Lecture 4: Classic String Matching Algorithm: Karp-Rabin and Aho-Corasick Algorithms

1 The Karp-Rabin algorithm

The Karp-Rabin algorithm uses a different idea doing exact string matching: it is a randomized algorithm. For simplicity, treat \( T \) and \( P \) as binary. The idea is to treat \( P \) and portion of \( T \) to match \( P \) as binary integers: 

\[
H(P) = \sum_{i} 2^{n-i} P[i].
\]

And do the same for the portion of \( T \) being compared with \( P \). Clearly \( P \) occurs in \( T \) at position \( i \) iff \( H(P) = H(T_i) \). However, computing \( H \) for long \( P \) is hard: even for 100 character long, \( 2^{100} \) is huge. Karp and Rabin found a simple algorithm that makes this both theoretical and practically interesting. The idea is simple: they use a hashing approach with theoretical justification. That is, it ensures with proof that it can find the matching with certain chance of failure (which means the algorithm can declare a match possible but really not there, i.e. false positive). The hash is called fingerprint here. For a prime number \( p \), \( H_p(P) = H(P) \mod p \). The first observation is, \( H_p(P) \) is easy to compute: it does not use too large number (at most int up to value \( 2p \) is used). The idea is based on Horner’s rule (a well-known in evaluating polynomial): 

\[
H_p(P) = ((P[1]*2 \mod p + P[2])*2 \mod 2 + P[3])*2 \mod p \ldots
\]

That is, each time we do a multiplication we do a mod. In other words, we perform the modulo operation early. And thus only the maximum value of \( 2p \) value is needed. For \( H_p(T_i) \): we can compute based on value from \( H_p(T_{i-1}) \):

\[
H_p(T_i) = (2H_p(T_{i-1} \mod p) - (2^n \mod p) * T[i - 1] + T[i + n - 1]) \mod p
\]

Here, \( 2^n \mod p \) can also be computed with mod introduced earlier: \( 2^n \mod p = (2^n - 1 \mod p) \mod p \).

This makes computing fingerprint easier.

However, we can have false positives: fingerprint matches may not always suggest text matches. Many approaches take the following approach: if fingerprint matches, then just do a direct re-examination one by one. However, in worst case you may still need \( O(nm) \): most positions of \( T \) give false matches and each takes \( O(n) \) to examine. Karp-Rabin is more rigorous in the sense that it has a justification why the algorithm is ensured to only have a small chance of making errors. The key is choose \( p \) small (so arithmetic does not involve larger numbers) while \( p \) is also large enough to make false positive chance small.

\textit{Definition:} \( \pi(u) \) is the number of primes less or equal to an integer \( u \). The famous prime number theorem states that \( \pi(u) \approx \frac{u}{\ln(u)} \). Now another fact about prime number is the Lemma 4.4.2:

\textbf{Lemma 1.1.} If \( u \geq 29 \), the product of all primes that are less or equal to \( u \) is larger than \( 2^u \).

The consequence is, if \( u \geq 29 \) and \( x \leq 2^n \), then \( x \) has less than \( \pi(u) \) distinct prime divisors. Why? Suppose \( x \) has \( k > \pi(u) \) distinct prime divisors \( q_1, q_2, \ldots, q_k \). Then, \( 2^n \geq x \geq \prod_{i=1}^{k} q_i \), which is greater than the product of the first \( \pi(u) \) prime numbers (since \( k > \pi(u) \)). However, by the previous lemma, the product of the first \( \pi(u) \) prime numbers is at least \( 2^u \). So we have \( 2^u > 2^n \) which is a contradiction. \textbf{YW:} note this is the point I did not explain in details in class.

Now we have our main theorem.

\textbf{Theorem 1.2.} Suppose \( nm \geq 29 \), and \( I \) is some integer, and \( p \leq I \) is a random prime number. Then probability of false positive of declaring \( P \) match \( T \) is no bigger than \( \frac{\pi(nm)}{\pi(I)} \).
Proof. Let $R$ be the set of positions in $T$ where $P$ does not match $T_s$ (i.e. starting at position $s$), and so $H(P) \neq H(T_s)$. So consider $\prod_{s \in R} (H(P) - H(T_s))$. Each term is at most: $2^n$ (recall $P$ and $T_s$ are binary numbers of length $n$), and thus the (absolute value) of this product is at most $2^{nm}$ and so it has fewer than $\pi(nm)$ prime divisors. Now, suppose $p$ is a false match choice and thus $p$ divides $H(P) - H(T_s)$ for some $s$, and $\frac{\text{error probability}}{\text{hit probability}} = p$. We omit more detailed analysis.

Now if we choose proper $I$, we can ensure the false positive rate is low. Claim: when $I = nm^2$, the false positive probability is at most $2.53/m$ (and note $m$ is usually large). Note $nm^2$ is not too large (in terms of bits). This is because: error probability $p_e$ is at most $\pi(nm)/\pi(nm^2) \approx 1.26^{\frac{\text{nm} \ln((nm)^2)}{\text{nm} \ln(nm)}} \leq 1.26 * 2/m \leq \frac{2.53}{m}$. Here we use a particular form of prime number theorem: $\pi(n) \leq 1.26 \frac{n}{\ln(n)}$. When $m$ grows, the error probability is low. In case $m$ is relatively short, that can be done by picking $K$ random primes and combine the results, and then error probability becomes $p_e^K$. When we choose large enough random $p_i$, effectively we have zero probability of being wrong. We omit more detailed analysis.

Still, there is a remaining issue. If a probable match is declared, we need to $O(n)$ time to check each position explicitly reported by the KR. That is, the worst time seems to be still $O(mn)$. But there is a trick to even making the worst-case time linear. Well with a little caveat: it will ensure checking to find at least one false match and stop (but may miss other false matches). This is well acceptable: the chance of error is small for Karp-Rabin and so more than one false positives are even less likely. The idea is related to the periodicity analysis. To apply periodicity, we need to consider closer matching positions. So we divide potential matching position found by Karp-Rabin as $l_1, l_2, \ldots$ into phases where each phase has two consecutive positions differ by at most $n/2$ (i.e. two matches will overlap by at least half of $n$, a common trick for periodicity checking). We will process each phases (called run) separately. For a single run, explicitly compare first two matches and if does not match, then declare false positive found (and stop). Otherwise, both positions match and this implies pattern $P$ is semi-periodic with length $d = l_2 - l_1$. Note $d$ must be the smallest period: otherwise between $l_1$ and $l_2$ we will have more matches (but this is impossible since Karp-Rabin will not miss any matches; it can only have false positives). Then, each consecutive pair of hits within this run $l_i$ and $l_{i+1}$ must have $l_{i+1} - l_i = d$. Why? This is due to the GCD lemma.

**Lemma 1.3.** The GCD lemma. If a string $\alpha$ is semi-periodic with period of length $p$ and also with period of length $q$. Then $\alpha$ is also semi-periodic with the greatest common divisor of $p$ and $q$.

By the GCD lemma, if we have different $d_1 < n/2$, then $P$ must be semi-periodic with $\text{GCD}(d, d_1) < d$ since $d_1 + d < n$. So we first ensure each run, they all separate by $d$ positions (all supposed matches). This alone will reveal many false ones. That is, if a hit found by Karp-Rabin does not follow this pattern in this run, we know this is a false positive. But what if it passes this test? Then, to check each position in this run, only compare the last $d$ symbols of the supposed matching position of $T$ with the last $d$ symbols of $P$ (due to periodicity of $P$ and the close positions of matching). If matching, yes it occurs. Otherwise, the hit is a false positive. Time analysis: no position in $T$ examined by more than twice in a run. Moreover, runs are separated by at least $n/2$, and the overlap between two consecutive runs is at most $n/2$ long (since $P$ is at least $n$). So no positions of $T$ will be compared by more than two runs. Total number of comparisons is thus $O(m)$. 


2 The Aho-Corasick algorithm

Aho-Corasick algorithm works well for the problem of matching multiple patterns $P$ with $T$. Why? Sometimes there are fixed set of patterns (say words in dictionary), and $T$ can vary a lot. We may want to know what words appear in $T$. Suppose there are $z$ patterns, $n$ is the total length of $P$ and $m = |T|$. Simple approach: search each of $z$ patterns on $T$, and this takes $O(n + zm)$ time. When $z$ is large, this is not very efficient. Aho-Corasick algorithm has the $O(n + m + k)$, where $k$ is the number of occurrences of some $P$.

The idea is using a tree structure called “keyword tree” to store information about $P$. Frequently used idea (e.g. suffix tree). In the keyword tree, nodes may be labeled by pattern id; edge label is a single symbol; two edges out of a node cannot start with the same symbol; a path from root to a pattern spells out the pattern. Example: you should try to build the keyword tree $P = \{potato, poetry, pottery, science, school\}$. First, building Keyword tree is easy: $O(n)$ if fixed size alphabet (many satisfy this property). This means we just process each pattern word one by one and create new branches whenever possible. Easy to give algorithm that insert each $P$ into the keyword tree one by one. However, it is unclear why the keyword tree helps: if search for $P$ in some places of $T$. We can follow paths of $T$ into the keyword tree (the path is unique). But we seem to need to do this for each start position of $T$ and that seems to give $O(nm)$. We will come back to this issue in the next lecture.