The Exact and Approximate String Matching Algorithms

李家同
• In this talk, we discuss two problems:
• Problem 1: The Exact String Matching Problem
• Problem 2: The Approximate String Matching Problem
• We shall discuss Problem 1 first.
Problem 1

The Exact String Matching Problem:

We are given a text string \( T = t_1 t_2 \cdots t_n \)
and a pattern string \( P = p_1 p_2 \cdots p_m \)
and we want to find all occurrences of \( P \) in \( T \).
Consider the following example:

\[ T = AGCCTAAGCTCCTAAGGTC \]

\[ P = CCTA \]

There are two occurrences of \( P \) in \( T \) as shown below:

\[ \underline{AGCCTAAGC} \underline{TCTAAGGTC} \]
A brute force method for exact string matching algorithm:

\[ T = ACCACTAGA \]
\[ P = ACTA \]

ACTA

ACTA

ACTA

ACTA
If the brute force method is used, many characters which had been matched will be matched again because each time a mismatch occurs, the pattern is moved only one step.
Let us consider the following case. The mismatch occurs at \( p_{11} \).
That is, \( P(1,10) = T(4,13) \).

\[
T = \text{GCCTAAGCTCCTCAGTC...}
\]

\[
P = \text{TAAGCTCCTCCA}
\]

\[t_{14}\]

\[p_{11}\]
Besides, no suffix of $T(4,13)$ is equal to any prefix of $P(1,10)$ which means that if we move $P$ less than 10 steps, there will be no matching. We may slide $P$ all the way to the right as shown below.

\[
T = \underline{GCCTAAGCTCCTCAGTC}... \\
\quad t_{14}
\]

\[
P = \underline{TAAGCTCCTCCA} \\
\quad p_{11}
\]

Besides, no suffix of $T(4,13)$ is equal to any prefix of $P(1,10)$ which means that if we move $P$ less than 10 steps, there will be no matching. We may slide $P$ all the way to the right as shown below.

\[
T = \underline{GCCTAAGCTCCTCAGTC}... \\
\quad t_{14}
\]

\[
P = \underline{TAAGCTCCTCCA} \\
\quad p_{11}
\]
For the following case, since there is a suffix of the window in $T$, namely CCGA, which is a prefix of $P$, we can only slide the window such that the prefix matches with the suffix of the window, as shown below.

$$T = \text{GCCCGA\_CTCCGAATCC...}$$
$$P = \text{CCGAATCC\_CGAGA}$$

$$T = \text{GCCCGA\_CTCCGAATCC...}$$
$$P = \text{CCGAATCC\_CGAGA}$$
There are many exact string matching algorithms. Nearly all of them are concerned with how to slide the pattern.

In the following, we shall list the important ones.
Boyer and Moore Algorithm (BM77, 1, 2, 2-1, 3-1)
Colussi Algorithm (C91, 1)
Crochemore and Perrin Algorithm (CP91, 5)
Galil Gianardo Algorithm (GG92, 1)
Horsepool Algorithm (H80, 2-2)
Knuth Morris and Pratt Algorithm (KMP77, 1)
KMP Skip Algorithm (CTJ98, 2)
Max-Suffix Matching Algorithm (R2002, 2,3)
Morris and Pratt Algorithm (MP70, 1)
Navarro and Raffinot Algorithm (NR98, 1)
Nebel Algorithm (2-2)
Quick Searching Algorithm (S90, 2-2)
Raita Algorithm  (R92, 2-2)
Reverse Factor Algorithm  (CCGJLPR94, 1)
Reverse Colussi Algorithm (C94, 1,2)
Self Max-Suffix Algorithm  (R2002, 1)
Simon Algorithm  (S93,1)
Skip Search Algorithm  (CTJ98, 2-2, 4)
Smith Algorithm  (S91, 2-2)
Tuned Boyer and Moore Algorithm  (HS91, 2-2)
Two Way Algorithm  (CP91, 5)
Uniqueness Algorithm  (CL2007, 3-1, 3-2, 3-3)
Wide Window Algorithm  (HFS2005, 4)
Zhu and Takaoka Algorithm  (ZT87, 2)
Although there are so many algorithms, there are some common rules.

It is surprising that all of these algorithms are actually based upon these rules.
Table of Rules for Exact String Matching

• Rule 1: The Suffix to Prefix Rule
• Rule 2: The Substring Matching Rule
  – Rule 2-1: Character Matching Rule
  – Rule 2-2: 1-Suffix Rule
  – Rule 2-3: The 2-Substring Rule
• Rule 3: The Uniqueness Property Rule
  – Rule 3-1: Unique Substring Rule
  – Rule 3-2: Longest Substring with a Unique Character Rule
  – Rule 3-3: The Unique Pairwise Substring Rule
• Rule 4: The Two Window Rule
• Rule 5: Non-Tandem-Repeat Rule
• Nearly all of the exact string matching algorithms use the slide window approach.
• Whenever a mismatching is found, the pattern is moved to the right.
Rule 1: The Suffix to Prefix Rule

- For a window to have any chance to match a pattern, in some way, there must be a suffix of the window which is equal to a prefix of the pattern.
The Implication of Rule 1:

- Find the longest suffix $U$ of the window which is equal to some prefix of $P$. Skip the pattern as follows:

\[\text{T} \quad \text{P} \]
Example

\[ T = GCATCGACAGACGTATAAGCTACGTACG \]

\[ P = \textcolor{red}{GACGGATCA} \]

\[ \because \text{The longest suffix of the window which is equal to a prefix of } P \text{ is } "GAC" = P(1, 3), \]

\[ \text{slide the window by 6.} \]

\[ T = GCATCGACAGACGTATAAGCTACGTACG \]

\[ P = \textcolor{red}{GACGGATCA} \]
The MP Algorithm

- Assume that a mismatch occurs as shown below and we have already found the longest suffix of the matched string $V$ which is equal to a prefix of $P$.

<table>
<thead>
<tr>
<th>T</th>
<th>V</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>V</td>
<td>b</td>
</tr>
</tbody>
</table>
The MP Algorithm

- Skip the pattern by using Rule 1.
But, if we have to do the finding of the longest suffix in run time, the algorithm will be very inefficient. A preprocessing can eliminate the problem because $u$ also exists in $P$. 
MP Algorithm

- The MP Algorithm pre-processes the pattern $P$ and produces the prefix function to determine the number of steps the pattern skips.
Example

\[ T = \text{GCATCGACGAGAGTATACAGTG} \]
\[ P = \text{GACGACGAG} \]

\[ \therefore P(1, 2) = P(4, 5) = \text{`GA'}, \text{ slide the window by 3.} \]

\[ T = \text{GCATCGACGAGAGTATACAGTG} \]
\[ P = \text{GACGACCGAG} \]

Note that the MP Algorithm knows it can skip 3 steps because of the preprocessing.

The prefix function can be obtained recursively.
The KMP Algorithm

- The KMP algorithm makes a further checking on $P$. If $x = y$, skip further.
Example

\[ T = GCATCGACGAGAGTATACAGTACG \]
\[ P = \underline{GACGACGAG} \]

\[ P(1, 2) = P(4, 5) = \text{‘GA’}. \text{ But } p_3 = p_6 = \text{‘C’}. \]

Slide the window by 5.

\[ T = GCATCGACGAG_{\text{AG}}GAGTATACAGTACG \]
\[ P = \underline{GACGACGAG} \]
Simon’s Algorithm

• Simon’s Algorithm improves the KMP Algorithm a little bit further. It checks whether $y$ which is after prefix $u$ in $P$ is the character $x$ after $u$ in $T$. If not, skip further.
• Example

\[ T = GCATCGAGGAGGAGTATACAGTACG \]
\[ P = \underline{GAGGACGAG} \]

\[ P(1, 2) = P(4, 5) = 'GA', \text{ and } p_3 = 'G' = t_{11}. \]
Slide the window by 3.

\[ T = GCATCGAGGAGGAGGAGTATACAGTACG \]
\[ P = \underline{GAGGACGAG} \]
The Navarro and Raffinot Algorithm

- \( u \) is the longest suffix of the window which is equal to a prefix of \( P \). The Navarro and Raffinot Algorithm uses Rule 1. This algorithm also uses a preprocessing mechanism. But the finding of \( u \) is still done during the run-time, with the result of preprocessing.
• Example

$T = GCATCGAGGAGAGGAGTATACAGTACG$

$P = GAGCGAAC$

The longest prefix of $P$ is “GAG”, which is equal to a suffix of the window of $T$. Slide the window by 5.

$T = GCATCGAGGAGAGGAGTATACAGTACG$

$P = GAGCGAAC$
• The Reverse Factor Algorithm uses Rule 1, by incorporating the idea of suffix trees.
Self Max Suffix Algorithm

• A maximal suffix of a string is a suffix which is lexicographically maximal of all suffixes of a string.

• The maximal suffix of string \( w \) is denoted by \( \text{MaxSuf}(w) \)

• A string \( w \) is said to be self-maximal if \( \text{MaxSuf}(w) = w \).

Example:
\( w = \text{TCAAAATATCA} \) is a self-maximal string.
Since $P$ is a self-maximal string, from the prefix $u$, we may conclude that $y > x$. Contradiction!
• The Self Max Suffix Algorithm uses Rule 1, by noting in a special case, we don’t need to store any table for deciding how many steps we may jump.

• The number of steps we jump is done in the run-time, and we use a function, called Naive-Period function to compute.
Rule 2: The Substring Matching Rule

- For any substring $u$ in $T$, find a nearest $u$ in $P$ which is to the left of it. If such an $u$ in $P$ exists, move $P$ such then the two $u$’s match; otherwise, we may define a new partial window.
Boyer and Moore Algorithm

- The Good Suffix Rule 1 in the BM Algorithm uses Rule 2, except $u$ is a suffix.

- If no such $u$ exists to the left of $x$, the suffix $u$ in $P$ is unique in $P$. This is a very important property.
• Example

\[ T = GCATCGAGGA_GAGTATACAGTACG \]
\[ P = \textcolor{red}{GGAGC}CCGAG \]

\[ \therefore P(2, 4) = 'GAG' \]

Slide the window by 5.

\[ T = GCATCGAGGA_GAGTATACAGTACG \]
\[ P = \textcolor{red}{GGAGC}CCGAG \]
Rule 2-1: Character Matching Rule (A Special Version of Rule 2)

• For any character $x$ in $T$, find the nearest $x$ in $P$ which is to the left of $x$ in $T$. 

\[ T \quad x \]

\[ P \quad x \]
Implication of Rule 2-1

- Case 1. If there is an $x$ in $P$ to the left of $T$, move $P$ so that the two $x$’s match.

\[
\begin{array}{c}
T & \quad \quad \quad \quad x \\
\hline
P & \quad \quad \quad \quad x \\
\end{array}
\]
• Case 2: If no such $x$ exists in $P$, move $P$ in the following way.
Boyer and Moore Algorithm

- The Bad Character Rule in BM Algorithm uses Rule 2-1 in a limited way except it starts from the end as shown below:

\[ \begin{array}{c|c|c}
T & x & u \\
\hline
P & x & y & u \\
\end{array} \]
• Example

\[
T = GCATCGAGGAGCGTATACAGTACG
\]

\[
P = \underline{GAGGCCGCG}
\]

\[
\therefore p_2 = 'A', \text{ slide the window by 4.}
\]

\[
T = GCATCGAGGAGCGTATACAGTACG
\]

\[
P = \underline{GAGGCCGCG}
\]
Rule 2-2: 1-Suffix Rule (A Special Version of Rule 2)

- Consider the 1-suffix $x$. We may apply Rule 2-2 now.
The Skip Search Algorithm

• The Skip Search Algorithm (Christian, Thierry and Joseph Algorithm) uses Rule 2-2 together with Rule 4 in a very clever way.
• The Horspool Algorithm, Quick Search Algorithm, Raita Algorithm, Tuned Boyer-Moore Algorithm, Nebel Algorithm, Liu and Lee Algorithm and Smith Algorithm all use Rule 2-2, in some way. The difference is on which character to select.
Rule 2-3: The 2-Substring Rule (A Special Version of Rule 2)

• Consider the following case:
• We match from right to left

\( T = GAATCAATCATGAA \)
\( P = TCATGAA \)

\( T = GAATCAATCATGAA \)
\( P = TCATGAA \)
• Berry and Ravindran Algorithm is based up on Rule 2-3.
Rule 3-1: Unique Substring Rule
(The Lu’s Algorithm)

- The substring \( u \) appears in a prefix of \( P \) exactly once.
- If the substring \( u \) matches with \( T(i, j) \), no matter whether a mismatch occurs in some position of \( P \) or not, we can slide the window by \( l \). Here, we are using Rule 1.

The string \( s \) is the longest suffix of \( u \) which is equal to a prefix of \( P \).
• Note that the above rule also uses Rule 2.
• It should also be noted that the unique substring is the shorter and the more right-sided the better.
• A short $u$ guarantees a short (or even empty) $s$ which is desirable.
• Example

\[ T = GCATCGAGGCGAGTATACAGTACG \]
\[ P = GGAGGCCGAG \]

Unique substring \( u = \text{‘CG’} \)

\( \therefore u = T(10, 11) = \text{‘CG’}, \) and a mismatch occurs in \( p_1 \).
Within CG, suffix G is a prefix of \( P \).
Slide the window by 6.

\[ T = GCATCGAGGGCGAGTATACAGTACG \]
\[ P = GGAGGCCGAG \]
Boyer and Moore Algorithm

• In Boyer and Moore Algorithm (BM Algorithm), there is a Good Suffix Rule 2 which is a combination of Rule 2 and Rule 3-1.
• The Good Suffix Rule 2 is used after the Good Suffix Rule 1, which is actually Rule 2-1, fails to work.
• When Good Suffix Rule 1 fails, it means that the suffix $u$ in $P$ is unique. That’s why Rule 3-1 can be used.

No such $u$ exists.
• Example

$T = GCATCGGAGGACTATACAGTACG$

$P = GACGACGGAC$

∵ The suffix “GGAC” of window is unique in $P$, and $P(1, 3) = GAC$ is a suffix of “GGAC”, slide the window by 7.

$T = GCATCGGGAGGACGTATACAGTACG$

$P = GACGACGGAC$
Rule 3-2: Longest Substring with a Unique Character Rule

- Find the longest substring of $P$, $P(i, j)$, where $p_j$ is the unique character in $P(i, j)$. Thus $p_{i+1} = p_j$
- If $p_j$ matches with $t_k$, we can slide the window by $j-i+1$ in next step.

![Diagram](image-url)
• Example

$T = GCATCGCGGGCA\underline{G}GTATAACAGTACG$

$P = \underline{GGAGGCCGAG}$

The longest substring $P(4, 8) = \text{GCCGA}$, which has a unique character ‘A’ in $P(4, 8)$.

$\therefore p_8 = t_{12} = \text{A}$, and a mismatch occurs in $p_1$.

Slide the window by 5.

$T = GCATCGCGGGCAG\underline{A}GTATAACAGTACG$

$P = \underline{GGAGGCCGAG}$
Rule 3-3: The Unique Pairwise Substring Rule

- The substring $p_i p_{i+1} \ldots p_{j-1} p_j$ is called an unique pairwise substring if it satisfies the condition that $p_i p_{i+1} \ldots p_{j-1} p_j$ occurs in the prefix $p_1 p_2 \ldots p_{j-1} p_j$ of $P$ exactly once, and no $p_k p_{k+1} \ldots p_{k+j-i}$ exists in $p_1 p_2 \ldots p_{j-1} p_j$ such that $p_k = p_i$ and $p_{k+j-i} = p_j$. 

![Diagram](image_url)
• Example

\[ T = \textcolor{red}{GCA}TC\textcolor{green}{GCGC}G\textcolor{red}{GC}CAGTATACAGTACG \]
\[ P = \textcolor{green}{GCAGG}\textcolor{red}{GC}\textcolor{green}{GAG} \]

The substring CGA is an unique pairwise substring, and because \( p_6 = t_{10} = 'C' \), \( p_8 = t_{12} = 'A' \), we could slide the window by 6.

\[ T = \textcolor{red}{GCA}TC\textcolor{green}{GCGC}G\textcolor{red}{GC}CAGTATACAGTACG \]
\[ P = \textcolor{green}{GCAGG}\textcolor{red}{GC}\textcolor{green}{GAG} \]
Rule 4: The Two Window Rule

- Open a window with length $2m$. If (the length of a suffix of $u_l$ which is equal to a prefix of $P$) + (the length of a prefix of $u_r$ which is equal to a suffix of $P$) = $m$, output the position. Slide the window by $m$. 

\[ T: \quad \begin{array}{c|c|c|c|c|c} \hline & & v & & & \\ \hline & u_l & & u_r & & \\ \hline \end{array} \]

\[ P: \quad \begin{array}{c|c|c|c|c|c} \hline & & v & & & \\ \hline & & & & v & \\ \hline \end{array} \]
• Example

\[ T = GCATCGAGAGAGCGTATACAGTACG \]

\[ P = AGAGC \]

The suffixes of \( u_l \) which are equal to prefixes of \( P \): “AG” and “AGAG”. Return the lengths: 2, 4.

The prefix of \( u_r \) which is equal to a suffix of \( P \): “AGC”. Return the length: 3.

\[ \therefore 2 + 3 = 5 = m, \text{ find a position in } T_9. \]

\[ T = GCATCGAGAGAGCGTATACAGTACG \]

\[ u_l \]

\[ u_r \]
• The Wide Window Algorithm (He, Fang and Sui Algorithm) uses Rule 4.
Rule 5: The Non-Tandem-Repeat Rule

- We divide pattern $P$ into two parts $uv$ in such a way that no suffix of $u$ is a prefix of $v$. 

\[
\begin{array}{c|c}
 u & v \\
\end{array}
\]
- Example:

\[ P = A \ G \ G \ A \ T \ G \ A \ T \ C \ C \ A \ T \]

\[ P = \boxed{A \ G \ G \ A} \ T \ G \ A \ T \ C \ C \ A \ T \]
Example:

\[
T = \underbrace{G\ C\ T\ A\ T\ G\ C\ A}_{\text{CACATG}}\ T\ G\ C\ A \\

P = \underbrace{C\ A\ T\ G\ C\ A}_{\text{CACATG}} \\
\]
• Maximal Suffix: (alphabetically)
  bacdabc
  ________
  maximal suffix

  cabebaa
  ________
  maximal suffix
• Given a string $S$, divide it into $uv$ such that $v$ is the maximal suffix.

• Then $uv$ must follow the Non-tandem Repeat Rule.

• Besides, $v$ does not appear in $u$. Then the uniqueness rule can be used.
An excellent method to speed up the searching: Encoding (The Chen’s Algorithm)

T= Whatever you did to the least brother of mine, you did it for me.

P=least brother

Choose r .

P’=r4r, T’=r18r4r17r. P’=r4r occurs in T’.

A further checking can be done.
Final Sample Examples of Algorithms for Each Rule

• Rule 1: The Suffix to Prefix Rule
• Exemplary Algorithm: The MP Algorithm

\[
T = \underline{C \ G \ C \ A \ C \ G \ G} \ T \ A \ C \ G \ G \ A \ C \ C
\]
\[
P = \underline{C \ G \ G \ A \ C}
\]

\[
\begin{align*}
&\underline{C \ G \ G \ A \ C} \\
&\underline{C \ G \ G \ A \ C} \\
&\underline{C \ G \ G \ A \ C} \\
&\underline{C \ G \ G \ A \ C} \\
&\underline{C \ G \ G \ A \ C}
\end{align*}
\]
Another MP Algorithm Example

\[ T = \text{CGC T ACGC AAT CGC ACG} \]
\[ P = \text{CGC ACG} \]

\[ \text{CGC ACG} \]
\[ \text{CGC ACG} \]
\[ \text{CGC ACG} \]
\[ \text{CGC ACG} \]
\[ \text{CGC ACG} \]
\[ \text{CGC ACG} \]
Rule 2: The Substring Matching Rule

- Exemplary Algorithm: The Tuned Boyer and Moore Algorithm.

\[ T = \text{C G C A C G G T A C G G A C C C} \]
\[ P = \text{C G G A C} \]
Rule 3: The Uniqueness Rule

• Exemplary Algorithm: Rule 3-3 (Unique Pairwise Substring Rule)

\[ T = \text{A T C A T C G C A C C C} \]
\[ P = \text{C G C A C C} \]

\[ \text{C G C A C C} \]
\[ \text{C G C A C C} \]
Rule 4: Two Window Rule

$T = \text{CGCACA CGGGTACCTTTACGGGT}$

$P = \text{CTTA}$

$w_1 \quad w_2$

$C \quad G \quad C \quad A \quad C \quad G \quad G \quad T$

No prefix of $P$ = a suffix of $W_1$.

No suffix of $P$ = a prefix of $W_2$.

$w_3 \quad w_4$

$A \quad C \quad C \quad T \quad T \quad A \quad C \quad G$

$C \quad T \quad T \quad A$

Matched!
Rule 5: Non Tandem Repeat

\[ P = \begin{array}{cccc} A & G \\ \hline \end{array} \ \begin{array}{cccc} C & G & A & C \\ u & v \end{array} \]

(No suffix of \( u \) = a prefix of \( v \)).

\[ T = \begin{array}{cccccccccccc} C & A & A & C & G & C & A & G & C & G & A & C & C & T \end{array} \]

\[ P = \begin{array}{cccc} A & G & C & G & A & C \\ A & G & C & G & A & C \\ A & G & C & G & A & C \\ A & G & C & G & A & C \\ A & G & C & G & A & C \end{array} \]
The Encoding Method: The Chen’s Algorithm

$T =$ The woods are lovely, dark and deep. But I have promises to keep, and miles to go before I sleep, and miles to go before I sleep.

$P =$ promises to keep
$P’ = p_{12}p$
$T’ = p_{9}p_{12}p_{24}p_{26}p$

$P’$ occurs in $T’$. 
The End of Problem 1.

The Starting of Problem 2.
Problem 2

- The approximate string matching problem: We are given a text string $T = t_1 t_2 \ldots t_n$ and a pattern string $P = p_1 p_2 \ldots p_m$. Find each substring $S$ in $T$ where the edit distance between $S$ and $P$ is smaller than or equal to some constant $k$. 
String $X$ : ATGAATCTTTACCGCCTCG
String $Y$ : ATGAGGCTCTCTGGCCCCTG

Transformation (from string $Y$ to string $X$)

String $X$: A T G A A – – T C T T A C C G C C T C G
String $Y$: A T G A G G C T C T G G C C – C C C C T – G

$EDIT(X, Y)$=7 (2 insertions, 2 deletions and 3 changes).
Dynamic Programming for Edit Distance:

\[
EDIT[i, j] = \min \begin{cases} 
  EDIT[i, j-1] + 1 & \text{(Delete)} \\
  EDIT[i-1, j] + 1 & \text{(Insert)} \\
  EDIT[i-1, j-1] + \partial(x[i], y[j]) & \text{(Substitute)} 
\end{cases}
\]

where \( \partial(x[i], y[j]) = 0 \), if \( x[i] = y[j] \), and \( \partial(x[i], y[j]) = 1 \) otherwise.

\( EDIT[i, 0] = i, EDIT[0, j] = j \).
Given

\[ X = \text{abcabba} \]
\[ Y = \text{cbabac} \]

\[ \text{EDIT}(X, Y) = 4 \]

\[ \text{abcabba} \rightarrow \text{cb-ab-ac} \]
Given
$X=abcabba$
$Y=cbabac$

$EDIT(X, Y)=4$

$abcaabba-$
$cb-a-bac$
Problem 2

- We now come to our Problem 2.
- The approximate string matching problem: We are given a text string $T = t_1t_2 \cdots t_n$ and a pattern string $P = p_1p_2 \cdots p_m$. Find each substring $S$ in $T$ where $ED(S, P) \leq k$. 
There are essentially three approaches:

1. The global approach.

2. The local approach.

3. The exhaustive searching approach.

We first introduce the global approach.
The spirit of the global approach is as follows:
1. We first find all substrings in T which exactly match \( P(1,j) \), \( 1 \leq j \leq m \).
2. Based upon the above finding, we find all substrings in T whose edit distances with \( P(1,j) \leq 1 \), \( 1 \leq j \leq m \).
3. The process continues until we find all substrings in T whose edit distances with \( P \leq k \).
A dynamic programming approach.
Rule 1

Consider two substrings $A_1$ and $A_2$ as shown below:

\[ A_1 \quad P_1 \quad S_1 \]
\[ A_2 \quad P_2 \quad S_2 \]

If $ed(A_1, A_2) \leq k$ and $S_1 = S_2$, then $ed(P_1, P_2) \leq k$. 
A Recursive Operation for the Global Dynamic Programming Approach Wu and Manber Algorithm

Consider $T_{1,i}$ and $P_{1,j}$.

Case 1: $T_i = P_j$. We denote prefix $P_{1,j-1}$ in $P$ to be $B$. We consider whether there is a suffix $A$ in $T_{1,i-1}$ such that $d(A,B) \leq k$. 

![Diagram](image-url)
Case 2: \( T_i \neq P_j \). We consider three cases:

2.1 We denote \( P_{1,j} \) to be \( B \). We consider whether there is a suffix \( A \) in \( T_{1,i-1} \) such that \( d(A,B) \leq k-1 \). This corresponds to an insertion as illustrated below:

\[
\begin{align*}
T &: \quad \begin{array}{cccc}
\text{i-1} & \text{i} \\
\hline
A & & & \\
\end{array} \\
\begin{array}{cccc}
1 & j \\
\hline
P : \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
T &: \quad \begin{array}{cccc}
\text{i-1} & \text{i} \\
\hline
A & & & \\
\end{array} \\
\begin{array}{cccc}
1 & j \\
\hline
P : \\
\end{array} \\
\end{align*}
\]

\text{insertion}
Case 2: $T_i \neq P_j$. We consider three cases:

2.2 We denote $P_{1,j-1}$ to be $B$. We consider whether there is a suffix $A$ in $T_{1,i}$ such that $d(A,B) \leq k-1$. This corresponds to a deletion as illustrated below:

```
T : 1 2 3 4 5 6 7 8
    |---|---|
    |   |   |
    |   A |
    |---|---|
    |   |   |
    |   j |

P : 1 2 3 4 5 6 7 8
    |---|---|
    |   |   |
    |   B |
    |---|---|
    |   |   |
    |---|---|
    |   j |
```

deletion
Case 2: $T_i \neq P_j$. We consider three cases:

2.3 We denote $P_{1,j-1}$ to be $B$. We consider whether there is a suffix $A$ in $T_{1,i-1}$ such that $d(A,B) \leq k-1$. This corresponds to a substitution as illustrated below:

```
T:  i-1  i
   A
P:  1  j-1  j
   B
```

substitution
To solve our approximate string matching problem, we start with a table, called $R^k[n, m]$. Let $S=T_{1,i}$.

$$R^k(i,j) \begin{cases} 1 & \text{if there exists a suffix } A \text{ of } S \text{ such that } d(A, P_{1,j}) \leq k. \\ 0 & \text{otherwise.} \end{cases}$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$.

**Example:**

$T$:aabaacaabacab, $P$:aabac and $k=1$.

Consider $i=9$, $j=4$.

$S=T_{1,9}$=aabaacaab

$P_{1,4}$=aaba

$A=T_{7,9}$=aab

$d(A, P_{1,4})=d(aab, aaba)=1$

$\therefore R^1(9,4)=1$
The meaning of $R^0(i,j)$’s

$R^0(i,j) \begin{cases} 
= 1 & \text{if there exists a suffix } A \text{ of } S \text{ such that } d(A, P_{1,j})=0. \\
= 0 & \text{otherwise.} 
\end{cases}$

Example: Text = aabaacaabacab. Pattern = aabac. $k=0$.

<table>
<thead>
<tr>
<th>$R^0[13,5]$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>1 a</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3 b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4 a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5 c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Question: How can we find $R^k(i, j)$?

Answer: Dynamic Programming.

There are three types of operations in edit distance:

(1) Insertion
(2) Deletion
(3) Substitution

We consider them separately and combine the results later.
Let $R_I^k(i,j)$, $R_D^k(i,j)$ and $R_S^k(i,j)$ denote the $R^k(i,j)$ related to insertion, deletion and substitution respectively.

$R_I^k(i,j)=1$, $R_D^k(i,j)=1$ and $R_S^k(i,j)=1$ indicate that we can perform an insertion, deletion and substitution respectively without violating the error bound which is $k$. 
If \( T_i \neq P_j \), after every \( R_I^k(i,j) \), \( R_D^k(i,j) \) and \( R_S^k(i,j) \) have found, we immediately determine \( R^k(i,j) \) by

\[
R^k(i,j) = R_I^k(i,j) \lor R_D^k(i,j) \lor R_S^k(i,j).
\]

If \( T_i = P_j \) and \( R^k(i-1,j-1) = 1 \), \( R^k(i,j) = 1 \).

If \( T_i = P_j \) and \( R^k(i-1,j-1) = 0 \), \( R^k(i,j) = 0 \).

**Example:** Text = aabaacaabacab. Pattern = aabac. \( k = 1 \).
Rule 1 is used in Amir, Keselman, Landau, Lewenstein, Lewenstein and Rodeh Algorithm, Heikki Algorithm, Huynh, Hon, Lam and Sung, Algorithm, Holub and Melichar Algorithm, Landau Vishkin Algorithm, Navarro and Baeza-Yates Algorithm, Sellers Algorithm, Tarhio and Ukkonen Algorithm, and Wu and Manber Algorithm.
The Local Search Approach

- As in the exact string matching algorithms, we open a window \( W \) and calculate the edit distance between each prefix of \( W \) and pattern \( P \). We shift the window step by step. Through this way, we can find every substring \( S \) in \( T \) such that \( ED(S, P) \leq k \).

The dynamic programming approach to find edit distances can now be used.
No substring $S$ such that $ED(S, P) \leq 2$ can be found.

Substring $S = T(4, 13) = \text{acaacabacb}$ is found.
But, it is obvious that the window size can be neither too small nor too large.

Thus, we have Rule 2.
Rule 2

If $ed(A, B) \leq k$, then the length of $A$ must be between $m-k$ and $m+k$. 
• Rule 2 is used in Fredriksson and Navarro Algorithm, Navarro and Baeza-Yates Algorithm and Tarhio and Ukkonen Algorithm.
Rule 3

If $S_1$ contains $S_1'$ completely and the edit distance between $S_1'$ and any substring of $P$ is larger than $k$, then $ed(S_1, P) > k$. 
Based upon Rule 3 and Rule 2, we have Rule 4

If the window size is $(m-k)$ and there exists a substring $S_1$ in the window such that the distance between $S_1$ and any substring of $P$ is larger than $k$, then we can safely move $P$ as follows:
If Rule 4 is not satisfied, it means the following:

For every substring $S_1$ in $T$, there exists a substring $S_2$ in $P$ such that $ed(S_1, S_2) \leq k$. 
Rule 4-1

If Rule 4 is not satisfied, we can only move 1 step as follows:

\[ T \quad m-k \quad S_1 \quad P \]

\[ T \quad m-k \quad S_1 \quad P \rightarrow \]
Rule 5

Hamming Distance\((A, B)\) $\geq$ Edit Distance\((A, B)\).

If the lengths of A and B are 2, the Hamming distance between them is equal to the edit distance.
Rule 6

Let $P$ and $Q$ be two strings.
Let $P$ be divided as follows:

\[
P_1 \quad P_2 \quad \ldots \quad P_n
\]

Let $Q_i$ be the substring in $Q$ and that $ed(P_i, Q_i)$ is the smallest.

\[
P_1 \quad P_2 \quad \ldots \quad P_n
\]
\[
Q_2 \quad Q_1 \quad \ldots \quad Q_N
\]

If $\sum_{i=1}^{N} ed(P_i, Q_i) > k$, $ed(P, Q) > k$. 

104
In the following, we introduce an algorithm which incorporates Rules 4, 5 and 6.

The Fredriksson and Navarro Algorithm
The FN-algorithm scans from the right as shown below:

\[ T: \quad m - k \]

\[ P: \]

For a window of size \( m-k \), if there exists a substring \( S_1 \) in this window such that its edit distance with every substring of \( P \) is greater than \( k \), we move \( P \) (Rule 4).
To determine whether $\text{ED}(S_1, S_2) > k$, we may use Rule 6.

We divide the window into small pieces: $t_1, t_2, \ldots, t_a$.

For each $t_i$, we find the substring $p_i$ in $P$ where $\text{ED}(p_i, t_i)$ is the smallest.

![Diagram](image-url)
In general, to find such a $p_i$, we may use dynamic programming.

But, we may use a special kind of small pieces.

It is customary to call a small piece with size $L$ a $L$-gram.

Let us use the 2-gram (Rule 5).
Smallest edit distance between “aa” and all substrings of $P = 1$.

Smallest edit distance between “gc” and all substrings of $P = 1$.

∴ $\sum > k$
Rule 7

For strings $A$ and $B$, if there are $k+1$ characters which do not appear in $B$, then $ed(A, B) > k$.

Rule 7-1

Let $A$ and $B$ be two strings. Let there be $k+1$ characters $a_1, a_2, \ldots, a_{k+1}$ in $A$ and $a_i$ is aligned with $b_i$ in $B$. If no $a_i$ appear in $B[i-k, i+k]$, then $ed(A, B) > k$.

The Tarhio and Ukkonen Algorithm.
Example

<table>
<thead>
<tr>
<th>Σ</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D[i=8, a]$</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>$D[i=7, a]$</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

$k=1$

Shifting is needed now.

$t_8=A$ appears in $P(7,8)$

$t_7=C$ does not appear in $P(6,8)$

$t_6=G$ appears in $P(5,7)$

$t_5=C$ does not appear in $P(4,6)$
Rule 8

Let there be two strings $A$ and $B$. Let $B$ be divided into $j$ pieces $B_1, B_2, \ldots, B_j$. If $ed(A, B) \leq k$, there is at least one substring $A_i$ in $A$ such that $ed(A_i, B_i) \leq \left\lfloor \frac{k}{j} \right\rfloor$. 

Rule 8-1

Let $A$ and $B$ be two strings. Let $B$ be divided into $j$ pieces $B_1, B_2, \ldots, B_j$. If for every $B_i$ and every substring $S$ of $A$, $ed(S, B_i) > \lfloor k/j \rfloor$, $ed(A, B) > k$.

The Navarro and Baeza-Yates algorithm.
If we let $j=k+1$, then $\left\lfloor \frac{k}{j} \right\rfloor = 0$.

In this case, if $\text{ed}(T, P) \leq k$, then at least one $p_i$ occurs in $T$ exactly.

If, in a certain window, we find an exact matching of a $p_i$ inside the window, we use the dynamic programming approach to determine whether there exists an approximate matching of $P$ allowing $k$ errors in this window.
If, in a window, we cannot find any exact matching of $p_i$ inside the window, we ignore the window. That is, we do not have to check whether there is an approximate matching inside the window.
Suppose we have $P = ATCCTC$ with $k = 2$.
We divide $P$ into three pieces: $p_1 = AT$, $p_2 = CC$ and $p_3 = TC$.

$T = TCCAAGTTATAGCTC$

We will find AT occurring at 9 in $T$, CC occurring at 2 in $T$ and TC occurring at 1 and 14 in $T$.

We open a window starting at $i-k = 9 - 2 = 7$ and ending at $i + m + k = 9 + 6 + 2 = 17$.

We use dynamic programming to check the edit distances.

Of course, we also open windows at locations 1, 2 and 14.
Rule 9 (The Lu’s Algorithm)

Let $A$ and $B$ be two strings with lengths $m+k$ and $m$ respectively.
Let $A'$ be the prefix of $A$ with length $m-k$.
Let there be $j$ alphabets $a_1, a_2, \ldots, a_j$ in $A'$.
Let the number of times that $a_i$ appears in $A'$ and $B$ be $N_{a_i}(A')$ and $N_{a_i}(B)$ respectively.
Let $C_1$ be the set of all alphabets $a_i$ such that $N_{a_i}(A') > N_{a_i}(B)$. Let $AP$ be any prefix of $A$.

If $\sum_{c \in C_1} [N_c(A') - N_c(B)] > k$, $ed(AP, B) > k$. 

Rule 9-1

Let $A$ and $B$ be two strings with lengths $m+k$ and $m$ respectively.
Let there be $j$ alphabets $a_1, a_2, \ldots, a_j$ in $A$.
Let the number of times that $a_i$ appears in $A$ and $B$ be $N_{a_i}(A)$ and $N_{a_i}(B)$ respectively.
Let $C_2$ be the set of all alphabets $a_i$ such that $N_{a_i}(A) < N_{a_i}(B)$. Let $AP$ be any prefix of $A$.

If $\sum_{c \in C_2} [N_c(B) - N_c(A)] > k$, $ed(AP, B) > k$. 

Example

\[
\begin{array}{cccccccccccccccc}
\hline
T & a & a & a & a & c & a & a & a & c & a & b & a & c & b & a & c & a & a & a & a \\
\end{array}
\]

\[m-k\]

\[
\begin{array}{cccccccccccccccc}
\hline
P & a & c & c & c & a & c & a & b & c & b & b & & & & & & & & & & \\
\end{array}
\]

\[k = 2\]

\[N_a(P) = 3, \quad N_b(P) = 2, \quad N_c(P) = 4.\]

\[N_a(T(1, 7)) = 6, \quad N_b(T(1, 7)) = 0, \quad N_c(T(1, 7)) = 1.\]

The number of the character \(a\) in \(T(1, 7)\) is larger than in \(P\).

\[N_a(T(1, 7)) - N_a(P) = 6 - 3 = 3 > k.\]

Thus, the edit distances of all substrings starting at position 1 with \(P\) are larger than \(k\).
Example

\[ T \]
\[ \underline{m+k} \]

\[ P \]

\( N_a(P) = 3, \quad N_b(P) = 2, \quad N_c(P) = 4. \)

\( N_a(T(1, 11)) = 8, \quad N_b(T(1, 11)) = 1, \quad N_c(T(1, 11)) = 2. \)

The numbers of the characters \( b \) and \( c \) in \( T(1, 11) \) are smaller than in \( P \).

\[
[ N_b(P) - N_b(T(1, 11))] + [N_c(P) - N_c(T(1, 11))] \\
= (2-1) + (4-2) = 3 > k.
\]

Thus, the edit distances of all substrings starting at position 1 with \( P \) are larger than \( k \).
T=You are the big drop of the drew under the lotus leaf. I am the smaller one on its upper side, said the drew drop to the lake.

P=drew drop \quad k = 1

\[ N_d(P)=2, \; N_r(P)=2, \; N_e(P)=1, \; N_w(P)=1, \; N_o(P)=1, \; N_p(P)=1 \]

T(1, 7)=“You are t”. Y and u appear in T(1,7) once and not all in P.
T(1, 9)=“You are the” Character d appears 2 times in P and 0 times in T(1,9).

Conclusion: All substrings starting from t_{1} can be ignored.
Rule 10

• *Diagonal* $d$ is defined as all of the $D_{i,j}$’s where $d = i - j$.

• The Diagonal Property: In dynamic approach, $D_{i,j} - D_{i-1,j-1} = 0$ or 1. It means that on the diagonal, the values are monotonically increasing.

The Landau and Vishkin Algorithm
Given

\[ X = \text{abcabba} \]
\[ Y = \text{cbabac} \]

\[ EDIT(X, Y) = 4 \]

Redraw the string:

\[ \text{abcabba} \rightarrow \text{cb – a – bac} \]
Consider two substrings $A_1$ and $A_2$ as shown below:

$$
\begin{array}{c}
T_1 \\
\hline
A_1 \\
S_1 \\
\hline
T_2 \\
\hline
A_2 \\
S_2 \\
\end{array}
$$

If $ED(A_1, A_2) \leq k$ and $S_1 = S_2$, then $ED(A_1 + S_1, A_2 + S_2) \leq k$. 
• Observe the following two strings:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th></th>
<th>$A_1$</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>$i$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T_2$</th>
<th></th>
<th>$A_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>$j$</td>
<td></td>
</tr>
</tbody>
</table>

• If $i$ and $j$ are the largest $i$ and $j$ such that $ED(T_1[1\ldots i], T_2[1\ldots j]) = k$ and $T_1[i+1] \neq T_2[j+1]$, then $ED(A_1+x, A_2+y) = k+1$. 
When we find the $ED(A_1, A_2) = k$, we want to determine whether the longest common prefix $S$ of $B_1$ and $B_2$ exists.

\[
\begin{array}{c|c|c|c}
S_1 & \quad & A_1 & S \\
S_2 & \quad & A_2 & S \\
\end{array}
\]

We will use LCA (lowest common ancestor) to find $S$. 
\item \( T = \text{gggtcta} \) and \( P = \text{gttc} \). Let us concatenate \( T \) and \( P \) to be a new string as follows:

\[
\begin{array}{cccc}
ggg & \text{tct}a & \text{gt} & \text{tc} \\
\end{array}
\]

The substring after \( \text{ggg} \) is \( \text{tctagttc}=S_1' \). The substring after \( \text{gt} \) is \( \text{tc}=S_2' \). Note that \( S_2' \) and \( S_1' \) have a common prefix with length 2. Consider \( D_{3,2}=1 \). Thus we have that \( D_{3,2}=D_{4,3}=D_{5,4}=1 \).

\[
\begin{array}{cccccccc}
i & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
j & \text{g} & \text{g} & \text{g} & \text{t} & \text{c} & \text{t} & \text{a} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
2 & 2 & 1 & 1 & 1 & 0 & 1 & 1 & 2 \\
3 & 3 & 2 & 2 & 2 & 1 & 1 & 1 & 2 \\
4 & 4 & 3 & 3 & 3 & 2 & 1 & 2 & 2 \\
\hline
\end{array}
\]

\( d = 1 \)
Let us concatenate \( T \) and \( P \) to be a new string as follows: 
gggtctagttaa. And then we construct the suffix tree of it. The 
substring after ggg is tctagttc=\( S_1' \). The substring after gt is tc=\( S_2' \).
Note that \( S_2' \) and \( S_1' \) have a common ancestor tc of length 2.

\[
\begin{array}{|c|c|c|c|}
\hline
T=gggtccta & ggg & tctagttaa & gt \\
\hline
P=gttc & tc & & \\
\hline
\end{array}
\]
The Exhaustive Searching Approach

• In this approach, we generate from P all possible strings whose edit distances with P are less than or equal to $k$.

• For instance, let $P=ac$ and $k=1$. 

\( P=ac \)

Then the following strings all have edit distances with \( P \) smaller than or equal to \( k=1 \):

\( S=\{c, a, ac, cc, aa, aca, acc, aac, cac\} \)

This set can be generated by using Rule 1.
After this set is generated, the approximate string matching problem becomes the exact string matching problem.

Example:
Given $T=\text{acaaccac}$, $P=\text{ac}$ and $k=1$. $S=\{\text{c, a, ac, cc, aa, aca, acc, aac, cac}\}$
We now want to find whether the strings of $S$ occur in $T$. 
Example:
Given $T=\text{acaaccac}$, $P=\text{ac}$ and $k=1$.
$S=\{c, a, ac, cc, aa, aca, acc, aac, cac\}$

We can find aa occurs in $T(3,4)$ and the edit distance between aa and $P=ac$ is 1. There is an approximate matching occurring in $T(3)$.
• The Huynh, Hon, Lam and Sung Algorithm and the Holub and Melichar Algorithm are based upon this approach.
Exact String Matching Reference


Approximate String Matching Reference


• I would like to express my thanks to my brilliant graduate students who are now exhausted and tired after an exhaustive searching through many difficult to read papers (except WM92 and HHLS2006).

• But they are happy, proud and grateful to their adviser who has been yelling at them all the time.
Thank you.