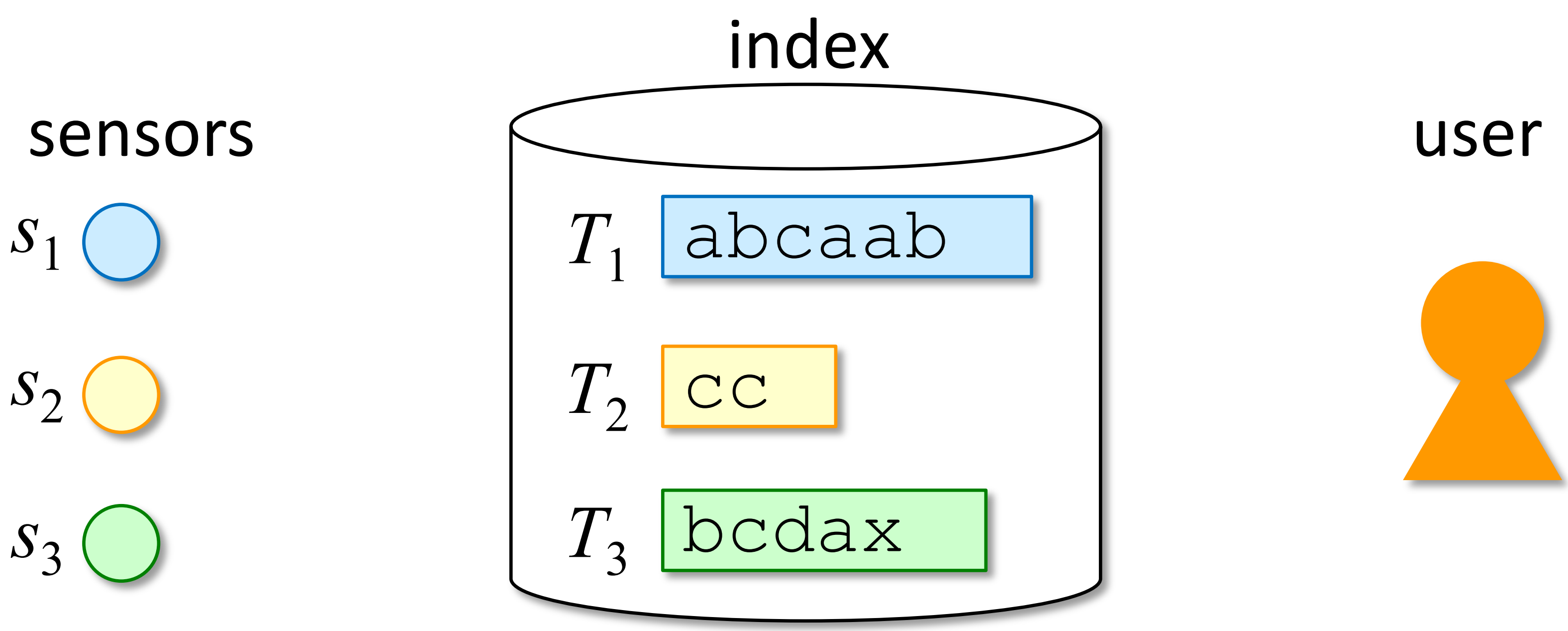
PSC 2020

Pointer-Machine Algorithms for Fully-Online Construction of Suffix Trees and DAWGs on Multiple Strings

Shunsuke Inenaga
Kyushu University, Japan

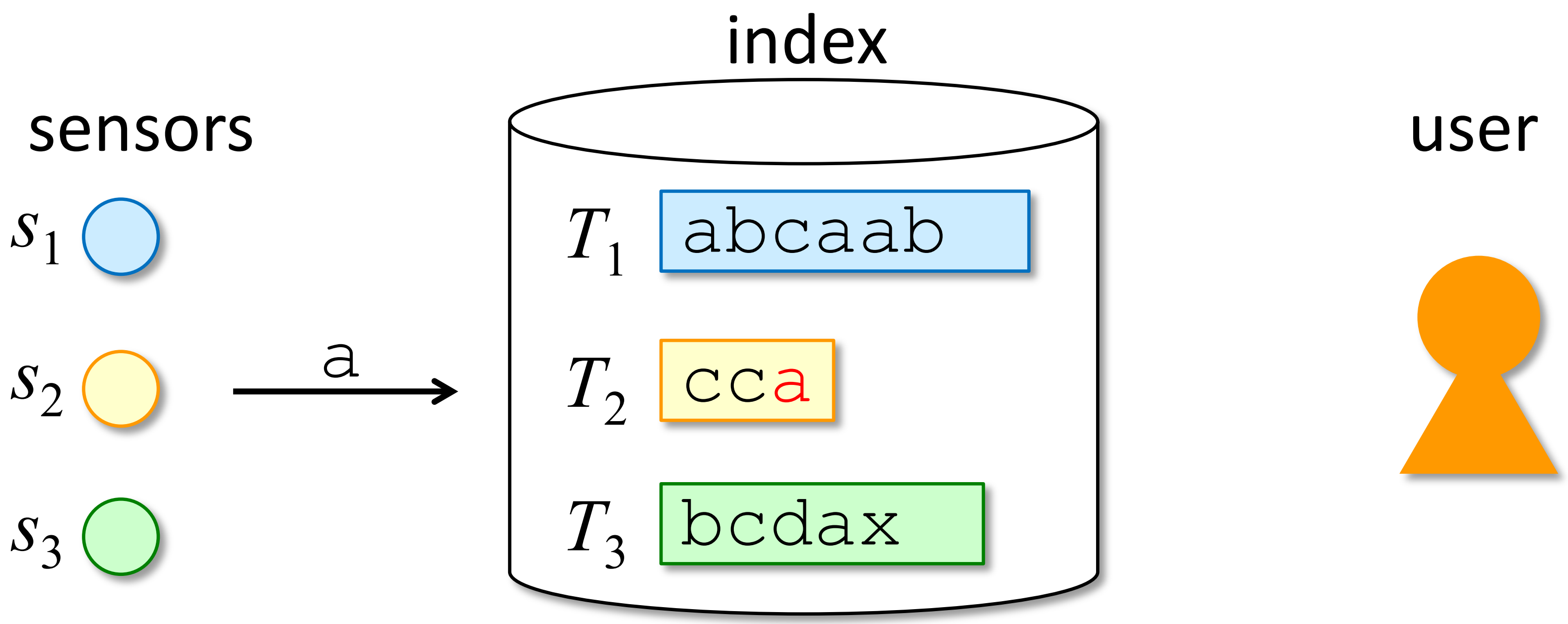
Fully-Online Indexing of Multiple Strings

- Goal: Indexing multiple strings in a **fully-online manner** where each string can grow **any time**.
- Motivation: Indexing multi online/streaming data.
  - Sensing data, trajectory data, SNS, etc.



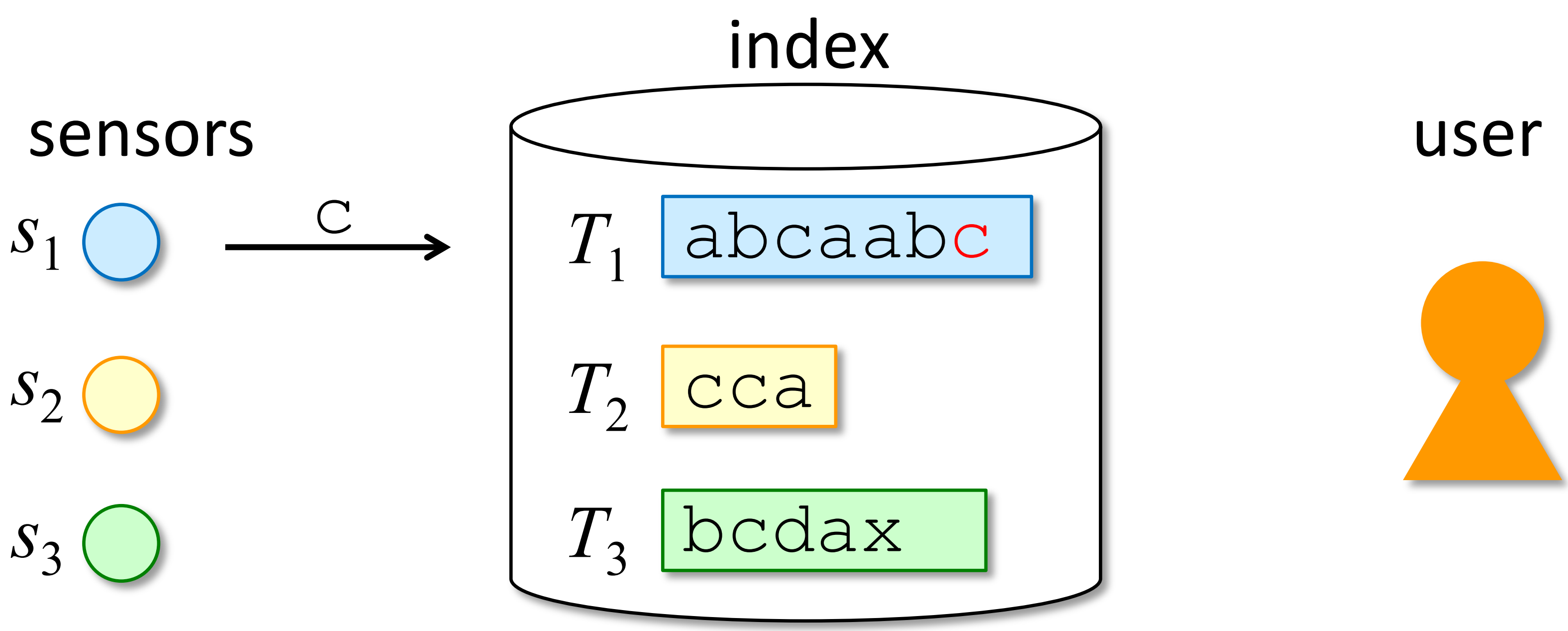
Fully-Online Indexing of Multiple Strings

- Goal: Indexing multiple strings in a **fully-online manner** where each string can grow **any time**.
- Motivation: Indexing multi online/streaming data.
  - Sensing data, trajectory data, SNS, etc.



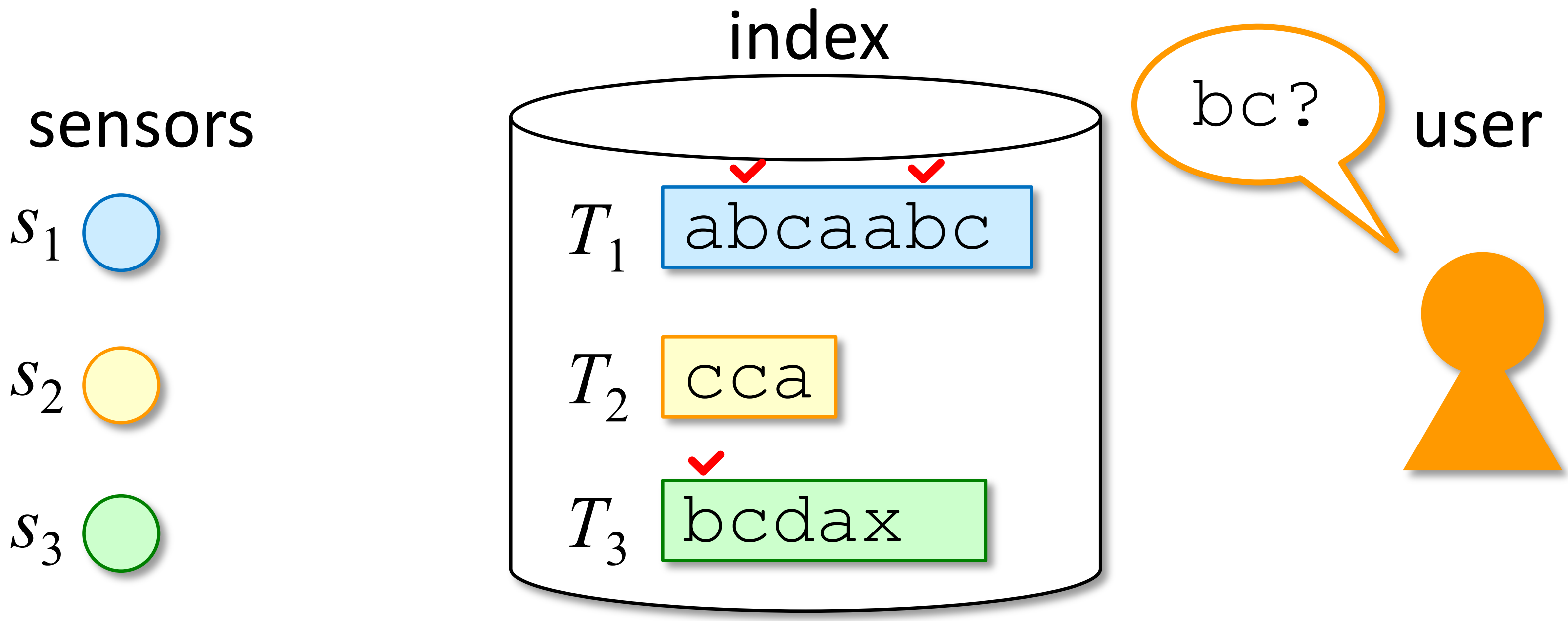
Fully-Online Indexing of Multiple Strings

- Goal: Indexing multiple strings in a **fully-online manner** where each string can grow **any time**.
- Motivation: Indexing multi online/streaming data.
  - Sensing data, trajectory data, SNS, etc.



Fully-Online Indexing of Multiple Strings

- Goal: Indexing multiple strings in a **fully-online manner** where each string can grow **any time**.
- Motivation: Indexing multi online/streaming data.
  - Sensing data, trajectory data, SNS, etc.



Overview of This Work

- We will consider **suffix trees** and **DAWGs** as indexing structures for fully-online multiple strings.
- For **suffix trees**, we propose a Weiner-type algorithm where strings grow **from right to left**.
- For **DAWGs**, we propose a Blumer et al.-type algorithm where strings grow **from left to right**.
- Our model of computation is the **pointer machine** that is strictly weaker than the word RAM.

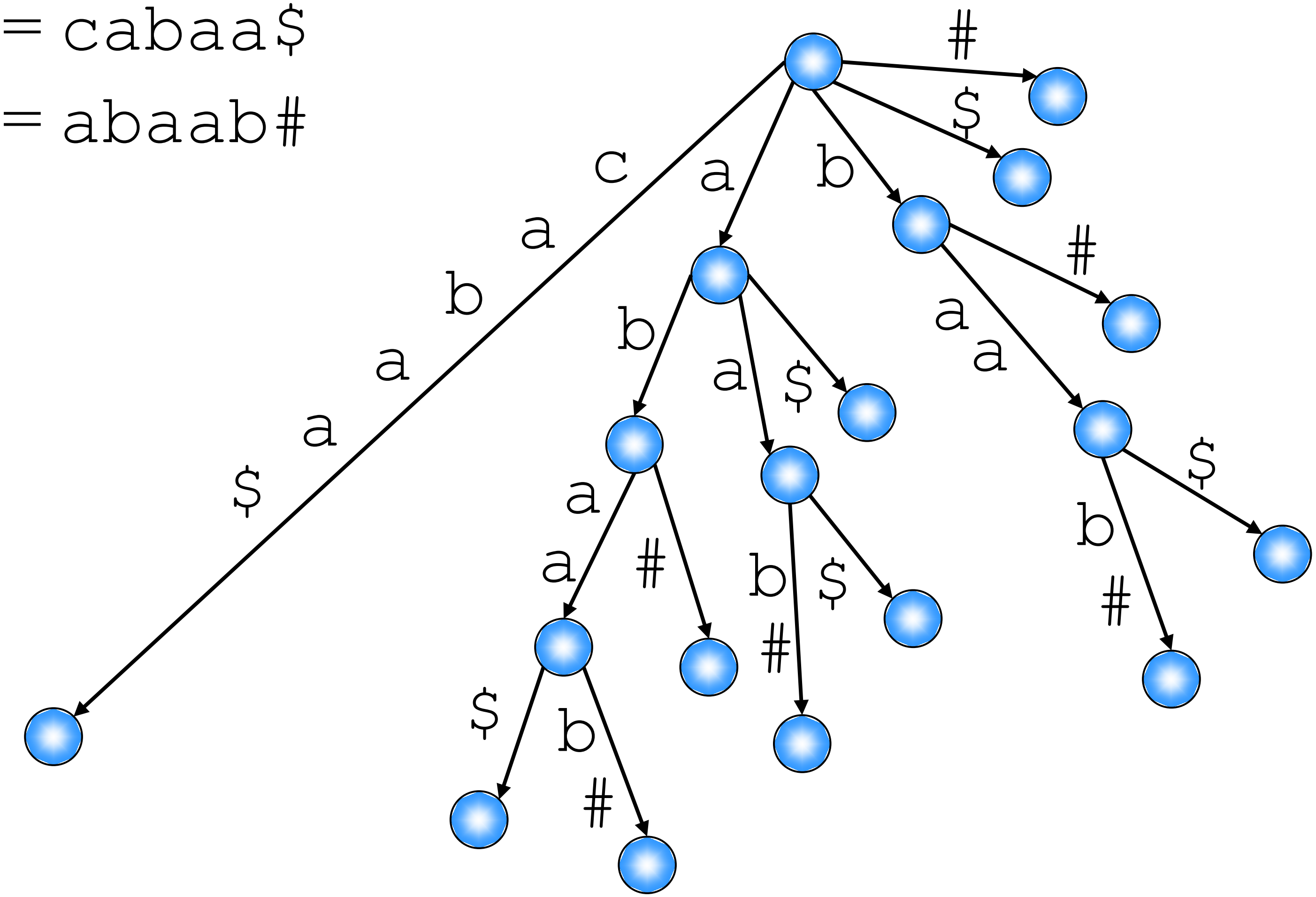
Suffix Trees

[Weiner 1973]

The **suffix tree** of multiple strings is a path-compressed trie that represents all suffixes of the strings.

$T_1 = \texttt{cabaa\$}$

$T_2 = \texttt{abaab\#}$

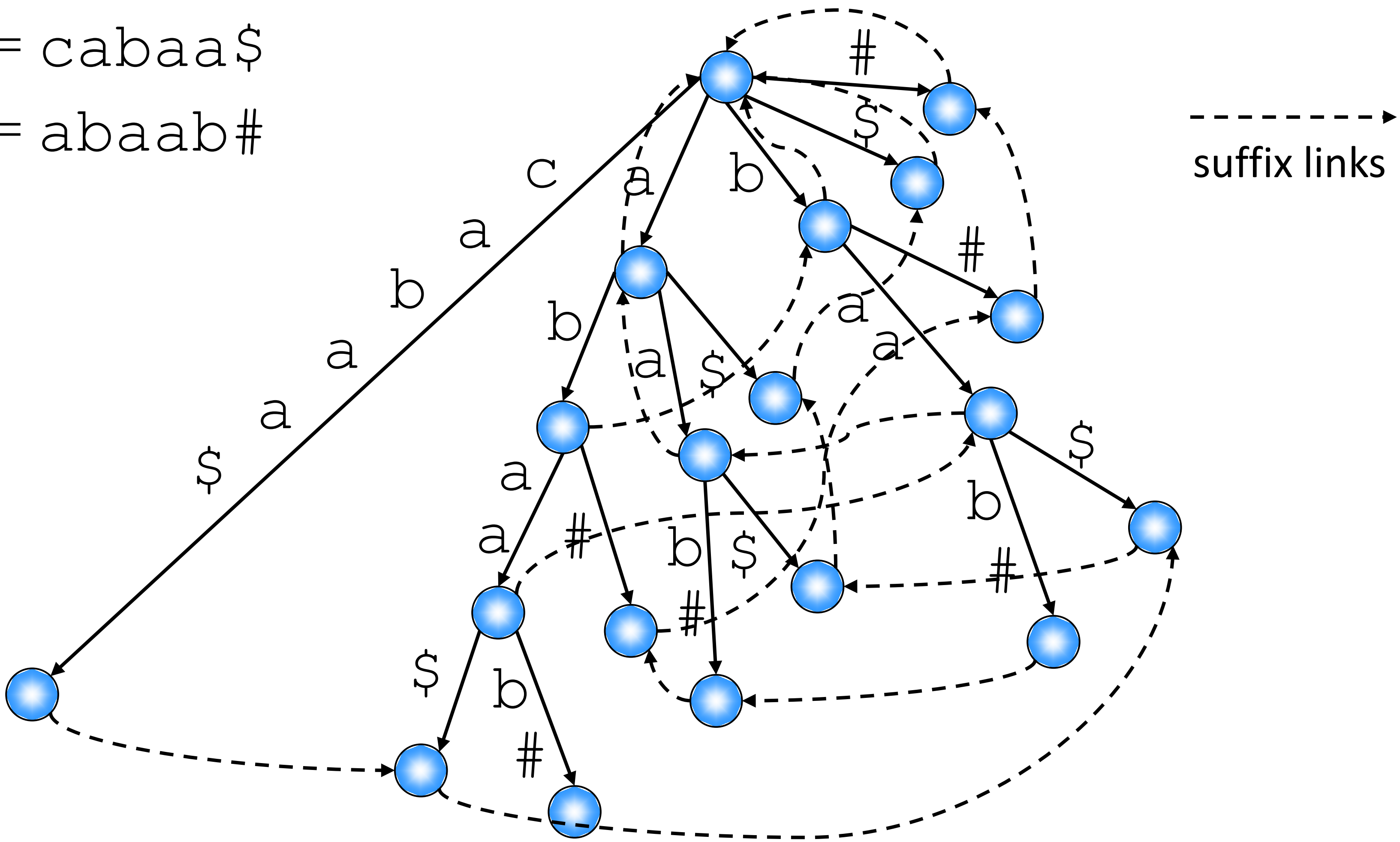


Suffix Links

If av is a node and a is a character, then $\text{suffix\_link}(av) = v$.

$T_1 = \texttt{cabaa\$}$

$T_2 = \texttt{abaab\#}$

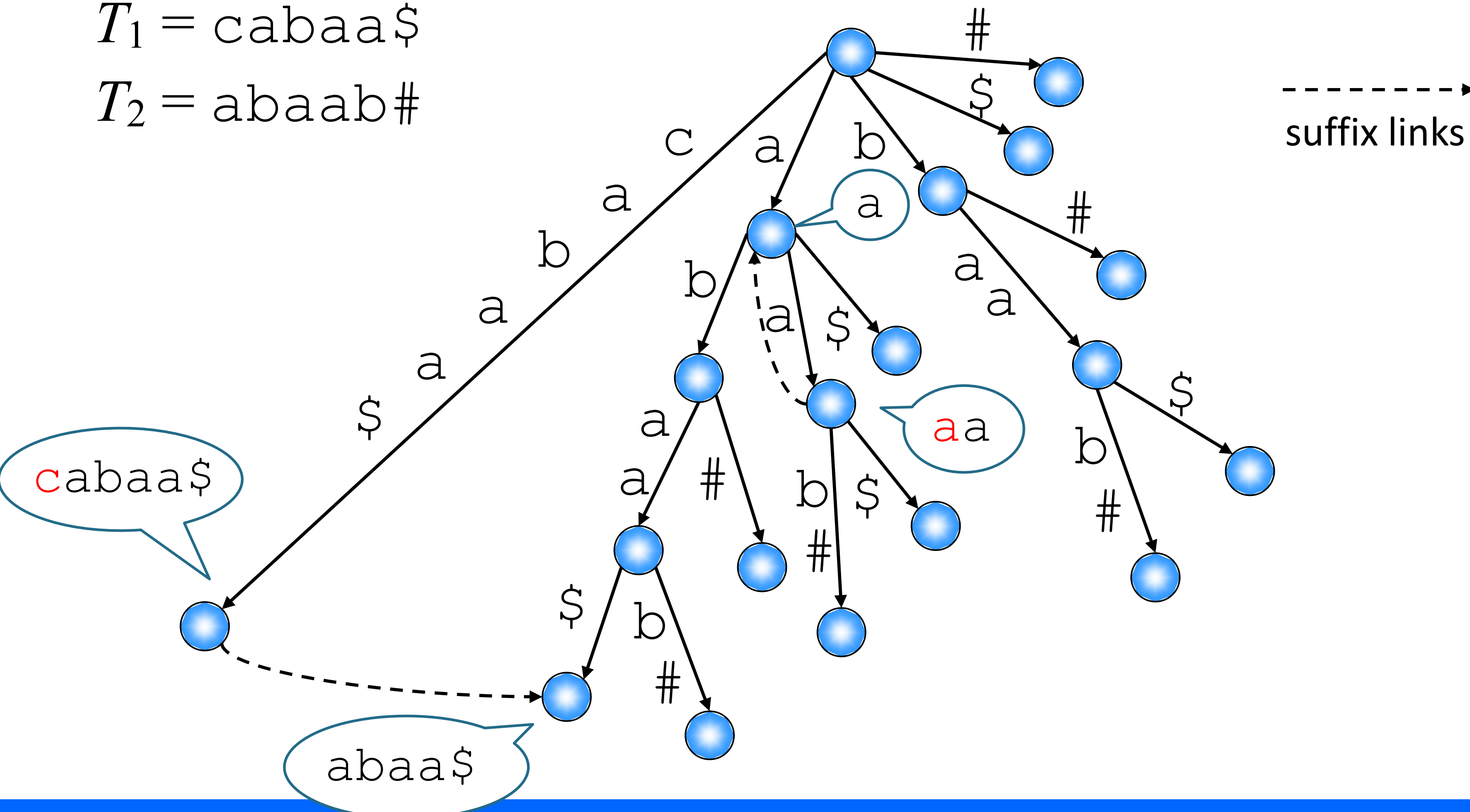


Suffix Links

If av is a node and a is a character, then $\text{suffix\_link}(av) = v$.

$T_1 = \texttt{cabaa\$}$

$T_2 = \texttt{abaab\#}$

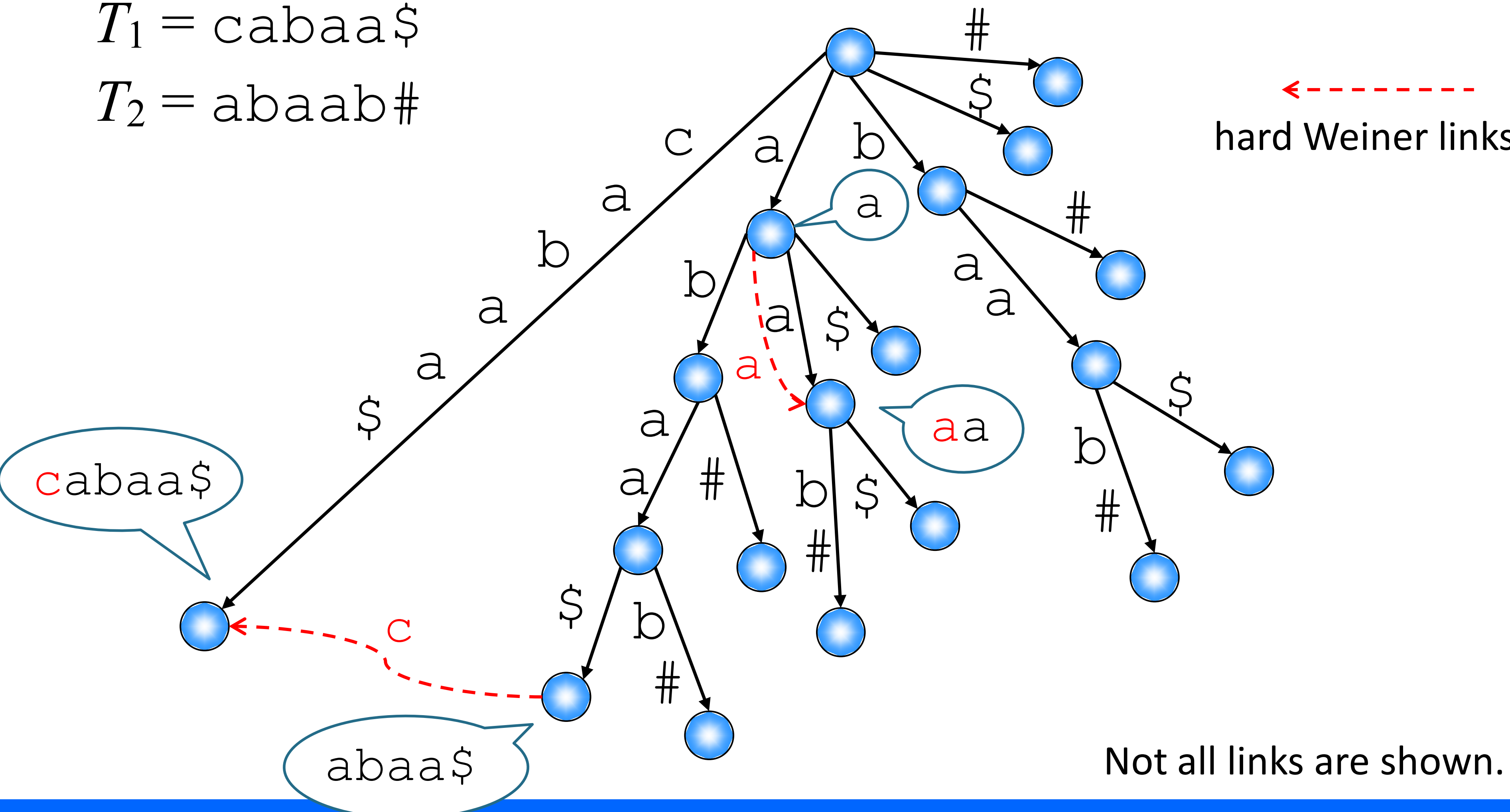


Hard Weiner Links

The *reversed* suffix links with character labels are called **hard Weiner links**.

$T_1 = \mathtt{cabaa\$}$

$T_2 = \mathtt{abaab\#}$



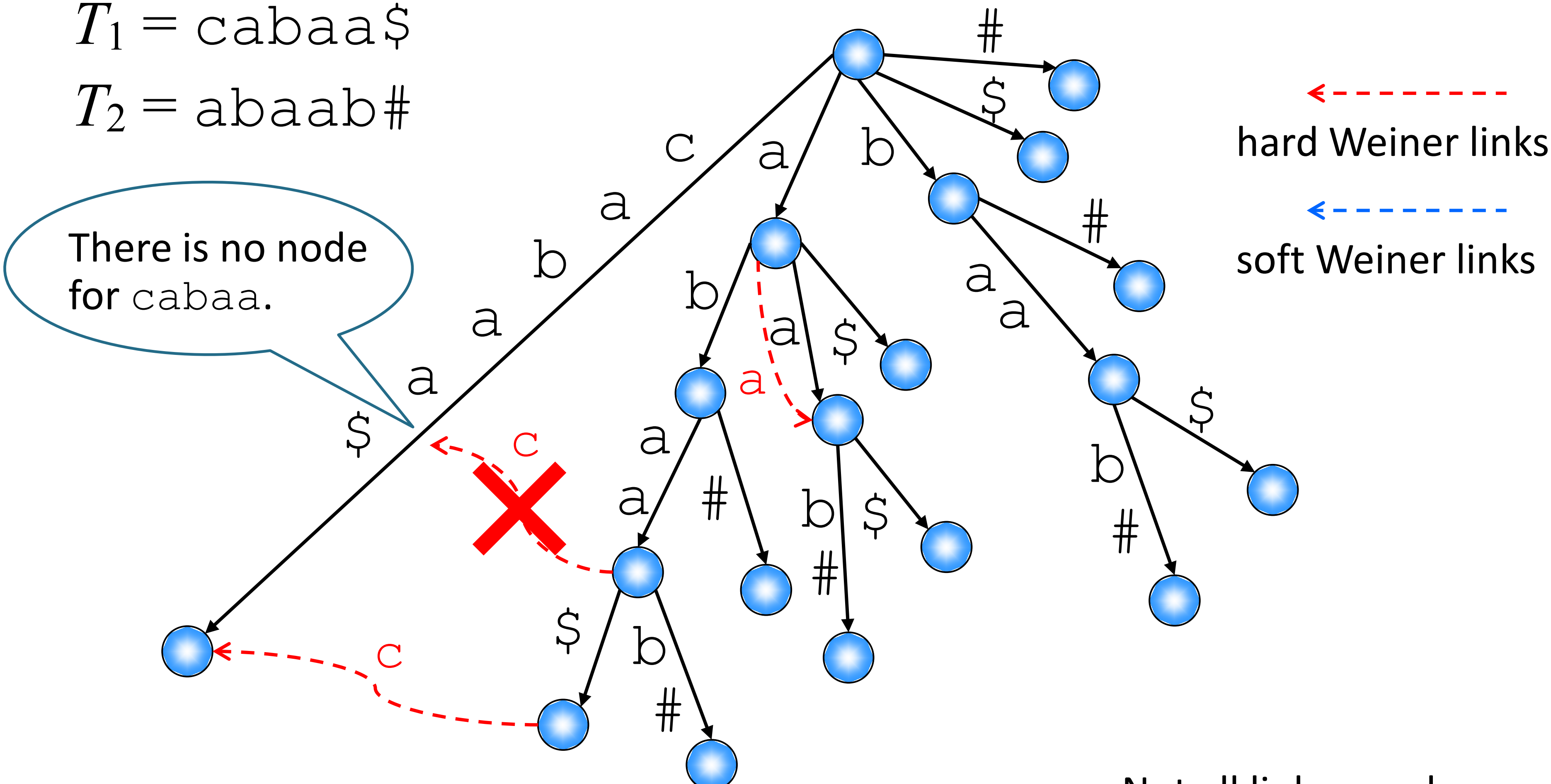
Not all links are shown.

Soft Weiner Links

Soft Weiner links are "generalized" Weiner links.

$T_1 = \texttt{cabaa\$}$

$T_2 = \texttt{abaab\#}$



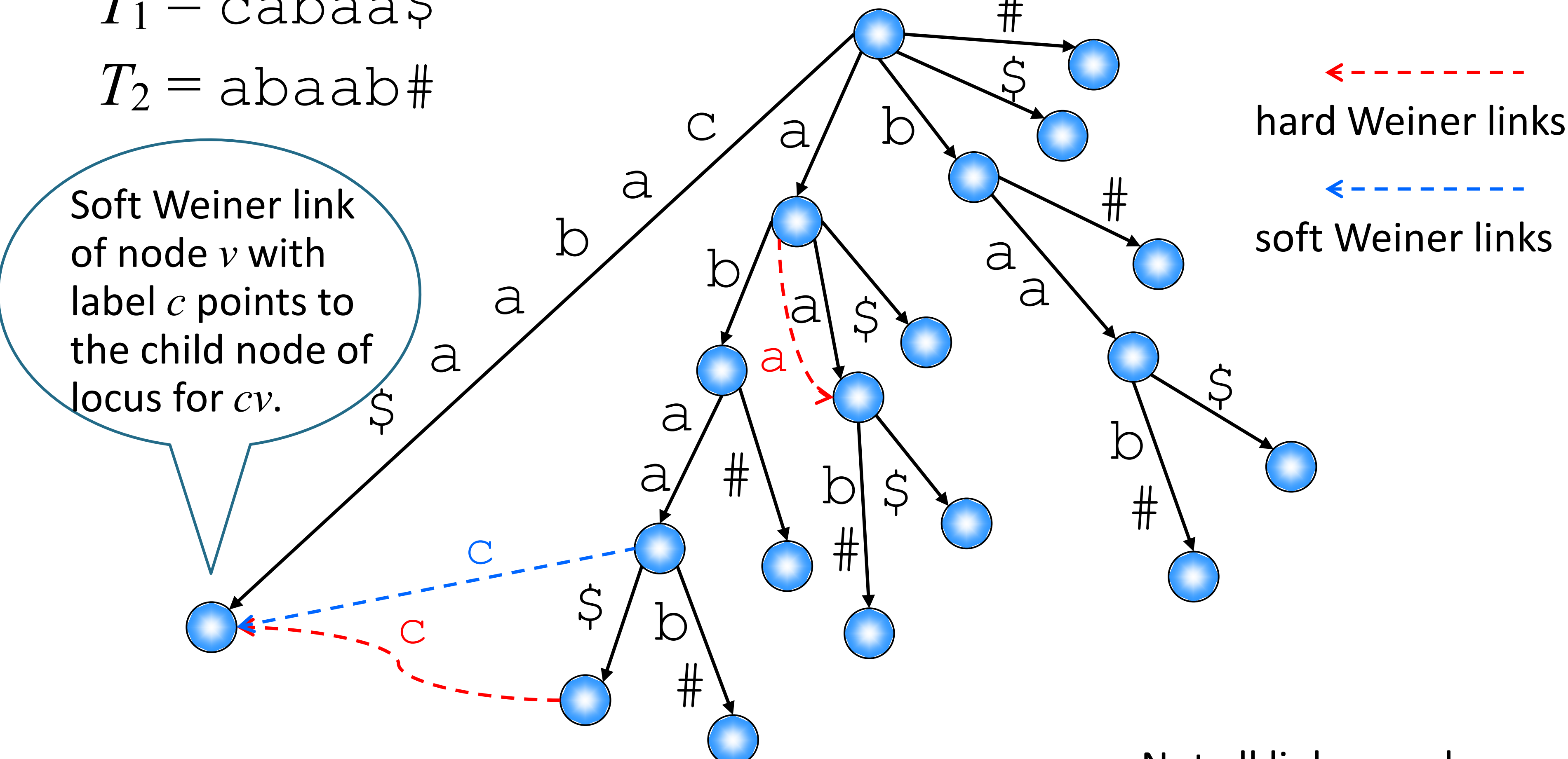
Not all links are shown.

Soft Weiner Links

Soft Weiner links are "generalized" Weiner links.

$T_1 = \texttt{cabaa\$}$

$T_2 = \texttt{abaab\#}$



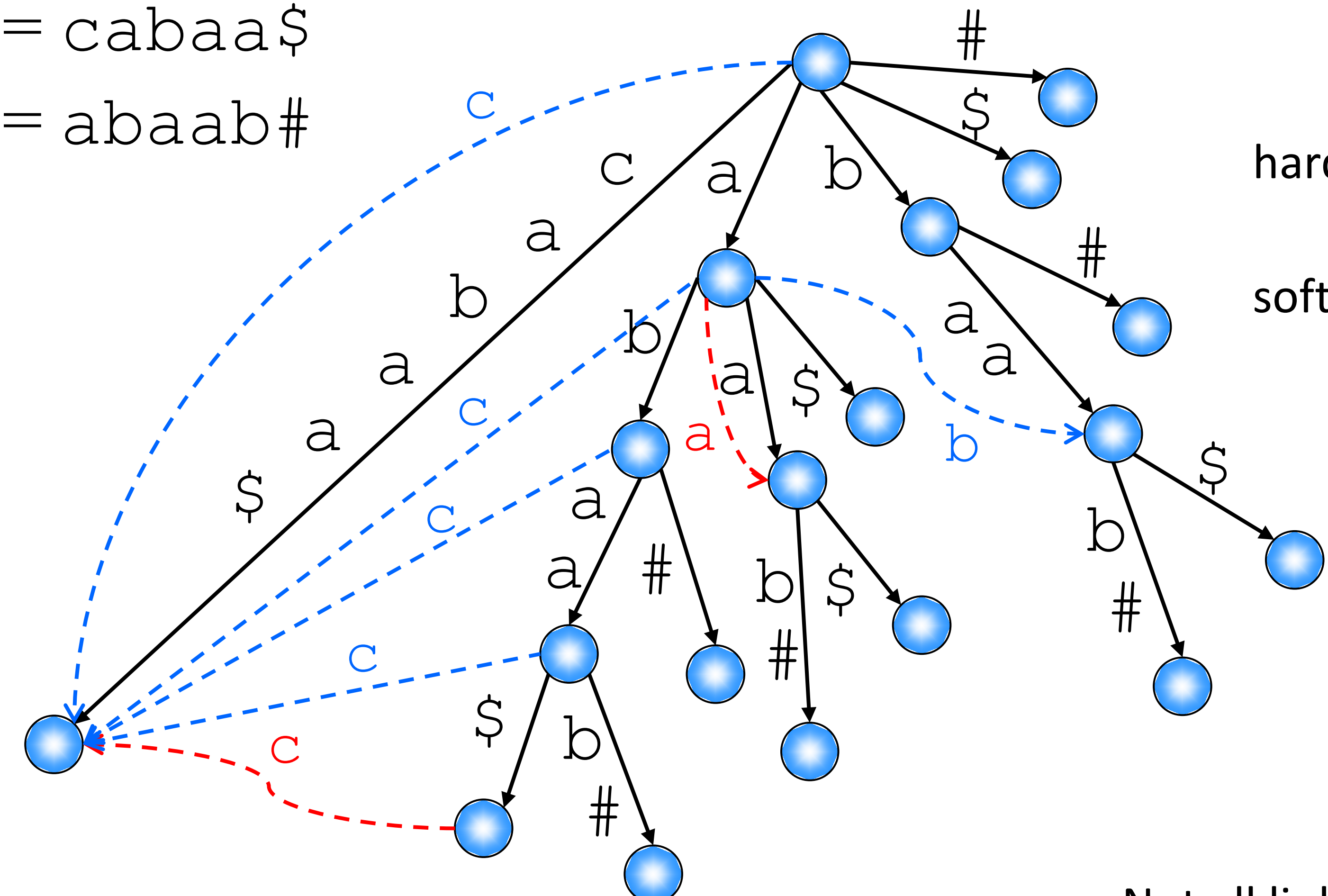
Not all links are shown.

Soft Weiner Links

Soft Weiner links are "generalized" Weiner links.

$T_1 = \texttt{cabaa\$}$

$T_2 = \texttt{abaab\#}$



hard Weiner links

soft Weiner links

Not all links are shown.

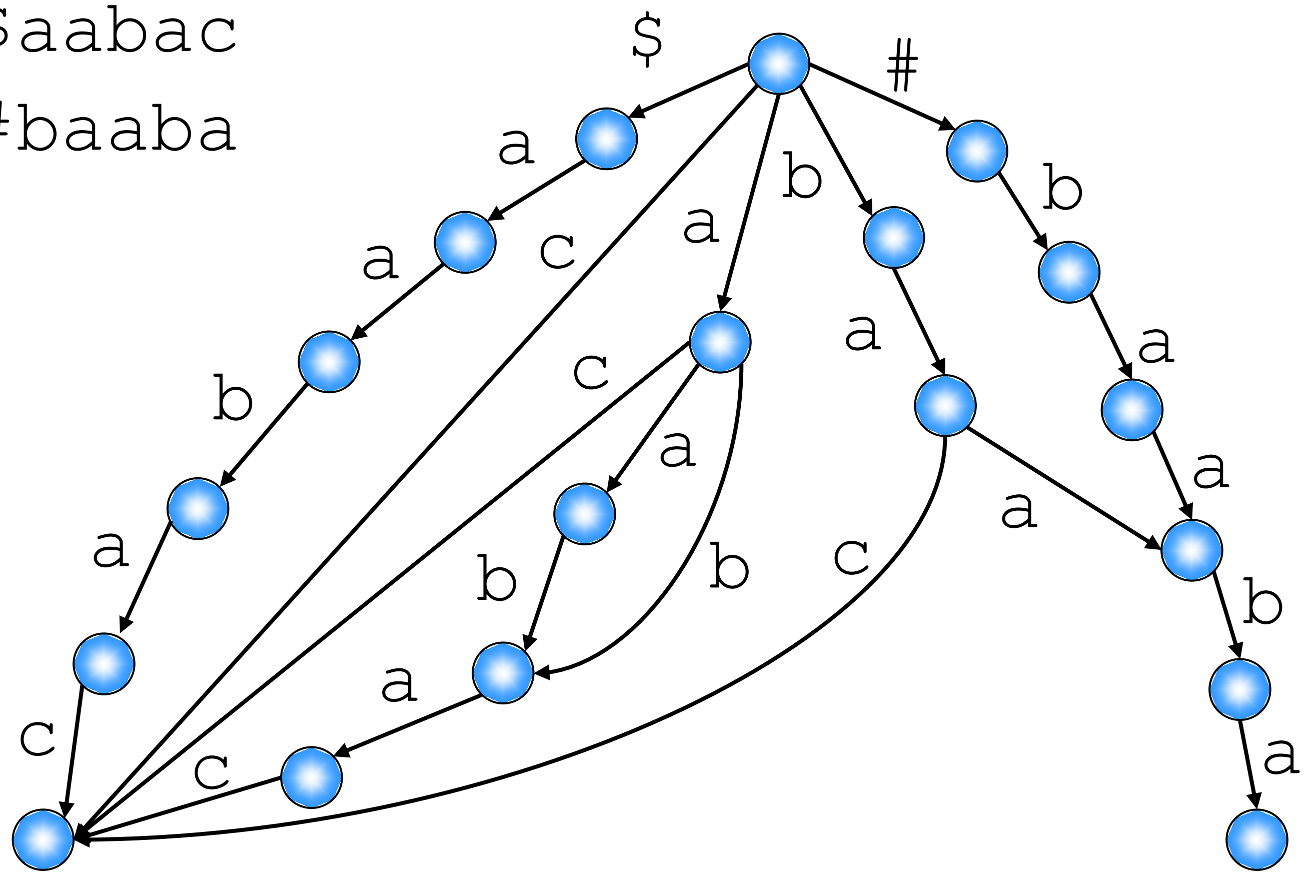
DAWGs

[Blumer et al. 1987]

The **DAWG** of multiple strings is a linear-size automaton that recognizes all substrings of the strings.

$S_1 = \texttt{\$aabac}$

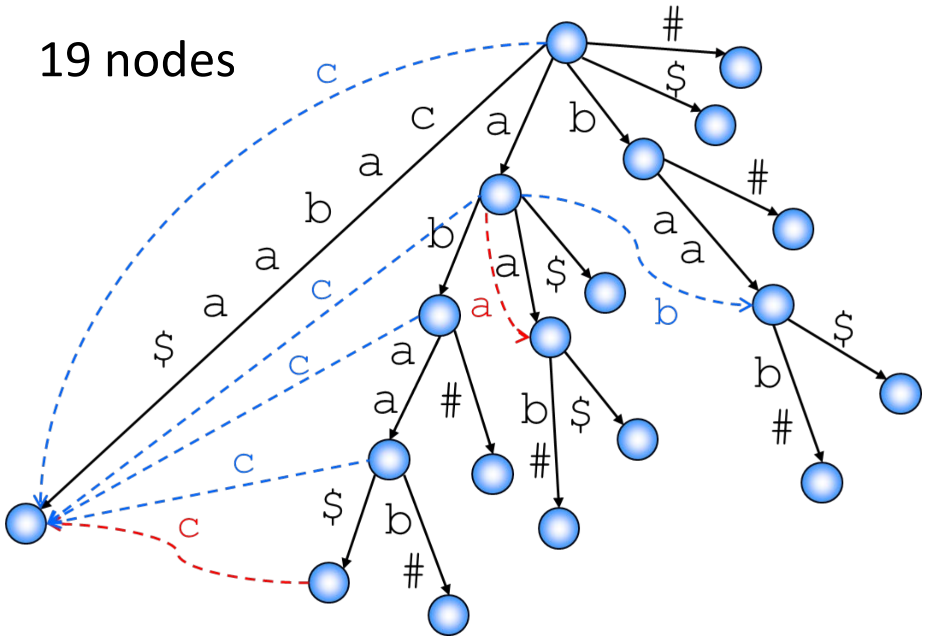
$S_2 = \texttt{\#baaba}$



Duality of Suffix Trees and DAWGs

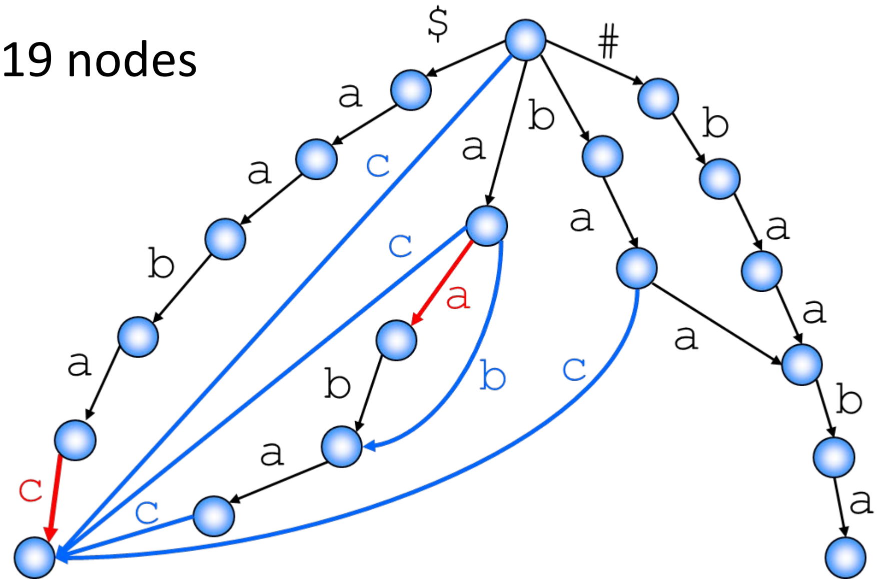
A) There is a one-to-one correspondence between **the nodes of the suffix tree of strings** and **the nodes of the DAWG of the reversed strings**.

19 nodes



Suffix Tree of $T_1 = \mathtt{cabaa\$}$, $T_2 = \mathtt{abaab\#}$

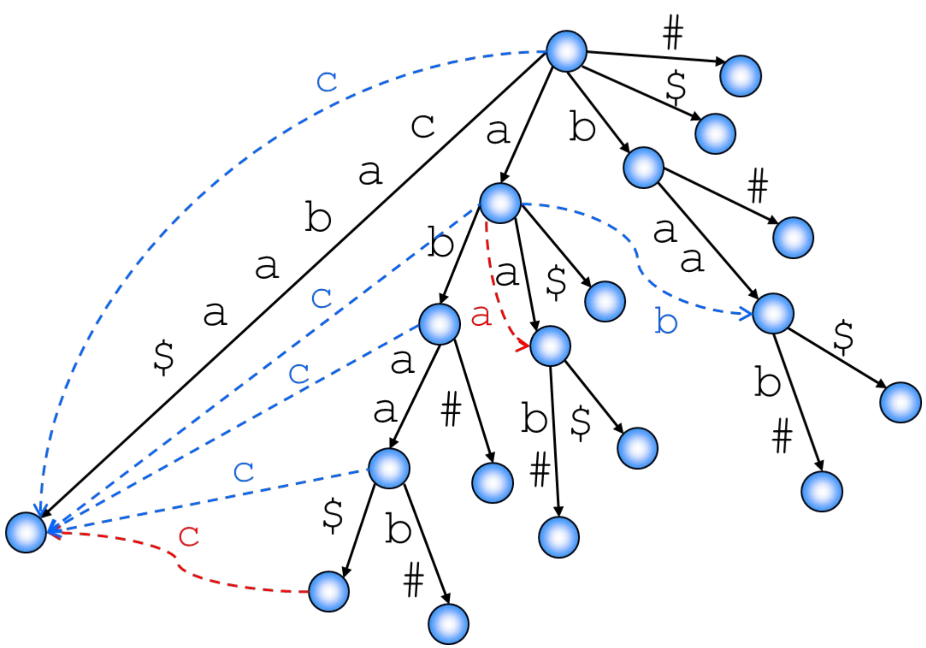
19 nodes



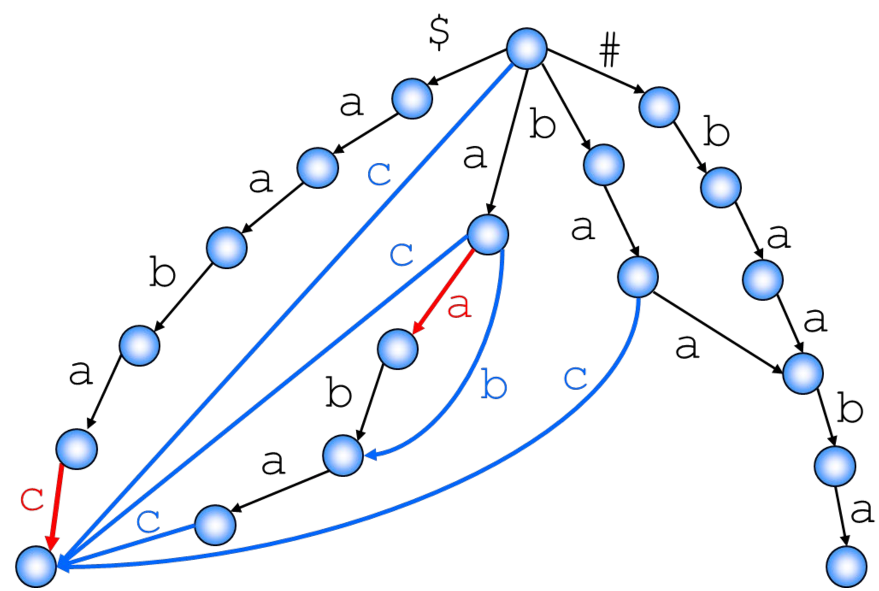
DAWG of $S_1 = \mathtt{\$aabac}$, $S_2 = \mathtt{\#baaba}$

Duality of Suffix Trees and DAWGs

B) There is a one-to-one correspondence between **the Weiner links of the suffix tree of strings** and **the edges of the DAWG of the reversed strings**.



Suffix Tree of $T_1 = \texttt{cabaa\$}$, $T_2 = \texttt{abaab\#}$



DAWG of $S_1 = \texttt{\$aabac}$, $S_2 = \texttt{\#baaba}$

Previous and This Work (Suffix Trees)

Right-to-Left Fully-Online Suffix Tree Construction Time			
algorithm	single string	multiple strings	model
Weiner	$O(n \log \sigma)$	$\Omega(n^{1.5})$	pointer machine
Takagi et al.	$O(n \log \sigma)$	$O(n \log \sigma)$	word RAM
This work	$\boldsymbol{O(n\ (\log \sigma + \log d))}$	$\boldsymbol{O(n\ (\log \sigma + \log d))}$	**pointer machine**

n : total string length , σ : alphabet size, d : max. # in-coming Weiner links

Both $O(n \log \sigma) \subseteq O(n \log n)$ and $O(n\ (\log \sigma + \log d)) \subseteq O(n \log n)$ hold

➔ The new algorithm achieves the same worst-case complexity on a **weaker model** of computation (**pointer machine**).

Previous and This Work (DAWGs)

Left-to-Right Fully-Online DAWG Construction Time			
algorithm	single string	multiple strings	model
Blumer et al.	$O(n \log \sigma)$	$\Omega(n^{1.5})$	pointer machine
Takagi et al.	$O(n \log \sigma)$	$O(n \log \sigma)$	word RAM
This work	$\boldsymbol{O(n\ (\log \sigma + \log d))}$	$\boldsymbol{O(n\ (\log \sigma + \log d))}$	**pointer machine**

n : total string length , σ : alphabet size, d : max. # in-coming Weiner links

Takagi et al.'s method only maintains an implicit representation of DAWG.

➔ The new algorithm is the **first non-trivial algorithm** that maintains an **explicit representation of DAWG** for fully-online multiple stings.

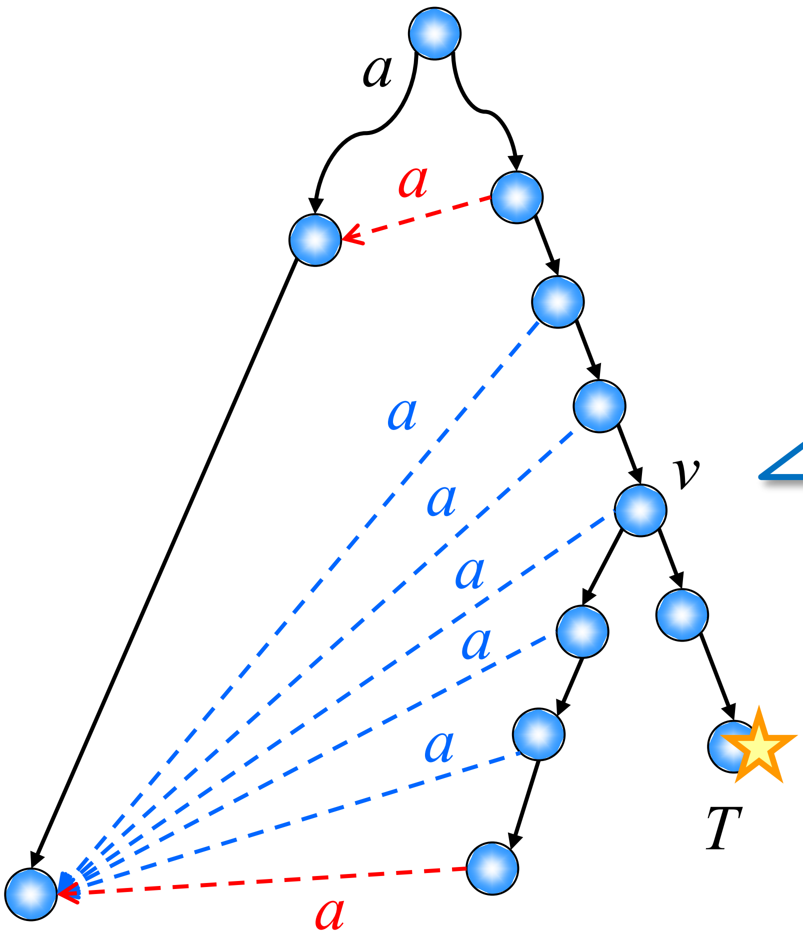
Pointer Machine [cf. Tarjan 1979]

- The **pointer machine** is an abstract model of computation where the state of computation is stored as a digraph. Each node contains a const. number of data and pointers.
- The pointer machine **supports** instructions **(1)-(3):**
  (1) creating / deleting nodes and pointers;
  (2) manipulating data;
  (3) performing comparisons,

  but it **does NOT support** word RAM instructions **(4)-(5):**
  (4) address arithmetics;
  (5) unit-cost bit-wise operations.
- Still, the pointer machine serves as a good basis for modelling linked structures such as trees and graphs.

Weiner's Algorithm (Blumer et al. version)

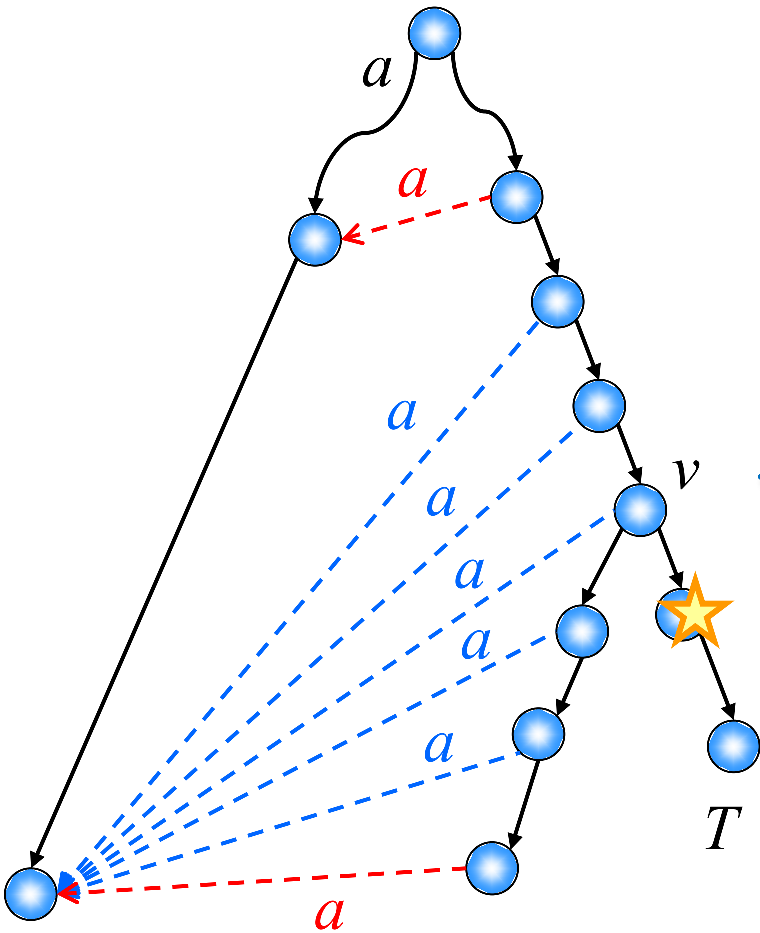
Update string T to aT.



Find the lowest ancestor v of leaf T that has Weiner link with character a.

Weiner's Algorithm (Blumer et al. version)

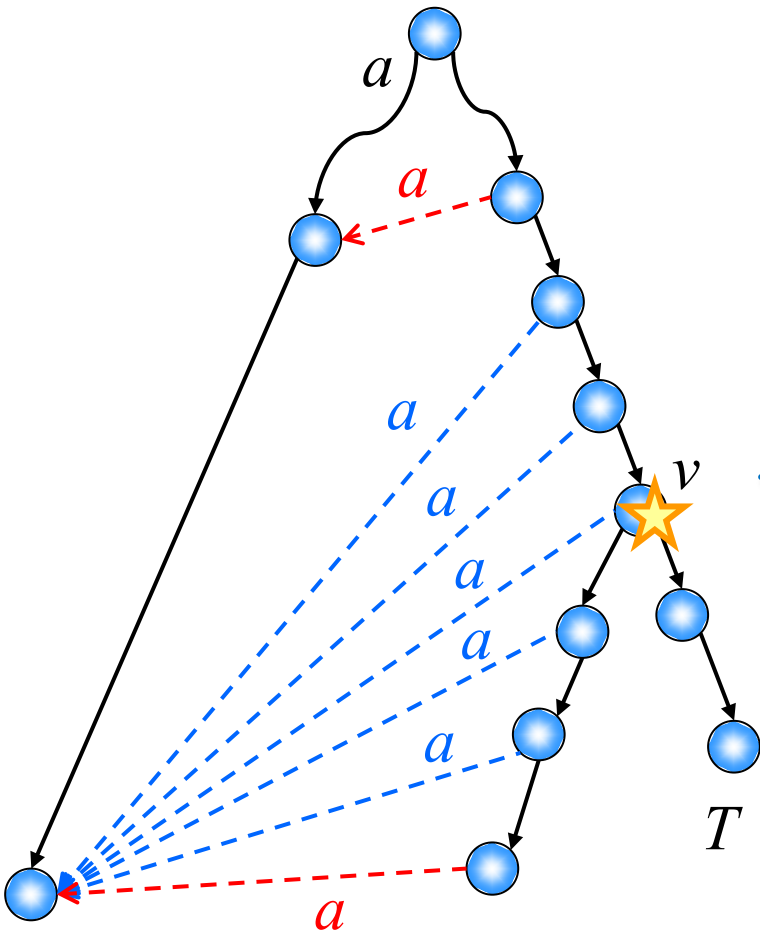
Update string T to aT.



Find the lowest ancestor v of leaf T that has Weiner link with character a.

Weiner's Algorithm (Blumer et al. version)

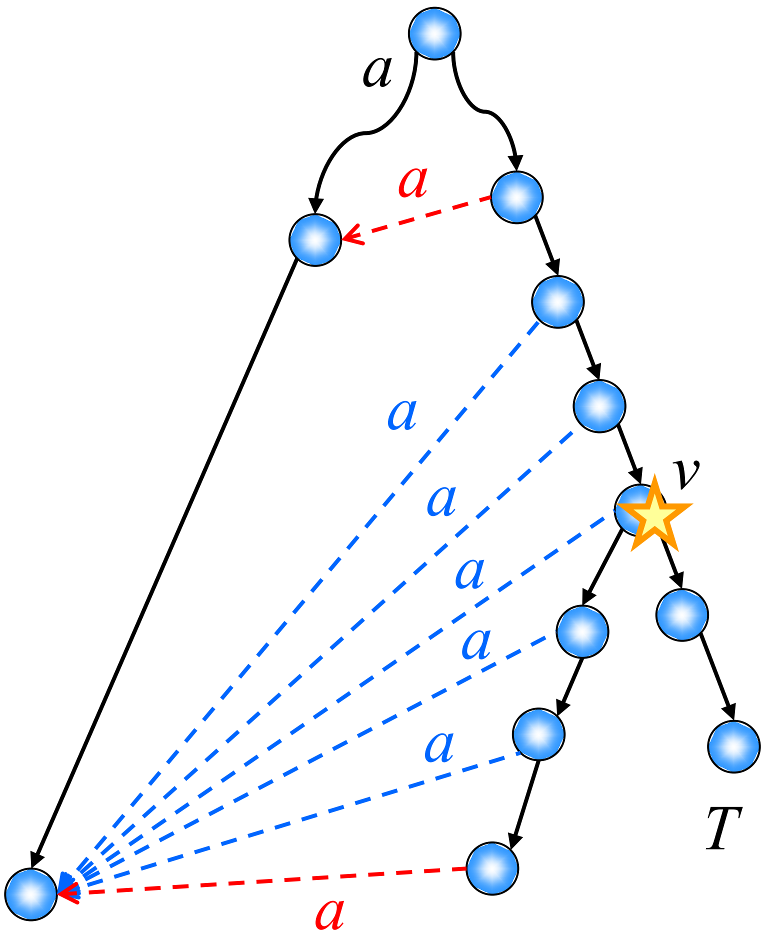
Update string T to aT.



Find the lowest ancestor v of leaf T that has Weiner link with character a.

Weiner's Algorithm (Blumer et al. version)

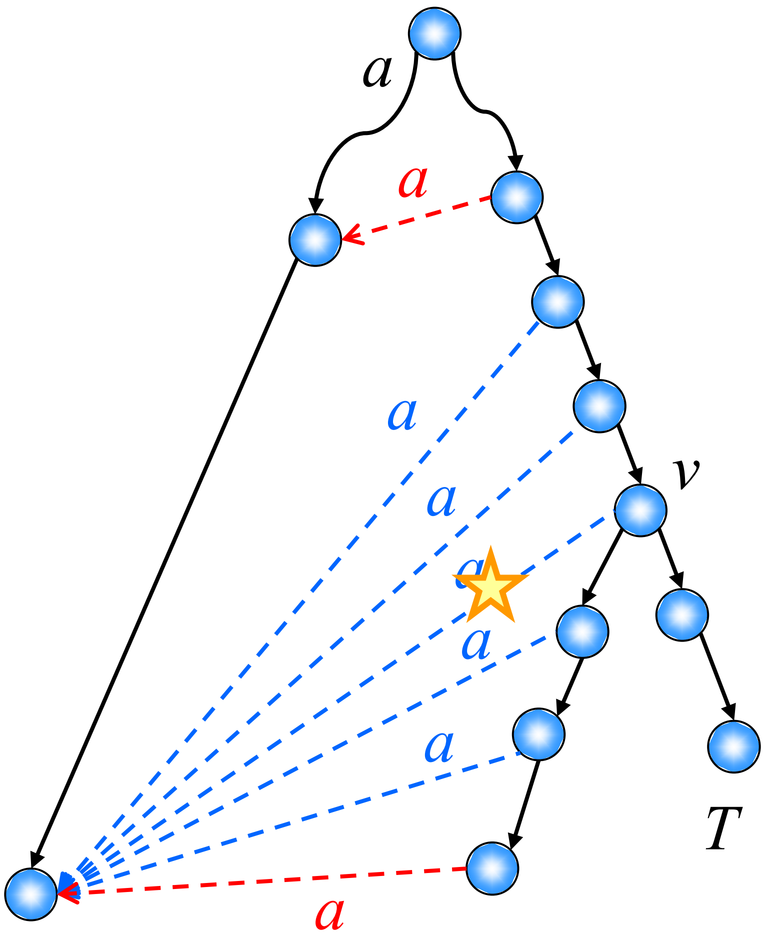
Update string T to aT.



Follow the Weiner link labeled a from v.

Weiner's Algorithm (Blumer et al. version)

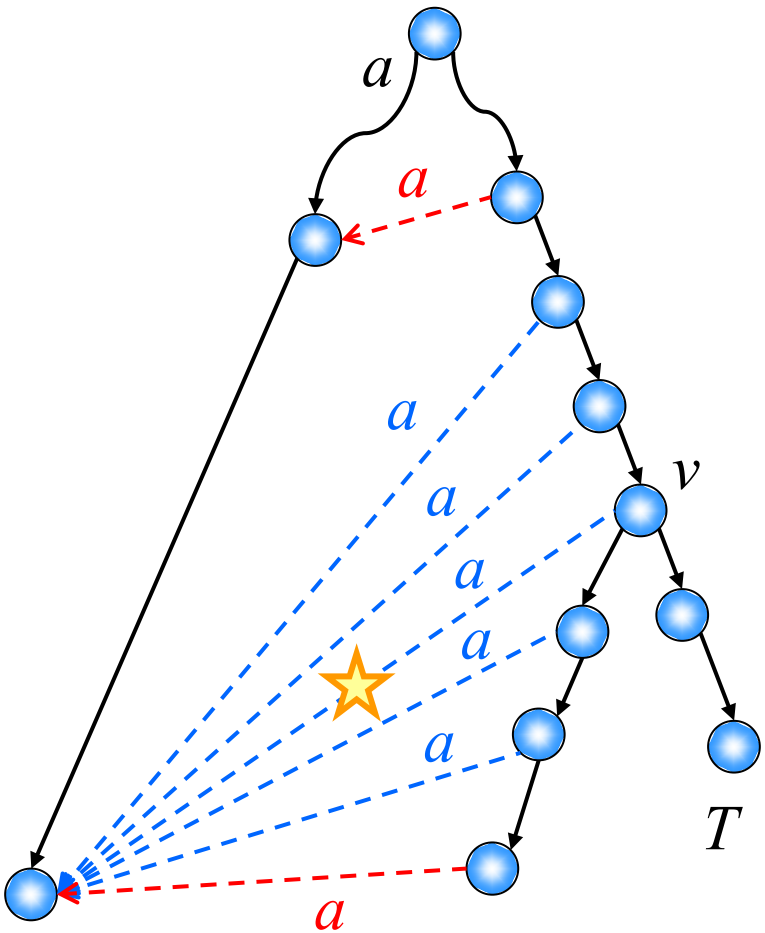
Update string T to aT.



Follow the Weiner link labeled a from v.

Weiner’s Algorithm (Blumer et al. version)

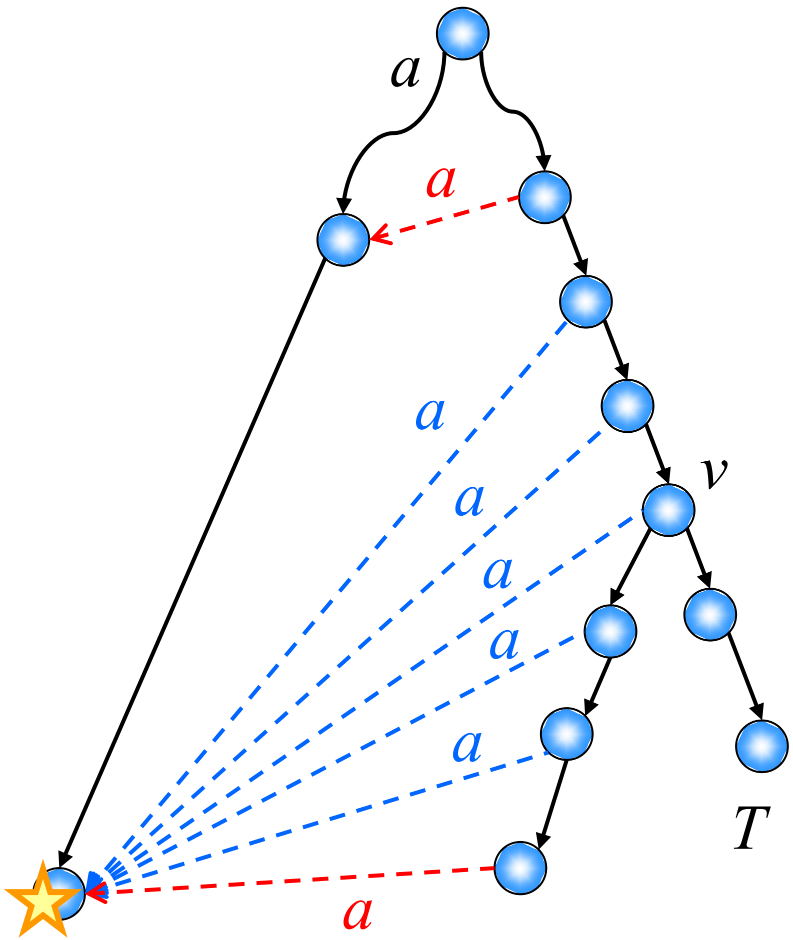
Update string T to aT.



Follow the Weiner link labeled a from v.

Weiner's Algorithm (Blumer et al. version)

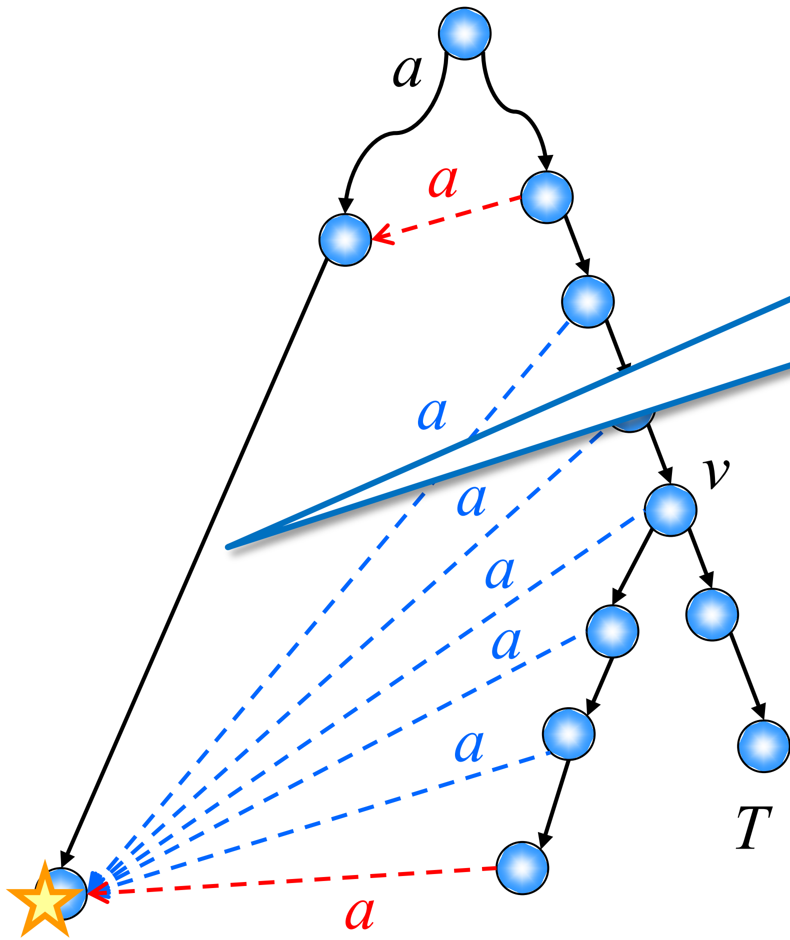
Update string T to aT.



Follow the Weiner link labeled a from v.

Weiner's Algorithm (Blumer et al. version)

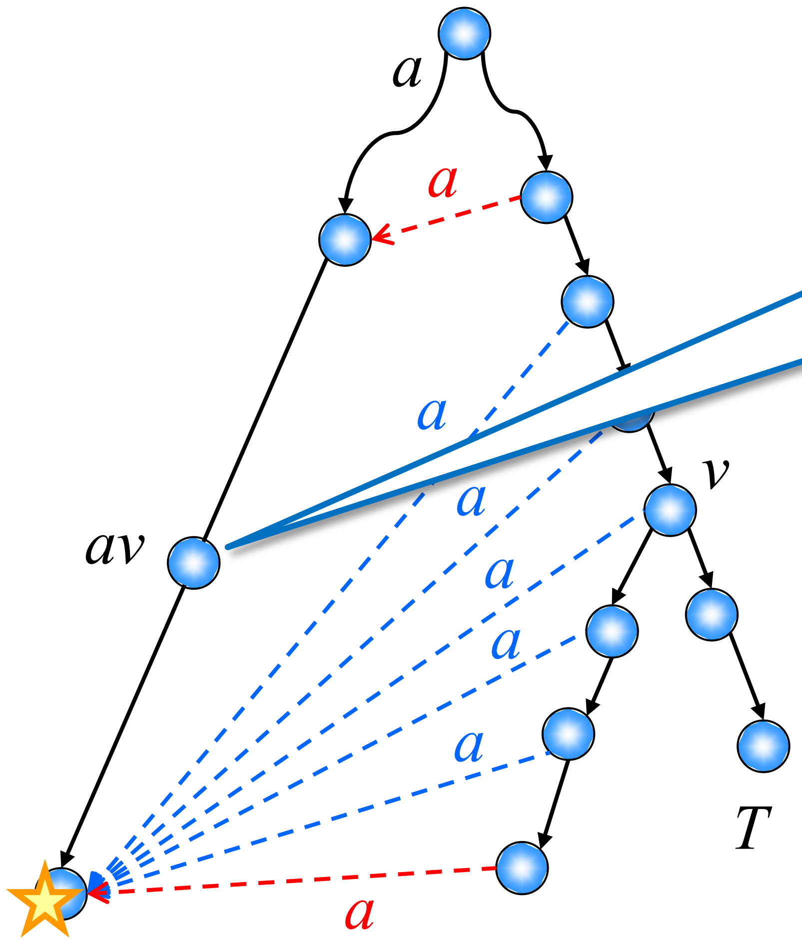
Update string T to aT.



Split the incoming edge at string depth $|av| = |v|+1$.

Weiner's Algorithm (Blumer et al. version)

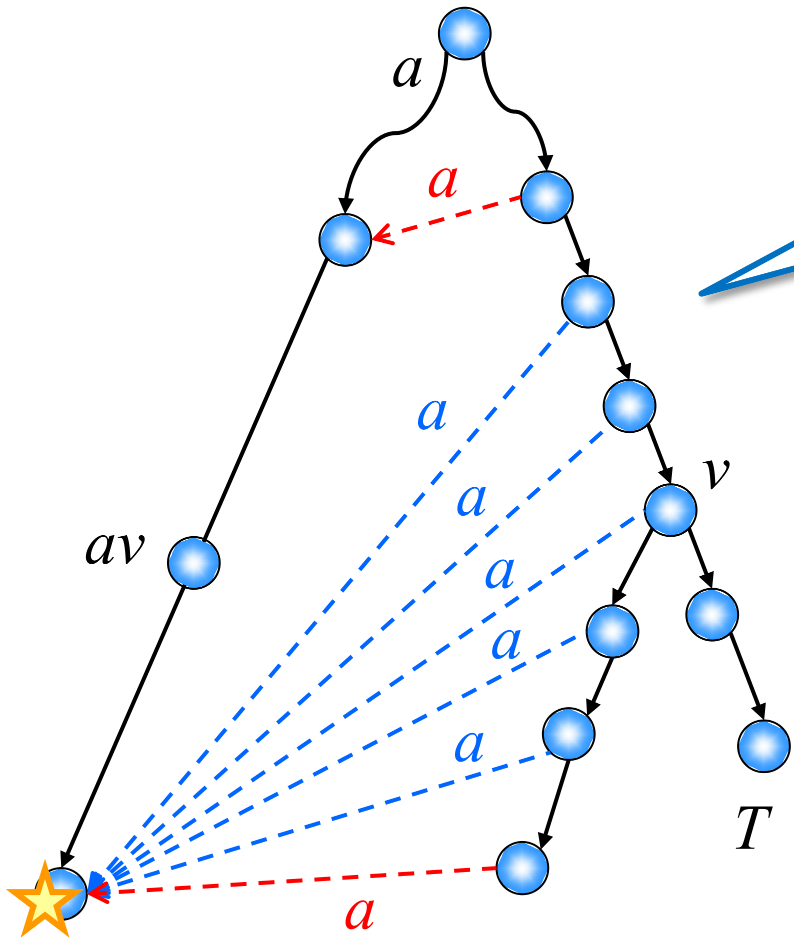
Update string T to aT.



Split the incoming edge at string depth $|av| = |v|+1$.

Weiner's Algorithm (Blumer et al. version)

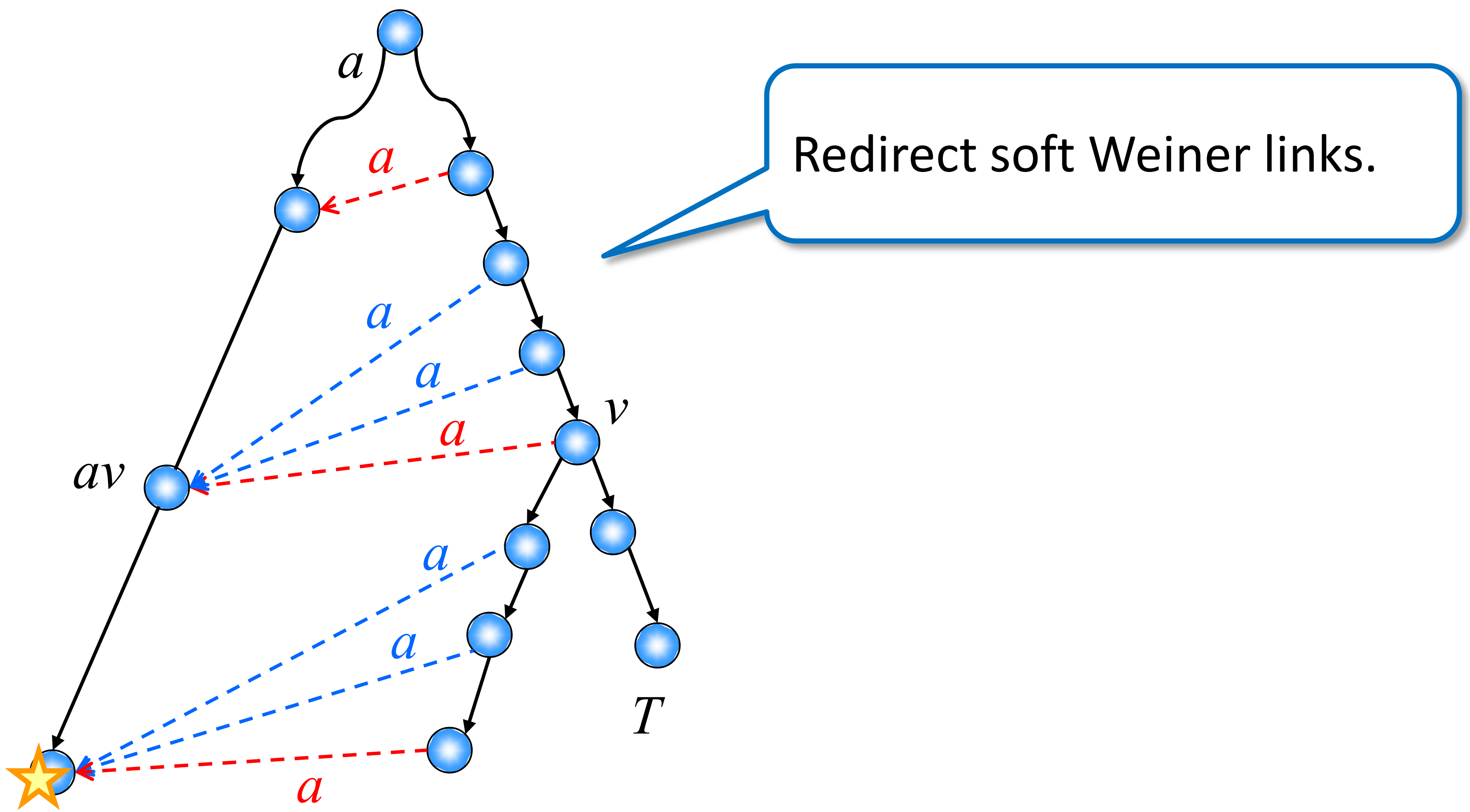
Update string T to aT.



Redirect soft Weiner links.

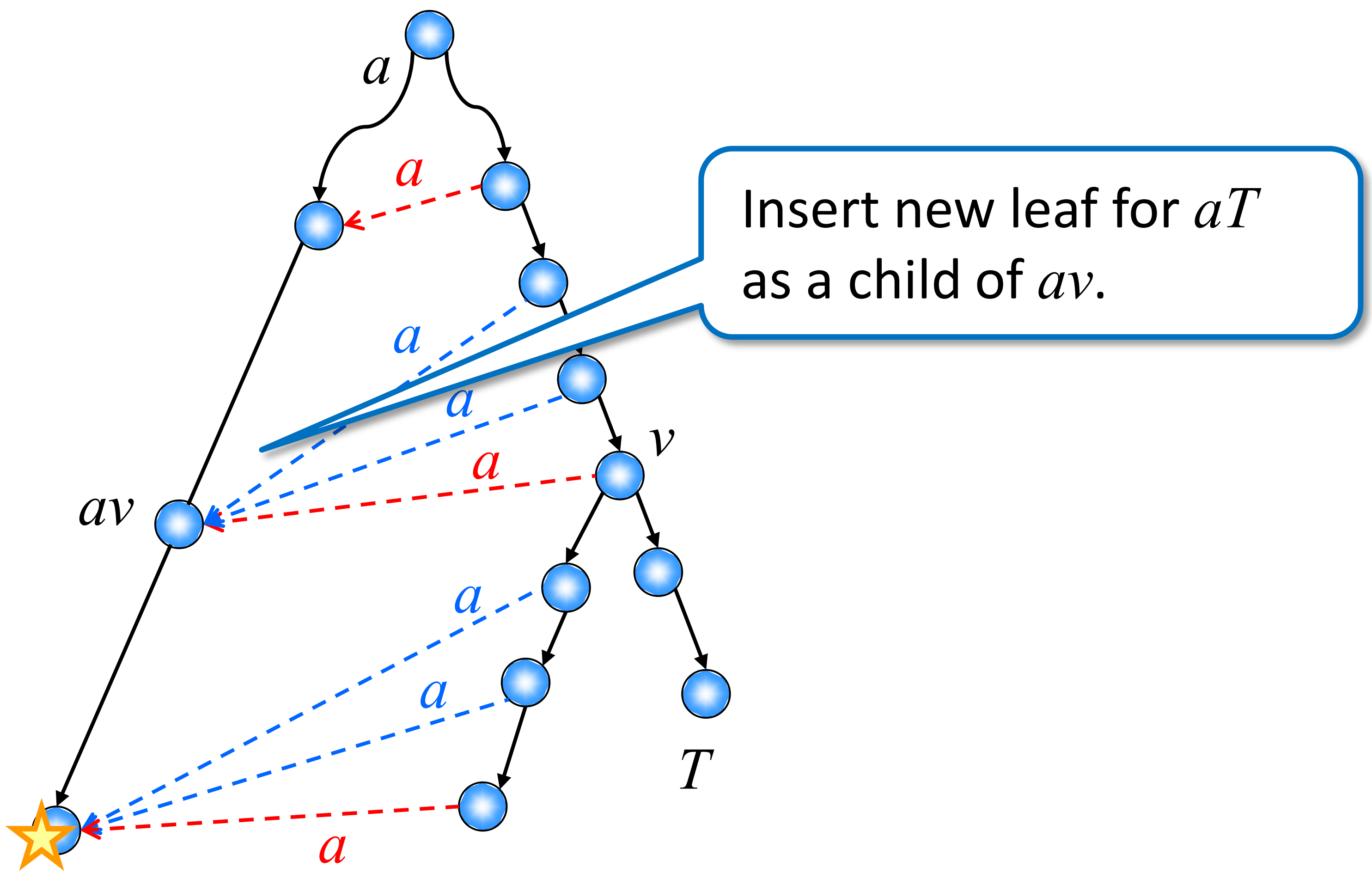
Weiner's Algorithm (Blumer et al. version)

Update string T to aT.



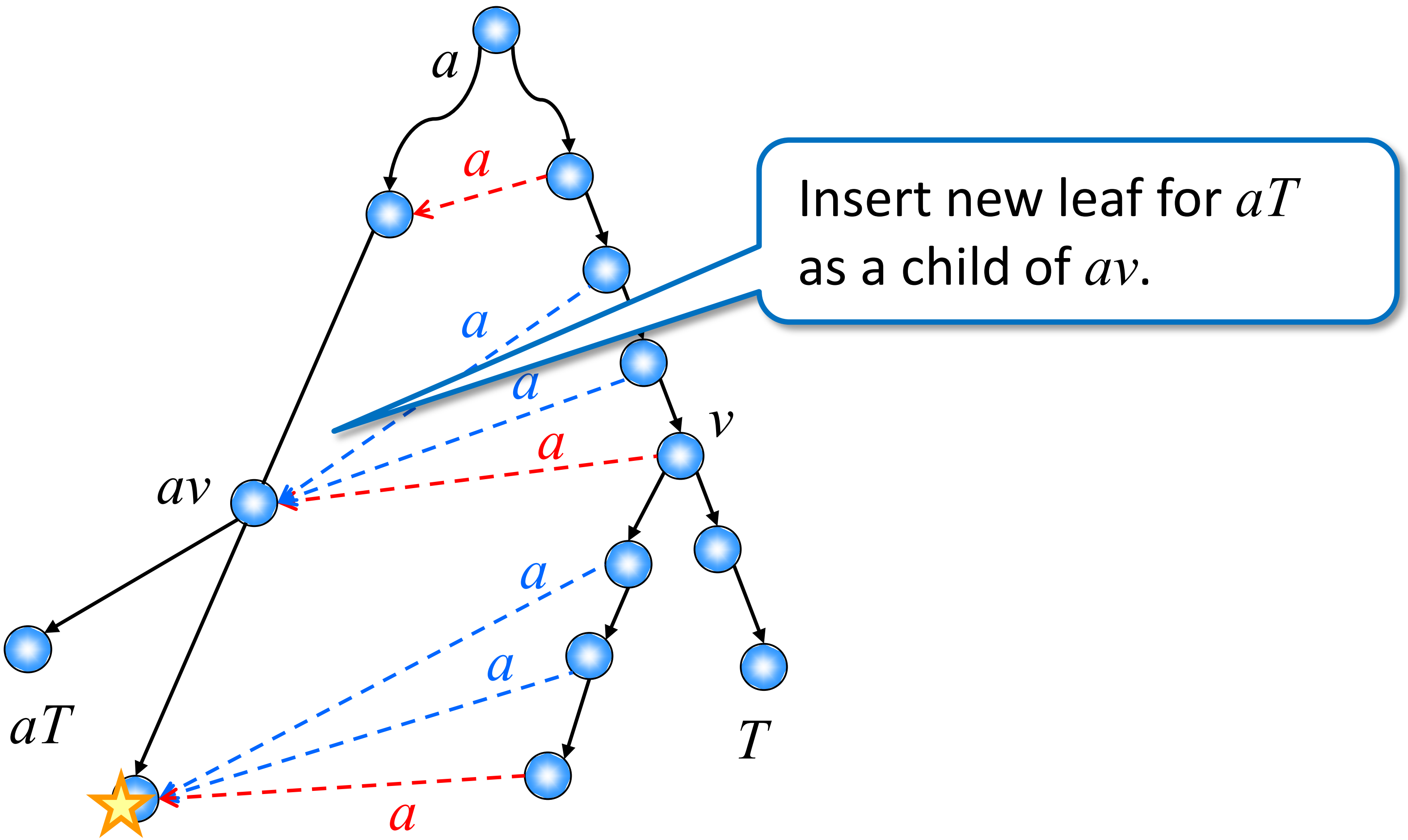
Weiner's Algorithm (Blumer et al. version)

Update string T to aT.



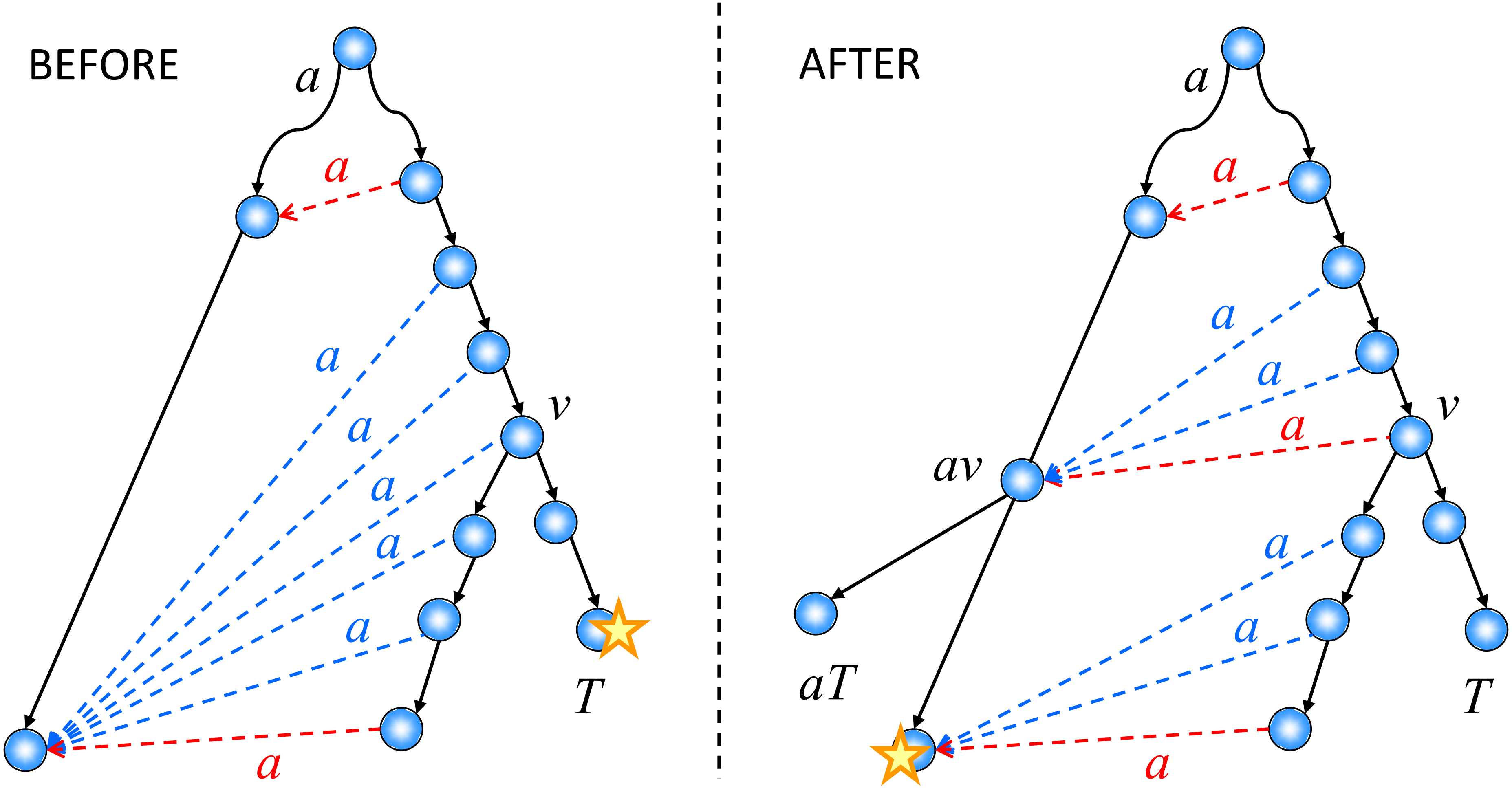
Weiner's Algorithm (Blumer et al. version)

Update string T to aT.



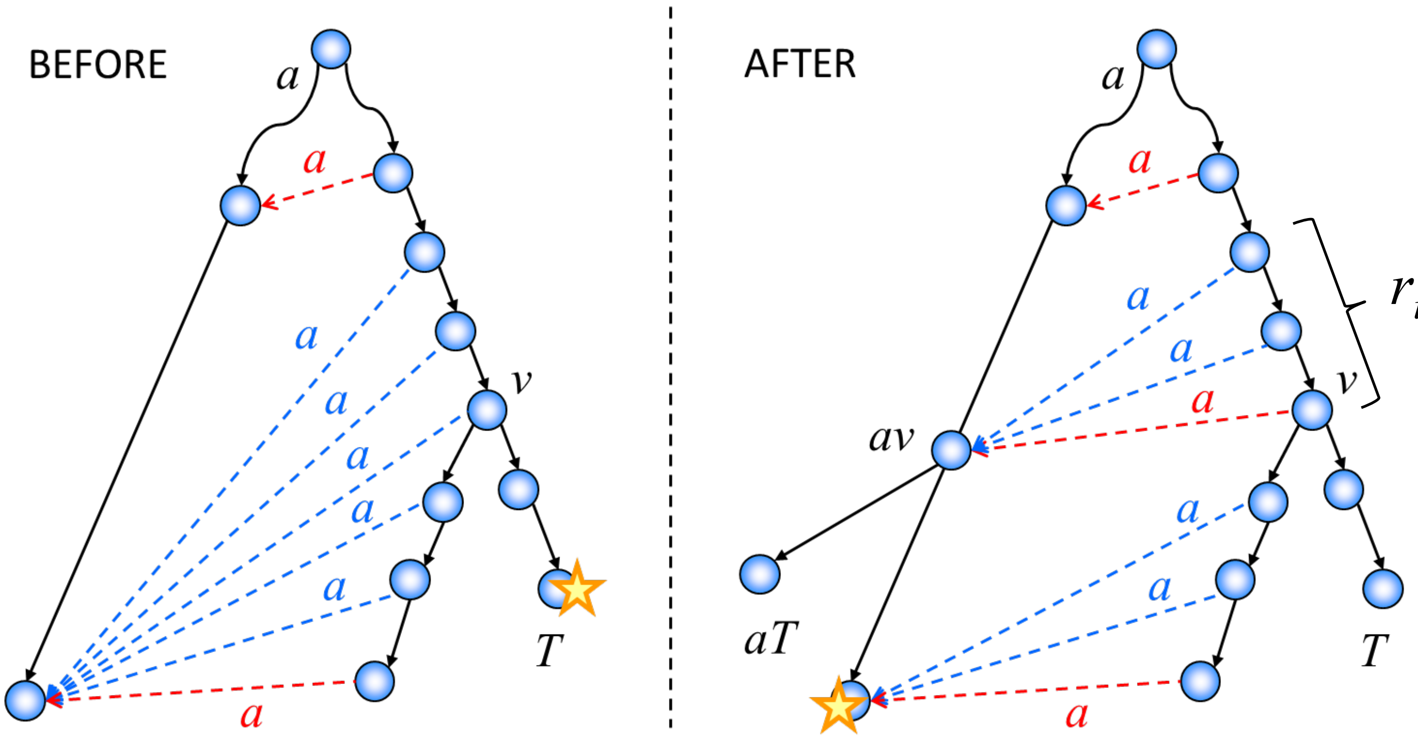
Weiner's Algorithm (Blumer et al. version)

Update string T to aT.



Weiner's Algorithm (Blumer et al. version)

Update string T to aT.

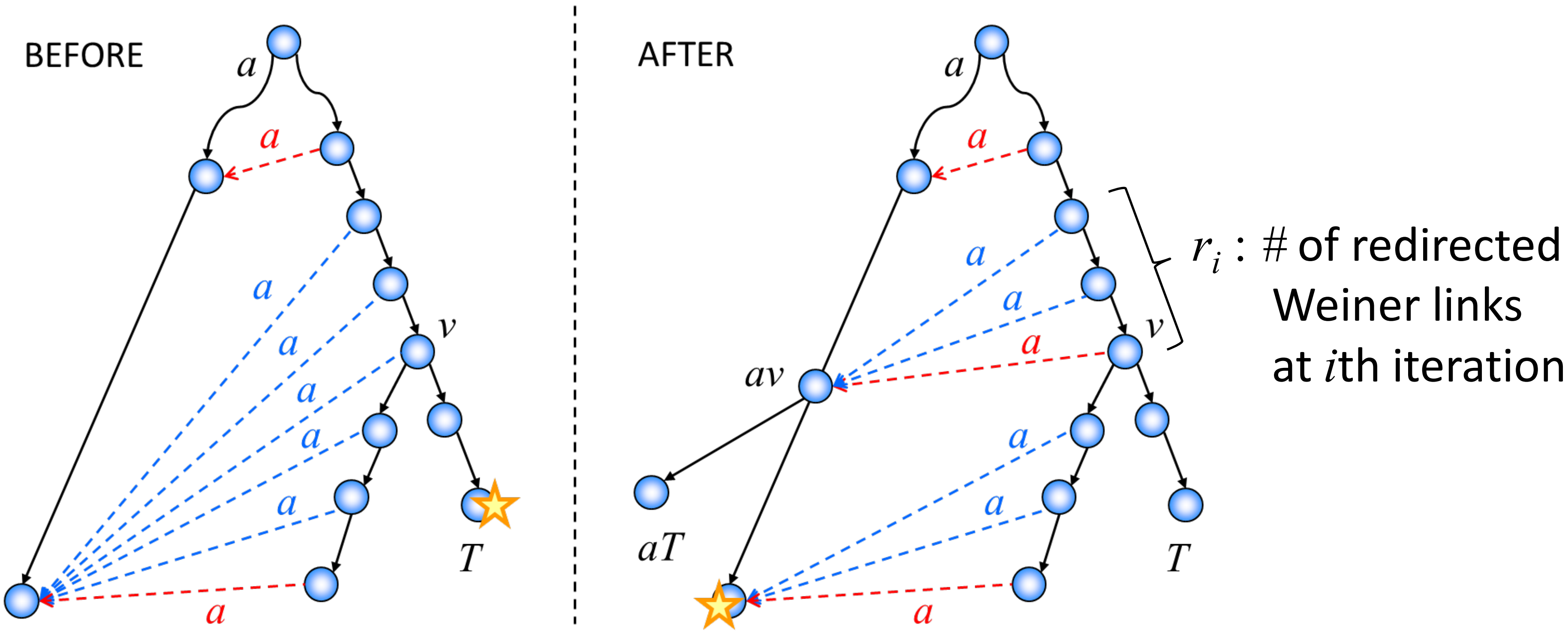


r_i : # of redirected Weiner links at ith iteration.

Online single string	: $\sum_{i=1}^{n} r_i \in O(n)$	[Blumer et al. 1985]
Fully-online multiple strings	: $\sum_{i=1}^{n} r_i \in \Omega(n^{1.5})$	[Takagi et al. 2020]

Weiner's Algorithm (Blumer et al. version)

Update string T to aT.



To avoid the $\Omega(n^{1.5})$ work, we reduce the sub-problem of redirecting Weiner links to the **ordered split-insert-find problem**.

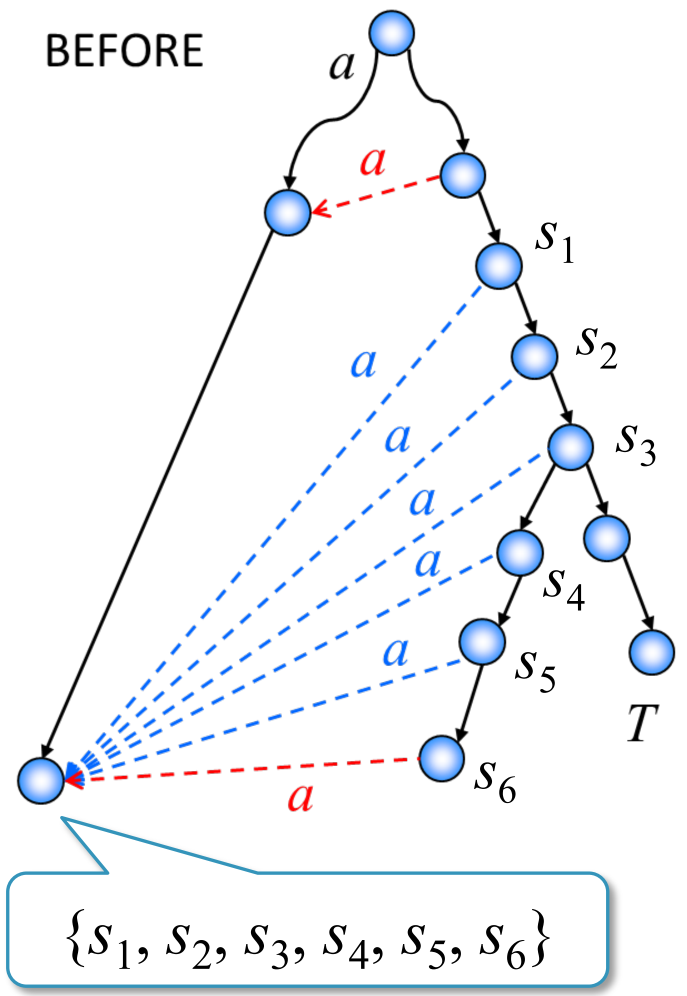
Ordered Split-Insert-Find

The **ordered split-insert-find** problem is to maintain a data structure on ordered sets which supports the following operations and queries efficiently:

- **Make-set**, which creates a new list that consists only of a single element;
- **Split**, which splits a given set into two disjoint sets, so that one set contains only smaller elements than the other set;
- **Insert**, which inserts a new single element to a given set;
- **Find**, which answers the name of the set that a given element belongs to.

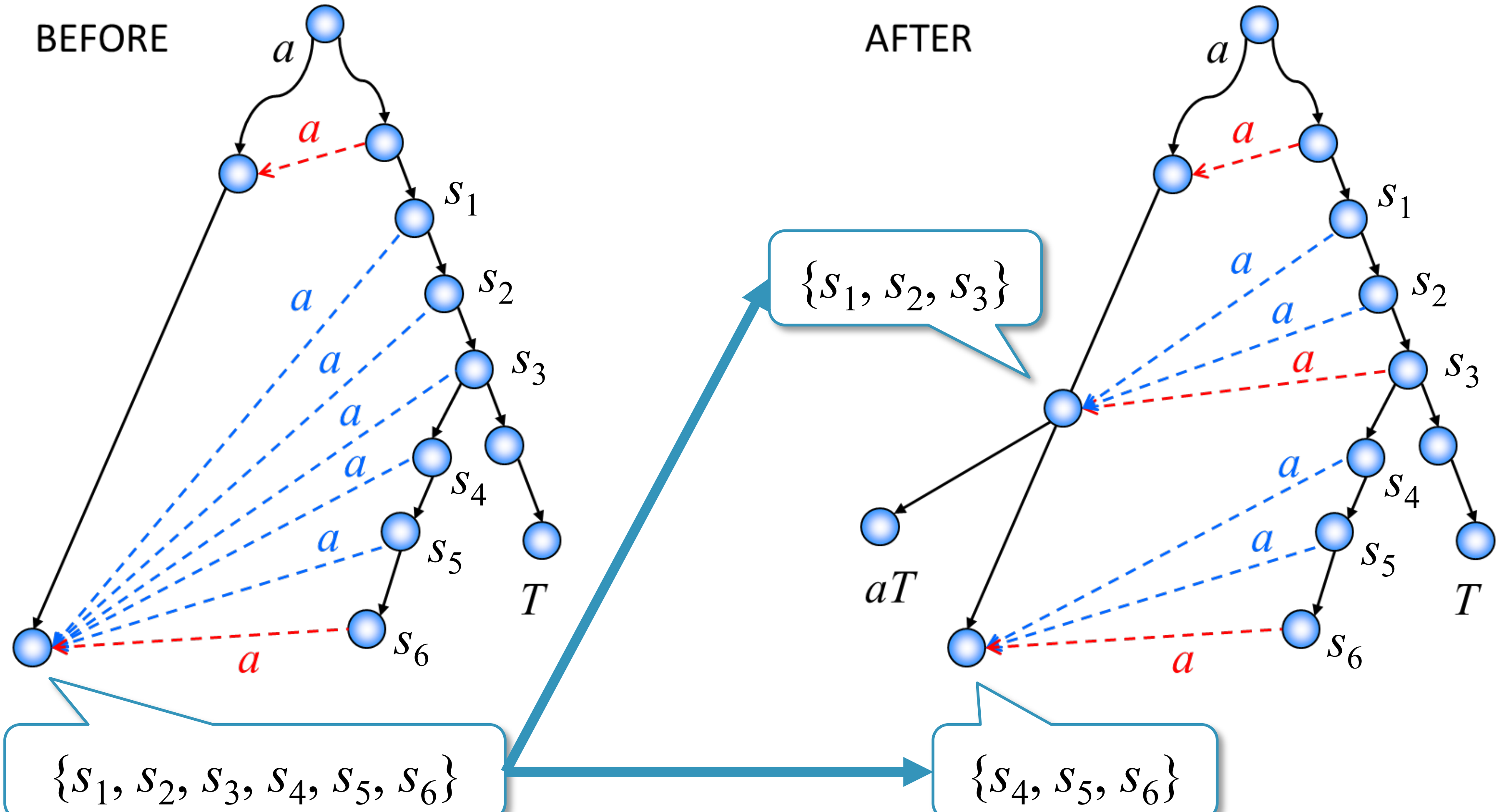
Reduction to Ordered Split-Insert-Find

Maintain the set of string depths of origin nodes of Weiner links

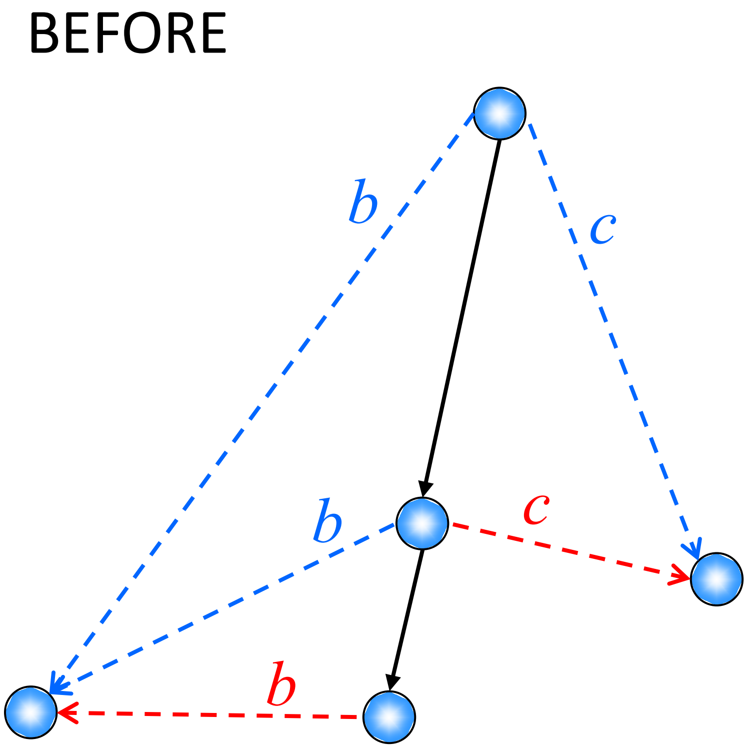


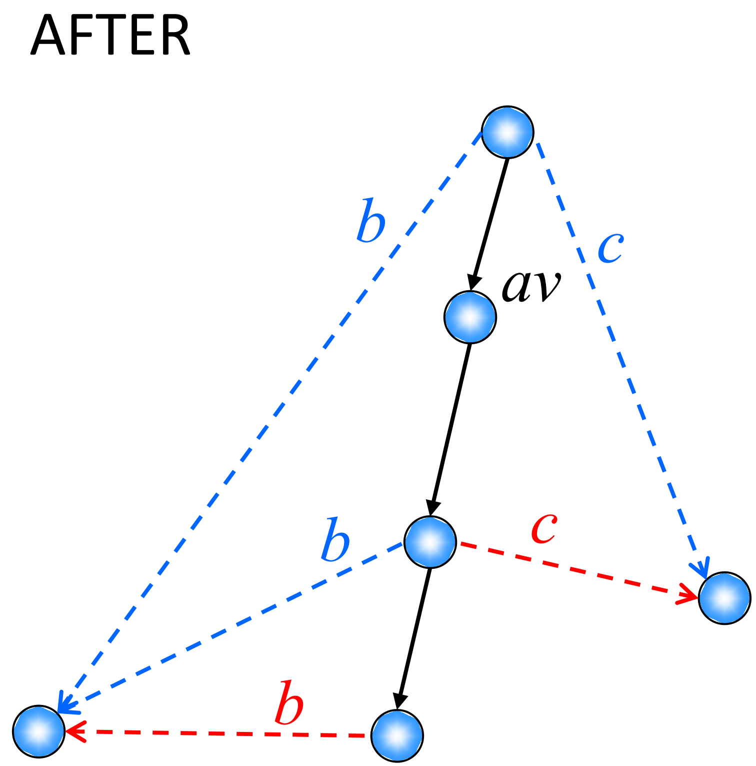
Reduction to Ordered Split-Insert-Find

Maintain the set of string depths of origin nodes of Weiner links



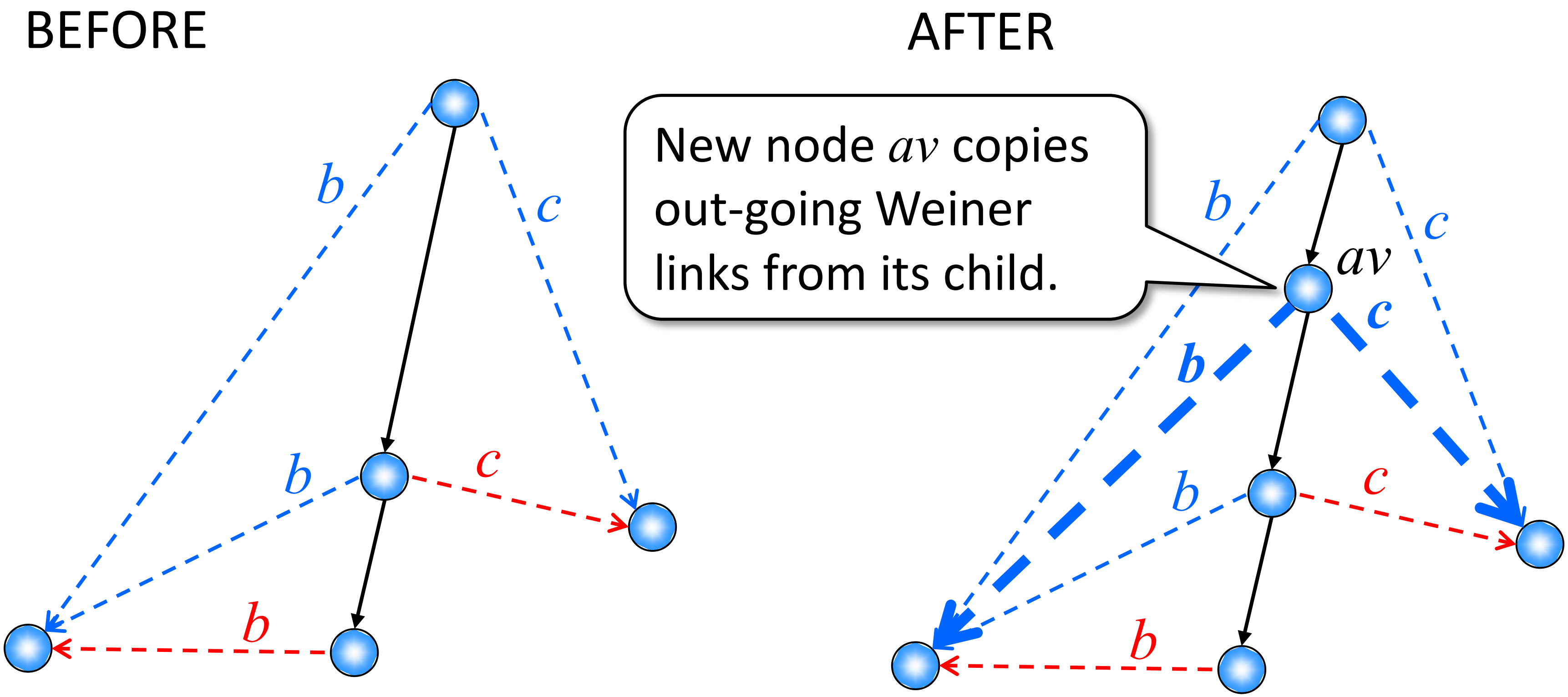
Reduction to Ordered Split-Insert-Find





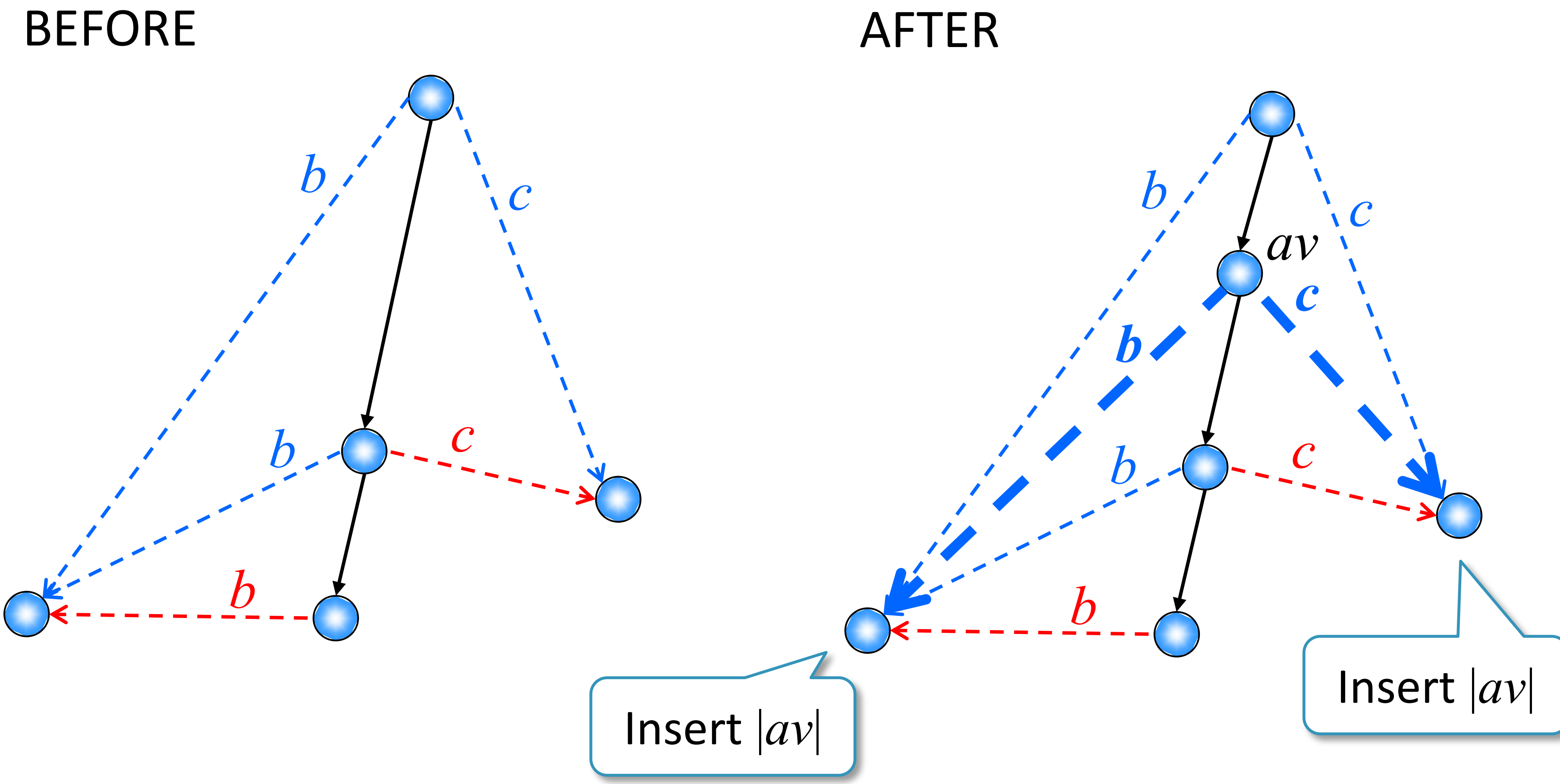
av is the parent of new leaf aT

Reduction to Ordered Split-Insert-Find



av is the parent of new leaf aT

Reduction to Ordered Split-Insert-Find



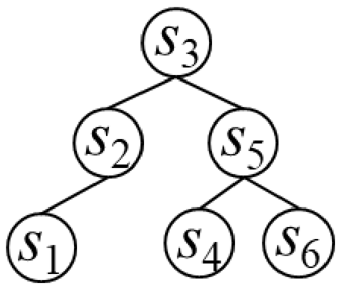
av is the parent of new leaf aT

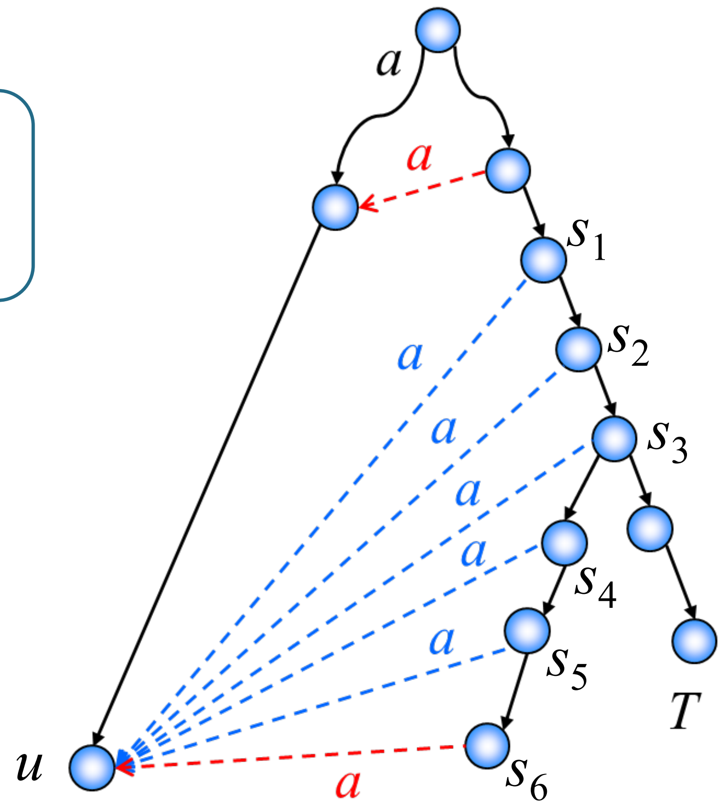
Ordered Split-Insert-Find by AVL-trees

For each suffix tree node u, we maintain an **AVL tree** such that each AVL tree node stores the string depth of the origin node of an in-coming Weiner link.

Works on the pointer machine.

AVL tree for node u

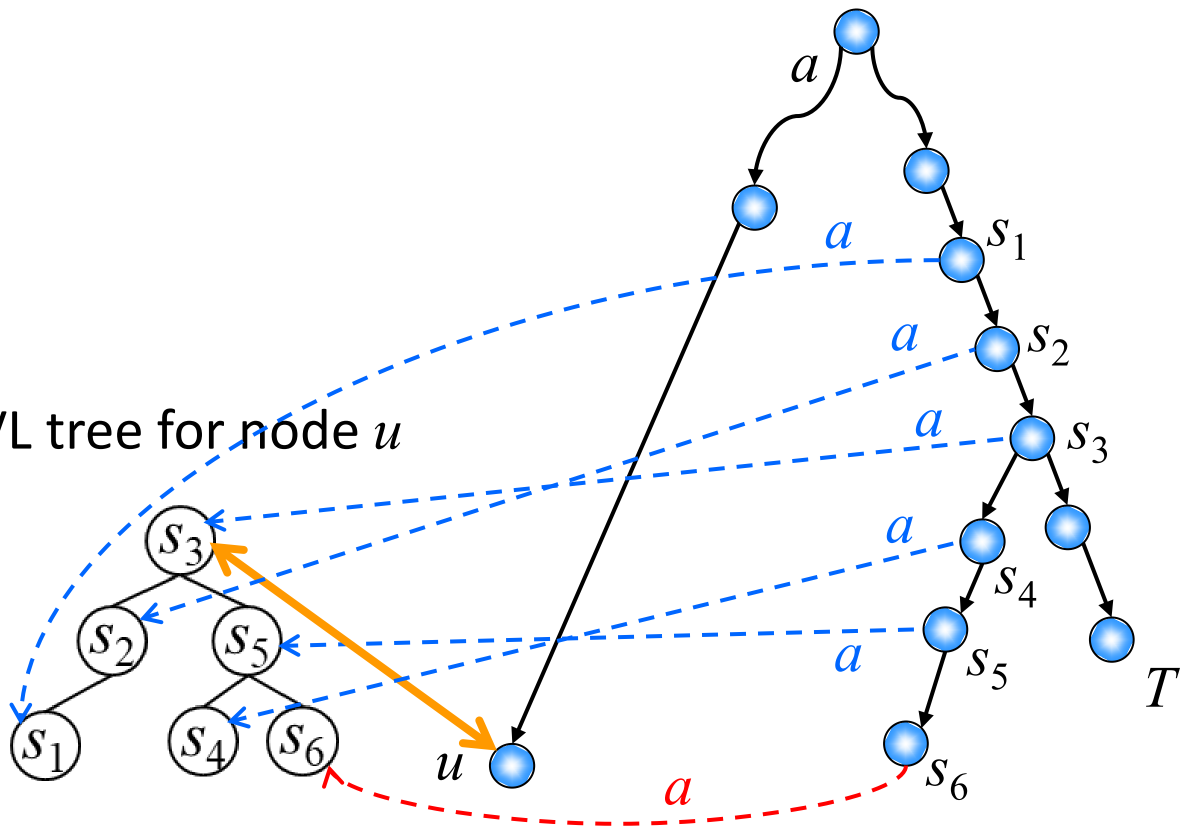




Ordered Split-Insert-Find by AVL-trees

Now, each Weiner link to a suffix tree node u points to the corresponding node in the AVL tree for node u.
The root of this AVL tree is connected to suffix tree node u.

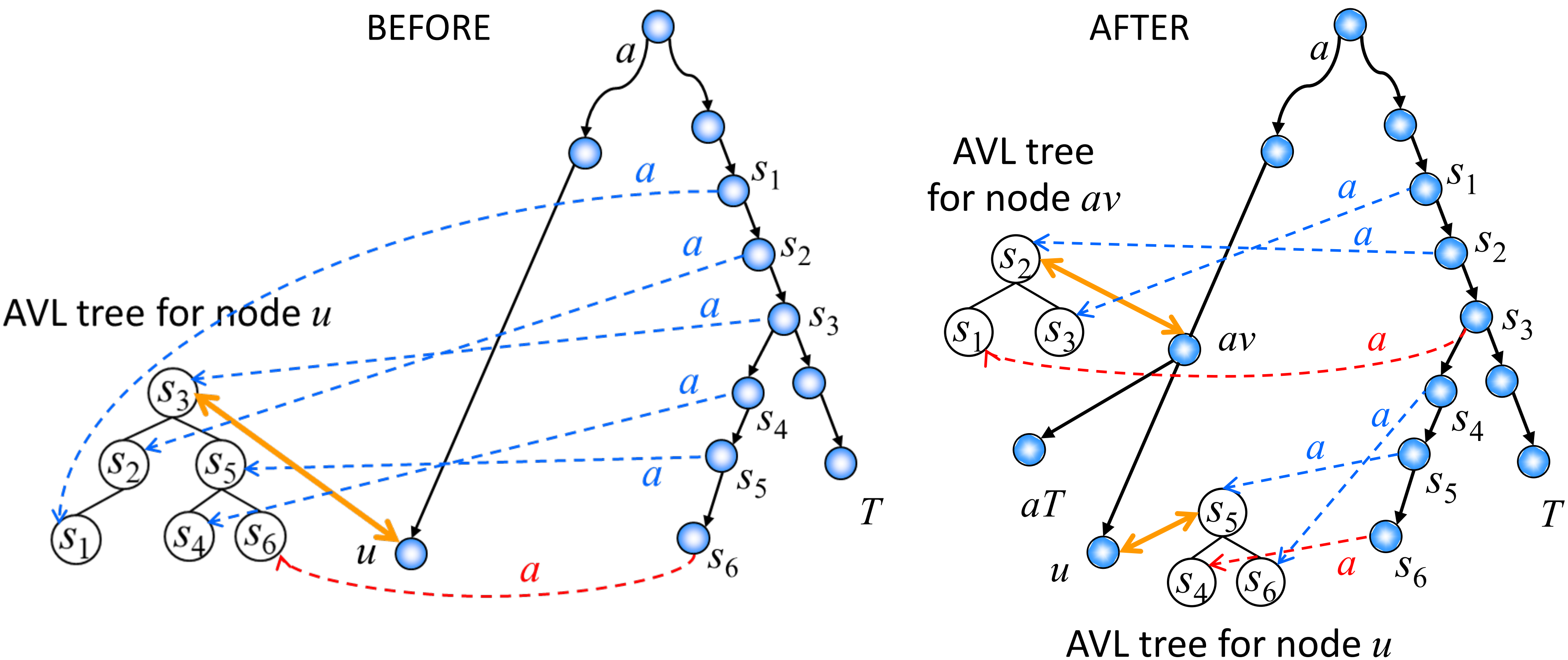
AVL tree for node u



Ordered Split-Insert-Find by AVL-trees

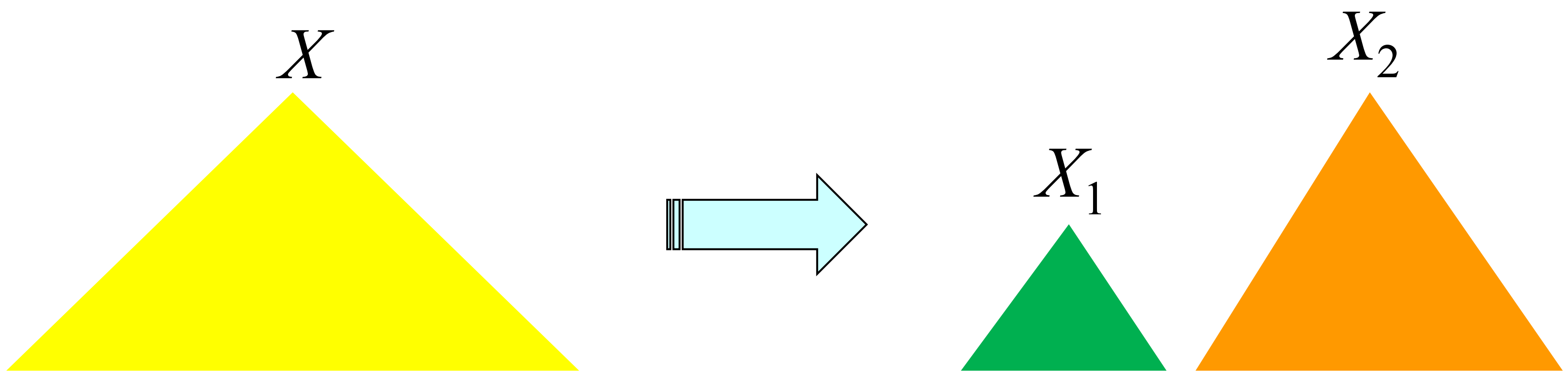
An AVL tree of d elements supports operations Make-set, Split, Inert, and Find in $O(\log d)$ time each.

➔ $O(\log d)$-time maintenance of Weiner links.



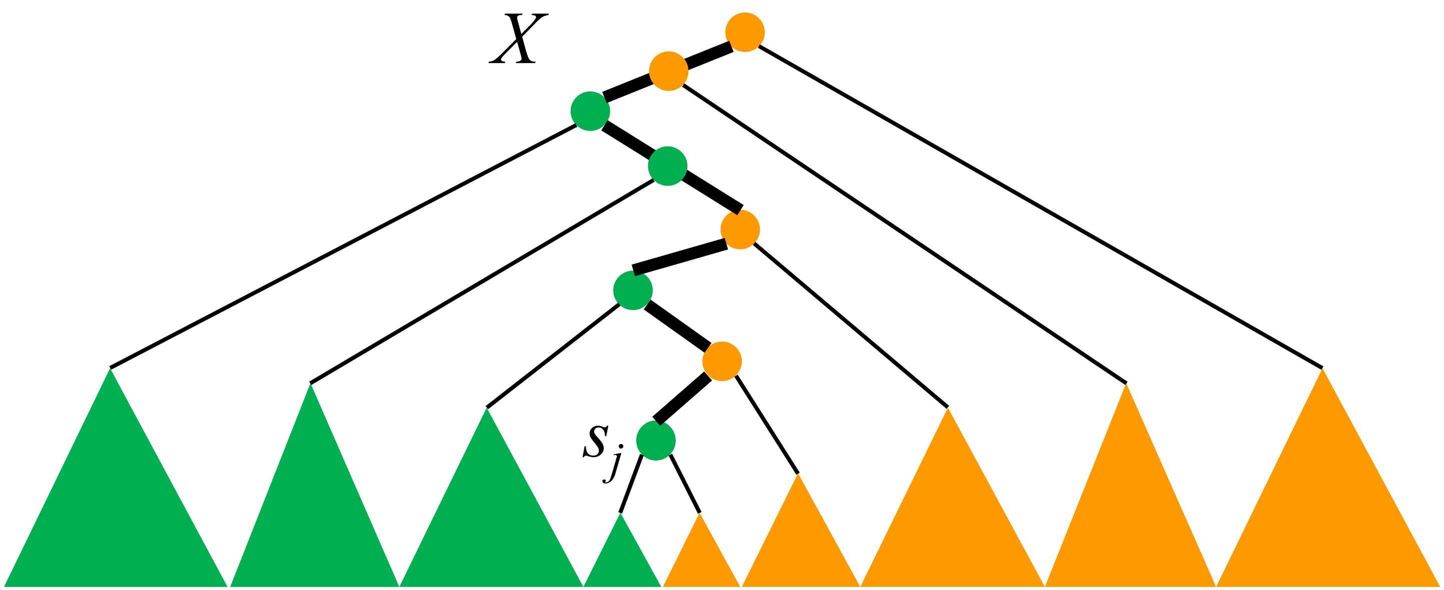
Splitting an AVL-tree

- Let X be an AVL tree for the set $\{s_1, \ldots, s_d\}$ of integers.
- Given an element s_j in the set, we split the AVL tree X into two AVL trees, X_1 for $\{s_1, \ldots, s_j\}$ and X_2 for $\{s_{j+1}, \ldots, s_d\}$.
- This split operation can be done in $O(\log d)$ time (next slide).



Splitting an AVL-tree

- Consider the search path for s_j in the AVL tree X.
- By splitting X with this path, we obtain
  - green nodes and subtrees containing elements at most s_j
  - orange nodes and subtrees containing elements larger than s_j
- Using monotonicity, we can merge each of them in $O(\log d)$ time.



Main Results

Theorem 1

There is a pointer-machine algorithm which builds the **suffix tree** of **right-to-left** fully-online multiple strings in $O(n (\log \sigma + \log d))$ time and $O(n)$ space.
Each suffix-tree edge traversal takes $O(\log \sigma)$ time.

Theorem 2

There is a pointer-machine algorithm which builds the **DAWG** of **left-to-right** fully-online multiple strings in $O(n (\log \sigma + \log d))$ time and $O(n)$ space.
Each DAWG-edge traversal takes $O(\log \sigma + \log d)$ time.

n : total string length , σ : alphabet size, d : max. # in-coming Weiner links

Conclusions and Open Question

We proposed pointer-machine algorithms for fully-online construction of suffix trees and DAWGs on multiple strings running in $O(n\ (\log \sigma + \log d))$ time and $O(n)$ space.

We have not found an instance where the $n \log d$ term in our time complexity becomes $\Theta(n \log n)$ or $\omega(n)$.

We have only found a bad instance which requires sub-linear $\Omega(\sqrt{n} \log n)$ work to maintain the AVL trees.

Would it be possible to construct suffix trees / DAWGs for fully-online multiple strings in $O(n \log \sigma)$ time on the pointer machine?