1 Local string matching with bounded error

Now what about local alignment with at most \( k \) difference (not global alignment)? The following is based on Section 12.2.4 of Gusfield’s book.

Clearly the previous global bounded error alignment idea does not work. Definition: the main diagonal \((i, i)\) is called diagonal 0. The diagonals above the main diagonal called diagonal \(i\); below, called diagonal \(-i\). Here, row corresponds to pattern, and column correspond to \(T\). Thus, row 0 has value 0. Definition: \(d\)-path: a path starts from row 0 and has exactly \(d\) mismatches/spaces. Definition: \(d\)-path is farthest-reaching at diagonal \(i\) if this \(d\)-path ends at diagonal \(i\) and column \(c\) along diagonal \(i\) is no smaller than any other \(d\)-path ending in diagonal \(i\). (That is, no other \(d\)-path reaches further along diagonal \(i\)). Note by requiring farthest, we ask for the best among all possible \(d\)-path.

Basic idea: consider each \(d = 0\) \ldots \(k\); for each \(d\), find the farthest reaching path ending at each possible diagonal (ranged from \(-n\) to \(m\)). This iteration will run in \(O(n + m)\) time. In details: (1) Initialize \(d = 0\). This is perfect matching case: for diagonal \(i\), match exact \(P[1..n]\) with \(T[i,m]\) (start at position \(i\)). That is just the longest common extension. This is done in \(O(1)\) for each \(i\). So this iteration takes \(O(m)\) time. (2) For \(d > 0\), the idea is kind of like original dynamic programming: the farthest \(d\)-path on diagonal \(i\) may be (i) a farthest reaching \((d-1)\)-path along diagonal \(i+1\) and then follow a space to diagonal \(i\), or (ii) a farthest reaching \(d-1\)-path along \(i-1\) and then a horizontal space or (iii) farthest reaching \(d-1\)-path along diagonal \(i\) and then a mismatch followed by a common extension along the diagonal.

This is just like the original DP: take the one giving the farthest reaching path. Time: clearly \(O(km)\): each farthest reaching path can be found in \(O(1)\) for each \(i\) during an iteration.

2 BYP: a partition-based method

This section is based on Section 12.3 of Gusfield’s book.

The general partition-based approach for finding matches with at most \(k\) differences works as follows:

Partition \(T\) or \(P\) into regions of some length \(r\).

Search Search \(T\) to identify possible regions that may contain \(P\).

Check For each survived intervals, perform dynamic programming alignment.

There are many different variations of this partition-based approach. We first describe the so-called BYP method. This is a simple idea: break \(P\) into consecutive \(r\)-length regions, where \(r = n/(k + 1)\). So there are \(k + 1\) regions of length-\(r\) in \(P\). Since the max number of errors is \(k\), at least one \(r\)-region appears exactly in \(T\). Then, we find which positions of \(T\) matches one of the \(r\)-region. For these candidate positions, we use simple dynamic programming to check each candidate positions. The finding of candidates can be done with the Aho-Corasic algorithm using the keyword tree, which takes \(O(m + n)\) time.

Analysis: first how many candidates can there be? Assume \(T\) is composed symbols drawn uniformly from alphabet. Then, for each \(r\)-region, the expected number of matchings is roughly \(m/\sigma^r\), where \(\sigma\) is the size of alphabet. We have \(k + 1\) \(r\)-regions, and thus the expected number of hits is \(m(k+1)/\sigma^r\). For each candidate, it takes \(O(n^2)\) time. So total time is: \(mn^2(k+1)/\sigma^r\). Note \(k+1 < n\). So we can replace \(k\) by \(n-1\). We now check which \(r\) will make \(mn^3/\sigma^r = cm\) for some \(c\). Then, \(r = \log_\sigma(n^3) - \log_\sigma(c)\). But \(r = n/(k+1)\). Combine these two and we obtain the time is \(O(m)\) when \(k = O(n/logn)\).
3 The Chang-Lawler method

Consider the $k$-mismatch version (i.e. no spaces). We break $T$ into length $r = n/2$ regions. This way, for any pattern $P$ to occur, some full regions of $T$ will match against $P$ with at most $k$ mismatches. Thus, we can first filter out regions of $T$ that can not matches with $k$ mismatches. How to test whether a region of $T$ survives? Here is a simple approach: we test whether we can find at most $k+1$ segments of perfect match of $P$. Here, do not concern about the order just the number. In other words, the segments can match anywhere in the $P$. This can be done by the so-called matching statistics. Briefly, we let $ms(i)$ be the length of the longest matches of the substring starting at position $i$ of $T$ in $P$ somewhere. I will not go to the details here but $ms(i)$ can be collected for all positions of $T$ in $O(m)$ time. Once a region survive, we apply the usual dynamic programming on the nearby $n/2$ to the left and $n/2$ to the right (and so the total length of the region to compare is $3n/2$). Note: this can be done $O(kn)$ time (because the total region is $O(n)$ and difference is bounded by $k$).

Analysis: there are $2m/n$ regions of $T$. For each such region $R$, the filtering check of the region (used to test the segment of $P$ matches) time for this is $O(j' - j*)$ where $j'$ is the ending position of $R$ search and $j*$ is the initial position. Note the expected value of $j' - j*$ is less than $k$ times the expected value of $ms(i)$ (which is denoted as $E(M)$). So the expected number of survived regions is $O(2mkE(M)/n)$. Here is a simple lemma (Lemma 12.3.3): $E(M) = O(log_{\sigma} n)$. This is not hard: there are roughly $n$ substrings of length $d$ in $P$, and there are $\sigma^d$ substrings of length $d$ possible. So, for any length-$d$ substring, chance of finding a match in $P$ is about $n/\sigma^d$. Let $X$ be a random variable that is equal to $log_{\sigma}(n)$ for $ms(i) \leq log_{\sigma}(n)$, and equal to $ms(i)$ otherwise. So, $E(M) < E(X) < log_{\sigma}(n) + \sum_{i=log_{\sigma}n}^{\infty} \frac{1}{\sigma^i} = log_{\sigma} n + 2.$