Module 9: String Matching

CS 240 - Data Structures and Data Management

Module 0: Introduction

Collin Roberts and Arne Storjohann

Based on lecture notes by many previous cs240 instructors

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

Fall 2022

References:

version 2022-11-14 12:24

Pattern Matching

- 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

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

- 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

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** 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.

```
BruteforcePM(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
    match ← true
    j ← 0
    while j < m and match do
        if T[i + j] = P[j] then
            j ← j + 1
        else
            match ← false
    if match then
        return i
return FAIL
```

Example

- Example: $T = \texttt{abbbababbab}$, $P = \texttt{abba}$

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 $\Theta((n - m + 1)m)$
- $m \leq n/2 \Rightarrow \Theta(mn)$

Pattern Matching

More sophisticated algorithms

- **KMP** and **Boyer-Moore**
- Do extra preprocessing on the pattern P
- We eliminate guesses based on completed matches and mismatches.

KMP Algorithm

- Knuth-Morris-Pratt algorithm (1977)
- Compares the pattern to the text in left-to-right
- Shifts the pattern more intelligently than the brute-force algorithm
- When a mismatch occurs, what is the most we can shift the pattern (reusing knowledge from previous matches)?

$T =$	a	b	c	d	c	a	b	c	?	?	?
	a	b	c	d	c	a	b	a			
						a	b	c	d	c	a

- KMP Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$

KMP Failure Array

- Preprocess the pattern to find matches of prefixes of the pattern with the pattern itself
- The failure array F of size m: $F[j]$ is defined as the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- $F[0] = 0$
- If a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F[j-1]$
- Consider $P =$ `abacaba`

j	$P[1..j]$	P	$F[j]$
0	—	`abacaba`	0
1	`b`	`abacaba`	0
2	`ba`	`abacaba`	1
3	`bac`	`abacaba`	0
4	`baca`	`abacaba`	1
5	`bacab`	`abacaba`	2
6	`bacaba`	`abacaba`	3

KMP Algorithm

```
KMP(T, P)
T: String of length n (text), P: String of length m (pattern)
    F ← failureArray(P)
    i ← 0
    j ← 0
    while i < n do
        if T[i] = P[j] then
            if j = m − 1 then
                return i − j //match
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0 then
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 // no match
```

KMP: Example

$P =$ abacaba

$T =$ abaxyabacabbaababacaba

0	1	2	3	4	5	6	7	8	9	10	11
a	b	a	x	y	a	b	a	c	a	b	b
a	b	a	c								
		(a)	b								
			a								
				a							
					a	b	a	c	a	b	a
									(a)	(b)	a

Exercise: continue with $T =$ abaxyabacabbaababacaba

Computing the Failure Array

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

KMP: Analysis

failureArray

- At each iteration of the while loop, either
  1. i increases by one, or
  2. the guess index $i - j$ increases by at least one ($F[j-1] < j$)
- There are no more than $2m$ iterations of the while loop
- Running time: $\Theta(m)$

KMP: Analysis

failureArray

- At each iteration of the while loop, either
  1. i increases by one, or
  2. the guess index $i - j$ increases by at least one ($F[j-1] < j$)
- There are no more than $2m$ iterations of the while loop
- Running time: $\Theta(m)$

KMP

- failureArray can be computed in $\Theta(m)$ time
- At each iteration of the while loop, either
  1. i increases by one, or
  2. the guess index $i - j$ increases by at least one ($F[j-1] < j$)
- There are no more than $2n$ iterations of the while loop
- Running time: $\Theta(n)$

KMP: Another Example

- $T =$ abacaabaccabacabaabb
- $P =$ abacab

Boyer-Moore Algorithm

Based on three key ideas:

- Reverse-order searching: Compare P with a subsequence of T moving backwards
- Bad character jumps: When a mismatch occurs at $T[i] = c$
  - If P contains c, we can shift P to align the last occurrence of c in P with $T[i]$
  - Otherwise, we can shift P to align $P[0]$ with $T[i+1]$
- Good suffix jumps: If we have already matched a suffix of P, then get a mismatch, we can shift P forward to align with the previous occurence of that suffix (with a mismatch from the actual suffix). Similar to failure array in KMP.
- When a mismatch occurs, Boyer-Moore chooses whichever of bad character or good suffix shifts the pattern further to the right.
- Can skip large parts of T

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

			o								

P = m o o r e

T = b o y e r m o o r e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				

P	=	m	o	o	r	e				
T	=	b	o	y	e	r	m	o	o	r	e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				
													o

P	=	m	o	o	r	e				
T	=	b	o	y	e	r	m	o	o	r	e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				
												d	o

P	=	m	o	o	r	e				
T	=	b	o	y	e	r	m	o	o	r	e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				
											l	d	o

P	=	m	o	o	r	e				
T	=	b	o	y	e	r	m	o	o	r	e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				
										a	l	d	o

P	=	m	o	o	r	e				
T	=	b	o	y	e	r	m	o	o	r	e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				
										a	l	d	o

6 comparisons (checks)

P	=	m	o	o	r	e				
T	=	b	o	y	e	r	m	o	o	r	e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				
										a	l	d	o

6 comparisons (checks)

P	=	m	o	o	r	e					
T	=	b	o	y	e	r	m	o	o	r	e
						e					

Bad character examples

P =	a	l	d	o								
T =	w	h	e	r	e	i	s	w	a	l	d	o
				o								
								o				
									a	l	d	o

6 comparisons (checks)

P =	m	o	o	r	e					
T =	b	o	y	e	r	m	o	o	r	e
					e					
					[r]	e				

Bad character examples

P =	a	l	d	o								
T =	w	h	e	r	e	i	s	w	a	l	d	o
				o								
								o				
									a	l	d	o

6 comparisons (checks)

P =	m	o	o	r	e					
T =	b	o	y	e	r	m	o	o	r	e
					e					
					[r]	e				
						[m]				e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				
										a	l	d	o

6 comparisons (checks)

P	=	m	o	o	r	e					
T	=	b	o	y	e	r	m	o	o	r	e
						e					
						[r]	e				
							[m]			r	e

Bad character examples

P	=	a	l	d	o								
T	=	w	h	e	r	e	i	s	w	a	l	d	o
					o								
									o				
										a	l	d	o

6 comparisons (checks)

P	=	m	o	o	r	e					
T	=	b	o	y	e	r	m	o	o	r	e
						e					
						[r]	e				
							[m]	o	o	r	e

7 comparisons (checks)

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							×	(e)	(l)	(l)	(s)							

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							×	(e)	(l)	(l)	(s)							

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							×	(e)	(l)	(l)	(s)							

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										
							(o)	(d)										

Good suffix examples

$P =$ sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							×	(e)	(l)	(l)	(s)							

$P =$ odetofood

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										
							(o)	(d)										

- Crucial ingredient: longest suffix of $P[i{+}1..m{-}1]$ that occurs in P.

Last-Occurrence Function

- Preprocess the pattern P and the alphabet Σ
- Build the last-occurrence function L mapping Σ to integers
- $L(c)$ is defined as
  - the largest index i such that $P[i] = c$ or
  - -1 if no such index exists
- Example: $\Sigma = \{a, b, c, d\}, P = abacab$

c	a	b	c	d
$L(c)$	4	5	3	-1

- The last-occurrence function can be computed in time $O(m + |\Sigma|)$
- In practice, L is stored in a size-$|\Sigma|$ array.

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

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

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

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

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

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

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

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

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

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

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

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

Suffix skip array

- Again, we preprocess P to build a table.
- Suffix skip array S of size m: for $0 \leq i < m$, $S[i]$ is the largest index j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ **and** $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.

Example: $P =$ bonobobo

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

- Computed similarly to KMP failure array in $\Theta(m)$ time.

Boyer-Moore Algorithm

```
boyer-moore(T,P)
1.   L ← last occurrance array computed from P
2.   S ← suffix skip array computed from P
3.   i ← m − 1,      j ← m − 1
4.   while i < n and j ≥ 0 do
5.       if T[i] = P[j] then
6.           i ← i − 1
7.           j ← j − 1
8.       else
9.           i ← i + m − 1 − min(L[T[i]], S[j])
10.          j ← m − 1
11.  if j = −1 return i + 1
12.  else return FAIL
```

Exercise: Prove that $i - j$ always increases on lines 9–10.

Boyer-Moore algorithm conclusion

- Worst-case running time $\in O(n + |\Sigma|)$
- This complexity is difficult to prove.
- What is the worst case?
- On typical English text the algorithm probes approximately 25% of the characters in T
- Faster than KMP in practice on English text.

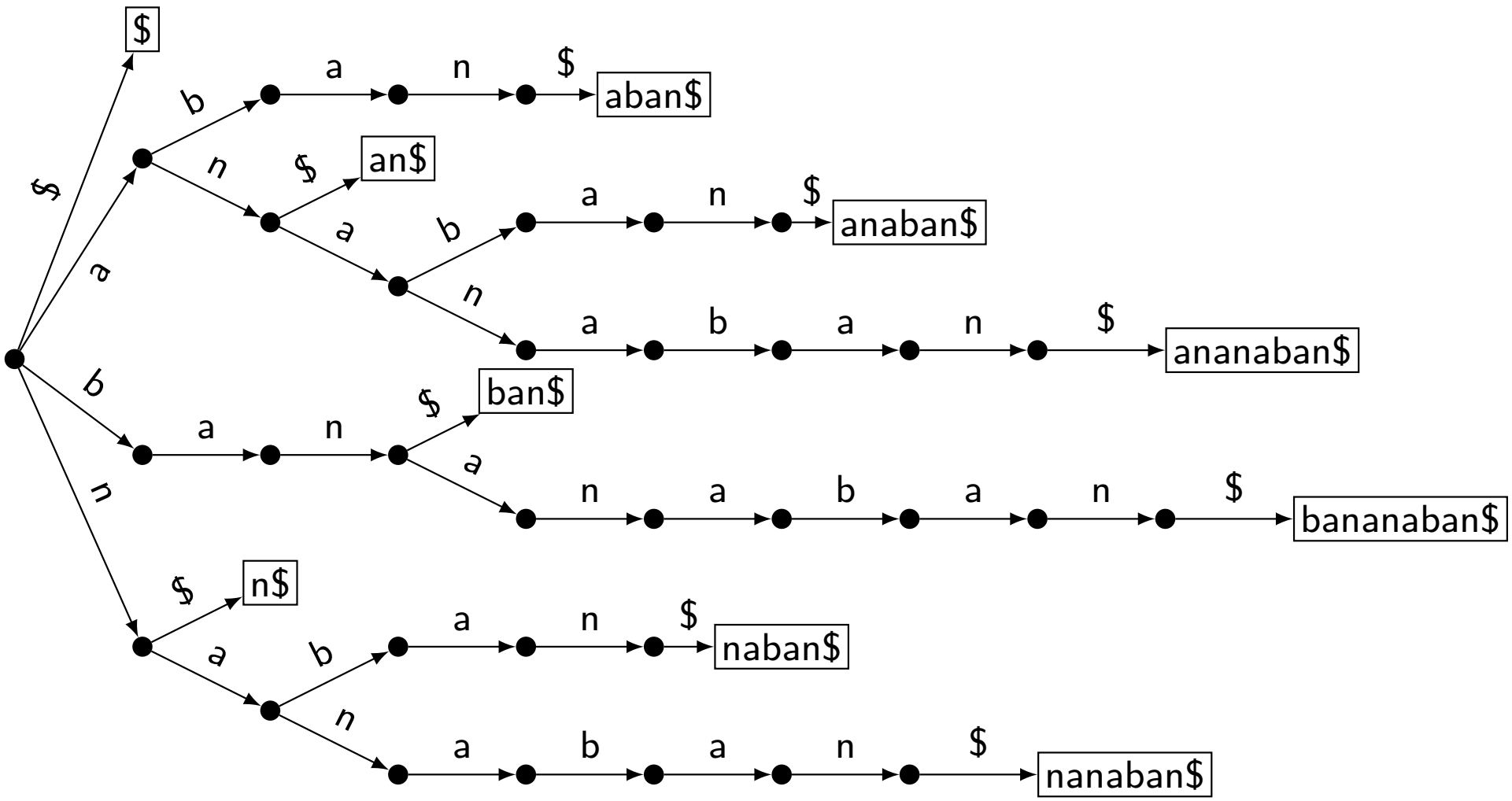
Tries of Suffixes and Suffix Trees

- 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}

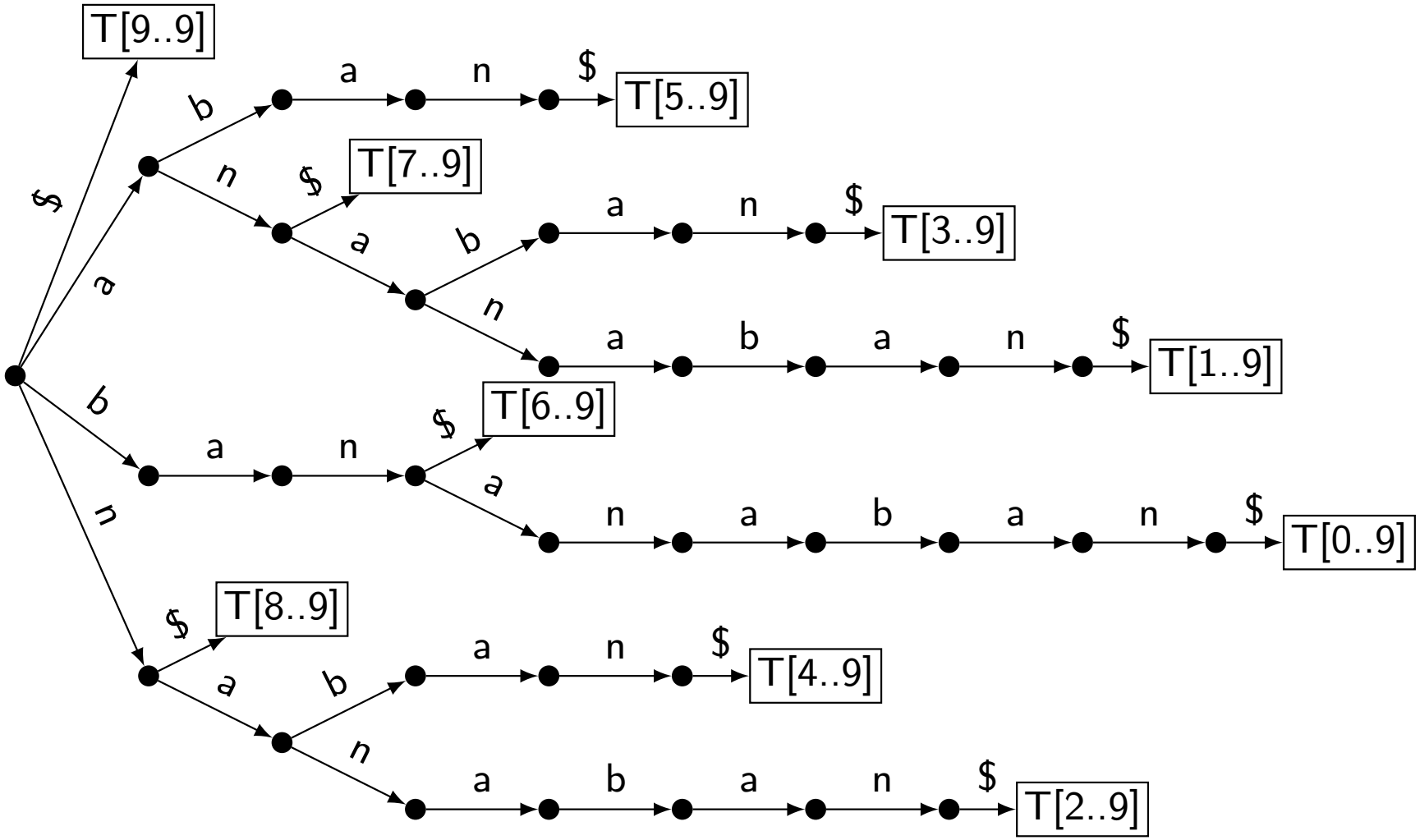


Tries of suffixes

Store suffixes via indices:

$T =$

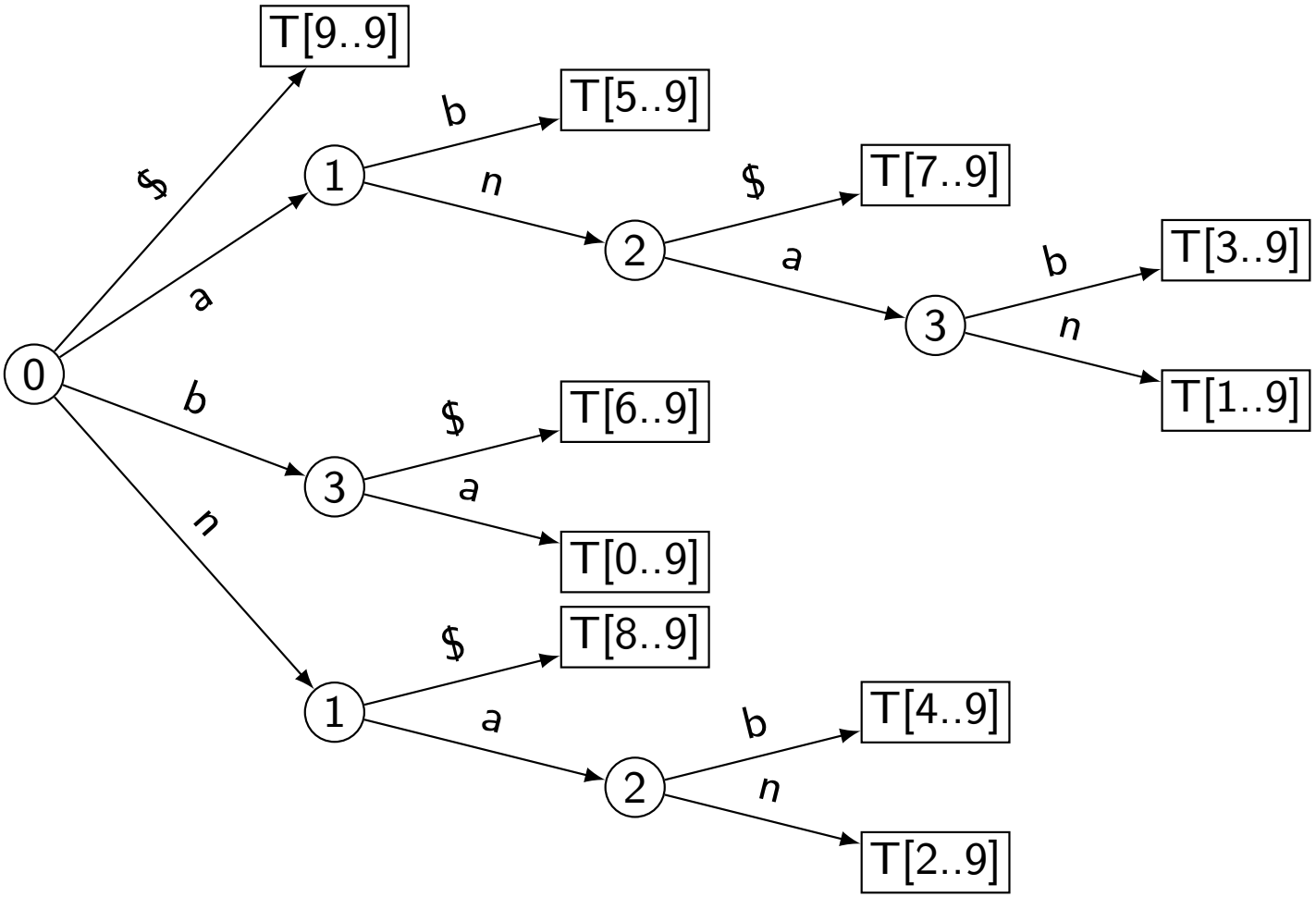
0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	$



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 $\Theta(n^2)$.
- There *is* a way to build a suffix tree of T in $\Theta(n)$ time.
  This is quite complicated and beyond the scope of the course.

Suffix Trees: String Matching

Assume we have a suffix tree of text T.

To search for pattern P of length m:

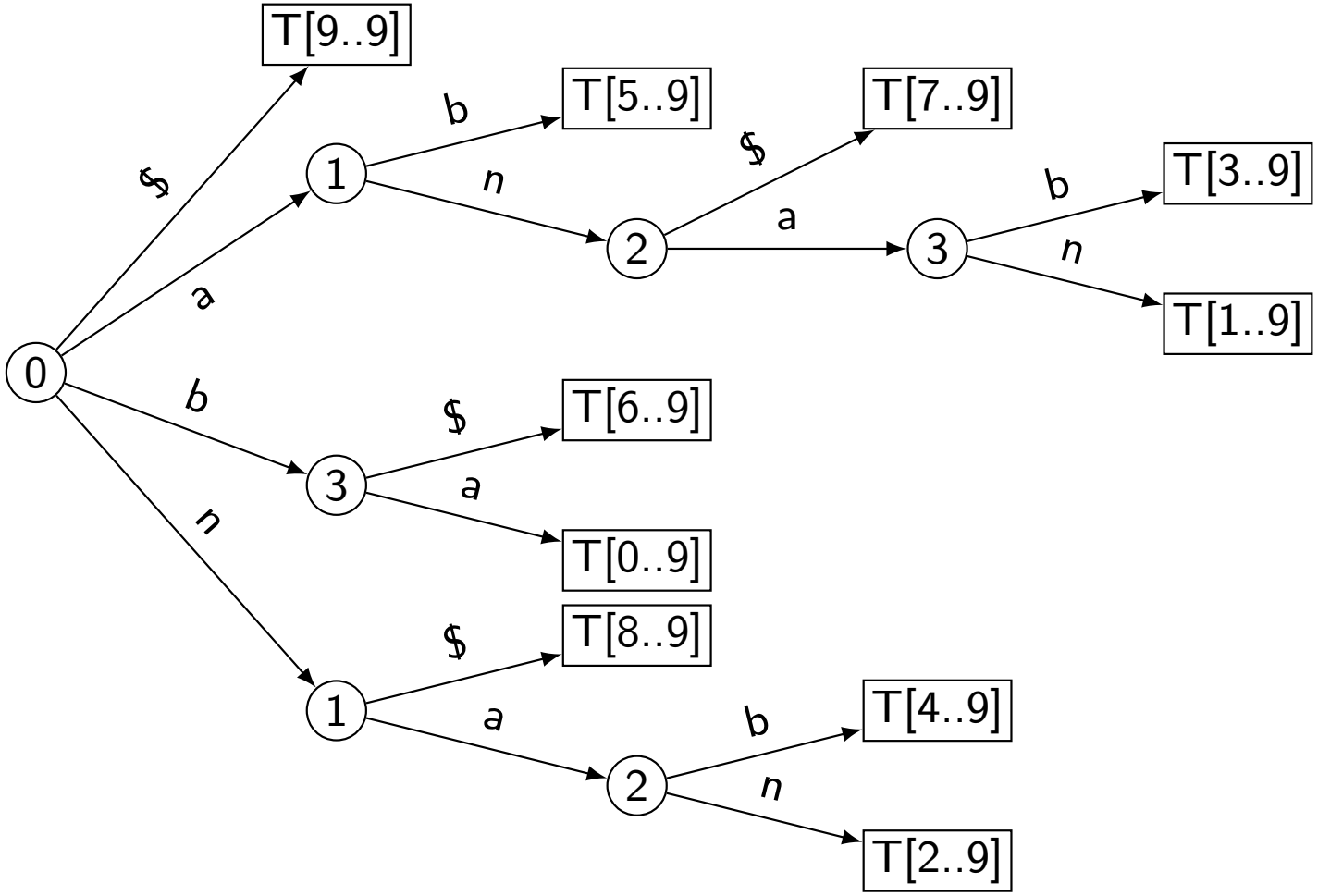
- We assume that P does not have the final \$.
- P is the prefix of some suffix of T.
- In the *uncompressed* trie, searching for P would be easy: P exists in T if and only search for P reaches a node in the trie.
- In the suffix tree, search for P until one of the follow occurs:
  1. If search fails due to *"no such child"* then P is not in T
  2. If we reach end of P, say at node v, then jump to leaf ℓ in subtree of v. (We presume that suffix trees stores such shortcuts.)
  3. Else we reach a leaf $\ell = v$ while characters of P left.
- Either way, left index at ℓ gives the shift that we should check.
- This takes $O(|P|)$ time.

Pattern Matching in Suffix Tree: Example 1

$T =$

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

$P = \texttt{ann}$

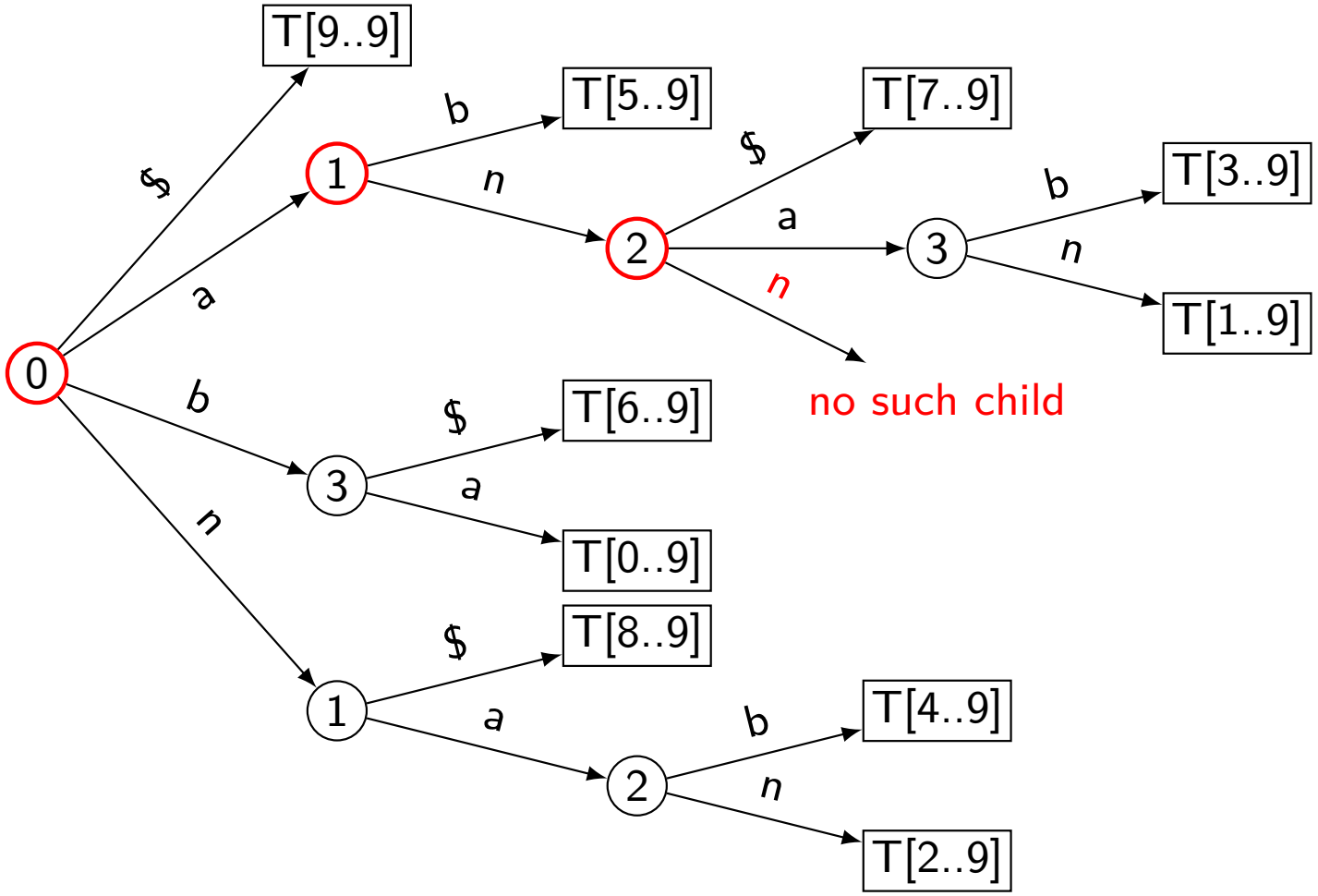


Pattern Matching in Suffix Tree: Example 1

$T =$

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

$P = \texttt{ann}$

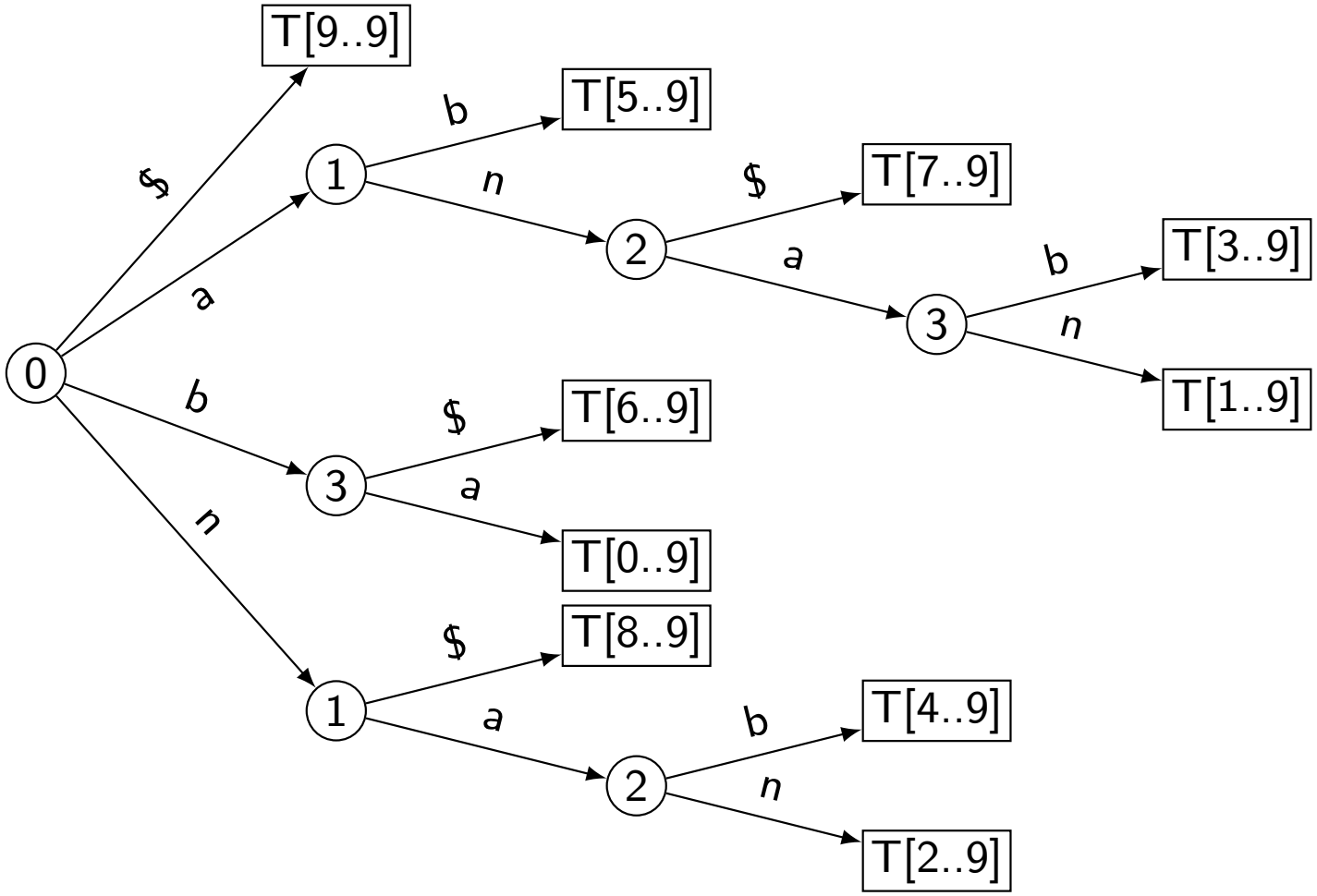


Pattern Matching in Suffix Tree: Example 2

$T =$

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

$P = \texttt{ana}$

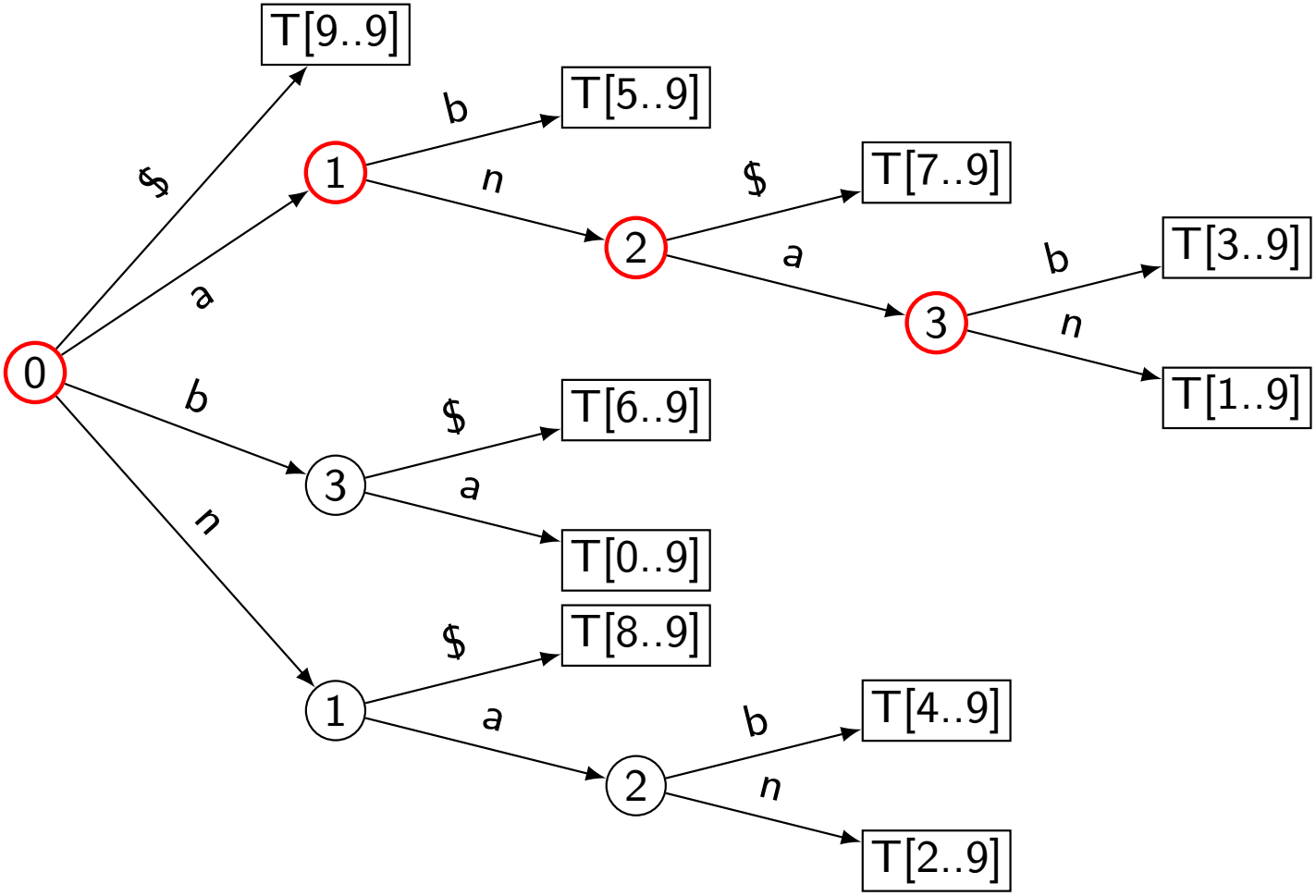


Pattern Matching in Suffix Tree: Example 2

$T =$

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

$P = \texttt{ana}$

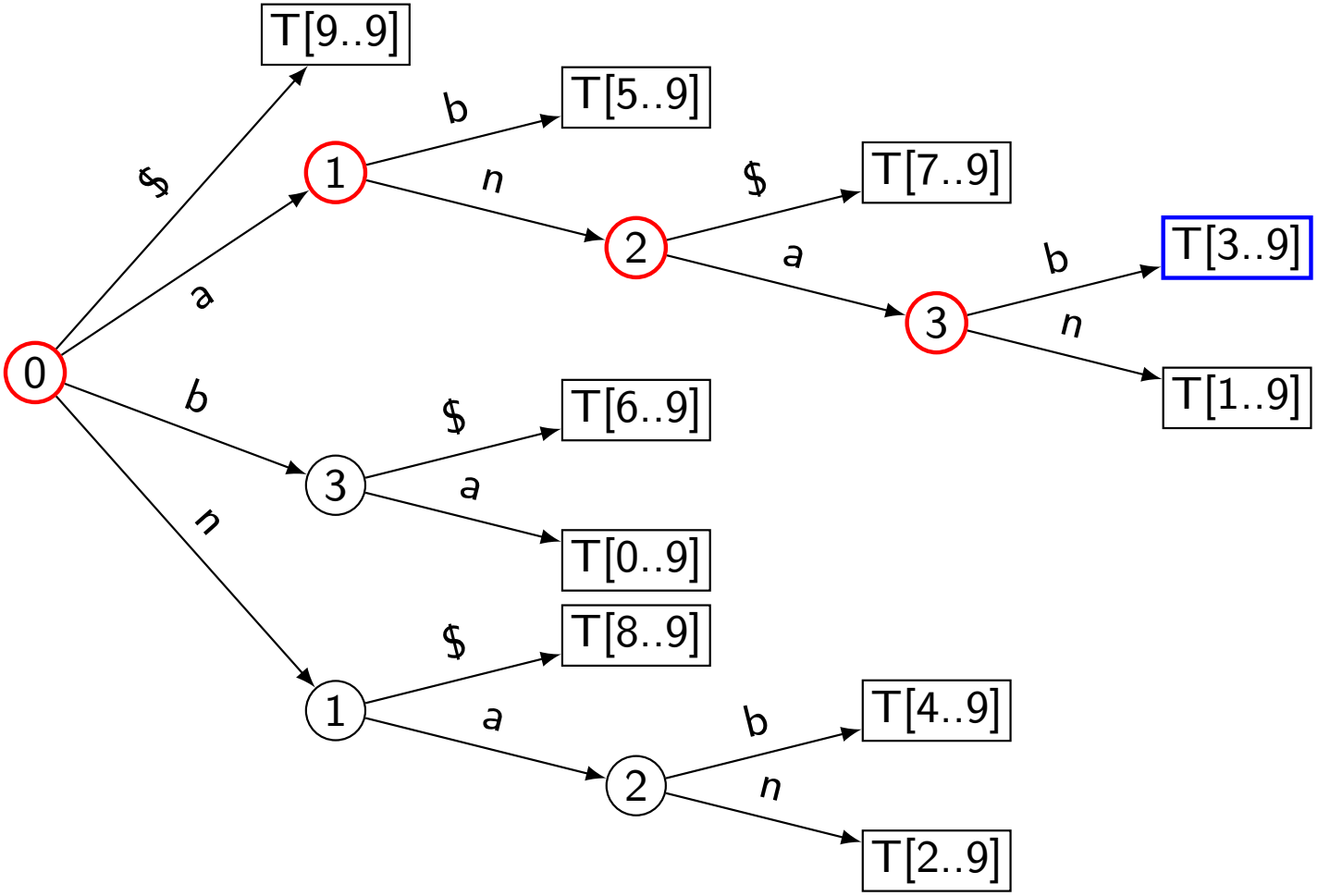


Pattern Matching in Suffix Tree: Example 2

$T =$

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

$P = \texttt{ana}$

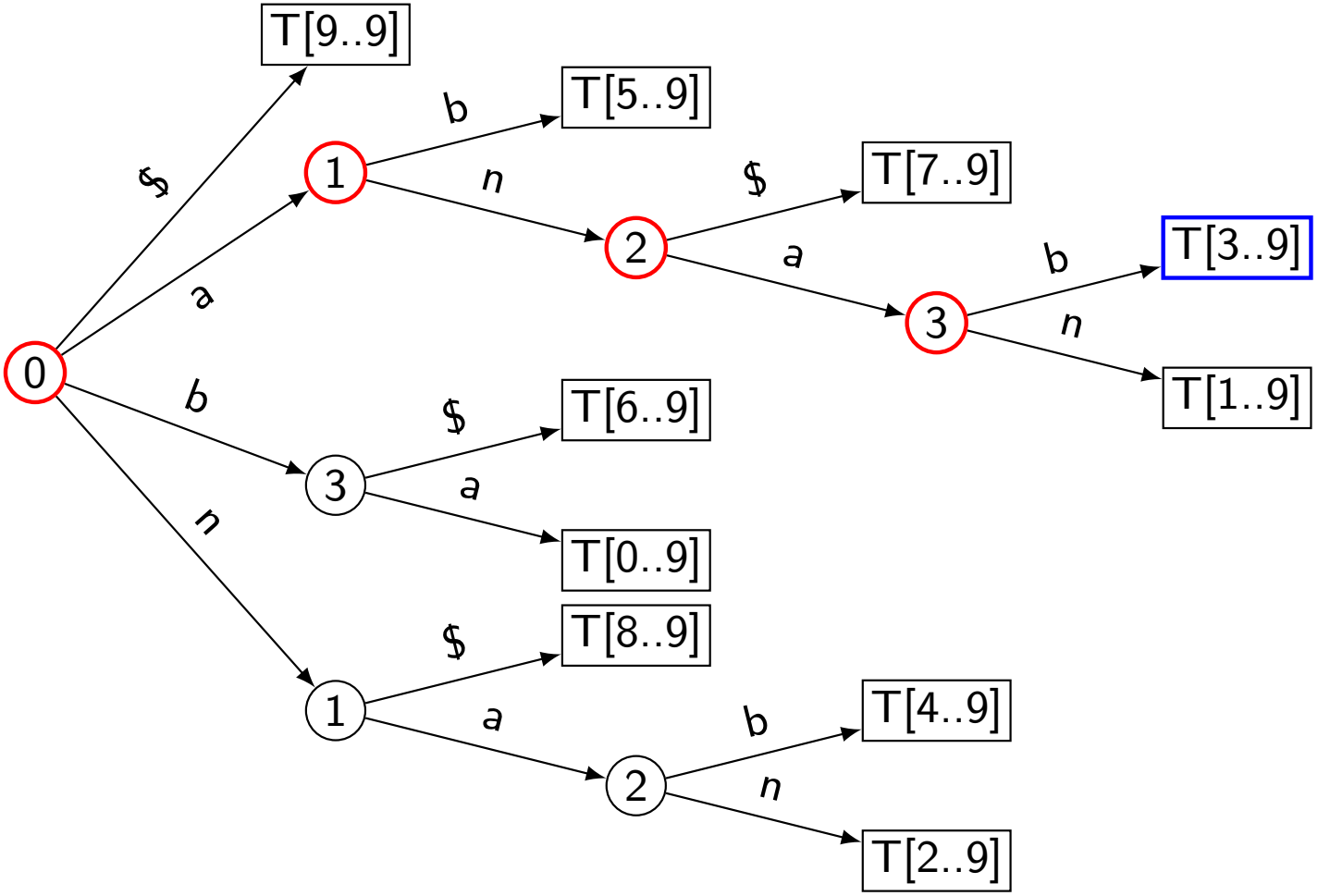


Pattern Matching in Suffix Tree: Example 2

$T =$

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

$P = \texttt{ana}$

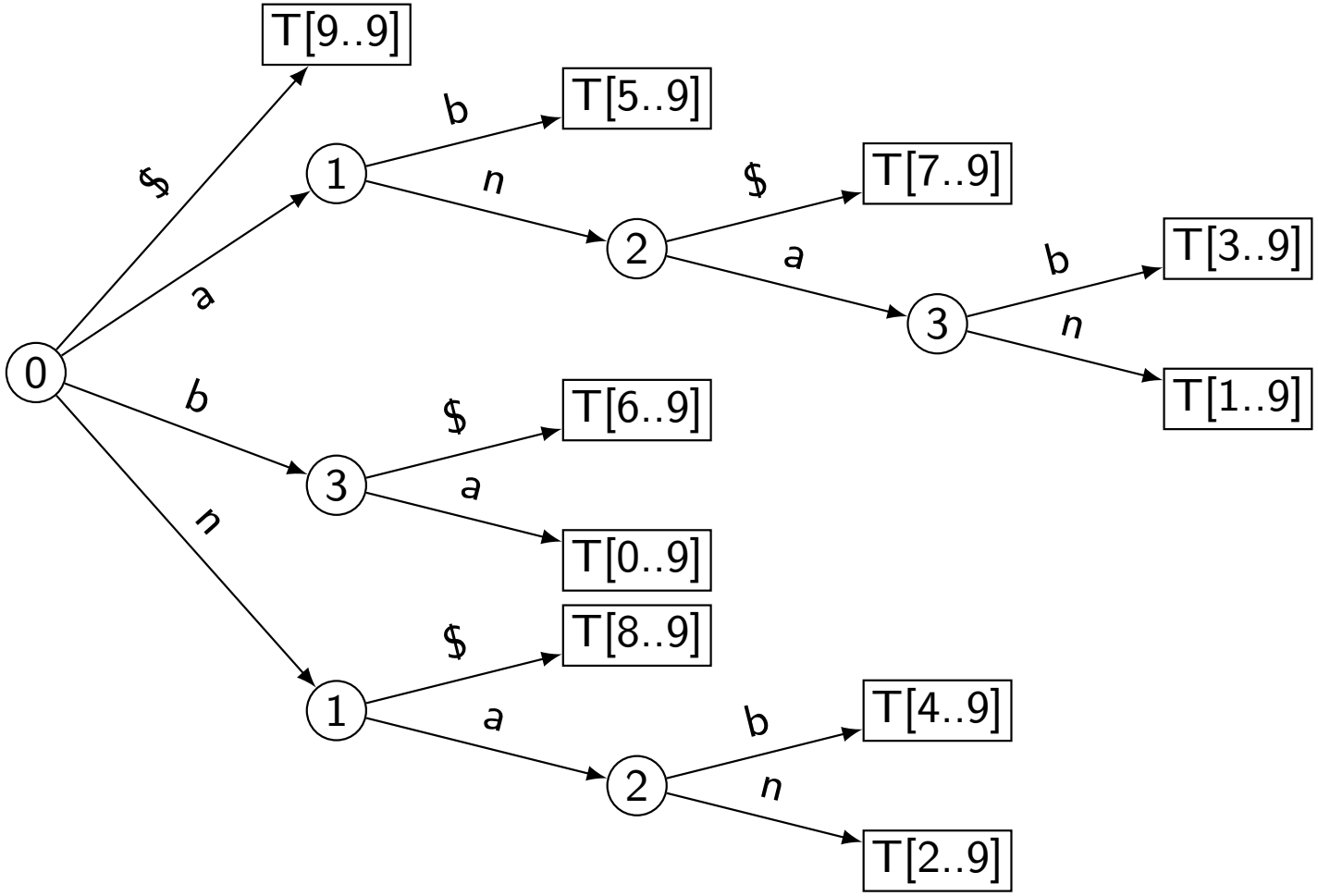


Pattern Matching in Suffix Tree: Example 3

$T =$

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

$P =$ `briar`

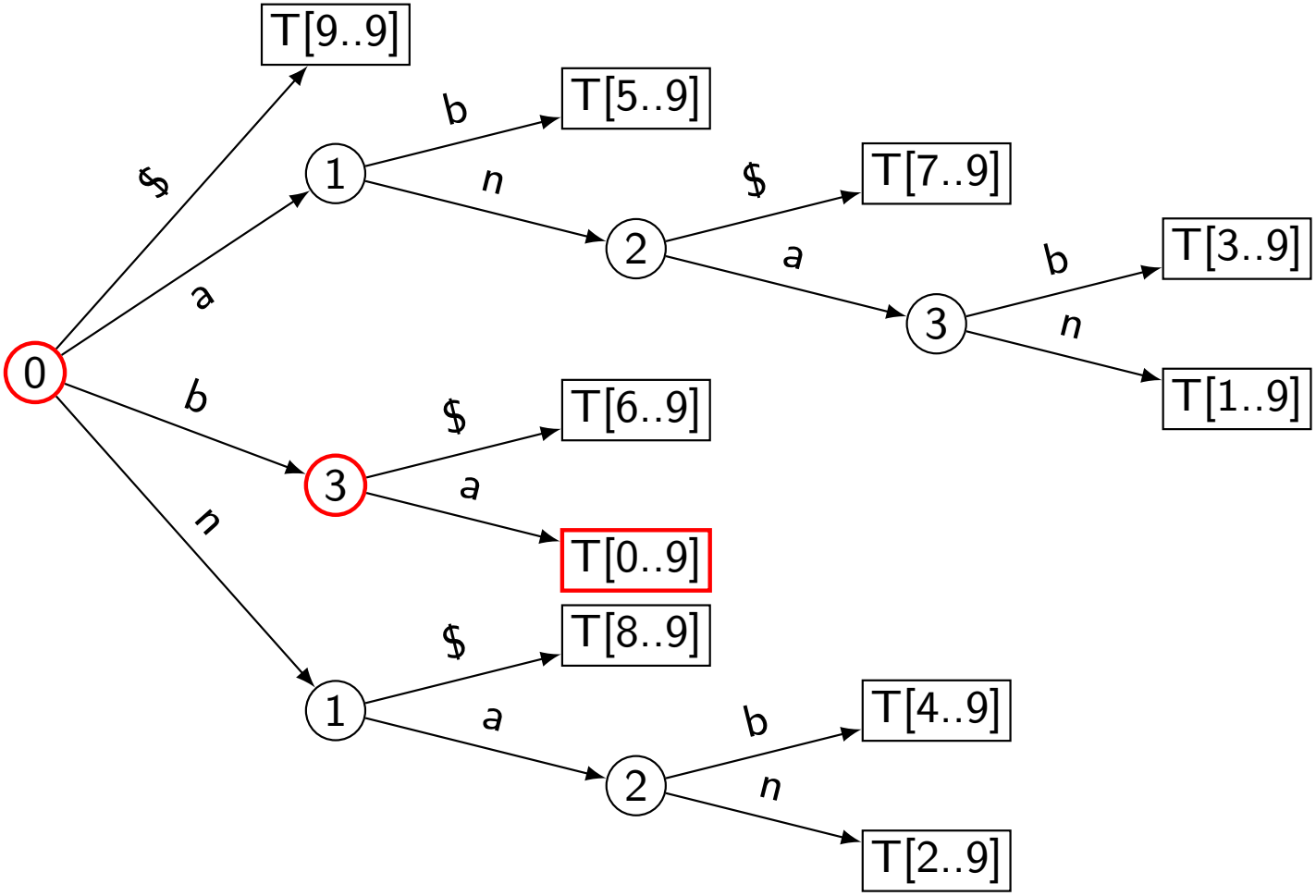


Pattern Matching in Suffix Tree: Example 3

$T =$

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

$P =$ `briar`

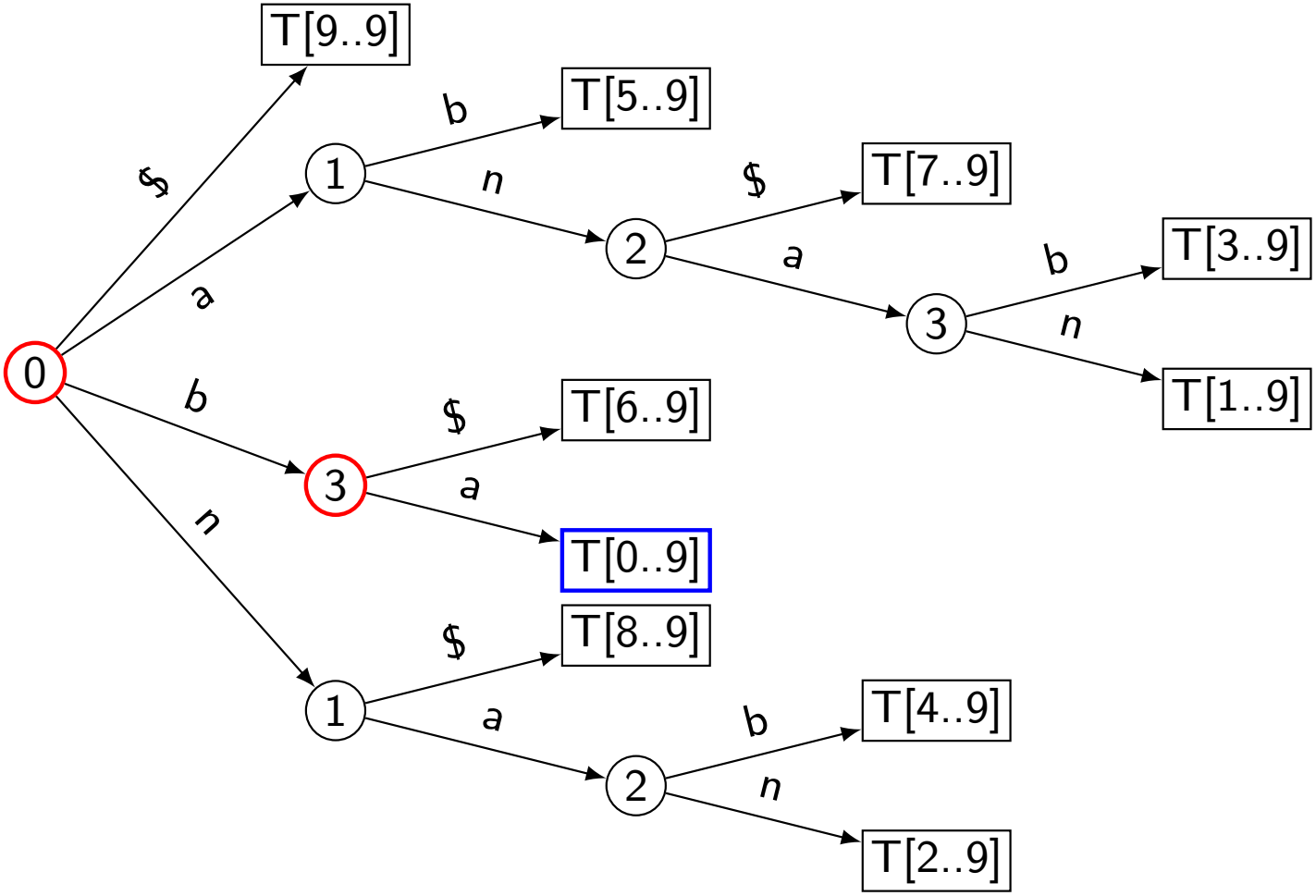


Pattern Matching in Suffix Tree: Example 3

$T =$

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

$P =$ `briar`

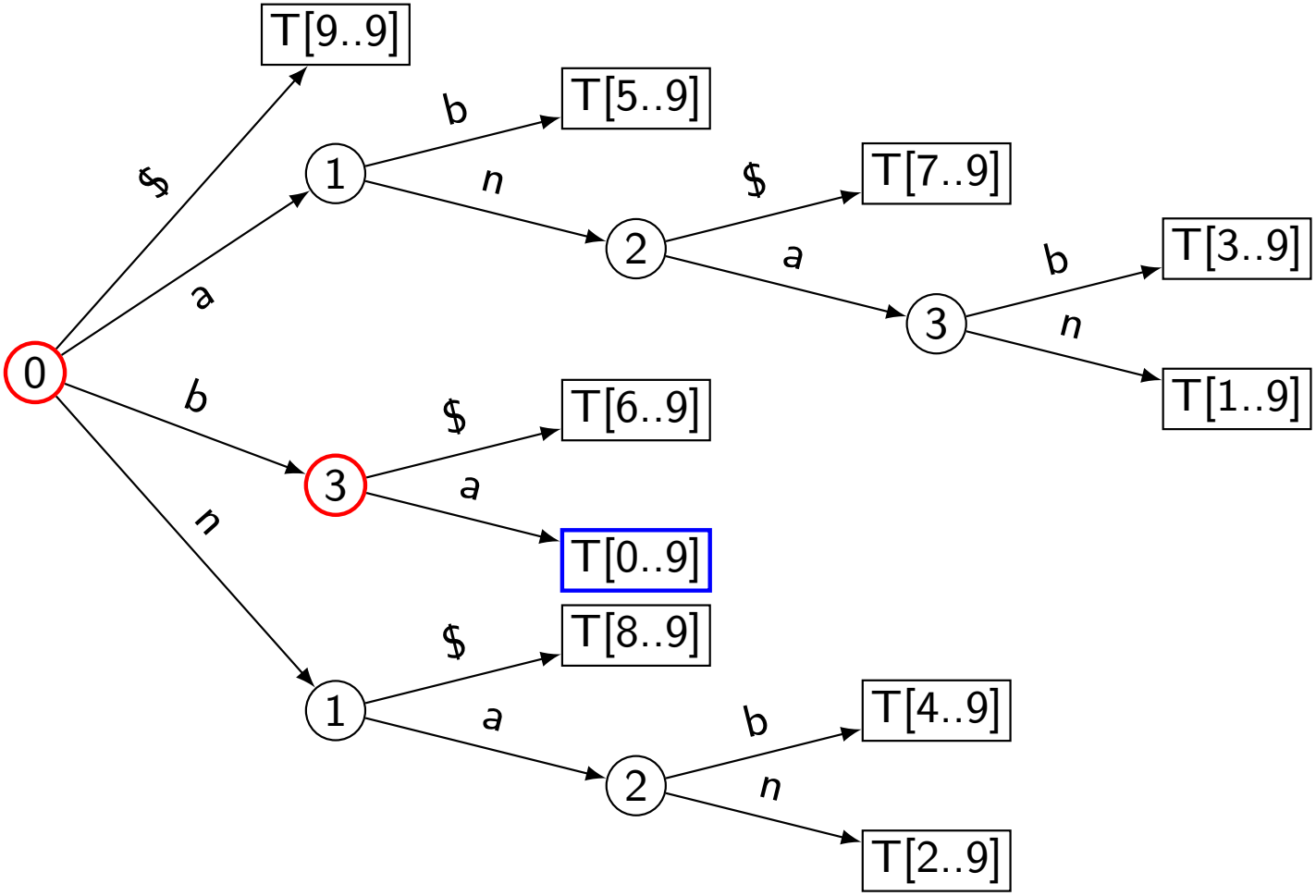


Pattern Matching in Suffix Tree: Example 3

$T =$

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

$P =$ `briar`



Pattern Matching Conclusion

	Brute-Force	KMP	Boyer-Moore	Suffix trees
Preprocessing:	–	$O(m)$	$O(m + \|\Sigma\|)$	$O(n^2)$
Search time:	$O(nm)$	$O(n)$	$O(n)$ (often better)	$O(m)$
Extra space:	–	$O(m)$	$O(m + \|\Sigma\|)$	$O(n)$