Local Alignments

23/03/2010

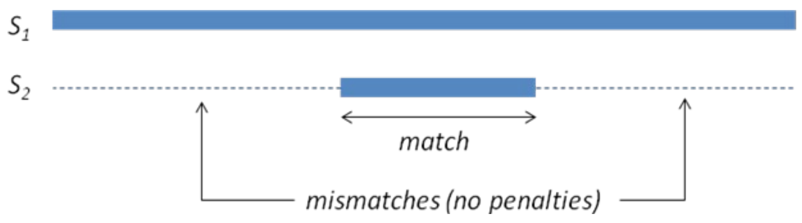
Dynamic programming can be used for either the exact matching problem or the inexact matching problem. Solving the exact matching problem yields solutions that are always correct, but the process is slow. For the inexact matching problem, on the other hand, the results are less accurate, but the process works comparatively faster. It is useful to consider a middle ground between the two.

We have discussed global alignments, where for strings S_1 and S_2, all of S_1 is matched (or 'aligned' to be more accurate) with all of S_2. However, this is not a feasible approach if you want to match a small part of the text. This brings us to the **local alignment problem**:

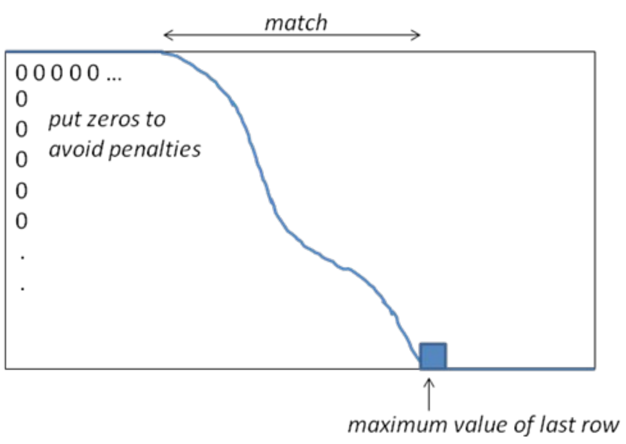
> *Find the substrings S_1' of S_1 and S_2' of S_2 such that S_1' and S_2' form the best alignment between all substrings of S_1 and S_2.*

An immediate consideration is: will this blow out the running time? A trivial algorithm for solving this will take at least quadratic time, because there are $\Theta(n^2m^2)$ pairs of candidate substrings for S_1' and S_2'. But our intuition from suffix trees is that it can be made smaller. If the strings S_1 and S_2 are of size n and m respectively, then we can solve it in the same time bound as the global alignment problem, that is, *O(nm)*.

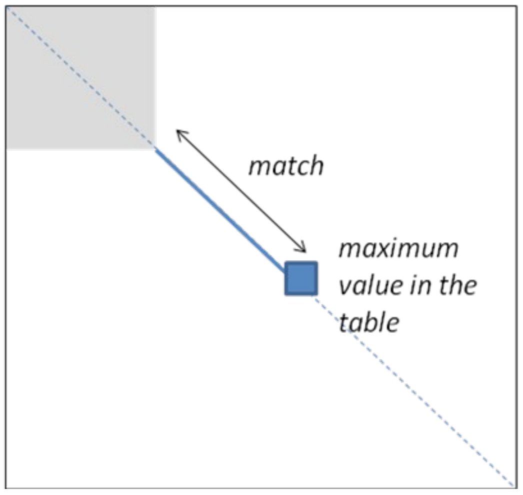
Intuitively, we can think of aligning S_1' with S_2', and making the rest of S_1 and S_2 into a global alignment without caring about the mismatches outside our local alignment. That is, no penalties will be assigned to the regions outside the local alignment. Another way to consider it, is to think of taking a candidate for S_2' and finding it in S_1:



When creating the dynamic programming table, we can fill the first row and column with zeros to avoid penalties outside the local alignment. The box with the maximum value in the last row gives the best local alignment score.



Now we can extend this to solve the local alignment problem. We need to create a 'contained alignment' in the dynamic programming table:



Although we could try filling the first row and column with zeros, how do we fill the shaded region? We can use the following recurrence to solve this problem:

$$V(i, j) = \max \begin{cases} 0, \\ V(i-1, j-1), \\ V(i, j-1) \\ V(i-1, j) \end{cases}$$

The zero means that you can start with a 0 where ever you started your local alignment. Finding the largest value in any cell in the table and backtracking from there until a 0 is found provides the result. Filling in each cell takes a constant number of operations, hence the total time to build the table is *O(nm)*. Finding the maximum value and backtracking also take the same order of time, hence the total time is *O(nm).*

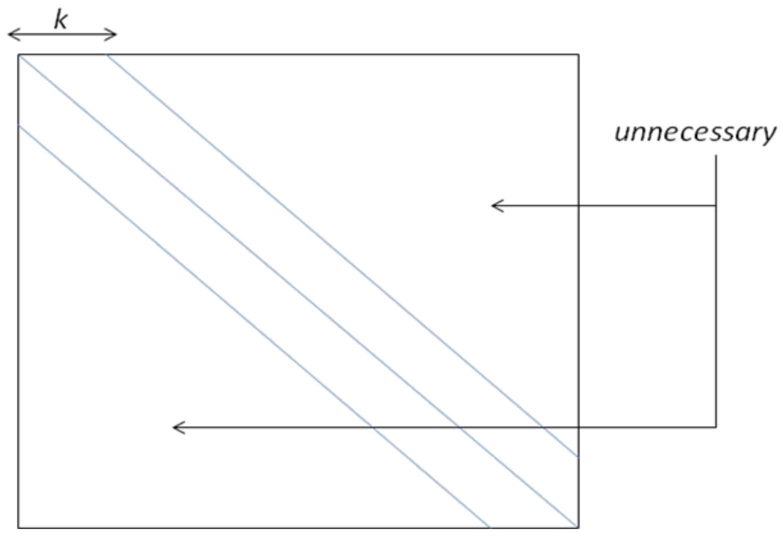
These algorithms are inefficient for aligning long strings, such as part of the human genome. Therefore we focus only on ‘relatively good’ alignments:

k-difference problem: *Align strings S_1 and S_2 with the total number of insertions, deletions and substitutions being at most k.*

k-mismatch problem: *Align strings S_1 and S_2 with at most k substitutions.*

Note that with the k-difference problem, you’re allowed to skip characters, while with the k-mismatch problem, you are not.

Consider building the dynamic programming table for the k-difference problem:



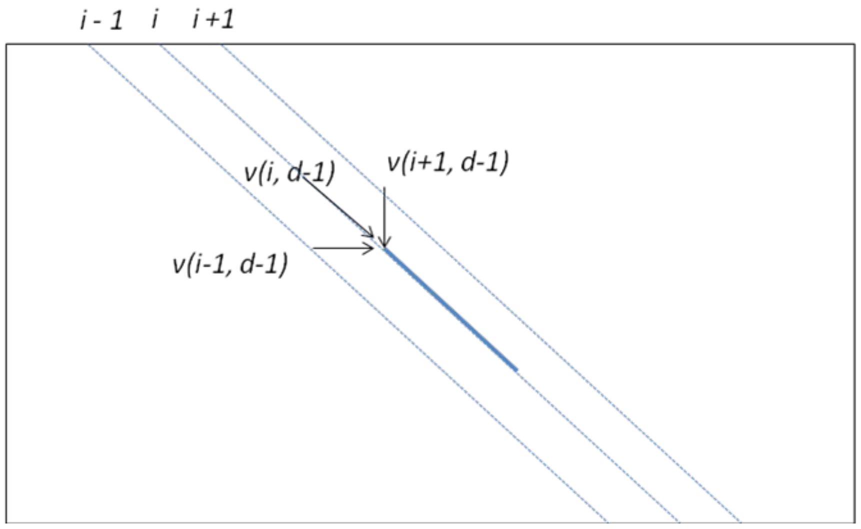
In our original alignment method, for an insertion or a deletion we would step off the diagonal. Therefore in the k-difference problem, it is only possible to step off the main diagonal at most k times. Consider the diagonals with a k-gap from the main diagonal: the regions below and to the right of these in the table are unnecessary. Thus we are filling only an *O(2kn)* portion of the table.

However, the last box in the table can still lead to more than k differences. If the best alignment found has differences less than or equal to k, we have found our alignment. But if it has more than k differences, then it is possible that there is a better alignment outside the region that was considered. This can be done by repeating the procedure with *k, 2k, 4k, …* : at any iteration, you’re guaranteed not to overshoot by more than the current k value.

Let k* be the best alignment value found in the last iteration. If the amount of work done in the i th round is *O(2k*n)*, then the amount of work for the i-1th round was *O(½[2k*n])*. Therefore the total amount of work is *O(2k*n + [2k*n]/2 + [2k*n]/4+…) = O(2[2k*n])*. In the worst case, this is not better than a full alignment.

The k-mismatch problem is more difficult because it allows gaps. But it is still possible to solve this in *O(kn)* time. Instead of looking at the neighboring cells in the dynamic programming table, consider a diagonal. Find a path on this diagonal to the farthest cell you can go with exactly d mismatches and gaps – such a path is called a d-path.

The solution runs in k iterations, where for each iteration we find the d path that we can go the farthest (i.e. if we find the farthest reaching cell in diagonal i then we cannot reach a cell further along diagonal i with any d-path). To find the path, we consider the farthest reaching d-1 length paths on the diagonals i-1, i, and i+1.



The recurrence relation can be written as

$$V(i,d) = f\{\ V(i-1, d-1),\ V(i, d-1),\ V(i+1, d-1)\ \}$$

The horizontal and vertical transitions (corresponding to V(i-1, d-1) and V(i+1, d-1) in the recurrence, respectively) indicate gaps, while the diagonal transitions (corresponding to V(i, d) in the recurrence) indicate mismatches. From that cell onwards, there is a match along diagonal i as indicated in the diagram. The value of d ranges from 0 to k. The alignment value will be in the lowest row.

For each d and i value, there are 3 cases to be considered in the recurrence, which takes constant time. The value of d ranges from 0 to k, and the number of values for i is *O(n+m)* (i.e. the number of diagonals), hence the total time taken to find the paths are of order *O(kn)*. Finding the identical substrings in each path can be done in constant time by using lowest common ancestor queries on a suffix tree. Thus the total time taken by the algorithm is *O(kn)*. The space used is also *O(kn)*.

This method is a combination of dynamic programming and exact matching.

Note: *Calculating the lowest common ancestor of two nodes in a tree can be done by preprocessing the tree in linear time with storing the tree using 0s and 1s to number paths, and then the result for a query is obtained in constant time by using bitwise operations on the stored information.*