Agenda for today

- **Suffix Trees**
  - **Introduction and definitions**
  - **Linear-time construction**
  - **Deterministic exact string matching**
- **Time permitting, further related topics**
  - **Suffix automata**
  - **Suffix arrays**

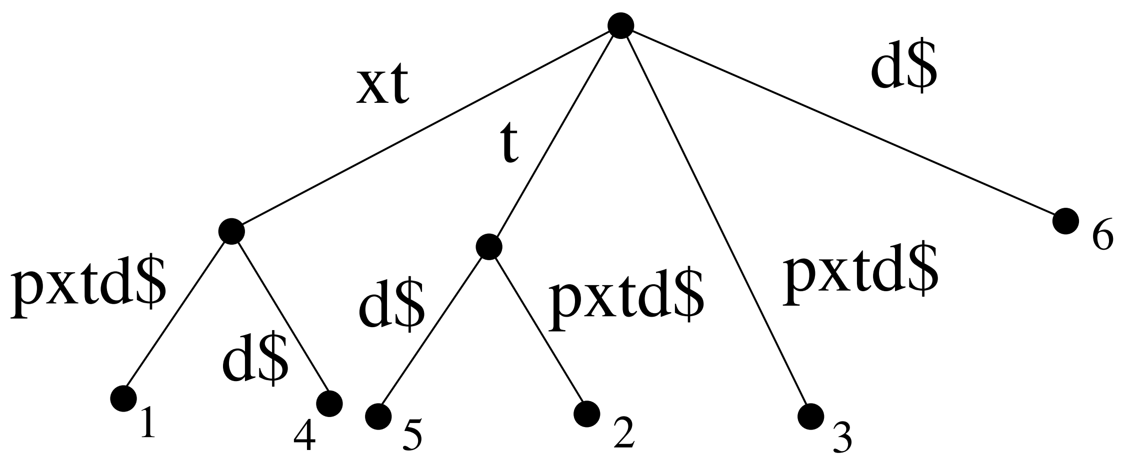
Suffix Trees

- Given a string S of length m, a suffix tree T encodes all suffixes, i.e., $S[i, m]$ for $1 \leq i \leq m$
- Concatenation of symbols labeling edges along the path from root to leaf spells out suffix
- Leaves are numbered with i, the start index of the suffix
- Edges can be labeled with one or more symbols
- Internal (non-leaf) nodes must have more than one child
- No two children leaving a node can start with the same symbol
- Tremendously useful data structure for many problems

x t p x t d $

Start	suffix
1	**x t p x t d** $
2	**t p x t d** $
3	**p x t d** $

Start	suffix
4	**x t d** $
5	**t d** $
6	**d** $



How can we use it?

- **Exact match: e.g., how many times does *xt* occur?**
- **Algorithm: order n (size of pattern) rather than order m (size of text)**
  - **Start at the root**
  - **Match pattern along branches in tree**
  - **Number of leaf nodes is the number of instances**
  - **Indices give you the locations**
- **Useful in scenarios where text is given in advance, and used for many pattern matches**

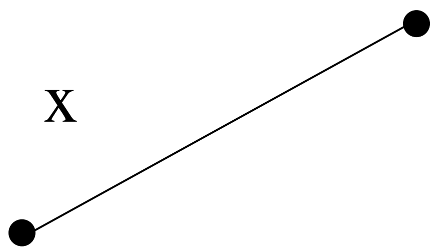
Problem: how to build

- **Naive method too expensive:**
  - **Put suffix $S[1, m]$ into the tree**
  - **Then put $S[i, m]$ into the tree for $2 \leq i \leq m$**
  - **m^2 complexity**
- **Several linear-time suffix tree construction algorithms**
- **We will consider Ukkonen's algorithm, which has several key ideas, including**
  - **Suffix links (similar to failure links)**
  - **String indices instead of characters**

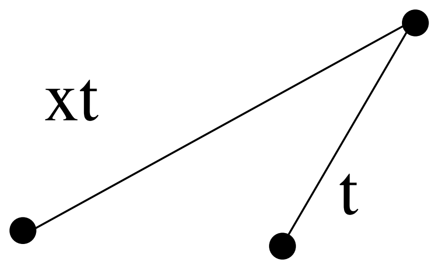
Basic idea

- **Build suffix tree incrementally from left-to-right**
- **Build "implicit" suffix trees (no end-of-string marker)**
- **Extend the implicit suffix trees from the previous step**
- **Convert implicit suffix tree to explicit in final step**

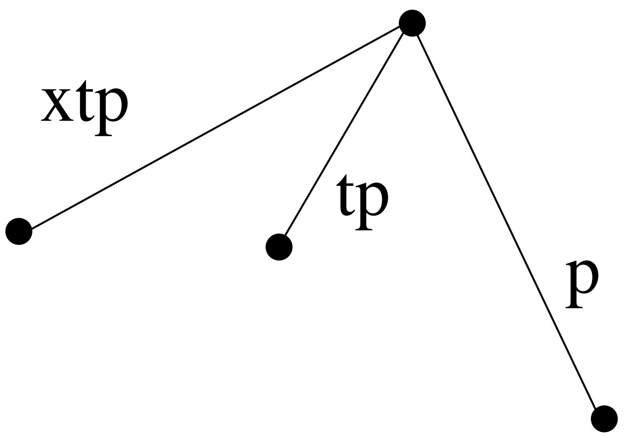
Step 1: x t p x t d



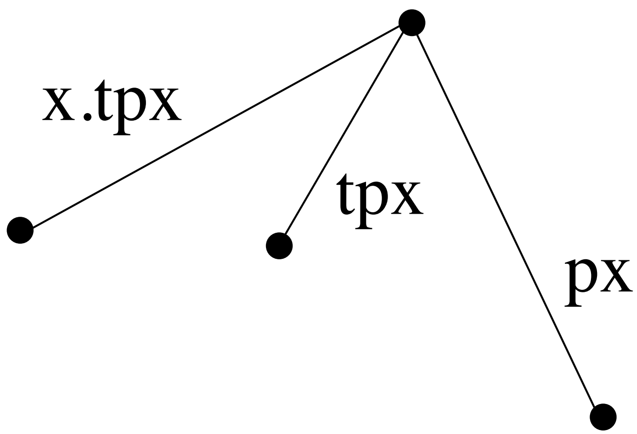
Step 2: x t p x t d



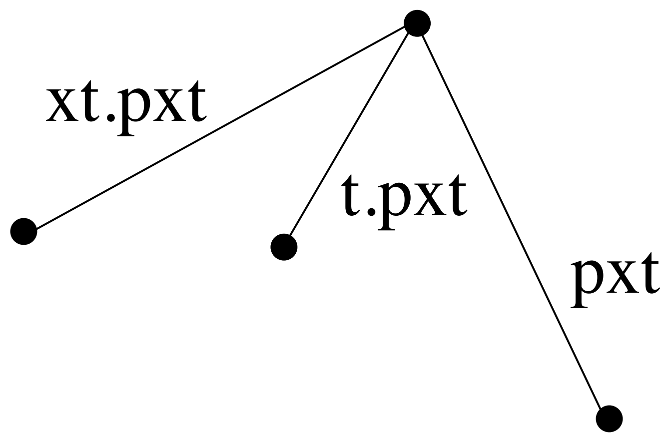
Step 3: x t p x t d



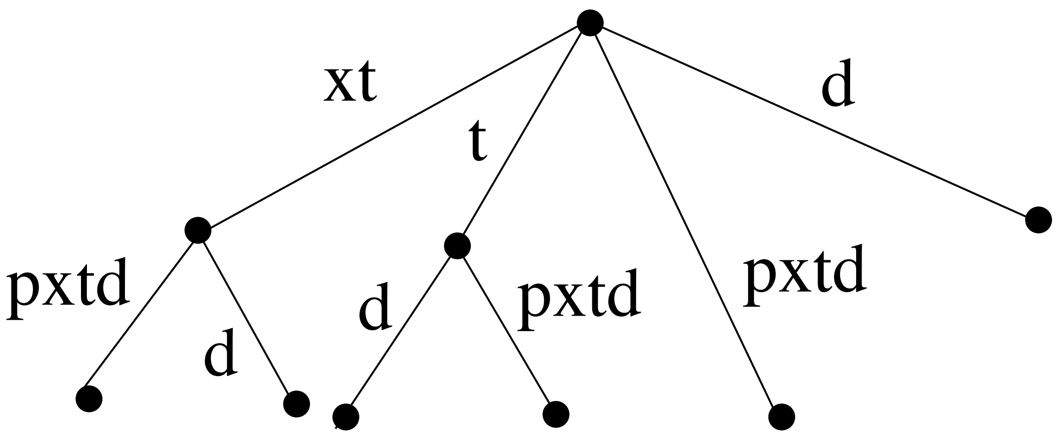
Step 4: x t p x t d



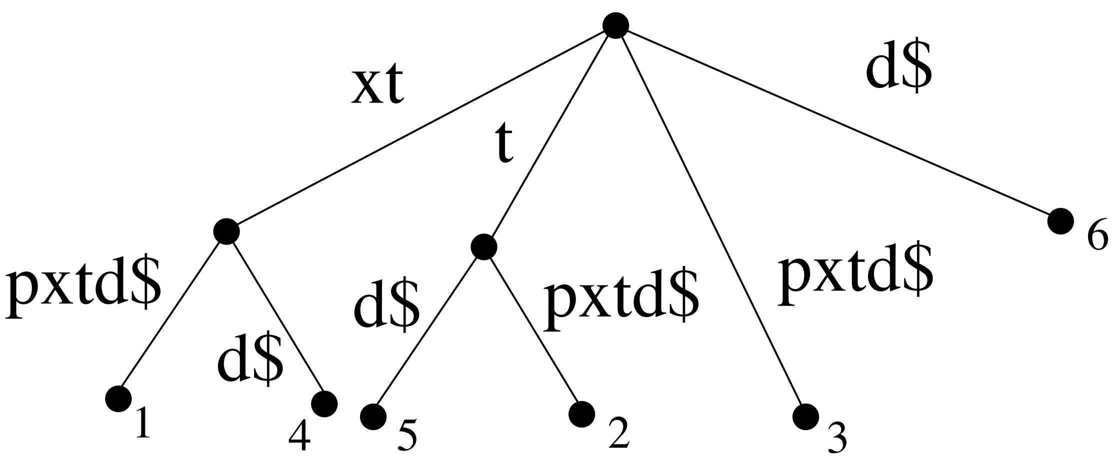
Step 5: x t p x t d



Step 6: x t p x t d



Step 7: finalize



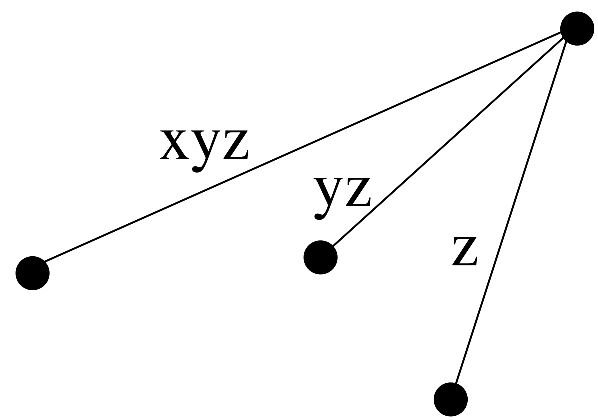
Efficient construction

- To really make this tractable, several clever “tricks”
  - Suffix links (like failure links)
  - Represent sequences as beginning and ending indices
  - Automatic extension for leaves (via global “end” index)
  - Early stopping of suffix link extensions
  - Check only first letter when following suffix link extensions (“Skip/count” trick)
- In aggregate, these make for tractable suffix tree construction

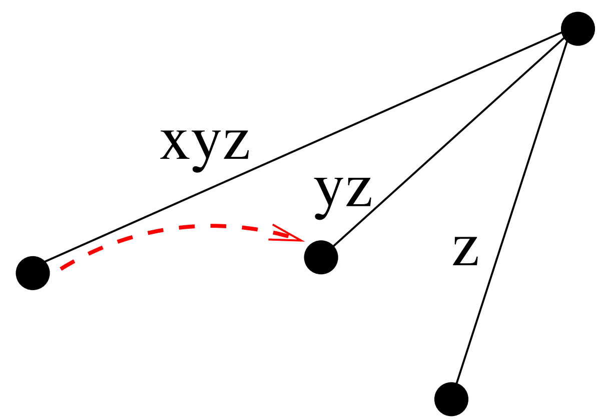
Suffix links

- **Note that every suffix of a suffix in the suffix tree must also be in the suffix tree**
- **For example: if** *abcd* **is in the tree, so are:** *bcd*, *cd*, *d*
- **If we extend a suffix, its suffixes will also be extended**
  - **e.g., if** *abcd+e*, **then** *bcd+e*, *cd+e*, *d+e*, *e*
- **Need to find them (via suffix links) to extend them**
- **Can use the suffix links of parents in tree for quick access**

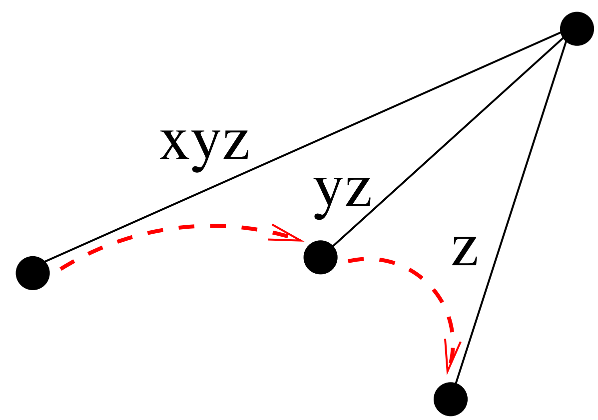
Suffix links: xyz



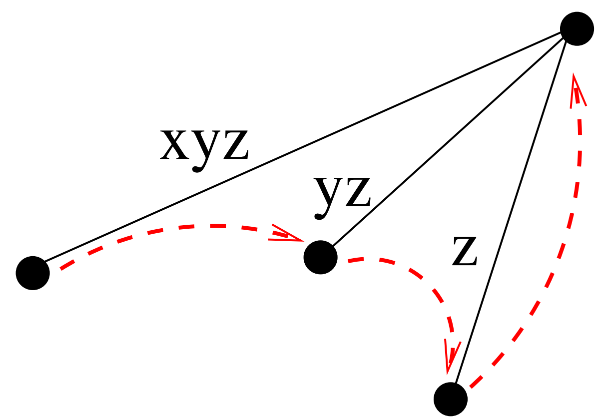
Suffix links: xyz



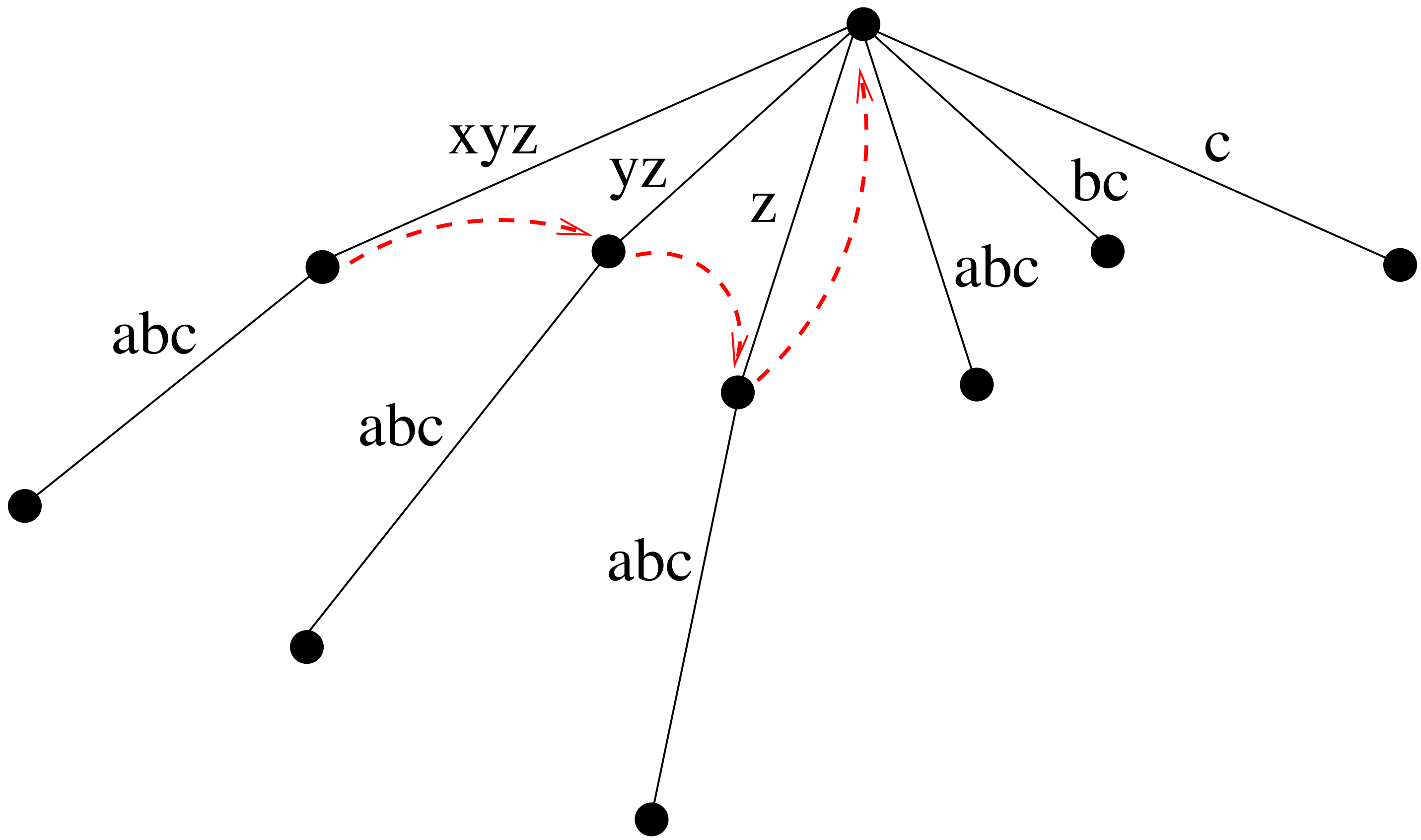
Suffix links: xyz



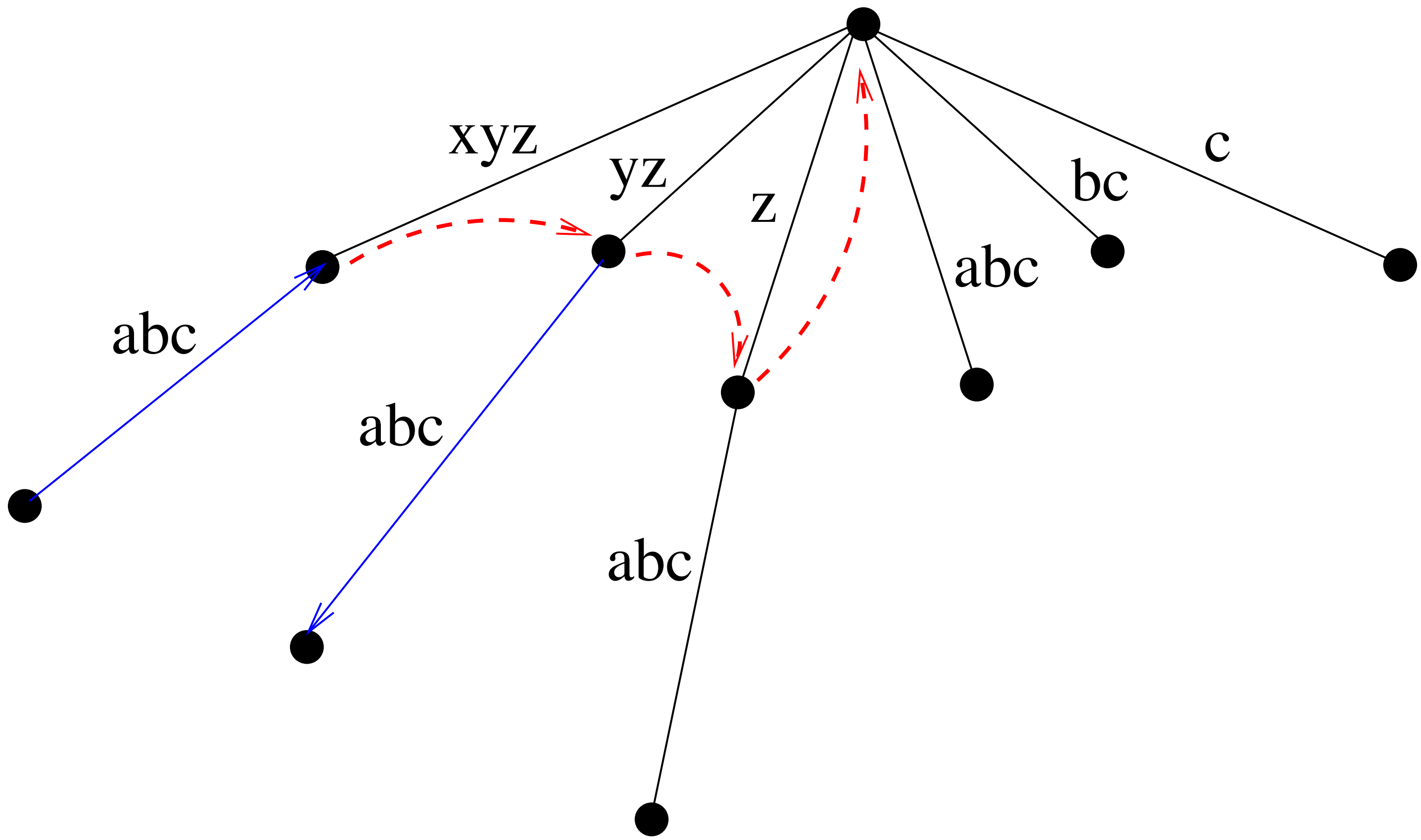
Suffix links: xyz



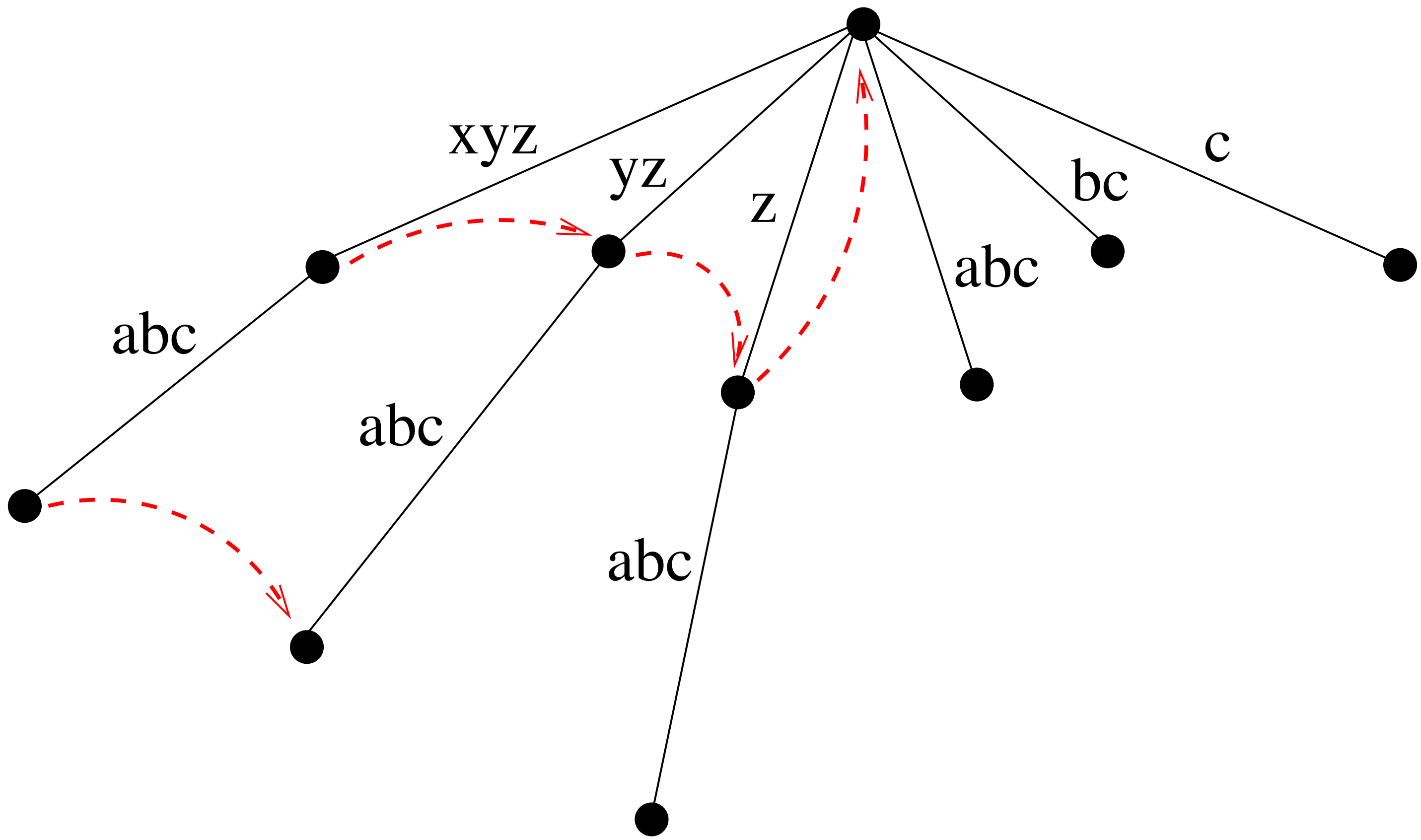
Suffix links: xyzabc



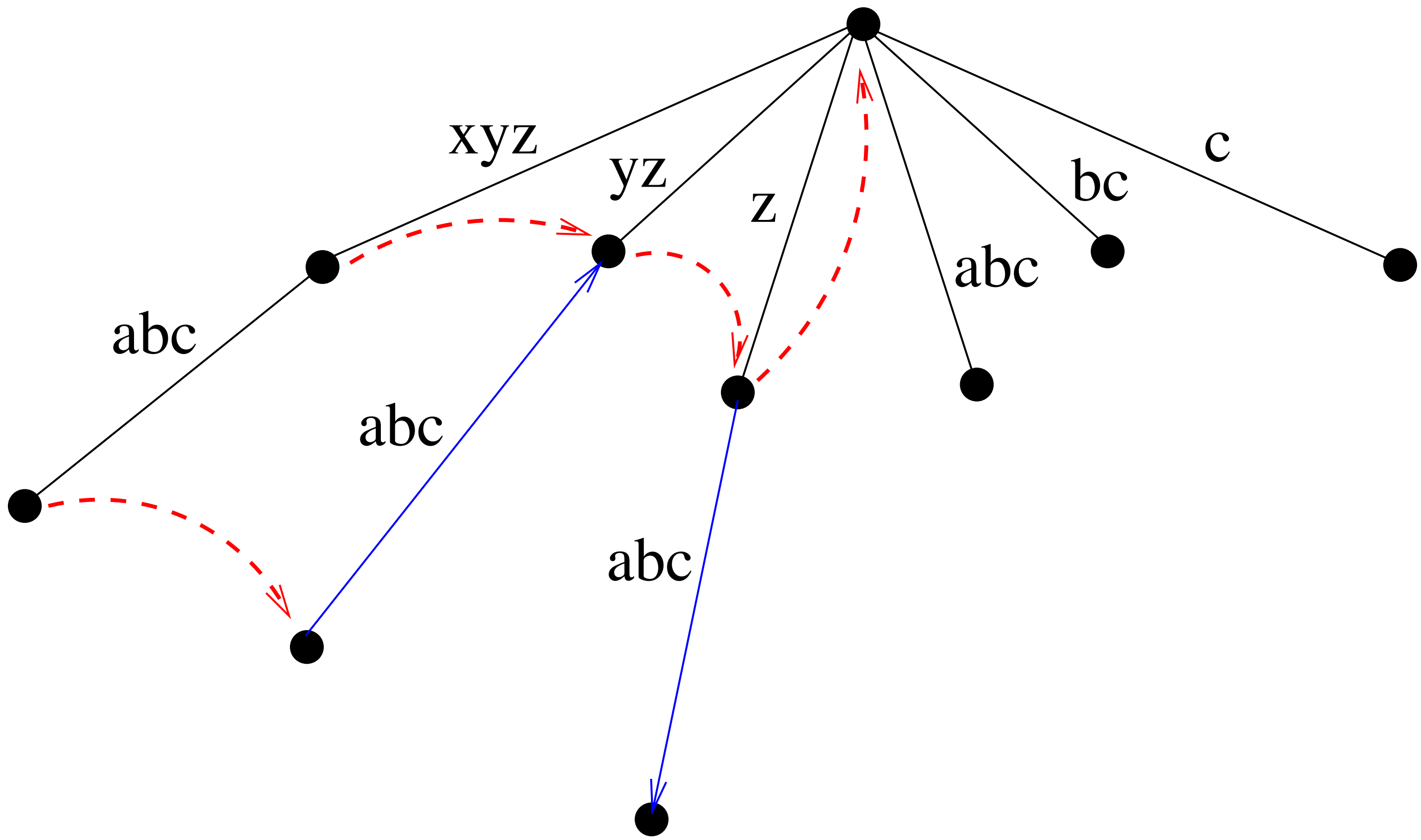
Suffix links: xyzabc



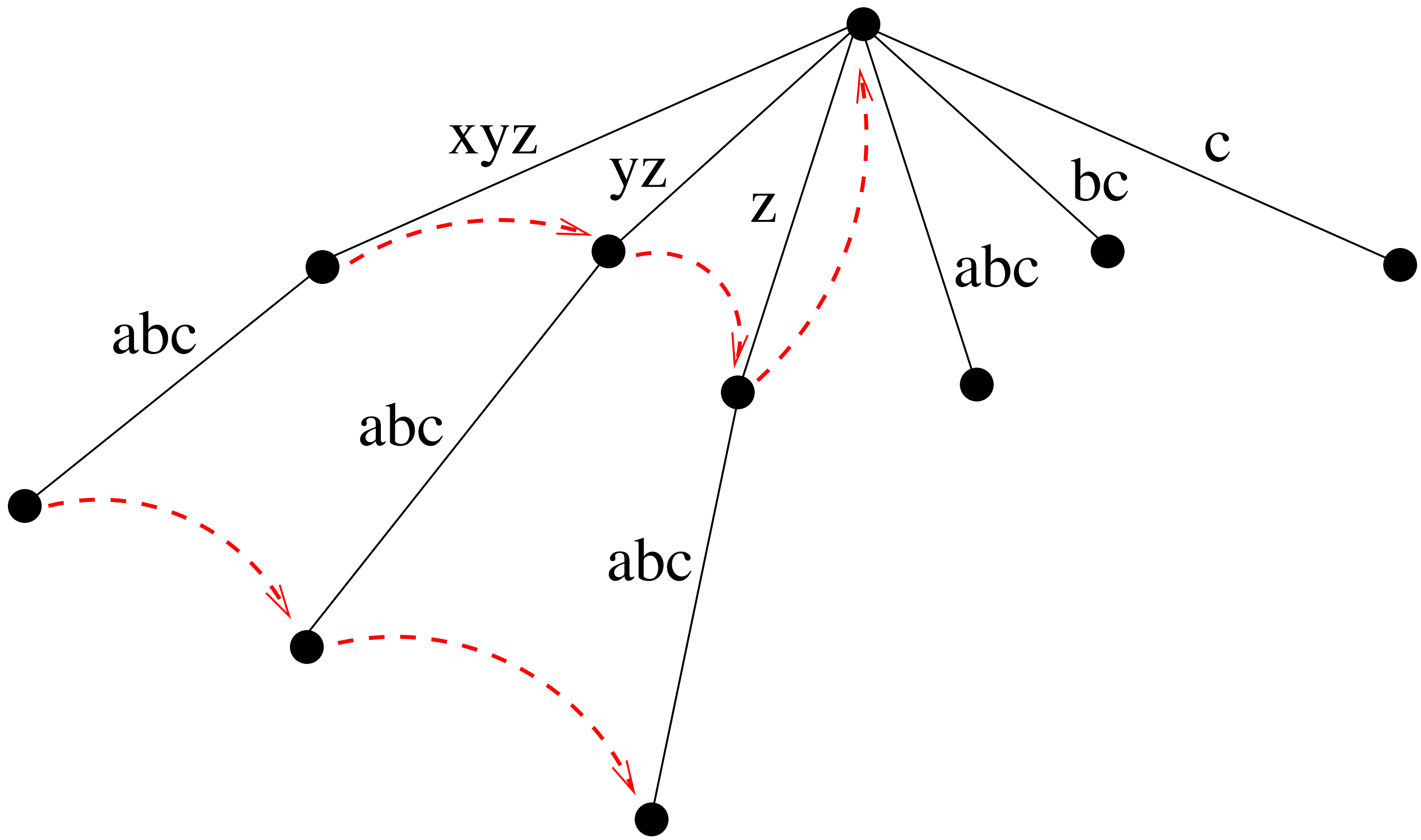
Suffix links: xyzabc



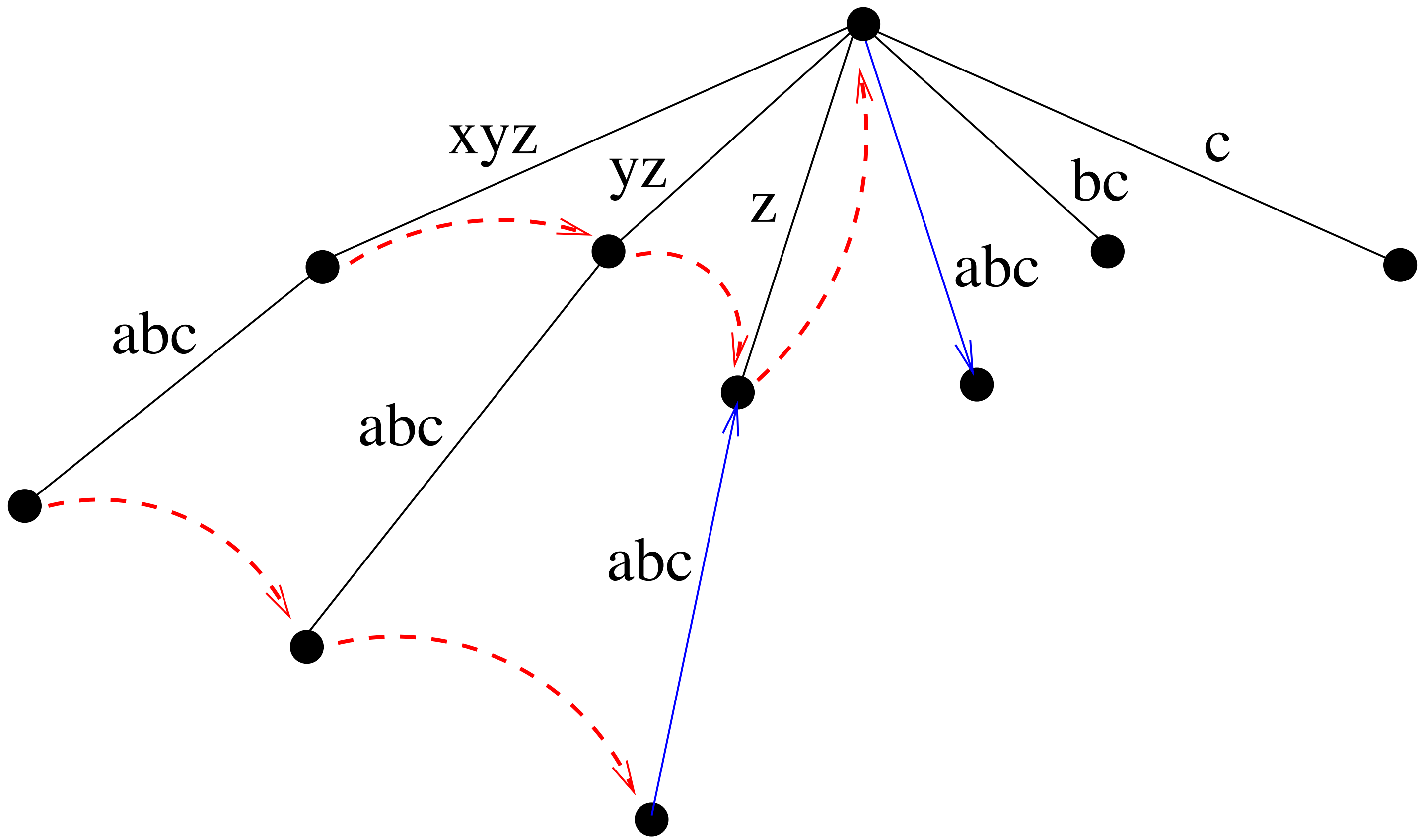
Suffix links: xyzabc



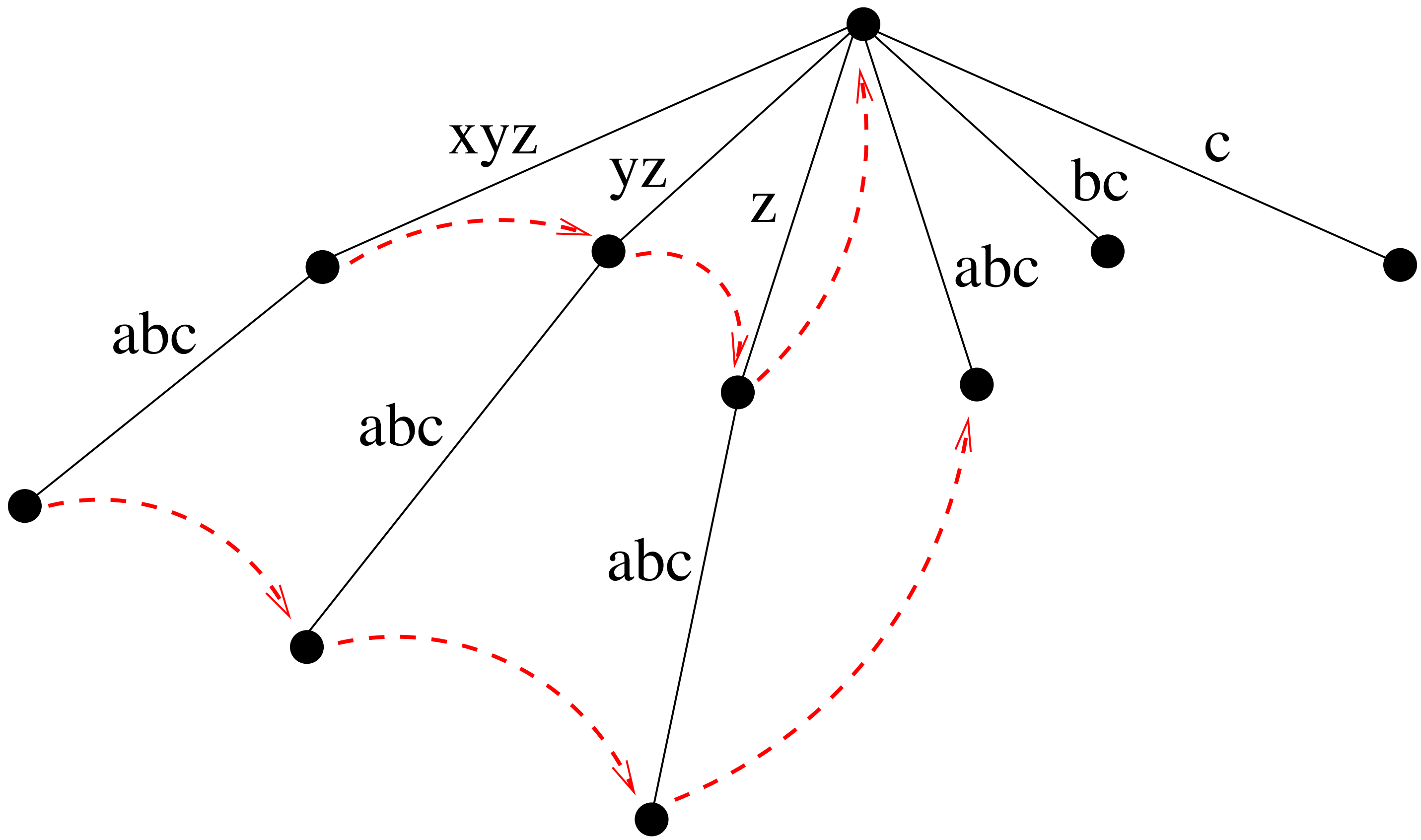
Suffix links: xyzabc



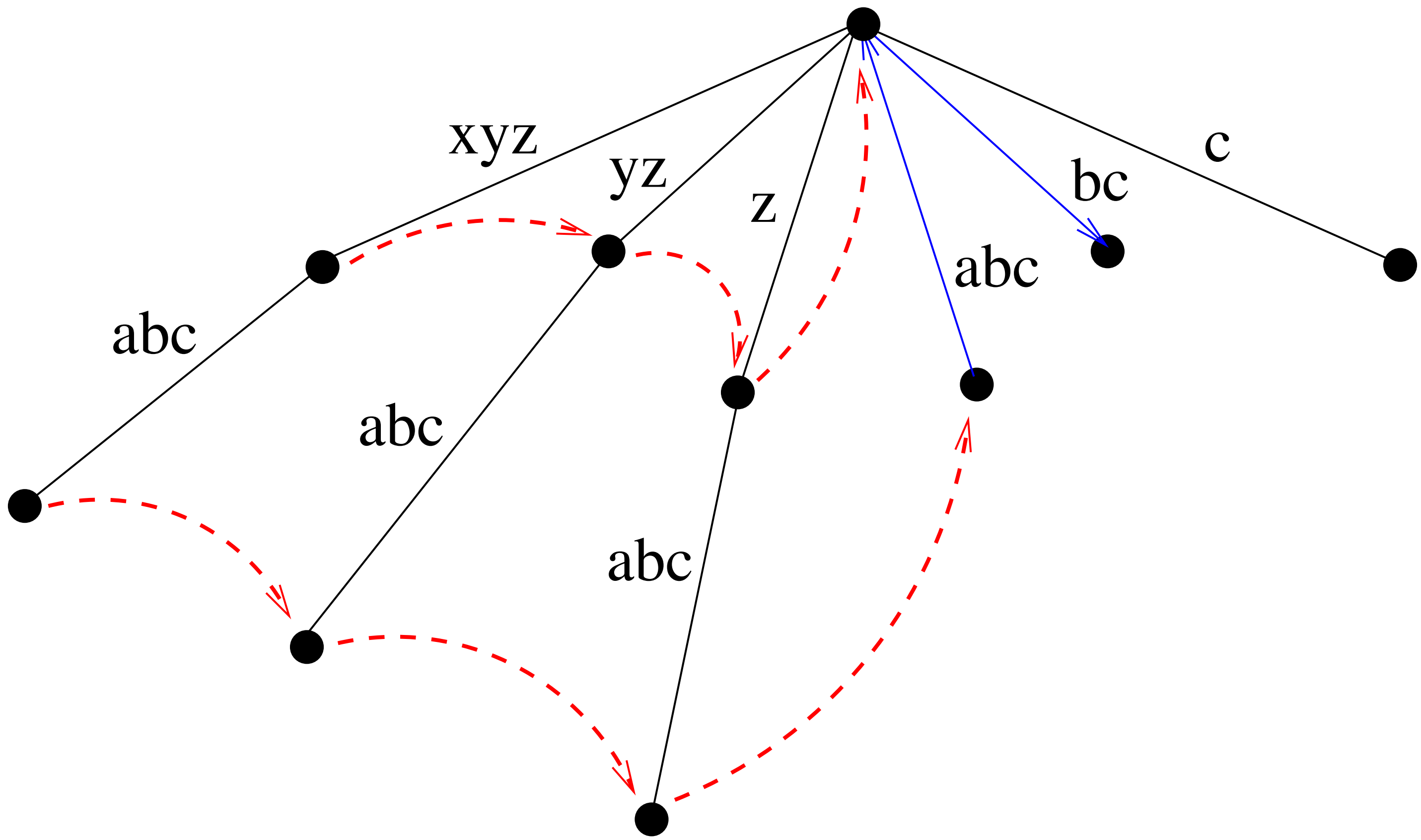
Suffix links: xyzabc



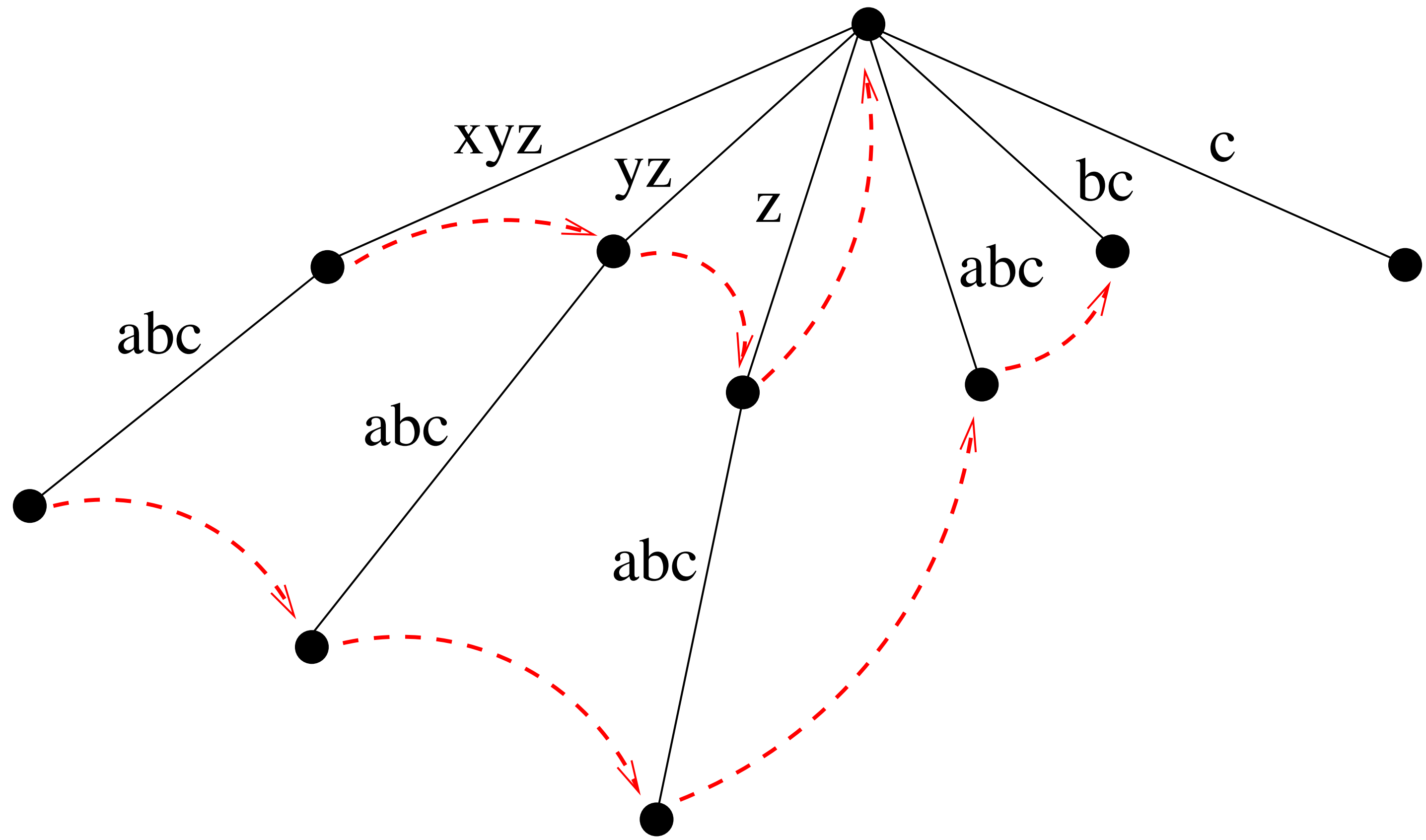
Suffix links: xyzabc



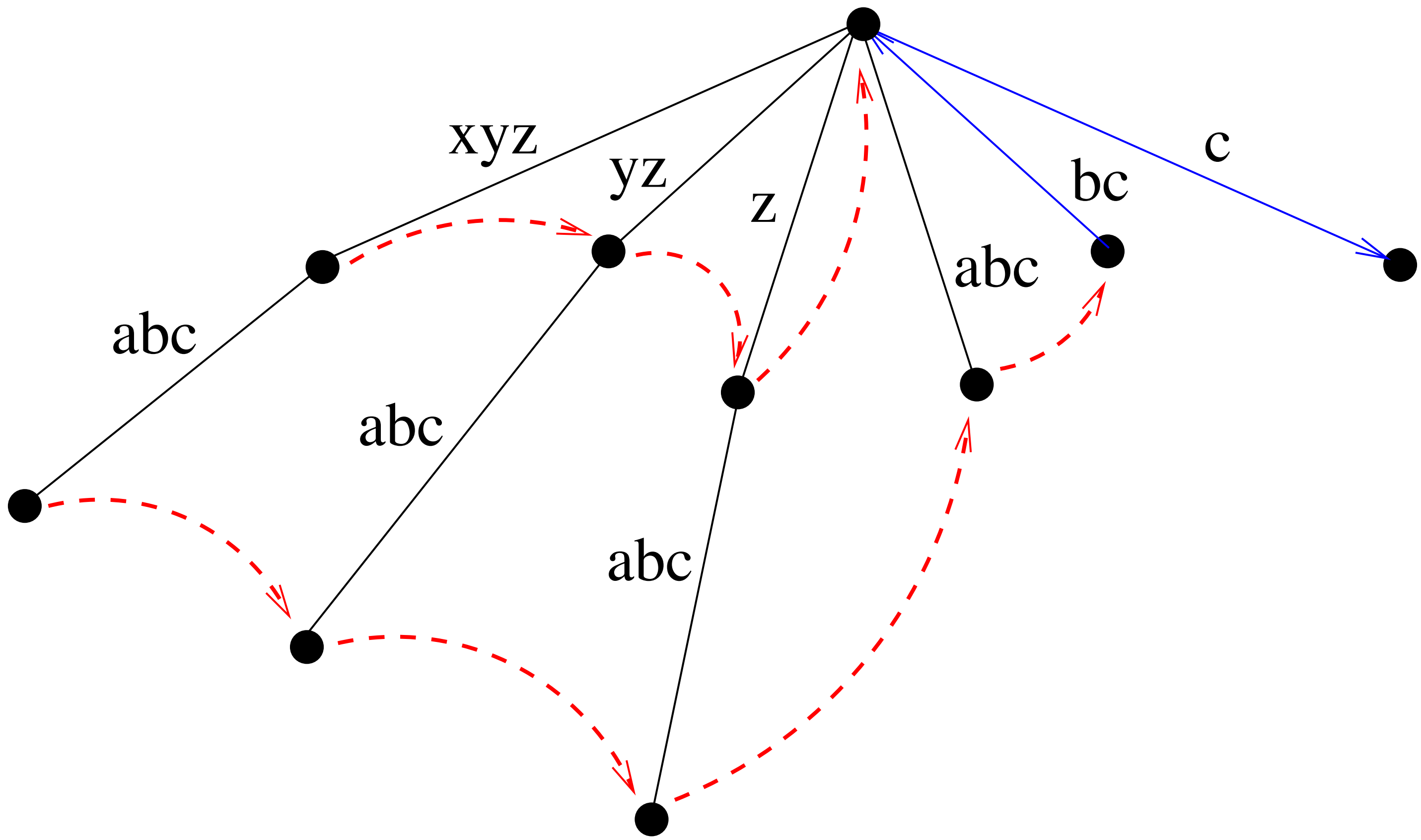
Suffix links: xyzabc



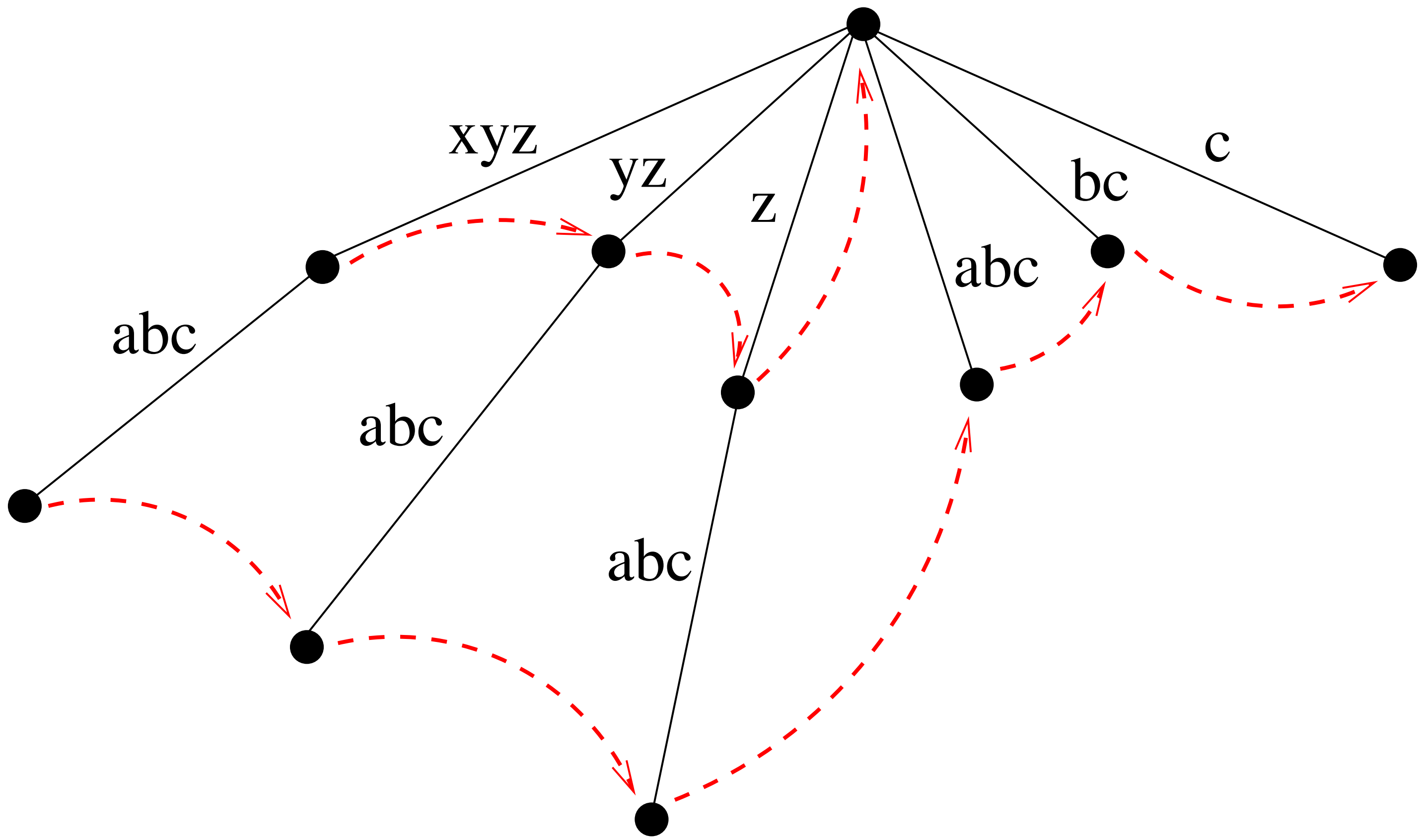
Suffix links: xyzabc



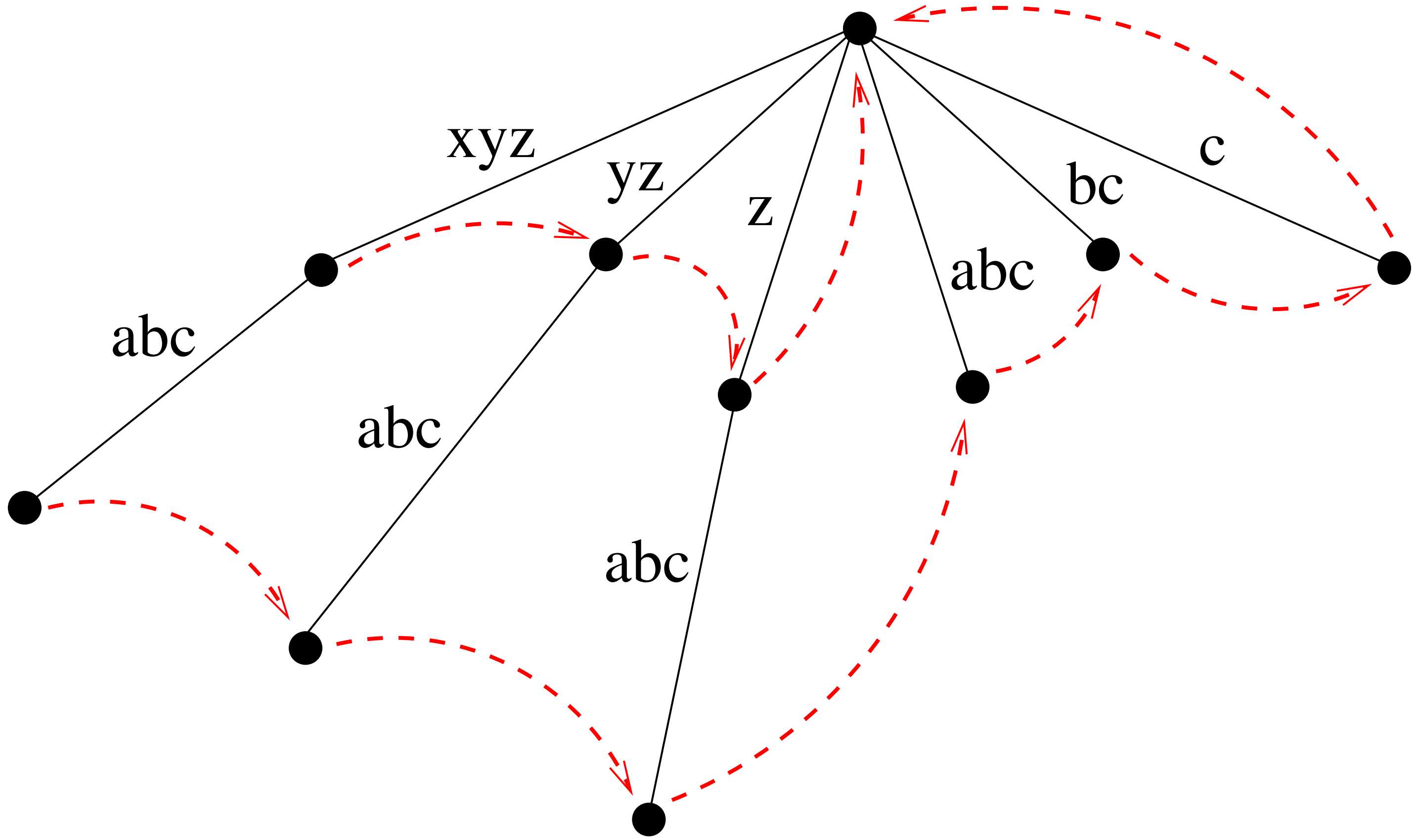
Suffix links: xyzabc



Suffix links: xyzabc



Suffix links: xyzabc



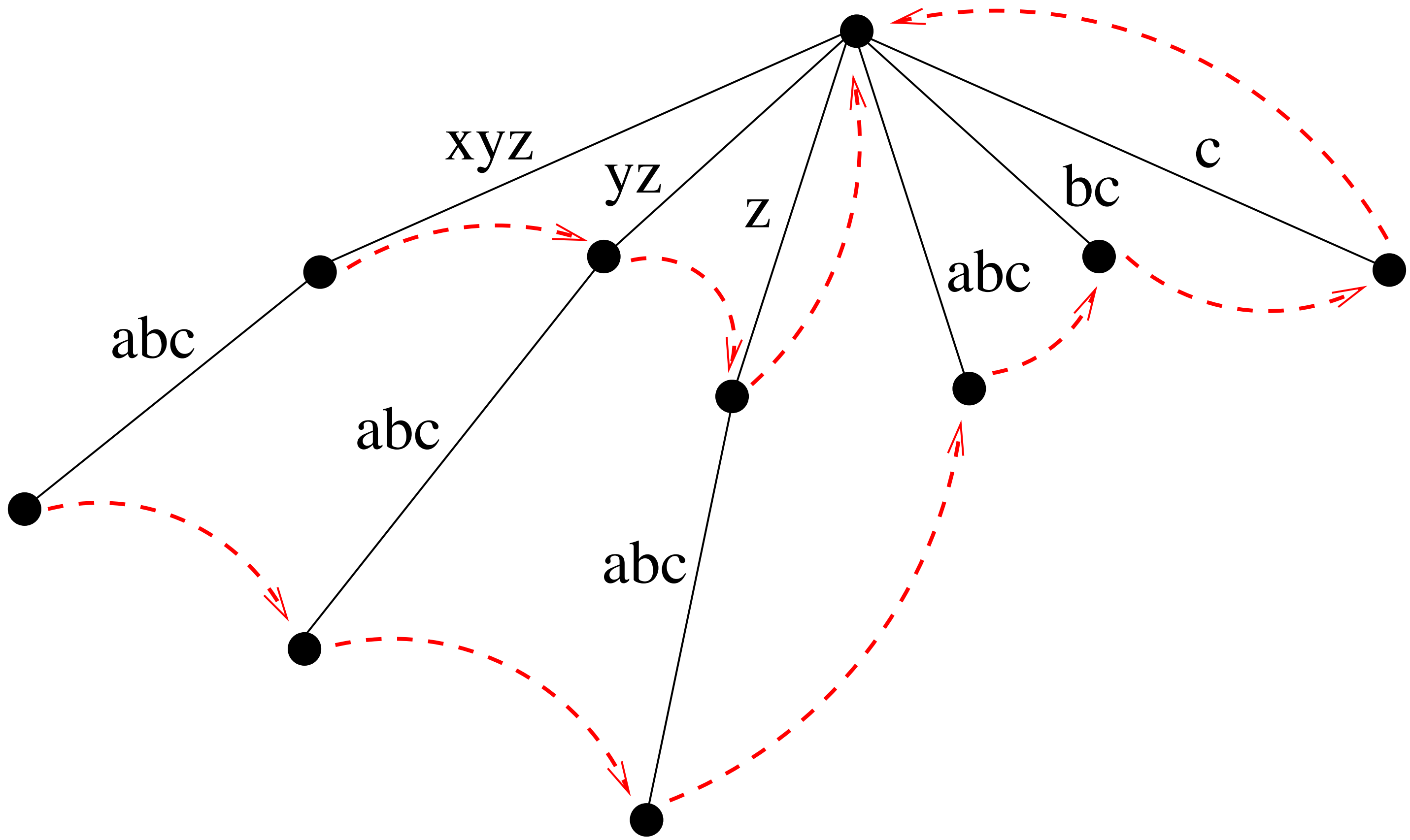
Finding suffix link for node X

- Go to parent node Y
- Denote $x\alpha$ string labeling $Y \rightarrow X$, $x \in \Sigma$, $\alpha \in \Sigma^*$
- If Y is the root of the tree
  - If $\alpha = \epsilon$, set Z to Y
  - Else set Z to state reached from Y with transitions labeled α
- Else
  - Set Y to destination state of suffix link from Y
  - Set Z to state reached from Y with transitions labeled $x\alpha$
- Create suffix link from X to Z

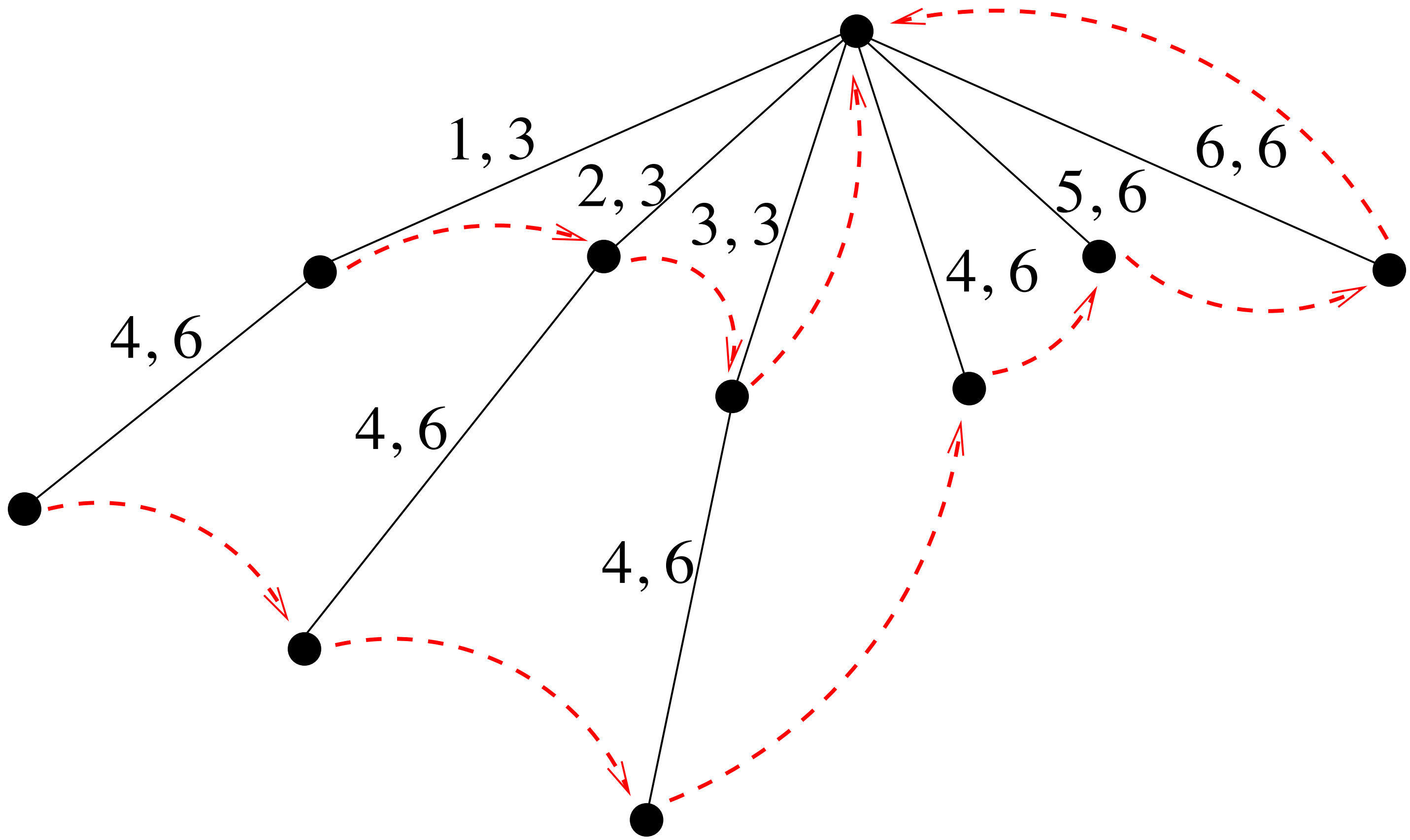
Some notes about suffix links

- **Every previously created node (except root) has suffix link**
- **May require multiple transitions to generate all of string**
- **String is guaranteed to exist from suffix link of the parent (or root)**
  - **Only one transition leaving each state per letter**
  - **Hence only need to check first letter (skip/count)**

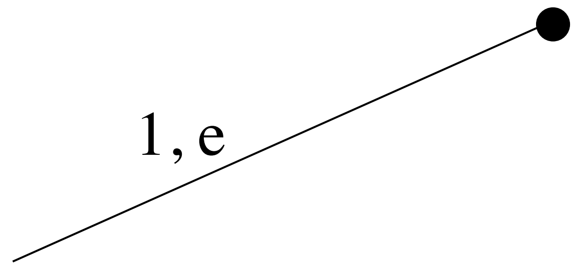
Begin/End indices: xyzabc



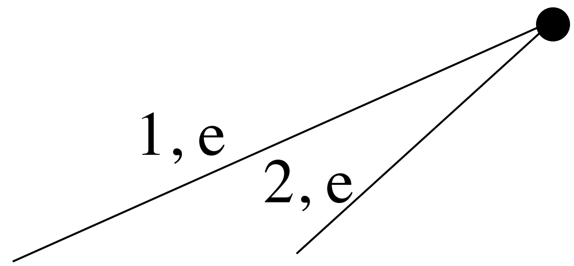
Begin/End indices: xyzabc



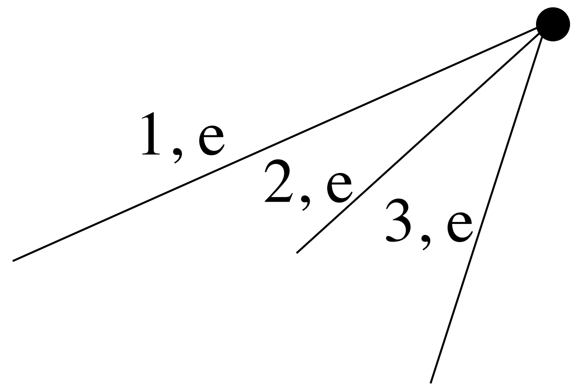
Global end index: xyzabc



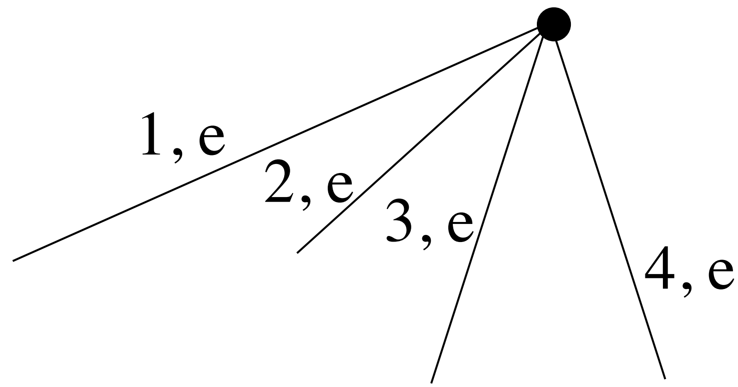
Global end index: xyzabc



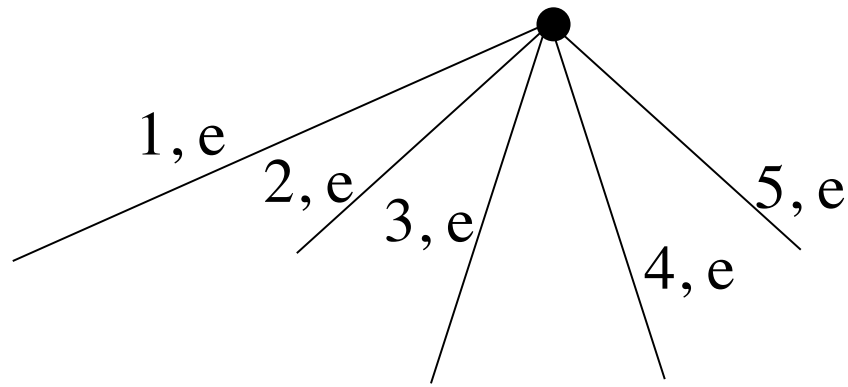
Global end index: xyzabc



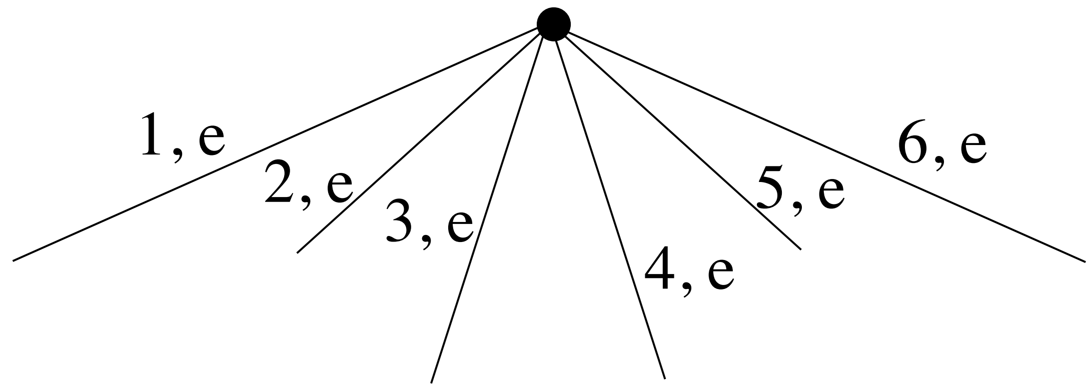
Global end index: xyzabc



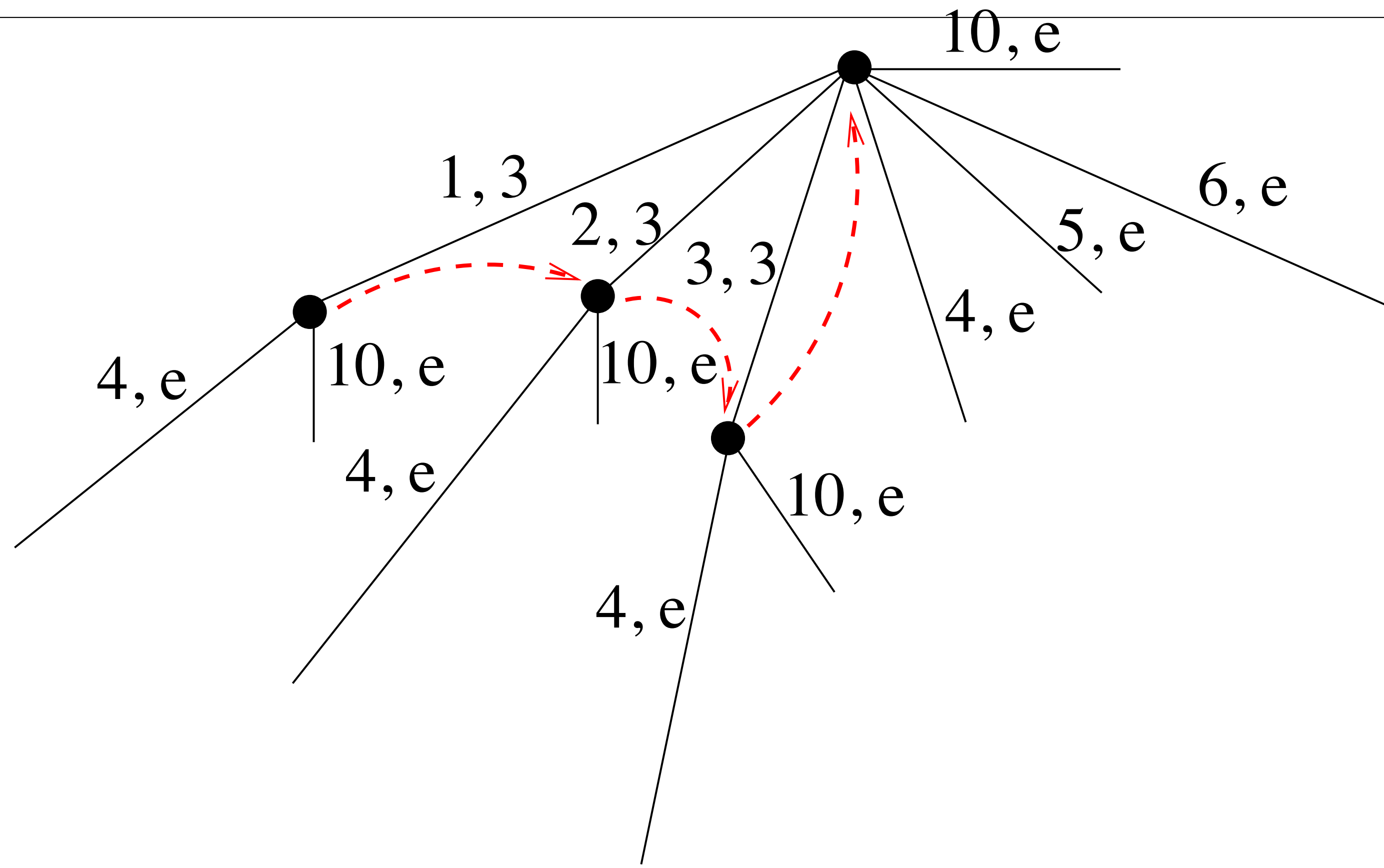
Global end index: xyzabc



Global end index: xyzabc



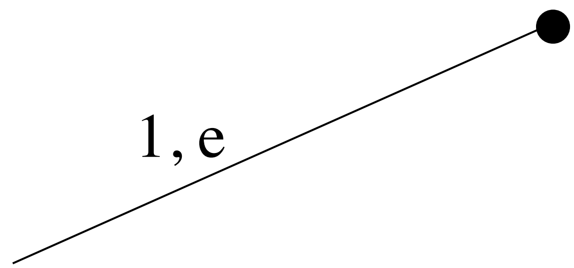
Global end index: xyzabcxyzr



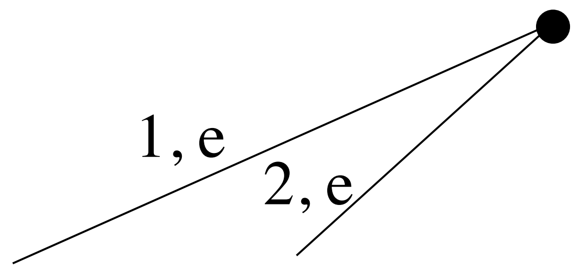
Final “tricks”

- **Two situations as yet unobserved**
- **Early stopping**
  - **The first node where an extension is found is sufficient**
  - **All subsequent suffix links will have a match**
- **Skip count**
  - **When finding suffix link, just match first character**
  - **Rest of characters on arc MUST match**

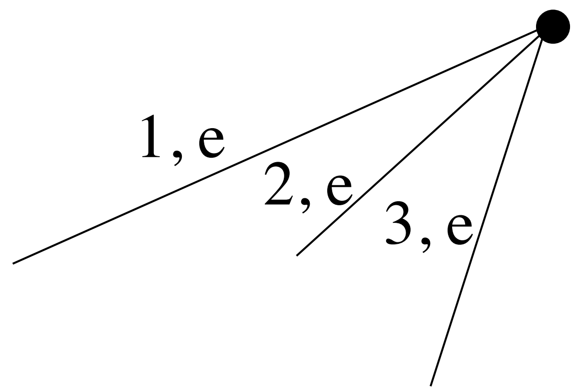
Early stopping and Skip/count: xyztyzrxyzr



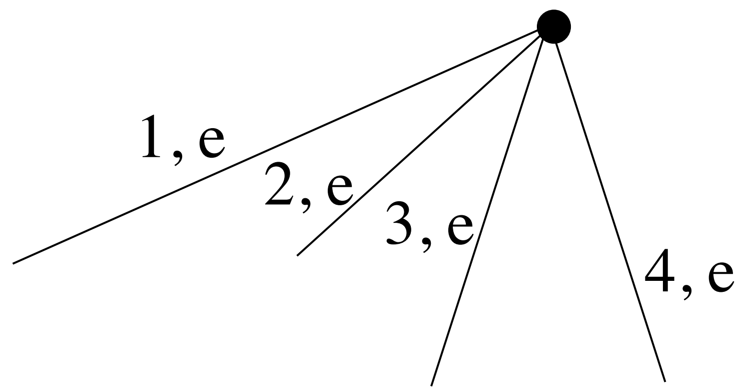
Early stopping and Skip/count: xyztyzrxyzr



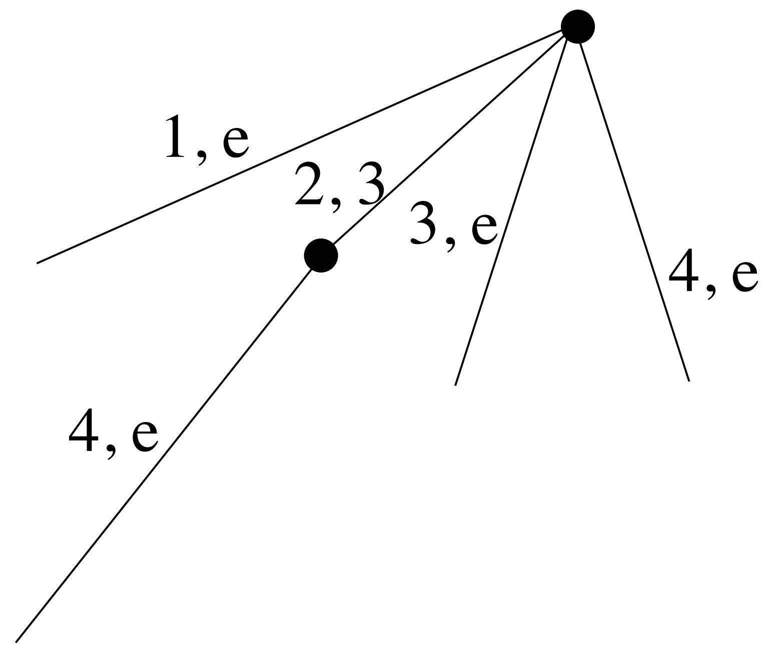
Early stopping and Skip/count: xyztyzrxyzr



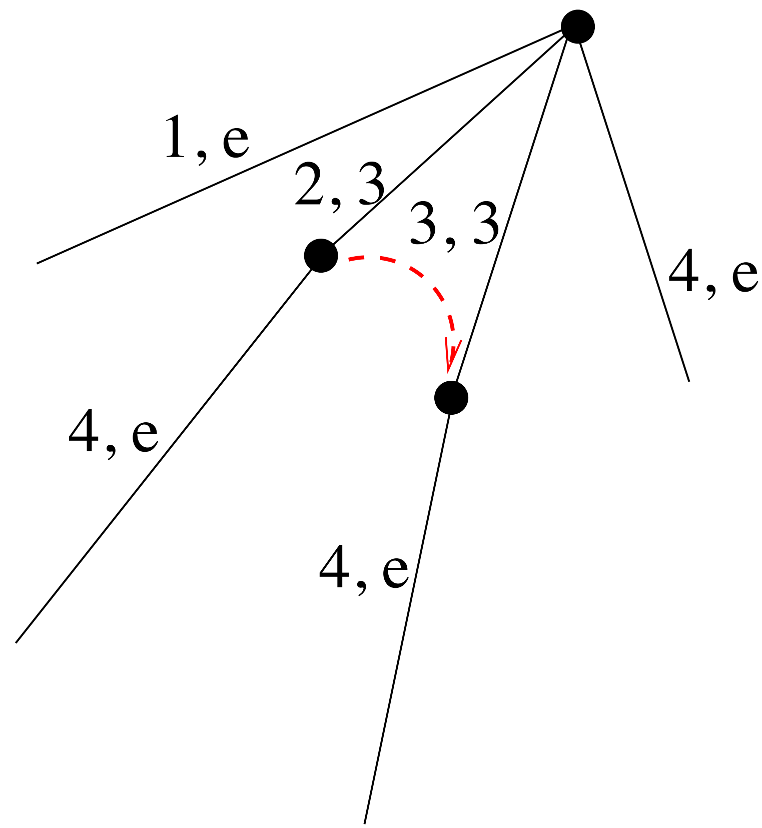
Early stopping and Skip/count: xyztyzrxyzr



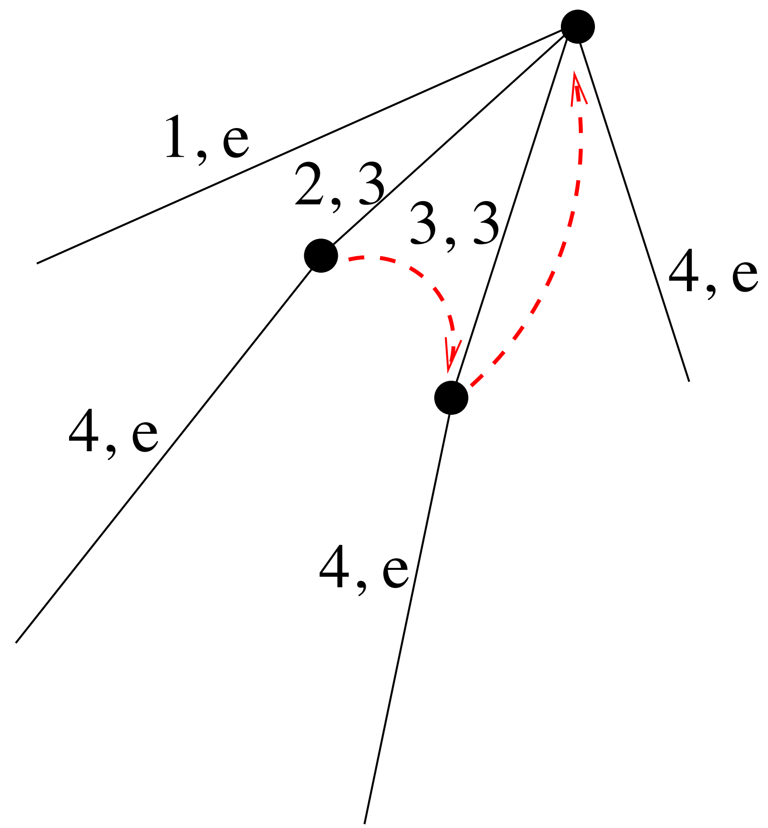
Early stopping and Skip/count: xyztyzrxyzr



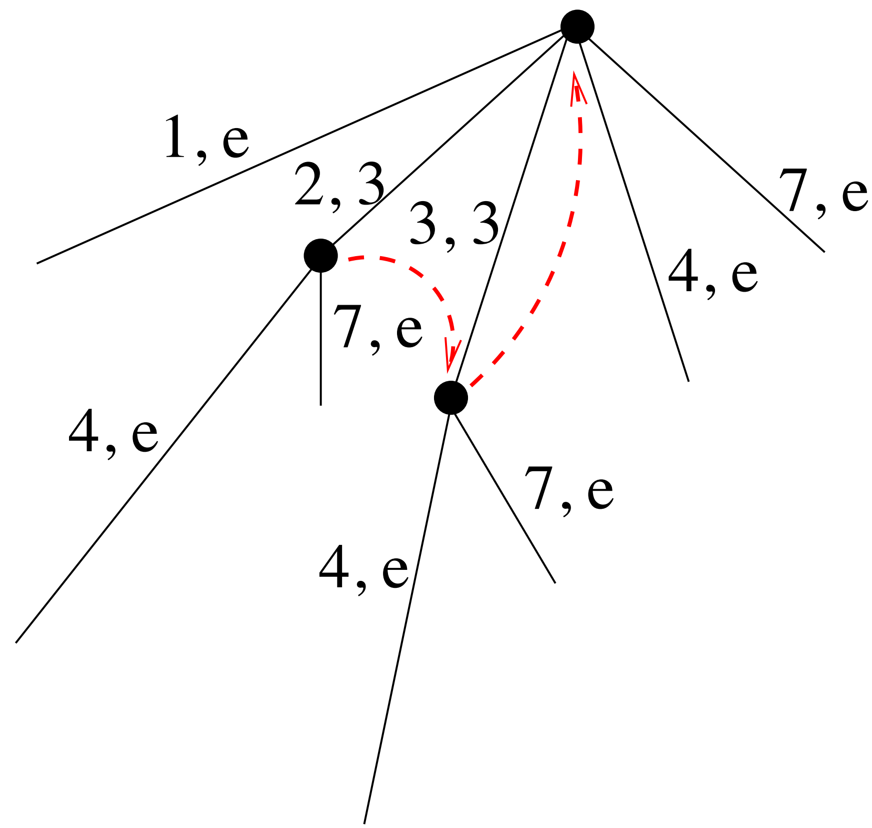
Early stopping and Skip/count: xyztyzrxyzr



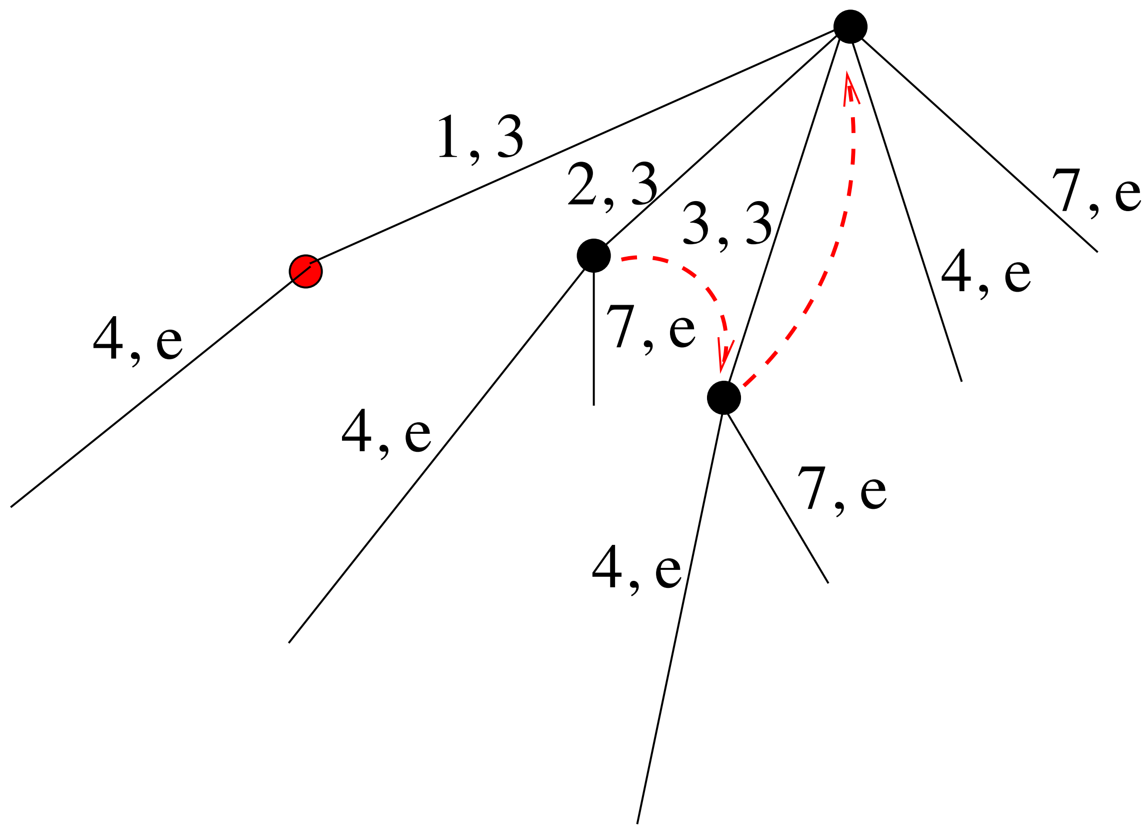
Early stopping and Skip/count: xyztyzrxyzr



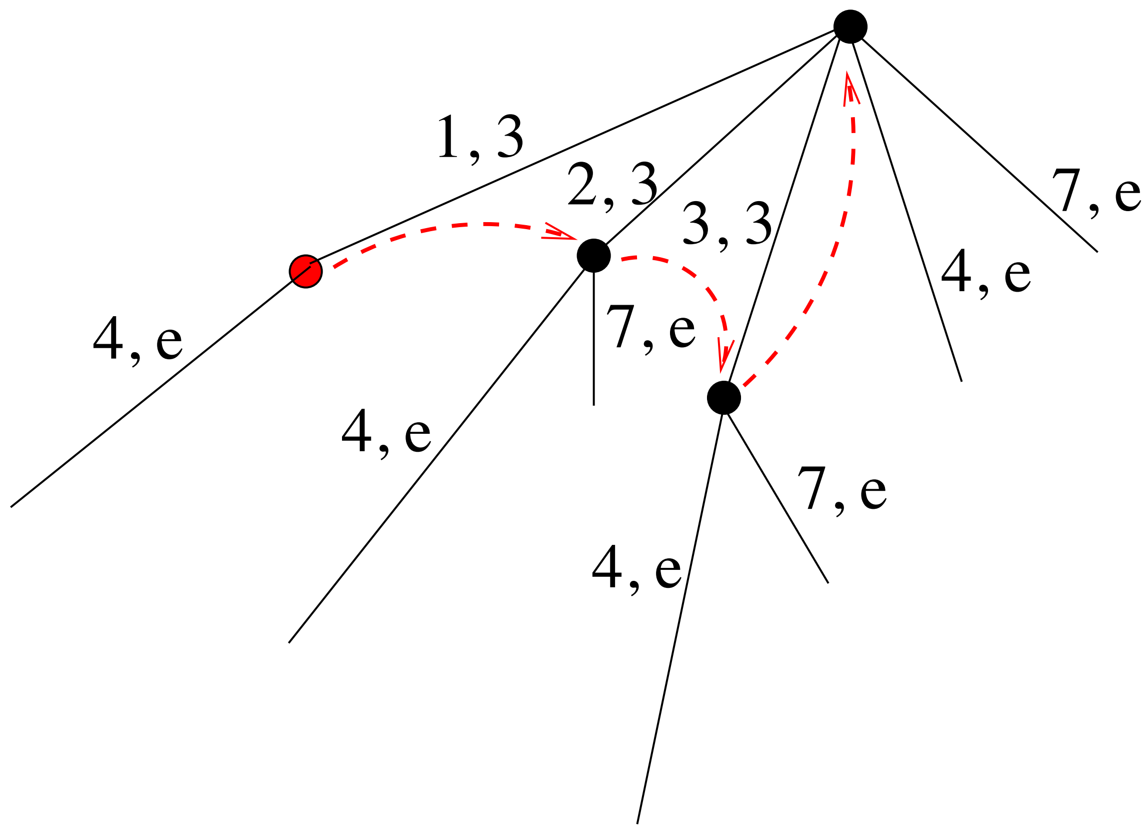
Early stopping and Skip/count: xyztyzrxyzr



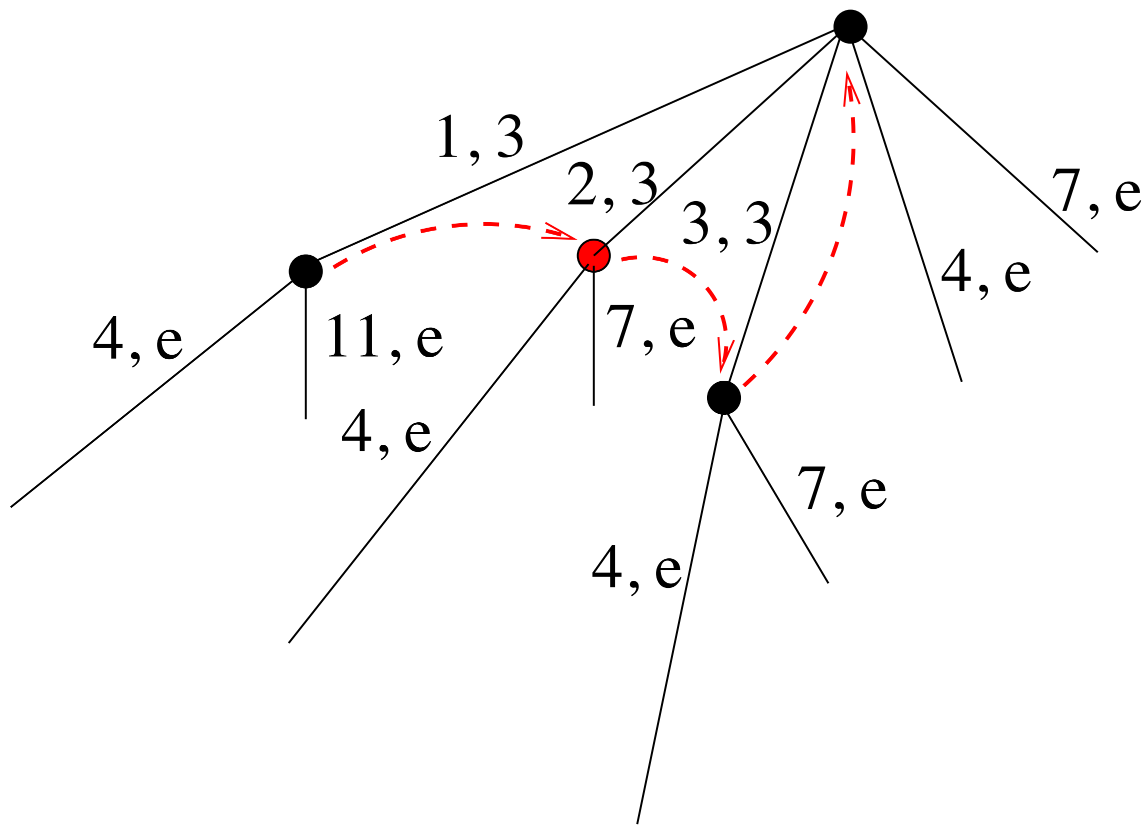
Early stopping and Skip/count: xyztyzrxyzr



Early stopping and Skip/count: xyztyzrxyzr

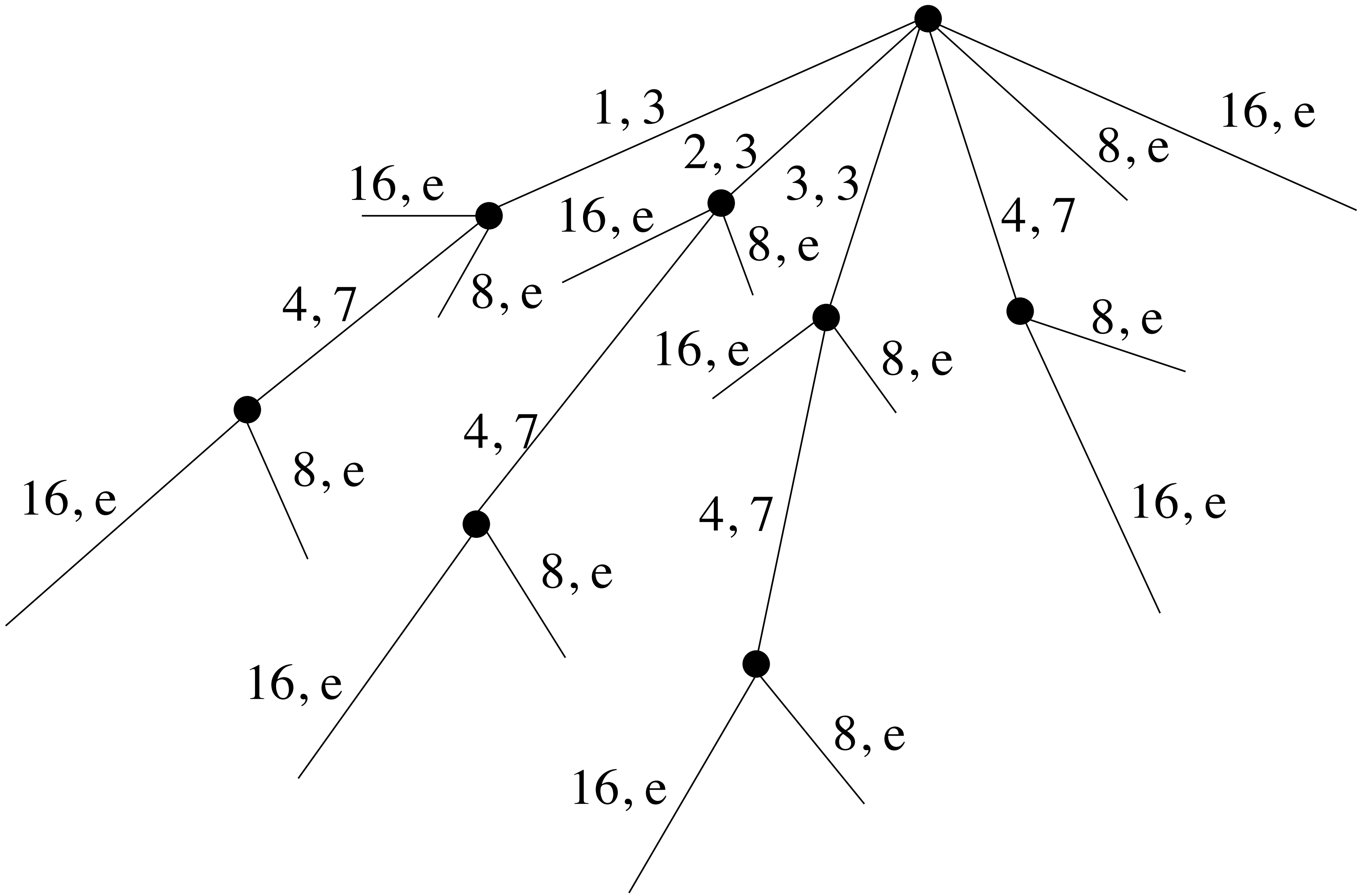


Early stopping and Skip/count: xyztyzrxyzr



Example:

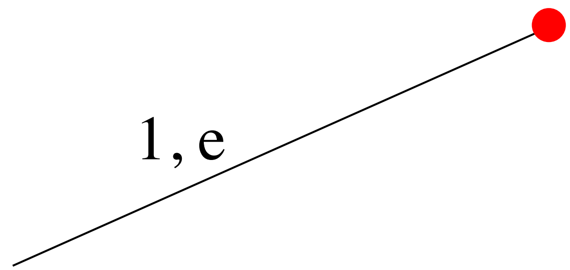
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r



Example:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	x	**t**	**p**	**y**	**x**	**t**	**p**	**z**	**x**	**t**	**p**	**y**	**x**	**t**	**p**	**r**

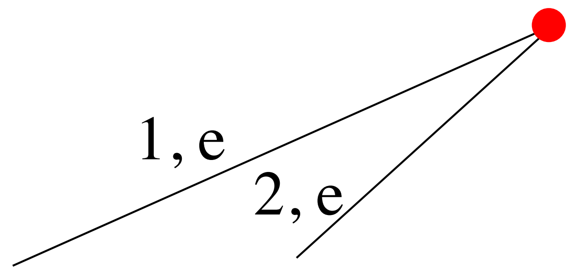
Example:

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r



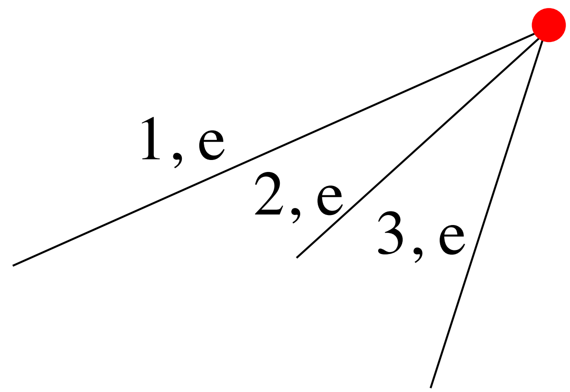
Example:

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r

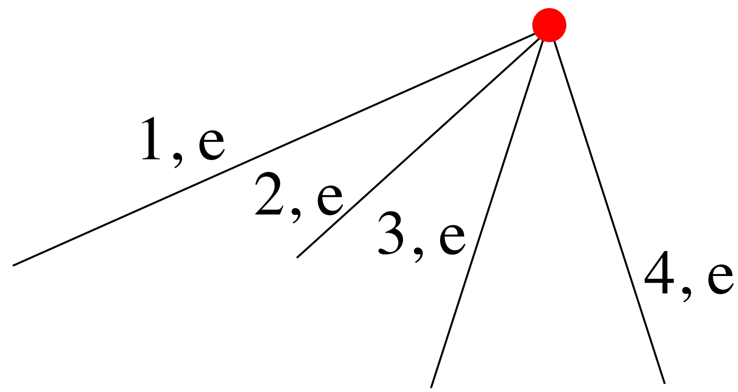


Example:

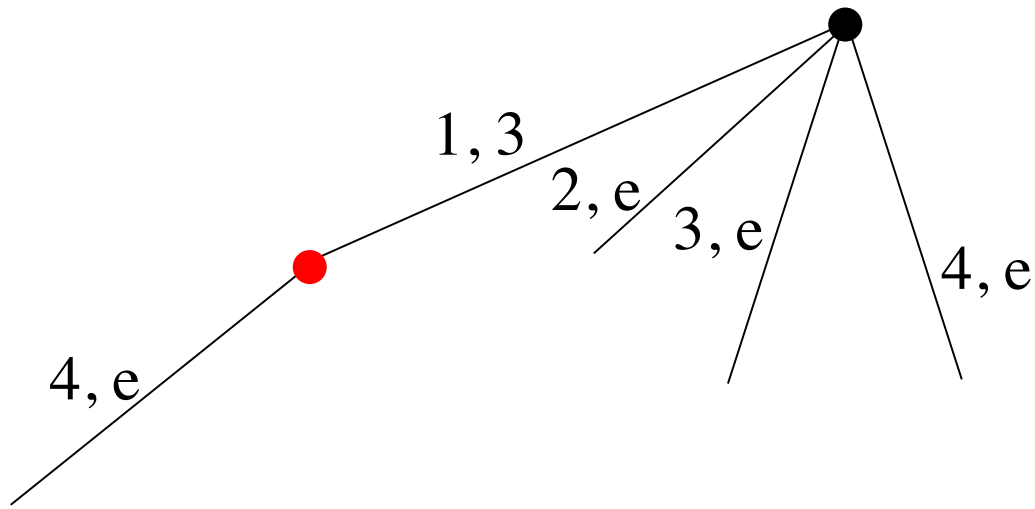
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r



Example:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	x	**t**	**p**	**y**	**x**	**t**	**p**	**z**	**x**	**t**	**p**	**y**	**x**	**t**	**p**	**r**

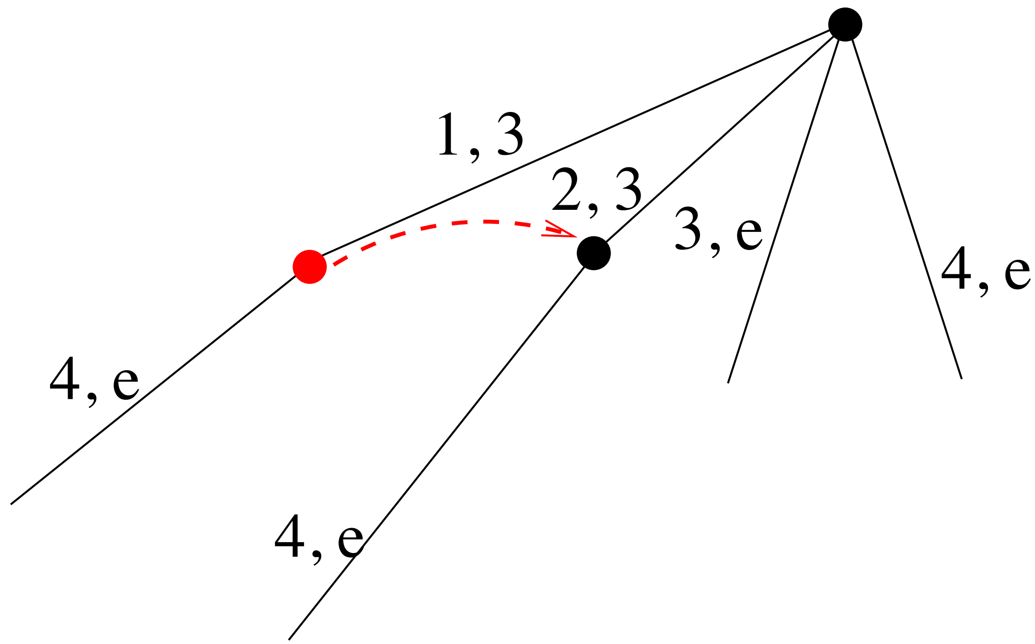


Example:	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
	x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r



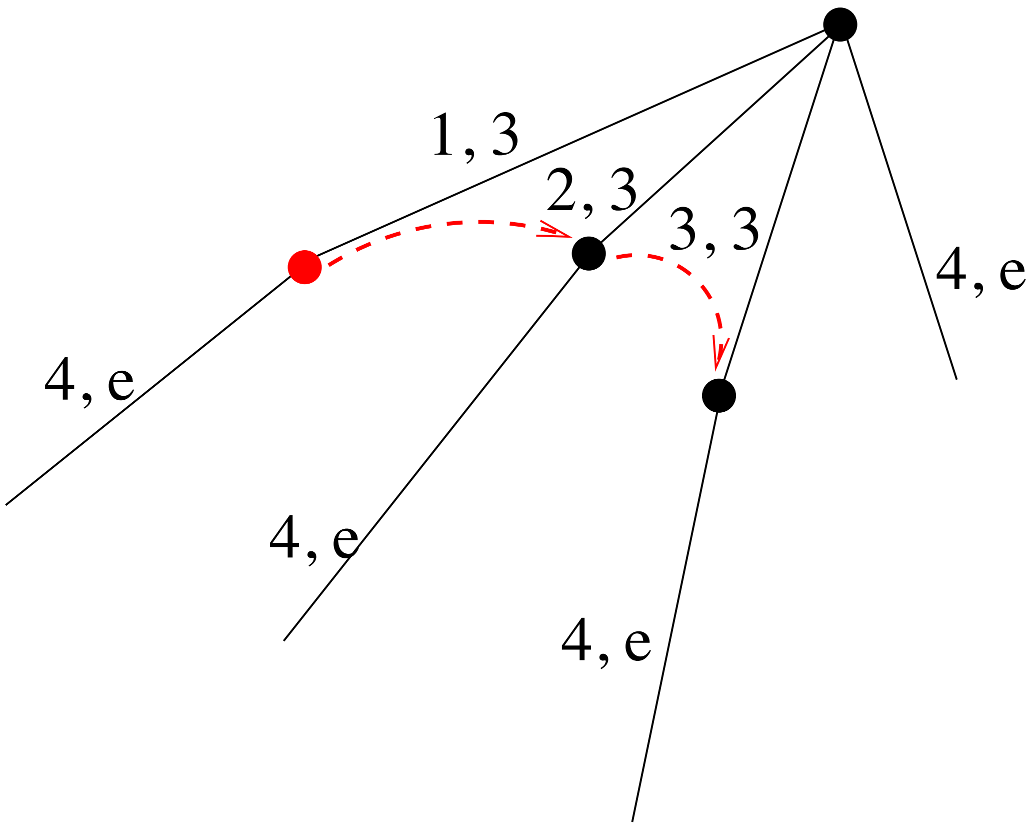
Example:

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r



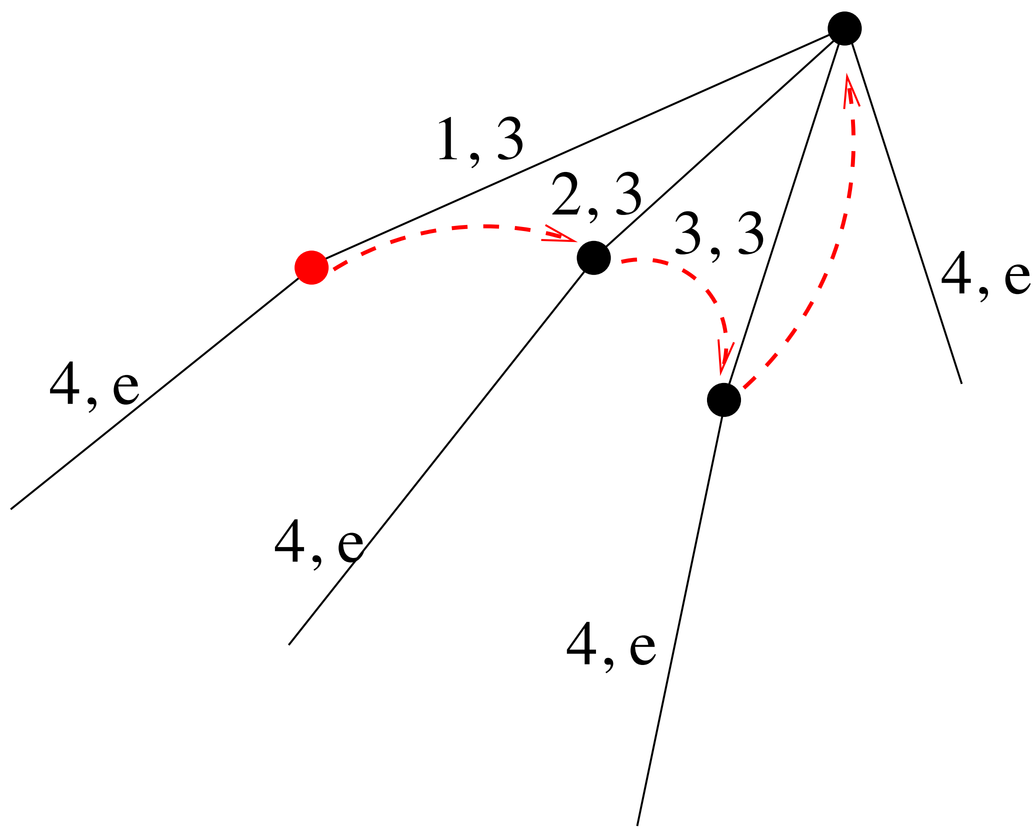
Example:

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r



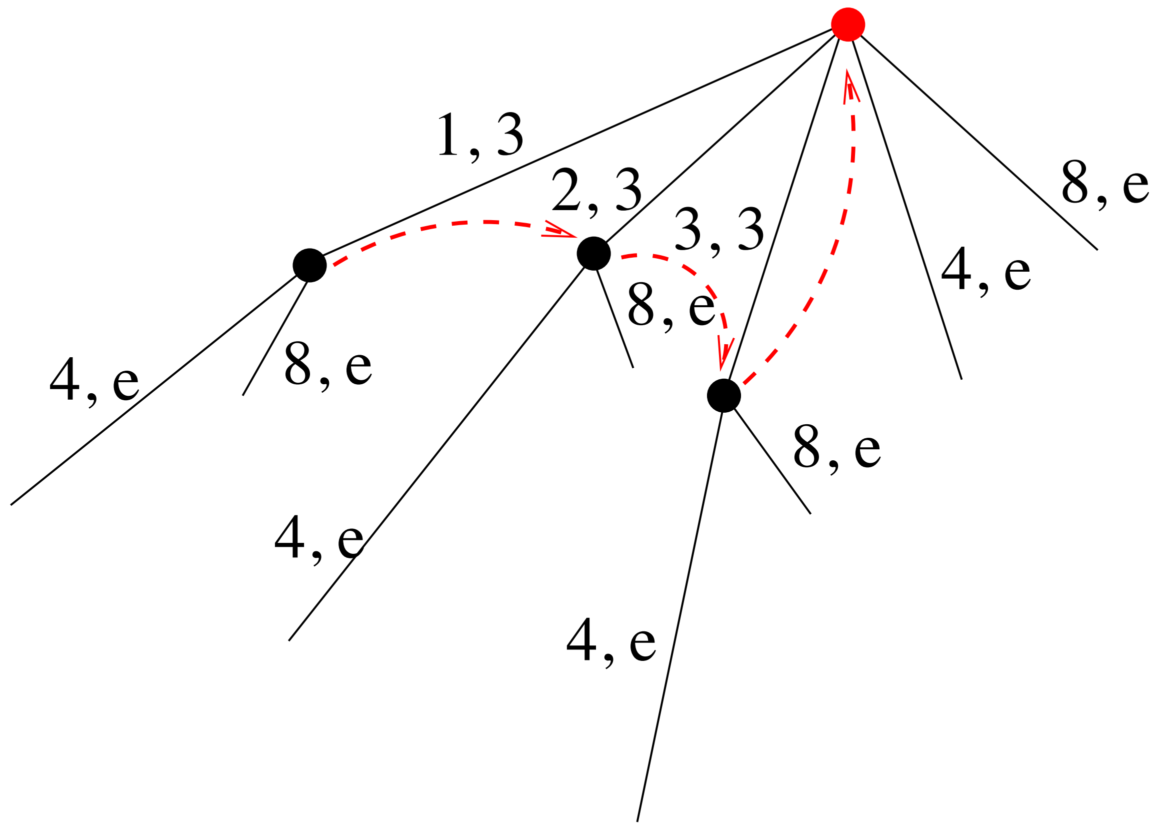
Example:

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
x	**t**	**p**	**y**	**x**	**t**	**p**	**z**	**x**	**t**	**p**	**y**	**x**	**t**	**p**	**r**

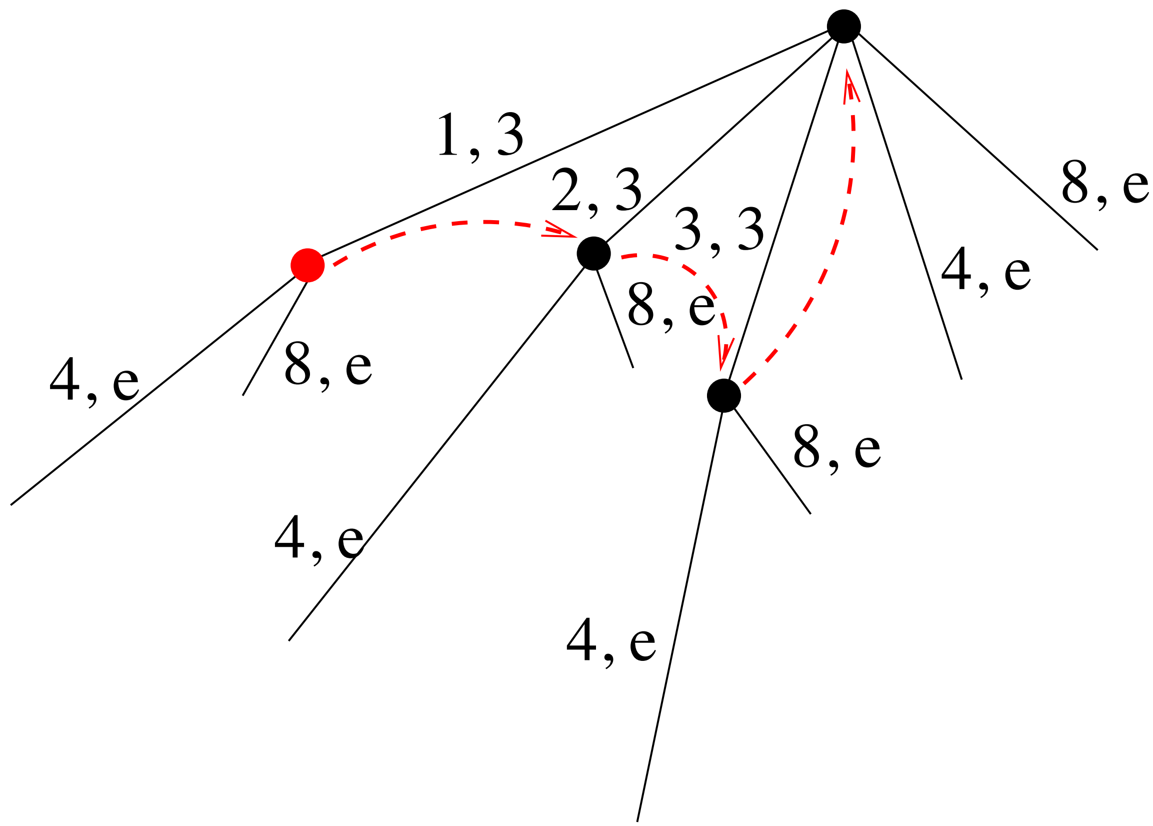


Example:

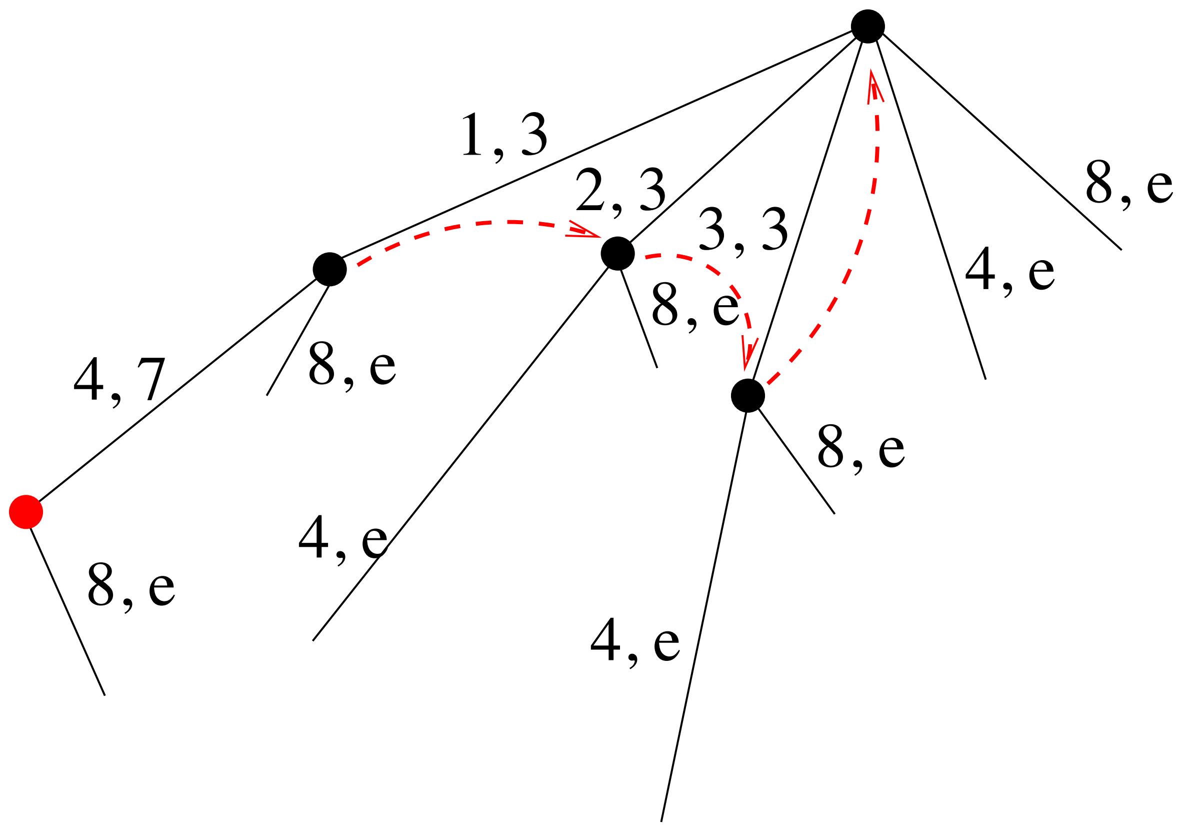
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
x	**t**	**p**	**y**	**x**	**t**	**p**	**z**	**x**	**t**	**p**	**y**	**x**	**t**	**p**	**r**



Example: 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
x t p y x t p z x t p y x t p r

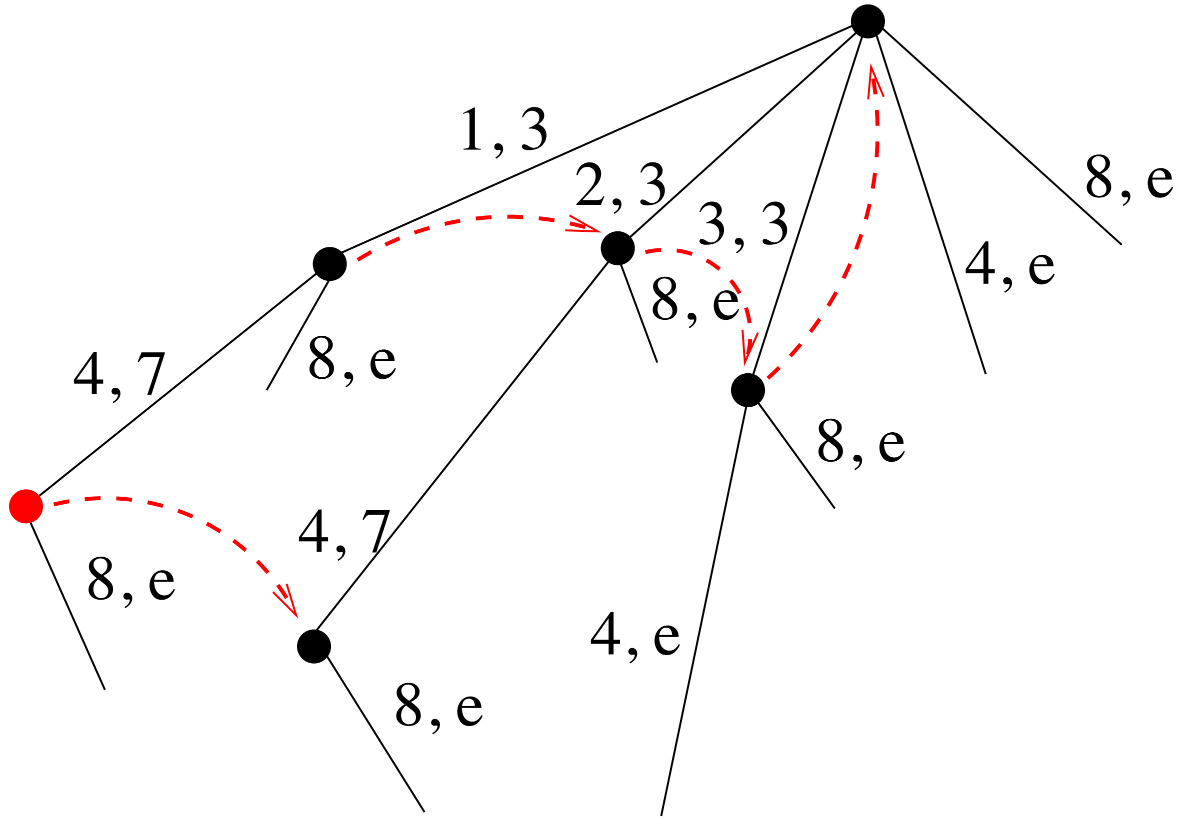


Example: 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
x t p y x t p z x t p y x t p r



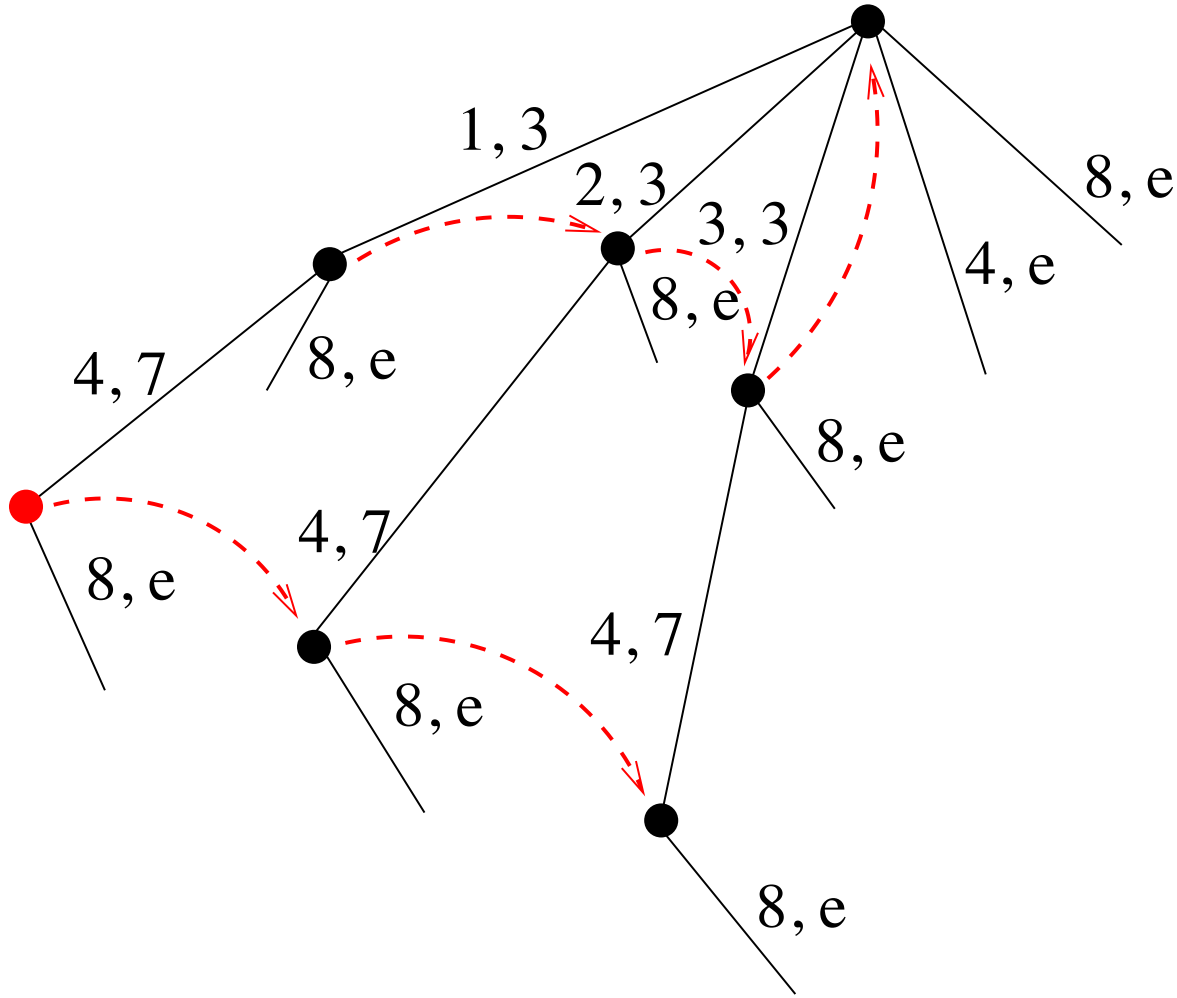
Example:

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r

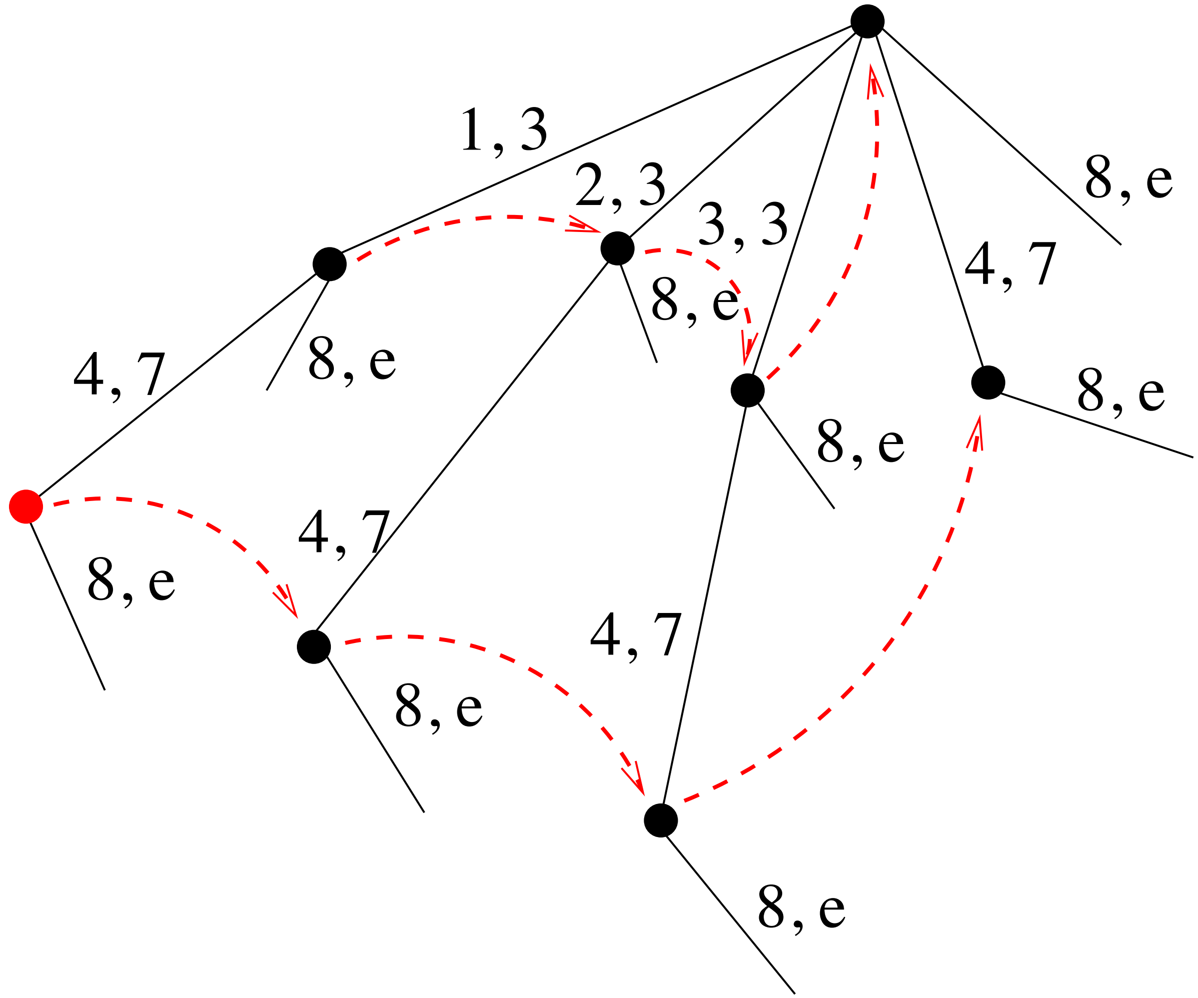


Example:

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r

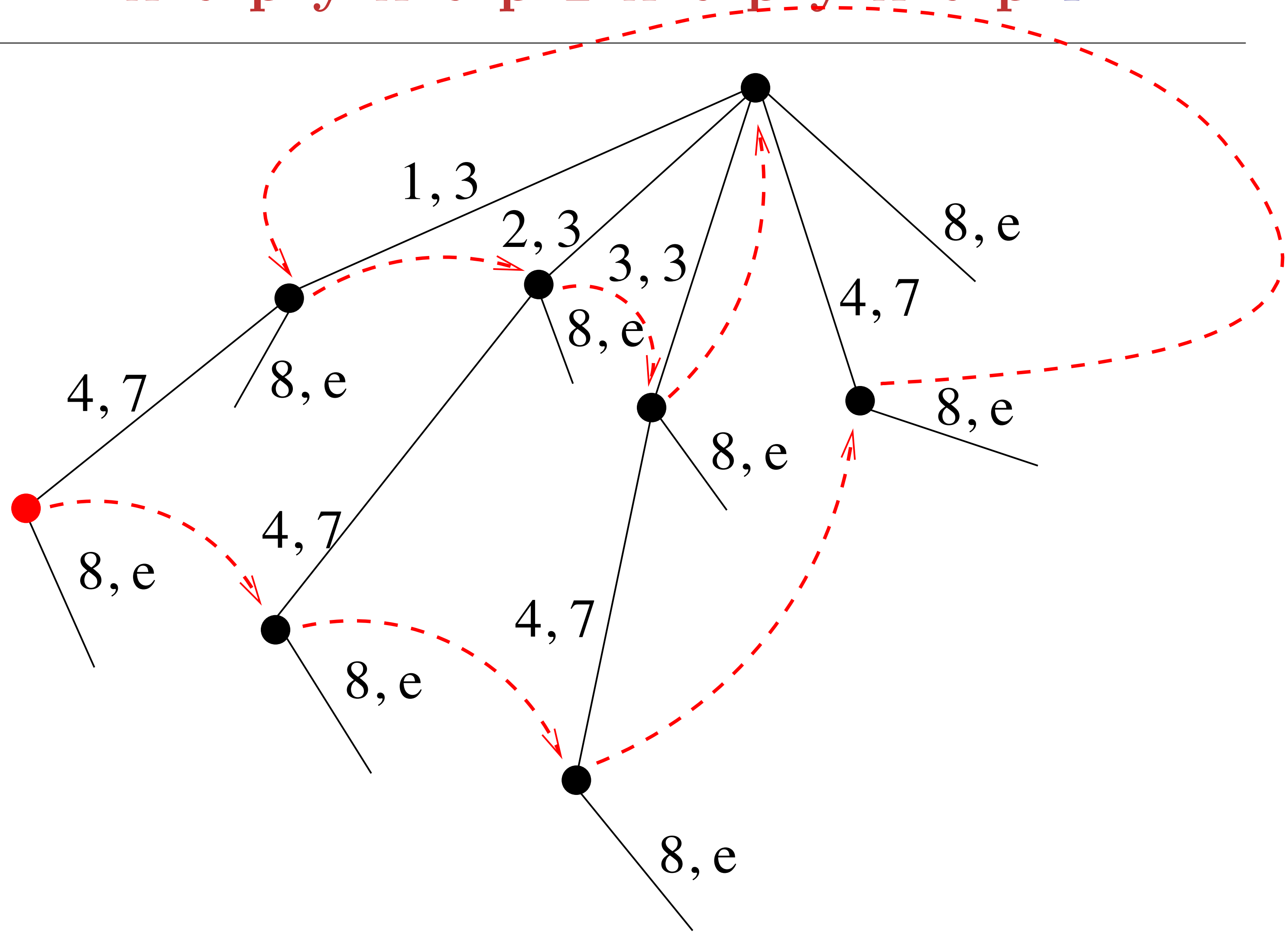


Example:	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
	x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r



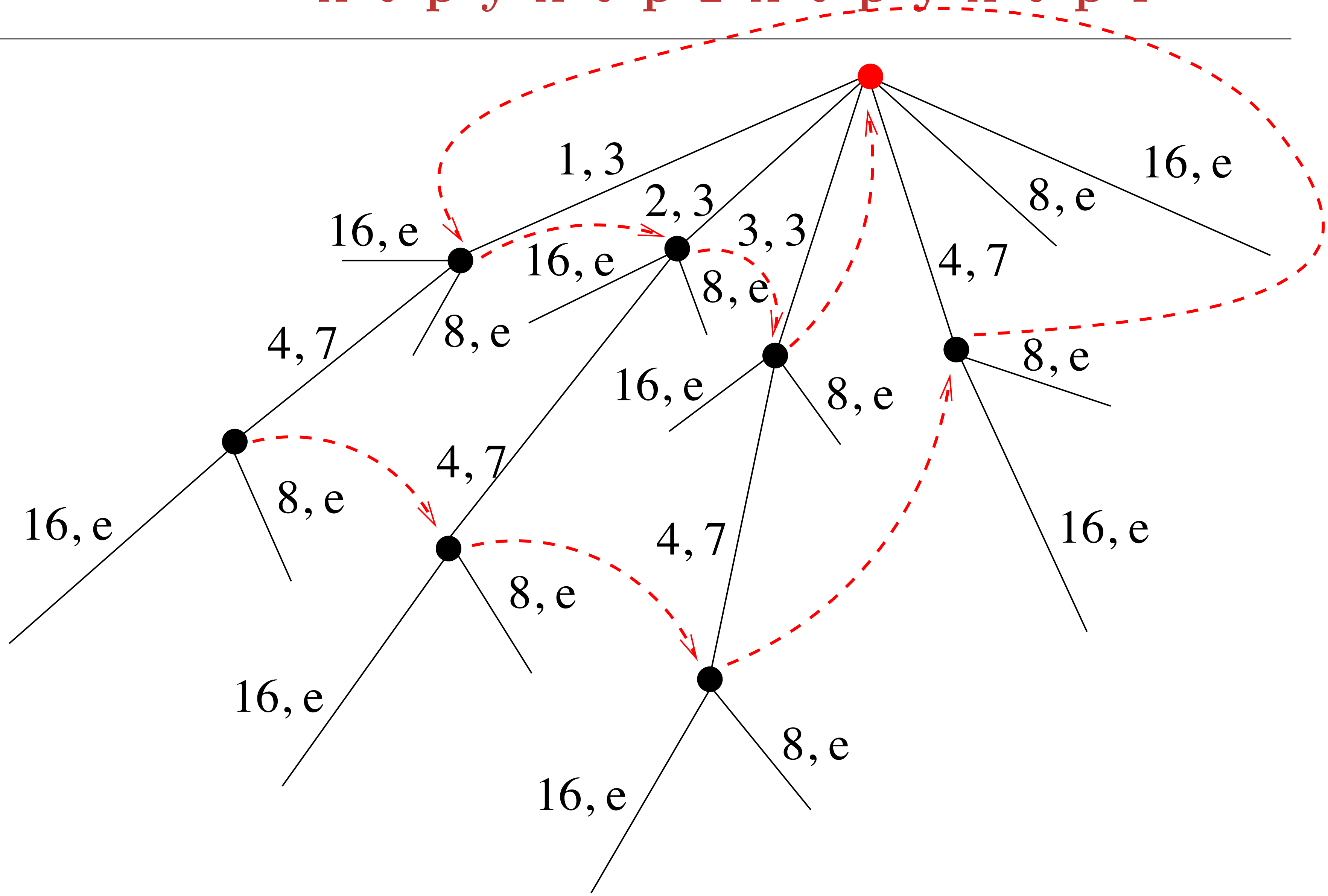
Example:

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
x	**t**	**p**	**y**	**x**	**t**	**p**	**z**	**x**	**t**	**p**	**y**	**x**	**t**	**p**	**r**



Example:

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
x	**t**	**p**	**y**	**x**	**t**	**p**	**z**	**x**	**t**	**p**	**y**	**x**	**t**	**p**	**r**



Searching for occurrences

- **Start at the root**
- **Search for match in tree**
- **Count the number of leaf nodes below point where word matches**
- **Linear in the length of the pattern, not the text**
- **Example: how many times does** xt **occur in text?** t**?** py**?**

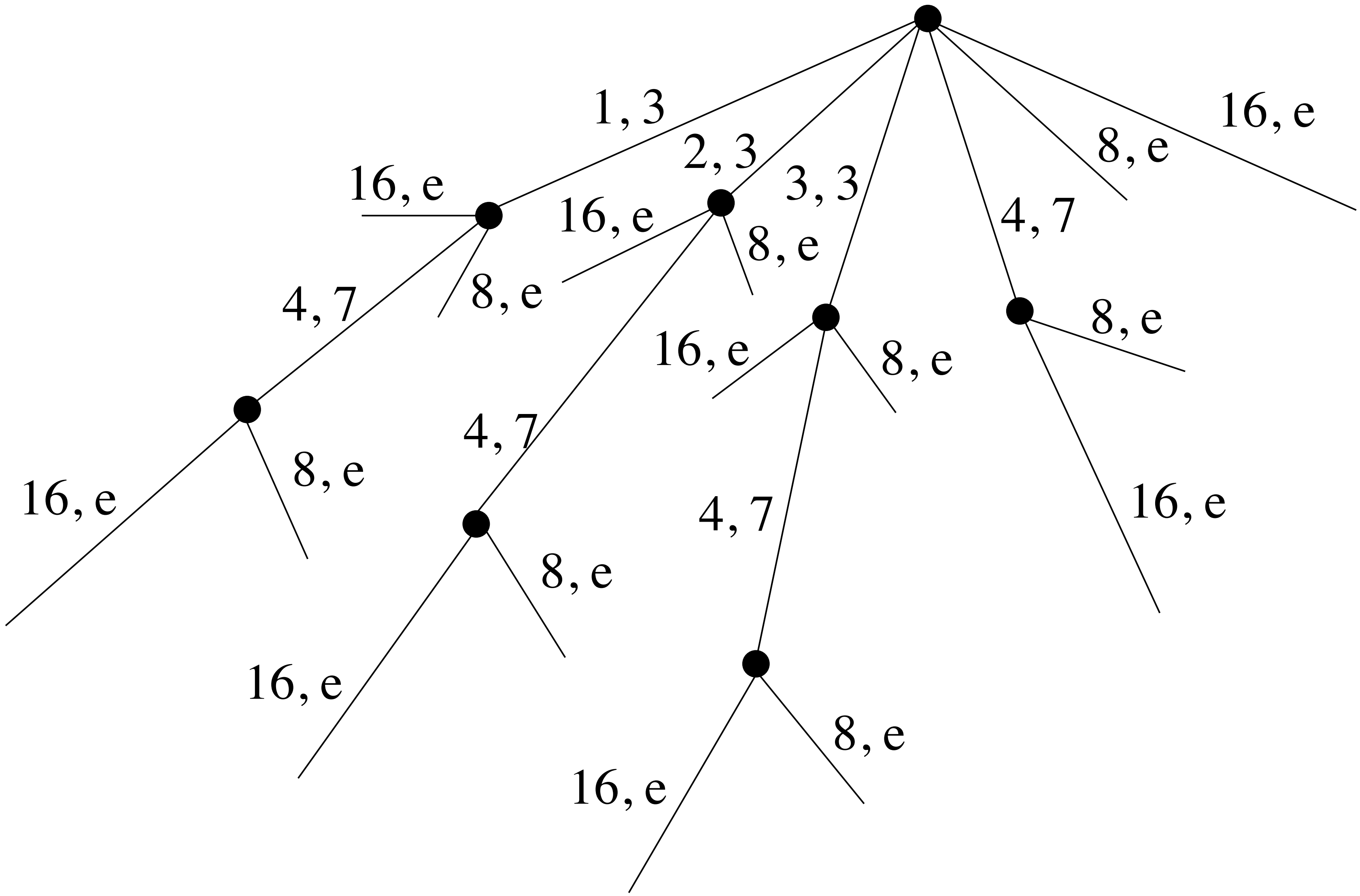
Characteristics of suffix trees

- **One transition out of each state for any given symbol in Σ**
- **Size requirements quite large: $\mathrm{O}(m|\Sigma|)$ for length m**
  - **m states, Σ continuations per state**
  - **Can hash, resulting in less space usage, but $\mathrm{O}(n \log |\Sigma|)$ time to search, rather than $\mathrm{O}(n)$**
- **With linear-time algorithm to build, construction not the problem**
- **Storage and use is the problem**

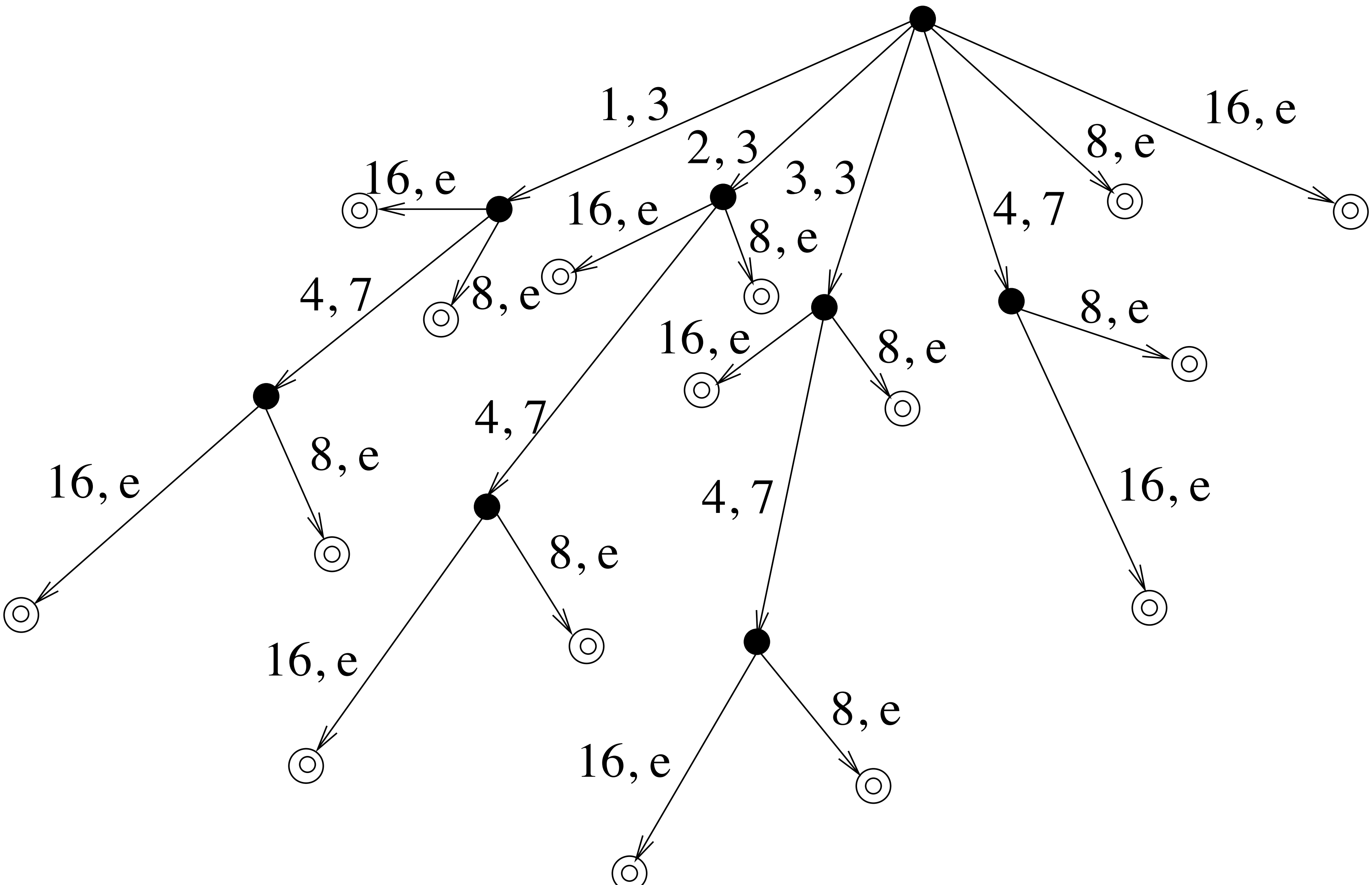
Suffix automata

- **One space saving idea: turn tree into directed acyclic graph (DAG)**
- **Convert tree to finite-state automaton with re-entrancies**
  - **Tree is a special case of FSA: start state at root, final states at leaves**
  - **Further, tree is deterministic FSA**
  - **Minimize the tree to a minimal automaton**
- **Potentially massive space savings on final graph**

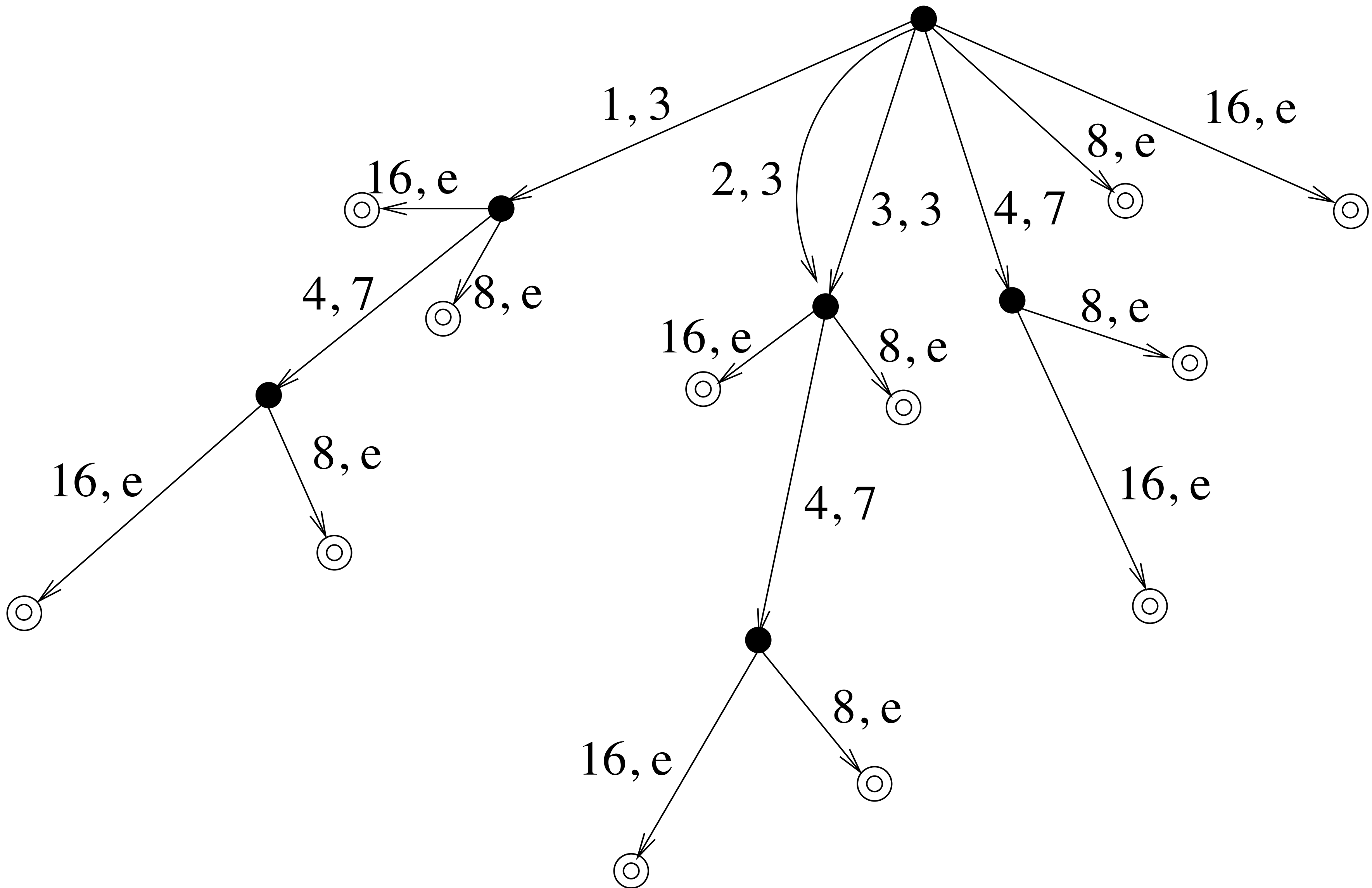
Example suffix tree



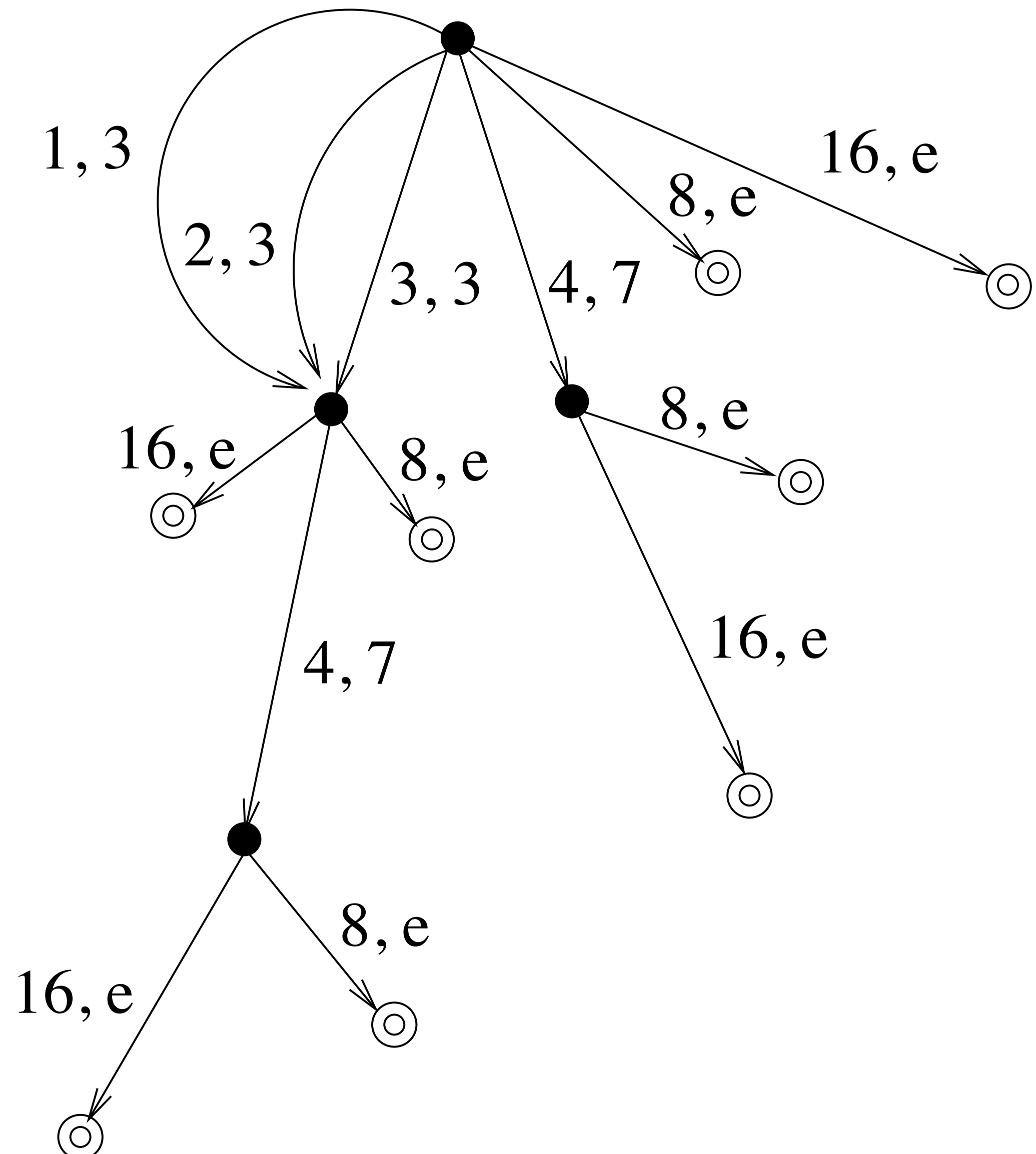
Change to suffix automata: directed edges, final states



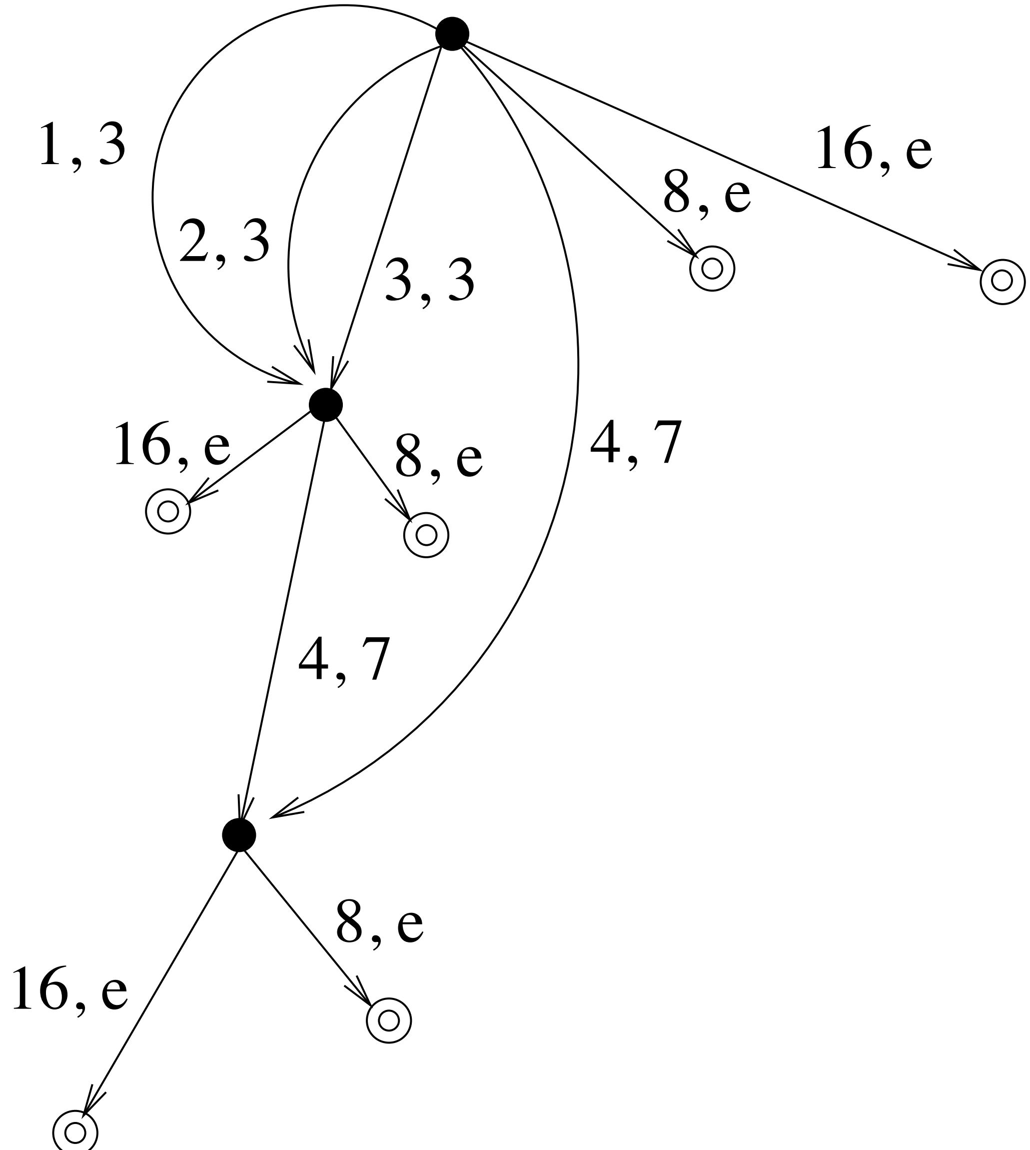
Change to suffix automata: collapse states



Change to suffix automata: collapse states



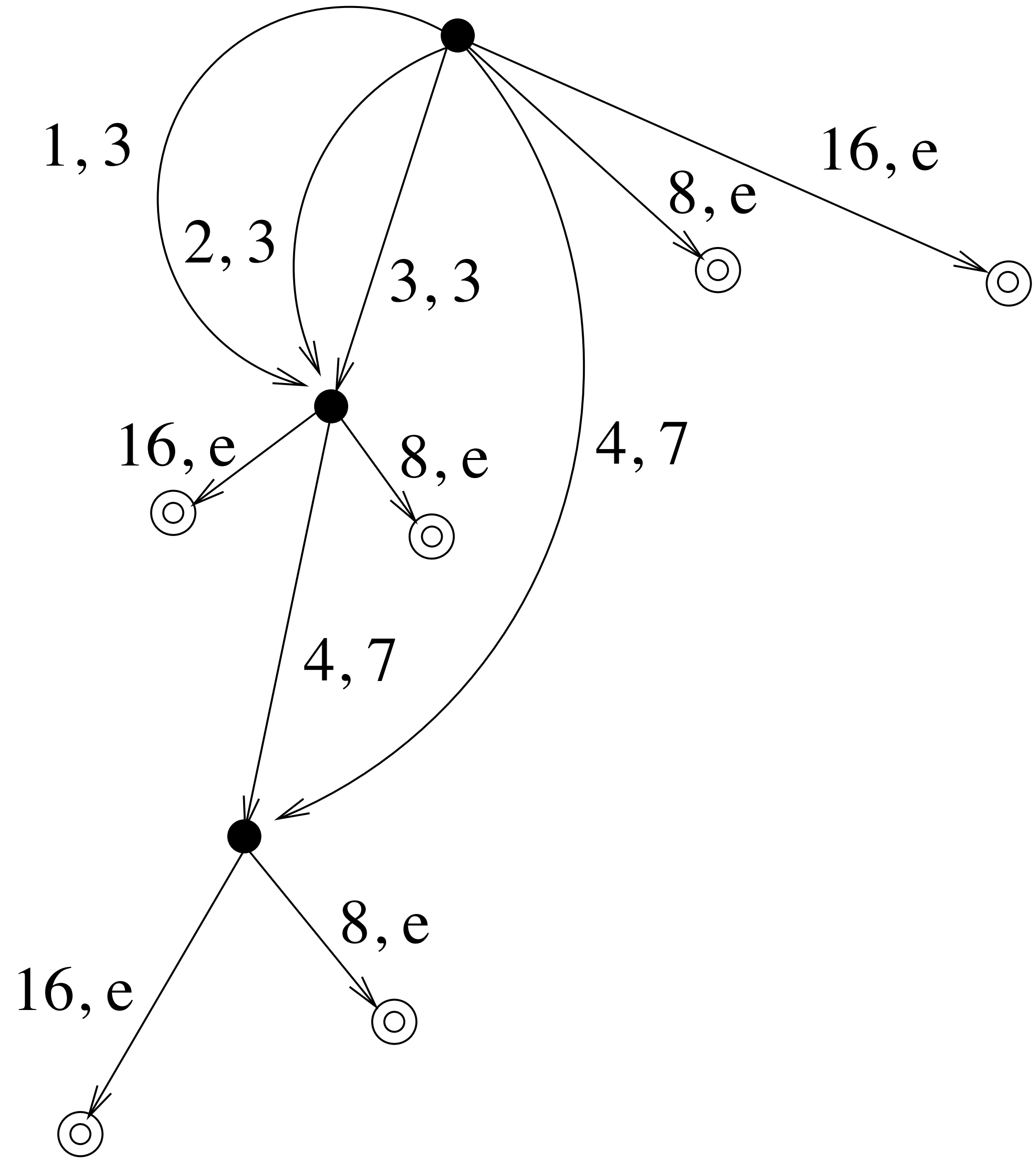
Change to suffix automata: collapse states



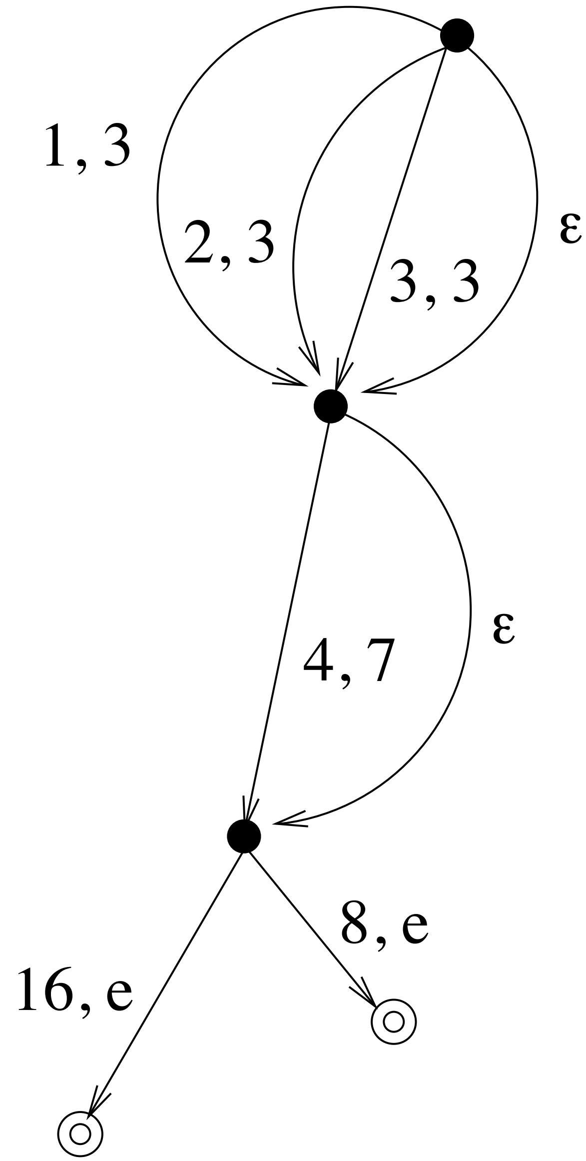
Deterministic suffix automata

- **Large space savings. In this case:**
  - **from 16 final states, 8 internal states, 23 arcs**
  - **to 6 final states, 3 internal states, 11 arcs**
- **Can be achieved with well-known graph minimization algorithms**
- **One particular problem in exact match:**
  - **For a given final state, may have reached there via various paths**
  - **Hence extra work to identify indices of matches**
- **Can get additional space savings with ϵ transitions**

Suffix automata with no ϵ transitions



Suffix automata with ϵ transitions



Suffix array

- **Another clever space savings idea is the suffix array**
- **Note that, for text of length m, there are exactly m suffixes**
- **Thus, in an array of m ints, we can store the suffix indices in alphabetical order**
- **For any given pattern, all of the suffixes beginning with the pattern would be adjacent in the array**
- **Much better space usage, not bad exact match time**

Suffix array:

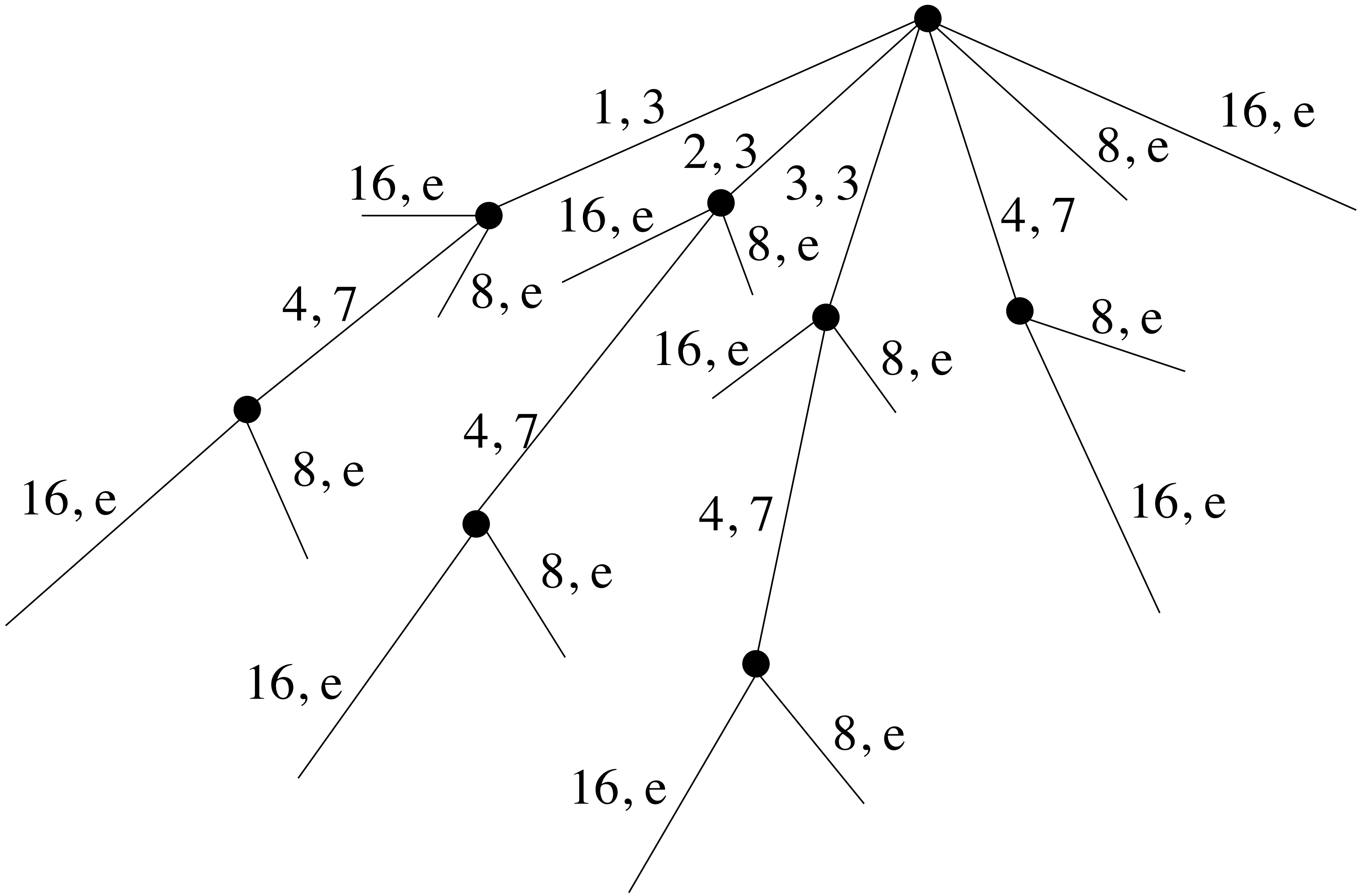
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
x	t	p	y	x	t	p	z	x	t	p	y	x	t	p	r

Suffix idx	Suffix	String pos
1	pr	15
2	pyxtpr	11
3	pyxtpzxtpyxtpr	3
4	pzxtpyxtpr	7
5	r	16
6	tpr	14
7	tpyxtpr	10
8	tpyxtpzxtpyxtpr	2
9	tpzxtpyxtpr	6
10	xtpr	13
11	xtpyxtpr	9
12	xtpyxtpzxtpyxtpr	1
13	xtpzxtpyxtpr	5
14	yxtpr	12
15	yxtpzxtpyxtpr	4
16	zxtpyxtpr	8

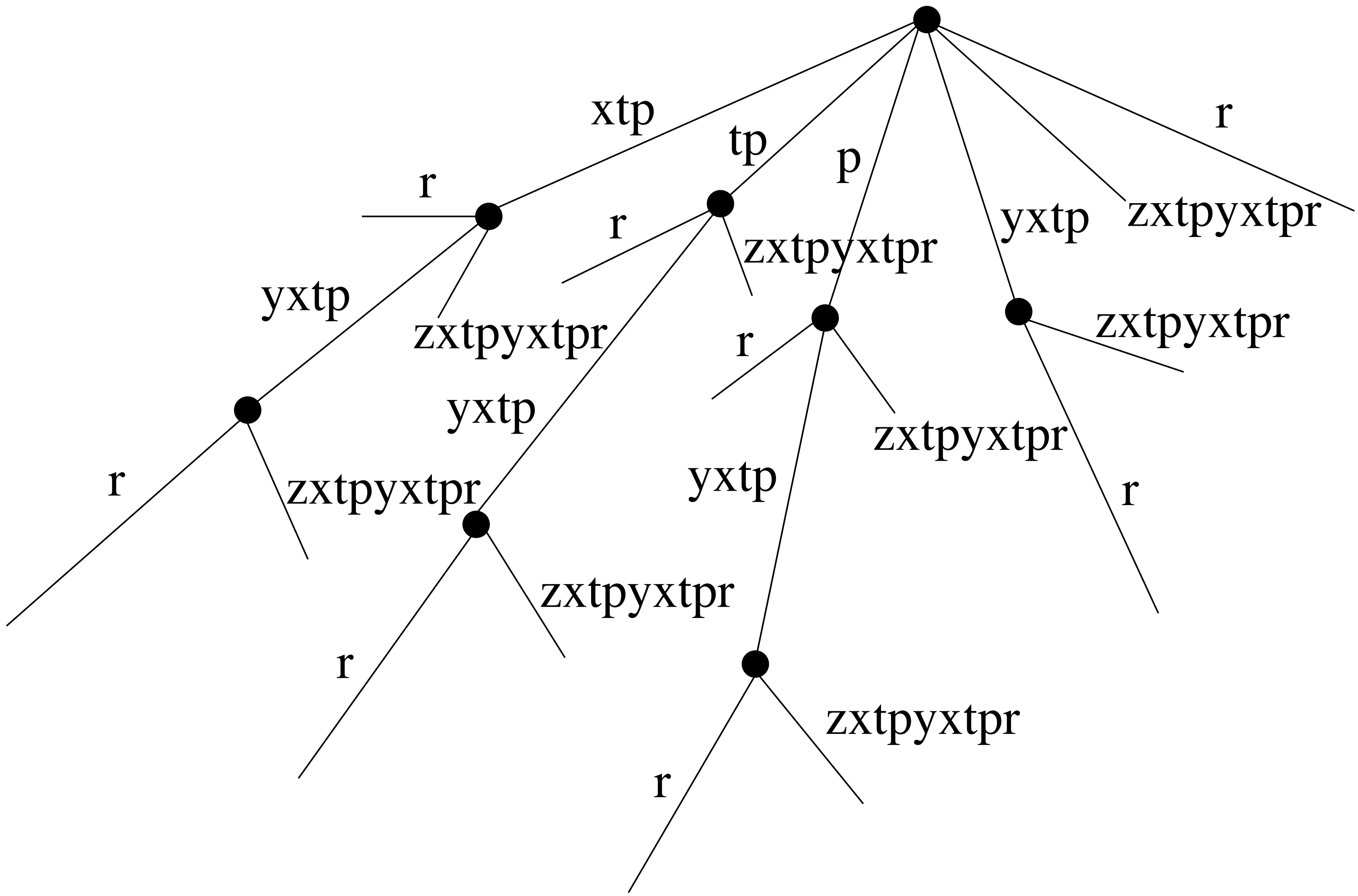
Building suffix array

- **Each state in suffix tree has at most one arc for each letter**
- **If we sort the arcs leaving each node in alphabetical order**
  - **Follow a pre-order (depth first) traversal of the tree**
  - **Emit indices of leaf nodes as encountered**

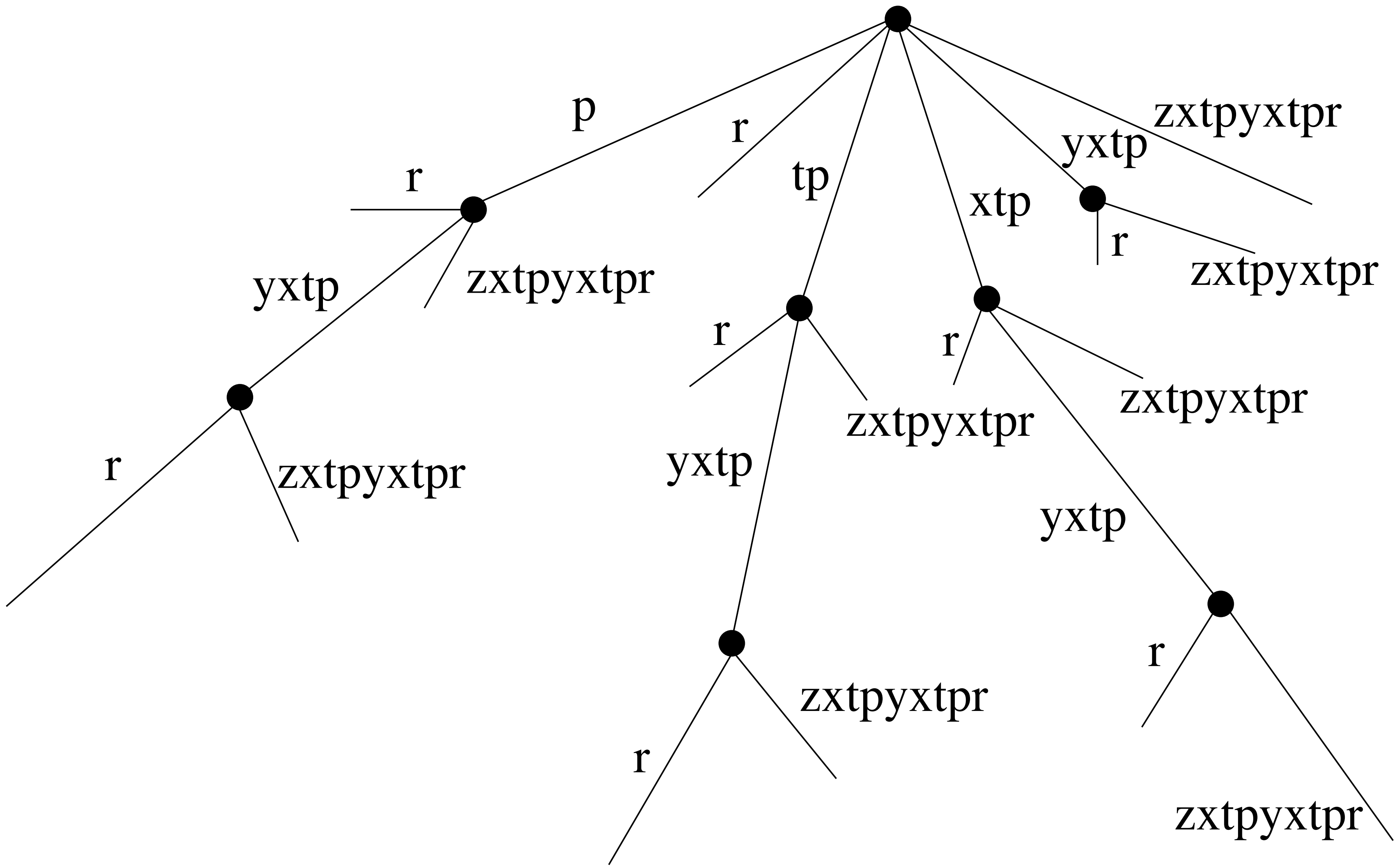
Example suffix tree



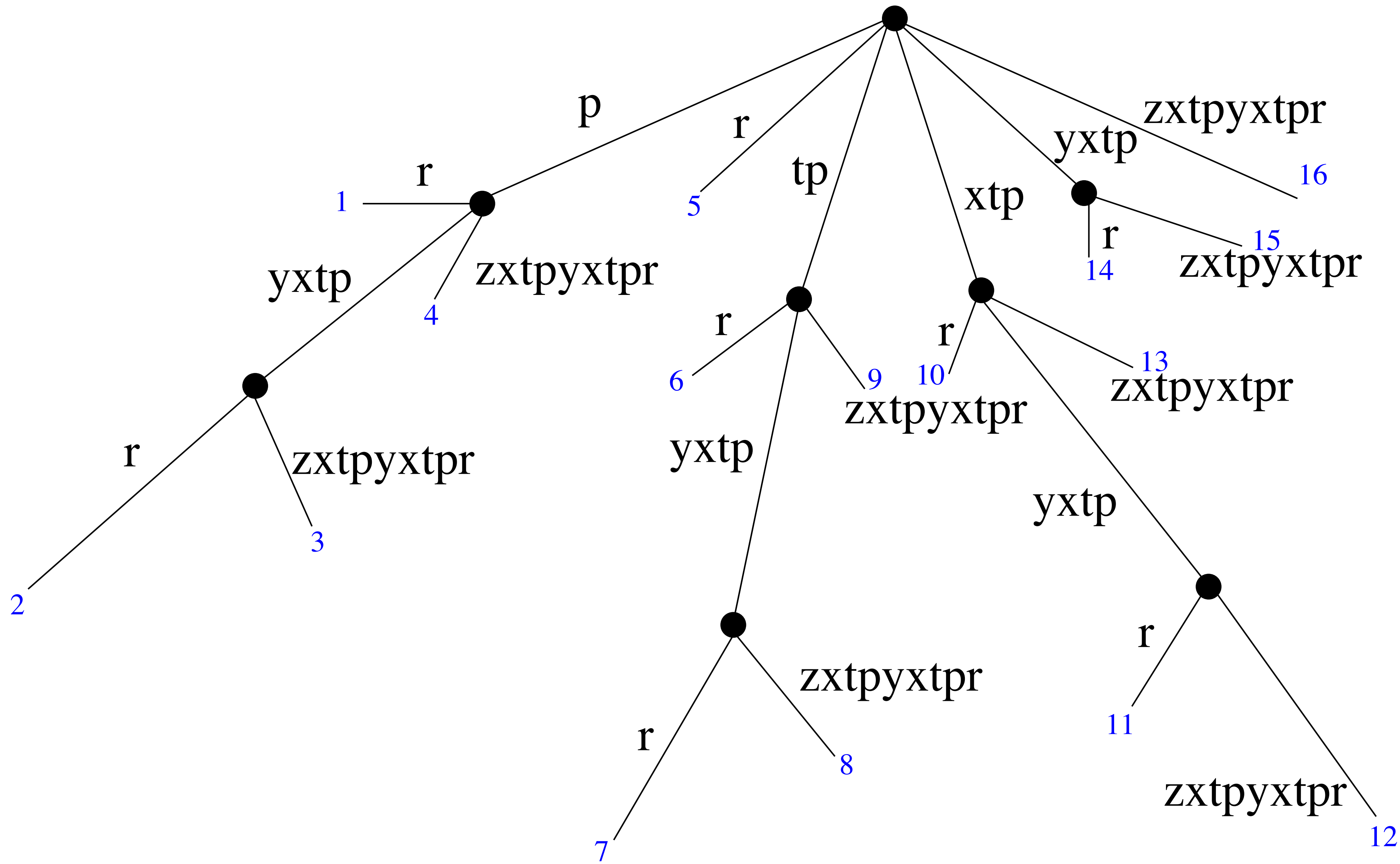
Suffix tree, letters again



Suffix tree, node order



Suffix tree, pre-order enumeration



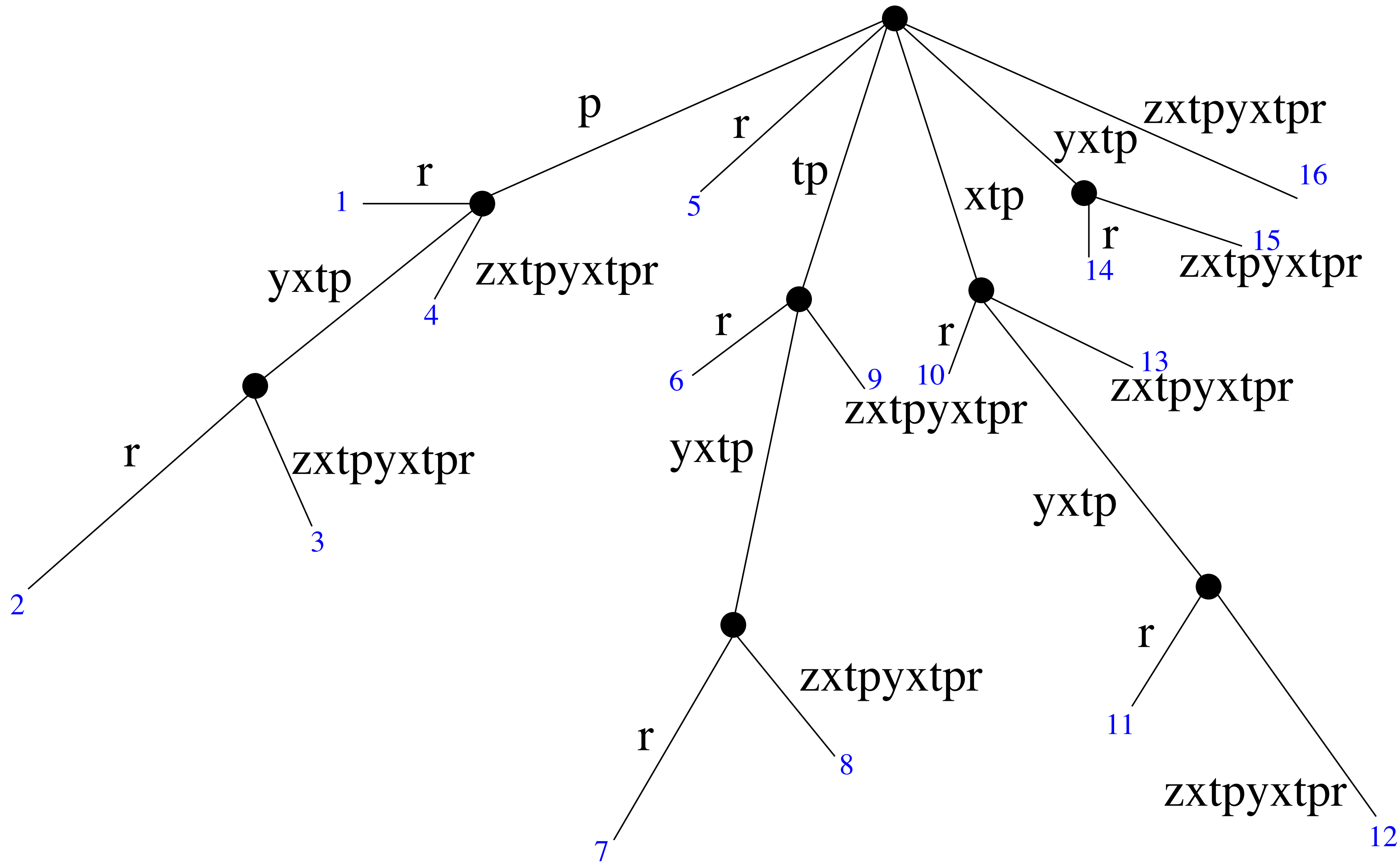
Searching with suffix array

- Perform binary search on the suffix array: $\mathrm{O}(n \log m)$
- Can do better than this with clever preprocessing
  - Want to avoid comparisons that are not necessary
- Suppose we have one match in the suffix array
  - The neighbors may or may not be matches
  - How many characters do we need to check?
  - If we know the *longest common prefix* (LCP) of the neighbors, we know where to begin
- Can build a binary search tree with pre-compiled LCP values at each node

Longest common prefix

- Have enough information to calculate pair-wise LCP in suffix tree
- In pre-order traversal of suffix tree to build the suffix array:
  - The LCP between suffix idx i and suffix idx $i+1$ is the depth in the tree of the lowest common ancestor node of the suffixes
  - (Recall that the depth of a node is the length of the string labeling arcs reaching that node)
- Critical lemma for easy LCP accumulation in binary tree:
  For any i, j such that $j > i+1$,
  $$\mathrm{LCP}(i, j) = \min_{i \le k \le j-1} \mathrm{LCP}(k, k+1)$$
- In binary tree, take the smallest value from the two children

Suffix tree, pre-order enumeration



Suffix array:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	x	**t**	**p**	**y**	**x**	**t**	**p**	**z**	**x**	**t**	**p**	**y**	**x**	**t**	**p**	**r**

Suffix idx	Suffix	String pos	LCP(i, i+1)
1	pr	15	1
2	pyxtpr	11	5
3	pyxtpzxtpyxtpr	3	1
4	pzxtpyxtpr	7	0
5	r	16	0
6	tpr	14	2
7	tpyxtpr	10	6
8	tpyxtpzxtpyxtpr	2	2
9	tpzxtpyxtpr	6	0
10	xtpr	13	3
11	xtpyxtpr	9	7
12	xtpyxtpzxtpyxtpr	1	3
13	xtpzxtpyxtpr	5	0
14	yxtpr	12	4
15	yxtpzxtpyxtpr	4	0
16	zxtpyxtpr	8	