Exact String Matching

CS 4390/5390

Content mainly Chapters 1 and 5-7 from "Algorithms on Strings, Trees, and Sequences" by Dan Gusfield

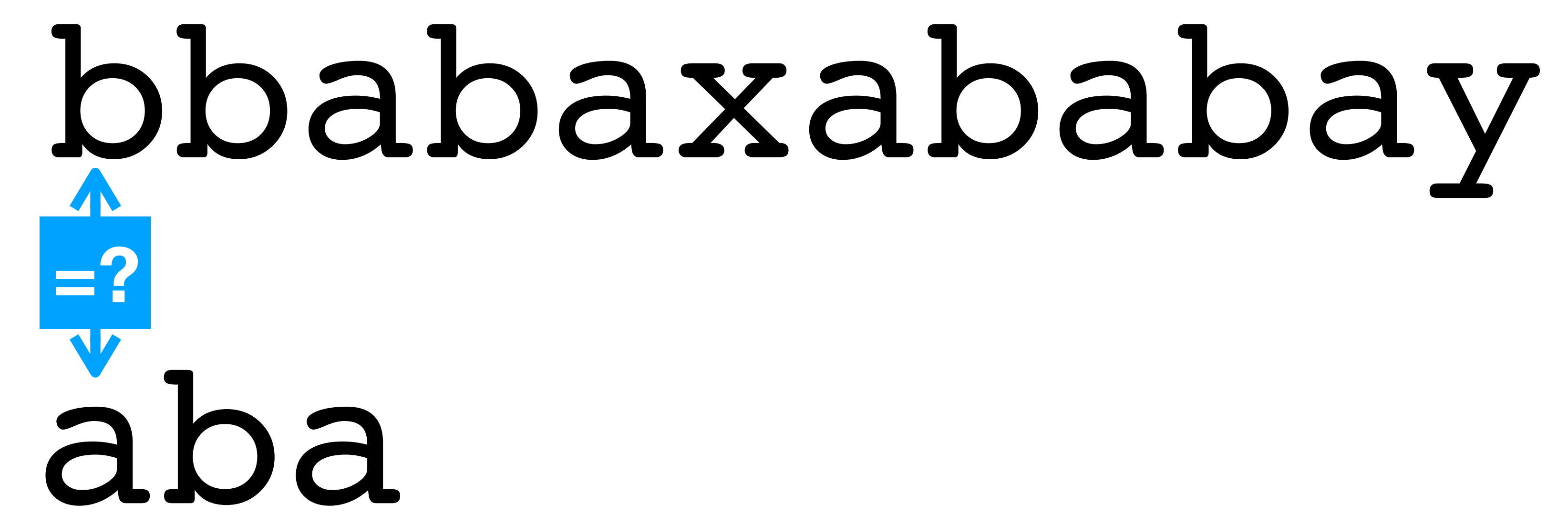
Exact String Matching

- Given string *P*, called the pattern, and a longer string *T*, called the text, the **exact matching** problem is to find all occurrences, if any, of *P* in *T*.
- Example:
  - *P* = "`aba`", *T* = "`bbabaxababay`"
  - *P* occurs in *T* at positions: 3, 7, & 9
  - Note, that 2 occurrences overlap

The naive method

```
bbabaxababay
aba
```

The naive method



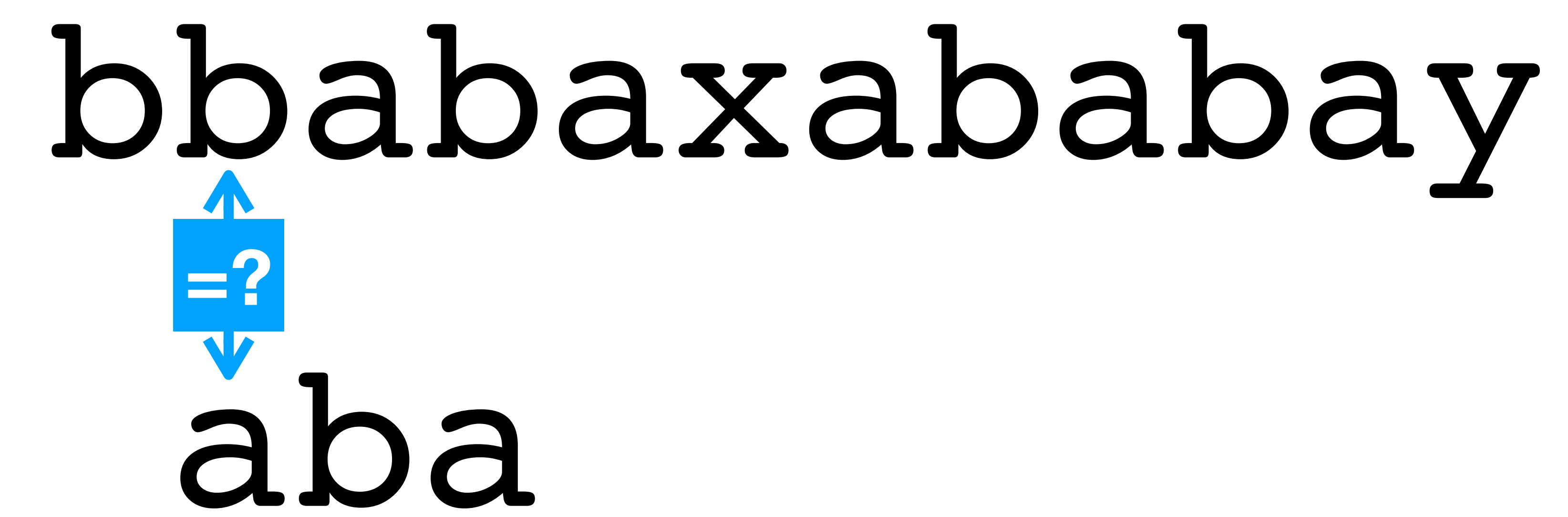
The naive method

```
bbabaxababay
✘
aba
```

The naive method

```
bbabaxababay
 aba
```

The naive method



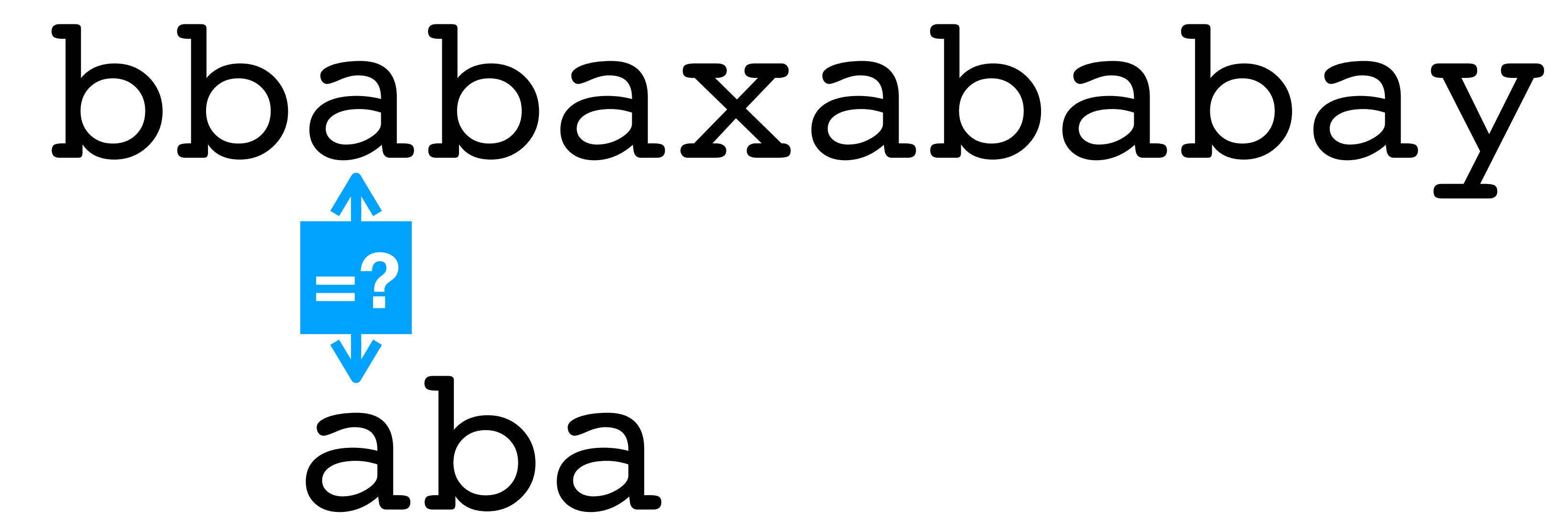
The naive method

```
bbabaxababay
 ✗
 aba
```

The naive method

```
bbabaxababay
  aba
```

The naive method

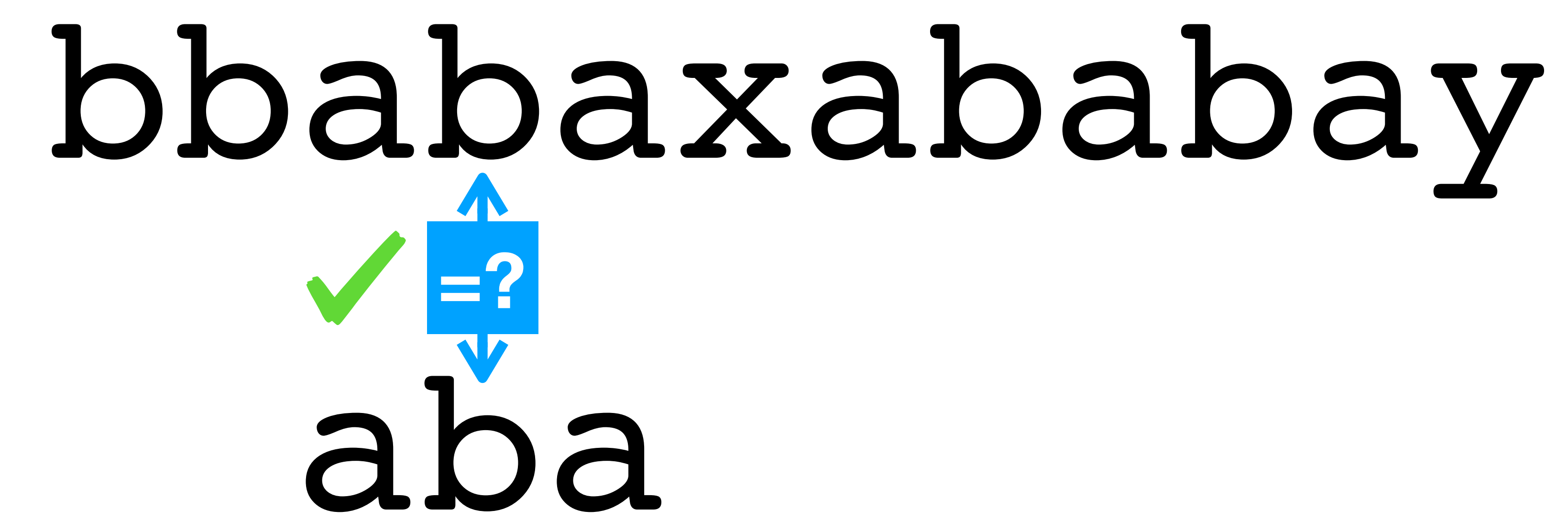


The naive method

```
bbabaxababay
```

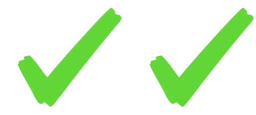
```
   aba
```

The naive method



The naive method

bbabaxababay



aba

The naive method

```
bbabaxababay
  ✓✓=?
  aba
```

The naive method

```
bbabaxababay
  ✓✓✓
  aba
```

The naive method

bbabaxababay

✓✓✓

aba

The naive method

```
bbabaxababay
    aba
```

The naive method

bbabaxababay

=?

aba

The naive method

bbabaxababay

✗

aba

The naive method

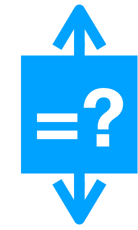
Another example

```
aaaaaaaaaaaa
aaa
```

The naive method

Another example

aaaaaaaaaaaa



aaa

The naive method

Another example

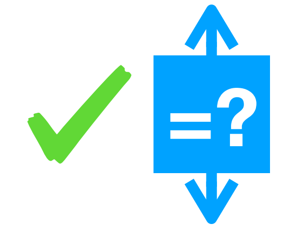
```
aaaaaaaaaaaa
```

```
aaa
```

The naive method

Another example

aaaaaaaaaaaa



aaa

The naive method

Another example

```
aaaaaaaaaaaa
✓✓
aaa
```

The naive method

Another example

aaaaaaaaaaaa

✓ ✓ =?

aaa

The naive method

Another example

```
aaaaaaaaaaaa
✓✓✓
aaa
```

The naive method

Another example

```
aaaaaaaaaaaa
✓✓✓
aaa
```

The naive method

Another example

aaaaaaaaaaaa

aaa

The naive method

Another example

aaaaaaaaaaaaa

=?

aaa

The naive method

Another example

aaaaaaaaaaaa

aaa

The naive method

Another example

aaaaaaaaaaaaa

=?

aaa

The naive method

Another example

aaaaaaaaaaaaa

✓✓

aaa

The naive method

Another example

aaaaaaaaaaaaa

=?

aaa

The naive method

Another example

```
aaaaaaaaaaaa
 ✓✓✓
 aaa
```

The naive method

Another example

```
aaaaaaaaaaaa
 ✓✓✓
 aaa
```

How many comparisons would the naive algorithm do on this example?

The naive method

Another example

aaaaaaaaaaaaa

✓✓✓

aaa

How many comparisons would the naive algorithm do on this example?

What did we already know about this comparison before we started?

Example of faster methods

```
xabxyabxyabxz
abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
  abxyabxz
    abxyabxz
      abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
   abxyabxz
    abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
   abxyabxz
    abxyabxz
     abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
   abxyabxz
    abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
   abxyabxz
    abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
abxyabxz
 abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
  abxyabxz
    abxyabxz
      abxyabxz
        abxyabxz
          abxyabxz
```

```
xabxyabxyabxz
abxyabxz
  abxyabxz
          abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
   abxyabxz
    abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
abxyabxz
 abxyabxz
     abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
   abxyabxz
    abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
abxyabxz
 abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
abxyabxz
 abxyabxz
     abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
   abxyabxz
    abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
abxyabxz
 abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
abxyabxz
 abxyabxz
     abxyabxz
```

Example of faster methods

```
xabxyabxyabxz
abxyabxz
 abxyabxz
  abxyabxz
   abxyabxz
    abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
abxyabxz
 abxyabxz
     abxyabxz
```

```
xabxyabxyabxz
abxyabxz
 abxyabxz
     abxyabxz
```

If we knew ahead of time the comparisons we could skip (based on where in P the mismatches are), we could decrease the running time!

Preprocessing

Definition Given a string S and a position $i>1$, let $\boldsymbol{Z_i(S)}$ be the length of the longest substring if S that matches a prefix of S.

that is, $Z_i(S)=j$ if $S[1...j] = S[i...(i+j)]$ (and $S[1...(j+1)] \neq S[i...(i+j+1)]$)

for S = "`aabcaabxaaz`"

$Z_5(S) = 3$

$Z_6(S) = 1$

$Z_7(S) = Z_8(S) = 0$

$Z_9(S) = 2$

Preprocessing

Definition For any position $i>1$ where $Z_i>0$, the **Z-box** at i is defined by the interval starting at i and ending at position $i+Z_i-1$

Preprocessing

Definition For any position $i>1$ where $Z_i>0$, the **Z-box** at i is defined by the interval starting at i and ending at position $i+Z_i-1$

Definition For every $j>1$, $\mathbf{r_j}$ is the right most endpoint of the Z-boxes that begin at or before position j.

Or: r_j is the largest value $k+Z_k-1$ over all $0<k<=j$

Preprocessing

Definition For any position $i>1$ where $Z_i>0$, the **Z-box** at i is defined by the interval starting at i and ending at position $i+Z_i-1$

Definition For every $j>1$, $\boldsymbol{r_j}$ is the right most endpoint of the Z-boxes that begin at or before position j.

Or: r_j is the largest value $k+Z_k-1$ over all $0<k<=j$

Definition For every $j>1$, $\boldsymbol{l_j}$ is the left most position k such that $r_i = k+Z_k-1$.

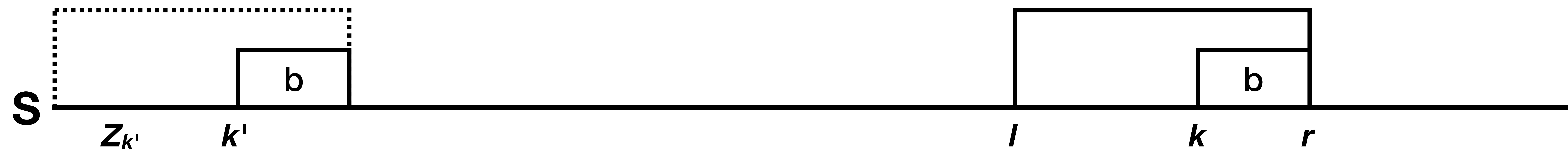
Preprocessing

Preprocessing

Preprocessing

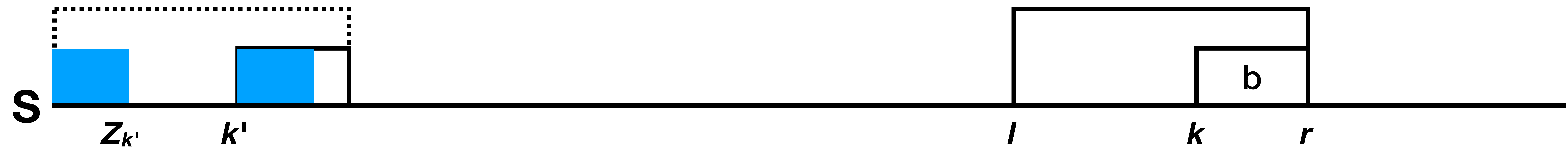
Preprocessing

Preprocessing



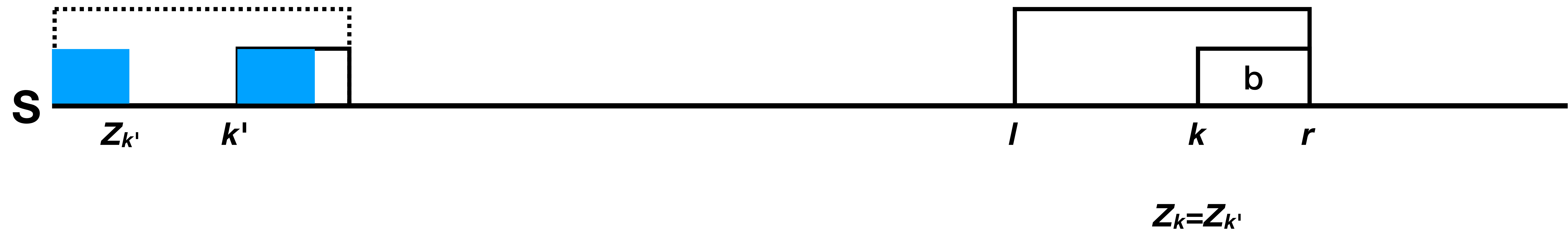
$Z_{k'} < |b|$

Preprocessing



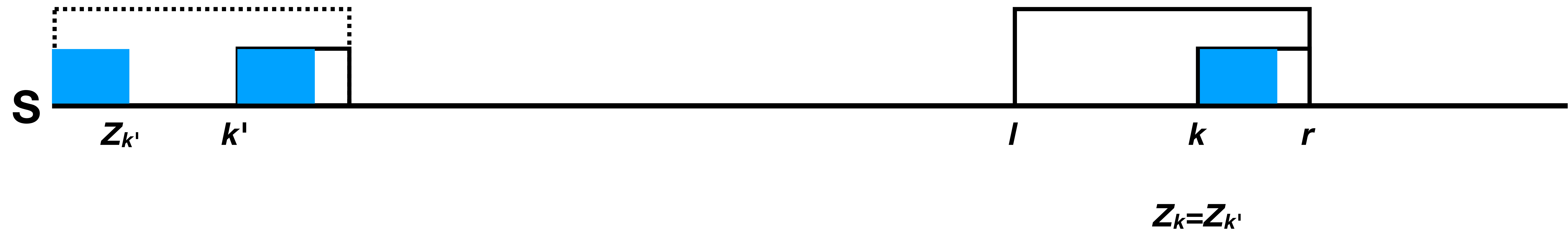
$Z_{k'} < |b|$

Preprocessing



$Z_{k'} < |b|$

Preprocessing

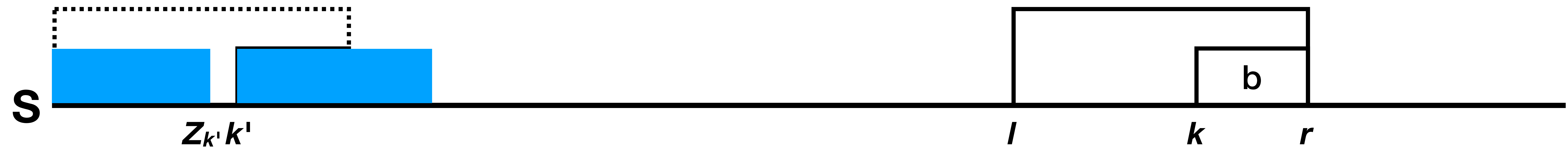


$Z_{k'} < |b|$

Preprocessing

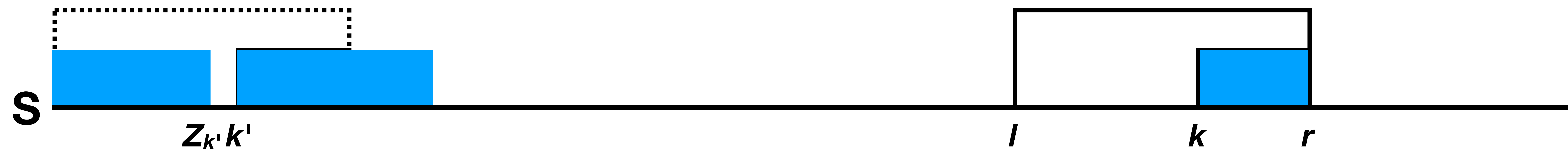
$Z_{k'} >= |b|$

Preprocessing



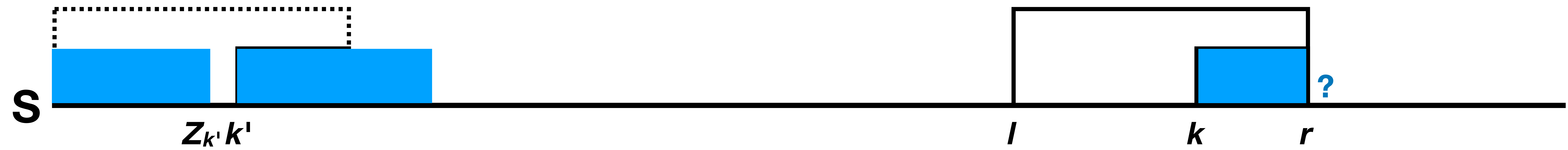
$Z_{k'} >= |b|$

Preprocessing



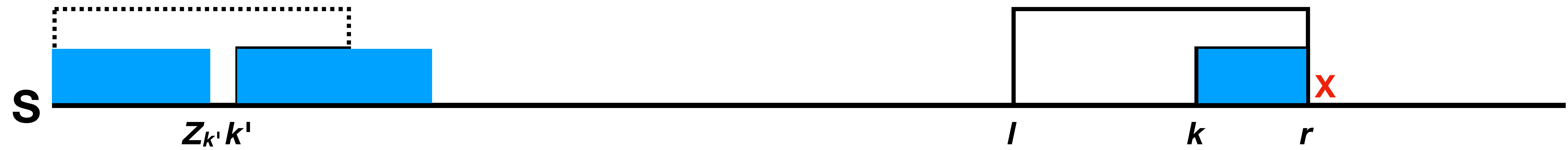
$Z_{k'} >= |b|$

Preprocessing



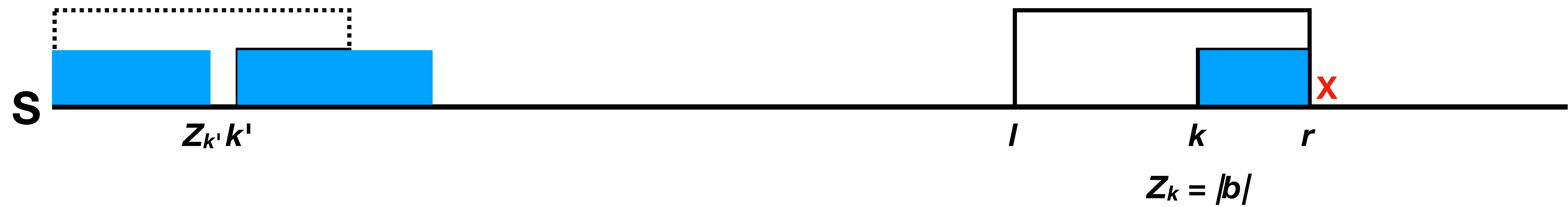
$Z_{k'} >= |b|$

Preprocessing



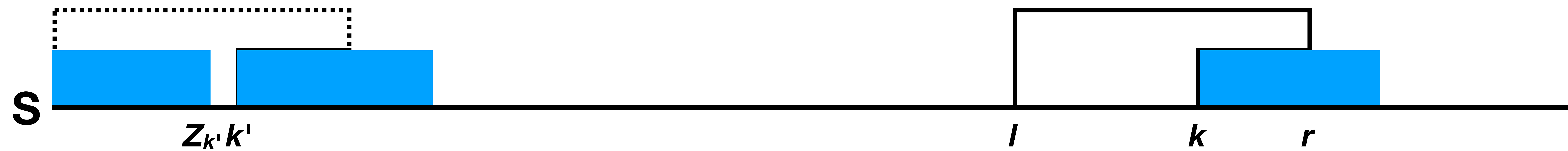
$Z_{k'} >= |b|$

Preprocessing



$Z_{k'} >= |b|$

Preprocessing



$Z_{k'} >= |b|$

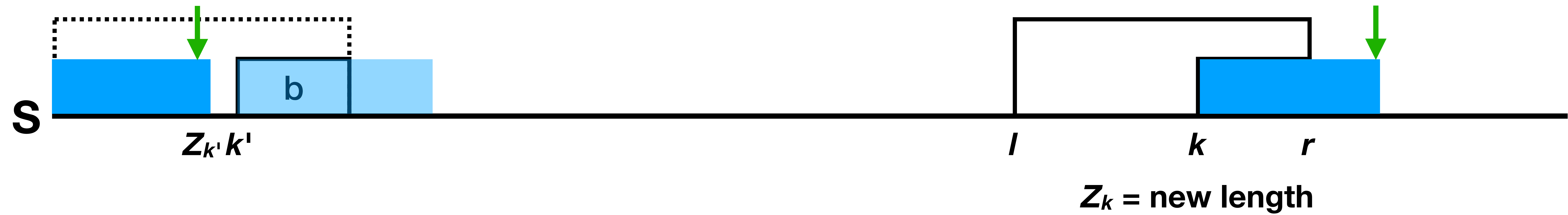
Preprocessing

$Z_{k'} >= |b|$

Preprocessing

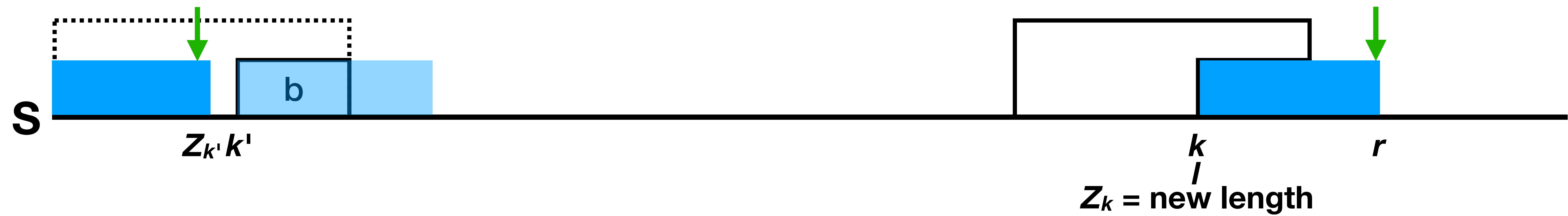
$Z_{k'} >= |b|$

Preprocessing



$Z_{k'} >= |b|$

Preprocessing



$Z_{k'} >= |b|$

Preprocessing

- Calculation of all Z_i can be done in $O(n)$ time (where $|S| = n$)

Linear-time exact sequence matching

Linear-time exact sequence matching

- Construct a new sequence $S = P\$T$ where $\$$ is a letter thats not in either of the two strings

Linear-time exact sequence matching

- Construct a new sequence $S = P\$T$ where $\$$ is a letter thats not in either of the two strings
- What happens when we compute all of the Z values for S?

Linear-time exact sequence matching

- Construct a new sequence $S = P\$T$ where \$ is a letter thats not in either of the two strings
- What happens when we compute all of the Z values for S?
- Do we need to save $Z_{(n+5)}$? (where $|P| = n$)

Boyer-Moore (at a high level)

- Worst-case running time is still $O(m+n)$, but in practice it is sub-linear
- Re-think the problem and look at suffixes of the pattern, then use some preprocessing information to skip the pattern ahead by more than 1

Boyer-Moore (at a high level)

- The bad character rule
  - when matching (from the right) and a mismatch is found (say *x* in *T* ≠ *y* in *P*)
  - shift the pattern so that the next instance of *x* in *P* that is to the left of the current position is at the current position in *T*

```
xabxyabxyabxz
 abxyabxx
```

Boyer-Moore (at a high level)

- The bad character rule
  - when matching (from the right) and a mismatch is found (say x in $T \neq y$ in P)
  - shift the pattern so that the next instance of x in P that is to the left of the current position is at the current position in T

```
xabxyabxyabxz
 abxyabxz
```

Boyer-Moore (at a high level)

- The bad character rule
  - when matching (from the right) and a mismatch is found (say x in $T \neq y$ in P)
  - shift the pattern so that the next instance of x in P that is to the left of the current position is at the current position in T

```
xabxyabxyabxz
    abxyabxz
```

Boyer-Moore (at a high level)

- The bad character rule
  - when matching (from the right) and a mismatch is found (say x in $T \neq y$ in P)
  - shift the pattern so that the next instance of x in P that is to the left of the current position is at the current position in T

```
xabxyabxyabxz
        abxyabxz
```

Does this skip any instances of *P* in *T*?

Boyer-Moore (at a high level)

- The good suffix rule
  - suppose a substring of *T*, *t* matches a suffix of *P* but a mismatch occurs at the next character
  - find the right most copy of *t* in *P* such that the preceding character is different
  - shift *P* so that this substring is matched with *t*

```
prstabstubabvqxrst
  qcabdabdab
```

Boyer-Moore (at a high level)

- The good suffix rule
  - suppose a substring of *T*, *t* matches a suffix of *P* but a mismatch occurs at the next character
  - find the right most copy of *t* in *P* such that the preceding character is different
  - shift *P* so that this substring is matched with *t*

```
prstabstubabvqxrst
  qcabdabdab
```

Boyer-Moore (at a high level)

- The good suffix rule
  - suppose a substring of *T*, *t* matches a suffix of *P* but a mismatch occurs at the next character
  - find the right most copy of *t* in *P* such that the preceding character is different
  - shift *P* so that this substring is matched with *t*

```
prstabstubabvqxrst
        qcabdabdab
```

Boyer-Moore (at a high level)

- The good suffix rule
  - suppose a substring of *T*, *t* matches a suffix of *P* but a mismatch occurs at the next character
  - find the right most copy of *t* in *P* such that the preceding character is different
  - shift *P* so that this substring is matched with *t*

```
prstabstubabvqxrst
          qcabdabdab
```

Does this skip any instances of *P* in *T*?

Other methods

- Knuth-Morris-Pratt (KMP)
  - able to work with *T* arriving **online**
  - running time is still $O(m+n)$
  - uses pattern suffixes (closer to suffix tree which we will see next)

- Aho-Corasick
  - efficiently finds all occurrences of a **set** of patterns
  - puts patterns into a tree to search all at once

- Karp-Rabin
  - uses bit operations in place of character comparisons
  - structure similar to on-hot-encodings to represent strings

Suffix Trees

CS 4390/5390

Sequence Search Problem

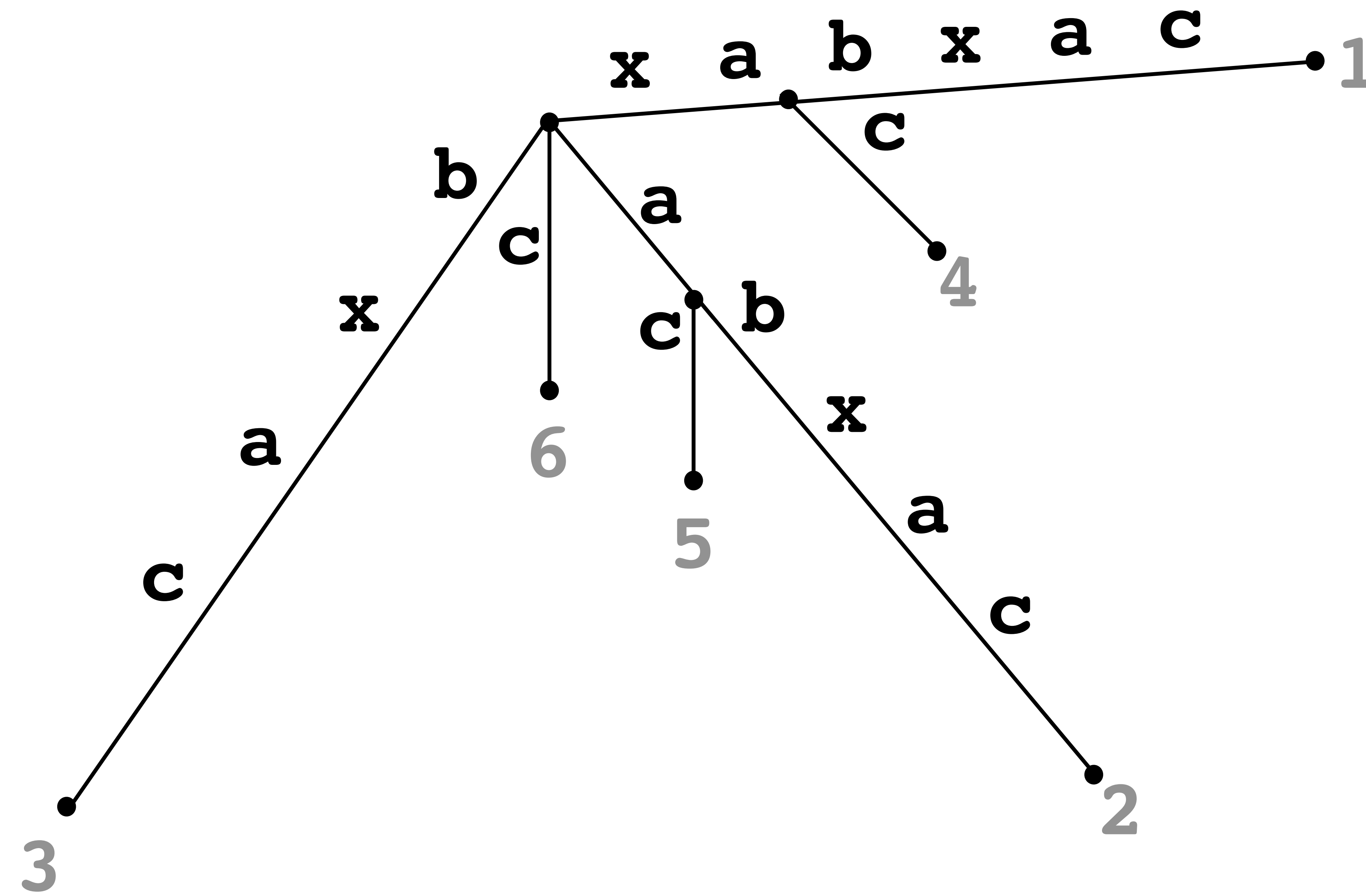
- Given a text *T* and a pattern *P* answer the question: does *P* exist as a substring in *T*?
- Example:
  - *P* = "`aba`", *T* = "`bbabaxababay`"
  - yes, *P* occurs in *T*
  - note, that there are multiple occurrences, but the question is binary

Sequence Search Problem

- We know that for $|P|=m$ & $|T|=n$, can be answered in $O(m+n)$ time
- What if there are k patterns?
  - using Boyer-Moore (or others) can be answered in $O(k(m+n))$ time
- What happens when $m=10^9$ (i.e. a human genome or an entire textbook)?

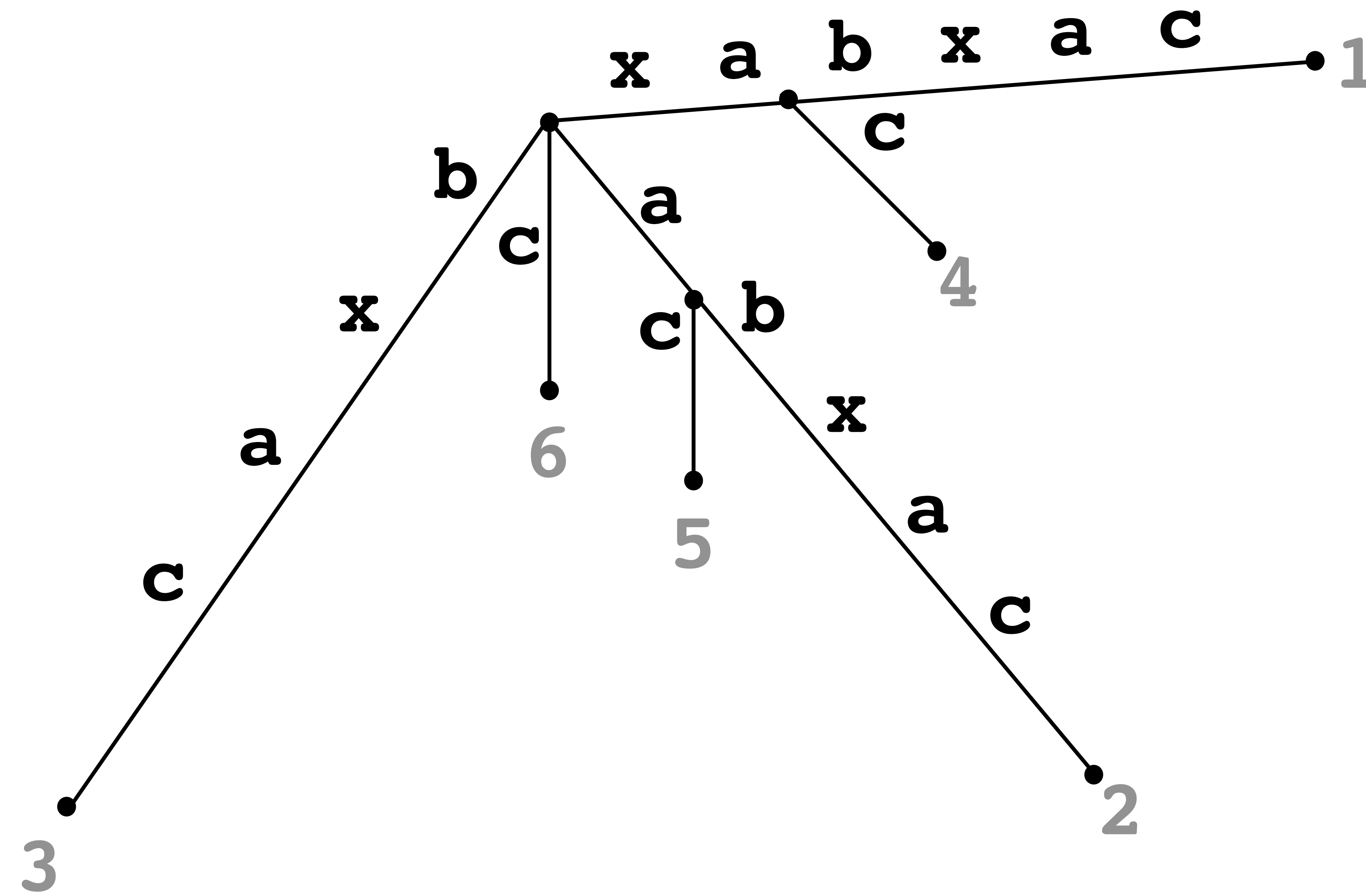
Suffix Tree

123456
xabxac



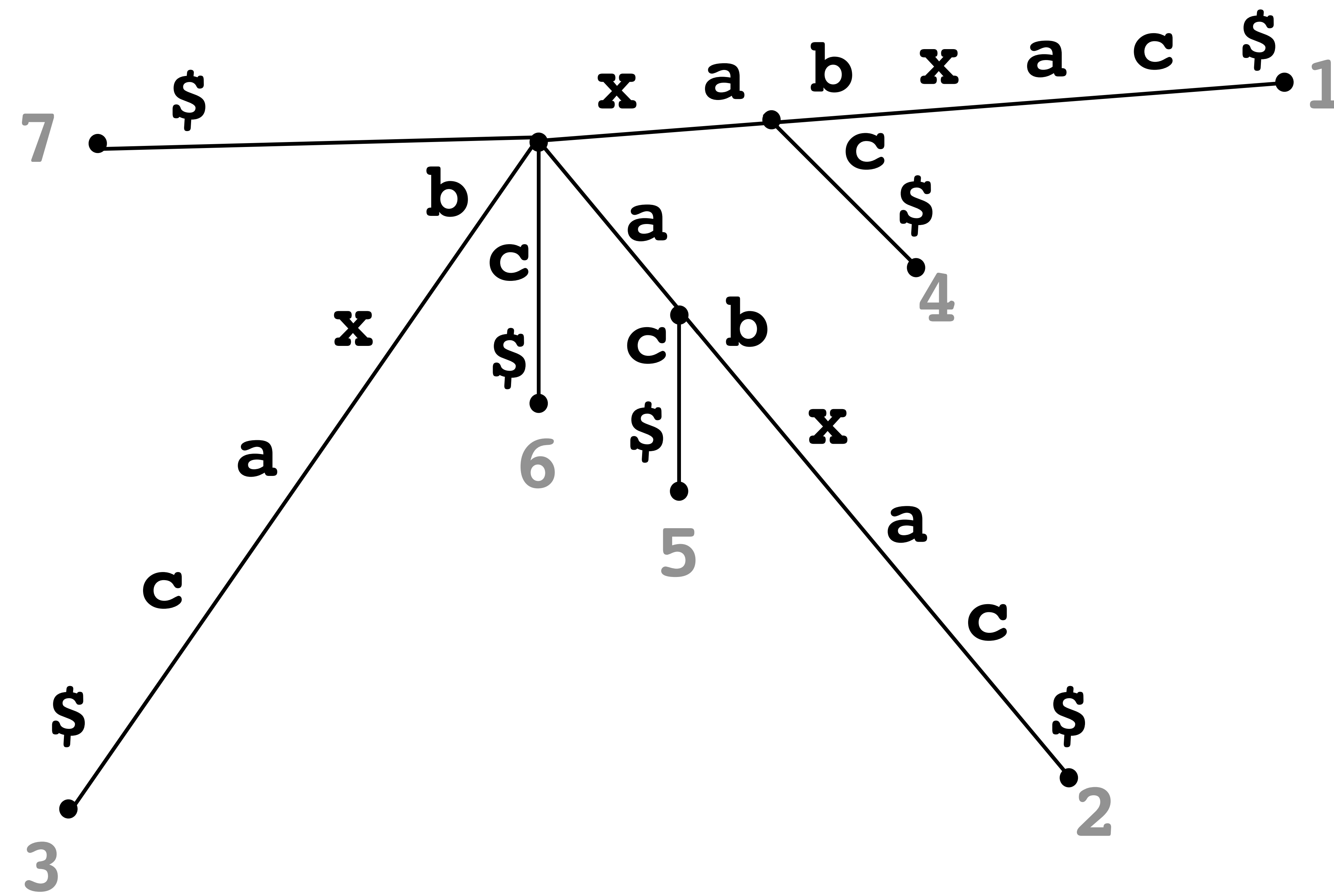
Suffix Tree

```
1234567
xabxac$
```



Suffix Tree

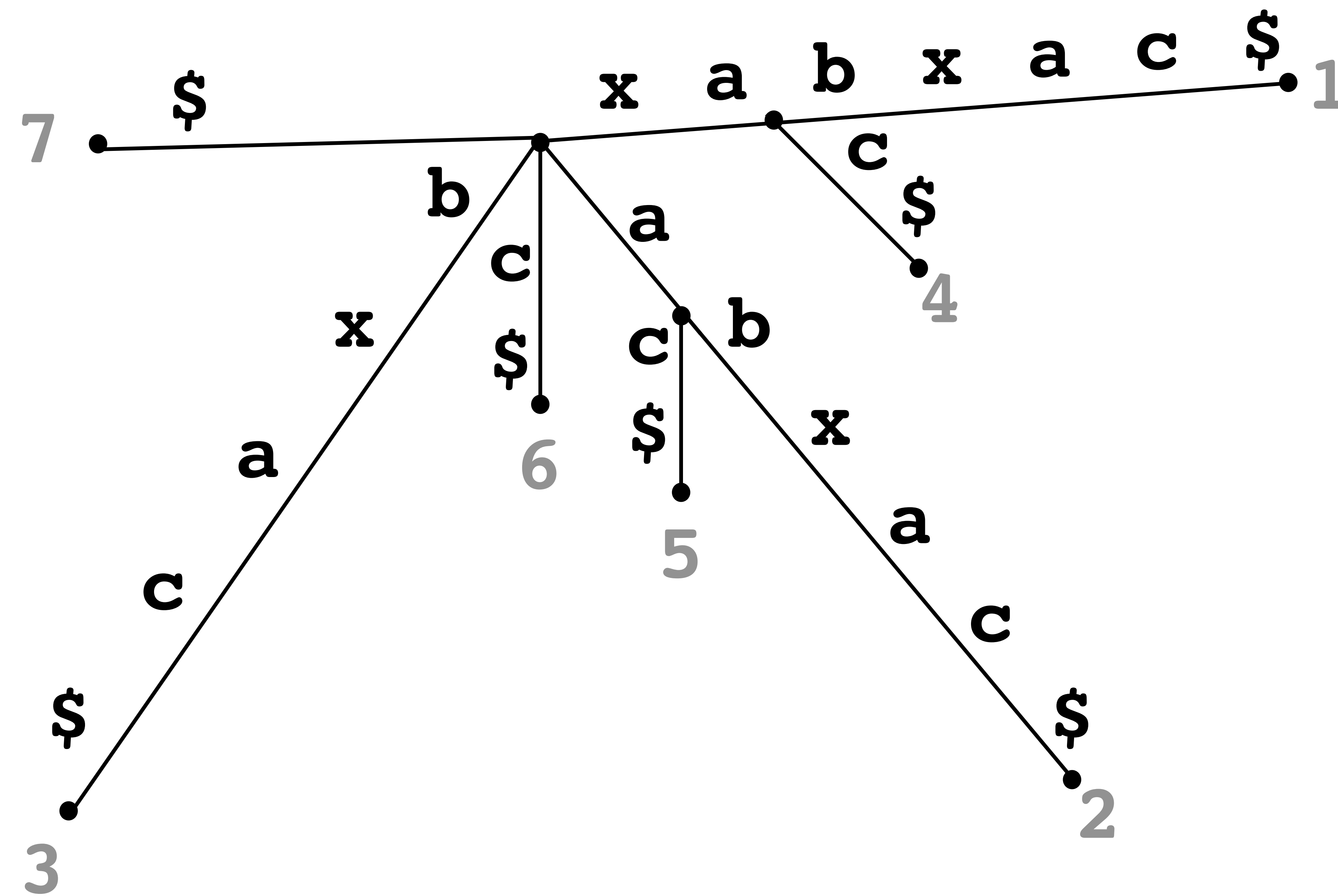
1234567
xabxac$



Suffix Tree

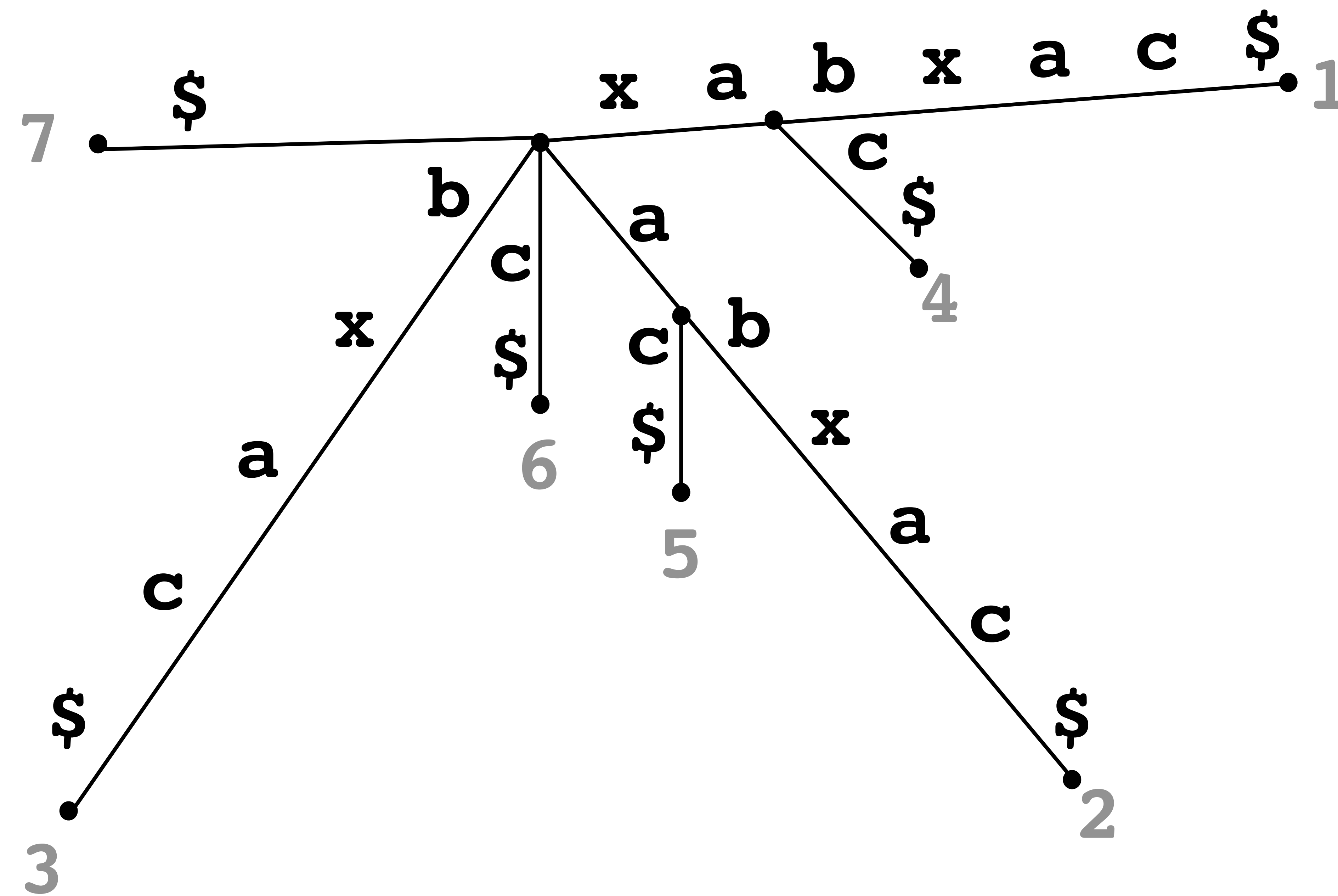
Does `xabxac` contain `xa`?

1234567
xabxac$



Suffix Tree

1234567
xabxac$

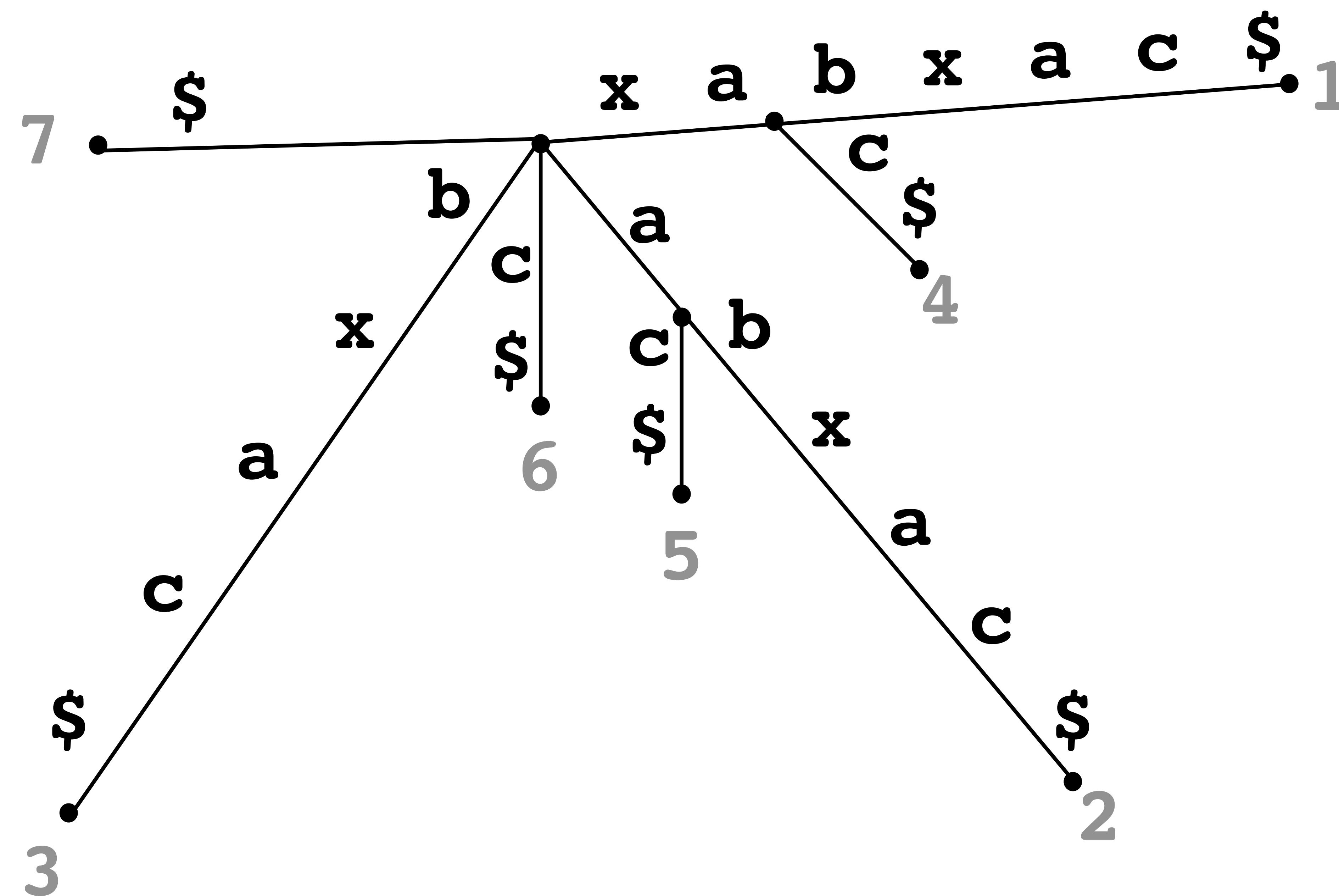


Does xabxac contain xa?

How long does it take to answer that question?

Suffix Tree

1234567
xabxac$



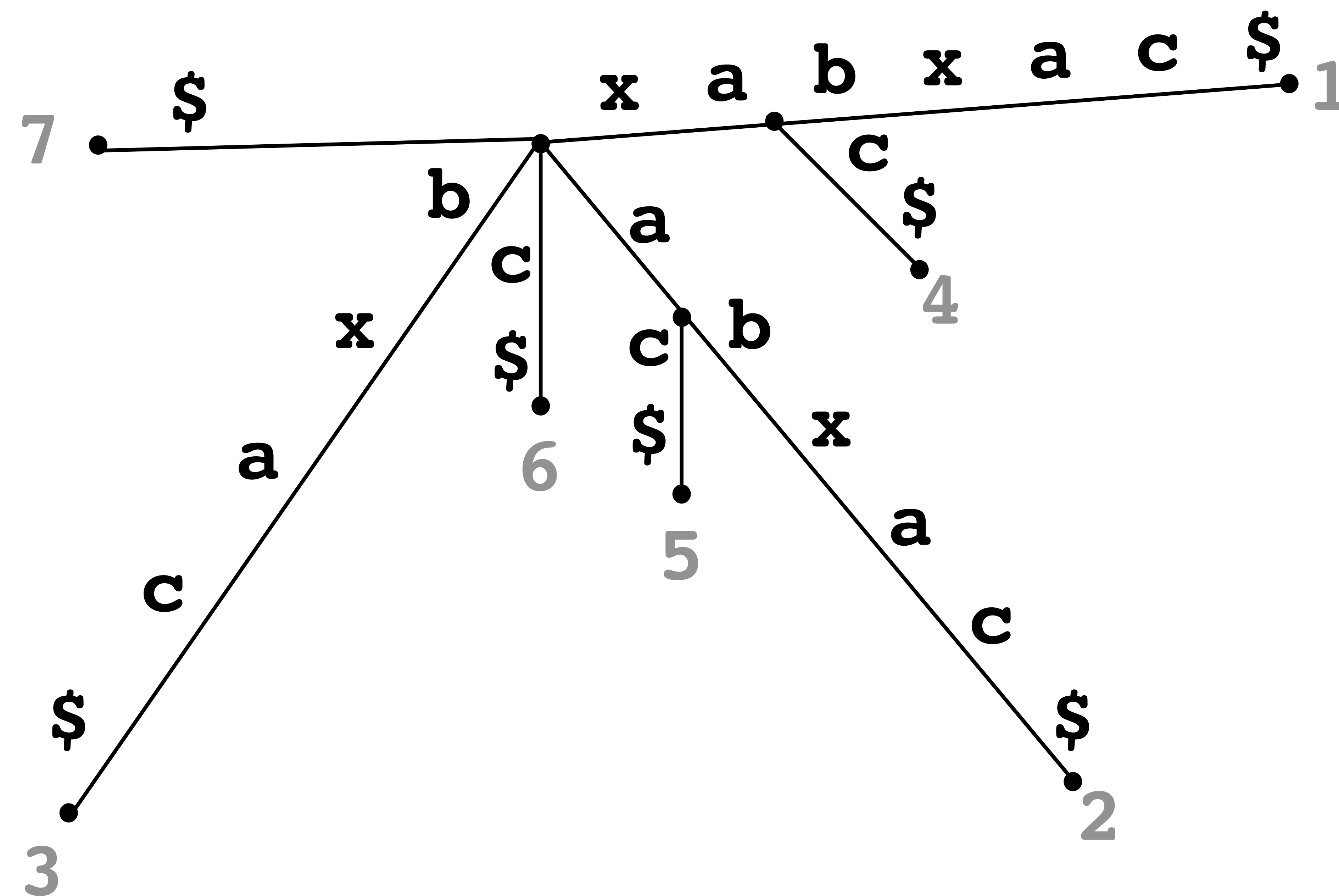
Does xabxac contain xa?

How long does it take to answer that question?

How many instances of xa does xabxac contain?

Suffix Tree

1234567
`xabxac$`



Does `xabxac` contain `xa`?

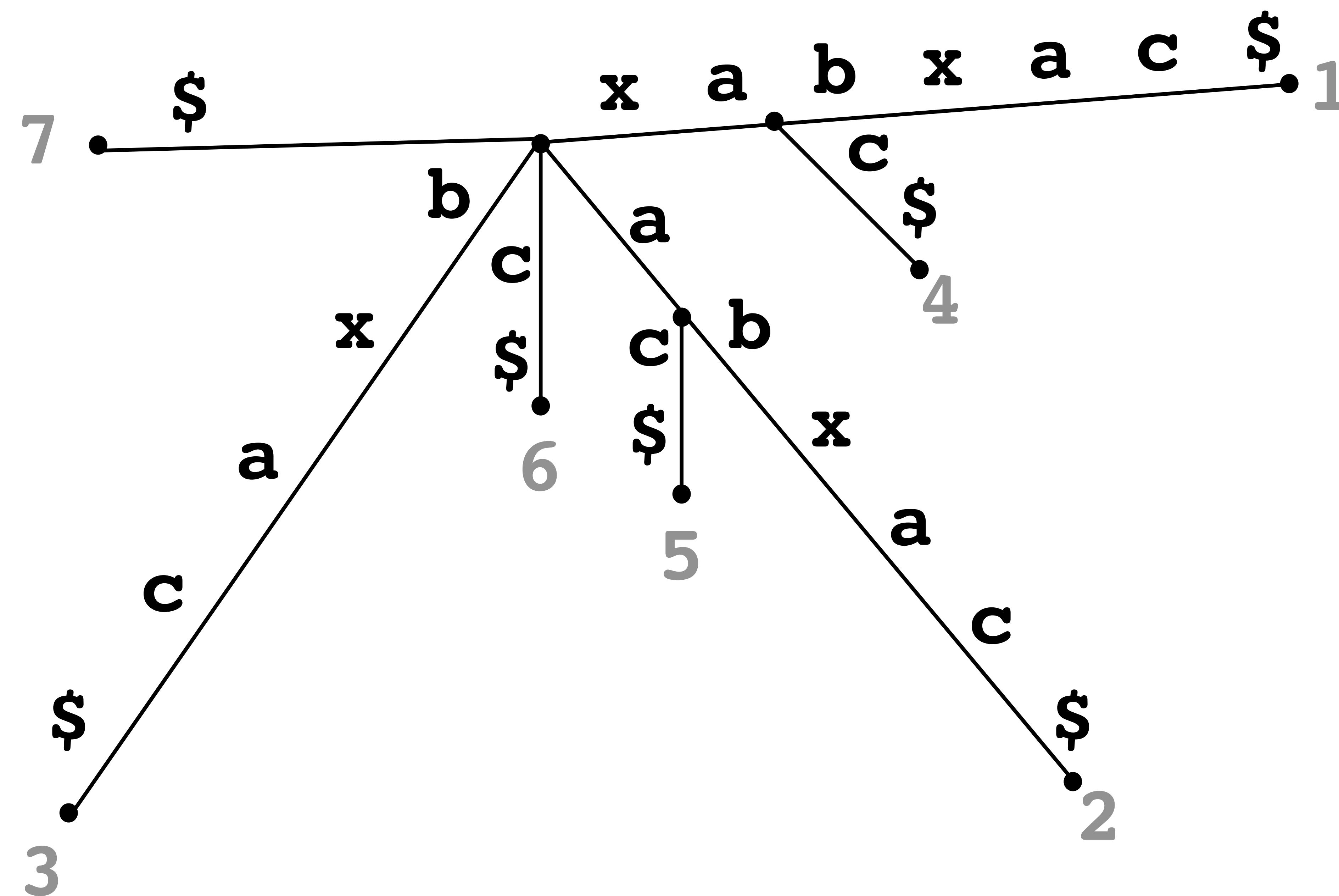
How long does it take to answer that question?

How many instances of `xa` does `xabxac` contain?

Where are the instances of `xa` within `xabxac`?

Suffix Tree

1234567
`xabxac$`



Does `xabxac` contain `xa`?

How long does it take to answer that question?

How many instances of `xa` does `xabxac` contain?

Where are the instances of `xa` within `xabxac`?

How long would it take to construct the tree?

Ukkonen's Algorithm

- Builds a suffix tree in *O(m)*-time
- Uses the idea of "implicit suffix trees" which don't include terminating characters, and iterative extension to build the tree in linear time

Ukkonen's Algorithm

Concept

•

Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

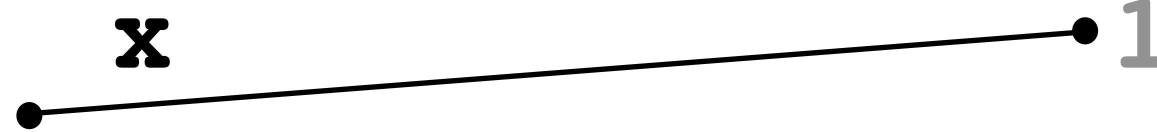
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

1
x



Three rules for adding suffix $S[i...j+1]$ to the implicit tree up to j

Rule 1 In the current tree $S[i...j]$ ends at a leaf, append character $S[j+1]$ to the label.

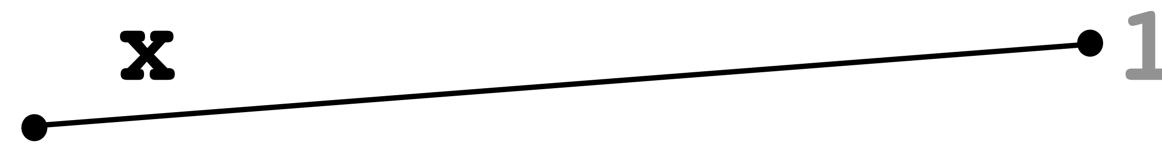
Rule 2 $S[i...j]$ ends at an internal node or in the middle of a label, and no extension starts with $S[j+1]$, add new leaf.

Rule 3 Some path from $S[i...j]$ starts with $S[j+1]$, do nothing.

Ukkonen's Algorithm

Concept

```
12
xa
```



Three rules for adding suffix $S[i...j+1]$ to the implicit tree up to j

Rule 1 In the current tree $S[i...j]$ ends at a leaf, append character $S[j+1]$ to the label.

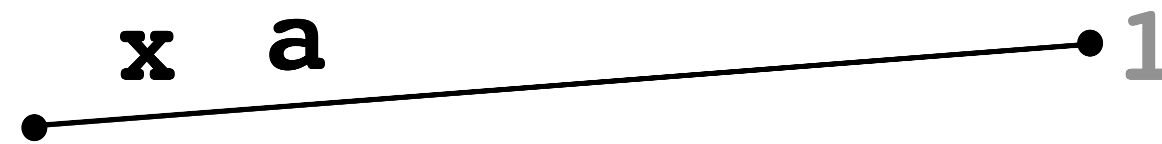
Rule 2 $S[i...j]$ ends at an internal node or in the middle of a label, and no extension starts with $S[j+1]$, add new leaf.

Rule 3 Some path from $S[i...j]$ starts with $S[j+1]$, do nothing.

Ukkonen's Algorithm

Concept

12
xa



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

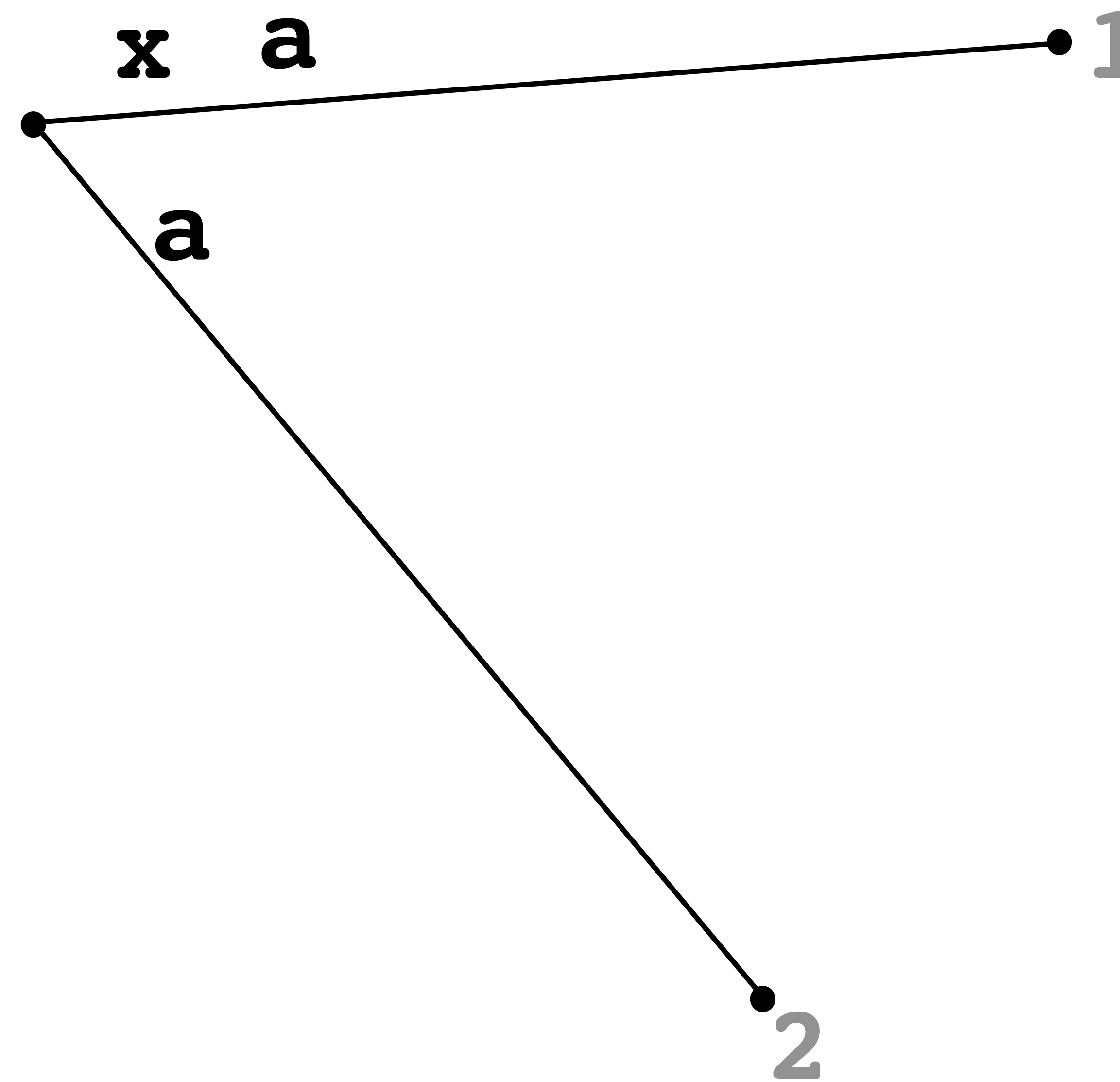
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

12
xa



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

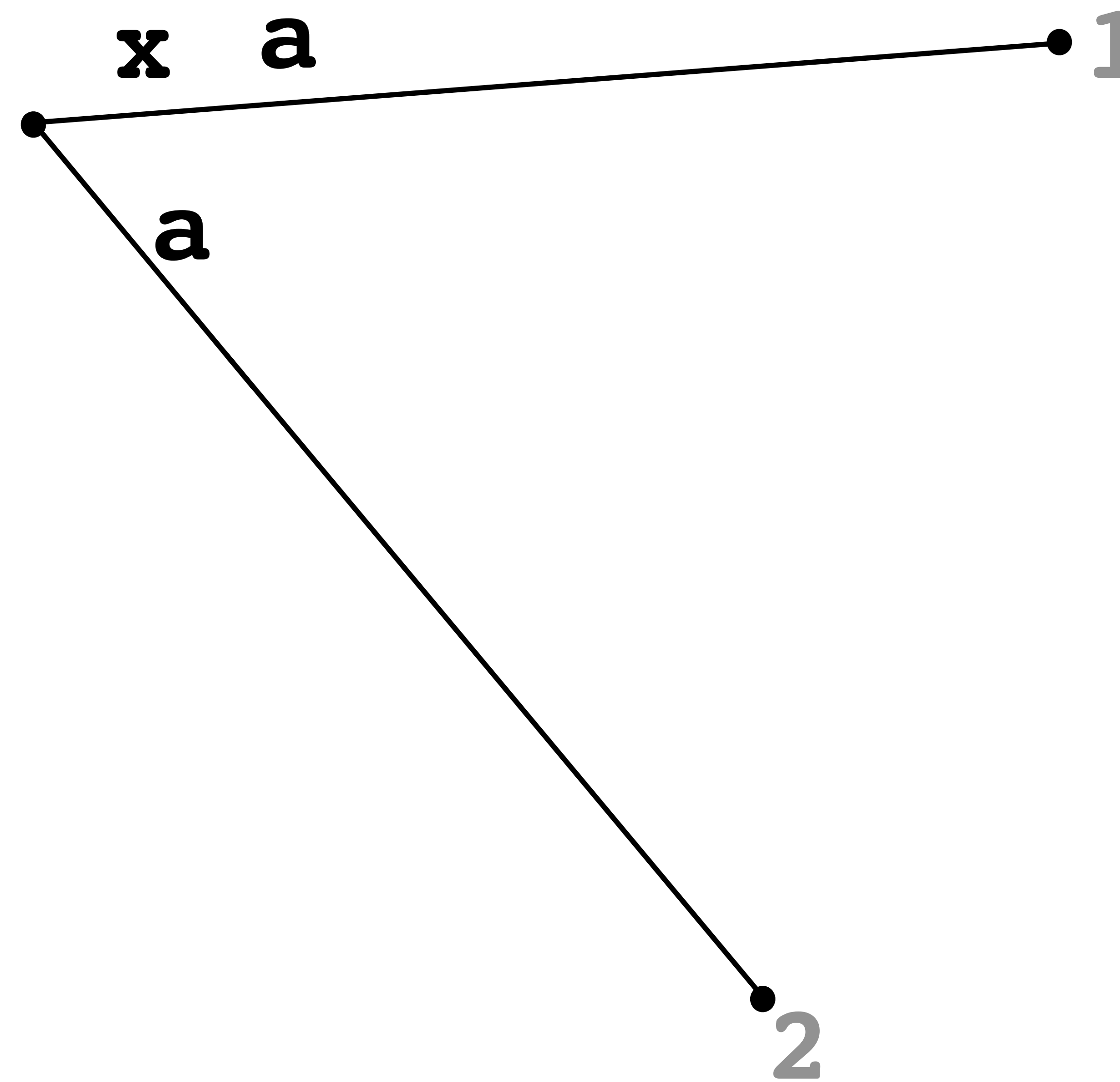
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

```
123
xab
```



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

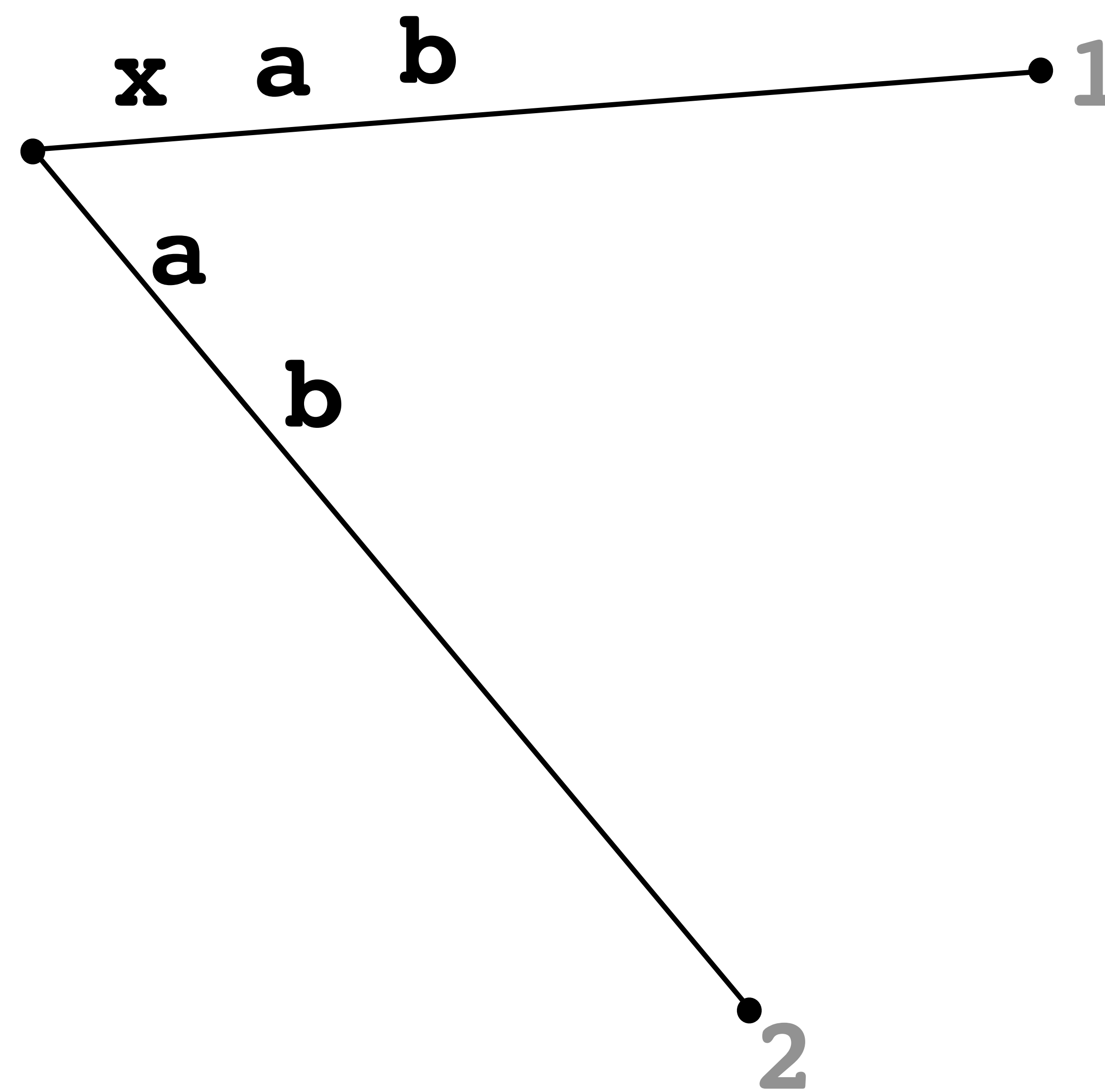
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

123
xab



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

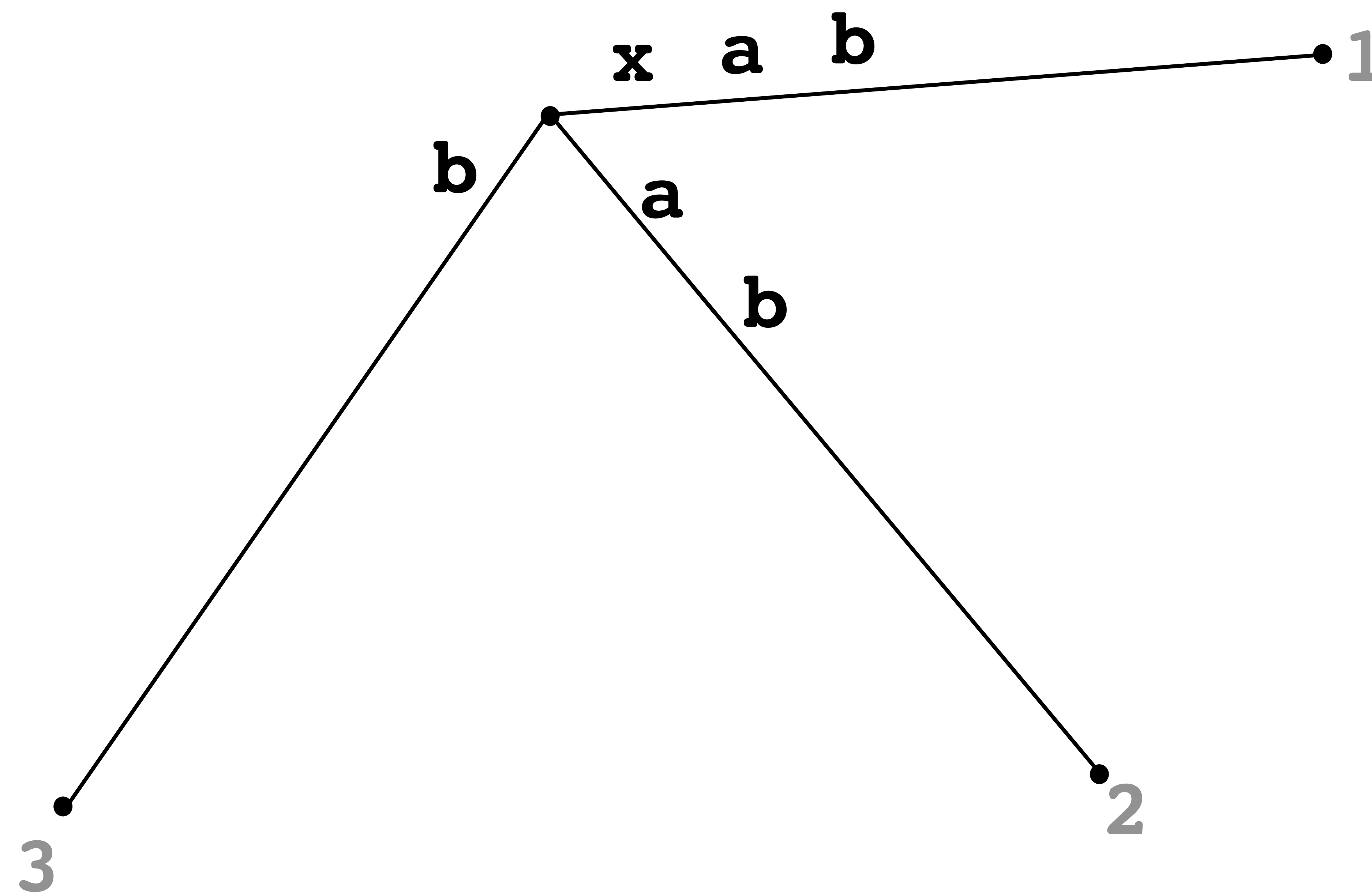
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

123
xab



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

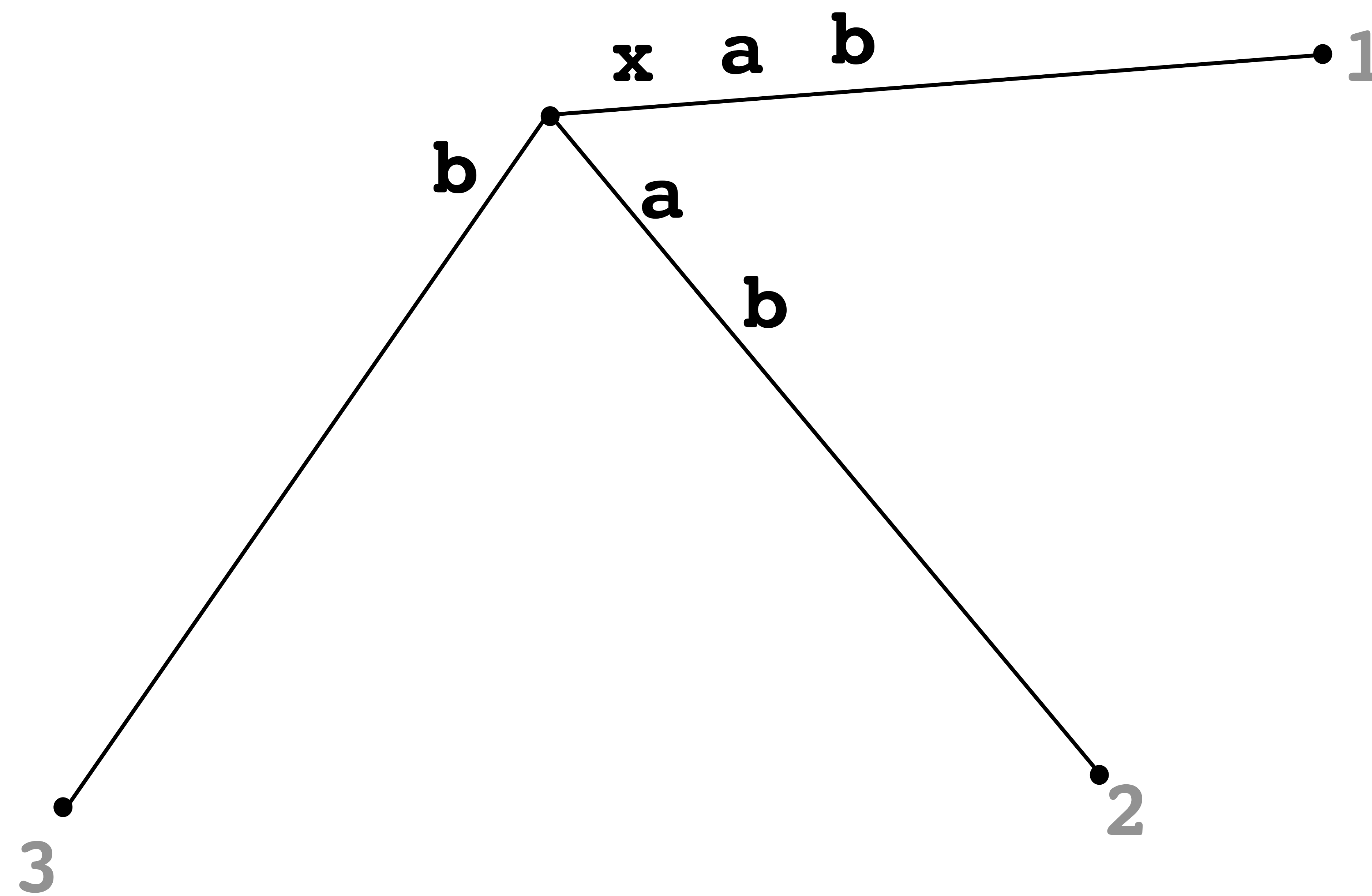
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

1234
xabx



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

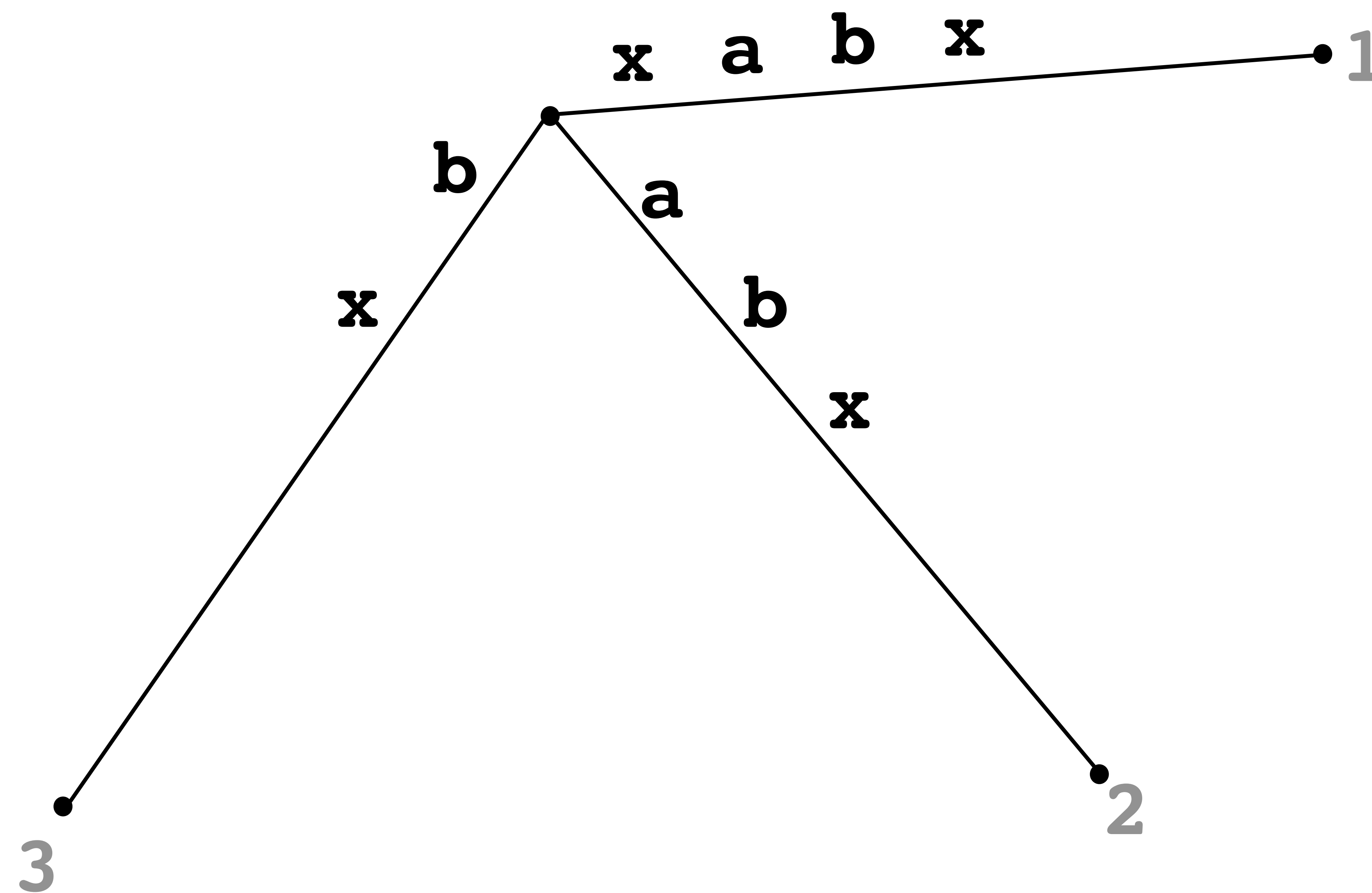
Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

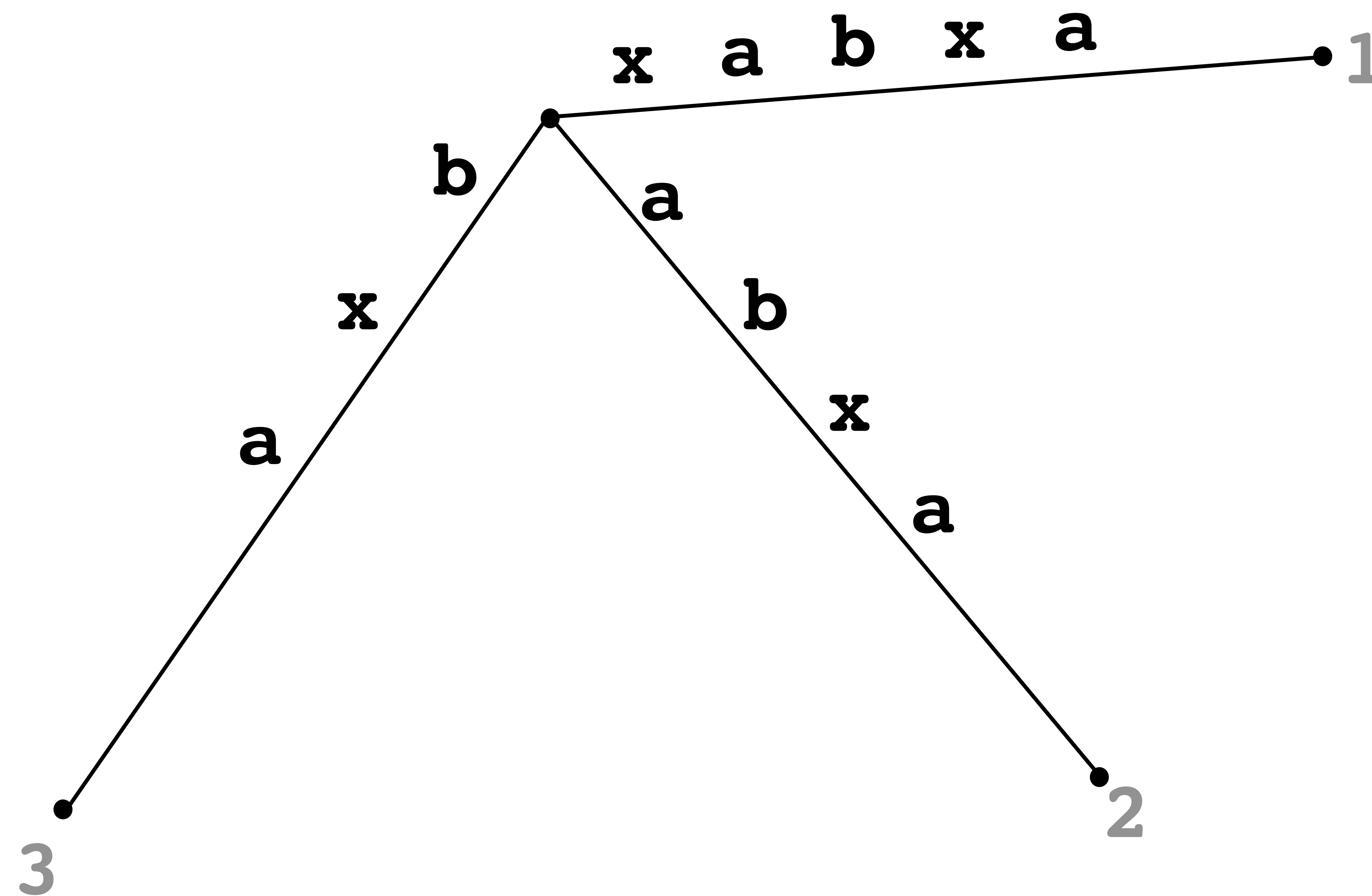
Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

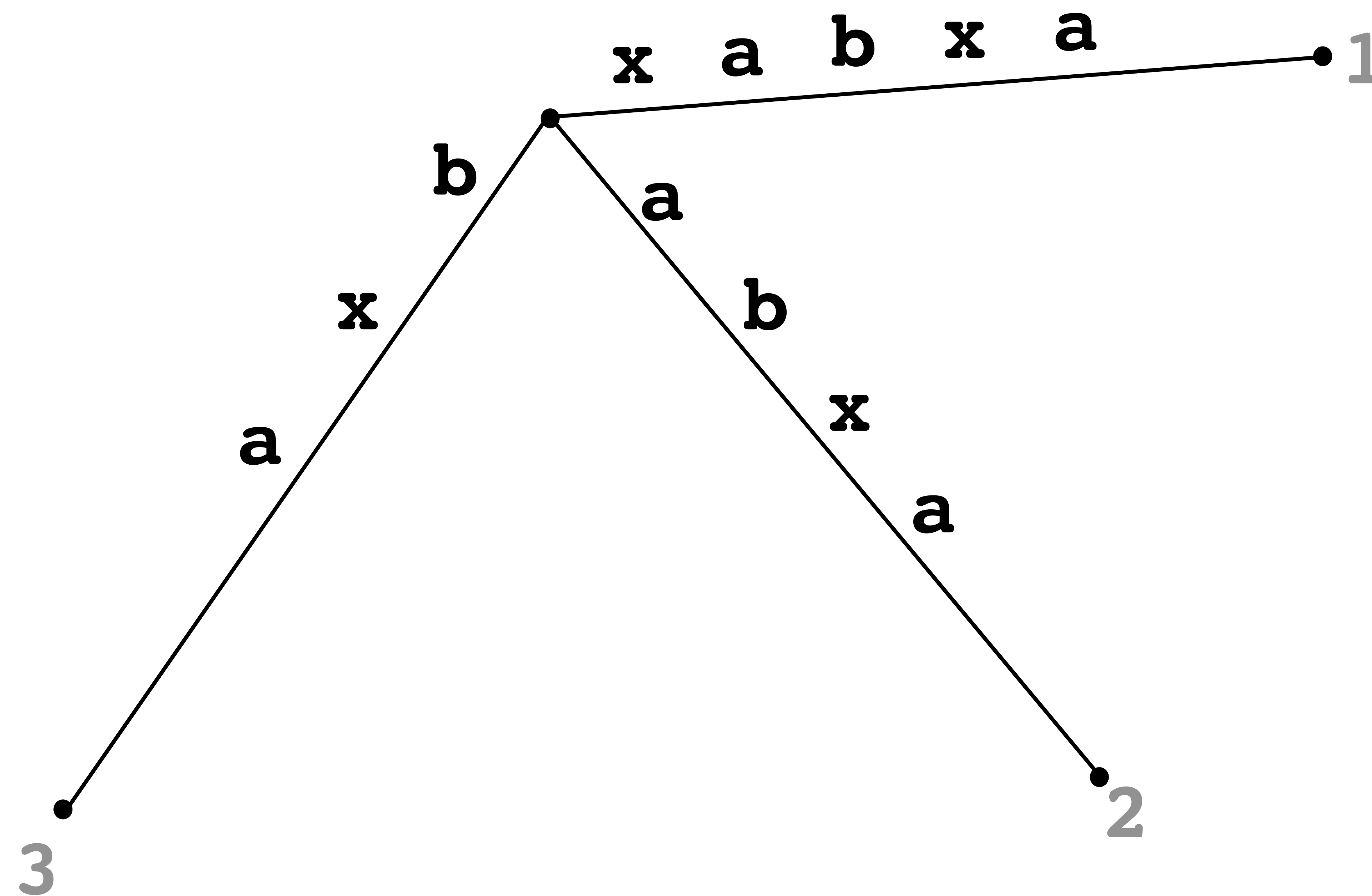
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

```
123456
xabxac
```



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

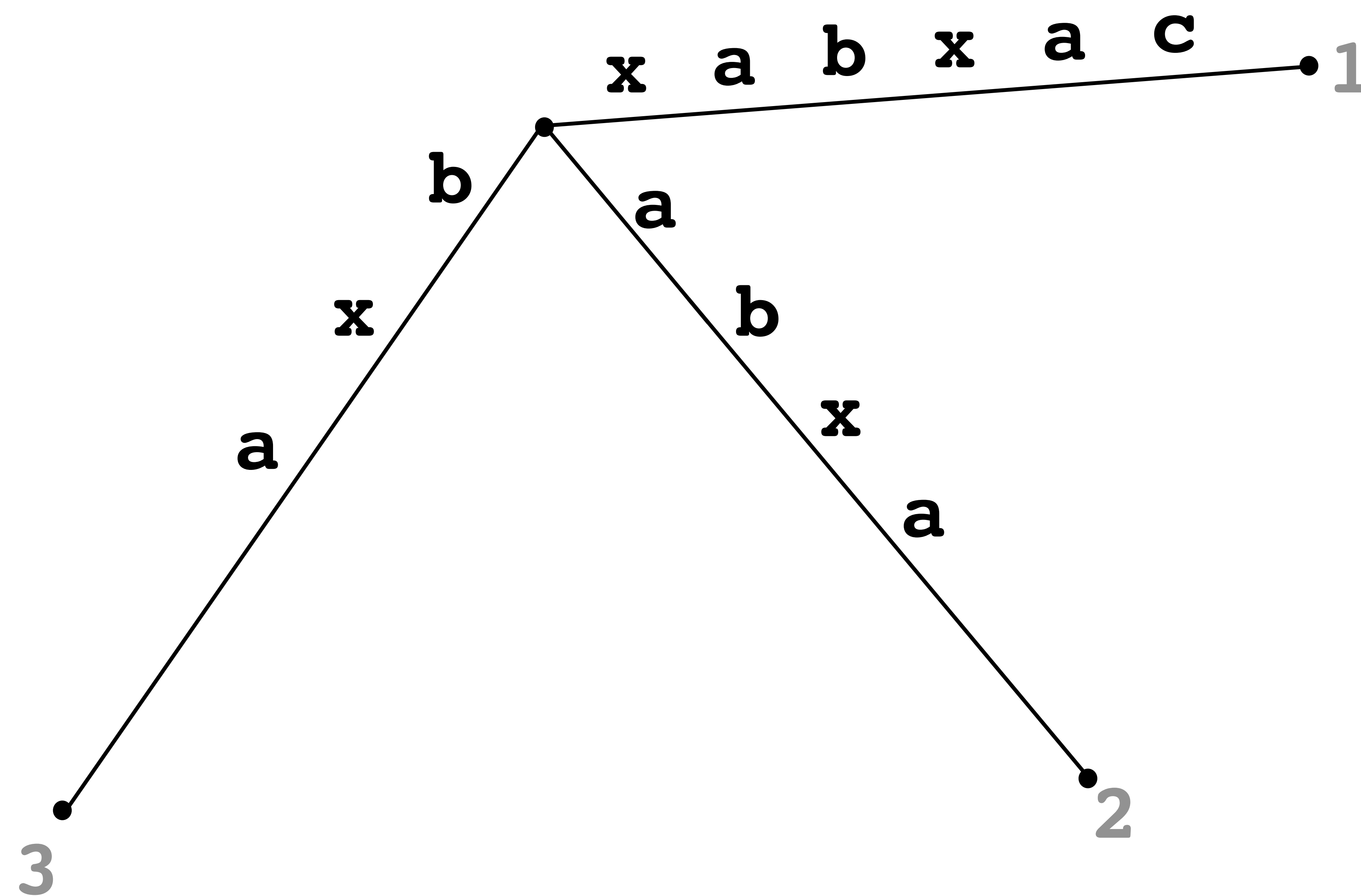
Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

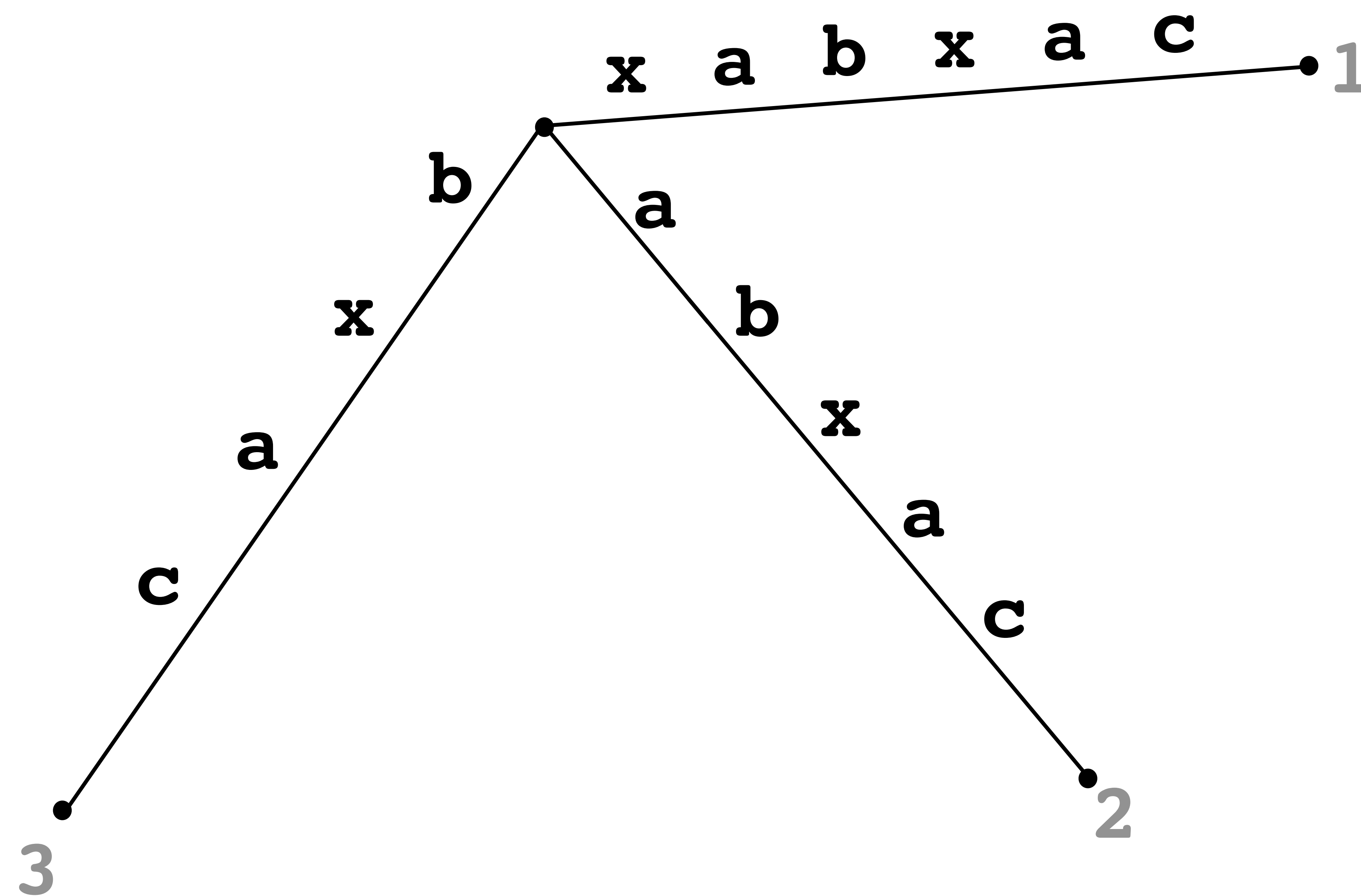
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

123456
xabxac



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

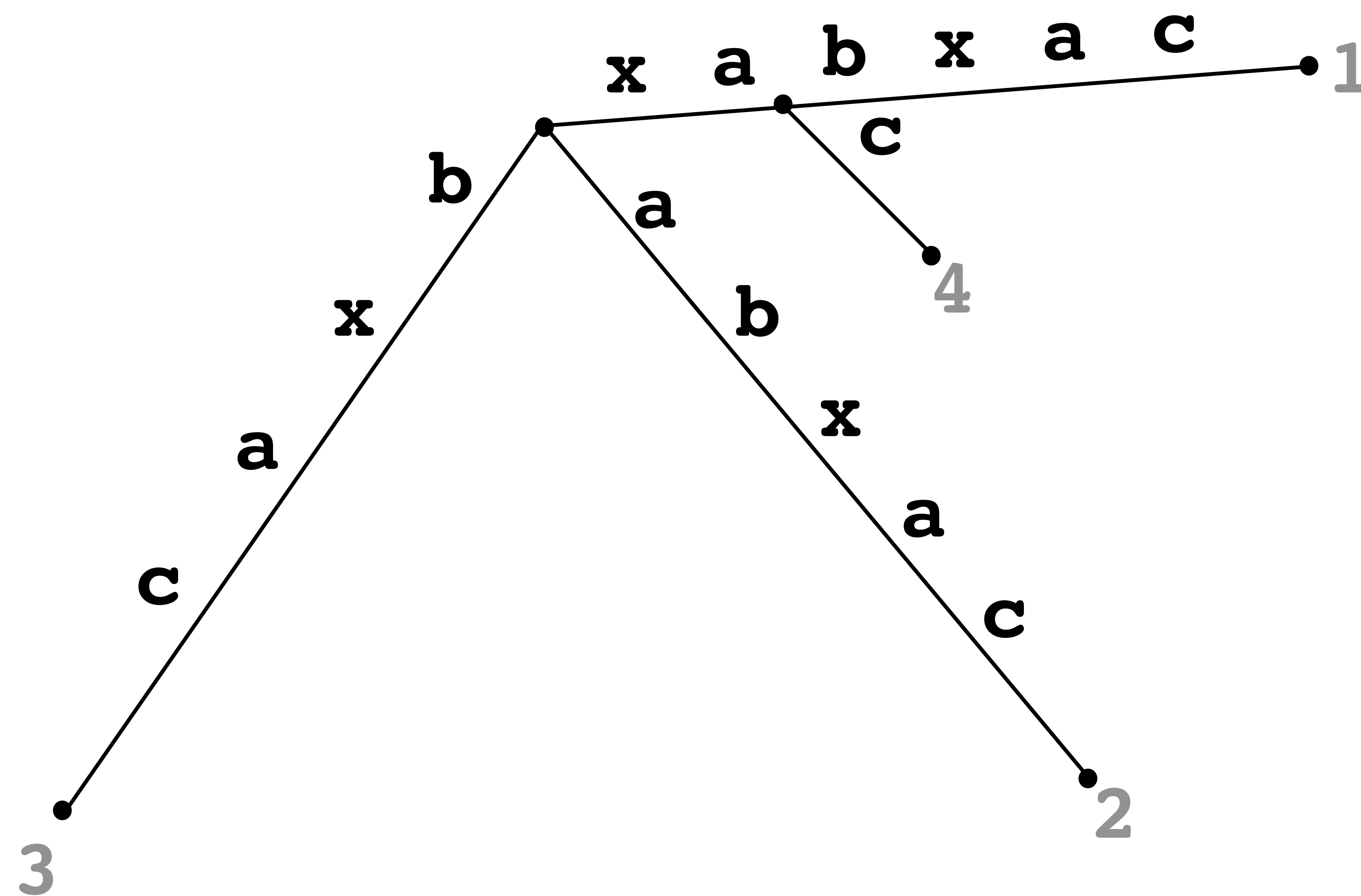
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

123456
xabxac



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

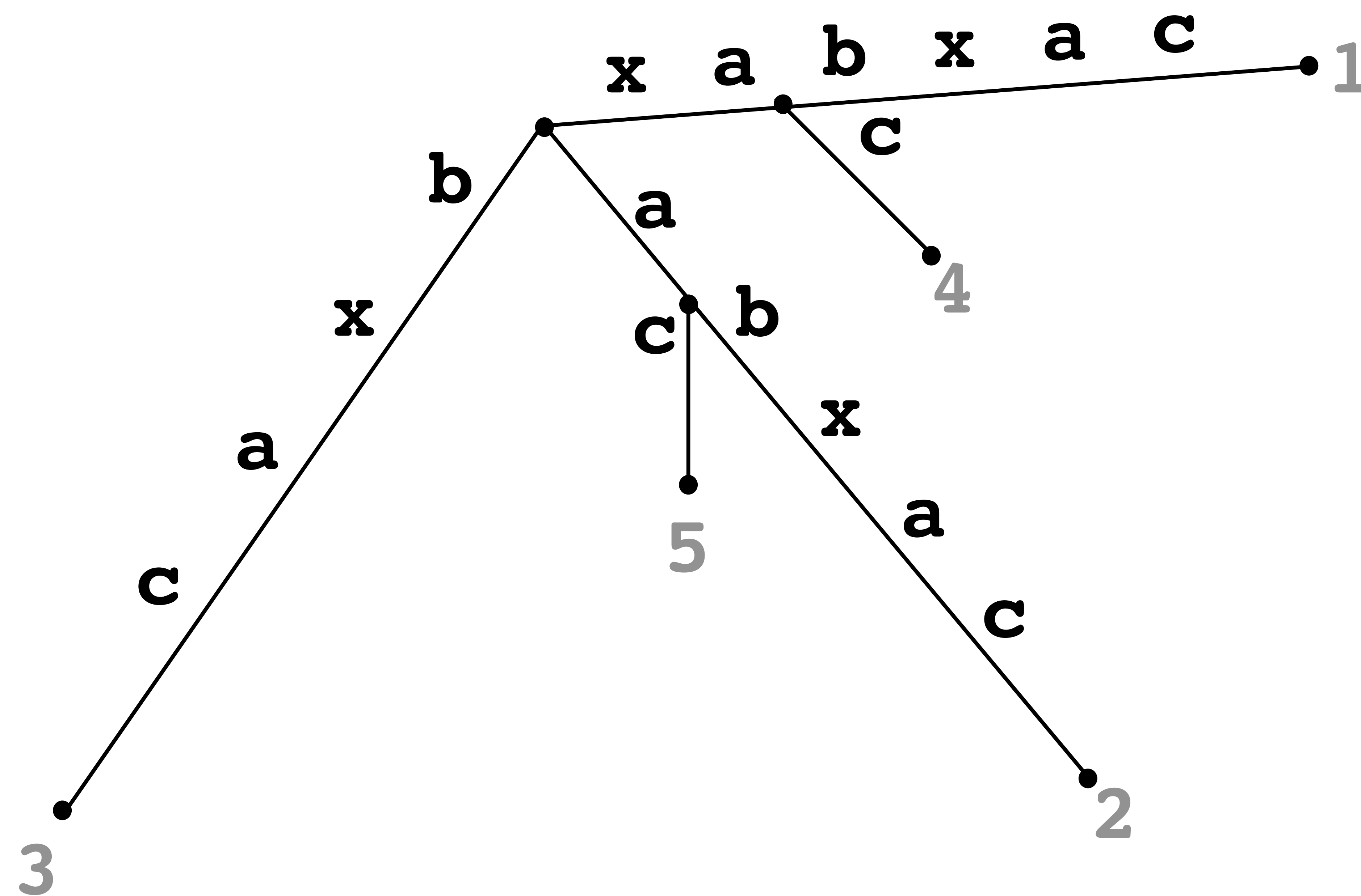
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

123456
xabxac



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

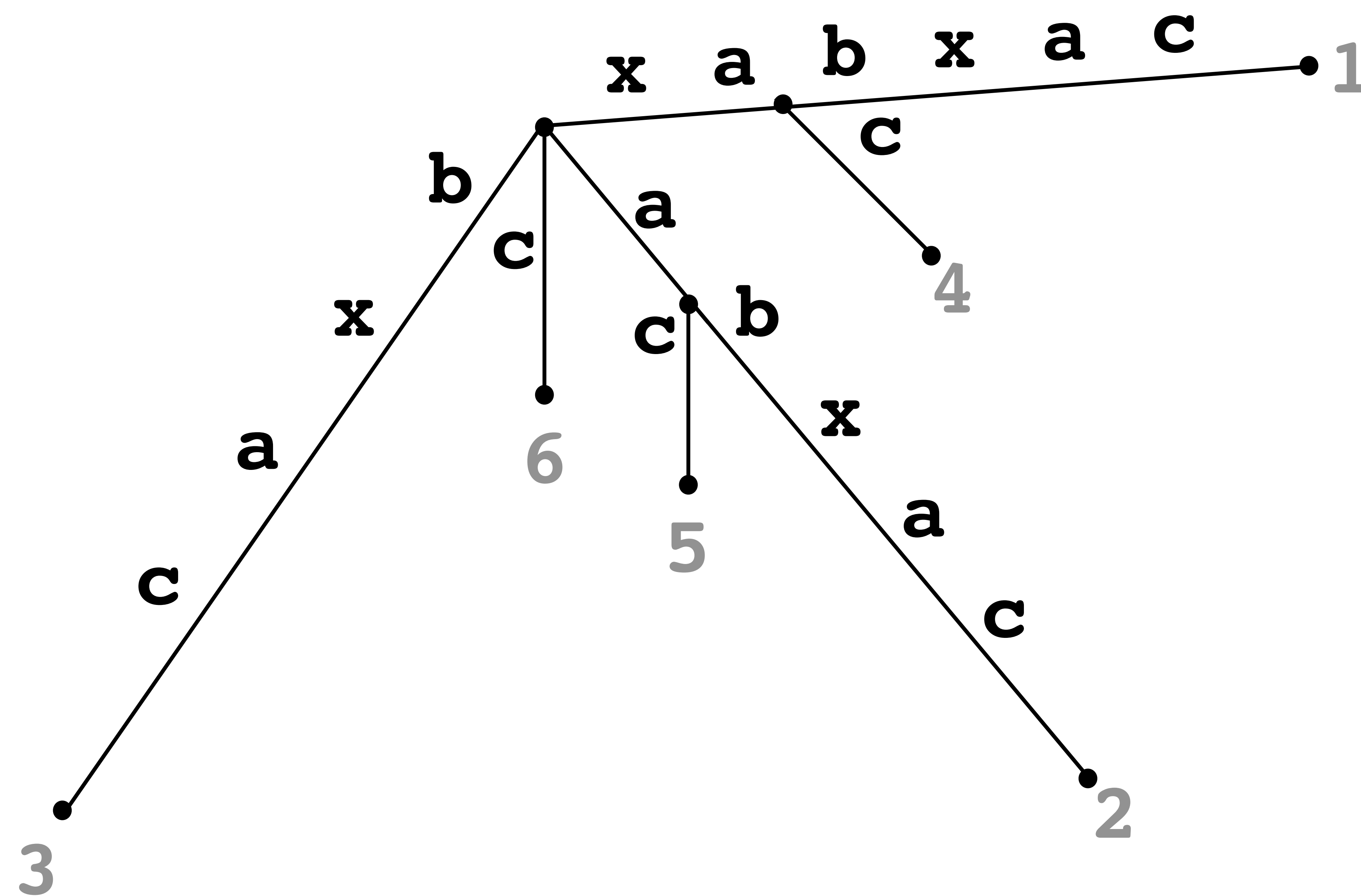
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

123456
xabxac



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

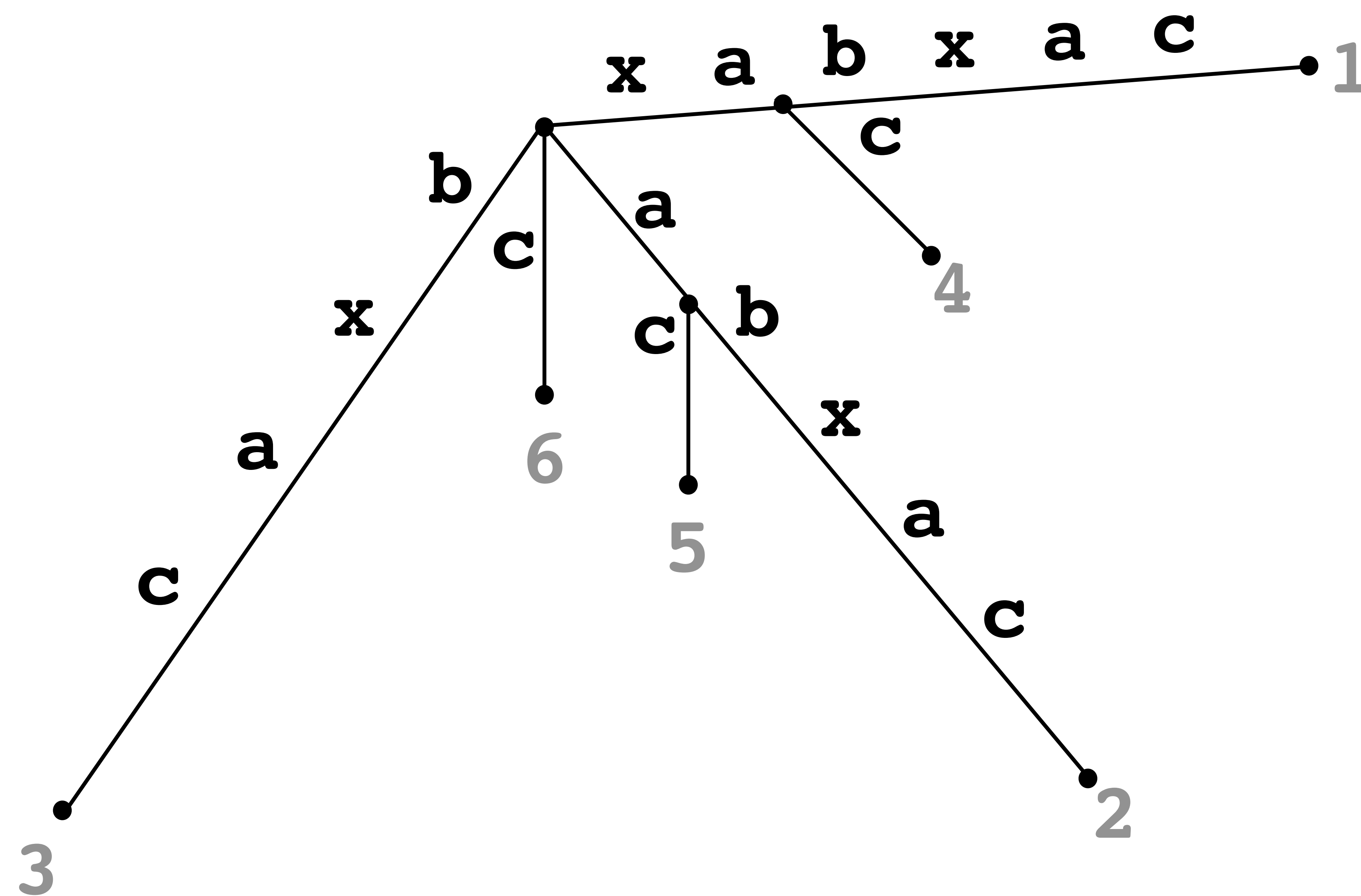
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

Concept

123456
xabxac



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

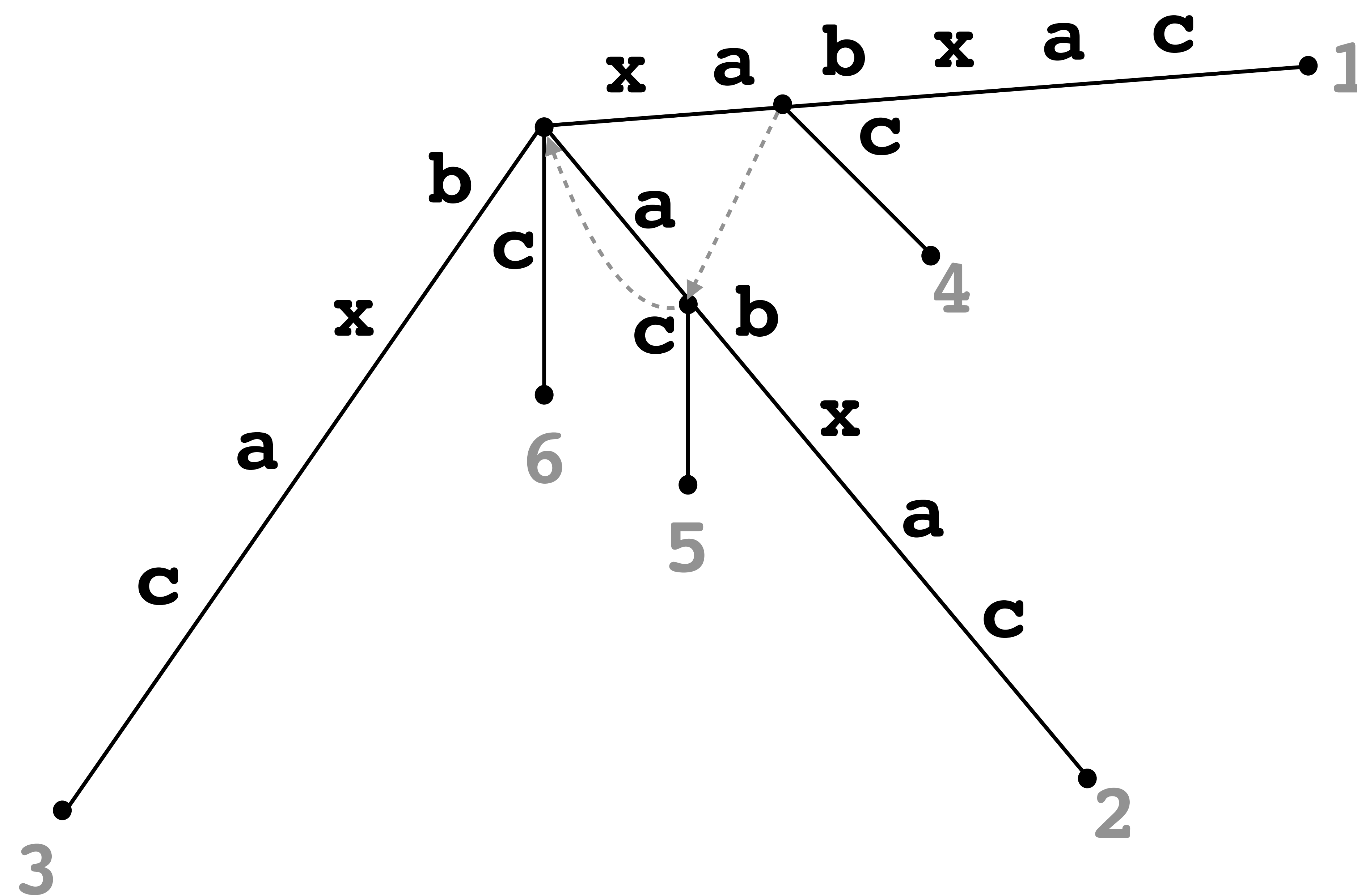
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

How long would it take to run this construction method?

Ukkonen's Algorithm

Concept

123456
xabxac



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

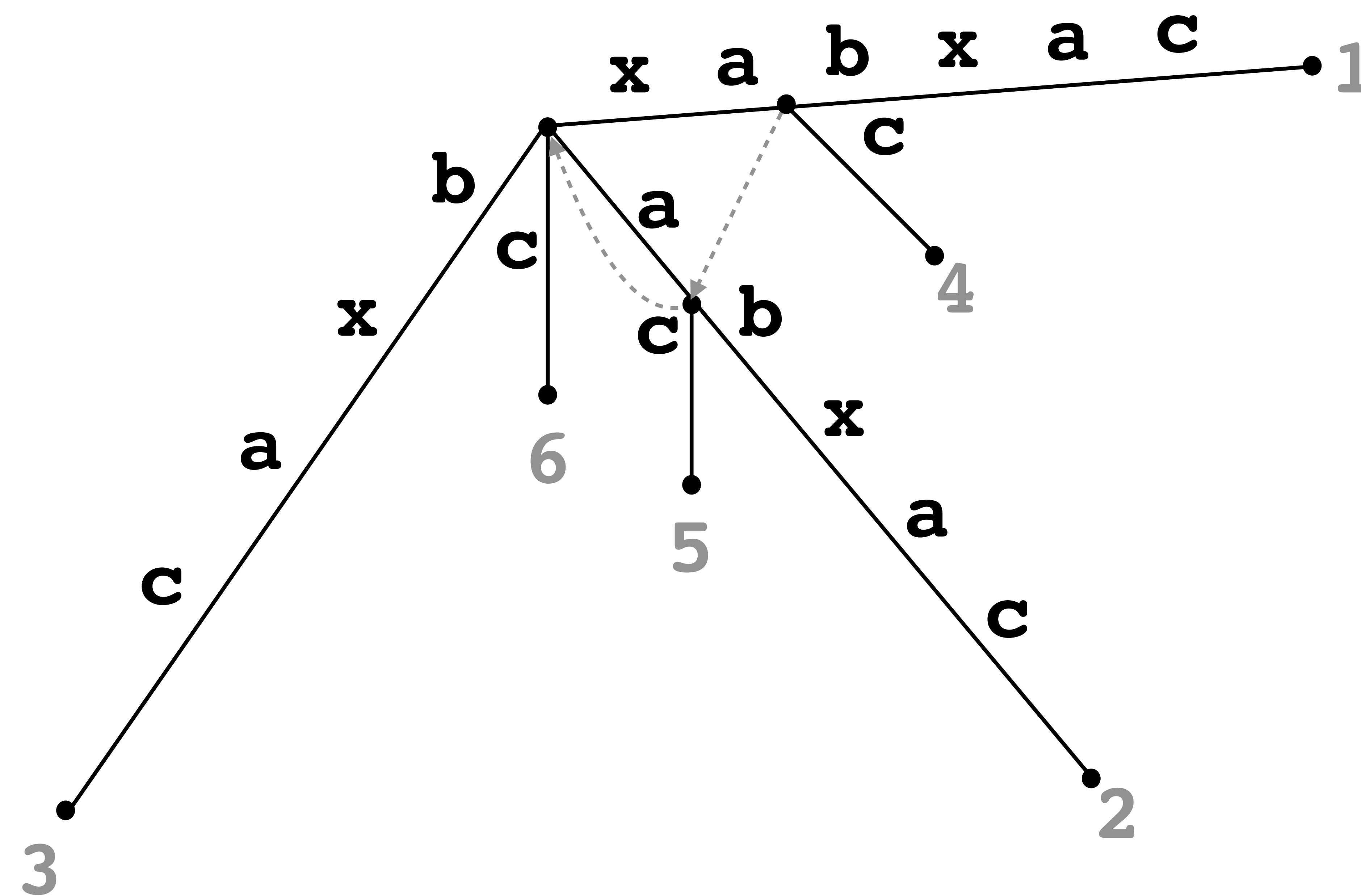
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

Concept

123456
xabxac



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

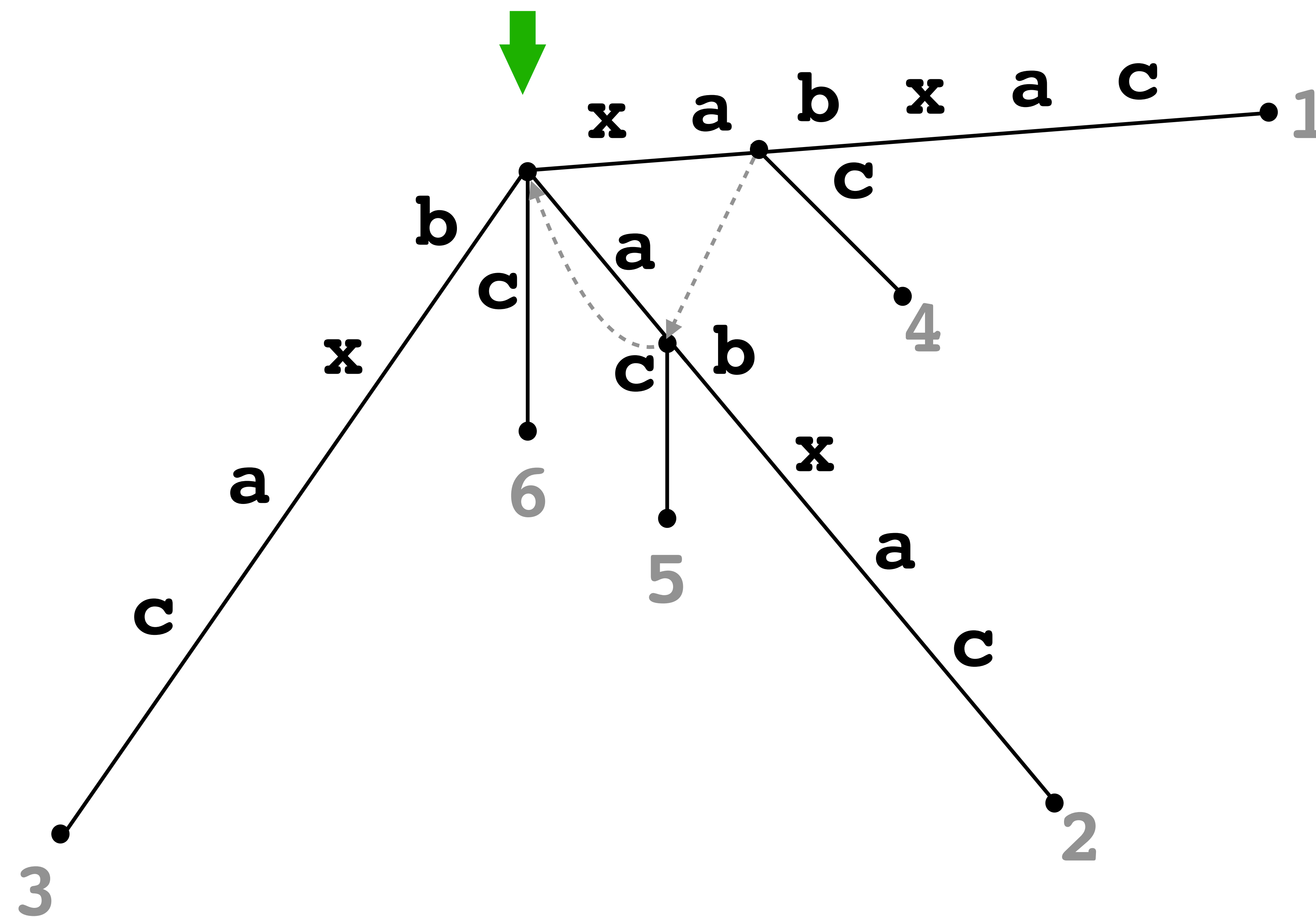
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Each newly created node (by Rule 2) will have a suffix link from it by the end of the next extension.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

1234567
xabxacf



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

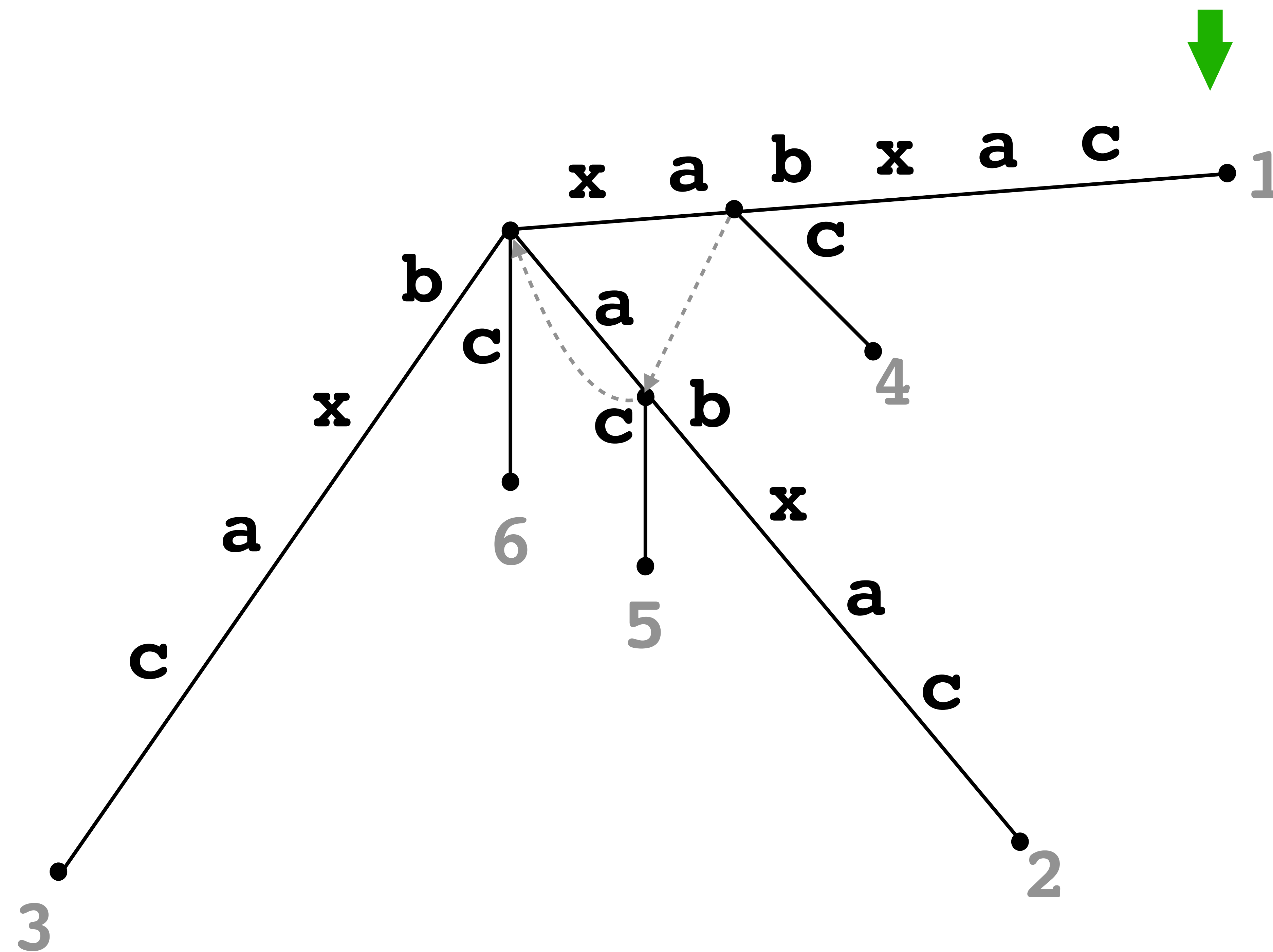
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

1234567
xabxacf



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

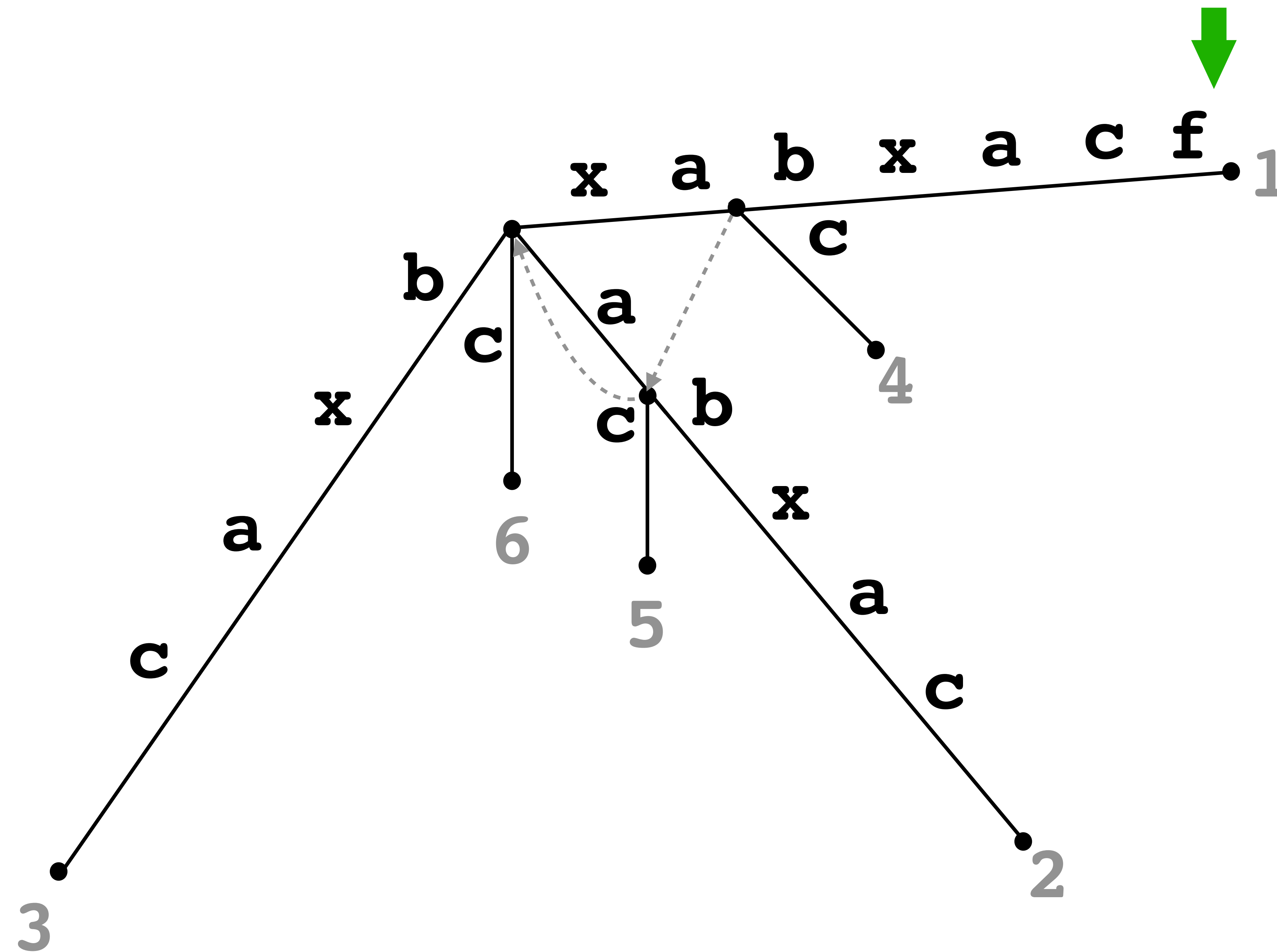
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

1234567
xabxacf



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

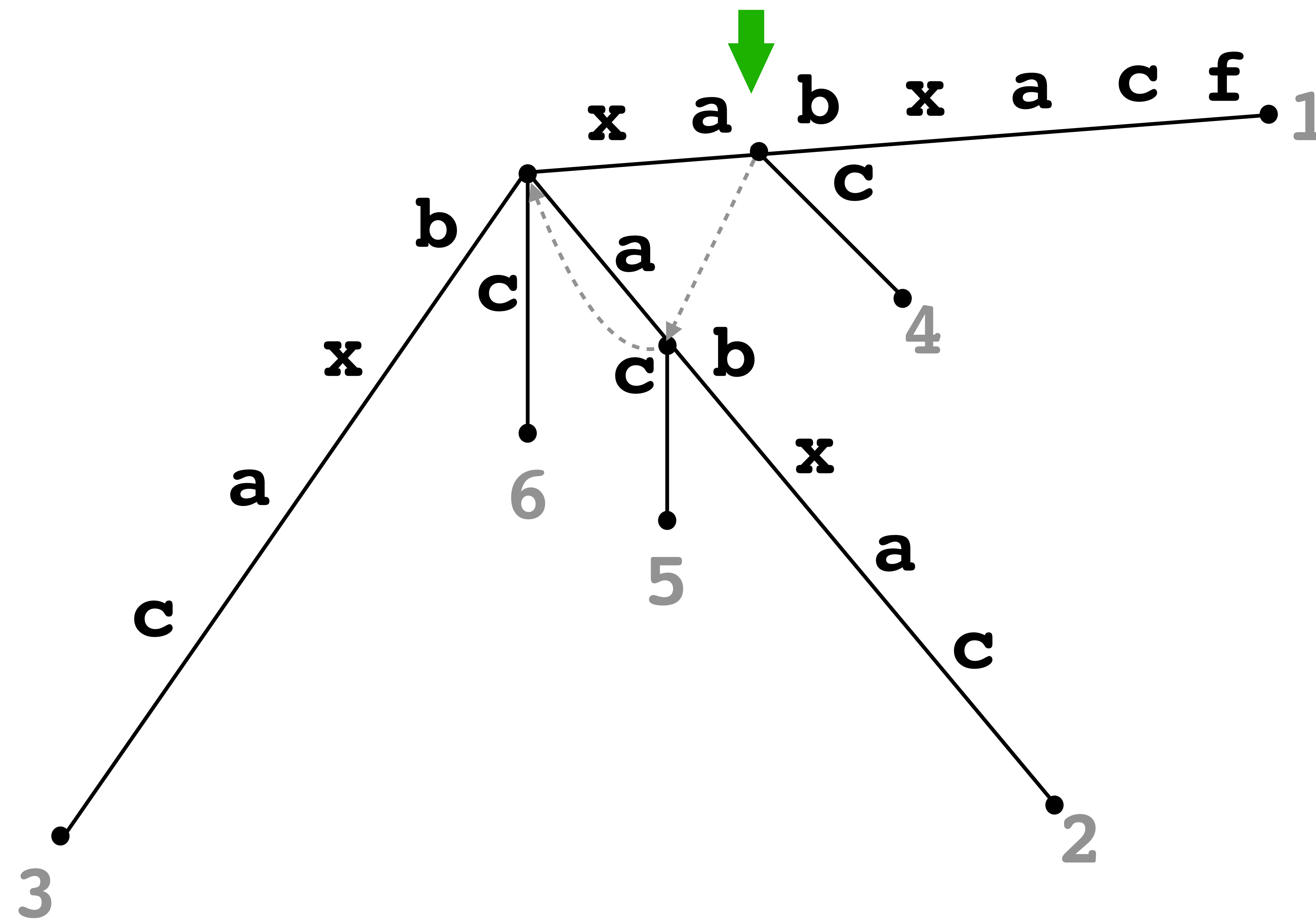
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

```
1234567
xabxacf
```



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

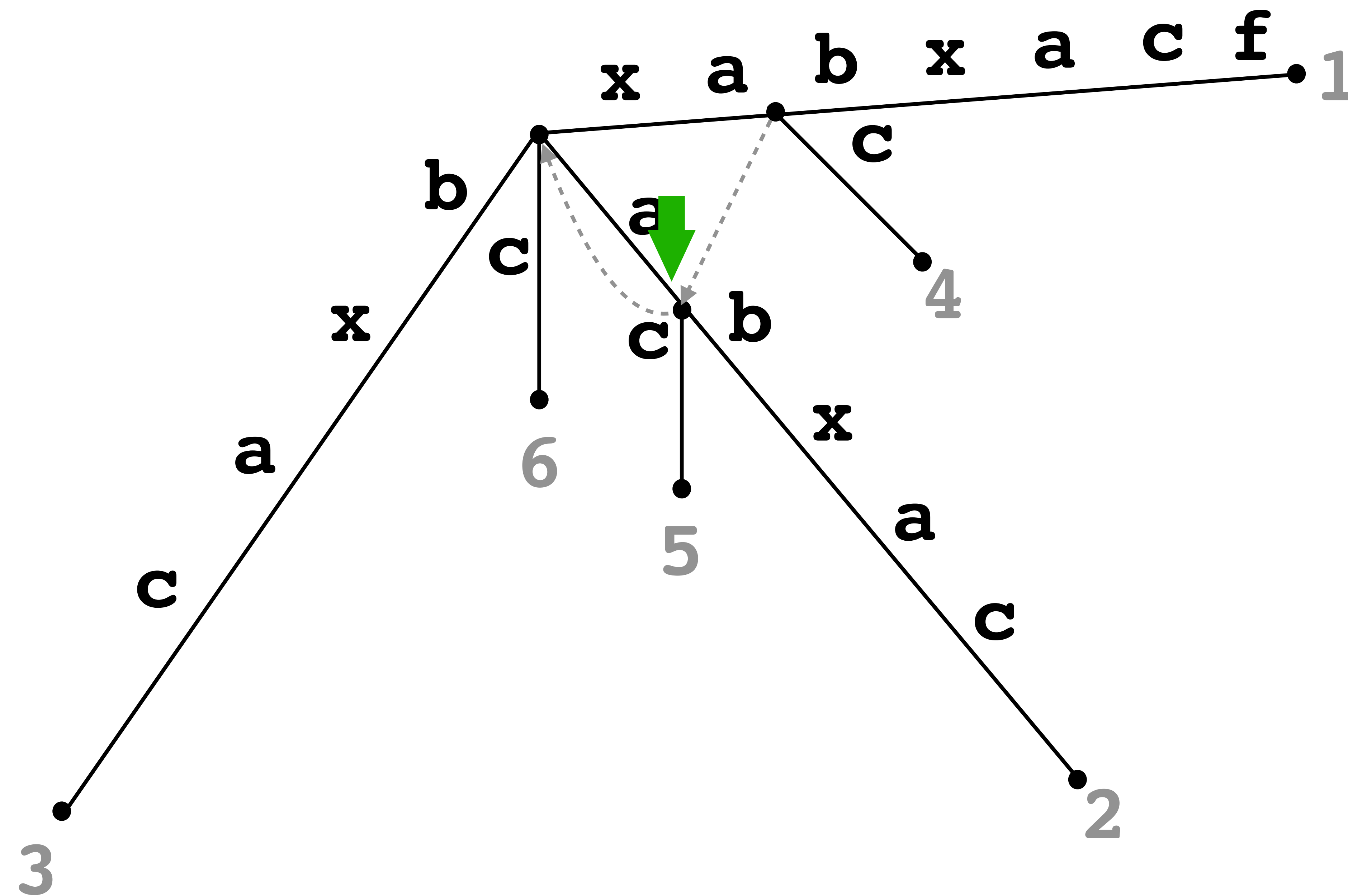
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

1234567
xabxacf



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

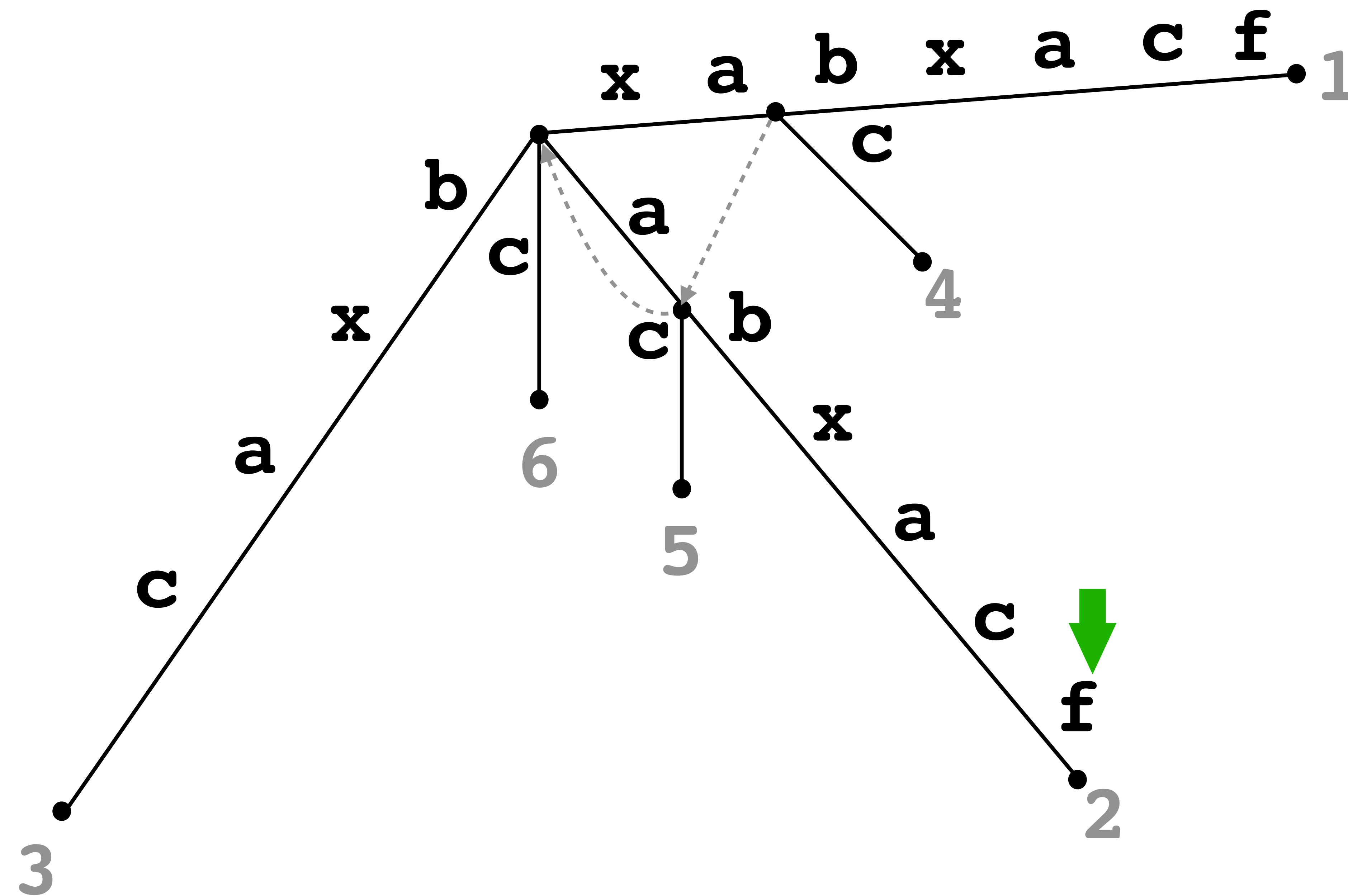
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

1234567
xabxacf



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

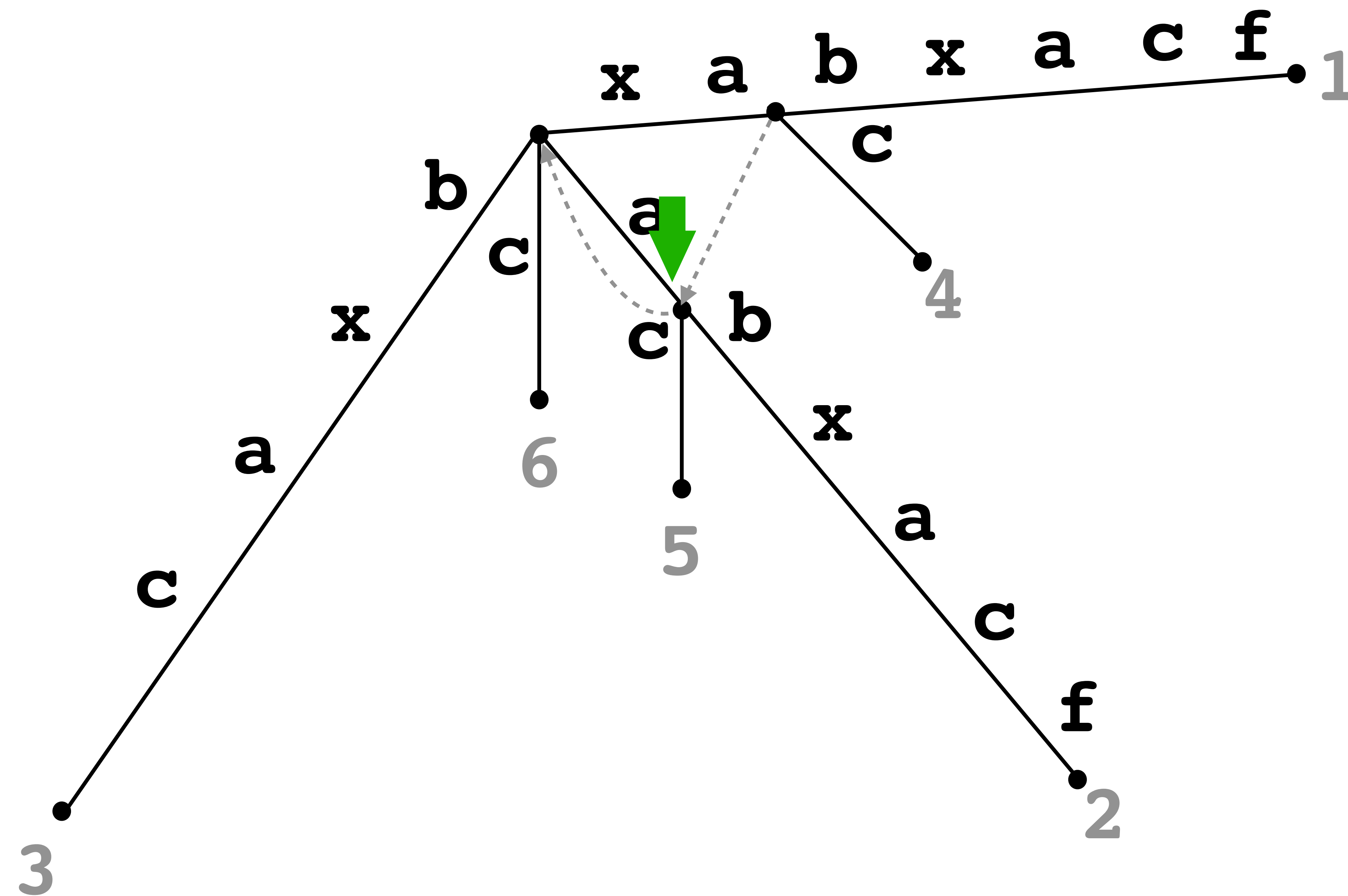
Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

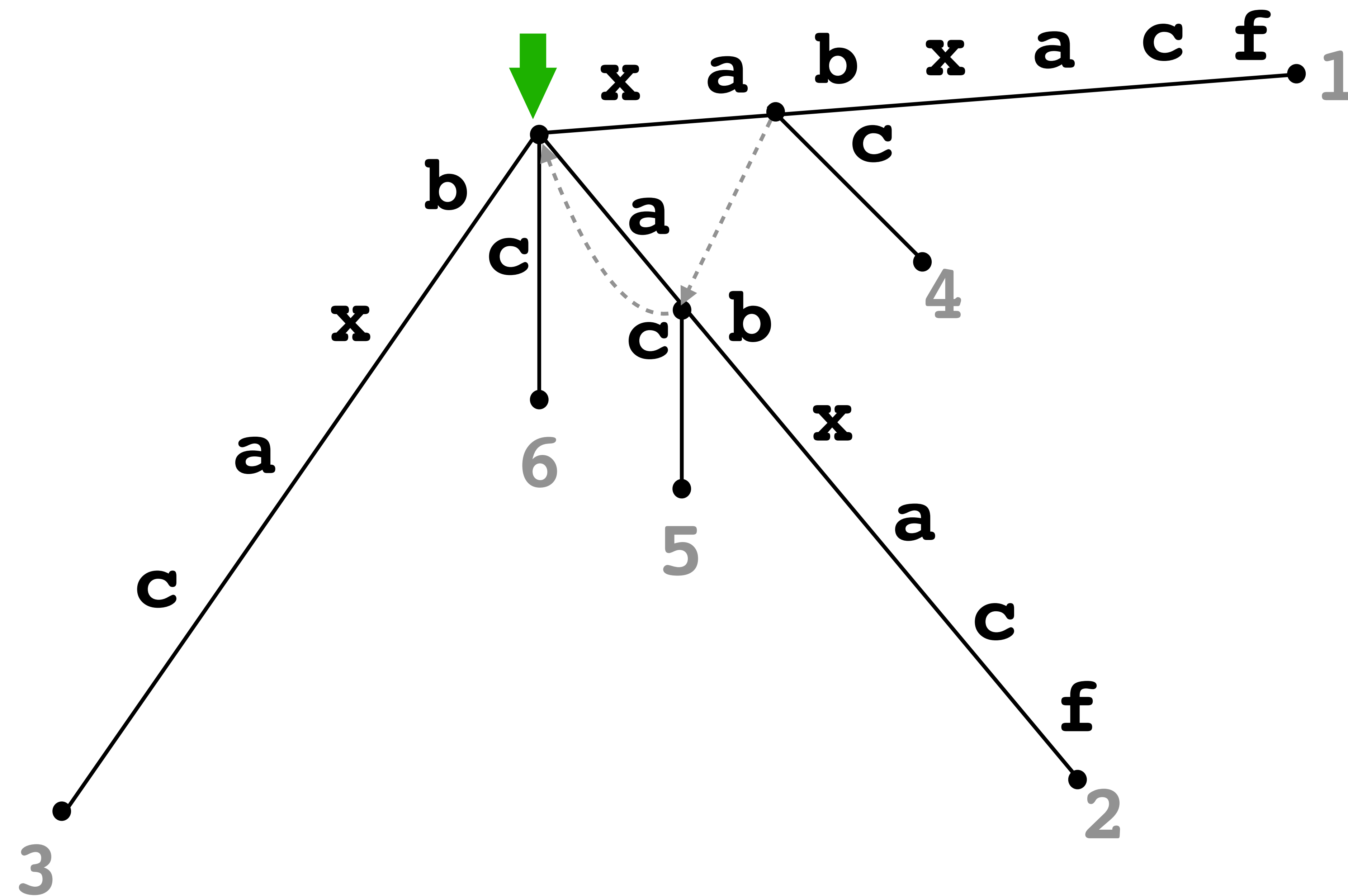
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

1234567
xabxacf



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

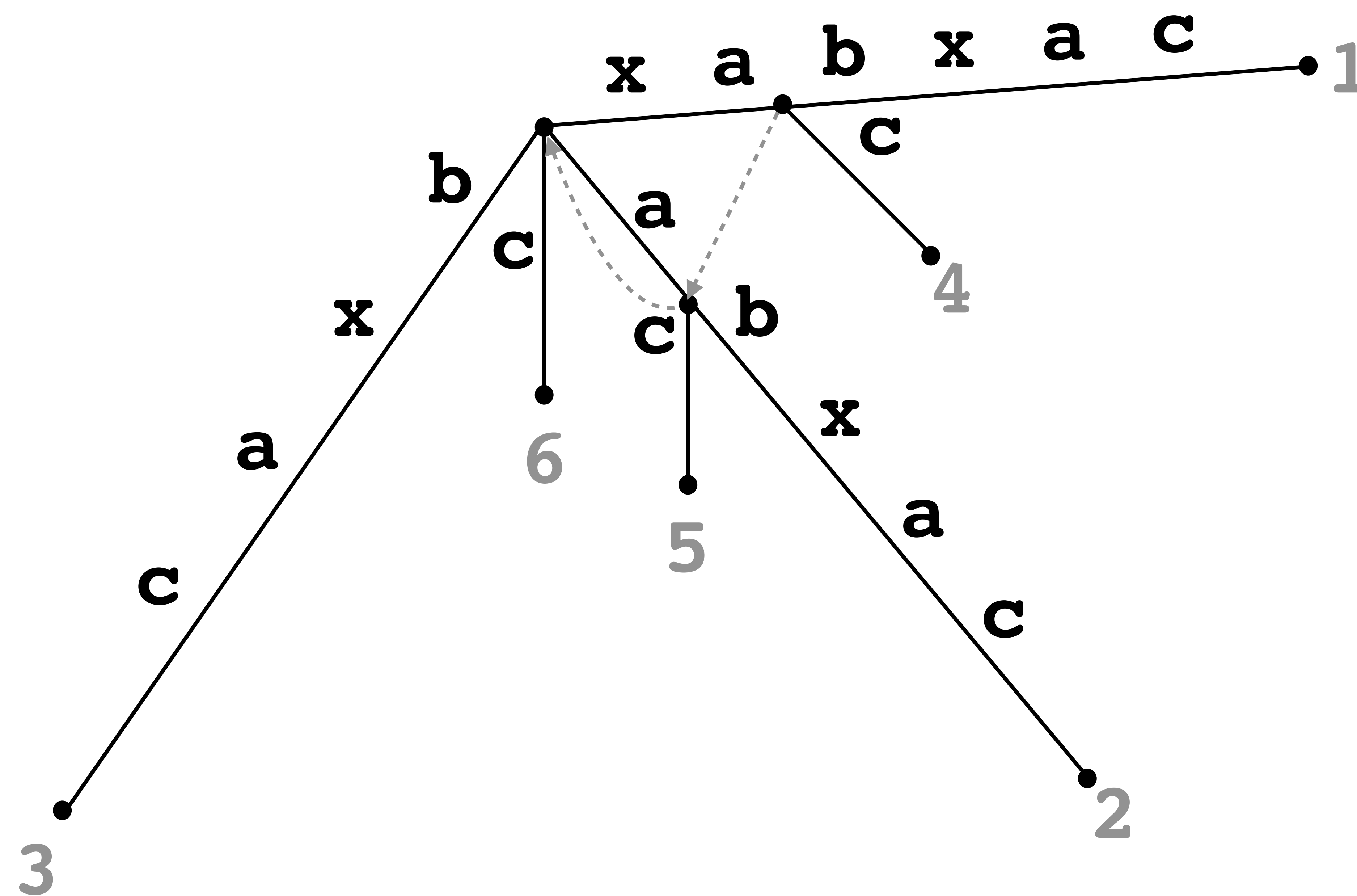
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

a **suffix link** connects an internal node labeled by some string *xa* to an internal node labeled by *a* where *x* is a single character and *a* is an arbitrary (possibly empty) string.

Ukkonen's Algorithm

Concept

123456
xabxac



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

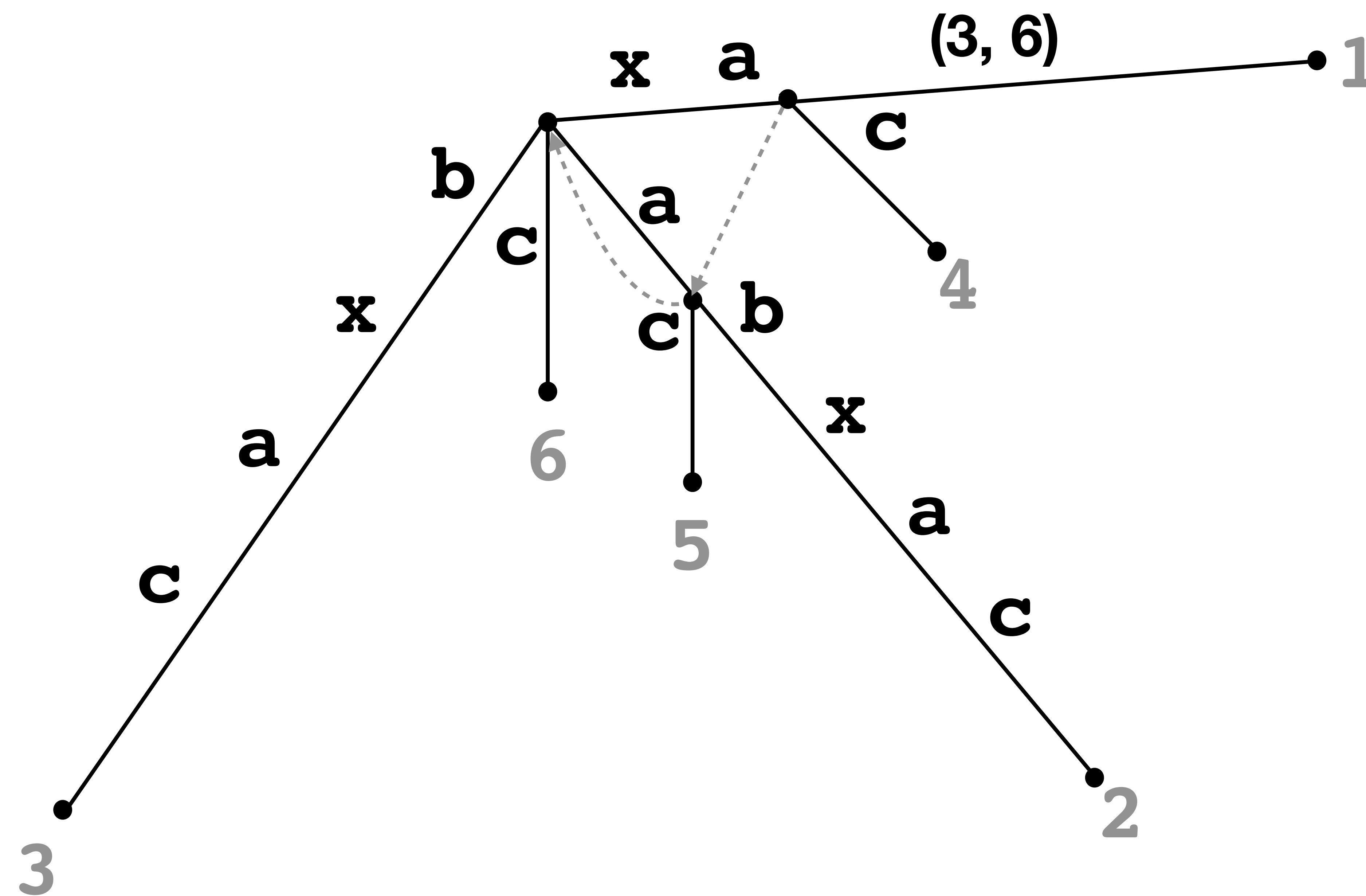
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

node compression is used to save time and space when constructing and using a suffix tree. On each edge store only the start and end indexes rather than substring since they all come from a single string *S.*

Ukkonen's Algorithm

Concept

```
123456
xabxac
```



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

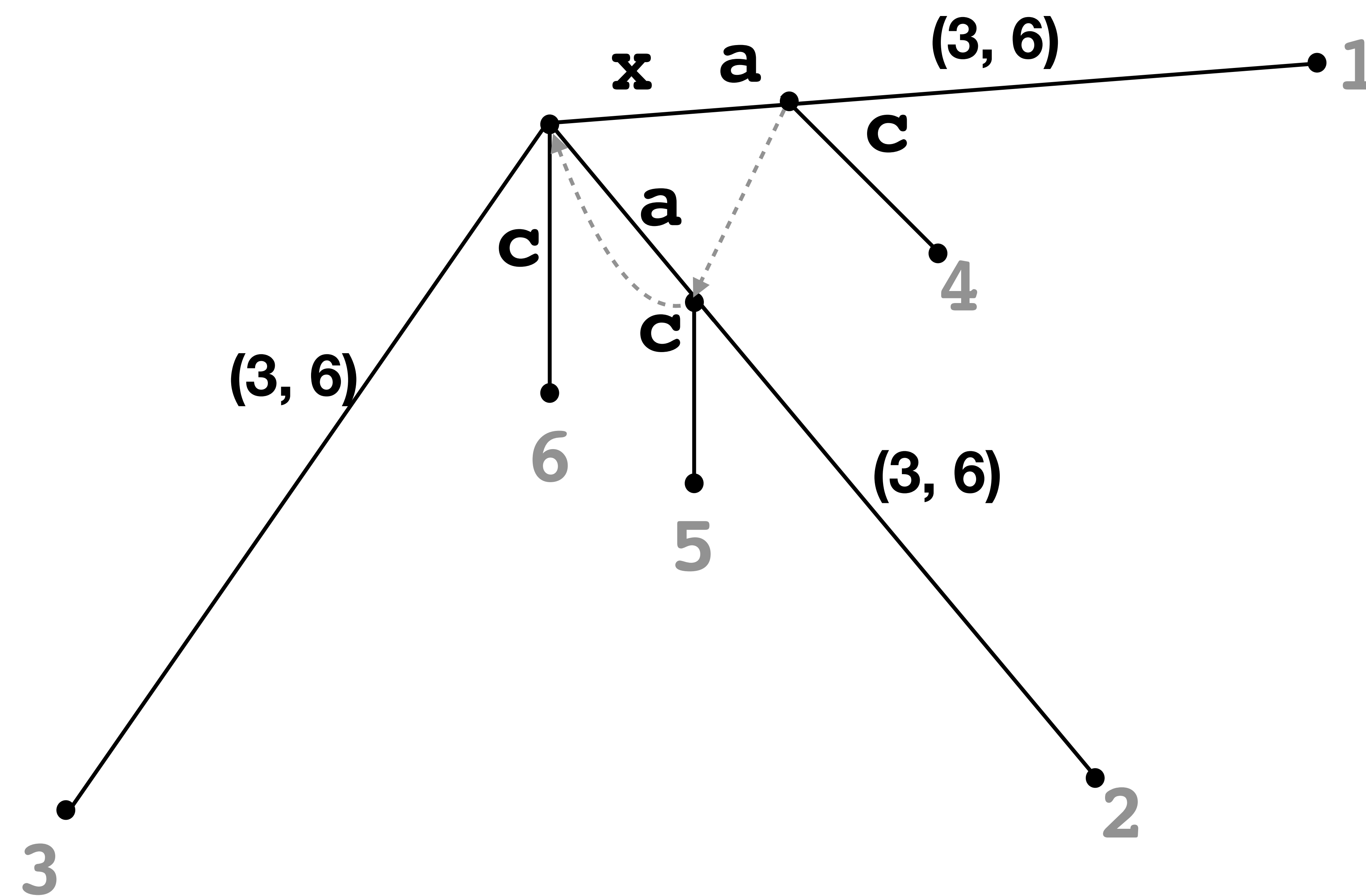
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

node compression is used to save time and space when constructing and using a suffix tree. On each edge store only the start and end indexes rather than substring since they all come from a single string *S.*

Ukkonen's Algorithm

Concept



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

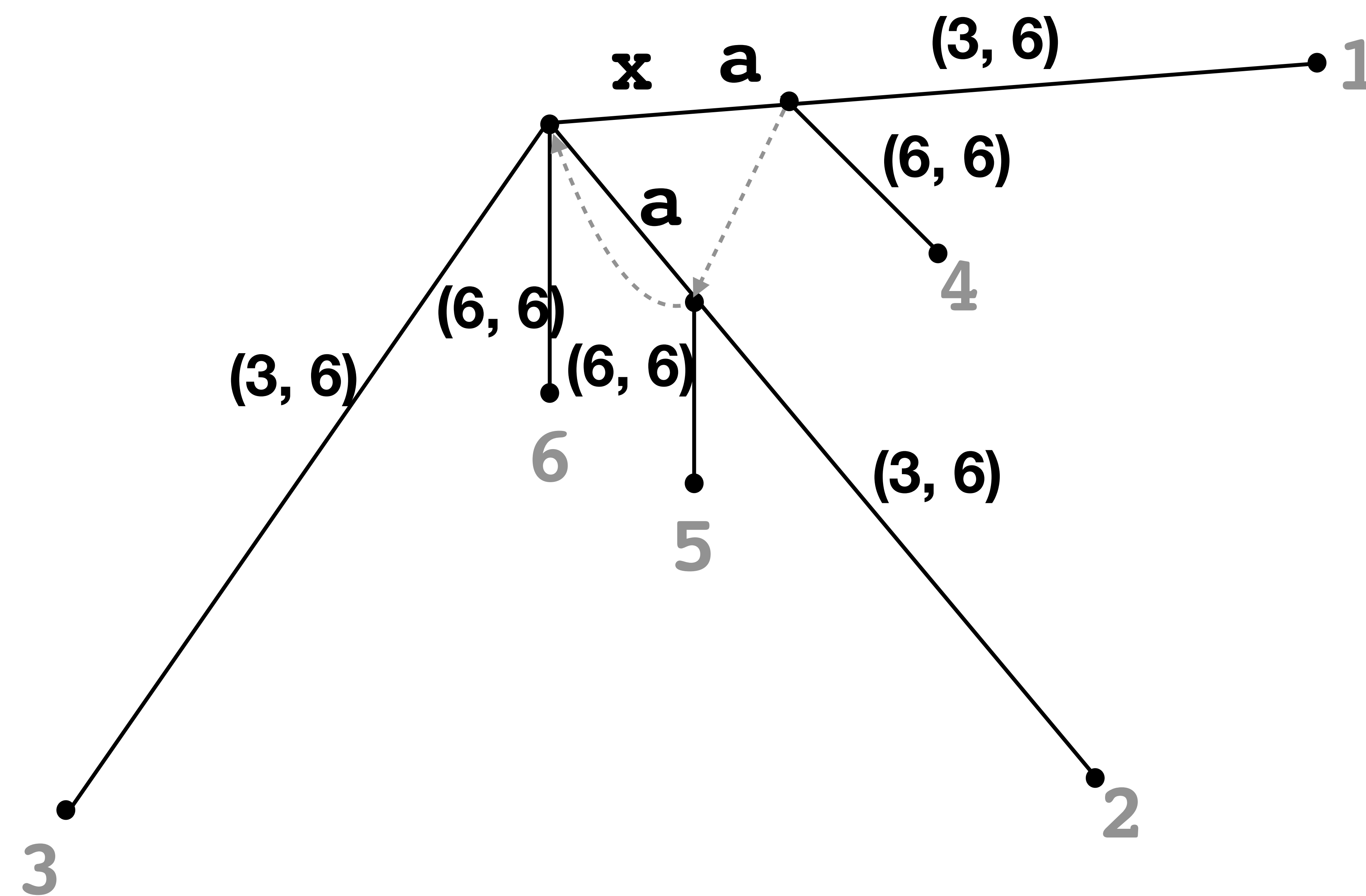
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

node compression is used to save time and space when constructing and using a suffix tree. On each edge store only the start and end indexes rather than substring since they all come from a single string *S.*

Ukkonen's Algorithm

Concept



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

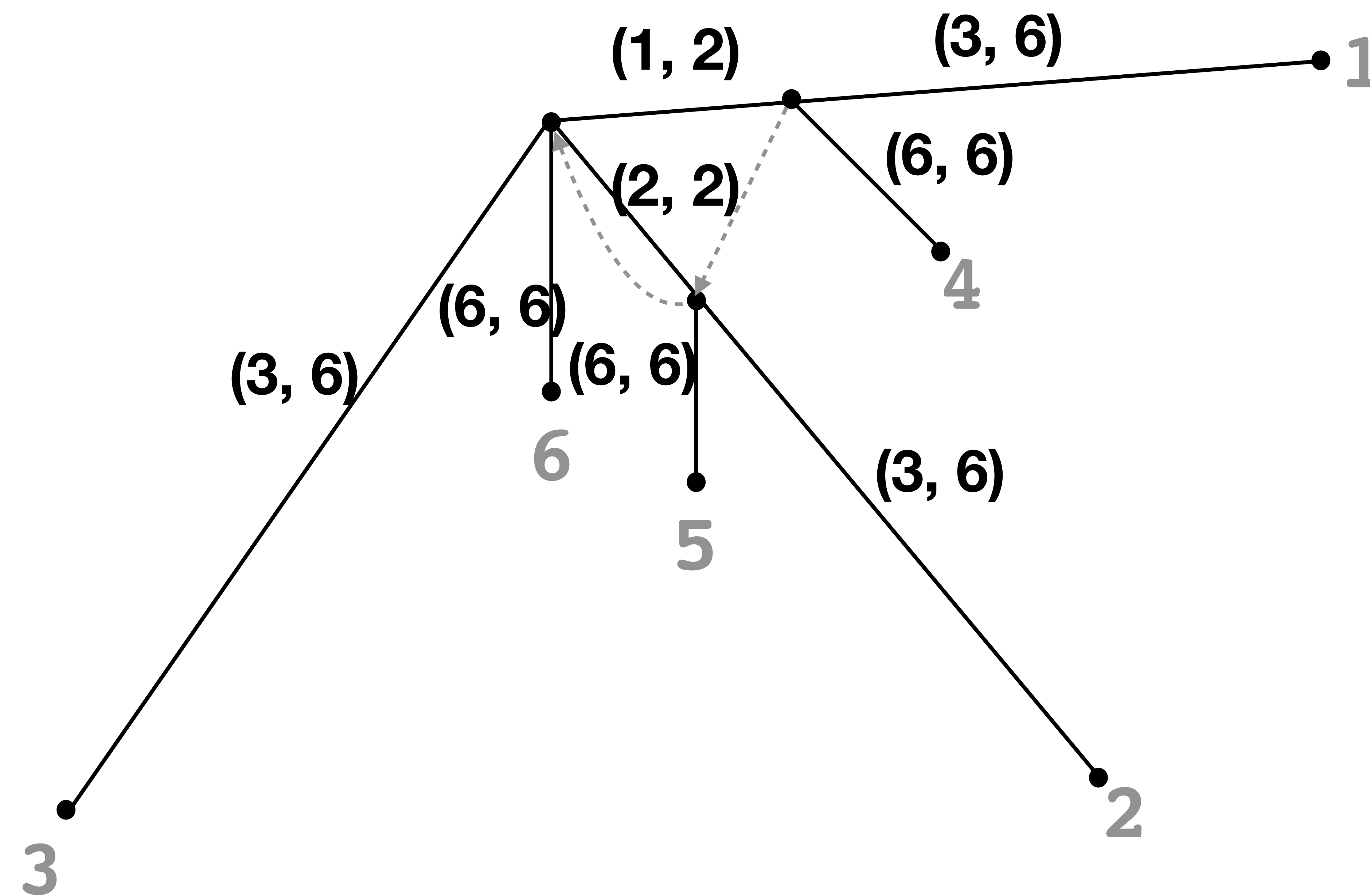
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

node compression is used to save time and space when constructing and using a suffix tree. On each edge store only the start and end indexes rather than substring since they all come from a single string *S.*

Ukkonen's Algorithm

Concept



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

node compression is used to save time and space when constructing and using a suffix tree. On each edge store only the start and end indexes rather than substring since they all come from a single string *S*.

Ukkonen's Algorithm

- Trick 1: the "skip/count trick"
  - when finding a string *f* in the tree (for which we know the string thats all but the last character exists) we can use the length of the string and the edge labels to speed up search
  - at node *v* in the tree, one must start with $f[1]$ call that edge $e = (v,u)$
    - if $|\text{label}(e)| < |f| - 1$ set $v = u$, $f = f[|\text{label}(e)|...|f|]$, repeat
    - if $|\text{label}(e)| = |f| - 1$ extend label(*e*) by $f[|f|]$, end (**Rule 1**)
    - otherwise we know *f* ends somewhere in *e* and we can apply the appropriate rule (**Rule 2** or **Rule 3**)

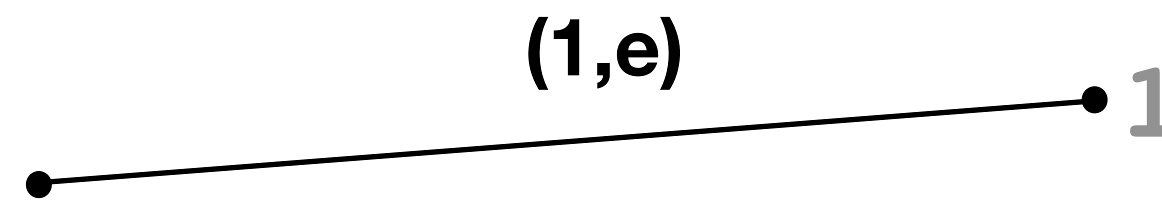
Ukkonen's Algorithm

- Trick 2: Rule 3 is a "show stopper"
  - stop any extension round once **Rule 3** is applied, all further rounds will already be in the tree
  - if *S[i...j]* exists, then *S[i+1...j]*, *S[i+2...j]* since they are suffixes of some other prefix
- Trick 3: Once a leaf, always a leaf
  - let the edge label for leaves end at a global variable ***e*** that will be updated each round
  - **Rule 1** will always apply to these nodes
  - keep a value i_j which is the index before the one where trick 2 started to apply, only examine from i_j+1 on to j

Ukkonen's Algorithm

123456
xabxac

i = 1



e = 1

$i_j = 0$

Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

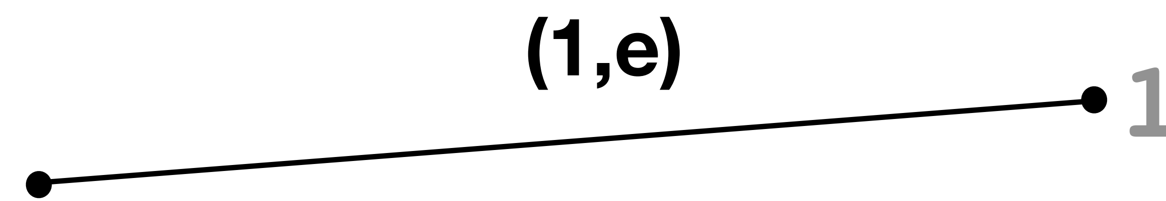
Ukkonen's Algorithm

```
123456
xabxac
```

$i = 1$

$\boldsymbol{e} = 1$

$i_j = 1$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

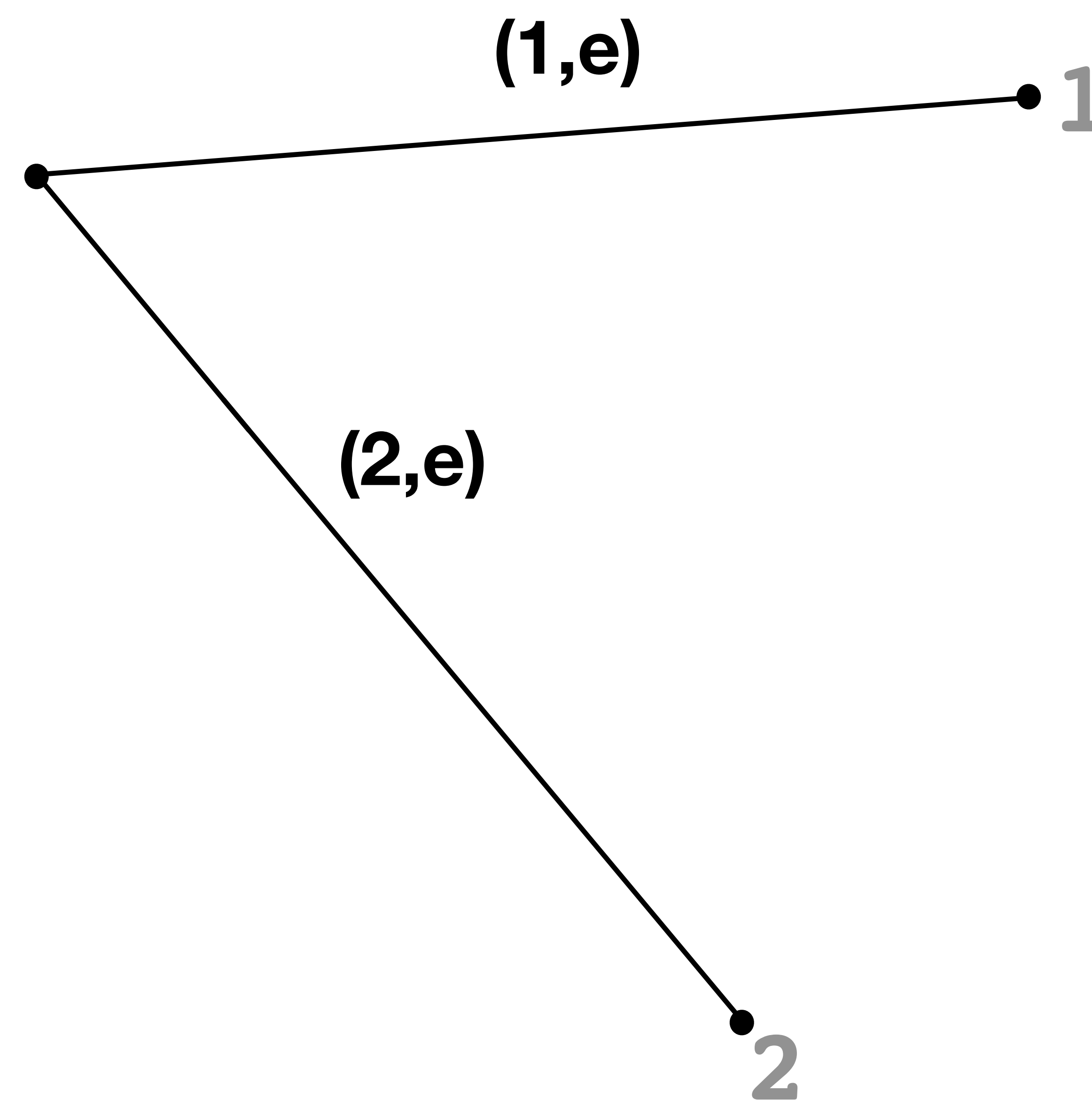
Ukkonen's Algorithm

123456
xabxac

i = 2

e = 2

$i_j = 1$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

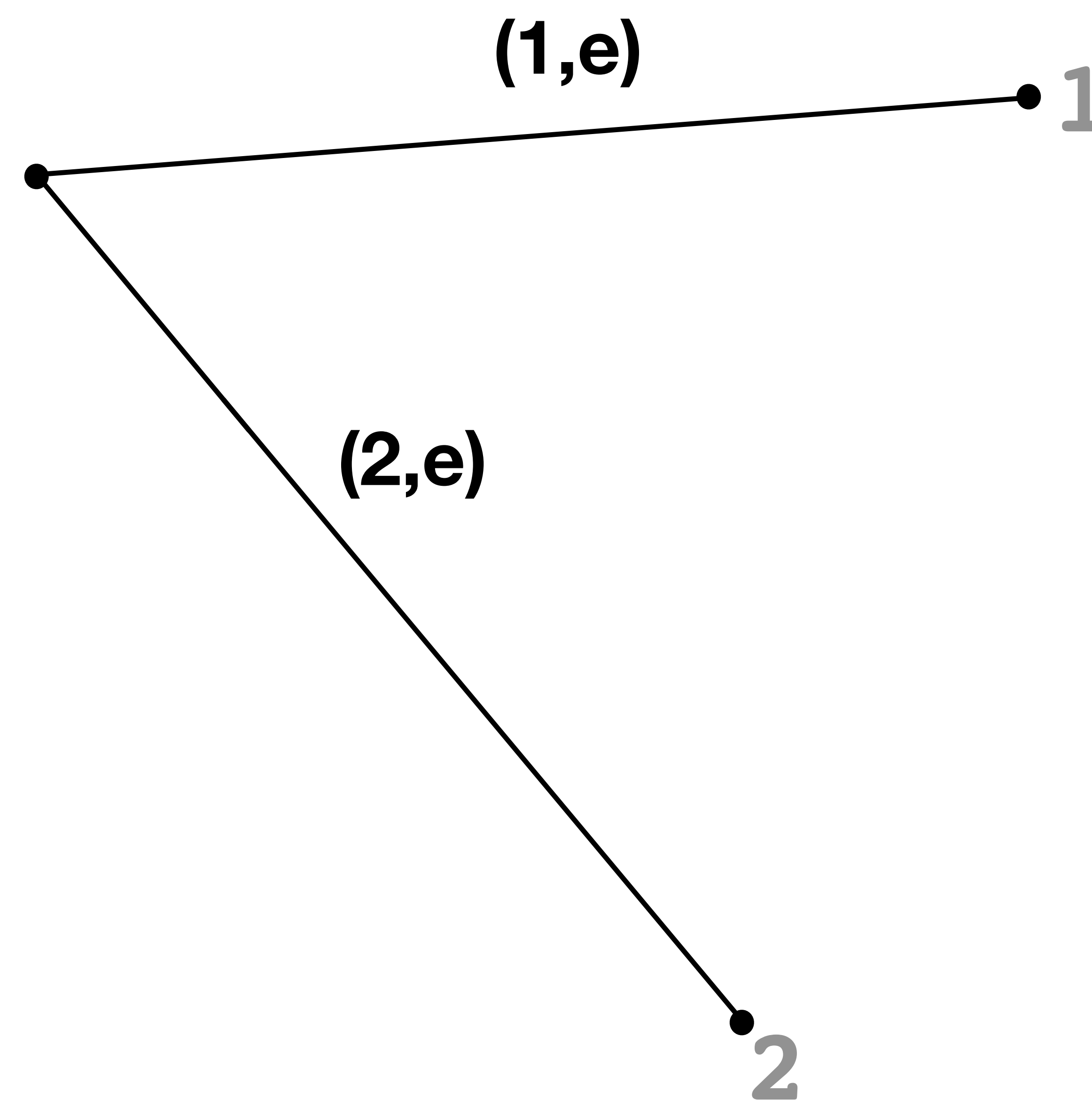
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

123456
xabxac

i = 2

e = 2
$i_j = 2$



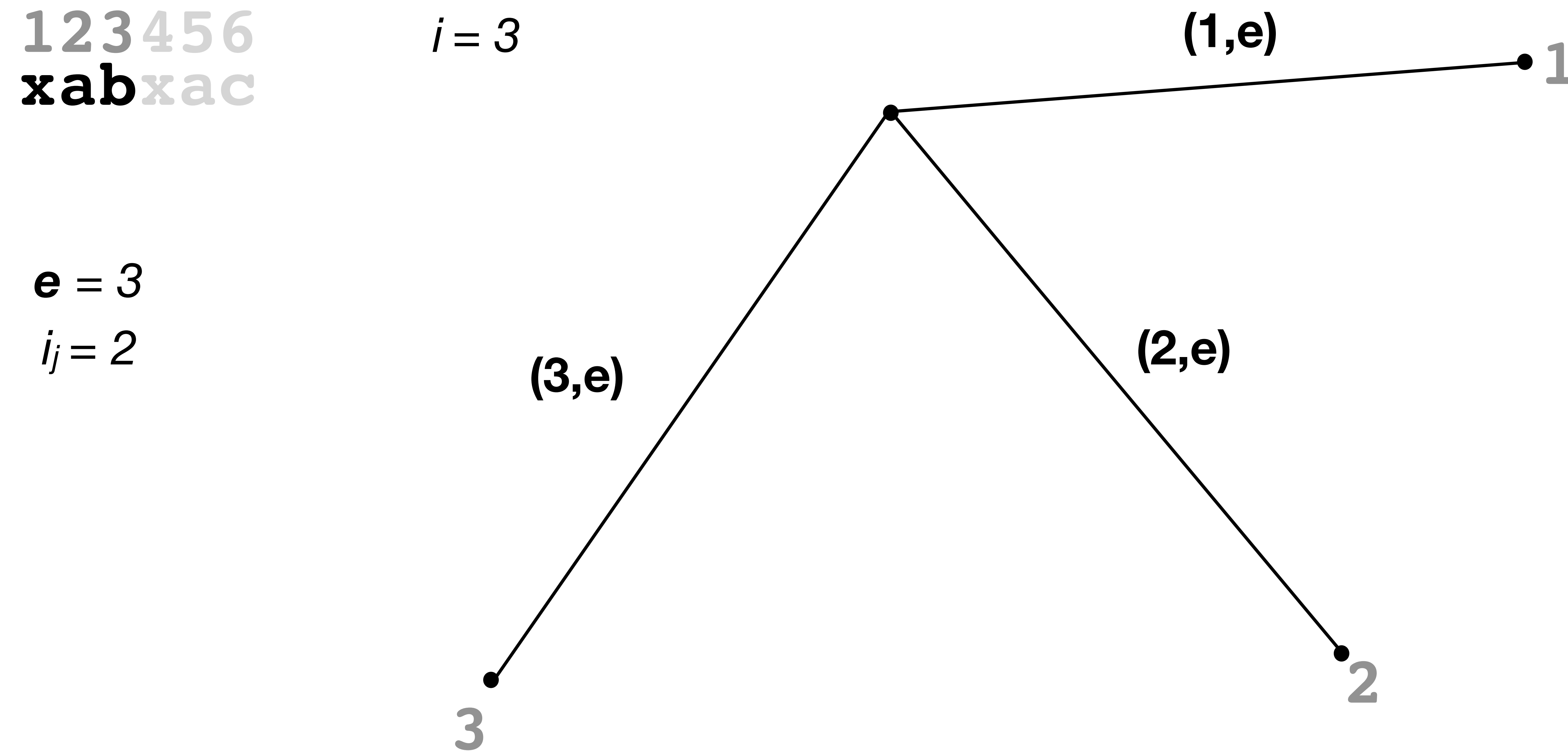
Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

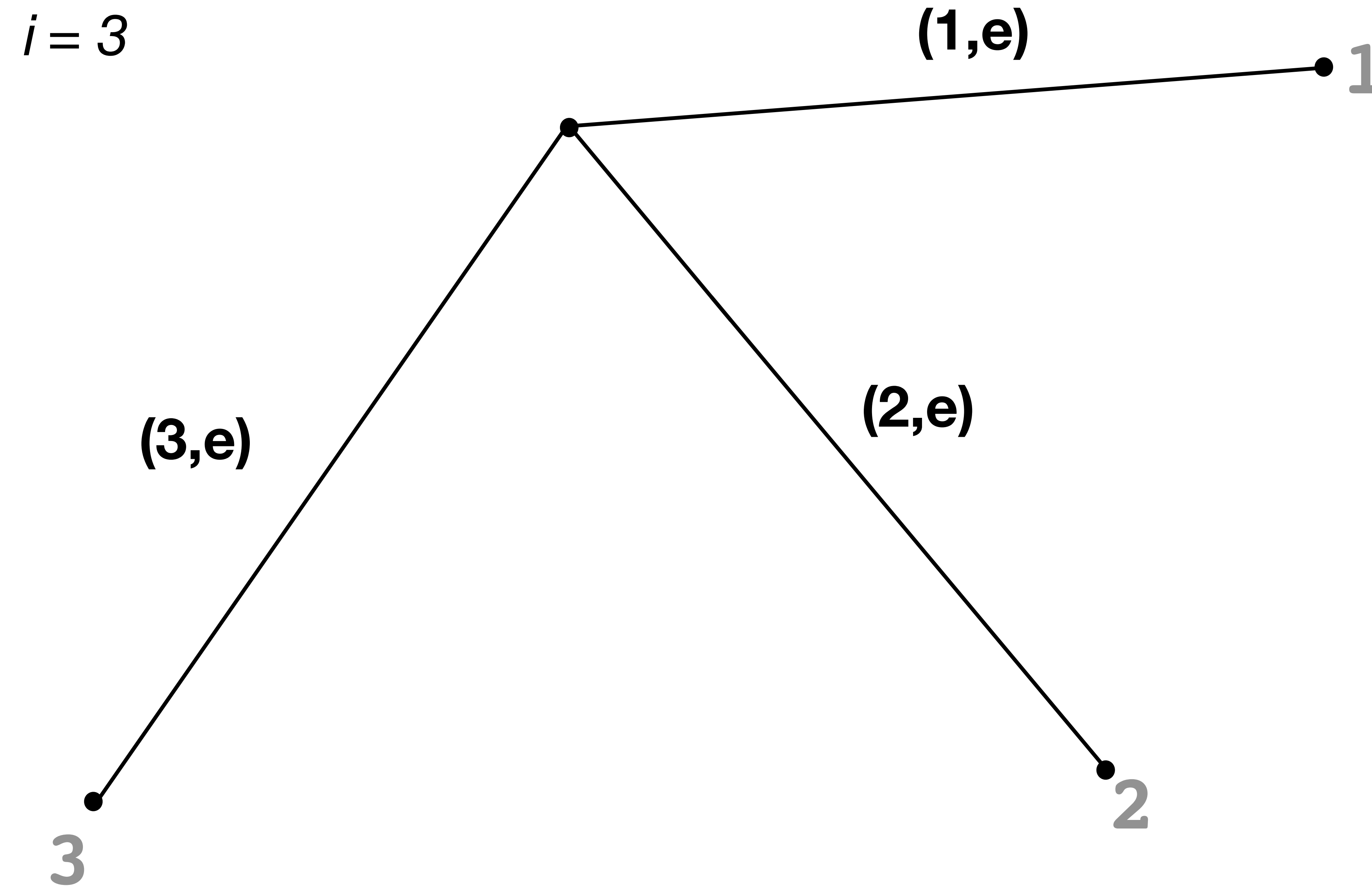
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

123456
xabxac

i = 3

e = 3
$i_j = 3$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

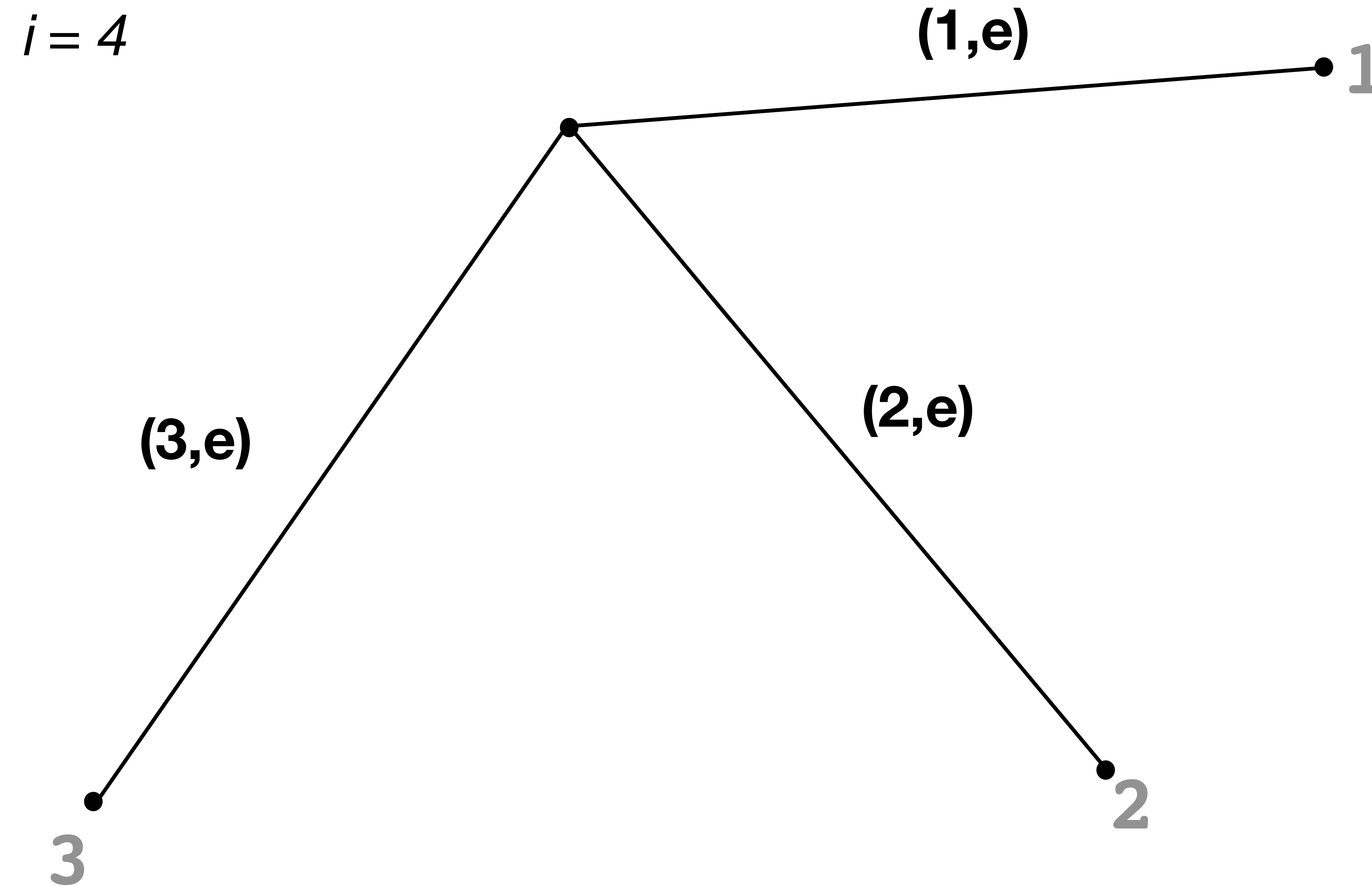
Ukkonen's Algorithm

123456
xabxac

i = 4

e *= 4*

$i_j = 3$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

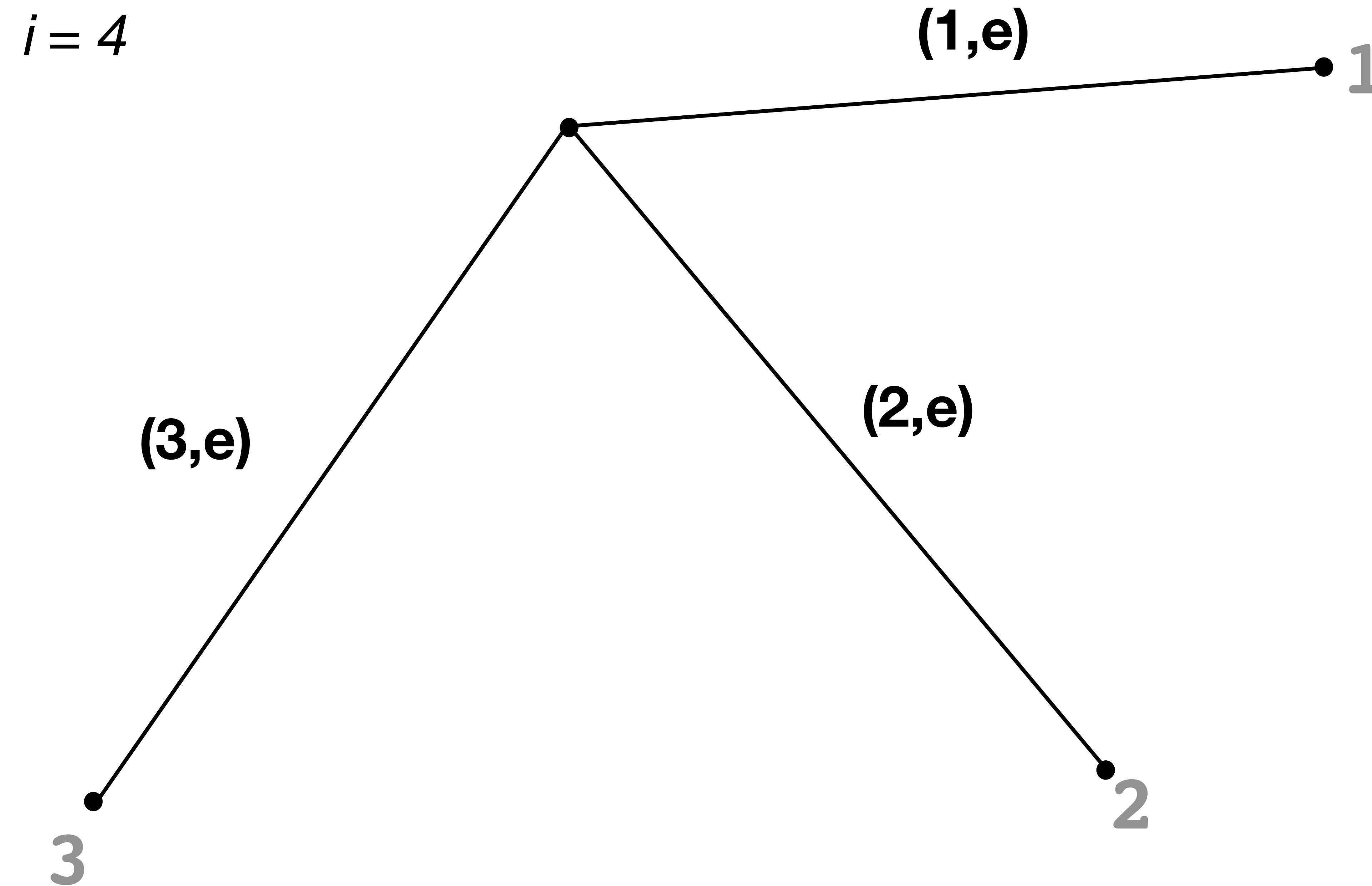
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

123456
xabxac

e $= 5$

$i_j = 3$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

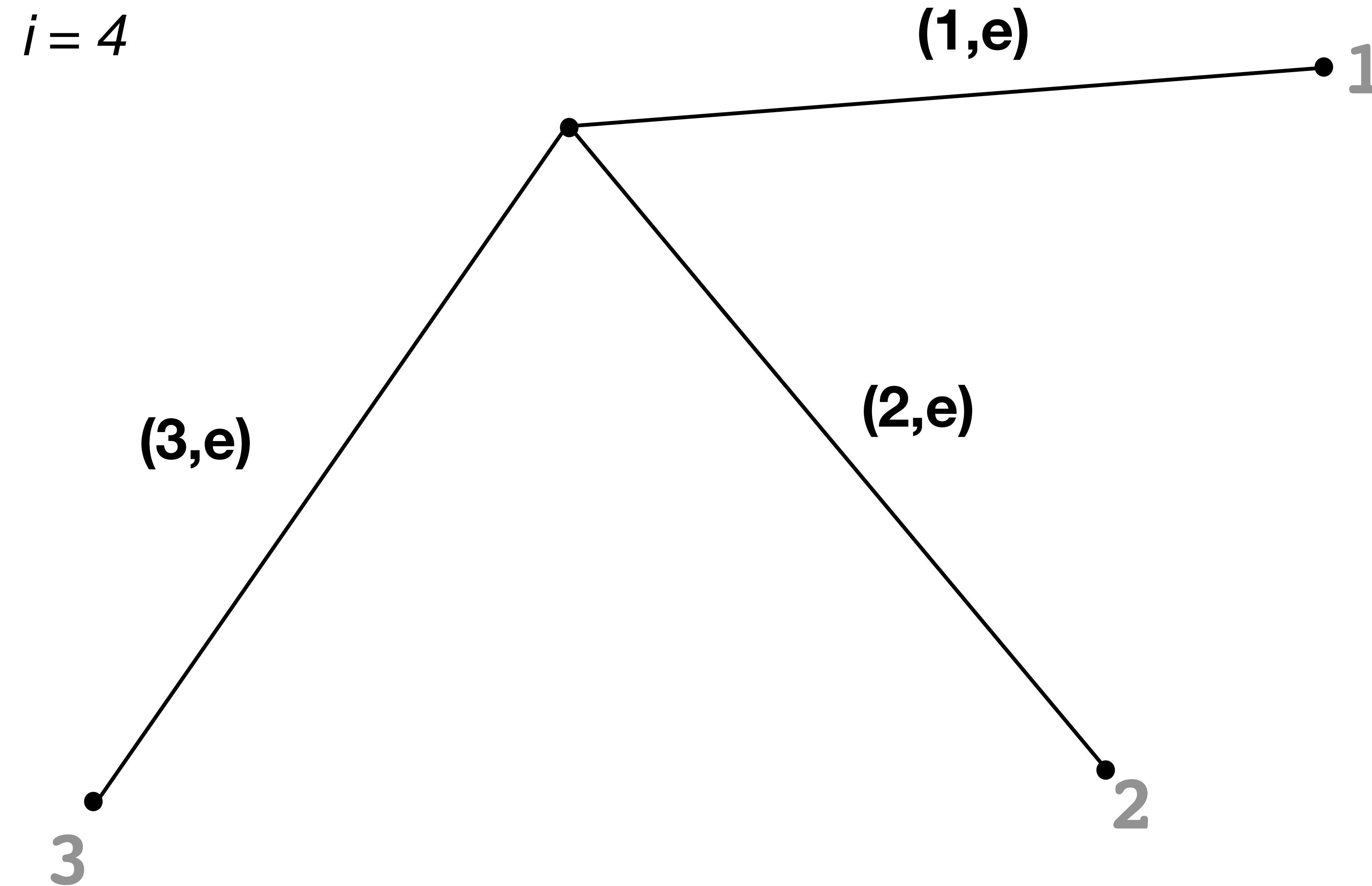
Ukkonen's Algorithm

123456
xabxac

i = 4

e = 6

$i_j = 3$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

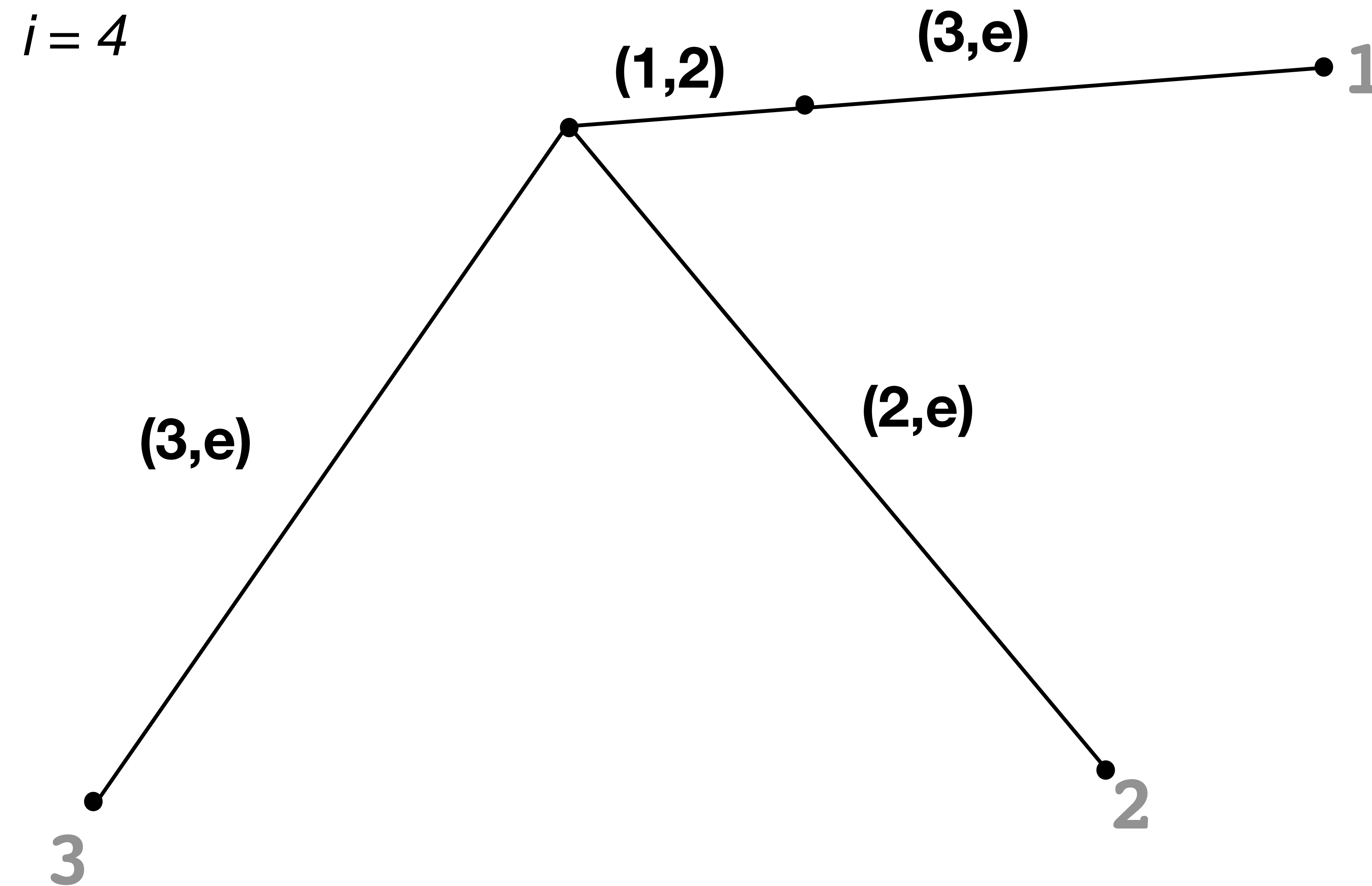
Ukkonen's Algorithm

`123456`
`xabxac`

i = 4

e = 6

$i_j = 3$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

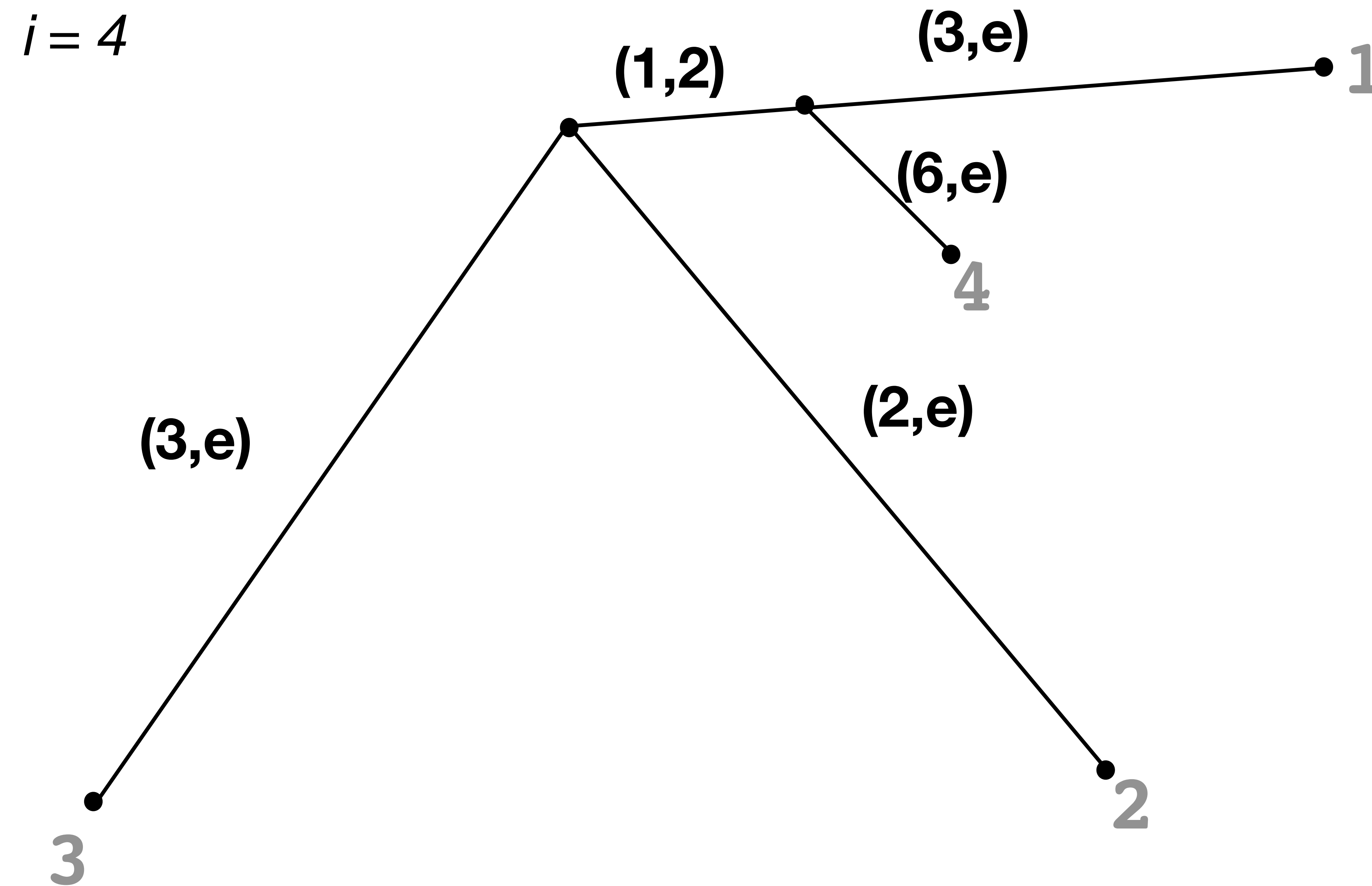
Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

123456
xabxac

e = 6

$i_j = 3$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

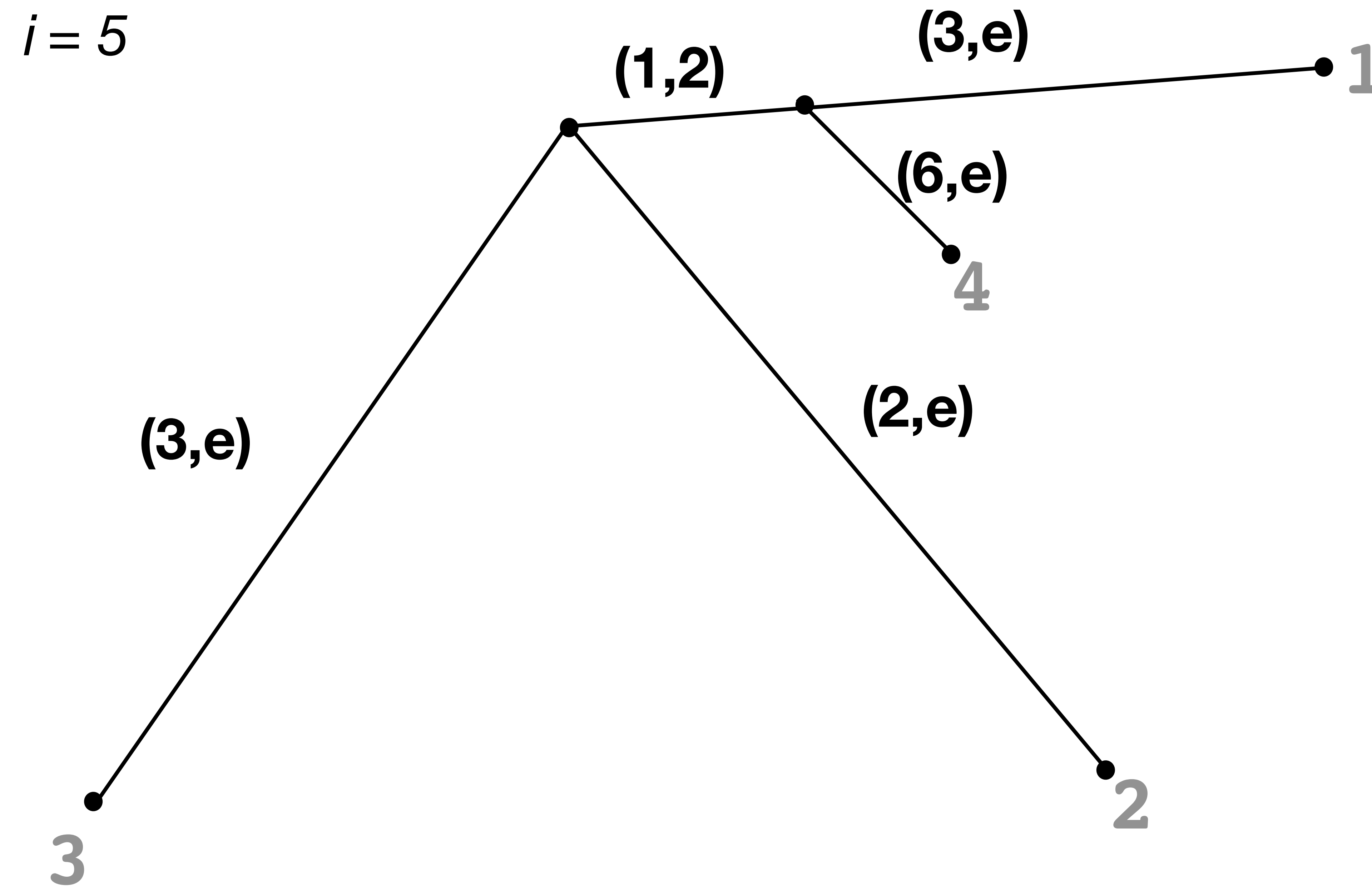
Ukkonen's Algorithm

123456
xabxac

$i = 5$

$\boldsymbol{e} = 6$

$i_j = 3$



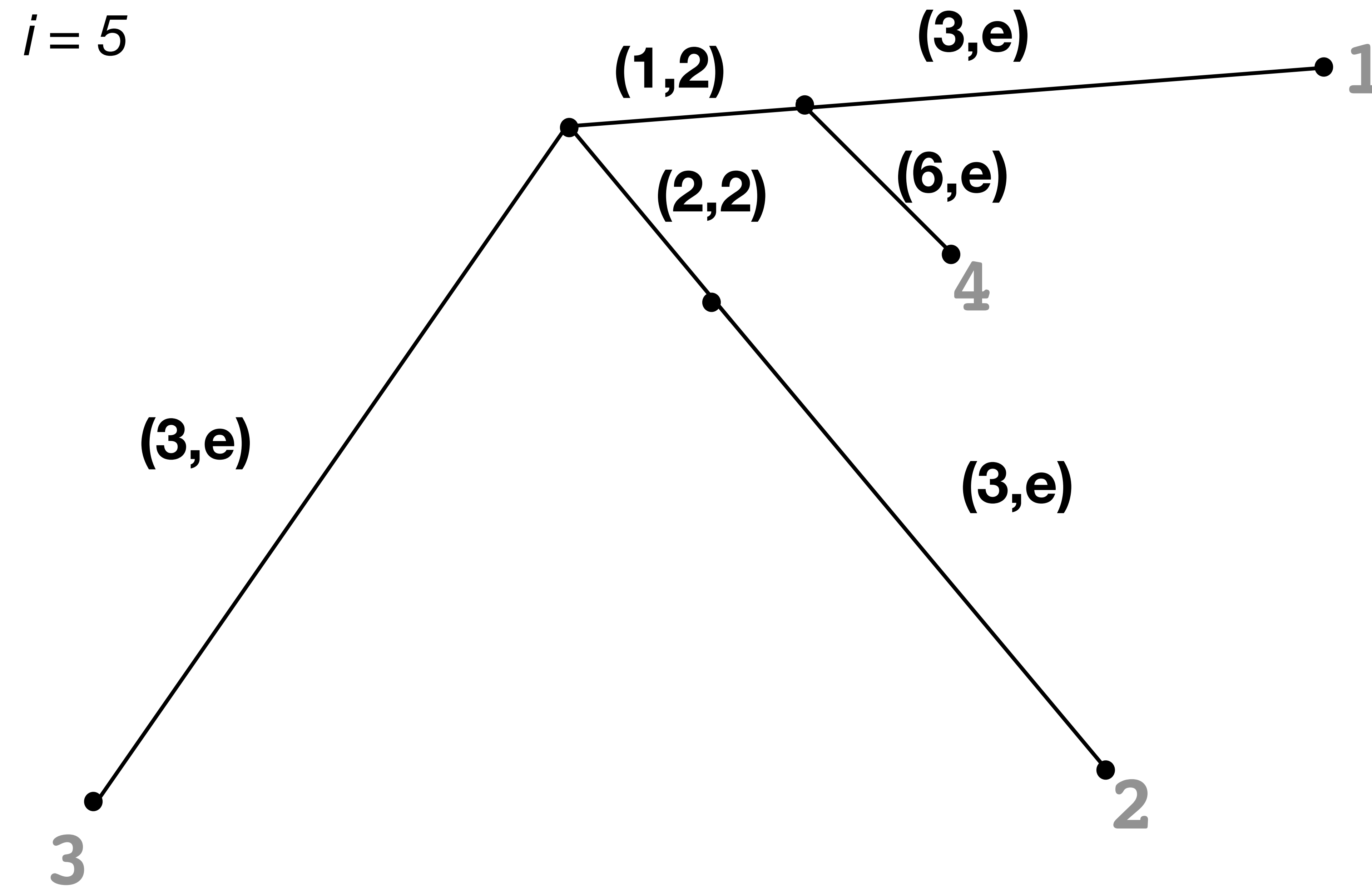
Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

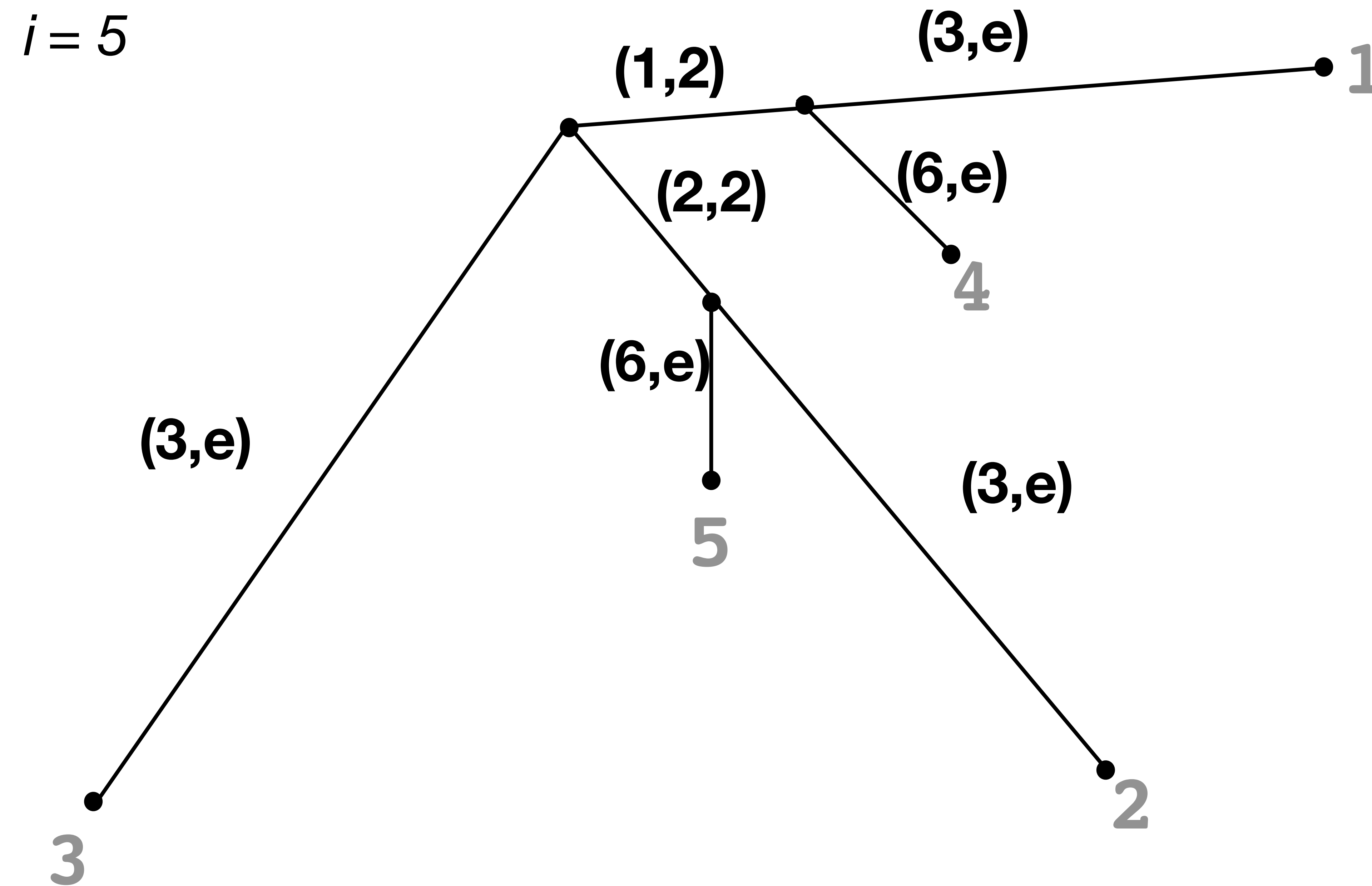
Ukkonen's Algorithm

```
123456
xabxac
```

i = 5

e = 6

$i_j = 3$



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

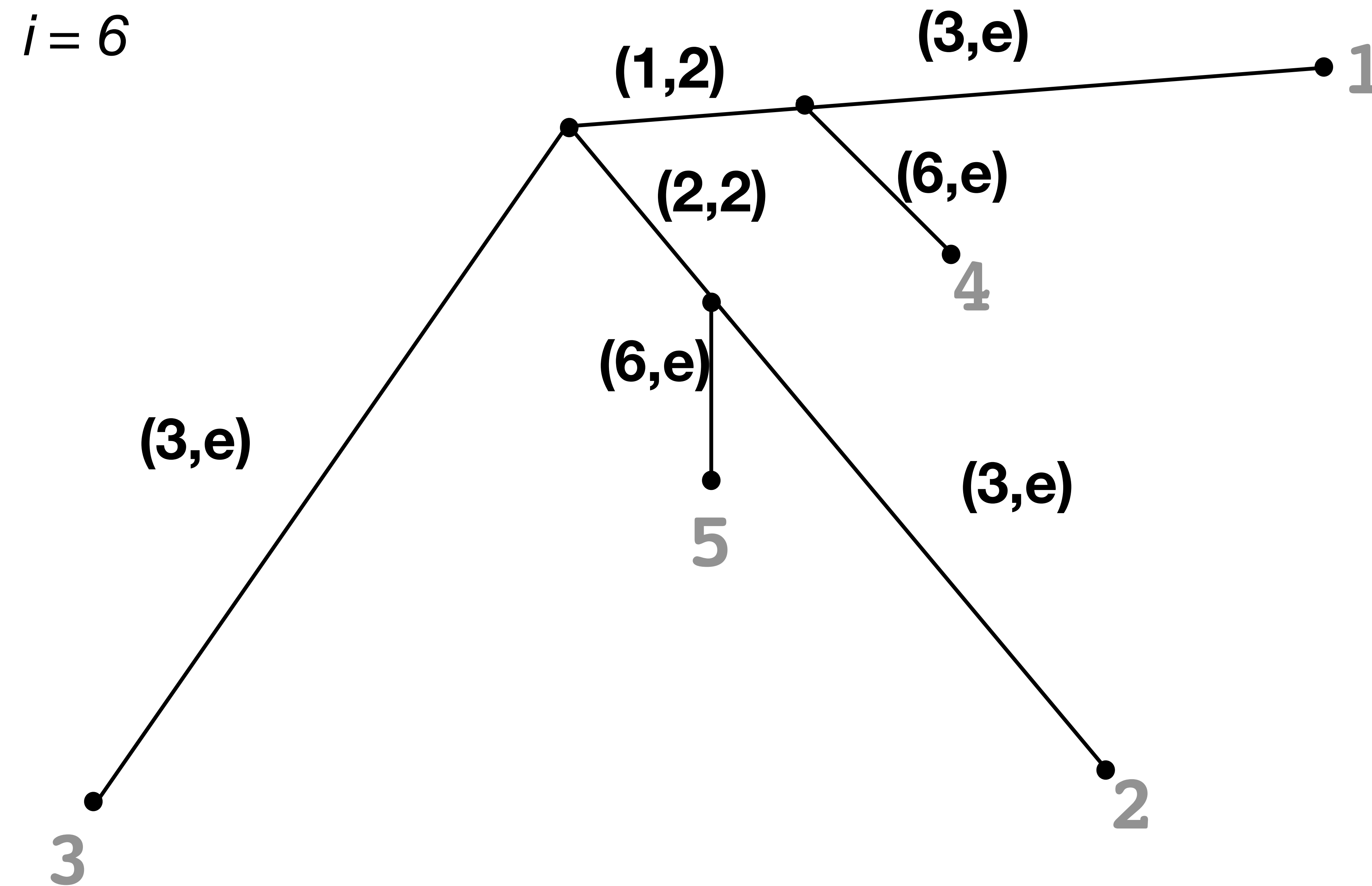
Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

```
123456
xabxac
```

i = 6



e = 6

$i_j = 3$

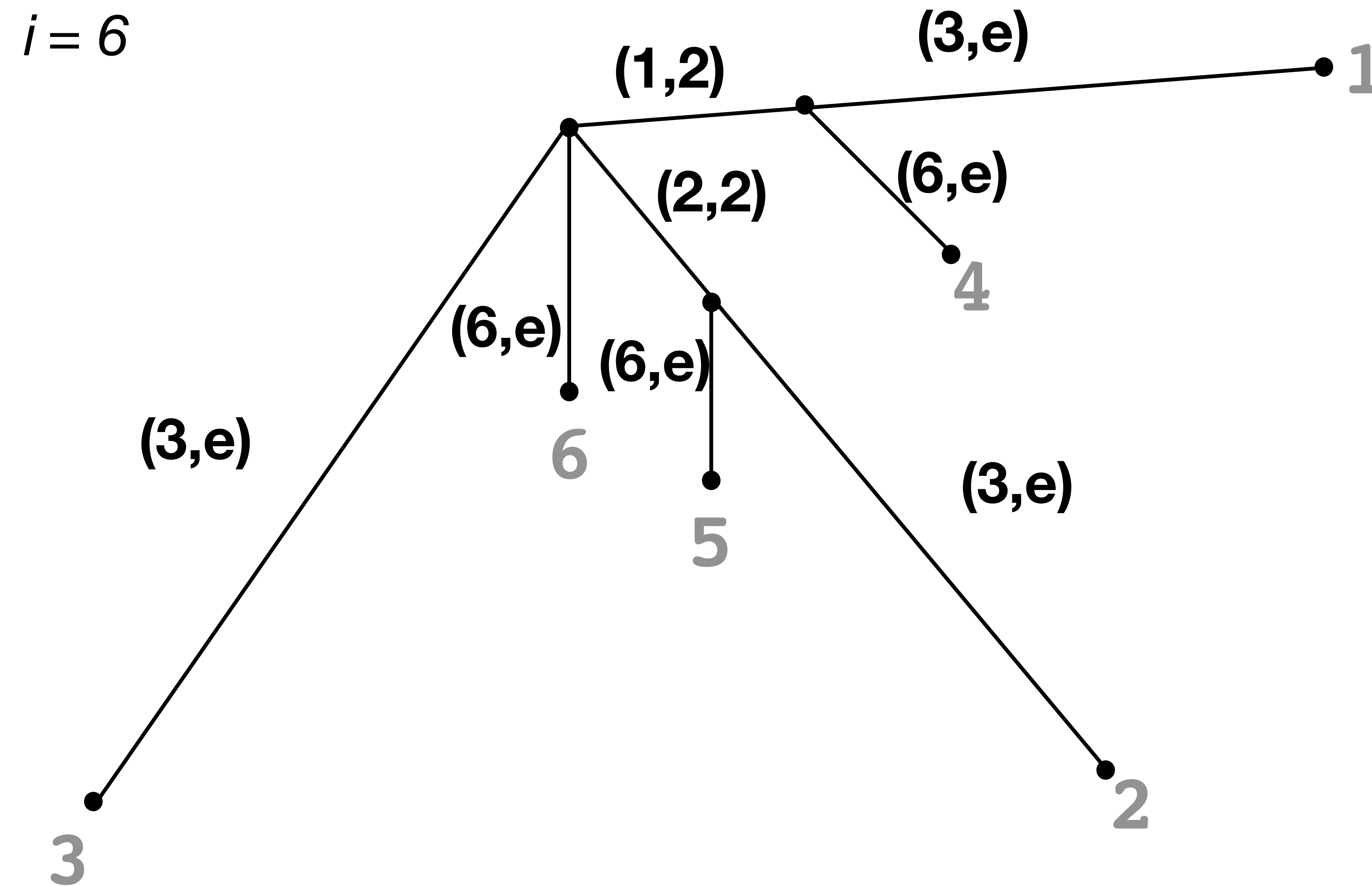
Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm



Three rules for adding suffix *S[i...j+1]* to the implicit tree up to j

Rule 1 In the current tree *S[i...j]* ends at a leaf, append character *S[j+1]* to the label.

Rule 2 *S[i...j]* ends at an internal node or in the middle of a label, and no extension starts with *S[j+1]*, add new leaf.

Rule 3 Some path from *S[i...j]* starts with *S[j+1]*, do nothing.

Ukkonen's Algorithm

- **Theorem** Using suffix links, and tricks 1, 2, and 3, Ukkonen's algorithm builds an implicit suffix tree in $O(m)$ time (for $|S| = m$)
  - **Proof sketch**
    - implicit extensions are constant time per round, so total time is $O(m)$
    - j_i only ever increases, is bounded by m and is always incremented when an explicit extension is performed (i.e. **Rule 2**), therefore only m explicit extensions are performed
    - walking down to get to an explicit extension is a total of at most $2m$ because of the sufflix links, the fact you're always starting at the last explicit extension that was made, and the tree depth being $O(m)$

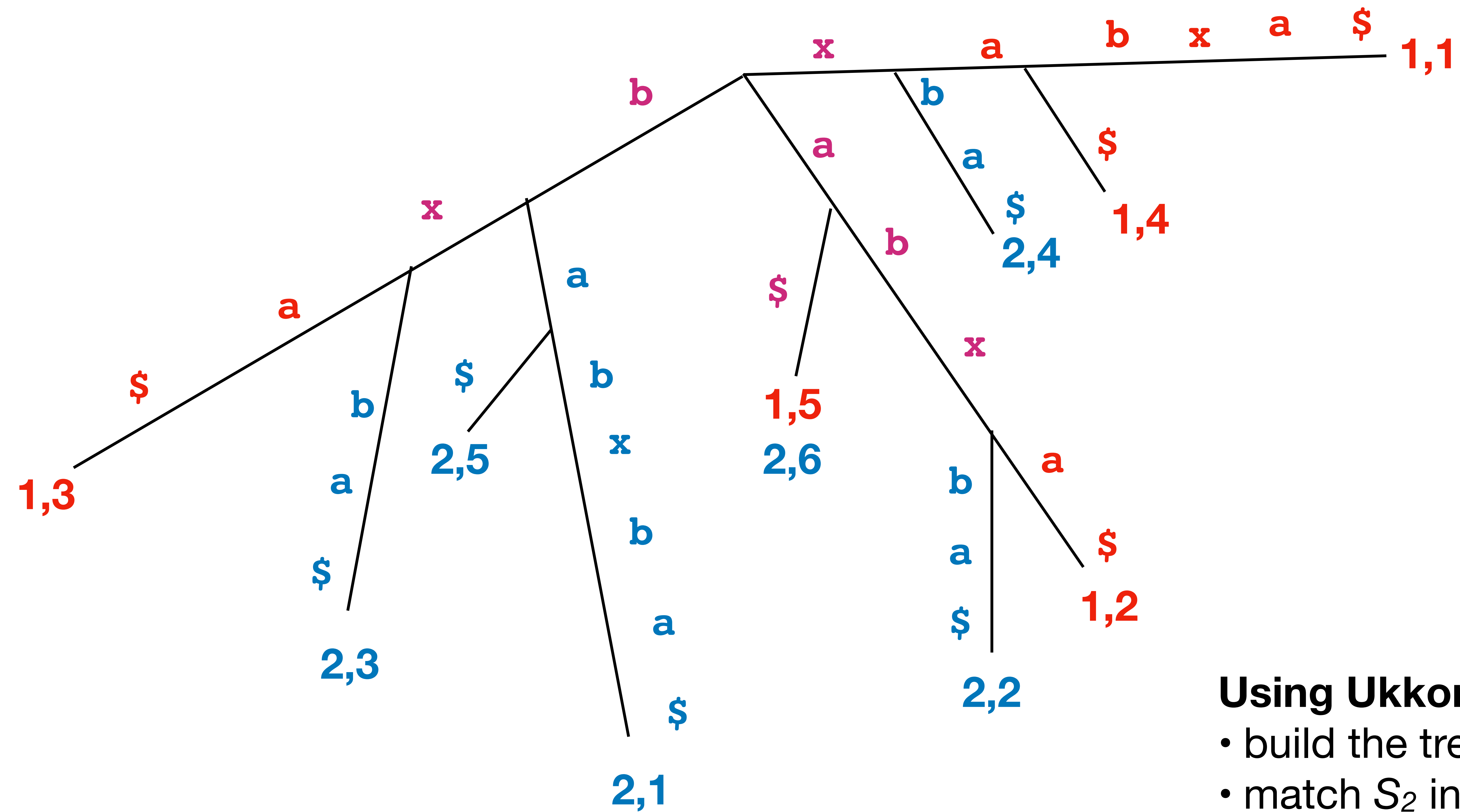
Ukkonen's Algorithm

- Making a true suffix tree is still $O(m)$
  - create the implicit suffix tree for $S\$$ (which takes $O(m+1)$ time)
  - make a single pass to convert the variable ***e*** to a number

Generalized Suffix Trees

```
     123456
```
S_1 = xabxa

S_1 = babxba



Using Ukkonen's Algorithm

- build the tree for S_1
- match S_2 in the tree until a mismatch is found at $S_2[j]$
- restart the Ukkonen algorithm from j (all suffixes of $S[1...j-1]$ are already in the tree)
- repeat for S_3, S_4, ... , S_k

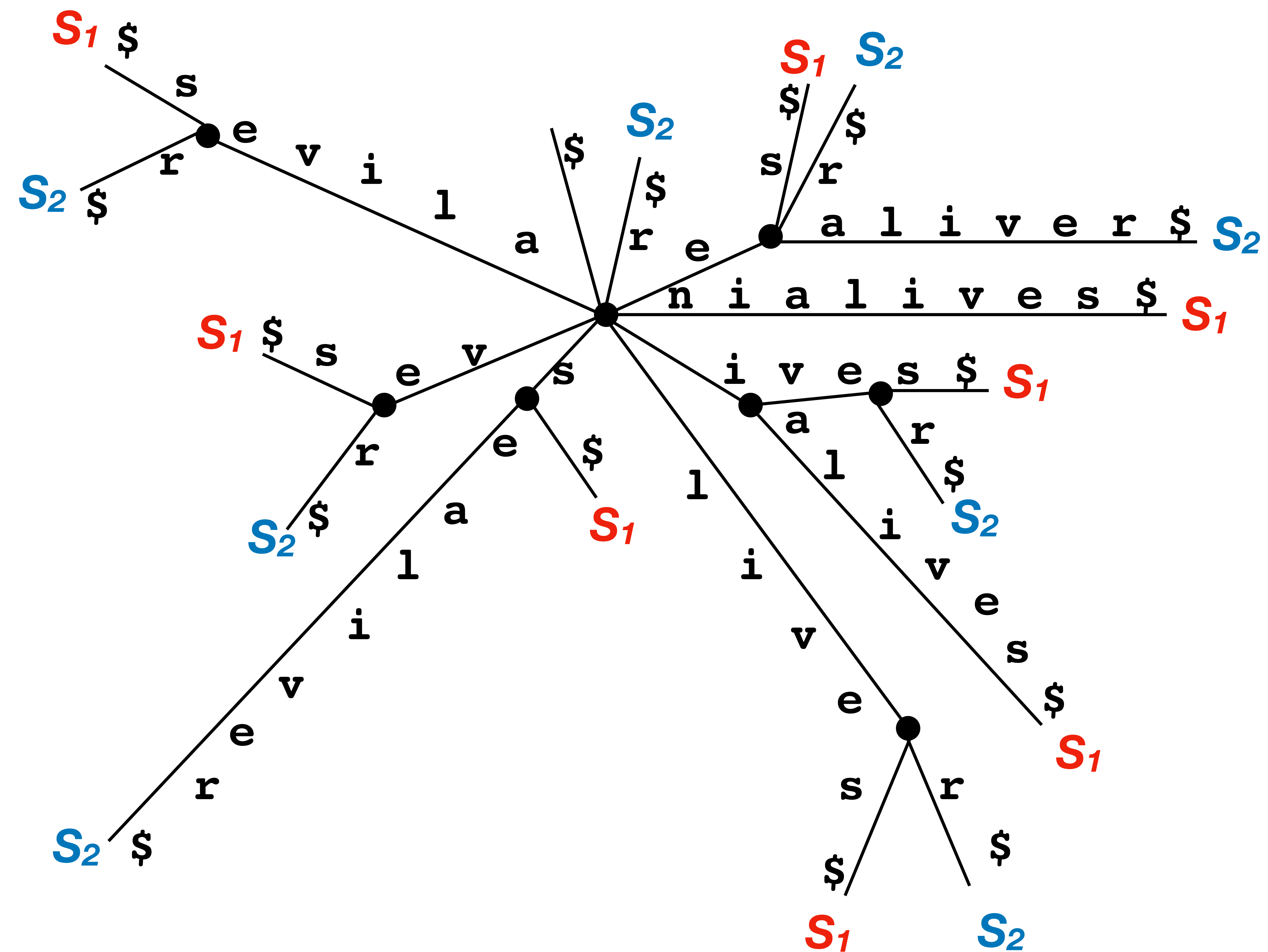
Longest Common Substring Problem

- Given two sequences S_1 and S_2 find the longest common substring between the two.
  - That is, find the largest k such that for some locations $i<|S_1|$ and $j<|S_1|$ such that $S_1[i...(i+k)] = S_2[j...(j+k)]$.
- Example S_1 = `californialives` S_2 = `sealiver`
  - $k = 5$, $i = 10$, $j = 3$ (`alive`)

Longest Common Substring Problem

S_1 = `nialives$`

S_2 = `sealiver$`



Suffix Arrays

- How much space does a suffix tree over an alphabet $|\Sigma| = \sigma$ consume?
  - If each internal node contains a σ length array of pointers its $O(m\ \sigma)$.

- Manber and Myers show that the same running time for algorithms on suffix trees can be achieved while only storing a single array of integers, called a *suffix array*.

Suffix Arrays

A suffix array contains the starting position of the suffixes of a string when listed in lexicographic order.

```
s = mississippi

11: i
 8: ippi
 5: issippi
 2: ississippi
 1: mississippi
10: pi
 9: ppi
 7: sippi
 4: sissippi
 6: ssippi
 3: ssissippi
```

Suffix Arrays

Is `sip` is contained in `mississippi`?

`s = mississippi`

11:	`i`
8:	`ippi`
5:	`issippi`
2:	`ississippi`
1:	`mississippi`
10:	`pi`
9:	`ppi`
7:	`sippi`
4:	`sissippi`
6:	`ssippi`
3:	`ssissippi`

Suffix Arrays

`s = mississippi`

Binary search!

```
11: i
 8: ippi
 5: issippi
 2: ississippi
 1: mississippi
10: pi
 9: ppi
 7: sippi
 4: sissippi
 6: ssippi
 3: ssissippi
```

Is `sip` is contained in `mississippi?`

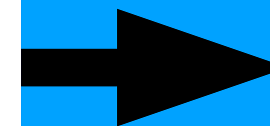
Suffix Arrays

`s = mississippi`

Binary search!

Is `sip` is contained in `mississippi?`

s[10...] =? sip



11: `i`
8: `ippi`
5: `issippi`
2: `ississippi`
1: `mississippi`
10: `pi`
9: `ppi`
7: `sippi`
4: `sissippi`
6: `ssippi`
3: `ssissippi`

Suffix Arrays

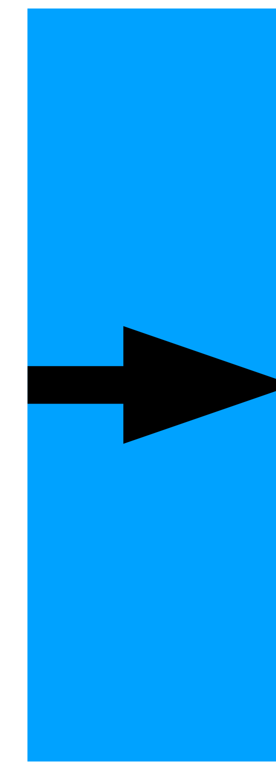
`s = mississippi`

Binary search!

11:	i
8:	ippi
5:	issippi
2:	ississippi
1:	mississippi
10:	pi
9:	ppi
7:	sippi
4:	sissippi
6:	ssippi
3:	ssissippi

Is `sip` is contained in `mississippi?`

s[4...] =? sip



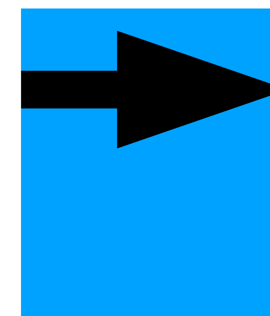
Suffix Arrays

s = mississippi

Binary search!

Is `sip` is contained in `mississippi?`

s[9...] =? sip



11: i
8: ippi
5: issippi
2: ississippi
1: mississippi
10: pi
9: ppi
7: sippi
4: sissippi
6: ssippi
3: ssissippi

Suffix Arrays

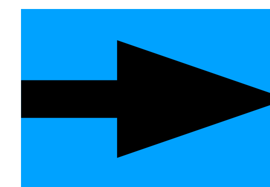
s = mississippi

Binary search!

11:	i
8:	ippi
5:	issippi
2:	ississippi
1:	mississippi
10:	pi
9:	ppi
7:	sippi
4:	sissippi
6:	ssippi
3:	ssissippi

Is `sip` is contained in `mississippi?`

s[7...] =? sip



Suffix Arrays

One more concept:

lcp(i,j) for positions *i* and *j* is the length of the longest common prefix of the suffixes at position *i* and *j* in the suffix array

lcp(10,11) = 3 (`ssi`)
lcp(8,11) = 1 (`s`)

The *lcp* for non-adjacent positions is the minimum of adjacent *lcp* values between the two positions

lcp(8,9) = 2
lcp(9,10) = 1
lcp(10,11) = 3

`s = mississippi`

11:	`i`	**1**
8:	`ippi`	**1**
5:	`issippi`	**4**
2:	`ississippi`	**0**
1:	`mississippi`	**0**
10:	`pi`	**1**
9:	`ppi`	**0**
7:	`sippi`	**2**
4:	`sissippi`	**1**
6:	`ssippi`	**3**
3:	`ssissippi`	**-**

Suffix Arrays

Find all occurrences of `issi` in `mississippi`.

`s = mississippi`

		index	suffix	lcp
T lcp with p = 1		11:	`i`	**1**
		8:	`ippi`	**1**
		5:	`issippi`	**4**
		2:	`ississippi`	**0**
		1:	`mississippi`	**0**
M		10:	`pi`	**1**
	lcp(T,M) = 0	9:	`ppi`	**0**
		7:	`sippi`	**2**
		4:	`sissippi`	**1**
		6:	`ssippi`	**3**
B lcp with p = 0		3:	`ssissippi`	**-**

Suffix Arrays

S = mississippi

T lcp with p = 1	11:	i	1
	8:	ippi	1
M	5:	issippi	4
	2:	ississippi	0
	1:	mississippi	0
B lcp with p = 0	10:	pi	1
	9:	ppi	0
	7:	sippi	2
	4:	sissippi	1
	6:	ssippi	3
	3:	ssissippi	-

lcp(T,M) = 1

Find all occurrences of `issi` in `mississippi`.

Suffix Arrays

s = mississippi

Find all occurrences of `issi` in `mississippi`.

T lcp with p = 4

B lcp with p = 0

Index	Suffix	LCP
11:	i	1
8:	ippi	1
5:	issippi	4
2:	ississippi	0
1:	mississippi	0
10:	pi	1
9:	ppi	0
7:	sippi	2
4:	sissippi	1
6:	ssippi	3
3:	ssissippi	-

Suffix Arrays

Find all occurrences of `issi` in `mississippi`.

`s = mississippi`

T lcp with p = 4

B lcp with p = 4

11:	i	**1**
8:	ippi	**1**
5:	issippi	**4**
2:	ississippi	**0**
1:	mississippi	**0**
10:	pi	**1**
9:	ppi	**0**
7:	sippi	**2**
4:	sissippi	**1**
6:	ssippi	**3**
3:	ssissippi	**-**

Suffix Arrays

- We will see more about suffix arrays when we get to genome assembly in a few weeks as they are the basis for the Burroughs-Wheeler Transform (or BWT)

Exact String Matching

- So far we have only looked at searching for sequence that are exactly the same, one is an exact sub-sequence of another.
- What happens when there are small differences? These cannot be handled in the methods in this section.
- Next time we will talk about the **sequence alignment** problem, i.e. *inexact* string matching.