Lecture 2: Classic String Matching ALgorithm

The brute-force string matching algorithm is the basis of several classic string matching algorithms, including the well-known KMP algorithm. The brute-force algorithm would scan (say from left to right) each position \( i \) in \( T \) to see if \( P \) matches \( T \) starting at \( i \). Clearly, this would run in \( O(mn) \) time. Obviously, this seems to be a lot of rooms for improvement and indeed that is the case.

1 The Knuth-Morris-Pratt (KMP) algorithm

Recall that \( P \) has length \( n \) and \( T \) of length \( m \). The naive algorithm may check all position of \( T \) to test the presence of \( P \). If a mismatch occurs, it will move to right by one position and repeat. In order to beat the \( O(mn) \) time, it is desirable to move at longer steps, which is what the KMP will do. We use an example \( P = \text{abcxabcde} \) from Gusfield’ book. Suppose mismatch at 8th (d) position and we need to move to right. Instead by 1, you can move by 4 (to the other a). Why? In order to find a match, the first letter of \( P \) should match \( a \) but \( bcx \) in \( T \) (implied by the previous matching of the first 7 characters) do not match with \( a \). Thus, the content of \( P \) tells a lot about what to do in determining how to move to the right.

Definition: \( sp(i) \)=length of longest “proper” suffix of \( P[1..i] \) that matches a prefix of \( P \). Here, proper means the exclusion of the trivial matching of the whole of \( P[1..i] \). Ex: \( P=\text{abcxabcde} \) sp(1..9)=\{0,0,0,0,1,2,3,0,0\}. A related definition: \( sp(i) \) is almost the same as \( sp(i) \) but has an additional requirement: \( P(i+1) \neq P(sp(i)+1) \). So sp(6)=2 but sp′(6)=0.

The key idea of the KMP algorithm is: suppose mismatch occurs at position \( i + 1 \) of \( P \) and position \( k \) of \( T \), so shift \( P \) to right (wrt \( T \)) s.t. character \( sp(i)+1 \) of \( P \) align with \( T[k] \) (and the shift size is \( (i + 1) - (sp(i)+1) = i - sp(i) \)) \geq 1. You should draw a picture and it becomes self-clear. In the above example, sp′(7)=3 if mismatch occur at 8, then shift by 7-3=4. Here, the stronger \( sp' \) ensures we will not have the same mismatch at \( i + 1 \). Also, why does the KMP requires the longest proper suffix? And why the shift is always at least one? This should be clear if you draw a picture and think about it. With the new shift, we may shift over by more than 1 places and also the first portion of \( P \) is often already known to be matched to \( T \) and so no need to compare. Therefore, comparison starts from \( T[k] \) again.

Why does KMP work? That is, why long shifts do not miss any potential matches? The key is based on the definition of \( sp'(i) \): if we ever miss by jumping for too long, then we should have a larger value \( sp'(i) \) than what is said in \( sp'(i) \) now. This can be shown by again drawing a picture. Let us say we miss a match at position \( i'' \) when KMP shifts from \( i \) to \( i' \). To be precise, the mismatch occurs at position \( k \) of \( T \) and character \( sp'(i) \) is aligned with \( T[k-1] \). Then KMP moves \( P \) s.t. \( P[i'] \) now is aligned with \( T[k-1] \), where \( i' = sp'(i) \). The danger is there exists \( i'' > i' \) and a full match occurs when \( P[i''] \) is aligned with \( T[k-1] \). It is easy to see that both \( P[1..i'] \) and \( P[1..i''] \) match some proper suffix of \( P[1..i] \) and \( P[1..i''] \) is longer than \( P[1..i'] \). Now I claim \( P[i''+1] \neq P[i+1] \) and if so, \( sp(i) \geq i'' > i' = sp'(i) \) which is a contradiction. Why this is the case? This is because we assume a full occurrence of \( P \) when \( P[i''] \) is opposite to \( T[k-1] \) and thus \( P[i''+1] \) matches \( T[k] \). but we know \( T[k] \neq P[i+1] \) by definition of \( sp'(i) \). So \( P[i''+1] \neq P[i+1] \).

Now time analysis. I claim KMP performs no more than \( 2m \) character comparisons. The Key is noting that \( P \) shifts to right by no more than \( m \) times. Each shift will involves at most one previously compared characters (the one where we have a mismatch), and so the total number of
comparisons is number of shifts (where a letter in $T$ can be repetitively compared) plus $m$ (those compared involving uncomapred characters in $T$), which is at most $2m$.

Now next question: how to get the value $sp'(i)$? These values can be collected by Z algorithm. With this definition, $sp'(i) = Z_j$ where $j$ is the smallest $j$ whose Z-box ending at $j$ (or $Z_j = i - j + 1$). If no such $j$ exists, then $sp'(i) = 0$. Why? Note it is the longest match in Z definition and so the character to the right by 1 will not match. Also, we pick $j$ to the earliest and thus the match will be longest. This can be done by scanning from the right to left and update the $sp'$ values by the proper Z values.

2 The Boyer-Moore algorithm

The Boyer-Moore algorithm is one of the best in exact matching. The basic idea of Boyer-Moore is also to improve original naive algorithm by moving by more than 1 positions each time. That is, Boyer-Moore has similar basic structure as naive but has rules to allow it move faster (of course without missing any true matches). These tricks allow worst case $O(m + n)$ time and in practice often fast (called sub-linear). That is, Boyer-Moore may skip some parts of $T$ by “jumping” over them. We should pause for a moment. Is sub-linear feature is really something new? Can the KMP algorithm also run in the sub-linear fashion? I think this worths some thoughts.

The first feature of Boyer-Moore is, for each matching of $P$ against somewhere in $T$, compare from right to left. This looks odd initially: why does it matter? A moment of thoughts suggests sometime it indeed can help. Suppose letter $y$ of $P$ and symbol $x$ of $T$ mismatch at position $p$, which is the the rightmost position. Clearly we can move $P$ so the rightmost $x$ of $P$ matches that $x$ of $T$ (i.e. move by more than 1). Or even better, if there is no $x$ at all in $P$, we can just move $P$ over $x$ of $T$. Formally, for each $x$, we define $R(x)$ to be the rightmost position of $x$ in $P$, and $R(x) = 0$ if $x$ does not appear in $P$. $R(x)$ can be processed by linear scan.

Now the first rule of Boyer-Moore: the bad character rule. When $P[i]$ mismatch $T[k]$, shift $P$ by $max(1, i - R(T[k]))$ positions. That is, if $R(T[k]) < i$ (i.e. the rightmost of $T[k]$ is to the left of $i$), then we can move more to make that $x$ of $P$ aligns with $T[k]$. Now pause for a moment. If $R(T[k]) > i$, can only move right by 1 since we do not know where is $x$ in $P$ to the right. This can be tightened: if we know the closest $T[k]$ to the left of $i$, then we can move to there. If alphabet size is small, this can be done. See Gusfield’s book for more details.

Now the second rule: the good suffix rule. This is key of BM. Suppose a suffix of $P$ matches somewhere in text and them mismatch at $x$. Where should we shift $P$ to? After shifting for a match, then there must be somewhere in $P$ matching this suffix and with a different preceding symbol. So, we should move $P$ to right so that the rightmost such occurrence of suffix $t$ align with matched part in $T$. It is clear this shifting rule will never miss any matches. For example, let $T = prstabstubabvqxrst$ and $P = qcabdabdab$. We are aligning $P[1]$ with $T[2]$ at the moment and recall we compare from the right to left. How much should we shift? It seems we can shift by 3 positions at least. In fact, we can shift by 6: shift of 3 will get the same mismatch (d).