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

Terminology

String: List $S[1..i]$ of characters over an alphabet Σ.

Terminology

String: List $S[1..i]$ of characters over an alphabet Σ.

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 Σ.

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 Σ.

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 Σ.

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 Σ.

Let R be the set of all regular expressions over Σ. 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 $\{\mathrm{a}\}$

$a \mid b$: stands for the set $\{\mathrm{a}, \mathrm{b}\}$

- *Union* of sets corresponding to REs a and b

ab : stands for the set $\{\mathrm{ab}\}$

- Analogous to set *product* on REs for a and b
  - $(a|b)(a|b)$: stands for the set $\{\mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}\}$.

a^* : stands for the set $\{\epsilon, \mathrm{a}, \mathrm{aa}, \mathrm{aaa}, \ldots\}$ 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.

$$
\begin{array}{rcl}
\mathcal{L}(\epsilon) & = & \{\epsilon\} \\
\mathcal{L}(\alpha) & = & \{\alpha\} \qquad (\alpha \in \Sigma) \\
\mathcal{L}(r_1 \mid r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1\ r_2) & = & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(r^*) & = & \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))
\end{array}
$$

Finite State Automata

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

FSAs are represented by a labeled directed graph.

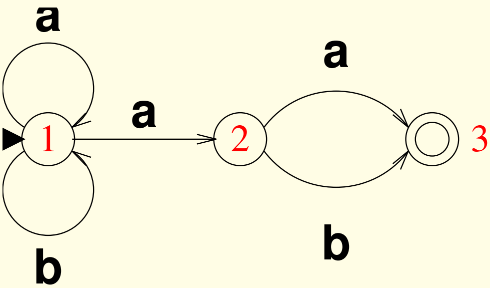
- A finite set of *states* (vertices).
- *Transitions* between states (edges).
- *Labels* on transitions are drawn from $\Sigma \cup \{\epsilon\}$.
- One distinguished *start* state.
- One or more distinguished *final* states.

Finite State Automata: An Example

Consider the Regular Expression $(a \mid b)^* a(a \mid b)$.

$\mathcal{L}((a \mid b)^* a(a \mid b)) = \{\text{aa}, \text{ab}, \text{aaa}, \text{aab}, \text{baa}, \text{bab}, \text{aaaa}, \text{aaab}, \text{abaa}, \text{abab}, \text{baaa}, \ldots\}$.

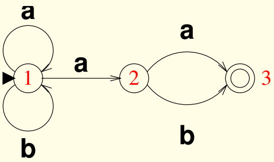
The following (non-deterministic) automaton determines whether an input string belongs to $\mathcal{L}((a \mid b)^* a(a \mid b)$:



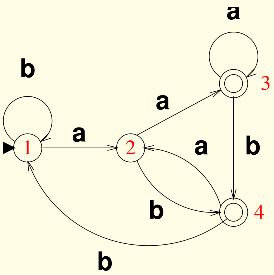
Determinism

$(a \mid b)^{*}a(a \mid b)$:

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.

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

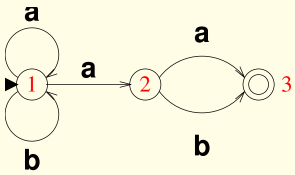
. . . such that the sequence of edge labels spells x

. . . 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 $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?

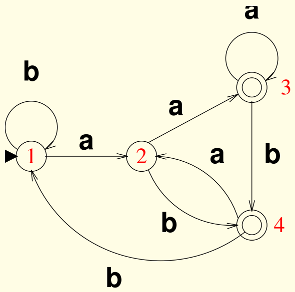


Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\perp$	$\perp$	

Accept

Recognition with a DFA

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



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 ϵ.

  (Spontaneous transitions)
- All transition labels in a DFA belong to Σ.
- 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 = Size of Regular Expression (pattern)

m = Length of Input String (subject)

	NFA	**DFA**
Size of Automaton	$O(n)$	$O(2^n)$
Recognition time per input string	$O(n \times m)$	$O(m)$

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.

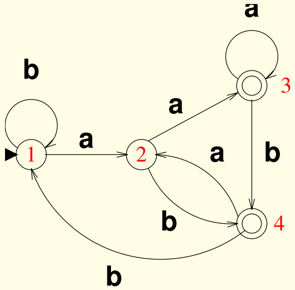
Converting RE to FSA

- 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 $\boxed{R = (a^1|b^2)* \, a^3(a^4|b^5)\$^6}$: $@(R, 1) = a$, $@(R, 5) = b$.

- $filter(R, P, s)$: $\{p \in P \mid @(R, p) = s\}$

  Example: $filter(R, \{1, 2, 3\}, a) = \{1, 3\}$

  $filter(R, \{1, 2, 4, 5\}, b) = \{2, 5\}$

- $first(R)$: First positions in R

$$\begin{array}{rcl} first(a) & = & pos(a) \\ first(R_1|R_2) & = & first(R_1) \cup first(R_2) \\ first(R_1 \cdot R_2) & = & first(R_1), \text{ if } R_1 \text{ doesn't match } \epsilon \\ first(R_1 \cdot R_2) & = & first(R_1) \cup first(R_2), \text{ otherwise} \\ first(R*) & = & first(R) \end{array}$$

Example: $first(R) = \{1, 2, 3\}$

McNaughton-Yamada: Definitions (Continued)

- $\mathit{follow}(R, p)$: Positions immediately following p in R.

$$\begin{aligned} \mathit{follow}(R_1 \cdot R_2, p) &\supseteq \mathit{first}(R_2), \text{ if } p \text{ is rightmost in } R_1 \\ \mathit{follow}(R*, p) &\supseteq \mathit{first}(R), \text{ if } p \text{ is rightmost in } R \end{aligned}$$

Example for $\boxed{R = (a^1|b^2)*\, a^3(a^4|b^5)\$^6}$:

$\mathit{follow}(R, 1) = \{1, 2, 3\}$

$\mathit{follow}(R, 2) = \{1, 2, 3\}$

$\mathit{follow}(R, 3) = \{4, 5\}$

$\mathit{follow}(R, 4) = \{6\}$

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

Example: $\mathit{follow}(R, \{3, 4\}) = \{4, 5, 6\}$

McNaughton-Yamada Algorithm

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

Create an automaton state S labeled P

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

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

  Call $BuildMY(R, follow(R, filter(R, P, a))$ 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 $BuildMY(R, follow(\{0\}))$. The root of the resulting automaton is marked as a start state.

BuildMY Illustration on $R = {}^0(a^1|b^2)*\, a^3(a^4|b^5)\$^6$

Computations Needed

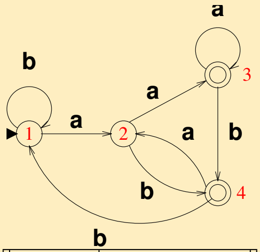
$follow(\{0\}) = \{1,2,3\}$
$follow(\{1\}) = follow(\{2\}) = \{1,2,3\}$
$follow(\{3\}) = \{4,5\}$
$follow(\{4\}) = follow(\{5\}) = \{6\}$

$filter(\{1,2,3\},a) = \{1,3\}$, $filter(\{1,2,3\},b) = \{2\}$
$follow(\{1,3\}) = \{1,2,3,4,5\}$

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

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

Resulting Automaton



State	Pset
1	{1,2,3}
2	{1,2,3,4,5}
3	{1,2,3,4,5,6}
4	{1,2,3,6}

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.

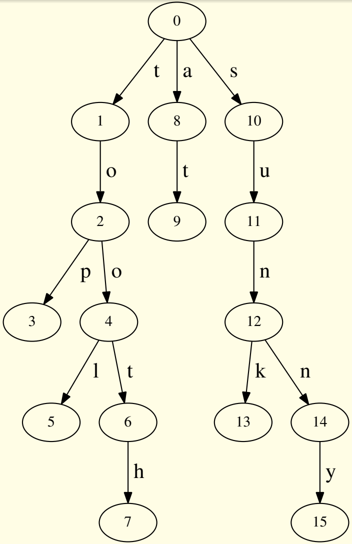
String Lookup (Not Search)

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

$$R_0 = top|tool|tooth|at|sunk|sunny$$



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: Σ^* matches any arbitrary string (incl. ϵ)
- 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

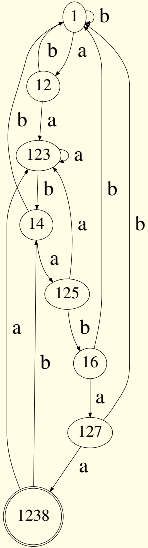
Consider $R_0 = (\Sigma^0)^* a^1 a^2 b^3 a^4 b^5 a^6 a^7 \$^8$

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 ϵ, a and aa.

- Or equivalently, Σ^*, $\Sigma^* a$ 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?

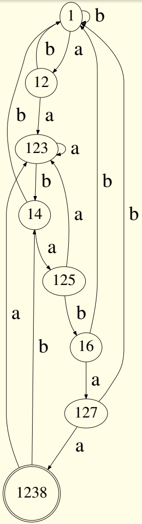
Improving Exact Search: Observations

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

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.

S	a	a	b	a	b	a	a	$\cdots\cdots\cdots$
Viable match 1	a^1	a^2	b^3	a^4	b^5	a^6	a^7	$\$^8$
Viable match 2	Σ	Σ	Σ	Σ	Σ	a^1	a^2	b^3
Viable match 3	Σ	Σ	Σ	Σ	Σ	Σ	a^1	a^2
Viable match 4	Σ	Σ	Σ	Σ	Σ	Σ	Σ	a^1

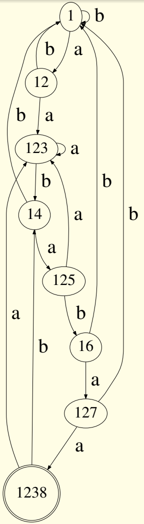


Improving Exact Search: Key Ideas

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

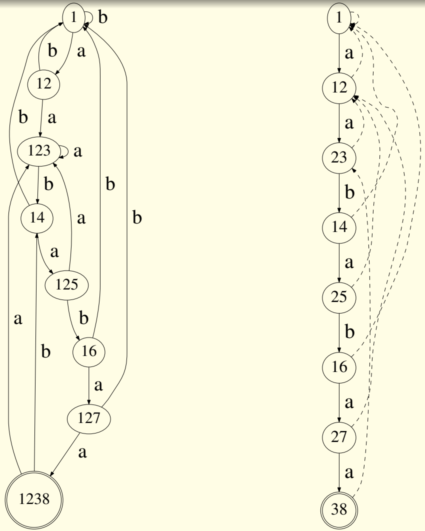
Main Idea

- 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 $pset \Longrightarrow O(n)$ 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 $pset \Longrightarrow O(n)$ size



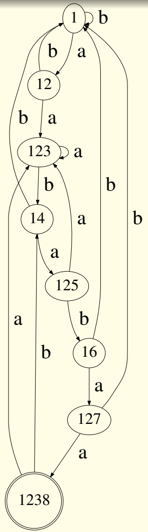
Exact Search: KMP Automaton

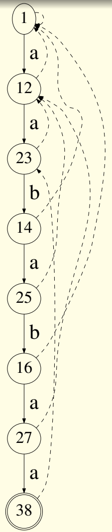
- Only two positions per state: $\{j, i\}$
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset $\{j+1, i+1\}$



Exact Search: KMP Automaton

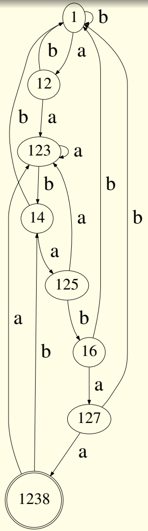
- Only two positions per state: $\{j, i\}$
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset $\{j+1, i+1\}$
- Otherwise, the match at j cannot advance on the symbol at i.

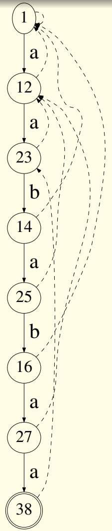




Exact Search: KMP Automaton

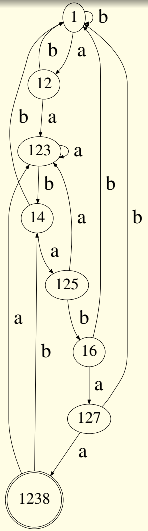
- Only two positions per state: $\{j, i\}$
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset $\{j+1, i+1\}$
- Otherwise, the match at j cannot advance on the symbol at i.
  - Use the fail link to get to the next shorter prefix

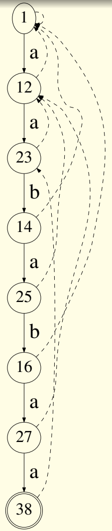




Exact Search: KMP Automaton

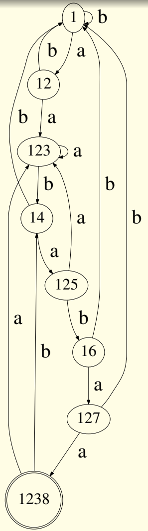
- Only two positions per state: $\{j, i\}$
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset $\{j + 1, i + 1\}$
- Otherwise, the match at j cannot advance on the symbol at i.
  - Use the fail link to get to the next shorter prefix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

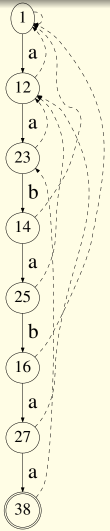




Exact Search: KMP Automaton

- 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

$BuildAuto(P[1..m])$

$j = 0$
for $i = 1$ **to** m **do**
  $fail[i] = j$
  while $j > 0$ **and** $P[i] \neq P[j]$ **do**
    $j = fail[j]$
  $j{+}{+}$

$KMP(P[1..m], S[1..n])$

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

- 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 *fail* array.
    - So, *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

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.

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

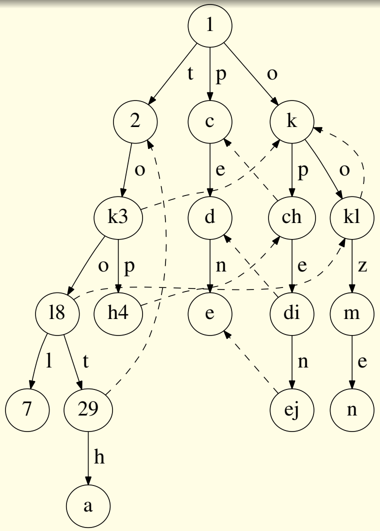
- As with KMP, we can think of AC as a specialization of MY.
  - Retain just the largest two numbers i and j in the pset.
  - Use the value of j as target for failure link, and to find j' in the successor state's pset $\{j', i+1\}$
- But there is an extra wrinkle:
  - With KMP, there is one pattern; 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

$$(\Sigma^0)^*(t^1o^2p^3\$^4|too^5l^6\$^7|toot^8h^9\$^a$$
$$|p^be^cn^d\$^e|o^fp^ge^hn^i\$^j|oo^kz^le^m\$^n)$$

- 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

$$p = 10^{n-1}P[1] + 10^{n-2}P[2] + \cdots 10^{n-n}P[n]$$

- 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

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 $O((n+m)\log(n+m))$

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!

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

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]

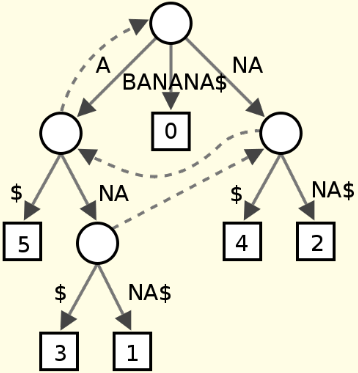
- Versatile data structure: wide applications in string search, computational biology
- *"Compressed" trie of all suffixes of a string appended with "$"*
  - Linear chains in the trie are compressed
    - Edges can now be substrings.
    - Each state has at least two children.
  - Leaves identify starting 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

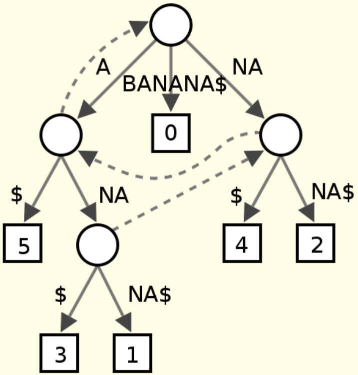
Finding Substrings and Suffixes

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.

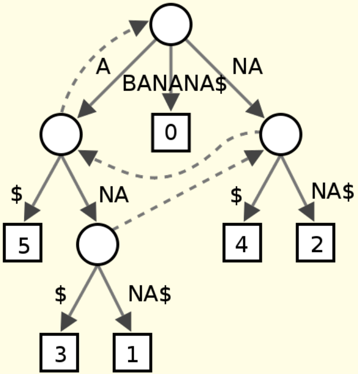


Counting # of Occurrences of p

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.

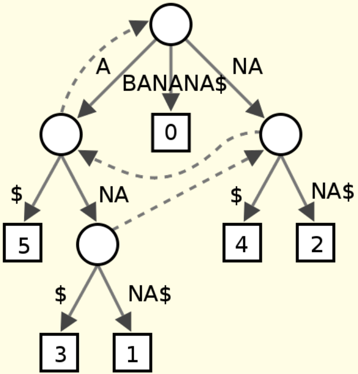


Self-LCS (Or, Longest Common Repeat)

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 *ana*.

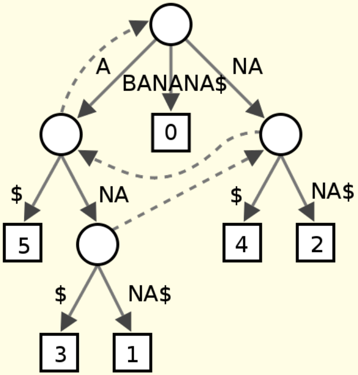


LC extension of *i* and *j*

Longest Common Extension

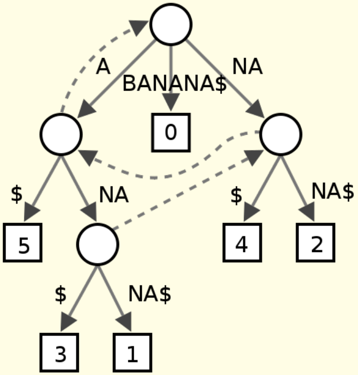
Longest common prefix of suffixes starting at *i* and *j*

- 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*



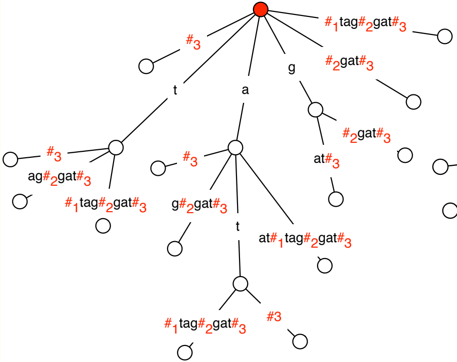
Generalized Suffix Trees

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

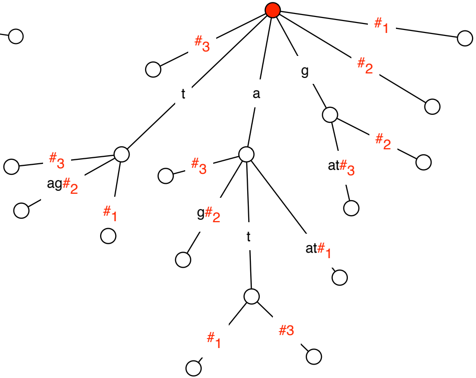
Example. att, tag, gat

Simple solution:

(1) build suffix tree for string aat$\#_1$tag$\#_2$gat$\#_3$



(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
- Use binary search to locate suffixes etc.

i	T_i	A_i	T_{A_i}
1	mississippi$	12	$
2	ississippi$	11	i$
3	ssissippi$	8	ippi$
4	sissippi$	5	issippi$
5	issippi$	2	ississippi$
6	ssippi$	1	mississippi$
7	sippi$	10	pi$
8	ippi$	9	ppi$
9	ppi$	7	sippi$
10	pi$	4	sissippi$
11	i$	6	ssippi$
12	$	3	ssissippi$

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 $O(|t|)$ time
- Use binary search to locate suffixes etc.

i	T_i	A_i	T_{A_i}	LCP
1	mississippi$	12	$	$\perp$
2	ississippi$	11	i$	0
3	ssissippi$	8	ippi$	1
4	sissippi$	5	issippi$	1
5	issippi$	2	ississippi$	4
6	ssippi$	1	mississippi$	0
7	sippi$	10	pi$	0
8	ippi$	9	ppi$	1
9	ppi$	7	sippi$	0
10	pi$	4	sissippi$	2
11	i$	6	ssippi$	1
12	$	3	ssissippi$	3