Sparse Compact Directed Acyclic Word Graphs

Shunsuke Inenaga

(Japan Society for the Promotion of Science
& Kyushu University)

Masayuki Takeda

(Kyushu University
& Japan Science and Technology Agency)

Traditional Pattern Matching Problem

- Given : text ***T*** in Σ^* and pattern ***P*** in Σ^*
- Return : whether or not ***P*** appears in ***T***

  - Σ: *alphabet* (set of *characters*)
  - Σ^*: set of *strings*

- A text indexing structure for ***T*** enables you to solve the above problem in $O(m)$ time (for fixed Σ).

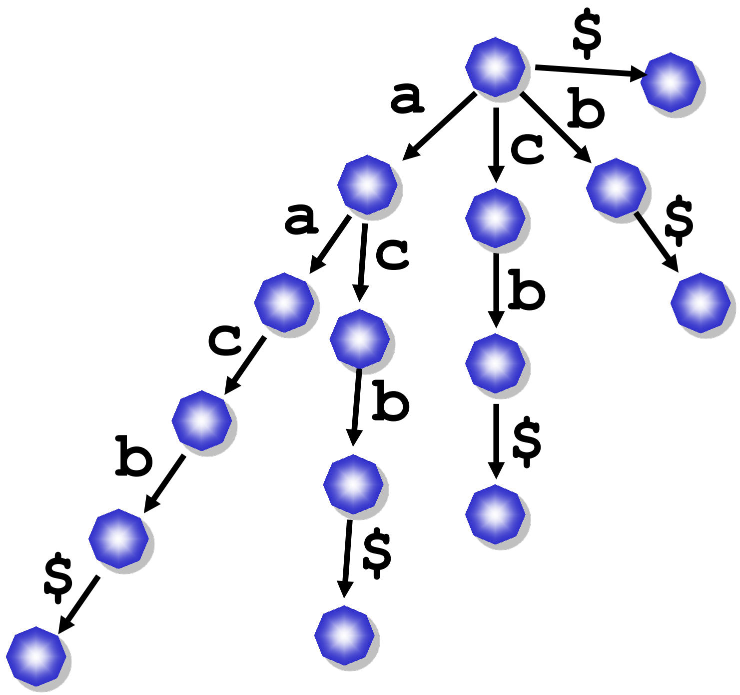
  - *m*: the length of ***P***

Suffix Trie

- A trie representing **all** suffixes of ***T***

T = `aacb$`

`aacb$`
`acb$`
`cb$`
`b$`
`$`



Introducing Word Separator #

- # : word separator - special symbol *not* in Σ
- ***D*** = Σ^* # : dictionary of *words*

- Text ***T*** : an element of $\boldsymbol{D}^+$
(***T*** is a sequence $\boldsymbol{T}_1\boldsymbol{T}_2\ldots\boldsymbol{T}_k$ of *k* words in ***D***)

- e.g., ***T*** = This#is#a#pen#

  - Σ = {A,…,z}
  - ***D*** = {...,This#,...,a#,...is#,...pen#,...}

Word-level Pattern Matching Problem

- Given: text ***T*** in $\boldsymbol{D}^+$ and pattern ***P*** in $\boldsymbol{D}^+$
- Return: whether or not ***P*** appears at the beginning of any word in ***T***

e.g.

T = The#space#runner#is#not#your#good#pace#runner#

P = pace#runner#

Word-level Pattern Matching Problem

- Given: text ***T*** in ***D***$^+$ and pattern ***P*** in ***D***$^+$
- Return: whether or not ***P*** appears at the beginning of any word in ***T***

e.g.

✕ ○

T = The#space#runner#is#not#your#good#**pace#runner#**

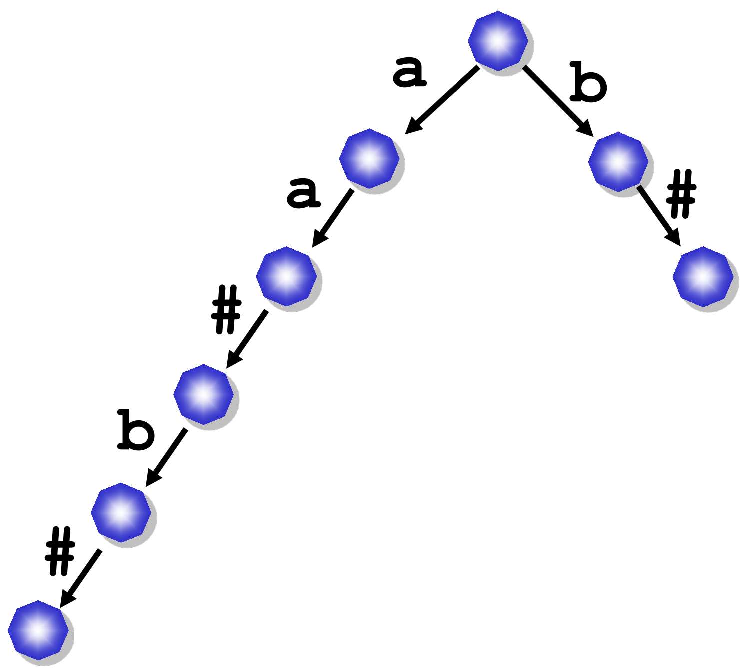
P = pace#runner#

Word Suffix Trie

- A trie representing the suffixes of ***T*** which begin at a word.

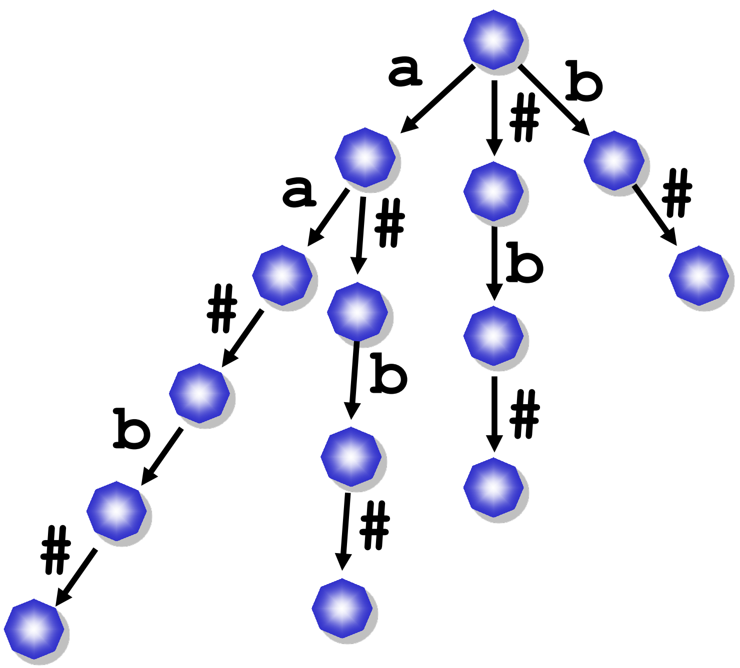
T = **`aa#b#`**

`aa#b#`
`a#b#`
`#b#`
`b#`
`#`

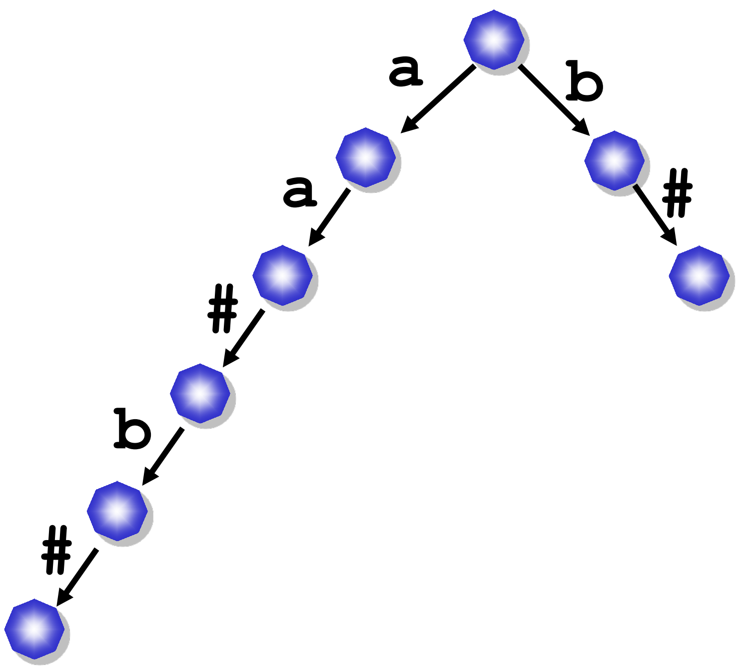


Normal and Word Suffix Tries

T **= `aa#b#`**



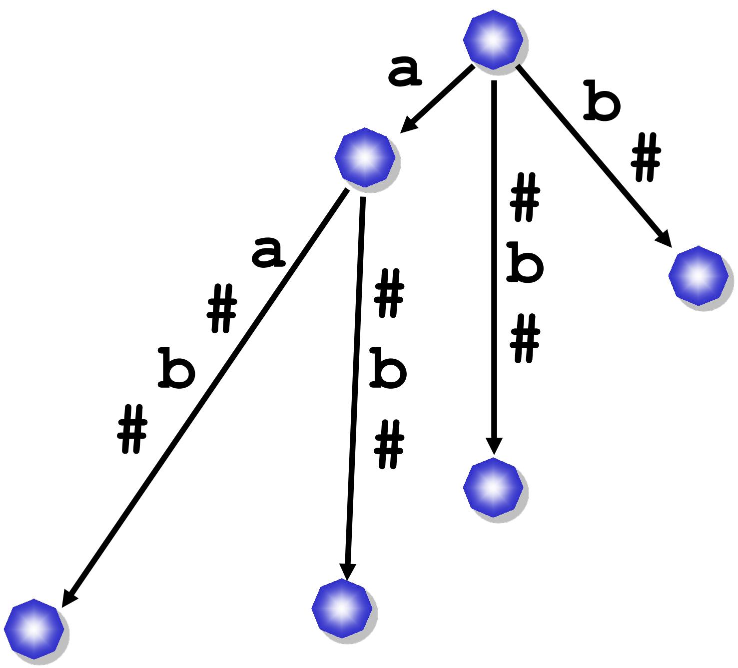
Normal Suffix Trie



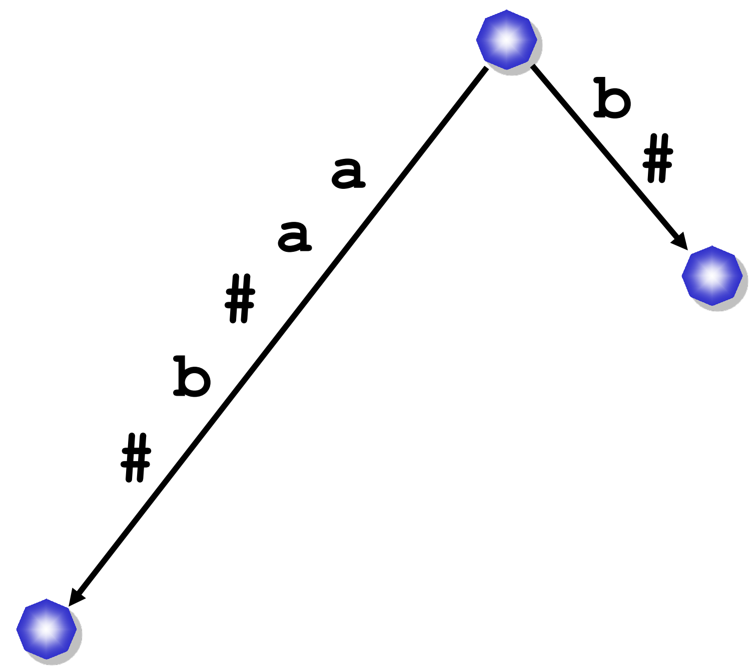
Word Suffix Trie

Normal and Word Suffix Trees

T **= aa#b#**



Normal Suffix Tree



Word Suffix Tree

Sizes of Word Suffix Tries and Trees

- For text $\boldsymbol{T} = \boldsymbol{T}_1\boldsymbol{T}_2\ldots\boldsymbol{T}_k$ of length n,
  - the word suffix trie of $\boldsymbol{T}$ requires $O(nk)$ space, but
  - the word suffix tree of $\boldsymbol{T}$ requires $O(k)$ space!!
    - because the word suffix tree has only k leaves and has only branching internal nodes.

Construction of Word Suffix Trees

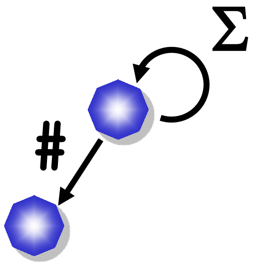
- Algorithm by Andersson et al.（1996）
  - for text $\boldsymbol{T} = \boldsymbol{T_1 T_2 \ldots T_k}$ of length n, constructs word suffix trees in $O(n)$ expected time with $O(k)$ space.
- Our algorithm (CPM’06)
  - builds word suffix trees in $O(n)$ time in the worst case, with $O(k)$ space.

Our Construction Algorithm

- We modify Ukkonen's on-line normal suffix tree construction algorithm by using *minimum DFA accepting dictionary **D***
  - We replace the root node of the suffix tree with the final state of the DFA.

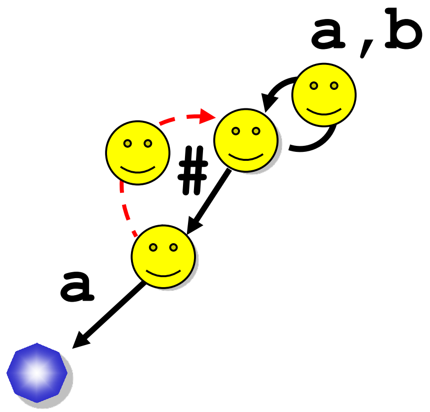
Minimum DFA

- The minimum DFA accepting $\boldsymbol{D} = \Sigma^* \#$ clearly requires constant space (for fixed Σ).



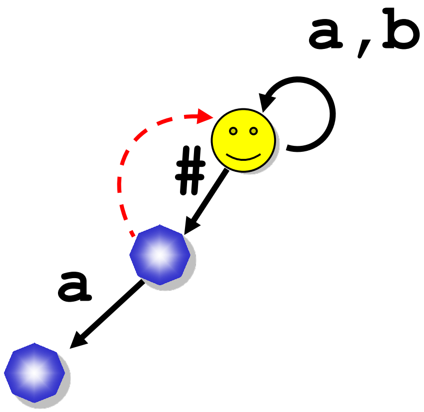
On-line Construction of Word Suffix Trees

T = **aa#b#**



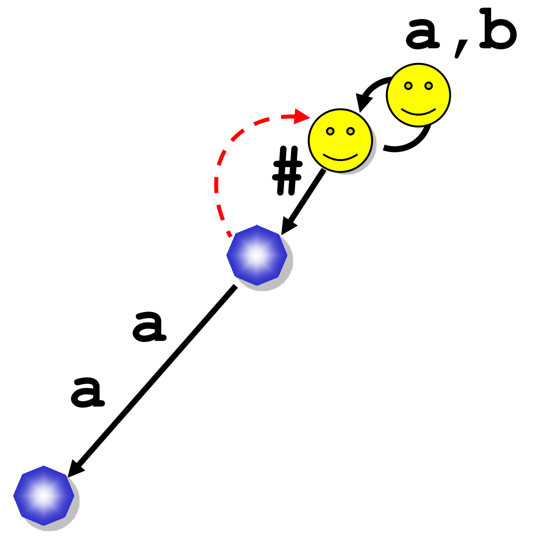
On-line Construction of Word Suffix Trees

T = **aa#b#**



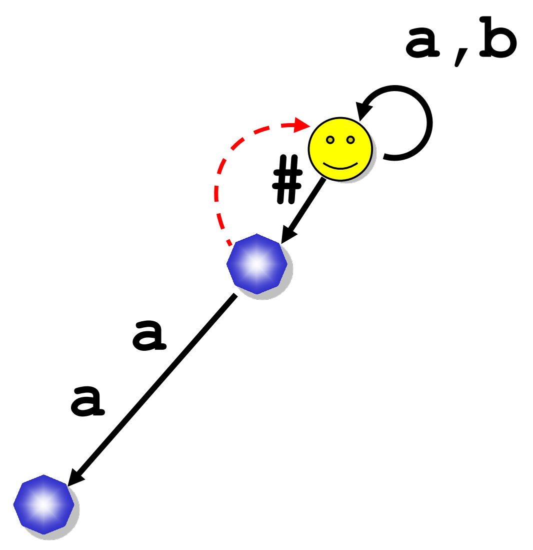
On-line Construction of Word Suffix Trees

T = aa#b#



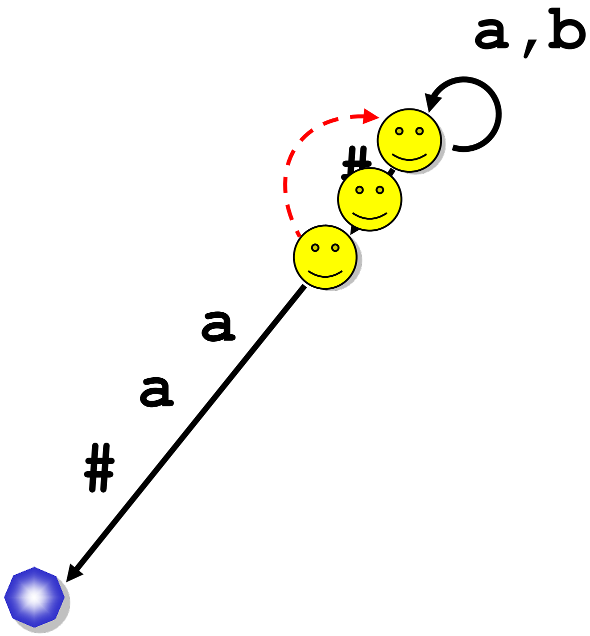
On-line Construction of Word Suffix Trees

T = **aa#b#**



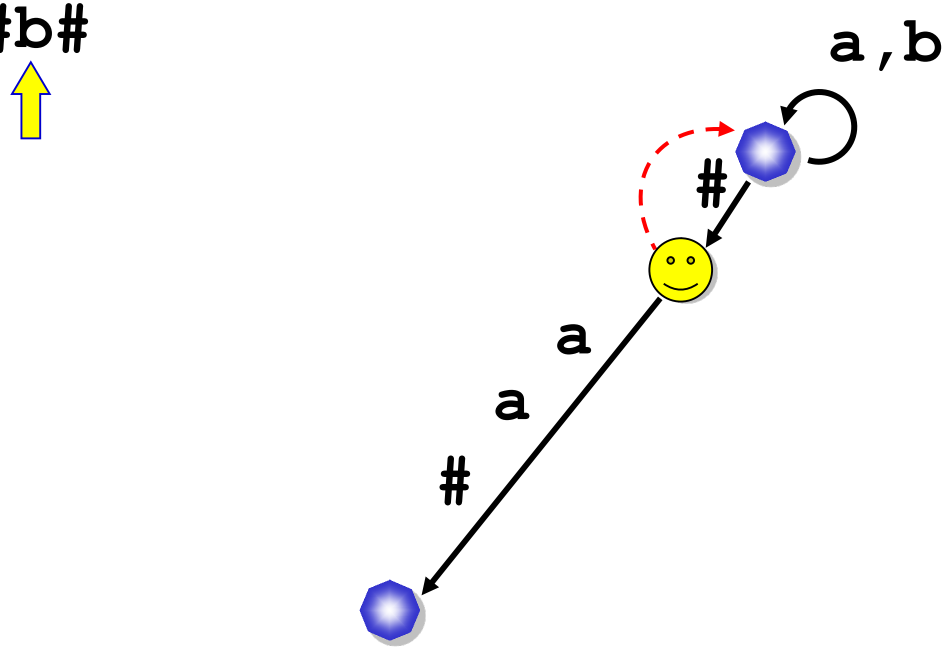
On-line Construction of Word Suffix Trees

T = aa#b#



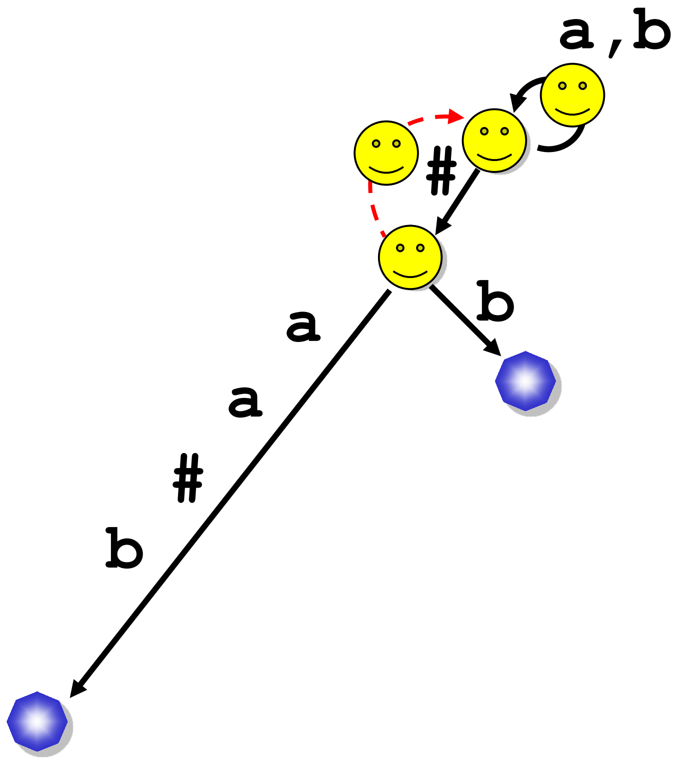
On-line Construction of Word Suffix Trees

T = aa#b#



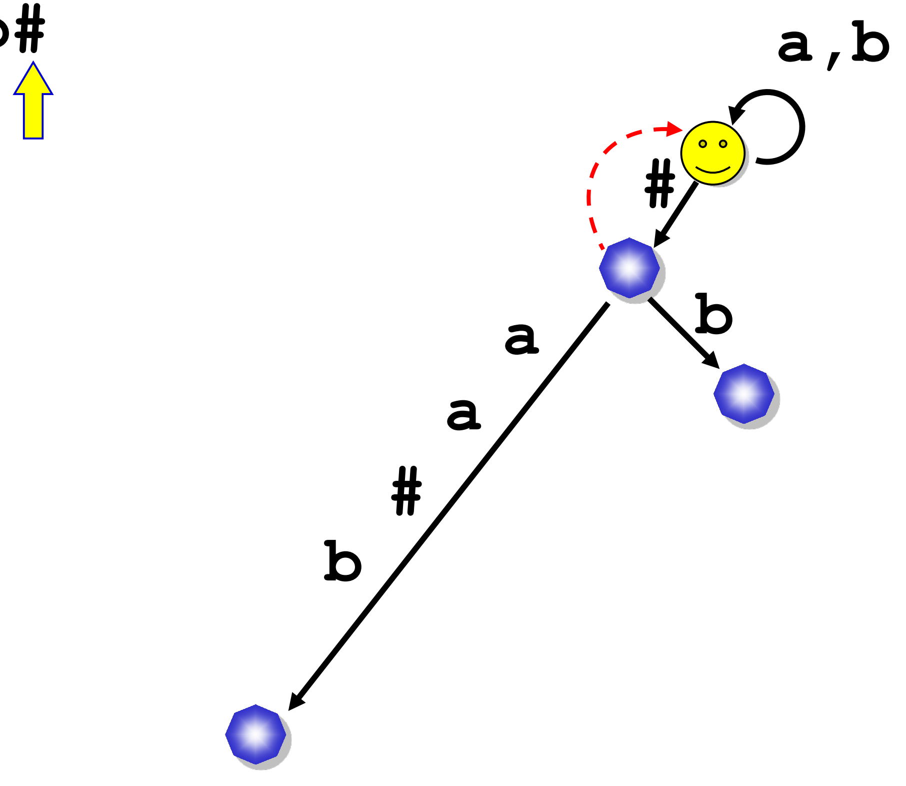
On-line Construction of Word Suffix Trees

T **= aa#b#**



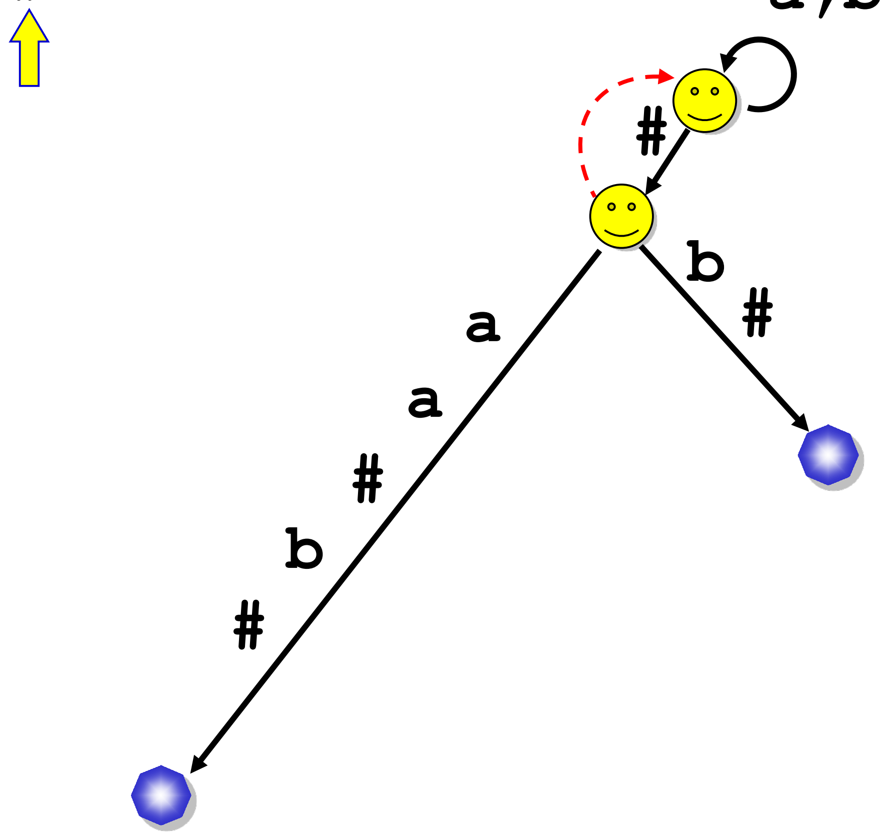
On-line Construction of Word Suffix Trees

T = aa#b#



On-line Construction of Word Suffix Trees

T = aa#b#



Pseudo-Code

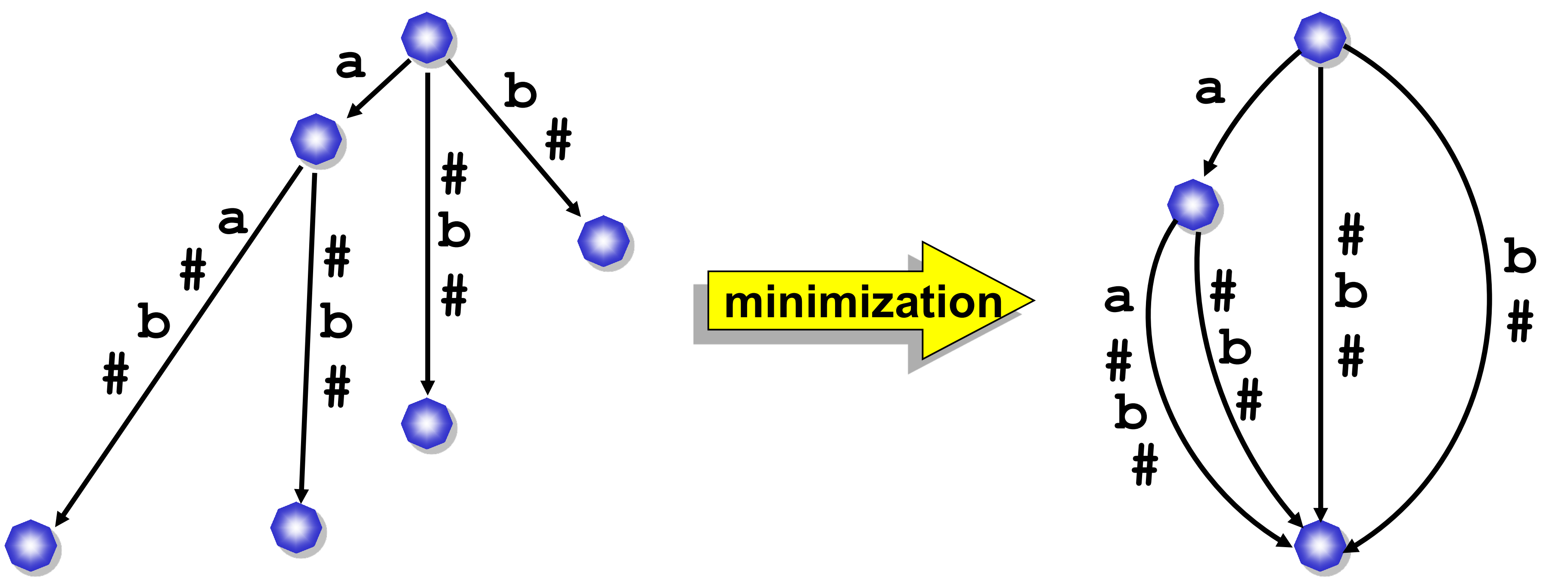
Just change here

```
Input:   w = w[1..n] ∈ D+ and auxiliary DFA M_D.
Output:  Word suffix tree of w[1..n] w.r.t. D.
{
    /* We assume Σ = {w[−1], w[−2], . . . , w[−m]} */.
    /* Replace the edge labels of M_D with appropriate integer pairs */.
    root = the final state of M_D;
    Suf(root) = the initial state of M_D;
    (s, k) = (root, 1);
    for (i = 1; i ≤ n; i++) {
        oldr = nil;
        while (CheckEndPoint(s, (k, i − 1), w[i]) == false) {
            if (k ≤ i − 1)    /* (s, (k, i − 1)) is implicit. */
                r = SplitEdge(s, (k, i − 1));
            else              /* (s, (k, i − 1)) is explicit. */
                r = s;
            t = CreateNewNode();
            create a new edge r --(i,∞)--> t;
            if (oldr ≠ nil) Suf(oldr) = r;
            oldr = r;
            (s, k) = Canonize(Suf(s), (k, i − 1));
        }
        if (oldr ≠ nil) Suf(oldr) = s;
        (s, k) = Canonize(s, (k, i));
    }
}
```

```
boolean CheckEndPoint(s, (k, p), c) {
    if (k ≤ p) {    /* (s, (k, p)) is implicit. */
        let s --(k′,p′)--> s′ be the w[k]-edge from s;
        return (c == w[k′ + p − k + 1]);
    } else return (there is a c-edge from s);
}

(node,integer)-pair Canonize(s, (k, p)) {
    if (k > p) return (s, k);    /* (s, (k, p)) is explicit. */
    find the w[k]-edge s --(k′,p′)--> s′ from s;
    while (p′ − k′ ≤ p − k) {
        k += p′ − k′ + 1; s = s′;
        if (k ≤ p) find the w[k]-edge s --(k′,p′)--> s′ from s;
    }
    return (s, k);
}

node SplitEdge(s, (k, p)) {
    let s --(k′,p′)--> s′ be the w[k]-edge from s;
    r = CreateNewNode();
    replace this edge by edges s --(k′,k′+p−k)--> r and r --(k′+p−k+1,p′)--> s′;
    return r;
}
```

Compact Directed Acyclic Word Graphs

T **= aa#b#**

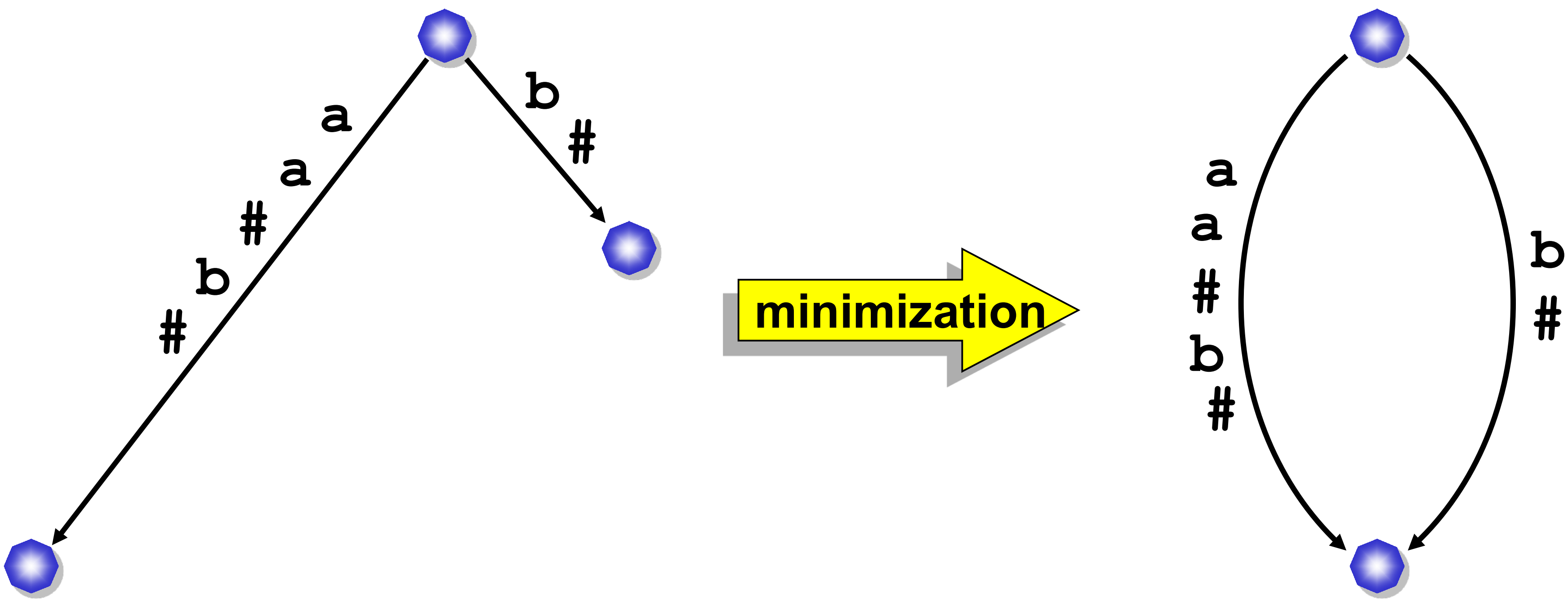


Suffix Tree

Compact Directed Acyclic Word Graph (CDAWG)

Sparse CDAWGs

T **=** **aa#b#**

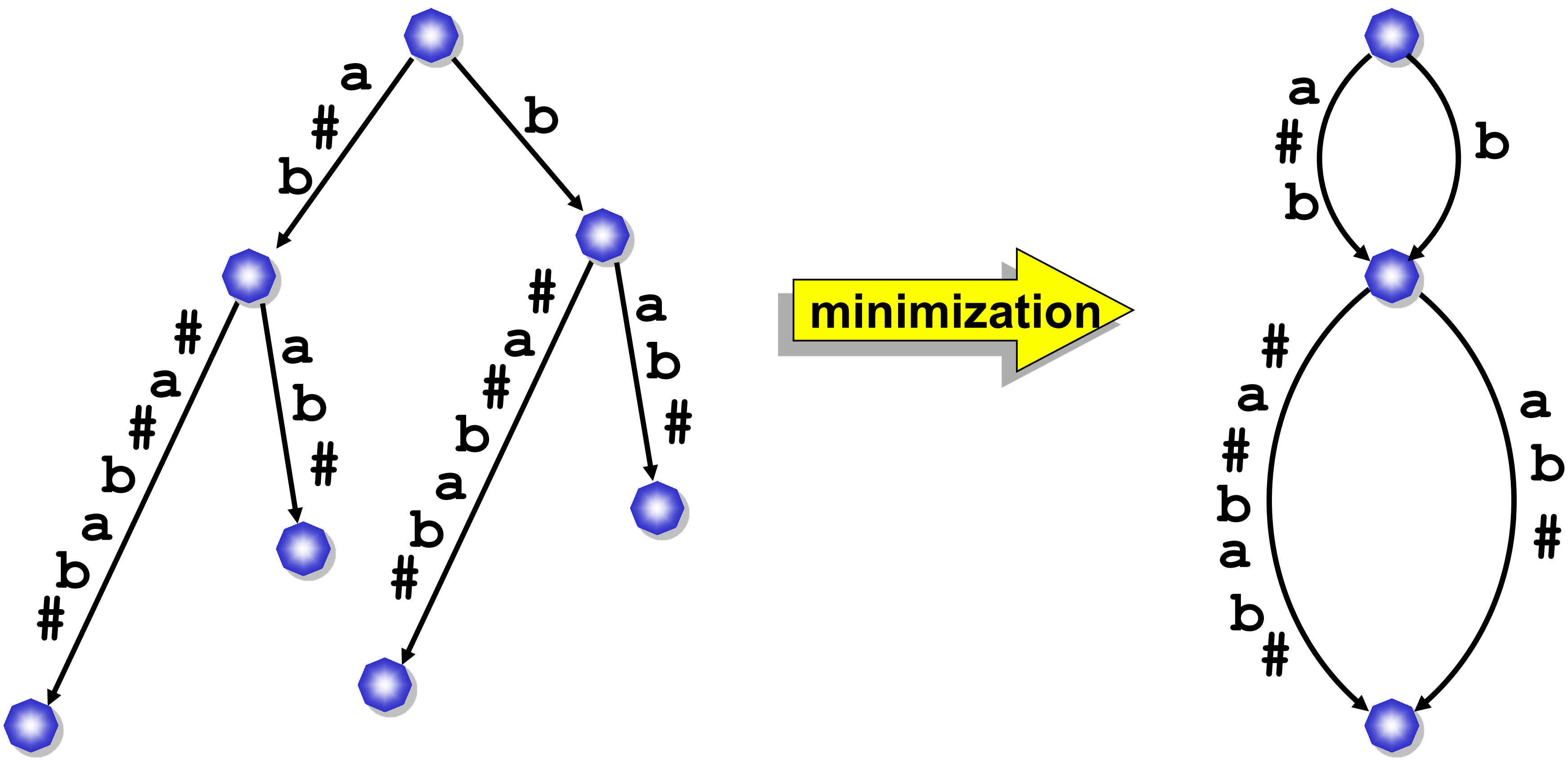


Word Suffix Tree

Sparse Compact Directed Acyclic Word Graph (SCDAWG)

Sparse CDAWGs [cont.]

T **= `a#b#a#bab#`**



Word Suffix Tree *SCDAWG*

SCDAWG Construction

- SCDAWGs can be constructed by minimizing word suffix trees in $O(k)$ time.
  - using Revuz's DAG minimization algorithm (1992)

SCDAWG Construction [cont.]

- Question : Direct construction for SCDAWGs?
- Answer : **YES!**
  Using minimal DFA accepting dictionary ***D***,
  we can directly build SCDAWGs in $O(n)$ time and $O(k)$ space.
  - We modify the CDAWG on-line construction algorithm (Inenaga et al. 05) by using the above DFA.

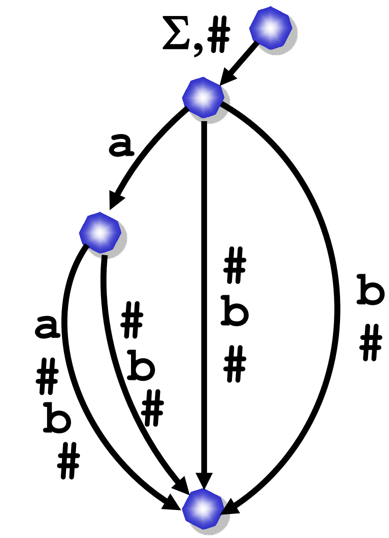
Pseudo-Code

Just change here

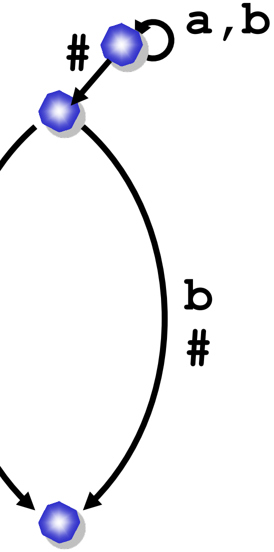
```
Input:    w = w[1..n] ∈ D+ and M_D with initial state q_s and final state q_f.
Output:   SCDAWG_D(w).
{
    /* We assume Σ = {w[−1], w[−2], ..., w[−m]} */.
    /* Replace the edge labels of M_D with appropriate integer pairs */.
    length(q_f) = 0;   length(q_s) = −1;   length(sink) = ∞;
    source = q_f;   link(source) = q_s;   link(sink) = nil;
    (s, k) = (source, 1);
    for (i = 1; i ≤ n; i++)  (s, k) = Update(s, (k, i));
}

(node,integer)-pair Update(s, (k, i)) {
    oldr = nil; s' = nil;
    while (CheckEndPoint(s, (k, i − 1), w[i]) == false) {
        if (k ≤ i − 1) {  /* (s, (k, i − 1)) is implicit. */
            if (s' == Extension(s, (k, p − 1))) {
                let (s, (k', p'), s') be the w[k]-edge from s;
                replace the edge by edge (s, (k', k' + p − k − 1), r);
                (s, k) = Canonize(link(s), (k, p − 1));
                continue;
            }
            else {
                s' = Extension(s, (k, p − 1));
                r = CreateNode(s, (k, p − 1));
            }
        } else r = s;     /* (s, (k, i − 1)) is explicit. */
        create new edge (r, (i, ∞), sink);
        if (oldr ≠ nil) link(oldr) = r;
        oldr = r;
        (s, k) = Canonize(link(s), (k, i − 1));
    }
    if (oldr ≠ nil) link(oldr) = s;
    return SplitNode(s, (k, i));
}
```

Body is the same!!

```
boolean Che
    if (k ≤ p
        let (
        retu
    } else re
}

node Extens
    if (k > p
    find the
    return
}

(node,integ
    if (k > p
    find the
    while (p
        k =
        s =
        if (k
    }
    return (
}

node Create
    let (s, (k
    create ne
    replace t
    length(r)
    return r
}

(node,integ
    (s', k') =
    if (k' ≤ p
    /* (s', (k
    if (length
    create no
    link(r') =
    length(r'
    do {
        repla
        (s, k
    } while (
    return (r'
}
```



Different structures



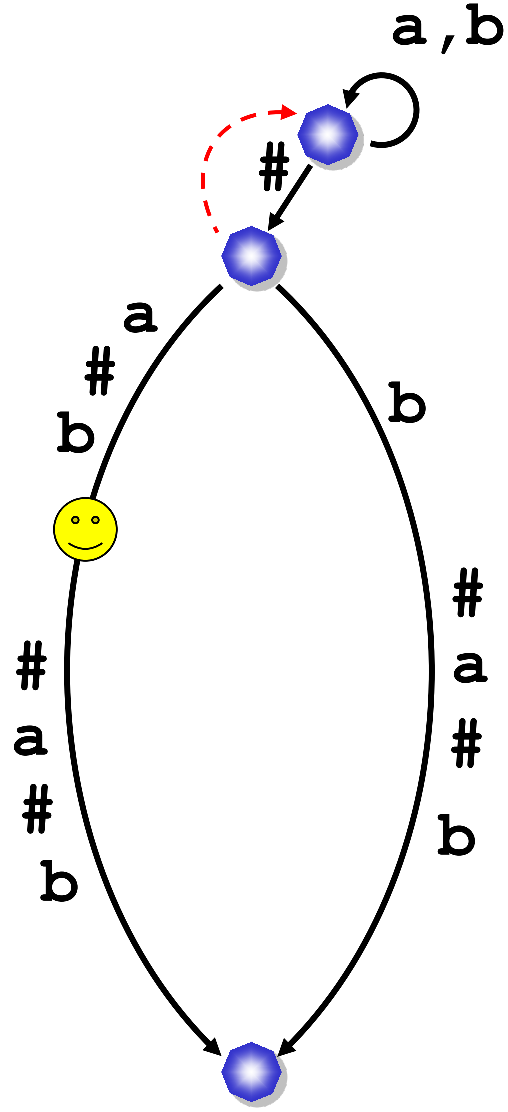
Some Events

- Basically on-line construction of SCDAWGs is similar to that of word suffix trees.

- Except for the two following unique events:
  - *Edge merging*
  - *Node splitting*

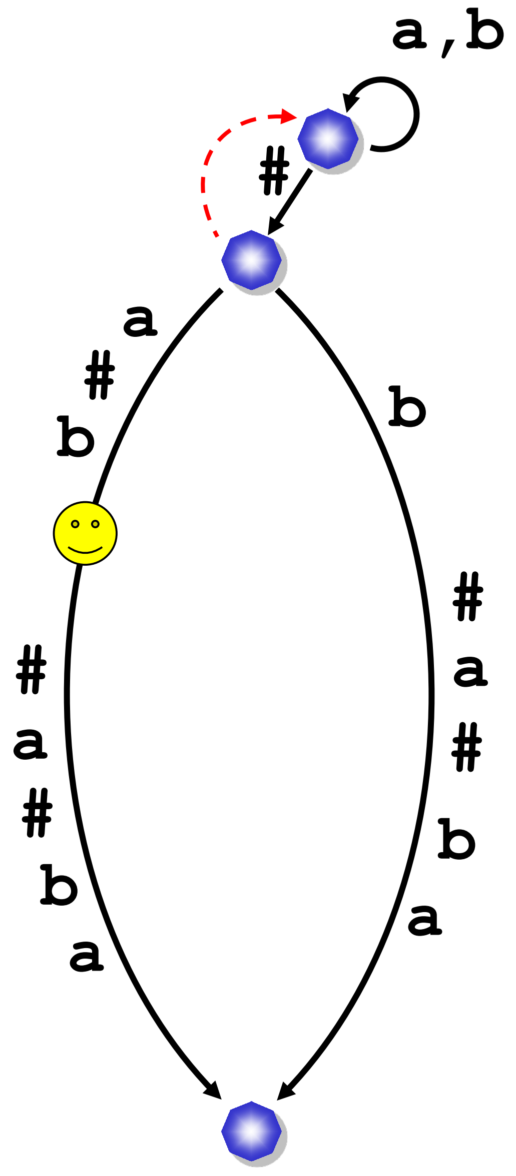
Edge Merging

T = a#b#a#bab#bc...



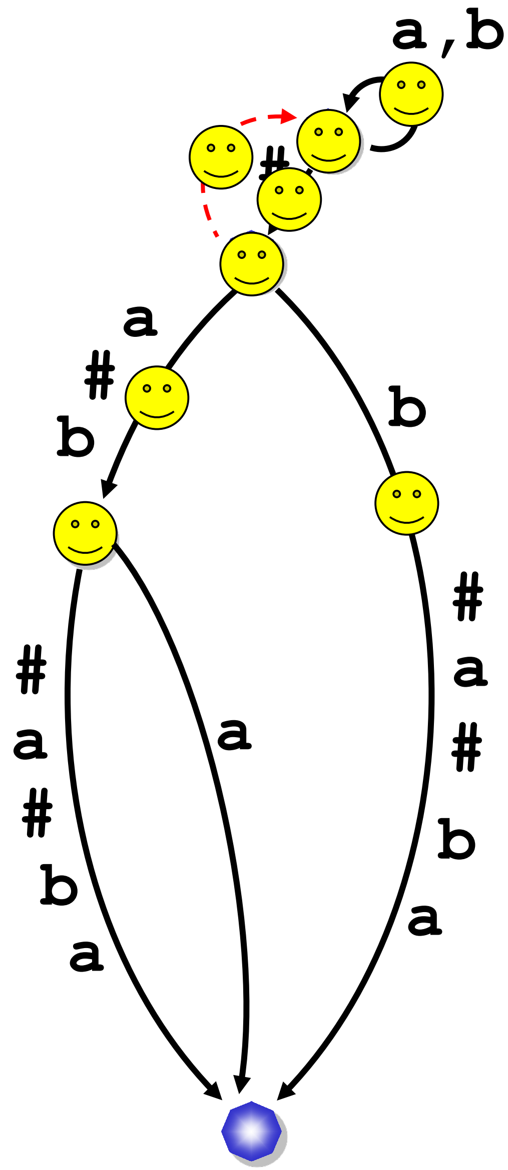
Edge Merging

T = a#b#a#bab#bc...



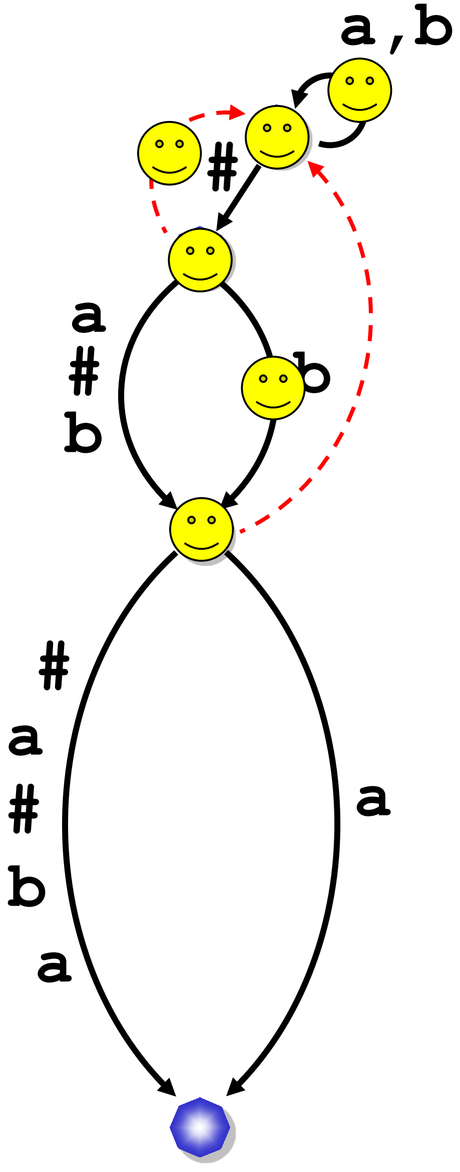
Edge Merging

T = a#b#a#bab#bc...



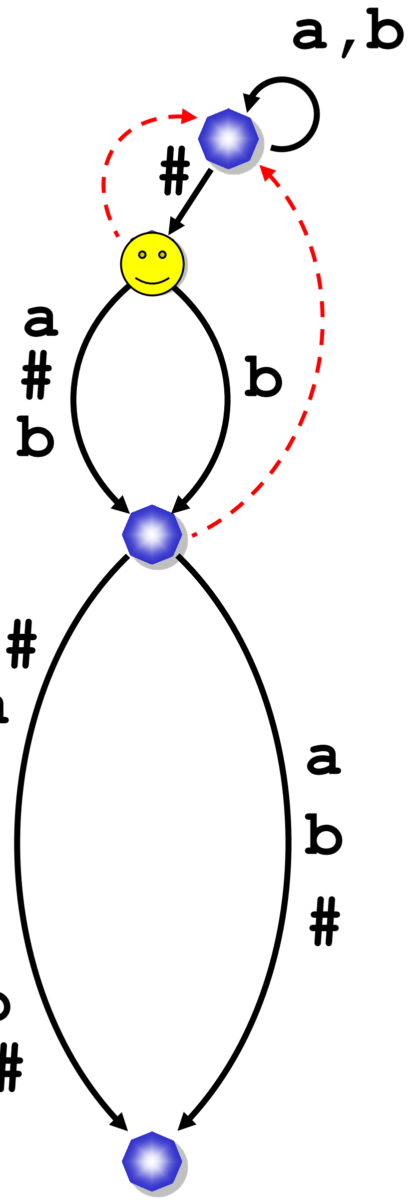
Edge Merging

T = a#b#a#bab#bc...



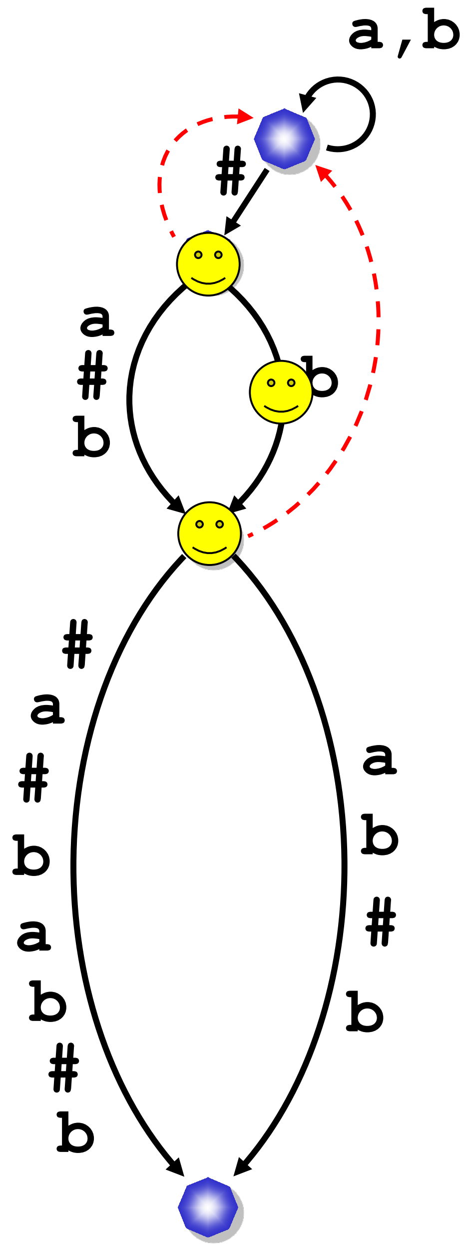
Node Splitting

T = a#b#a#bab#bc...



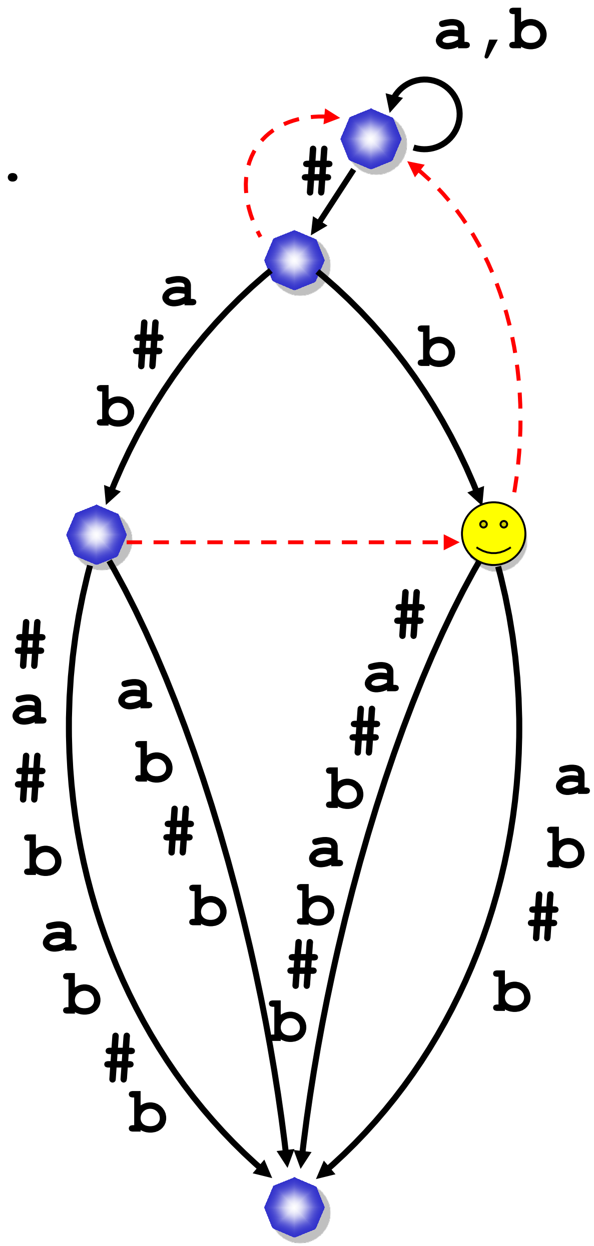
Node Splitting

T **= a#b#a#bab#bc...**



Node Splitting

T = `a#b#a#bab#bc...`



Conclusion

- We introduced new text indexing structure ***sparse compact directed acyclic word graphs*** (***SCDAWGs***) for word-level pattern matching.

- We presented an on-line algorithm to construct SCDAWGs directly, in $O(n)$ time with $O(k)$ space.

    - The key is the use of minimum DFA accepting dictionary $\boldsymbol{D}$.

Related Work

- **“Sparse Directed Acyclic Word Graphs”**
  by *Shunsuke Inenaga and Masayuki Takeda*
  Accepted to SPIRE’06