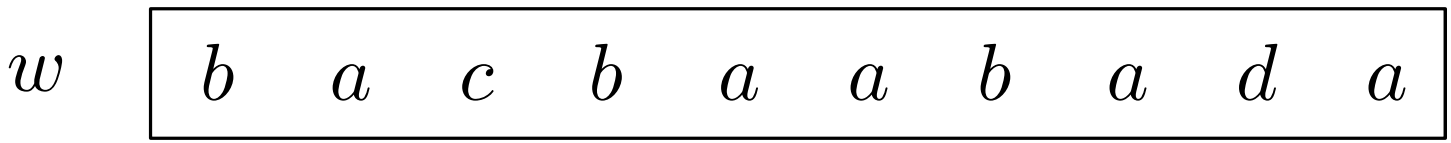
Efficiently Testing Simon's Congruence

Florin Manea

Göttingen University

One World Combinatorics on Words Seminar
June 2021

Subsequences



Subsequences

ababa

d

bbb

ba

bc

w | b | a | c | b | a | a | b | a | d | a |

baab

bb

a

c

aaaaa

b

Subsequences

ababa

d

bbb

abc

ba

bc

w | b | a | c | b | a | a | b | a | d | a |

baab

bb

a

c

aaaaa

b

Subsequences

ababa

d

bbb

ba

~~abc~~

bc

w | b | a | c | b | a | a | b | a | d | a |

baab

a

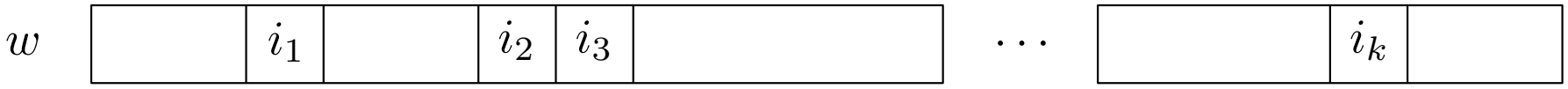
bb

c

aaaaa

b

Subsequences



Subsequence

We call w' a subsequence of length k of a word w, where $|w| = n$, if there exist positions $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, such that $w' = w[i_1]w[i_2] \cdots w[i_k]$.

Set of Subsequences of length k

Let $Subseq_k(i, w)$ denote the set of subsequences of length k of $w[i : n]$. Accordingly, the set of subsequences of length k of the entire word w will be denoted by $Subseq_k(1, w)$.

Example: $Subseq_2(1, abaca) = \{aa, ab, ac, ba, bc, ca\}$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.

Example: $w = abacab, w' = baacabba$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.

Example: $w = abacab, w' = baacabba$

$$Subseq_2(1, w) = \{aa, ab, ac, ba, bb, bc, ca, cb\}$$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.

Example: $w = abacab, w' = baacabba$

$$Subseq_2(1, w) = \{aa, ab, ac, ba, bb, bc, ca, cb\}$$
$$Subseq_2(1, w') = \{aa, ab, ac, ba, bb, bc, ca, cb\}$$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.

Example: $w = abacab, w' = baacabba$

$$Subseq_2(1, w) = \{aa, ab, ac, ba, bb, bc, ca, cb\}$$
$$Subseq_2(1, w') = \{aa, ab, ac, ba, bb, bc, ca, cb\}$$
$$Subseq_2(1, w) = Subseq_2(1, w') \Rightarrow w \sim_2 w'$$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.

Example: $w = abacab, w' = baacabba$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.

Example: $w = abacab, w' = baacabba$

$$bbb \notin Subseq_3(1, w), bbb \in Subseq_3(1, w')$$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.

Example: $w = abacab, w' = baacabba$

$$bbb \notin Subseq_3(1, w), bbb \in Subseq_3(1, w')$$
$$Subseq_3(1, w) \neq Subseq_3(1, w') \Rightarrow w \not\sim_3 w'$$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.
(ii) A word u of length k distinguishes w and w' w.r.t. $\sim_k$ if u occurs in exactly one of the sets $Subseq_k(1, w)$ and $Subseq_k(1, w')$.

Example: $w = abacab, w' = baacabba$

$$bbb \notin Subseq_3(1, w), bbb \in Subseq_3(1, w')$$
$$Subseq_3(1, w) \neq Subseq_3(1, w') \Rightarrow w \not\sim_3 w'$$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.
(ii) A word u of length k distinguishes w and w' w.r.t. $\sim_k$ if u occurs in exactly one of the sets $Subseq_k(1, w)$ and $Subseq_k(1, w')$.
(iii) Let $i, j \in w$. We define $i \sim_k j$ (w.r.t. w) if $w[i : n] \sim_k w[j : n]$, and we say that the positions i and j are k-equivalent.

Example: $w = abacab, w' = baacabba$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.
(ii) A word u of length k distinguishes w and w' w.r.t. $\sim_k$ if u occurs in exactly one of the sets $Subseq_k(1, w)$ and $Subseq_k(1, w')$.
(iii) Let $i, j \in w$. We define $i \sim_k j$ (w.r.t. w) if $w[i : n] \sim_k w[j : n]$, and we say that the positions i and j are k-equivalent.

Example: $w = abacab, w' = baacabba$

$$w[2 : 6] = bacab, w[3 : 6] = acab$$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.
(ii) A word u of length k distinguishes w and w' w.r.t. $\sim_k$ if u occurs in exactly one of the sets $Subseq_k(1, w)$ and $Subseq_k(1, w')$.
(iii) Let $i, j \in w$. We define $i \sim_k j$ (w.r.t. w) if $w[i : n] \sim_k w[j : n]$, and we say that the positions i and j are k-equivalent.

Example: $w = abacab, w' = baacabba$

$$w[2 : 6] = bacab, w[3 : 6] = acab$$
$$w[2 : 6] \sim_1 w[3 : 6]$$

Simon's Congruence

Simon's Congruence

(i) Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ if $Subseq_k(1, w) = Subseq_k(1, w')$.
(ii) A word u of length k distinguishes w and w' w.r.t. $\sim_k$ if u occurs in exactly one of the sets $Subseq_k(1, w)$ and $Subseq_k(1, w')$.
(iii) Let $i, j \in w$. We define $i \sim_k j$ (w.r.t. w) if $w[i : n] \sim_k w[j : n]$, and we say that the positions i and j are k-equivalent.

Example: $w = abacab, w' = baacabba$

$$w[2 : 6] = bacab, w[3 : 6] = acab$$
$$w[2 : 6] \sim_1 w[3 : 6]$$
$$w[2 : 6] \nsim_2 w[3 : 6]$$

Problem Definition

SimK

Given two words s and t over an alphabet Σ, with $|s| = n$ and $|t| = n'$, with $n \geq n'$, and a natural number k, decide whether $s \sim_k t$.

MaxSimK

Given two words s and t over an alphabet Σ, with $|s| = n$ and $|t| = n'$, with $n \geq n'$, find the maximum k for which $s \sim_k t$.

Basic Assumption

Σ integer alphabet, i.e., $\Sigma \subseteq \{1, \ldots, n\}$.

History

- Line of research originating in the PhD thesis of Imre Simon from 1972
- Long history of algorithm designs and improvements for associated problems. State of the art:
  – SimK optimal linear time [DLT 2020], prompted by linear time solution for constant alphabets, via shortlex form [Kufleitner, Fleischer, MFCS 2018]
  – MaxSimK $O(n \log n)$ time
  – Simon claimed a linear time solution for MaxSimK in 2003, but never published it
  – optimal linear-time algorithm for the MaxSimK problem using a new data structure called Simon-Tree [STACS 2021]

SimK

Definition

The *shortlex normal form* of a word $w \in \Sigma^*$ w.r.t. $\sim_k$, where Σ is an ordered alphabet, is the shortest word u with $u \sim_k w$ which is also lexicographically smallest (w.r.t. the given order on Σ) amongst all words $v \sim_k w$ with $|v| = |u|$.

Idea

Checking whether $w' \sim_k w''$ is equivalent to checking whether the shortlex normal forms w.r.t. $\sim_k$ of w' and w'' are equal.

SimK

[Kufleitner and Fleischer, MFCS 2018]

To compute the shortlex normal form of a word $w \in \Sigma^n$ w.r.t. $\sim_k$, we define for each position of w define the x- and y-coordinates:

- The x-coordinate of i, denoted x_i, is the length of the shortest sequence of indices $1 \leq i_1 < i_2 < \ldots < i_t = i$ such that i_1 is the position where the letter $w[i_1]$ occurs in w for the first time and, for $1 < j \leq t$, i_j is the first position where $w[i_j]$ occurs in $w[i_{j-1} + 1..i]$. Obviously, if a occurs for the first time on position i in w, then $x_i = 1$.

Property

If $w[\ell] = w[i] = \mathsf{a}$ for some $i > \ell$ such that $w[j] \neq \mathsf{a}$ for all $\ell + 1 \leq j \leq i - 1$, then $x_i = \min\{x_\ell, x_{\ell+1}, \ldots, x_{i-1}\} + 1$.

SimK

[Kufleitner and Fleischer, MFCS 2018]

To compute the shortlex normal form of a word $w \in \Sigma^n$ w.r.t. $\sim_k$, we define for each position of w define the x- and y-coordinates:

- The y-coordinate of a position i, denoted y_i, is defined symmetrically: y_i is the length of the shortest sequence of indices $n \geq i_1 > i_2 > \ldots > i_t = i$ such that i_1 is the position where the letter $w[i_1]$ occurs last time in w and, for $1 < j \leq t$, i_j is the last position where $w[i_j]$ occurs in $w[i..i_{j-1} - 1]$.

Property

If $w[\ell] = w[i] = \mathsf{a}$ for some $i < \ell$ such that $w[j] \neq \mathsf{a}$ for all $\ell - 1 \geq j \geq i + 1$, then $y_i = \min\{y_{i+1}, \ldots, y_{\ell-1}, y_\ell\} + 1$.

[Kufleitner and Fleischer, MFCS 2018]

Computing the coordinates is done in two phases:

- The x-coordinates are computed and stored (in an array $x : x_1, \ldots, x_n$) from left to right, and the y-coordinates are stored in an array $y : y_1, \ldots, y_n$ and computed from right to left (while dynamically deleting a position whenever the sum of its coordinates is greater then $k + 1$).
- To compute the shortlex normal form, go left to right in the word: if letters $\mathtt{b} > \mathtt{a}$ occur consecutively, they are interchanged whenever they have the same x- and y-coordinates and the sum of these coordinates is $k + 1$ (until this situation does not occur anymore).

[DLT 2020]

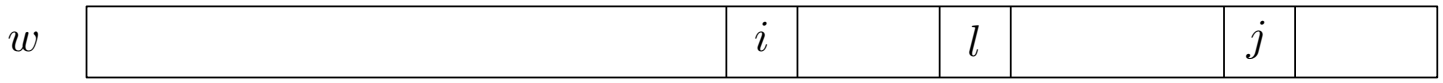
This can be done in linear time for integer alphabets (based on interval union-find data structures + radix-sort).

SimK

Theorem

- *Given a word w over an integer alphabet Σ, with $|w| = n$, and a number $k \leq n$, we can compute the shortlex normal form of w w.r.t. $\sim_k$ in time $O(n)$.*
- *Given two words w', w'' over an integer alphabet Σ, with $|w'| \leq |w''| = n$, and a number $k \leq n$, we can test if $w' \sim_k w''$ in time $O(n)$.*

MaxSimK: Simon-tree – Equivalence Classes



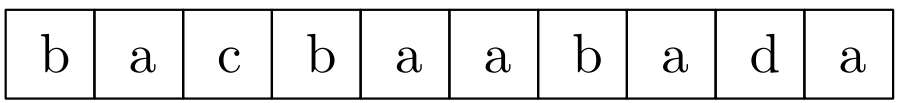
$$Subseq_k(i, w) \supset Subseq_k(l, w) \supset Subseq_k(j, w)$$

- Splitting a word suffixwise into blocks of equivalence classes w.r.t. $\sim_k$
- If $i \sim_k j$, then
  $Subseq_k(i, w) = Subseq_k(l, w) = Subseq_k(j, w)$
  and we say that i, l, and j are in the same *k-block*
- $\sim_{k+1}$ is a refinement of $\sim_k$
- Index i is a $(k+1)$-*splitting position* if $i \sim_k i+1$ but not $i \sim_{k+1} i+1$

Equivalence Classes

Use these properties to build a block structure for a word

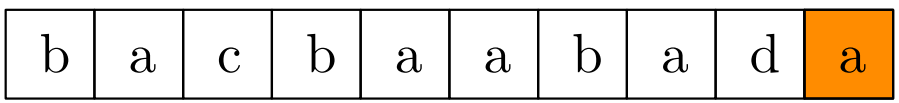
1. $i \sim_1 j$ iff $\text{alph}(w[i:n]) = \text{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions



Equivalence Classes

Use these properties to build a block structure for a word

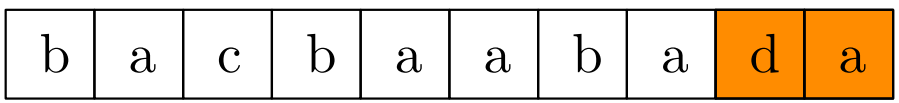
1. $i \sim_1 j$ iff $\text{alph}(w[i:n]) = \text{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions



Equivalence Classes

Use these properties to build a block structure for a word

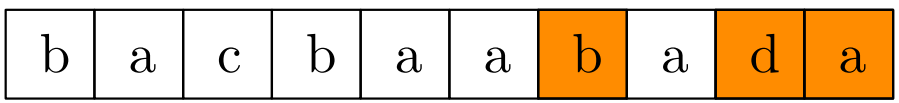
1. $i \sim_1 j$ iff $\mathrm{alph}(w[i:n]) = \mathrm{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions



Equivalence Classes

Use these properties to build a block structure for a word

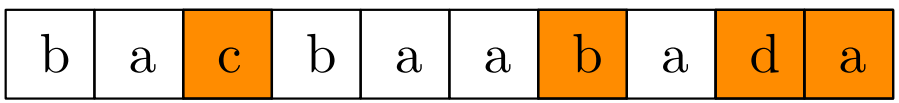
1. $i \sim_1 j$ iff $\text{alph}(w[i:n]) = \text{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions



Equivalence Classes

Use these properties to build a block structure for a word

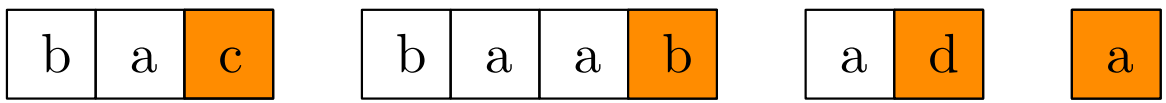
1. $i \sim_1 j$ iff $\text{alph}(w[i : n]) = \text{alph}(w[j : n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions



Equivalence Classes

Use these properties to build a block structure for a word

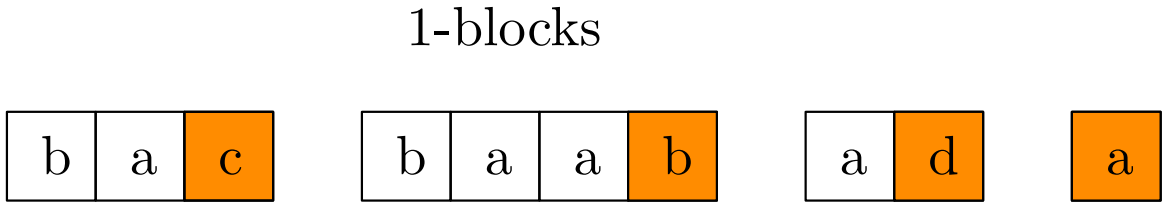
1. $i \sim_1 j$ iff $\mathrm{alph}(w[i:n]) = \mathrm{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions



Equivalence Classes

Use these properties to build a block structure for a word

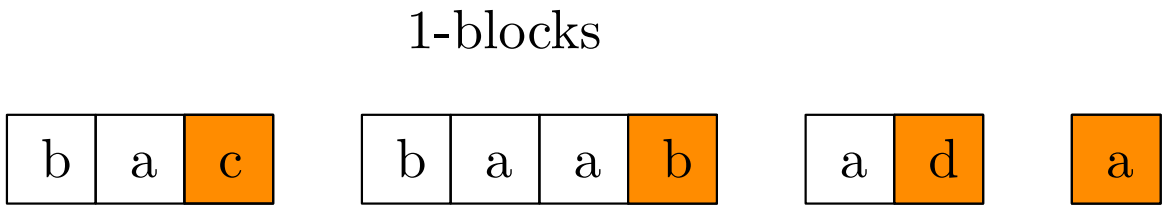
1. $i \sim_1 j$ iff $\mathrm{alph}(w[i:n]) = \mathrm{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions



Equivalence Classes

Use these properties to build a block structure for a word

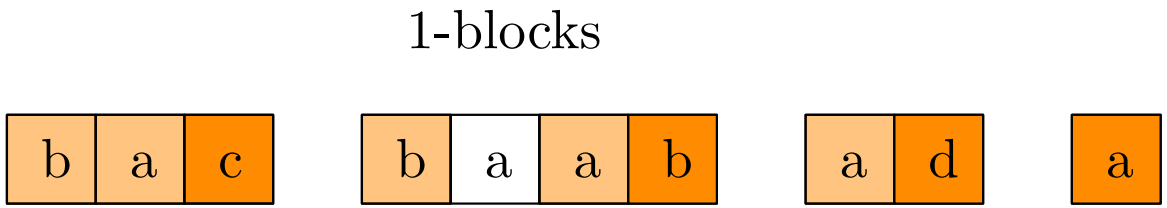
1. $i \sim_1 j$ iff $\mathrm{alph}(w[i:n]) = \mathrm{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions
2. Split a k-block into $(k+1)$-blocks by going from right to left through the block (without its last letter, which forms a singleton $(k+1)$-block) and determine $(k+1)$-splitting positions **exactly** as for 1-splitting positions (but w.r.t. the block only)



Equivalence Classes

Use these properties to build a block structure for a word

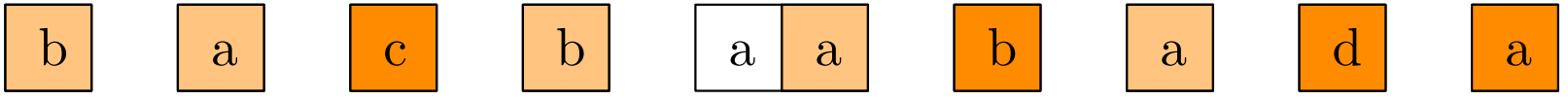
1. $i \sim_1 j$ iff $\text{alph}(w[i:n]) = \text{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions
2. Split a k-block into $(k+1)$-blocks by going from right to left through the block (without its last letter, which forms a singleton $(k+1)$-block) and determine $(k+1)$-splitting positions **exactly** as for 1-splitting positions (but w.r.t. the block only)



Equivalence Classes

Use these properties to build a block structure for a word

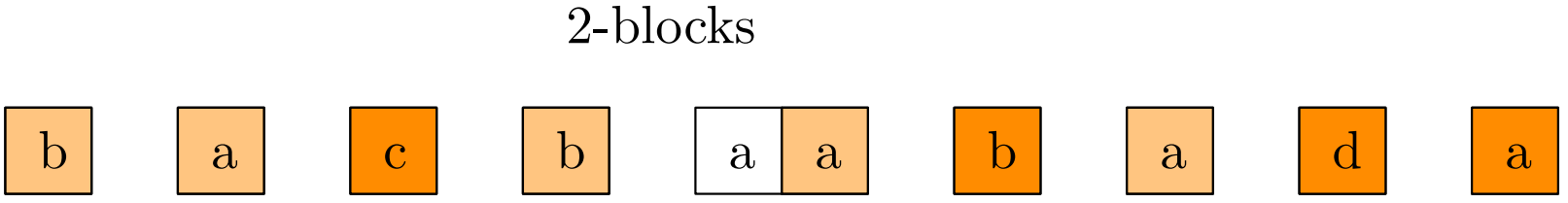
1. $i \sim_1 j$ iff $\text{alph}(w[i:n]) = \text{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions
2. Split a k-block into $(k+1)$-blocks by going from right to left through the block (without its last letter, which forms a singleton $(k+1)$-block) and determine $(k+1)$-splitting positions **exactly** as for 1-splitting positions (but w.r.t. the block only)



Equivalence Classes

Use these properties to build a block structure for a word

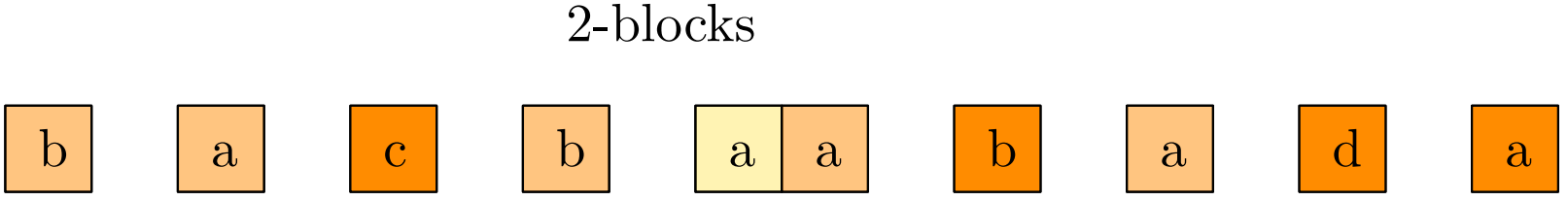
1. $i \sim_1 j$ iff $\text{alph}(w[i:n]) = \text{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions
2. Split a k-block into $(k+1)$-blocks by going from right to left through the block (without its last letter, which forms a singleton $(k+1)$-block) and determine $(k+1)$-splitting positions **exactly** as for 1-splitting positions (but w.r.t. the block only)



Equivalence Classes

Use these properties to build a block structure for a word

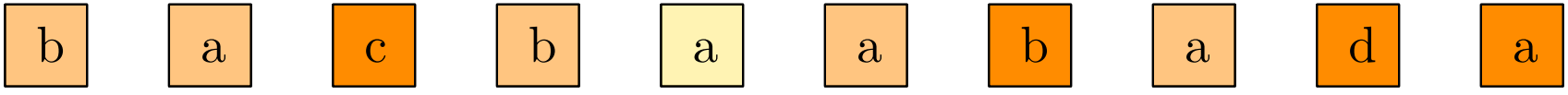
1. $i \sim_1 j$ iff $\text{alph}(w[i:n]) = \text{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions
2. Split a k-block into $(k+1)$-blocks by going from right to left through the block (without its last letter, which forms a singleton $(k+1)$-block) and determine $(k+1)$-splitting positions **exactly** as for 1-splitting positions (but w.r.t. the block only)



Equivalence Classes

Use these properties to build a block structure for a word

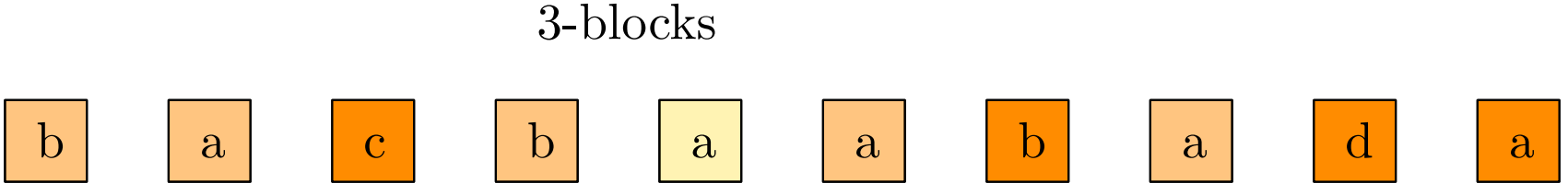
1. $i \sim_1 j$ iff $\mathrm{alph}(w[i:n]) = \mathrm{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions
2. Split a k-block into $(k+1)$-blocks by going from right to left through the block (without its last letter, which forms a singleton $(k+1)$-block) and determine $(k+1)$-splitting positions **exactly** as for 1-splitting positions (but w.r.t. the block only)



Equivalence Classes

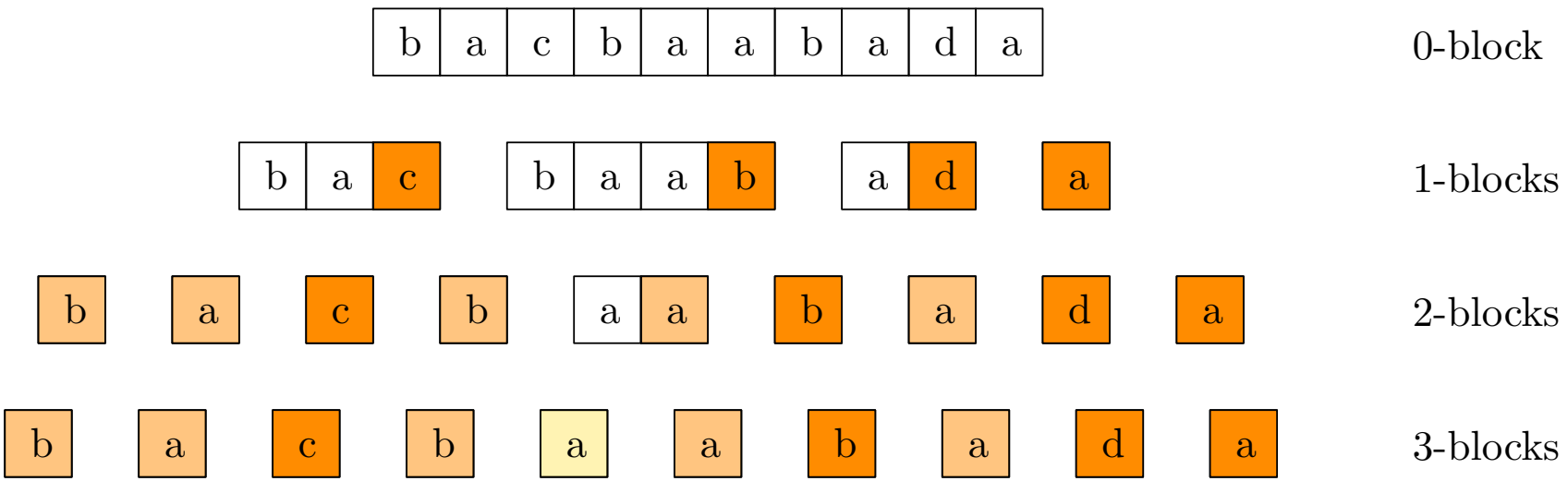
Use these properties to build a block structure for a word

1. $i \sim_1 j$ iff $\text{alph}(w[i:n]) = \text{alph}(w[j:n])$ for any $i, j \in w$
   $\rightarrow$ We can go from right to left through the word and determine 1-splitting positions
2. Split a k-block into $(k+1)$-blocks by going from right to left through the block (without its last letter, which forms a singleton $(k+1)$-block) and determine $(k+1)$-splitting positions **exactly** as for 1-splitting positions (but w.r.t. the block only)



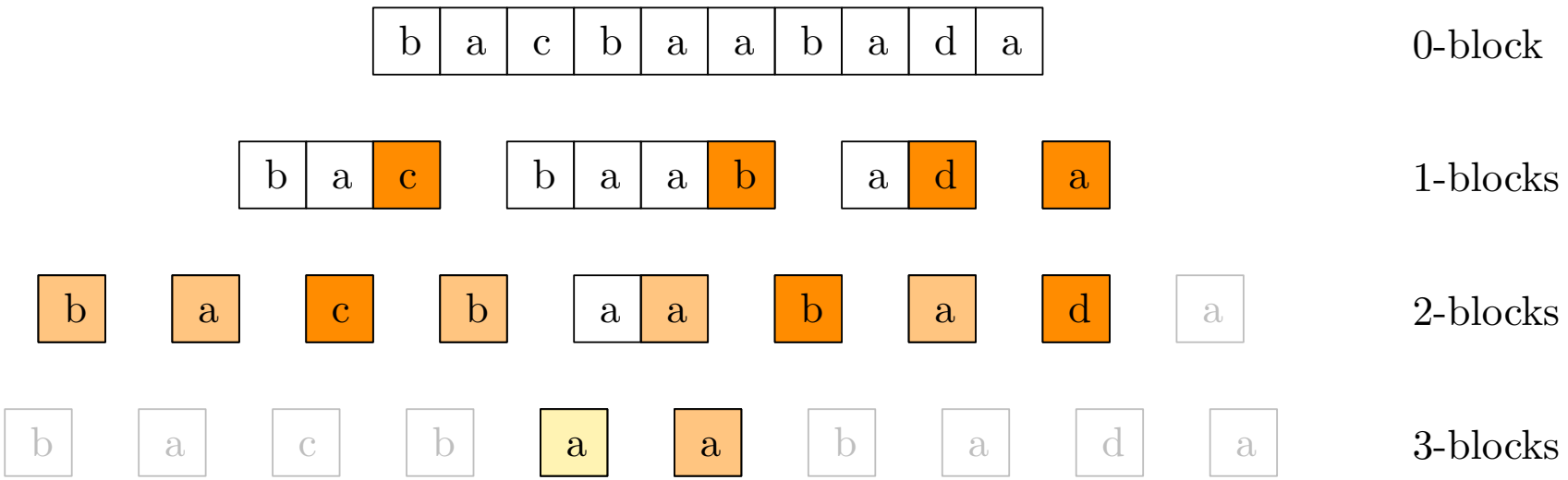
Simon-tree Definition

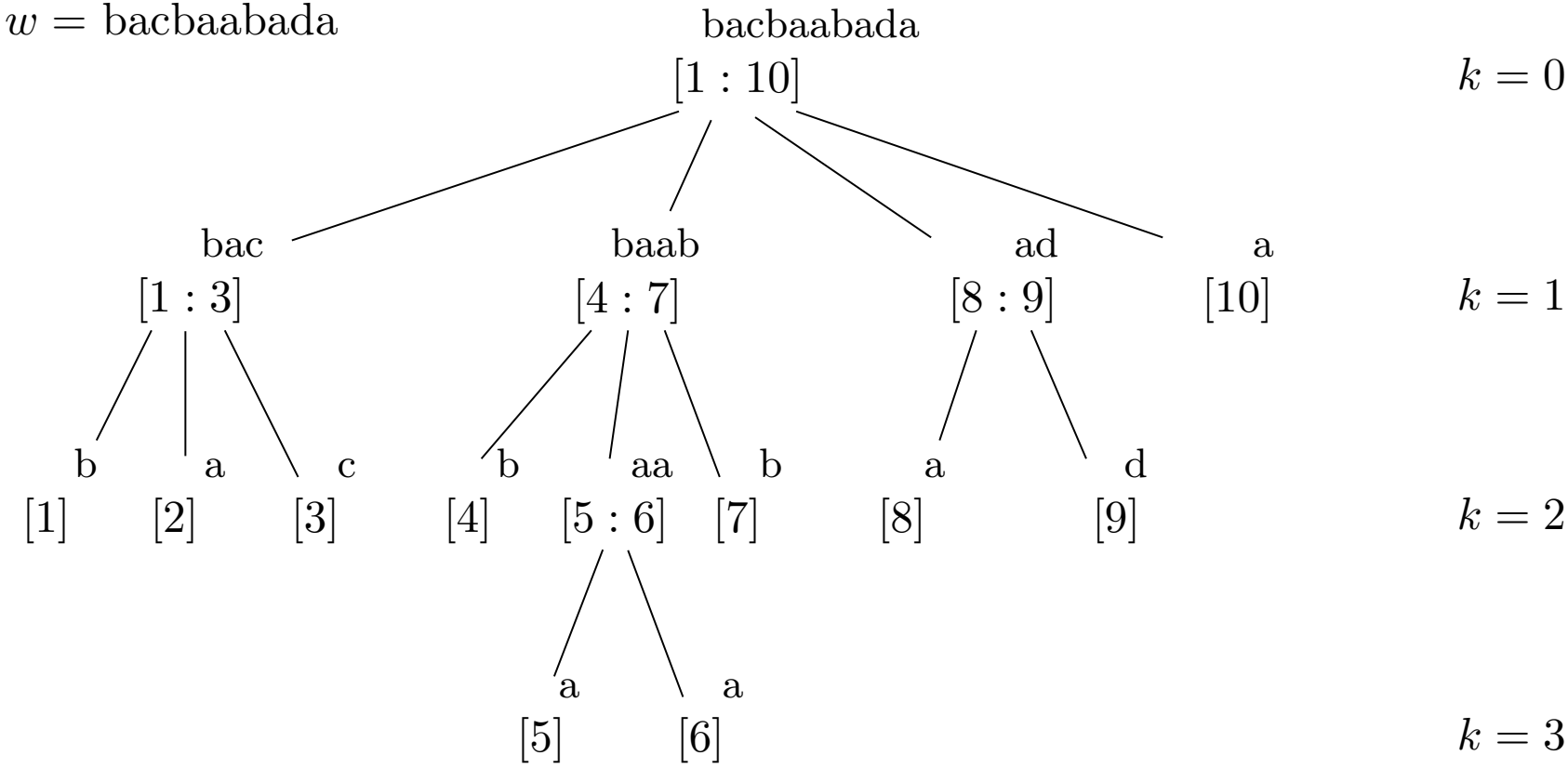
- New data structure: Simon-tree
- Represents presented block structure
- Efficiently partition positions of a given word
- Construction takes linear time



Simon-tree Definition

- New data structure: Simon-tree
- Represents presented block structure
- Efficiently partition positions of a given word
- Construction takes linear time



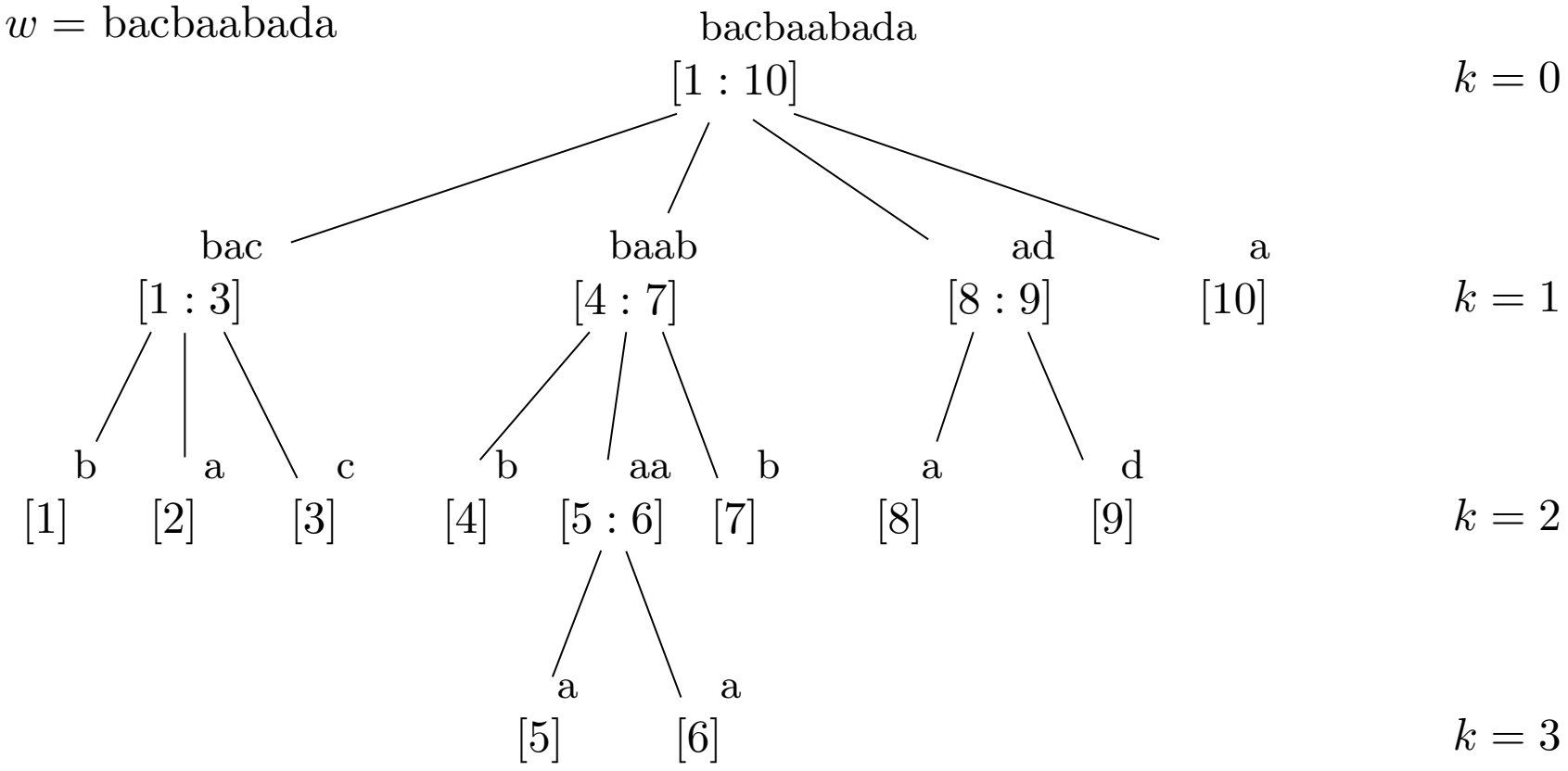


- The root corresponds to the 0-block of w
- Children correspond to the blocks obtained by refinement
- Nodes corresponding to singletons have no children

Simon-tree Construction

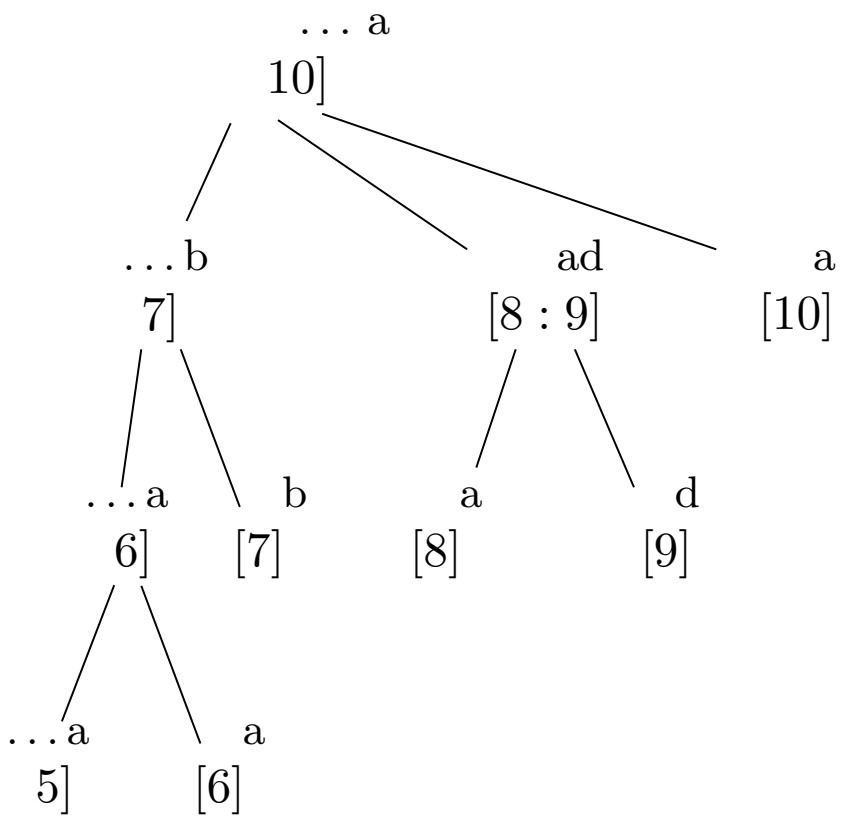
- **Algorithm:** Build the Simon-tree right to left as the word is traversed right to left. Only the leftmost branch is edited during construction.
  1. Insert the new position/letter into the tree by moving up the leftmost branch from leaf to root.
  2. Find lowest node that is not split by this position (and close all the others on the way).
  3. Insert the new position/letter as a leftmost child of this node.

Simon-tree Construction



Simon-tree Construction

$w = \text{bacbaabada}$



$k = 0$

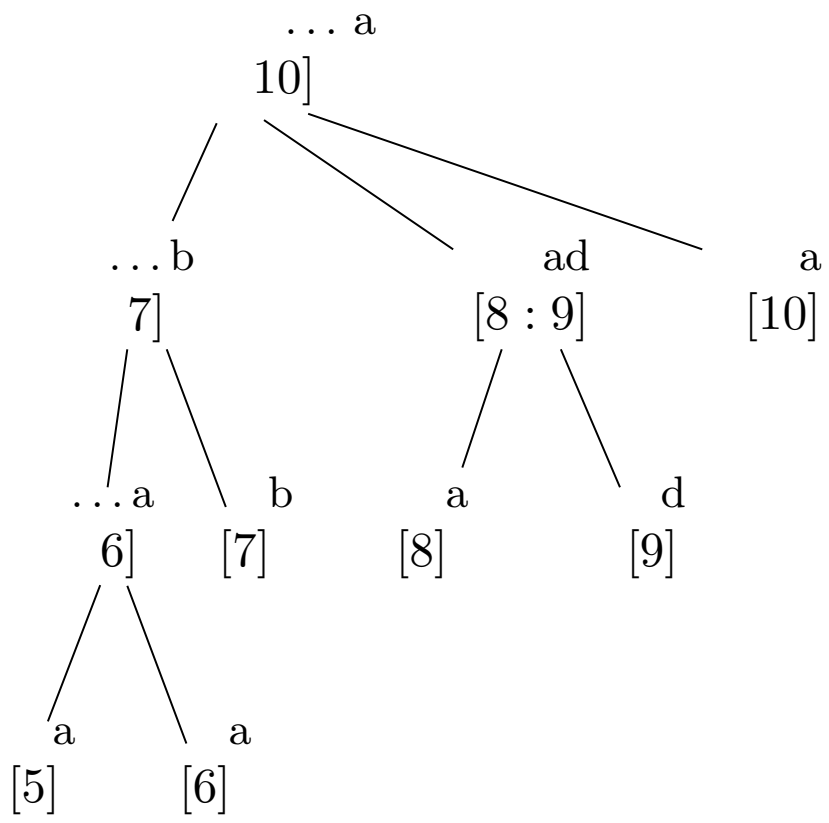
$k = 1$

$k = 2$

$k = 3$

Simon-tree Construction

$w = \text{bacbaabada}$



$k = 0$

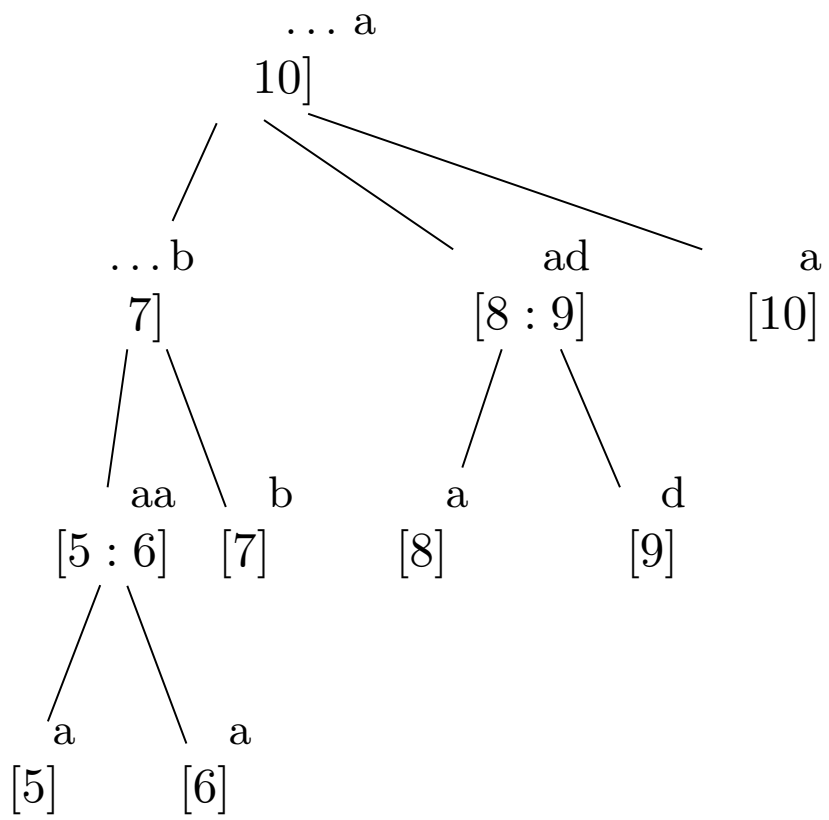
$k = 1$

$k = 2$

$k = 3$

Simon-tree Construction

$w = \text{bacbaabada}$



$k = 0$

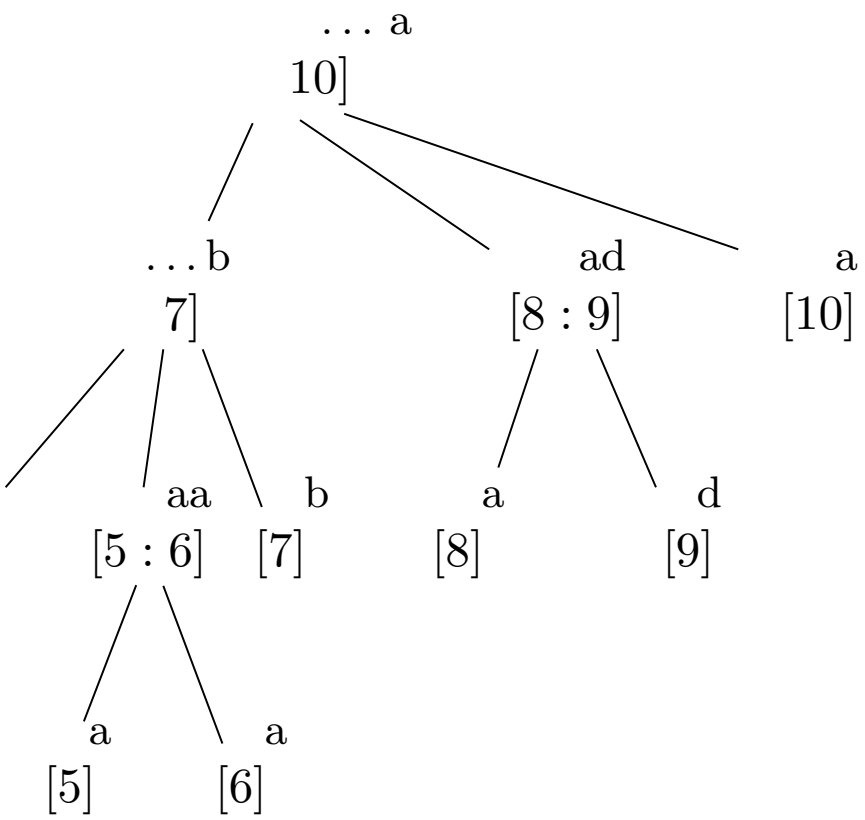
$k = 1$

$k = 2$

$k = 3$

Simon-tree Construction

$w = \text{bacbaabada}$



$k = 0$

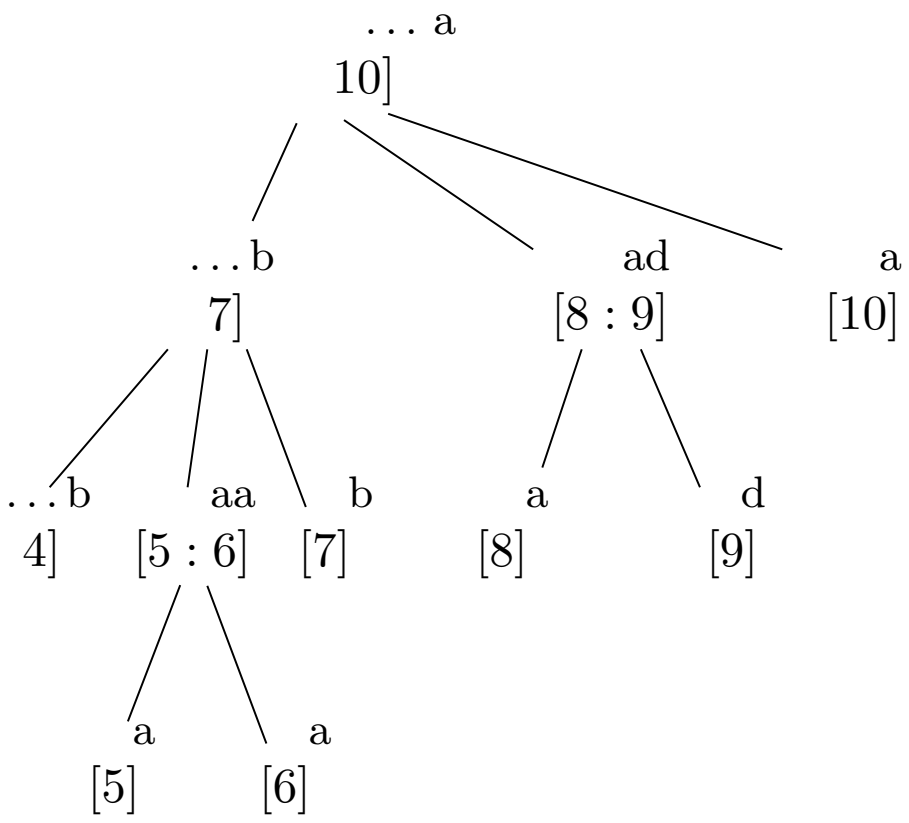
$k = 1$

$k = 2$

$k = 3$

Simon-tree Construction

$w = \text{bacbaabada}$



$k = 0$

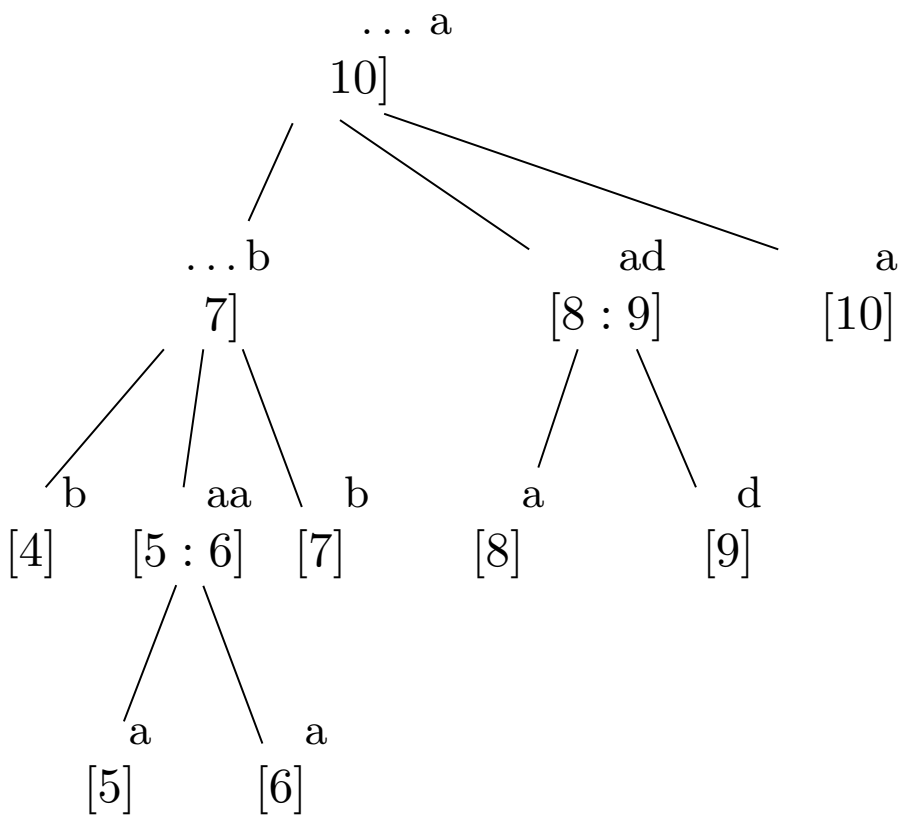
$k = 1$

$k = 2$

$k = 3$

Simon-tree Construction

$w = \text{bacbaabada}$



$k = 0$

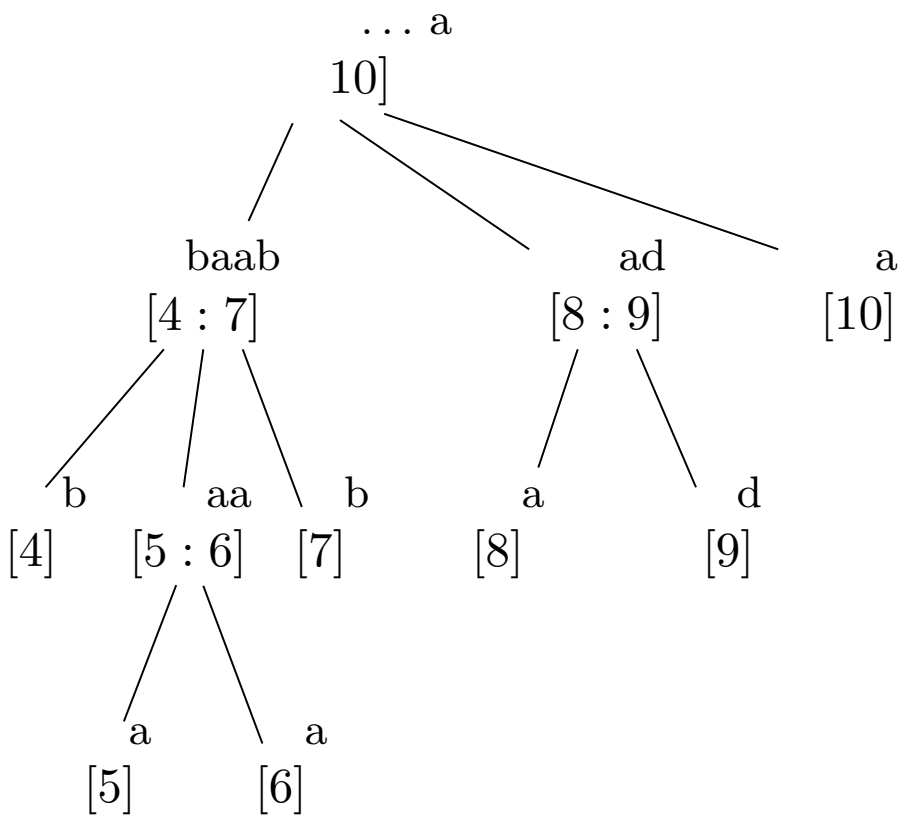
$k = 1$

$k = 2$

$k = 3$

Simon-tree Construction

$w = \text{bacbaabada}$



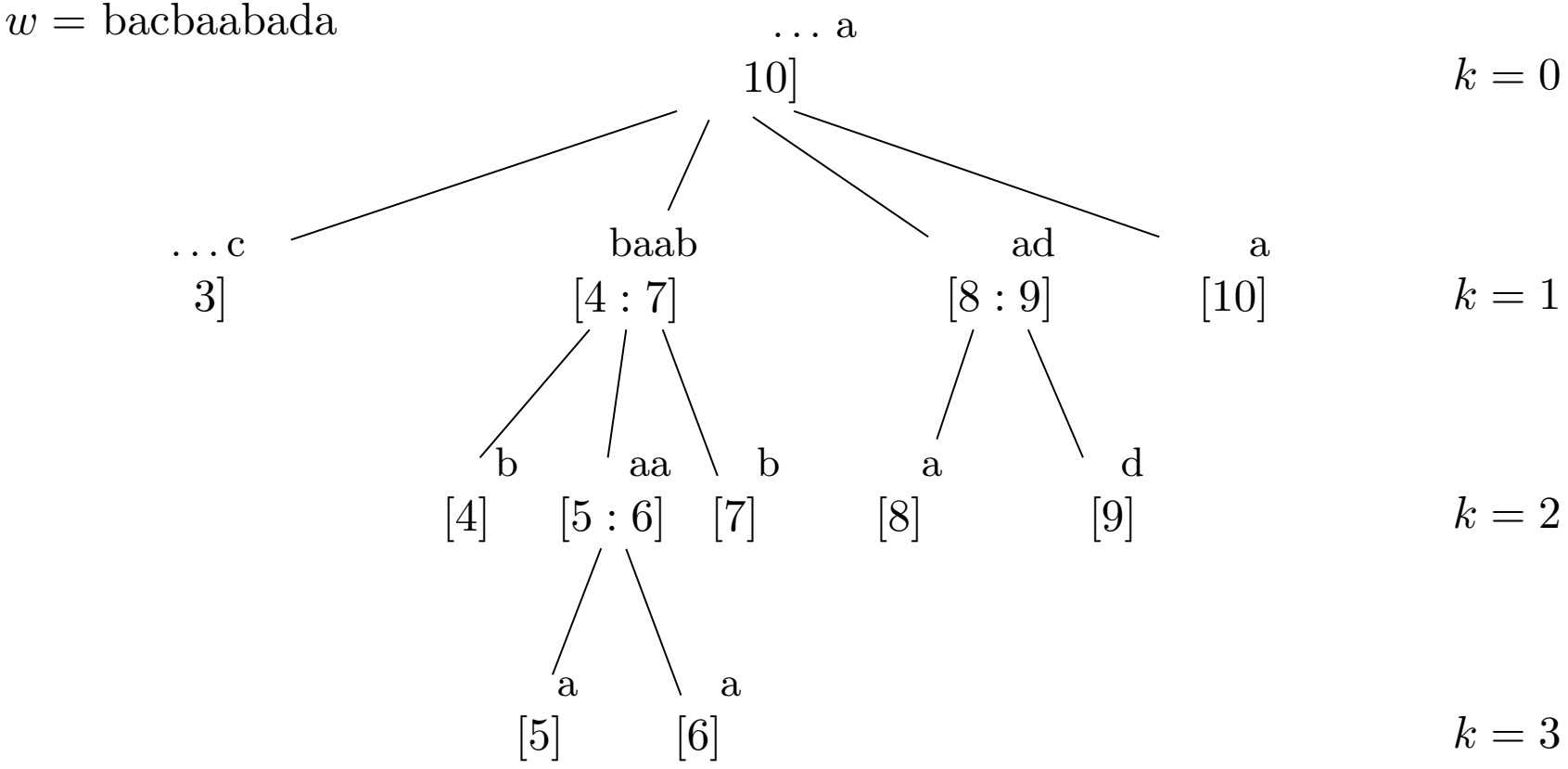
$k = 0$

$k = 1$

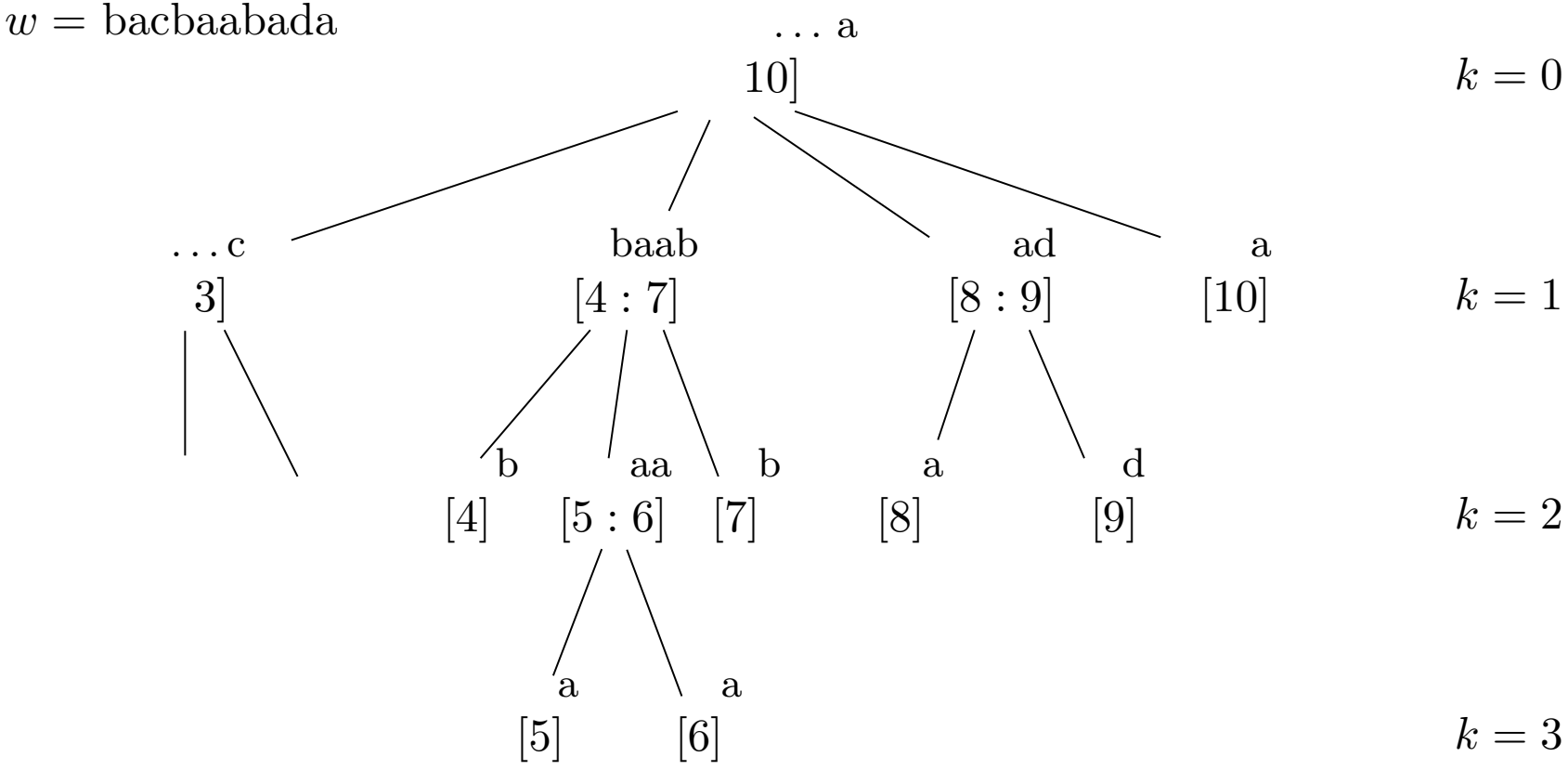
$k = 2$

$k = 3$

Simon-tree Construction

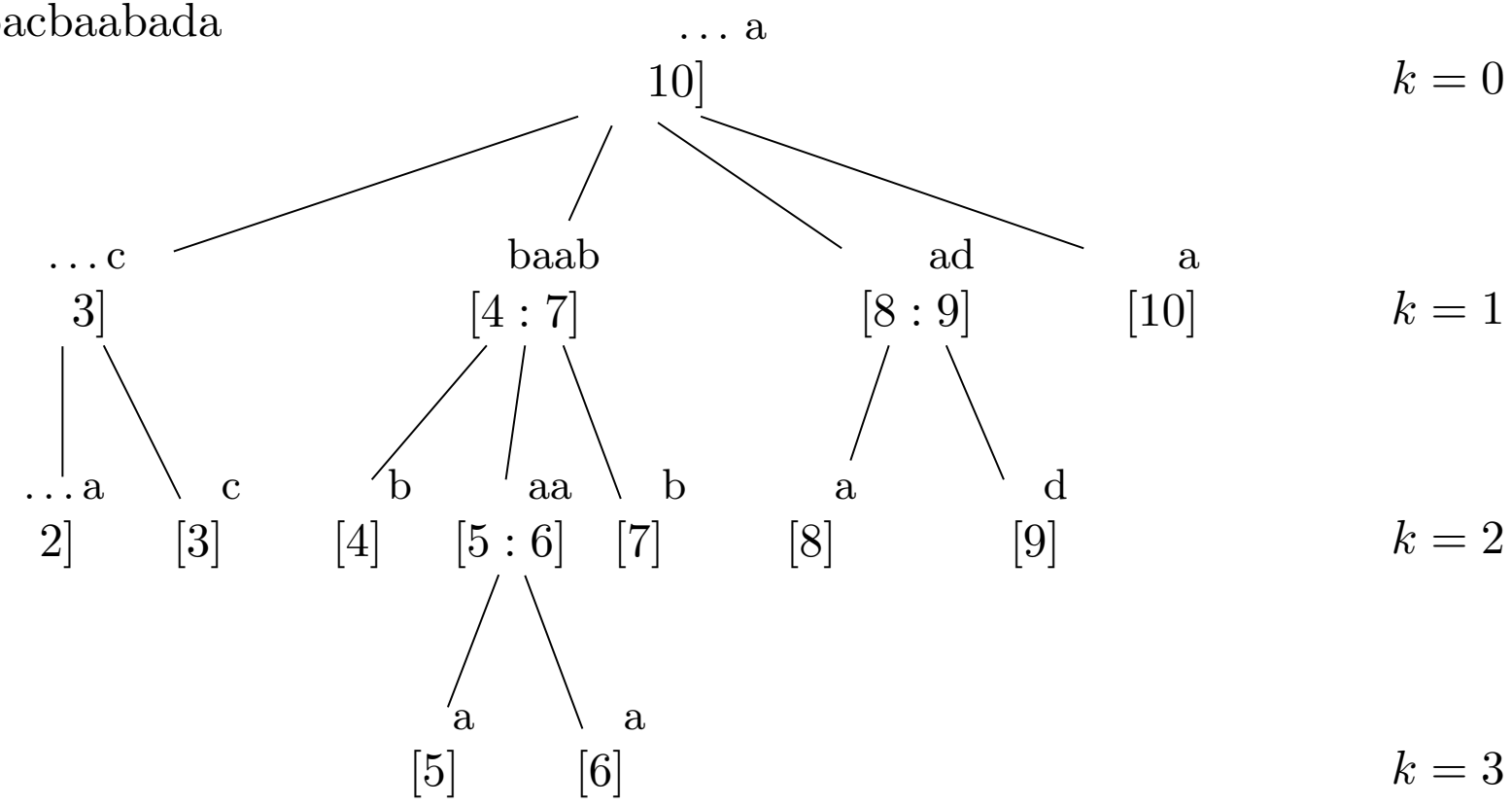


Simon-tree Construction

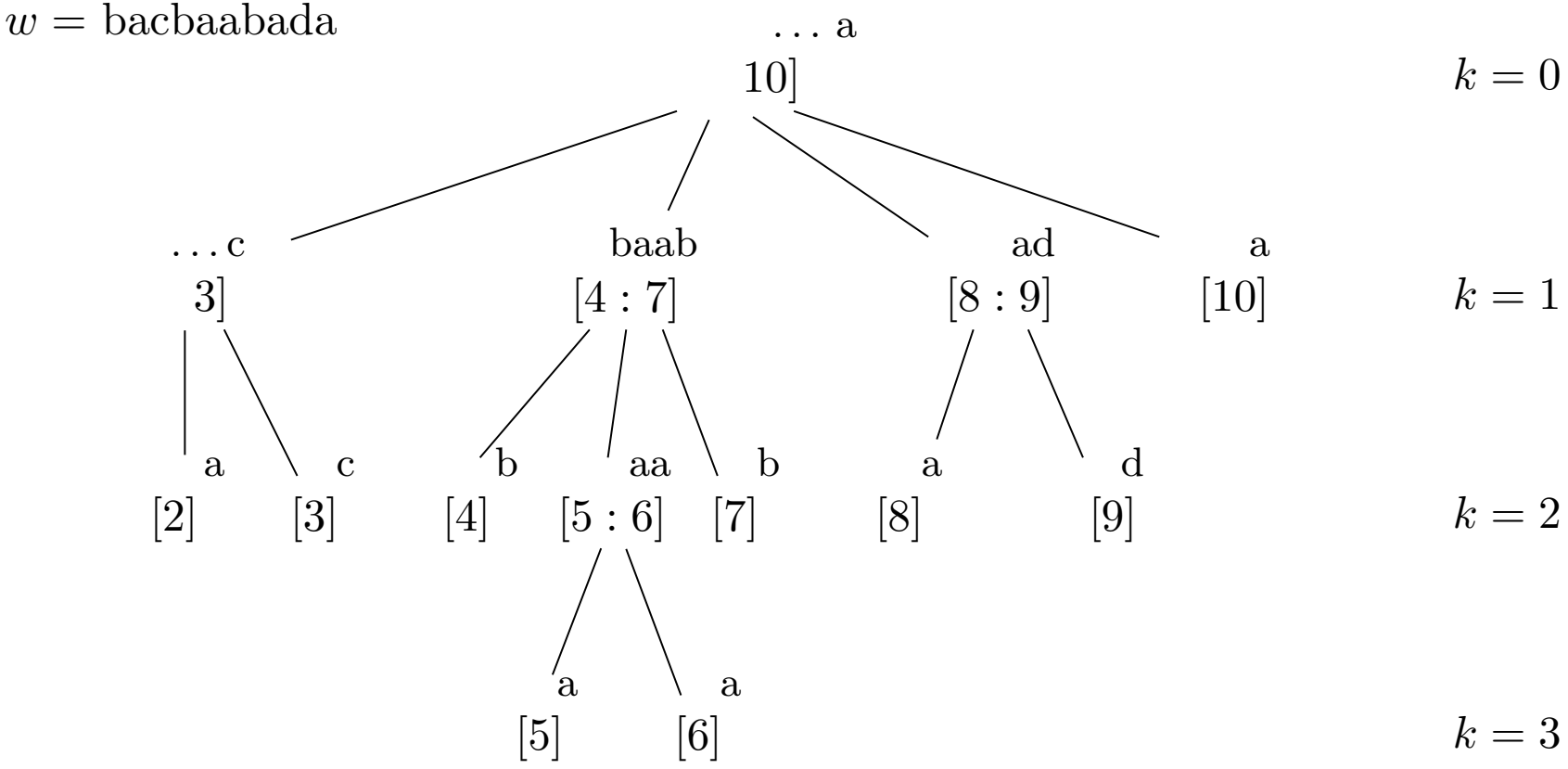


Simon-tree Construction

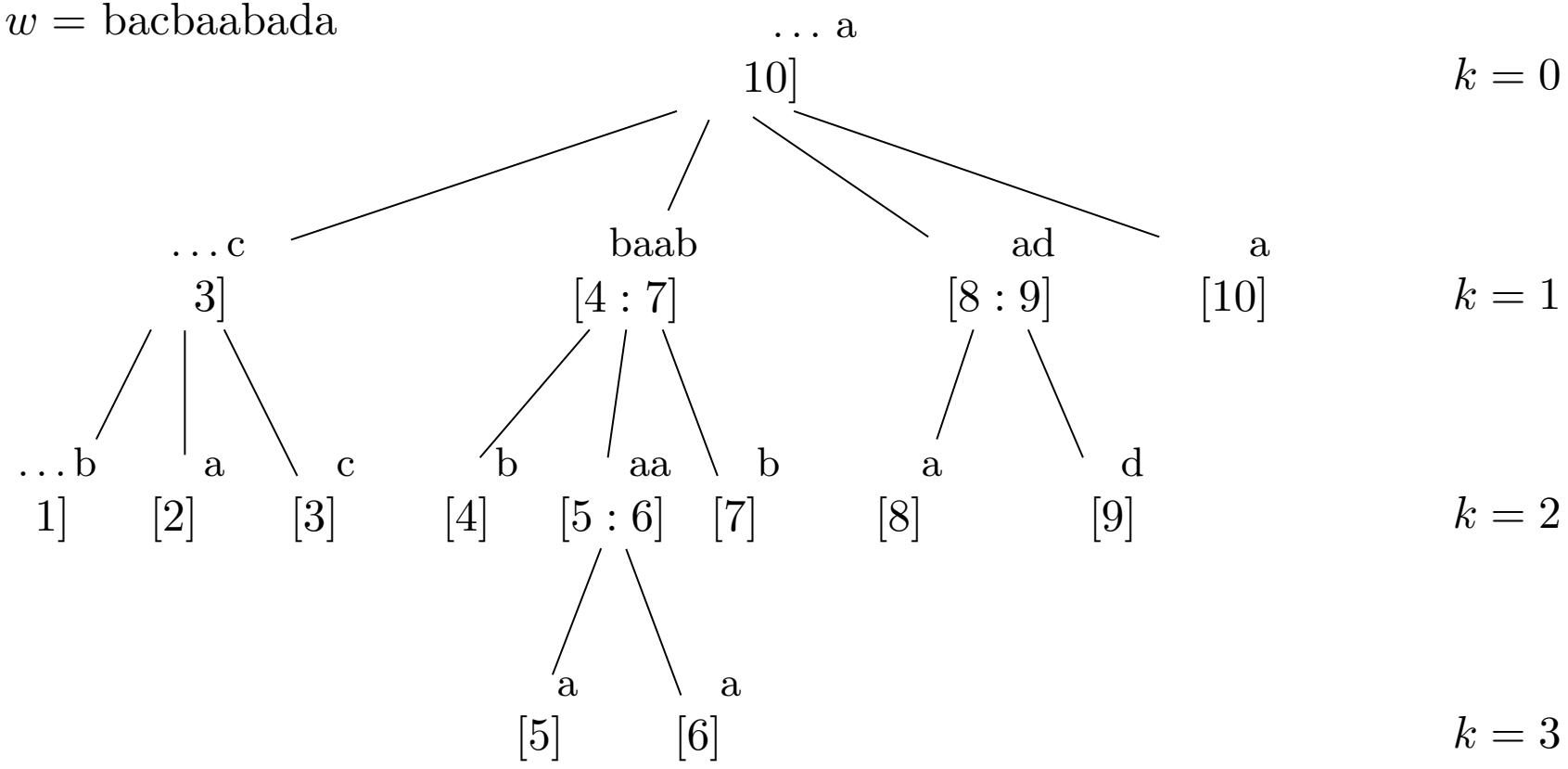
$w = \text{bacbaabada}$



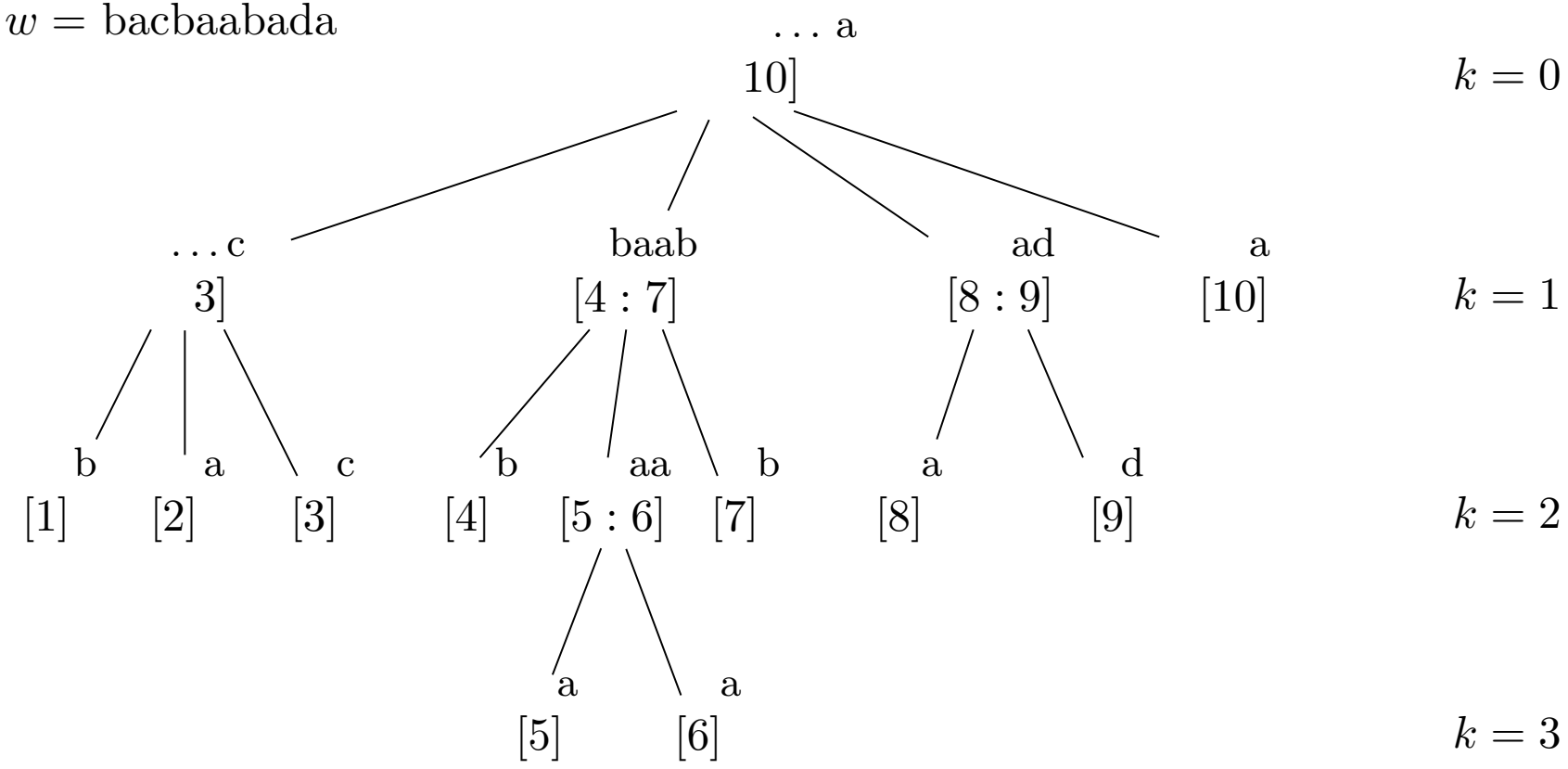
Simon-tree Construction



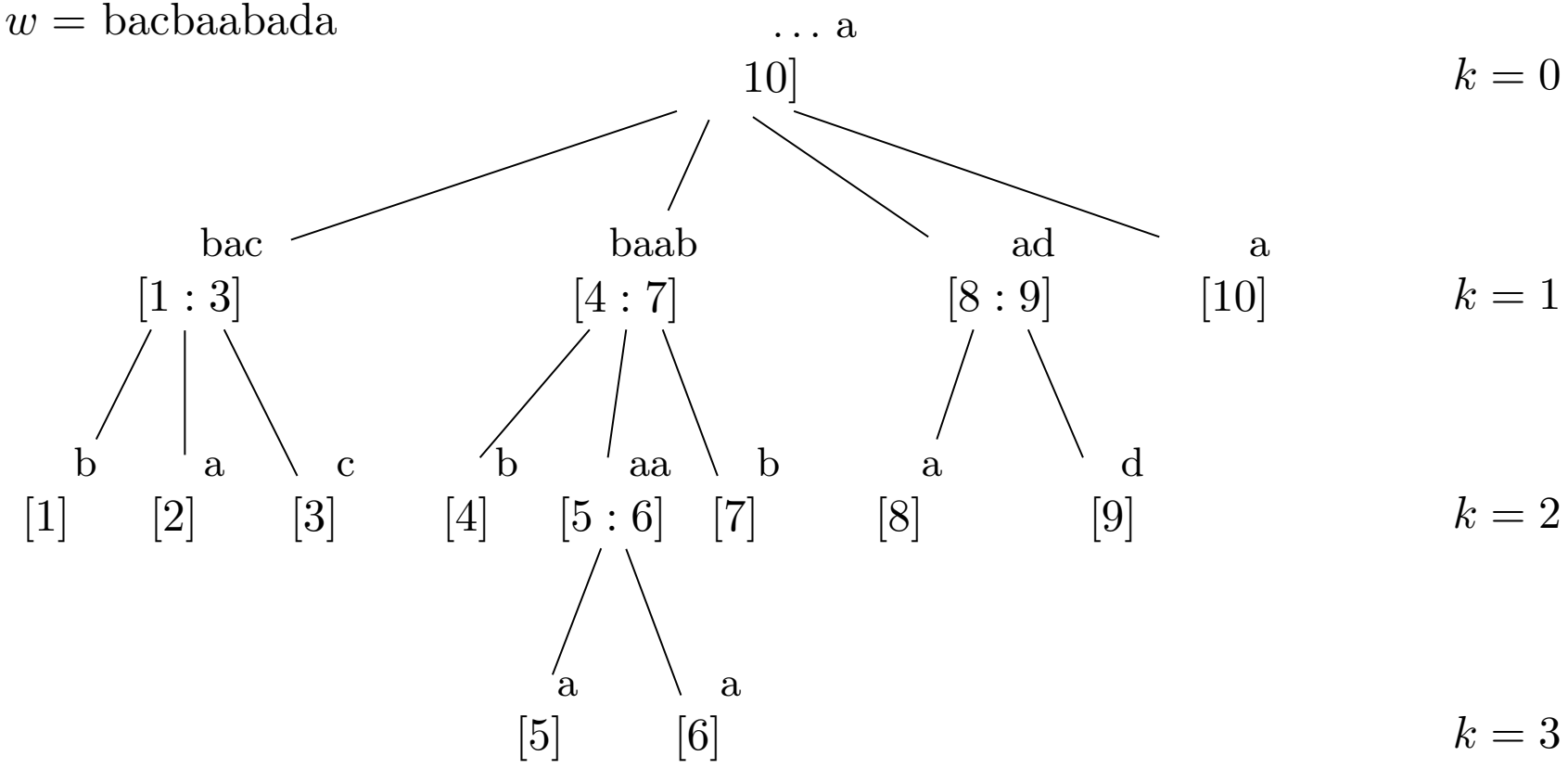
Simon-tree Construction



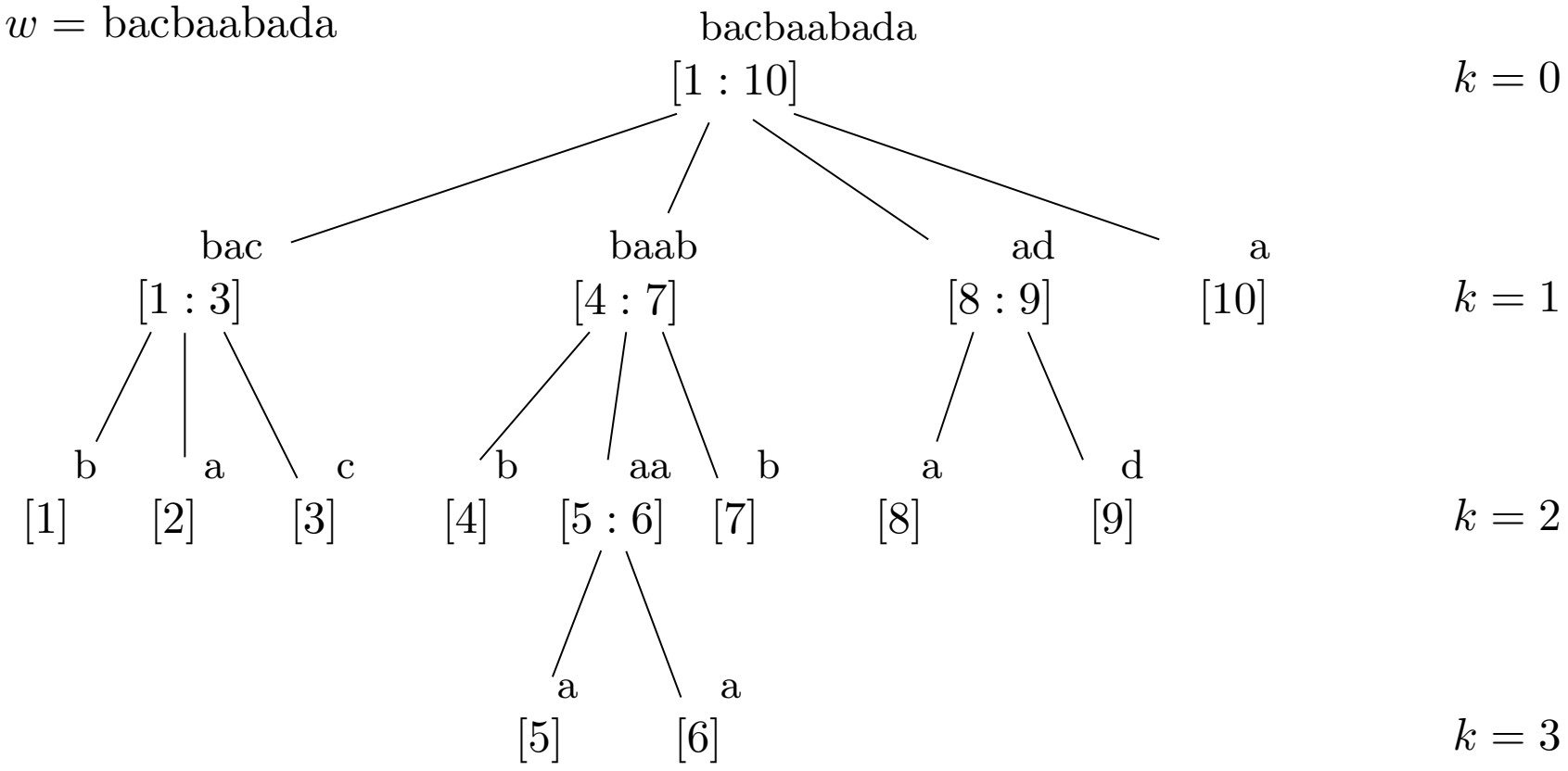
Simon-tree Construction



Simon-tree Construction



Simon-tree Construction



Short Recap

So far:
Structure for one word representing the equivalence classes w.r.t. $\sim_k$

MaxSimK

Given two words s and t over an alphabet Σ, with $|s| = n$ and $|t| = n'$, with $n \geq n'$, find the maximum k for which $s \sim_k t$.

Now:
Set two words in relation to each other by using their respective Simon-trees

Connecting Two Simon-trees

- Transform the words s and t into Simon-trees as shown
- Use the tree structure to connect equivalent nodes of the two words.

Connecting Two Simon-trees

- Transform the words s and t into Simon-trees as shown
- Use the tree structure to connect equivalent nodes of the two words.

S-Connection

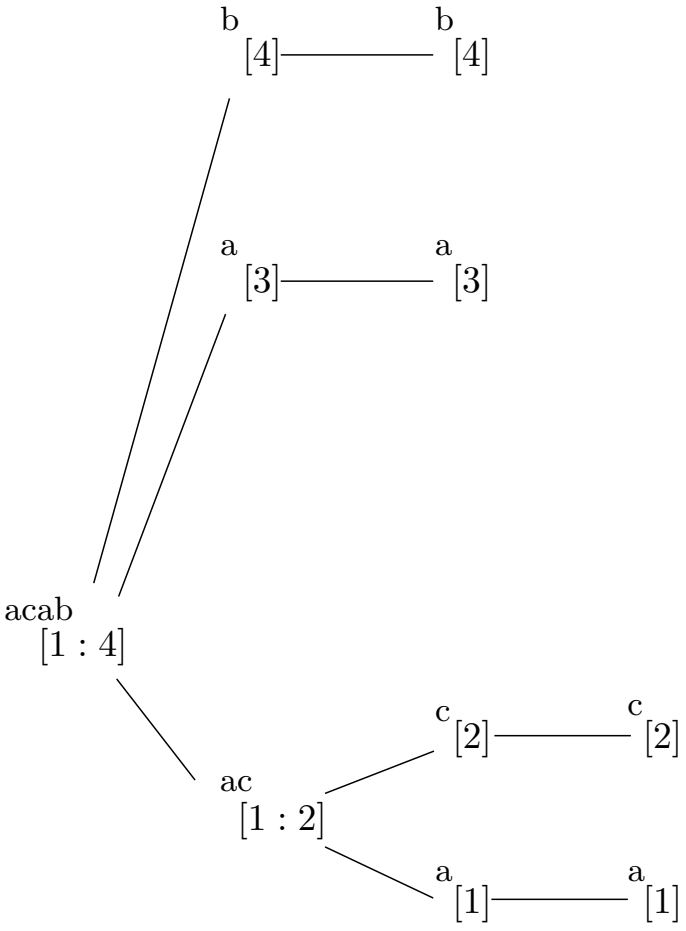
The k-node a of T_s and the k-node b of T_t are S-connected (i.e., the pair (a, b) is in the S-connection) if and only if $s[i : n] \sim_k t[j : n']$ for all positions i in block a and positions j in block b.

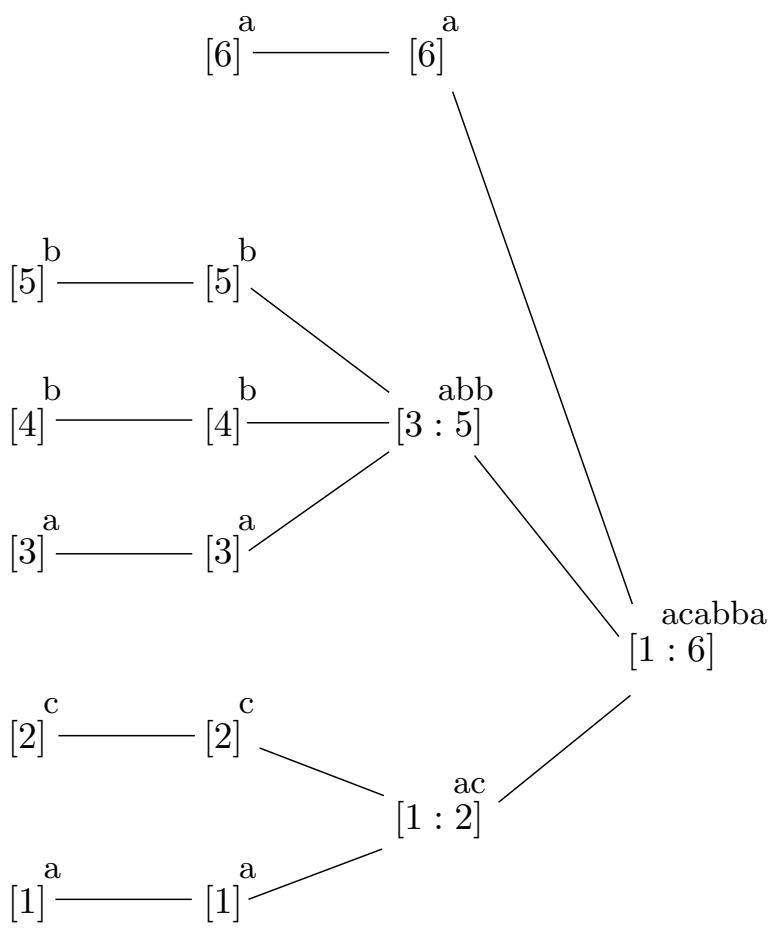
From P-Connection to S-Connection

Starting from a larger relation (P-Connection) which contains the S-Connection, and refine it

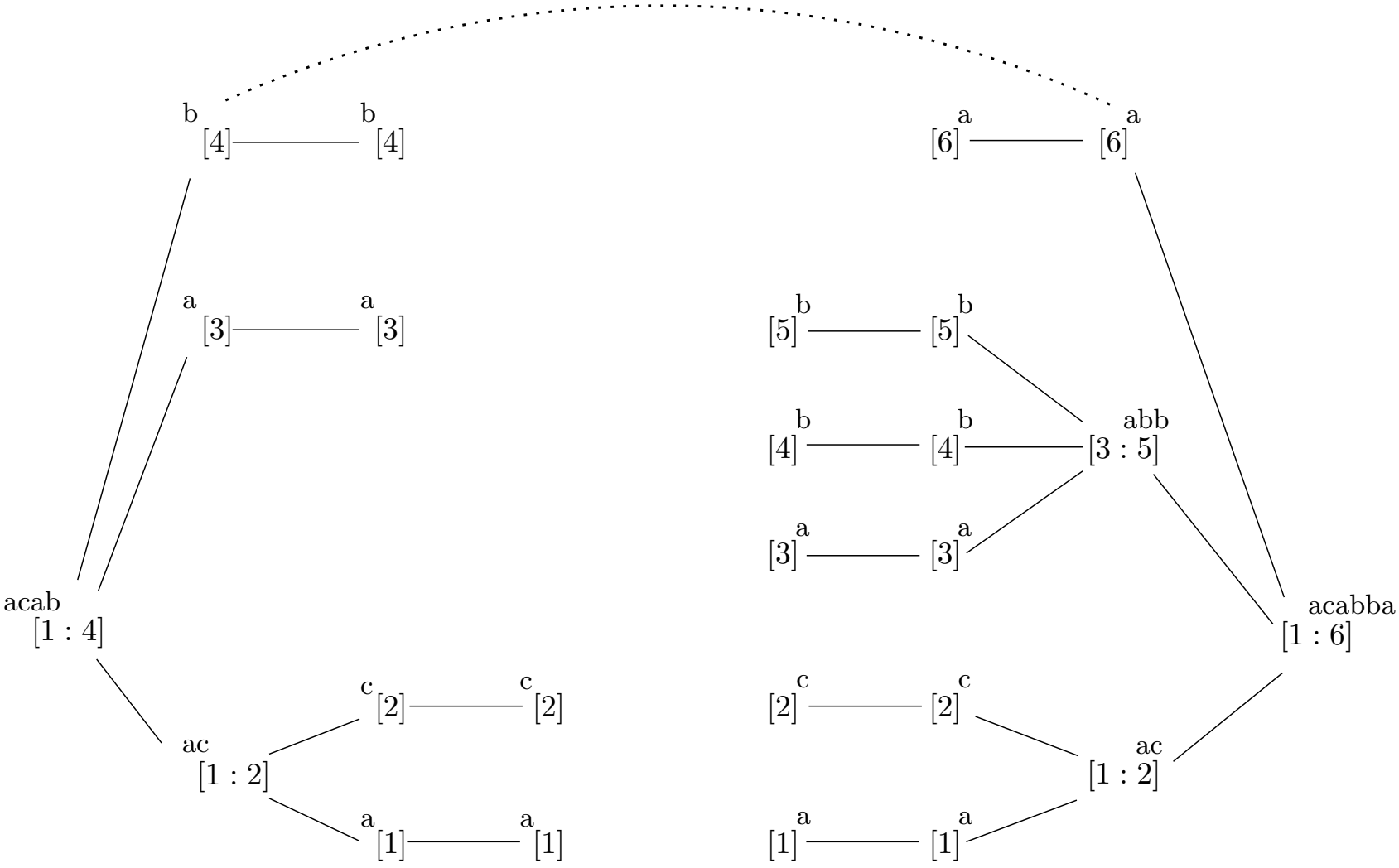
- The 0-nodes of T_s and T_t are P-connected.
- For all levels k of T_s, if the explicit or implicit k-nodes a and b (from T_s and T_t, respectively) are P-connected, then the i^{th} child of a is P-connected to the i^{th} child of b, for all i.
- No other nodes are P-connected.

From P-Connection to S-Connection

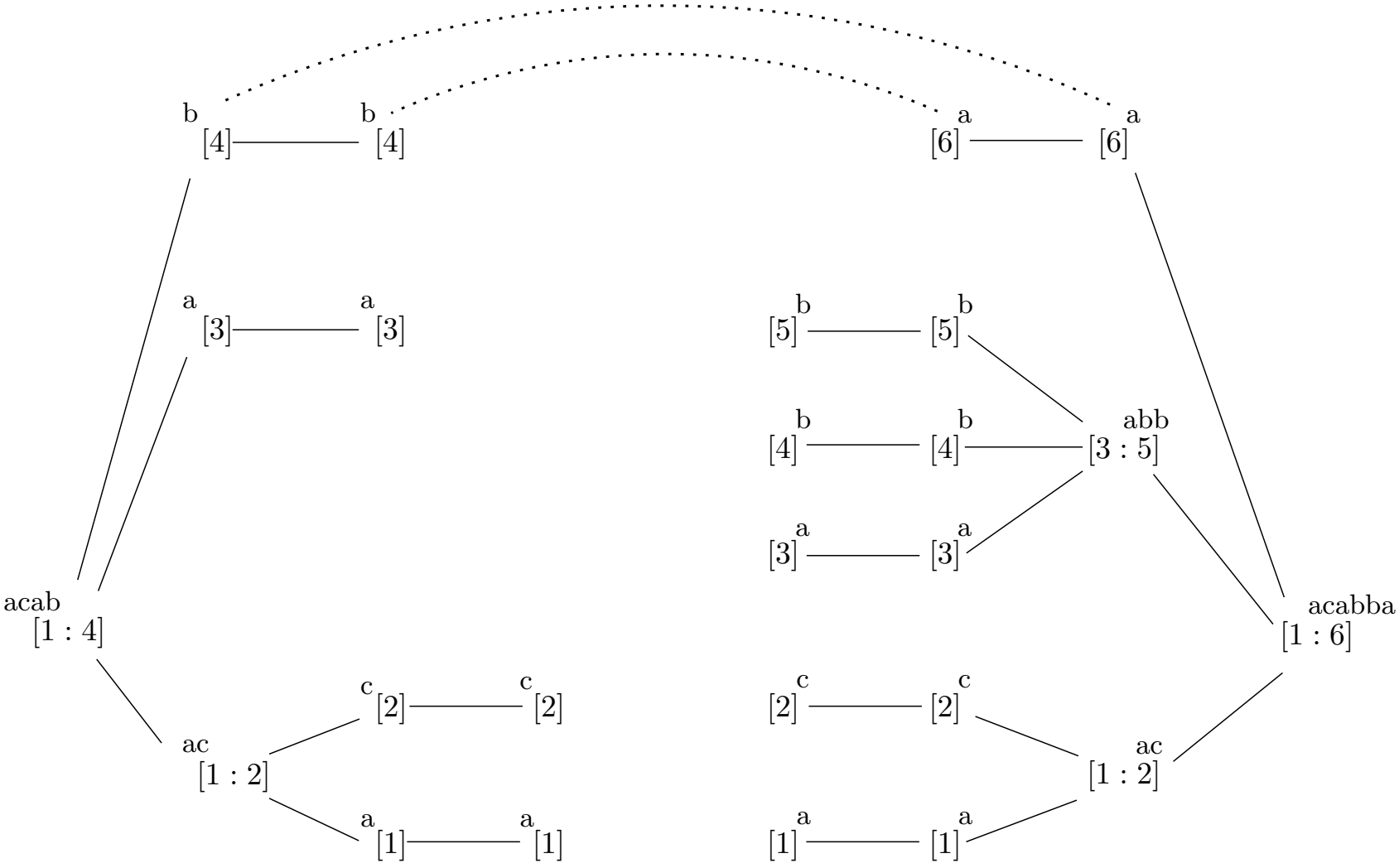




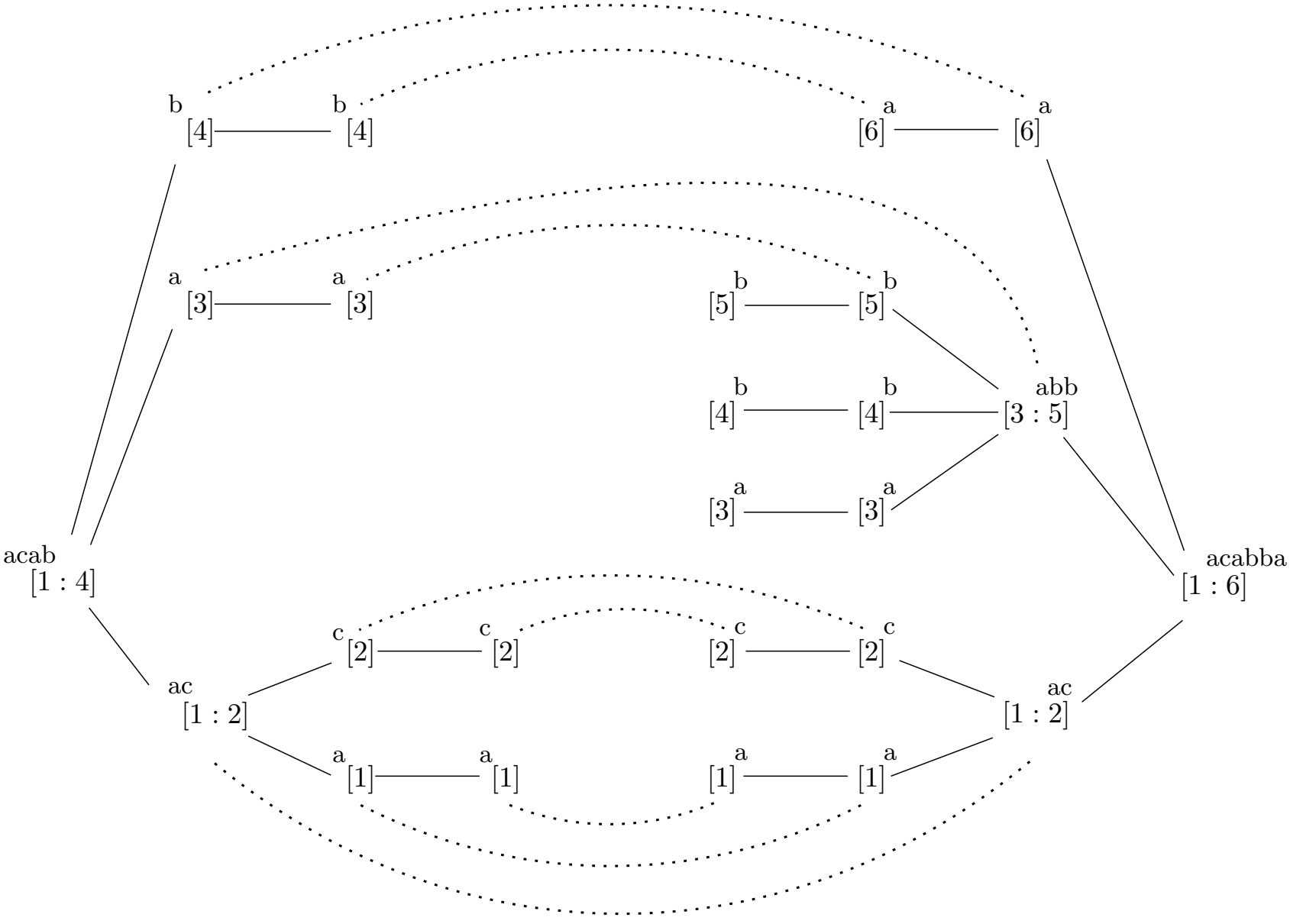
From P-Connection to S-Connection



From P-Connection to S-Connection



From P-Connection to S-Connection



From P-Connection to S-Connection

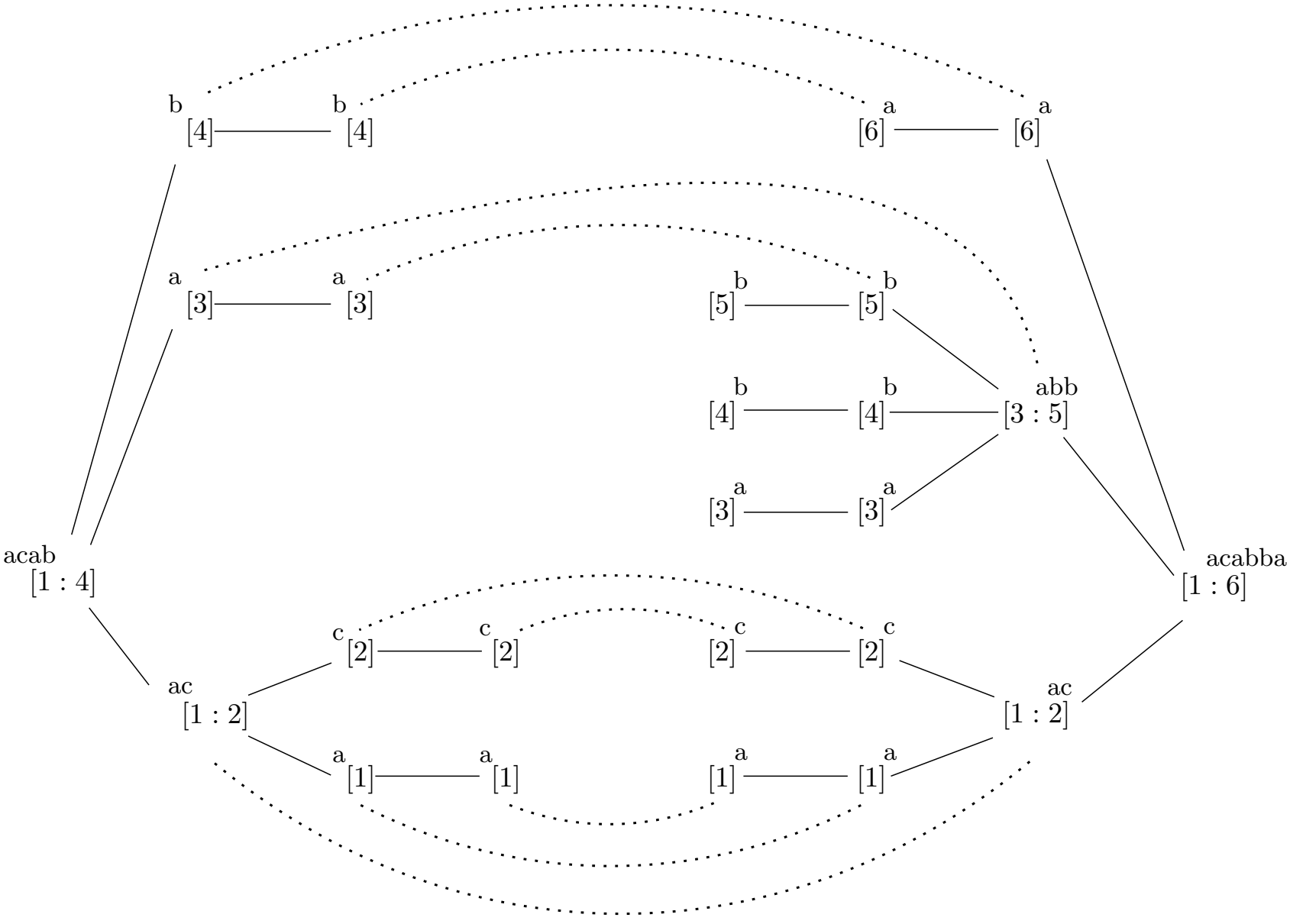
How to refine the P-Connection:

- Let $k \geq 1$. Let a, b be k-blocks in the word t, resp. s, with $a \sim_k b$.
- Let a' be child of a, b' be child of b.
- $a' \nsim_{k+1} b'$ if and only if there exists a letter x such that $s[\mathtt{next}_s(a', x) + 1 : n] \nsim_k t[\mathtt{next}_t(b', x) + 1 : n']$.

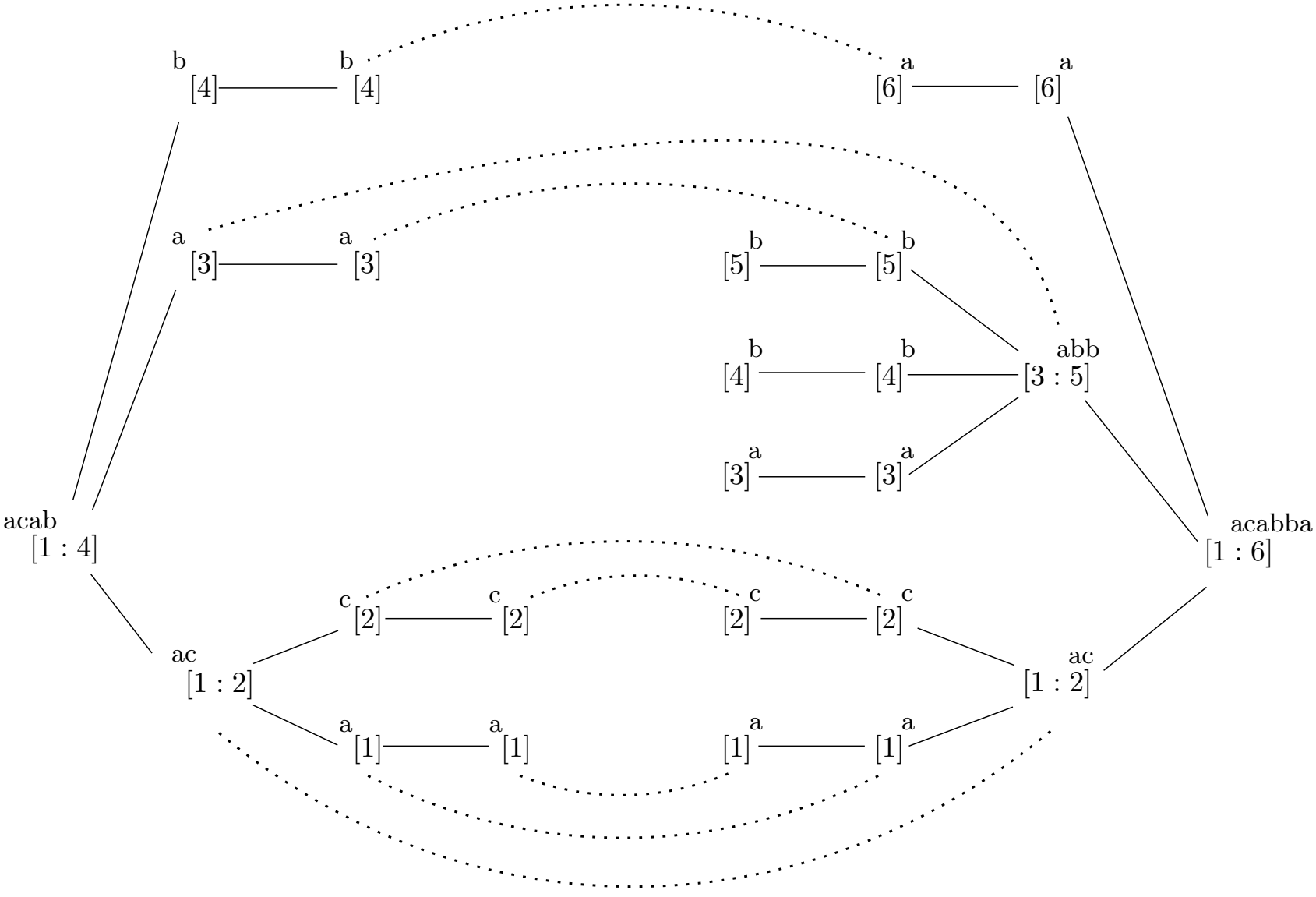
...where next is defined as...

$\mathtt{next}_w(j, x)$: the leftmost position of x in $w[j : |w|]$.
For block $a = [m_a : n_a]$ of w and letter x:
$\mathtt{next}_w(a, x) = \mathtt{next}_w(n_a, x)$.

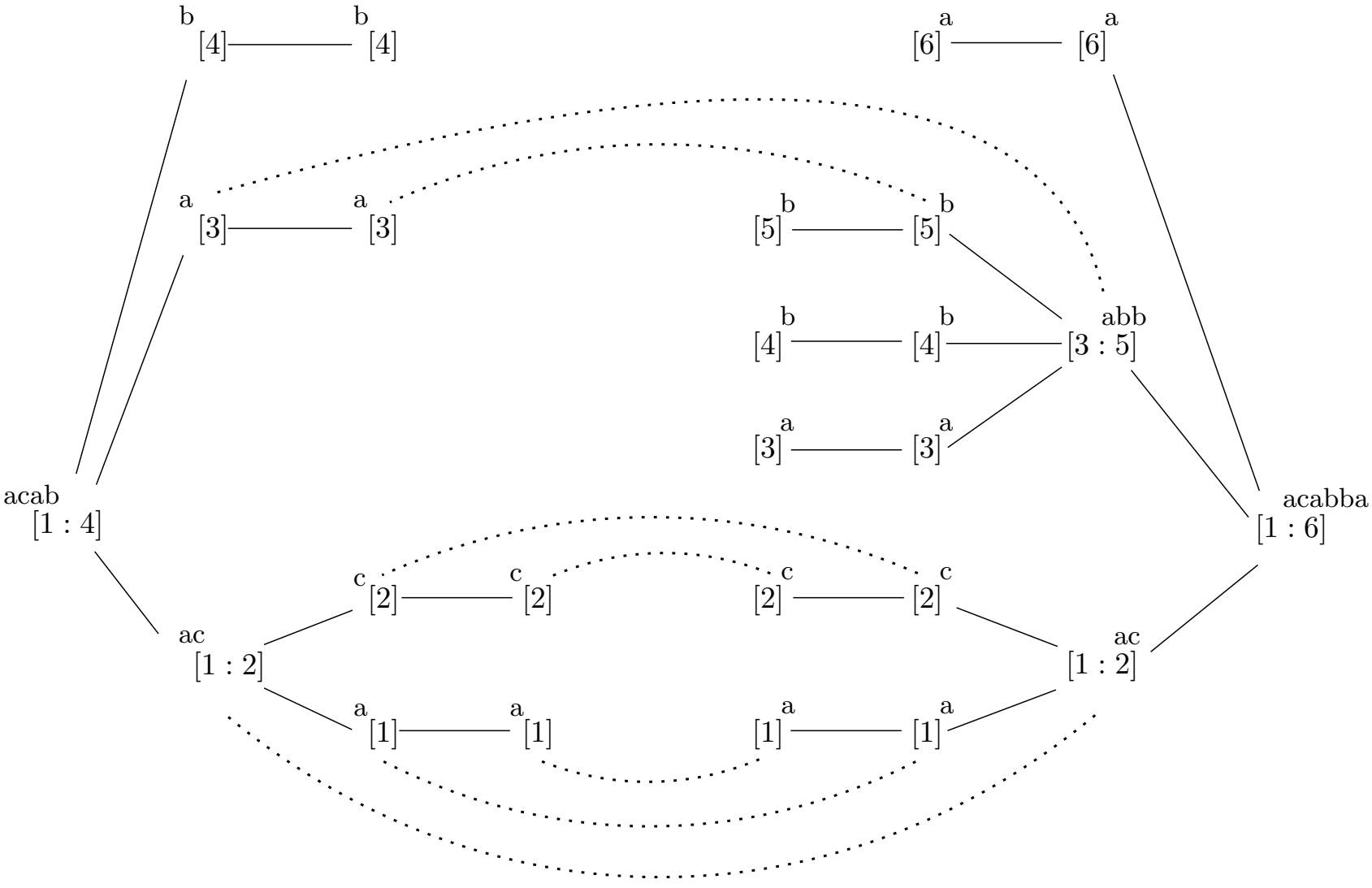
From P-Connection to S-Connection



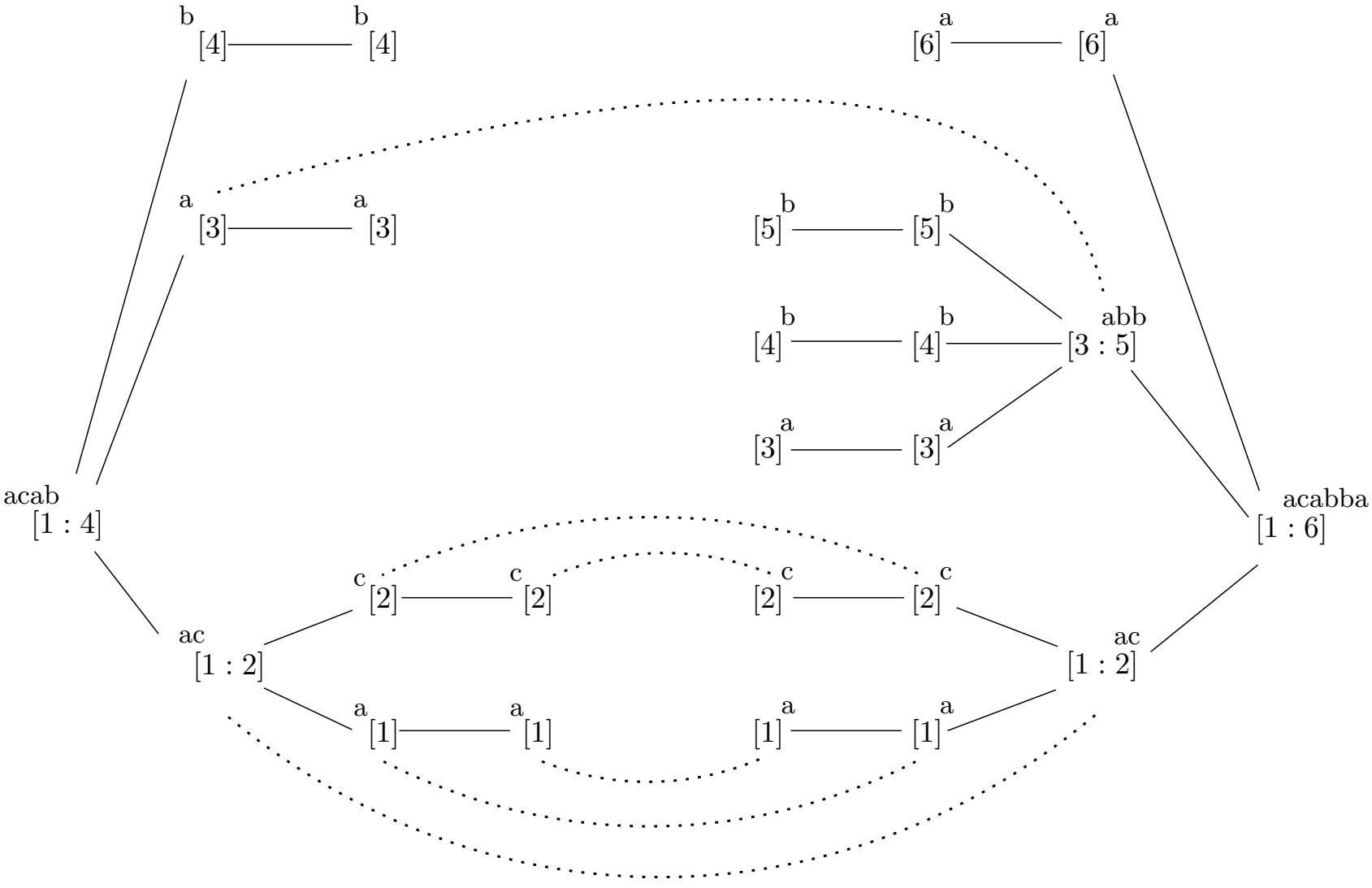
From P-Connection to S-Connection



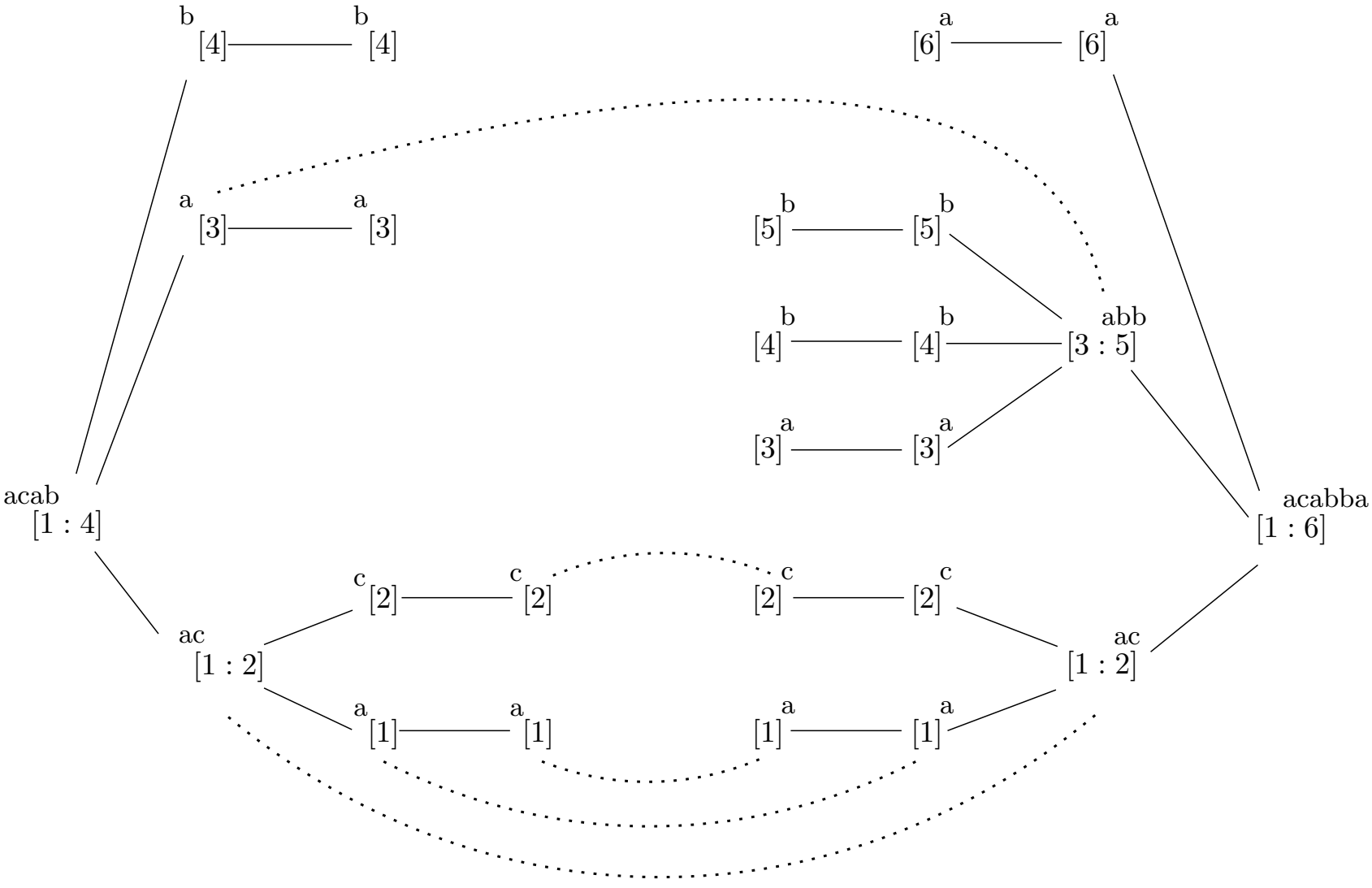
From P-Connection to S-Connection



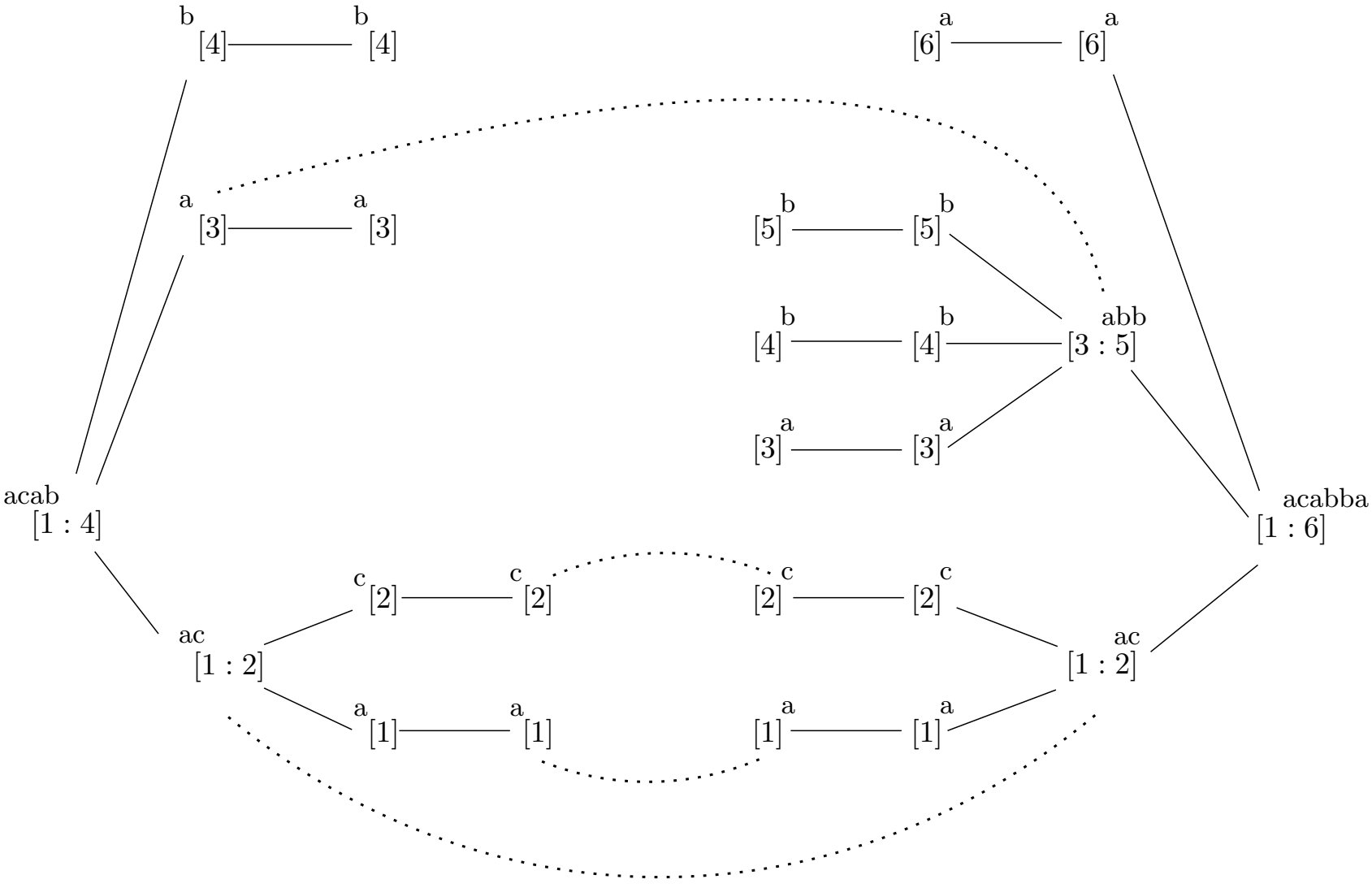
From P-Connection to S-Connection



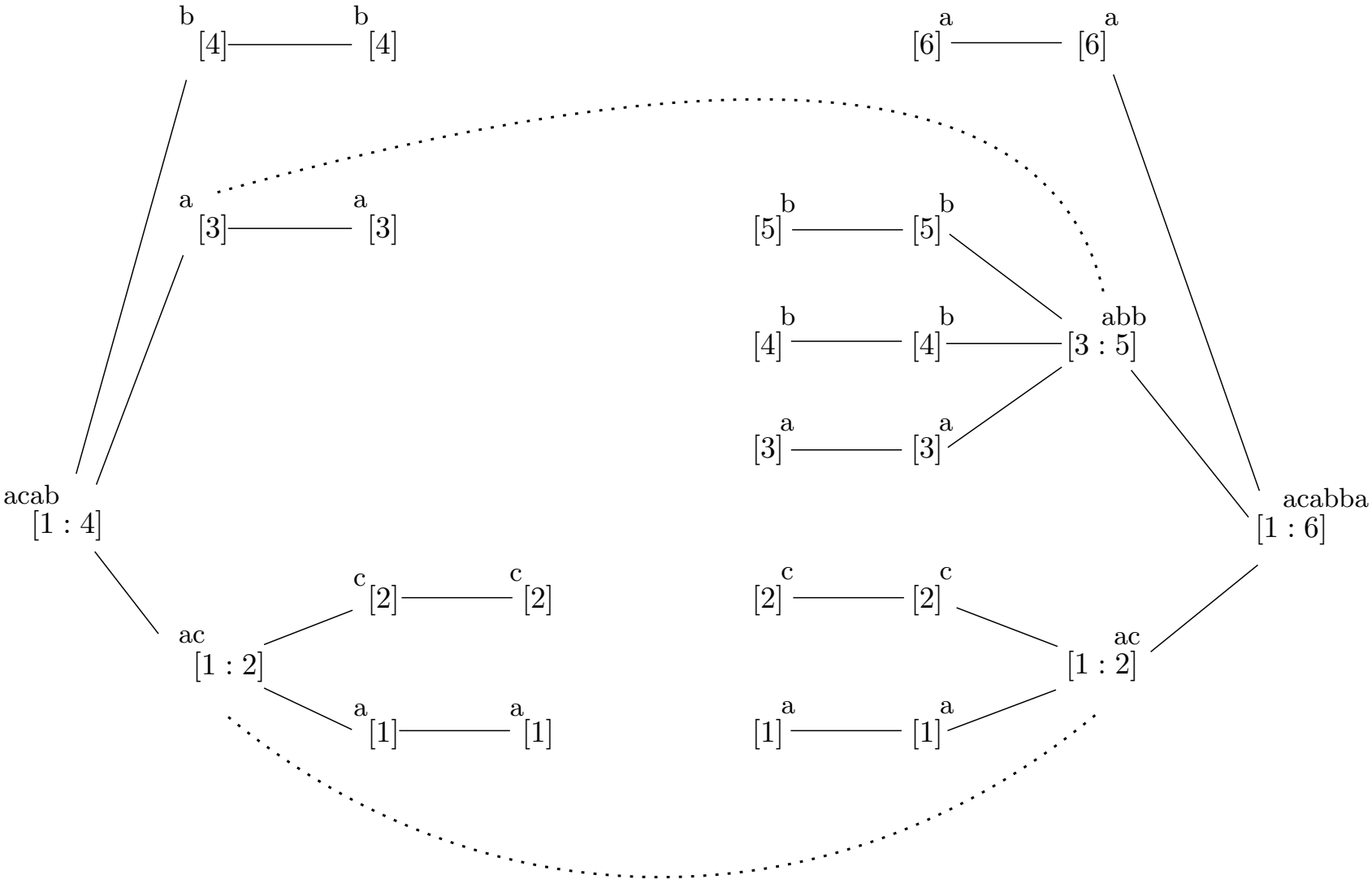
From P-Connection to S-Connection



From P-Connection to S-Connection



From P-Connection to S-Connection



Additional Notes and Analysis

- Solution of MaxSimK: last level k where the k-blocks containing position 1 of the input words are equivalent
- Distinguishing word can be obtained
- By efficiently using interval-union-find and -split-find data structures the algorithm achieves an optimal linear runtime

Theorem

MaxSimK *can be solved in linear time.*

Connected and Future Work

Edit distance to $\sim_k$-equivalence:

- What is the minimum number of edit operations we need to apply to u such that we obtain a word $\sim_k$-equivalent to w?

Connected and Future Work

Edit distance to $\sim_k$-equivalence:

- What is the minimum number of edit operations we need to apply to u such that we obtain a word $\sim_k$-equivalent to w?
- Preliminary results [STACS 2021]: what is the minimum number of edit-operations we need to apply to u such that we obtain a word which has all possible words of length k as subsequences (i.e., it is k-universal)?

Connected and Future Work

Edit distance to $\sim_k$-equivalence:

- What is the minimum number of edit operations we need to apply to u such that we obtain a word $\sim_k$-equivalent to w?
- Preliminary results [STACS 2021]: what is the minimum number of edit-operations we need to apply to u such that we obtain a word which has all possible words of length k as subsequences (i.e., it is k-universal)?
  - To increase (respectively, decrease) the *universality* insertions (respectively, deletions) are enough.

Connected and Future Work

Edit distance to $\sim_k$-equivalence:

- What is the minimum number of edit operations we need to apply to u such that we obtain a word $\sim_k$-equivalent to w?
- Preliminary results [STACS 2021]: what is the minimum number of edit-operations we need to apply to u such that we obtain a word which has all possible words of length k as subsequences (i.e., it is k-universal)?
  - To increase (respectively, decrease) the *universality* insertions (respectively, deletions) are enough. Substitutions only $\rightarrow$ *Hamming distance*

Connected and Future Work

Edit distance to $\sim_k$-equivalence:

- What is the minimum number of edit operations we need to apply to u such that we obtain a word $\sim_k$-equivalent to w?
- Preliminary results [STACS 2021]: what is the minimum number of edit-operations we need to apply to u such that we obtain a word which has all possible words of length k as subsequences (i.e., it is k-universal)?
  - To increase (respectively, decrease) the *universality* insertions (respectively, deletions) are enough. Substitutions only $\rightarrow$ *Hamming distance*
  - $O(nk)$, based on dynamic programming + efficient data structures.
  - $(n \log^{O(1)} \sigma)$ when we are interested in *increasing the universality*. The problem is equivalent to computing a minimum weight k-link path in a DAG which fulfills the concave Monge property.
  - So $O(n)$ for constant k or constant σ (but only when increasing the universality).

Connected and Future Work

Absent subsequences:

- Absent substrings: an important topic in stringology. How about absent subsequences?
- Preliminary results:
  - *Minimal absent subsequences*: absent subsequences whose every subsequence is not absent,
  - *Shortest absent subsequences*: absent subsequences of minimal length,
  - Deep connection to the notion of universality,

Connected and Future Work

Absent subsequences:

- Absent substrings: an important topic in stringology. How about absent subsequences?
- Preliminary results:
  - *Minimal absent subsequences*: absent subsequences whose every subsequence is not absent,
  - *Shortest absent subsequences*: absent subsequences of minimal length,
  - Deep connection to the notion of universality,
  - Precise characterizations of the set of minimal absent subsequences and shortest absent subsequences occurring in a word,
  - Examples of words w having an exponential number (w.r.t. the length of $|w|$) of minimal absent subsequences and shortest absent subsequences, respectively.
  - We identified the word with a maximum number of shortest absent subsequences among all k-universal words

Connected and Future Work

Absent subsequences:

- Absent substrings: an important topic in stringology. How about absent subsequences?
- Preliminary results:
  - *Minimal absent subsequences*: absent subsequences whose every subsequence is not absent,
  - *Shortest absent subsequences*: absent subsequences of minimal length,
  - Deep connection to the notion of universality,
  - Precise characterizations of the set of minimal absent subsequences and shortest absent subsequences occurring in a word,
  - Examples of words w having an exponential number (w.r.t. the length of $|w|$) of minimal absent subsequences and shortest absent subsequences, respectively.
  - We identified the word with a maximum number of shortest absent subsequences among all k-universal words
  - We can compute efficiently minimal/shortest absent subsequences, as well as compact representations of all such words.

The End

Thank You!

And many thanks to my co-authors:
Joel D. Day, Pamela Fleischmann, Paweł Gawrychowski,
Maria Kosche, Tore Koß, Stefan Siemer.