CSE 548: Algorithms

String Algorithms

R. Sekar
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, ...
Topics

1. Intro
   - Motivation
   - Background
2. RE
   - Regular expressions
3. FSA
   - DFA and NFA
4. To DFA
   - McNaughton-Yamada
5. Trie
   - Tries
6. grep
7. Fingerprint
   - KMP
   - Aho-Corasick
   - Rabin-Karp
   - Rolling Hashes
   - Common Substring and rsync
8. Suffix trees
   - Overview
   - Applications
   - Suffix Arrays
   - Shift-And
9. agrep
**Terminology**

**String:** List $S[1..i]$ of characters over an alphabet $\Sigma$. 
Terminology

**String:** List $S[1..i]$ of characters over an alphabet $\Sigma$.

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

**String:** List $S[1..i]$ of characters over an alphabet $\Sigma$.

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

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

**String:** List $S[1..i]$ of characters over an alphabet $\Sigma$.

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

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

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

**String:** List $S[1..i]$ of characters over an alphabet $\Sigma$.

**Substring:** A string $P[1..j]$ such that for $P[1..j] = S[l+1..l+j]$ 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 $p_1, .., p_n$ 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** : $r_1, r_2 \in R \Rightarrow r_1 r_2 \in R$

**Alternative** : $r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R$

**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 for \( a \) 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:
\[
\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}
\]

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

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

$\mathcal{L}(\epsilon) = \{\epsilon\}$

$\mathcal{L}(\alpha) = \{\alpha\} \quad (\alpha \in \Sigma)$

$\mathcal{L}(r_1 \mid r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

$\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$

$\mathcal{L}(r^*) = \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))$
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 \((a | b)^*a(a | b)\).

\[\mathcal{L}((a | b)^*a(a | b)) = \{aa, ab, aaa, aab, baa, bab, aaaa, aaab, abaa, abab, baaa, \ldots\} \]

The following (non-deterministic) automaton determines whether an input string belongs to \(\mathcal{L}((a | b)^*a(a | b))\):

![Finite State Automaton Diagram](image-url)
Determinism

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

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.
Acceptance Criterion

A finite state automaton (NFA or DFA) \textit{accepts} an input string $x$

\begin{itemize}
  \item if beginning from the start state
  \item we can trace some path through the automaton
  \item such that the sequence of edge labels spells $x$
  \item and end in a final state.
\end{itemize}

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 \( \text{abab} \in \mathcal{L}((a \mid b)^*a(a \mid b)) \)?

![NFA Diagram]

Input: \( \text{abab} \)

Path 1: 1 1 1 1 1
Path 2: 1 1 1 2 3 Accept
Path 3: 1 2 3 \( \perp \) \( \perp \)

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.
NFA vs. DFA

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

<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 per input string</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.
Converting RE to FSA

- 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.
McNaughton-Yamada Construction

- Positions in RE are numbered, e.g., $^0(a^1|b^2)^*a^3(a^4|b^5)$.$^6$.
- RE suffix that remains to be matched is identified by its start position
  - Or more generally, a set of suffixes is identified by a set of positions
- Each DFA state corresponds to a position set (pset)

\[
R_1 \equiv \{1, 2, 3\} \\
R_2 \equiv \{1, 2, 3, 4, 5\} \\
R_3 \equiv \{1, 2, 3, 4, 5, 6\} \\
R_4 \equiv \{1, 2, 3, 6\}
\]
McNaughton-Yamada: Definitions

- \( (R, p) \): symbol at position \( p \) in \( R \)
  
  **Example** for \( R = (a^1b^2) a^3(a^4b^5)^6 \): \( (R, 1) = a \), \( (R, 5) = b \).

- \( \text{filter}(R, P, s) \): \( \{ p \in P \mid (R, p) = s \} \)
  
  **Example**: \( \text{filter}(R, \{1, 2, 3\}, a) = \{1, 3\} \)
  
  \( \text{filter}(R, \{1, 2, 4, 5\}, b) = \{2, 5\} \)

- \( \text{first}(R) \): First positions in \( R \)
  
  \[
  \begin{align*}
  \text{first}(a) &= \text{pos}(a) \\
  \text{first}(R_1|R_2) &= \text{first}(R_1) \cup \text{first}(R_2) \\
  \text{first}(R_1 \cdot R_2) &= \text{first}(R_1), \text{ if } R_1 \text{ doesn’t match } \epsilon \\
  \text{first}(R_1 \cdot R_2) &= \text{first}(R_1) \cup \text{first}(R_2), \text{ otherwise} \\
  \text{first}(R^*) &= \text{first}(R)
  \end{align*}
  \]

  **Example**: \( \text{first}(R) = \{1, 2, 3\} \)
McNaughton-Yamada: Definitions (Continued)

- \( \text{follow}(R, p) \): Positions immediately following \( p \) in \( R \).
  
  \[
  \text{follow}(R_1 \cdot R_2, p) \supseteq \text{first}(R_2), \text{ if } p \text{ is rightmost in } R_1
  \]
  
  \[
  \text{follow}(R^*, p) \supseteq \text{first}(R), \text{ if } p \text{ is rightmost in } R
  \]

**Example** for \( R = (a^1|b^2)^* a^3(a^4|b^5)^6 \):

- \( \text{follow}(R, 1) = \{1, 2, 3\} \)
- \( \text{follow}(R, 2) = \{1, 2, 3\} \)
- \( \text{follow}(R, 3) = \{4, 5\} \)
- \( \text{follow}(R, 4) = \{6\} \)

- \( \text{follow}(R, P) : \bigcup_{p \in P} \text{follow}(R, p) \)

**Example:** \( \text{follow}(R, \{3, 4\}) = \{4, 5, 6\} \)
McNaughton-Yamada Algorithm

**BuildMY(\(R, P\))**

Create an automaton state \(S\) labeled \(P\)

Mark this state as final if \(\$ \in \@(R, P)\)

**foreach** symbol \(a \in \@(R, P) - \{\$\} \textbf{do}**

Call **BuildMY\((R, \text{follow}(R, \text{filter}(R, P, a)))\)** if hasn’t previously been called

Create a transition on \(a\) from \(S\) to the root of this subautomaton

DFA construction begins with the call **BuildMY\((R, \text{follow}([0]))\)**. The root of the resulting automaton is marked as a start state.
**BuildMY** Illustration on $R = 0(a^1|b^2) \ast a^3(a^4|b^5)$$^6$

### Computations Needed

- $\text{follow}(\{0\}) = \{1, 2, 3\}$
- $\text{follow}(\{1\}) = \text{follow}(\{2\}) = \{1, 2, 3\}$
- $\text{follow}(\{3\}) = \{4, 5\}$
- $\text{follow}(\{4\}) = \text{follow}(\{5\}) = \{6\}$

\[
\text{filter}(\{1, 2, 3\}, a) = \{1, 3\}, \quad \text{filter}(\{1, 2, 3\}, b) = \{2\}
\]

- $\text{follow}(\{1, 3\}) = \{1, 2, 3, 4, 5\}$

\[
\text{filter}(\{1, 2, 3, 4, 5\}, a) = \{1, 3\}
\]

- $\text{filter}(\{1, 2, 3, 4, 5\}, b) = \{2, 5\}$

- $\text{follow}(\{1, 3, 4\}) = \{1, 2, 3, 4, 5, 6\}$

- $\text{follow}(\{2, 5\}) = \{1, 2, 3, 6\}$

\[
\text{filter}(\{1, 2, 3, 4, 5, 6\}, a) = \{1, 3\}
\]

- $\text{filter}(\{1, 2, 3, 4, 5, 6\}, b) = \{2, 5\}$

\[
\text{filter}(\{1, 2, 3, 6\}, a) = \{1, 3\}
\]

- $\text{filter}(\{1, 2, 3, 6\}, b) = \{2\}$

### Resulting Automaton

<table>
<thead>
<tr>
<th>State</th>
<th>Pset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3,4,5}</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,3,4,5,6}</td>
</tr>
<tr>
<td>4</td>
<td>{1,2,3,6}</td>
</tr>
</tbody>
</table>
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 (Not Search)

**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 $s$?
- Results in an FSA that is a tree
- More commonly known as a *trie*
- We can use *BuildMY*.
  - But since the construction is obvious, we won’t.
Trie Example

\[ R_0 = \text{top}|\text{tool}|\text{tooth}|\text{at}|\text{sunk}|\text{sunny} \]
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 subject $S[1..m]$.
  - 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 Example

Consider $R_0 = (\Sigma^0)^* a^1 a^2 b^3 a^4 b^5 a^6 a^7 \$^8$

We use McNaughton-Yamada. Recall that:

- States are identified by position sets.
- A position $j$ denotes:
  - a match for the pattern prefix upto but not including $j$, or
  - the need to match pattern suffix starting at $j$ in order to complete the match.

For instance, position set $\{0, 2, 3\}$ means that we have so far matched $\epsilon$, $a$ and $aa$.

- Or equivalently, $\Sigma^*$, $\Sigma^* a$ and $\Sigma^* aa$. 

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 $O(n^2)$ rather than $O(n)$.
- Upto $|\Sigma|$ transitions per state
  - Automaton size is $O(n|\Sigma|)$ rather than $O(n)$.

**Question:** Can we do better?
Improving Exact Search: Observations

The DFA has a linear structure, with states 0 to \( n \):

- State \( i \) is reached on matching the pattern prefix \( P[1..i] \)
- \( pset(i) \) identifies all viable prefix matches of \( P \)
  - i.e., \( \forall j \in pset(i) \), \( P[1..j] \) matches a subject suffix.

<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>
<th>( \ldots \ldots \ldots )</th>
</tr>
</thead>
</table>
| Viable match 1 | \( a^1 \) | \( a^2 \) | \( b^3 \) | \( a^4 \) | \( b^5 \) | \( a^6 \) | \( a^7 \) | \$^8 \)
| Viable match 2 | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( a^1 \) | \( a^2 \) | \( b^3 \)
| Viable match 3 | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( a^1 \) | \( a^2 \)
| Viable match 4 | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( \Sigma \) | \( a^1 \) |
Improving Exact Search: Key Ideas

Main Idea

- Remember only the second largest $j$ in $pset(i)$
- You can look at $pset(j)$ for the next smaller prefix
- Add failure links from state $i$ to $j$ for this purpose
- Two positions per $pset \Rightarrow O(n)$ construction time

- Failure links eliminate the need for all backward transitions
- Go to $j$ and take forward transitions from there.
- One forward and failure transition $pset \Rightarrow O(n)$ size
**Exact Search: KMP Automaton**

- Only two positions per state: \( \{j, i\} \)
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset \( \{j + 1, i + 1\} \)
Exact Search: KMP Automaton

- Only two positions per state: \( \{j, i\} \)
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset \( \{j + 1, i + 1\} \)
- Otherwise, the match at \( j \) cannot advance on the symbol at \( i \).
Exact Search: KMP Automaton

- Only two positions per state: \( \{j, i\} \)
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset \( \{j + 1, i + 1\} \)
- Otherwise, the match at \( j \) cannot advance on the symbol at \( i \).
- Use the fail link to get to the next shorter prefix
Exact Search: KMP Automaton

- Only two positions per state: \( \{j, i\} \)
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset \( \{j + 1, i + 1\} \)
- Otherwise, the match at \( j \) cannot advance on the symbol at \( i \).
  - Use the fail link to get to the next shorter prefix
  - Keep following fail links until you can advance
Exact Search: KMP Automaton

- Only two positions per state: \( \{j, i\} \)
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset \( \{j + 1, i + 1\} \)
- Otherwise, the match at \( j \) cannot advance on the symbol at \( i \).
  - Use the fail link to get to the next shorter prefix
  - Keep following fail links until you can advance
- Failure link chase is amortized \( O(1) \) time, while other steps are \( O(1) \) time.
KMP Algorithm

\[ \text{BuildAuto}(P[1..m]) \]

\[
\begin{align*}
    j &= 0 \\
    \text{for } i &= 1 \text{ to } m \text{ do} \\
    \quad \text{fail}[i] &= j \\
    \quad \text{while } j &> 0 \text{ and } P[i] \neq P[j] \text{ do} \\
    \quad \quad \text{fail}[j] &= j \\
    \quad j &= j + + \\
\end{align*}
\]

\[ \text{KMP}(P[1..m], S[1..n]) \]

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

- Same algorithm as on previous slide, but avoids an explicit automaton.
- Automaton state numbers are sequential, so use an integer var \( i \) to keep track of state.
- Use another variable \( j \) to keep track of second largest matching prefix
- Failure links are stored in the \textit{fail} array.
  - So, \textit{BuildAuto} only needs to construct this array.
Multi-pattern Exact Search

- Can we extend KMP to support multiple patterns?
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 KMP, not so much.
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 KMP, not so much.
- 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
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 KMP, not so much.

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
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 KMP, not so much.

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

As with KMP, McNaughton-Yamada can build an automaton similar to AC.
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 KMP, not so much.

- 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

- As with KMP, McNaughton-Yamada can build an automaton similar to AC.
  - One can understand Aho-Corasick as a specialization of McNaughton-Yamada,
  - 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 many patterns, so a state’s pset contains positions from many patterns.
    - If many patterns share a prefix, the corresponding state includes all their next positions.
    - We can retain just (one of) the longest prefix(es).
Aho-Corasick Example

Consider RE

\[(\Sigma^0)^* (t^1 o^2 p^3 s^4 \mid too^5 l^6 s^7 \mid toot^8 h^9 s^a \mid p^b e^c n^d s^e \mid o^f p^g e^h n^i s^j \mid oo^k z^l e^m s^n)\]

- To reduce clutter, positions that occur with previously numbered positions are not explicitly numbered, e.g., o's in tooth (occurs with the o's in tool)
- Figure omits failure links that go to start state.
Using arithmetic for exact matching

**Problem:** Given $P[1..n]$ and $S[1..m]$, find occurrences of $P$ in $S$ in $O(n + m)$ time.
Using arithmetic for exact matching

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

**Idea:** To simplify presentation, assume $P, S$ range over $[0-9]$
Using arithmetic for exact matching

Problem: Given $P[1..n]$ and $S[1..m]$, find occurrences of $P$ in $S$ in $O(n + m)$ time.

Idea: To simplify presentation, assume $P$, $S$ range over $[0-9]$.

- Interpret $P[1..n]$ as digits of a number
  
  \[ p = 10^{n-1}P[1] + 10^{n-2}P[2] + \cdots + 10^{n-n}P[n] \]

- Similarly, interpret $S[i..(i + n - 1)]$ as the number $s_i$
Using arithmetic for exact matching

Problem: Given $P[1..n]$ and $S[1..m]$, find occurrences of $P$ in $S$ in $O(n + m)$ time.

Idea: To simplify presentation, assume $P$, $S$ range over $[0-9]$

- Interpret $P[1..n]$ as digits of a number
  $$p = 10^{n-1}P[1] + 10^{n-2}P[2] + \cdots 10^{n-n}P[n]$$
- Similarly, interpret $S[i..(i+n-1)]$ as the number $s_i$
- Note: $P$ is a substring of $S$ at $i$ iff $p = s_i$
Using arithmetic for exact matching

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

**Idea:** To simplify presentation, assume $P$, $S$ range over $[0-9]$.

- Interpret $P[1..n]$ as digits of a number
  
  $$p = 10^{n-1}P[1] + 10^{n-2}P[2] + \cdots + 10^n P[n]$$

- Similarly, interpret $S[i..(i+n-1)]$ as the number $s_i$

- Note: $P$ is a substring of $S$ at $i$ iff $p = s_i$

- To get $s_{i+1}$:
  - shift $S[i]$ out of $s_i$: $s_{i+1} = s_i - 10^{n-1}S[i]$
  - shift in $S[i+n]$: $s_{i+1} = s_{i+1} \cdot 10 + S[i+n]$
Using arithmetic for exact matching

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

**Idea:** To simplify presentation, assume $P, S$ range over $[0-9]$

- Interpret $P[1..n]$ as digits of a number
  \[ p = 10^{n-1}P[1] + 10^{n-2}P[2] + \cdots + 10^{n-n}P[n] \]
- Similarly, interpret $S[i..(i + n - 1)]$ as the number $s_i$
- Note: $P$ is a substring of $S$ at $i$ iff $p = s_i$
- To get $s_{i+1}$:
  - shift $S[i]$ out of $s_i$: $s_{i+1} = s_i - 10^{n-1}S[i]$
  - shift in $S[i + n]$: $s_{i+1} = s_{i+1} \cdot 10 + S[i + n]$

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

- To avoid very large numbers, use computations modulo $q$ for a fixed size $q$, say, 64-bits.
- Use a random base $r$ (instead of 10) and make $q$ prime in order to minimize collisions.

What is the likelihood of false matches?

We are treating $P$ as a polynomial $p(x) = \sum_{i=1}^{n} P[n-i] \cdot x^i$.

A false match occurs when $p(x) = s(x)$, or when $p(x) - s(x) = 0$.

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.

Example: for $n = 10^6$ and 64-bit $q$, probability is just $\approx 10^{-15}$. 

$61/112$
Carter-Wegman-Rabin-Karp Algorithm

- To avoid very large numbers, use computations modulo $q$ for a fixed size $q$, say, 64-bits.
- Use a random base $r$ (instead of 10) and make $q$ prime in order to minimize collisions.

What is the likelihood of false matches?
Carter-Wegman-Rabin-Karp Algorithm

- To avoid very large numbers, use computations modulo $q$ for a fixed size $q$, say, 64-bits.
- Use a random base $r$ (instead of 10) and make $q$ prime in order to minimize collisions.

What is the likelihood of false matches?
- We are treating $P$ as a polynomial

$$p(x) = \sum_{i=1}^{n} P[n - i] \cdot x^i$$
Carter-Wegman-Rabin-Karp Algorithm

- To avoid very large numbers, use computations modulo $q$ for a fixed size $q$, say, 64-bits.
- Use a random base $r$ (instead of 10) and make $q$ prime in order to minimize collisions.

What is the likelihood of false matches?

- We are treating $P$ as a polynomial

$$p(x) = \sum_{i=1}^{n} P[n - i] \cdot x^i$$

- A false match occurs when $p(x) = s_i(x)$, or when $p(x) - s_i(x) = 0$.
- 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.
- Example: for $n = 10^6$ and 64-bit $q$, probability is just $\approx 10^{-15}$
Rolling Hashes

CWRK is an example of a rolling hash:

- Hash computed on text within a sliding window
Rolling Hashes

CWRK is an example of a rolling hash:

- 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:

\[ t_{i+1} = (t_i - x^{n-1} T[i]) \cdot x + T[i + n] \]
Rolling Hashes

CWRK is an example of a rolling hash:

- 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
Rolling Hashes

CWRK is an example of a rolling hash:

- 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
Other Rolling Hashes

In the past, multiplication/modulo were too expensive. (May still be true to some extent and/or on some hardware.)

- 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.

\[
A_l = 1 + \sum_{k=0}^{l-1} t_{i+k} \mod 65521
\]

\[
B = \sum_{k=1}^{n} A_k = n + \sum_{k=0}^{n-1} (n - k)t_{i+k} \mod 65521
\]

\[
H = (B \ll 16) + A
\]
Rolling Hash and Common Substring Problem

- To find a common substring of length $l$ or more
  - Compute rolling hashes of $P$ and $S$ with window size $l$
    - Takes $O(n + m)$ time.
  - Store hashes from $P$ in a hash table
  - For each rolling hash from $S$, check if it is in the table
    - Effectively, $O(nm)$ comparisons, so expected number of collisions increases.
      - Unless collision probability is $O(1/nm)$, expected runtime can be nonlinear

- Can find longest common substring (LCS) using a binary-search like process, with a total complexity of $O((n + m) \log(n + m))$
RSYNC: Synchronizes directories across network
   - Need to minimize data transferred
     - A diff requires entire files to be copied to client side first!
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

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).
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]

- Versatile data structure: wide applications in string search, computational biology
- “Compressed” trie of all suffixes of a string appended with “$”
**Suffix Trees [Weiner 1973]**

- Versatile data structure: wide applications in string search, 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!
Suffix Trees [Weiner 1973]

- Versatile data structure: wide applications in string search, 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.
Suffix Trees [Weiner 1973]

- Versatile data structure: wide applications in string search, 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!
Suffix Tree Example

Key Property Behind Suffix Trees

Substrings are prefixes of suffixes

- Failure links used only during construction
- Uses end-marker “$”
- Leaves identify starting position of suffix
- Typically, we preprocess the text, not the pattern.

Images from Wikipedia commons
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 $t$. 

![Suffix Tree Diagram](image_url)
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 *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 Trees

Suffix trees for multiple strings $p_1, \ldots, p_n$

**Example.** att, tag, gat

Simple solution:
1. build suffix tree for string $aat\#_1tag\#_2gat\#_3$
2. For every leaf node, remove any text after the first $\#$ symbol.
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 $p_1, \ldots, p_n$

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 [Manber and Myers 1989]

- **Drawbacks of suffix trees:**
  - Multiple pointers per internal node: significant storage costs
  - Pointer-chasing is not cache-friendly

- **Suffix arrays address these drawbacks.**
  - Requires same asymptotic storage ($O(n)$) but constant factors a lot smaller — 4x or so.
  - Instead of navigating down a path in the tree, relies on binary search
    - Increases asymptotic cost by $O(\log n)$, but can be faster in practice due to better cache performance etc.
Suffix Arrays

- Construct a sorted array of suffixes, rather than tries
  - Can use 2 to 4 bytes per symbol

<table>
<thead>
<tr>
<th>( i )</th>
<th>( T_i )</th>
<th>( A_i )</th>
<th>( T_{A_i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mississippi$</td>
<td>12</td>
<td>$</td>
</tr>
<tr>
<td>2</td>
<td>ississippi$</td>
<td>11</td>
<td>i$</td>
</tr>
<tr>
<td>3</td>
<td>ssissippi$</td>
<td>8</td>
<td>ippi$</td>
</tr>
<tr>
<td>4</td>
<td>sissippi$</td>
<td>5</td>
<td>issippi$</td>
</tr>
<tr>
<td>5</td>
<td>issippi$</td>
<td>2</td>
<td>ississippi$</td>
</tr>
<tr>
<td>6</td>
<td>ssippi$</td>
<td>1</td>
<td>mississippi$</td>
</tr>
<tr>
<td>7</td>
<td>sippi$</td>
<td>10</td>
<td>pi$</td>
</tr>
<tr>
<td>8</td>
<td>ippi$</td>
<td>9</td>
<td>ppi$</td>
</tr>
<tr>
<td>9</td>
<td>ppi$</td>
<td>7</td>
<td>sippi$</td>
</tr>
<tr>
<td>10</td>
<td>pi$</td>
<td>4</td>
<td>sissippi$</td>
</tr>
<tr>
<td>11</td>
<td>i$</td>
<td>6</td>
<td>ssippi$</td>
</tr>
<tr>
<td>12</td>
<td>$</td>
<td>3</td>
<td>ssissippi$</td>
</tr>
</tbody>
</table>

- Use binary search to locate suffixes etc.
Finding Suffix Arrays

- Maintaining LCP of successive suffixes speeds up algorithms
  - Search for substring $p$ in $O(\mid p \mid + \log \mid t \mid)$
  - Count number of occurrences of $p$ in $O(\mid p \mid + \log \mid t \mid)$ time
  - Search for longest common repeat $O(\mid t \mid)$ time

<table>
<thead>
<tr>
<th>$i$</th>
<th>$T_i$</th>
<th>$A_i$</th>
<th>$T_{A_i}$</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mississippi$</td>
<td>12</td>
<td>$</td>
<td>⊥</td>
</tr>
<tr>
<td>2</td>
<td>ississippi$</td>
<td>11</td>
<td>i$</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>ssissippi$</td>
<td>8</td>
<td>ippi$</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>sissippi$</td>
<td>5</td>
<td>ississippi$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>ississippi$</td>
<td>2</td>
<td>ississippi$</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ssippi$</td>
<td>1</td>
<td>mississippi$</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>sippi$</td>
<td>10</td>
<td>pi$</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>ippi$</td>
<td>9</td>
<td>ppi$</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>ppi$</td>
<td>7</td>
<td>sippi$</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>pi$</td>
<td>4</td>
<td>sissippi$</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>i$</td>
<td>6</td>
<td>ssippi$</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>$</td>
<td>3</td>
<td>ssissippi$</td>
<td>3</td>
</tr>
</tbody>
</table>

- Use binary search to locate suffixes etc.
Alternative Approaches for Exact Search

- DFA approach had significant preprocessing ("compiling") costs, but optimized runtime — exactly $m$ comparisons.
- KMP reduces compile-time\(^1\) by shifting more work (up to $2m$ comparisons) to runtime.
  - DFA states contain information about all matching prefixes, but KMP states retain just the two longest ones.
  - Other prefixes are essentially being computed at runtime by following fail links.
- Can we remember even less in automaton states?
  - Can we leave all matching prefixes to be computed at runtime?

\(^1\)while also simplifying automaton structure
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”
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.
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:

$$T = [(T \& \delta_{S[k]}) \ll 1] | 1$$
Shift-And Method Illustration

\[ k = 11 \]

| \( S \) | aabcdaababaaba |
| \( P \) | aababaa |
| \( T \) | 00100001 |
| \( \delta_a \) | 01101011 |
| \( T_{new} \) | 01000011 |
Shift-And Method Illustration

\[ k = 12 \]

\[ S \quad aabcdaababaaba \]

\[ P \quad aababaa \]

\[ T \quad 01000011 \]

\[ \delta_a \quad 01101011 \]

\[ T_{\text{new}} \quad 10000111 \]
Shift-And Method Illustration

\[ k = 13 \]

\[
\begin{align*}
S & \quad \text{aabcedaababa} \\
P & \quad \text{aababaa} \\
T & \quad \text{10000111} \\
\delta_b & \quad \text{00010100} \\
T_{\text{new}} & \quad \text{00001001}
\end{align*}
\]
Shift-And Method Illustration

\[ k = 14 \]

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
</tr>
<tr>
<td>( P )</td>
</tr>
<tr>
<td>( T )</td>
</tr>
<tr>
<td>( \delta_a )</td>
</tr>
<tr>
<td>( T_{\text{new}} )</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
Levenshtein Automaton

No errors permitted.
Levenshtein Automaton

- Up to one missing character (deletion).
Levenshtein Automaton

- Up to one deletion and one insertion.
Levenshtein Automaton

- Up to one deletion, or insertion, or substitution
Levenshtein Automaton

- Up to a total of two deletions, insertions, or substitution
Compare with:
- Structure of cost matrix for edit-distance problem
- Finding least-cost paths from \( T_0 \) to \( T_7 \), \( U_7 \) or \( V_7 \)

Illustrates the relationship between shortest path and edit-distance problem.
Matching Using Levenshtein Automaton

- Convert to DFA (subset construction)
  - Potentially $O(n^k)$ states, where $k$ is the max edit distance permitted
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.
We extend the notation to explicitly include of the current position $k$ in $T$. With this extension, our original equation for $T$ becomes

$$T^k = 1 \mid [ (T^{k-1} \& \delta_{S[k]}) \ll 1 ]$$
We extend the notation to explicitly include the current position $k$ in $T$. With this extension, our original equation for $T$ becomes

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

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

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

$$T^{k-1} \ll 1$$  // move ↑ with substitution

$$T^k \ll 1$$  // move ↑ with deletion

$$[(U^{k-1} \& \delta_{S[k]}) \ll 1]|1$$  // move →
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 $l$
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
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.