EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Knuth-Morris-Pratt Algorithm

- Computing π[j] from P[0:m-1]:

```
k= 0; j = 1;
π[0]=0;
While j < m:
  If P[k]==P[j]:
    k++; π[j]=k; j++;
 else if k == 0:  # first character did not match
    π[j] = 0; j++;
 else:  #mismatch after first character
    k = π[k-1]
```

w	c	a	b	c	a
arr	0	0	0	1	2

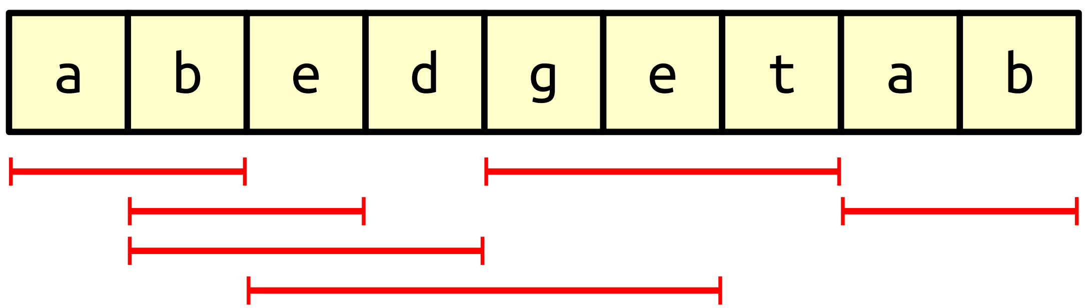
P	A	A	B	A	A	C	A	A	B
Pi	0	1	0	1	2	0	1	2	3
K	0	1	0	1	2	0	1	2	3
J	1	2	3	4	5	6	7	8	9

Search for Multiple Strings

- KMP good for 1 pattern P
- What if we are looking for k patterns $P_1, P_2, \ldots, P_k$?

Pattern Strings

a	b			
a	b	o	u	t
a	t			
a	t	e		
b	e			
b	e	d		
e	d	g	e	
g	e	t		

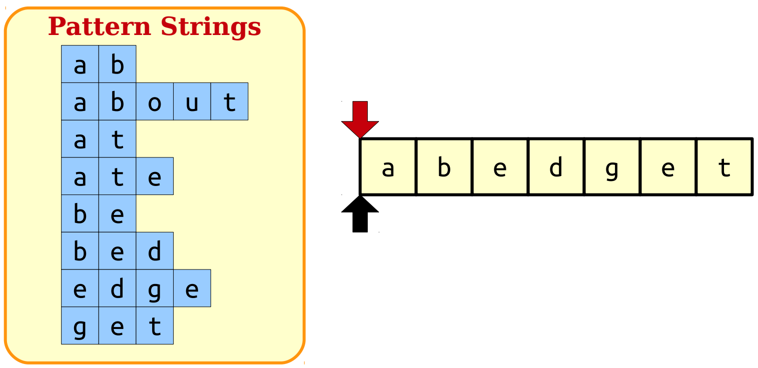


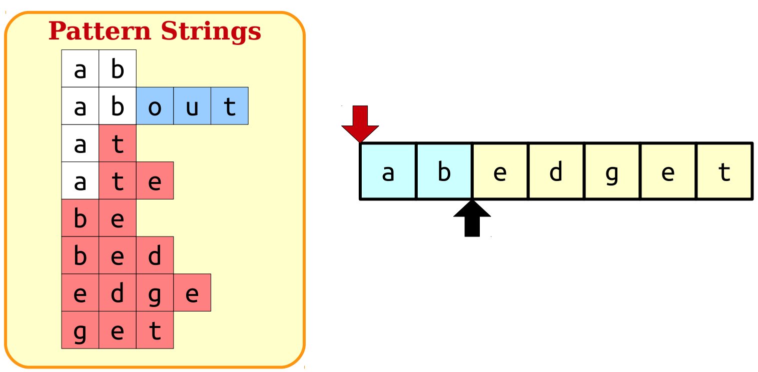
Search for Multiple Strings

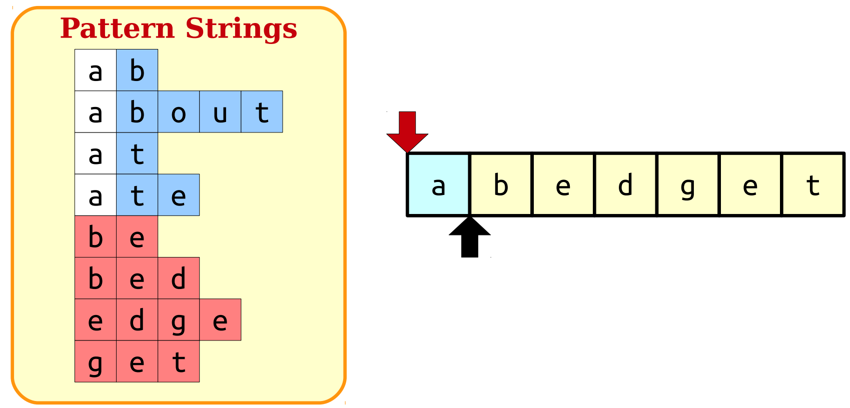
- Notation: m is T string length; n is total length of k patterns; L_{max} length of longest pattern
- Naive approach:
  - For each position in T:
    - For each pattern string P_j:
      - Check if P_j appears at that position
  - $\Theta(mn)$
  - Can we do better?

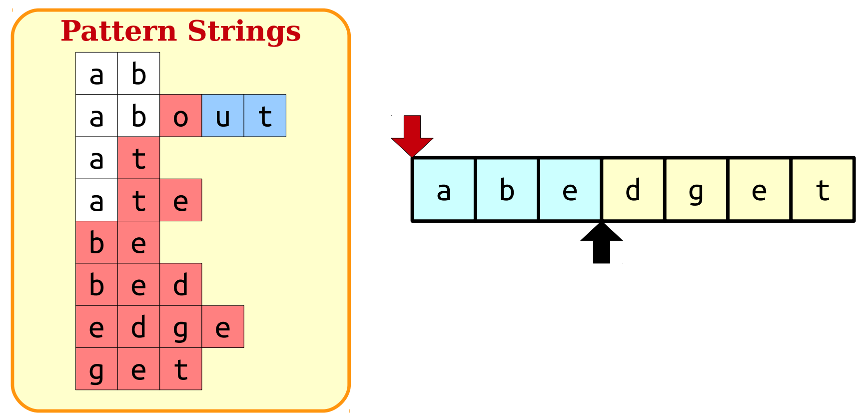
Search for Multiple Strings

- Can we do better?







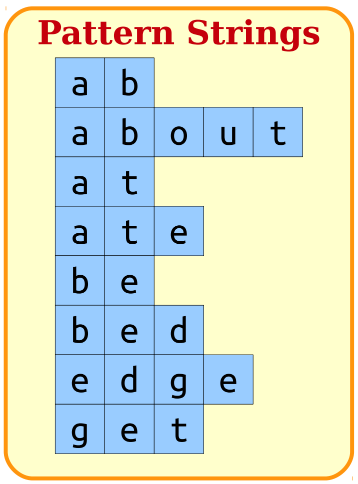


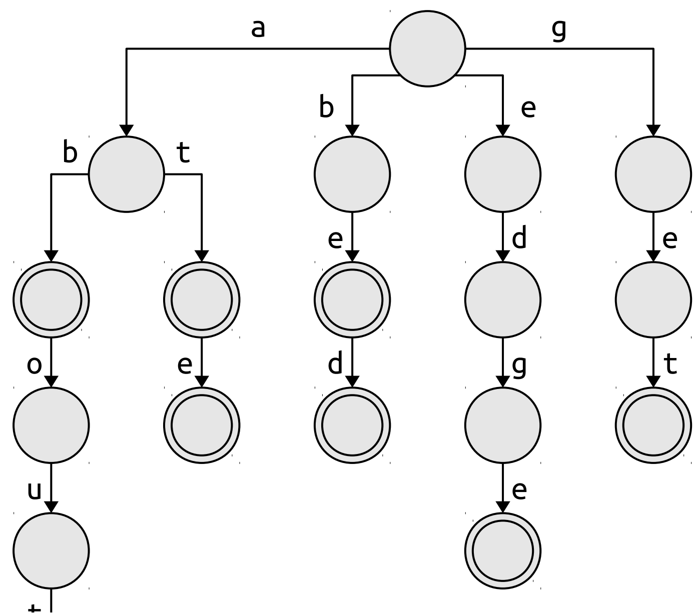
Search for Multiple Strings

- Idea: Search for all strings in parallel
- How? Form the strings into a **trie**: word comes from retrieval
- A trie: a tree structure where the branches at each level correspond to symbol values
  - Example trie: index tables at end of documents
  - Important idea: In a trie, you wait to subdivide a node until there is a collision of two elements…saves storage! Can identify nodes with key words as potential end nodes

Search for Multiple Strings

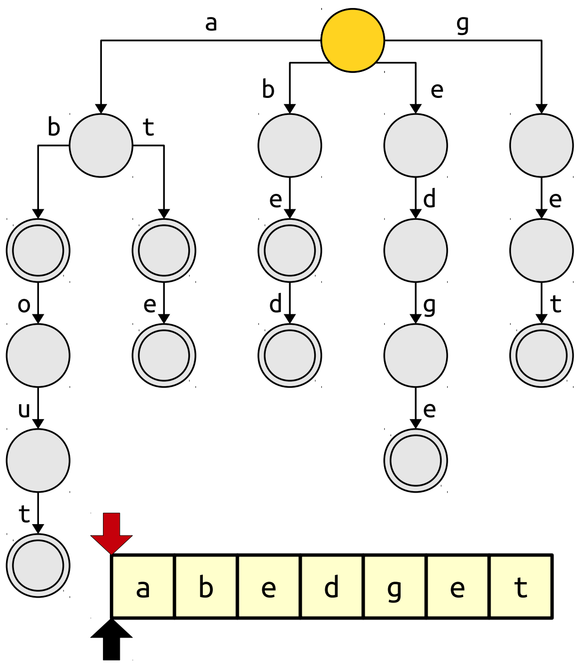
- Example trie for words

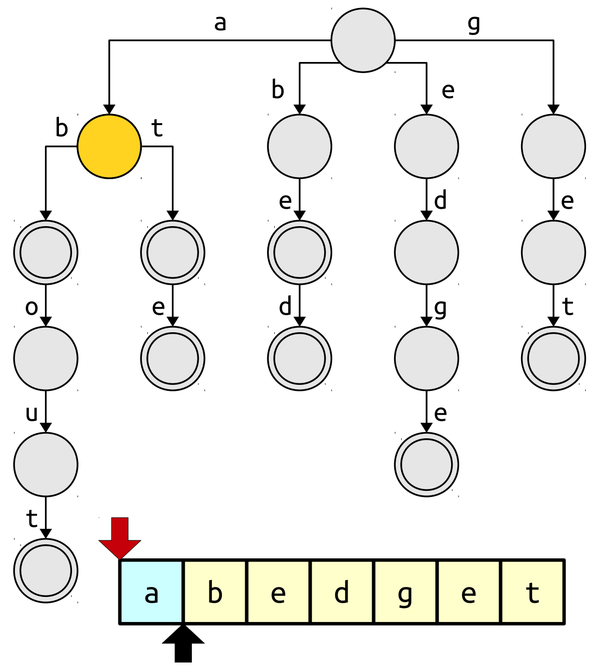




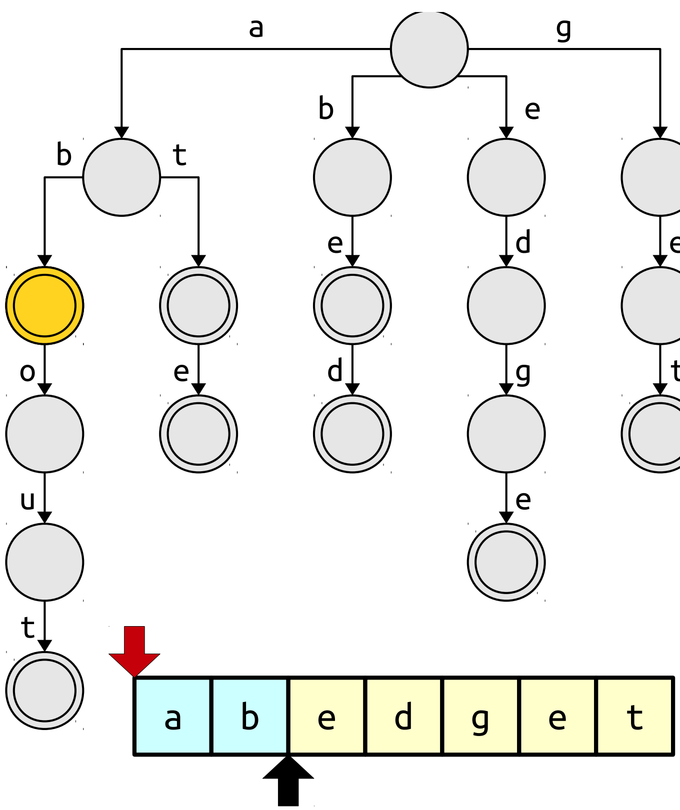
Representing Tries

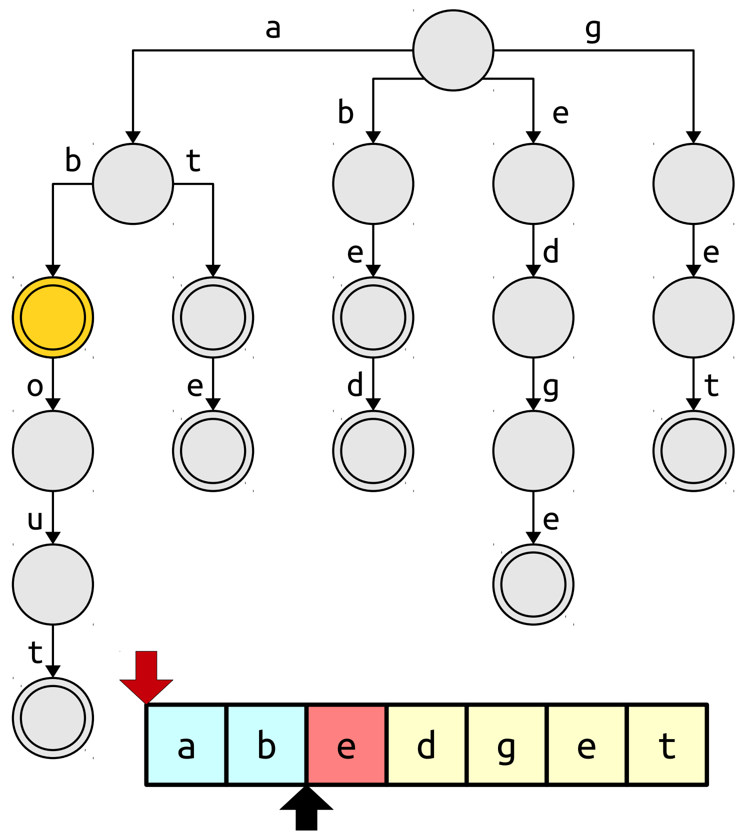
- Each trie node has pointers to its possible children
  - Assume you have an array of pointers at each node, many of which could be null
- Matching



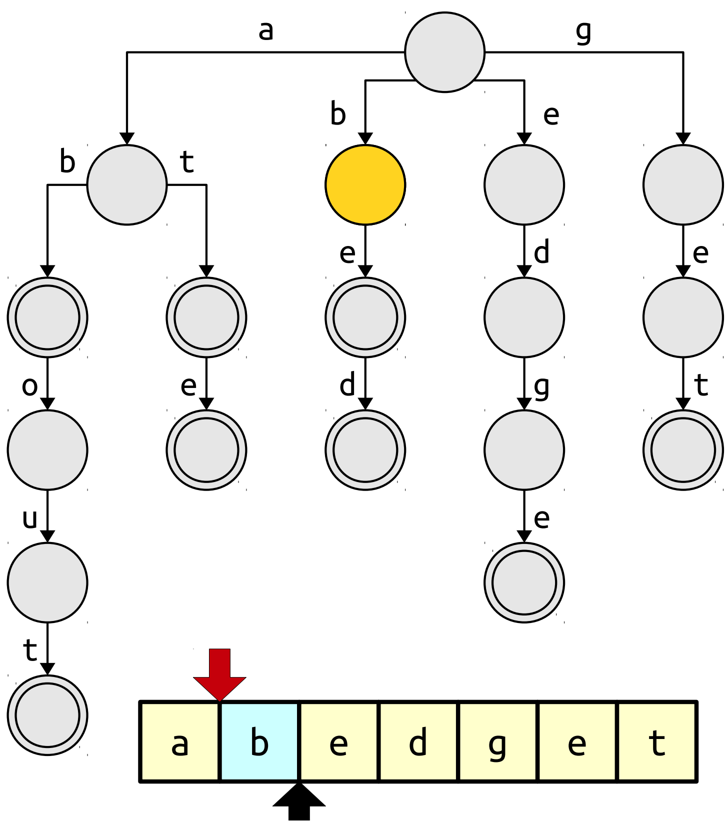


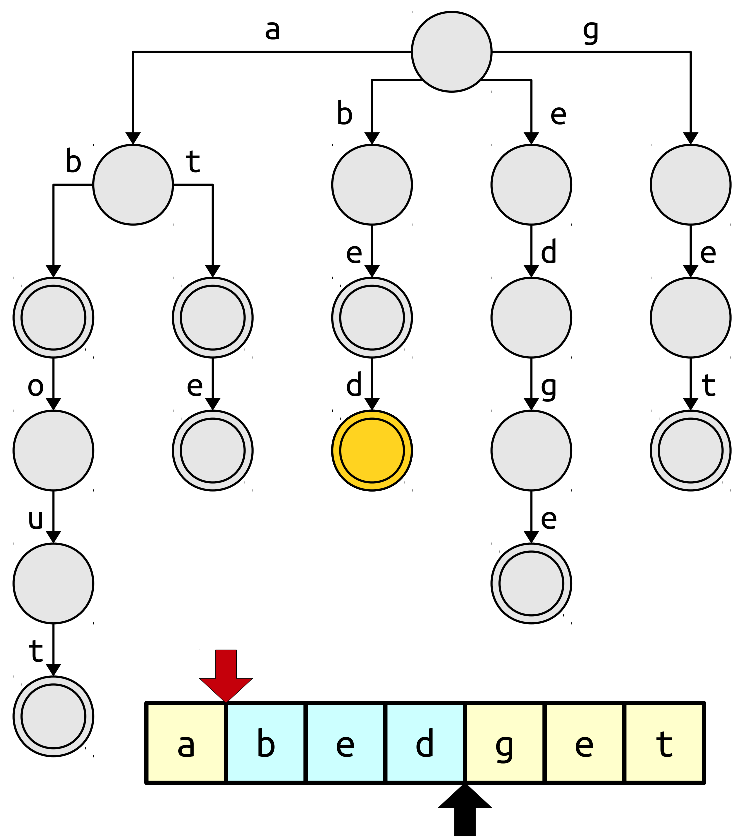
Matching Tries





Matching Tries





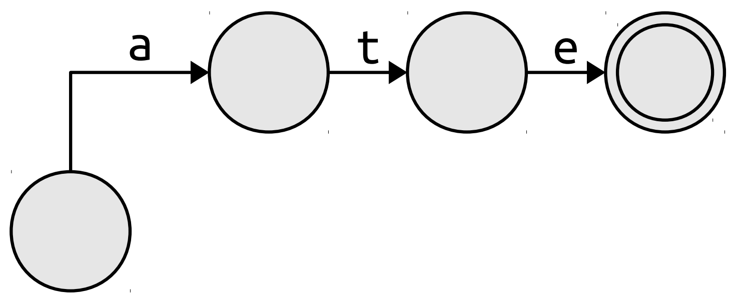
Analysis of algorithm

- Complexity: $\Theta(mL_{max})$
  - Good reduction if n is large (several patterns)
- But: How long to build a trie? Need that preprocessing step

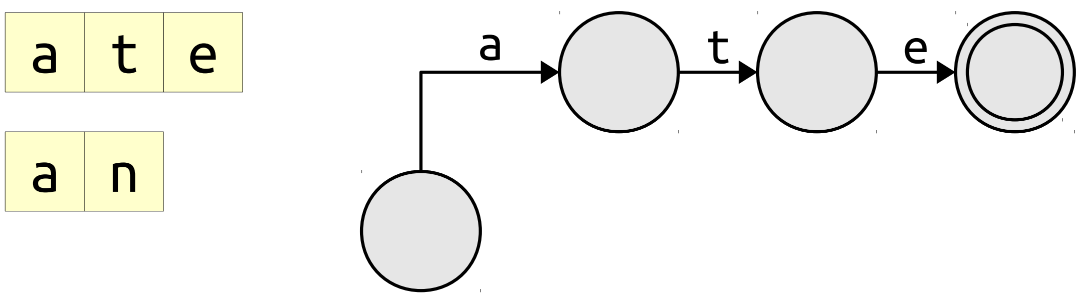
- Claim: Given a set of strings $P_1, P_2, \ldots, P_k$ of total length n, it's possible to build a trie for those strings in time $\Theta(n)$

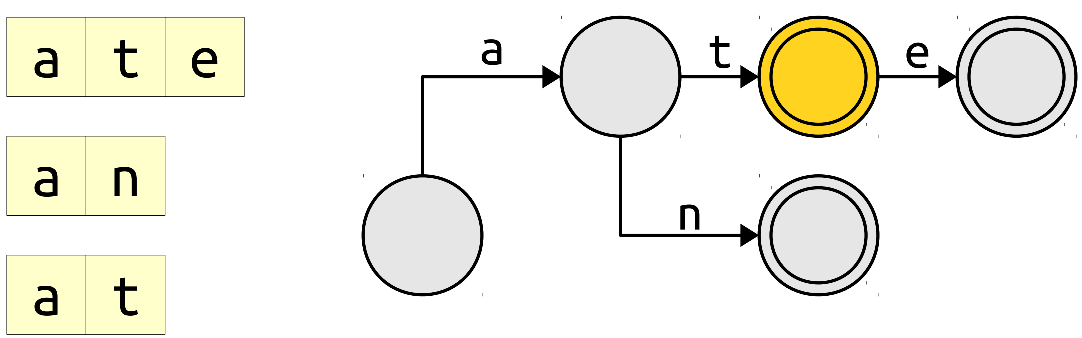
a	t	e

a	n



Building Tries

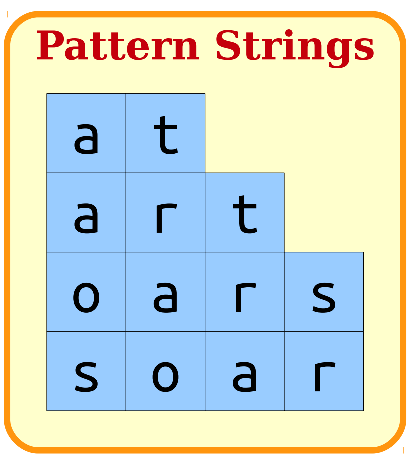


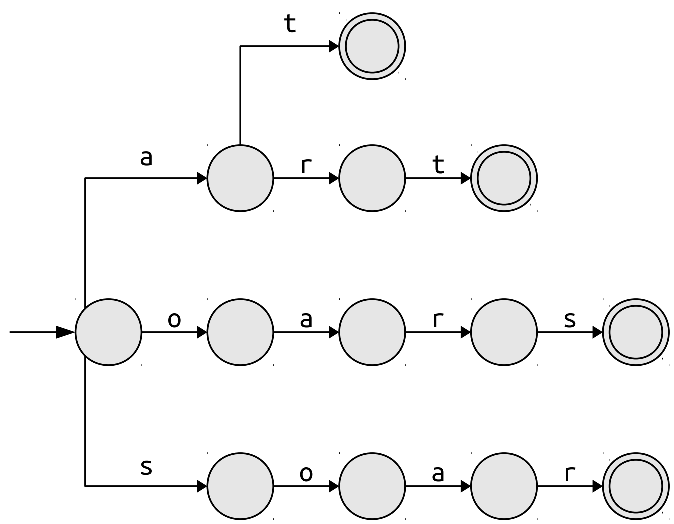


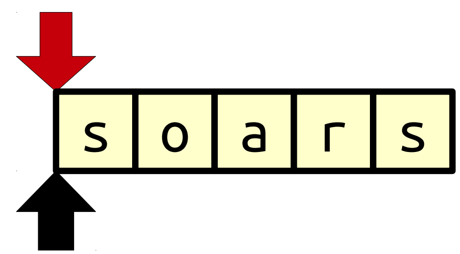
Analysis of algorithm

- Two algorithms so far
  - Naive algorithm: no preprocessing run time $\Theta(mn)$
  - Trie algorithm: $\Theta(n)$ preprocessing, $\Theta(mL_{max})$ run time

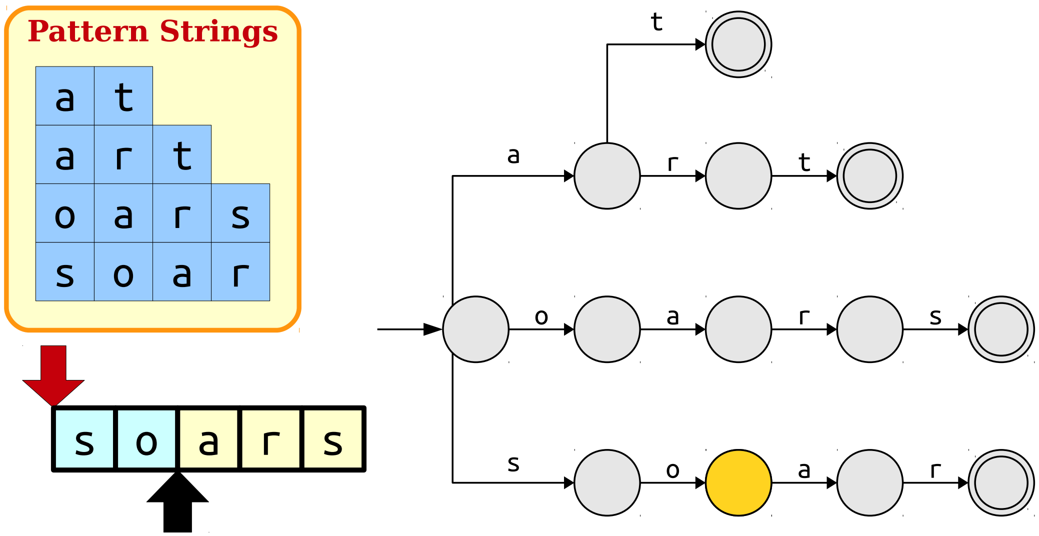
- Can we do better? Can we exploit prefix functions, as in KMP?

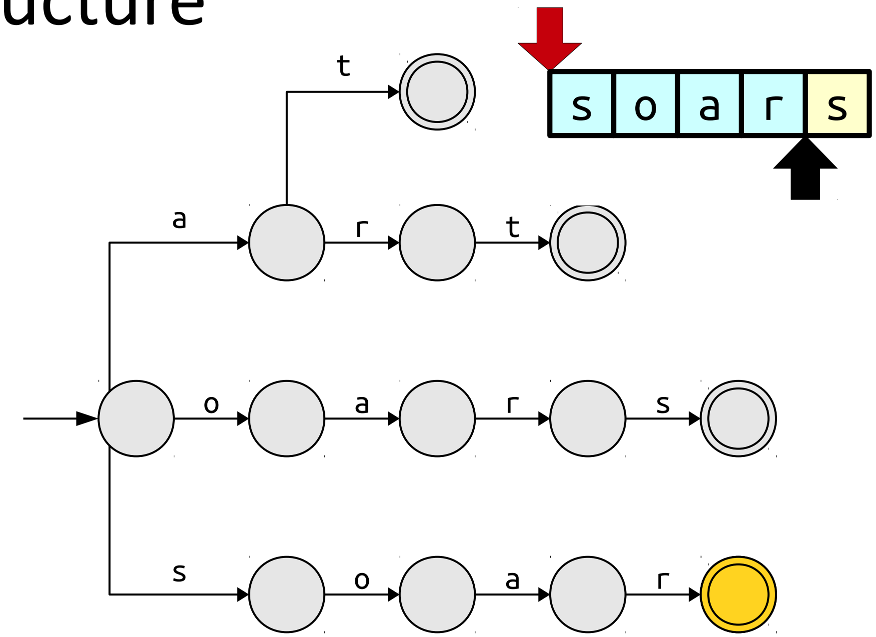






Exploiting structure

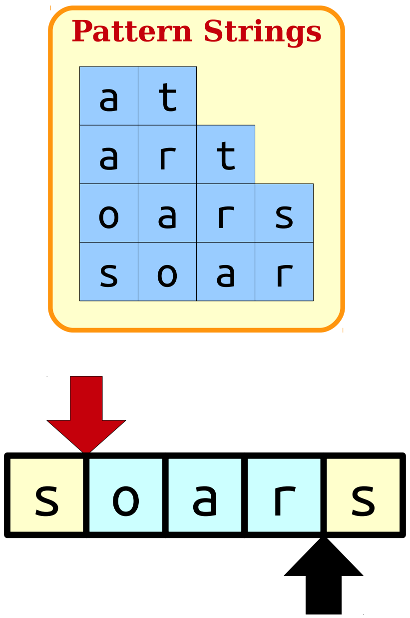


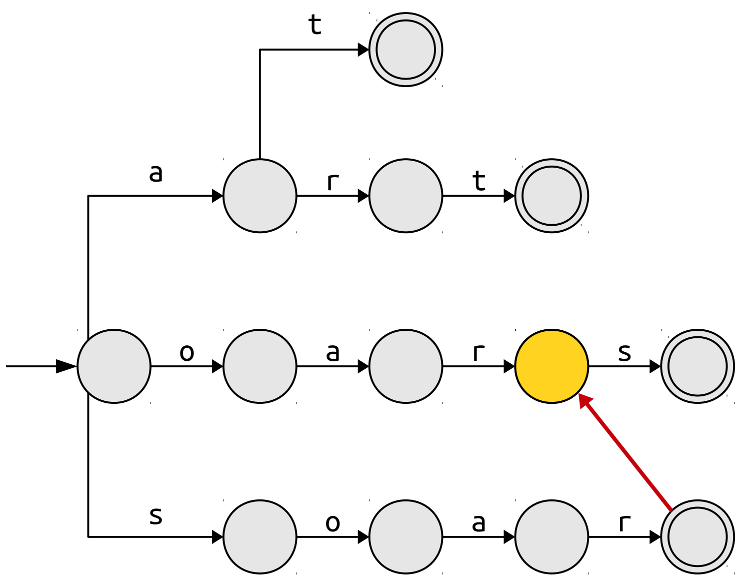


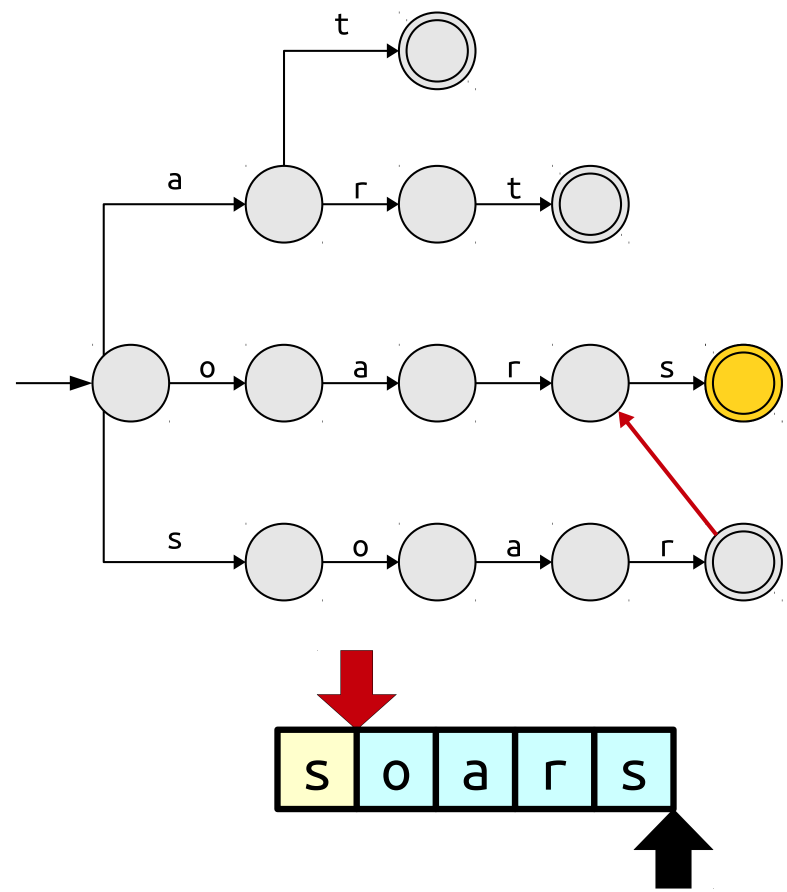
Observe: Suffix oar is prefix in oars

Want to exploit that!

Exploiting structure

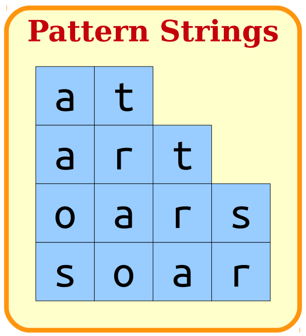


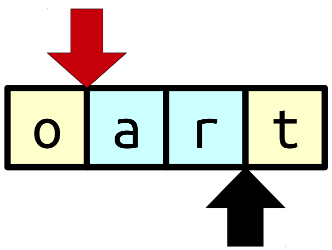


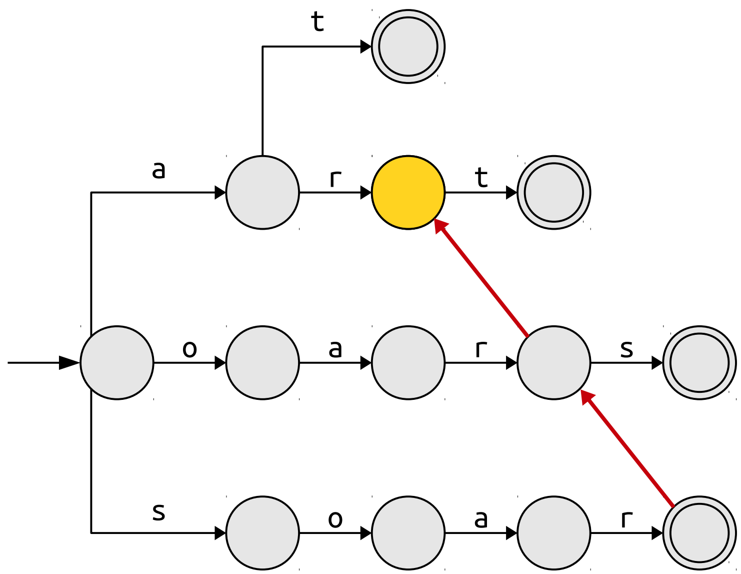


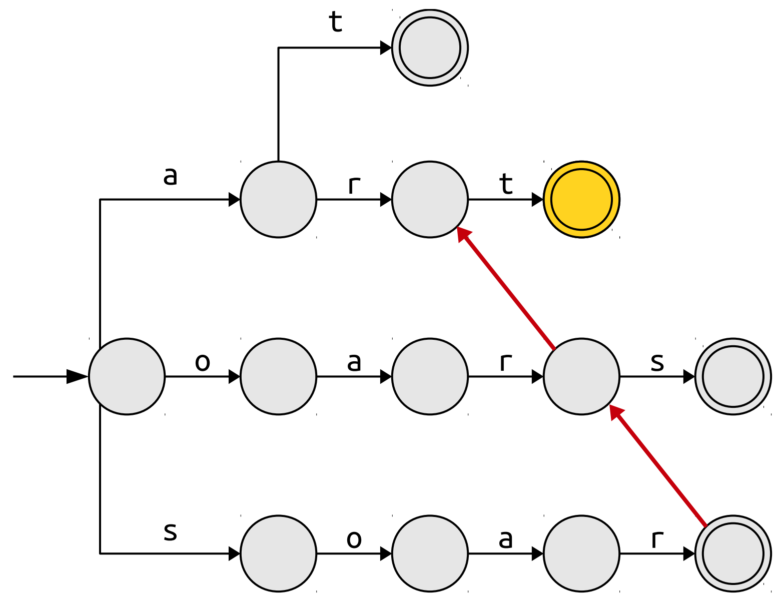
Red link is a suffix link

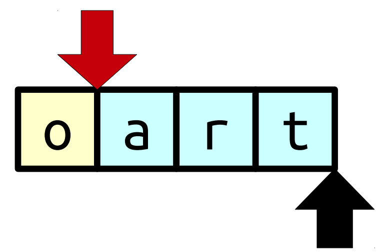
Exploiting structure





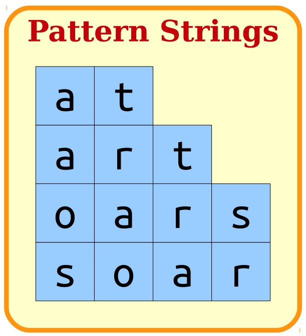


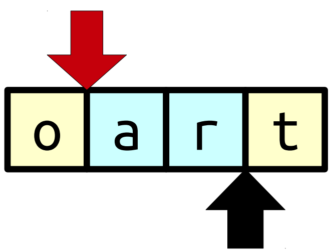


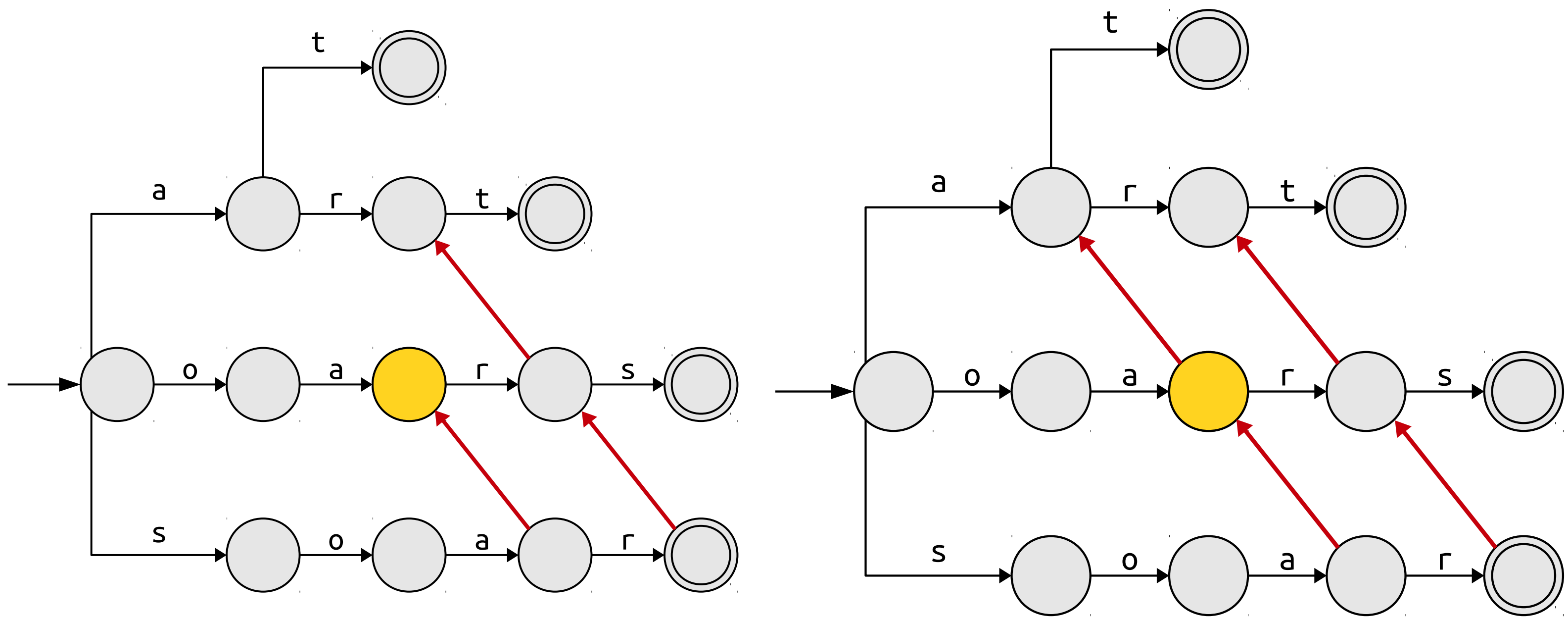


Red link is a suffix link

Exploiting structure

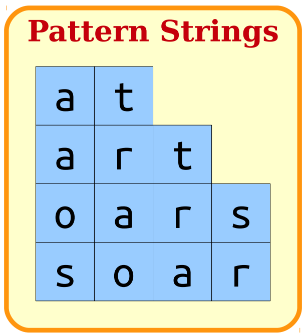


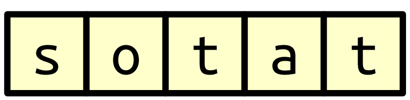


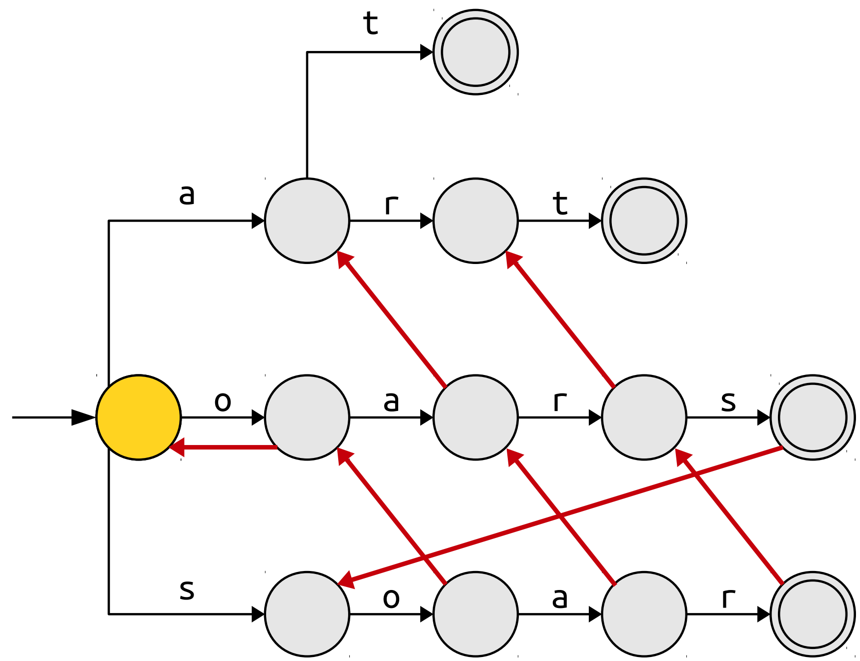


Red link is a suffix link

Exploiting structure







Red link is a suffix link

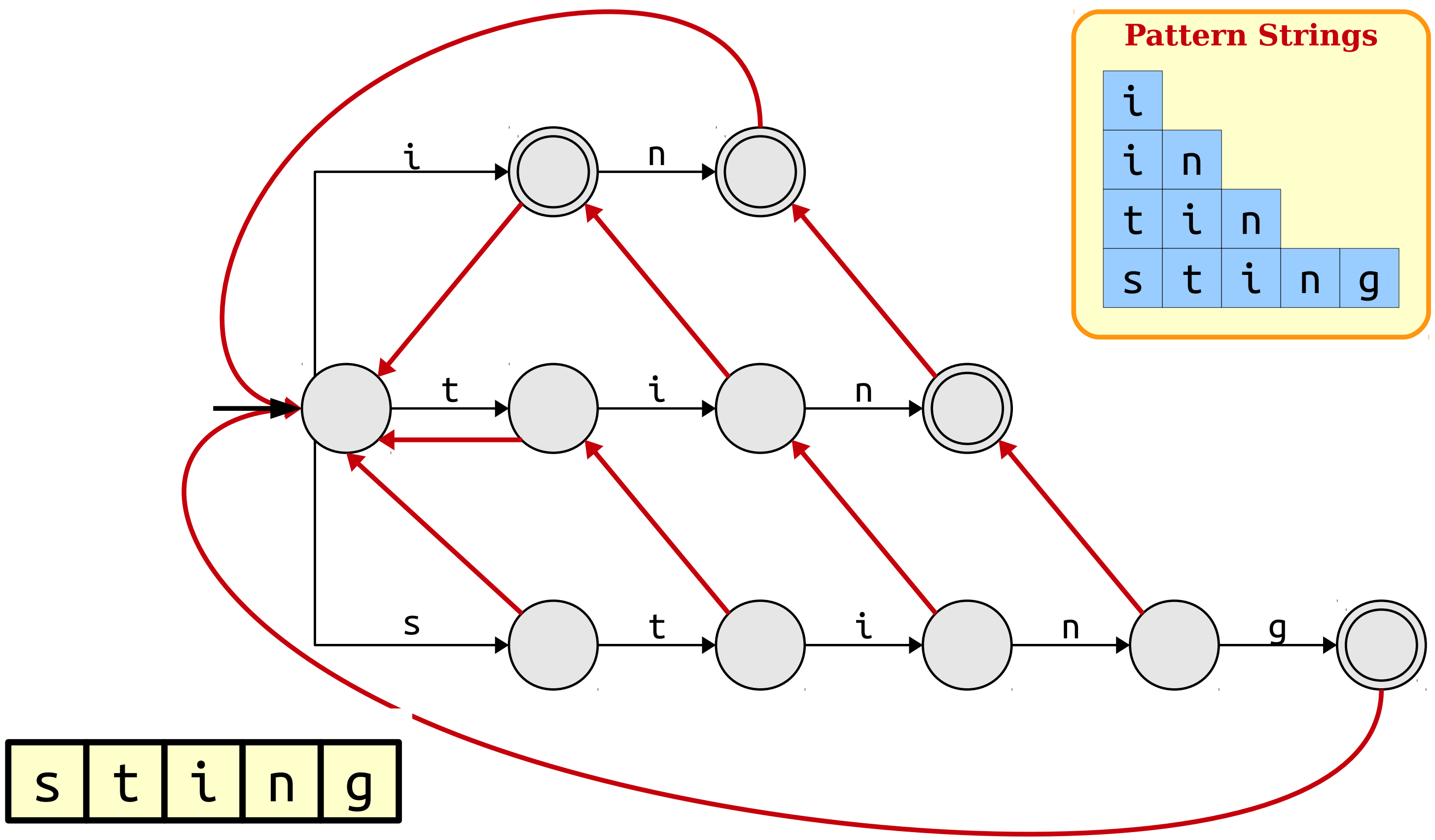
Suffix Links

- A suffix link is a red edge from a trie node corresponding to string α to the trie node corresponding to a string ω such that ω is the longest proper suffix of α that is still in the trie
- Intuition: When we hit a part of the string where we cannot continue to read characters, we fall back by following suffix links to try to preserve as much context as possible
- Every node in the trie, except the root (which corresponds to the empty string ε), will have a suffix link associated with it

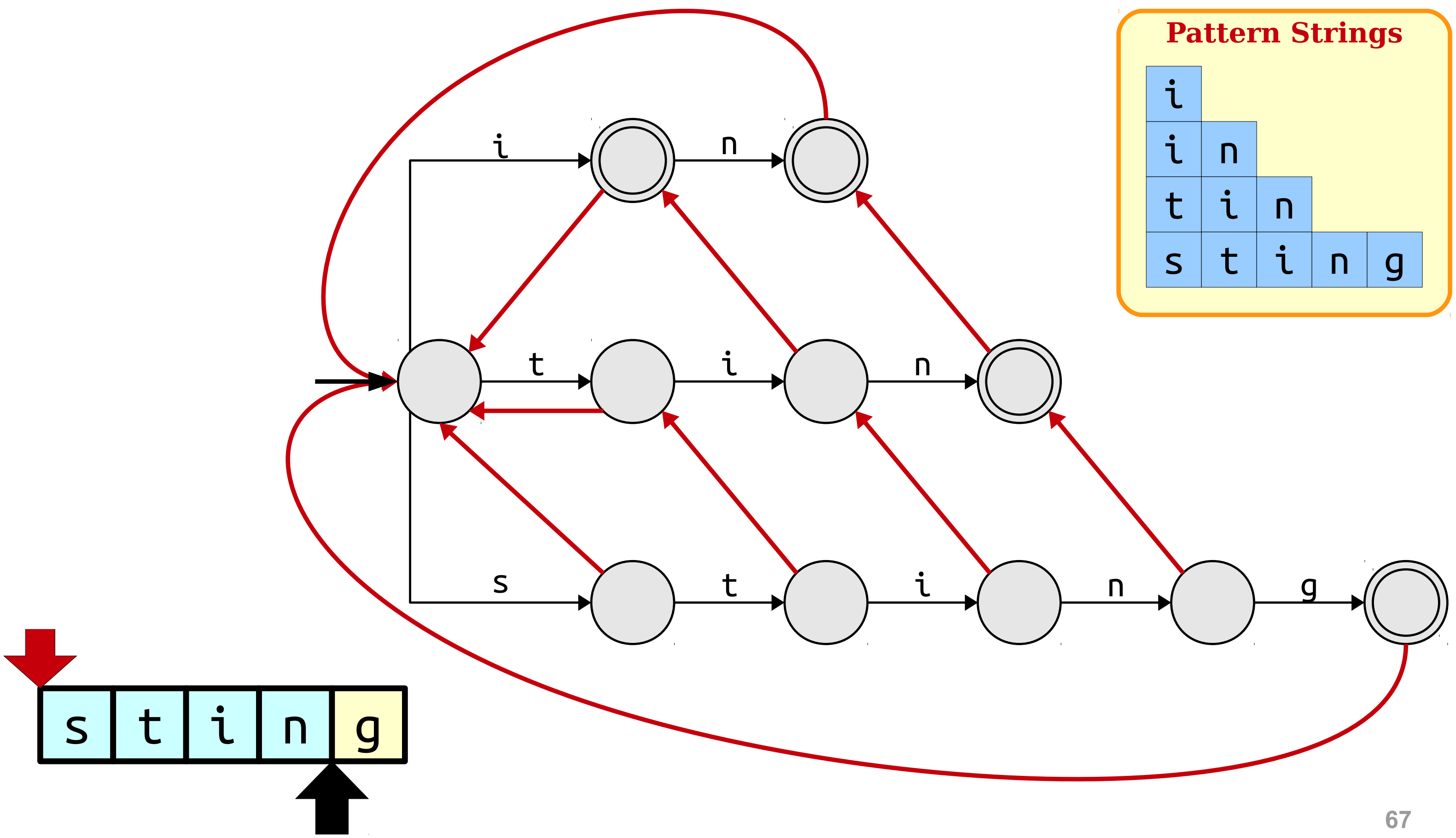
Suffix Links Are Useful

- Suffix links can substantially improve the performance of our string search
  - At each step, we either advance the black (end) pointer forward in the trie, or advance the red (start) pointer forward
- Each pointer can advance forward at most O(m) times
- This reduces the amount of time spent scanning characters from $O(mL_{max})$ down to $\Theta(m)$
- This is only useful if we can compute suffix links quickly... which we'll see how to do it

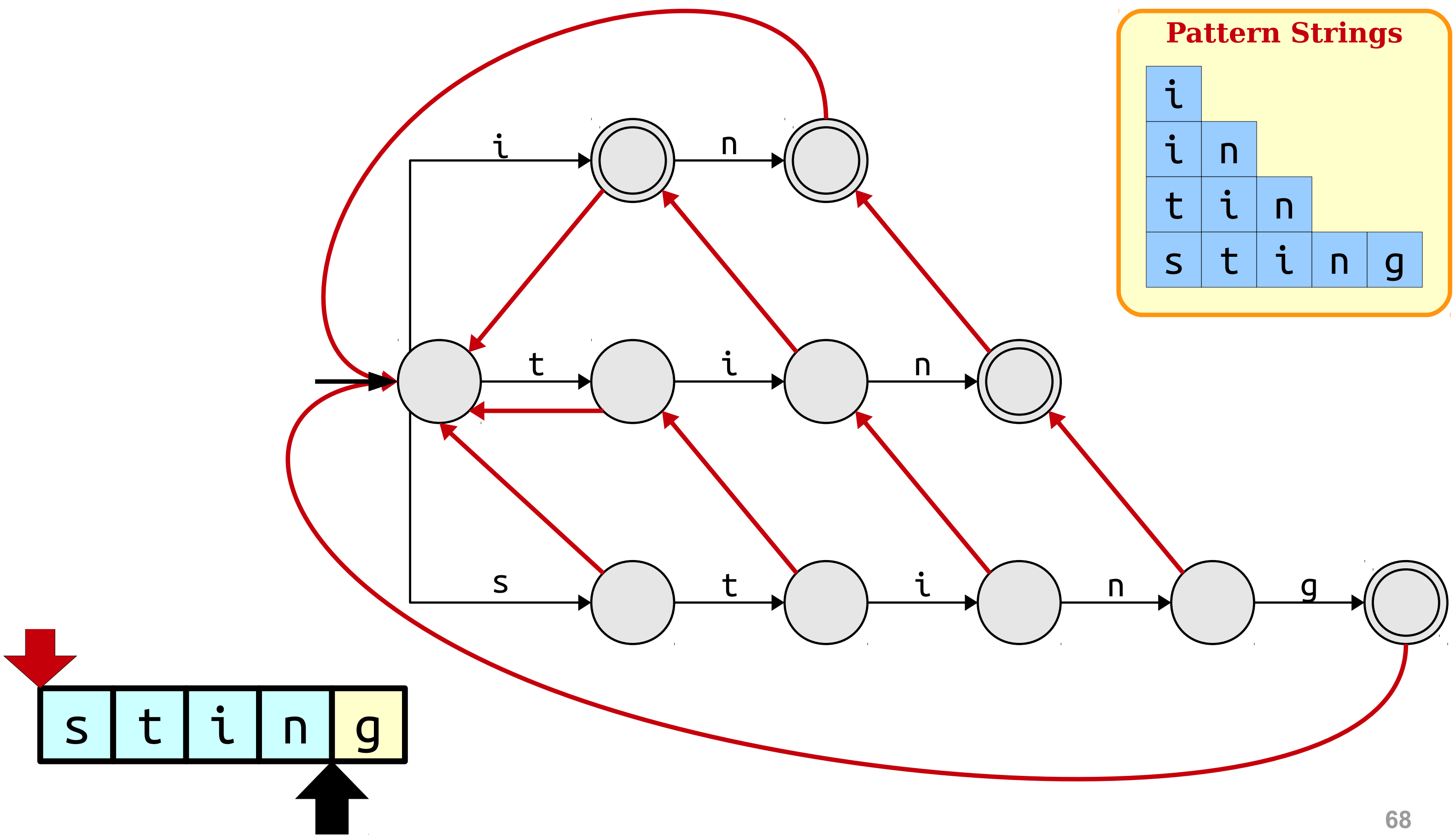
A Possible Problem



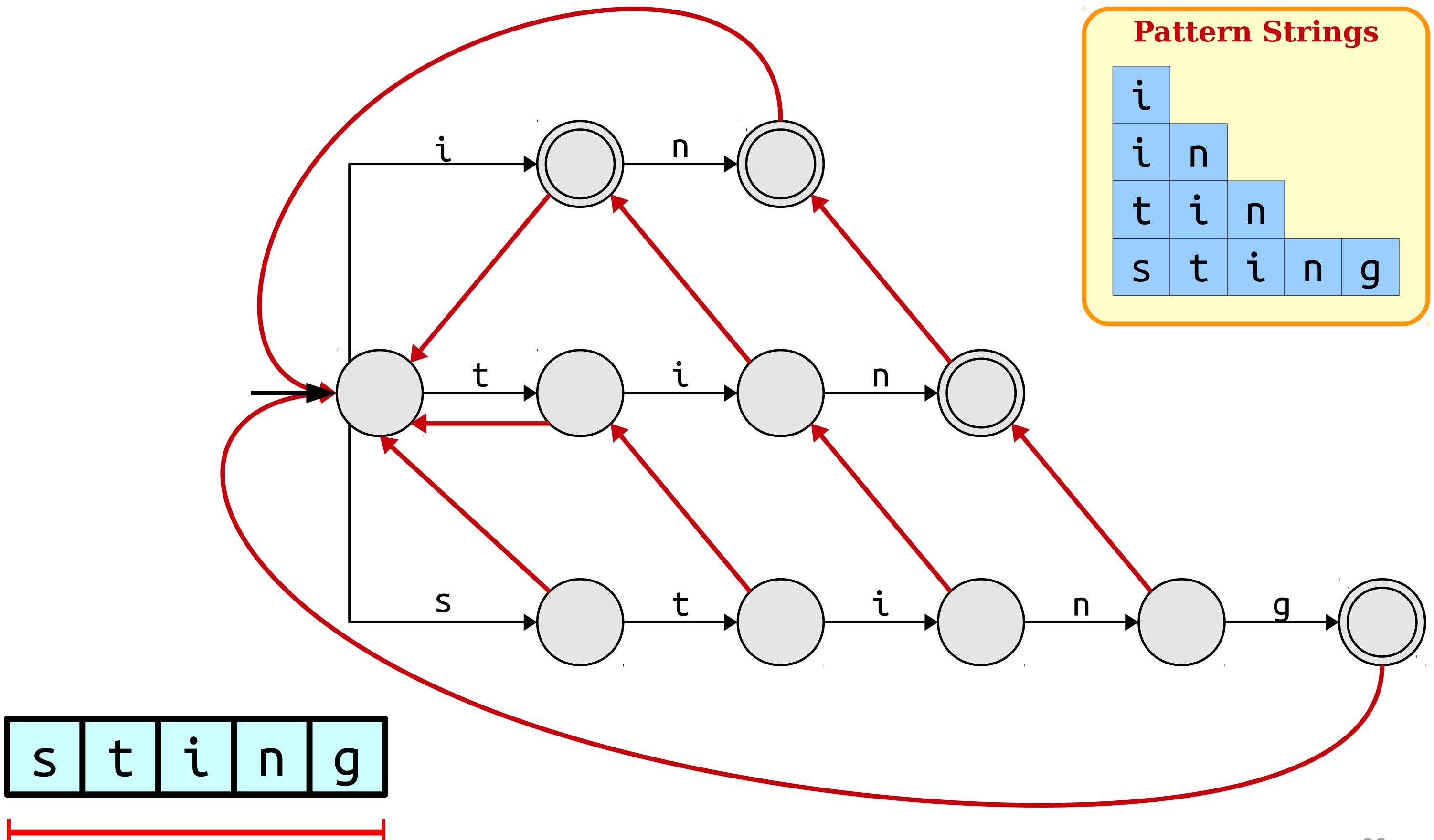
A Possible Problem



A Possible Problem

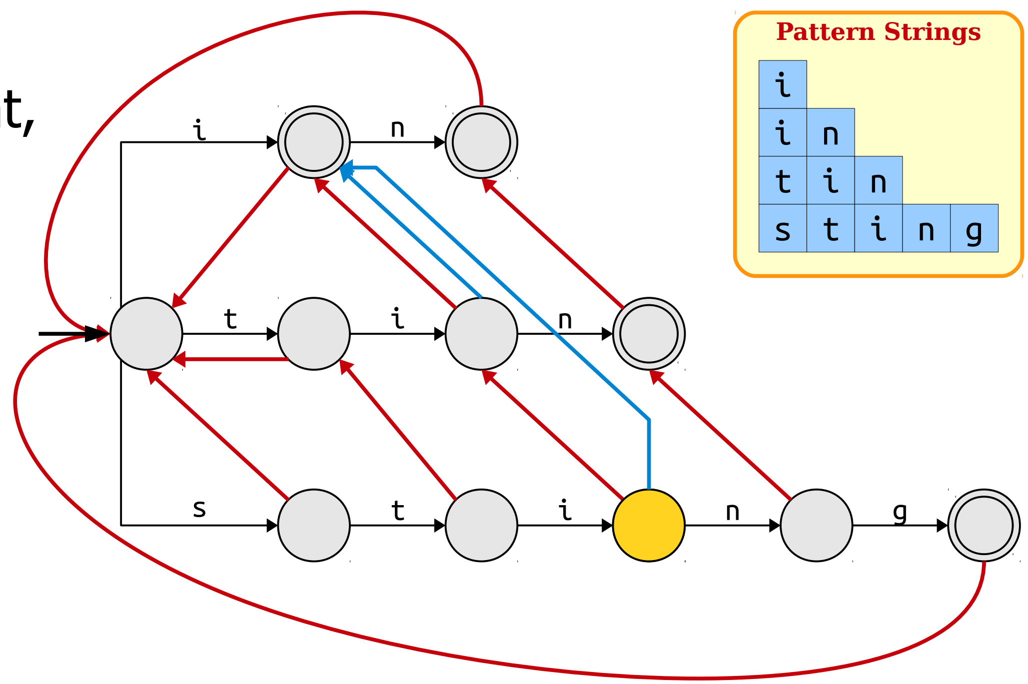


A Possible Problem

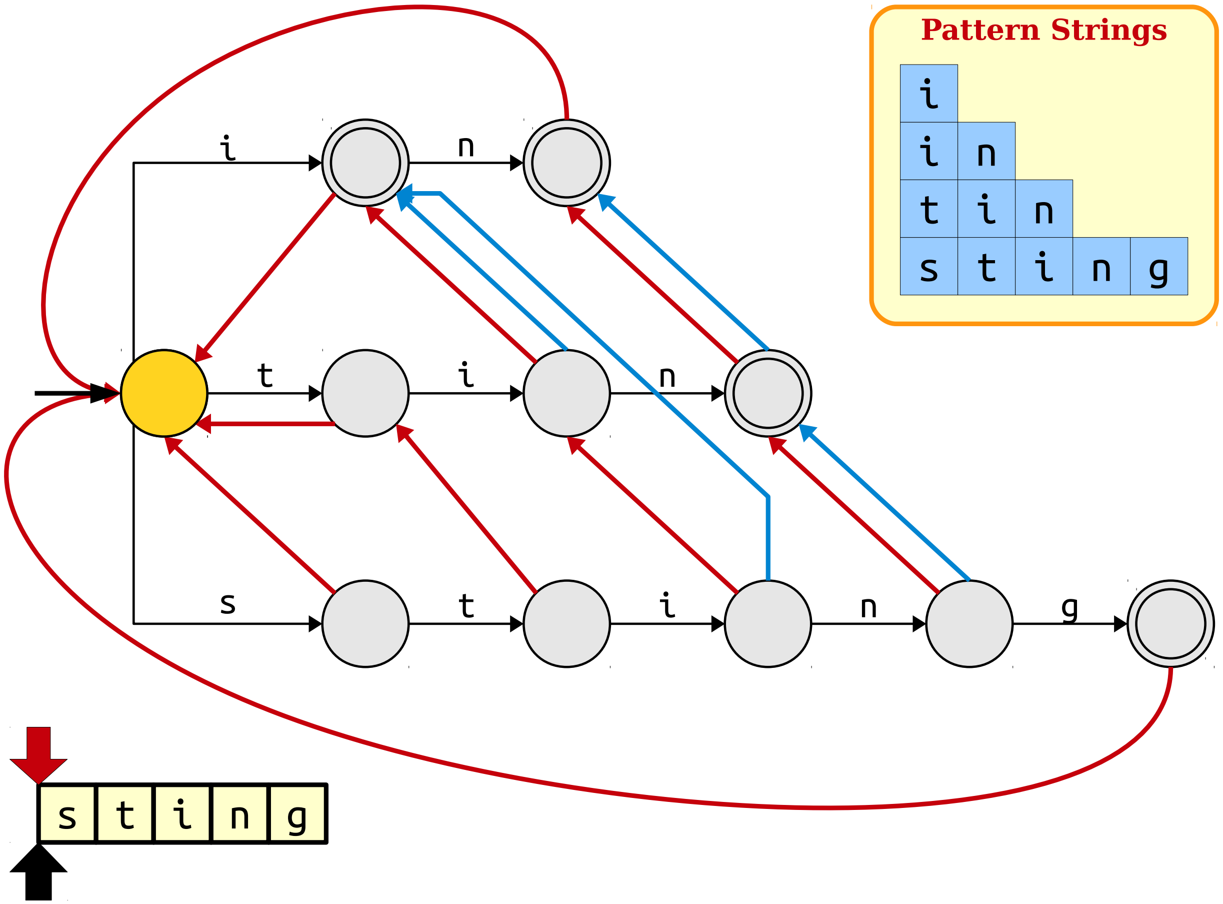


Problem?

- We found "sting", but need to find patterns inside of "sting"
  - Need to figure out how to output "i", "in", "tin" while searching
- Solution: Output links (blue links!)
  - Whenever output links are present, output string pointed to output link when visiting a node
- Precomputing where we eventually need to end up, we can read off any extra patterns we find
  - Can be done quickly!



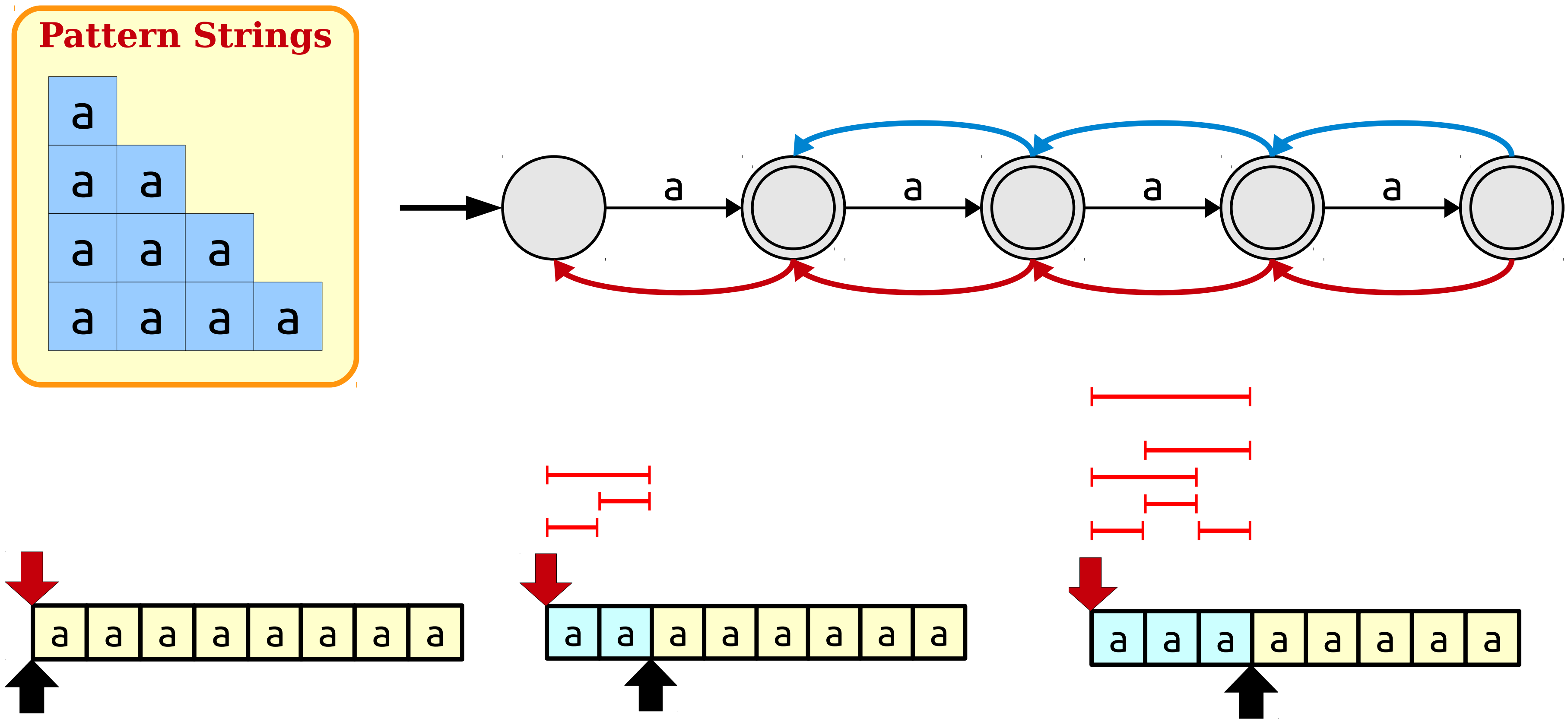
Output Links



Final Algorithm

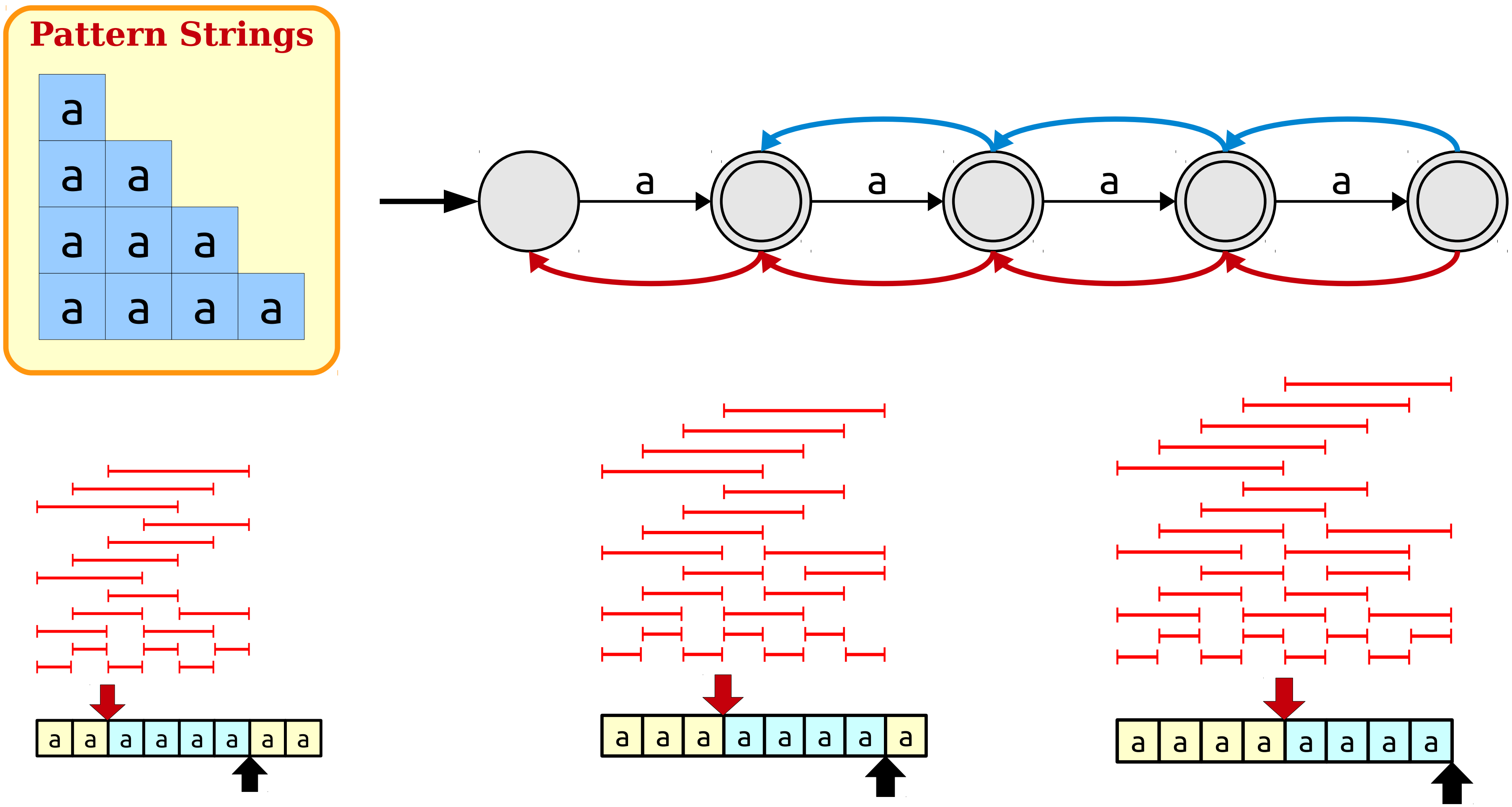
- Start at the root node in the trie
- For each character c in the string:
  - While there is no edge labeled c:
    - If you are at root, break out of loop; otherwise, follow suffix link
  - If there is an edge labeled c, follow it
    - If the current node corresponds to a pattern, output that pattern
    - Output all words in the chain of output links originating at this node
- How bad can this be? How much backtracking?

Bad Case



Pattern Strings
a
a a
a a a
a a a a
a
a
a
a
a a a a a a a a
a a a a a a a a
a a a a a a a a

Bad Case

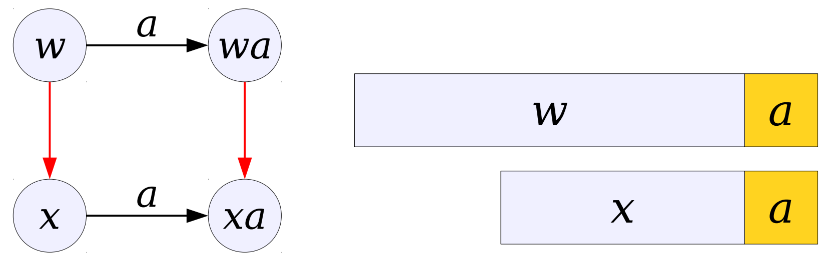

Pattern Strings
a
a a
a a a
a a a a
a a a a
a a a a a a a a
a a a a a a a a
a a a a a a a a

Run Time Analysis

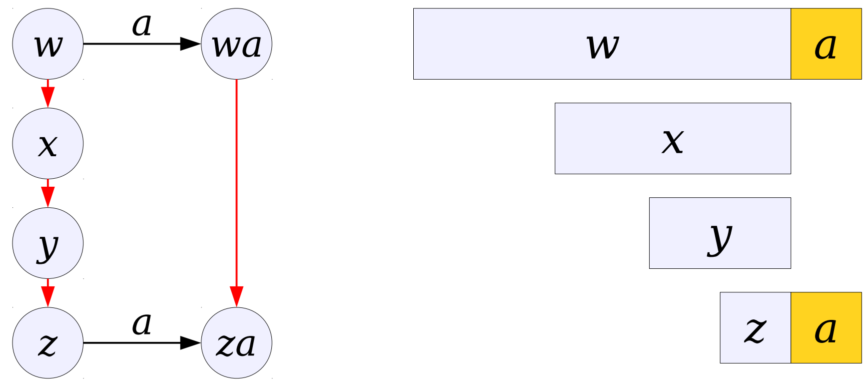
- This is known as the Aho-Corasick algorithm
  - May spend a lot of time outputting matches
  - That is a function of the problem, not the algorithm
  - Any algorithm that matches strings would have to spend the time reporting matches
- Let z denote the number of matches reported by our algorithm.
  - Runtime of match phase is $\Theta(m + z)$
- Only issue: How long does it take to build a trie, with suffix links and output links?
  - Building a trie is $\Theta(n)$, so need to add suffix links and output links

Fast Suffix Link Construction

- Key insight: Suppose we know the suffix link for a node labeled w. After following a trie edge labeled a, there are two possibilities.
  - Case 1: xa exists



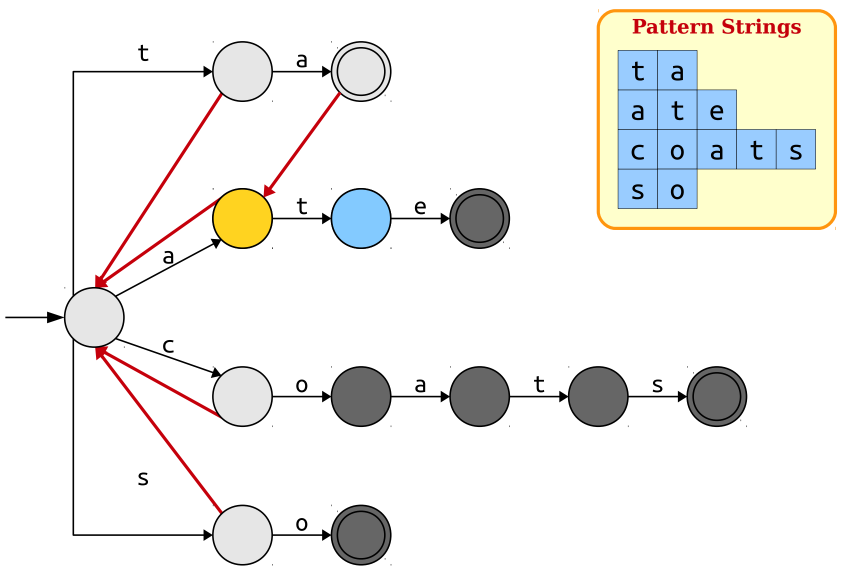
  - Case 2: xa does not exist

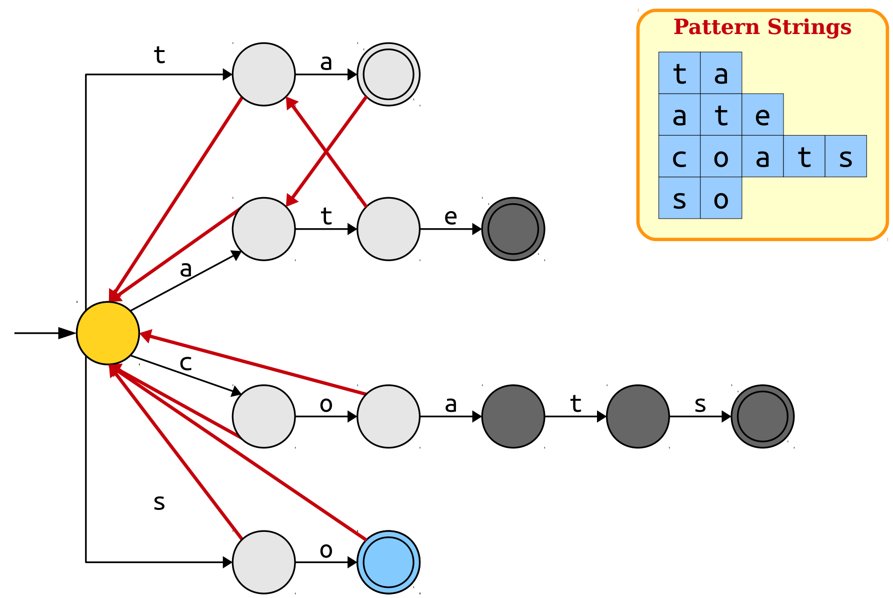


Fast Suffix Link Construction

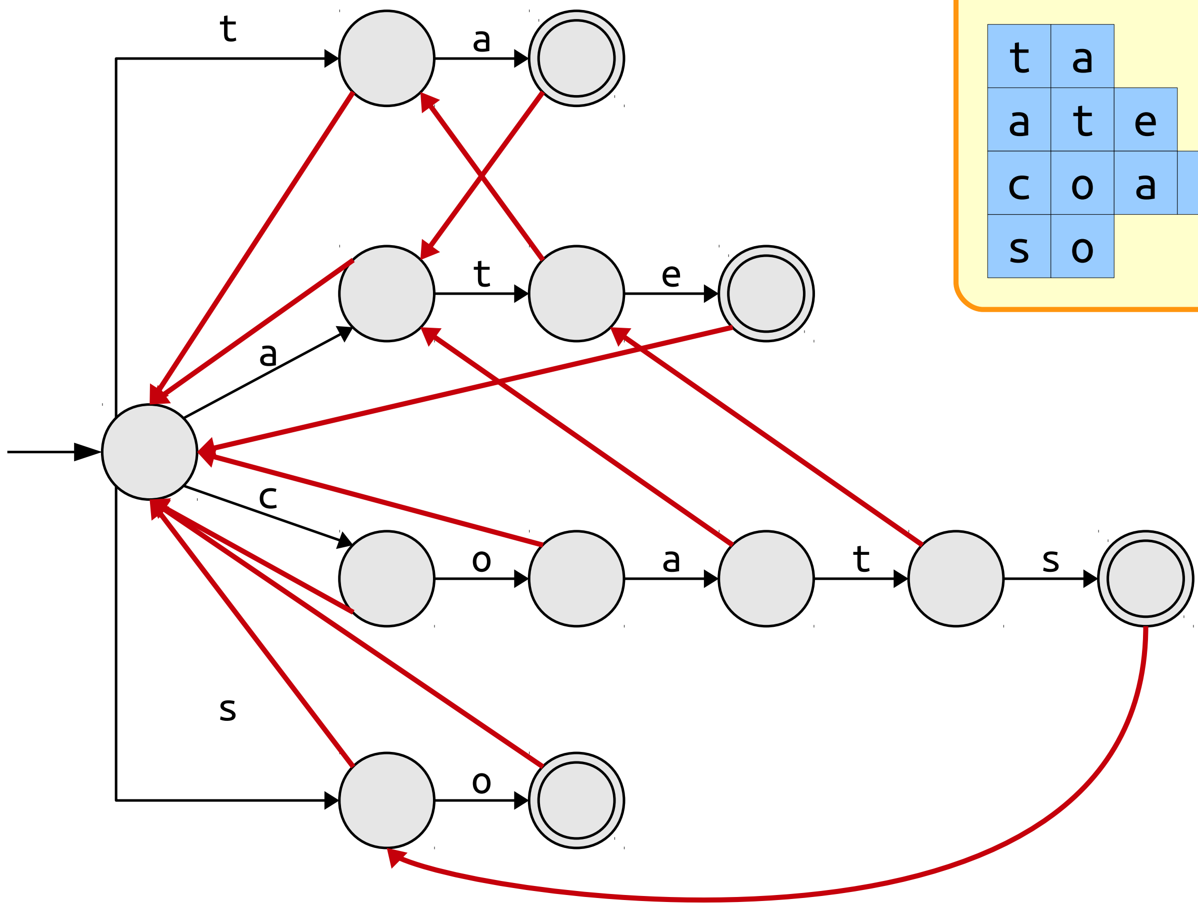
- To construct the suffix link for a node wa:
  - Follow w's suffix link to node x
  - If node xa exists, wa has a suffix link to xa
  - Otherwise, follow x's suffix link and repeat
  - If you need to follow backwards from the root, then wa's suffix link points to the root
- Observations
  - Suffix links point from longer strings to shorter strings
  - Precomputing suffix links for nodes in **ascending order** of string length, information needed will be available at the time we need it
  - Breadth first search!

Growing the Trie





Growing the Trie



Pattern Strings

t	a			
a	t	e		
c	o	a	t	s
s	o			

Fast Suffix Link Algorithm

- Do a breadth-first search of the trie, performing the following operations:
  - If the node is the root, it has no suffix link
  - If the node is one hop away from the root, its suffix link points to the root
  - Otherwise, the node corresponds to some string wa. Let x be the node pointed at by w's suffix link. Then, do the following:
    - If the node xa exists, wa's suffix link points to xa
    - Otherwise, if x is the root node, wa's suffix link points to the root
    - Otherwise, set x to the node pointed at by x's suffix link and repeat

- Can be done in O(n) !
  - Total time required to construct suffix links for a pattern of length h: O(h)

Computing Output Links

- Initially, set every node's output link to be a null pointer
- While doing the BFS to fill in suffix links, set the output link of the current node v as follows:
  - Let u be the node pointed at by v's suffix link
  - If u corresponds to a pattern, set v's output link to u itself
  - Otherwise, set v's output link to u's output link.

- Can be done in O(n) !
  - Exploit Breadth-first search order

Summary: Aho-Corasick Algorithm

- Finds k patterns with total length n in text of length m by building a suffix trie in O(n) and finding z matches in O(m+z)
  - Generalizes KMP algorithm
  - Exploits trie structures and prefix backtracking

- Lots of work continues in this area
  - Driven by computer search of text sources

- Most language searches for regular expressions are slow (use simple methods)
  - But fast methods exist

New Data Structure: Disjoint Sets

- Problem of Interest: keep track of dynamic relations
  - e.g. who is connected to whom
  - Who belongs to same club
  - Which computers are in same network
  - Which web pages link on the Internet
- Abstraction
  - Have a set of objects
  - Have relations between objects:
    - symmetric, $a \sim b$, transitive $a \sim b$, $b \sim c \longrightarrow a \sim c$
  - Want to keep track of subsets of objects that are related as relations are added

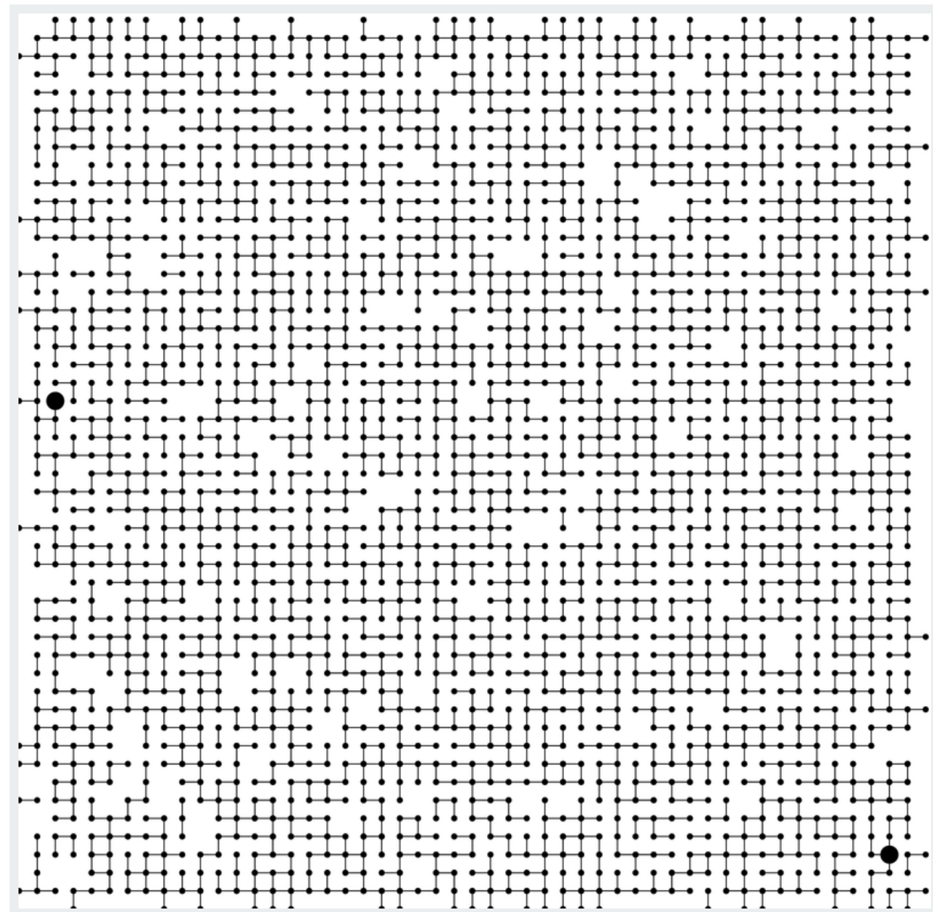
Disjoint Sets

- Operations
  - Find(a): Find the subset that contains element a
  - Union(a,b): Add a relation between two elements a, b which merges the subsets containing a, b
  - Often known as Union-Find problem

- Would like to do Unions and Find in O(1)
  - Seems really hard…
  - We'll come close with a disjoint set data structure

Disjoint Sets

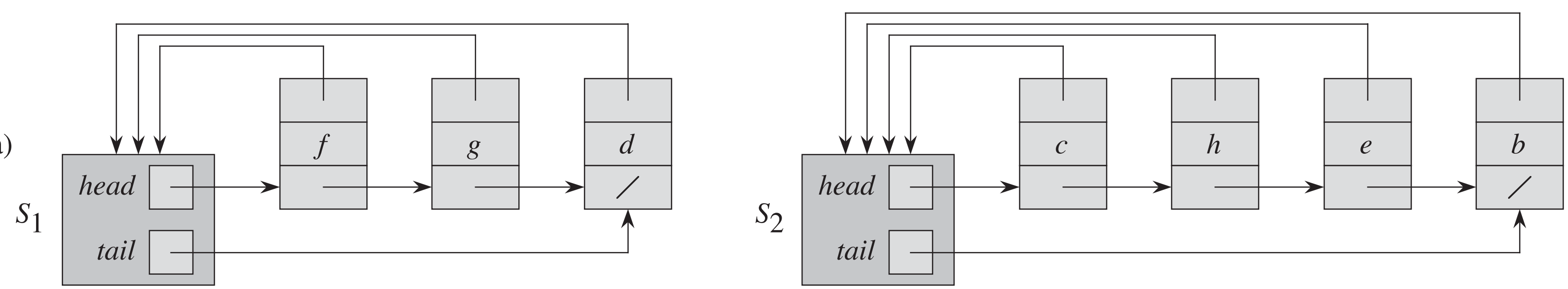
- Application: Find path in graph
  - Do start, end belong to same subset
  - Relation: a,b related if edge exists between them



Disjoint Sets

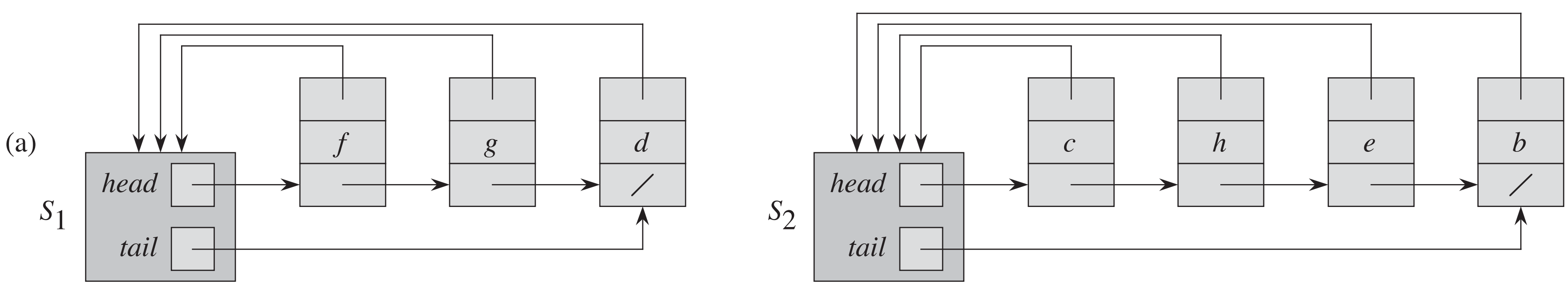
- Possible solution
  - Represent subsets as linked lists, with a head object as its identifier
  - All list elements point to head, and to next in list
  - Head points to first child and to last child

(a)



Disjoint Sets

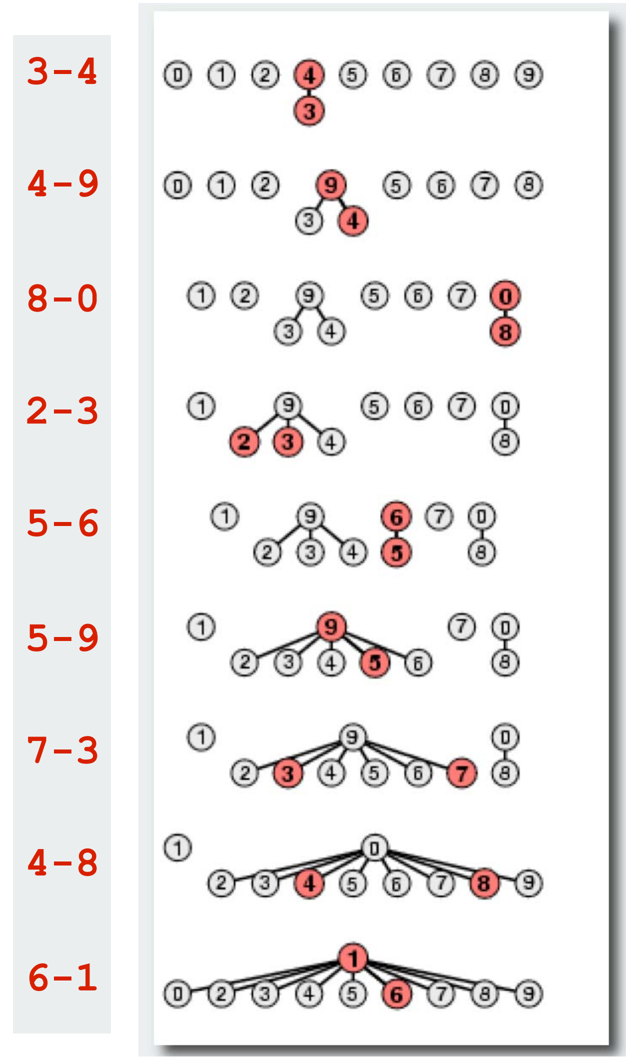
- Possible solution Complexity
  - Find(a) is O(1) using pointers to head
  - Union(a,b): Slow, needs to update pointers to new head
    - O(n) worst case



Disjoint Sets

- Better Solution
  - Subsets as trees, children point to parent, root is subset label
  - Find: get root label of tree, moving up the tree
  - Union: if new relation a ~ b, Find(a) and Find(b) If a, b are in different trees, merge trees
  - How to merge: Keep track of height of trees
    - Merge shorter tree as direct child taller tree

- Complexity
  - Find complexity: height of tree O(log(n))
  - Union complexity: O(Find) + O(merge) = O(find)



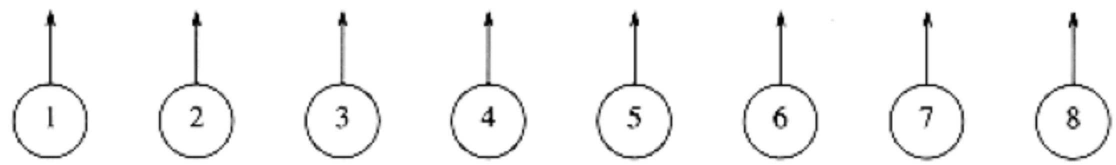


Figure 8.1 Eight elements, initially in different sets

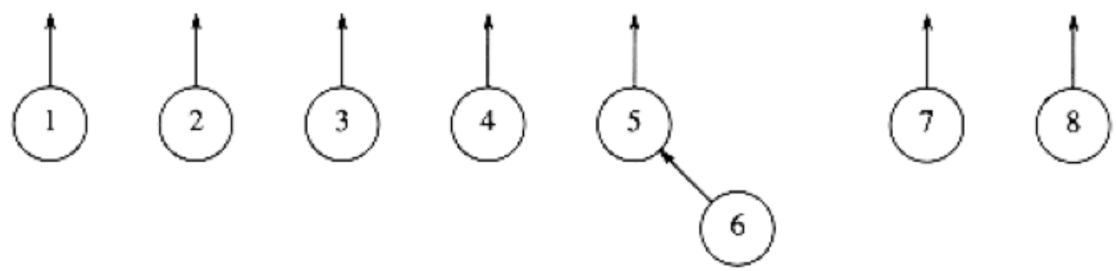


Figure 8.2 After union (5, 6)

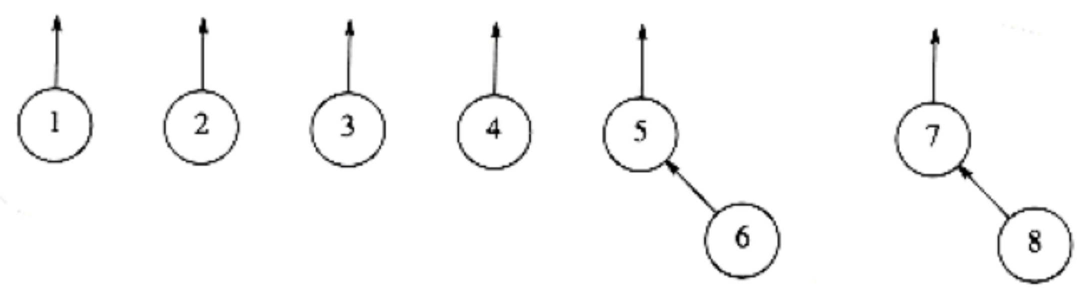


Figure 8.3 After union (7, 8)

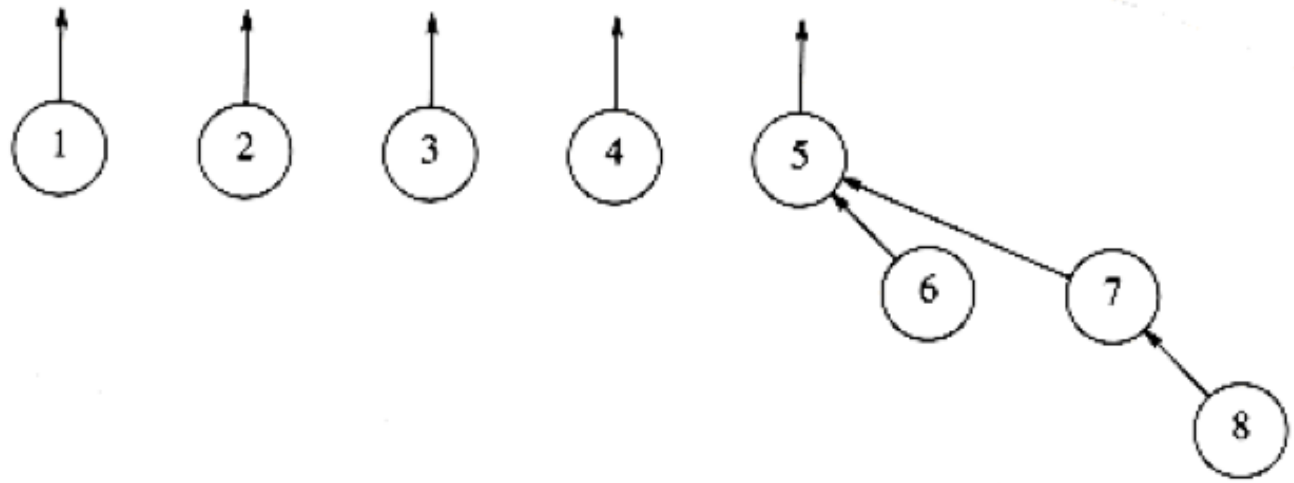


Figure 8.4 After union (5, 7)

0	0	0	0	0	5	5	7
1	2	3	4	5	6	7	8

Union (4,5)

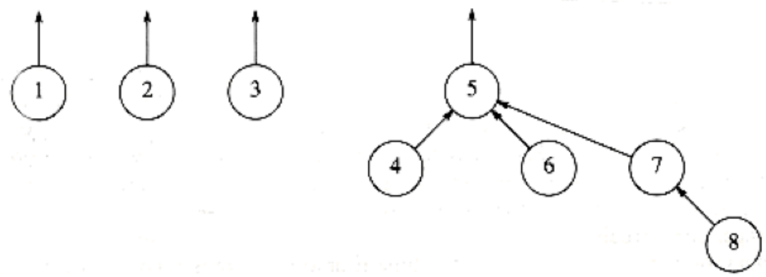


Figure 8.10 Result of union-by-size

Small tree links to large

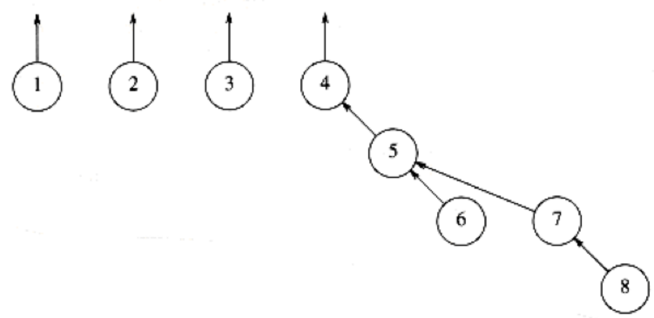


Figure 8.11 Result of an arbitrary union

Worst case of union-by-size
depth is O(log(N))

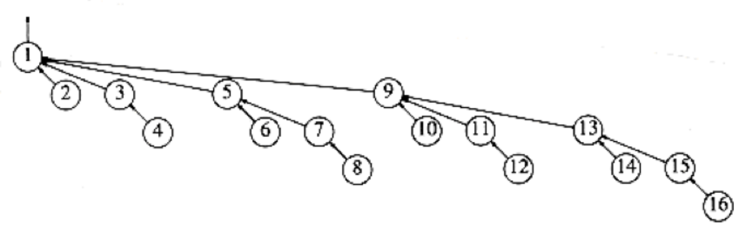
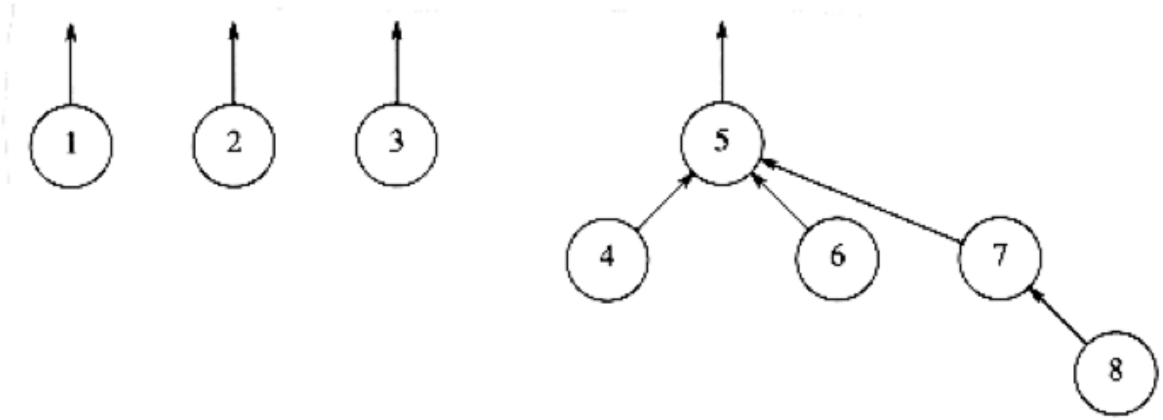


Figure 8.12 Worst-case tree for n = 16

The following figures show a tree and its implicit representation for both union-by-size and union-by-height. The code in Figure 8.13 implements union-by-height.

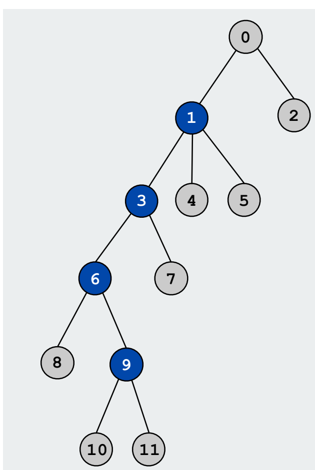


−1	−1	−1	5	−5	5	5	7	- size
1	2	3	4	5	6	7	8	

0	0	0	5	−2	5	5	7	- height
1	2	3	4	5	6	7	8	

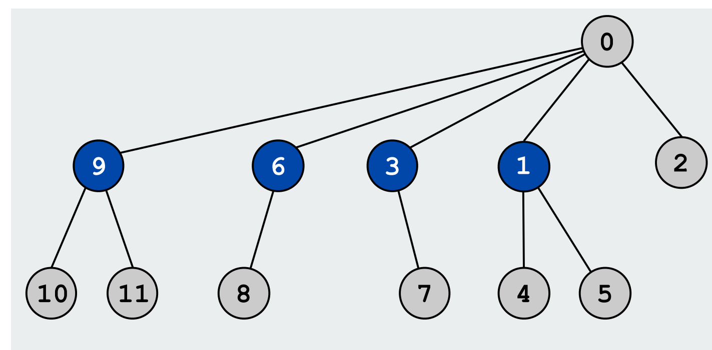
Disjoint Sets

- Best Solution
  - Subsets as trees, children point to parent, root is subset label
  - Find: get root label of tree, moving up the tree
    - Restructure tree by making all ancestors have root as parent
  - Union: if new relation a ~ b, Find(a) and Find(b). If a, b are in different trees, merge trees



Path Compression

Find 9



Disjoint Sets

- Theorem. Starting from an empty data structure, any sequence of M union and find operations on N objects takes $O(N + M \log^*(N))$ time
  - $\log^*(N)$: Ackerman function
    - $A(i) = 2^{A(i-1)}$; A(0) = 1
    - A(1) = 2, A(2) = 4, A(3) = 16, A(4) = 216 = 65536, A(5) = 265636,
    - A(6) = VERY VERY VERY BIG!
  - Inverse Ackerman: $i = \log^*(N)$
    - log*(N) = min number times you take $\log_2(\log_2(\cdots))$ to get smaller than or equal to 1
    - For all practical purposes, it is O(1) (Ch. 21, CLRS)

Example: Connected Components