CS 240 – Data Structures and Data Management

Module 9: String Matching

T. Biedl   E. Schost   O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2021

References: Goodrich & Tamassia 23

version 2020–12–04 13:23

Outline

Outline

Pattern Matching Definition [1]

- Search for a string (pattern) in a large body of text
- $T[0..n-1]$ – The **text** (or **haystack**) being searched within
- $P[0..m-1]$ – The **pattern** (or **needle**) being searched for
- Strings over **alphabet** Σ
- Return the first i such that

$$P[j] = T[i+j] \quad \text{for} \quad 0 \leq j \leq m-1$$

- This is the first **occurrence** of P in T
- If P does not **occur** in T, return `FAIL`
- Applications:
  - Information Retrieval (text editors, search engines)
  - Bioinformatics
  - Data Mining

Pattern Matching Definition [2]

Example:

- $T =$ "`Where is he?`"
- $P_1 =$ "`he`"
- $P_2 =$ "`who`"

Definitions:

- **Substring** $T[i..j]$ $0 \leq i \leq j < n$: a string of length $j - i + 1$ which consists of characters $T[i], \ldots T[j]$ in order
- A **prefix** of T:
  a substring $T[0..i]$ of T for some $0 \leq i < n$
- A **suffix** of T:
  a substring $T[i..n-1]$ of T for some $0 \leq i \leq n - 1$

General Idea of Algorithms

Pattern matching algorithms consist of **guesses** and **checks**:

- A **guess** or **shift** is a position i such that P might start at $T[i]$. Valid guesses (initially) are $0 \leq i \leq n - m$.
- A **check** of a guess is a single position j with $0 \leq j < m$ where we compare $T[i + j]$ to $P[j]$. We must perform m checks of a single **correct** guess, but may make (many) fewer checks of an **incorrect** guess.

We will diagram a single run of any pattern matching algorithm by a matrix of checks, where each row represents a single guess.

Brute-force Algorithm

Idea: Check every possible guess.

Bruteforce::patternMatching$(T[0..n-1], P[0..m-1])$
T: String of length n (text), P: String of length m (pattern)

```
for i ← 0 to n − m do
    if strcmp(T[i..i+m−1], P) = 0
        return "found at guess i"
return FAIL
```

Note: *strcmp* takes $\Theta(m)$ time.

strcmp$(T[i..i{+}m{-}1], P[0..m-1])$

```
for j ← 0 to m − 1 do
    if T[i + j] is before P[j] in Σ then return -1
    if T[i + j] is after P[j] in Σ then return 1
return 0
```

Brute-Force Example

- Example: $T =$ `abbbababbab`, $P =$ `abba`

a	b	b	b	a	b	a	b	b	a	b
a	b	b	**a**							
	a									
		a								
			a							
				a	b	**b**				
					a					
						a	b	b	a	

- What is the worst possible input?
  $P = a^{m-1}b,\ T = a^n$
- Worst case performance $\Theta((n - m + 1)m)$
- This is $\Theta(mn)$ e.g. if $m = n/2$.

How to improve?

More sophisticated algorithms

- Do extra **preprocessing** on the pattern P
  - **Karp-Rabin**
  - **Boyer-Moore**
  - Deterministic finite automata (**DFA**), **KMP**
  - We **eliminate guesses** based on completed matches and mismatches.
- Do extra **preprocessing** on the text T
  - **Suffix-trees**
  - We **create a data structure** to find matches easily.

Outline

1. String Matching
   - Introduction
   - Karp-Rabin Algorithm
   - String Matching with Finite Automata
   - Knuth-Morris-Pratt algorithm
   - Boyer-Moore Algorithm
   - Suffix Trees
   - Conclusion

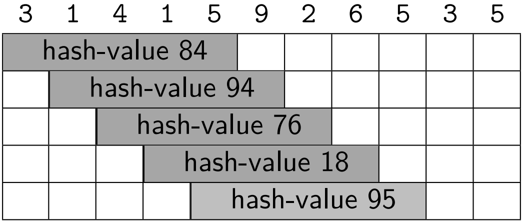
Karp-Rabin Fingerprint Algorithm – Idea

Idea: use hashing to eliminate guesses

- Compute hash function for each guess, compare with pattern hash
- If values are unequal, then the guess cannot be an occurrence
- Example: $P = 5\ 9\ 2\ 6\ 5$, $\quad T = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5$
  - Use standard hash-function: flattening + modular (radix $R = 10$):

$$h(x_0 \ldots x_4) = \big(x_0 x_1 x_2 x_3 x_4\big)_{10} \bmod 97$$

  - $h(P) = 59265 \bmod 97 = 95$.



Karp-Rabin Fingerprint Algorithm – First Attempt

```
Karp-Rabin-Simple::patternMatching(T, P)
1.   h_P ← h(P[0..m−1]])
2.   for i ← 0 to n − m
3.       h_T ← h(T[i..i+m−1])
4.       if h_T = h_P
5.           if strcmp(T[i..i+m−1], P) = 0
6.               return "found at guess i"
7.   return FAIL
```

- Never misses a match: $h(T[i..i{+}m{-}1]) \neq h(P) \Rightarrow$ guess i is not P
- $h(T[i..i{+}m{-}1])$ depends on m characters, so naive computation takes $\Theta(m)$ time per guess
- Running time is $\Theta(mn)$ if P not in T (how can we improve this?)

Karp-Rabin Fingerprint Algorithm – Fast Rehash

The initial hashes are called **fingerprints**.

Crucial insight: We can update these fingerprints in constant time.

- Use previous hash to compute next hash
- $O(1)$ time per hash, except first one

Example:

- Pre-compute: $10000 \bmod 97 = 9$
- Previous hash: $\mathbf{4}1592 \bmod 97 = 76$
- Next hash: $1592\mathbf{6} \bmod 97 = ??$

Observe: $15926 = (41592 - 4 \cdot 10\,000) \cdot 10 + 6$

$$15926 \bmod 97 = \Big((\underbrace{41592 \bmod 97}_{76 \text{ (previous hash)}} - 4 \cdot \underbrace{10000 \bmod 97}_{9 \text{ (pre-computed)}} \Big) \cdot 10 + 6) \bmod 97$$

$$= \Big((76 - 4 \cdot 9) \cdot 10 + 6 \Big) \bmod 97 = 18$$

Karp-Rabin Fingerprint Algorithm – Conclusion

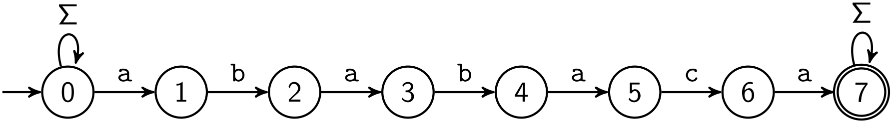
```
Karp-Rabin-RollingHash::patternMatching(T, P)
    h_P ← h(P[0..m−1)])
    p ← suitable prime number
    s ← 10^(m−1) mod p
    h_T ← h(T[0..m−1)])
    for i ← 0 to n − m
        if i > 0 // compute hash-value for next guess
            h_T ← ((h_T − T[i] · s) · 10 + T[i+m]) mod p
        if h_T = h_P
            if strcmp(T[i..i+m−1], P) = 0
                return "found at guess i"
    return "FAIL"
```

- Choose "table size" p at **random** to be huge prime
- Expected running time is $O(m+n)$
- $\Theta(mn)$ worst-case, but this is (unbelievably) unlikely

Outline

String Matching with Finite Automata

Example: Automaton for the pattern $P =$ `ababaca`



(
You should be familiar with:

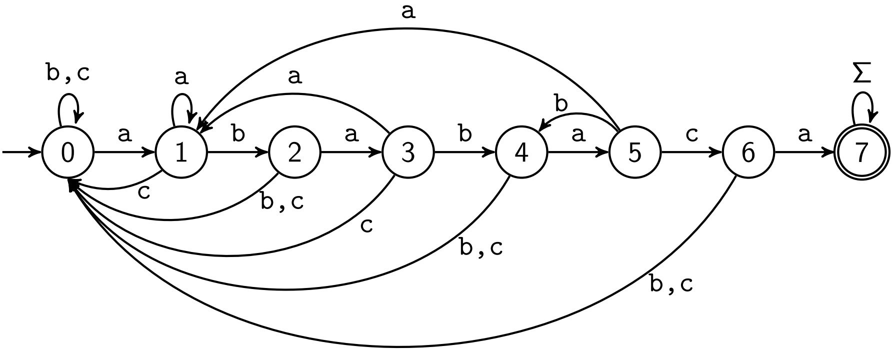
- finite automaton, DFA, NFA, converting NFA to DFA
- transition function δ, states Q, accepting states F

)

- The above finite automation is an **NFA**
- State q expresses "we have seen $P[0..q-1]$"
  - NFA accepts T if and only if T contains `ababaca`
  - But evaluating NFAs is very slow.

String matching with DFA

Can show: There exists an equivalent small DFA.

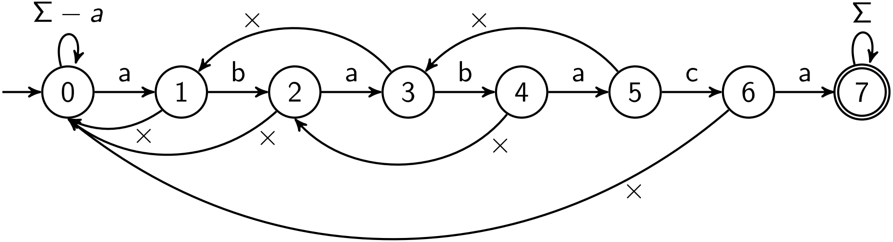


- Easy to test whether P is in T.
- But how do we find the arcs?
- We will not give the details of this since there is an even better automaton.

Outline

1. String Matching
   - Introduction
   - Karp-Rabin Algorithm
   - String Matching with Finite Automata
   - Knuth-Morris-Pratt algorithm
   - Boyer-Moore Algorithm
   - Suffix Trees
   - Conclusion

Knuth-Morris-Pratt Motivation



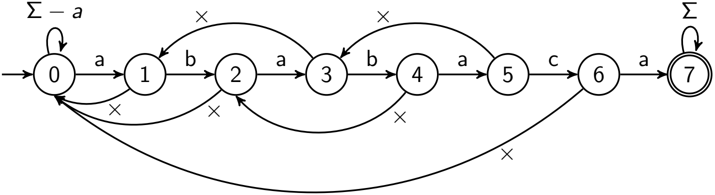
- Use a new type of transition × (*"failure"*):
  - Use this transition only if no other fits.
  - Does **not** consume a character.
  - With these rules, computations of the automaton are deterministic. (But it is formally not a valid DFA.)
- Can store **failure-function** in an array $F[0..m-1]$
  - The failure arc from state j leads to $F[j-1]$
- Given the failure-array, we can easily test whether P is in T: Automaton accepts T if and only if T contains `ababaca`

Knuth-Morris-Pratt Algorithm

```
KMP::patternMatching(T, P)
    F ← failureArray(P)
    i ← 0 // current character of T to parse
    j ← 0 // current state that we are in
    while i < n do
        if P[j] = T[i]
            if j = m − 1
                return "found at guess i − m + 1"
            else
                i ← i + 1
                j ← j + 1
        else // i. e. P[j] ≠ T[i]
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return FAIL
```

String matching with KMP – Example

Example: $T =$ `abababababaca`, $P =$ `ababaca`

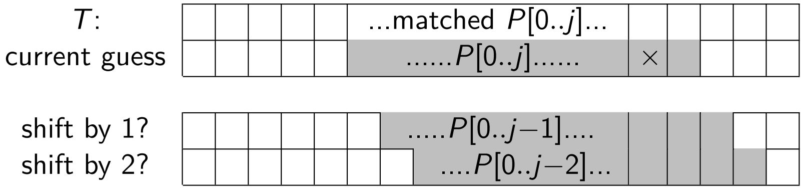


T:	a	b	a	b	a	b	b	c	a	b	a	b	a	c	a
	a	b	a	b	a	×									
			(a)	(b)	(a)	b	×								
					(a)	(b)	×								
							×								
								×							
									a	b	a	b	a	c	a
q:	1	2	3	4	5	3, 4	2, 0	0	1	2	3	4	5	6	7

(after reading this character)

String matching with KMP – Failure-function

Assume we reach state $j+1$ and now have mismatch.



- Can eliminate “shift by 1” if $P[1..j] \neq P[0..j-1]$.
- Can eliminate “shift by 2” if $P[1..j]$ does not end with $P[0..j-2]$.
- Generally eliminate guess if that prefix of P is not a suffix of $P[1..j]$.
- So want longest prefix $P[0..\ell-1]$ that is a suffix of $P[1..j]$.
- The ℓ characters of this prefix are matched, so go to state ℓ.

$$\begin{aligned} F[j] &= \text{head of failure-arc from state } j+1 \\ &= \text{length of the longest prefix of } P \text{ that is a suffix of } P[1..j]. \end{aligned}$$

KMP Failure Array – Example

$F[j]$ is the length of the longest prefix of P that is a suffix of $P[1..j]$.

Consider P = ababaca

j	$P[1..j]$	Prefixes of P	longest	$F[j]$
0	Λ	Λ, a, ab, aba, abab, ababa, ...	Λ	0
1	b	Λ, a, ab, aba, abab, ababa, ...	Λ	0
2	ba	Λ, a, ab, aba, abab, ababa, ...	a	1
3	bab	Λ, a, ab, aba, abab, ababa, ...	ab	2
4	baba	Λ, a, ab, aba, abab, ababa, ...	aba	3
5	babac	Λ, a, ab, aba, abab, ababa, ...	Λ	0
6	babaca	Λ, a, ab, aba, abab, ababa, ...	a	1

This can clearly be computed in $O(m^3)$ time, but we can do better!

Computing the Failure Array

```
KMP::failureArray(P)
P: String of length m (pattern)
    F[0] ← 0
    j ← 1
    ℓ ← 0
    while j < m do
        if P[j] = P[ℓ]
            ℓ ← ℓ + 1
            F[j] ← ℓ
            j ← j + 1
        else if ℓ > 0
            ℓ ← F[ℓ − 1]
        else
            F[j] ← 0
            j ← j + 1
```

Correctness-idea: $F[j]$ is defined via pattern matching of P in $P[1..j]$.
So KMP uses itself! Already-built parts of $F[\cdot]$ are used to expand it.

KMP – Runtime

failureArray

- Consider how $2j - \ell$ changes in each iteration of the while loop
  - j and ℓ both increase by $1 \Rightarrow 2j - \ell$ increases –OR–
  - ℓ decreases ($F[\ell - 1] < \ell$) $\Rightarrow 2j - \ell$ increases –OR–
  - j increases $\Rightarrow 2j - \ell$ increases
- Initially $2j - \ell \geq 0$, at the end $2j - \ell \leq 2m$
- So no more than $2m$ iterations of the while loop.
- Running time: $\Theta(m)$

KMP main function

- failureArray can be computed in $\Theta(m)$ time
- Same analysis gives at most $2n$ iterations of the while loop since $2i - j \leq 2n$.
- Running time KMP altogether: $\Theta(n + m)$

Outline

Boyer-Moore Algorithm

Brute-force search with three changes:

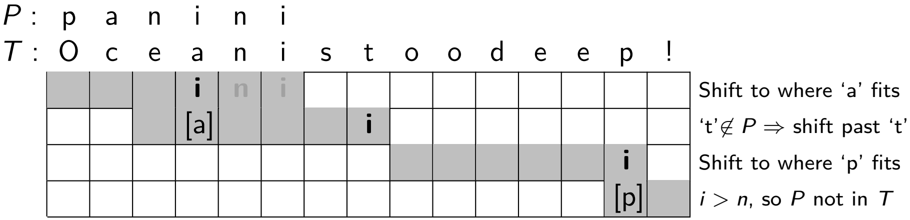
- **Reverse-order searching**: Compare P with a guess moving **backwards**
- **Bad character jumps**: When a mismatch occurs, then eliminate guesses where P does not agree with this char of T
- **Good suffix jumps**: When a mismatch occurs, then use recently seen suffix of P to eliminate guesses.
- This gives two possible shifts (locations of next guess to try). Use the one that moves forward more.
- In practice large parts of T will not be looked at.

Boyer-Moore Algorithm

```
Boyer-Moore::patternMatching(T,P)
    L ← last occurrence array computed from P
    S ← good suffix array computed from P
    i ← m − 1,      j ← m − 1
    while i < n and j ≥ 0 do
        if T[i] = P[j]
            i ← i − 1
            j ← j − 1
        else
            i ← i + m − 1 − min(L[T[i]], S[j])
            j ← m − 1
    if j = −1 return i + 1
    else return FAIL
```

L and S will be explained below.

Bad character heuristic



- Build the **last-occurrence array** L mapping Σ to integers
- $L(c)$ is the largest index i such that $P[i] = c$ (or -1 if no such index exists)

c	p	a	n	i	all others
$L(c)$	0	1	4	5	-1

- Can build this in time $O(m + |\Sigma|)$ with simple for-loop
- Guesses are updated by aligning $T[i]$ with $P[L(T[i])]$

Good suffix heuristic

$P =$ onobobo

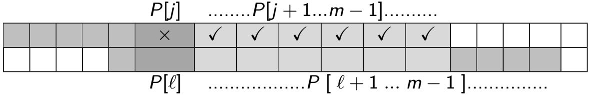
o	n	o	o	o	b	o	o	o	i	b	b	o	u	n	d	a	r	y
			b	**o**	**b**	**o**												
Do smallest shift so that **obo** fits in the new guess.																		
				(o)	(b)	(o)	**b**	**o**										
Do smallest shift so that **o** fits in the new guess.																		
								(o)										
But this *has* to fail at **b**, so could shift farther right away																		
							(not b)	(o)		**o**	**b**	**o**						
Again: the shift that matches **bo** would fail at **o**, so shift farther.																		
										(o)	(b)	(o)						
												(o)						
Note that we could not match all of **bo** but match as much as we can.																		

Good suffix array

- For $0 \le j < m$, if search failed at $T[i] \neq P[j]$
  - Had $T[i{+}1..k{+}m-1] = P[j{+}1..m{-}1]$ and $T[i] \neq P[j]$
  - Can precompute *good suffix array* of where to shift

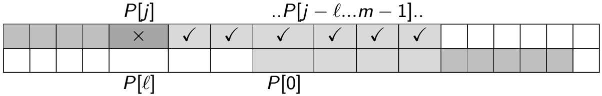
$S[j]$ is the maximum ℓ such that

- ★ $P[j{+}1 \ldots m{-}1]$ is a prefix of $P[\ell{+}1 \ldots m{-}1]$ and $P[j] \neq P[\ell]$



–OR–

- ★ $P[j-\ell \ldots m-1]$ is a prefix of P and $\ell < 0$.



–OR–

- ★ $\ell = j - m$ if neither of the above is possible

  - Then can update guess by aligning $T[i]$ with $P[S[j]]$

- $S[\cdot]$ computable (similar to KMP failure function) in $\Theta(m)$ time.

Good suffix array example

Example: bon**obobo**

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$	-6	-5	-4	-3	2	-1	2	6

Outline

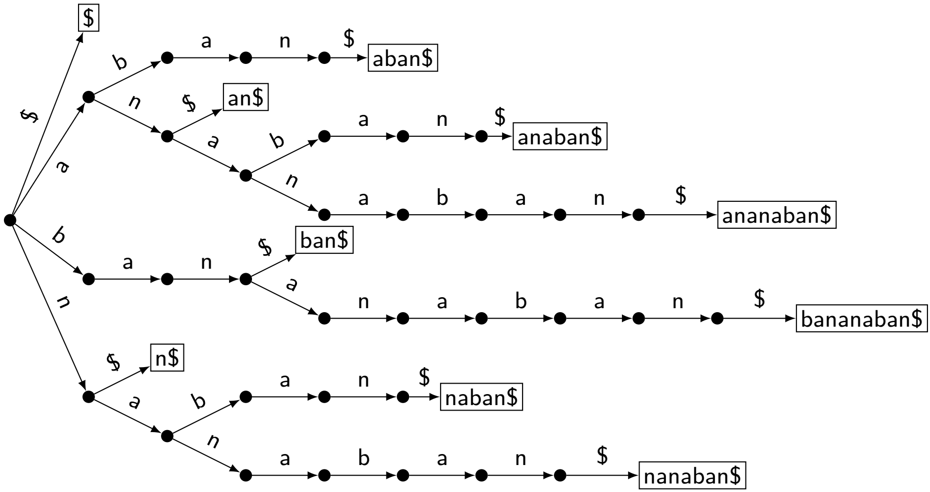
Tries of Suffixes and Suffix Trees

- What if we want to search for **many patterns** P within the same **fixed text** T?
- **Idea:** Preprocess the text T rather than the pattern P
- **Observation:** P is a substring of T if and only if P is a prefix of some suffix of T.
- So want to store all suffixes of T in a trie.
- To save space:
  - Use a compressed trie.
  - Store suffixes implicitly via indices into T.
- This is called a **suffix tree**.

Trie of suffixes: Example

$T =$ bananaban has suffixes

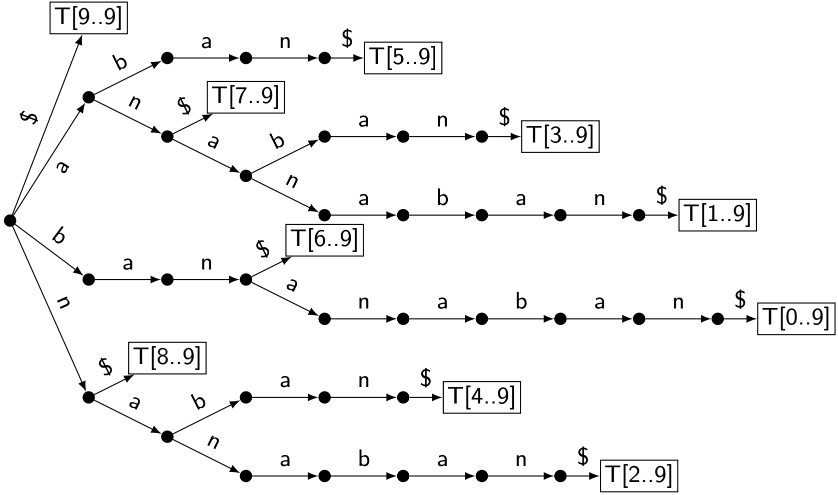
{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, Λ}



Tries of suffixes

Store suffixes via indices: $T =$

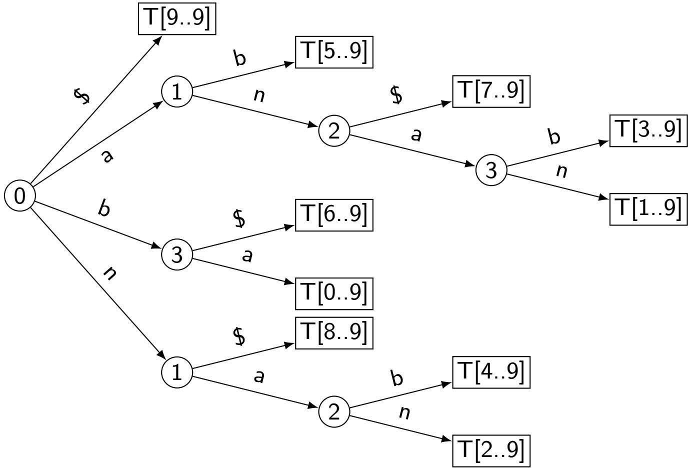
0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$



Suffix tree

Suffix tree: Compressed trie of suffixes

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$



Building Suffix Trees

- Text T has n characters and $n+1$ suffixes
- We can build the suffix tree by inserting each suffix of T into a compressed trie.
  This takes time $\Theta(n^2|\Sigma|)$.
- There *is* a way to build a suffix tree of T in $\Theta(n|\Sigma|)$ time.
  This is quite complicated and beyond the scope of the course.
- For pattern matching, suffix trees additionally need:
  - Every interior node w stores a reference $w.leaf$ to the leaf in its subtree with the longest suffix.
  - This can be found in $O(n)$ time by traversing the suffix tree.

Suffix Trees: String Matching

- In the *uncompressed* trie, searching for P would be easy.
- In the *compressed* suffix tree, search as in a compressed trie. Stop the search once P has run out of characters.

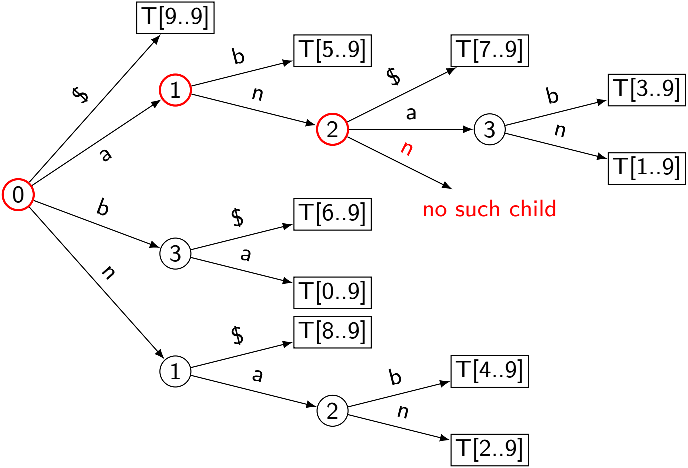
```
SuffixTree::patternMatching(T[0..n−1], P[0..m−1], 𝒯)
T: text, P: pattern, 𝒯: Suffix tree of T
    v ← 𝒯.root
    repeat
        if v.index ≥ m or v has no child corresponding to P[v.index]
            return FAIL
        w ← child of v corresponding to P[v.index]
        if w is leaf or w.index ≥ m // have gone beyond pattern P
            ℓ ← w.leaf
            i ← ℓ.start
            if (i+m ≤ n and strcmp(T[i..i+m−1], P) = 0)
                return "found at guess i"
            else return FAIL
        v ← w
```

Pattern Matching in Suffix Tree: Example 1

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$

$P =$ ann

FAIL

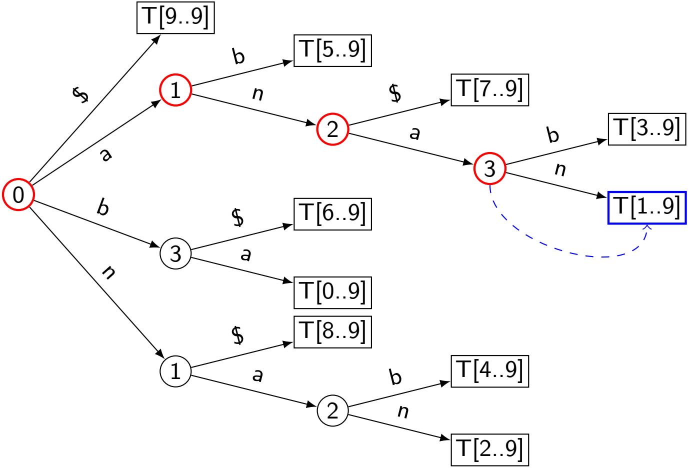


Pattern Matching in Suffix Tree: Example 2

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$

$P =$ ana

"found at guess 1"



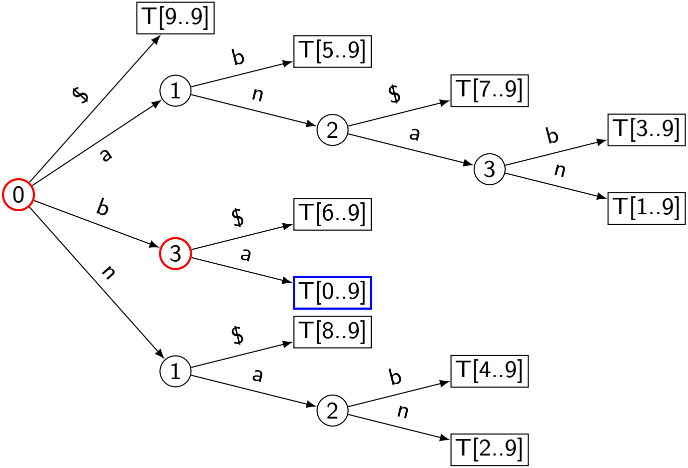
Pattern Matching in Suffix Tree: Example 3

$T =$

0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$

$P =$ briar

FAIL

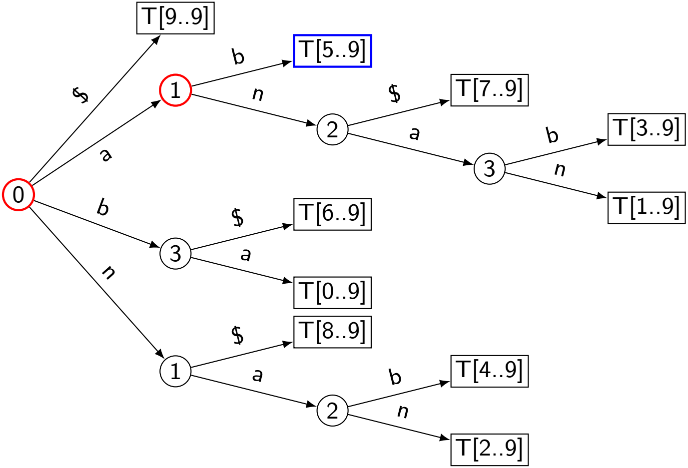


Pattern Matching in Suffix Tree: Example 4

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$

$P =$ abando

FAIL



Outline

String Matching Conclusion

	Brute-Force	Karp-Rabin	DFA	Knuth-Morris-Pratt	Boyer-Moore	Suffix Tree	Suffix Array [1]
Preproc.	—	$O(m)$	$O(m\|\Sigma\|)$	$O(m)$	$O(m+\|\Sigma\|)$	$O(n^2\|\Sigma\|)$ $[O(n)\|\Sigma\|]$	$O(n \log n)$ $[O(n)]$
Search time	$O(nm)$	$O(n+m)$ expected	$O(n)$	$O(n)$	$O(n)$ or better	$O(m)$	$O(m \log n)$
Extra space	—	$O(1)$	$O(m\|\Sigma\|)$	$O(m)$	$O(m+\|\Sigma\|)$	$O(n)$	$O(n)$

- Our algorithms stopped once they have found one occurrence.
- Most of them can be adapted to find *all* occurrences within the same worst-case run-time.

[1] studied only in the enriched section