Aho-Corasick Automata

String Data Structures

- Over the next few days, we're going to be exploring data structures specifically designed for string processing.
- These data structures and their variants are frequently used in practice.
  - In fact, the data structures we’ll talk about today were all invented by people implementing real systems!

Looking Forward

- Today: ***Aho-Corasick Automata***
  - A fast data structure for string matching.
- Thursday/Tuesday: ***Suffix Trees and Suffix Arrays***
  - Absurdly versatile string data structures.

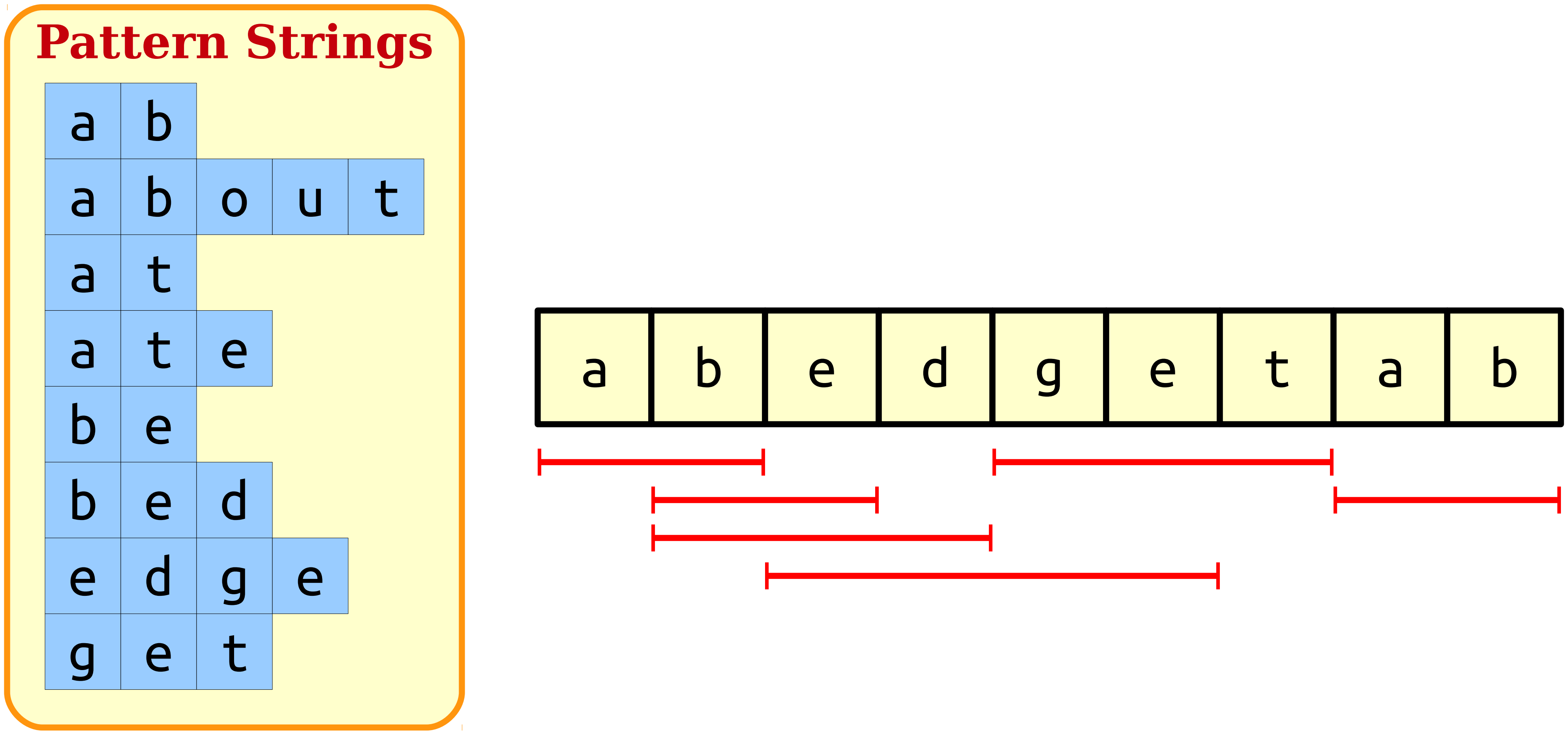
String Searching

The String Searching Problem

- Consider the following problem:

  Given a string T and k nonempty strings $P_1, \dots, P_k$, find all occurrences of $P_1, \dots, P_k$ in T.

- T is called the ***text string*** and $P_1, \dots, P_k$ are called ***pattern strings***.
- This problem was originally studied in the context of compiling indexes, but has found applications in computer security and computational genomics.

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
a b e d g e t a b

Some Terminology

- Let $\boldsymbol{m} = |\boldsymbol{T}|$, the length of the string to be searched.
- Let $\boldsymbol{n} = |\boldsymbol{P_1}| + |\boldsymbol{P_2}| + \dots + |\boldsymbol{P_k}|$ be the total length of all the pattern strings.
- Let $\boldsymbol{L_{max}}$ be the length of the longest pattern string.
- Assume that strings are drawn from an alphabet Σ, where $|\Sigma|$ is some constant.
- We'll use these terms when talking about the runtime of the algorithms and data structures we'll explore over the next couple of days.

How quickly can we solve the string searching problem?

Let's start with a naïve approach.

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		

```
For each position in $T$:
    For each pattern string $P_i$:
        Check if $P_i$ appears at that position.
```

a	b	e	d	g	e	t	a	b

Analyzing Our Approach

- As before, let ***m*** be the length of the text and ***n*** the total length of the pattern strings.
- For each character of the text string *T*, in the worst case, we scan over all *n* total characters in the patterns.
- Time complexity: **O(*mn*)**.
- Is this a tight bound?

$\Theta(mn)$

Pattern Strings

a	b			
a	a	b		
a	a	a	b	
a	a	a	a	b

a	a	a	a	a	a	a	a

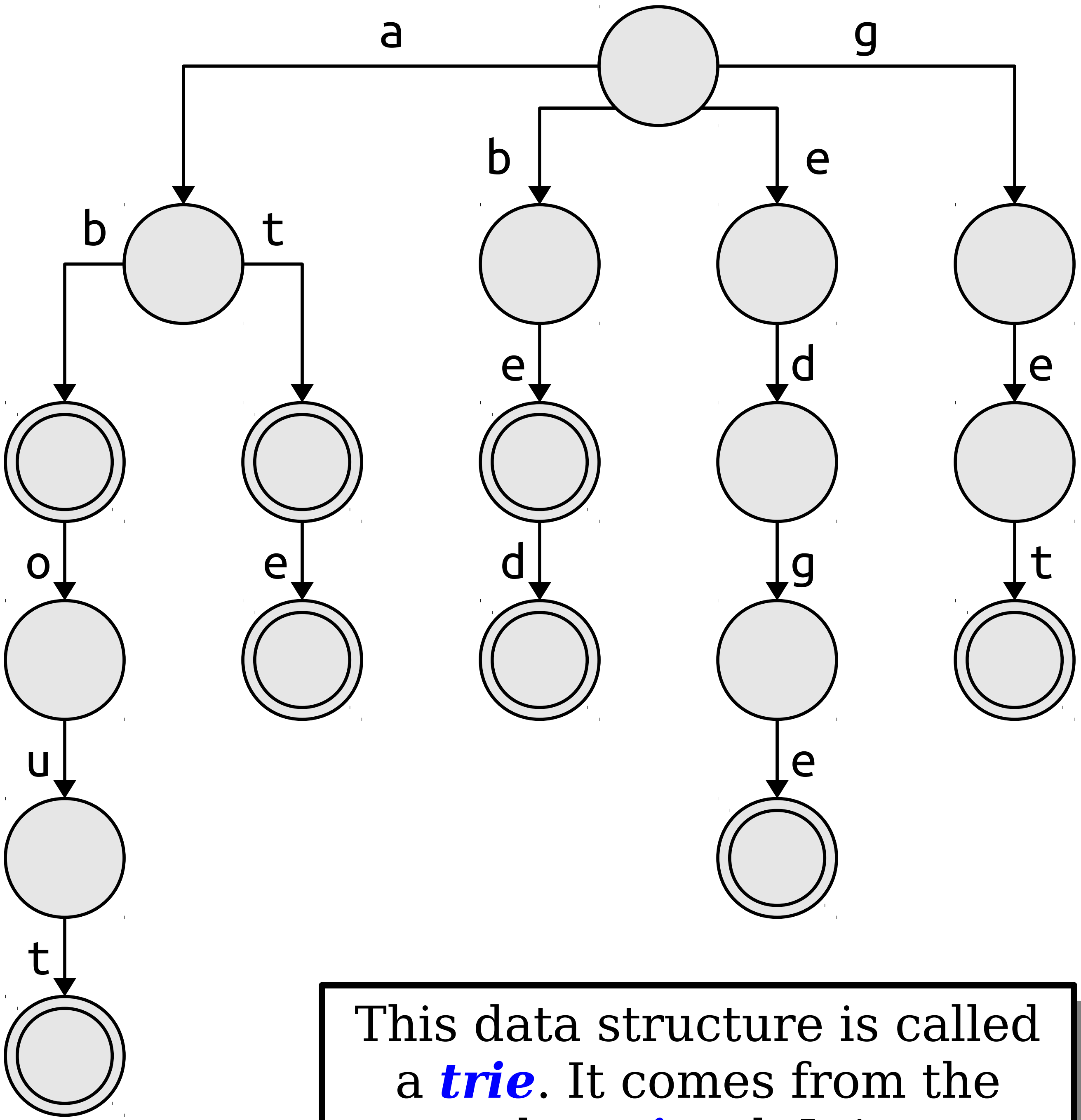
Can we do better?

Parallel Searching

- ***Idea:*** Rather than searching the pattern strings in *serial*, try searching them in *parallel*.
- Intuitively, this should cut down on a lot of the unnecessary rescanning that we're doing.
- ***Challenge:*** How exactly do we do this in practice?

Pattern Strings

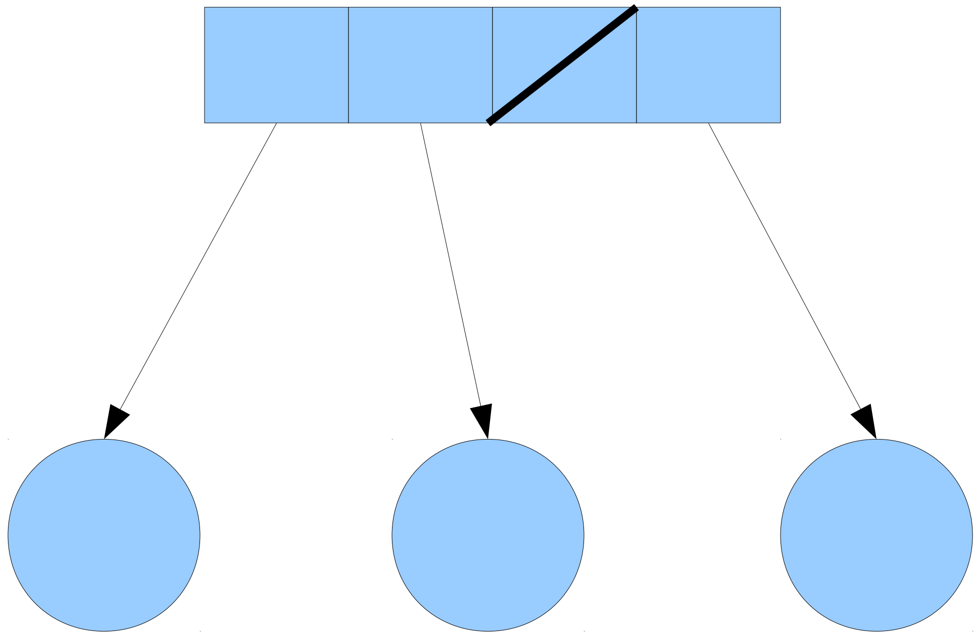
a	b			
a	b	o	u	t
a	t			
a	t	e		
b	e			
b	e	d		
e	d	g	e	
g	e	t		

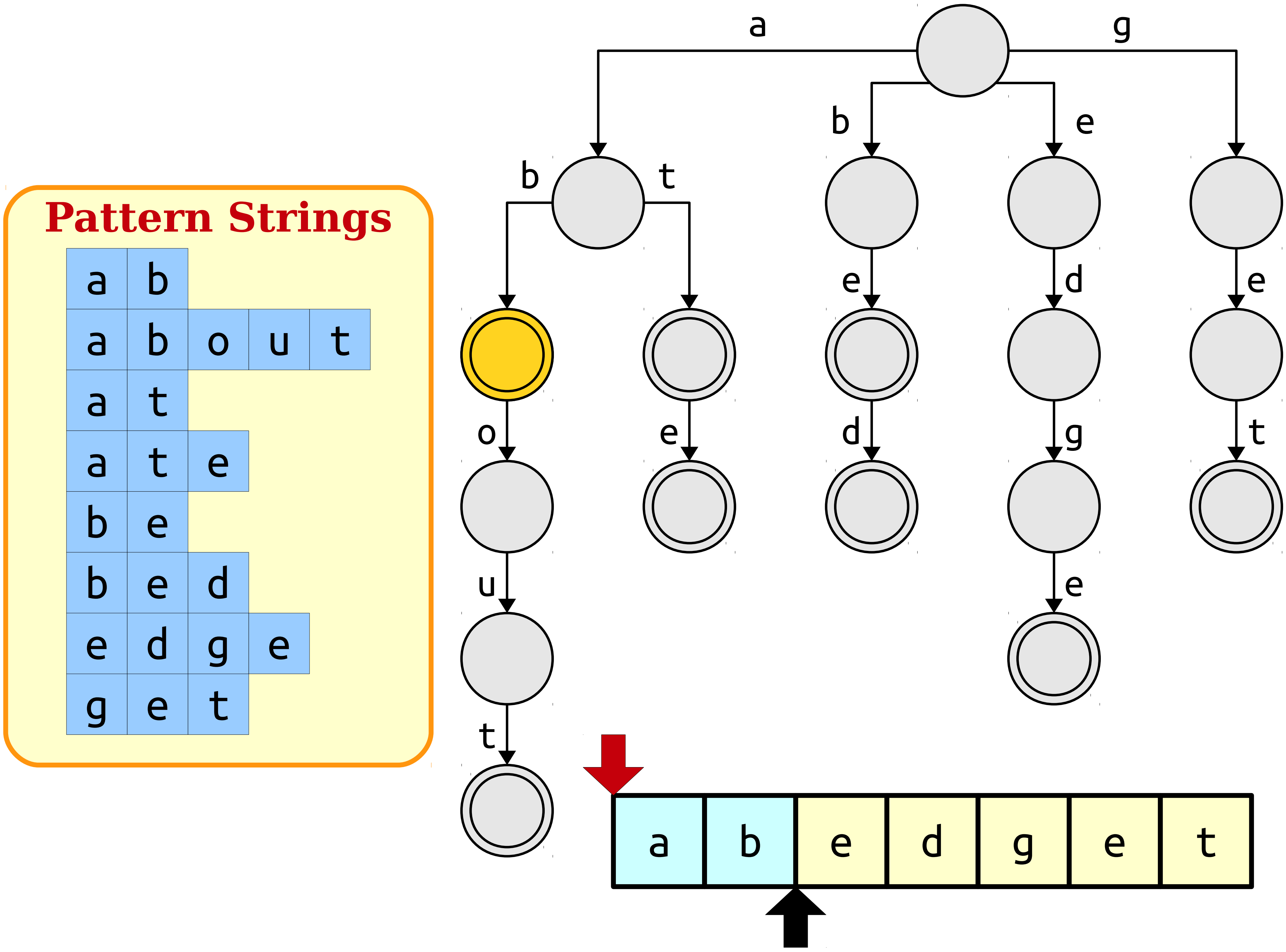


This data structure is called a ***trie***. It comes from the word re***trie***val. It is not pronounced like “retrieval.”

Representing Tries

- Each trie node needs to store pointers to its children.
- There are many different data structures we could use to store these pointers.
- For today, we'll assume we have an array of |Σ| pointers, one per possible child.
- You'll explore variants on this strategy in the problem set.



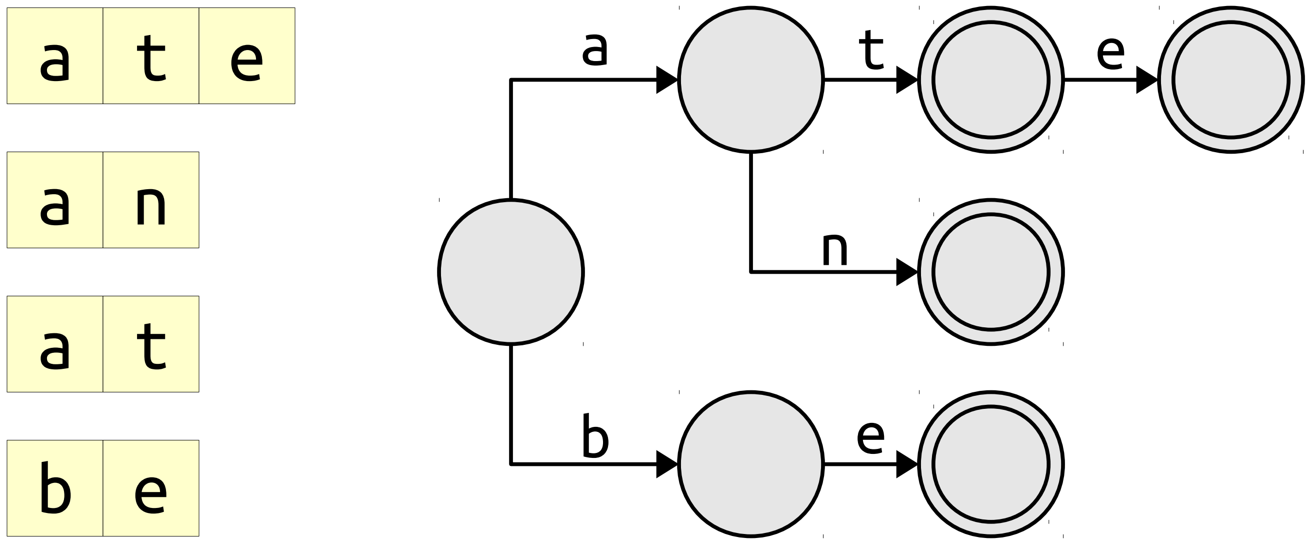

Pattern Strings
ab
about
at
ate
be
bed
edge
get
a
g
b
t
b
e
e
d
e
o
e
d
g
t
u
e
t
a b e d g e t

Analyzing our New Algorithm

- Let's suppose we've already constructed the trie. How much work is required to perform the match?
- For each character of *T*, we inspect as most as many characters as exist in the deepest branch of the trie.
- Time complexity: $\mathbf{O}(mL_{max})$, where L_{max} is the length of the longest pattern string. *(Do you see why?)*
- In the (reasonable) case where L_{max} is much smaller than n, this is a huge win over before. If L_{max} is "objectively" small, this is a pretty good runtime.
- How much time does it take to build the trie?

Building a Trie

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

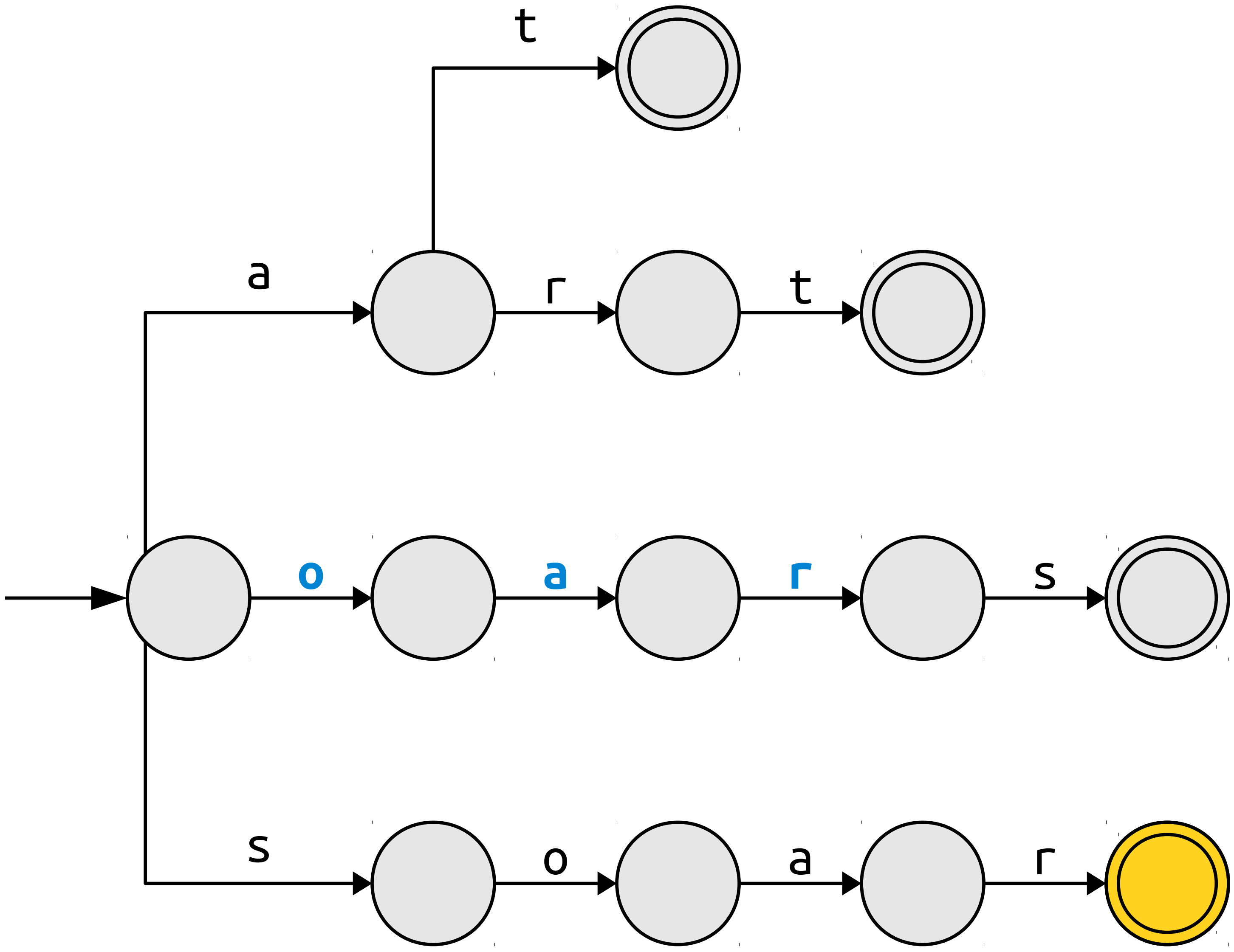


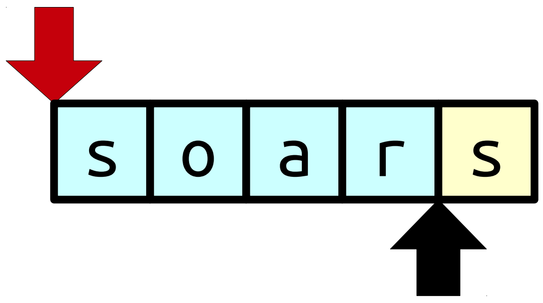
Our Strategies

- Following our foray into RMQ, we'll say that a solution to multi-string matching runs in time $\langle p, q \rangle$ if the preprocessing time is p and the matching time is q.
- We now have two approaches:
  - No preprocessing: $\langle O(1), O(\boldsymbol{mn}) \rangle$.
  - Trie searching: $\langle O(\boldsymbol{n}), O(\boldsymbol{mL_{max}}) \rangle$.
- ***Can we do better?***

Pattern Strings

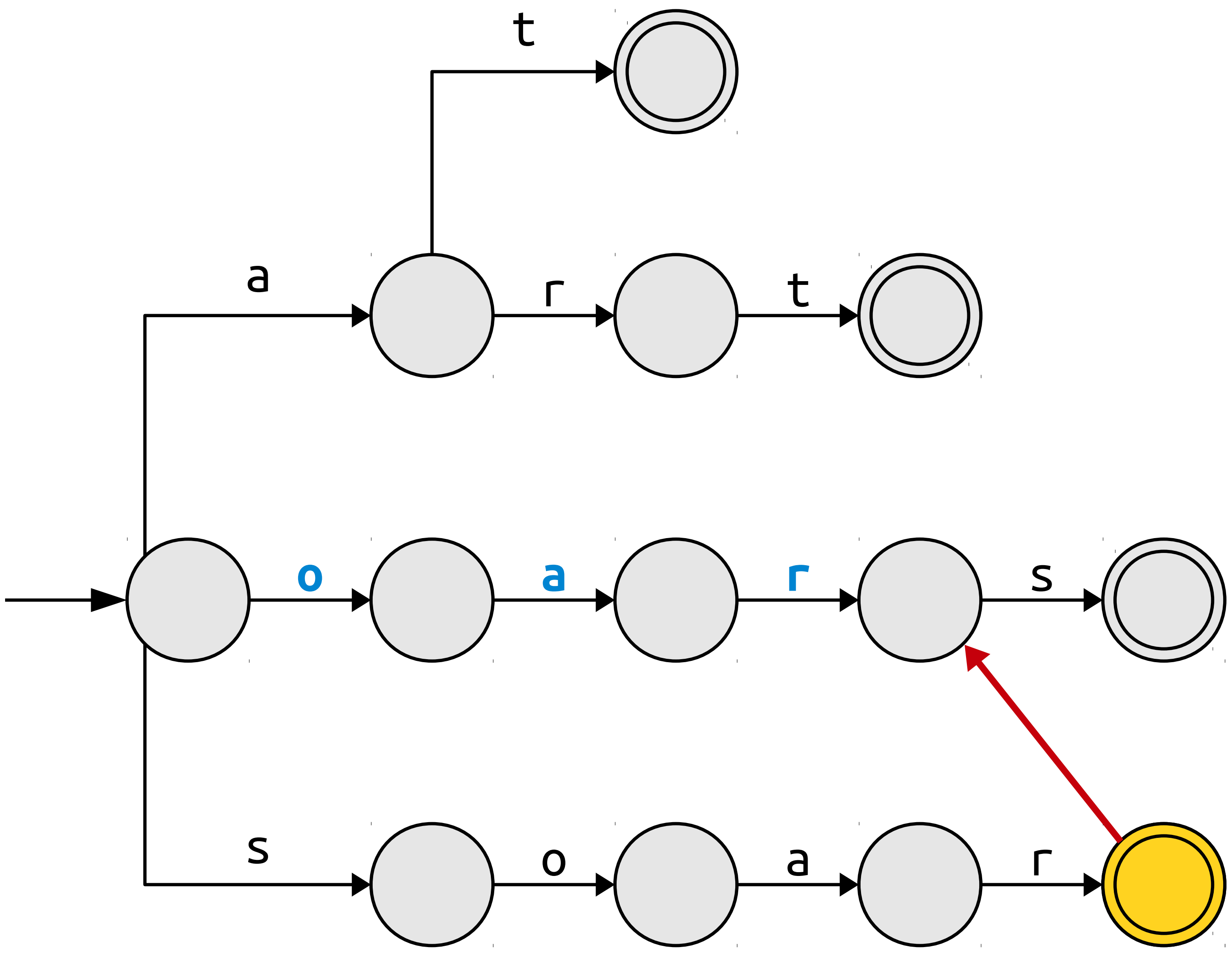
a	t		
a	r	t	
o	a	r	s
s	o	a	r

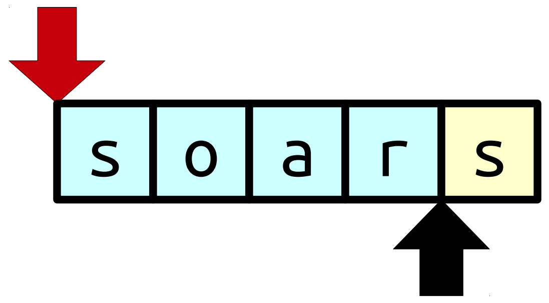




Pattern Strings

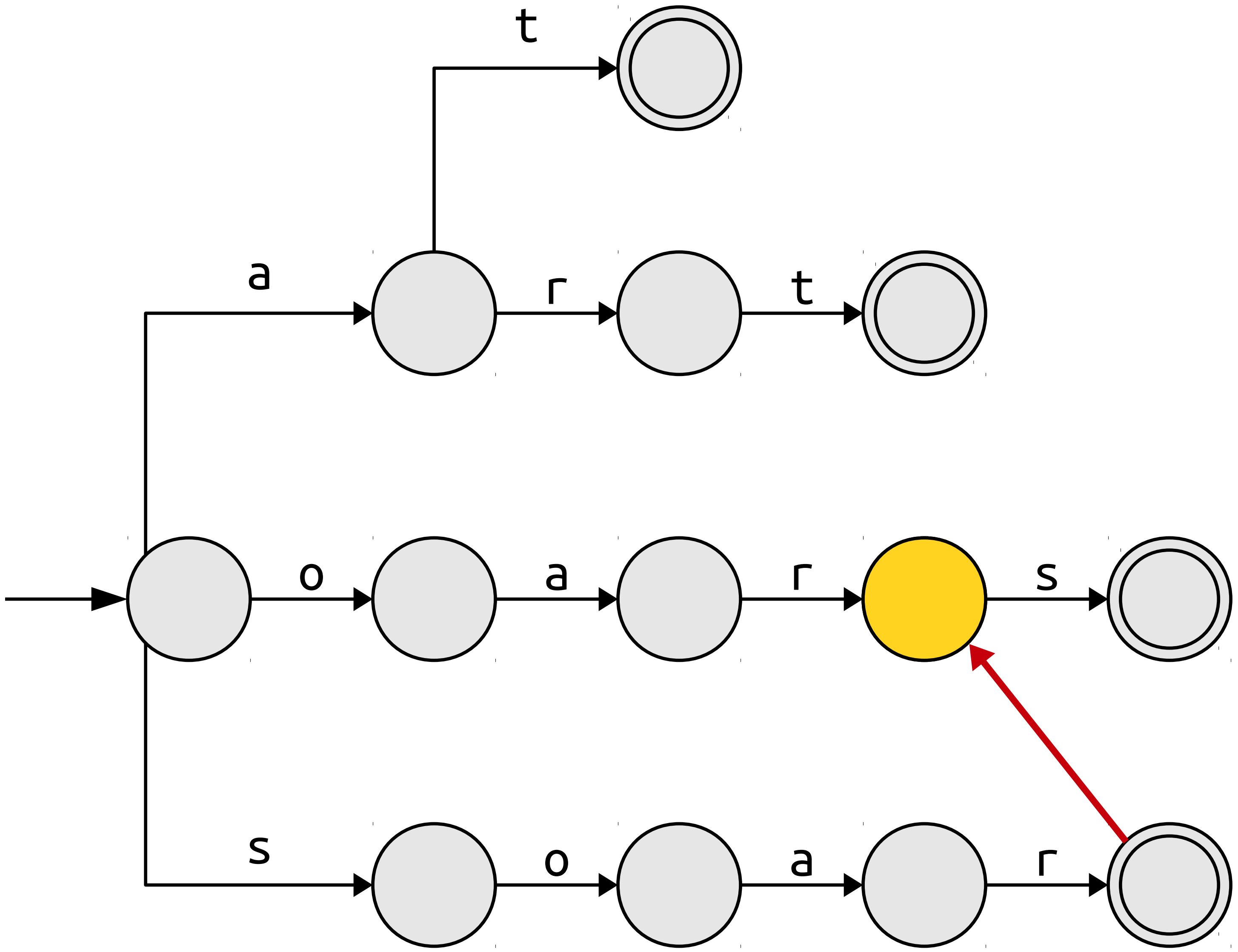
a	t		
a	r	t	
o	a	r	s
s	o	a	r

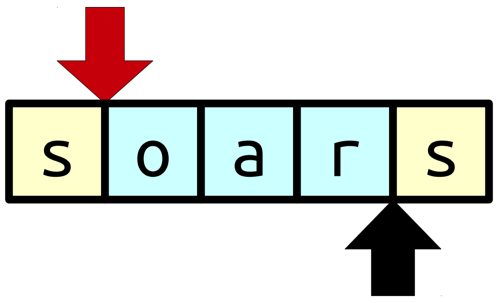




Pattern Strings

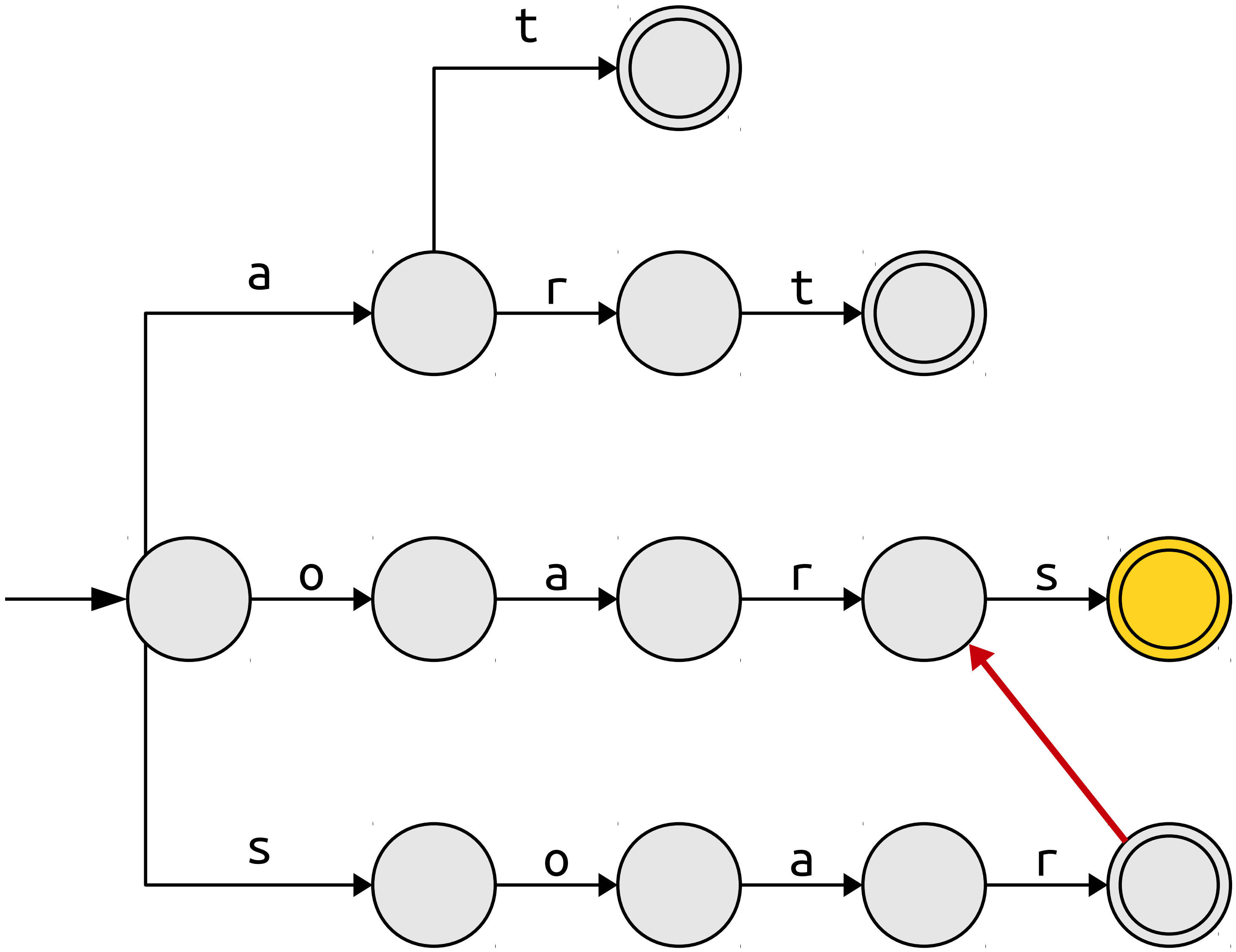
a	t		
a	r	t	
o	a	r	s
s	o	a	r





Pattern Strings

a	t		
a	r	t	
o	a	r	s
s	o	a	r

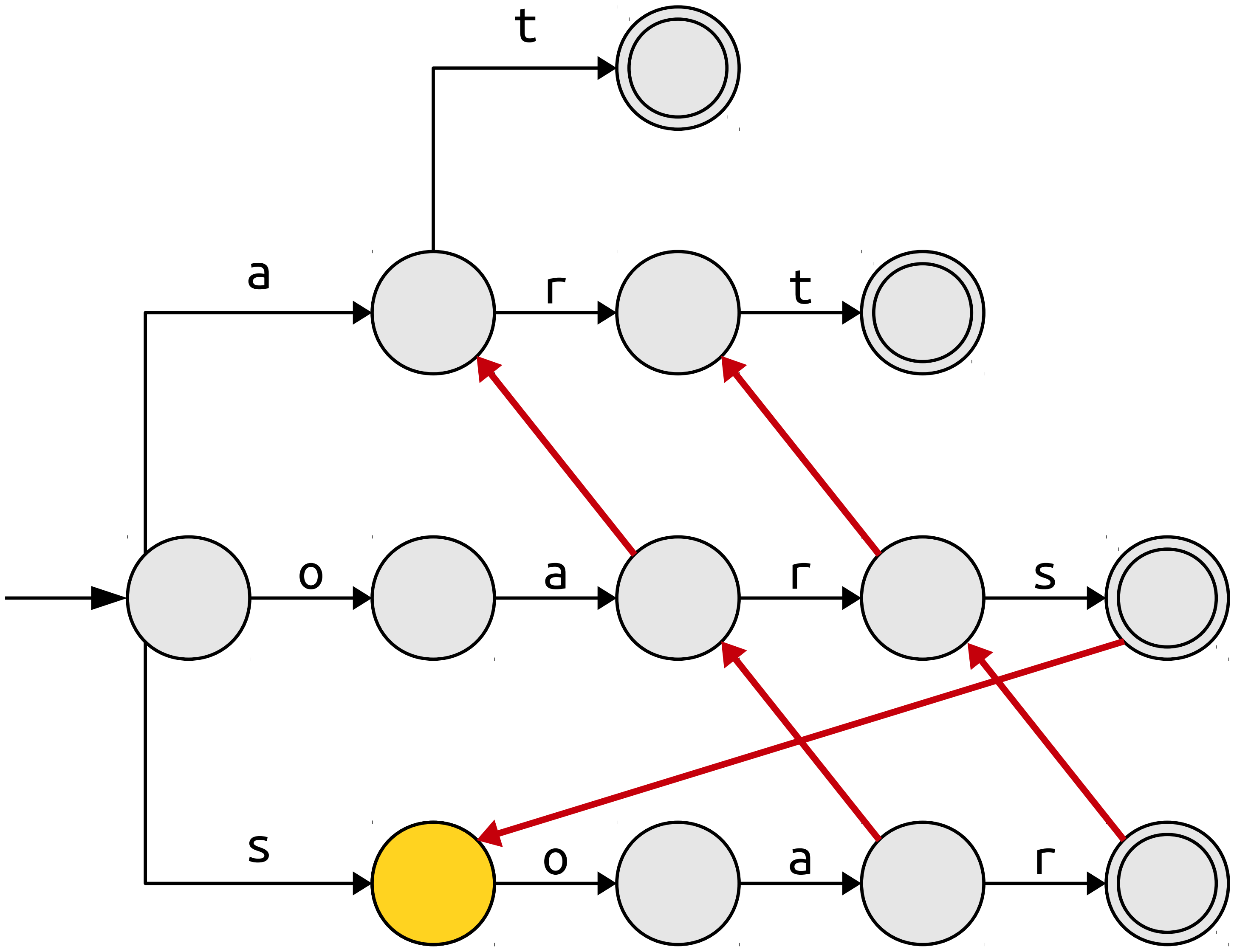


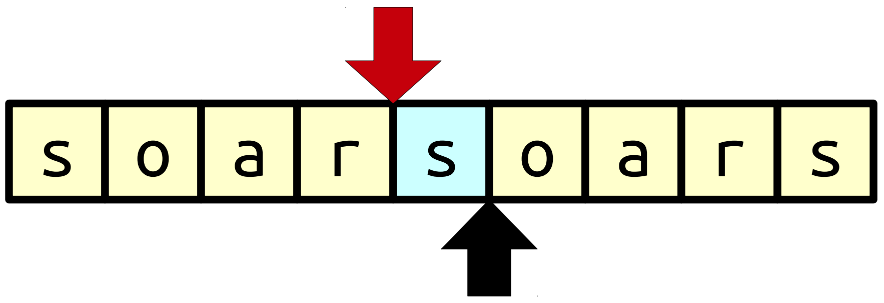
s	o	a	r	s

This red link is called a ***suffix link***. We'll talk about them more formally in a minute.

Pattern Strings

a	t		
a	r	t	
o	a	r	s
s	o	a	r





In general, suffix links might jump the red cursor forward more than one step. The number of steps taken is equal to the change of depth in the trie.

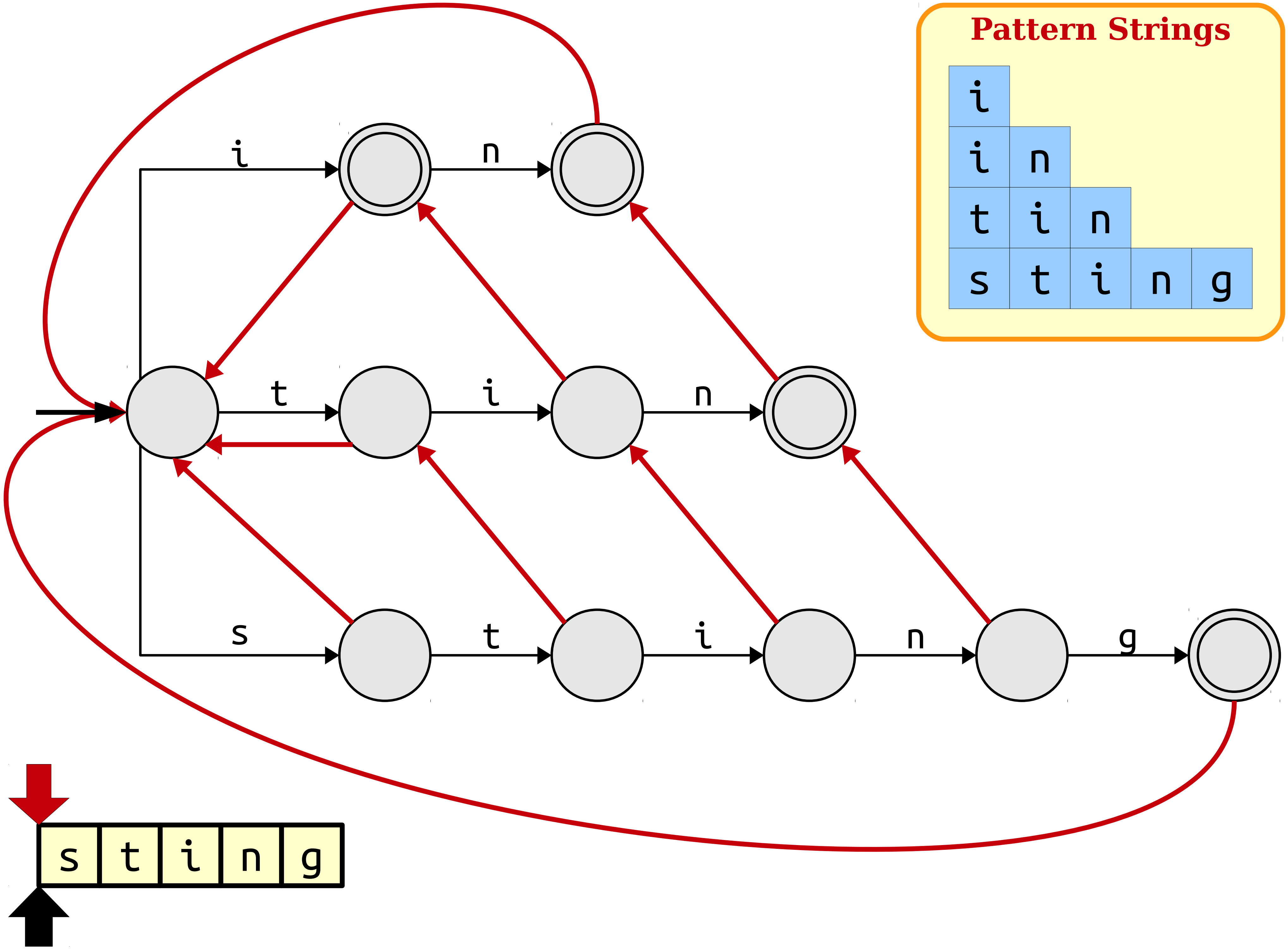
Suffix Links

- A ***suffix link*** (sometimes called a ***failure 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.

Why Suffix Links Matter

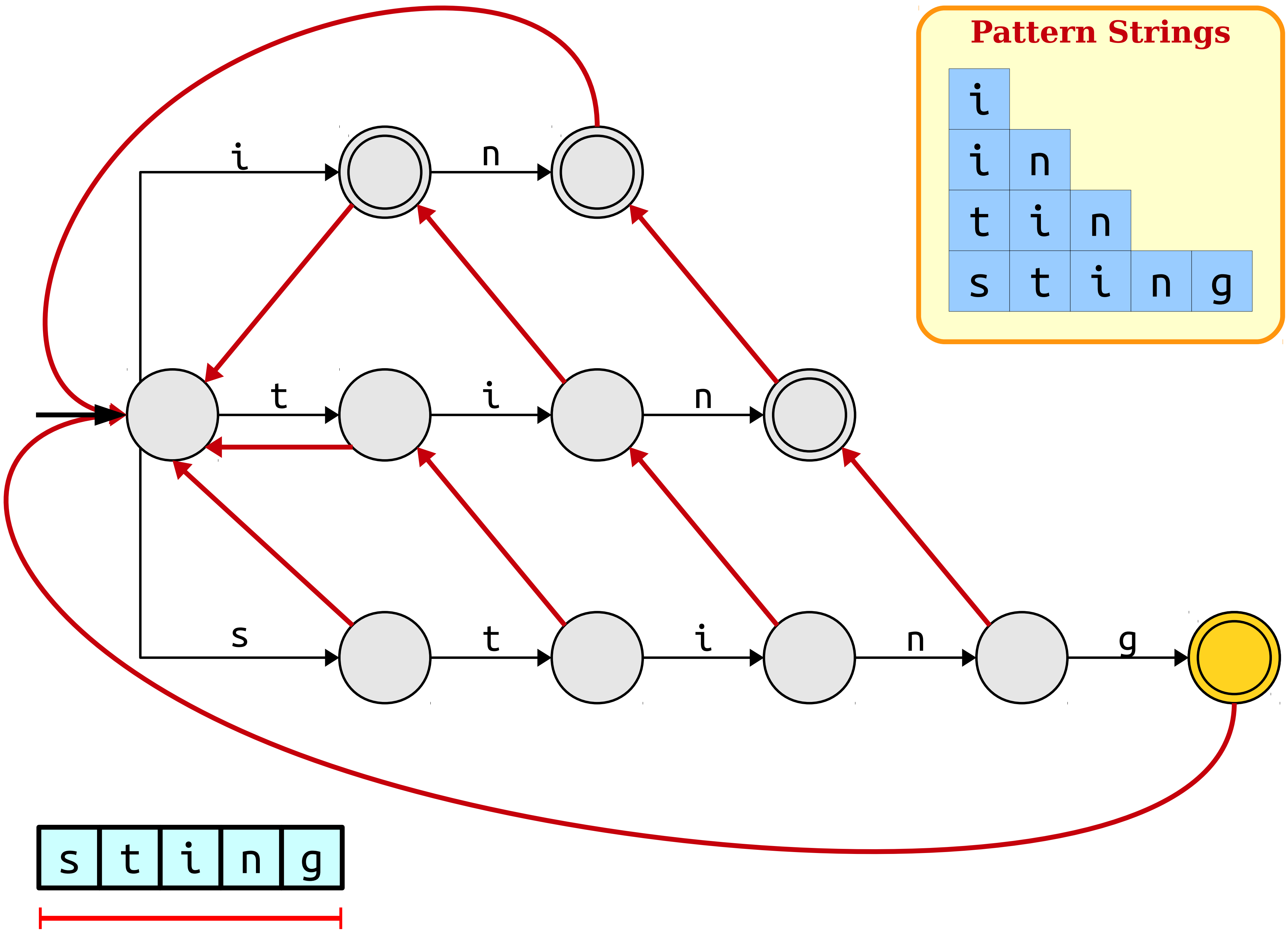
- Suffix links can substantially improve the performance of our string search.
- At each step, we either
  - advance the black (end) pointer forward in the trie, or
  - advance the red (start) pointer forward.
- Each pointer can advance forward at most O(m) times.
- This reduces the amount of time spent 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 later.

A Problem with our Optimization



Pattern Strings

i				
i	n			
t	i	n		
s	t	i	n	g



Pattern Strings

i

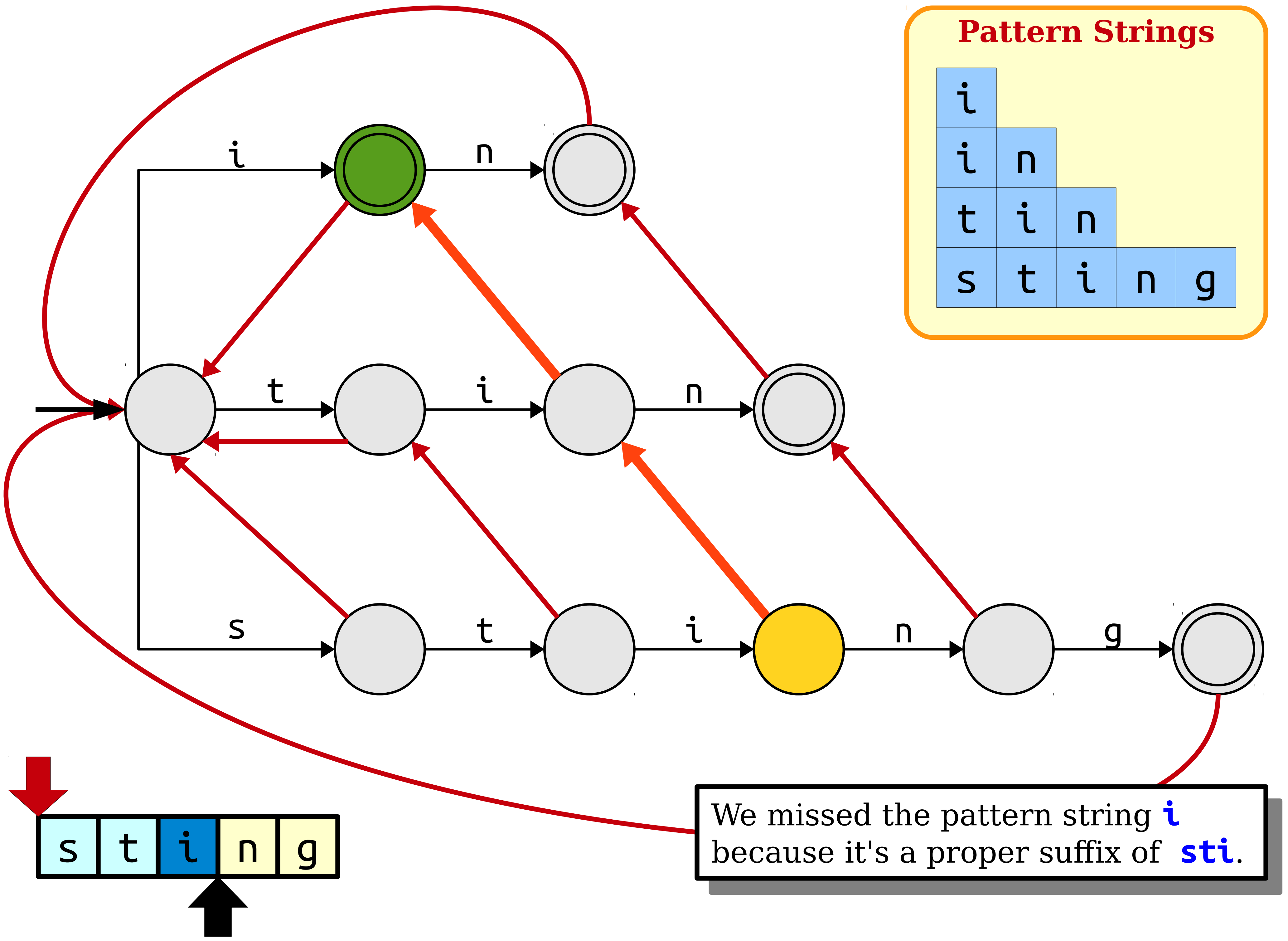
in

tin

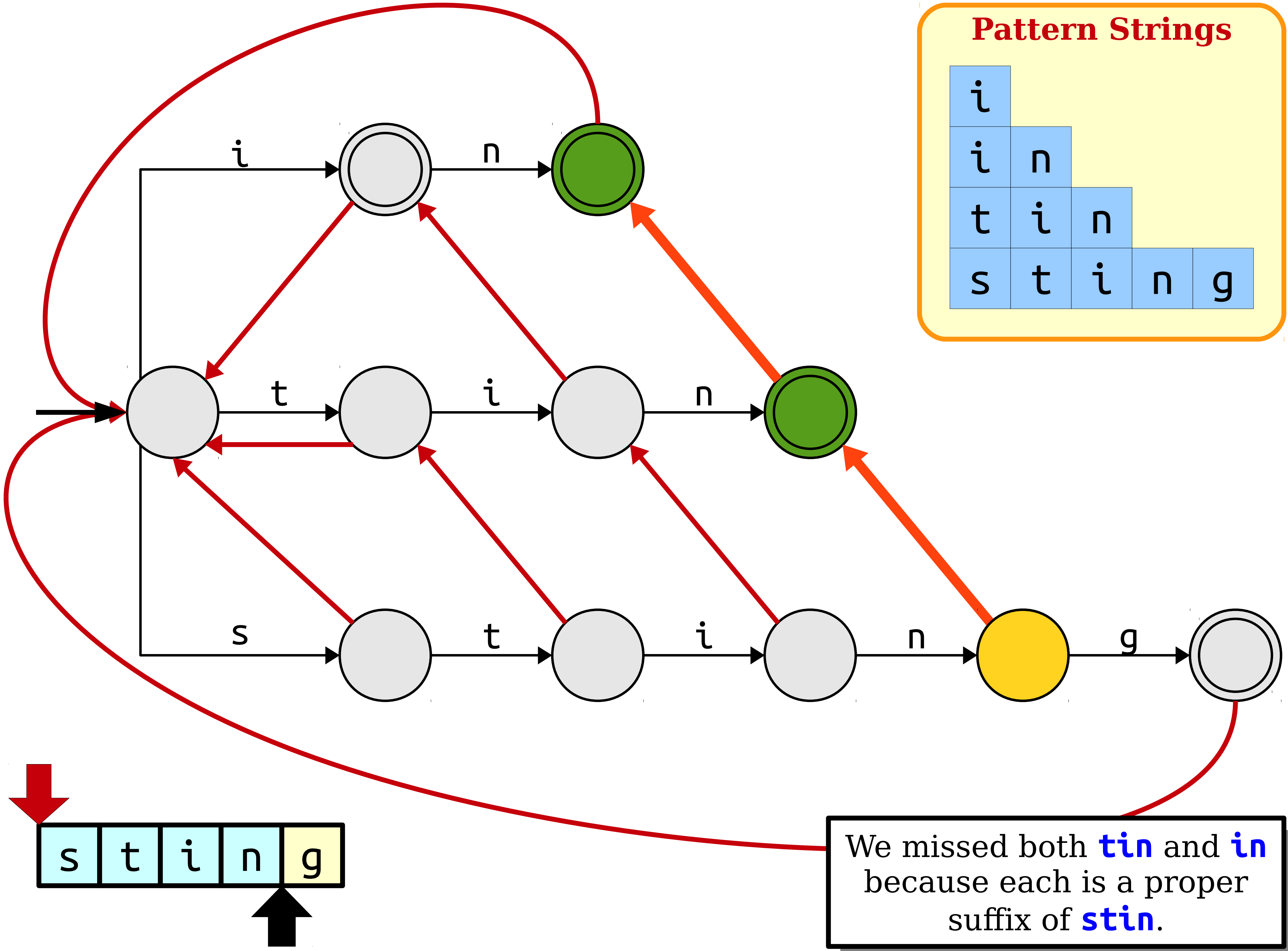
sting

What Happened?

- Our heavily optimized string searcher no longer starts searching from each position in the string.
- As a result, we now might forget to output matches in certain cases.
- We need to figure out
  - when this happens, and
  - how to correct for it.



We missed the pattern string **i** because it's a proper suffix of **sti**.



Pattern Strings

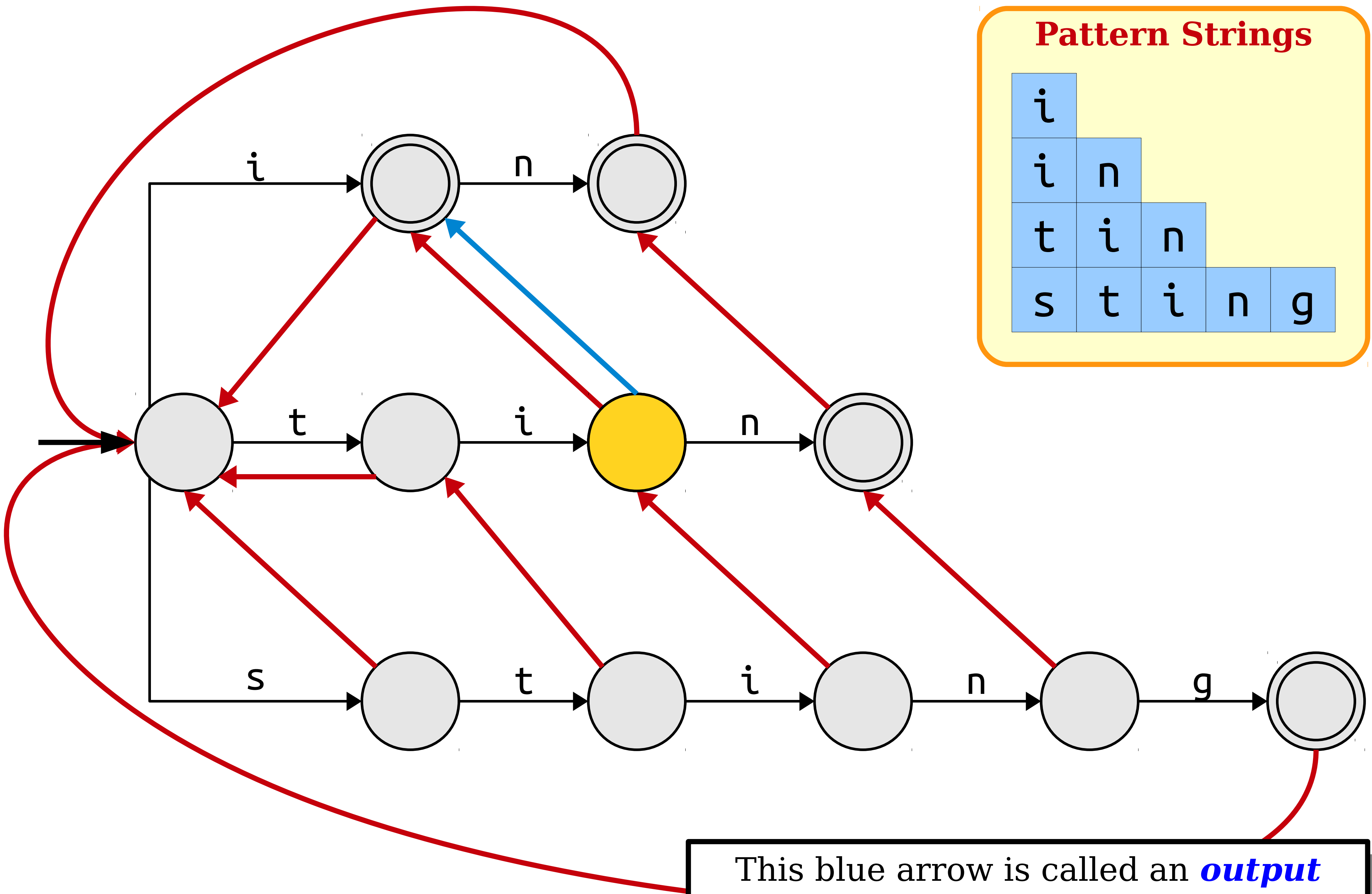
i				
i	n			
t	i	n		
s	t	i	n	g

We missed both **tin** and **in** because each is a proper suffix of **stin**.

How do we address this?

The Problem

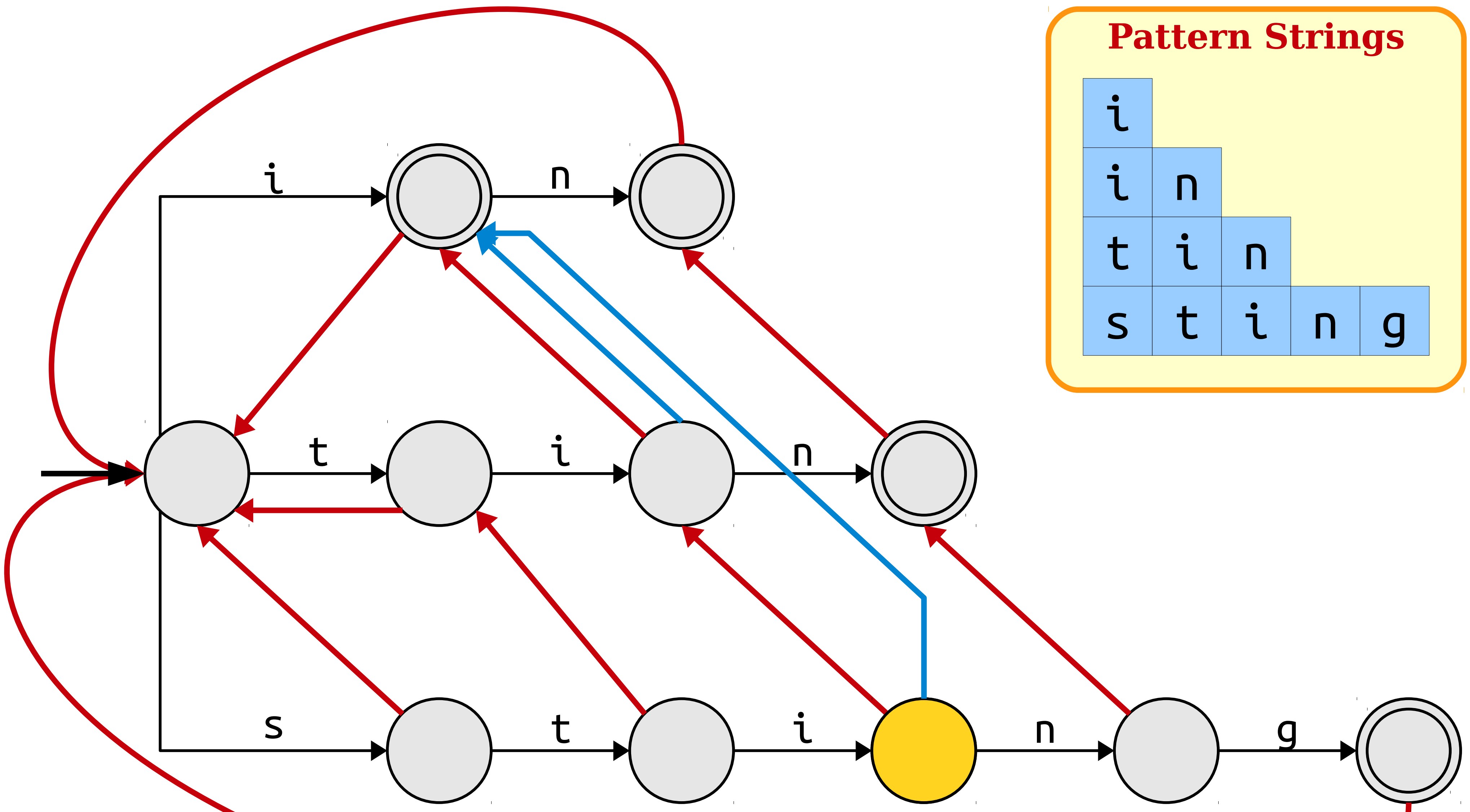
- The approach described previously will ensure that we don't miss any patterns, but it increases the complexity of the search.
- Specifically, we do O($\boldsymbol{L_{max}}$) additional work at each character following suffix links backwards.
- This brings our search cost back up to O($\boldsymbol{mL_{max}}$) again – and that's too slow.
- ***Can we do better?***



Pattern Strings

i				
i	n			
t	i	n		
s	t	i	n	g

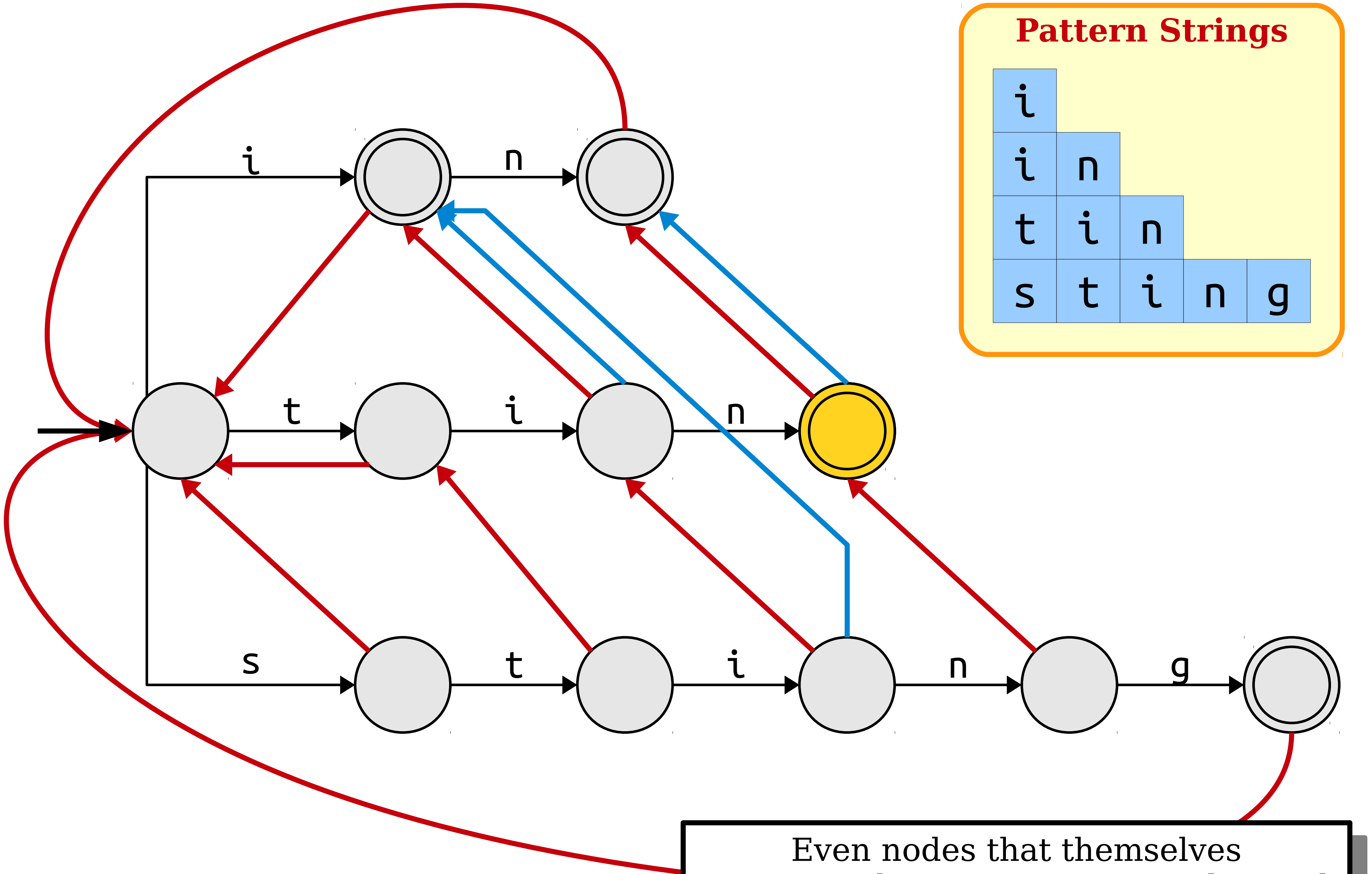
This blue arrow is called an ***output link***. Whenever we visit this gold node, we'll output the string represented by the node at the end of the blue arrow.



Pattern Strings

i				
i	n			
t	i	n		
s	t	i	n	g

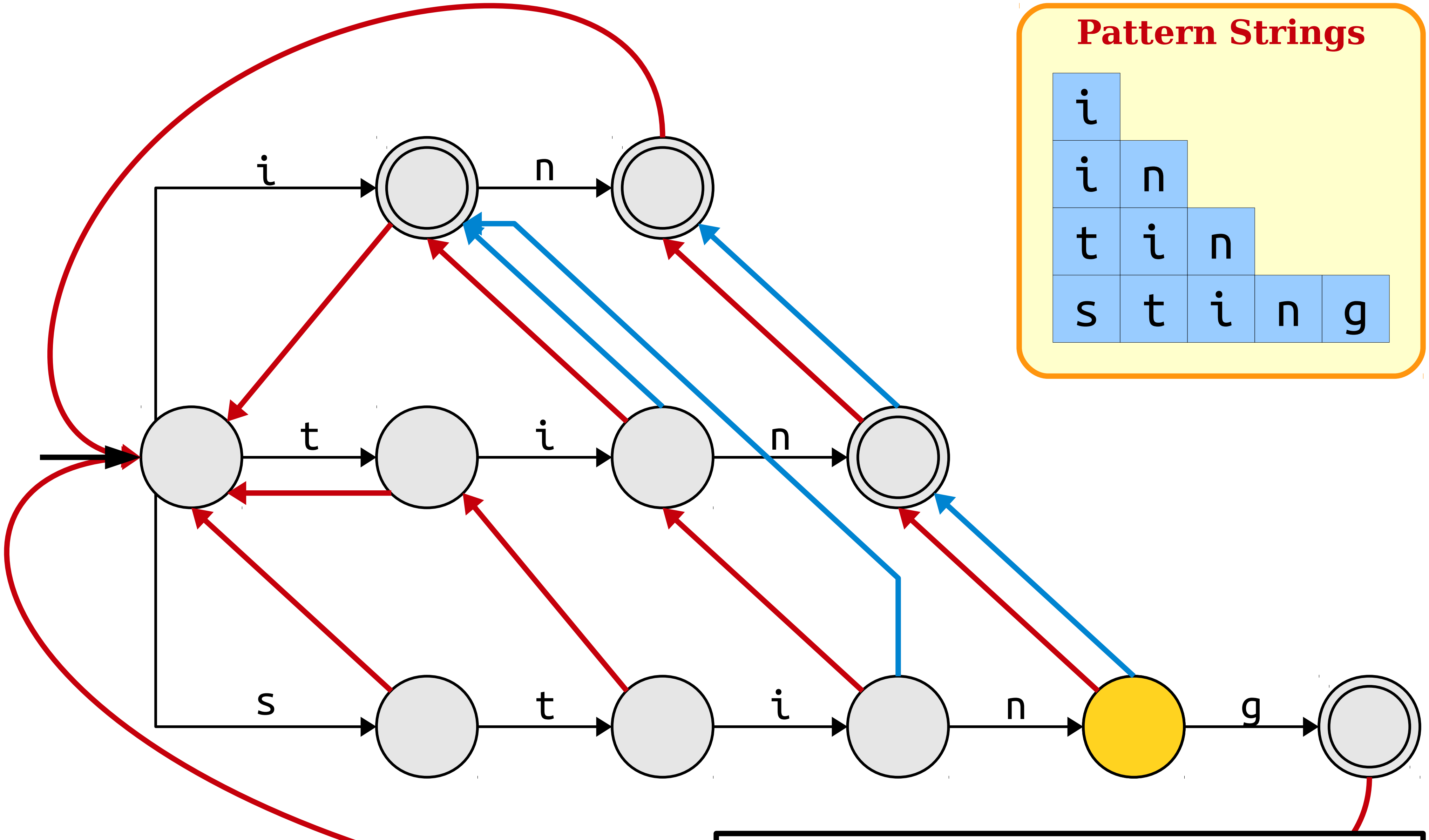
By precomputing where we eventually need to end up, we can instantly read off any extra patterns to emit at this point. As you'll see, we can precompute these links really quickly!



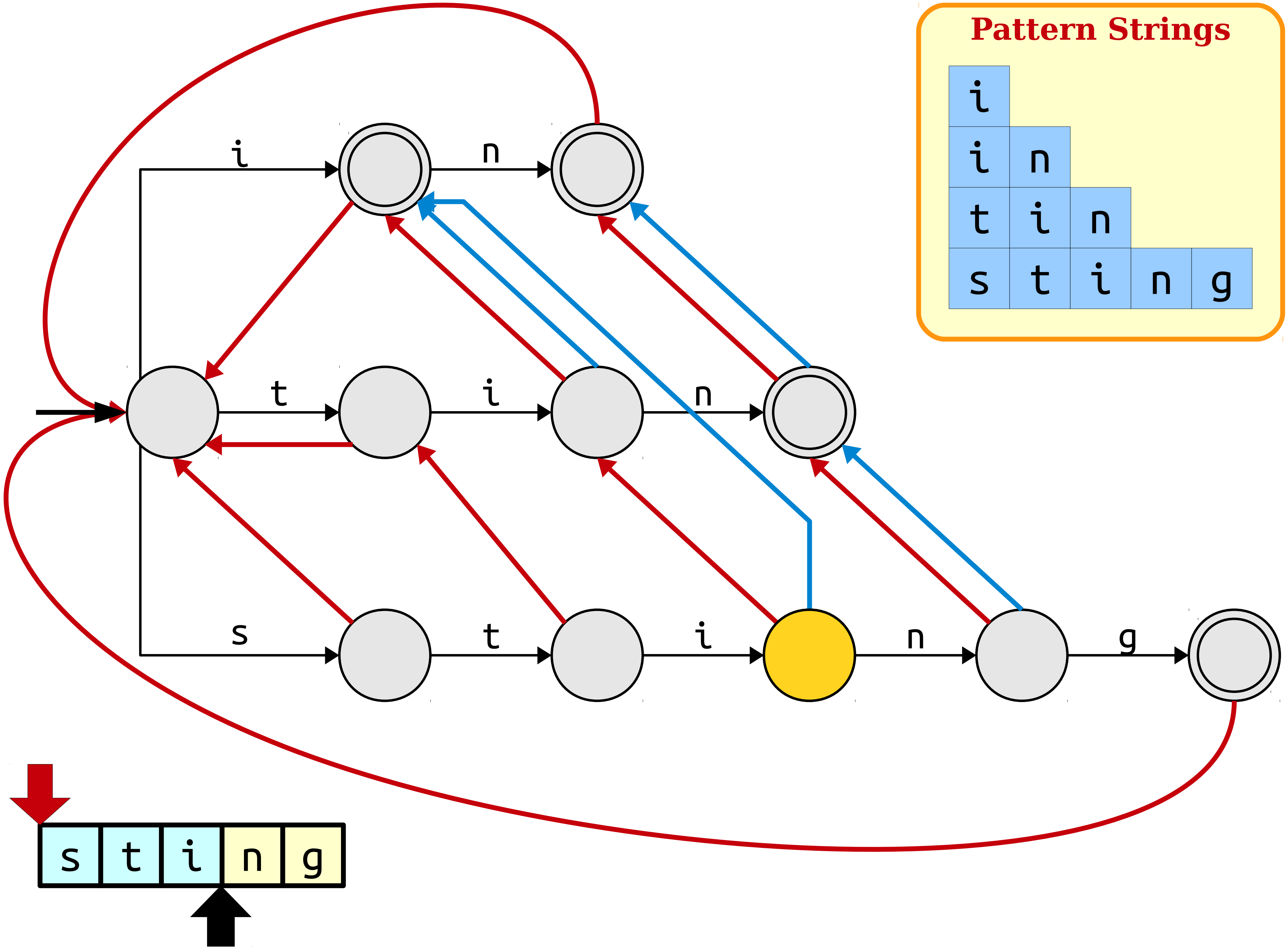
Pattern Strings

i				
i	n			
t	i	n		
s	t	i	n	g

Even nodes that themselves correspond to are patterns might need output links if other patterns also end at the corresponding string.

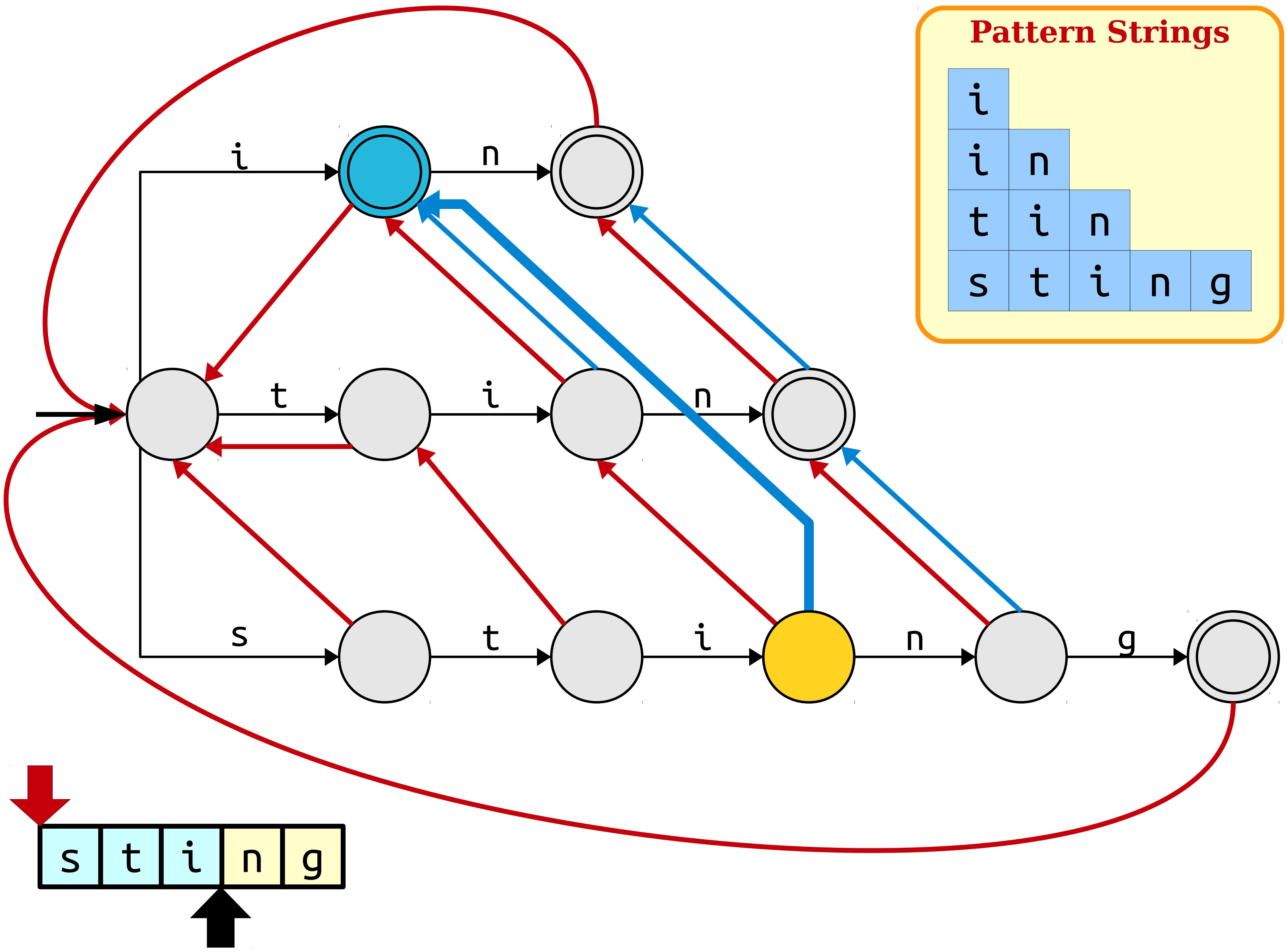


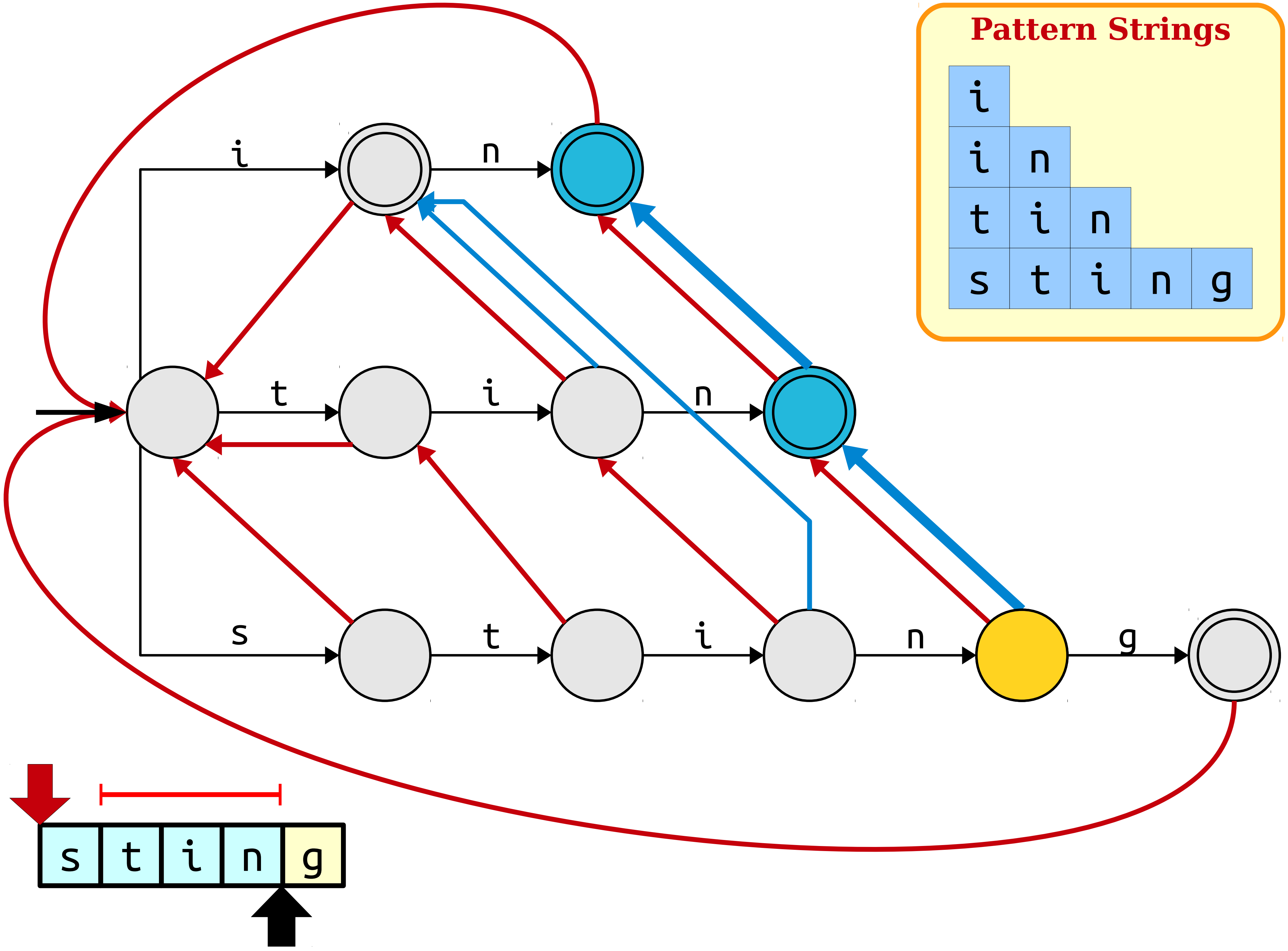
Notice that the blue edges here form a linked list. If we visit this node, we need to output everything in the chain, not just the “tin” node we're immediately pointing at.



Pattern Strings

i				
i	n			
t	i	n		
s	t	i	n	g


i
n
t
i
n
s
t
i
n
g
Pattern Strings
i
i n
t i n
s t i n g
s t i n g

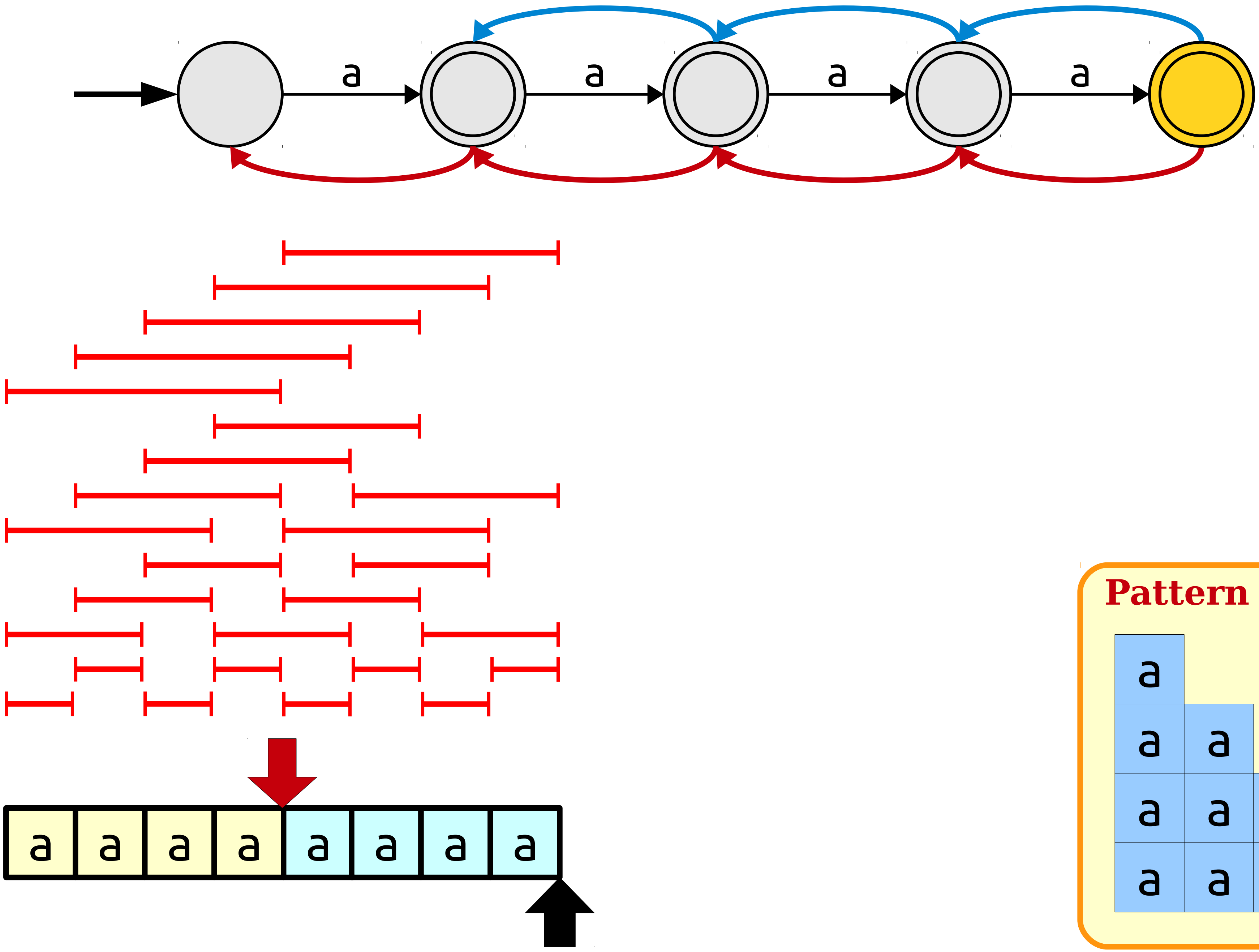


Pattern Strings
i
i n
t i n
s t i n g
i
n
t
i
n
s
t
i
n
g
s t i n g

The Final Matching 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're at the root, break out of this loop.
    - Otherwise, follow a 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.

The Runtime Impact



Pattern Strings

a			
a	a		
a	a	a	
a	a	a	a

The Runtime

- In the worst case, we may have to spend a *huge* amount of time listing off all the matches in the string.
- This isn't the fault of the algorithm – *any* algorithm that matches strings this way would have to spend the time reporting matches.
- To account for this, let z denote the number of matches reported by our algorithm.
- The runtime of the match phase is then $\mathbf{\Theta(m + z)}$, with the m term coming from the string scanning and the z term coming from the matches.
    - You sometimes hear algorithms whose runtime depends on how much output is generated referred to as ***output-sensitive algorithms***.

Where We Are

- Given the matching automaton (which is called an ***Aho-Corasick automaton*** or an ***AC automaton***), we can find all occurrences of the pattern strings in any text of length m in time $\Theta(m+z)$.
- To see whether this is worthwhile, we need to see how quickly we can build the automaton.

Time-Out for Announcements!

Problem Sets

- Problem Set 0 was due today at 2:30PM.
  - You have two 24-hour late days you can use throughout the quarter. Nothing more than 48 hours late will be accepted, since we'll be giving out solutions!
- Problem Set 1 goes out today. It's due next Tuesday at 2:30PM.
  - Play around with the RMQ structures we explored last week!
  - See how well they work in practice!
- As always, stop by office hours or visit Piazza if you have any questions!

HackOverflow

- Stanford WiCS is hosting HackOverflow, a hackathon for programmers of all skill levels. It's coming up on Saturday, April 14 from 10AM – 10PM. Everyone is welcome!
- ***Highly recommended!*** If you've never been to a hackathon before, this is one of the best places to start.

Statistics for Social Good

- Stanford Statistics for Social Good is holding a spring quarter event on ***Thursday, April 12*** from 4PM – 6PM in the BEAM conference room.
- Interesting? RSVP using **this link**.

Back to CS166!

Building the Aho-Corasick Automaton

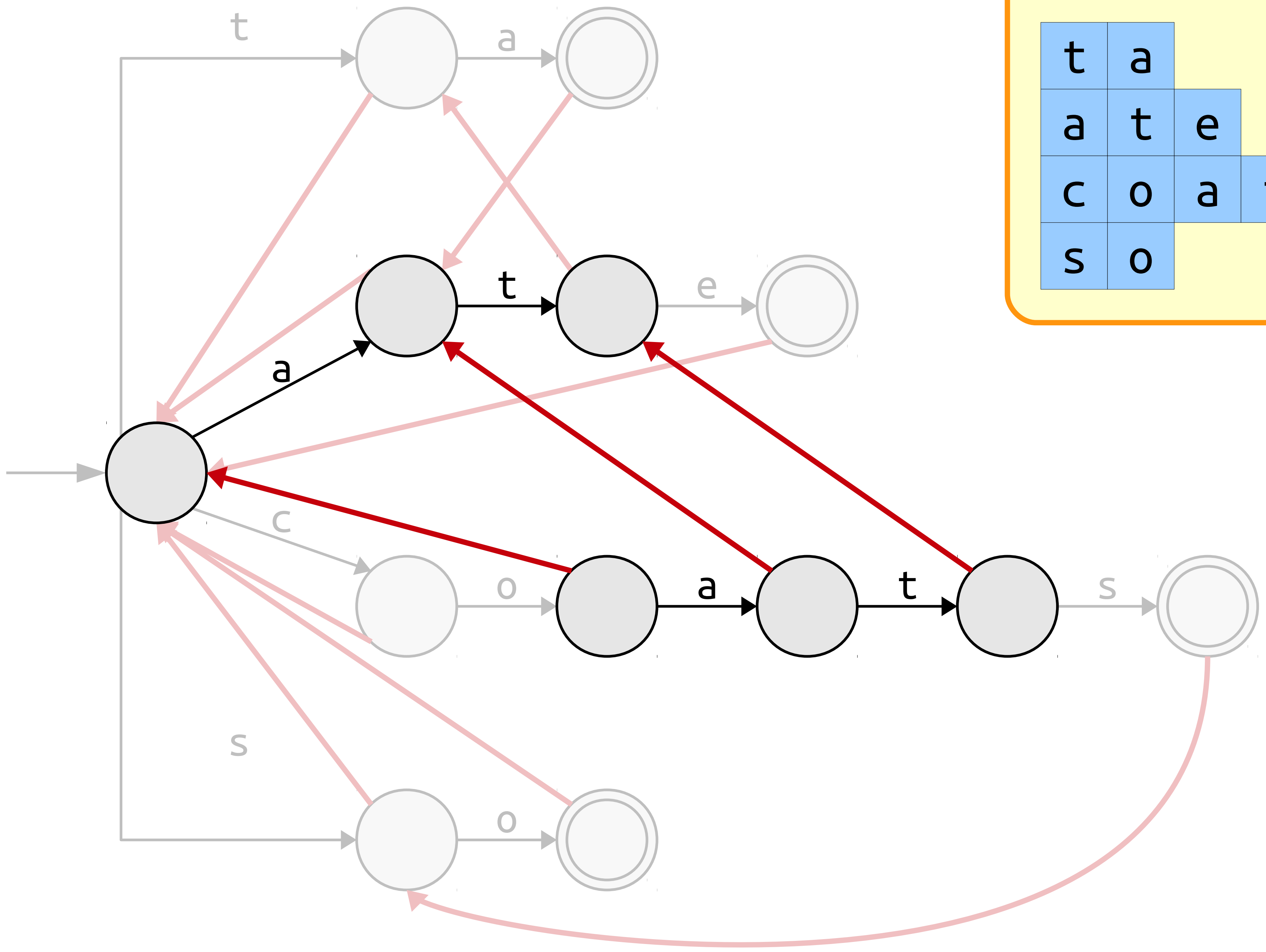
Building the Automaton

- To construct the Aho-Corasick automaton, we need to
  - construct the trie,
  - construct suffix links, and
  - construct output links.
- We know we can build the trie in time $\Theta(\boldsymbol{n})$ using our logic from before.
- How quickly can we construct suffix and output links?

Constructing Suffix Links

An Initial Algorithm

- Here is a simple, brute-force approach for computing suffix links:
  - For each node in the trie:
    - Let α be the string that this particular node corresponds to.
    - For each proper suffix ω of α:
      - Look up ω in the trie.
      - If the search ends up at some trie node, point the suffix link there and stop.
- This approach is not very efficient – that doubly-nested loop is exactly the sort of thing we're trying to avoid.
- ***Can we do better?***



Pattern Strings

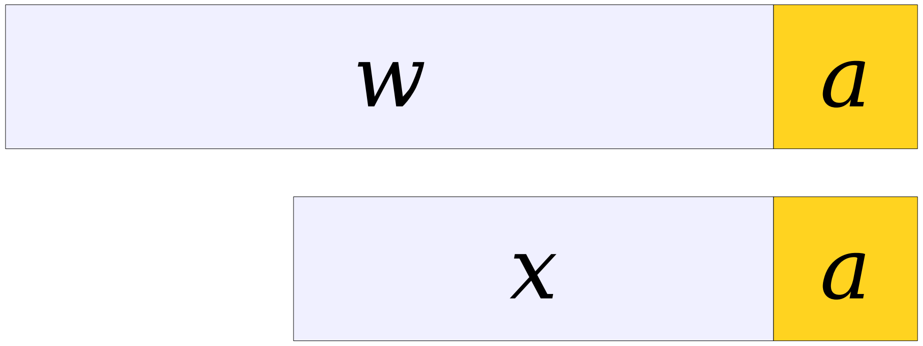
t	a			
a	t	e		
c	o	a	t	s
s	o			

Fast Suffix Link Construction

Constructing Suffix Links

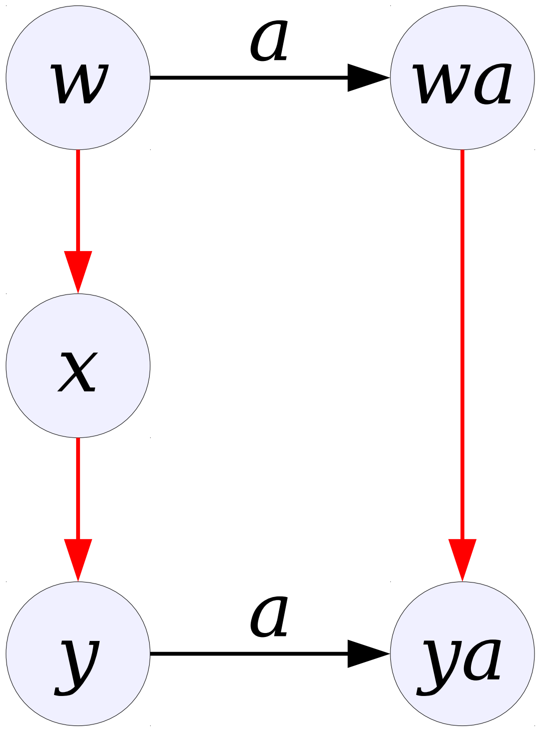
- ***Key insight***: Suppose we know the suffix link for *w* points to some string *x*. We can say a lot about the suffix link for *wa*.
- ***Case 1:*** *xa* exists.

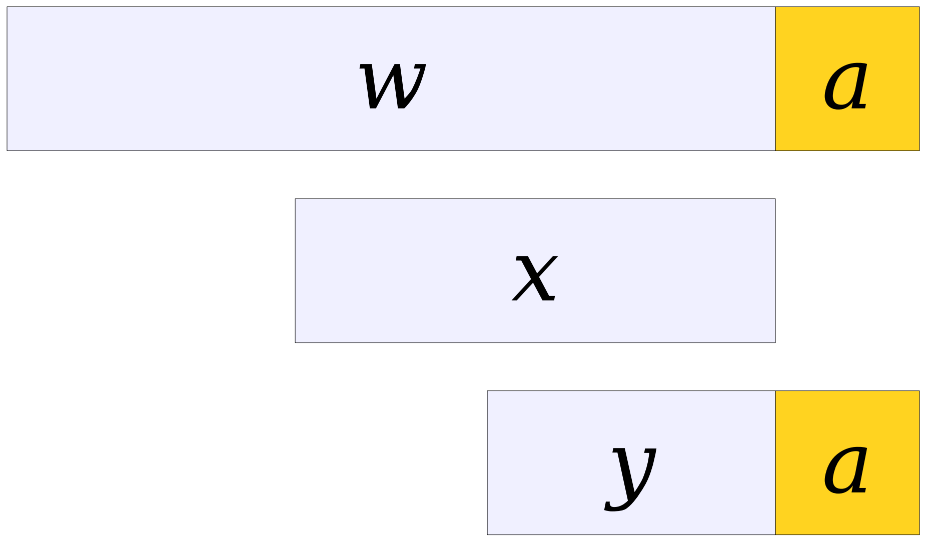
$$\begin{tikzcd} w \arrow[r, "a"] \arrow[d] & wa \arrow[d] \\ x \arrow[r, "a"] & xa \end{tikzcd}$$



Constructing Suffix Links

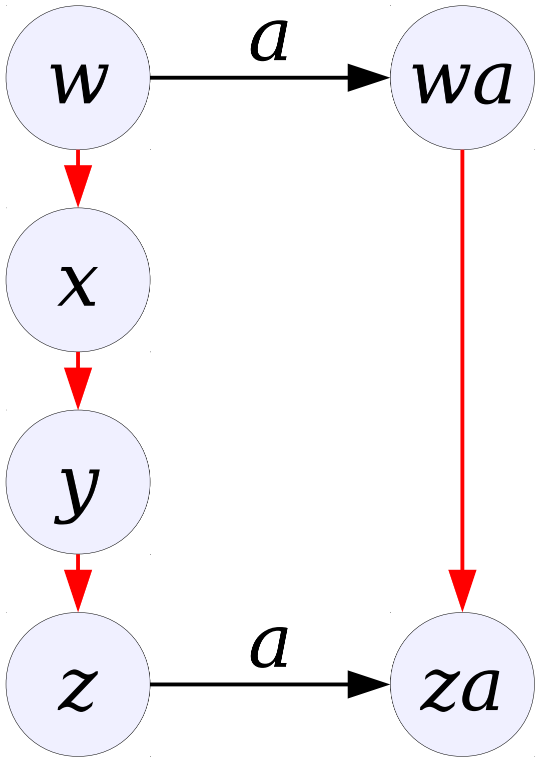
- ***Key insight***: Suppose we know the suffix link for *w* points to some string *x*. We can say a lot about the suffix link for *wa*.
- ***Case 2:*** *xa* does not exist.



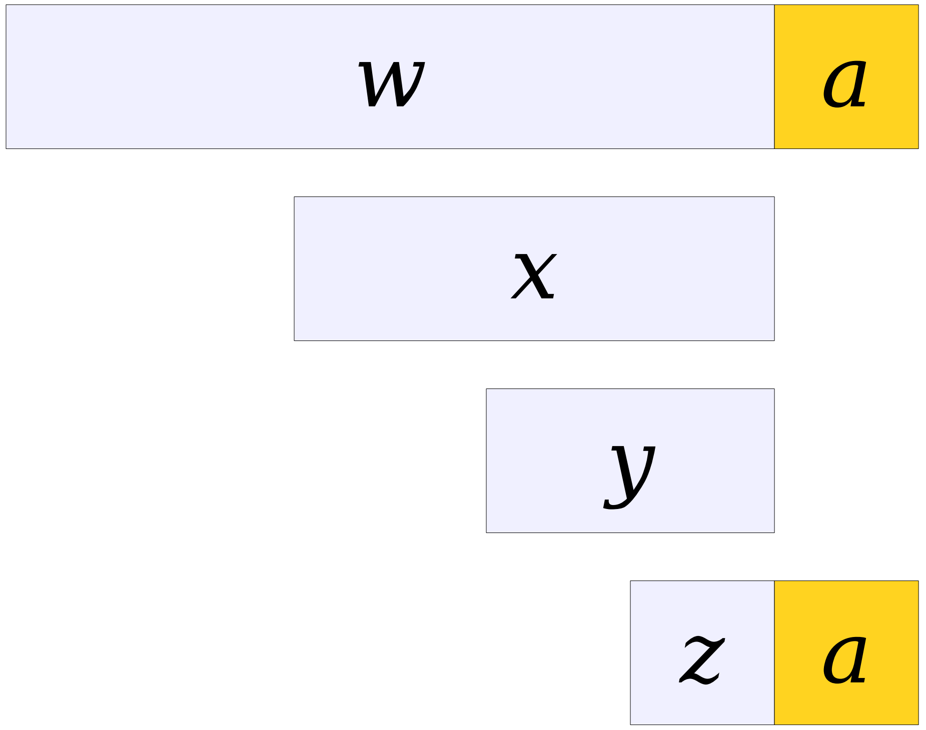


Constructing Suffix Links

- ***Key insight***: Suppose we know the suffix link for *w* points to some string *x*. We can say a lot about the suffix link for *wa*.
- ***Case 2:*** *xa* does not exist.



$$\begin{tikzcd}
w \arrow[r, "a"] \arrow[d] & wa \arrow[ddd] \\
x \arrow[d] & \\
y \arrow[d] & \\
z \arrow[r, "a"] & za
\end{tikzcd}$$



Constructing Suffix Links

- 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.
- ***Observation 1:*** Suffix links point from longer strings to shorter strings.
- ***Observation 2:*** If we precompute suffix links for nodes in ascending order of string length, all of the information needed for the above approach will be available at the time we need it.

Constructing Suffix Links

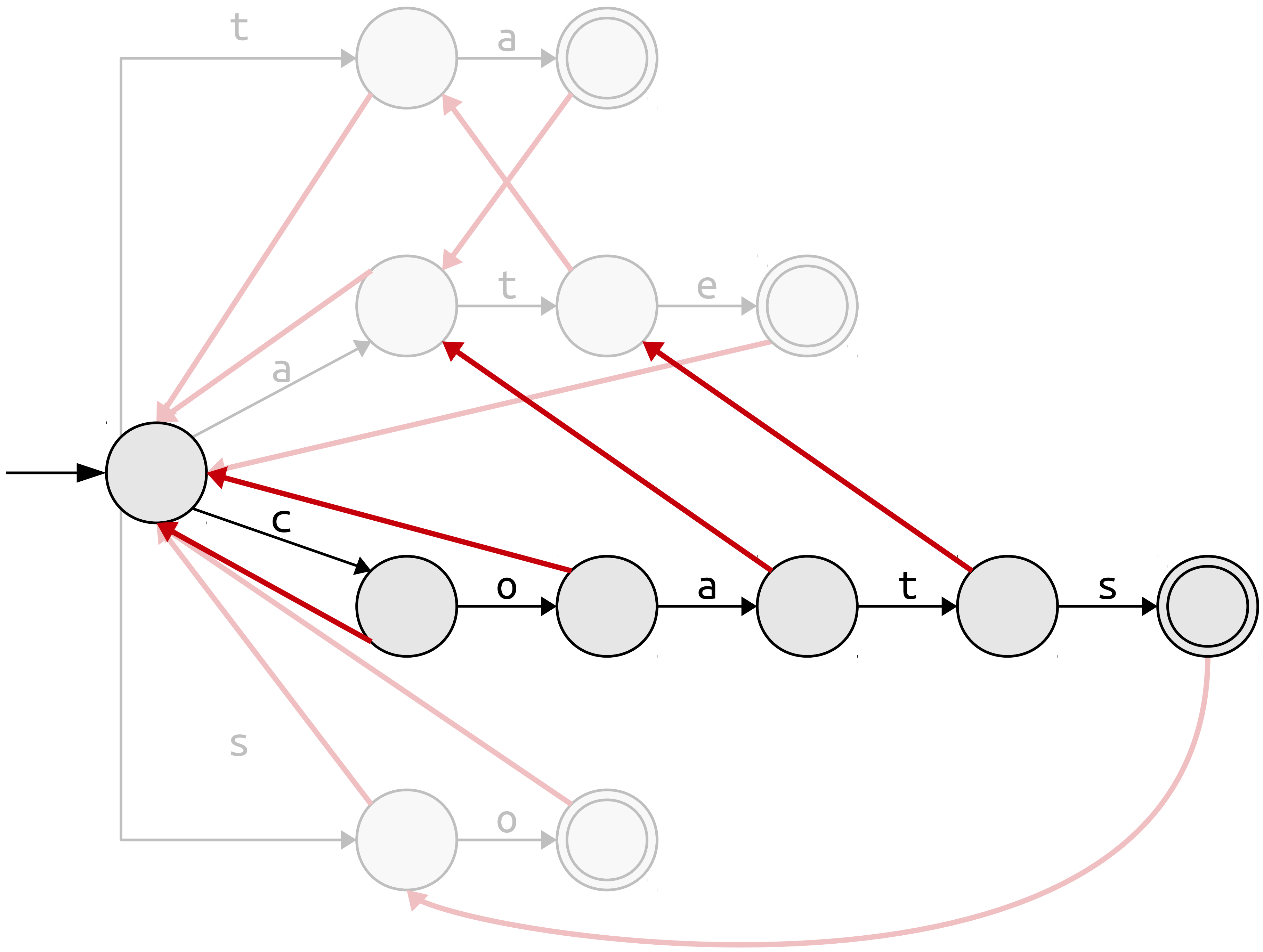
- Do a breadth-first search of the trie, performing the following operations:
  - If the node is the root, it has no suffix link.
  - If the node is one hop away from the root, its suffix link points to the root.
  - Otherwise, the node corresponds to some string *wa*.
  - Let *x* be the node pointed 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.

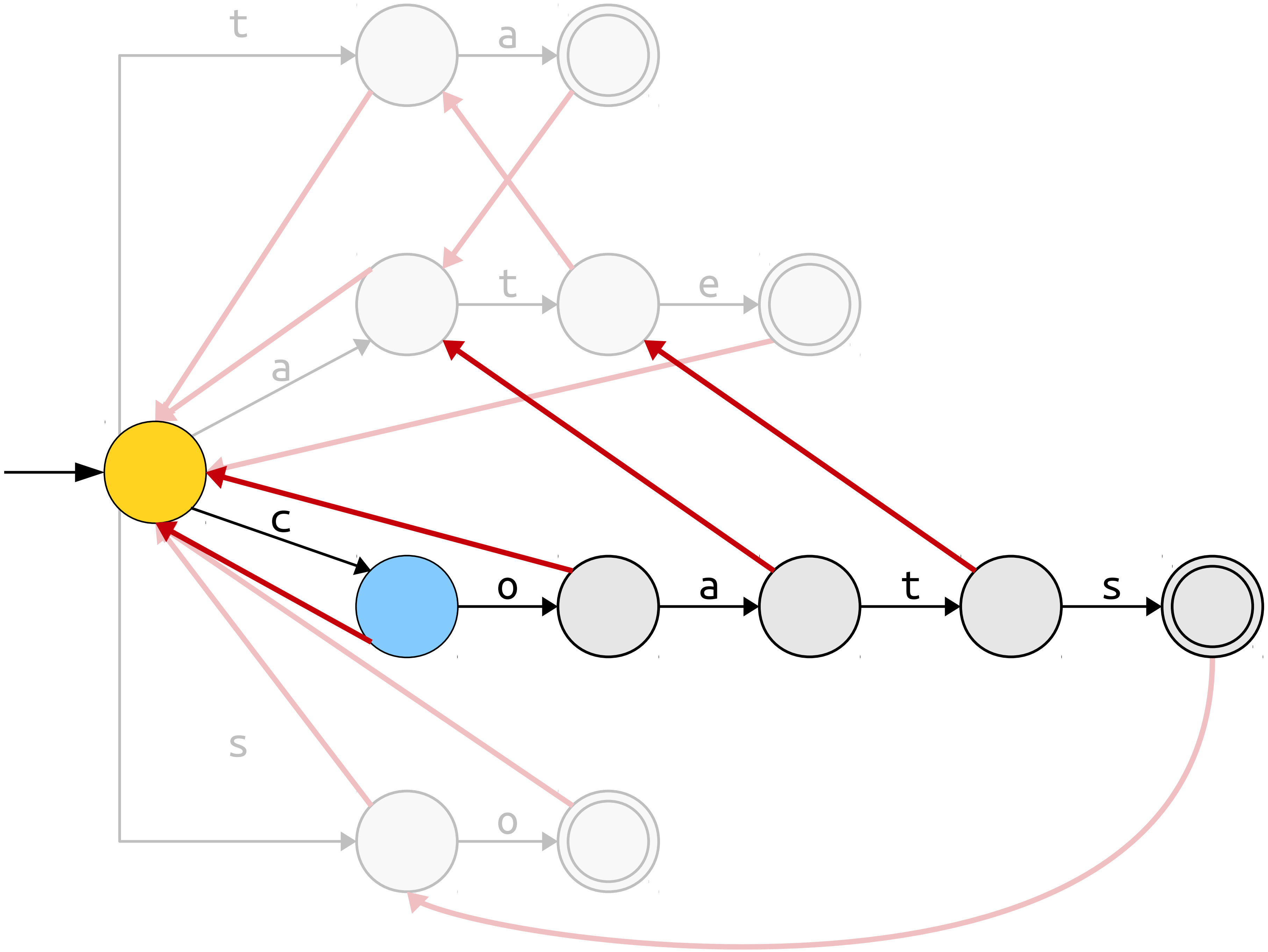
Analyzing Efficiency

- How much time does it take to actually build all the suffix links?
- When filling in any individual suffix link, we might have to keep walking backwards in the trie following suffix links repeatedly while searching for a place to extend.
- Intuitively, it seems like it should be quadratic in the length of the longest string in the trie.
- Is that bound tight?

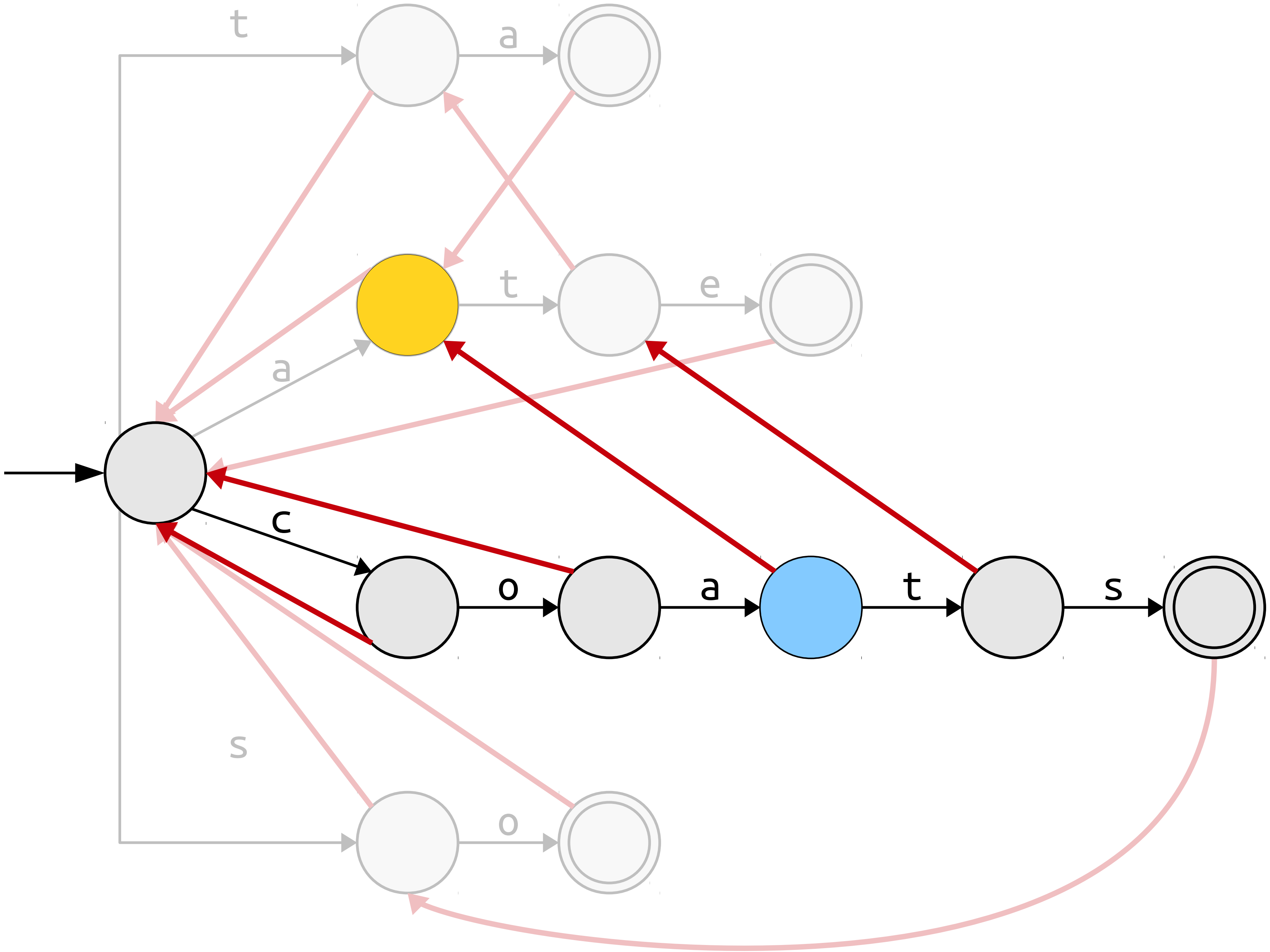
Analyzing Efficiency

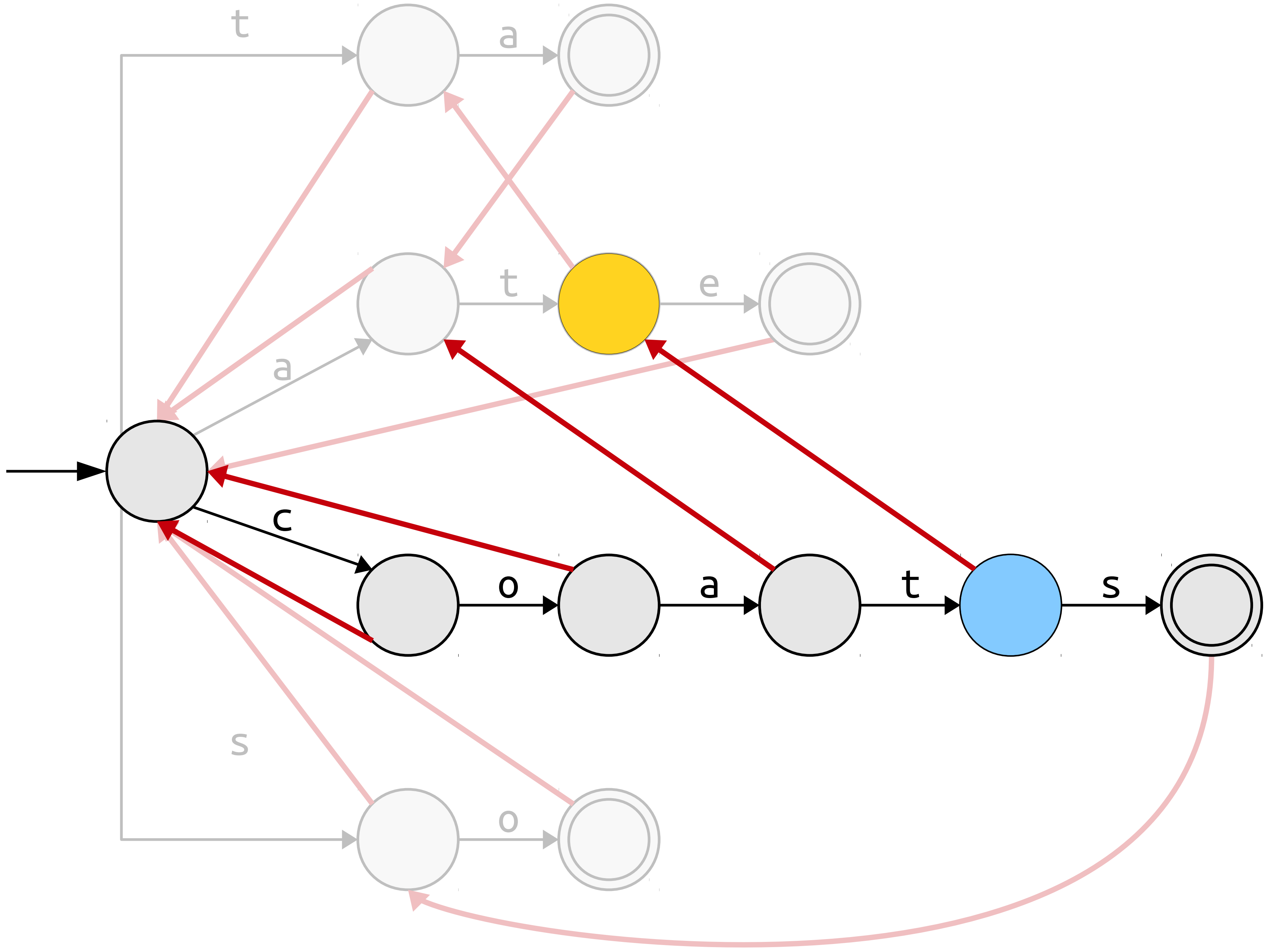
- ***Claim:*** The previously-described algorithm for computing suffix links takes time O(***n***).
- ***Intuition:*** Focus on any one word in the trie. As you add suffix links, keep track of the depth of the node pointed at by the current node's suffix link.

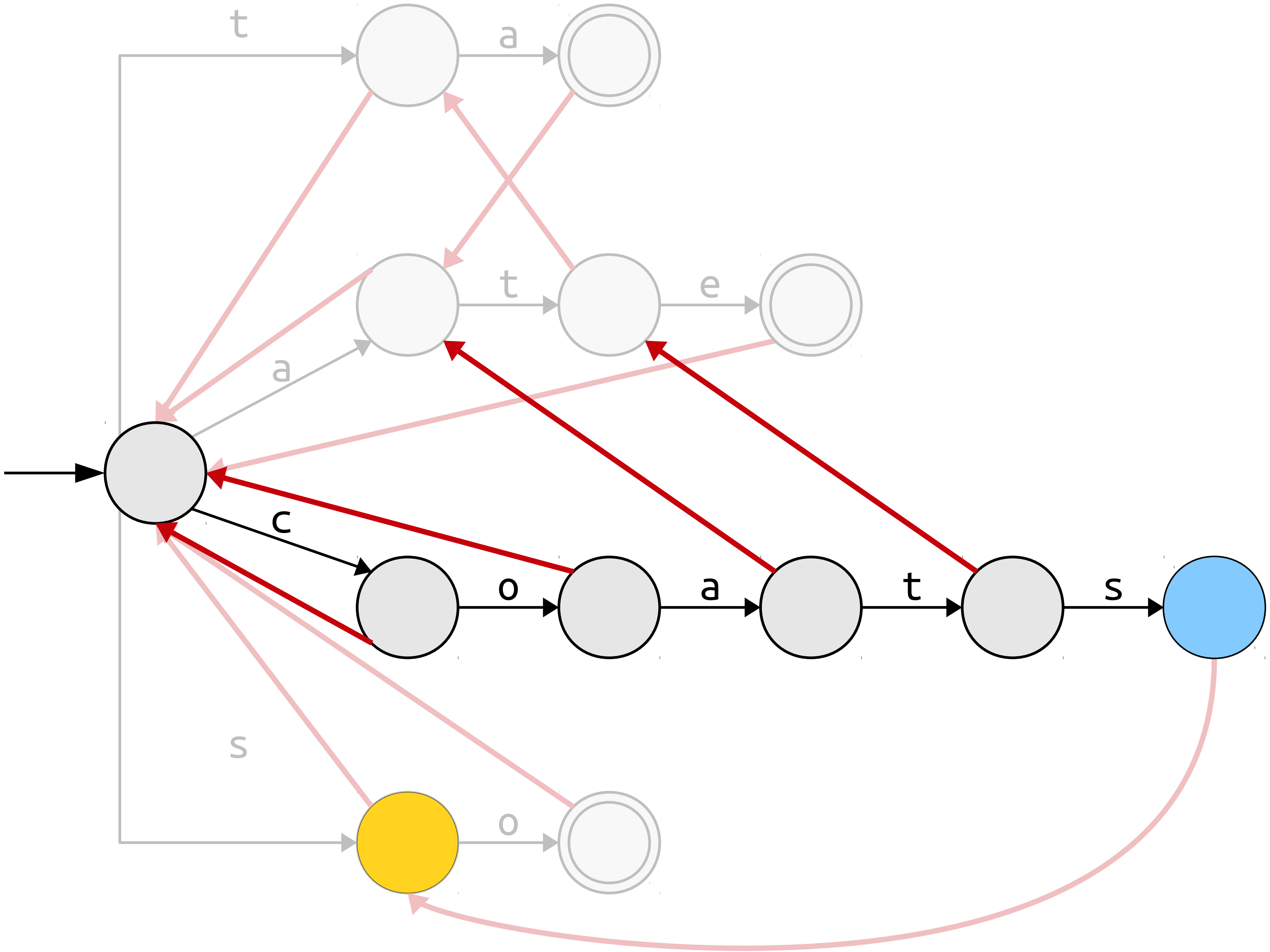

t
a
t
e
a
c
o
a
t
s
s
o



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


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


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



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

Construction Efficiency

- Focus on the time to fill in the suffix links for a single pattern of length h.
- The gold node (where the previous suffix link points) begins at the root. At each step, the gold node
  - takes some number of steps backward, then
  - takes at most one step forward.
- The gold node cannot take more steps backward than forward. Therefore, across the entire construction, the gold node takes at most h steps backward.
- Total time required to construct suffix links for a pattern of length h: $O(h)$.
- Total time required to construct *all* suffix links: $\mathbf{O(\boldsymbol{n})}$.

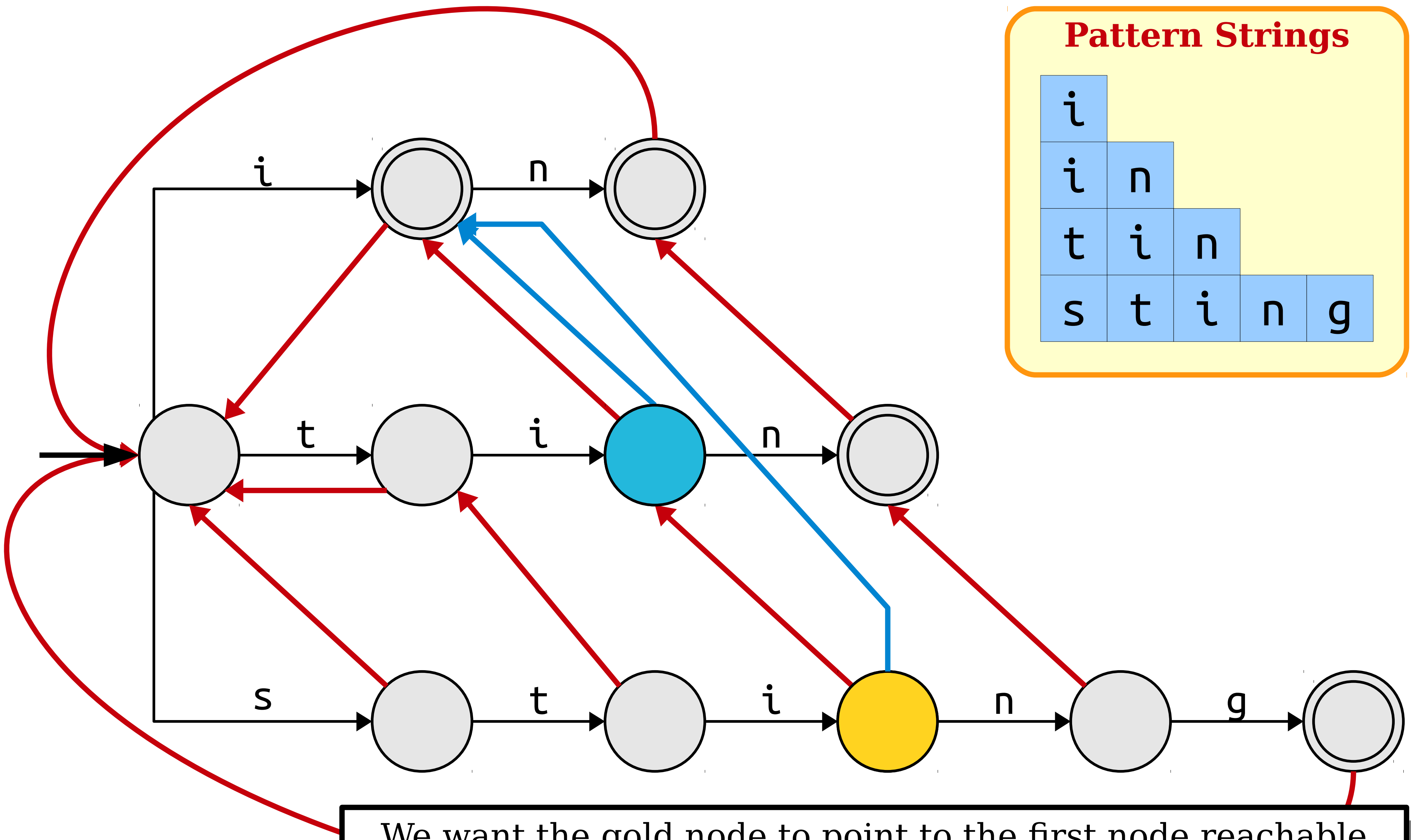
Computing Output Links

The Idea

- Some trie nodes represent strings that have a pattern string as a proper suffix.
- Our goal is to introduce output links so that, when these nodes are visited, the automaton outputs all the suffixes that end there.

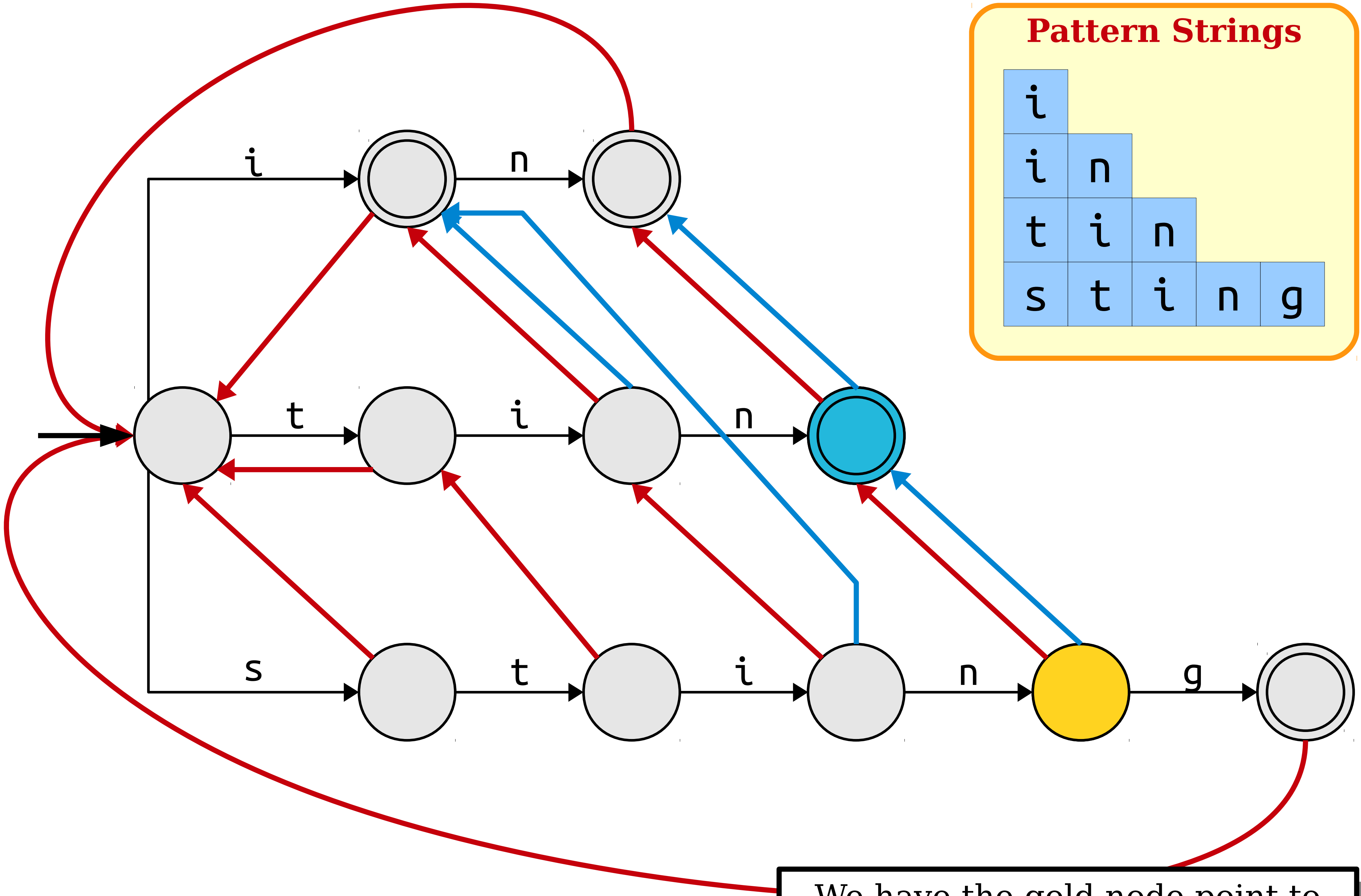
Output Links, Formally

- The output link at a node corresponding to a string *w* points to
    - the node corresponding to the longest proper suffix of *w* that is a pattern, or
    - null if no such suffix exists.
- By always pointing to the node corresponding to the *longest* such word, we ensure that we chain together all the patterns using output links.



We want the gold node to point to the first node reachable by suffix links that's also a pattern.

The blue node (at the end of the suffix link) isn't a pattern, but it knows where the first pattern is. We set the gold node's output link to equal the blue node's output link.



Pattern Strings

i				
i	n			
t	i	n		
s	t	i	n	g

We have the gold node point to the blue node because the blue node corresponds to a word.

Filling In 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.
- Time complexity of building all output links: **O(*n*)**.

The Net Complexity

- Our preprocessing time is
  - $\Theta(\boldsymbol{n})$ work to build the trie,
  - $O(\boldsymbol{n})$ work to fill in suffix links, and
  - $O(\boldsymbol{n})$ work to fill in output links.
- Total preprocessing time: **$\Theta(\boldsymbol{n})$**.

The Final Totals

- We now have a multi-string search data structure with time complexity

$$\langle \mathbf{O}(\boldsymbol{n}), \mathbf{O}(\boldsymbol{m} + \boldsymbol{z}) \rangle.$$

- In other words, this is exceptionally good in the case where there are a fixed set of patterns and a variable string to search.

Where We're Going

- A powerful data structure called the suffix tree lets us solve this problem in

$$\langle \mathbf{O}(\boldsymbol{m}), \mathbf{O}(\boldsymbol{n} + \boldsymbol{z}) \rangle.$$

- In other words, it excels when there's a fixed string to search and a variable set of patterns.

More to Explore

- There are a number of other approaches to solving this problem, and there's often a large gap between theory and practice!
- The ***Boyer-Moore*** algorithm searches for a single pattern in a large text. It can actually run in *sublinear time* if the string searched for isn't present, but runs in quadratic case if a match exists.
- The ***Commentz-Walter*** algorithm generalizes Boyer-Moore for multiple strings and has similar time guarantees, but tends to be a little chaotic in practice.
- The ***Knuth-Morris-Pratt*** algorithm is a special case of the Aho-Corasick algorithm when there is just one pattern. (Aho-Corasick is actually a generalization of KMP!)

Next Time

- ***Suffix Trees***
  - A highly versatile, flexible, powerful data structure for string processing.
- ***Suffix Arrays***
  - Shrinking down trie space usage.
- ***Applications of RMQ***
  - Getting some mileage out of Fischer-Heun.