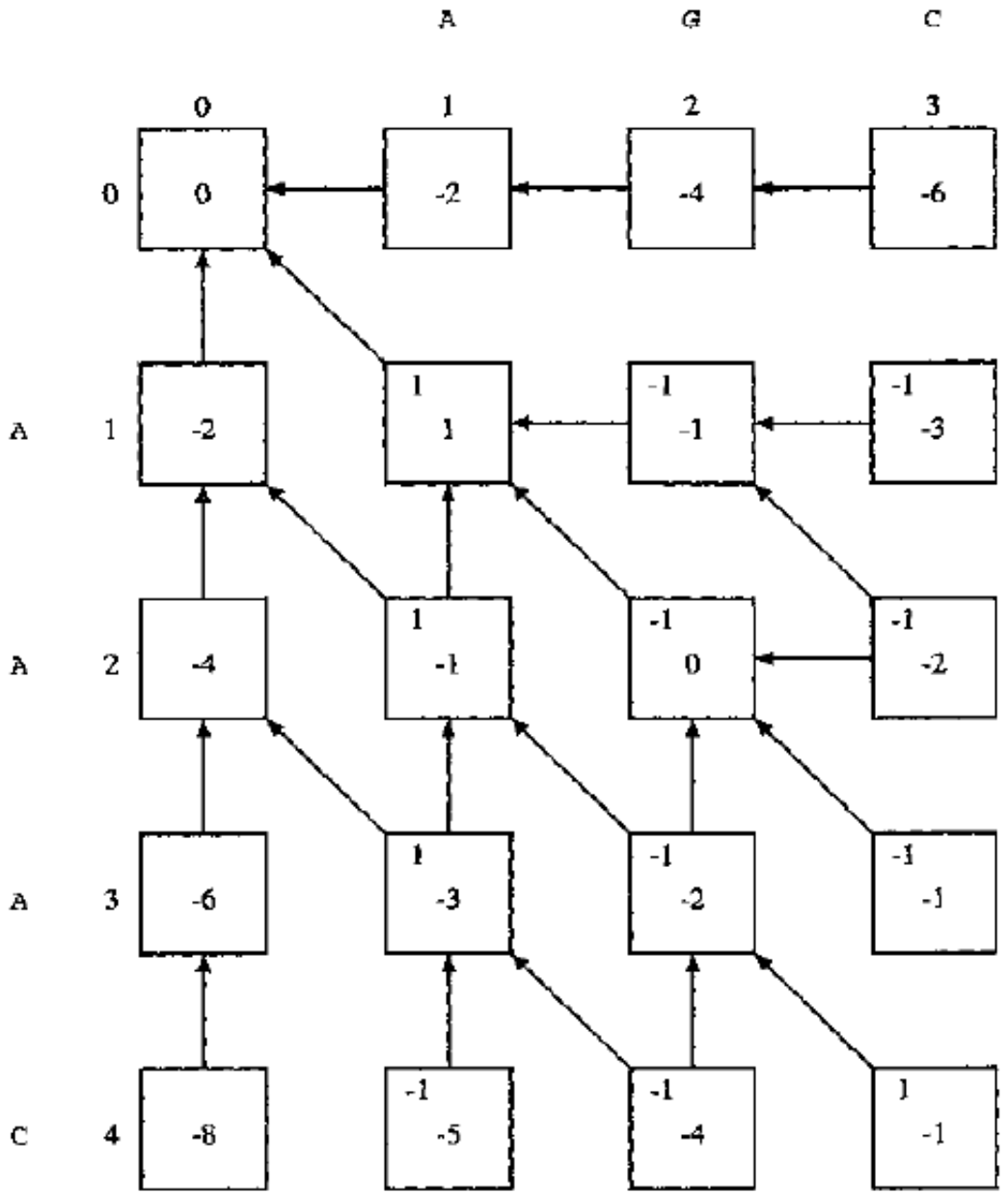


Global Comparison

Example: s = AAAC t = AGC.

- In fact, to get an alignment for s[1..i] and t [1.. j] , we have the following three choices:
- Align ,s[1..i] with *t[1..j — 1] and match a space with t[j], or*
- Align *s[1..i — 1] with t[1..j — 1] and match s[i] with t[j], or*
- Align *s[1..i - 1] with t[1..j] and match s[i] with a space.*

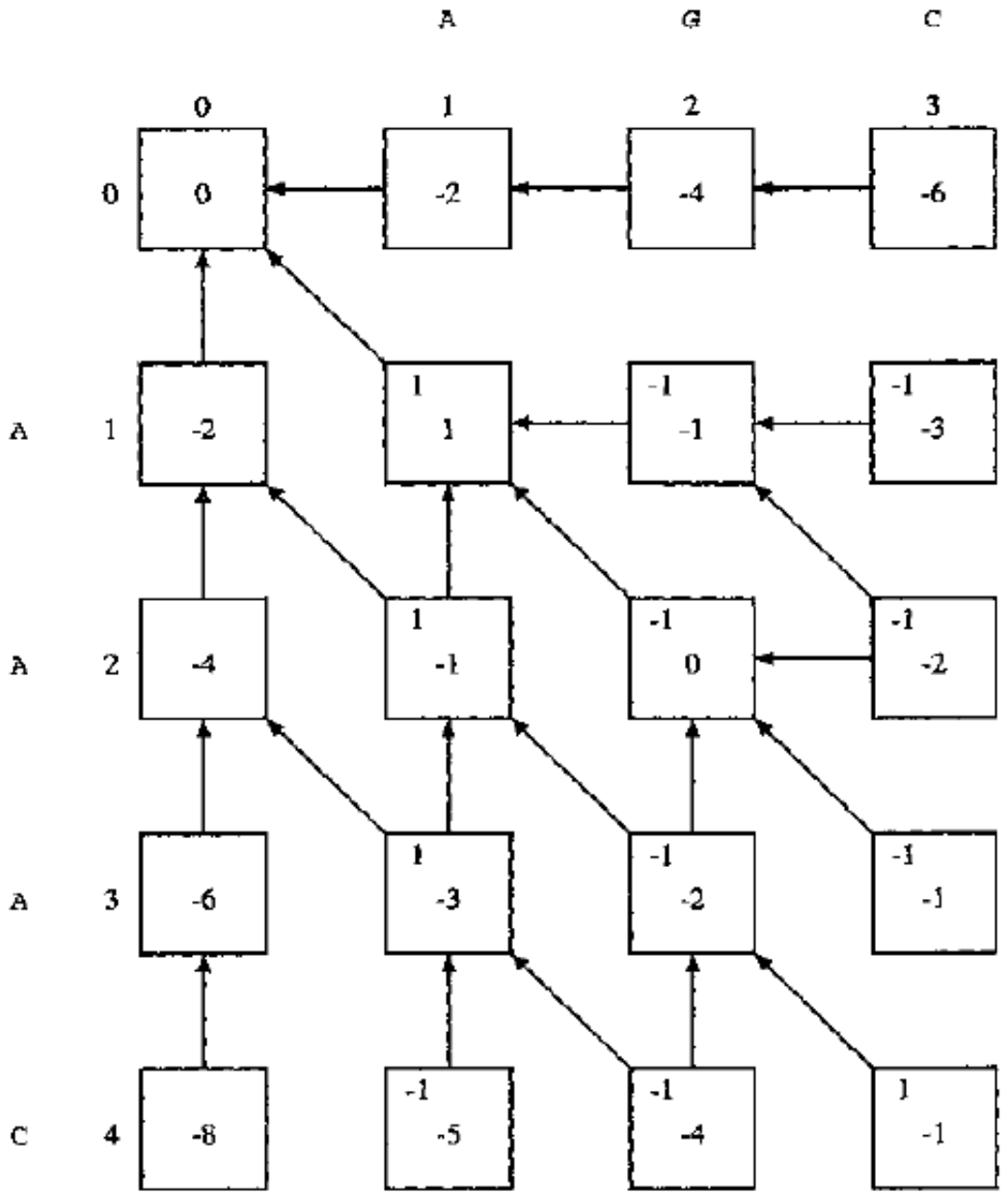
(N.B. we cannot have two spaces paired in the last column of the alignment*)*



Global Comparison

Example: s = AAAC t = AGC.

- **Dynamic programming concept**: Scores of the best alignments between smaller prefixes are already stored in the array if we choose an appropriate order in which to compute the entries. As a consequence, the similarity sought can be determined by a formula.
- *Given* p(i, *j) = +1 if s[i] = t[j] and —1 if s[i] <> t[j]… these values p(i, j) are written* in the upper left corners of the boxes



Global Comparison

- *Given* p(i, *j) = +1 if s[i] = t[j] and —1 if s[i] <> t[j]… these values p(i, j) are written* in the upper left corners of the boxes
- *The formula is:*

SIM(s[1..i],t[1..j]*) = max(*

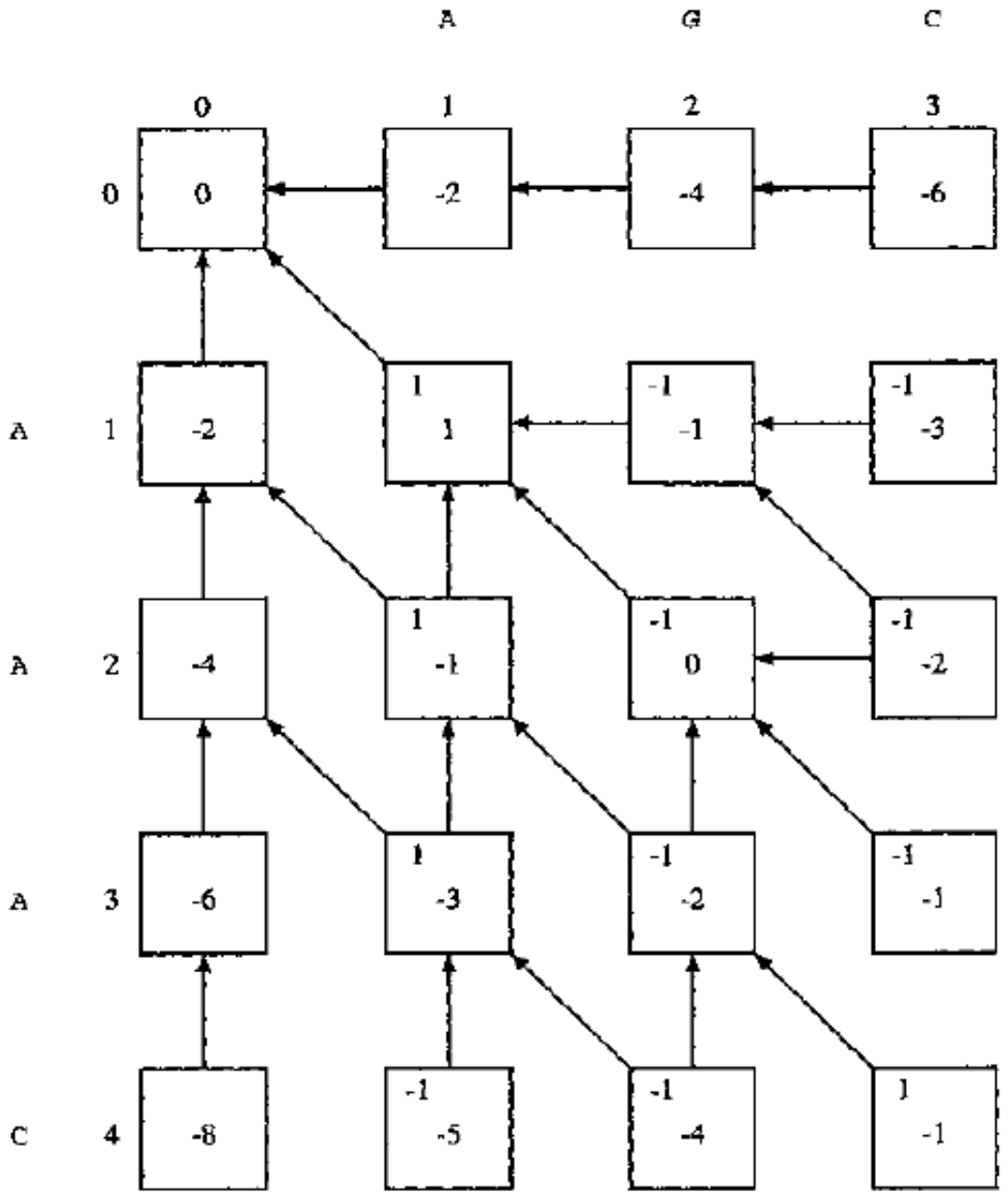
SIM(s[1..i],t[1..j-1])-2

SIM(s[1..i-1],t[1..j-1])+p(i,j)

SIM(s[1..i-1],t[1..j])-2

)

Any order that makes sure *a[i, j —1], a[i — 1, j — 1], and a[i — 1, j] are available when a[i, j] must be computed is fine.*



Global Comparison

- *If we denote the matrix as a[i,j], then every element a[i,j] can be computed as*

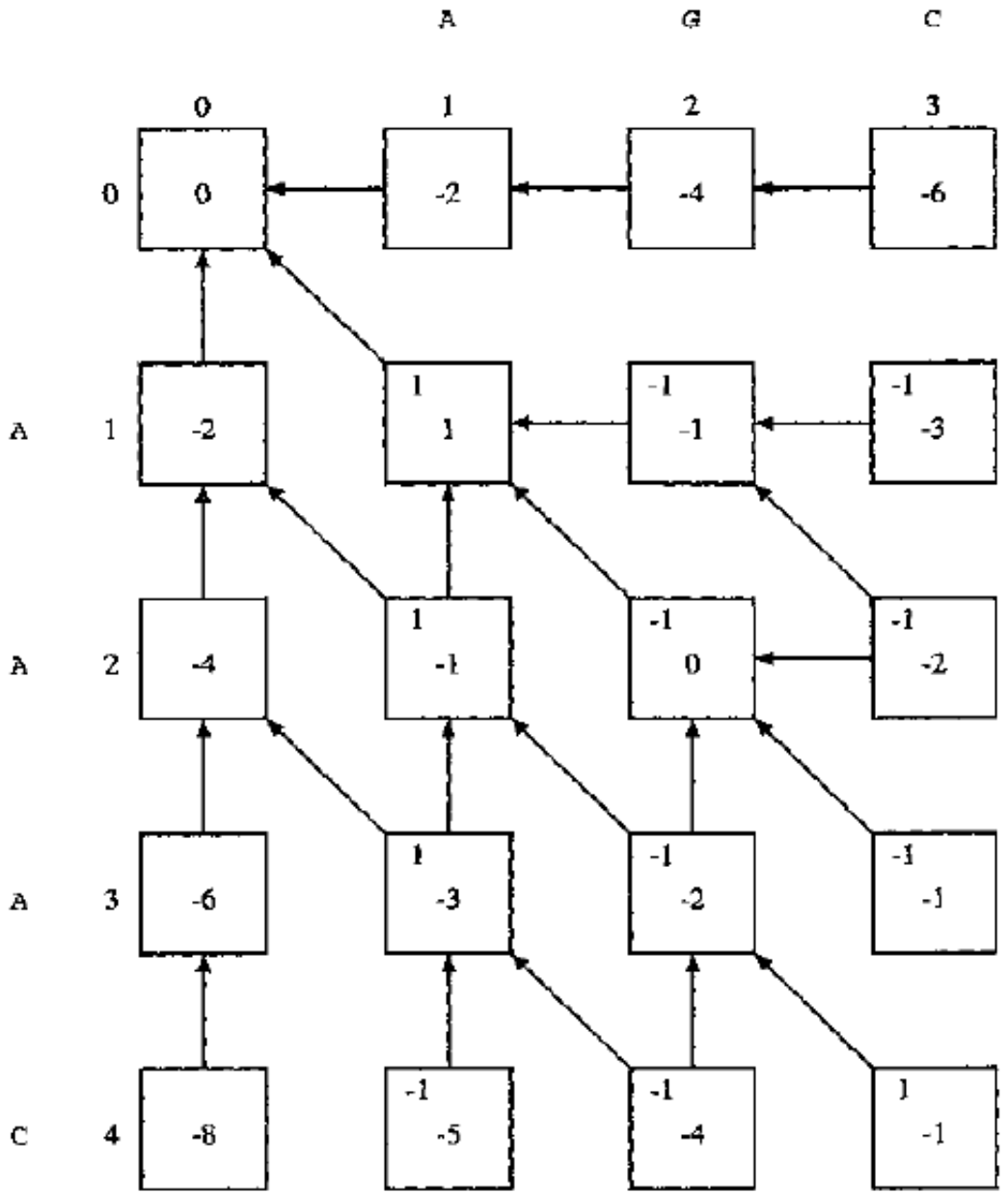
a(i,j) = *max(*

a(i,j-1])-2

a(i-1, j-1]) + p(i,j)

a(i-1, j)-2

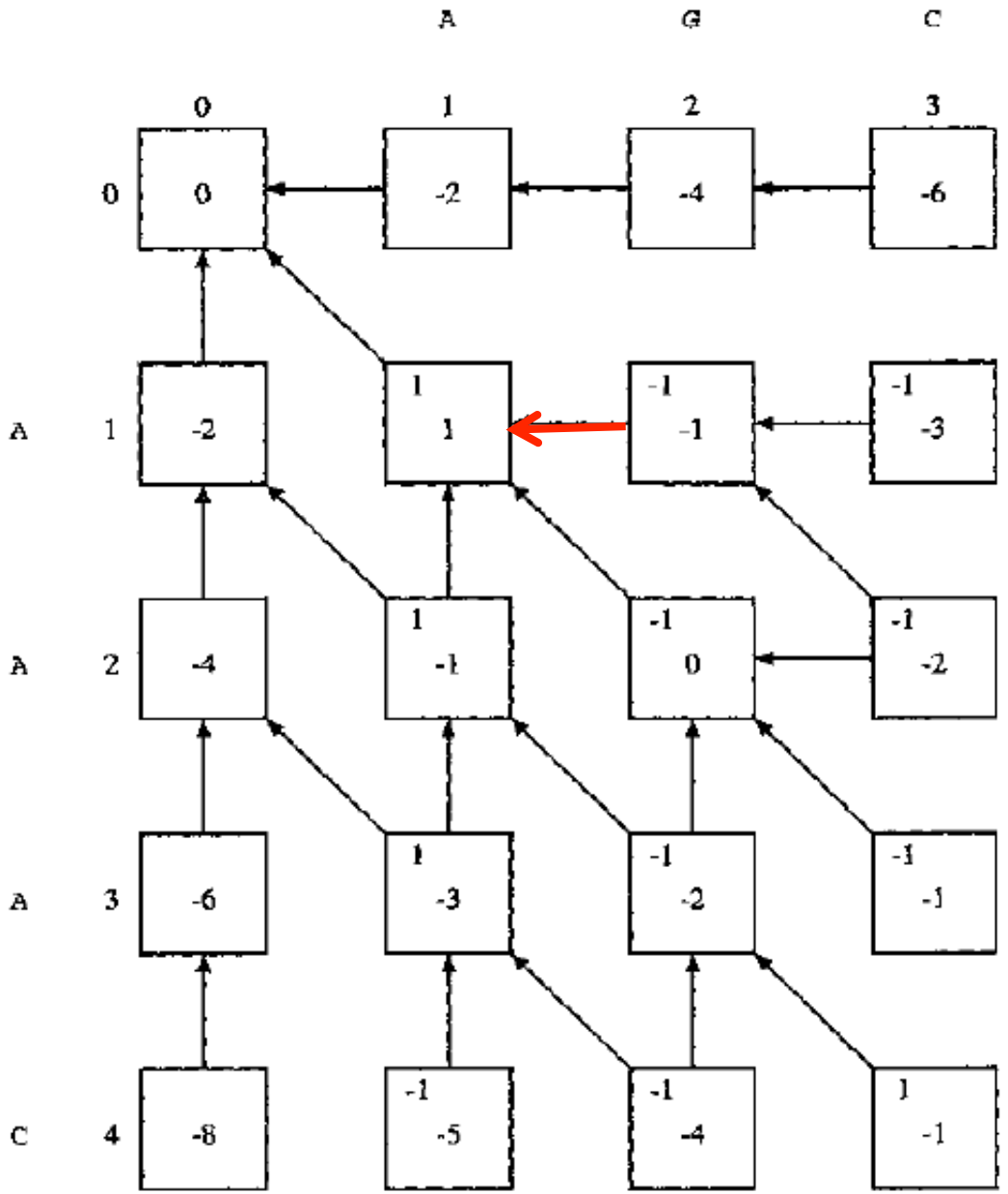
)



Global Comparison

- Finally, we drew arrows in Figure to indicate where the maximum value comes from according to Equation a[i,j]. For instance, the value of *a[1, 2] was taken as the maximum* among the following figures.
- A[1, 1] - 2 = 1 – 2 = -1
- a[0, 1] - 1 = -2 -1 = - 3
- a[0, 2] - 2 = -4 -2 = - 6 .

Therefore, there is only one way of getting this maximum value, namely, coming from entry (1, 1), and that is what the arrows show.



Global Comparison

Algorithm *Similarity*

input: sequences *s and t*

output: similarity between *s and t*

m = \s

n= \t

for *i = 0 to m do*

a[i, 0] = i x g

for *j = 0 to n do*

a[0, j] = j x g

For i = *1 to m do*

for *j = 1 to n do*

a[i, j] = max(a[i - 1, j] + g,

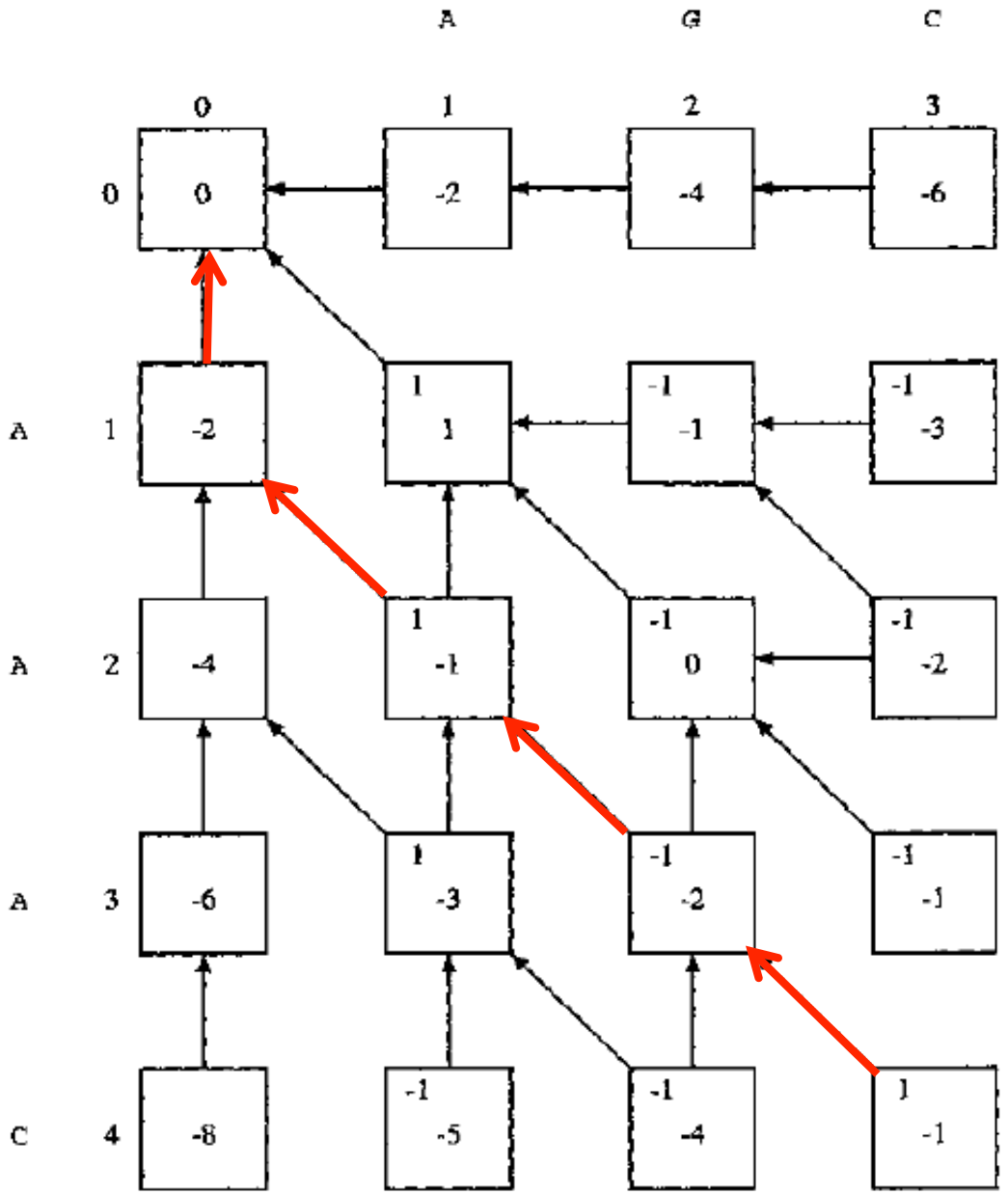
a[i - 1,j - 1] + p(i, j),

a[i, j – 1] + g)

return *a [in, n]*

Optimal Alignment

Now, we want to compute the **optimum alignemnt**: All we need to do is start at entry (m, *n)* *and follow the arrows until we* get to (0, 0). Each arrow used will give us one column of the alignment. In fact, consider an arrow leaving entry (i, *j). If this arrow is horizontal, it corresponds to a column with* a space in *s matched with t[j]. If it is vertical, it corresponds to s[i] matched with a* space in *t. finally, a diagonal arrow means s[i] matched with t[j].*



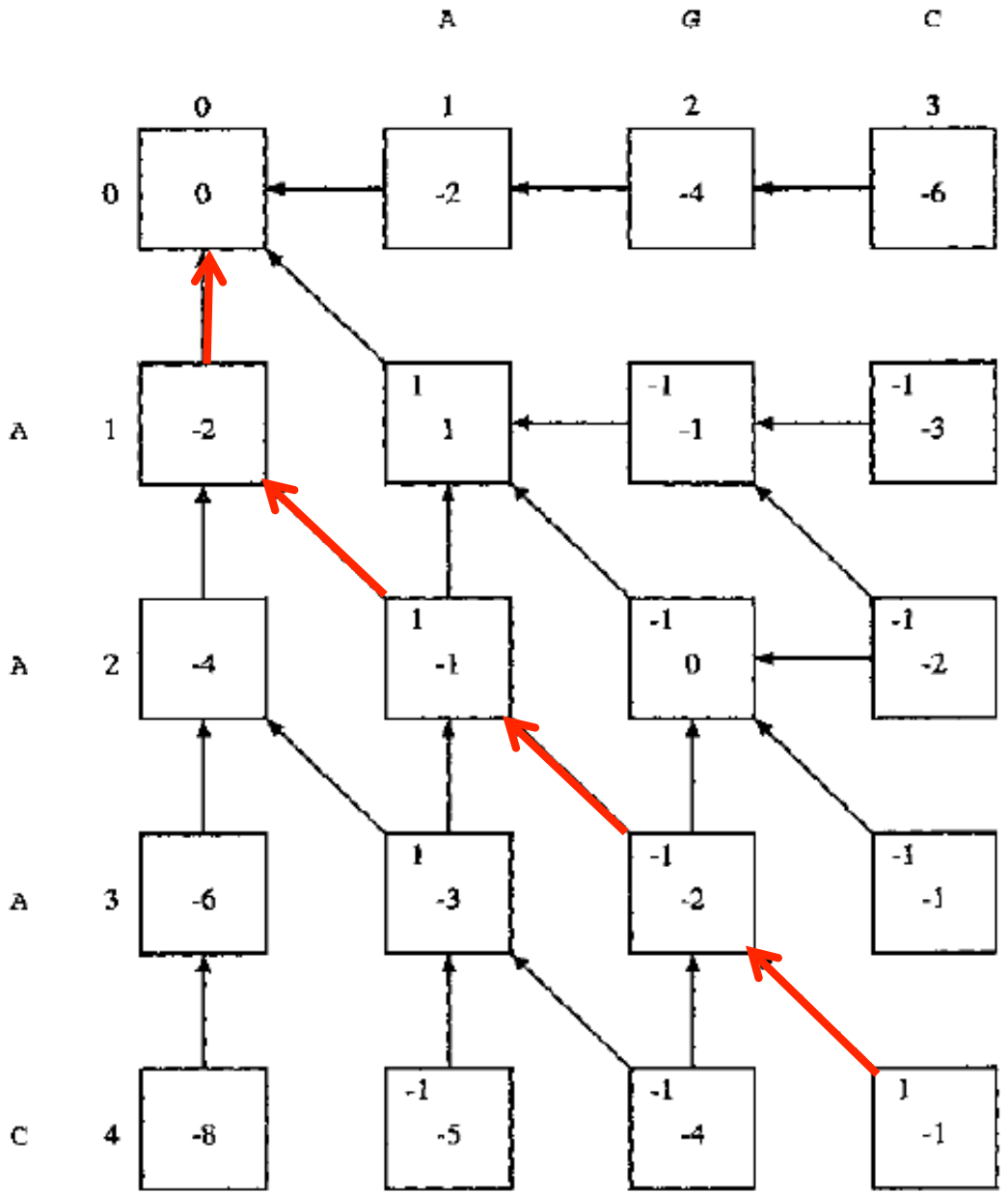
Optimal Alignment

Now, we want to compute the **optimum alignemnt**:

s= - A G C

t= A A A C

This is one of the optimum alignments. As we said previously, many optimal alignments may exist for a given pair of sequences. The algorithm of returns just one of them, giving preference to the edges leaving (i, *j) in counterclockwise order.*



Optimal Alignments

Algorithm *Align*

input: indices *i, j , array a given by algorithm Similarity*

output: alignment in *aligns, align-t, and length in len*

if i==0 and j==0 then

len ***= 0***

else if i > 0 and *a[i, j] = a[i — 1, j] + g then*

Align(i — 1, j , len)

len = len + 1

Align-s[len] = s[i]

align-t[len] = '-'

else if i > 0 and *j > 0 and a[i, j] = a[i — 1, j — 1] + p(i, j) then*

Align{i — 1, j — 1, len)

len = len + 1

align-s[len] = s[i]

align-t[len] = t[j]

else *// has to be j > 0 and a[i, j] = a[i, j — 1] + g*

Align(i, j — 1, len)

len = len + 1

aligns[len] ='-'

align-t[len] = t[j]

Optimal Alignments

Exercise at the blackboard: **align AGTATG and TGATA**

	-	A	G	T	A	T	G
-	0	-2	-4	-6	-8	-10	-12
T	-2	-1	-3	-3	-5	-7	-9
G	-4	-3	0	-2	-4	-6	-6
A	-6	-3	-2	-1	-1	-3	-5
T	-8	-5	-4	-1	-2	0	-2
A	-10	-7	-6	-3	0	-2	-1

t

s

Optimal Alignments

The optimal alignment returned by this algorithm has the following general characteristic:

-when there is choice, a column with a space in *t has precedence over a* column with two symbols, which in turn has precedence over a column with a space in *s.*

-*For instance, when aligning s = ATAT with t = TATA, we get:*

--ATAT

-TATA-

rather than

ATAT-

-TATA

which is the other optimal alignment for these two sequences.

Optimal Alignments

Similarly, when aligning *s = AA with t = AAAA, we get*

--AA

AAAA

although there are five other optimal alignments. This alignment is sometimes referred to as the ***upmost alignment*** *because it uses the arrows higher up in the matrix. To reverse* these preferences, we would reverse the **order of the if statements** in the code, obtaining the ***downmost alignment*** *in this case. A column appearing in both the upmost and downmost* alignments will be present in all optimal alignments between the two sequences considered.

Now, is it possible to modify the algorithm to produce *all optimal alignments between s* and *t? How?*

All the Optimal Alignments

Now, is it possible to modify the algorithm to produce *all optimal alignments between s* and *t? How?*

A: We need to keep a **stack** with the points at which there are options and backtrack to them to explore all possibilities of reaching (0, 0) through the arrows. However, there might be a very large number of optimal alignments.

Time Complexity

The time complexity of the matrix filling algorithm is O(mn), where |s|=m and |t|=n, both in time and in space. If the sequences are nearly the same length, the complexity is O(n^2) (quadratic). Then the algorithm extracting the alignment is O(n+m) following the arrows, since n+m is the maximum length of the (worst) alignment possible.

Local Comparison

A ***local alignment*** *between s and t is an alignment between a* ***substring of s and a substring of t.***

Problem: How would you define an algorithm to find the highest scoring local alignments between two sequences?

Local Comparison

A ***local alignment*** *between s and t is an alignment between a* ***substring of s and a substring*** **of *t.***

Problem: How would you define an algorithm to find the highest scoring local alignments between two sequences?

Main Idea: we exploit the previous "matrix algorithm", but this time the interpretation of the array values is different. Each entry (i, *j) will hold the highest score of an alignment between a* ***suffix*** of s[1..i] and a ***suffix*** *of t[1..j].*

- *The first row and the first column are initialized with* zeros.
- For any entry (i, *j) there is always the alignment between the empty suffixes of* s[1..i] and *t [1..j], which has score zero; therefore the array will have all entries greater* than or equal to zero. This explains in part the initialization above.

Local Comparison

The matrix can be filled in the usual way, with *a[i, j] depending* on the value of three previously computed entries. The resulting recurrence is:

$$a(i,j) = max($$

$$a(i,j-1])+g$$

$$a(i-1, j-1]) + p(i,j)$$

$$a(i-1, j)+g$$

$$0$$

$$)$$

(the same as in the basic algorithm, except that now we have a fourth possibility, not available in the global case, of an empty alignment.)

But which one is the result in the matrix we are interested now?

Local Comparison

But which one is the result in the matrix we are interested now?

A: it suffices to find the maximum entry in the whole array. This will be the score of an optimal local alignment.

Any entry containing this value can be used as a starting point to get such an alignment.

The rest of the alignment is obtained tracing back as usual, but stopping as soon as we reach an entry with no arrow going out.

Alternatively, we can stop as soon as we reach an entry with value zero.

Problem: In general, when doing local comparison, we are interested not only in the optimal alignments, but also in near optimal alignments with scores above a certain threshold. Any idea on how to obtain this result now?

Semi-Global Comparison

In a semiglobal comparison, we score alignments ignoring some of the *end spaces in the* sequences. An interesting characteristic of the basic dynamic programming algorithm is that we can control the penalty associated with end spaces by doing very simple modifications to the original scheme.

Let us begin by defining precisely **what we mean by *end spaces*** *and why it might* be better to let them be included for free in certain situations.

End spaces are those that appear before the first or after the last character in a sequence. For instance, all the spaces in the second sequence in the alignment below are end spaces, while the single space in the first sequence is not an end space.

CAGCA-CTTGGATTCTCGG

---CAGCGTGG--------

Semi-Global Comparison

CAGCA-CTTGGATTCTCGG

---CAGCGTGG--------

Notice that the lengths of these two sequences differ considerably. One has size 8, and the other has 18 characters. When this happens, there will be **many spaces in any alignment**, giving a large negative contribution to the score. Nevertheless, if we ignore end spaces, the alignment is pretty good, with 6 matches, 1 mismatch, and 1 space.

Observe that this is not the best alignment between these sequences. In alignment below we present another alignment with a higher score (—12 against —19 of the previous one) according to the scoring system we have been using so far.

CAGCACTTGGATTCTCGG

CAGC-----G-T----GG

However this alignment is not so interesting when finding similar regions in the sequences!

Semi-Global Comparison

However this alignment is not so interesting when finding similar regions in the sequences!

Problem: So, how to obtain such a good semi-global alignment only?

Again, we need a variation of the basic algorithm that will ignore end spaces… ☺

Consider initially the case where we do not want to charge for spaces after the last character of *s*. *Take an optimal alignment in this case. The spaces after the end of s are matched* to a suffix of *t*.

If we remove this final part of the alignment, we have an alignment between *s and a prefix of t, with score equal to the original alignment. Therefore,* to get the score of the optimal alignment between *s and t without charge for spaces after* the end of *s, all we need to do is to find the best similarity between s and a prefix of t. But* we know that the entry (i, *j) of the matrix a contains the similarity between s[1..i] and t[1..j]. Hence, it suffices to take the maximum value in the last row of matrix a, (when s has been eaten all), that* is,

Sim'(s, *t*) = *max(j=1..n) a[m, j] ,* *with* $m=|s|$

Notice that Sim' ignores the end spaces, differently than Sim

Semi-Global Comparison

Sim'(s, *t*) = *max(j=1..n) a[m, j]* , *with* $m=|s|$

Notice that Sim' ignores the end spaces, differently than Sim

Now, to extract the alignment itself, we proceed just as in the basic algorithm, but starting at (m, k) where k is such that sim'(s, t) = a[m, k].

An analogous argument solves the case in which we do not charge for final spaces in t. We take the maximum along the last column of a in this case. We can even combine the two ideas and seek the best alignment without charging for final spaces in either sequence.

The answer will be found by taking the maximum along the border of the matrix formed by the union of the last row and the last column. In all cases, to recover an optimal alignment, we start at an array entry that contains the similarity value and follow the arrows until we reach (0, 0). Each arrow will give one column of the alignment as in the basic case.

Now let us **turn our attention to the case of initial spaces**. Suppose that we want the best alignment that does not charge for initial spaces in *s.*

Semi-Global Comparison

Now let us **turn our attention to the case of initial spaces**. Suppose that we want the best alignment that does not charge for initial spaces in s.

This is equivalent to the best alignment **between s and a suffix of t**.

To get the desired answer, we use an (m + 1) x (n + 1) array just as in the basic algorithm, but with a slight difference. Each entry (i, j) now will contain the highest similarity between s[1..i] and **a suffix of a prefix of t**, which is to say a suffix of t[1..j].

Doing that, it is clear **that a[m, n] will be the answer**.

What is less clear, but nevertheless true, is that the array can be filled in using exactly the same formula as in the basic algorithm (the one with the zero option in slide 32), even though the initialization must be different: the first row must be initialized with zeros instead of multiples of the space penalty because of the new meaning of the entries.

Exercise: verify that Equation in slide 32 works in this case.

So, now…

Semi-Global Comparison

So, now…to summarize, we can apply the same trick and initialize the first column with zeros, and by doing this we will be forgiving spaces before the beginning of t.

If in addition we initialize both the first row and the first column with zeros, and proceed with Equation (in slide 32) for the other entries, we will be computing in each entry the highest similarity between a suffix of s and a suffix of t. To find an optimal alignment, we follow the arrows from the maximum value entry until we reach one of the borders initialized with zeros, and then follow the border back to the origin.

Intuitively, there are four places where we may not want to charge for spaces: beginning or end of s, and beginning or end of t.

Semi-Global Comparison

Intuitively, there are four places where we may not want to charge for spaces: beginning or end of s, and beginning or end of t.

We can combine these conditions independently in any way and use the variations above to find the similarity. The only things that change are the initialization and where to look for the maximum value. Forgiving initial spaces translates into initializing certain positions with zero. Forgiving final spaces means looking for the maximum along certain positions. But filling in the array is always the same process.

Place where spaces are not charged for	**Action**
Beginning of first sequence	Initialize first row with zeros
End of first sequence	Look for maximum in last row
Beginning of second sequence	Initialize first column with zeros
End of second sequence	Look for maximum in last column

Reducing memory cost

Problem: Now, We analyze the problem of **memory costs.** We want to reduce the space requirements of the algorithms from quadratic (mn) to linear (m+n)…. at a cost of roughly doubling computation time.

Any idea?

More Realistic Alignment

Problem: another possible improvement has to do with the biological interpretation of alignments. From the biological point of view, it is more realistic to consider a series of consecutive spaces instead of individual spaces in the alignment algorithm.

Any idea on how to do this?

Reducing memory cost

Problem: The quadratic complexity of the basic algorithms makes them unattractive in some applications involving very long sequences or repeated comparison of several sequences.

With respect to space, however, it is possible to improve complexity from quadratic to linear and keep the same generality. The price to pay is an increase in processing time, which will roughly double. Nevertheless, the asymptotic time complexity is still the same, and in many cases **space and not time is the limiting factor**, so this improvement is of great practical value. So we need to design an elegant space-saving technique!

We begin by noticing that computing sim(s, *t) can be easily done in linear space.*

In fact, each row (or column) of the matrix depends only on the preceding one, and it is possible to perform the calculations keeping only one vector in memory, which will hold partly the new row being computed and partly the previous row.

Reducing memory cost

Problem: The quadratic complexity of the basic algorithms makes them unattractive in some applications involving very long sequences or repeated comparison of several sequences.

With respect to space, however, it is possible to improve complexity from quadratic to linear and keep the same generality. The price to pay is an increase in processing time, which will roughly double. Nevertheless, the asymptotic time complexity is still the same, and in many cases **space and not time is the limiting factor**, so this improvement is of great practical value. So we need to design an elegant space-saving technique!

We begin by noticing that computing sim(s, *t) can be easily done in linear space.*

In fact, each row (or column) of the matrix depends only on the preceding one, and it is possible to perform the calculations keeping only one vector in memory, which will hold partly the new row being computed and partly the previous row. So what would you do if m>n?

Reducing memory cost

Algorithm for similarity in linear space. In the end, a[n] contains sim(s, t).

Algorithm *BestScore*

input: sequences *s and t*

output: vector *a*

m=|*s*|

n=|*t*|

For *j = 0 to n do*

a[j] = j x g

for i = 1 to *m do*

old = a[0]

a[0] = i x g

for j = *1 to n do*

temp = a[j]

a[j] = max(a[j]+g, old+p(i,j), a[j-1]+g)

old = temp

The key idea is the following:

Fix an optimal alignment and a position *i in s, and* consider what can possibly be matched with *s[i] in this alignment. There are only two* possibilities:

1. The symbol *t[j] will match s[i], for some j in 1..n.*

2. A space between *t[j] and t[j + 1] will match s [i], for some j in O..n.*

In the second case the index *j varies between 0 and n because there is always one* more position for spaces than for symbols in a sequence. We also abused notation when *j = 0 or j = n. What we mean in these cases is that the space will be before t[1] or* after *t[n], respectively.*

Continue in next slide…

Reducing memory cost

Now, let OPT(x, y) be the optimal alignment between x and y.

Every alignment between s and t, optimal or not, satisfies (1) or (2).

1. The symbol *t[j] will match s[i], for some j in 1..n.*

2. A space between *t[j] and t[j + 1] will match s[i], for some j in 0..n.*

In particular, our fixed optimal alignment must satisfy one of these as well. If it satisfies (1), to obtain all of it we must concatenate (A):

OPT(s[1..i-1]) + s[i] + OPT(s[i+1..m])

OPT(t[1..j-1]) + t[j] + OPT(t[j+1..n])

While if (2) is satisfied, we must concatenate (B):

OPT(s[1..i-1]) + s[i] + OPT(s[i+1..m])

OPT(t[1..j-1]) + "-" + OPT(t[j+1..n])

This provides a recursive method to compute an optimal alignment, as long as we can determine, for a given i, which (1) or (2) occurs and what is the corresponding value of *j* .

Reducing memory cost

A recursive method to compute an optimal alignment, splitting the problem into two equivalent subproblems…. Sounds like something already known right?

YES, **Divide et Impera!**

This can be done as follows. According to Equations A and B in previous slide, we need, for fixed *i, the similarities between s[1..i — 1] and an arbitrary prefix of t, and also the similarities* between *s[i + 1..m] and an arbitrary suffix of t. If we had these values, we could* explicitly compute the scores of the *j alignments represented in (A) and of the j + 1* alignments represented in (B). By choosing the best among these, we will have the information necessary to proceed in the recursion.

As we saw earlier, it is possible to compute in linear space the best scores between a given prefix of *s and all prefixes of t (BestScore alg.). A similar algorithm exists for* suffixes. Hence, our problem is almost solved. The only thing left is to decide which value of *i to use in each recursive call. What is the best choice for i?*

Reducing memory cost

A recursive method to compute an optimal alignment, splitting the problem into two equivalent subproblems…. Sounds like something already known right?

YES, **Divide et Impera!**

This can be done as follows. According to Equations A and B in previous slide, we need, for fixed *i, the similarities between s[1..i — 1] and an arbitrary prefix of t, and also the similarities* between *s[i + 1..m] and an arbitrary suffix of t. If we had these values, we could* explicitly compute the scores of the *j alignments represented in (A) and of the j + 1* alignments represented in (B). By choosing the best among these, we will have the information necessary to proceed in the recursion.

As we saw earlier, it is possible to compute in linear space the best scores between a given prefix of *s and all prefixes of t (BestScore alg.). A similar algorithm exists for* suffixes. Hence, our problem is almost solved. The only thing left is to decide which value of *i to use in each recursive call. What is the best choice for i?*

The best choice is to ***pick i as close as possible to*** **the middle of the sequence, of course.**

Reducing memory cost

Now, given the service functions implemented:

BestScore(s[a..i — 1], t[c..d],pref-sim)

Which returns in *pref-sim the similarities between s[a..i — 1] and t[c..j] for all j in [c-1..d].*

And the function:

BestScoreRev(s[i + 1..b], t[c..d], suff-sim)

returns in *suff-sim the similarities between s [i + 1 ..b] and t [j +1..d] for all j in [c—1..d].*

The call of the function (see next slide):

Align(1, m, 1, n, 1, len)

will return an optimal alignment in the global variables *align-s and align-t, and the size* of this alignment in *len.*

Reducing memory cost

Algorithm *Align*

input: sequences *s and r, indices a, b, c, d, start position start*

output: optimal alignment between *s[a..b] and t[c..d] placed in vectors align-s*

and *align-t beginning at position start and ending at position end*

if *s[a..b] empty or t[c..d] empty then*

// Base case: s[a..b] empty or t[c..d] empty

Align the nonempty sequence with spaces

end = start + max(|s|, |t|)

else

// General case…

Reducing memory cost

...else

// General case

i = bottom((a+b)/2)

BestScore(s[a..(i - 1)], t[c..d], pref-sim)

BestScoreRev(s[(i + 1)..b], t[c..d], suff-sim)

posmax = c — 1

typemax = SPACE

vmax = pref-sim[c — 1] + g + suff-sim[c — 1]

for *j = c to d do*

if (*pref-sim[j — 1] + p(i, j) + suff-sim[j]) > vmax then*

posmax = j

typemax = SYMBOL

vmax = *pref-sim[j — 1] + p(i, j) + suff-sim[j]*

Reducing memory cost

```
                    (((( vmax = pref-sim[j — 1] + p(i, j) + suff-sim[j] ))))….
          if pref-sim[j] + g + suff-sim[j] > vmax then
                    posmax = j
                    typemax = SPACE
                    vmax = pref-sim[j] + g + suff-sim[j]
endfor
if typemax = SPACE then
          Align(a, i — 1, c, posmax, start, middle)
          align-s[middle] = s[i]
          align-t[middle] = SPACE
          Align(i + 1, b, posmax + 1, d, middle +1, end)
else // typemax = SYMBOL…
```

Reducing memory cost

else // *typemax = SYMBOL…*

Align(a, i — 1, c, posmax — 1, start, middle)

align-s[middle] = s[i]

align-t[middle] = t[posmax]

Align{i +1, b, posmax + 1, d, middle + 1, end)

endAlign

Now, what about the additional time complexity cost of this compact-space implementation?

Reducing memory cost

Now, what about the additional time complexity cost of this compact-space implementation?

Let *T(m, n) be the number of times a maximum is computed in the internal loop of BestScore or BestScoreRev as a result of a call Align(a, b, c, d, start, end) where m =b — a + 1 and n = c — d + 1. It is easy to see that the total processing time will be* proportional to *T(m, n) plus linear terms due to control and initializations. We claim that* ***T(m,n) < 2mn.***

Proof by induction on *m*.

For m = 1 no maximum computations will occur, so obviously T(1, *n) < 2n.*

For m > 1 we will have a call to BestScore with at most *mn/2 maximum computations, another such amount for BestScoreRev, and* two recursive calls to *Align, producing at most T(m/2, j) and T(m/2, n — j) maximum* computations. Adding this all up, we have

T(m, n) <= *mn/2 + mn/2 + T(m/2, j) + T(m/2, n - j)*

<= mn + mj + m(n — j) = ***2mn***, proving the claim.

Reducing memory cost

Now, what about the additional time complexity cost of this compact-space implementation?

Let *T(m, n) be the number of times a maximum is computed in the internal loop of BestScore or BestScoreRev as a result of a call Align(a, b, c, d, start, end) where m =b — a + 1 and n = c — d + 1. It is easy to see that the total processing time will be* proportional to *T(m, n) plus linear terms due to control and initializations. We claim that* ***T(m,n) < 2mn.***

Proof by induction on *m*.

For m = 1 no maximum computations will occur, so obviously T(1, *n) < 2n.*

For m > 1 we will have a call to BestScore with at most *mn/2 maximum computations, another such amount for BestScoreRev, and* two recursive calls to *Align, producing at most T(m/2, j) and T(m/2, n — j) maximum* computations. Adding this all up, we have

T(m, n) <= *mn/2 + mn/2 + T(m/2, j) + T(m/2, n - j)*

<= mn + mj + m(n — j) = ***2mn***, proving the claim.

Gap Sequences

Now, Let us define a **gap as being a consecutive number *k > 1 of spaces.***

It is generally accepted that, when DNA mutations are involved, the occurrence of a gap with *k spaces is more* probable than the occurrence of *k isolated spaces. This is because a gap may be due to a* single mutational event that removed a whole stretch of residues, while separated spaces are most probably due to distinct events, and the occurrence of one event is more common than the occurrence of several events.

Up to now, we have not made any distinction between clustered or isolated spaces.

This means that a gap is penalized through a linear function. Denoting by *w(k), for k* >1, the penalty associated with a gap with *k spaces, we have w(k)=bk,* where *b is the absolute value of the (negative) score associated with a space.*

Exercise: Think to an algorithm that computes similarities with respect to more general gap penalty functions *w paying more those alignment with max gap (hint: such that penalties are not additive)*

Algorithms and Data Structures 2010 - 2011

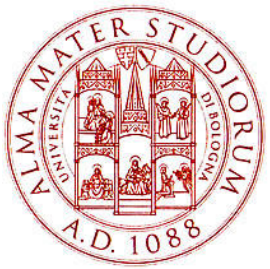
Lesson 9: some examples of algorithms for bioinformatics

Luciano Bononi

<bononi@cs.unibo.it>

http://www.cs.unibo.it/~bononi/

(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)



International Bologna Master in Bioinformatics

University of Bologna

20/05/2011, Bologna