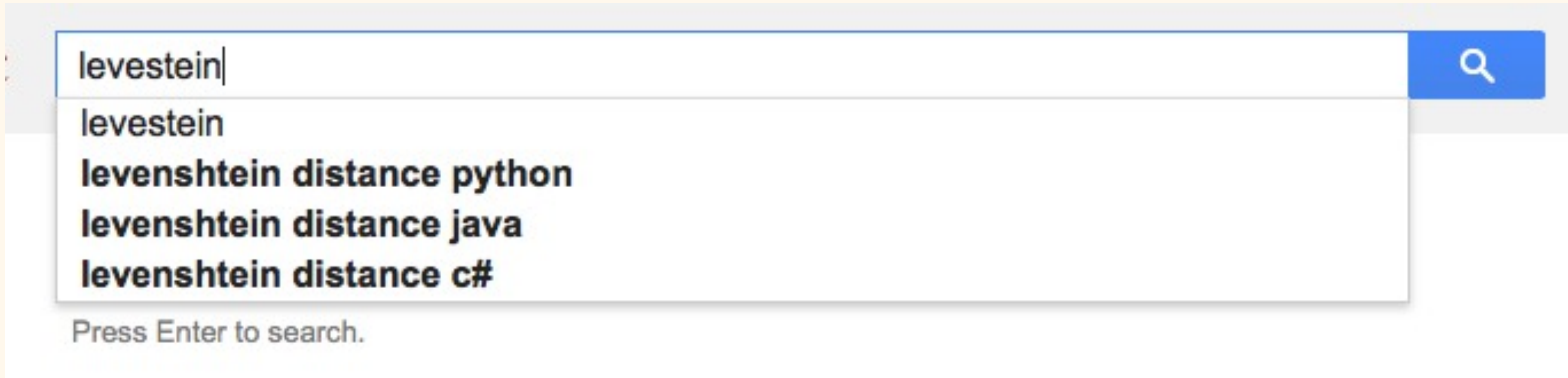
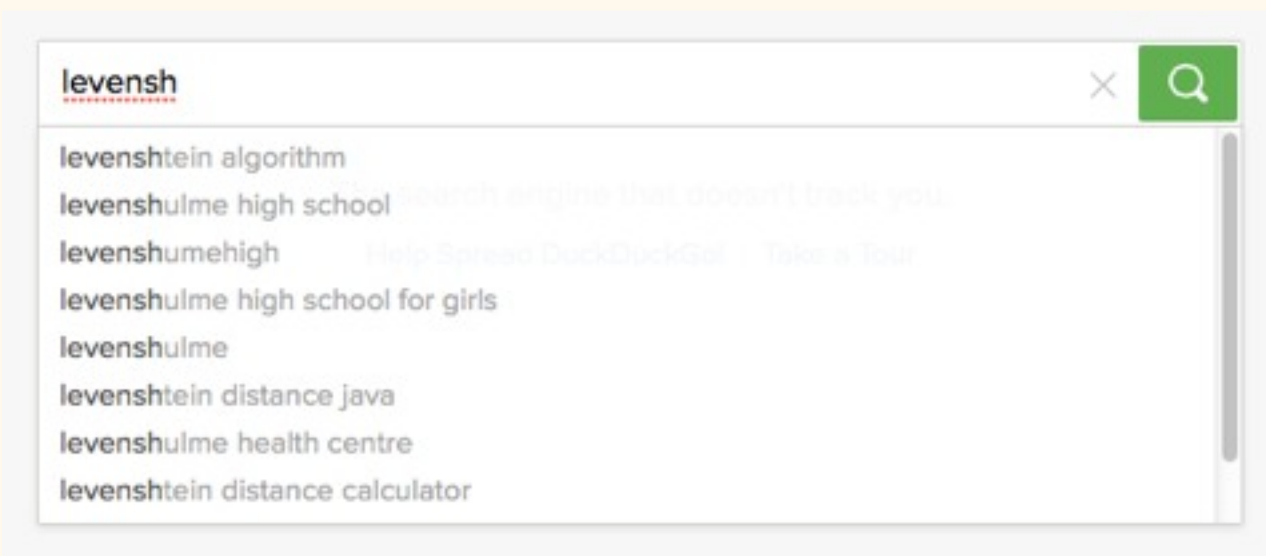
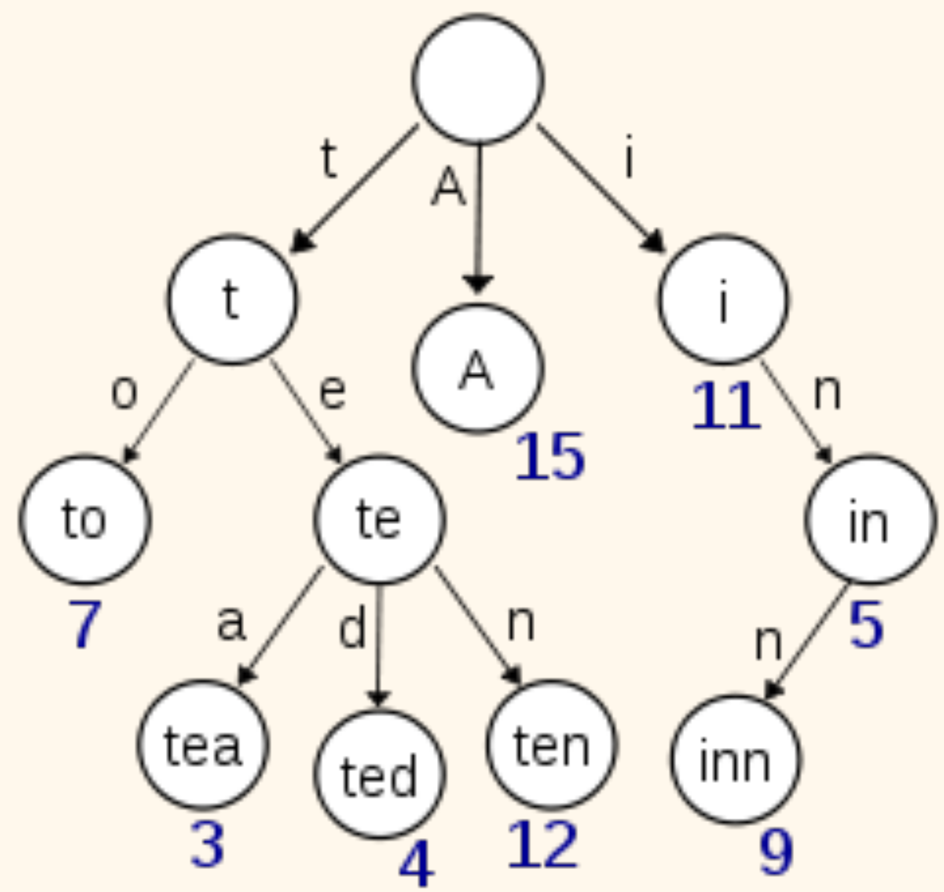
More fuzzy string matching!







Steven Bedrick

CS/EE 5/655, 12/1/14

Plan for today:

Tries

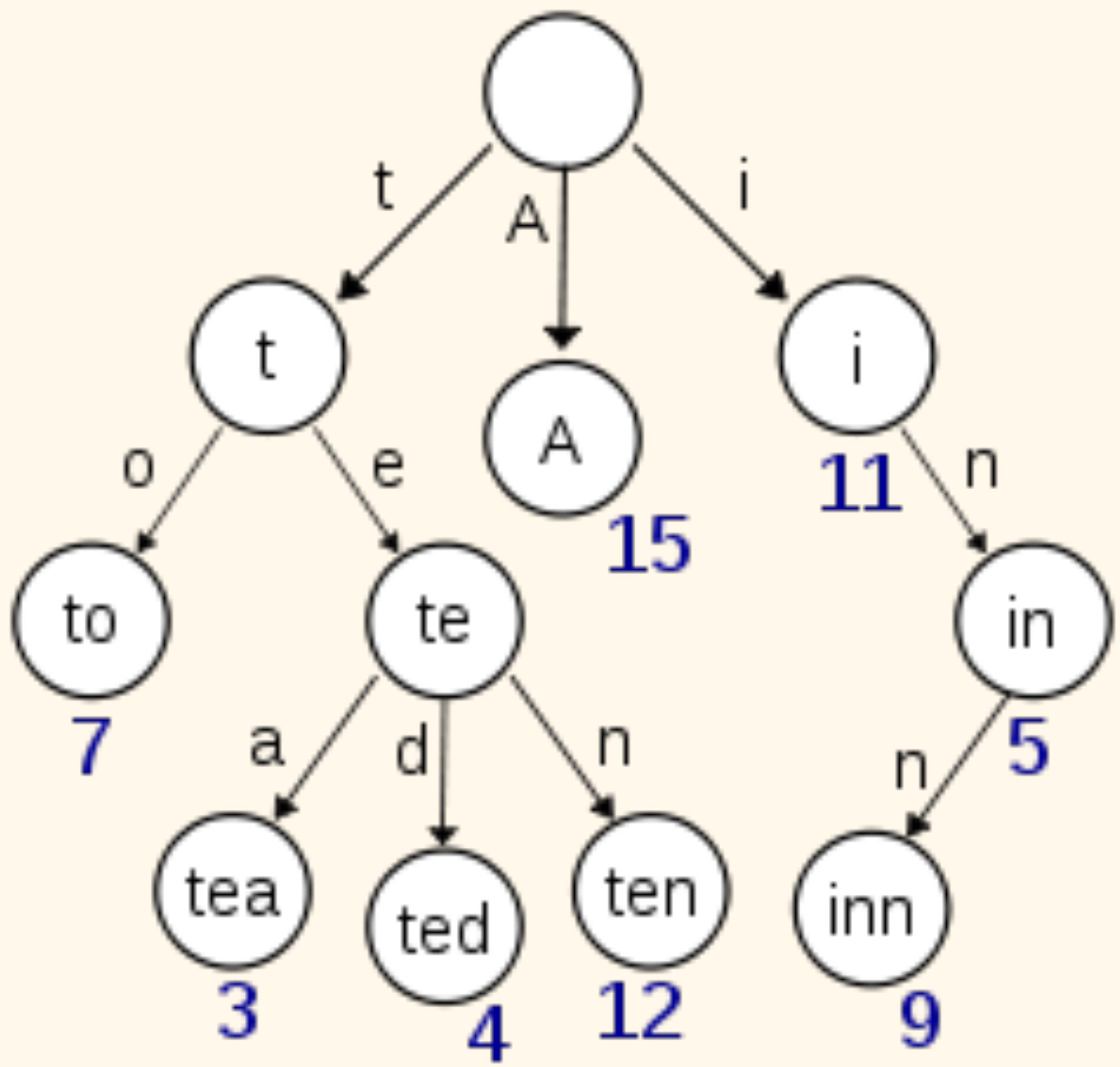
Simple uses of tries

Fuzzy search with tries

Levenshtein automata

A trie is essentially a *prefix tree*:

```
A: 15
i: 11
in: 5
inn: 9
to: 7
tea: 3
ted: 4
ten: 12
```



Simple uses of tries:

Key lookup in O(m) time, predictably.

(Compare to hash table: best-case O(1), worst-case O(n), depending on key)

Fast longest-prefix matching

IP routing table lookup

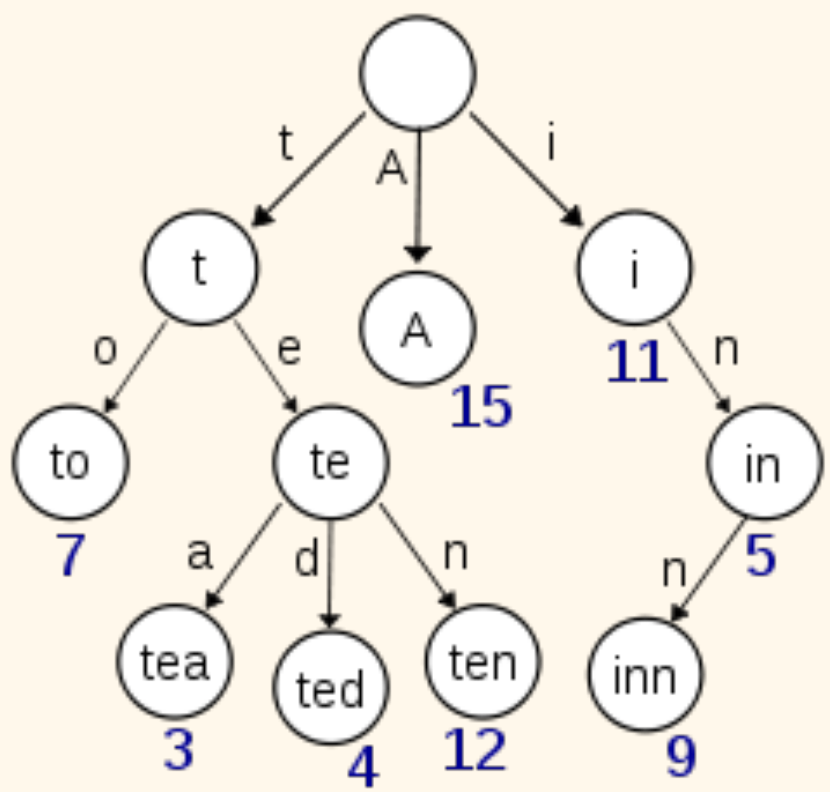
For an incoming packet, find the closest next hop in a routing table.

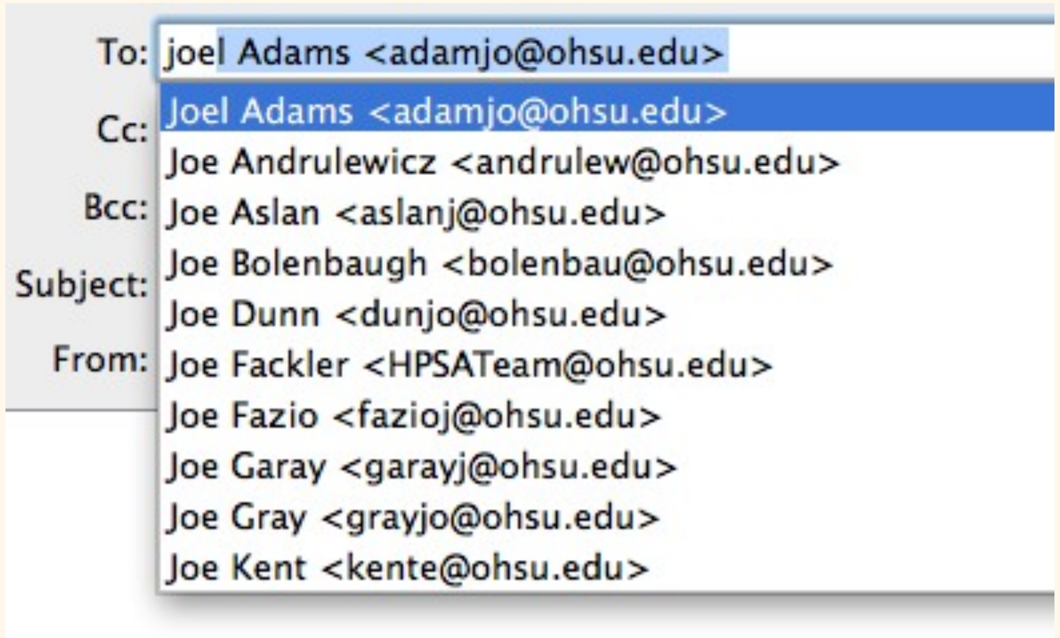
Simple uses of tries:

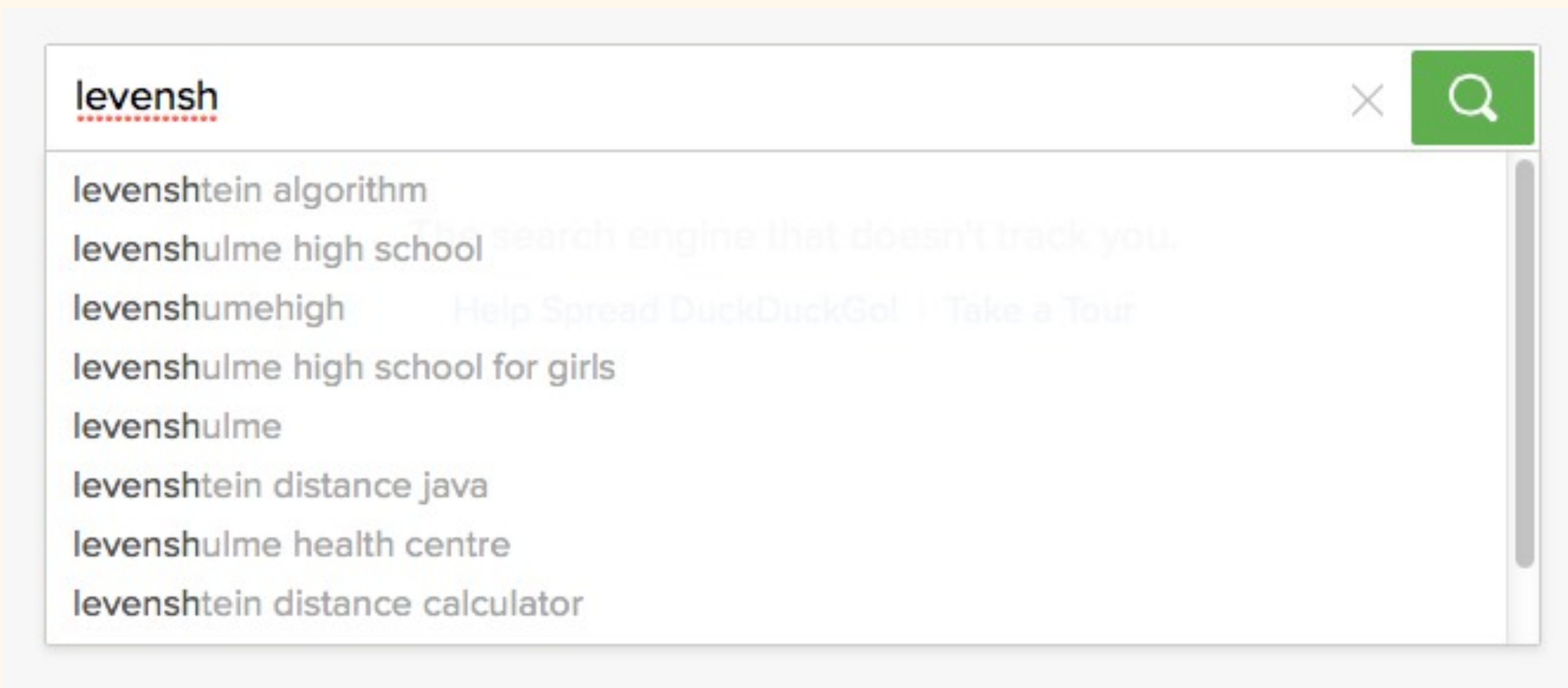
Fast longest-prefix matching

Useful for autocompletion:

"All words/names/whatevers that start with *XYZ*..."

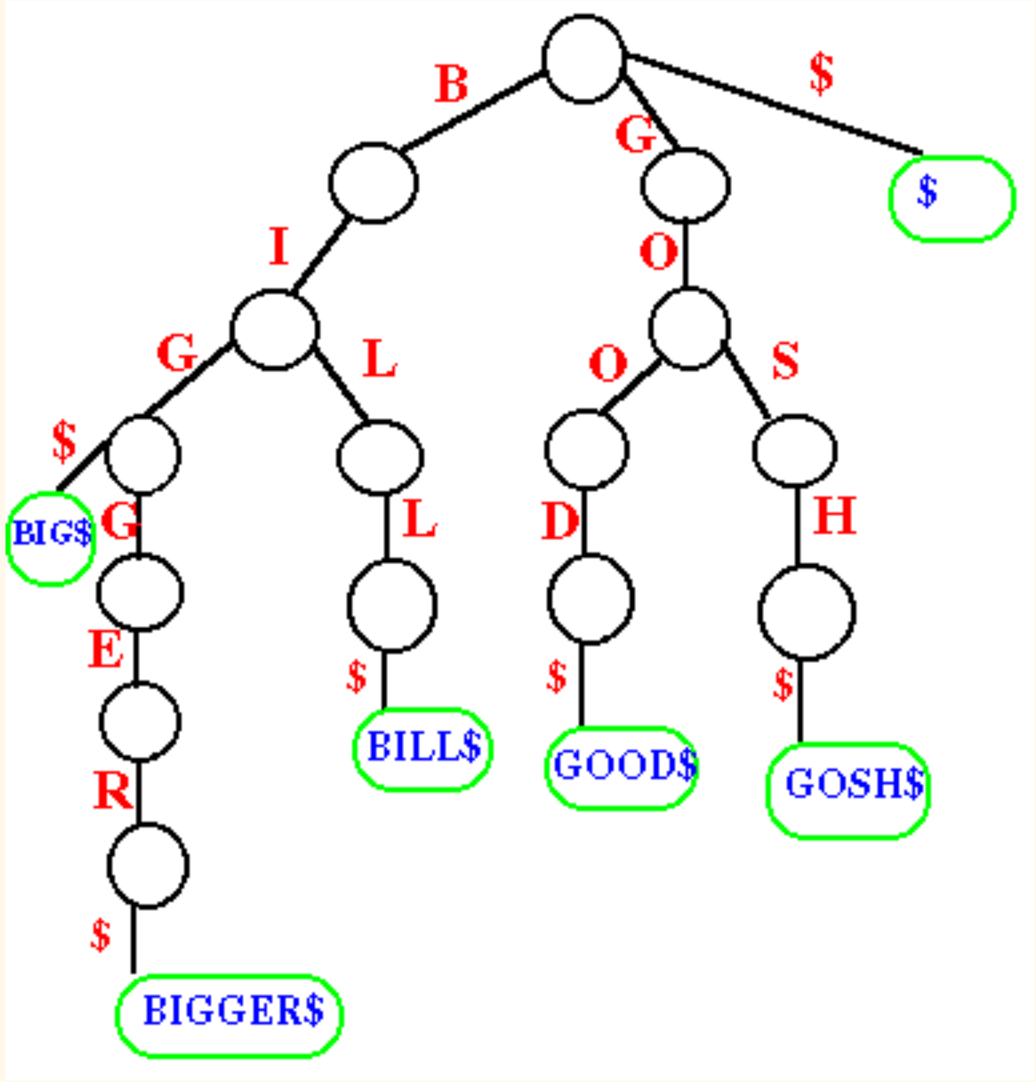



To: joel Adams <adamjo@ohsu.edu>
Cc:
Bcc:
Subject:
From:
Joel Adams <adamjo@ohsu.edu>
Joe Andrulewicz <andrulew@ohsu.edu>
Joe Aslan <aslanj@ohsu.edu>
Joe Bolenbaugh <bolenbau@ohsu.edu>
Joe Dunn <dunjo@ohsu.edu>
Joe Fackler <HPSATeam@ohsu.edu>
Joe Fazio <fazioj@ohsu.edu>
Joe Garay <garayj@ohsu.edu>
Joe Gray <grayjo@ohsu.edu>
Joe Kent <kente@ohsu.edu>


levensh
levenshtein algorithm
levenshulme high school
levenshumehigh
levenshulme high school for girls
levenshulme
levenshtein distance java
levenshulme health centre
levenshtein distance calculator

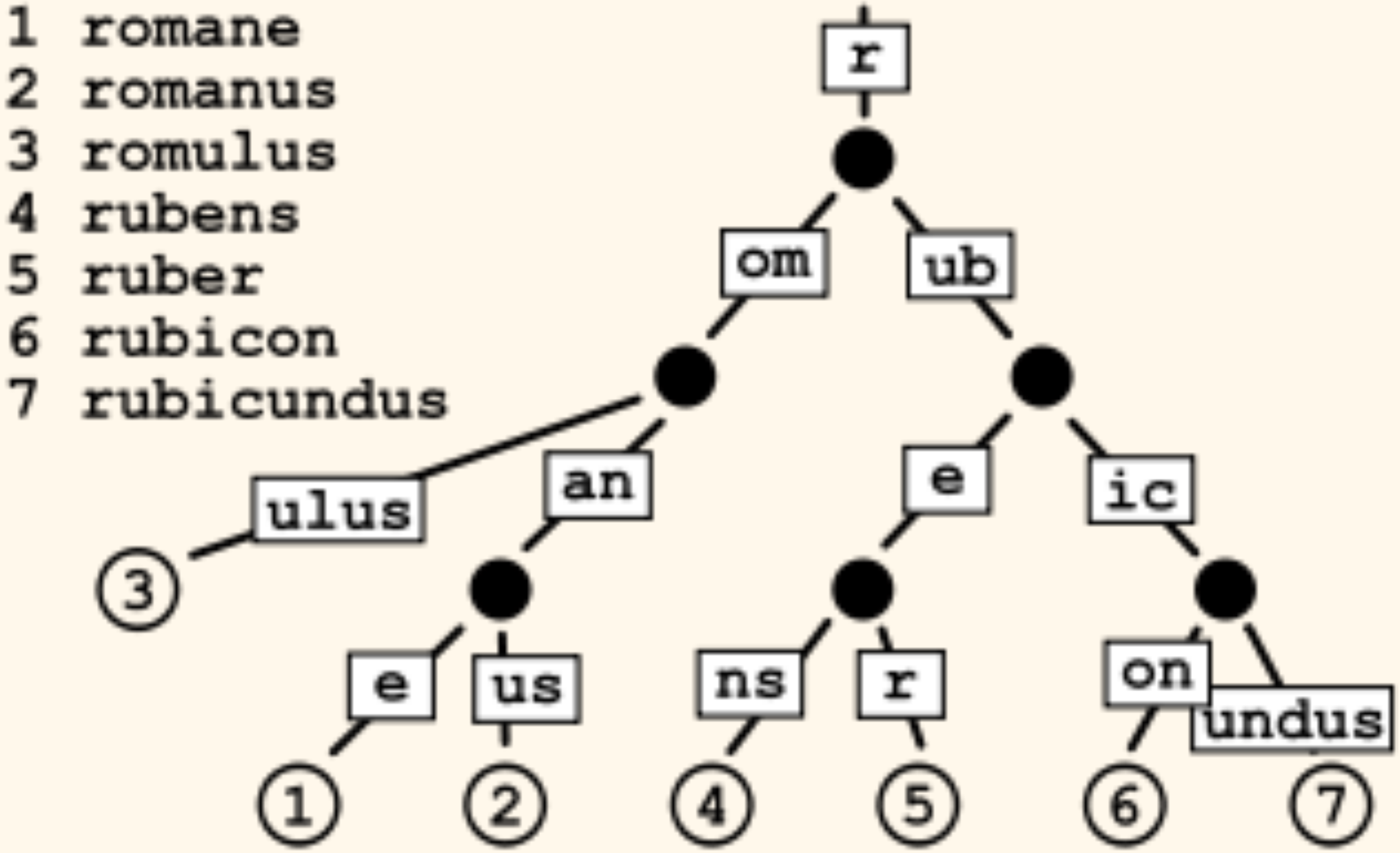
The problem with tries:

When the space of keys is sparse, the trie is not very compact:



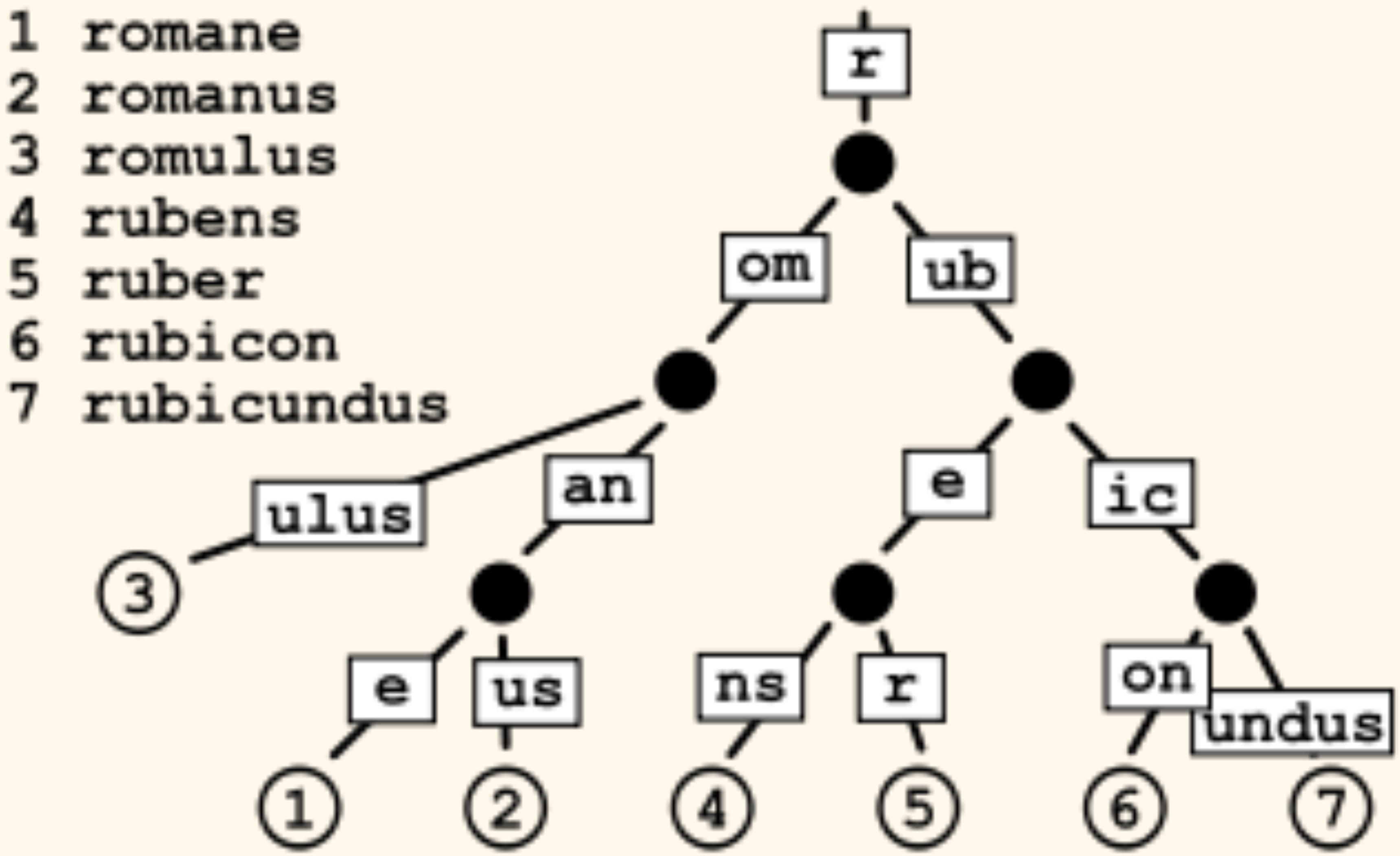
(One) Solution: PATRICIA Tries

(One) Solution: PATRICIA Tries



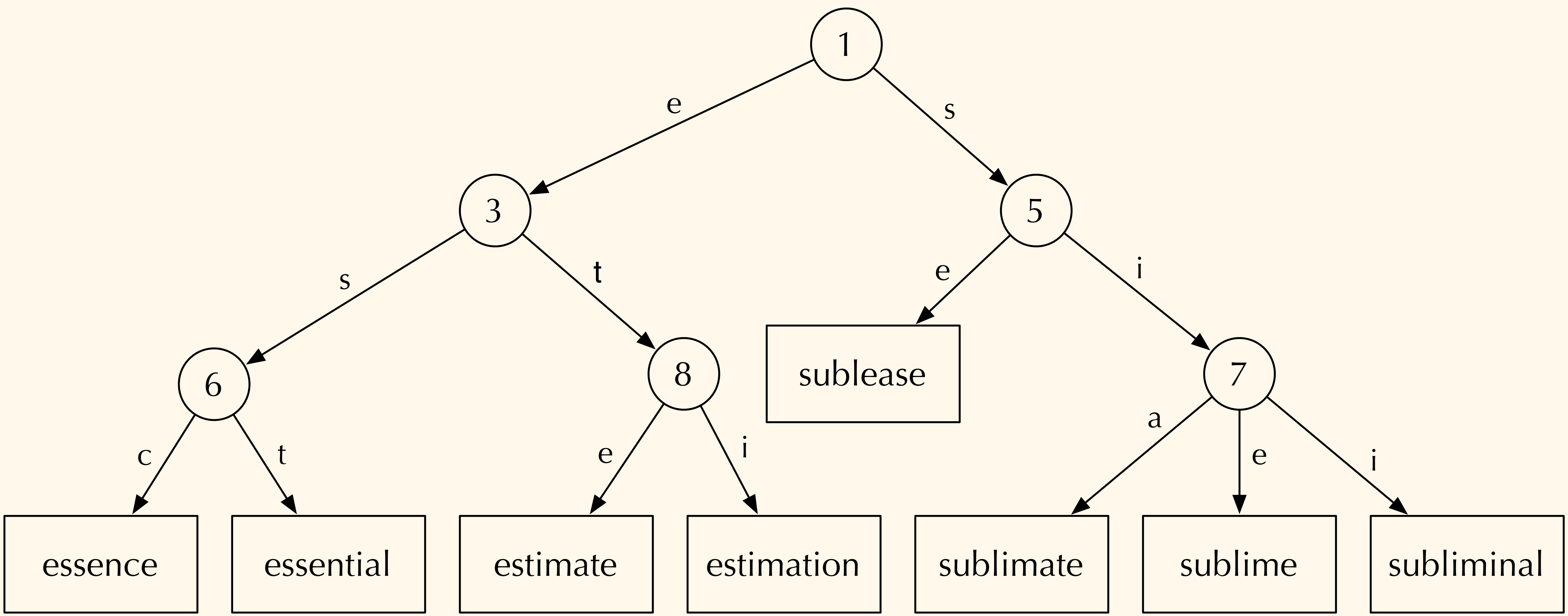
Key ideas: edges represent more than a single symbol; nodes with only one child get collapsed.

One could explicitly represent edges with multiple symbols...

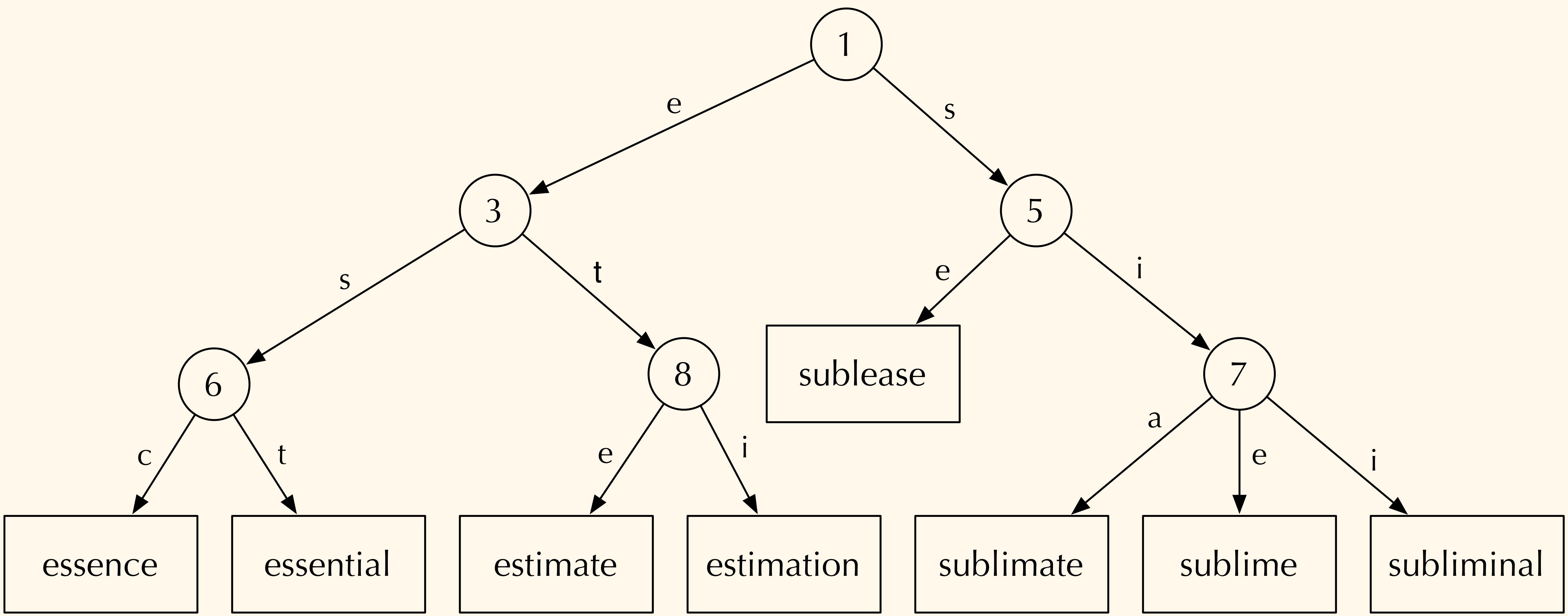


... but that would complicate matching.

Instead, each internal node stores the *offset* for the next difference to look for:

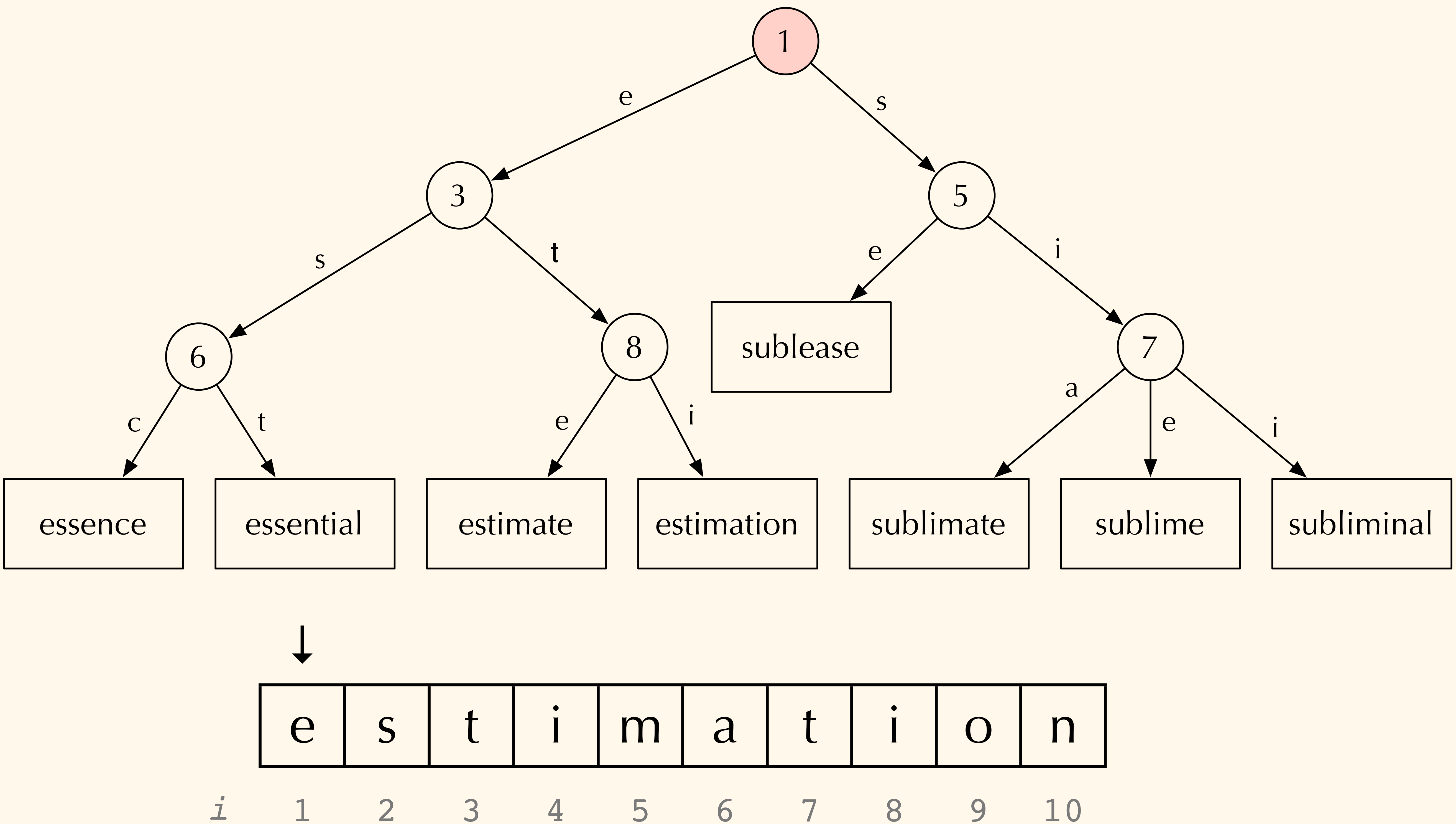


Instead, each internal node stores the *offset* for the next difference to look for:

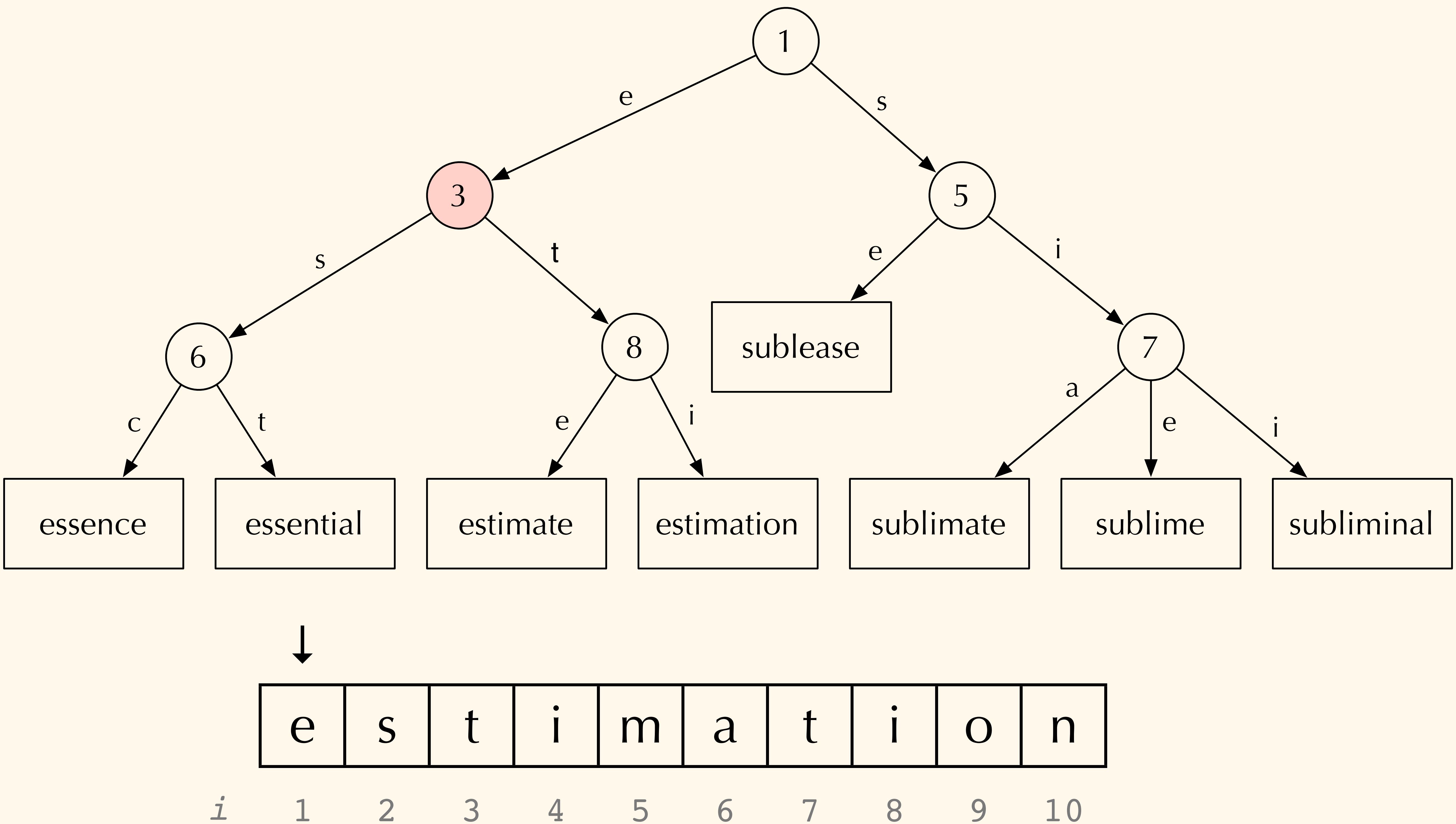


i	1	2	3	4	5	6	7	8	9	10
	e	s	t	i	m	a	t	i	o	n

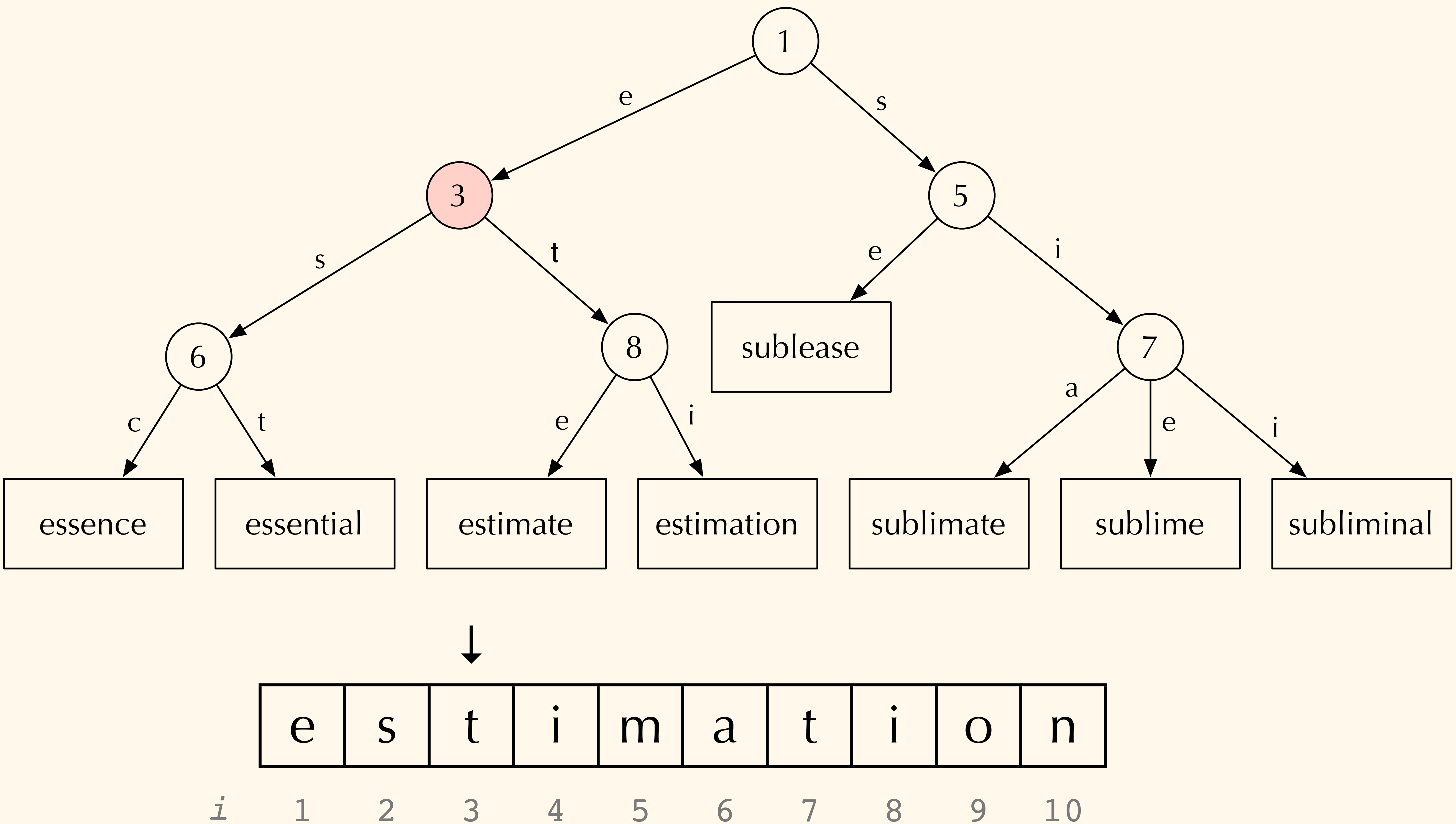
Instead, each internal node stores the *offset* for the next difference to look for:



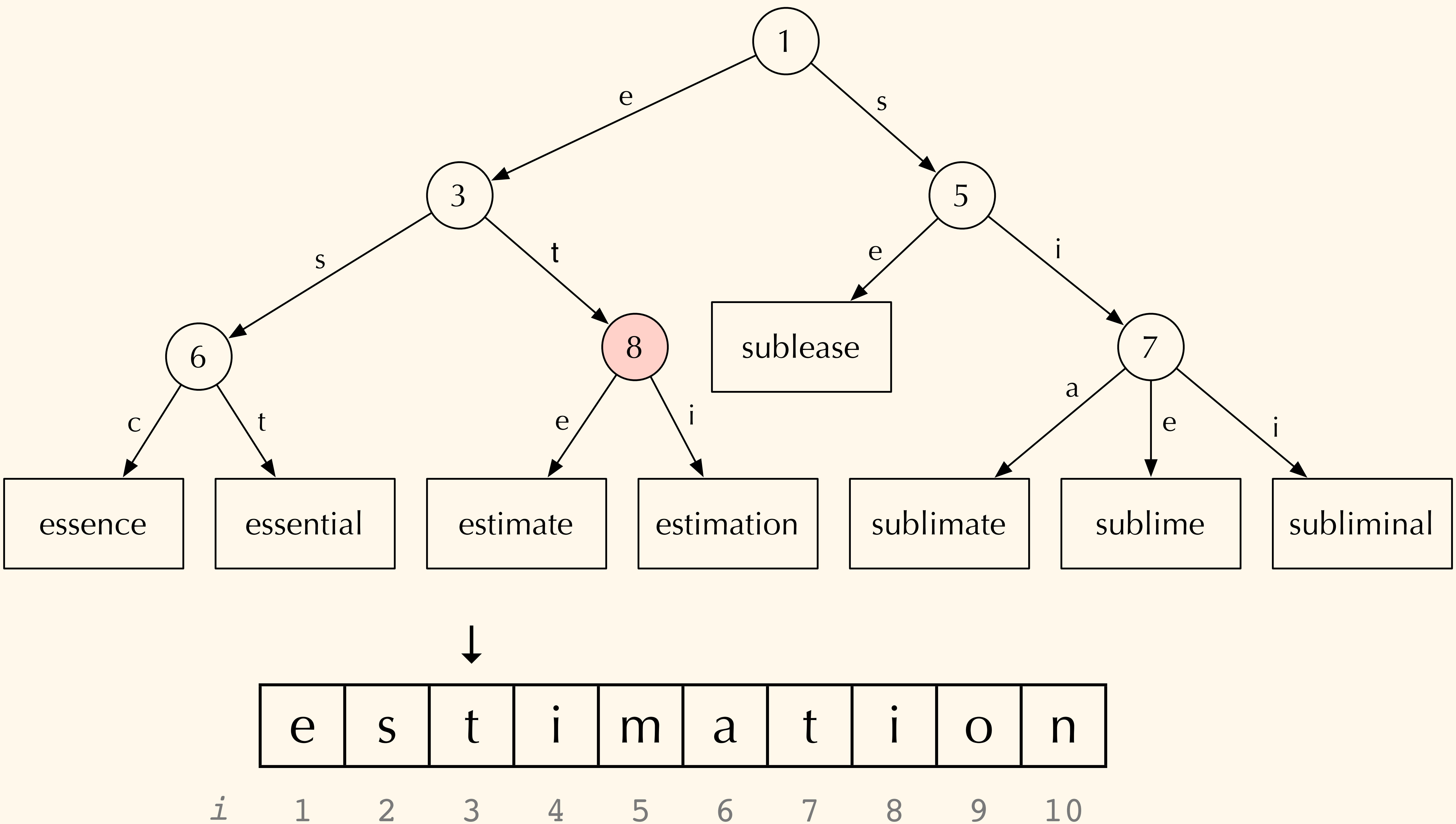
Instead, each internal node stores the *offset* for the next difference to look for:



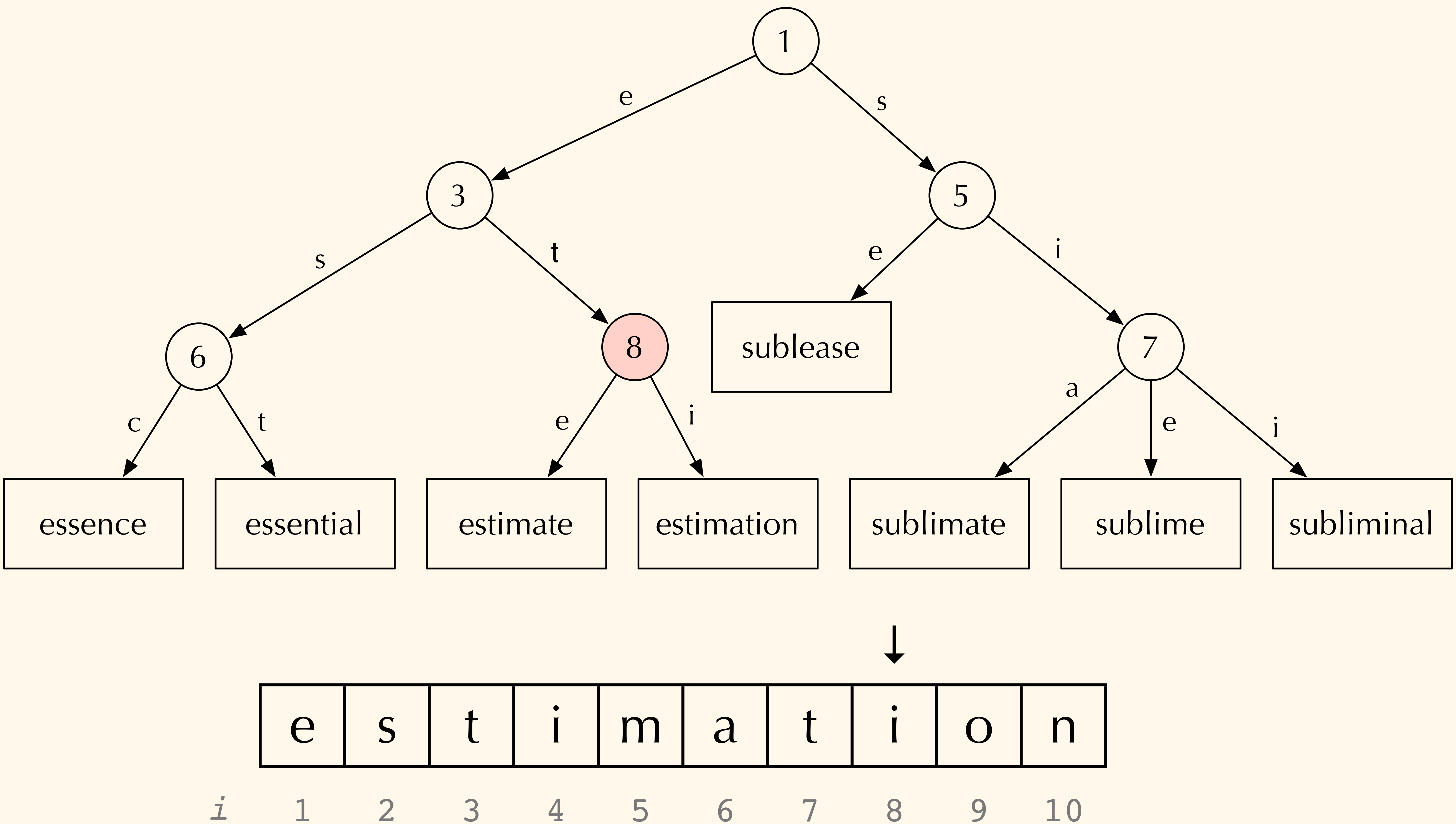
Instead, each internal node stores the *offset* for the next difference to look for:



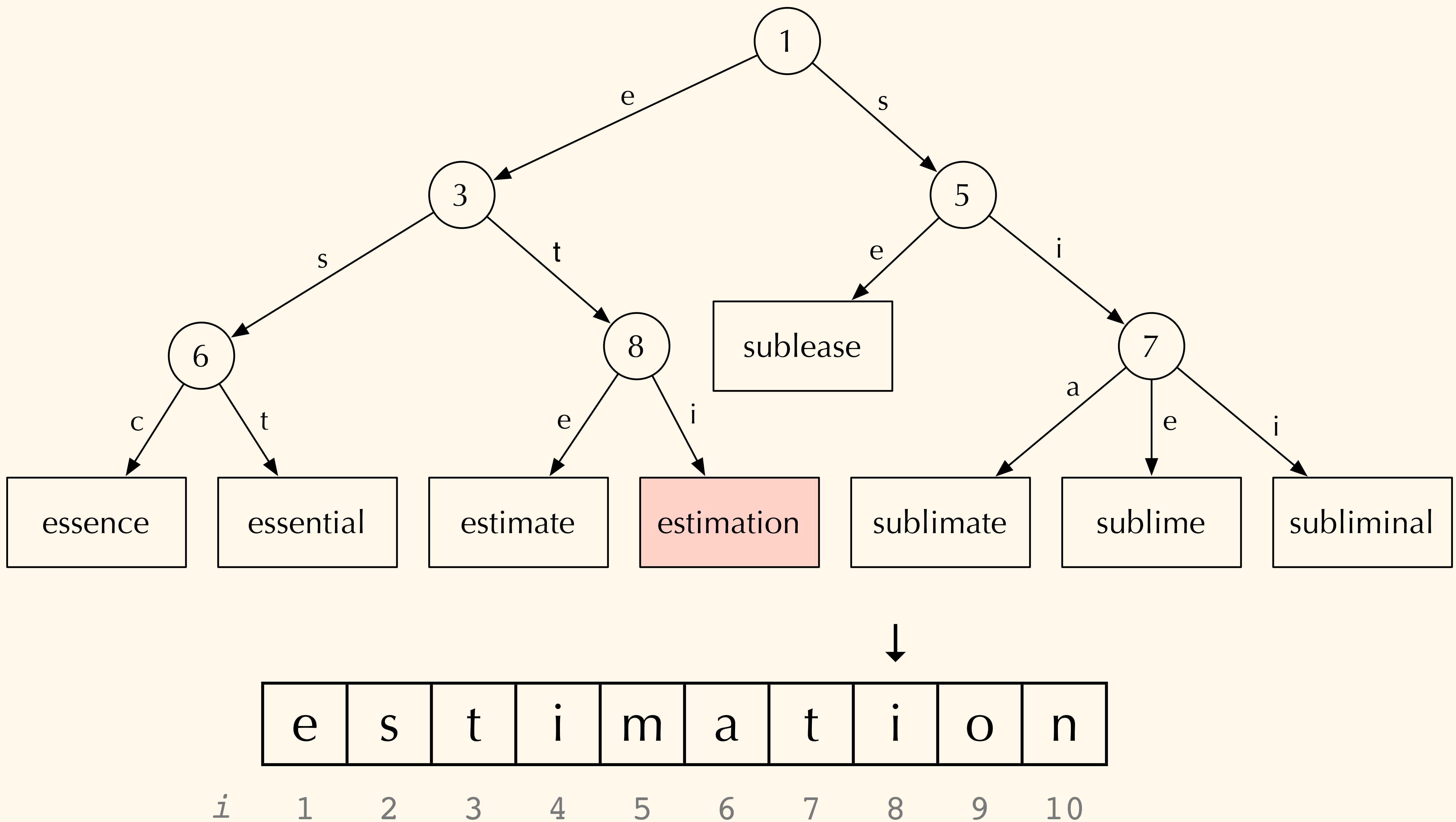
Instead, each internal node stores the *offset* for the next difference to look for:



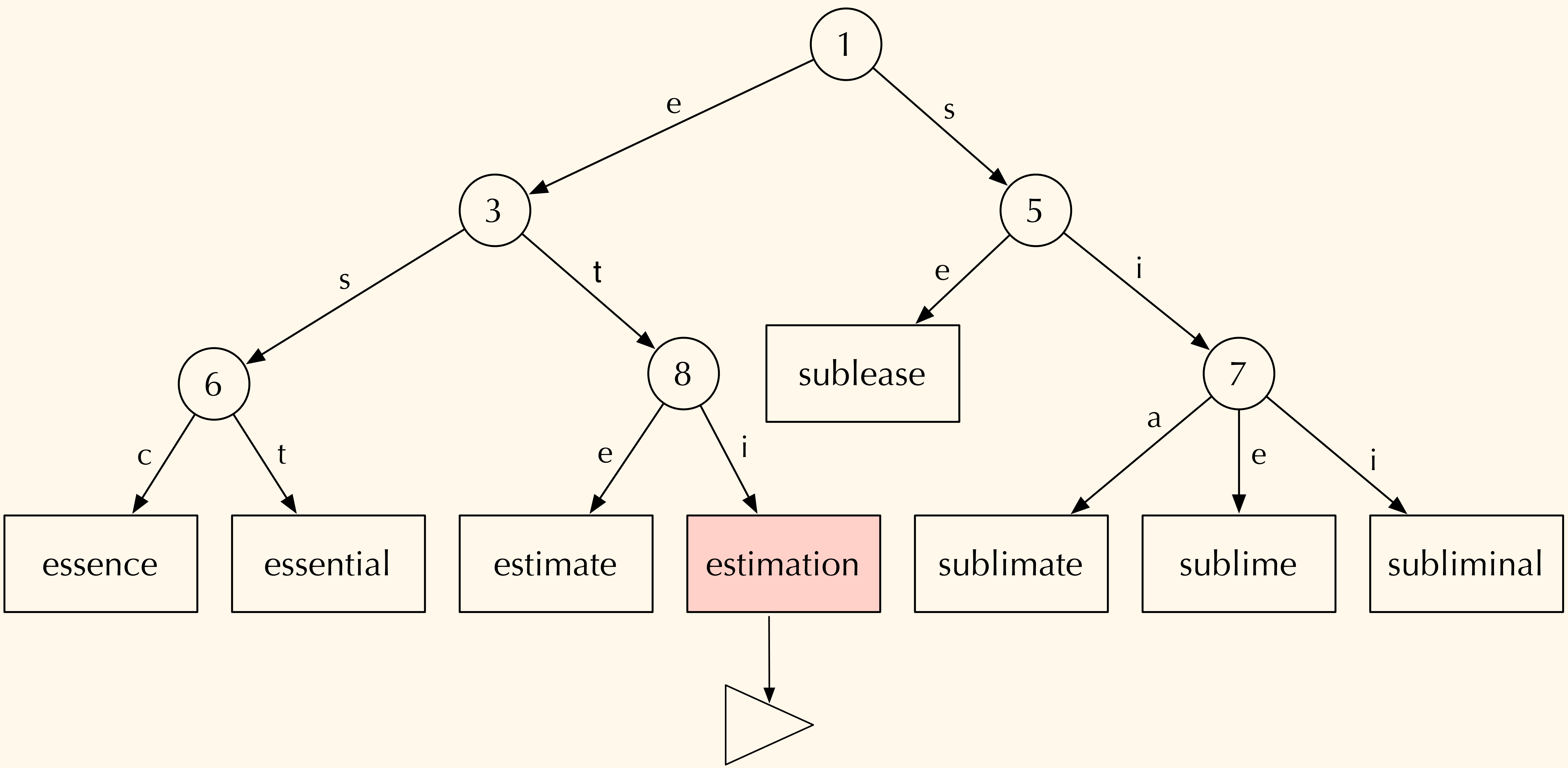
Instead, each internal node stores the *offset* for the next difference to look for:



Instead, each internal node stores the *offset* for the next difference to look for:



Instead, each internal node stores the *offset* for the next difference to look for:



Plan for today:

Tries

Simple uses of tries

Fuzzy search with tries

Levenshtein automata

Fuzzy search with tries

Problem: we want to search a dictionary for words *similar* to a query.

Example: “Smyth” and “Zmith” should retrieve “Smith”,

“Levenstien” should retrieve “Levenshtein”, etc.

By “similar,” we mean “edit distance less than some threshold δ.”

One solution:

Compute pairwise edit distance between our query q and every word w_i in our dictionary;

Match if $sim(q, w_i) \leq \delta$

A (slightly) better solution:

Speed up pairwise edit distance computation using *prefix pruning*.

Prefix pruning's key idea:

If we only care whether strings *r* and *s* have an edit distance less than some threshold…

i		j: 0	1: e	2: b	3: a	4: y
0		0	1	2	3	4
1	k	1	1	2	3	4
2	o	2	**2**	**2**	**3**	4
3	b	3	3	2	3	4
4	y	4	4	3	3	3

…we can do early termination of our computation as soon we exceed that threshold.

Wang J, Feng J, Li G. Trie-join: efficient trie-based string similarity joins with edit-distance constraints. VLDB Endowment; 2010.

One solution:

Compute pairwise edit distance between our query q and every word w_i in our dictionary;

Match if $sim(q, w_i) \leq \delta$

A (slightly) better solution:

Speed up pairwise edit distance computation using *prefix pruning*.

Neither are very good solutions for any kind of “on-line” use case:

Query autocompletion, fuzzy searching, spellchecking, etc.

(our dictionary is large, number of searches is high, etc. etc.)

A better solution: use a trie!

1. Build a trie out of our dictionary;

2. Iterate through *q*; at each point, identify a set of *active nodes* of the trie.

A node *n* is "active" with respect to a prefix q_i if the edit distance between q_i and the prefix represented by *n* is $\leq \delta$.

Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW '09

A better solution: use a trie!

1. Build a trie out of our dictionary;

2. Iterate through *q*; at each point, identify a set of *active nodes* of the trie.

3. Stop when we reach the end of *q* or no longer have active nodes.

Active nodes that happen to be leaves represent matches.

Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW '09

A better solution: use a trie!

1. Build a trie out of our dictionary;

2. Iterate through *q*; at each point, identify a set of *active nodes* of the trie.

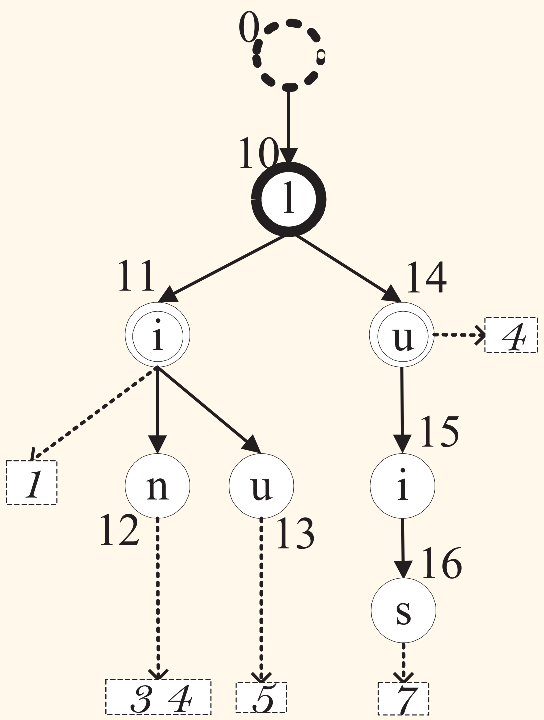
3. Stop when we reach the end of *q* or no longer have active nodes.

Active nodes that happen to be leaves represent matches.

Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW '09

Intuition: at each symbol *i* in *q*, the set of active nodes will be related to the set from q_{i-1}.

So, we don't need to visit every node in the trie!

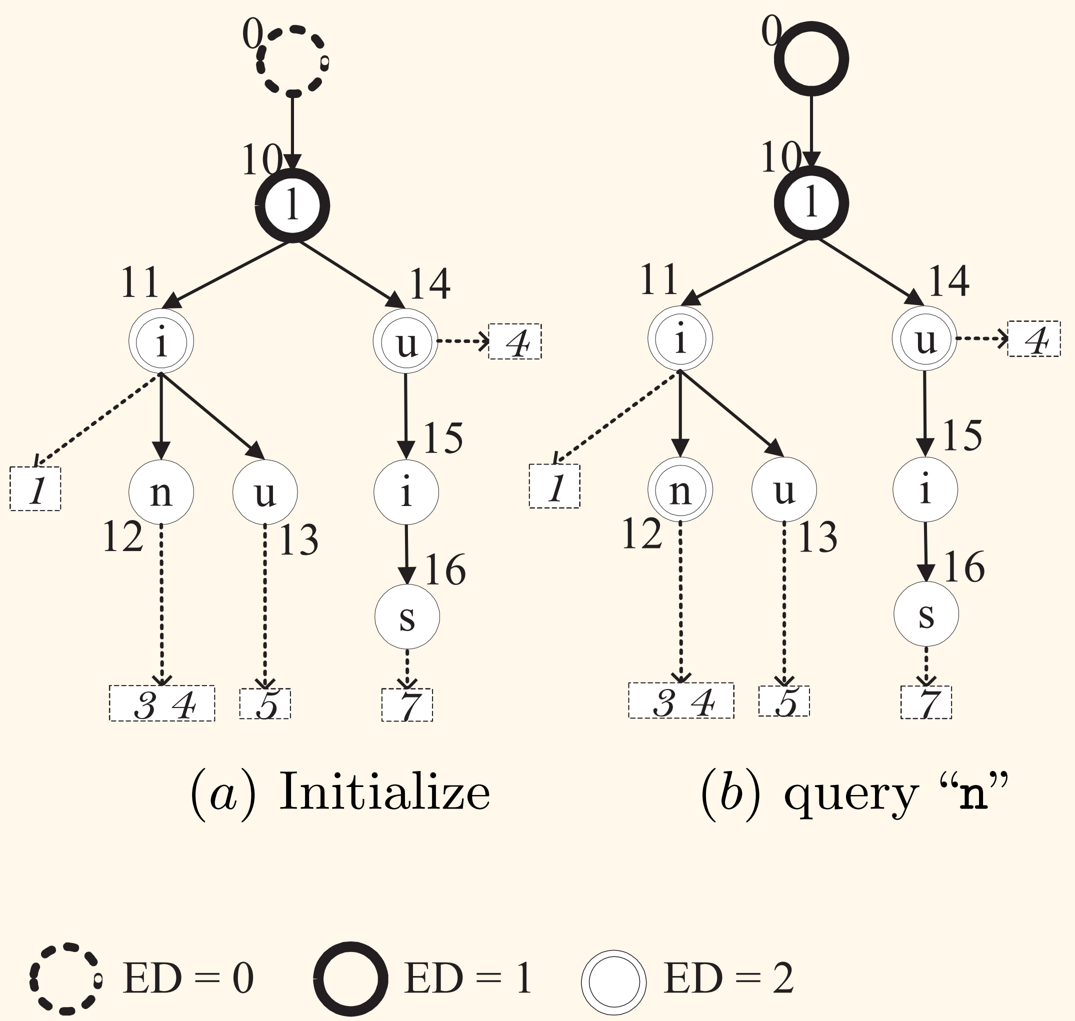


(*a*) Initialize

Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW '09

Intuition: at each symbol *i* in *q*, the set of active nodes will be related to the set from q_{i-1}.

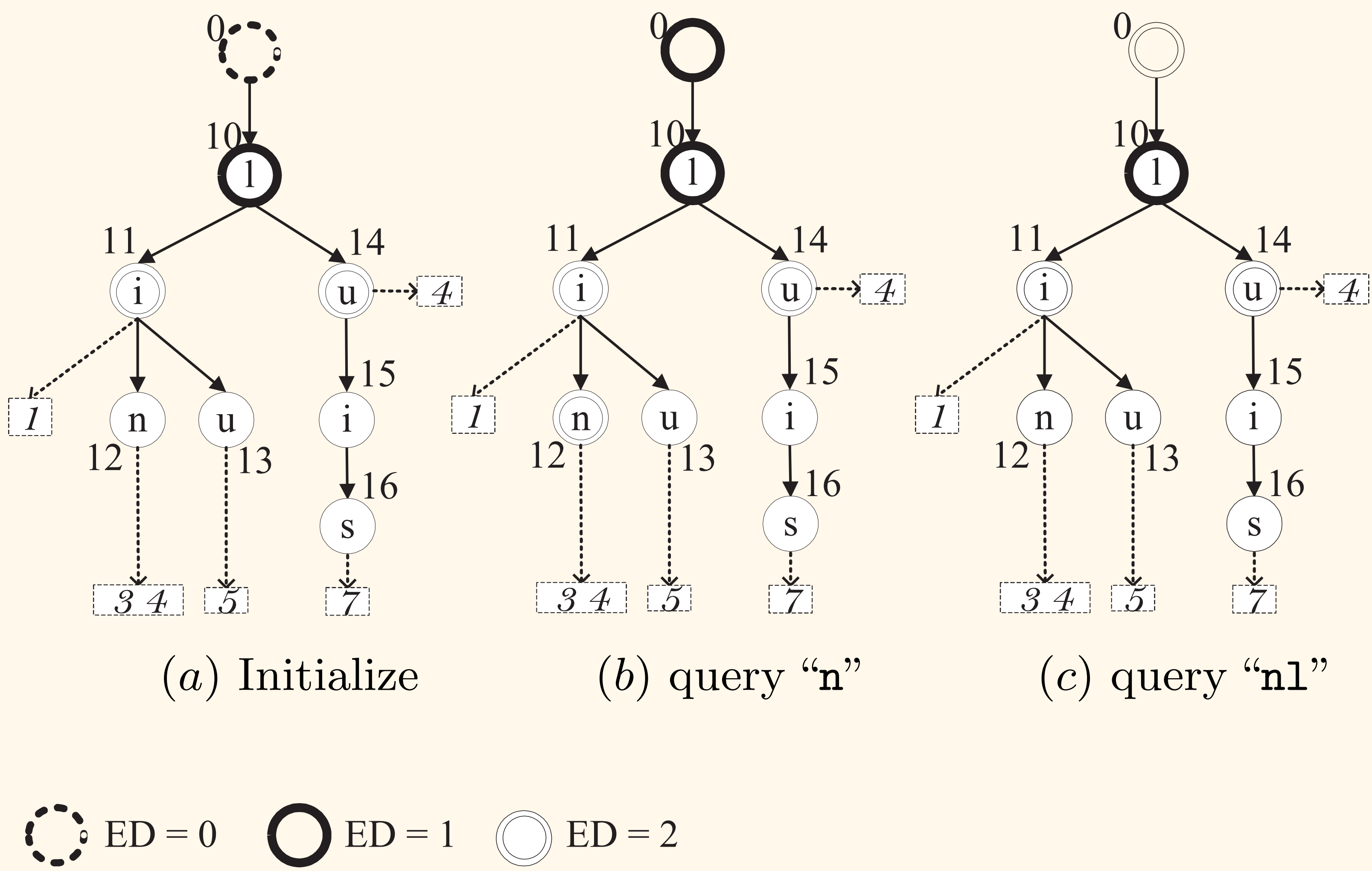
So, we don't need to visit every node in the trie!



Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW '09

Intuition: at each symbol *i* in *q*, the set of active nodes will be related to the set from q_{i-1}.

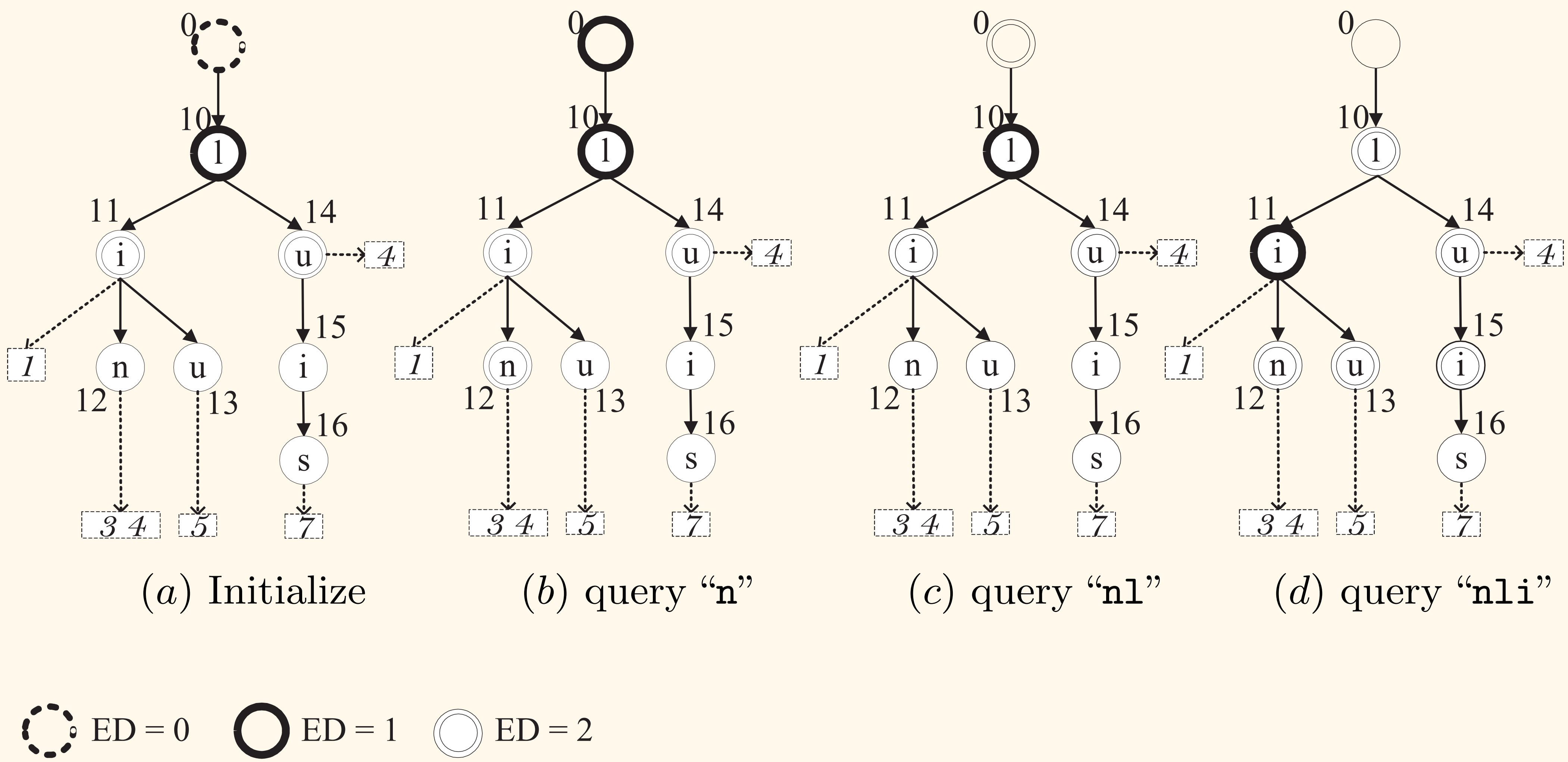
So, we don't need to visit every node in the trie!



Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW '09

Intuition: at each symbol *i* in *q*, the set of active nodes will be related to the set from q_{i-1}.

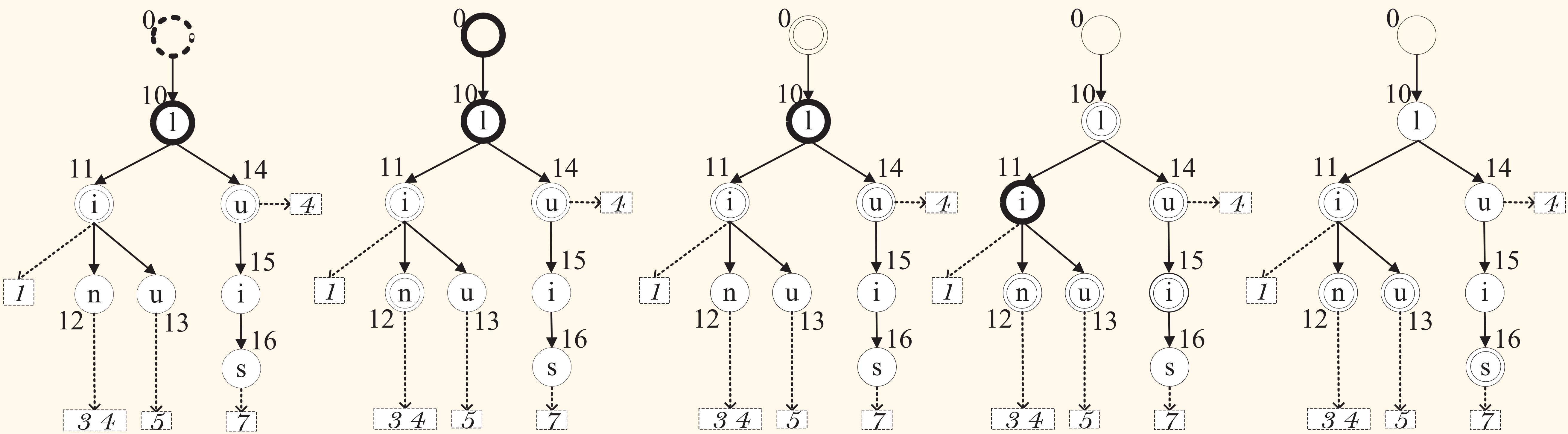
So, we don't need to visit every node in the trie!



Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW '09

Intuition: at each symbol *i* in *q*, the set of active nodes will be related to the set from q_{i-1}.

So, we don't need to visit every node in the trie!



(*a*) Initialize (*b*) query "**n**" (*c*) query "**nl**" (*d*) query "**nli**" (*e*) query "**nlis**"

Ji S, Li G, Li C, Feng J. Efficient interactive fuzzy keyword search. WWW '09

Related problem: the “similarity join”

We have two bags of words, *R* and *S*.

Goal: identify pairs of similar words.

Example:

R = { *kobe*, *ebay*, ...}

S = { *bag*, *koby*, ...}

We would want to identify pairs such as <*kobe*, *koby*>

Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).

Again, one solution is pairwise edit distance calculation...

... but if *R* and *S* are very large, that will be incredibly time consuming, even with prefix pruning!

One solution: use the trie search method!

Build a trie representing *R*;

For every string s in S, identify the active nodes A_s of R′s trie; for each leaf node *r* in A_s, produce <*s*,*r*>.

Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).

Another solution: use sub-trie pruning

Intuition: given the set of active nodes A_n for a particular trie node n...

... we can say that *only children of nodes in* A_n could possibly be similar to children of node n.

We can use this fact to speed up extraction of similar pairs.

Let us consider the case where our two sets are actually one set ($R = S$), and we simply want to identify similar pairs.

Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).

Algorithm:

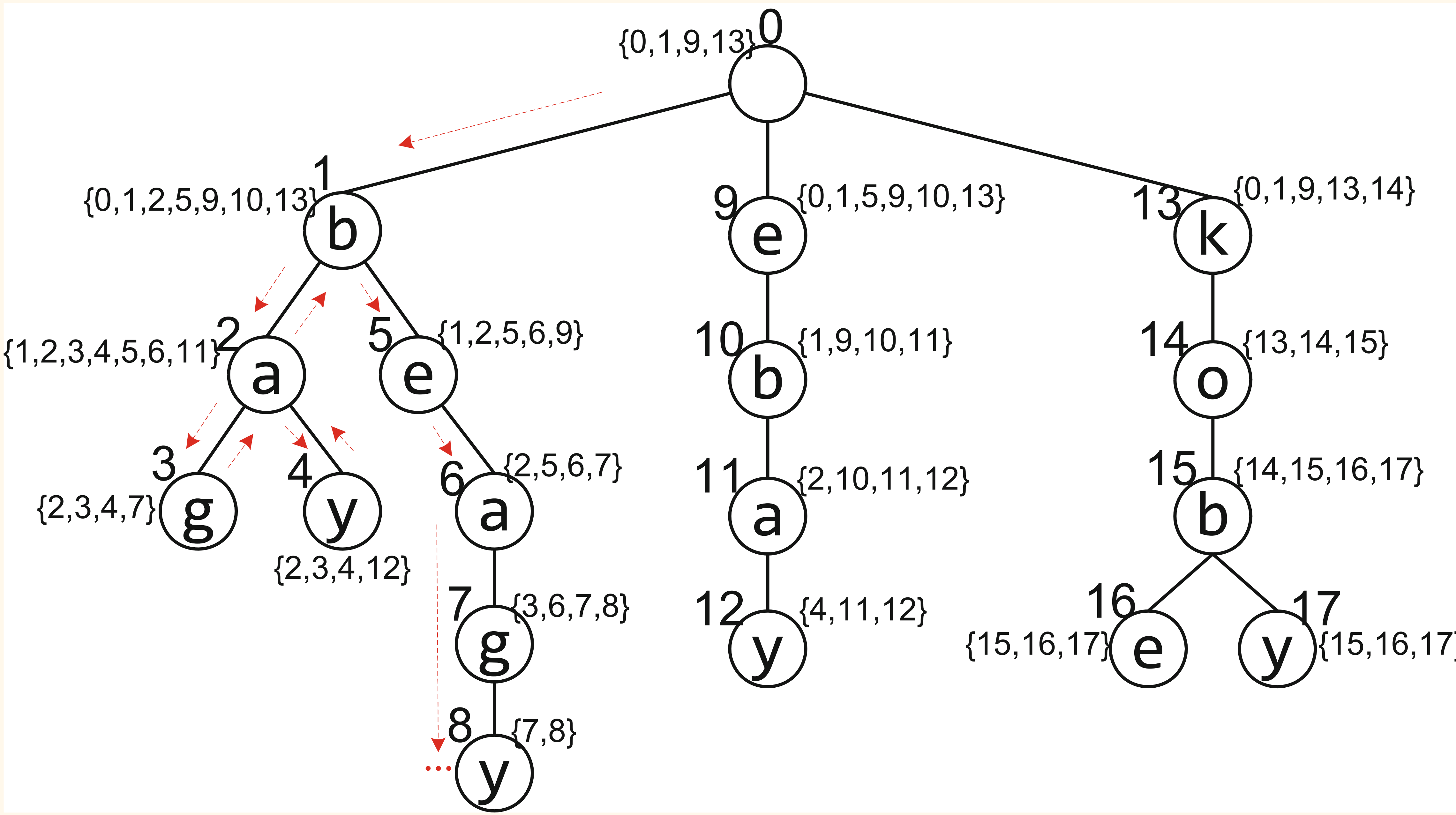
1. Build a trie for our set of words;

2. Traverse the trie in preorder. At each node, compute its set of active nodes A.

3. At each leaf node n, identify any leaf nodes in A_n; these are similar pairs.

As we traverse, we must keep the current node's ancestor's set of active nodes in memory; total time complexity is $O(\delta|A_T|)$.

Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).



Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).

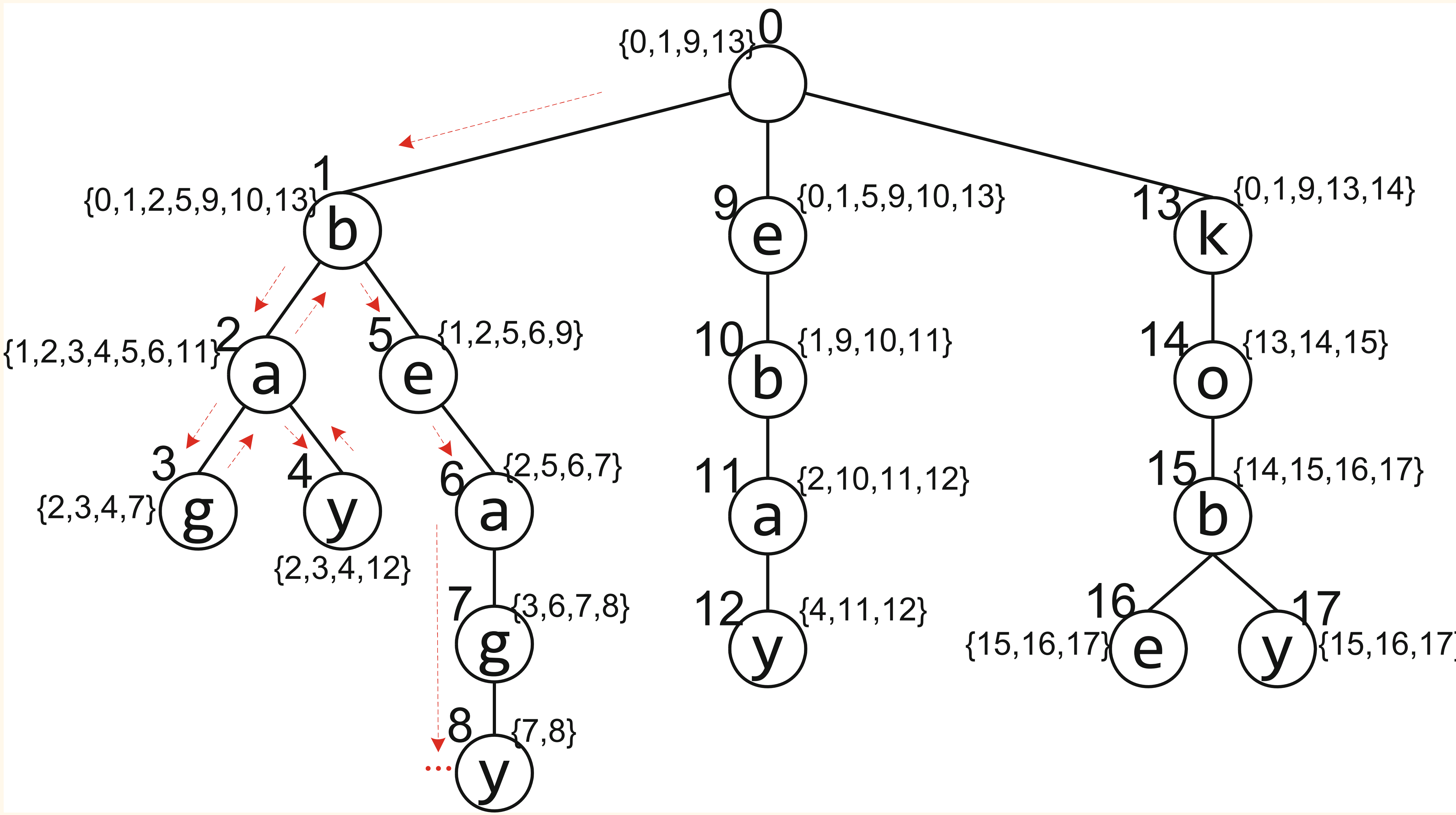
We can do better!

Intuition: given the set of active nodes A_n for a particular trie node *n*...

... we can say that *only children of nodes in* A_n could possibly be similar to children of node *n*.

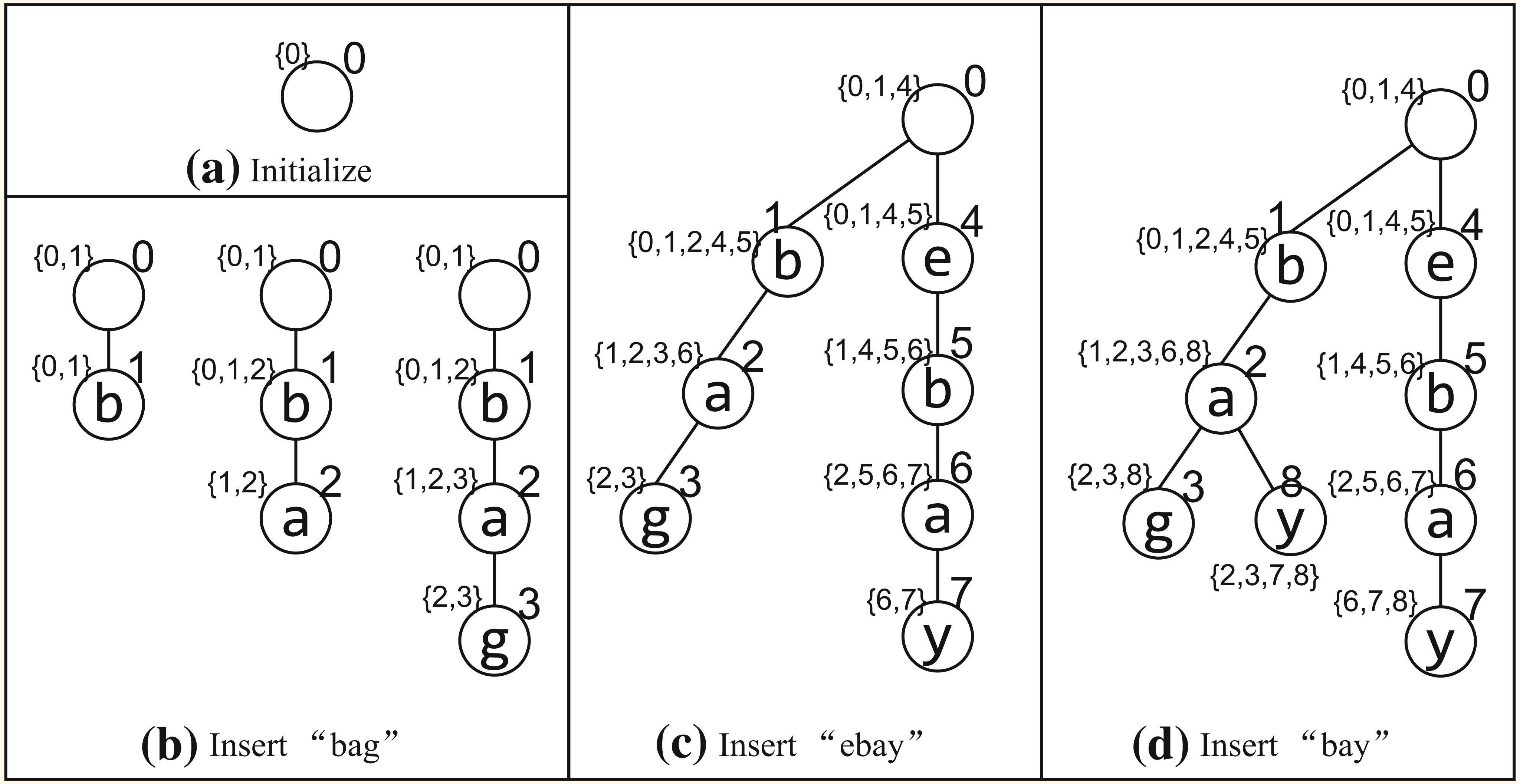
Also: if node *u* has node *v* in its active set, *v* must also have *u* in its set!

Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).



Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).

We can compute active nodes as we build the trie, and eliminate duplicate calculations.



By increasing our space complexity, we can reduce the time complexity to $O(\frac{\delta}{2}|A_T|)$.

Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).

There are extensions to the idea that allow for different sets of strings, more space-efficient construction, etc.

See the Feng, et al. article (cited below) for more details!

Feng J, Wang J, Li G. Trie-join: a trie-based method for efficient string similarity joins. The VLDB Journal — The International Journal on Very Large Data Bases. 2012 Aug;21(4).

UCI People Search

Search by Name, Nick Name, UCInetID, Title, Department, Major, Office Address, Phone Number, EMail Address and Zot Code

professor smyt | Search

Patrick J. **SMYT**H	pjsmyth	Padhraic SMYTH	**Professor**	Computer Science-Computing	(949) 824-2558	4062...
Janellen **Smit**h	jesmith7		**Professor**	Dermatology	(949) 824-5515	C252...
Clyde W **SMIT**H	smithcw		Clinical **Professor**	Radiological Sciences	(714) 456-5033	Bldg...
John H. **SMIT**H	jhsmith		**Professor** and Chair	German	(949) 824-6406, 6107	400G...

http://psearch.ics.uci.edu

Plan for today:

Tries

Simple uses of tries

Fuzzy search with tries

Levenshtein automata

Levenshtein automata

A different approach to solving the fuzzy matching problem uses finite-state automata.

The basic idea: construct an acceptor that will recognize an input string with up to δ edits.

Then, walk through the acceptor and our dictionary, emitting any final states we visit.

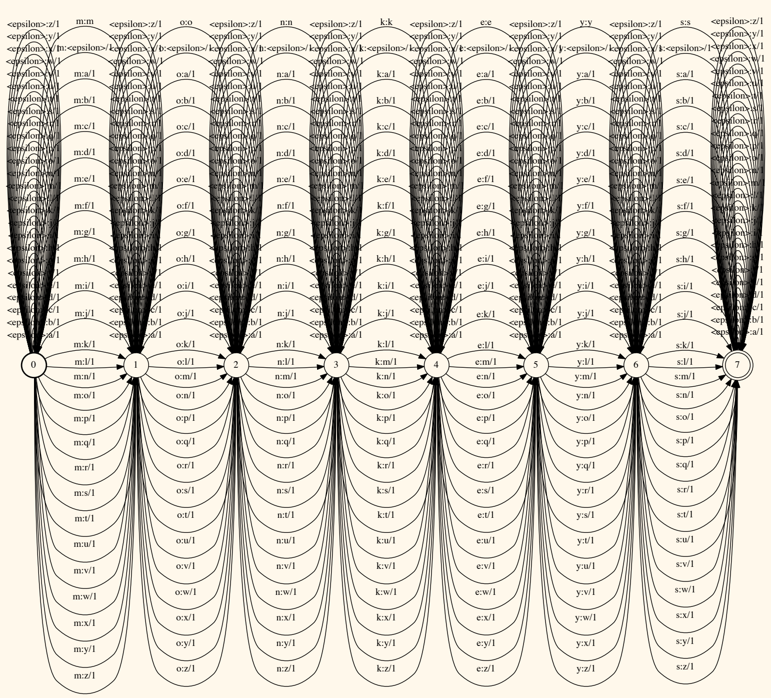
Schulz KU, Mihov S. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition. 2002 Nov 1;5(1):67–85.

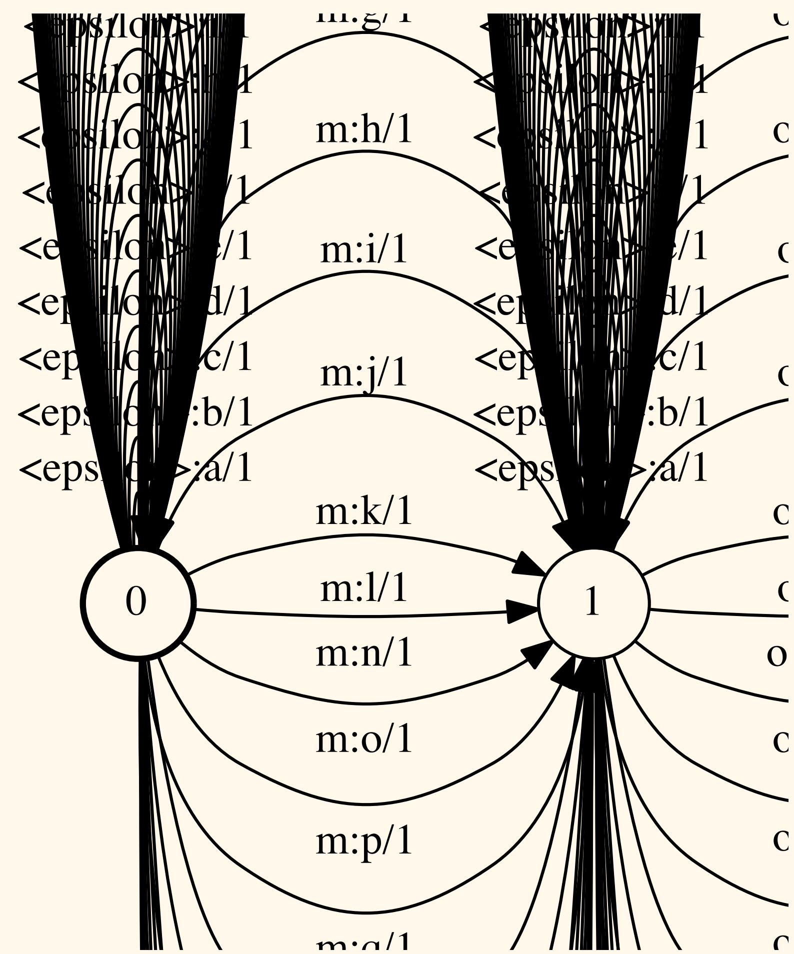
Mihov S, Schulz KU. Fast Approximate Search in Large Dictionaries. Computational Linguistics. 2004 Dec;30(4).

Levenshtein automata

The basic idea: construct an acceptor that will recognize an input string with up to δ edits.

We have seen something not entirely dissimilar:





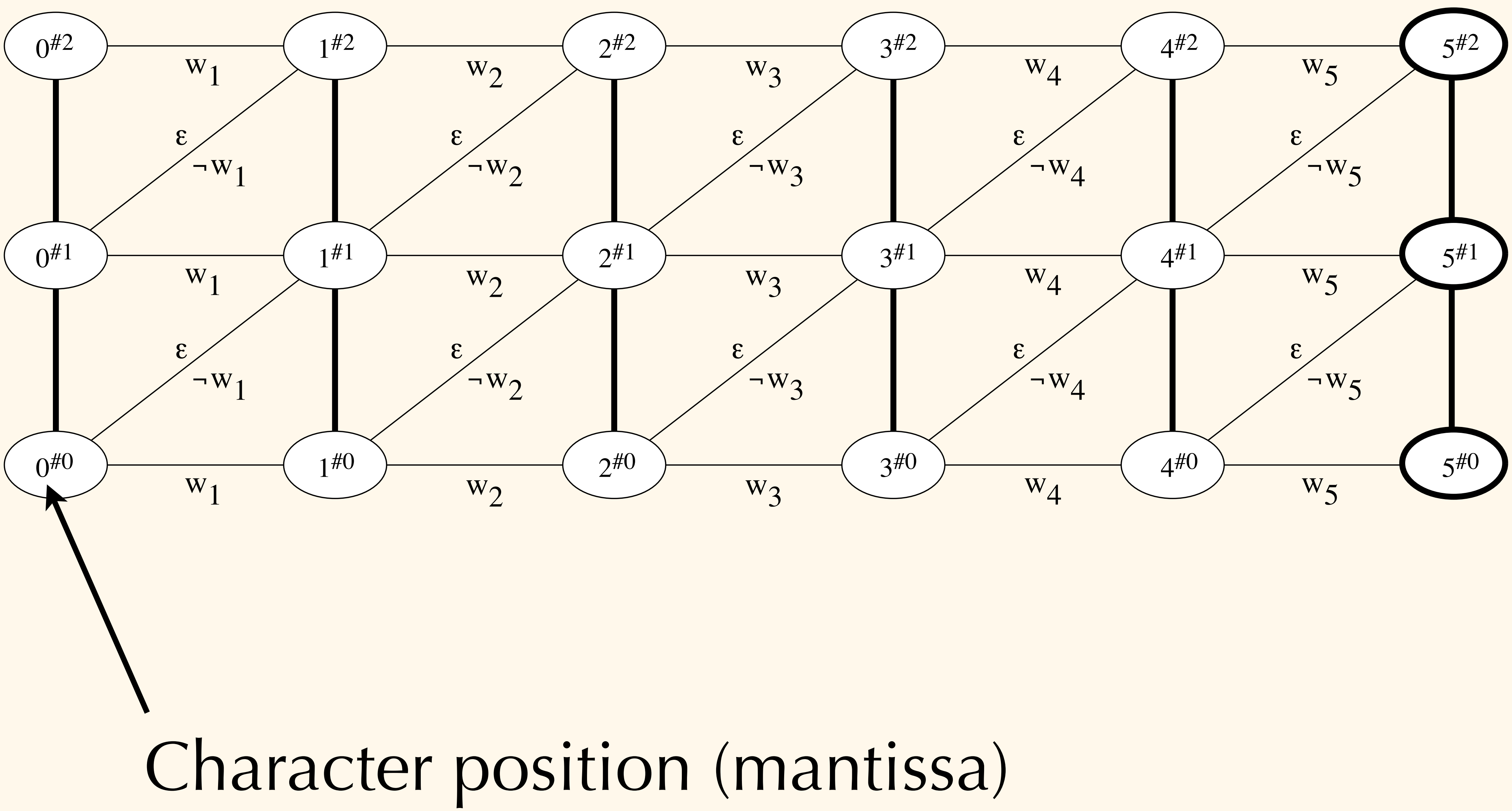
Levenshtein automata

In a sense, our old friend the edit-distance transducer is a step along the path towards a Levenshtein transducer.

The difference: the edit-distance transducer will allow infinite insertions or deletions...

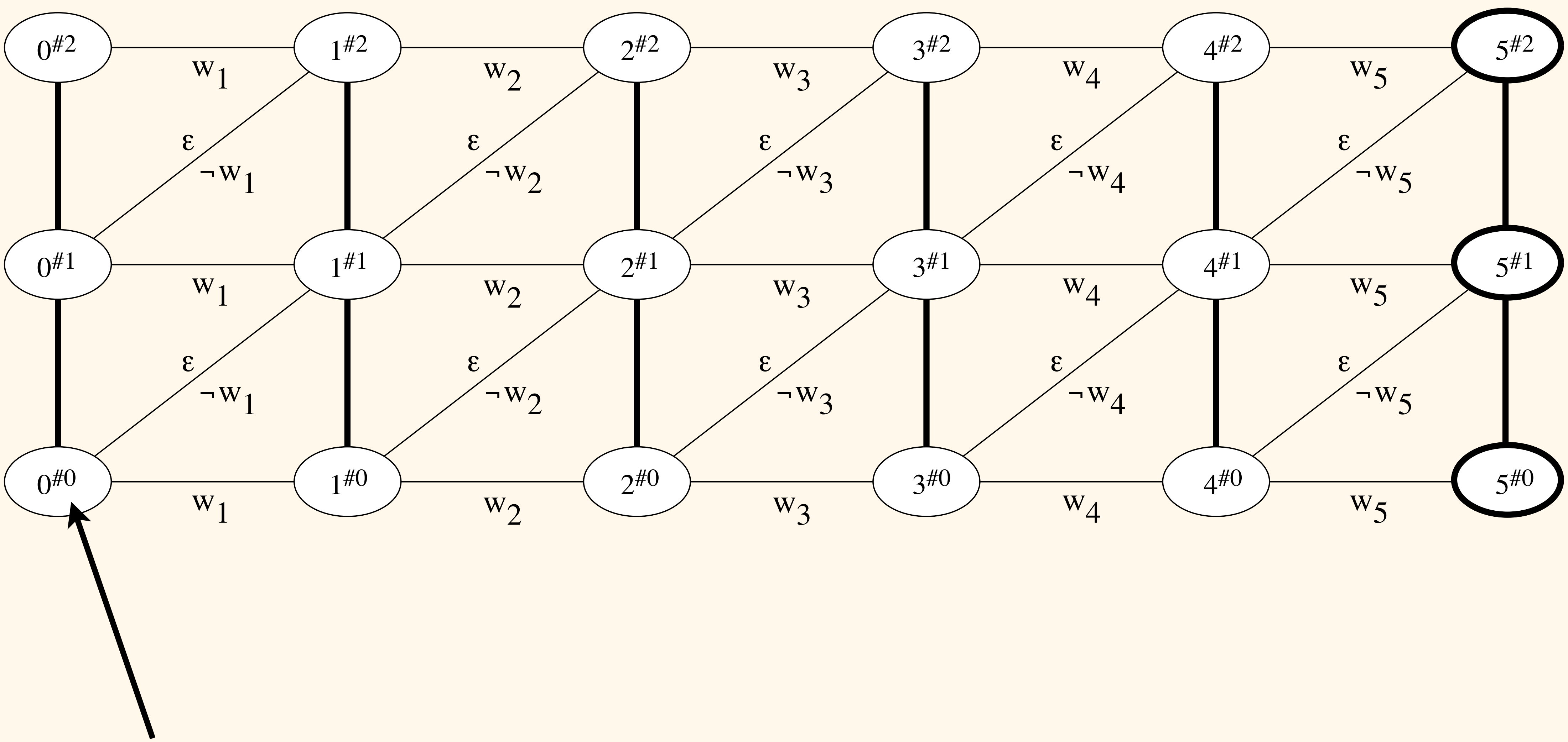
... and we need to limit the total number of such events.

Levenshtein automata



Schulz KU, Mihov S. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition. 2002 Nov 1;5(1):67–85.

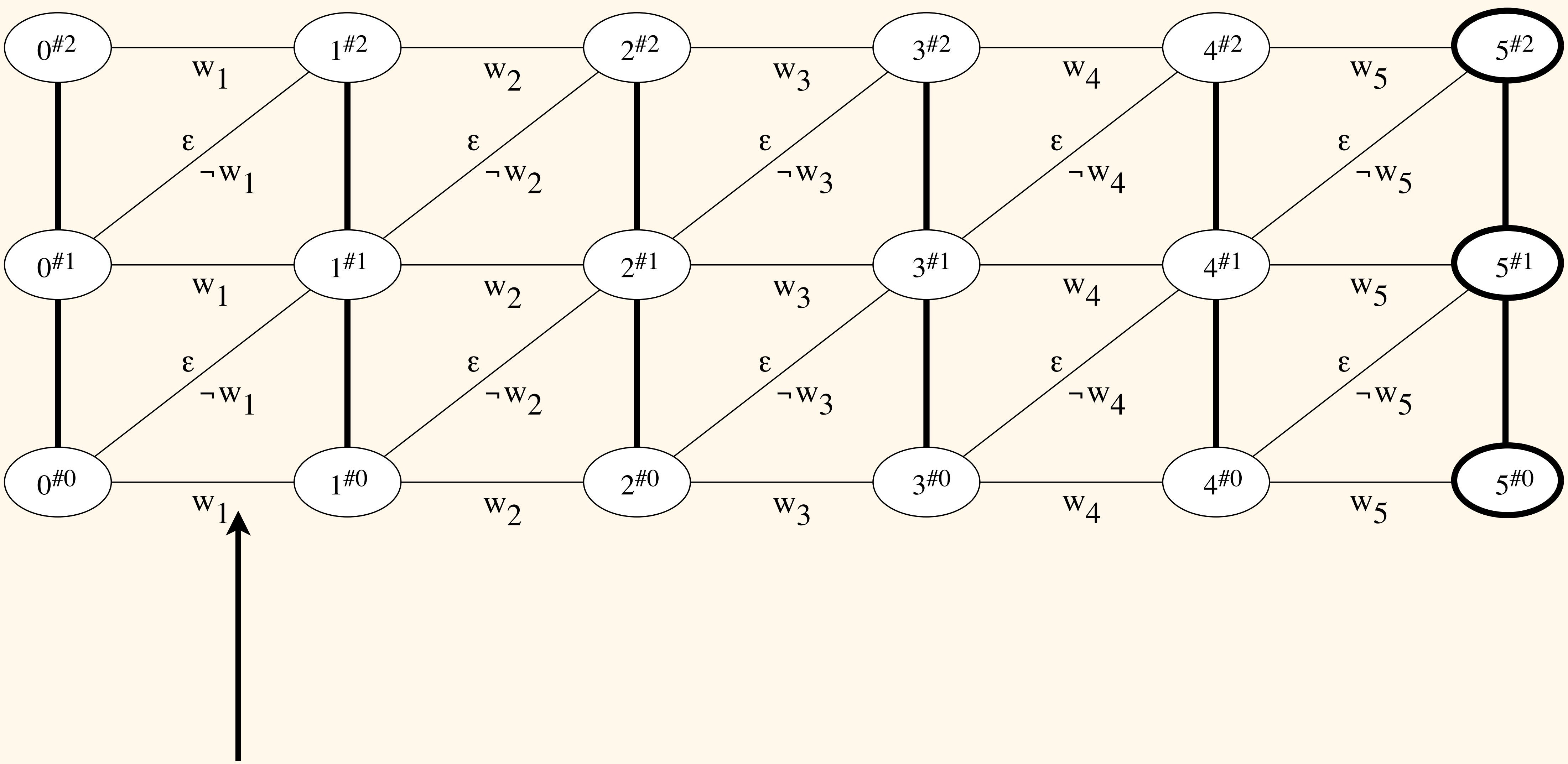
Levenshtein automata



Number of edits thus far (exponent)

Schulz KU, Mihov S. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition. 2002 Nov 1;5(1):67–85.

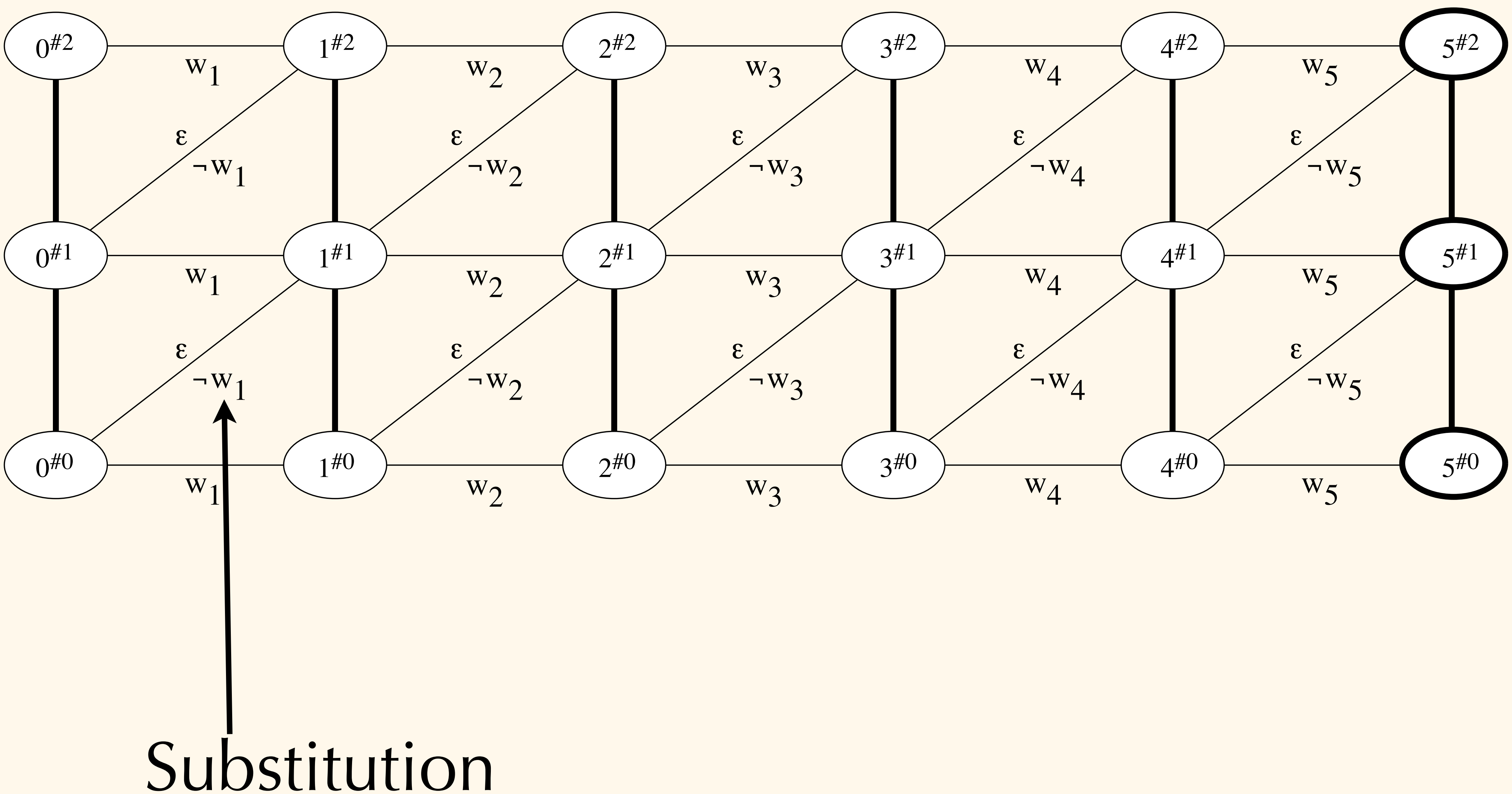
Levenshtein automata



Match

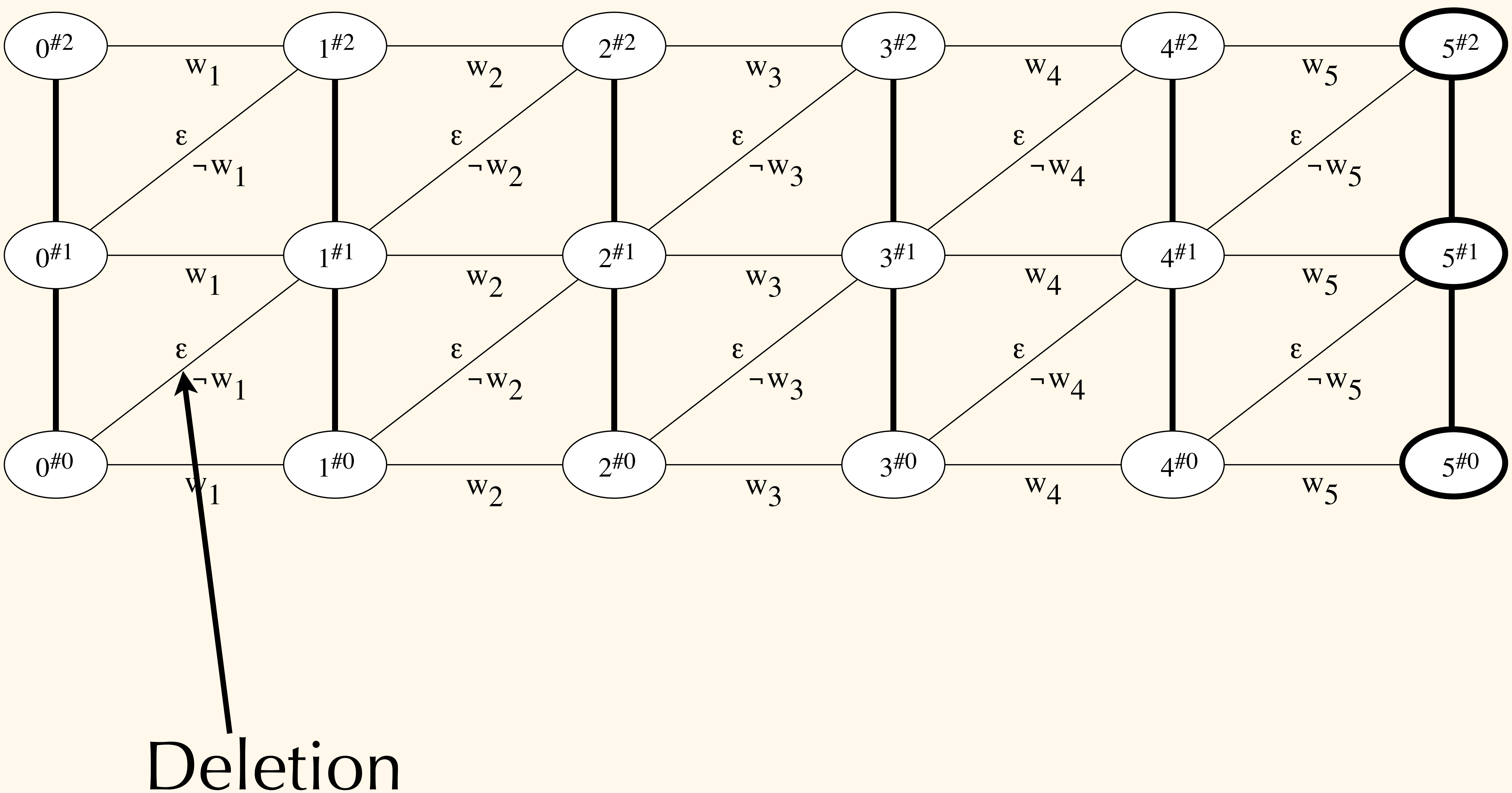
Schulz KU, Mihov S. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition. 2002 Nov 1;5(1):67–85.

Levenshtein automata



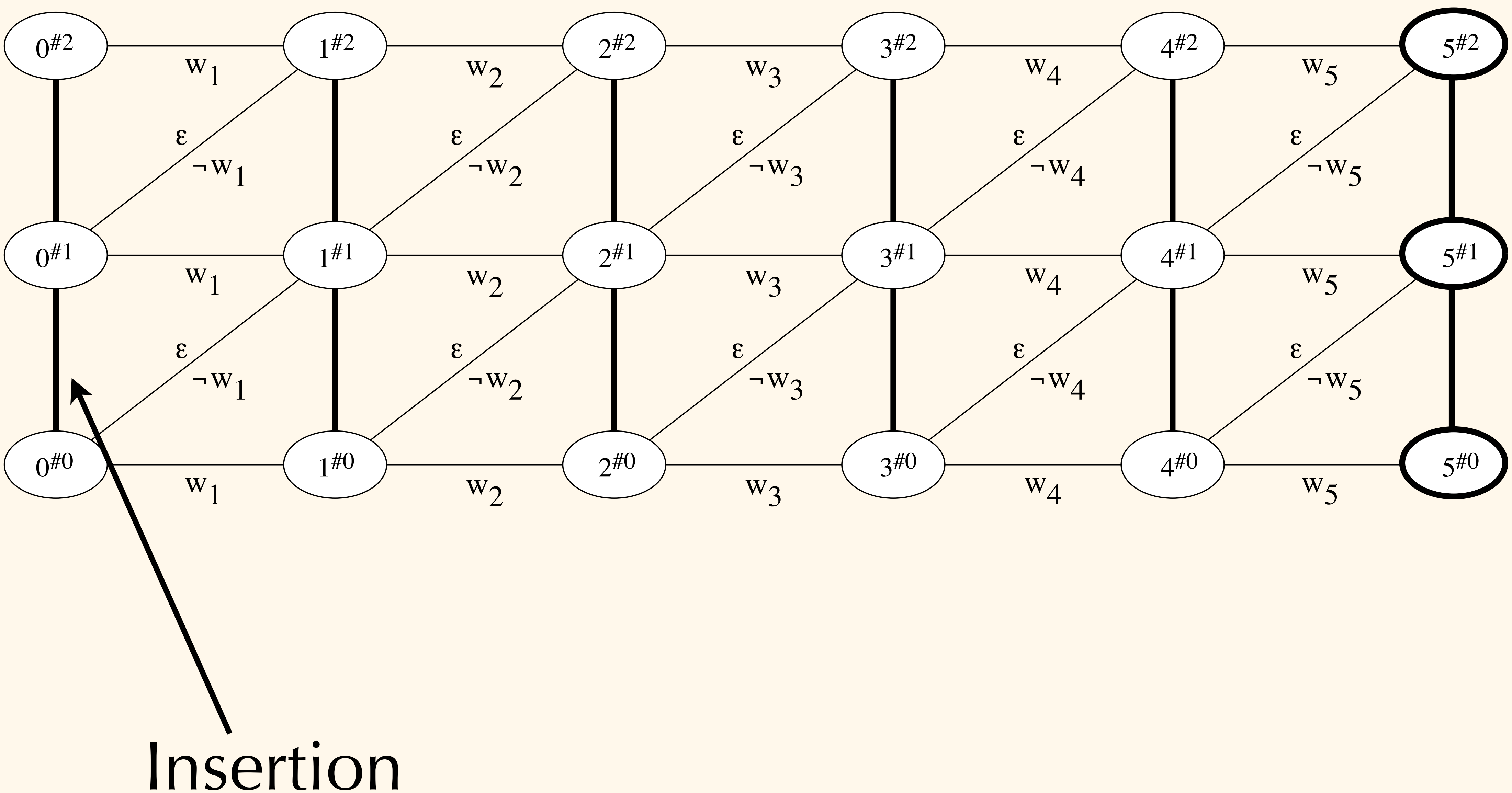
Schulz KU, Mihov S. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition. 2002 Nov 1;5(1):67–85.

Levenshtein automata



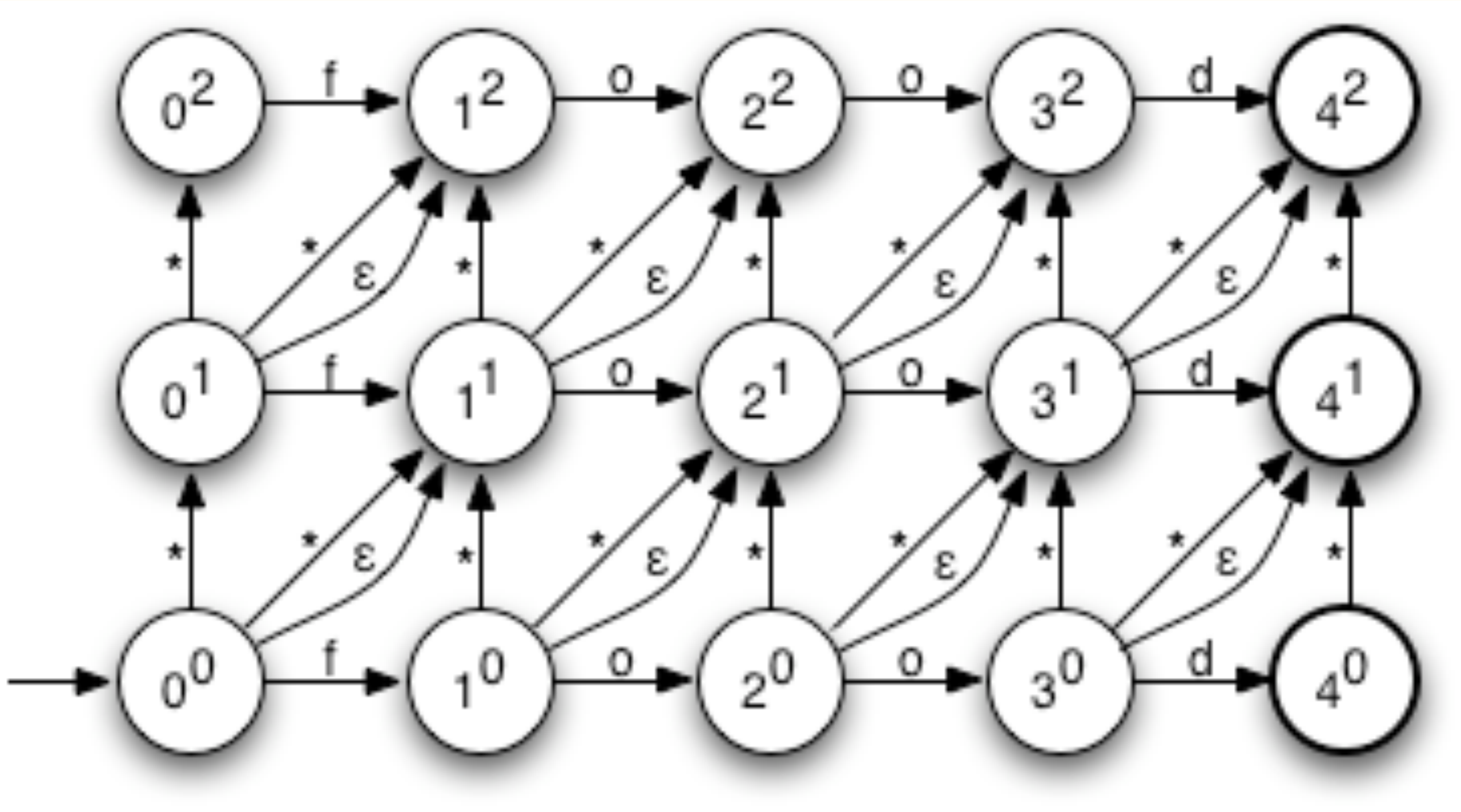
Schulz KU, Mihov S. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition. 2002 Nov 1;5(1):67–85.

Levenshtein automata



Schulz KU, Mihov S. Fast string correction with Levenshtein automata. International Journal on Document Analysis and Recognition. 2002 Nov 1;5(1):67–85.

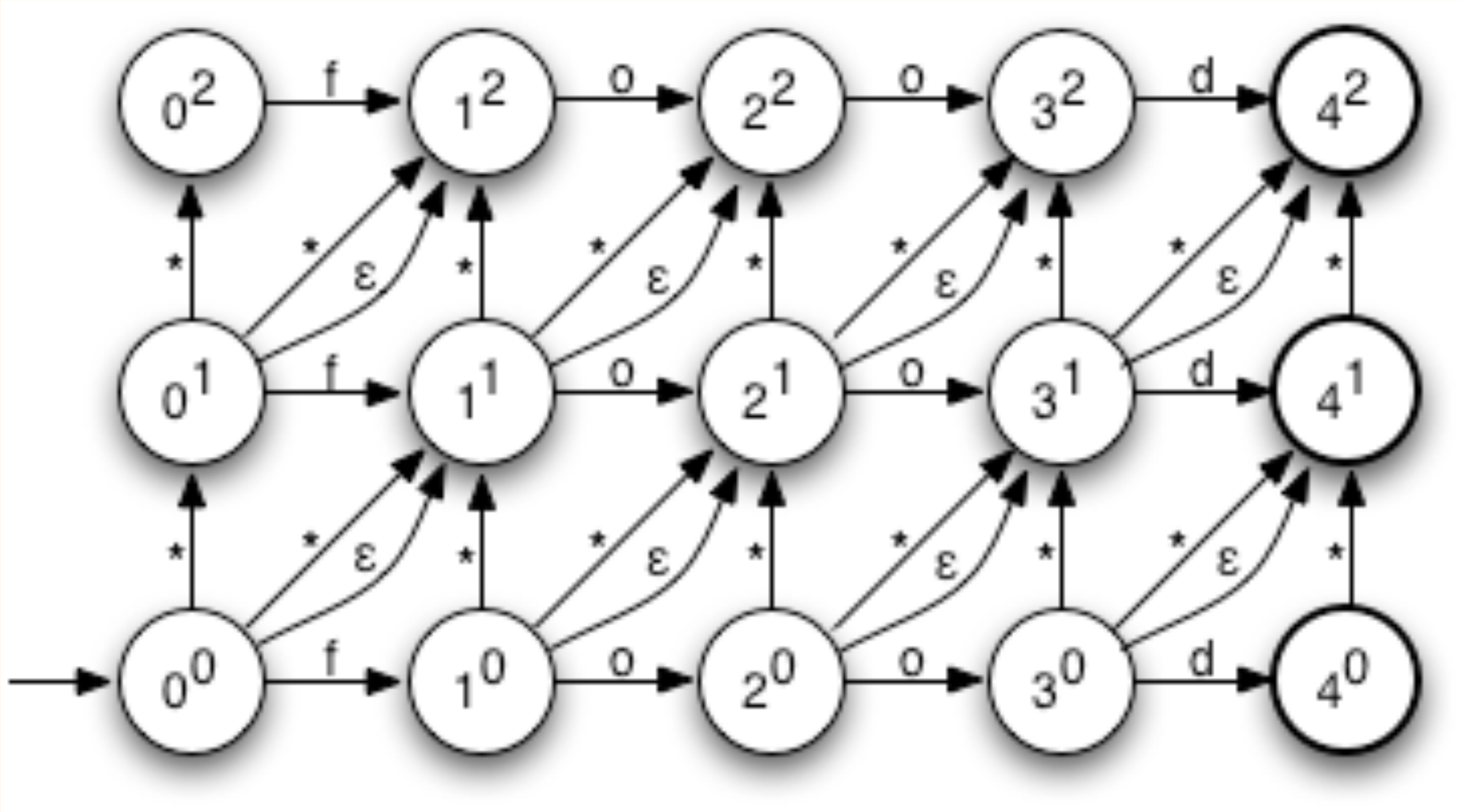
Levenshtein automata



foof doof

http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

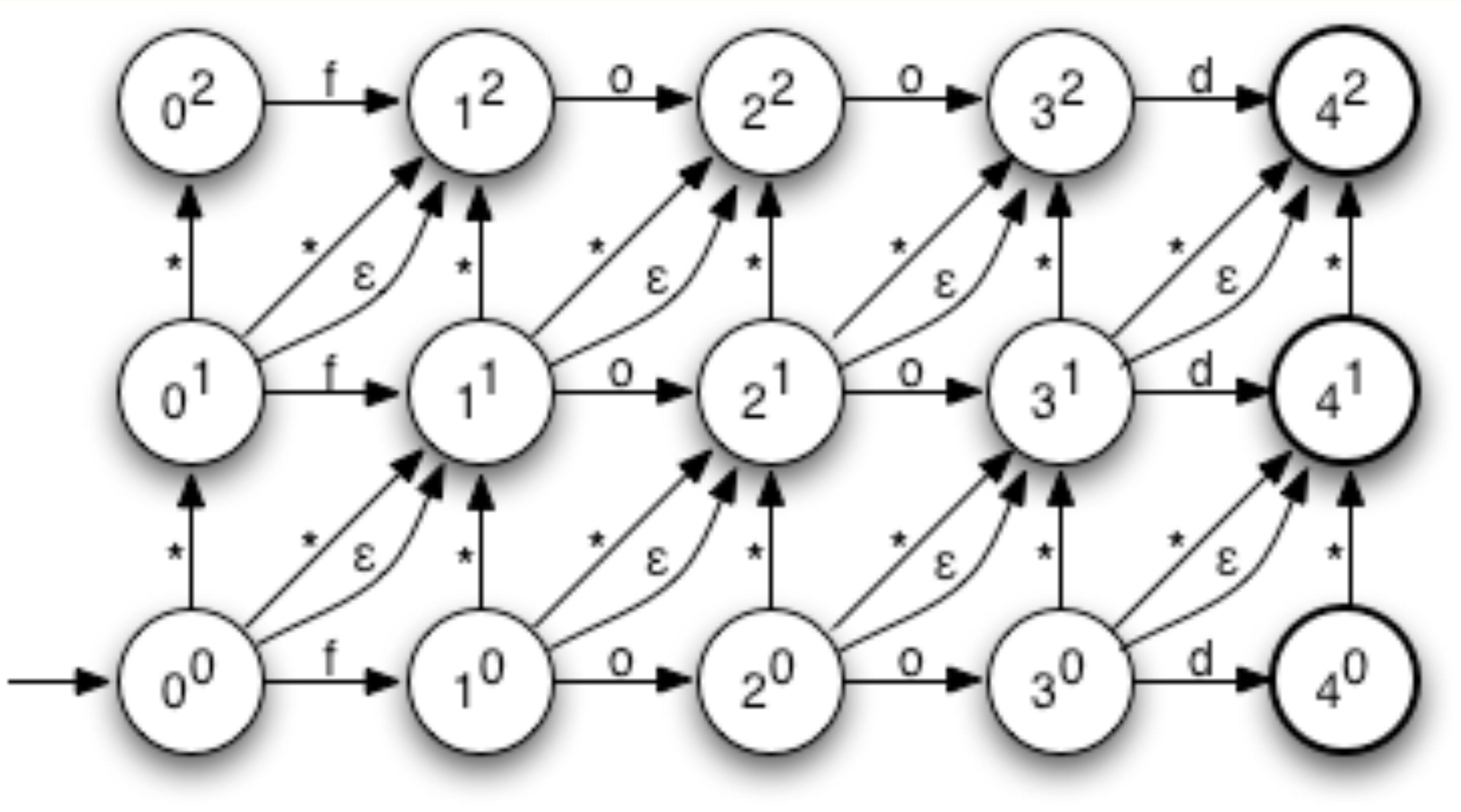
Levenshtein automata



foof   doof

http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

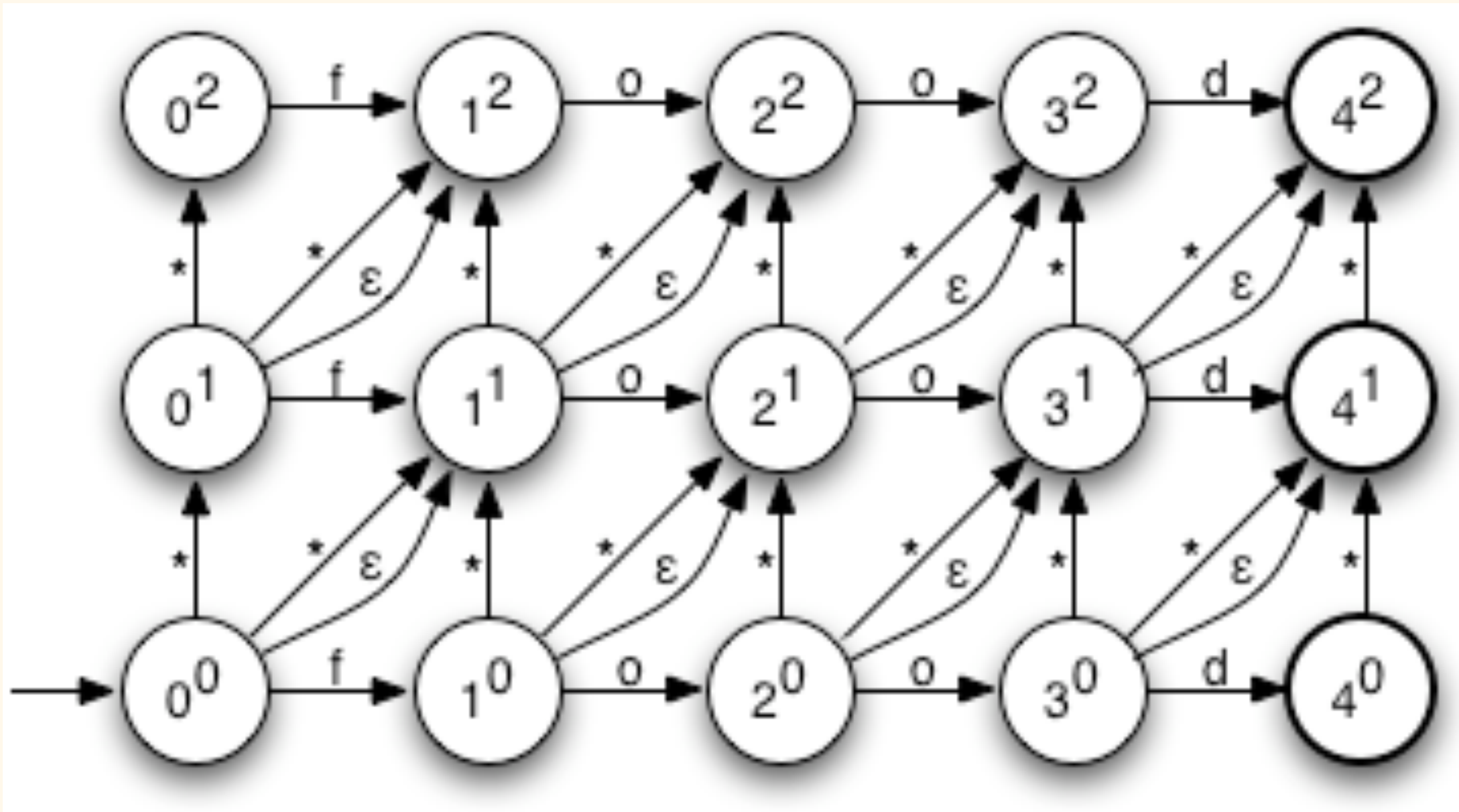
Levenshtein automata



foof doof dora

http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

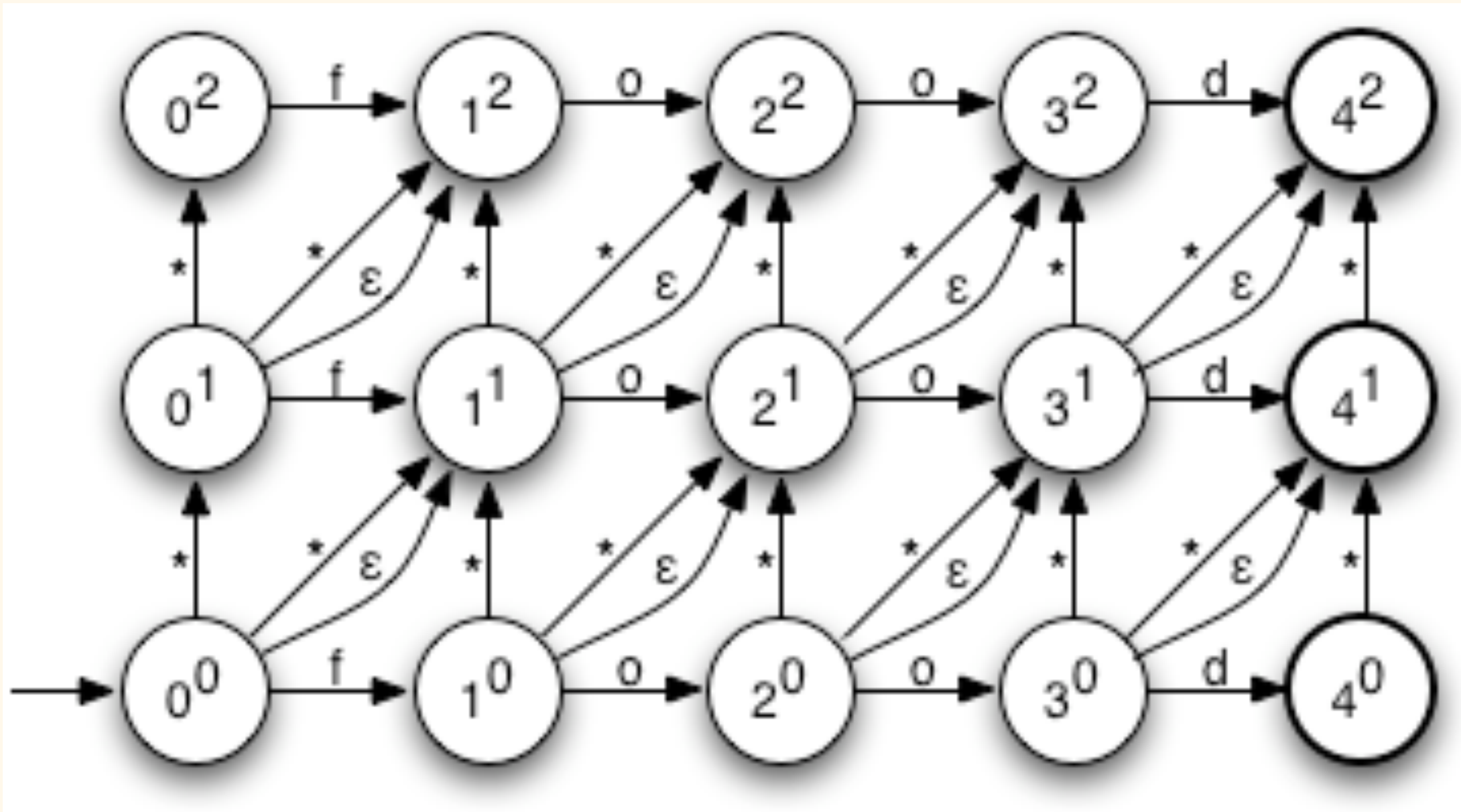
Levenshtein automata



foof doof ~~dora~~

http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

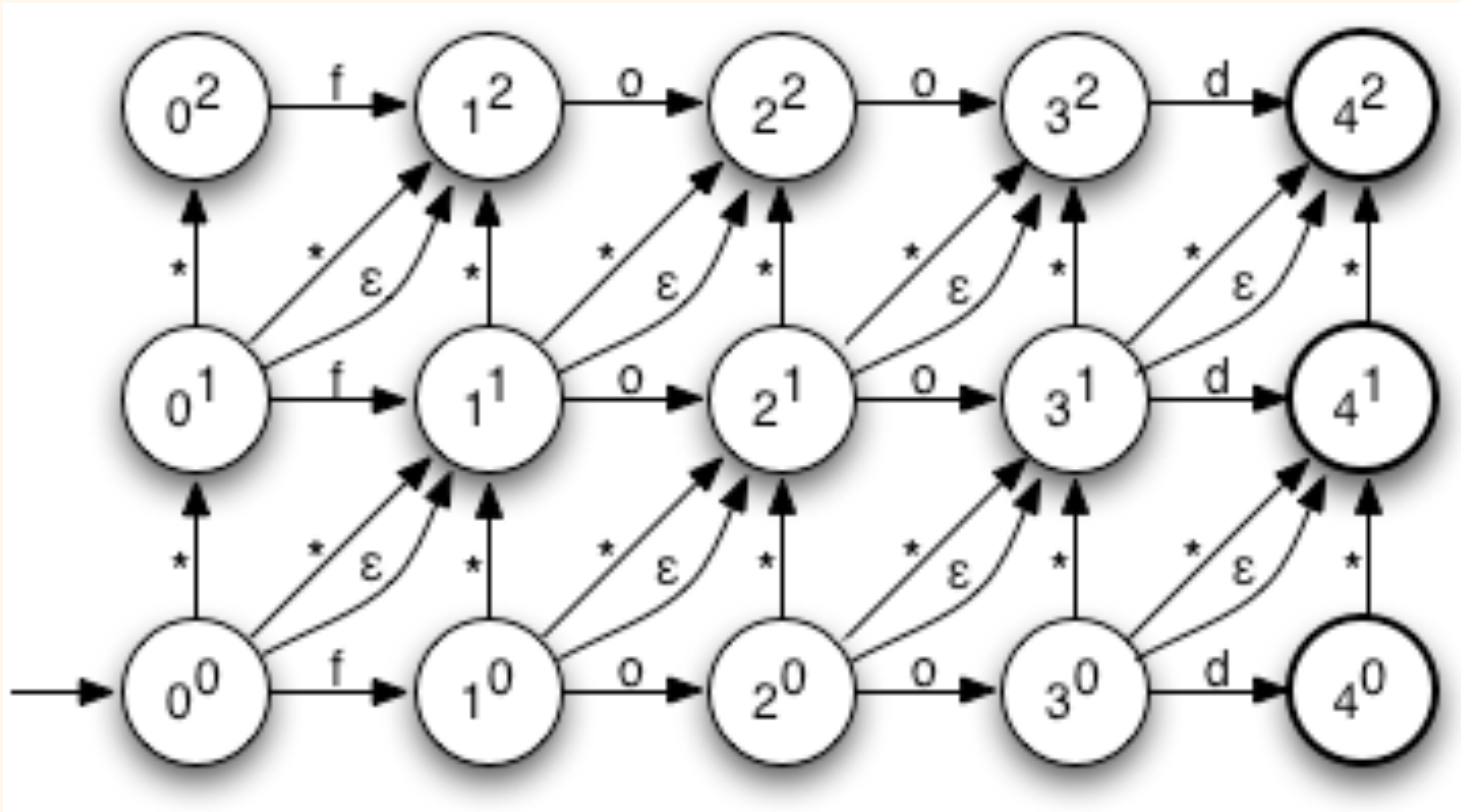
Levenshtein automata



foof   doof   ~~dora~~   foods

http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

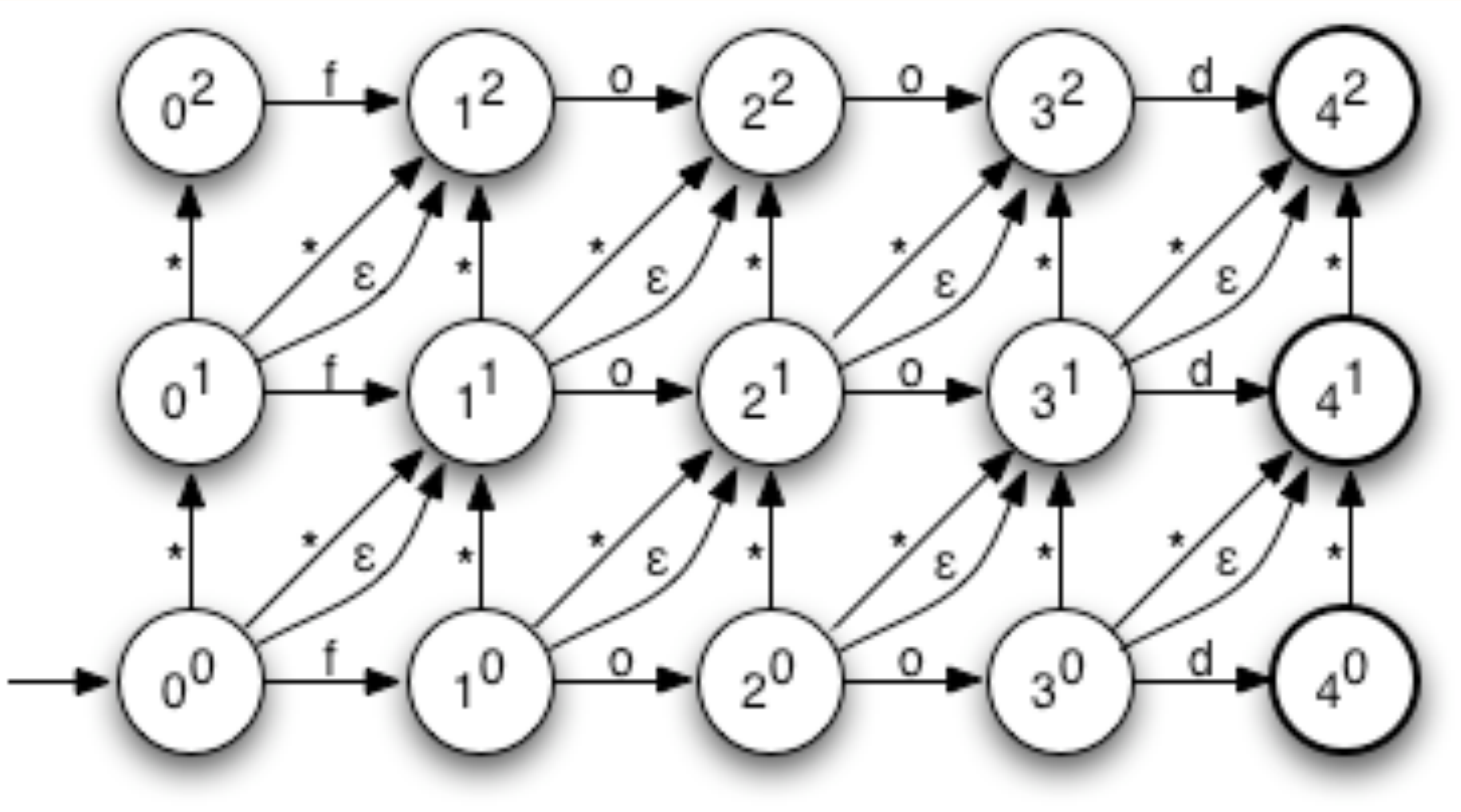
Levenshtein automata



foof doof ~~dora~~ foods feed

http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

Levenshtein automata



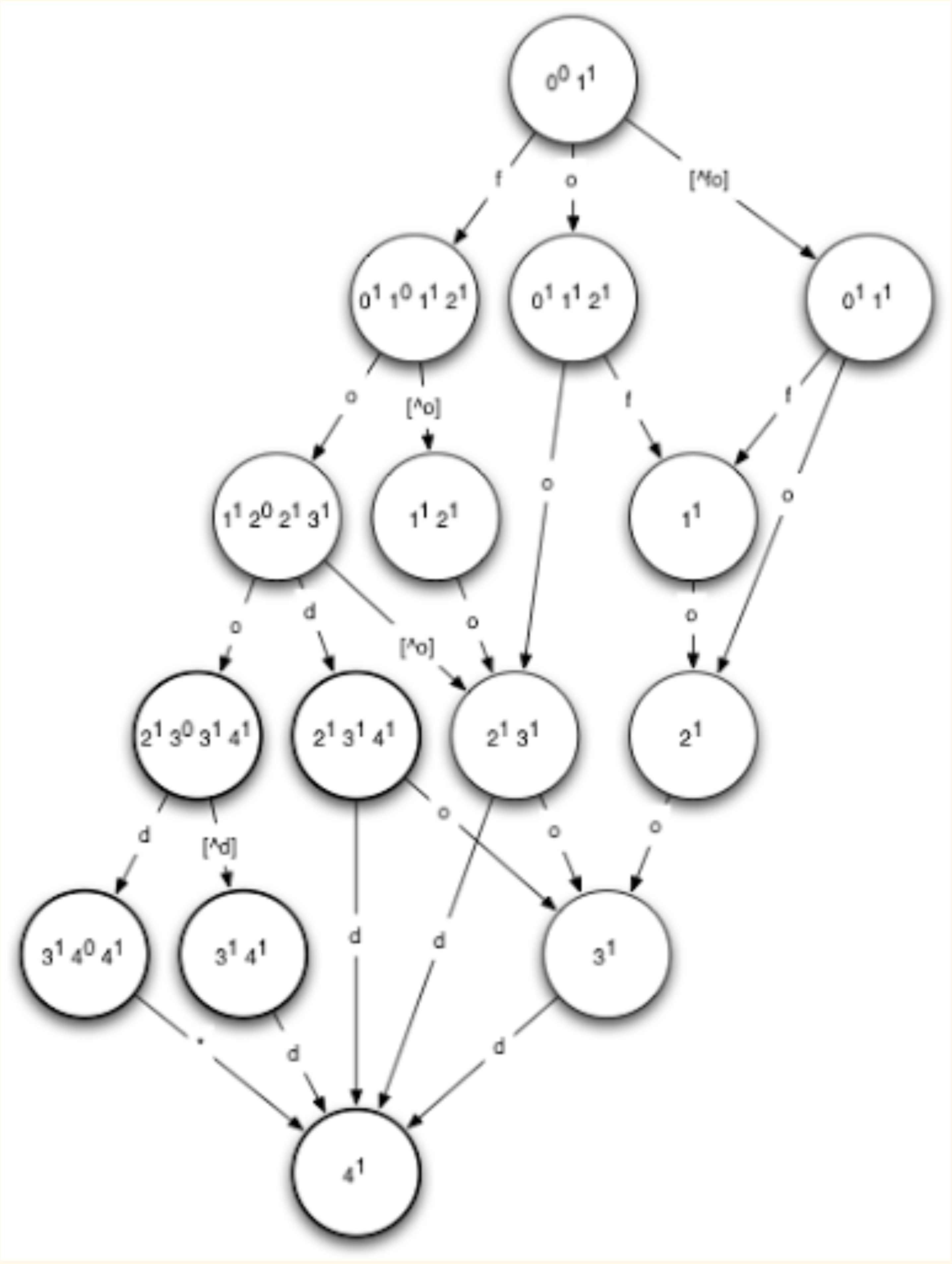
foof doof ~~dora~~ foods feed

http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

Levenshtein automata

This is a non-deterministic representation; actually using NFAs in practice is often tricky.

Luckily, NFAs can be determinized, which is generally how Levenshtein automata are actually used.



http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

Levenshtein automata

On its own, having a Levenshtein automaton of a query word improves even the naïve approach (pairwise comparison):

Instead of a large set of O(nm) computations, we have a large set of O(n) computations!

Levenshtein automata

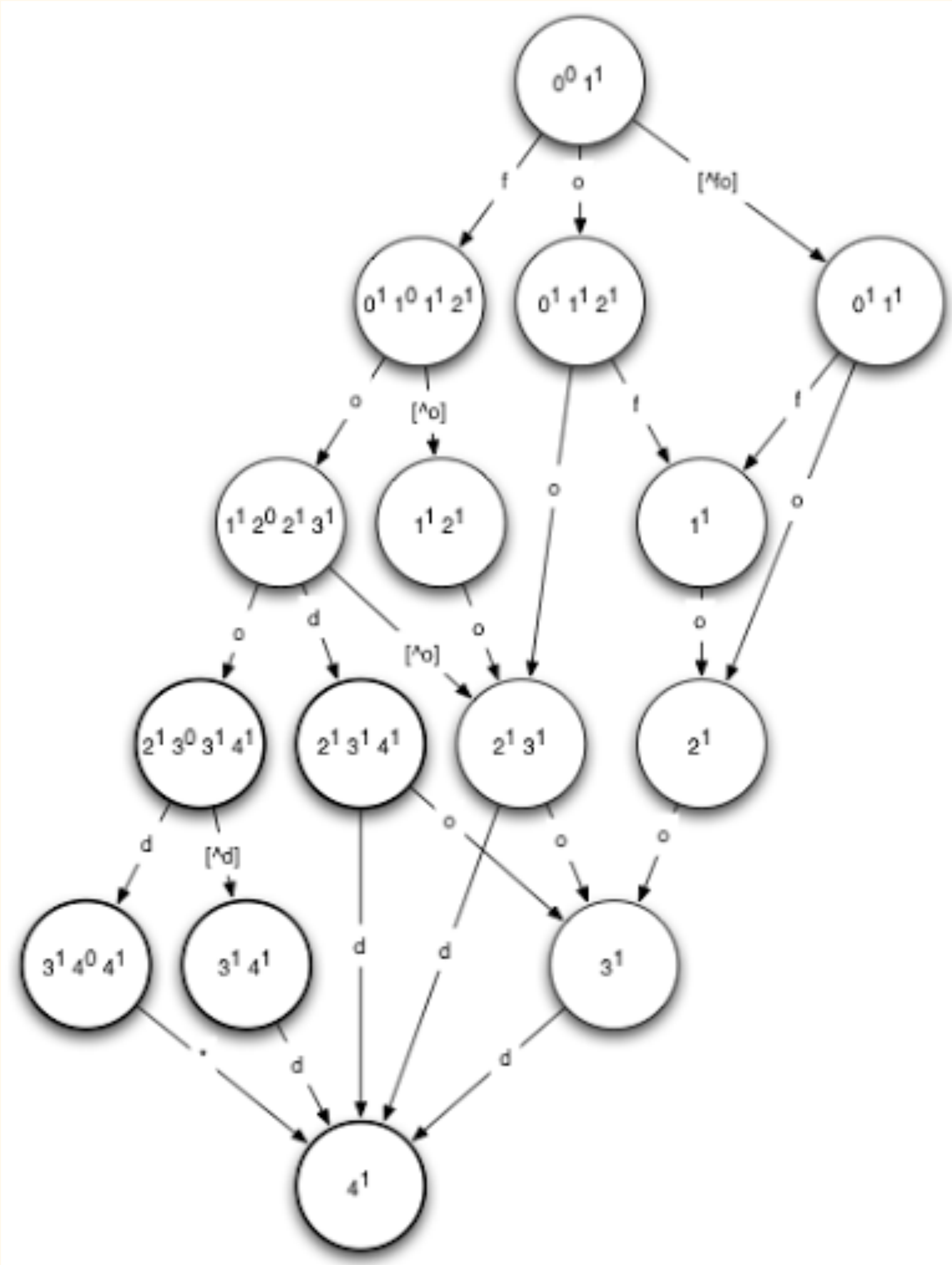
We can do better, however.

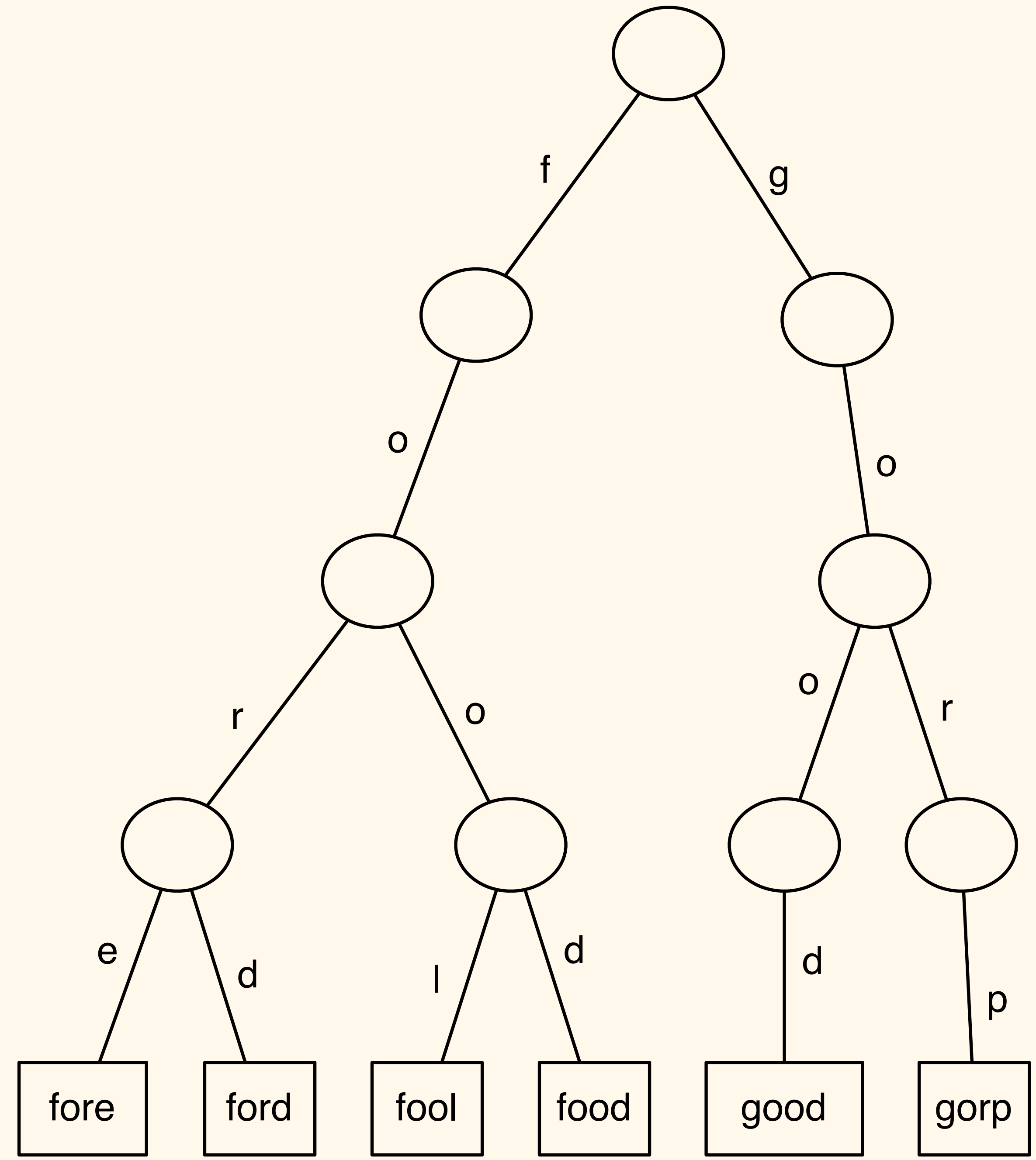
Represent our dictionary as a trie, DAWG, etc....

... and walk through it and our determinized automaton together in tandem.

At each state we encounter, follow edges that both have in common.

Any time both are in final states, we've got a match!


$0^0 1^1$
f
o
[^fo]
$0^1 1^0 1^1 2^1$
$0^1 1^1 2^1$
$0^1 1^1$
o
[^o]
o
f
f
o
$1^1 2^0 2^1 3^1$
$1^1 2^1$
1^1
o
d
[^o]
o
o
$2^1 3^0 3^1 4^1$
$2^1 3^1 4^1$
$2^1 3^1$
2^1
d
[^d]
o
o
o
d
d
$3^1 4^0 4^1$
$3^1 4^1$
3^1
d
d
4^1


f
g
o
o
r
o
o
r
e
d
l
d
d
p
fore
ford
fool
food
good
gorp