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

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 {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 $\{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$ 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.

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

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.

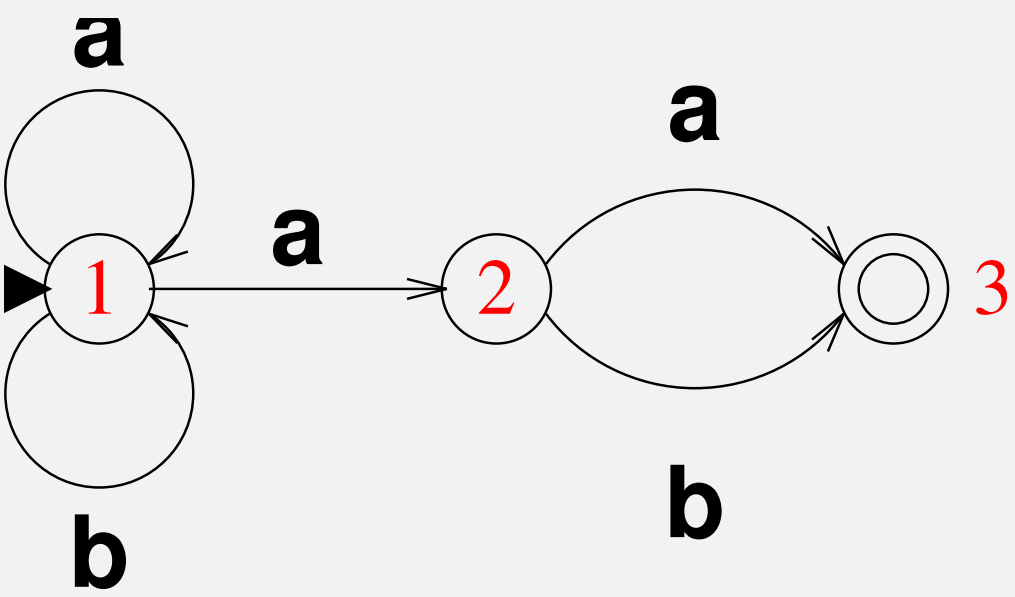
Finite State Automata: An Example

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

$\mathcal{L}((a \mid b)^* a(a \mid b)) = \{\mathtt{aa}, \mathtt{ab}, \mathtt{aaa}, \mathtt{aab}, \mathtt{baa}, \mathtt{bab},$

$\mathtt{aaaa}, \mathtt{aaab}, \mathtt{abaa}, \mathtt{abab}, \mathtt{baaa}, \ldots\}$.

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

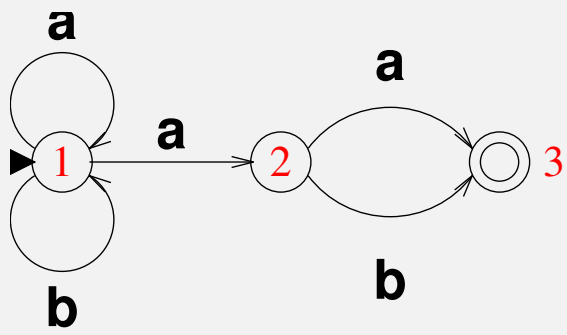


Determinism

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

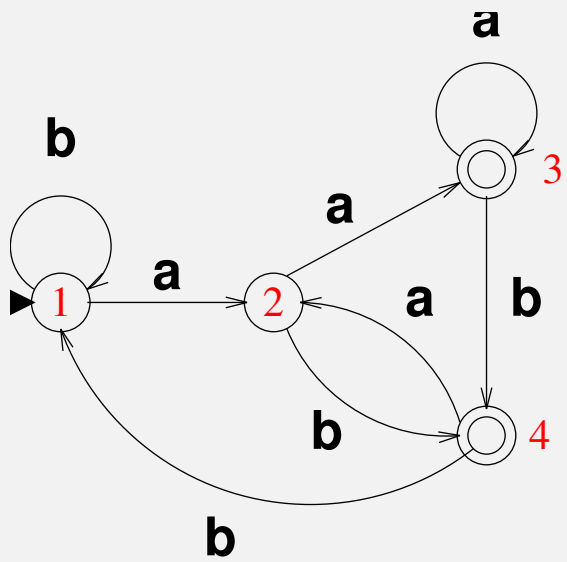
Nondeterministic:

(NFA)



Deterministic:

(DFA)



Acceptance Criterion

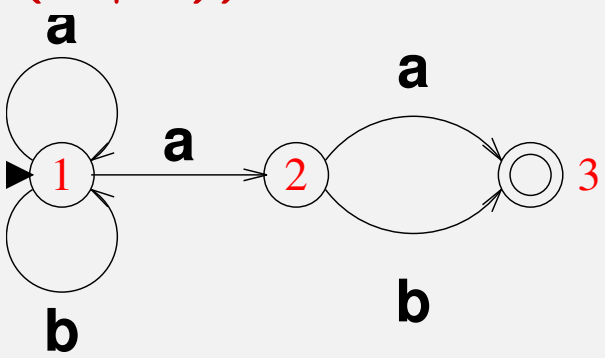
A finite state automaton (NFA or DFA) *accepts* an input string x

- ... 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.

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

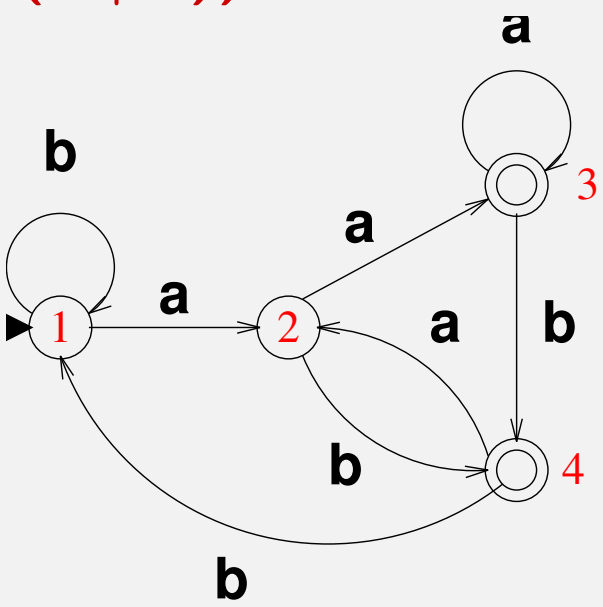


Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\perp$	$\perp$	

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.

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 $\mathcal{L}(R) = \mathcal{L}(xR')$

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

Examples

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

Definition of RE Derivative (1)

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

$$
\begin{aligned}
\mathit{inclEps}(a) &= \mathit{false}, \quad \forall a \in \Sigma \\
\mathit{inclEps}(R_1|R_2) &= \mathit{inclEps}(R_1) \vee \mathit{inclEps}(R_2) \\
\mathit{inclEps}(R_1R_2) &= \mathit{inclEps}(R_1) \wedge \mathit{inclEps}(R_2) \\
\mathit{inclEps}(R*) &= \mathit{true}
\end{aligned}
$$

Note *inclEps* can be computed in linear-time.

Definition of RE Derivative (2)

$$
\begin{array}{rcll}
\partial_a[a] & = & \epsilon & \\
\partial_a[b] & = & \emptyset & \\
\partial_a[R_1|R_2] & = & \partial_a[R_1]|\partial_a[R_2] & \\
\partial_a[R*] & = & \partial_a[R]R* & \\
\partial_a[R_1R_2] & = & \partial_a[R_1]R_2|\partial_a[R_2] & \text{if } \mathit{inclEps}(R_1) \\
 & = & \partial_a[R_1]R_2 & \text{otherwise}
\end{array}
$$

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

DFA Using Derivatives: Illustration

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

$\partial_a[R_1] = R_1|(a|b) = R_2$

$\partial_b[R_1] = R_1$

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

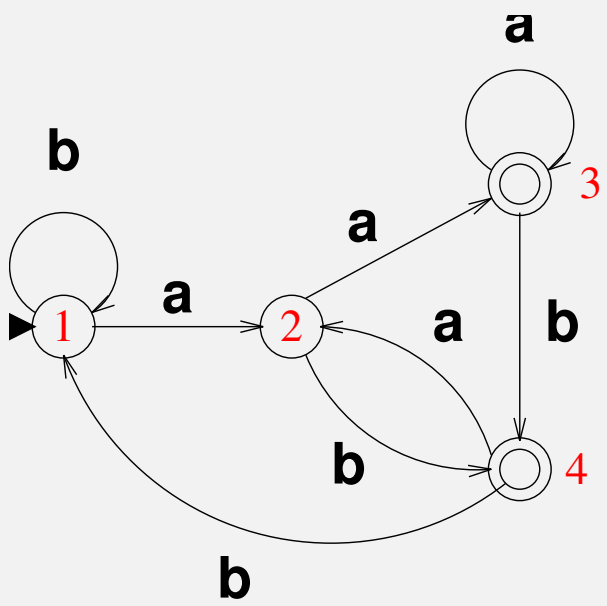
$\partial_b[R_2] = R_1|\epsilon = R_4$

$\partial_a[R_3] = R_1|(a|b)|\epsilon = R_3$

$\partial_b[R_3] = R_1|\epsilon = R_4$

$\partial_a[R_4] = R_1|(a|b) = R_2$

$\partial_b[R_4] = R_1$

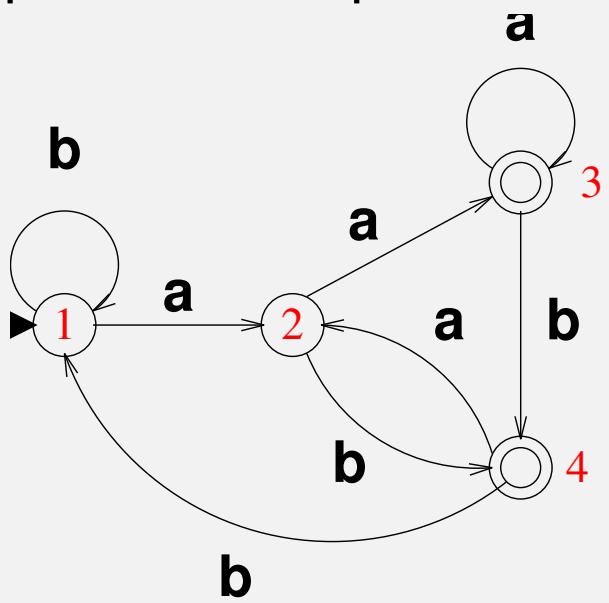


McNaughton-Yamada Construction

Can be viewed as a simpler way to represent derivatives

- Positions in RE are numbered, e.g., ${}^{0}(a^{1}|b^{2})* a^{3}(a^{4}|b^{5})\$^{6}$.
- 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)

$$
\begin{aligned}
R_1 &\equiv \{1,2,3\} \\
R_2 &\equiv \{1,2,3,4,5\} \\
R_3 &\equiv \{1,2,3,4,5,6\} \\
R_4 &\equiv \{1,2,3,6\}
\end{aligned}
$$



McNaughton-Yamada: Definitions

first(P): Yields the set of first symbols of RE denoted by pset P

Determines the transitions out of DFA state for P

Example: For the RE $(a^1|b^2)* a^3(a^4|b^5)\$^6$,

$$first(\{1,2,3\}) = \{a,b\}$$

$P|_s$: Subset of P that contain s, i.e., $\{p \in P \mid R \text{ contains } s \text{ at } p\}$

Example: $\{1,2,3\}|_a = \{1,3\}$, $\{1,2,4,5\}|_b = \{2,5\}$

follow(P): Yields the set of positions that immediately follow P.

Note: $follow(P) = \bigcup_{p \in P} follow(\{p\})$

Definition is very similar to derivatives

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

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

McNaughton-Yamada Construction (2)

$BuildMY(R, pset)$

Create an automaton state S labeled *pset*

Mark this state as final if \$ occurs in R at *pset*

foreach symbol $x \in first(pset) - \{\$\}$ **do**

Call $BuildMY(R, follow(pset|_x))$ 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 $BuildMY(R, follow(\{0\}))$. 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

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

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

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

$follow(\{4\}) = follow(\{5\}) = \{6\}$

$\{1,2,3\}|_a = \{1,3\}, \quad \{1,2,3\}|_b = \{2\}$

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

$\{1,2,3,4,5\}|_a = \{1,3,4\}$

$\{1,2,3,4,5\}|_b = \{2,5\}$

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

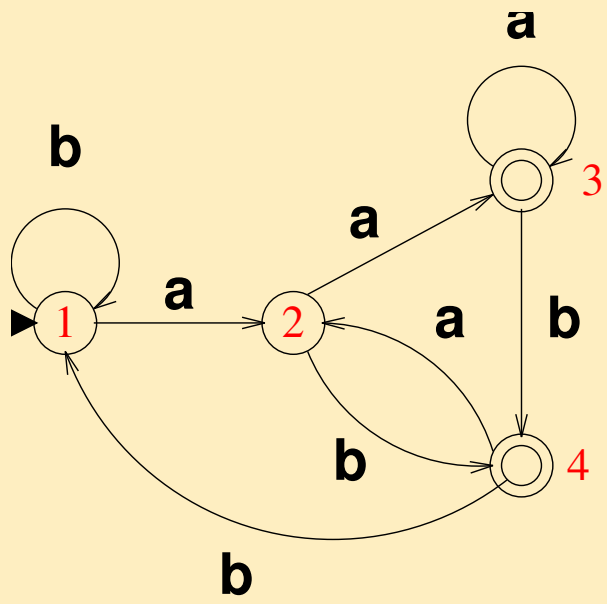
$follow(\{2,5\}) = \{1,2,3,6\}$

$\{1,2,3,4,5,6\}|_a = \{1,3,4\}$

$\{1,2,3,4,5,6\}|_b = \{2,5\}$

$\{1,2,3,6\}|_a = \{1,3\} \quad \{1,2,3,6\}|_b = \{2\}$

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}

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

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

$R_1 = \partial_t[R_0] = op|ool|ooth$

$R_2 = \partial_o[R_1] = p|ol|oth$

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

$R_4 = \partial_o[R_2] = l|th$

$R_5 = \partial_l[R_4] = \epsilon$

$R_6 = \partial_t[R_4] = h, R_7 = \partial_h[R_6] = \epsilon$

$R_8 = \partial_a[R_0] = t, R_9 = \partial_t[R_8] = \epsilon$

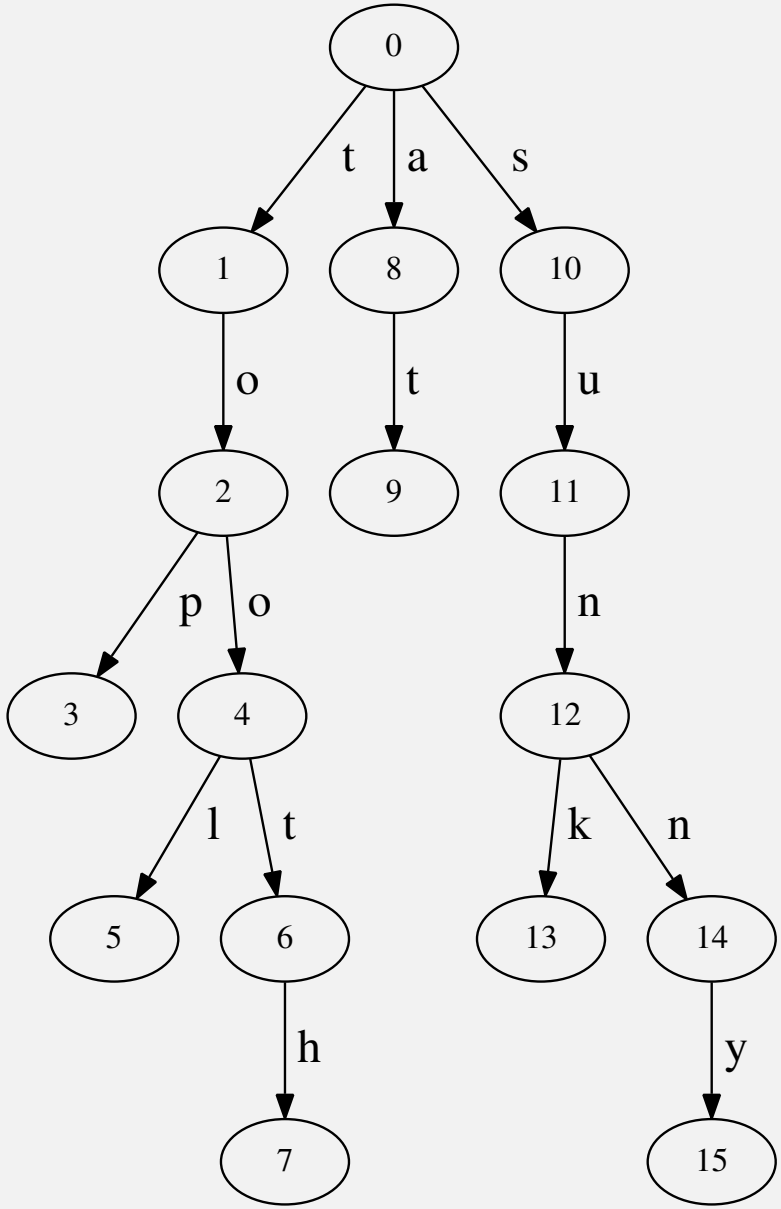
$R_{10} = \partial_s[R_0] = unk|unny$

$R_{11} = \partial_u[R_{10}] = uk|nny$

$R_{12} = \partial_n[R_{11}] = k|ny$

$R_{13} = \partial_k[R_{12}] = \epsilon$

$R_{14} = \partial_n[R_{12}] = y, R_{15} = \partial_y[R_{14}] = \epsilon$



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: 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

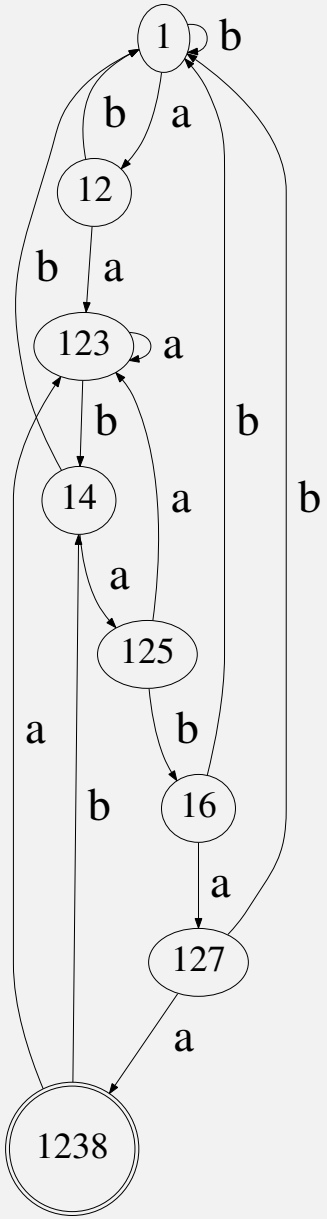
Consider $R_0 = (\Sigma^0)^* a^1 a^2 b^3 a^4 b^5 a^6 a^7 \$^8$

We use McNaughton-Yamada. Recall that 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

$$R_0 | a^2 b^3 a^4 b^5 a^6 a^7 | b^3 a^4 b^5 a^6 a^7$$



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

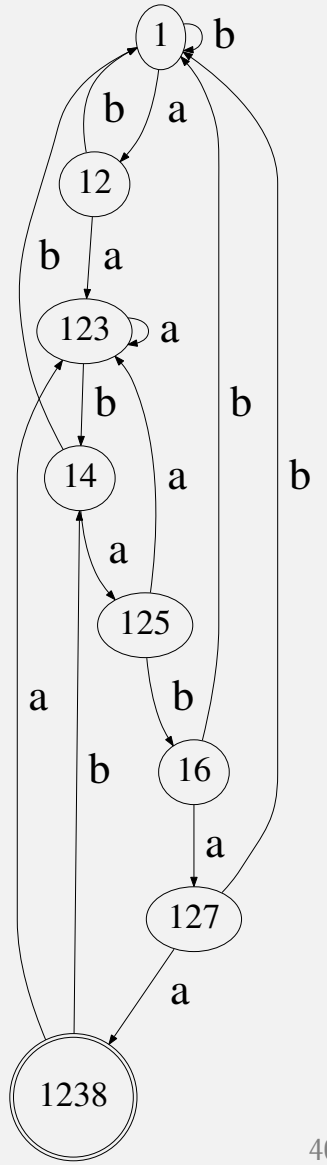
$(\Sigma^0)^* a^1 a^2 b^3 a^4 b^5 a^6 a^7 \$^8$

The DFA has a linear structure, with states 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 $P' = P[1..i-1] = S[k-i+2..k]$
  - *"Unwinding" of Σ^*:* A prefix of $S[k-i+2..k]$ can be matched with Σ^*, with the rest matching $P[1..j-1]$

  So, $pset(i)$ includes every j such that $S[k-i+2..k] = P[1..j-1] = P[..i-1]$

S	a	a	b	a	b	a	a	
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

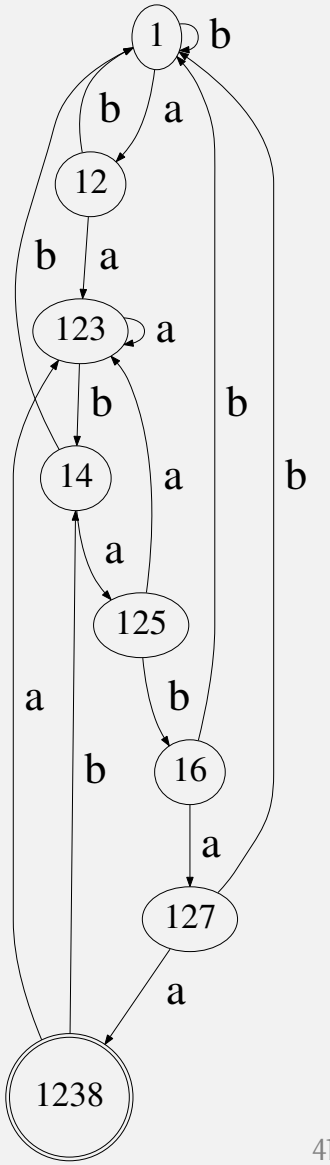


Improving Exact Search: Key Ideas

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

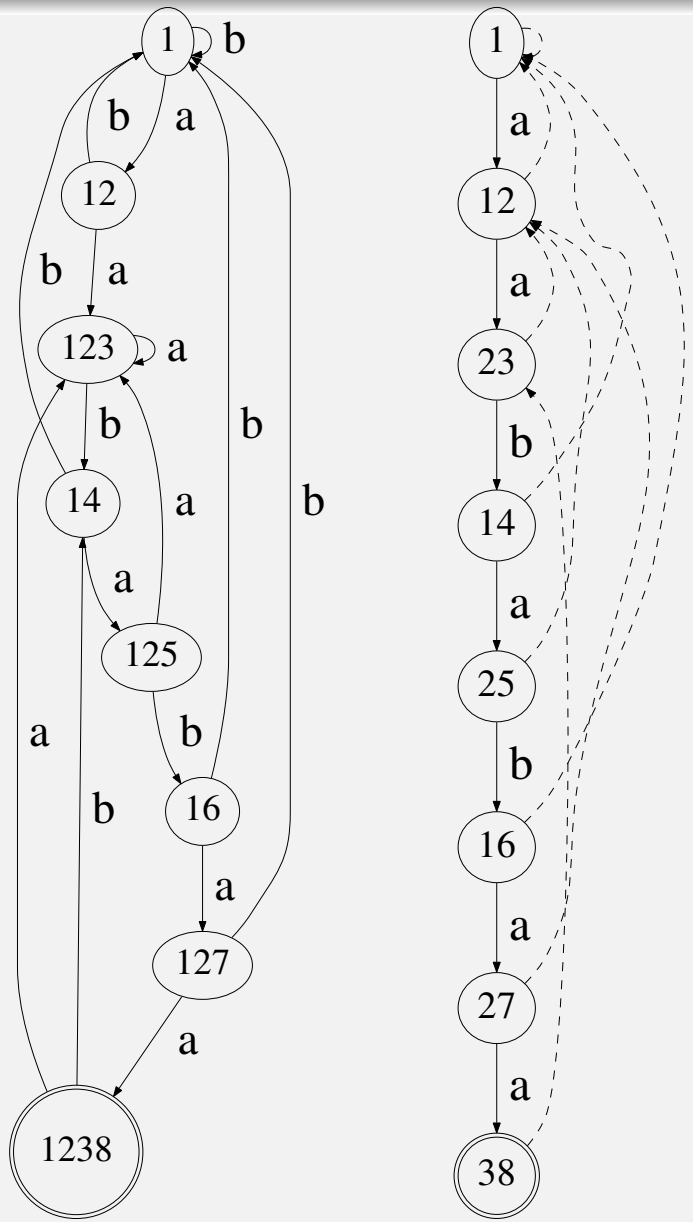
Main Idea

- Remember only the 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 $pset \Longrightarrow O(n)$ construction time



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. So, we use the fail link to identify the next shorter prefix that can advance:
  - Follow fail link to state u with pset $\{k, j\}$ and see if that match can advance
  - Otherwise, follow the fail link from u and so on.
- 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** $\underline{T[i]} \neq P[j]$ **do**
    $j = fail[j]$
  $j++$
  if $j > m$ **then return** $i - m + 1$

- 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 *fail*
- *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

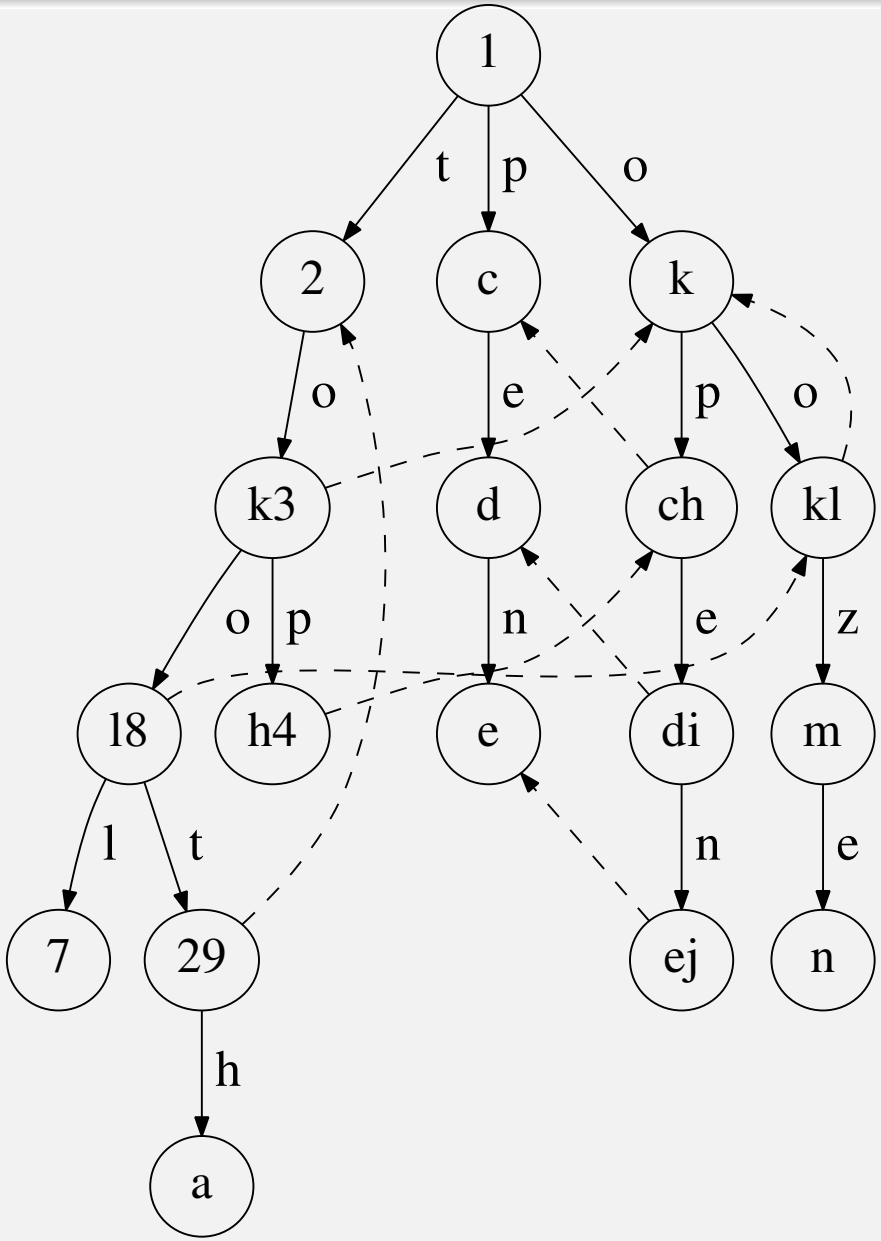
- As with KMP, we can think of AC as a specialization of MY.
  - Retain just the largest two numbers i and j in the pset.
  - Use the value of j as target for failure link, and to find j' in the successor state's pset $\{j', i+1\}$
- But there is an extra wrinkle:
  - With KMP, there is one pattern;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

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

- 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.



Alternative Approaches for Exact Search

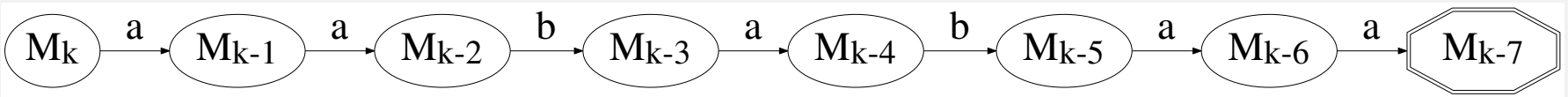
- DFA approach had significant preprocessing ("compiling") costs, but optimized runtime — exactly m comparisons.
- KMP reduces compile-time[1] by shifting more work (up to $2m$ comparisons) to runtime.
  - DFA states contain information about all matching prefixes, but KMP states retain just the two longest ones.
  - Other prefixes are essentially being computed at runtime by folowing 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

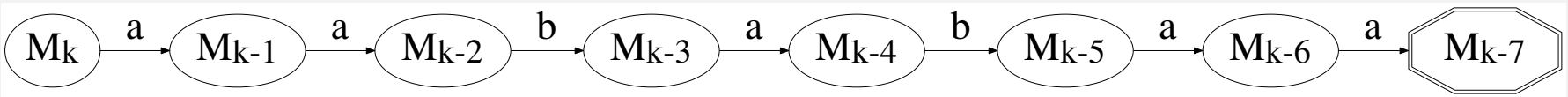
- *Key Idea:* Maintain all matching prefixes at runtime.
- Simultaneously advance the state of all these prefixes after reading next input character.
  - So, there will only be $O(m)$ "comparisons" total.
- *How can we do this?*
  - Think of KMP automaton, strip off all failure links
  - The automaton 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

Bit-parallel Exact Search: Shift-And Method



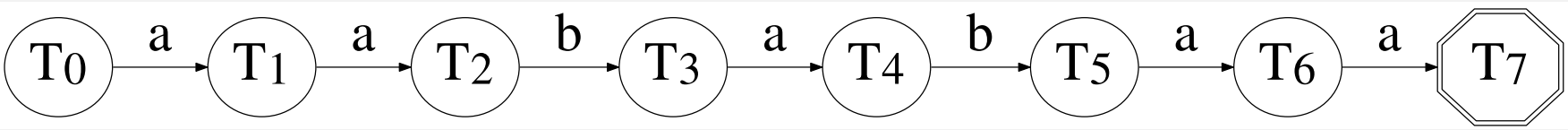
- A search for match begins on each symbol $S[k]$ in input
  - Think of placing a token in start state each time you slide P over S
  - If symbols continue to match, tokens advance through successive states until they reach the final state.
  - If there is a mismatch, the corresponding token "disappears" or "dies"

Bit-parallel Exact Search: Shift-And Method



- 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.

Bit-parallel Exact Search: Shift-And Method



- 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
δ_a     01101011
T_new    01000011
```

Shift-And Method Illustration

```
                  k=12
                   |
S     aabcdaababaaba
P        aababaa
T        01000011
δ_a      01101011
T_new      10000111
```

Shift-And Method Illustration

```
                  k=13
                   |
S     aabcdaababaaba
P           aababaa
T           10000111
δ_b         00010100
T_new         00001001
```

Shift-And Method Illustration

```
                    k=14
                     |
S    aabcdaababaaba
P          aababaa
T          00001001
δ_a        01101011
T_new       00010001
```

Approximate Search

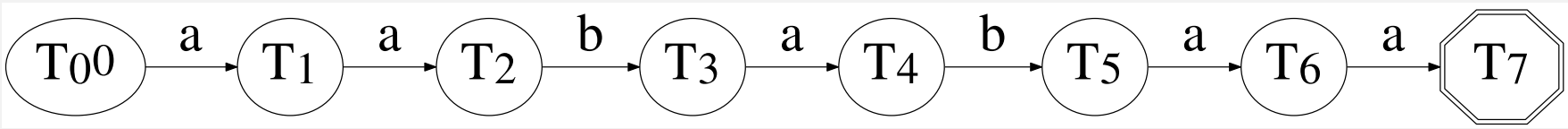
Approach 1: Use edit-distance algorithm

- Expensive
- Does not allow for multiple patterns
  - Unless you try the patterns one-by-one

Approach 2: Levenshtein Automaton

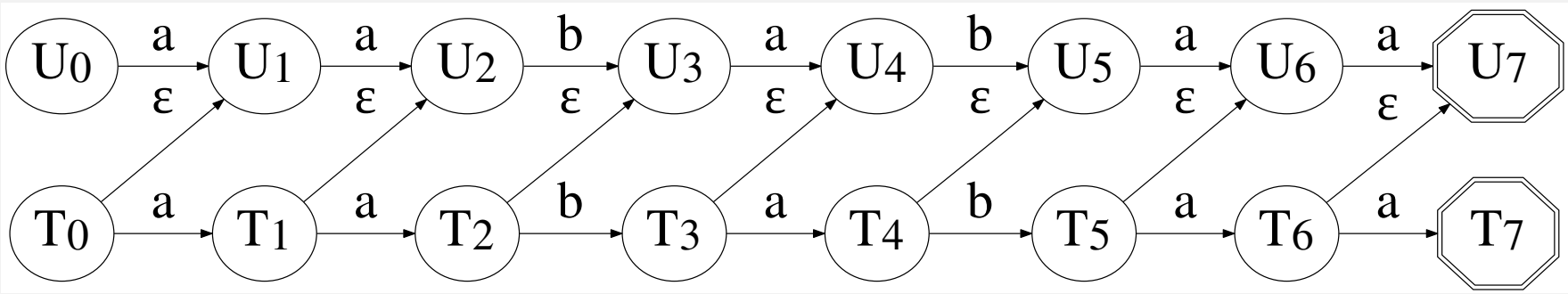
- Can be much faster, especially when p is small.
- Supports multiple patterns
- Enables applications such as spell-correction

Levenshtein Automaton



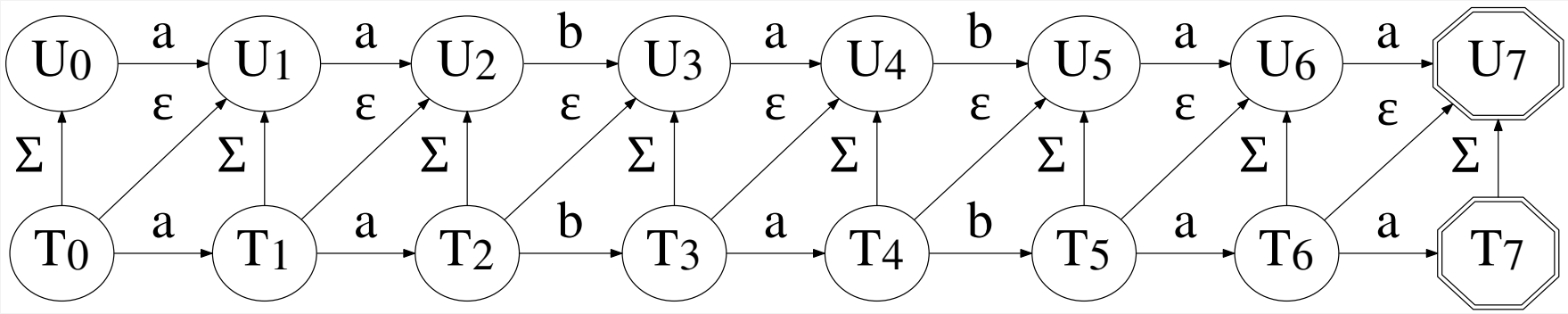
- No errors permitted.

Levenshtein Automaton



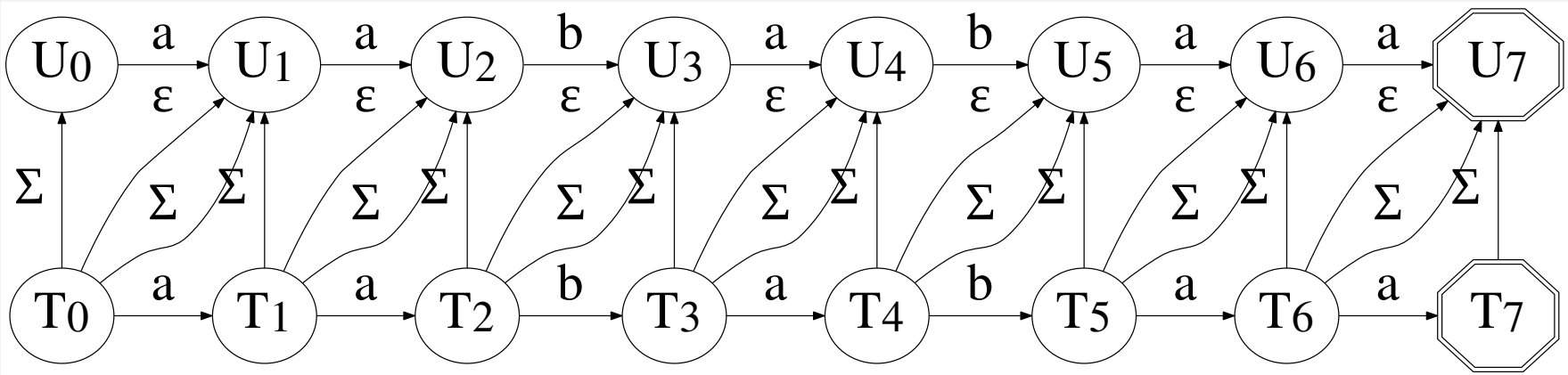
- Up to one missing character (deletion).

Levenshtein Automaton



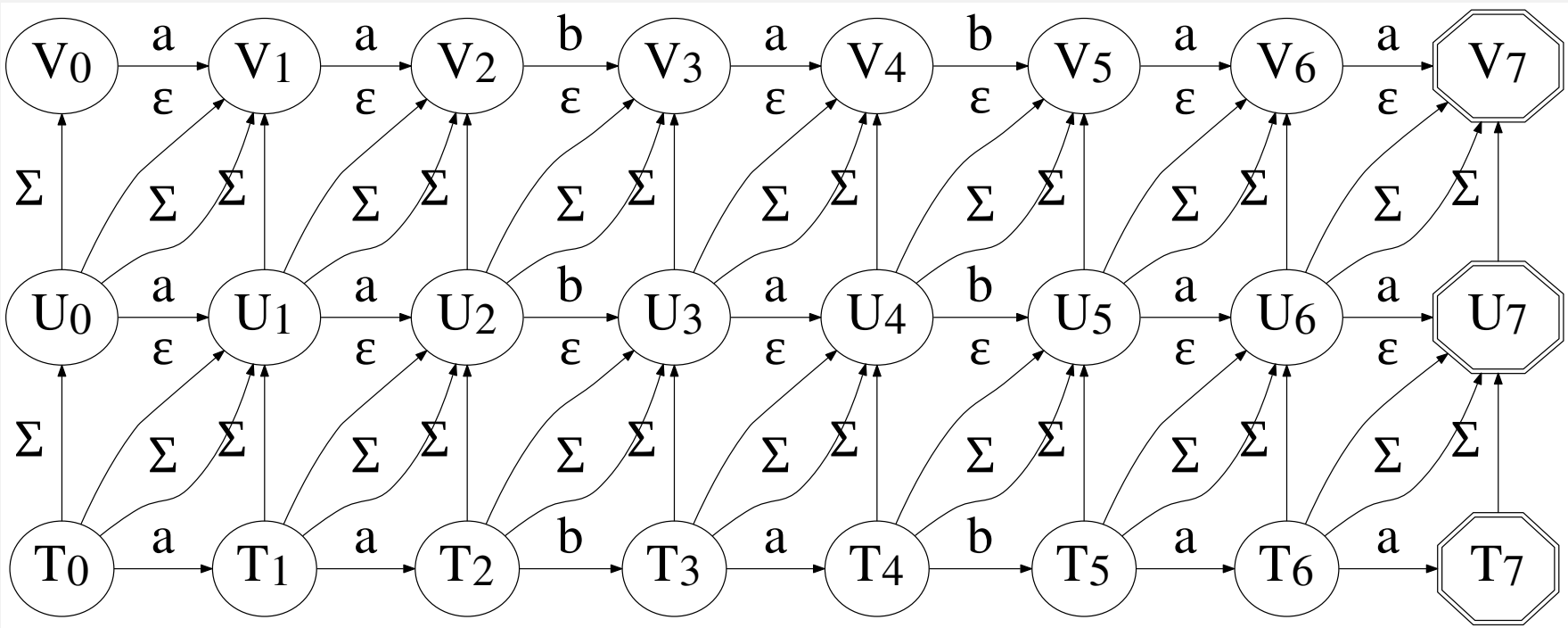
- Up to one deletion and one insertion.

Levenshtein Automaton



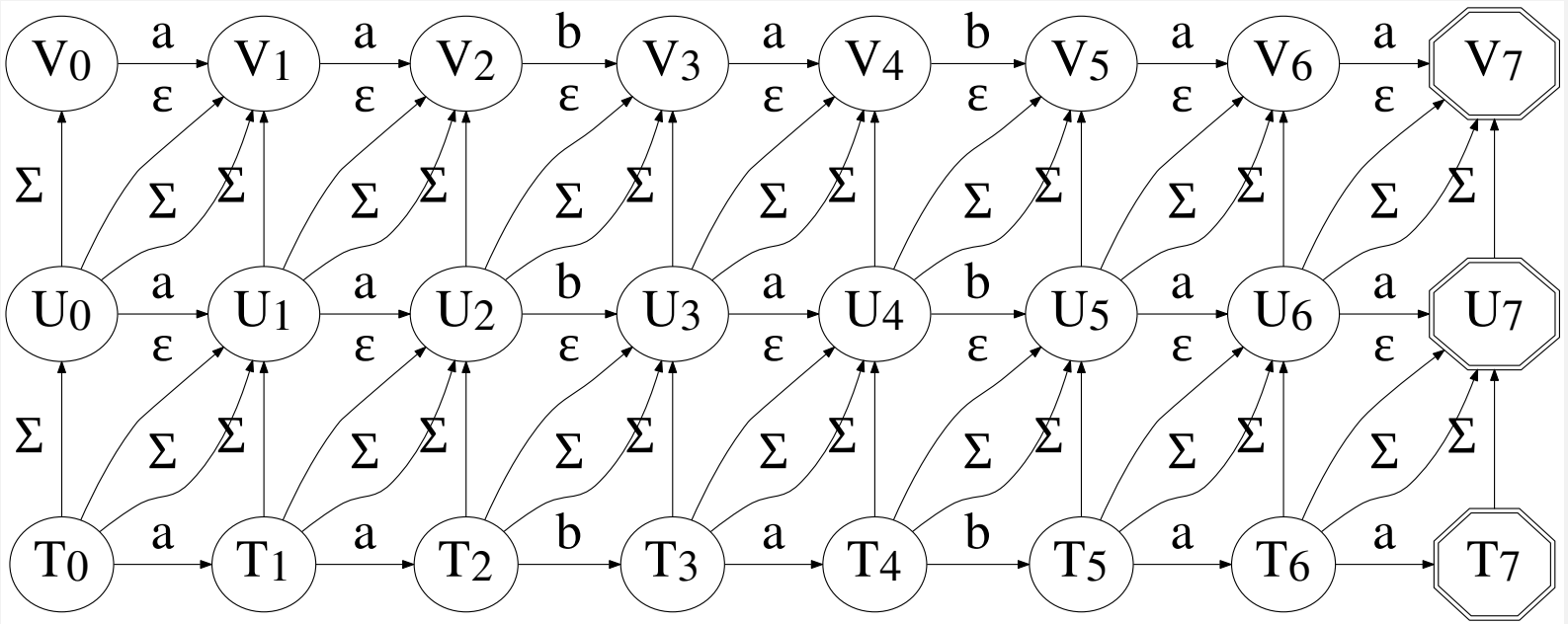
- Up to one deletion, or insertion, or substitution

Levenshtein Automaton



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

Levenshtein Automaton

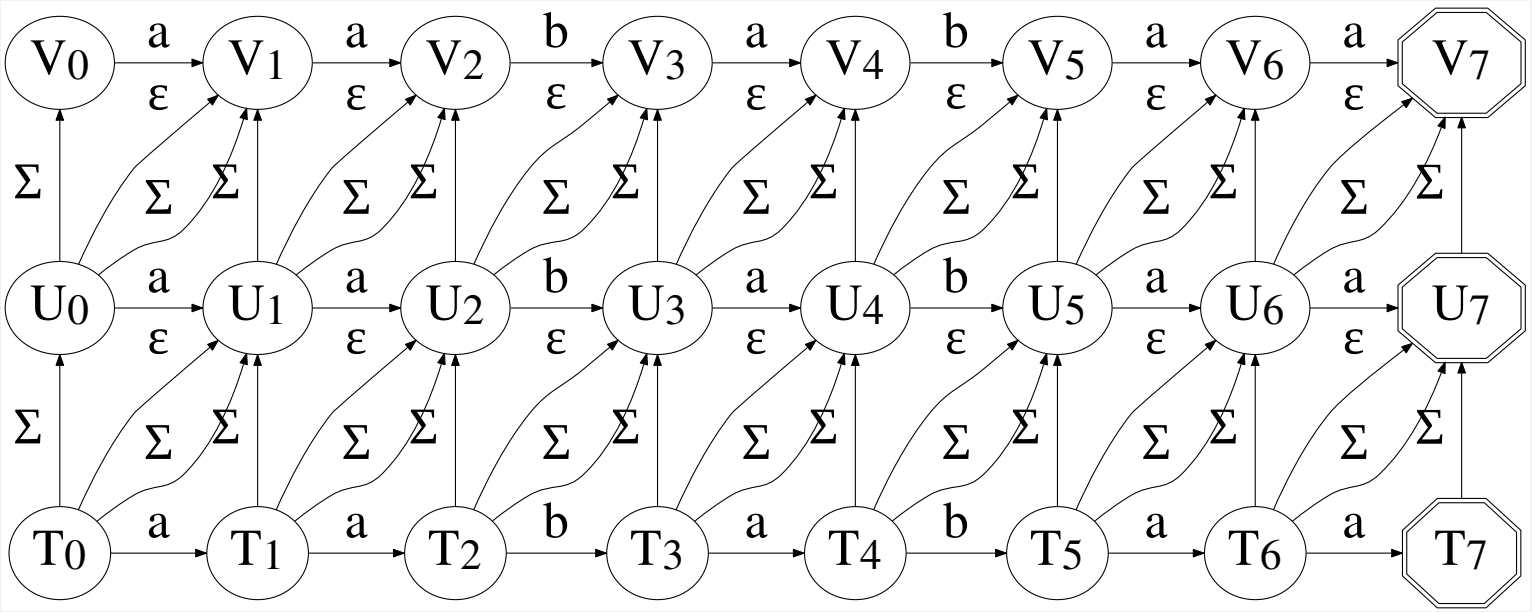


Compare with:

- Structure of cost matrix for edit-distance problem
- Finding least-cost paths from T_0 to T_7, U_7 or V_7

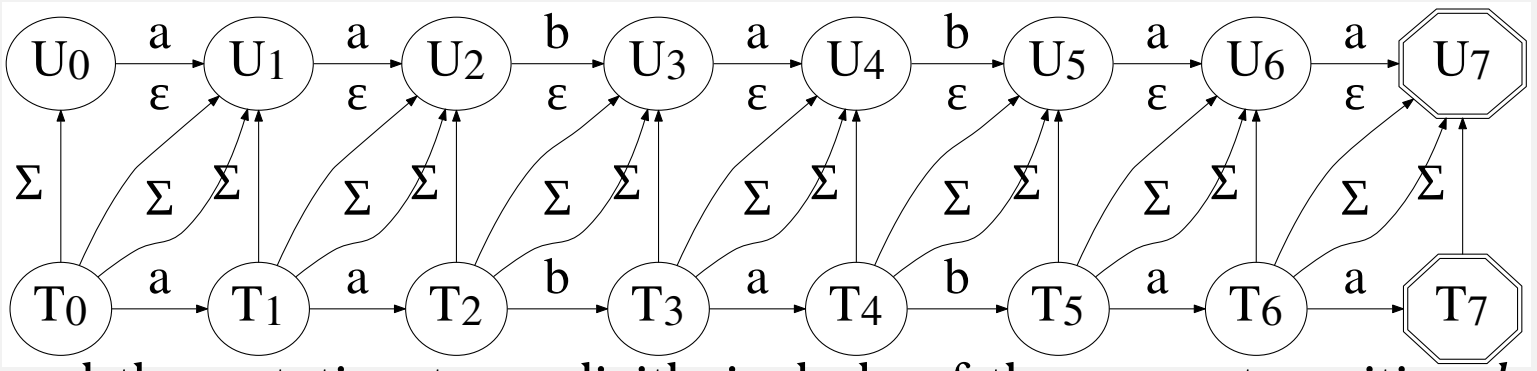
Illustrates the relationship between shortest path and edit-distance problem

Matching Using Levenshtein Automaton



- *Convert to DFA (subset construction)*
  - Potentially $O(n^k)$ states, where k is the max edit distance permitted
- *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.

Matching Using Levenshtein Automaton



We extend the notation to explicitly include of the current position k in T.

With this extension, our original equation for T becomes

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

Extending this to the case of U, we have

$$\begin{aligned} U^k \quad = \quad & T^{k-1} && \text{// move } \uparrow\text{, i.e., Insertion of } S[k] \\ | \quad & T^{k-1} \ll 1 && \text{// move } \nearrow \text{ with substitution} \\ | \quad & T^k \ll 1 && \text{// move } \nearrow \text{ with deletion} \\ | \quad & [(U^{k-1}\&\delta_{S[k]}) \ll 1]|1 && \text{// move } \rightarrow \end{aligned}$$

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

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

- 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 $T[i+m]$:

$$t_{i+1} = (t_i - 10^{n-1}T[i]) \cdot 10 + T[i+n]$$

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

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

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

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.

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

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

$$H = (B \ll 16) + A$$

Rolling Hash and Common Substring Problem

- To find a common substring of length l or more
  - Compute rolling hashes of P and T with window size l
    - Takes $O(n+m)$ time.
  - $O(nm)$ 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!
- 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]

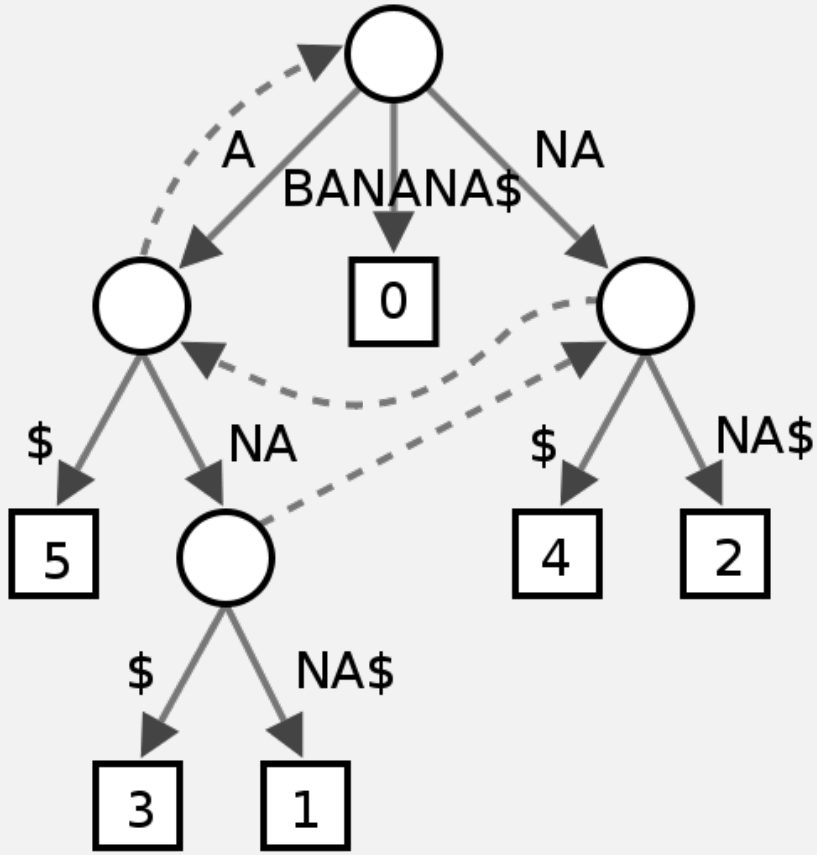
- A versatile data structure with wide applications in string search and computational biology
- *"Compressed" trie of all suffixes of a string appended with "$"*
  - Linear chains in the trie are compressed
    - Edges can now be substrings.
    - Each state has at least two children.
  - Leaves identify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

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, it is the text we preprocess, not the pattern.



Images from Wikipedia commons

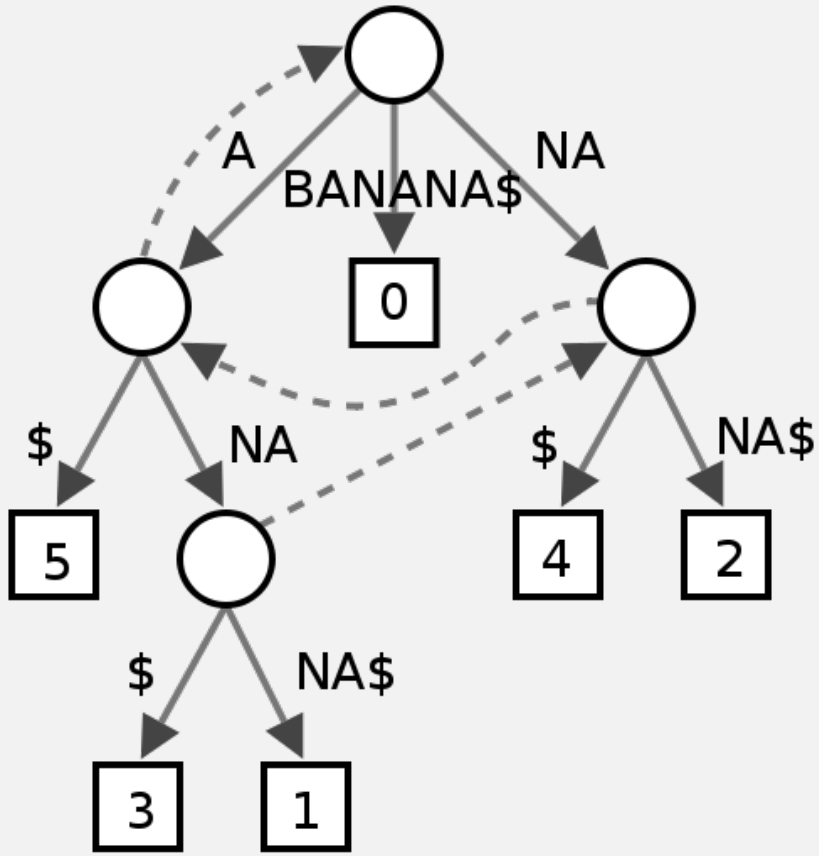
Finding Substrings and Suffixes

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

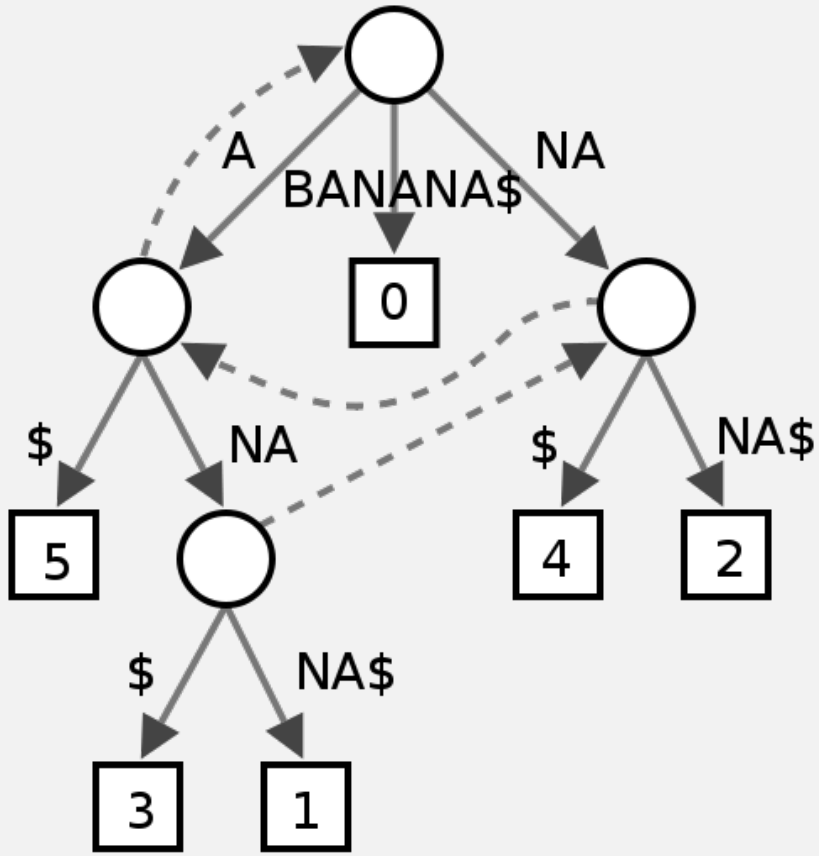


Counting # of Occurrences of p

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.

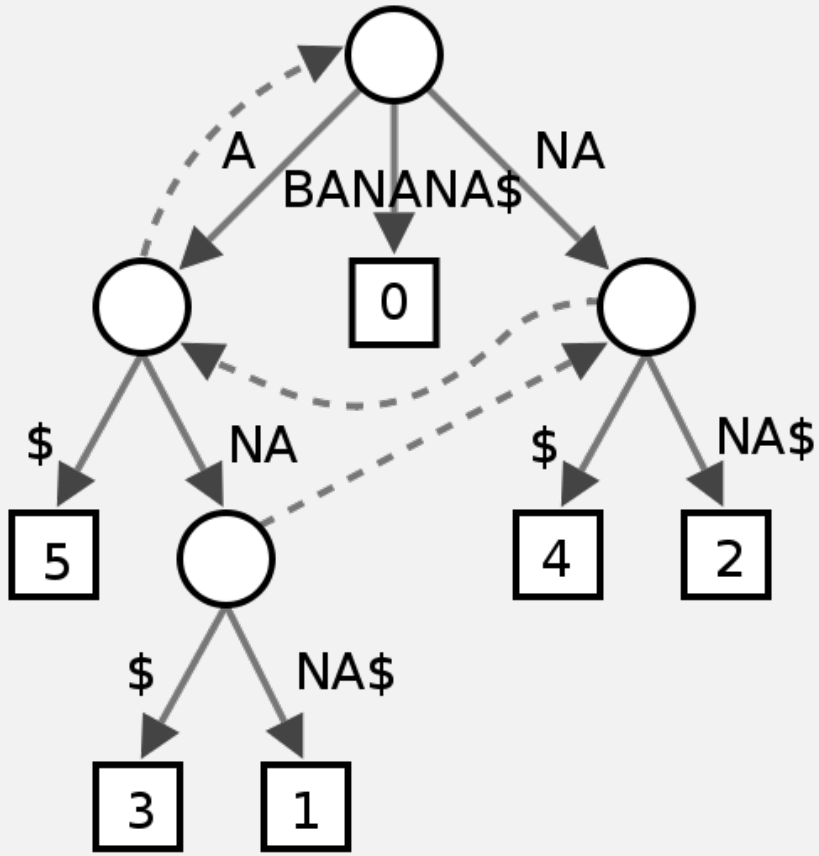


Self-LCS (Or, Longest Common Repeat)

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*.

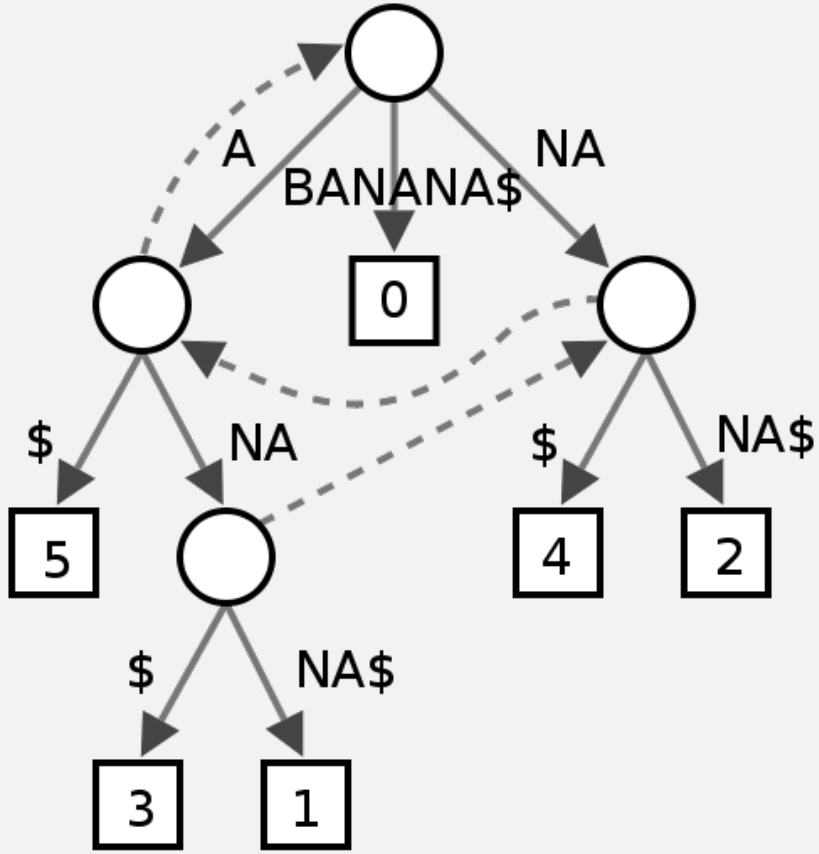


LC extension of i and j

Longest Common Extension

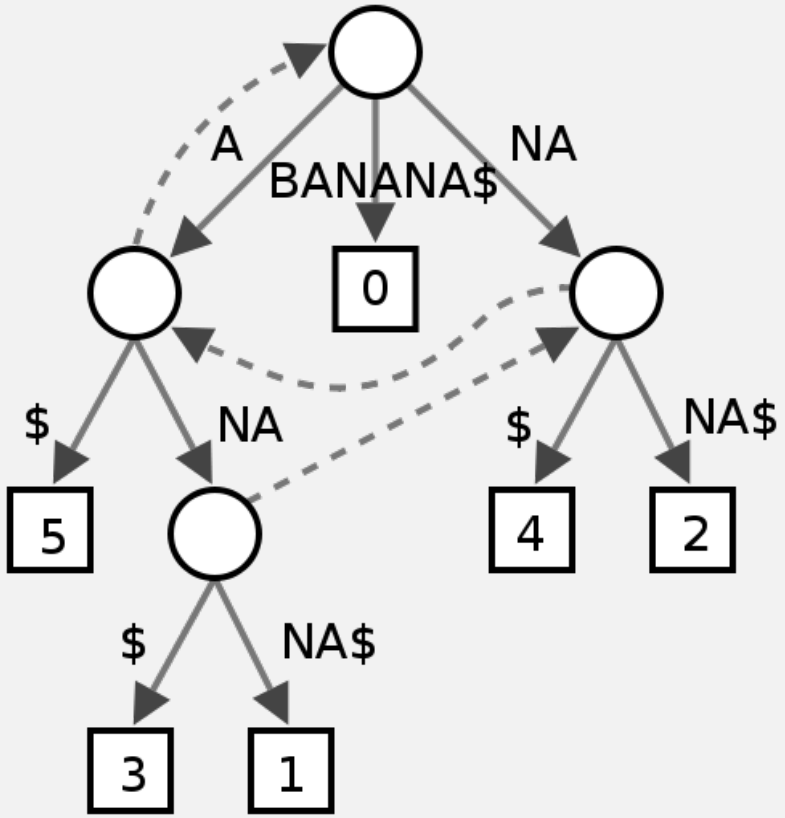
Longest common prefix of suffixes starting at i and j

- 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*



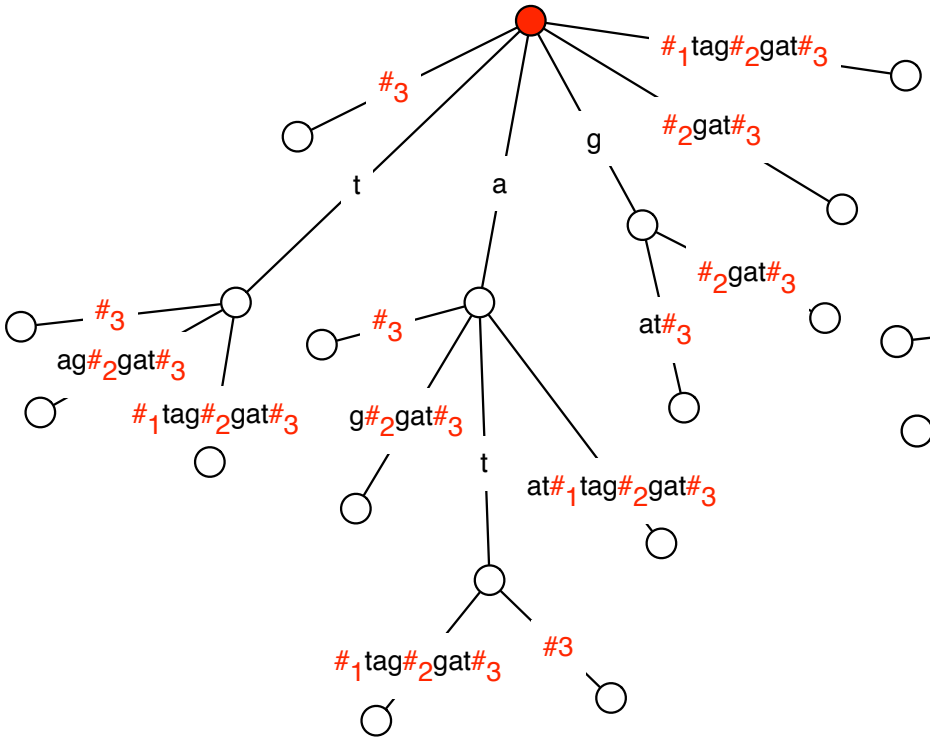
Generalized Suffix Trees

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

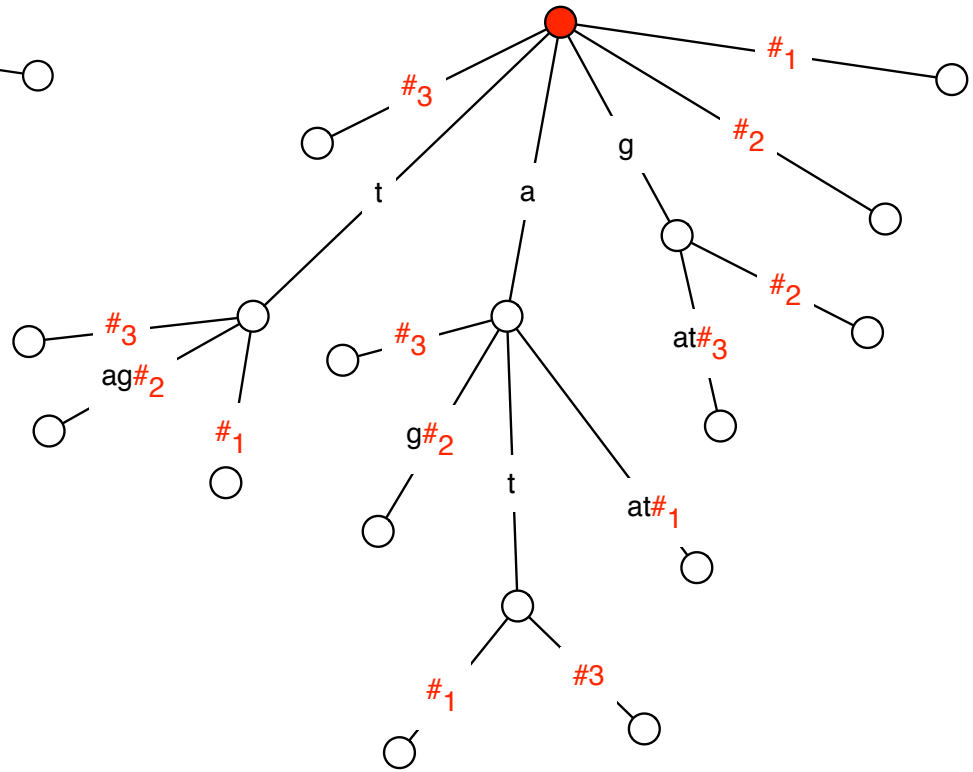
Example. att, tag, gat

Simple solution:

(1) build suffix tree for string aat#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
- 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