String Searching

Martin Kay

Stanford University

The String-searching Problems

- Decide whether a *text* (of t characters) contains a *given* substring, or *pattern* (of p characters)
- Identify all instances of the pattern in a text.

naive_match

```
def naiveMatch(pattern, text):
  for startPos in range(len(text) - len(pattern) + 1):
    matchLen = 0
    while pattern[matchLen] == text[startPos + matchLen]:
      matchLen += 1
      if matchLen == len(pattern):
        yield startPos
        break
```

Time: $O(p \times t)$

```
for i in naiveMatch(pat, text):
  print i
```

Complexity

```
> naive_search.py -qt aaaa aaaaaaaaaaaa
|....|....|....|....|....|....|....|....|....
text length = 12, pattern length = 4
9 iterations of text loop
36 iterations of pattern loop
```

Monotonic text scan

Examine text characters one-by-one from left to right.

- Convenient for handling long texts
- Reveals new algorithmic possibilities

Objects

```
class naive_search(object):
  def __init__(self, ...)
    ...
  def compile(self, pattern):
    self.pattern = pattern
    self.pattern_length=len(pattern}
    ...
  def seach(self, text):
    ...
```

Monotonic text scan

```
def search(self, text):
  locs=[0]
  for t in range(len(text)):
    new_locs=[0]
    for loc in locs:
      if text[t]==pattern[loc]:
        if loc+1==self.pattern_length:
          yield t-self.pattern_length+1
        else:
          new_locs.append(loc+1)
    locs=new_locs
```

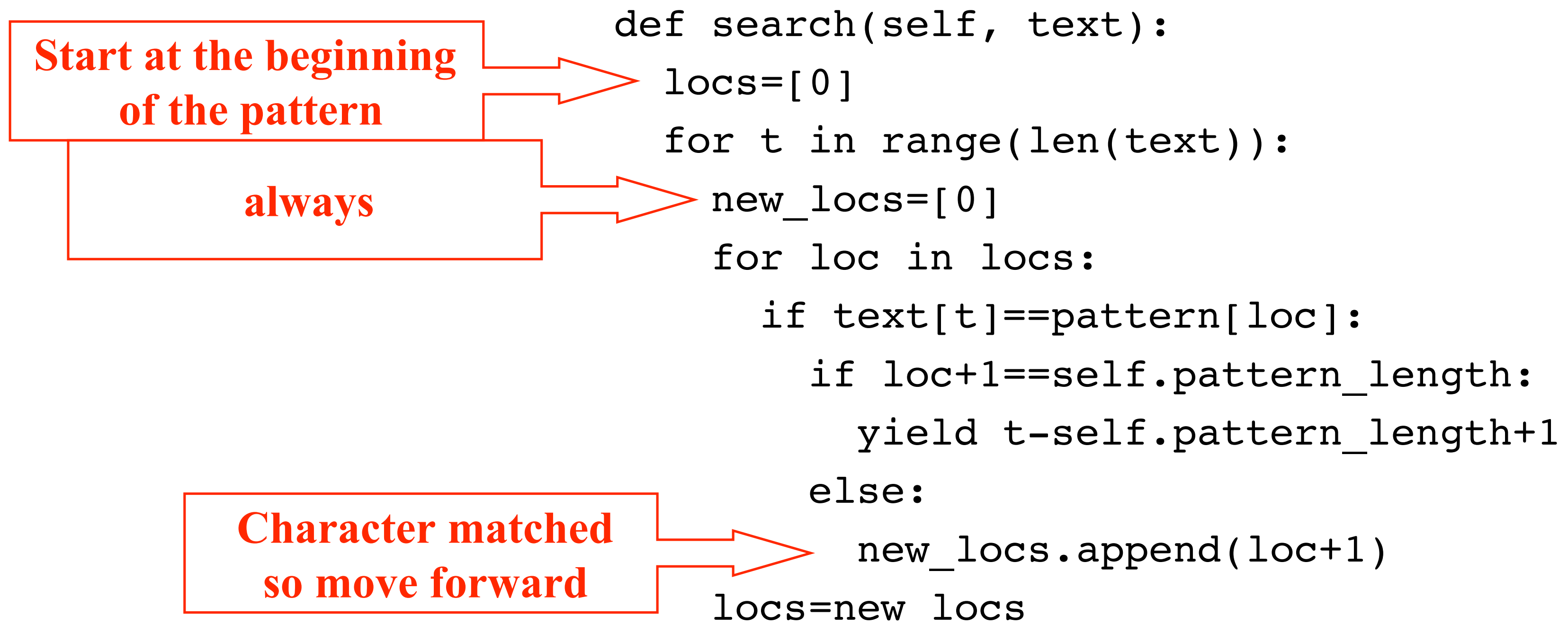
locs: *the positions in the pattern that have been reached when the next text character is encountered*

Monotonic text scan

```
def search(self, text):
  locs=[0]
  for t in range(len(text)):
    new_locs=[0]
    for loc in locs:
      if text[t]==pattern[loc]:
        if loc+1==self.pattern_length:
          yield t-self.pattern_length+1
        else:
          new_locs.append(loc+1)
    locs=new_locs
```

newlocs: *the positions in the pattern that will have been reached after the next text character is processed*

Monotonic text scan




```
def search(self, text):
  locs=[0]
  for t in range(len(text)):
    new_locs=[0]
    for loc in locs:
      if text[t]==pattern[loc]:
        if loc+1==self.pattern_length:
          yield t-self.pattern_length+1
        else:
          new_locs.append(loc+1)
    locs=new_locs
```

Monotonic text scan

```
def search(self, text):
  locs=[0]
  for t in range(len(text)):
    new_locs=[0]
    for loc in locs:
      if text[t]==pattern[loc]:
        if loc+1==self.pattern_length:
          yield t-self.pattern_length+1
        else:
          new_locs.append(loc+1)
    locs=new_locs
```

The whole pattern matched so ...

Get ready for next character

Eureka

Overlaps

Search for

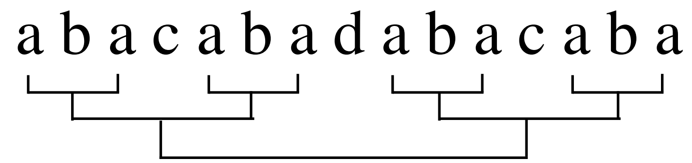
a b a c a b a d a b a c a b a

in the text

a b a b a c a b a d a b a c a b a d a b a c a b a b a

a b a c a b a d a b a c a b a
a b a c a b a d a b a c a b a
a b a c a b a d a b a c a b a
a b a c a b a d a b a c a b a
a b a c a b a d a b a c a b a
a b a c a b a d a b a c a b a
a b a c a b a d a b a c a b a
a b a c a b a d a b a c a b a
a b a c a b a d a b a c a b a

Overlaps



Déja vu

Search for

a b a c a b a d a b a c a b a

in the text

a b a b a c a b a d a b a c a b a d a b a c a b a b a

a b a c a b a d a b a c a b a

a b a c a b a d a b a c a b a

a b a c a b a d a b a c a b a

a b a c a b a d a b a c a b a

a b a c a b a d a b a c a b a

a b a c a b a d a b a c a b a

a b a c a b a d a b a c a b a

a b a c a b a d a b a c a b a

a b a c a b a d a b a c a b a

Search for

a b a c a b a d a b a c a b a

in the text

a b a b a c a b a d a b a c a b a d a b a c a b a b a

```
0 a [0]
1 b [0, 1]
2 a [0, 2]
3 b [0, 1, 3]
4 a [0, 2]
5 c [0, 1, 3]
6 a [0, 4]
7 b [0, 1, 5]
8 a [0, 2, 6]
9 d [0, 1, 3, 7]
10 a [0, 8]
11 b [0, 1, 9]
12 a [0, 2, 10]
13 c [0, 1, 3, 11]
14 a [0, 4, 12]
15 b [0, 1, 5, 13]
16 a [0, 2, 6, 14]
result 2
17 d [0, 1, 3, 7]
18 a [0, 8]
19 b [0, 1, 9]
20 a [0, 2, 10]
21 c [0, 1, 3, 11]
22 a [0, 4, 12]
23 b [0, 1, 5, 13]
24 a [0, 2, 6, 14]
result 10
25 b [0, 1, 3, 7]
26 a [0, 2]
```

Search for

a b a c a b a d a b a c a b a

in the text

a b a b a c a b a d

```
0 a [0]
1 b [0, 1]
2 a [0, 2]
3 b [0, 1, 3]
4 a [0, 2]
5 c [0, 1, 3]
6 a [0, 4]
7 b [0, 1, 5]
8 a [0, 2, 6]
9 d [0, 1, 3, 7]
10 a [0, 8]
11 b [0, 1, 9]
12 a [0, 2, 10]
13 c [0, 1, 3, 11]
14 a [0, 4, 12]
```

```
17 d [0, 1, 3, 7]
21 c [0, 1, 3, 11]
24 a [0, 2, 6, 14]
26 a [0, 2]
```

1. **The rightmost pointer always moves.**
2. **Others pointers move if they can do so over the same character**
3. **A new '0' is introduced on the left**

A pointer in a given position always has pointers in the same set of positions to its left

These are properties of the pattern *only*.

Therefore they can be *cached* or *precompiled.*

Therefore they can be *cached* or *precompiled.*

Caching and precompiling are the soul of *dynamic programming*, a technique for avoiding redundant computation that is central to artificial intelligence in general and computational linguistics in particular.

Dynamic programming can dramatically alter the complexity of an algorithm.

Search for

a b a c a b a d a b a c a b a

a b a b a c a b a d a b a c a b a d a b a c a b a b a

```
0 a  [0]
1 b  [0, 1]
2 a  [0, 2]
3 b  [0, 1, 3]
4 a  [0, 2]
5 c  [0, 1, 3]
6 a  [0, 4]
7 b  [0, 1, 5]
8 a  [0, 2, 6]
9 d  [0, 1, 3, 7]
10 a [0, 8]
11 b [0, 1, 9]
12 a [0, 2, 10]
13 c [0, 1, 3, 11]
14 a [0, 4, 12]
15 b [0, 1, 5, 13]
16 a [0, 2, 6, 14]
result 2
17 d [0, 1, 3, 7]
18 a [0, 8]
19 b [0, 1, 9]
20 a [0, 2, 10]
21 c [0, 1, 3, 11]
22 a [0, 4, 12]
23 b [0, 1, 5, 13]
24 a [0, 2, 6, 14]
result 10
25 b [0, 1, 3, 7]
26 a [0, 2]
```

If this matches ...

then so will these

Search for

a b a c a b a d a b a c a b a

a b a b a c a b a d a b a c a b a d a b a c a b a b a

```
0 a  [0]
1 b  [0, 1]
2 a  [0, 2]
3 b  [0, 1, 3]
4 a  [0, 2]
5 c  [0, 1, 3]
6 a  [0, 4]
7 b  [0, 1, 5]
8 a  [0, 2, 6]
9 d  [0, 1, 3, 7]
10 a [0, 8]
11 b [0, 1, 9]
12 a [0, 2, 10]
13 c [0, 1, 3, 11]
14 a [0, 4, 12]
15 b [0, 1, 5, 13]
16 a [0, 2, 6, 14]
result 2
17 d [0, 1, 3, 7]
18 a [0, 8]
19 b [0, 1, 9]
20 a [0, 2, 10]
21 c [0, 1, 3, 11]
22 a [0, 4, 12]
23 b [0, 1, 5, 13]
24 a [0, 2, 6, 14]
result 10
25 b [0, 1, 3, 7]
26 a [0, 2]
```

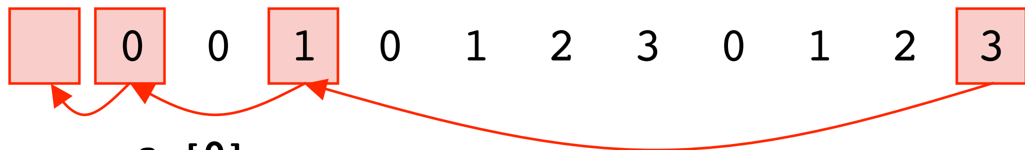
So try these

only if this fails!

0	1	2	3	4	5	6	7	8	9	10	11	12	...
a	b	a	c	a	b	a	d	a	b	a	c	a	...
	0	0	1	0	1	2	3	0	1	2	3	4	...

The *failure* function

```
a [0]
b [0, 1]
a [0, 2]
b [0, 1, 3]
a [0, 2]
c [0, 1, 3]
a [0, 4]
b [0, 1, 5]
a [0, 2, 6]
d [0, 1, 3, 7]
a [0, 8]
b [0, 1, 9]
a [0, 2, 10]
c [0, 1, 3, 11]
a [0, 4, 12]
```




```
a [0]
b [0, 1]
a [0, 2]
b [0, 1, 3]
a [0, 2]
c [0, 1, 3]
a [0, 4]
b [0, 1, 5]
a [0, 2, 6]
d [0, 1, 3, 7]
a [0, 8]
b [0, 1, 9]
a [0, 2, 10]
c [0, 1, 3, 11]
a [0, 4, 12]
```

0	1	2	3	4	5	6	7	8	9	10	11	12	...
a	b	a	c	a	b	a	d	a	b	a	c	a	...
-1	0	0	1	0	1	2	3	0	1	2	3	4	...

```
a [0]
b [0, 1]
a [0, 2]
b [0, 1, 3]
a [0, 2]
c [0, 1, 3]
a [0, 4]
b [0, 1, 5]
a [0, 2, 6]
d [0, 1, 3, 7]
a [0, 8]
b [0, 1, 9]
a [0, 2, 10]
c [0, 1, 3, 11]
a [0, 4, 12]
```

We will use -1 to mark the end of the list

Caching Search

```
def search(self, text):
    pattern=self.pattern
    loc=0
    for t in range(len(text)):
      last=-1
      while loc != -1:
        if text[t]==pattern[loc]:
          if loc+1==self.pattern_length:
            yield t-self.pattern_length+1
            last=self.alt[loc]
          last=loc
          break
        loc=self.alt[loc]
      loc=last+1
      if loc<self.pattern_length:
        if self.alt[last] > -1 and \
                text[t]==pattern[self.alt[last]]:
          self.alt[loc]=self.alt[last]+1
      else:
        loc=self.alt[last]+1
```

Beginning of pattern

Text loop is outermost

Pattern loop

Complete match

Remember matching location

Cache alternative

Use cached location

Caching Search

```
def search(self, text):
    pattern=self.pattern
    loc=0
    for t in range(len(text)):
      last=-1
      while loc != -1:
        if t
          if
          la
          break
        loc=self.alt[loc]
      loc=last+1
      if loc<self.pattern_length:
        if self.alt[last] > -1 and \
                 text[t]==pattern[self.alt[last]]:
•         self.alt[loc]=self.alt[last]+1
      else:
        loc=self.alt[last]+1
```

If the alternate of the current character would also have matched, then its successor is the alternate of the next character to be compared.

Caching Search Complexity

```
def search(self, text):
    pattern=self.pattern
    loc=0
    for t in range(len(text)):
      last=-1
      while loc != -1:
        if text[t]==pattern[loc]:
          if loc+1==self.pattern_length:
            yield t-self.pattern_length+1
            last=self.alt[loc]
          last=loc
          break
        loc=self.alt[loc]
      loc=last+1
```

How many iterations?

```
>caching_search.py abacaba ababacababa
|.|.|.|..|.|.|.|.|.|..|.
text length = 11, pattern length = 7
11 iterations of text loop
13 iterations of pattern loop
```

Caching Search Complexity

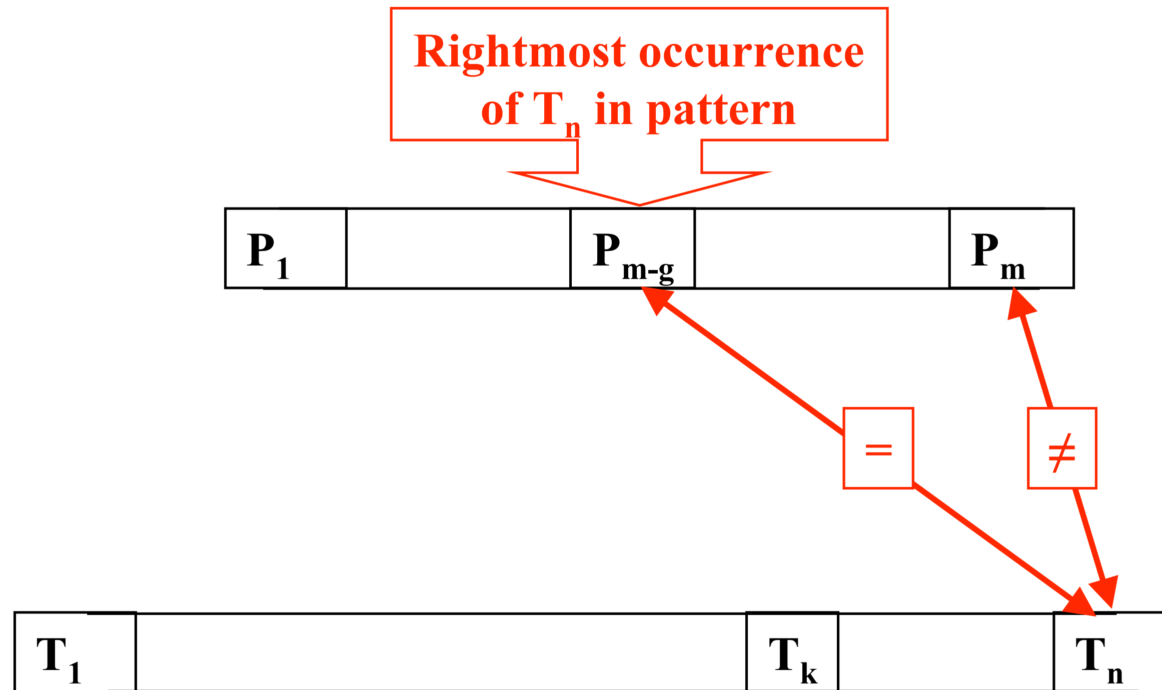
```
def search(self, text):
    pattern=self.pattern
    loc=0
    for t in range(len(text)):
      last=-1
      while loc != -1:
        if text[t]==pattern[loc]:
          if loc+1==self.pattern_length:
            yield t-self.pattern_length+1
            last=self.alt[loc]
          last=loc
          break
        loc=self.alt[loc]
      loc=last+1
      if loc<self.pattern_length:
        if self.alt[last] > -1 and \
                text[t]==pattern[self.alt[last]]:
•         self.alt[loc]=self.alt[last]+1
      else:
        loc=self.alt[last]+1
```

Each iteration of the outer loop sets up *one* (additional) iteration of the inner loop

Reference

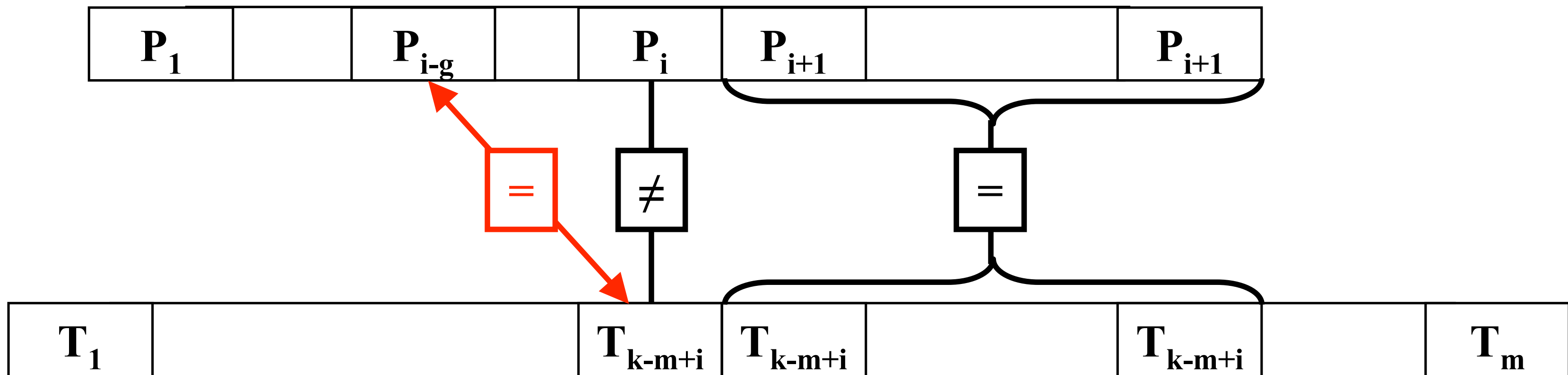
Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing* , 6(2):323-350, June 1977.

Boyer-Moore



Shift pattern right *g* places

Boyer-Moore



Shift *g* places

Shift *i* places if there is no matching $p_{i\text{-}g}$

Boyer-Moore-Horspool

```
def search(self, text):
  text_length = len(text)
  pattern_length=self.pattern_length
  if pattern_length < text_length:
      right_anchor = pattern_length - 1
      while right_anchor < text_length:
        pattern_loc = pattern_length - 1
        text_loc = right_anchor
        while pattern_loc >= 0 and text[text_loc] ==
 pattern[pattern_loc]:
          pattern_loc -= 1
          text_loc -= 1
        if pattern_loc == -1: yield (text_loc + 1)
        right_anchor += self.skip[ord(text[text_
```

Move right by pattern length

Match from right to left

Boyer-Moore-Horspool

```
def compile(self, pattern):
  self.pattern_length = pattern_length = len(pattern)
  skip = []
  for k in range(256): skip.append(pattern_length)
  for k in range(pattern_length - 1):
    skip[ord(pattern[k])] = pattern_length - k - 1
  self.skip = tuple(skip)
  self.show_skip()
  return self
```

The skip Table

a b a c a b a

0 : 7	1 : 7	2 : 7	3 : 7	4 : 7	5 : 7	6 : 7	7 : 7
8 : 7	9 : 7	10 : 7	11 : 7	12 : 7	13 : 7	14 : 7	15 : 7
16 : 7	17 : 7	18 : 7	19 : 7	20 : 7	21 : 7	22 : 7	23 : 7
24 : 7	25 : 7	26 : 7	27 : 7	28 : 7	29 : 7	30 : 7	31 : 7
32 : 7	33 : 7	34 : 7	35 : 7	36 : 7	37 : 7	38 : 7	39 : 7
40 : 7	41 : 7	42 : 7	43 : 7	44 : 7	45 : 7	46 : 7	47 : 7
48 : 7	49 : 7	50 : 7	51 : 7	52 : 7	53 : 7	54 : 7	55 : 7
56 : 7	57 : 7	58 : 7	59 : 7	60 : 7	61 : 7	62 : 7	63 : 7
64 : 7	65 A: 7	66 B: 7	67 C: 7	68 D: 7	69 E: 7	70 F: 7	71 G: 7
72 H: 7	73 I: 7	74 J: 7	75 K: 7	76 L: 7	77 M: 7	78 N: 7	79 O: 7
80 P: 7	81 Q: 7	82 R: 7	83 S: 7	84 T: 7	85 U: 7	86 V: 7	87 W: 7
88 X: 7	89 Y: 7	90 Z: 7	91 : 7	92 : 7	93 : 7	94 : 7	95 : 7
96 : 7	97 a: 2	98 b: 1	99 c: 3	100 d: 7	101 e: 7	102 f: 7	103 g: 7
104 h: 7	105 i: 7	106 j: 7	107 k: 7	108 l: 7	109 m: 7	110 n: 7	111 o: 7
112 p: 7	113 q: 7	114 r: 7	115 s: 7	116 t: 7	117 u: 7	118 v: 7	119 w: 7
120 x: 7	121 y: 7	122 z: 7	123 : 7	124 : 7	125 : 7	126 : 7	127 : 7

Avoiding multiple character examinations

Pattern: | a | b | c | a | b | c | a | b | c | d |

Pattern length = 10, so move 10 characters down the text

Text: | x | x | x | x | x | a | b | c | a | b | c | a | b | c | a | b | c | d |

Find the longest *prefix* of the pattern that ends there

Making a note of shorter ones on the way

Move down the text far enough to complete the match

Find the longest (partial) prefix of the pattern that ends there ...

but stop at the already examined region

Avoiding multiple character examinations

Pattern:

a	b	c	a	b	c	a	b	c	d

Pattern length = 10, so move 10 characters down the text

Text:

x	x	x	x	x	a	b	c	a	b	c	a	b	c	a	b	c	d

And there are five here

Trouble is that this needs two more characters

But remember this? Just what we need!

Avoiding multiple character examinations

Pattern:

a	b	c	a	b	c	a	b	c	d

Text:

x	x	x	x	x	a	b	c	a	b	c	a	b	c	a	b	c	d

So, we matched a longer prefix

Move down the text far enough to complete the match

Find the longest (partial) prefix of the pattern that ends there ...

Avoiding multiple character examinations

Pattern:

a	b	c	a	b	c	a	b	c	d

Text:

x	x	x	x	x	a	b	c	a	b	c	a	b	c	a	b	c	d

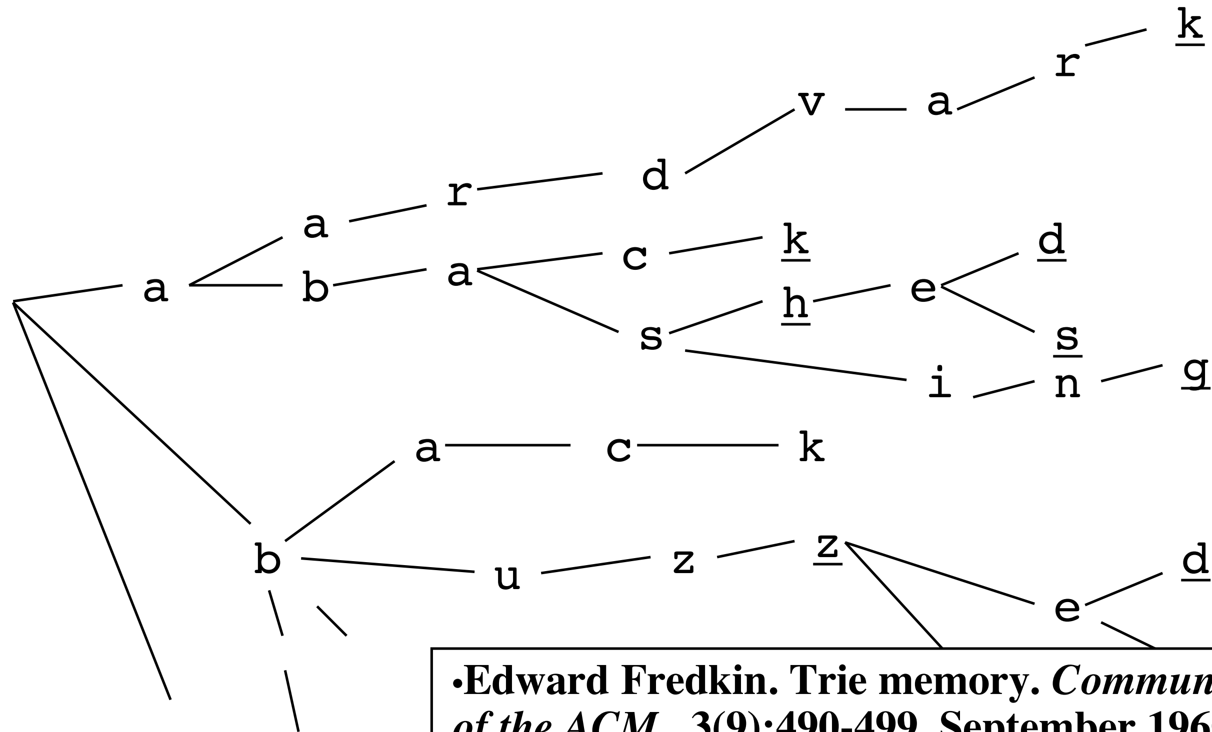
And that is just what we have !

This needs 7 characters to complete it

How to find the longest refix of the pattern efficiently

Answer: Prefix Trees
(= backwards suffix trees)

“Tries” (Digital Search Trees)

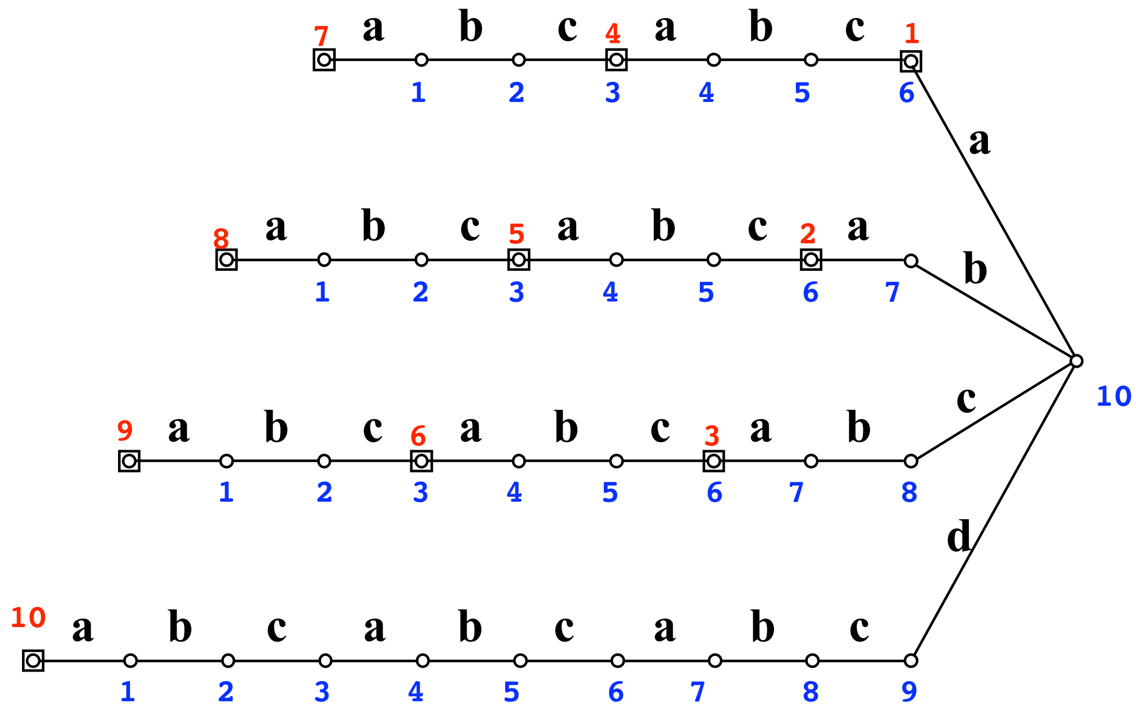


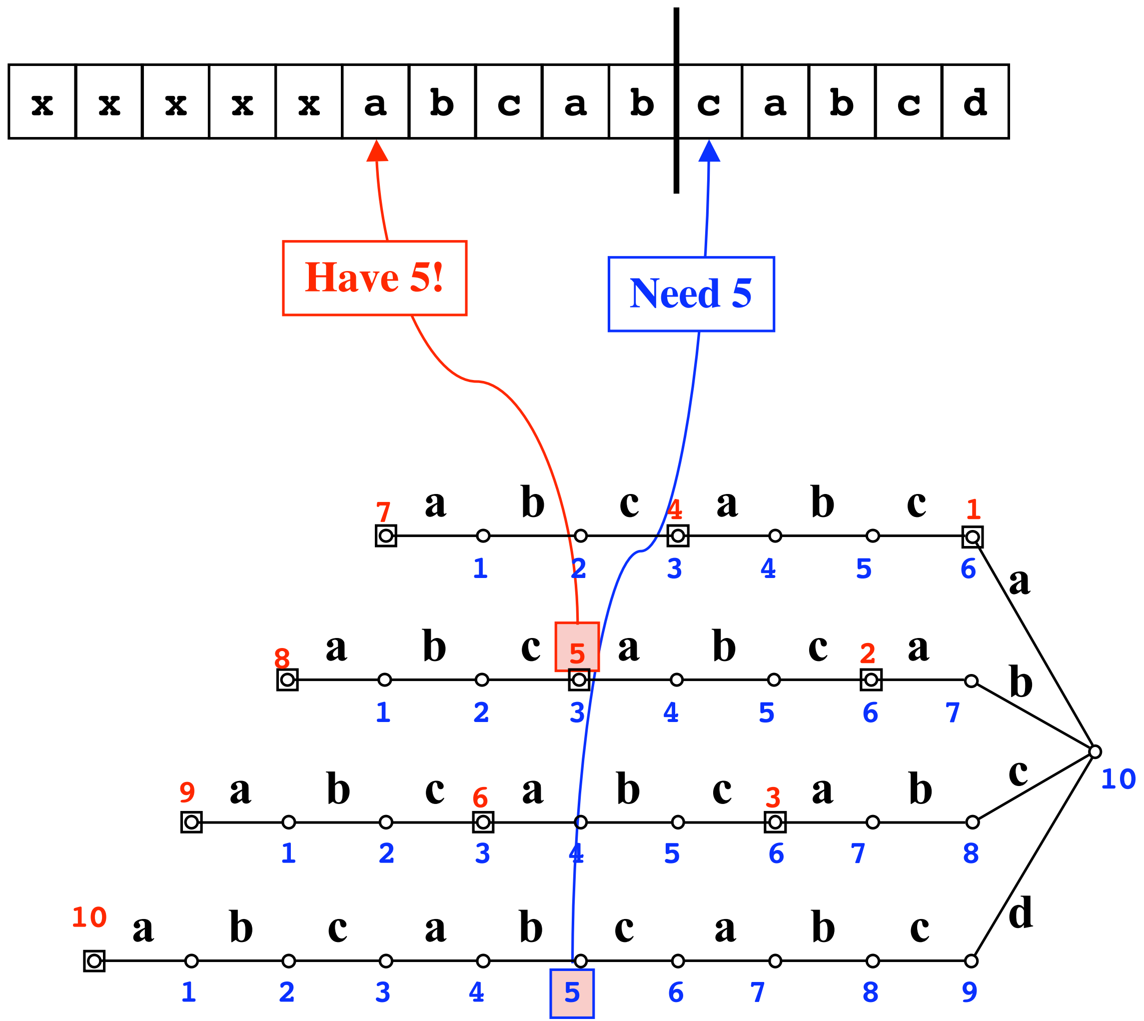
•**Edward Fredkin. Trie memory. *Communications of the ACM* , 3(9):490-499, September 1960**

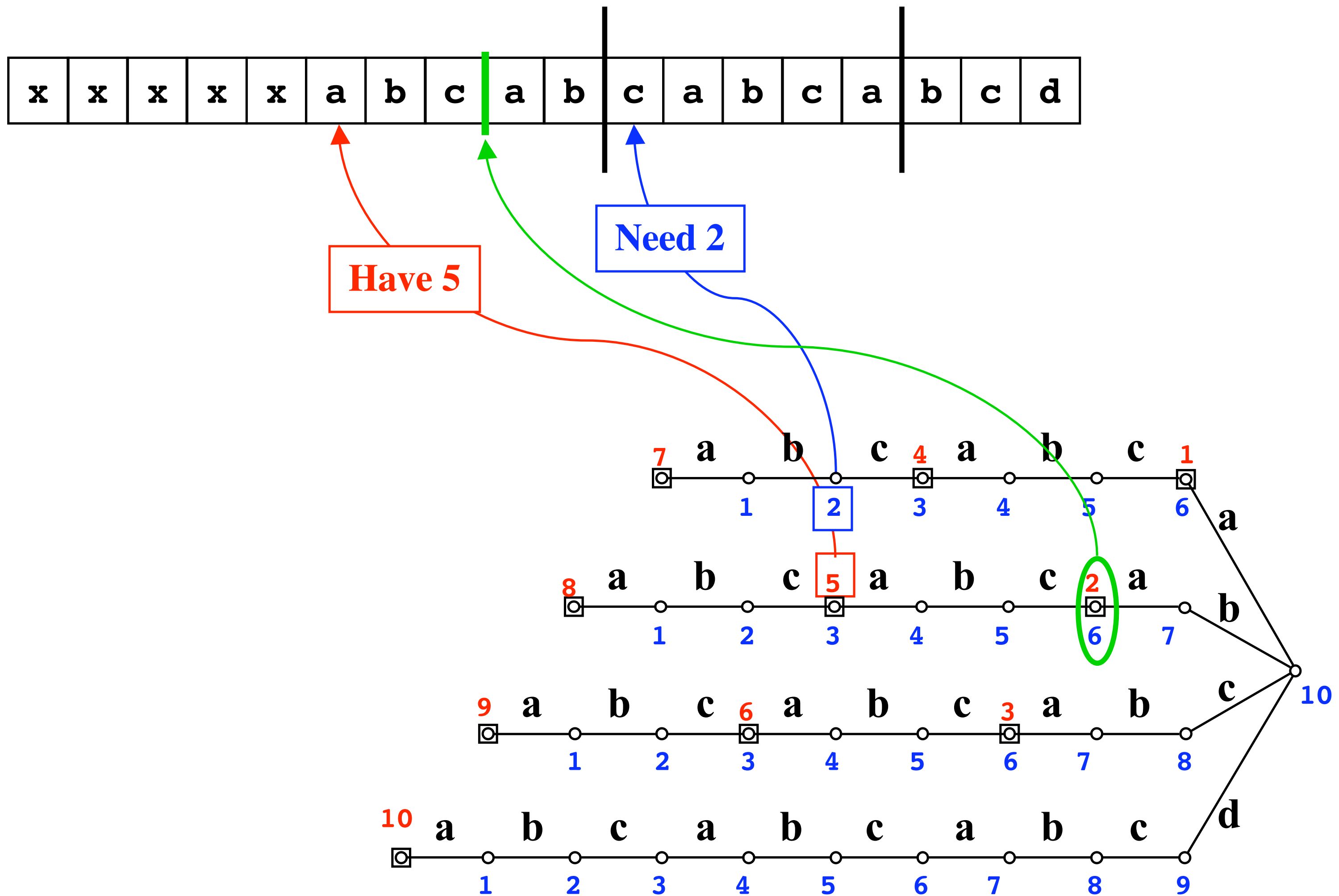
A suffix tree is a trie in which the words are the suffixes of some given string

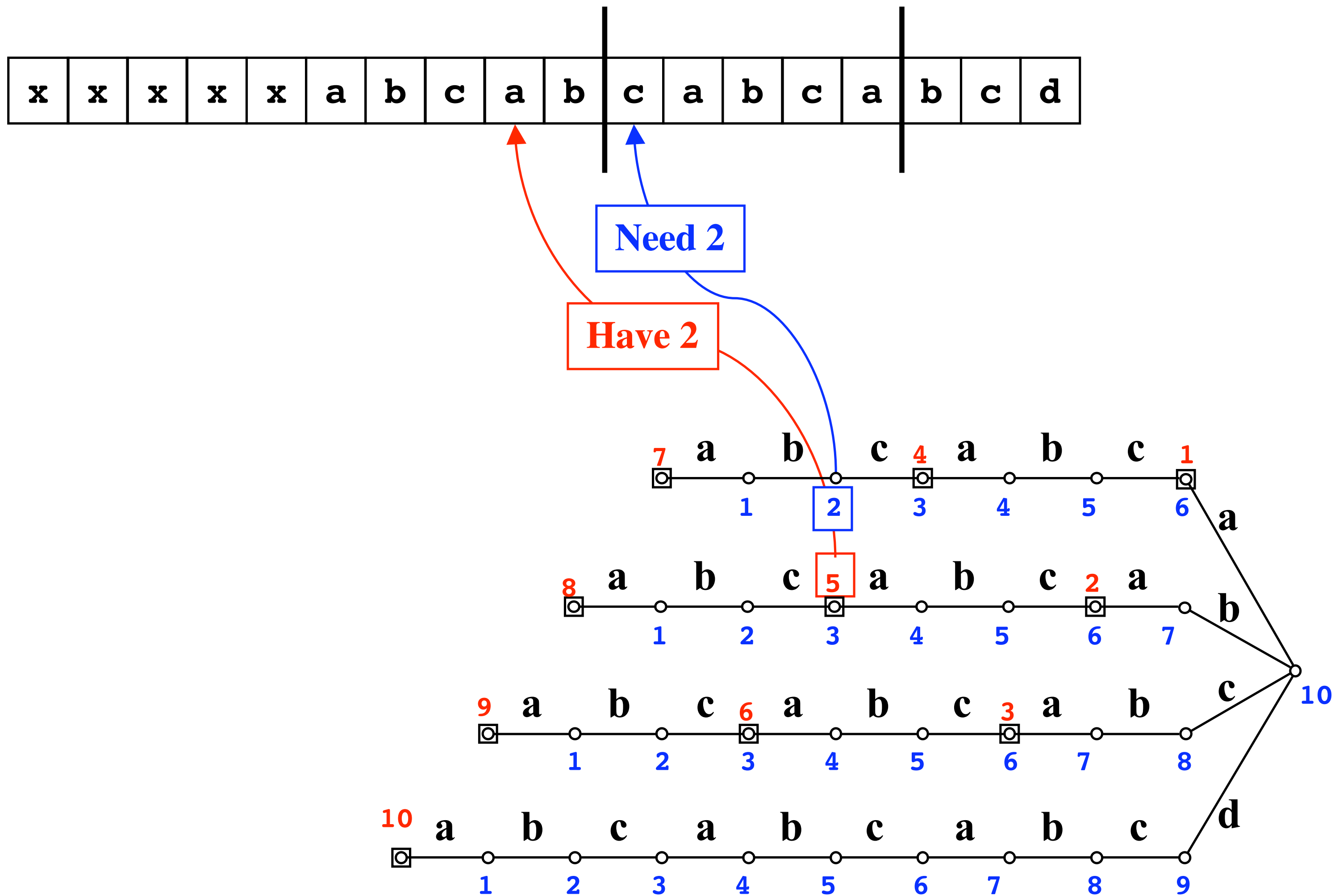
A prefix tree is (let us say) a trie in which the words are the prefixes for some given string, read from right to left.

10	9	8	7	6	5	4	3	2	1
	6	5	4	3	2	1			




x x x x x a b c a b c a b c d
Have 5!
Need 5


x x x x x a b c a b c a b c a b c d
Have 5
Need 2
a b c a b c
a b c a b c a
a b c a b c a b
a b c a b c a b c
a b c d
10


x x x x x a b c a b c a b c a b c d
Need 2
Have 2
a b c a b c
a b c a b c a
a b c a b c a b
a b c a b c a b c
a
b
c
d
10

x	x	x	x	x	a	b	c	a	b	c	a	b	c	a	b	c	d

Have 7

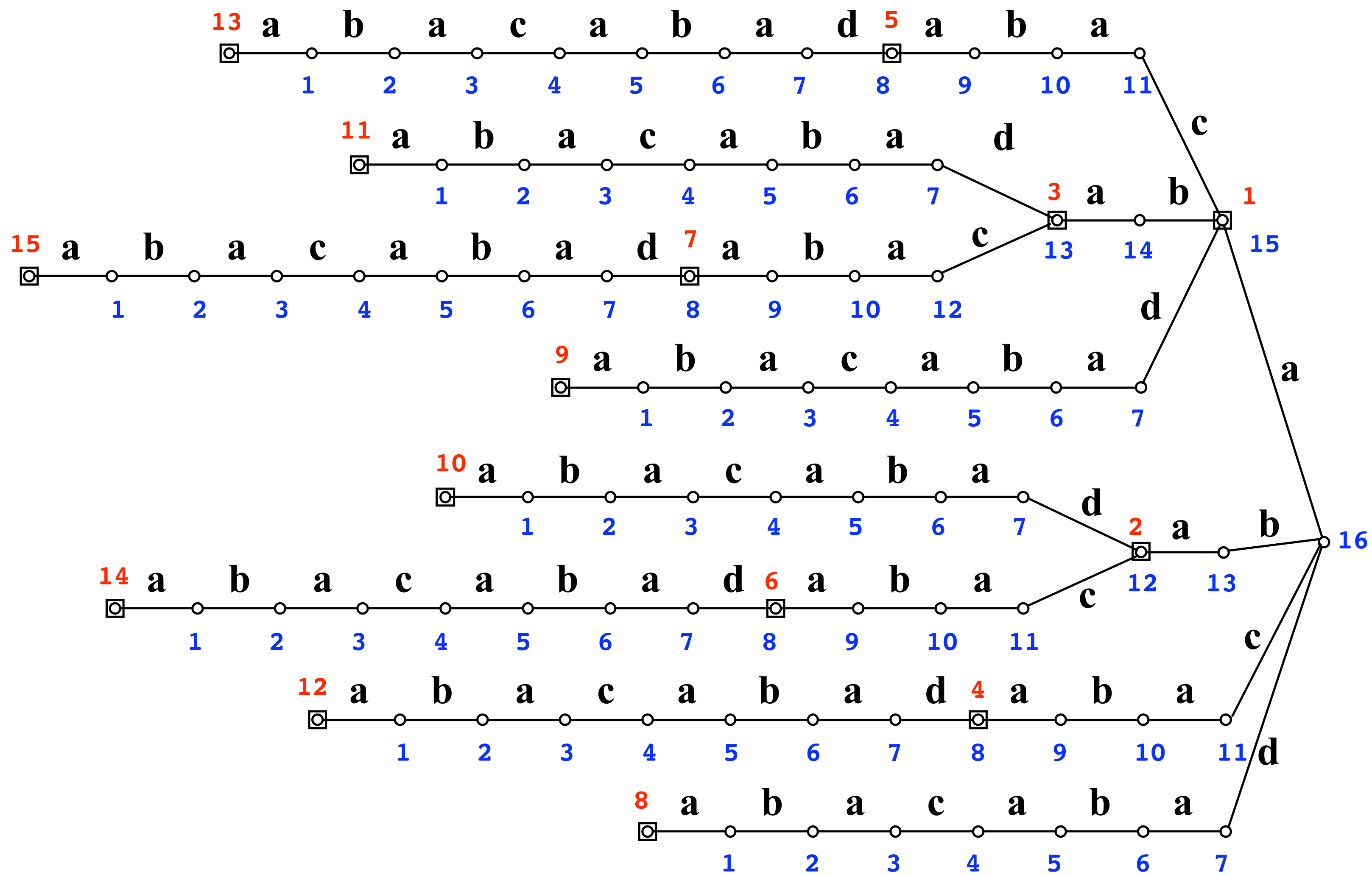
Need 7

7 a b c 4 a b c 1 a

8 a b c 5 a b c 2 a b

9 a b c 6 a b c 3 a b c

10 a b c a b c a b c d


13 a b a c a b a d 5 a b a
1 2 3 4 5 6 7 8 9 10 11
c
11 a b a c a b a d
1 2 3 4 5 6 7
3 a b 1
13 14 15
15 a b a c a b a d 7 a b a c
1 2 3 4 5 6 7 8 9 10 12
d
9 a b a c a b a
1 2 3 4 5 6 7
a
10 a b a c a b a
1 2 3 4 5 6 7
d 2 a b 16
12 13
14 a b a c a b a d 6 a b a c
1 2 3 4 5 6 7 8 9 10 11
c
12 a b a c a b a d 4 a b a
1 2 3 4 5 6 7 8 9 10 11
d
8 a b a c a b a
1 2 3 4 5 6 7

A *Really* Silly Idea

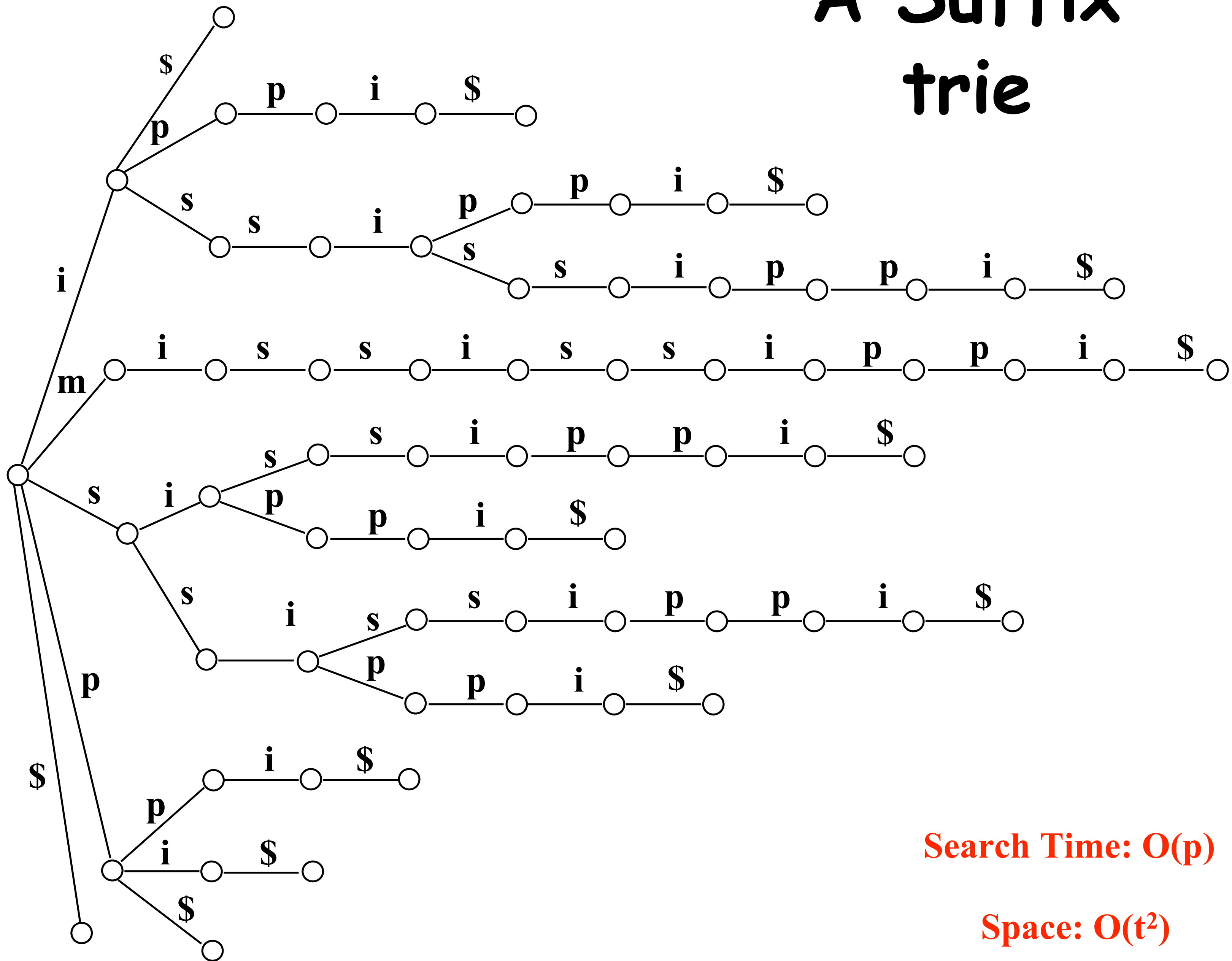
Index by all $\binom{n+1}{2}$ *substrings* of the text.

***Observe*: Every substring of a string is a prefix of some suffix of the string. So use a digital search tree to index on the suffixes of the text.**

A Corpus

mississippi

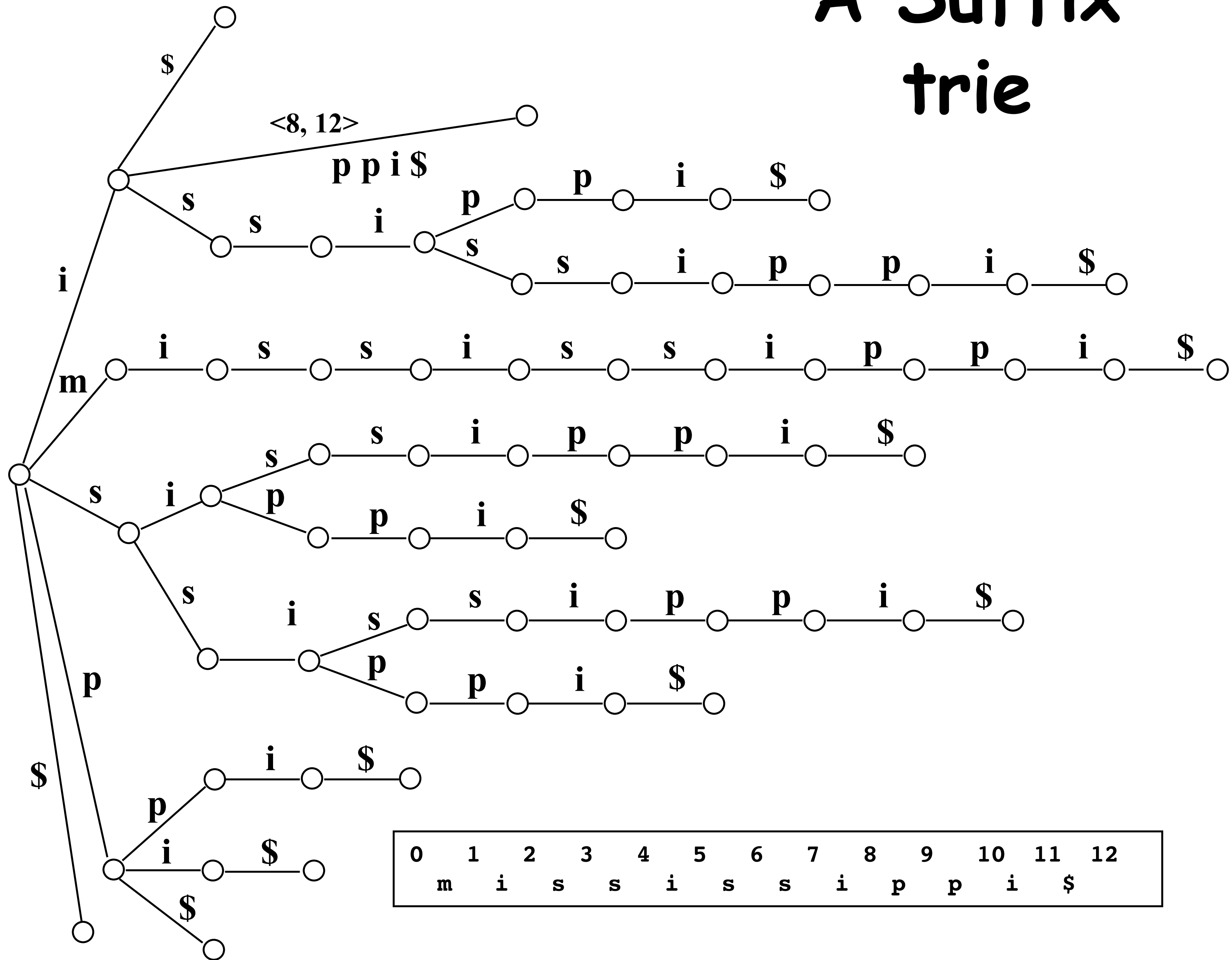
A Suffix trie



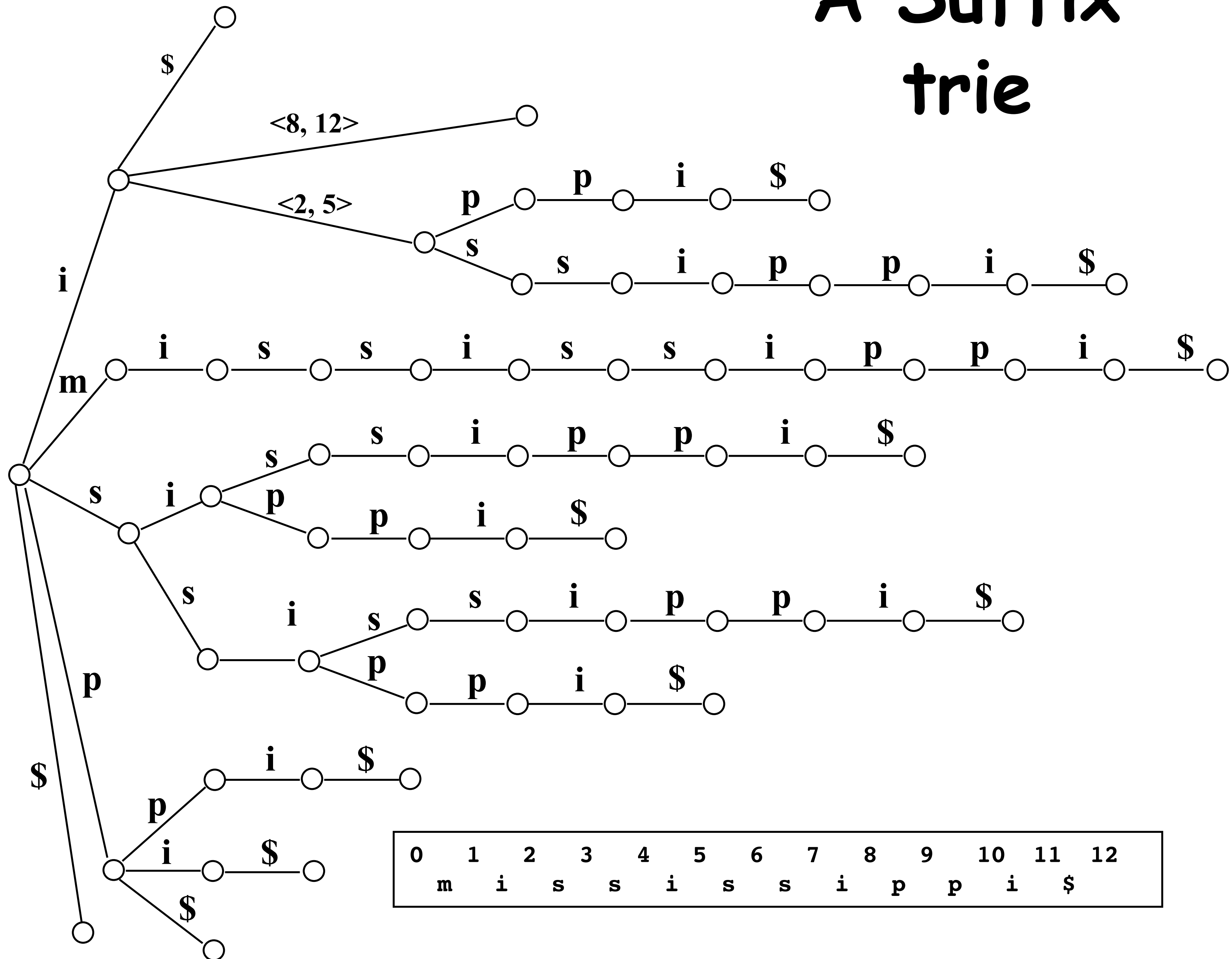
Search Time: O(p)

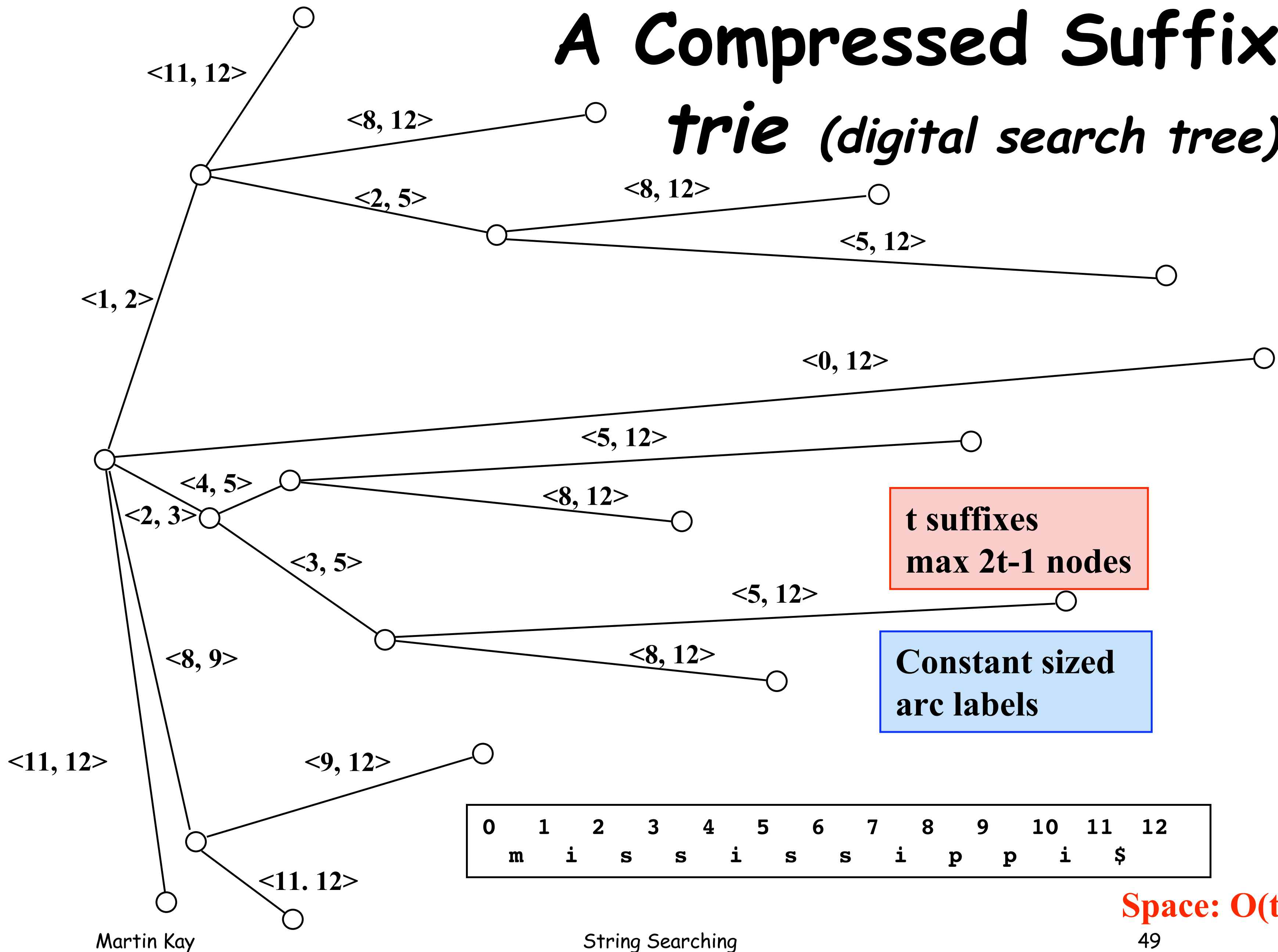
Space: $O(t^2)$

A Suffix trie



A Suffix trie

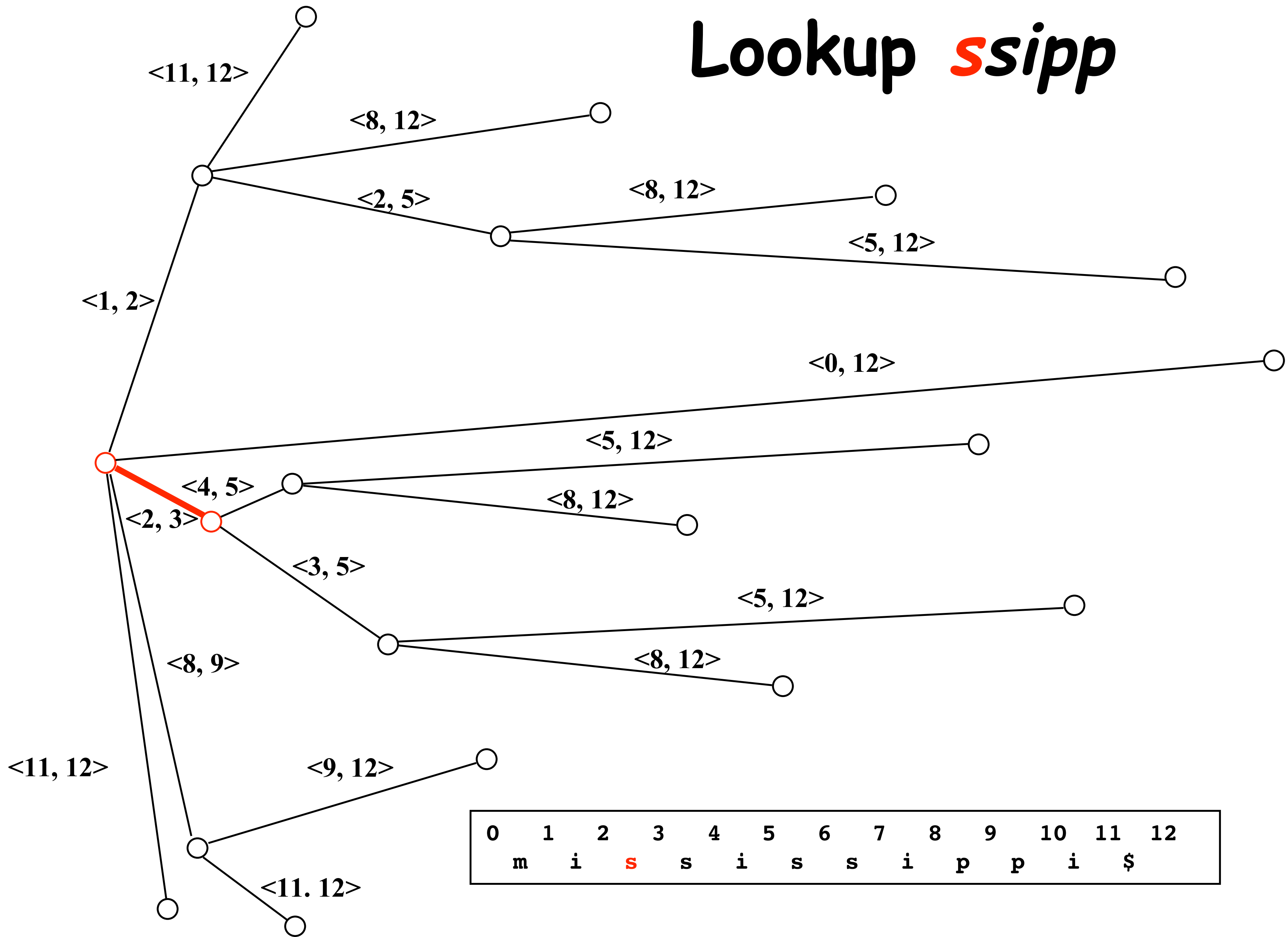


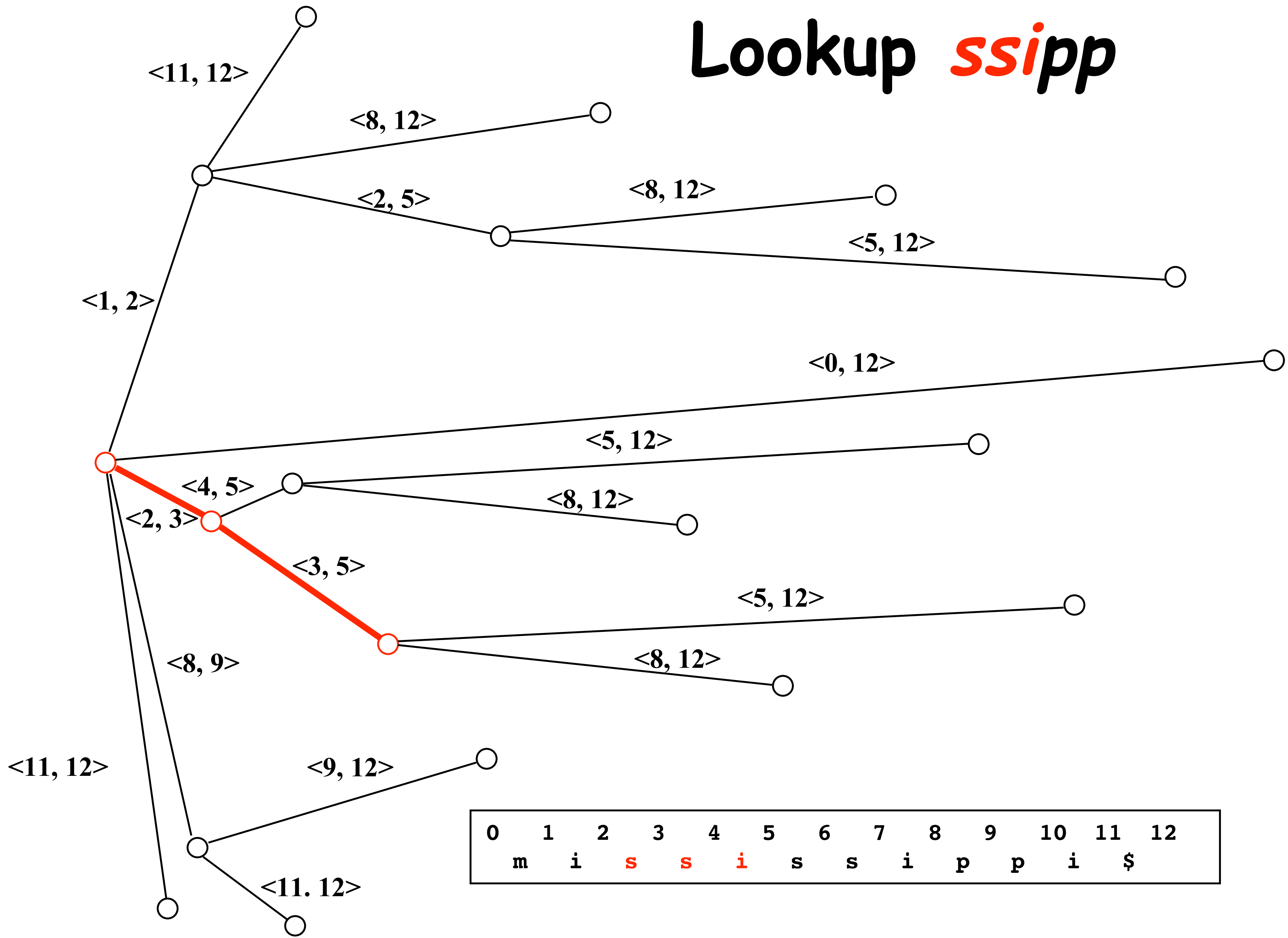

A Compressed Suffix
trie (digital search tree)
<11, 12>
<8, 12>
<2, 5>
<8, 12>
<5, 12>
<1, 2>
<0, 12>
<5, 12>
<4, 5>
<2, 3>
<8, 12>
<3, 5>
<5, 12>
<8, 9>
<8, 12>
<11, 12>
<9, 12>
<11. 12>
t suffixes
max 2t-1 nodes
Constant sized
arc labels
0 1 2 3 4 5 6 7 8 9 10 11 12
m i s s i s s i p p i $
Space: O(t)

Lookup *ssipp*

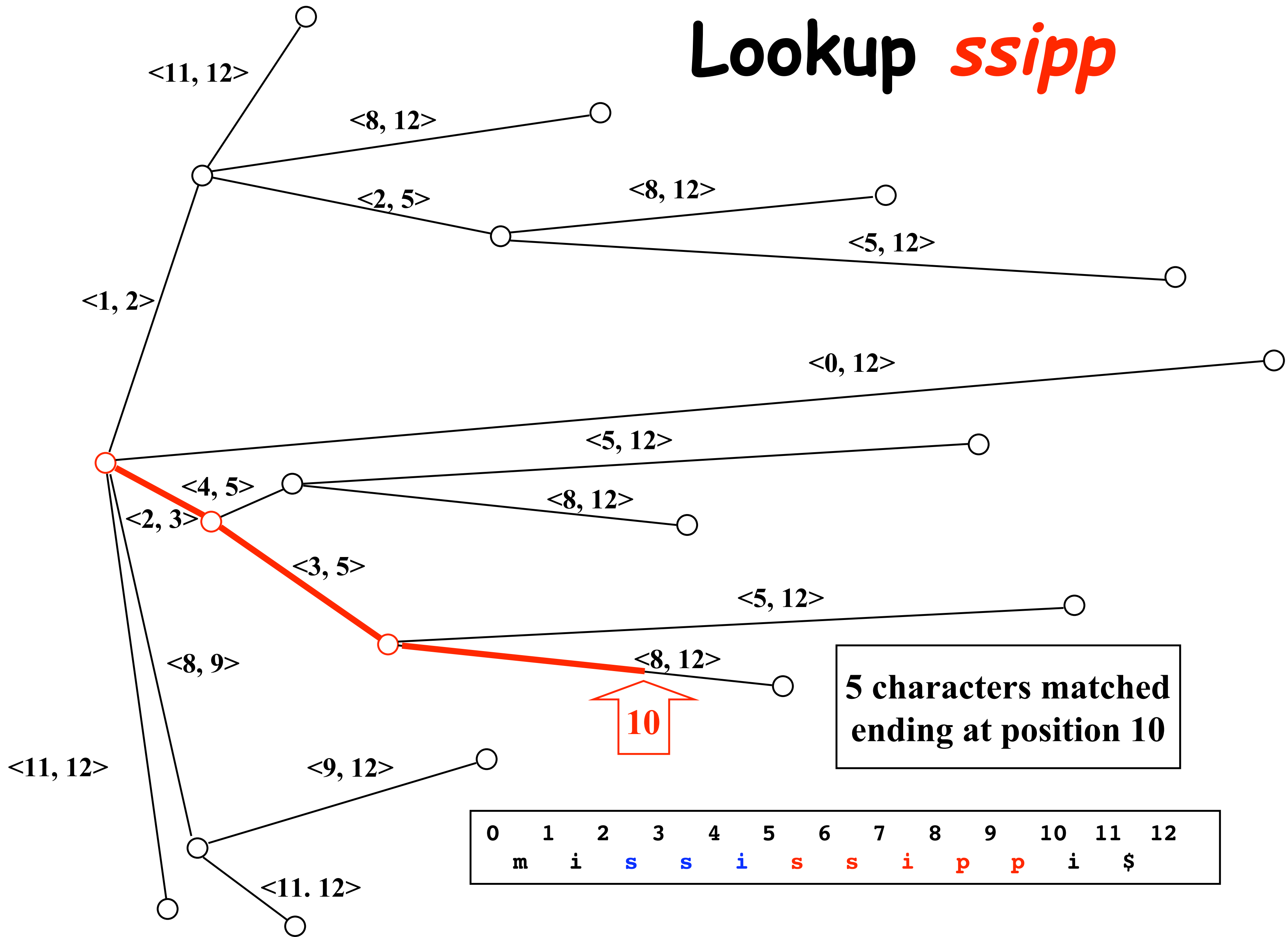
<11, 12>
<8, 12>
<2, 5>
<8, 12>
<5, 12>
<1, 2>
<0, 12>
<5, 12>
<4, 5>
<8, 12>
<2, 3>
<3, 5>
<5, 12>
<8, 9>
<8, 12>
<11, 12>
<9, 12>
<11. 12>

0	1	2	3	4	5	6	7	8	9	10	11	12
m	i	s	s	i	s	s	i	p	p	i	$	

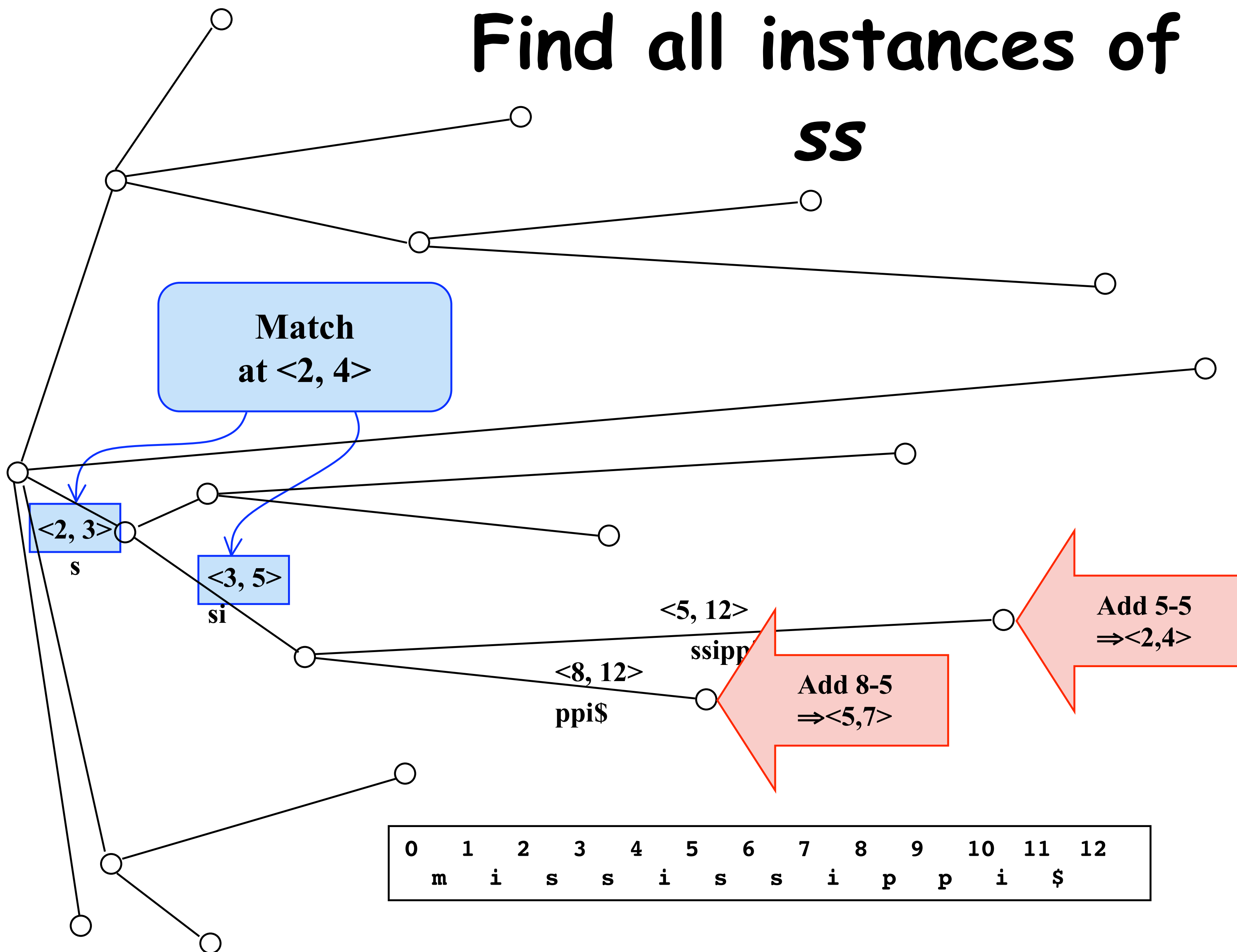

Lookup ssipp
<11, 12>
<8, 12>
<2, 5>
<8, 12>
<5, 12>
<1, 2>
<0, 12>
<5, 12>
<4, 5>
<8, 12>
<2, 3>
<3, 5>
<5, 12>
<8, 9>
<8, 12>
<11, 12>
<9, 12>
<11. 12>
0 1 2 3 4 5 6 7 8 9 10 11 12
m i s s i s s i p p i $

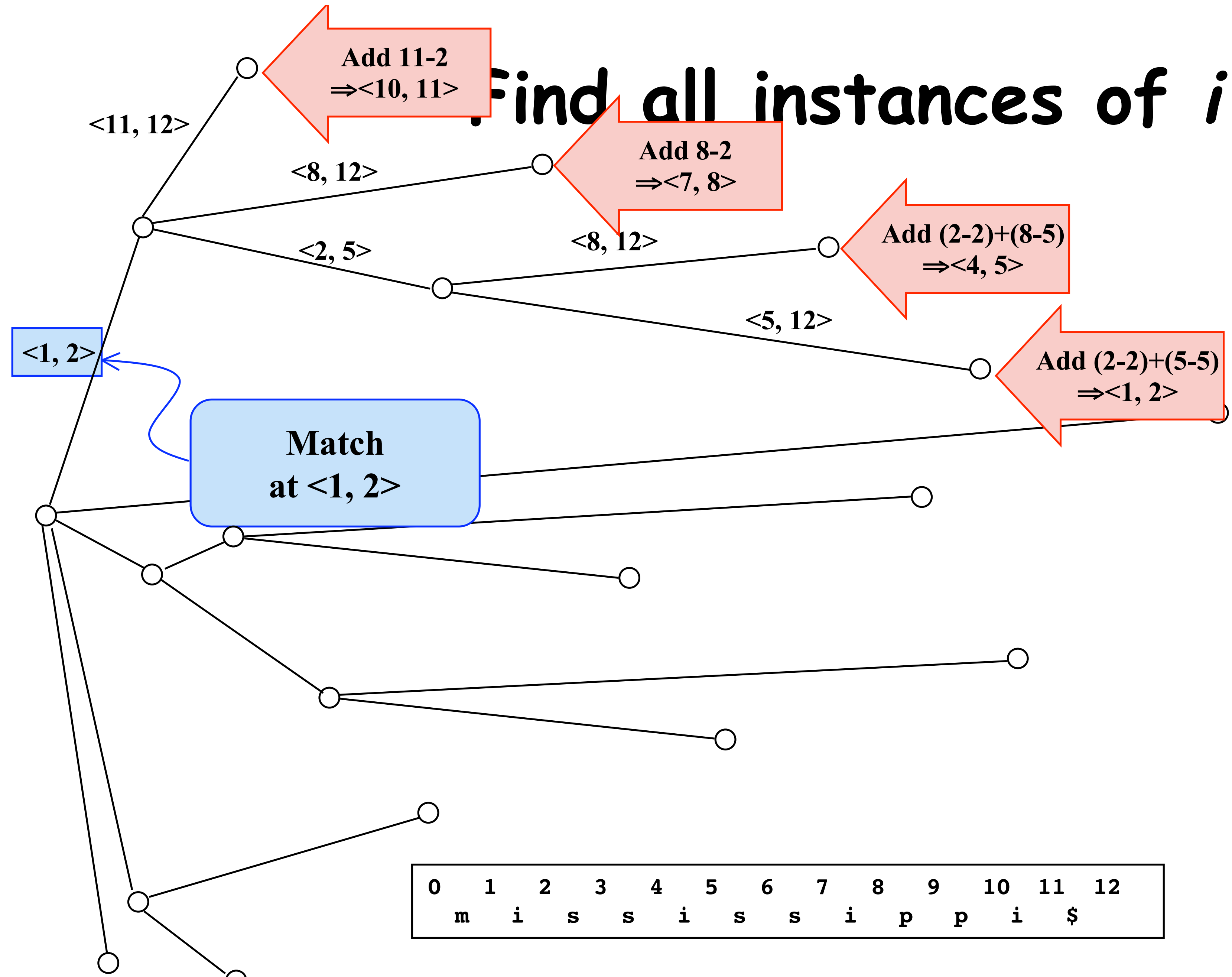

Lookup ssipp
<11, 12>
<8, 12>
<2, 5>
<8, 12>
<5, 12>
<1, 2>
<0, 12>
<5, 12>
<4, 5>
<8, 12>
<2, 3>
<3, 5>
<5, 12>
<8, 9>
<8, 12>
<11, 12>
<9, 12>
<11. 12>
0 1 2 3 4 5 6 7 8 9 10 11 12
m i s s i s s i p p i $

Lookup *ssipp*



Find all instances of *ss*




ind all instances of *i*
Add 11-2
⇒<10, 11>
<11, 12>
<8, 12>
Add 8-2
⇒<7, 8>
<2, 5>
<8, 12>
Add (2-2)+(8-5)
⇒<4, 5>
<5, 12>
Add (2-2)+(5-5)
⇒<1, 2>
<1, 2>
Match
at <1, 2>
0 1 2 3 4 5 6 7 8 9 10 11 12
m i s s i s s i p p i $

Observe:

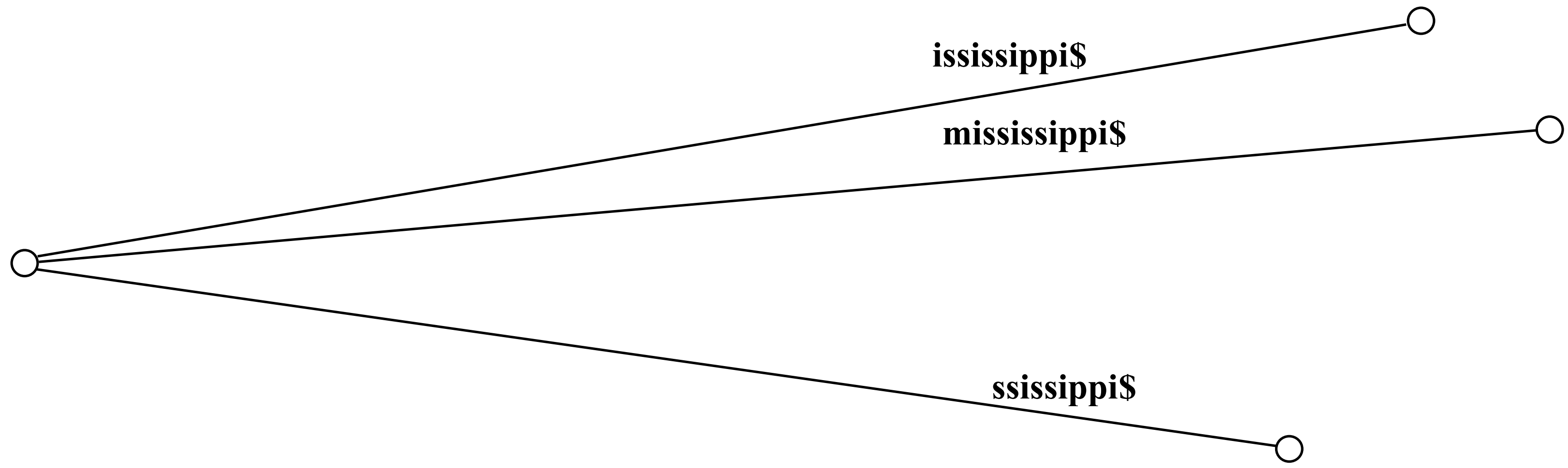
A (sub)tree with *n* terminals contains a total of at most 2*n*-1 nodes. Therefore finding all occurrences, once the first has been found requires, at most, that 2 nodes be visited for each hit.

Building the tree

Simple Labeling

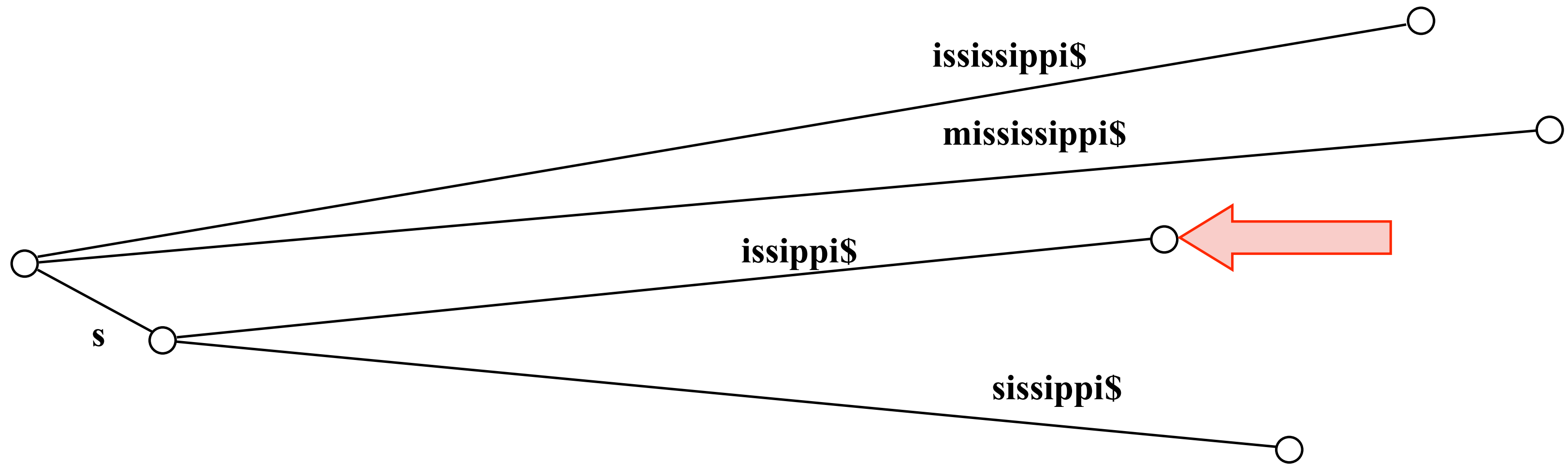
$
ppi$
ssi
ppi$
ssippi$
i
mississippi$
ssippi$
i
ppi$
s
si
ssippi$
ppi$
p
$
pi$
$

Building



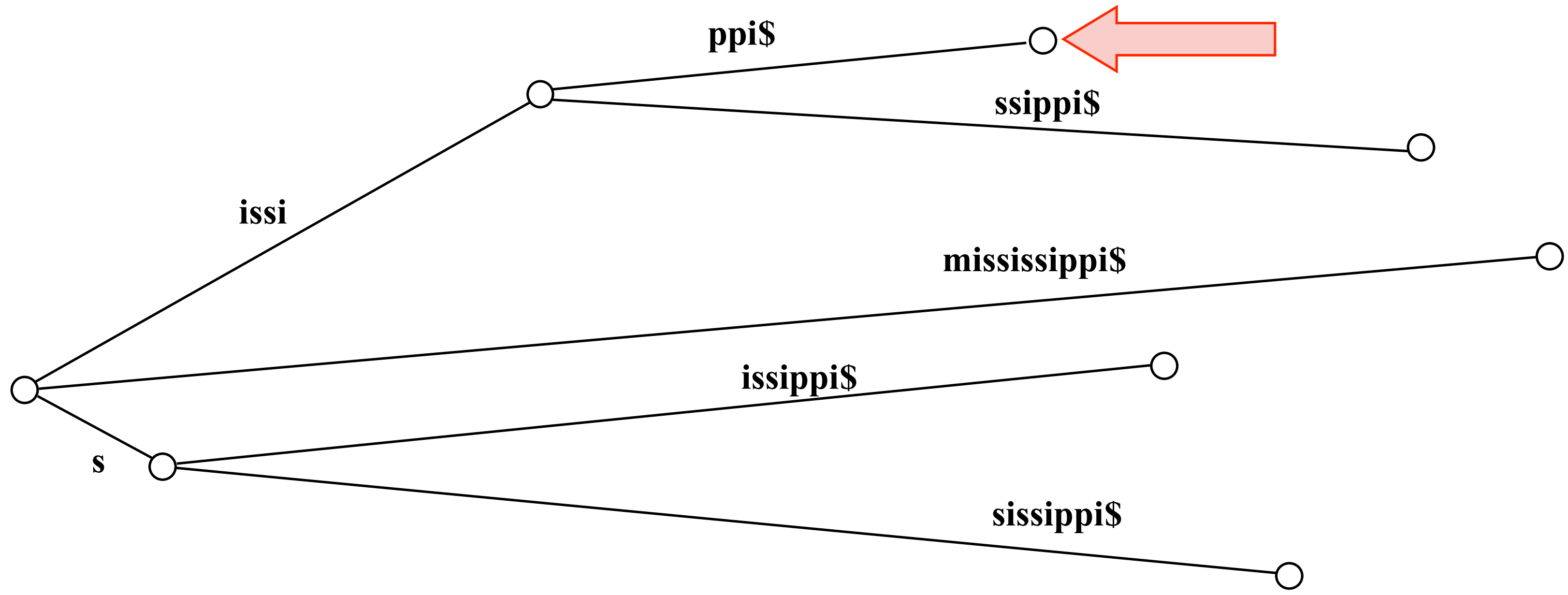
m i	s s i s s i p p i $

Building



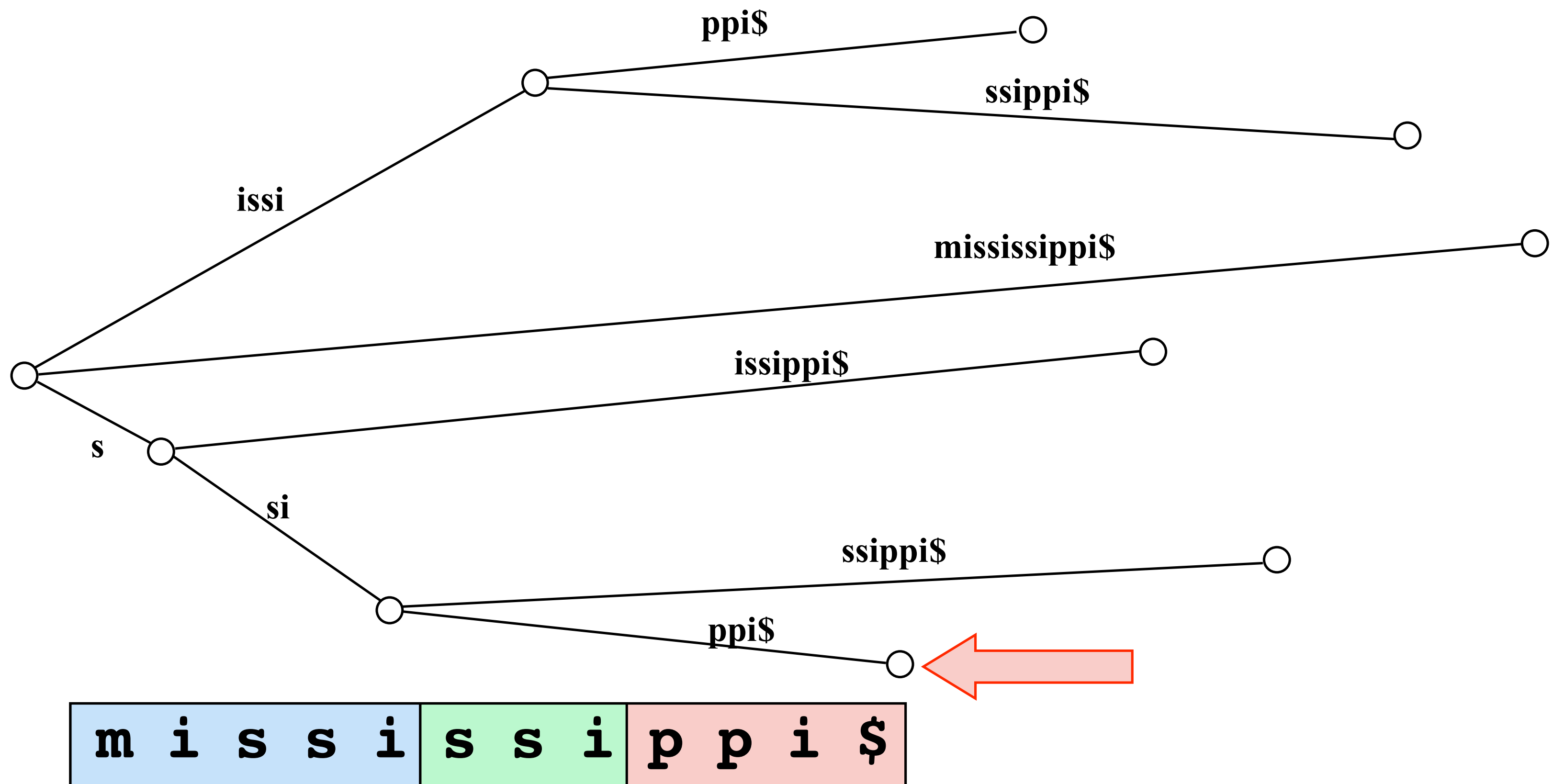
m i s s i s s i p p i $

Building

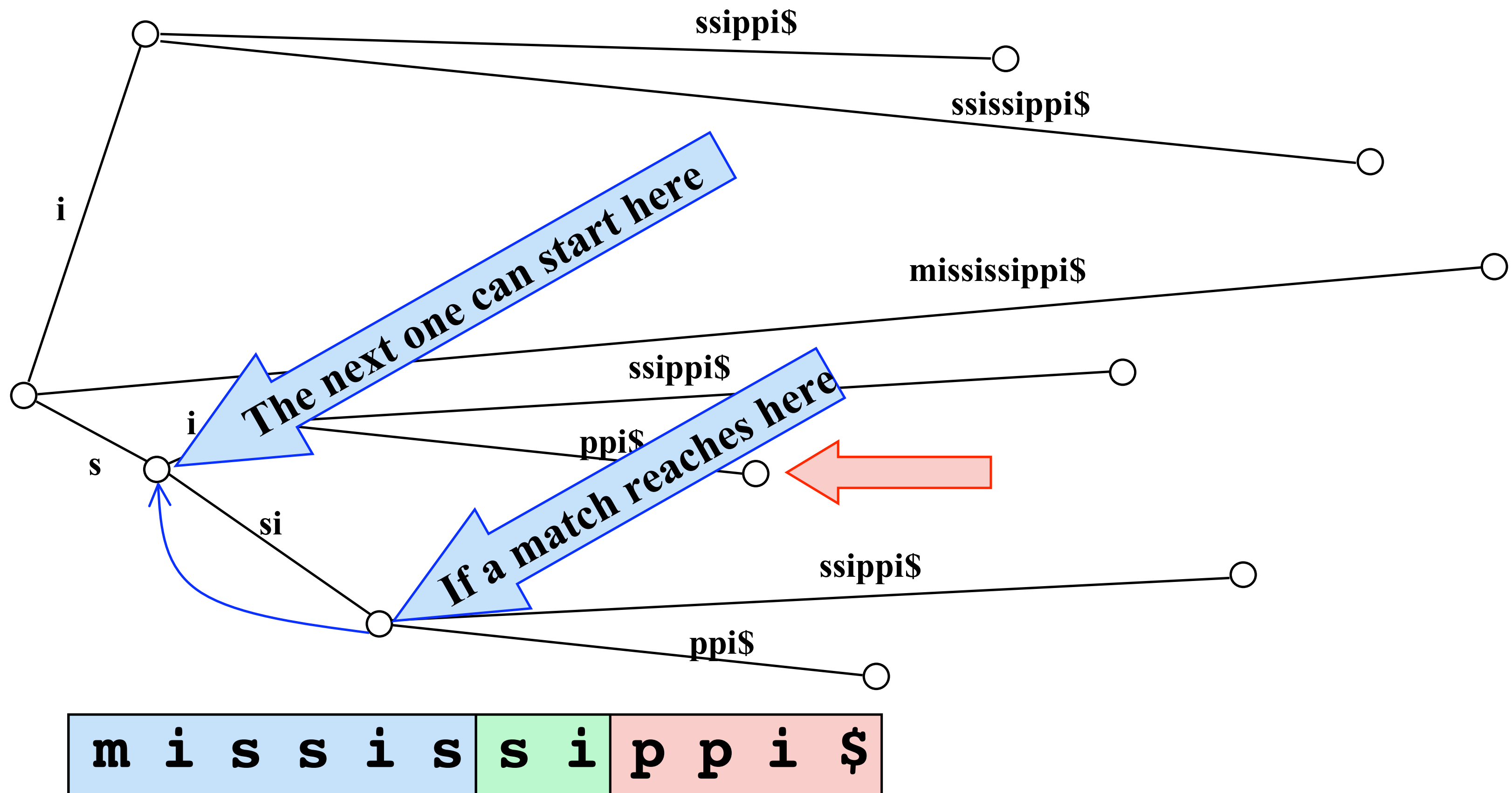


m	i	s	s	i	s	s	i	p	p	i	$

Building



Building

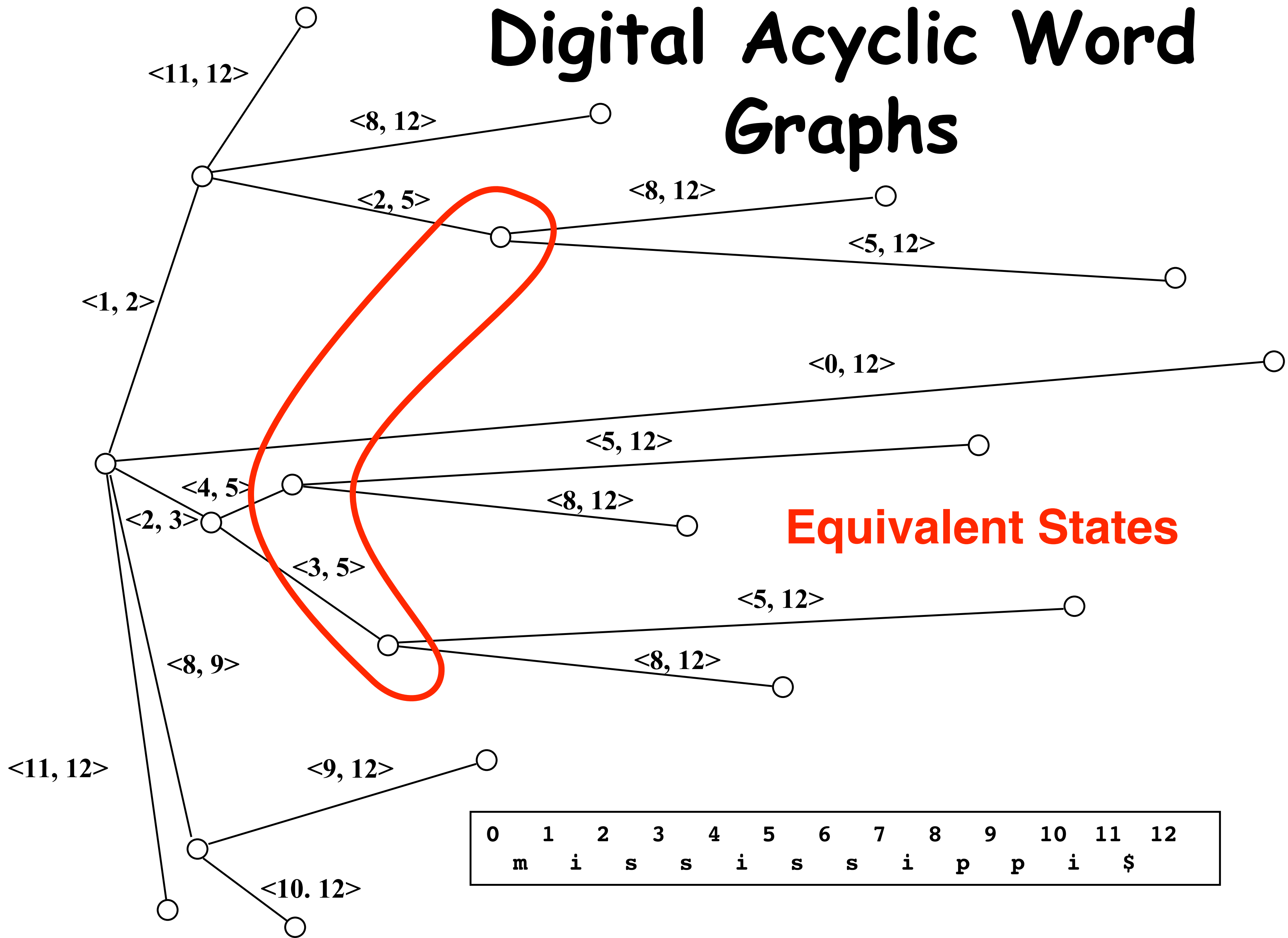


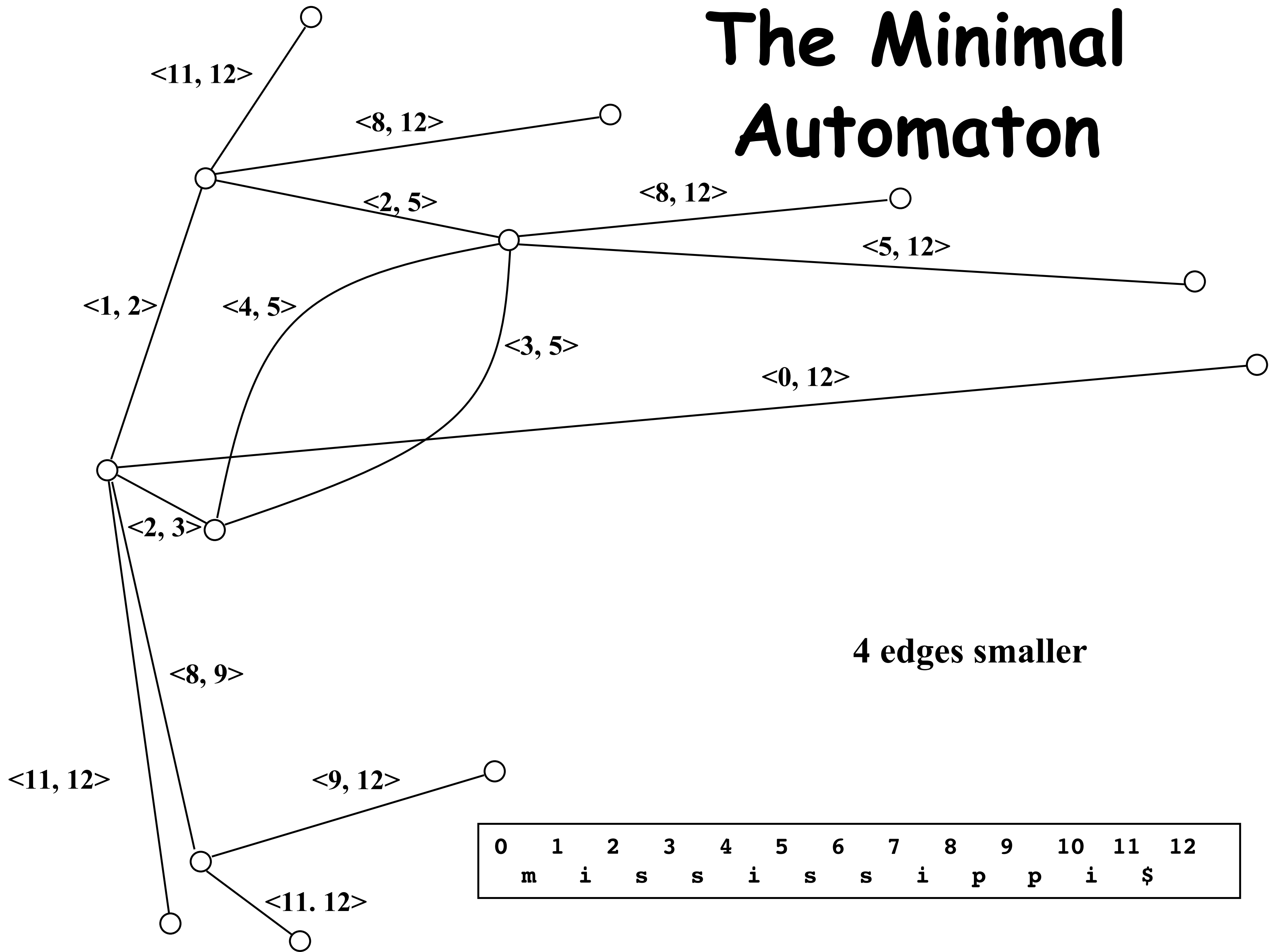
Building

ppi$
ssippi$
ssissippi$
i
mississippi$
ssippi$
i
ppi$
s
si
ssippi$
ppi$

m	i	s	s	i	s	s	i	p	p	i	$

Digital Acyclic Word Graphs




The Minimal Automaton
<11, 12>
<8, 12>
<2, 5>
<8, 12>
<5, 12>
<1, 2>
<4, 5>
<3, 5>
<0, 12>
<2, 3>
4 edges smaller
<8, 9>
<11, 12>
<9, 12>
<11. 12>
0 1 2 3 4 5 6 7 8 9 10 11 12
m i s s i s s i p p i $

Simple Boyer-Moore

```
def search(text)
   loc=@pattern_length-1
   while loc<text.length
     found = *match(text, loc)
     if found[0] == @pattern_length
       yield loc-found.shift+1
     end
     loc+=@pattern_length-found[0]
   end
 end
```

`found` is a list of prefix lengths

If the first member of the list is the length of the pattern, complete match has been found

Move the pattern just far enough to the right to complete a match

Simple Boyer-Moore

```
def match(text, loc)
    i=loc
    node=@root
    ret=[0]
    while node=node[text[i]]
      i-=1
      ret.unshift(loc-i) if node.final
    end
    return ret
  end
```