CSE 548: *(Design and)* Analysis of Algorithms

String Algorithms

R. Sekar

Topics

1. Intro
   - Motivation
   - Background

2. RE
   - Regular expressions

3. FSA
   - DFA and NFA

4. To DFA
   - RE Derivatives
   - McNaughton-Yamada

5. Trie
   - Tries

6. grep
   - Using Derivatives
   - KMP
   - Aho-Corasick
   - Shift-And
   - agrep
   - Levenshtein Automaton

7. Fing.print
   - Rabin-Karp
   - Rolling Hashes
   - Common Substring and sync

8. Suffix trees
   - Overview
   - Applications

9. Suffix Arrays

String Matching

Strings provide the primary means of interfacing to machines.

- programs, documents, ...

Consequently, string matching is central to numerous, widely-used systems and tools

- Compilers and interpreters, command processors (e.g., bash), text-processing tools (sed, awk, ...)
- Document searching and processing, e.g., grep, Google, NLP tools, ...
- Editors and word-processors
- File versioning and compression, e.g., rcs, svn, rsync, ...
- Network and system management, e.g., intrusion detection, performance monitoring, ...
- Computational biology, e.g., DNA alignment, mutations, evolutionary trees, ...

Terminology

**String:** List $S[l..j]$ of characters over an alphabet $\Sigma$.

**Substring:** A string $P[l..j]$ such that for $P[l..j] = S[l+1..l+j]$ for some $l$.

**Prefix:** A substring $P$ of $S$ occurring at its beginning

**Suffix:** A substring $P$ of $S$ occurring at its end

**Subsequence:** Similar to substring, but the the elements of $P$ need not occur contiguously in $S$.

For instance, $bcd$ is a substring of $abcde$, while $de$ is a suffix, $abcd$ is a prefix, and $acd$ is a subsequence. A substring (or prefix/suffix/subsequence) $T$ of $S$ is said to be *proper* if $T \neq S$. 
String Matching Problems

Given a "pattern" string $p$ and another string $s$:

**Exact match**: Is $p$ a substring of $s$?

**Match with wildcards**: In this case, the pattern can contain wildcard characters that can match any character in $s$.

**Regular expression match**: In this case, $p$ is regular expression.

**Substring/prefix/suffix**: Does a (sufficiently long) substring/prefix/suffix of $p$ occur in $s$?

**Approximate match**: Is there a substring of $s$ that is within a certain edit distance from $p$?

**Multi-match**: Instead of a single pattern, you are given a set $p_1, \ldots, p_n$ of patterns. Applies to all above problems.

String Matching Techniques

**Finite-automata and variants**: Regexp matching, Knuth-Morris-Pratt, Aho-Corasick

**Seminumerical Techniques**: Shift-and, Shift-and with errors, Rabin-Karp, Hash-based

**Suffix trees and suffix arrays**: Techniques for finding substrings, suffixes, etc.

Language of Regular Expressions

Notation to represent (potentially) infinite sets of strings over alphabet $\Sigma$.

Let $R$ be the set of all regular expressions over $\Sigma$. Then,

- **Empty String**: $\epsilon \in R$
- **Unit Strings**: $\alpha \in \Sigma \Rightarrow \alpha \in R$
- **Concatenation**: $n_1, r_2 \in R \Rightarrow n_1 r_2 \in R$
- **Alternative**: $n_1, r_2 \in R \Rightarrow (n_1 \mid r_2) \in R$
- **Kleene Closure**: $r \in R \Rightarrow r^* \in R$

Regular Expression

- $a$ : stands for the set of strings $\{a\}$
- $a \mid b$ : stands for the set $\{a, b\}$
  - *Union* of sets corresponding to REs $a$ and $b$
- $ab$ : stands for the set $\{ab\}$
  - Analogous to set *product* on REs $a$ and $b$
  - $(a|b)(a|b)$: stands for the set $\{aa, ab, ba, bb\}$
- $a^*$ : stands for the set $\{\epsilon, a, aa, aaa, \ldots\}$ that contains all strings of zero or more a’s.
  - Analogous to *closure* of the product operation.
Regular Expression Examples

\[(a|b)^*\] : Set of strings with zero or more a's and zero or more b's:
\[
\{\epsilon, a, b, aa, ab, ba, bb, aab, aba, \ldots\}
\]

\[(a^*b^*)\] : Set of strings with zero or more a’s and zero or more b’s such that all a's occur before any b:
\[
\{\epsilon, a, b, aa, ab, bb, aab, abb, \ldots\}
\]

\[(a^*b^*)^*\] : Set of strings with zero or more a’s and zero or more b’s:
\[
\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}
\]

Semantics of Regular Expressions

Semantic Function \(\mathcal{L}\): Maps regular expressions to sets of strings.

\[
\begin{align*}
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(\alpha) &= \{\alpha\} \quad (\alpha \in \Sigma) \\
\mathcal{L}(\eta \mid r_2) &= \mathcal{L}(\eta) \cup \mathcal{L}(r_2) \\
\mathcal{L}(\eta \cdot r_2) &= \mathcal{L}(\eta) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(r^*) &= \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))
\end{align*}
\]

Finite State Automata

Regular expressions are used for specification, while FSA are used for computation.

FSAs are represented by a labeled directed graph.
- A finite set of states (vertices).
- Transitions between states (edges).
- Labels on transitions are drawn from \(\Sigma \cup \{\epsilon\}\).
- One distinguished start state.
- One or more distinguished final states.

Finite State Automata: An Example

Consider the Regular Expression \((a \mid b)^*a(a \mid b)\).
\[
\mathcal{L}((a \mid b)^*a(a \mid b)) = \{aa, ab, aaa, aab, baa, bab, aaaa, aaab, abaa, abab, baaa, \ldots\}.
\]
The following (non-deterministic) automaton determines whether an input string belongs to \(\mathcal{L}((a \mid b)^*a(a \mid b))\):

[Diagram of FSA]
Determinism

\((a \mid b)^*a(a \mid b)\):

Nondeterministic: (NFA)

Deterministic: (DFA)

Acceptance Criterion

A finite state automaton (NFA or DFA) *accepts* an input string \(x\) if beginning from the start state we can trace some path through the automaton such that the sequence of edge labels spells \(x\) and end in a final state.

Or, there exists a path in the graph from the start state to a final state such that the sequence of labels on the path spells out \(x\).

Recognition with an NFA

Is \(abab \in L((a \mid b)^*a(a \mid b))\)?

Input: a b a b
Path 1: 1 1 1 1 1
Path 2: 1 1 1 2 3 Accept
Path 3: 1 2 3 ⊥ ⊥

Accept

Recognition with a DFA

Is \(abab \in L((a \mid b)^*a(a \mid b))\)?

Input: a b a b
Path: 1 2 4 2 4 Accept
NFA vs. DFA

For every NFA, there is a DFA that accepts the same set of strings.

- NFA may have transitions labeled by \( \epsilon \).
  (Spontaneous transitions)
- All transition labels in a DFA belong to \( \Sigma \).
- For some string \( x \), there may be many accepting paths in an NFA.
- For all strings \( x \), there is one unique accepting path in a DFA.
- Usually, an input string can be recognized faster with a DFA.
- NFAs are typically smaller than the corresponding DFAs.

\[ n = \text{Size of Regular Expression (pattern)} \]
\[ m = \text{Length of Input String (subject)} \]

<table>
<thead>
<tr>
<th></th>
<th>NFA</th>
<th>DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Automaton</td>
<td>( O(n) )</td>
<td>( O(2^n) )</td>
</tr>
<tr>
<td>Recognition time</td>
<td>( O(n \times m) )</td>
<td>( O(m) )</td>
</tr>
</tbody>
</table>

Converting RE to FSA

**NFA**: Compile RE to NFA (Thompson’s construction [1968]), then match.

**DFA**: Compile to DFA, then match

(A) Convert NFA to DFA (Rabin-Scott construction), minimize

(B) Direct construction: RE derivatives [Brzozowski 1964].
  - More convenient and a bit more general than (A).

(C) Direct construction of [McNaughton Yamada 1960]
  - Can be seen as a (more easily implemented) specialization of (B).
  - Used in Lex and its derivatives, i.e., most compilers use this algorithm.

- NFA approach takes \( O(n) \) NFA construction plus \( O(nm) \) matching, so has worst case \( O(nm) \) complexity.
- DFA approach takes \( O(2^n) \) construction plus \( O(m) \) match, so has worst case \( O(2^n + m) \) complexity.
- So, why bother with DFA?
  - In many practical applications, the pattern is fixed and small, while the subject text is very large. So, the \( O(mn) \) term is dominant over \( O(2^n) \)
  - For many important cases, DFAs are of polynomial size
  - In many applications, exponential blow-ups don’t occur, e.g., compilers.
Derivative of Regular Expressions

The derivative of a regular expression $R$ w.r.t. a symbol $x$, denoted $\partial_x[R]$ is another regular expression $R'$ such that $L(R) = L(xR')$.

Basically, $\partial_x[R]$ captures the suffixes of those strings that match $R$ and start with $x$.

**Examples**
- $\partial_a[a(b|c)] = b|c$
- $\partial_a[(a|b)cd] = cd$
- $\partial_a[(a|b)* cd] = (a|b)* cd$
- $\partial_a[(a|b)* cd] = d$
- $\partial_a[(a|b)* cd] = \emptyset$

Note: $L(\epsilon) = \{\epsilon\} \neq L(\emptyset) = \{\}$

Definition of RE Derivative (1)

$\text{inclEps}(R)$: A predicate that returns true if $\epsilon \in L(R)$

- $\text{inclEps}(a) = \text{false, } \forall a \in \Sigma$
- $\text{inclEps}(R_1 | R_2) = \text{inclEps}(R_1) \lor \text{inclEps}(R_2)$
- $\text{inclEps}(R_1R_2) = \text{inclEps}(R_1) \land \text{inclEps}(R_2)$
- $\text{inclEps}(R^*) = \text{true}$

Note $\text{inclEps}$ can be computed in linear-time.

DFA Using Derivatives: Illustration

Consider $R_1 = (a|b)^* a|b$

- $\partial_a[R_1] = R_1| (a|b) = R_2$
- $\partial_b[R_1] = R_1$
- $\partial_a[R_2] = R_1| (a|b)\epsilon = R_3$
- $\partial_b[R_2] = R_1| \epsilon = R_4$
- $\partial_a[R_3] = R_1| (a|b)\epsilon = R_3$
- $\partial_b[R_3] = R_1| \epsilon = R_4$
- $\partial_a[R_4] = R_1| (a|b) = R_2$
- $\partial_b[R_4] = R_1$

Note: $\text{inclEps}(R) = \text{true}$ does not implies $L(R) = \emptyset$.
McNaughton-Yamada Construction

Can be viewed as a simpler way to represent derivatives

- Positions in RE are numbered, e.g., \((a^1b^2) \ast a^3(a^1b^2)^6\).
- A derivative is identified by its beginning position in the RE
  - Or more generally, a derivative is identified by a set of positions
- Each DFA state corresponds to a position set (pset)

\[
\begin{align*}
R_1 & \equiv \{1, 2, 3\} \\
R_2 & \equiv \{1, 2, 3, 4, 5\} \\
R_3 & \equiv \{1, 2, 3, 4, 5, 6\} \\
R_4 & \equiv \{1, 2, 3, 6\}
\end{align*}
\]

McNaughton-Yamada Illustration

\[
\text{BuildMY}(R, \text{pset})
\]

Create an automaton state \(S\) labeled \(\text{pset}\)
Mark this state as final if \(\$\) occurs in \(R\) at \(p\)

\[\text{foreach} \text{ symbol } x \in \text{first}(\text{pset}) - \{$\} \text{ do} \]
Call BuildMY\((R, \text{follow}(\text{pset}|_a))\) if hasn't previously been called
Create a transition on \(x\) from \(S\) to the root of this subautomaton

DFA construction begins with the call \(\text{BuildMY}(R, \text{follow}\{0\})\). The root of the resulting automaton is marked as a start state.
McNaughton-Yamada (MY) Vs Derivatives

- Conceptually very similar
- MY takes a bit longer to describe, and its correctness a bit harder to follow.
- MY is also more mechanical, and hence is found in most implementations
- Derivatives approach is more general
  - Can support some extensions to REs, e.g., complement operator
  - Can avoid some redundant states during construction
    - Example: For $ac|bc$, DFA built by derivative approach has 3 states, but the one built by MY construction has 4 states
      The derivative approach merges the two $c$'s in the RE, but with MY, the two $c$'s have different positions, and hence operations on them are not shared.

Avoiding Redundant States

- Automata built by MY is not optimal
  - Automata minimization algorithms can be used to produce an optimal automaton.
- Derivatives approach associates DFA states with derivatives, but does not say how to determine equality among derivatives.
- There is a spectrum of techniques to determine RE equality
  - MY is the simplest: relies on syntactic identity
  - At the other end of the spectrum, we could use a complete decision procedure for RE equality.
    - In this case, the derivative approach yields the optimal RE!
  - In practice we would tend to use something in the middle
    - Trade off some power for ease/efficiency of implementation

RE to DFA conversion: Complexity

- Given DFA size can be exponential in the worst case, we obviously must accept worst-case exponential complexity.
- For the derivatives approach, it is not immediately obvious that it even terminates!
  - More obvious for McNaughton-Yamada approach, since DFA states correspond to position sets, of which there are only $2^n$.
- Derivative computation is linear in RE size in the general case.
- So, overall complexity is $O(n2^n)$
- Complexity can be improved, but the worst-case $2^n$ takes away some of the rationale for doing so.
  - Instead, we focus on improving performance in many frequently occurring special cases where better complexity is achievable.

RE Matching: Summary

- Regular expression matching is much more powerful than matching on plain strings (e.g., prefix, suffix, substring, etc.)
- Natural that RE matching algorithms can be used to solve plain string matching
  - But usually, you pay for increased power: more complex algorithms, larger runtimes or storage.
  - We study the RE approach because it seems to not only do RE matching, but yield simpler, more efficient algorithms for matching plain strings.
String Lookup

**Problem:** Determine if \( s \) equals any of the strings \( p_1, \ldots, p_k \).

- Equivalent to the question: does the RE \( p_1 | p_2 | \cdots | p_k \) match \( s \)?
- We can use the derivative approach, except that derivatives are very easy to compute.
- Or, we can use BuildMY — once again, follow() sets are very easy to compute for this class of regular expressions.
- Results in an FSA that is a tree
- More commonly known as a trie

### Trie Example

\[
R_0 = \text{top} \mid \text{tool} \mid \text{tooth} \mid \text{at} \mid \text{sunk} \mid \text{sunny}
\]

\[
R_1 = \partial_t[R_0] = \text{op} \mid \text{oool} \mid \text{ooth}
\]

\[
R_2 = \partial_o[R_1] = \text{p} \mid \text{ol} \mid \text{oth}
\]

\[
R_3 = \partial_p[R_2] = \epsilon
\]

\[
R_4 = \partial_o[R_2] = \text{h} \mid \text{th}
\]

\[
R_5 = \partial_t[R_4] = \text{unk} \mid \text{unny}
\]

\[
R_6 = \partial_u[R_5] = \text{uk} \mid \text{nny}
\]

\[
R_7 = \partial_n[R_6] = \text{k} \mid \text{ny}
\]

\[
R_8 = \partial_k[R_7] = \epsilon
\]

\[
R_9 = \partial_n[R_6] = \text{y} \mid \text{ny}
\]

\[
R_{10} = \partial_y[R_8] = \epsilon
\]

### Trie Summary

- A data structure for efficient lookup
  - Construction time linear in the size of keywords
  - Search time linear in the size of the input string
- Can also support maximal common prefix (MCP) query
- Can also be used for efficient representation of string sets
  - Takes \( O(|s|) \) time to check if \( s \) belongs to the set
  - Set union/intersection are linear in size of the smaller set
    - Sublinear in input size when one input trie is much larger than the other
  - Can compute set difference as well — with same complexity.

### Implementing Transitions

**How to implement transitions?**

**Array:** Efficient, but unacceptable space when \( |\Sigma| \) is large

**Linked list:** Space-efficient, but slow

**Hash tables:** Mid-way between the above two options, but noticeably slower than arrays. Collisions are a concern.
- But customized hash tables for this purpose can be developed.
- Alternatively, since transition tables are static, we can look for perfect hash functions

**Specialized representations:** For special cases such as exact search, we could develop specialized alternatives that are more efficient than all of the above.
Exact Search

- Determine if a pattern $P[1..n]$ occurs within text $S[1..m]$
  - Find $j$ such that $P[1..n] = S[j..(j+n-1)]$
- An RE matching problem: Does $\Sigma^* P \Sigma^*$ match $S$?
  - Note: $\Sigma^*$ matches any arbitrary string (incl. $\epsilon$)
- We consider $\Sigma^* P$ since it can identify all matches
  - A match can be reported each time a final state is reached.
  - In contrast, an automaton for $\Sigma^* P \Sigma^*$ may not report all matches

Exact Search: Complexity

- Positives:
  - Matching is very fast, taking only $O(m)$ time.
  - Only linear (rather than exponential) number of states
- Downsides:
  - Construction of psets for each state takes up to $O(n)$ time
  - Thus, overall complexity of automata construction is $O(n^2)$
  - Can be $O(n^2 |\Sigma|)$ since each state may have up to $|\Sigma|$ transitions
- Question: Can we do better?
  - Faster construction
    - $O(n)$ instead of $O(n^2)$?
  - More efficient representation for transitions.
    - constant number of transitions per state?

Improving Exact Search: Observations

The DFA has a linear structure, with states 1 to $n+1$:

- State $i$ is reached on matching the prefix $P[1..i-1]$
- The largest element of $pset(i)$ is $i$
- If you are in state $i$ after scanning $S[k]$: Let $P' = P[1..i-1] = S[k - i + 2..k]$ so, $pset(i)$ includes every $j$ such that $S[k - i + 2..k] = P[1..i-1] = P[1..i-1]$

<table>
<thead>
<tr>
<th>$S$</th>
<th>$a$</th>
<th>$a$</th>
<th>$b$</th>
<th>$a$</th>
<th>$b$</th>
<th>$a$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viable match 1</td>
<td>$d^2$</td>
<td>$d^2$</td>
<td>$b^2$</td>
<td>$a^2$</td>
<td>$a^2$</td>
<td>$a^2$</td>
<td>$a^2$</td>
</tr>
<tr>
<td>Viable match 2</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$a^2$</td>
<td>$a^2$</td>
</tr>
<tr>
<td>Viable match 3</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$d^2$</td>
<td>$a^2$</td>
</tr>
<tr>
<td>Viable match 4</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
</tbody>
</table>

We use McNaughton-Yamada. Recall that with this technique:

- States are identified by position sets.
- A position denotes a derivative starting at that position
- A position set indicates the union of REs corresponding to each position.

For instance, position set $\{0, 2, 3\}$ represents $R_0 | a^2 b^3 a^2 b^2 a^6 a^7$
Improving Exact Search: Key Ideas

Main Idea
- Remember only the largest \( j < i \) in \( pset(i) \)
  - You can look at \( pset(j) \) for the next smaller element
  - Add failure links from state \( i \) to \( j \) for this purpose
- Two positions per \( pset \) \( \implies O(n) \) construction time

KMP Algorithm

\begin{align*}
\text{BuildAuto}(P[1..m]) & \\
& \quad j = 0 \\
& \quad \text{for } i = 1 \text{ to } m \text{ do} \\
& \quad \quad \text{fail}[i] = j \\
& \quad \quad \text{while } j > 0 \text{ and } P[i] \neq P[j] \text{ do} \\
& \quad \quad \quad j = \text{fail}[j] \\
& \quad \quad j++ \\
& \quad \text{if } j > m \text{ then return } i - m + 1
\end{align*}

\begin{align*}
\text{KMP}(P[1..m], S[1..n]) & \\
& \quad j = 0; \text{BuildAuto}(P) \\
& \quad \text{for } i = 1 \text{ to } n \text{ do} \\
& \quad \quad \text{while } j > 0 \text{ and } T[i] \neq P[j] \text{ do} \\
& \quad \quad \quad j = \text{fail}[j] \\
& \quad \quad j++ \\
& \quad \text{if } j > m \text{ then return } i - m + 1
\end{align*}

- Simple, avoids explicit representation of states/transitions.
- Each state has two transitions: normal and failure.
  - Normal transition at state \( i \) is on \( P[i] \)
  - Fail links are stored in an array \( \text{fail} \)
  - \( \text{BuildAuto} \) is like matching pattern with itself!
- Algorithm is unbelievably short and simple!

Multi-pattern Exact Search

- Can we extend KMP to support multiple patterns?
  - Yes we can! It is called Aho-Corasick (AC) automaton
    - Note that AC algorithm was published before KMP!
    - Today, many systems use AC (e.g., grep, snort), but not KMP.
- KMP looks like a linear automaton plus failure links.
  - Aho-Corasick looks like a trie extended with failure links.
  - Failure links may go to a non-ancestor state
- Failure link computations are similar
  - McNaughton-Yamada and the derivatives algorithms build an automaton similar to AC, just as they did for KMP
  - One can understand Aho-Corasick as a specialization of these algorithms, as we did in the case of KMP
  - Or, as a generalization of KMP
Aho-Corasick Automaton

- As with KMP, we can think of AC as a specialization of MY.
- Retain just the largest two numbers \( i \) and \( j \) in the pset.
- Use the value of \( j \) as target for failure link, and to find \( j' \) in the successor state's pset \( \{ j', i+1 \} \)
- But there is an extra wrinkle:
  - With KMP, there is one pattern; we keep two positions from it.
  - With AC, we have multiple patterns, so a state's pset will contain positions from multiple patterns.
  - If two patterns share a prefix, the automaton state reached by this prefix will contain the next positions from both patterns.
- We will simply retain one one of these positions, say, from the higher numbered pattern.
- To avoid clutter in our example, we omit numbering of positions that will be dropped this way.

Aho-Corasick Example

Consider RE

\[
(\Sigma^*)^4 \cdot (l't'p'\$^3 | \text{too}^5 \text{\#}^7 | \text{toot}^8 \text{\#}^9 | p^p e^e r^r \#^e | o'p'p'p'p'| \text{toot}^9 \text{\#}^a)
\]

- To reduce clutter, positions that occur with previously numbered positions are not explicitly numbered, e.g., \( o' \)'s in \text{tooth} (occurs with the \( o' \)'s in \text{tool})
- Figure omits failure links that go to start state.

Parallel Exact Search

- **Key Idea**: Maintain all matching prefixes at runtime.
- Simultaneously advance the state of all these prefixes after reading next input character.
  - So, there will only be \( O(m) \) “comparisons” total.
- **How can we do this?**
  - Think of KMP automaton, strip off all failure links
  - The automaton is now linear: each state has a single successor
  - On the next input symbol, the prefix will
    - either be extended by transitioning to the successor state
    - or, the symbol doesn't match, and this match is aborted
A search for match begins on each symbol $S[k]$ in input
- Think of placing a token in start state each time you slide $P$ over $S$
- If symbols continue to match, tokens advance through successive states until they reach the final state.
- If there is a mismatch, the corresponding token “disappears” or “dies”

Each time $k$ is incremented, advance tokens forward by one state

- $T[j]$ advances if $S[k]$ matches the transition label
- Use a bitvector $\delta[0..n]$ to record transitions.
  - $\delta_x[j] = 1$ if the transition out of state $j$ is labeled $x$
  - In other words, $\delta_x[j] = 1 \iff P[j+1] = x$
  - Note $\delta_a = 01101011$, $\delta_b = 00010100$. (Note that bitvector indices go from right to left, while string indices go left-to-right.)

This means that when $k$ is incremented, $T$ should be updated as:

$$T = [(T \& \delta_{S[k]}) \ll 1]_1$$

At any time, token for matches beginning at $S[k-n]$ through $S[k]$ will be in the automaton.
- Use a bitvector $T[0..n]$ to record these tokens.
  - $T[j]$ indicates if the token for match beginning at $S[k-j]$ is still alive, i.e., $S[k-j..k-1] = P[1..j]$
  - $T[n] = 1$ indicates a completed match.

$k=11$

$$S \quad aabcdaababaaba$$

$$P \quad aababaa$$

$$T \quad 00100001$$

$$\delta_a \quad 01101011$$

$$T_{new} \quad 01000111$$
### Shift-And Method Illustration

**k=12**

<table>
<thead>
<tr>
<th>S</th>
<th>aabcdaababaaba</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>aababaa</td>
</tr>
<tr>
<td>T</td>
<td>010000111</td>
</tr>
<tr>
<td>$\delta_a$</td>
<td>01101011</td>
</tr>
<tr>
<td>$T_{new}$</td>
<td>10000111</td>
</tr>
</tbody>
</table>

**k=13**

<table>
<thead>
<tr>
<th>S</th>
<th>aabcdaababaaba</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>aababaa</td>
</tr>
<tr>
<td>T</td>
<td>100001111</td>
</tr>
<tr>
<td>$\delta_b$</td>
<td>00010100</td>
</tr>
<tr>
<td>$T_{new}$</td>
<td>00001001</td>
</tr>
</tbody>
</table>

### Approximate Search

**Approach 1:** Use edit-distance algorithm
- Expensive
- Does not allow for multiple patterns
  - Unless you try the patterns one-by-one

**Approach 2:** Levenshtein Automaton
- Can be much faster, especially when $p$ is small.
- Supports multiple patterns
- Enables applications such as spell-correction
No errors permitted.

Up to one missing character (deletion).

Up to one deletion and one insertion.

Up to one deletion, or insertion, or substitution.
Matching Using Levenshtein Automaton

- Convert to DFA (subset construction)
  - Potentially $O(n^k)$ states, where $k$ is the max edit distance permitted
- Adapt Shift-and algorithm
  - We already know how to maintain $T[0..n]$
  - Need to extend to compute $U$ from $T$, $V$ from $U$ and so on.

Extending this to the case of $U$, we have

$U^k = T^{k-1}$ \quad \text{// move ↑, i.e., Insertion of $S[k]$}

$T^{k-1} \ll 1$ \quad \text{// move ↗ with substitution}

$T^k \ll 1$ \quad \text{// move → with deletion}

$[(U^{k-1} \& S[k]) \ll 1] \ll 1$ \quad \text{// move ←}

Structure of cost matrix for edit-distance problem
Illustrates the relationship between shortest path and edit-distance problem

Up to a total of two deletions, insertions, or substitution

Compare with:
- $V_0 V_1 a V_2 a V_3 b V_4 a V_5 b V_6 a V_7 a$
  - $U_0$
  - $\Sigma$
  - $\epsilon$
  - $\Sigma$
  - $U_1 a$
  - $\Sigma$
  - $\epsilon$
  - $\Sigma$
  - $U_2 a$
  - $\Sigma$
  - $\epsilon$
  - $\Sigma$
  - $U_3 b$
  - $\Sigma$
  - $\epsilon$
  - $\Sigma$
  - $U_4 a$
  - $\Sigma$
  - $\epsilon$
  - $\Sigma$
  - $U_5 b$
  - $\Sigma$
  - $\epsilon$
  - $\Sigma$
  - $U_6 a$
  - $\Sigma$
  - $\epsilon$
  - $\Sigma$
  - $U_7 a$

$T_0 a$
$T_1 a$
$T_2 b$
$T_3 a$
$T_4 b$
$T_5 a$
$T_6 a$
$T_7 a$
Levenshtein automaton and spell-correction

- When a word \( w \) is misspelled, we want to find the closest matching word in the dictionary
- Or, list all matches within an edit distance of \( l \)

**Approach:**
- Build Levenshtein automaton for \( w \) with \( l + 1 \) “layers”
- Run the dictionary trie through the automaton
- List all matches

Alternatively, a DFA for the Levenshtein automaton could be built, and the trie run through this DFA.
- The DFA could be directly constructed as well, without going through an NFA and powerset construction.

Using arithmetic for exact matching

**Problem:** Given strings \( P[1..n] \) and \( T[1..m] \), find occurrences of \( P \) in \( T \) in \( O(n + m) \) time.

**Idea:** To simplify presentation, assume \( P, T \) range over \([0-9]\)
- Interpret \( P[1..n] \) as digits of a number
  \[ p = 10^{n-1}P[1] + 10^{n-2}P[2] + \ldots + 10^nP[n] \]
- Similarly, interpret \( T[i..(i+n-1)] \) as the number \( t_i \)
- Note: \( P \) is a substring of \( T \) at \( i \) iff \( p = t_i \)
- To get \( t_{i+1} \), shift \( T[i] \) out of \( t_i \), and shift in \( T[i+n] \):
  \[ t_{i+1} = (t_i - 10^{n-1}T[i]) \cdot 10 + T[i+n] \]

We have an \( O(n + m) \) algorithm. Almost: we still need to figure out how to operate on \( n \)-digit numbers in constant time!

Rabin-Karp Fingerprinting

**Key Idea**
- Instead of working with \( n \)-digit numbers,
- perform all arithmetic modulo a random prime number \( q \),
- where \( q > n^2 \) fits within wordsize

- All observations made on previous slide still hold
  - Except that \( p = t_i \) does not guarantee a match
  - Typically, we expect matches to be infrequent, so we can use \( O(n) \) exact-matching algorithm to confirm probable matches.

Carter-Wegman-Rabin-Karp Algorithm

**Difficulty with Rabin-Karp:** Need to generate random primes (not easy).

**New Idea:** Make the radix random, as opposed to the modulus
- We still compute modulo a prime \( q \), but it is not random.

**Alternative interpretation:** We treat \( P \) as a polynomial
\[
p(x) = \sum_{i=1}^{n} P[1..i] \cdot x^i
\]
and evaluate this polynomial at a randomly chosen value of \( x \)

What is the likelihood of false matches? Note that false match occurs when \( p(x) = t_i(x) \), or when \( p(x) - t_i(x) = 0 \).

Arithmetic modulo prime defines a field, so an \((n-1)\)th degree polynomial has \( n - 1 \) roots
- i.e., \((n-1)/q\) of the \( q \) possible choices of \( x \) result in a false match.
Rolling Hashes

RK and CWRK are examples of rolling hashes
- Hash computed on text within a sliding window
- Key point: Incremental computation of hash as the window slides.
Polynomial-based hashes are easy to compute incrementally:
\[ t_{i+1} = (t_i - x^{n-1} T[i]) \cdot x + T[i + n] \]

Complexity:
- \( x^{n-1} \) is fixed once the window size is chosen
- Takes just two multiplications, one modulo per symbol
- \( O(m + n) \) multiplication/modulo operations in total

Rolling Hash and Common Substring Problem

To find a common substring of length \( l \) or more
- Compute rolling hashes of \( P \) and \( T \) with window size \( l \)
  - Takes \( O(n + m) \) time.
  - \( O(nm) \) possible collisions, so expected number of collisions increases.
    - Unless collision probability is of the order of \( 1/nm \), expected runtime can be nonlinear
- Can find longest common substring (LCS) using a binary-search like process, with a total complexity of \( O((n + m) \log(n + m)) \)

Other Rolling Hashes

In some contexts, multiplication/modulo may be too expensive. **Alternatives:**
- Use shifts, cyclic shifts, substitution maps and xor operations, avoiding multiplications altogether
  - Need considerable research to find good fingerprinting functions.
- Example: Adler32 — used in zlib (used everywhere) and rsync.
  \[
  A_j = 1 + \sum_{k=0}^{l-1} t_{i+k} \mod 65521 \\
  B = \sum_{k=1}^{n} A_k = n + \sum_{k=0}^{n-1} (n - k) t_{i+k} \mod 65521 \\
  H = (B \ll 16) + A
  \]

zlib/gzip, rsync, binary diff, etc.

rsync: Synchronizes directories across network
- Need to minimize data transferred
  - A diff requires entire files to be copied to client side first!
  - Uses timestamps (or whole-file checksums) to detect unchanged files
  - For modified files, uses Adler-32 to identify modified regions
  - Find common substrings of certain length, say, 128-bytes
  - Relies on stronger MD-5 hash to verify unmodified regions

gzip: Uses rolling hash (Adler-32) to identify text that repeated from previous 32KB window
- Repeating text can be replaced with a “pointer” (offset, length).

Binary diff: Many programs such as xdelta and svn need to perform diffs on binaries; they too rely on rolling hashes.
- diff depends critically on line breaks, so does poorly on binaries
Suffix Trees [Weiner 1973]

- A versatile data structure with wide applications in string search and computational biology
- “Compressed” trie of all suffixes of a string appended with “$”
  - Linear chains in the trie are compressed
  - Edges can now be substrings.
  - Each state has at least two children.
  - Leaves identify starting position of that suffix.
- Key point: Can be constructed in linear time!
- Supports sublinear exact match queries, and linear LCS queries
  - With linear-time preprocessing on the text (to build suffix tree),
  - yields better runtime than techniques discussed so far.
- Applicable to single as well as multiple patterns or texts!

Finding Substrings and Suffixes

Is \( p \) a substring of \( t \)?

Example: Is \( anan \) a substring of \( banana \)?

Solution:
- Follow path labeled \( p \) from root of suffix tree for \( t \).
- If you fail along the way, then “no,” else “yes”
- \( p \) is a suffix if you reach a leaf at the end of \( p \)
- \( O(|p|) \) time, independent of \( |t| \) — great for large \( t \)

Counting # of Occurrences of \( p \)

How many times does “\( an \)” occur in \( t \)?

Solution:
- Follow path labeled \( p \) from root of suffix tree for \( t \).
- Count the number of leaves below.
- \( O(|p|) \) time if additional information (# of leaves below) maintained at internal nodes.
Self-LCS (Or, Longest Common Repeat)

What is the longest substring that repeats in \( t \)?

**Solution:**
- Find the deepest non-leaf node with two or more children!
- In our example, it is \( \text{ana} \).

LC extension of \( i \) and \( j \)

**Longest Common Extension**

Longest common prefix of suffixes starting at \( i \) and \( j \)

- Locate leaves labeled \( i \) and \( j \).
- Find their least common ancestor (LCA)
- The string spelled out by the path from root to this LCA is what we want.

LCS with another string \( p \)

- We can use the same procedure as LCR, if suffixes of \( p \) were also included in the suffix tree
- Leads to the notion of **generalized suffix tree**
Generalized Suffix Tree: Applications

LCS of $p$ and $t$: Build GST for $s$ and $t$, find deepest node that has
descendants corresponding to $s$ and $t$

LCS of $p_1, \ldots, p_k$: Build GST for $p_1$ to $p_k$, find deepest node that has
descendants from all of $p_1, \ldots, p_k$

Find strings in database containing $q$: • Build a suffix tree of all
strings in the database
• follow path that spells $q$
• $q$ occurs in every $p_i$ that appears below this node.

Suffix Arrays

• Construct a sorted array of suffixes, rather than tries
• Can use 2 to 4 bytes per symbol
• Use binary search to locate suffixes etc.

\[
\begin{array}{c|c|c|c|c|c|c}
\text{i} & T_i & A_i & T_A & LCP \\
\hline
1 & mississippi$ & 12 & $ & & \\
2 & ississippi$ & 11 & i$ & & \\
3 & ssissippi$ & 8 & ippi$ & & \\
4 & ssippi$ & 5 & ippi$ & & \\
5 & ippi$ & 2 & ississippi$ & & \\
6 & ssippi$ & 1 & mississippi$ & & \\
7 &ippi$ & 10 & pi$ & & \\
8 &ippi$ & 9 & ppi$ & & \\
9 & ppi$ & 7 & sippi$ & & \\
10 & pi$ & 4 & sissippi$ & & \\
11 & i$ & 6 & sissippi$ & & \\
12 & $ & 3 & sissippi$ & & \\
\end{array}
\]

Finding Suffix Arrays

• Maintaining LCP of successive suffixes speeds up algorithms
• Search for substring $p$ in $O(|p| + \log |t|)$
• Count number of occurrences of $p$ in $O(|p| + \log |t|)$ time
• Search for longest common repeat $O(|r|)$ time
• Use binary search to locate suffixes etc.

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{i} & T_i & A_i & T_A & LCP \\
\hline
1 & mississippi$ & 12 & $ & & 1 \\
2 & ississippi$ & 11 & i$ & & 0 \\
3 & ssissippi$ & 8 & ippi$ & & 1 \\
4 & ssippi$ & 5 & ippi$ & & 1 \\
5 & ippi$ & 2 & ississippi$ & & 4 \\
6 & ssippi$ & 1 & mississippi$ & & 0 \\
7 &ippi$ & 10 & pi$ & & 0 \\
8 &ippi$ & 9 & ppi$ & & 1 \\
9 & ppi$ & 7 & sippi$ & & 0 \\
10 & pi$ & 4 & sissippi$ & & 2 \\
11 & i$ & 6 & sissippi$ & & 1 \\
12 & $ & 3 & sissippi$ & & 3 \\
\end{array}
\]