Computational Biology
Lecture 5: Time speedup, General gap penalty function
Saad Mneimneh

We saw earlier that it is possible to compute optimal global alignments in linear space (it can also be done for local alignments). Now we will look at how to speedup time in some cases.

Similar sequences: Bounded dynamic programming

If we believe that the two sequences x and y are similar then we might be able to optimally align them faster. For simplicity assume that $m = n$ since our sequences are similar. If x and y align perfectly, then the optimal alignment corresponds to the diagonal in the dynamic programming table (now of size $n \times n$). This is because all of the back pointers will be pointing diagonally. Therefore, if x and y are similar, we expect the alignment not to deviate much from the diagonal.

For instance, consider the following two sequences $x = GCGCATGGATTGAGCGA$ and $y = TGCGCCATGGATGAGCA$. Their optimal alignment is shown below:

```
-GCGC-ATGGATTGAGCGA
TGCGCCATGGAT-GAGC-A
```

The following figure shows the alignment (back pointers) in the dynamic programming table.

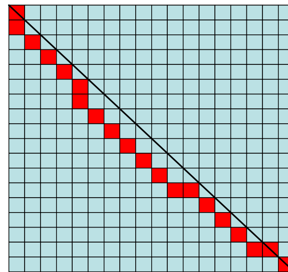

Figure 1: The optimal alignment of x and y above

Note that since x and y have the same length, their alignment will always contain gap pairs, i.e. for each gap in x there will be a corresponding gap in y. The deviation from the diagonal is at most equal to the number of gap pairs (it could be less). Therefore, if the number of gap pairs in the optimal alignment of x and y is k, we only need to look at a band of size k around the diagonal. This speeds up the time needed to compute the optimal alignment since we only update the entries $A(i, j)$ such that $A(i, j)$ falls inside the band. But we don't know the optimal alignment x and y to start with in order to obtain k. How should we set k then? Start with some value of k that you think is appropriate for your similar sequences. Could k be a wrong choice? Yes, and in that case the computed alignment will not be optimal. The book describes an iterative approach that doubles the value of k in each time until an optimal alignment is obtained. We will not describe the approach here, but we will assume that we make an educated guess on the value of k, and that we will be lucky and our computed alignment will be good enough if not optimal.

To summarize, the main idea is to bound the dynamic programming table in a band of size k around the diagonal as illustrated in the figure below:

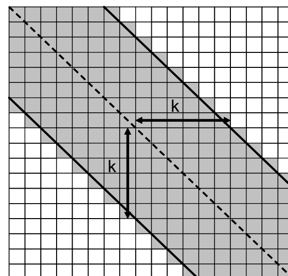

Figure 2: Bounded dynamic programming

Now we only need to change the basic algorithm in such a way that we do not touch or check $A(i, j)$ if it is outside the band. In other words, we need to assume that our dynamic programming table is just the band itself. It is easy to accomplish this since $A(i, j)$ inside the band implies $|i - j| \leq k$ (convince yourself with this simple fact). The modified Needleman-Wunsch is depicted below (pointer updates not shown, same as before).

- Initialization
  - A(0,j) = – d.j   if j ≤ k
  - A(i,0) = – d.i   if i ≤ k
- Main iteration
  For i= 1 to n
    For j = max(1,i-k) to min(n,i+k)

$$A(i,j) = \max \begin{cases} A(i-1, j-1) + s(i,j) & \\ A(i-1,j) - d & \text{if } |i-1-j| \leq k \\ A(i,j-1) - d & \text{if } |i-j+1| \leq k \end{cases}$$

- Termination
  $A^{opt} = A(n,n)$

Figure 3: Bounded Needleman-Wunsch

Now the running time is just $O(kn)$ as opposed to $O(n^2)$.

The Four Russians Speedup

In 1970 Arlazarov, V. L., E. A. Dinic, M. A. Kronrod, and I. A. Faradzev, in their paper, "On economical construction of the transitive closure of a directed graph" presented a technique to speedup Dynamic Programming. This technique is commonly referred to as Four Russians Speedup.

The idea behind the four russians speedup is to divide the dynamic programming table into $t \times t$ blocks called $t-blocks$ and then compute the values one $t - block$ at a time rather one cell at a time. Of course this by itself does not achieve any speedup, but the goal is to compute each block in $O(t)$ time only rather than $O(t^2)$, thus achieving a speedup of t.

The following figure shows a $t - block$ positioned at $A(i, j)$ in the dynamic programming table:

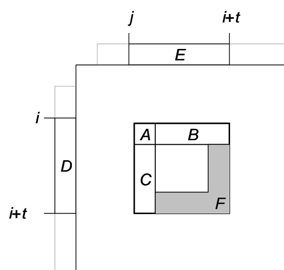

Figure 4: a $t - block$ at $A(i, j)$

Since the unit of computation is now the $t - block$, we do not really care about the values inside the $t - block$ itself. As the above figure illustrates, we are only interested in computing the boundary F, which we will call the output of the $t - block$. Note that given A, B, C, D, and E, output F is completely determined. We will call A, B, C, D, and E the $t - block$ input. Note also that the input and output of a $t - block$ are both $O(t)$ in size. Therefore, we will assume the existence of a $t - block$ function that, given A, B, C, D, and E, computes F in $O(t)$ time.

How can $t-blocks$ with the assumed $t-block$ function help? Here's how: divide the dynamic programming table into intersecting $t - blocks$ such that the last row of a $t - block$ is shared with the first row of the $t - block$ below it (if any), and the last column of a $t - block$ is shared with the first column of the $t - block$ to its right (if any). For simplicity assume $m = n = k(t-1)$ so that each row and each column in the table will have k intersecting $t - block$ (total number of $t - blocks$ is therefore k^2). The following figure illustrates the idea for $n = 9$ and $t = 4$.

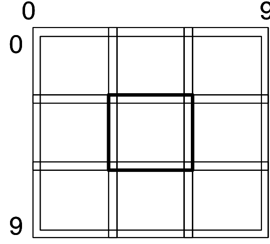

Figure 5: $4 - blocks$ in a 10×10 table ($n = 9$)

Now the computation of the dynamic programming table can be performed as follows:

- initialize the first row and first column of the whole table
- in a rowwise manner, use the $t - block$ function to compute the output of each $t - block$
- the output of one $t - block$ will serve as input to another
- finally return $A(n, n)$

The running time of the above algorithm is $O(k^2 t) = O(\frac{n^2}{t^2} t) = O(\frac{n^2}{t})$, hence achieving a speedup of t as we said before.

The big question now is how to construct the $t - block$ function which, given the input, computes the output in $O(t)$ time. This is conceptually easy to do. If we pre-compute outputs for all given inputs and store the results indexed by the inputs, then we can retrieve the output for every input in $O(t)$ time only (input and output both have size $O(t)$).

So how do we pre-compute the outputs. Again easy: for every input, compute the output the usual way (cell by cell in a rowwise manner) in $O(t^2)$ time and store it. The total time we spend on the pre-computation will be $O(It^2)$ where I is the number of possible inputs. You might say now, hey! didn't we say that we are not going to spend the $O(t^2)$ time on a $t - block$?). Yes, but the trick is that we are spending now $O(t^2)$ time for each input and not on each $t - block$, and we are hoping that the number of inputs is small enough. But it is not!

Let us find out the number of possible inputs. Well, D and E can each have $O(\alpha^t)$ possible values, where α is the size of the alphabet used for the sequences Moreover, C and D can each have $O(L^t)$ values, where L is the number of possible values that a cell in the dynamic programming table can have. Of course, we expect $L >> n$. Therefore, the number of possible inputs is $I = O(\alpha^{2t} L^{2t})$ and the running time for the pre-computation will be $O(\alpha^{2t} L^{2t} t^2)$. But t has to be at least 2 (otherwise there is no need for this whole approach). Therefore this bound exceed $O(n^2)$.

The problem is that the number of possible inputs is huge. Can we reduce it to something linear in n? The answer is yes, but we have to first look at the following observation:

Lemma: If the (+1,-1,-2) scoring scheme was used, then in any row, column, or diagonal, two adjacent cells differ by at most 3.

Proof: First, $A(i, j-1) - A(i, j) \leq 2$ because $A(i, j) \geq A(i, j-1) - 2$ from the update rule. Now consider $A(i, j) - A(i, j-1)$. Replace y_j with a gap in the alignment of $x_1..x_i$ and $y_1..y_j$ with score $A(i, j)$, we get an alignment of $x_1..x_i$ and $y_1..y_{j-1}$ with score at least $A(i, j) - 3$ (the worst case is when x_i was aligned with y_j, i.e. we replaced a match with a gap). So $A(i, j-1) \geq A(i, j) - 3$ because $A(i, j-1)$ is the optimal score for aligning $x_1..x_i$ and $y_1..y_{j-1}$. Thus $A(i, j) - A(i, j-1) \leq 3$. A symmetric argument works for the column. We leave the diagonal as an exercise (but it is not needed for the argument).

Now the trick. Since adjacent cells cannot differ a lot, we can use offset encoding to encode the input of a $t - block$. First we define an offset vector to be a vector of size t such that the values in the vector are in the set $\{-3, -2, -1, 0, 1, 2, 3\}$ and the first entry of the vector is 0. We can encode any row or column as an offset vector. For instance, consider the row 5, 4, 4, 5. This can be represented as 0, -1, 0, +1. Given this offset vector and the value 5, the row can be completely recovered as follows: we start with 5, we add an offset of -1 we obtain 4, we add an offset of 0 we obtain 4, we add an offset of +1 we obtain 5.

The other important observation is that given the input of a $t - block$ as offset vectors, the $t - block$ function can compute the output as offset vectors as well, without even knowing the initial values. Below is an example:

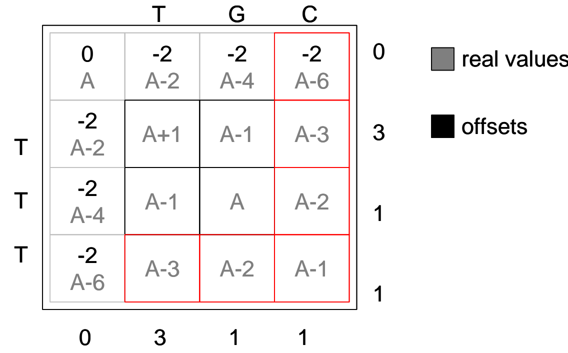

Figure 6: Computing with offset vectors

As you can see in the figure above, the initial value A is not needed, the $t - block$ function can simply work in terms of A to compute the output in offset vectors.

At the end, we can determine $A(n, n)$ by starting from $A(n, 0)$ and adding all the offsets in the offset vectors of the last row of $t - blocks$.

What have we achieved? Now the number of possible inputs is reduced considerably. Inputs are now offset vectors. An offset vector of size t can have at most 7^t values. Now the number of possible inputs $I = O(\alpha^{2t} 7^{2t})$. For DNA sequences, $\alpha = 4$. Therefore $I = O(28^{2t})$. If we set $t = \frac{1}{2} \log_{28} n$ we have $I = O(n)$. The running time of the pre-computation is now $O(It^2) = O(n \log^2 n)$. The total time for computing the the output of the whole dynamic programming table as offset vectors is $O(n \log^2 n + \frac{n^2}{\log n}) = O(\frac{n^2}{\log n})$.

General gap penalty function

So far we have considered a gap as some sort of individual mismatches and penalized them as such. There are some biological reasons to believe that once addition or deletion happens, it is very likely that it will happen over a cluster of contiguous positions, e.g. a gap of length k is more likely to occur that k contiguous gaps of length 1 each, occurring independently. Thus we more a general gap penalty function that gives us the option of not penalizing additional gaps as much as the first gap.

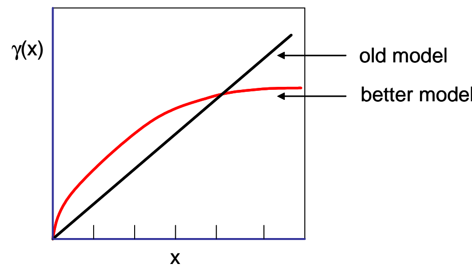

Figure 7: More general gap function

This new model for penalizing gaps introduces a new difficulty for our alignment algorithms. The additivity of the score does not hold anymore, a property that we have been relying upon a lot. Below is a figure that illustrates the idea.

xxxxxxxxxxxxxx
yyyyy-----yyyy

score ≠ (xxxxxxx / yyyyy-- $-\gamma(2)$) + (xxxxxxx / ---yyyy $-\gamma(3)$)

$\gamma(5) \neq \gamma(2) + \gamma(3)$

Figure 8: The score is not additive across a gap

This implies that if the alignment of $x_1...x_i$ and $y_1...y_j$ is optimal, a prefix (or suffix) of that alignment is not necessarily optimal (with the score being additive this was the case before). Here's an example: $x = ABAC$ and $y = BC$. Assume that the score of a match is 1, that of a mismatch is $-\epsilon$, and that $\gamma(1) = -2$, and $\gamma(2) = -2 - \epsilon$. Then the best alignment is:

```
ABAC
B--C
```

which gives a score of $-1 - 2\epsilon$. Obviously, the part:

```
AB
B-
```

is not optimal.

So our update rules have to change. Let us investigate how we should change our basic Needleman-Wunsch algorithm to work with the new gap penalty function.

The optimal alignment of $x_1...x_i$ and $y_1...y_j$ that aligns x_i with y_j still has a score of $A(i-1, j-1) + s(i, j)$. This is because the score is additive in that case. The alignment ends with x_i aligned with y_j so the cut does not go across a gap.

What about the two other cases. Well, they are symmetric so let us look at one of them. Consider the optimal alignment of $x_1...x_i$ and $y_1...y_j$ that aligns x_i with a gap. The score of that alignment is not necessarily $A(i-1, j) - \gamma(1)$. This is because the cut here might go across a gap. A good observation is the following: The optimal alignment of $x_1...x_i$ and $y_1...y_j$ that aligns x_i with a gap must end with a gap of a certain length. Therefore, if we try all possible lengths for the gap we will eventually find the optimal alignment of $x_1...x_i$ and $y_1...y_j$ that aligns x_i with a gap. Therefore, we will try $A(i-k, j) - \gamma(k)$ for $k = 1...i$.

Here's the new update rule:

$$A(i,j) = \max \begin{cases} A(i-1, j-1) + s(i,j), & Ptr(i,j) = (i-1, j-1) \\ A(i, j-k) - \gamma(k) \ \ k = 1...j, & Ptr(i,j) = (i, j-k) \\ A(i-k, j) - \gamma(k) \ \ k = 1...i, & Ptr(i,j) = (i-k, j) \end{cases}$$

The initialization will also change appropriately. $A(0,0) = 0$, $A(i,0) = -\gamma(i)$, and $A(0,j) = -\gamma(j)$.

For each entry $A(i,j)$ we now spend $O(1+i+j)$ time. Therefore, the total running time is $O(\sum_i \sum_j (1+i+j)) = O(m^2 n + n^2 m)$. This is now cubic compared to the quadratic time for previous algorithms. We will see next lecture how to go back to our quadratic time bound using an affine gap penalty function that approximates the general one.

Note that in coming up with the new update rule for $A(i,j)$, we did not consider anything about the function γ. What if γ does not penalize additional gaps less than the first gap? There was nothing in our analysis that relied on this property. There is a catch! Our new update rule requires that property. More formally, we expect γ to satisfy $\gamma(x+1) - \gamma(x) \leq \gamma(x) - \gamma(x-1)$. Try to think why this is needed by the algorithm. The book provides another algorithm where this condition is not required.

References

Setubal J., Meidanis, J., Introduction to Molecular Biology, Chapter 3.

Gusfield D., Algorithms on Strings, Trees, and Sequences, Chapter 12.