Lecture 14: Approximate string matching

When we allow some small differences (substitutions, insertions and deletions) in string matching, the computational time increases. In Bioinformatics, the most well-known methods for comparing two strings (or sequences) are based on dynamic programming. These algorithms are frequently covered in Bioinformatics and introductory algorithms courses. We would omit the details. In the following, we describe a less well known algorithm called Shift-Or algorithm.

1 Shift-Or algorithm

Shift-Or algorithm (sometimes also called shift-and-algorithm) builds on exact string matching. It works well when the pattern is short. This algorithm builds a binary array $M$, which is $n$ by $m+1$ size. Recall $n$ is the size of the pattern $P$, and $m$ is the size of text $T$. Here, $M[i,j] = 1$ if first $i$ characters of $P$ match exactly the $i$ letters ending at position $j$ in $T$. Otherwise $M[i,j] = 0$. Clearly, $M[n,j] = 1$ if $P$ occur in $T$ with ending position $j$.

Now we define another length-$n$ vector $U(x)$ where $U(x)[i] = 1$ if char $x$ appear in position $i$ of $P$. E.g. let $P = abacdeab$, then $U(a) = 10100010$. Definition: $BitShift(j-1)$: shift column $j$ by 1 position to the right (towards larger index) and have a 1 at the first position. For example, if the column $j-1$ was 0010011, then after shifting, it becomes 1001001 (lost the original last bit).

Now we describe how $M$ is constructed efficiently. For column 1 of $M$: $M[1,1] = 1$ if $T[1] = P[1]$. For all other positions $i$, $M[i,1] = 0$. Then, $M[i,j]$ for j-th column depends on column $j-1$ and $U$: $M(j) = BitShift(j-1)$ AND $U(T(j))$. Why? Consider each $i \in [1,n]$. To make $M[i,j] = 1$, $T[j] = P[i]$ (i.e. $U(T(j))[i] = 1$). Also, $M(i-1,j-1)$ (when $i > 1$) should be 1: meaning the first $i-1$ letters of $P$ match $T$ ending at position $j-1$. Note that the size of $M$ is $\Theta(nm)$. However if $n$ is small (say $n \leq 32$, which is within a computer word), then that can be done in one operation using bit operation. Even if $n$ is a few words size, you are still only doing bit operations for a few more times and it is fast. Thus if pattern size $n$ is small, string matching can be done in $O(m)$ time.

Now what if we allow inexact match? Let us say that we allow a small number (say $k$) differences (including mismatches/insertion/deletions). For now we focus on mismatches. Let $M_k(i,j) = 1$ if at least $i-k$ of the first $i$ letters of $P$ match the length-$i$ substring ending at position $j$ of $T$. Clearly the above exact matching case is the $k = 0$ case. And so $M_k(n,j)$ tells whether there is hit of whole pattern at position ending at position $j$ in $T$ with at most $k$ chars. So now question is how to compute $M_k$. We already know $M_0$. Here is the basic observation. $M_k(j) = M_{k-1}(j)$ or $BitShift(M_{k-1}(j-1))$ AND $U(T(j))$ OR $M_{k-1}(j-1)$. YW: this was given in Gusfield’s book but it seems to me the last term should be $BitShift(M_{k-1}(j-1))$. To see why this works, we observe there are three conditions for having a match with at most $l$ differences ending at text position $j$: (1) first $l-1$ letters of $P$ match with $T$ with at most $l-1$ mismatches at $j$; (2) the first $i-1$ letters of $P$ matches $j$ letters of $T$ ending at position $j$ with cost $l$ and $P[i] = T[j]$; (3) the first $i-1$ letters of $P$ matches a substring of $T$ ending at $j-1$ with at most $l-1$ mismatches. In fact, this is very similar to the dynamic programming formulation for the edit distance. The entire algorithm uses $O(knm)$ bit operations. Again, if $n$ is small, the algorithm runs in $O(km)$ time.

2 Global string matching with bounded error

What about general matching differences (with spaces i.e. insertions and deletions)? The following is from Sect. 12.2.3 of Gusfield’s book: the $k$-difference global alignment.

Recall that we can use dynamic programming to align two strings of length $n$ and $m$ in time $O(nm)$. Given two strings, we now show how to align them in $O(km)$ time (if the difference is bounded by $k$).
We will still use the dynamic programming approach. But here, instead of filling the entire table, we only fill a small part of it. First, let us suppose $n \leq m$. If $m - n > k$, then there is no solution. So we assume $m - n \leq k$. In this case, define the main diagonal as the line $(i, i)$ in the table. Note that any cell that is off horizontal (or vertical) by more than $k$ from the main diagonal, then that is not a solution. Why? That would mean you will add more than $k$ spaces: each move away from diagonal would mean you add one space. Thus, we only need to fill the strip of $2k$ cells off the main diagonal. When filling the table, only use values within the strip, but ignore the values outside the strip. Now, if $k$-difference is possible, you will get an edit distance at most $k$ in this strip. Otherwise, if you get a value larger than $k$ (note this may not be the true distance since we may miss some true paths), there is no solution. This algorithm clearly runs in $O(kn)$ time. What if we do not know the edit distance but still want to save space? We can use a search trick: start from $k = 1$ and then double $k$ (i.e. letting $k' = 2k$) each iteration; each time, use the current $k$ to use the above procedure to see if exist a feasible solution (distance at most $k$); if so stop; This will stop with $k' \leq 2k^*$ (where $k^*$ the true distance). The time it takes is $O(n + 2n + ...k'm/2 + k'm) = O(km)$. 