CPSC 490 – Problem Solving in Computer Science

Lecture 10: Aho Corasick

Jason Chiu and Raunak Kumar

2017/01/27

University of British Columbia

From one string to multiple strings

Recall that

- The KMP state is “prefix of string”
- The nodes of a Trie represent prefixes of string, for multiple strings

To search for multiple strings run KMP with trie nodes as states!

From one string to multiple strings

All we need to do is to draw some extra arrows on a trie!

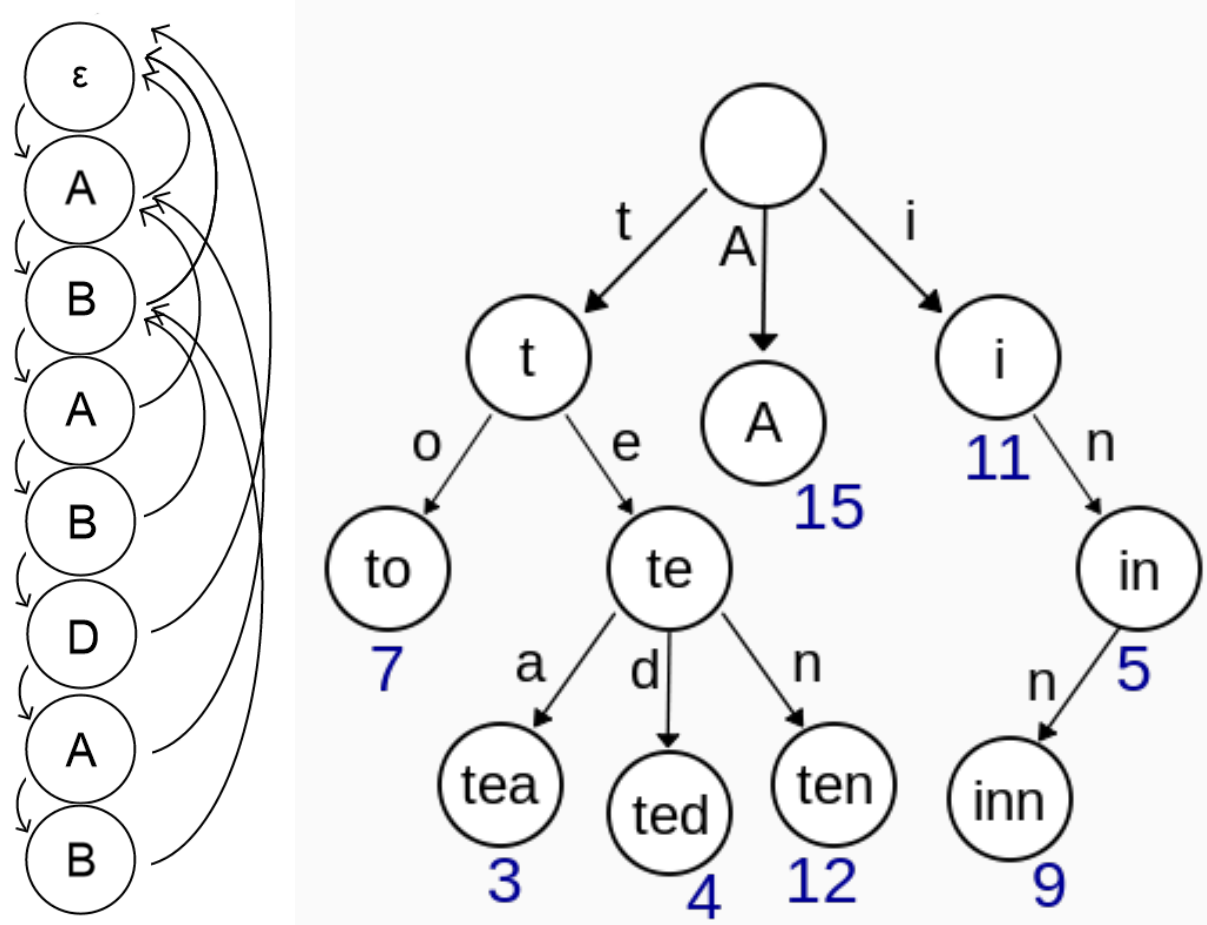
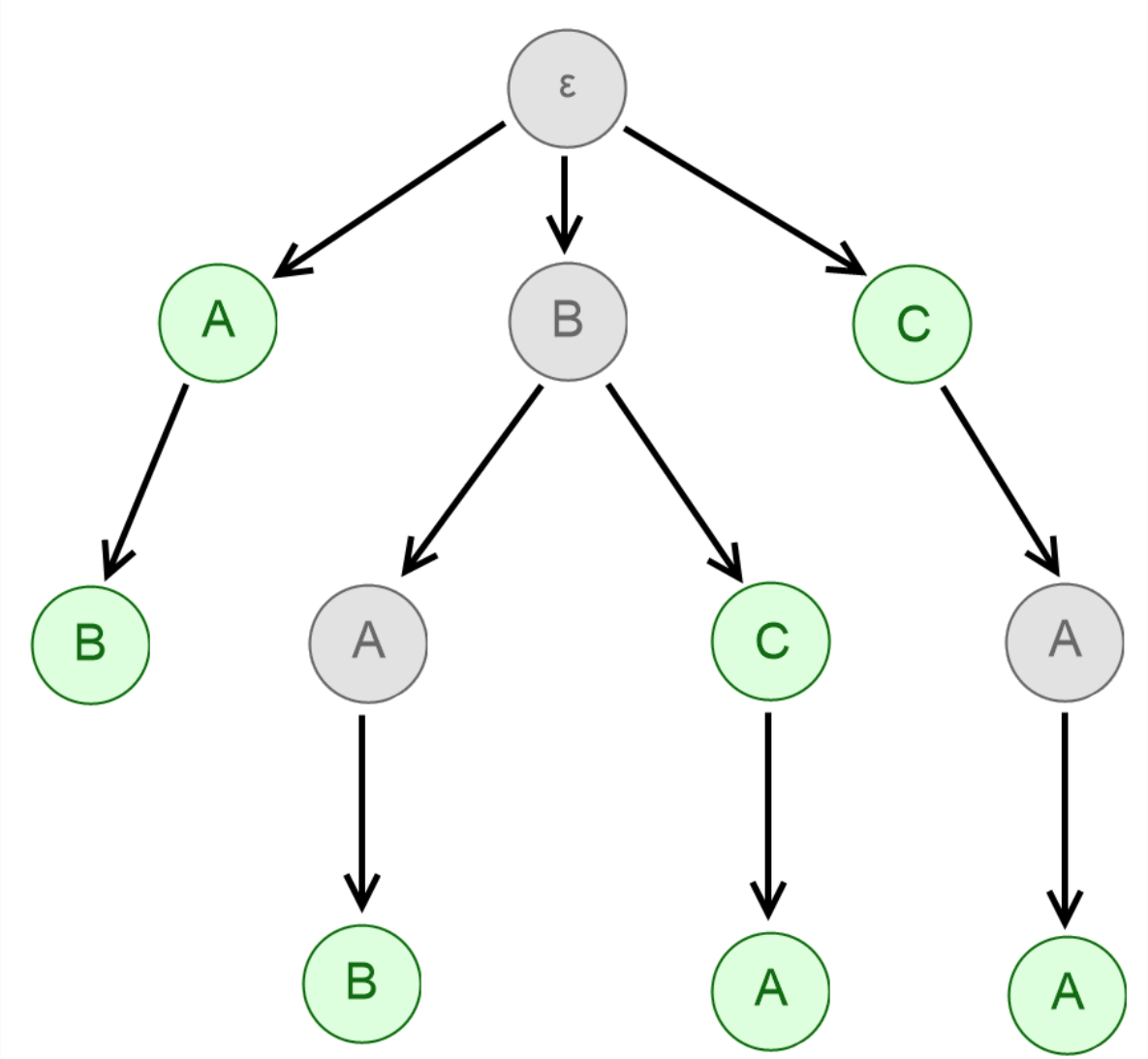


Figure 1: KMP automaton (left) and a trie (right, Source: Wikipedia)

Aho Corasick: Success Arrows

Success arrows are already in a trie! They point from parent to child!



Example: trie has {a, ab, bab, bc, bca, c, caa}

Aho Corasick: Fail Arrows

Q: How do we draw the Fail arrows?

Aho Corasick: Fail Arrows

Q: How do we draw the Fail arrows?

A: Same way! Follow the “previous” arrow until “next” char matches!
Of course, on a trie, “previous” = parent, “next” = children

Aho Corasick: Fail Arrows

Q: How do we draw the Fail arrows?

A: Same way! Follow the “previous” arrow until “next” char matches!
Of course, on a trie, “previous” = parent, “next” = children

$\Rightarrow$ Follow Fail arrow of parent node until we get to a node with current node’s character as child, then go to that child.
$\Rightarrow$ If we reach the root with no success, point arrow to root.

Aho Corasick: Fail Arrows

Q: How do we draw the Fail arrows?

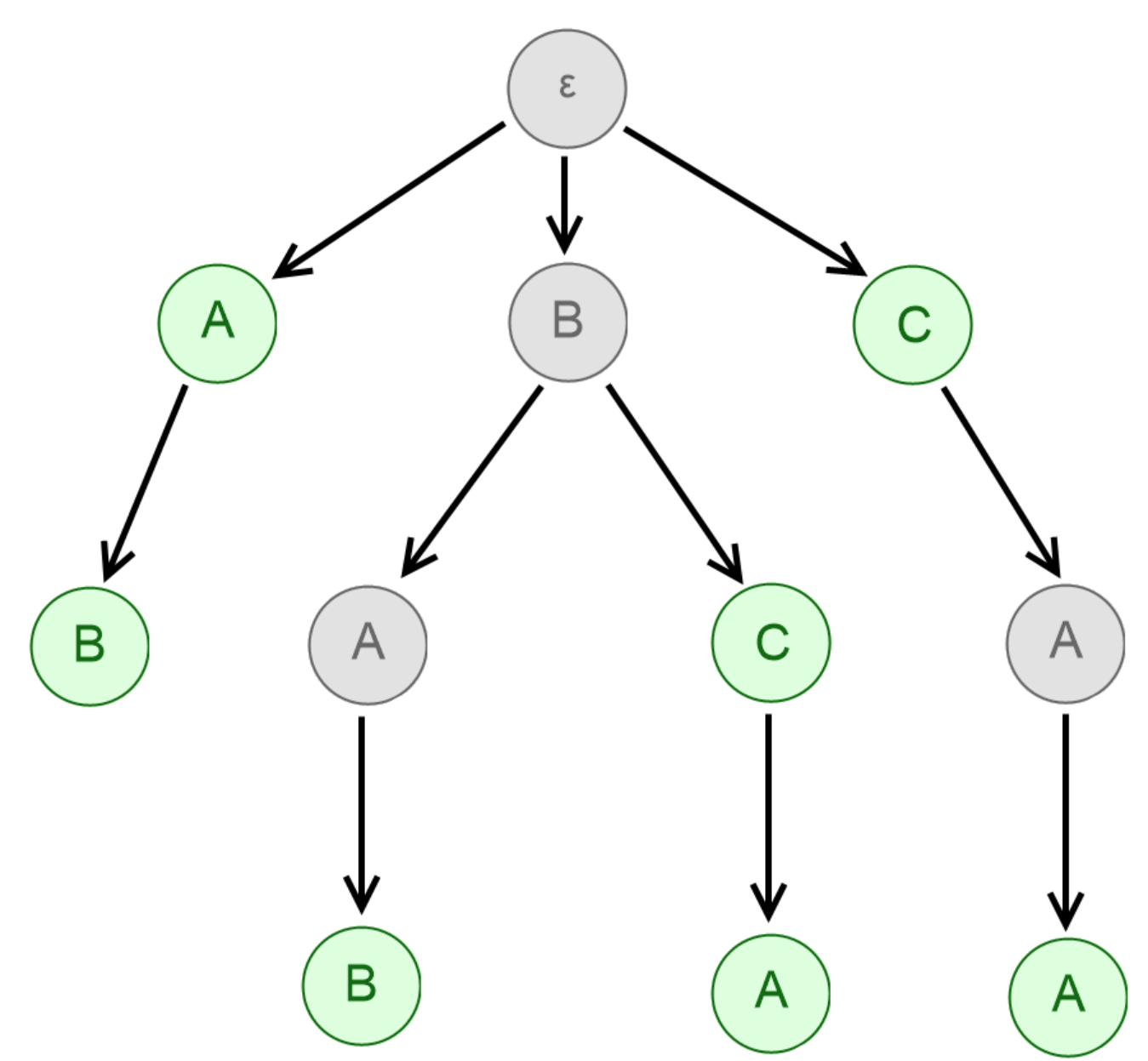
A: Same way! Follow the “previous” arrow until “next” char matches!
Of course, on a trie, “previous” = parent, “next” = children

$\Rightarrow$ Follow Fail arrow of parent node until we get to a node with current node’s character as child, then go to that child.
$\Rightarrow$ If we reach the root with no success, point arrow to root.

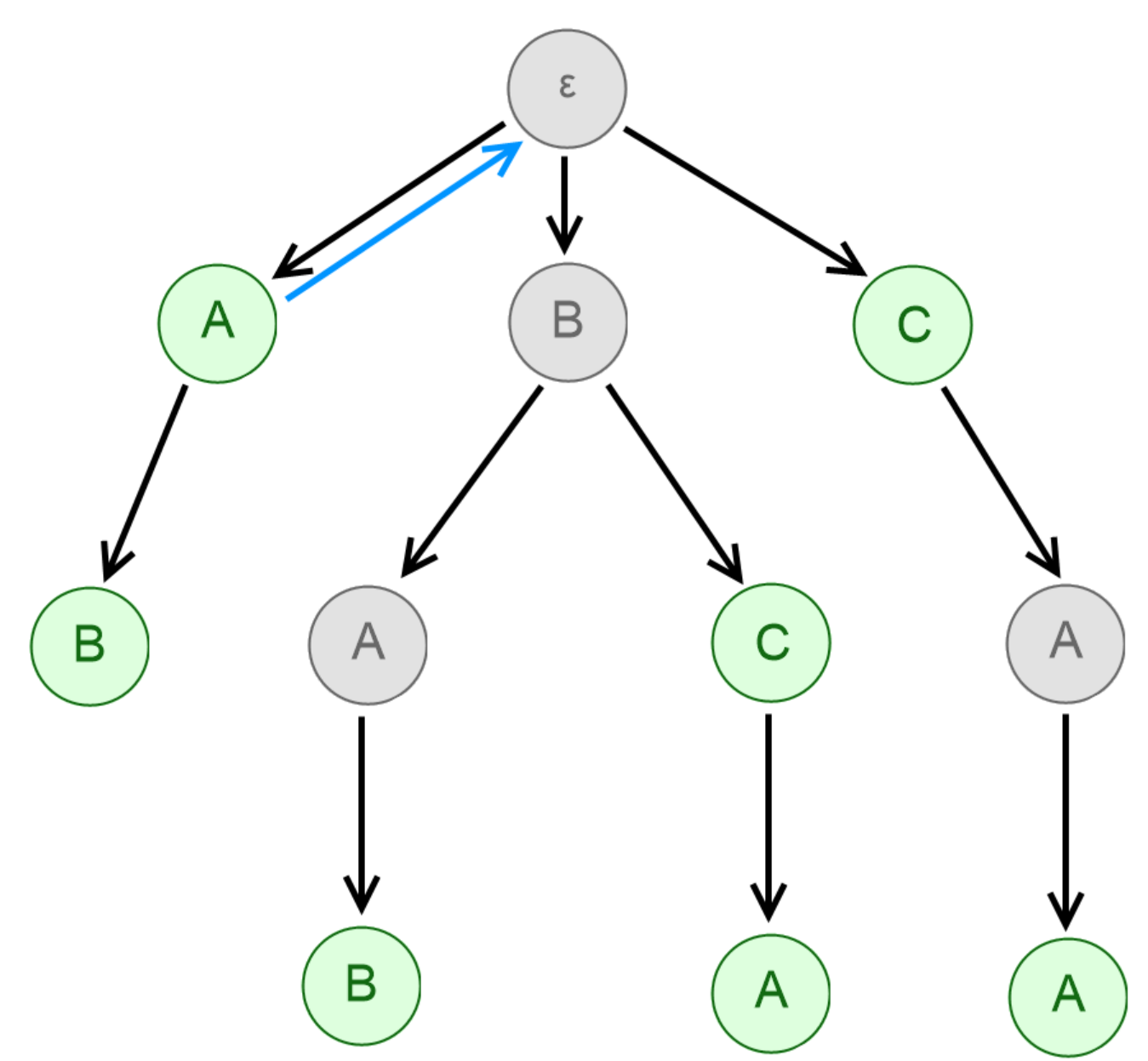
Observation: arrows point up at least one level of tree

$\Rightarrow$ draw arrows in BFS order!

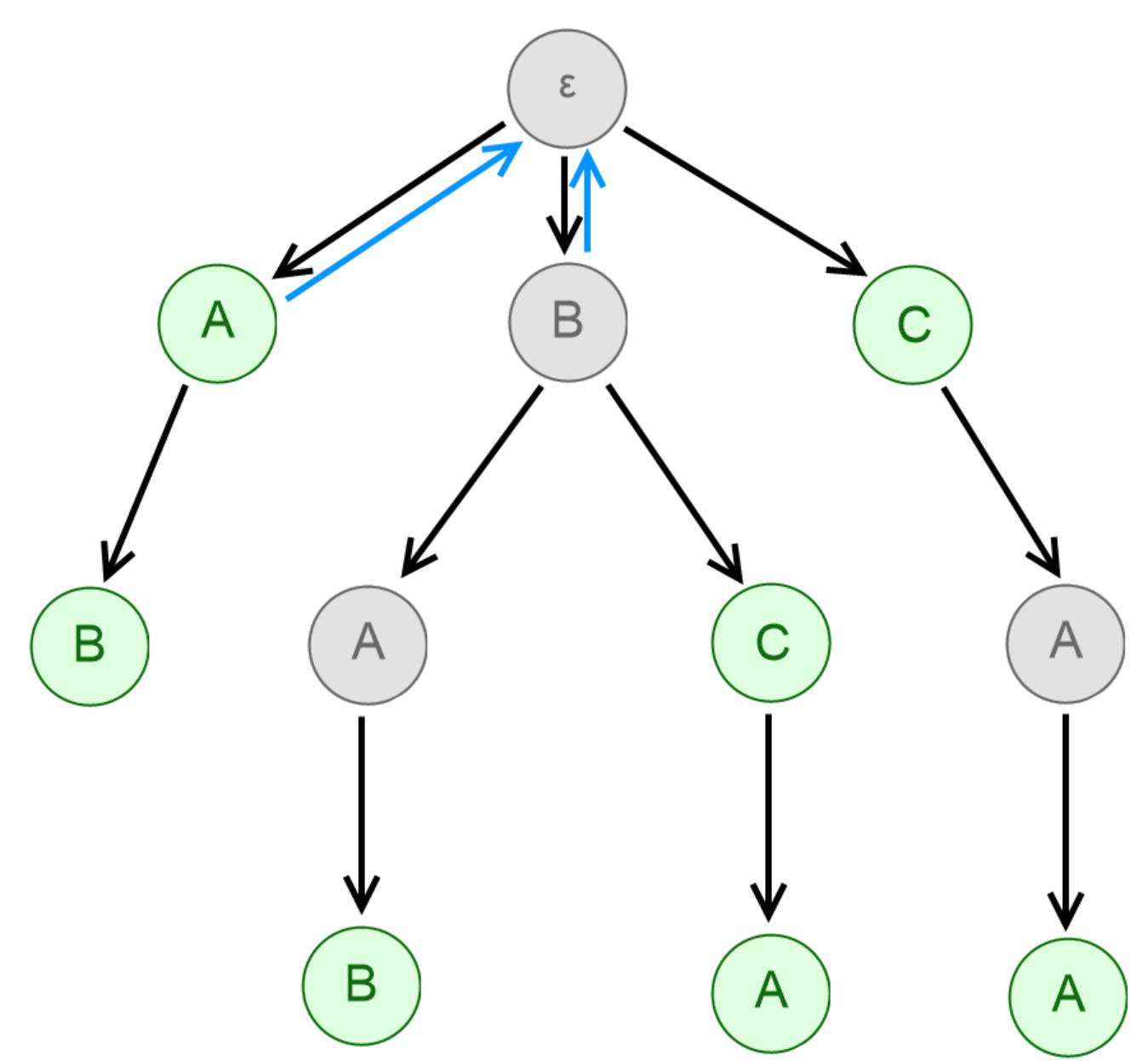
Aho Corasick: Fail Arrows



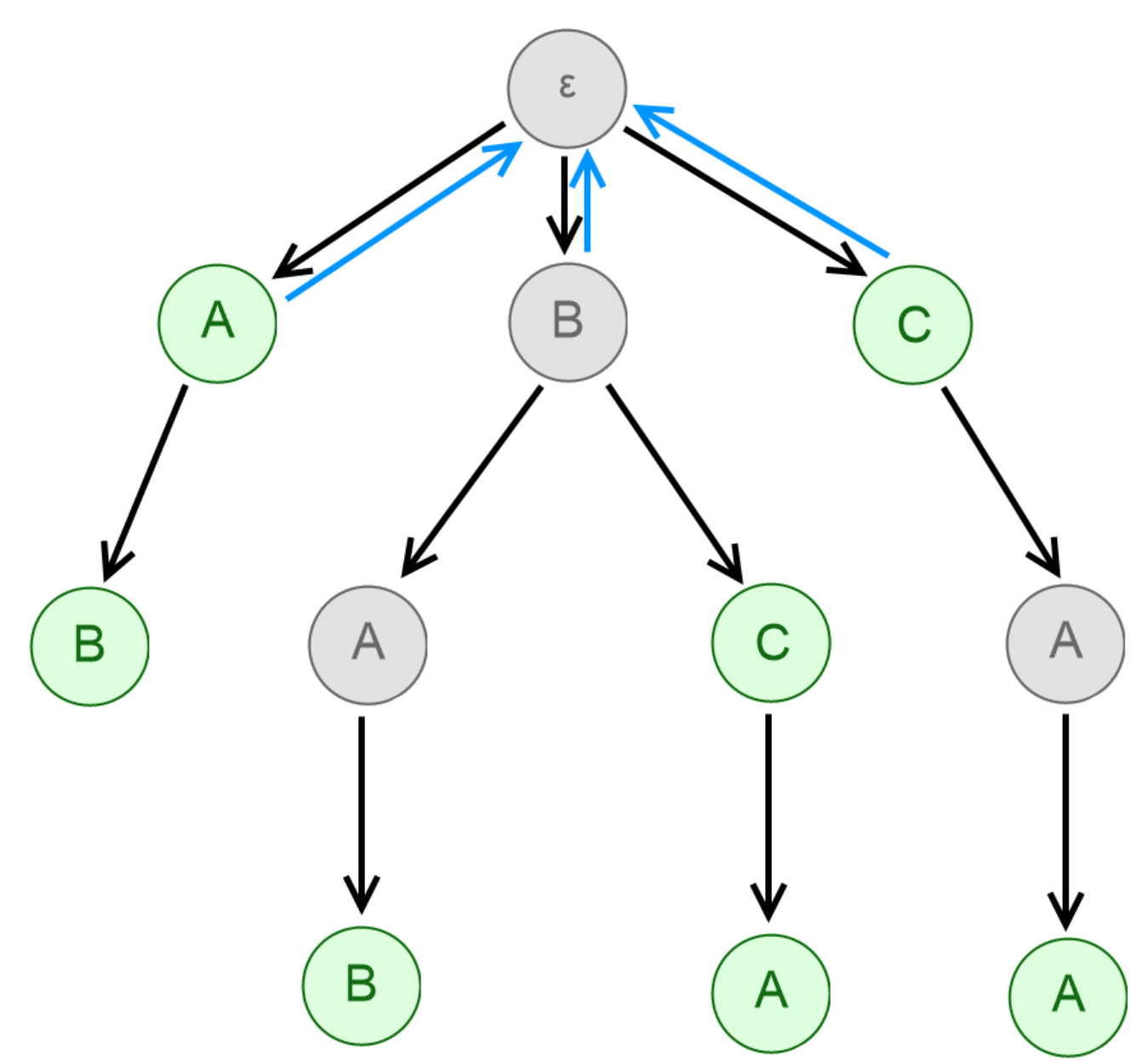
Aho Corasick: Fail Arrows



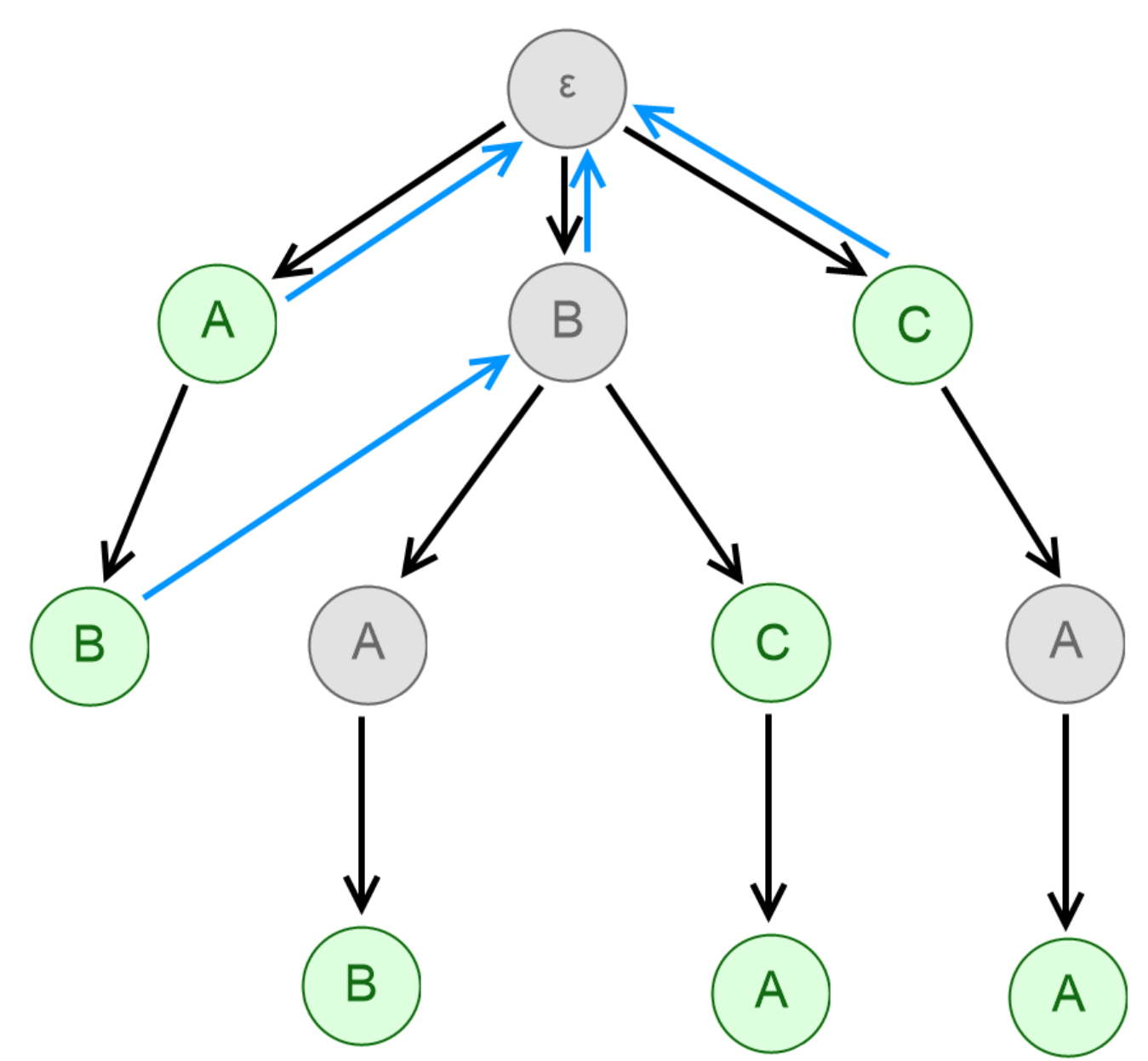
Aho Corasick: Fail Arrows



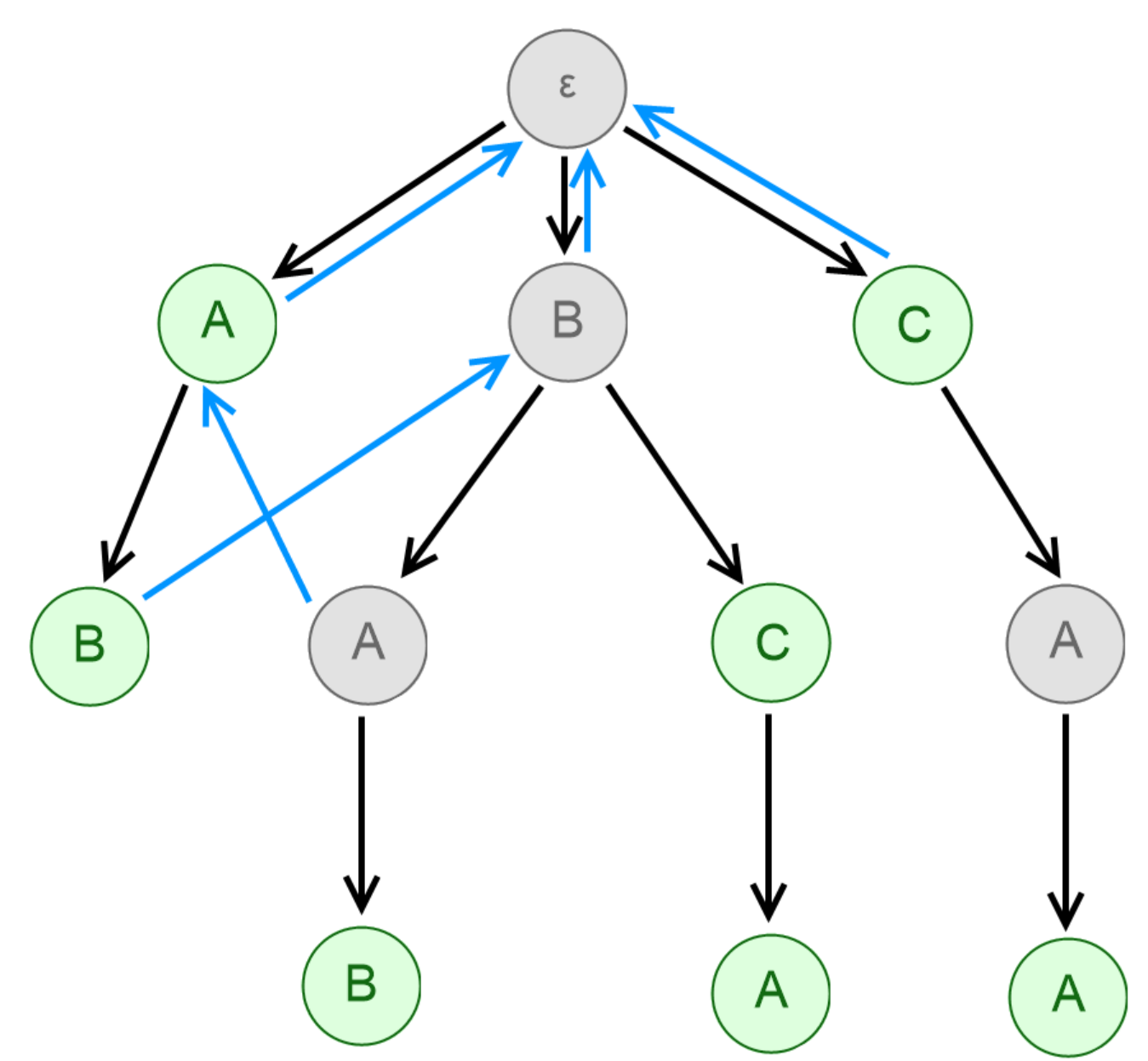
Aho Corasick: Fail Arrows



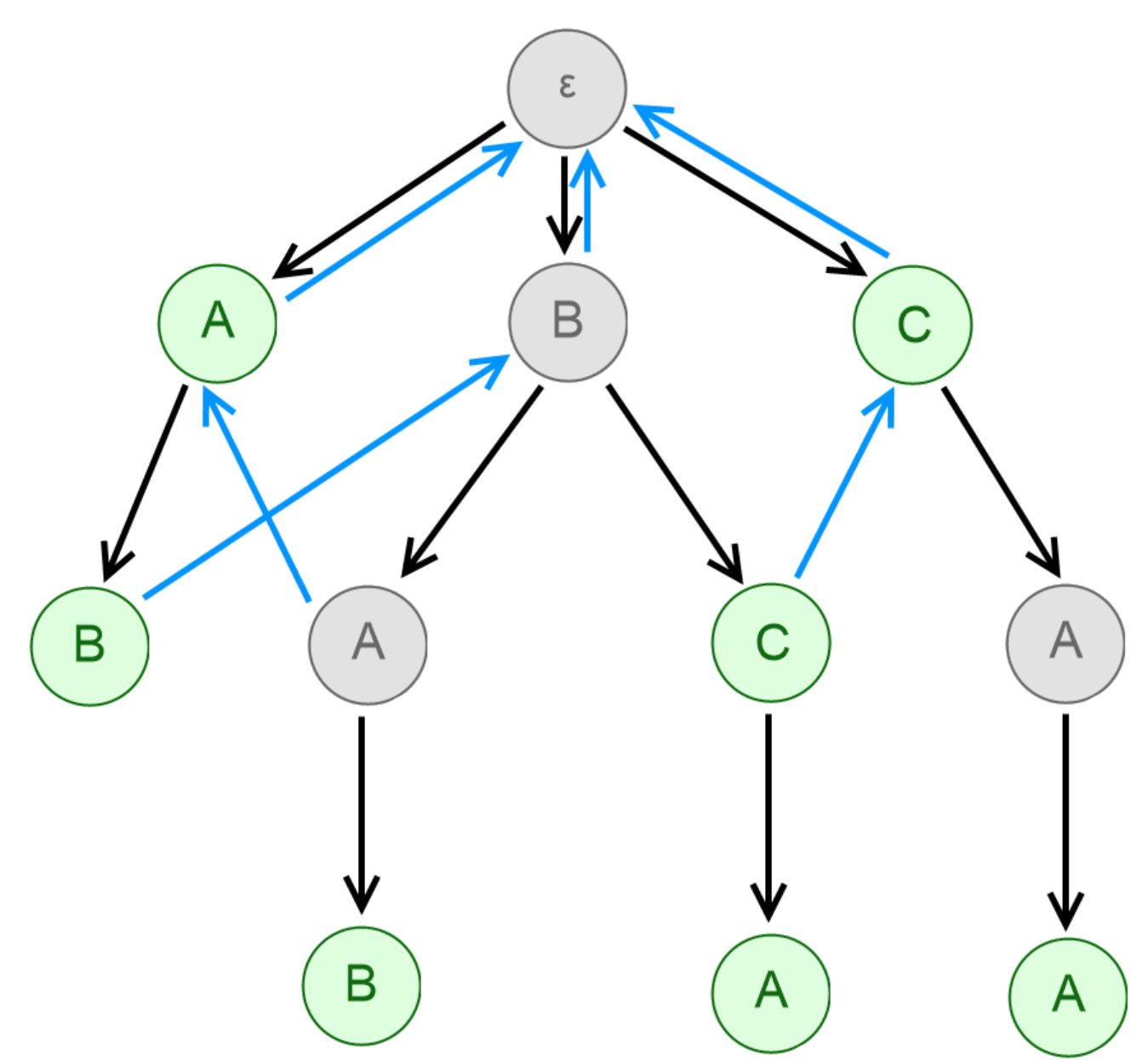
Aho Corasick: Fail Arrows



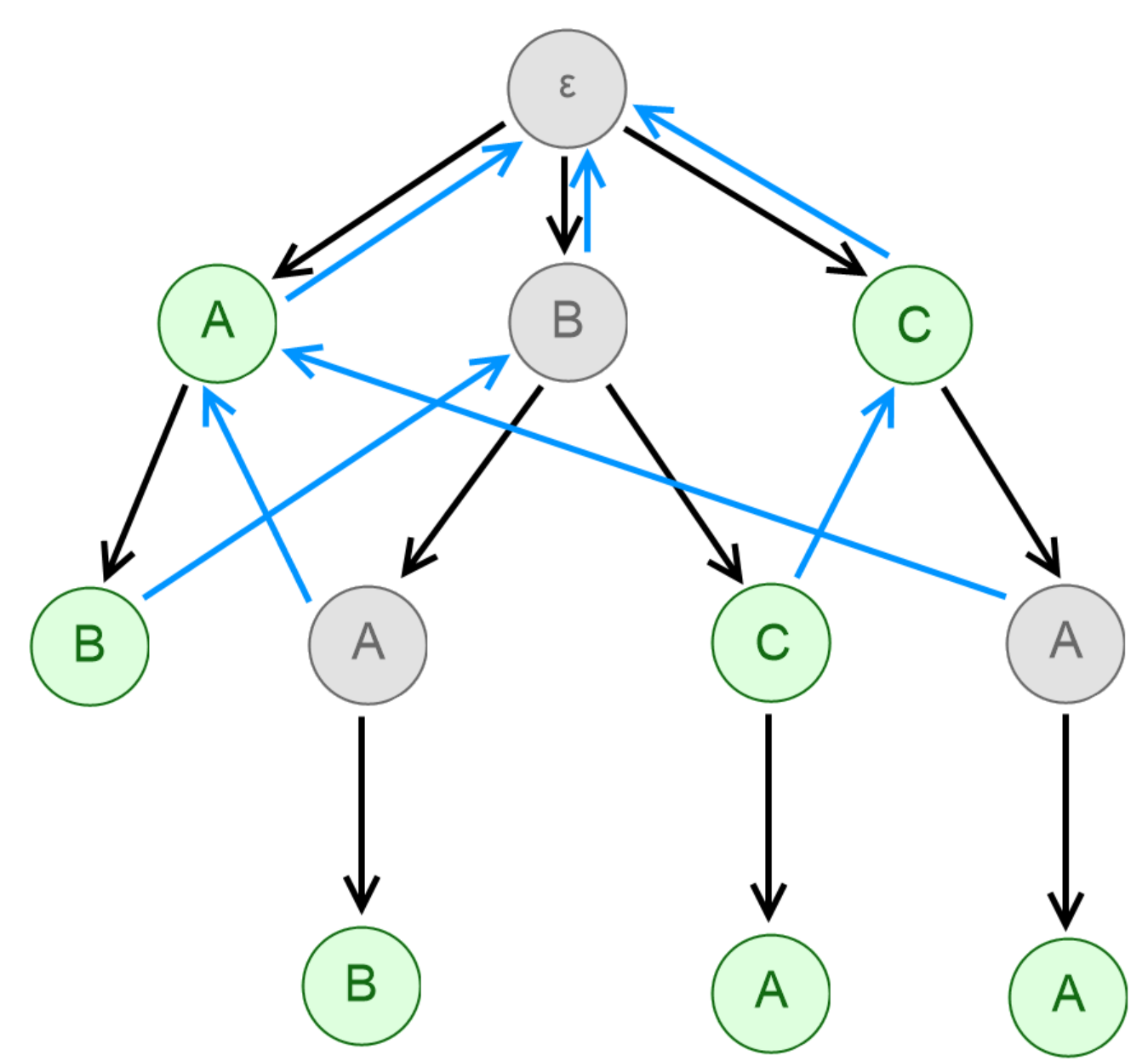
Aho Corasick: Fail Arrows



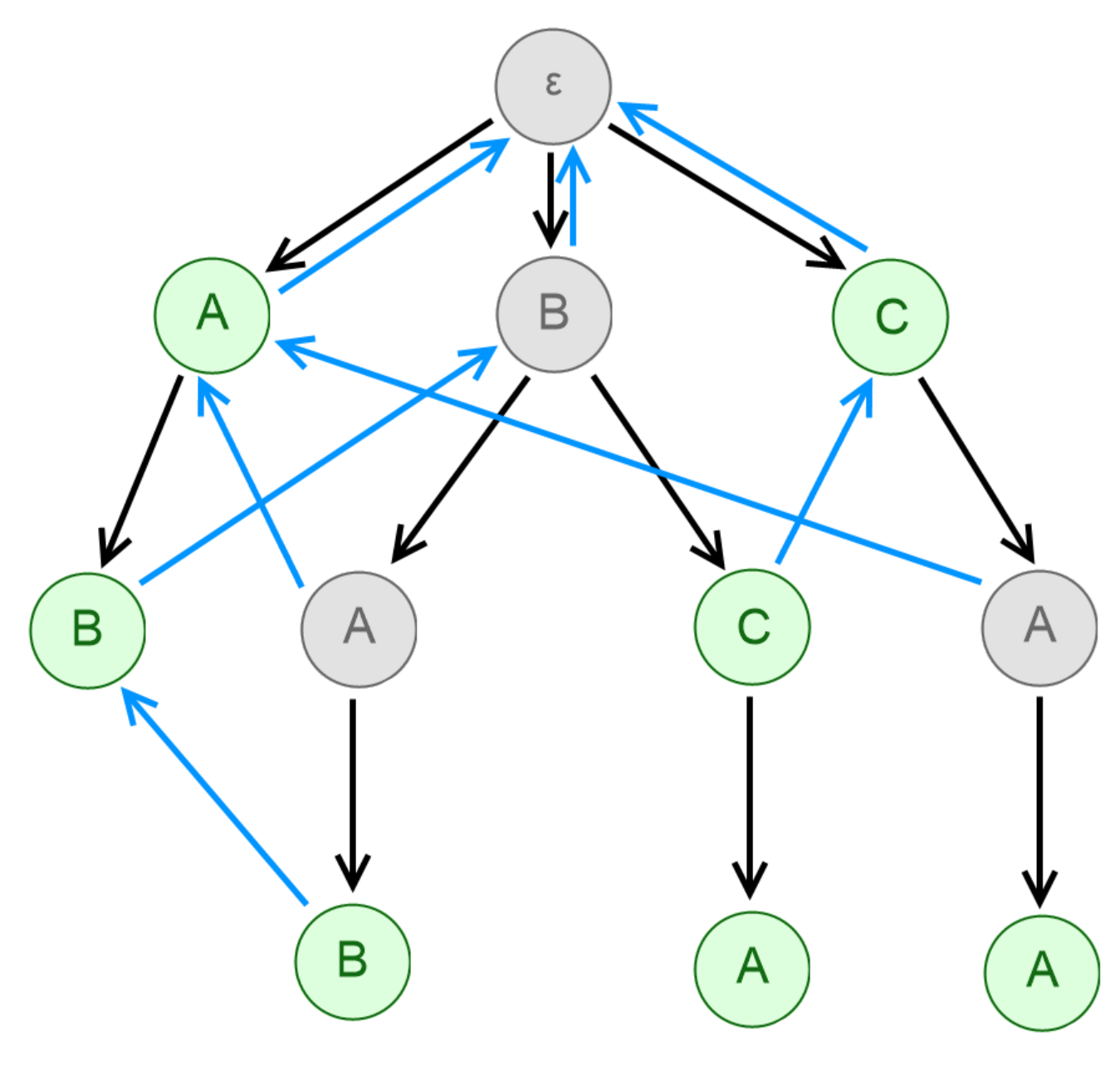
Aho Corasick: Fail Arrows



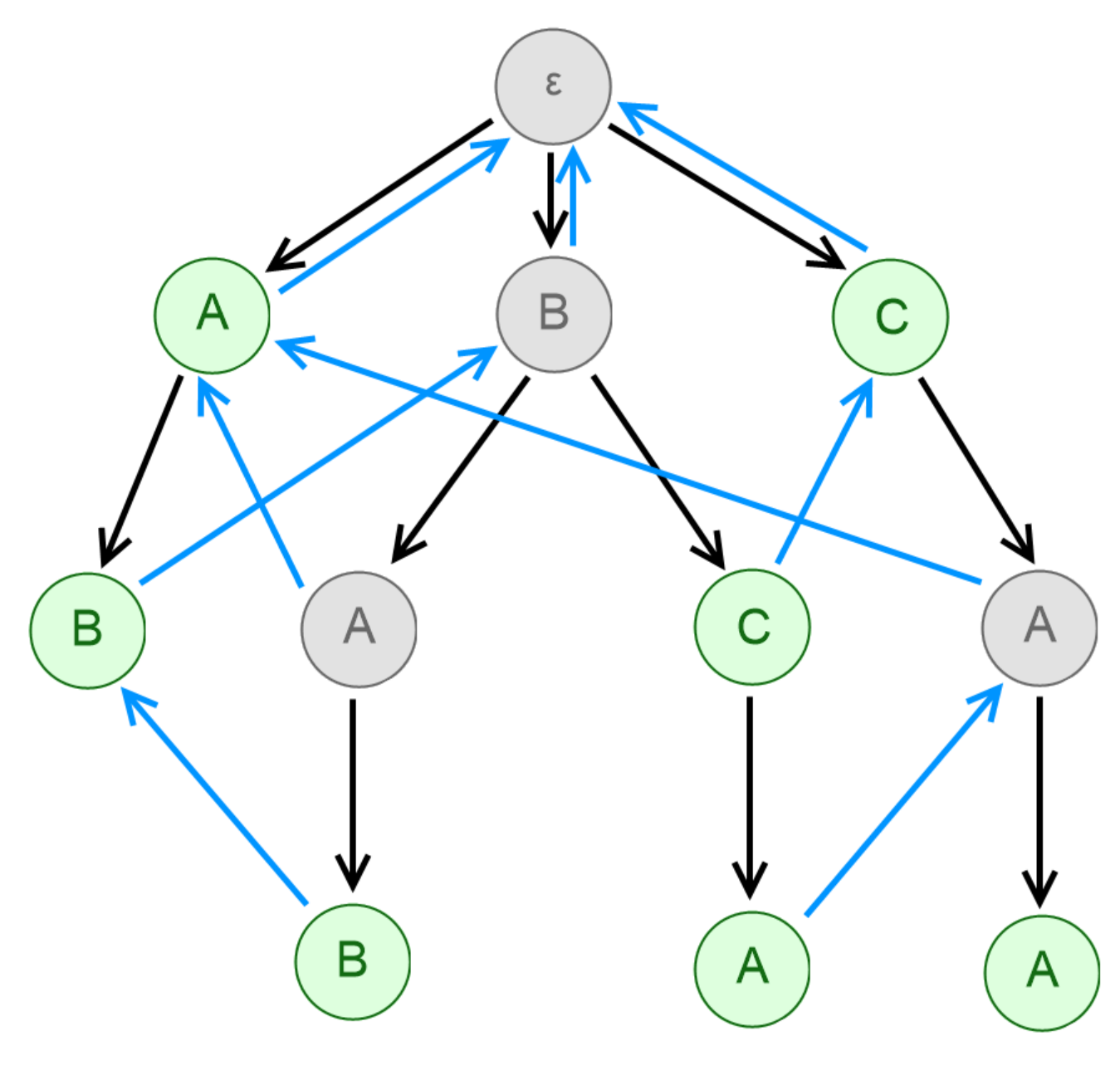
Aho Corasick: Fail Arrows



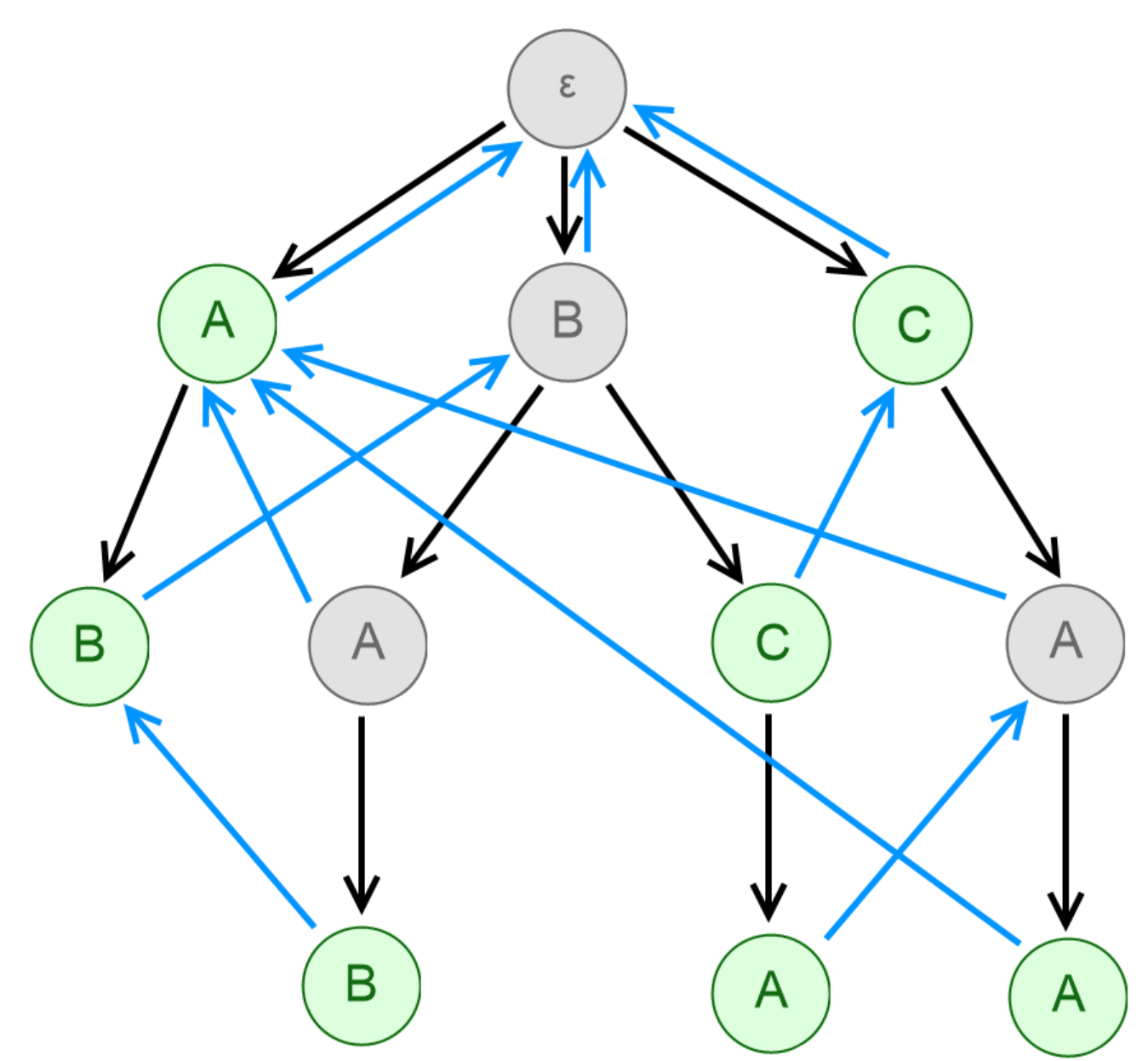
Aho Corasick: Fail Arrows



Aho Corasick: Fail Arrows



Aho Corasick: Fail Arrows



How to get matches?

Q: What are the possible “total matches” at a trie node?

How to get matches?

Q: What are the possible “total matches” at a trie node?

A: All the strings in the trie that match the suffix!

How to get matches?

Q: What are the possible “total matches” at a trie node?

A: All the strings in the trie that match the suffix!

Total matches are of course prefixes, so if we keep following Fail arrows we will eventually get all of them, but...

Problem: following all Fail arrows on every step is too slow

How to get matches?

Q: What are the possible “total matches” at a trie node?

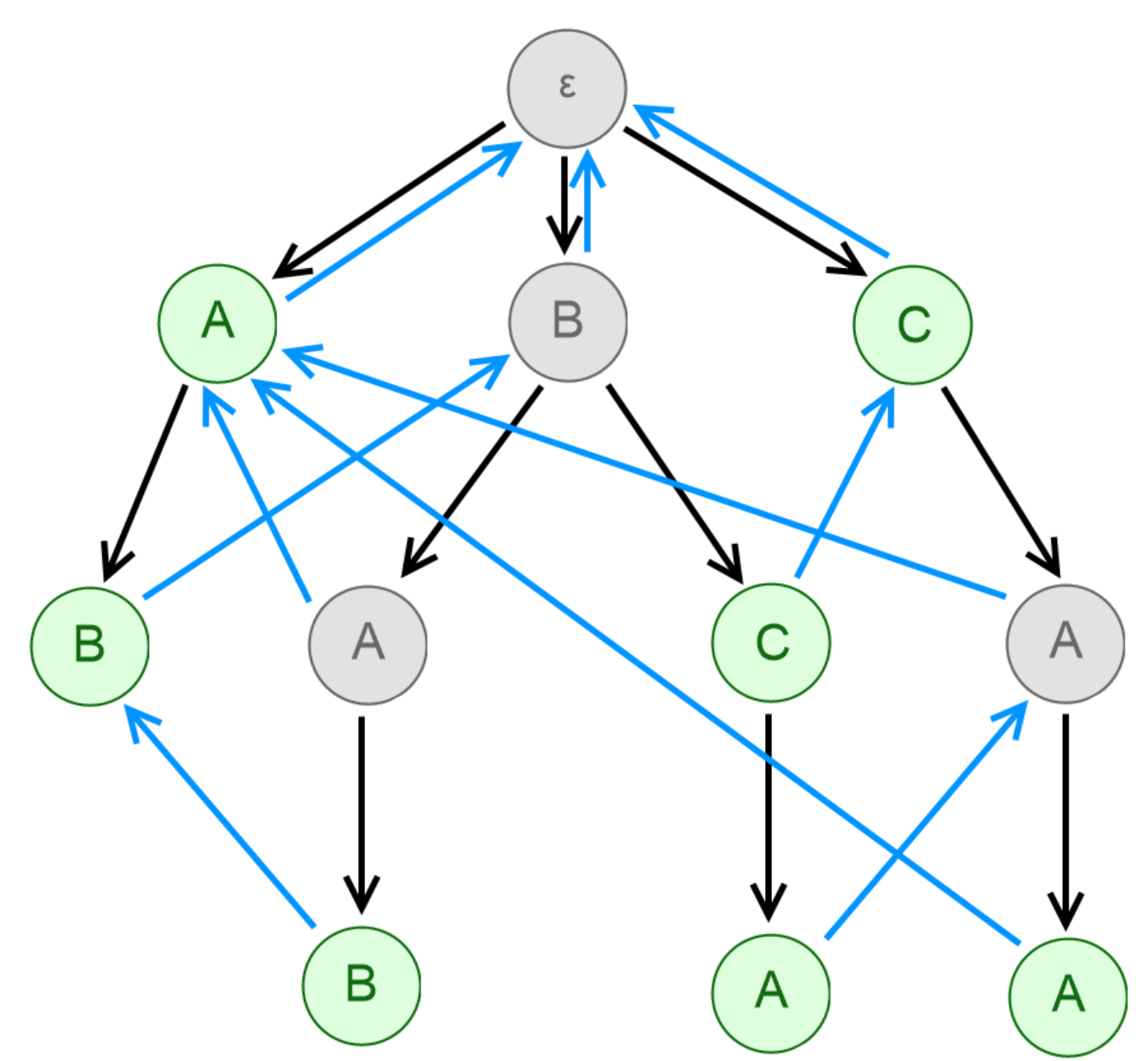
A: All the strings in the trie that match the suffix!

Total matches are of course prefixes, so if we keep following Fail arrows we will eventually get all of them, but...

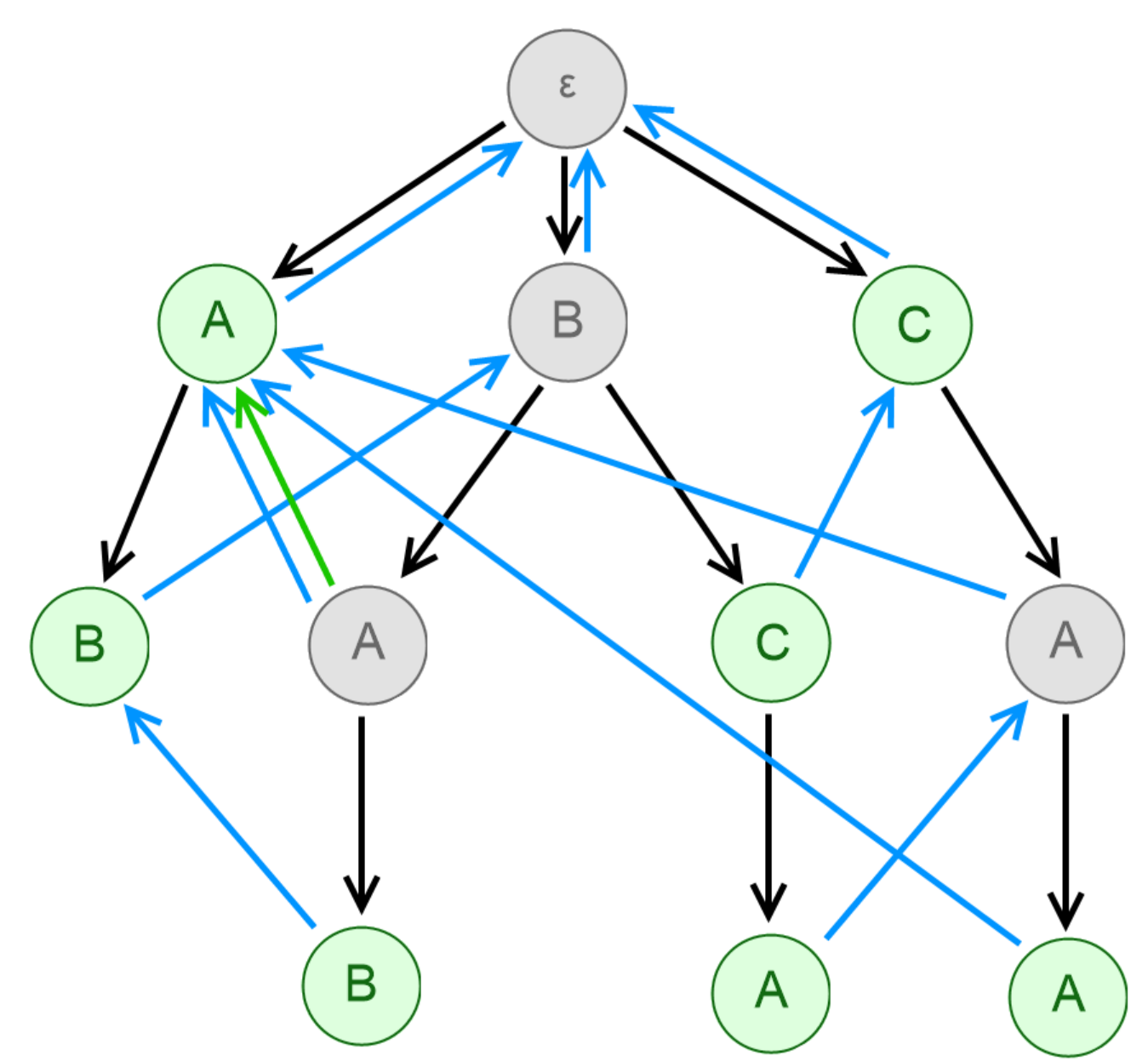
Problem: following all Fail arrows on every step is too slow
Solution: pre-compute this before hand $\Rightarrow$ Dictionary arrows!

Notice that we only need to follow Fail arrows until we either see a green node, or a node already with a Dictionary arrow, so at most 2 arrows.

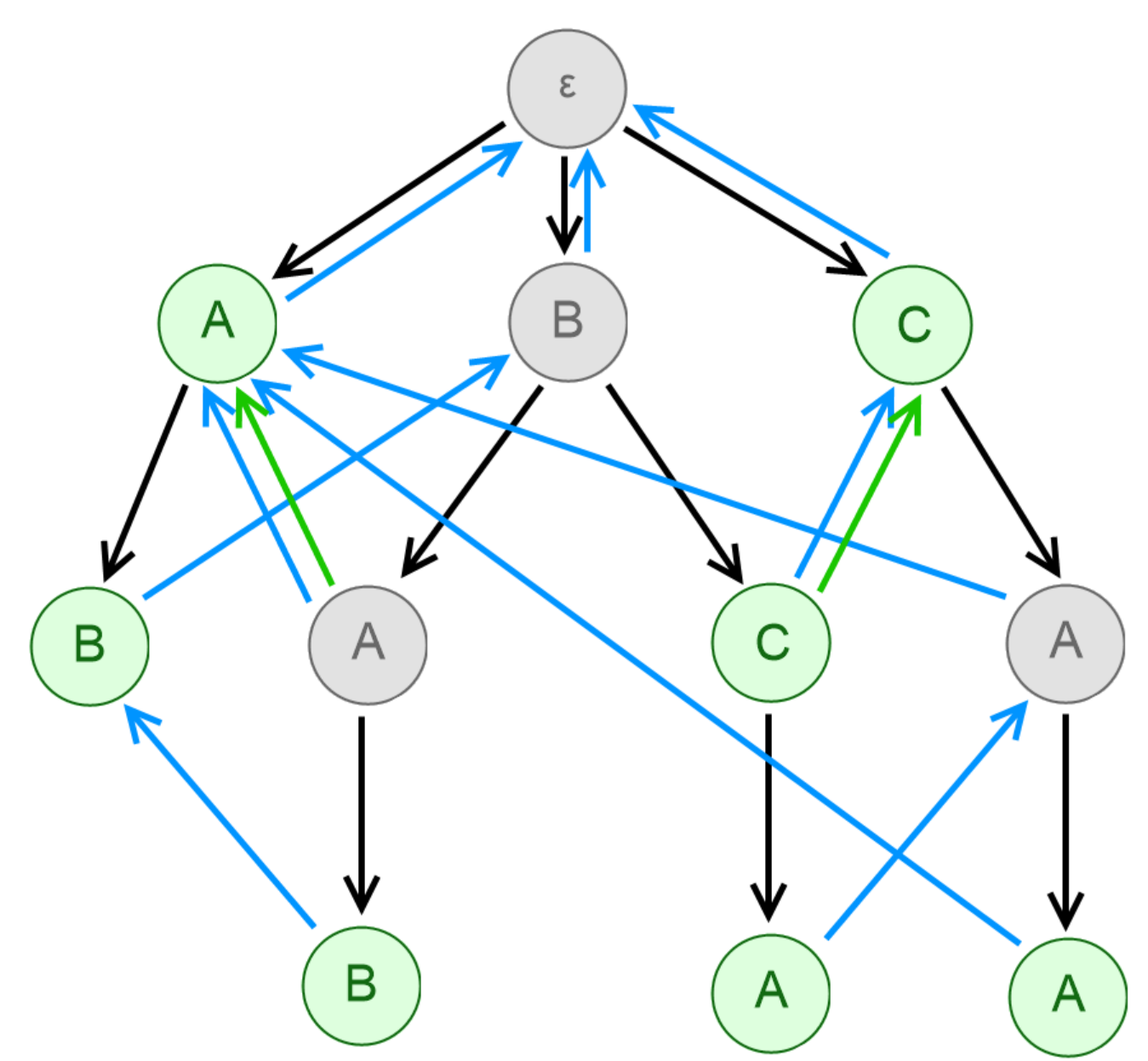
Aho Corasick: Dictionary Arrows



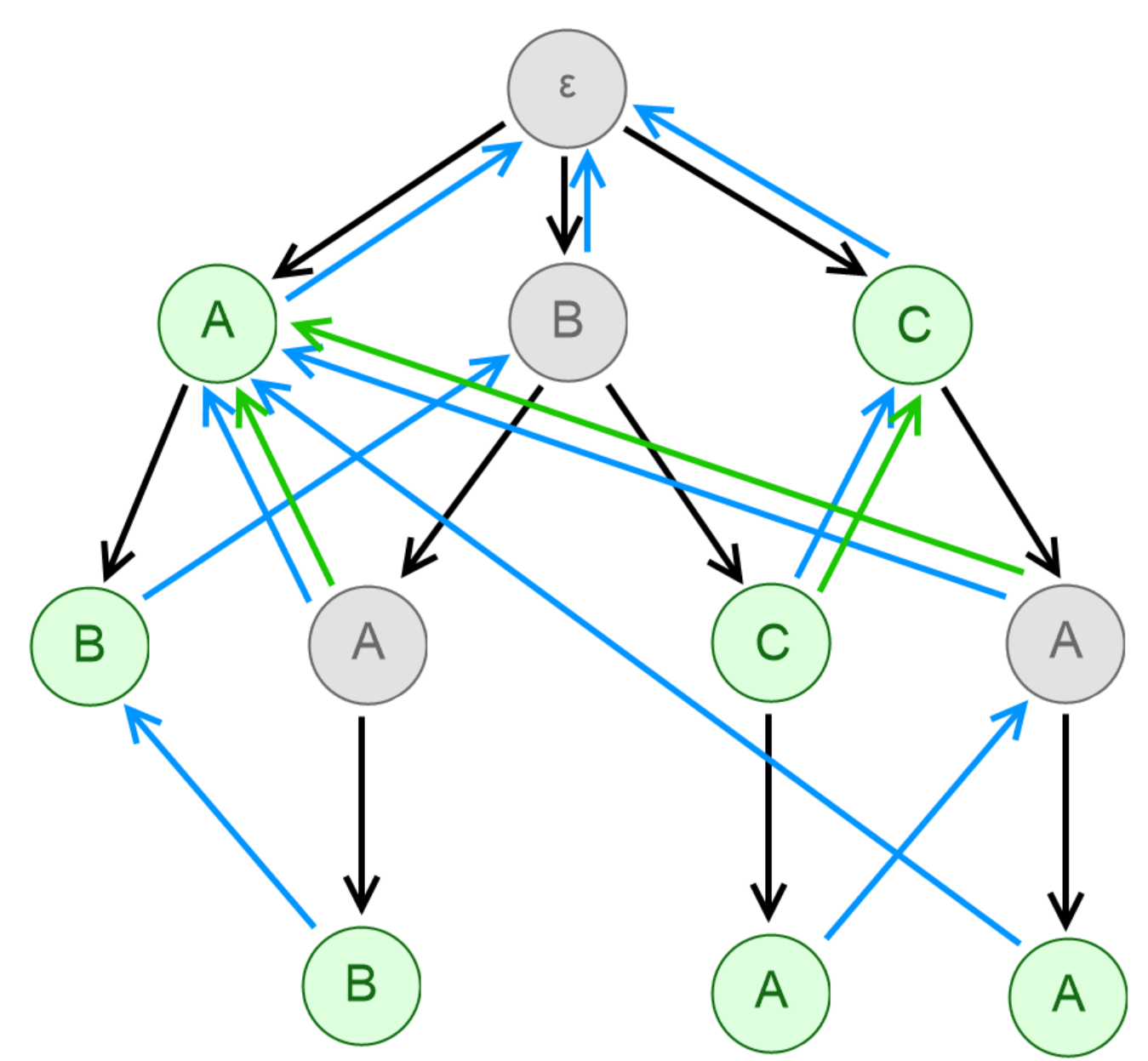
Aho Corasick: Dictionary Arrows



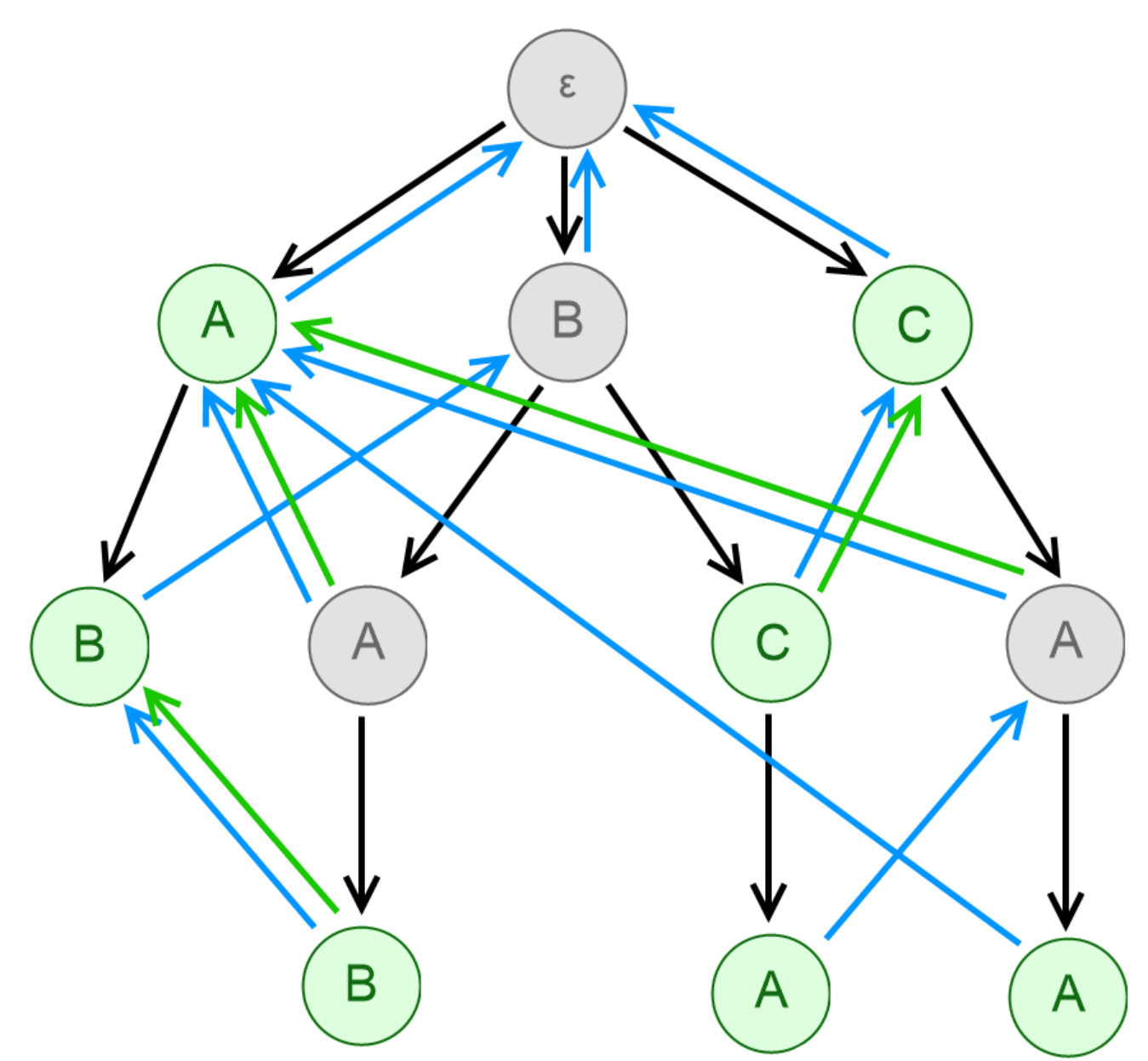
Aho Corasick: Dictionary Arrows



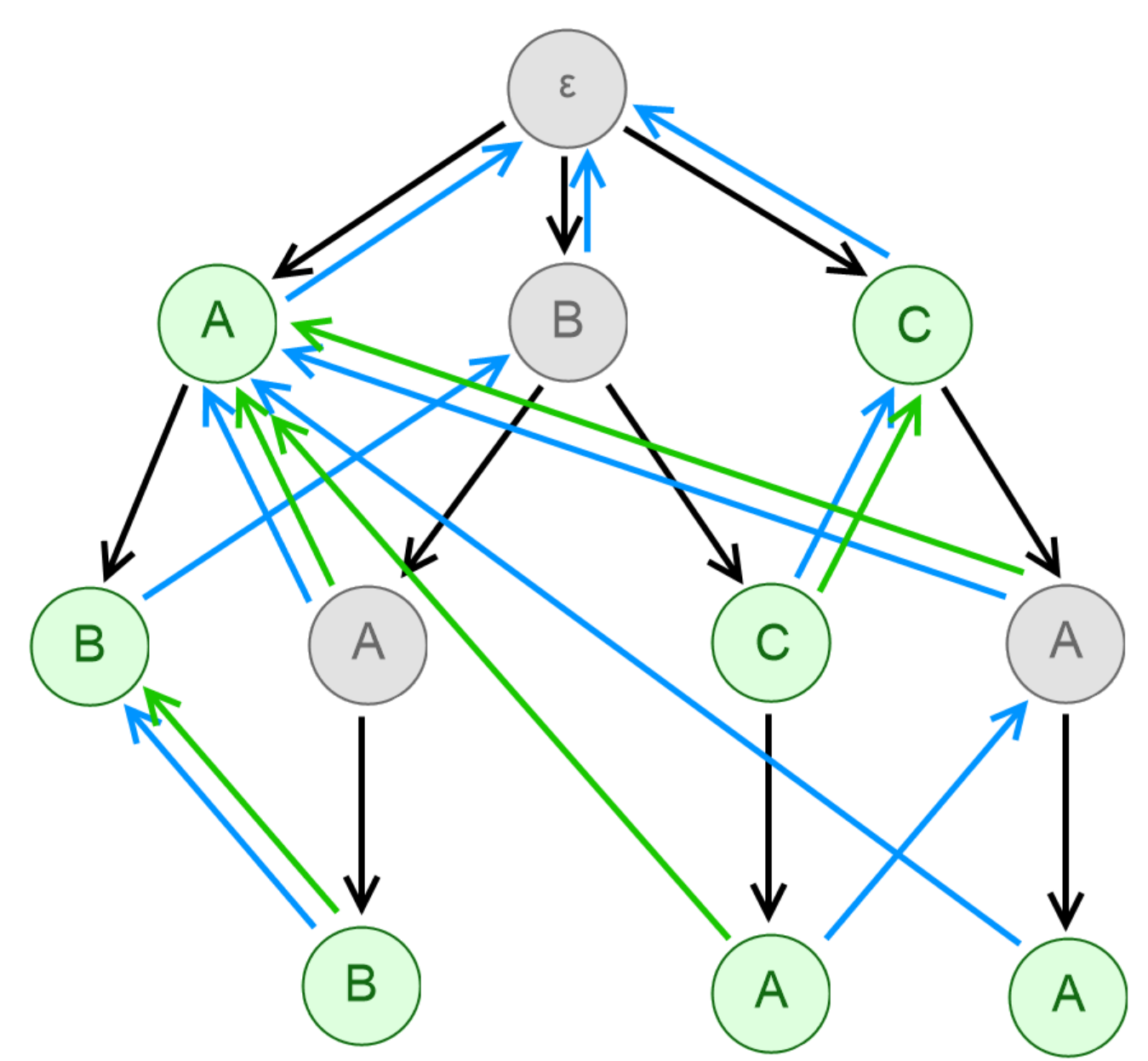
Aho Corasick: Dictionary Arrows



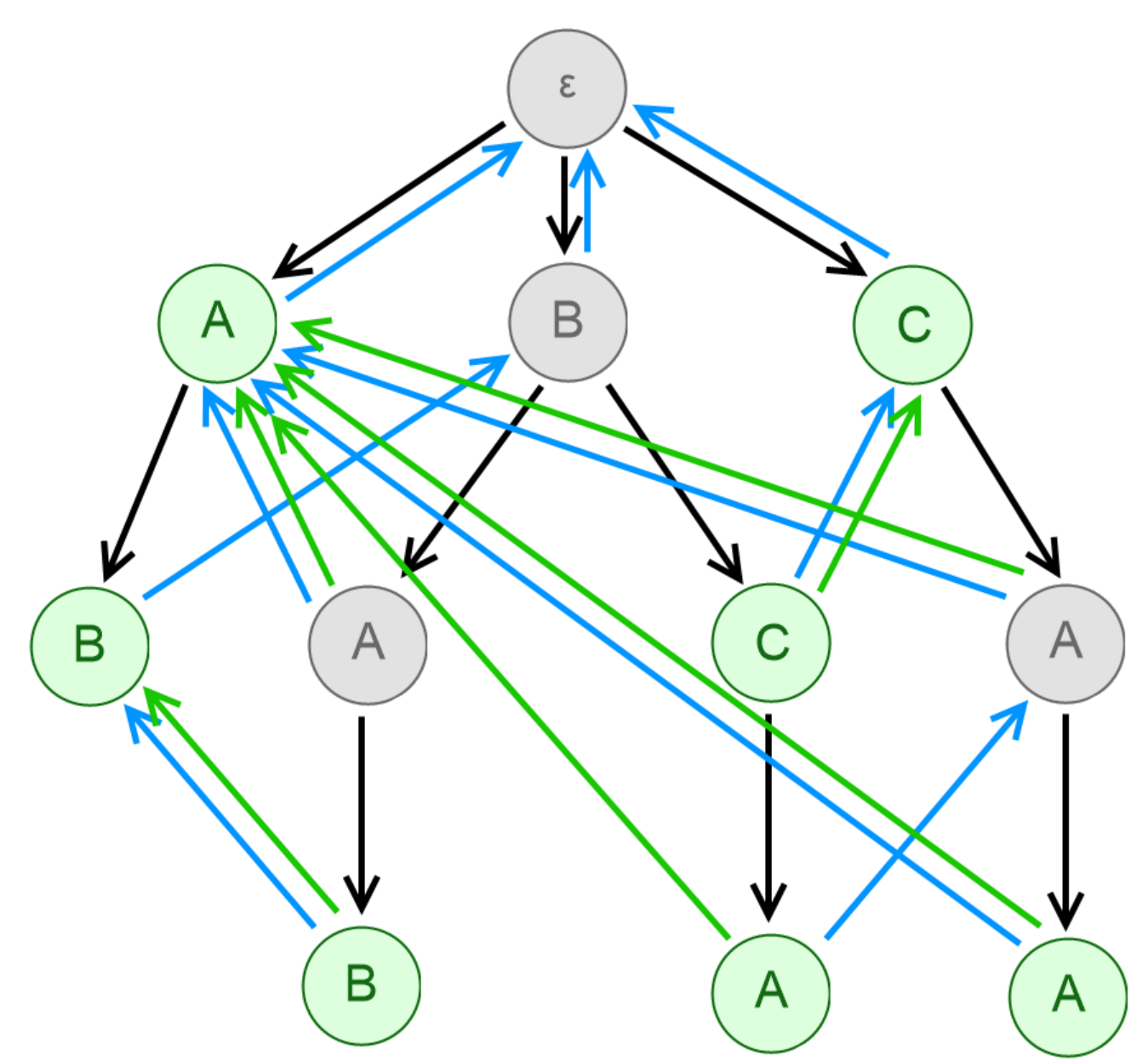
Aho Corasick: Dictionary Arrows



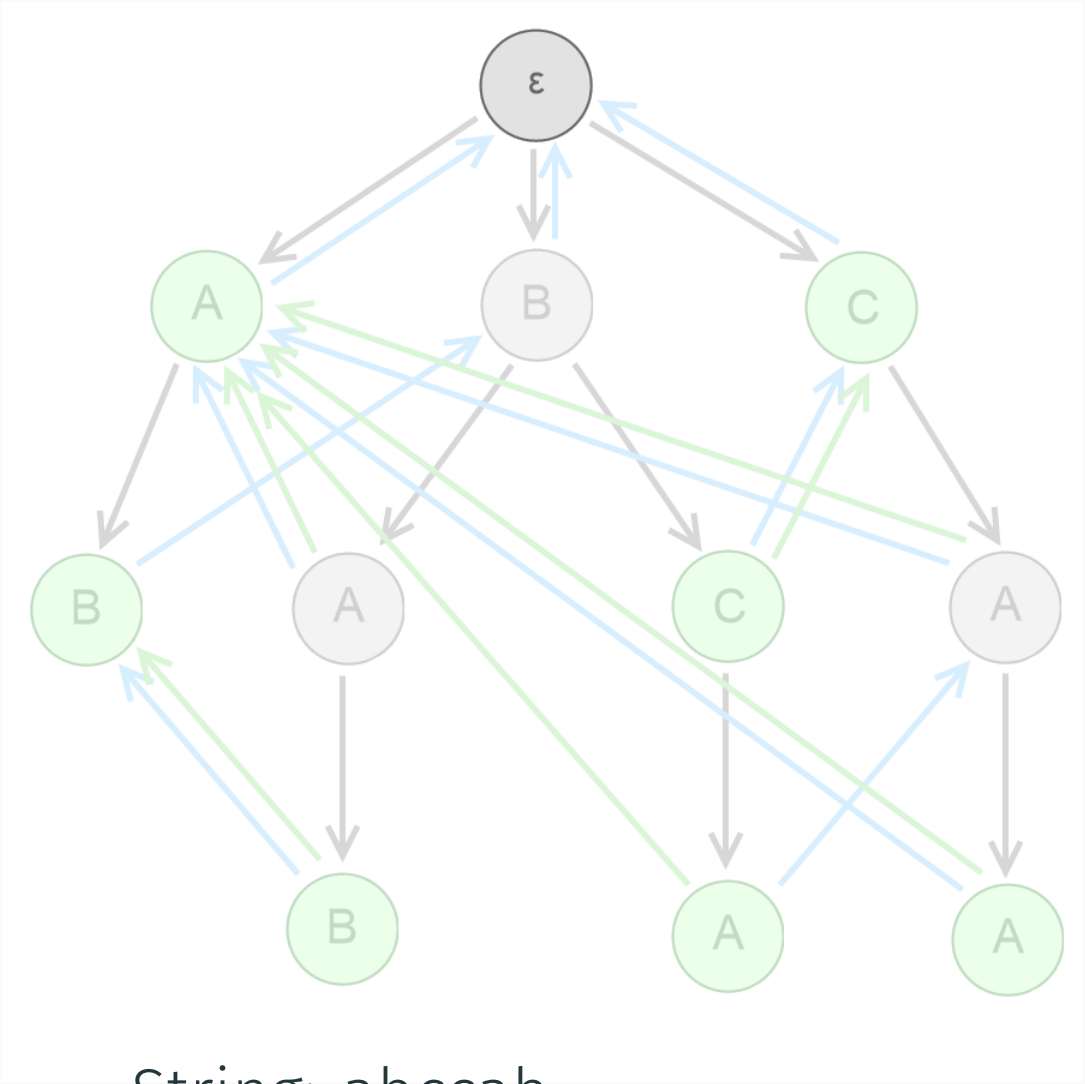
Aho Corasick: Dictionary Arrows



Aho Corasick: Dictionary Arrows



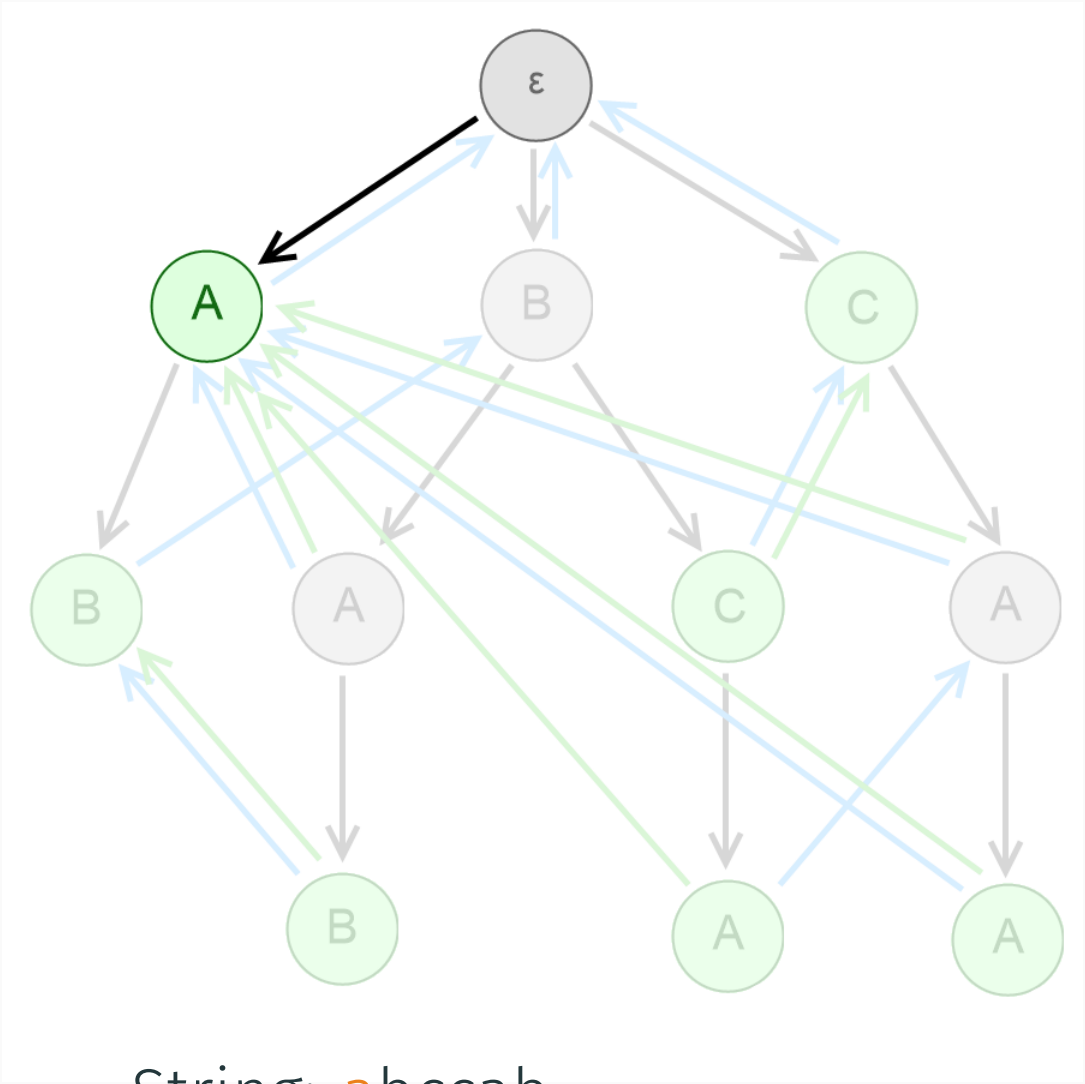
Aho Corasick: Matching Example



String: abccab

Output:

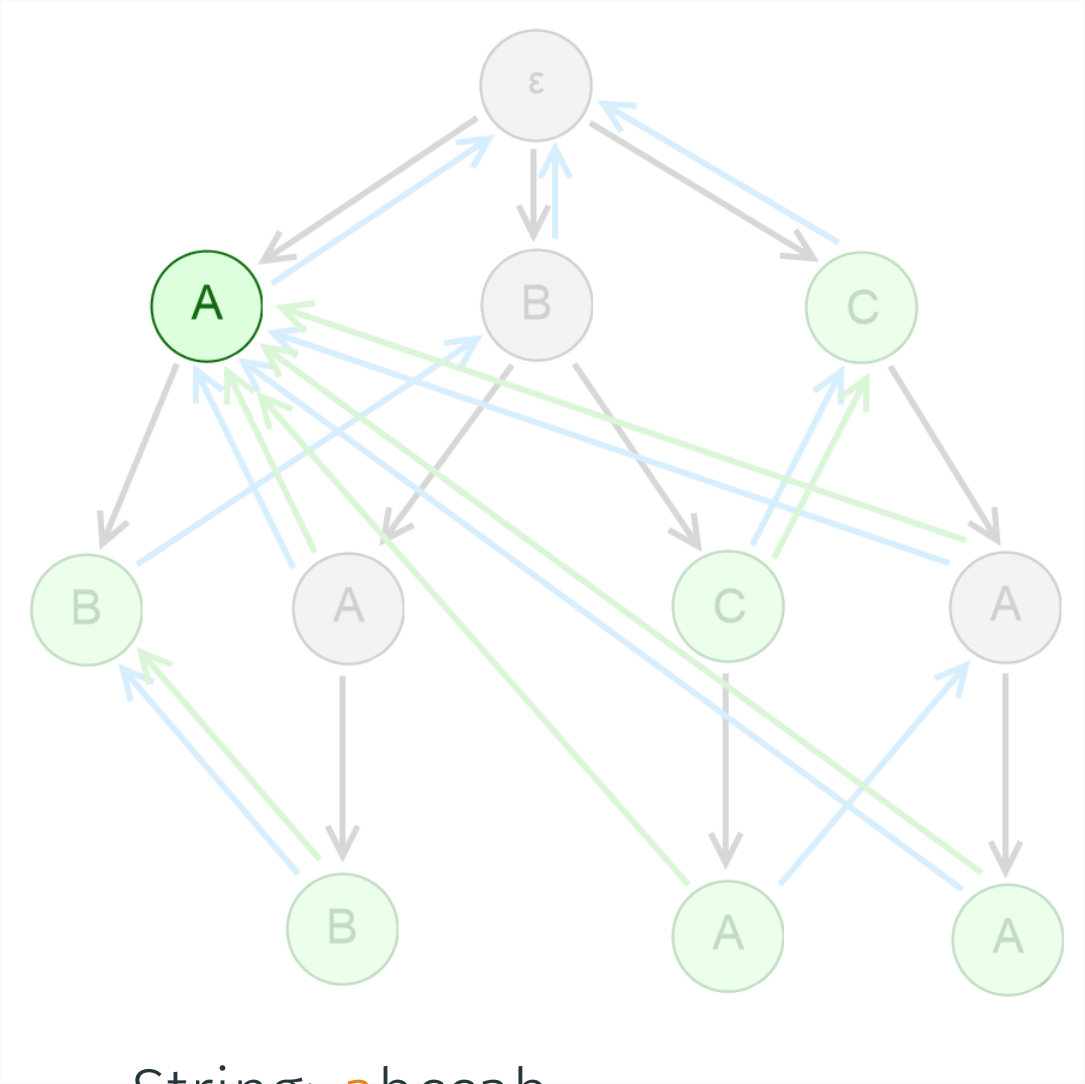
Aho Corasick: Matching Example



String: abccab

Output:

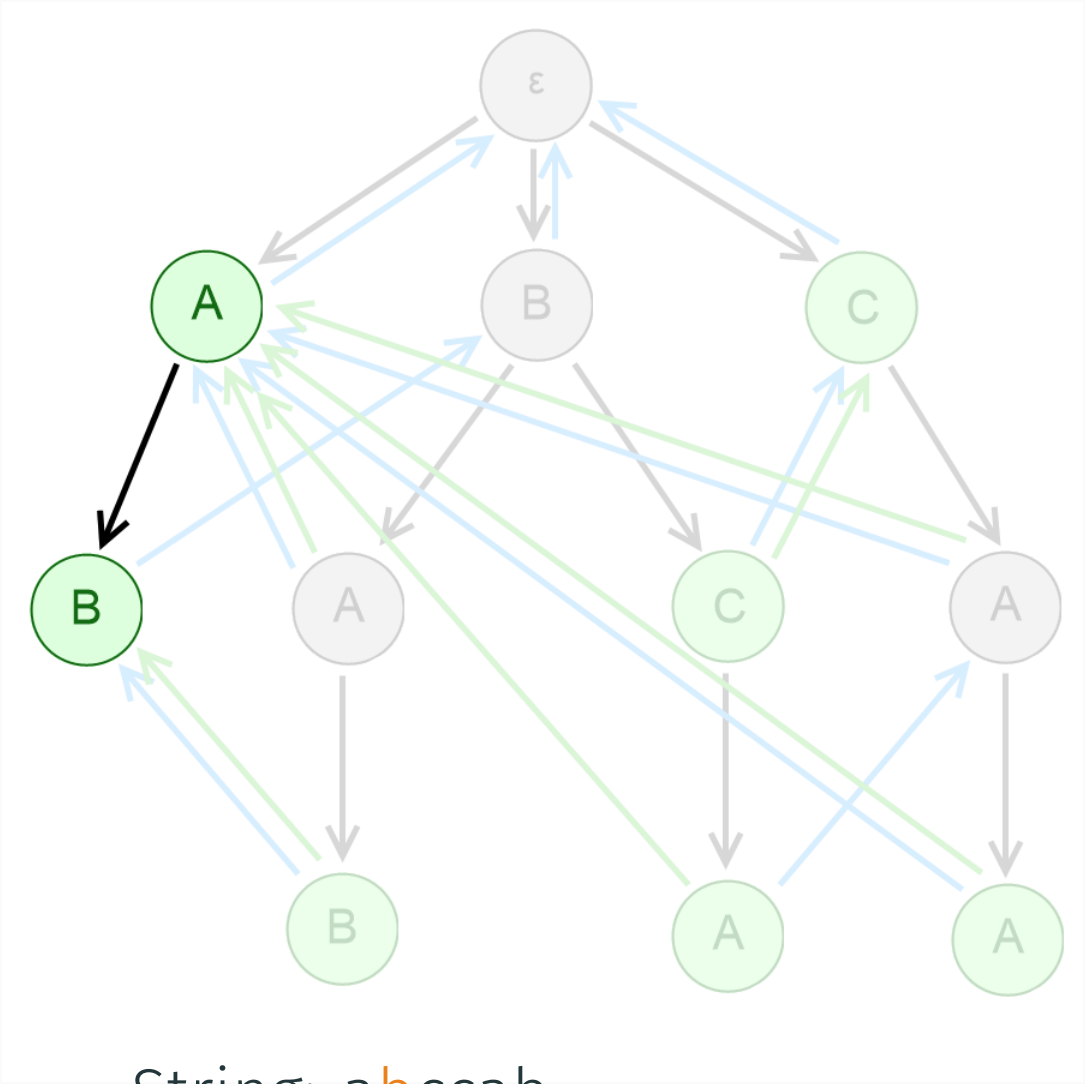
Aho Corasick: Matching Example



String: abccab

Output: a

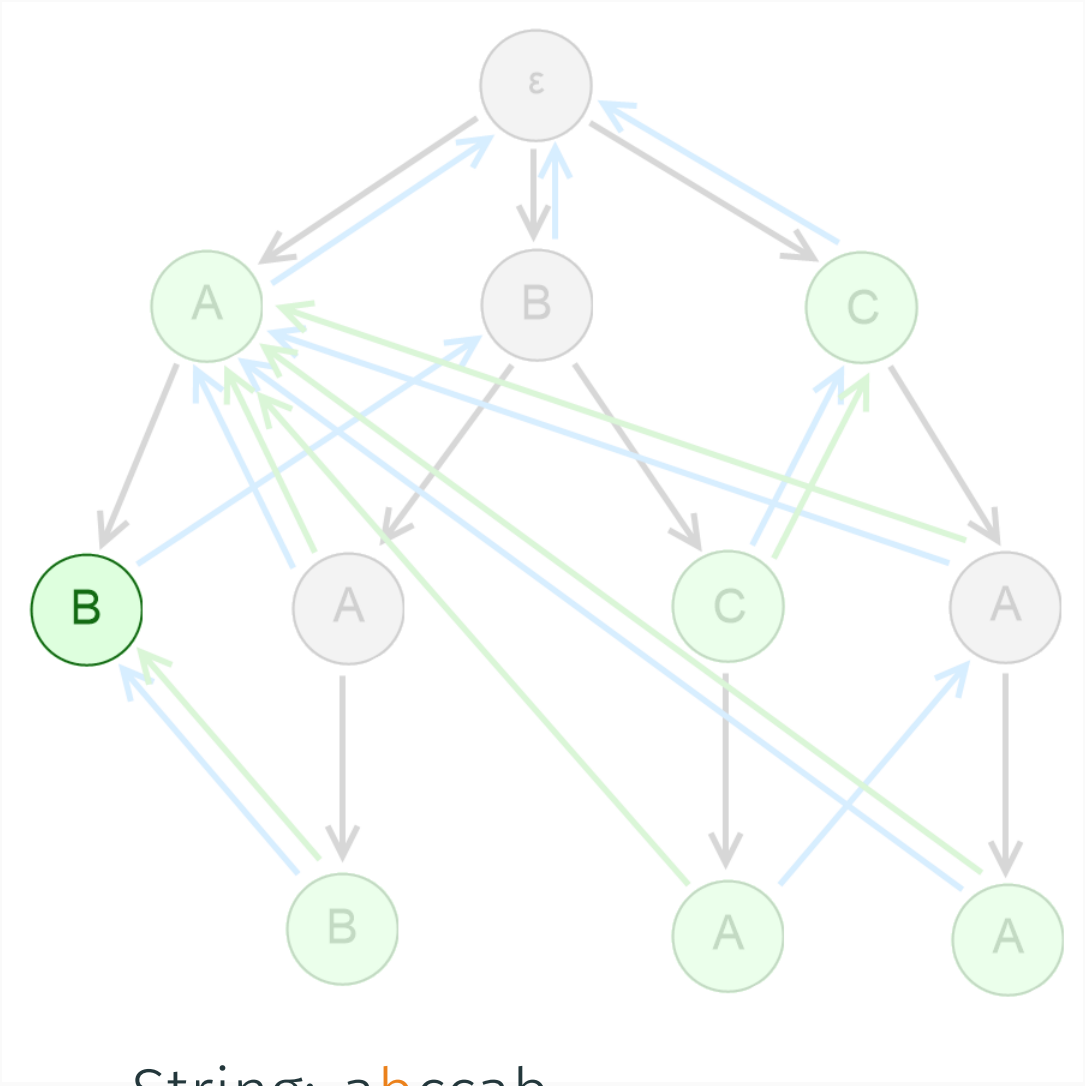
Aho Corasick: Matching Example



String: abccab

Output: a

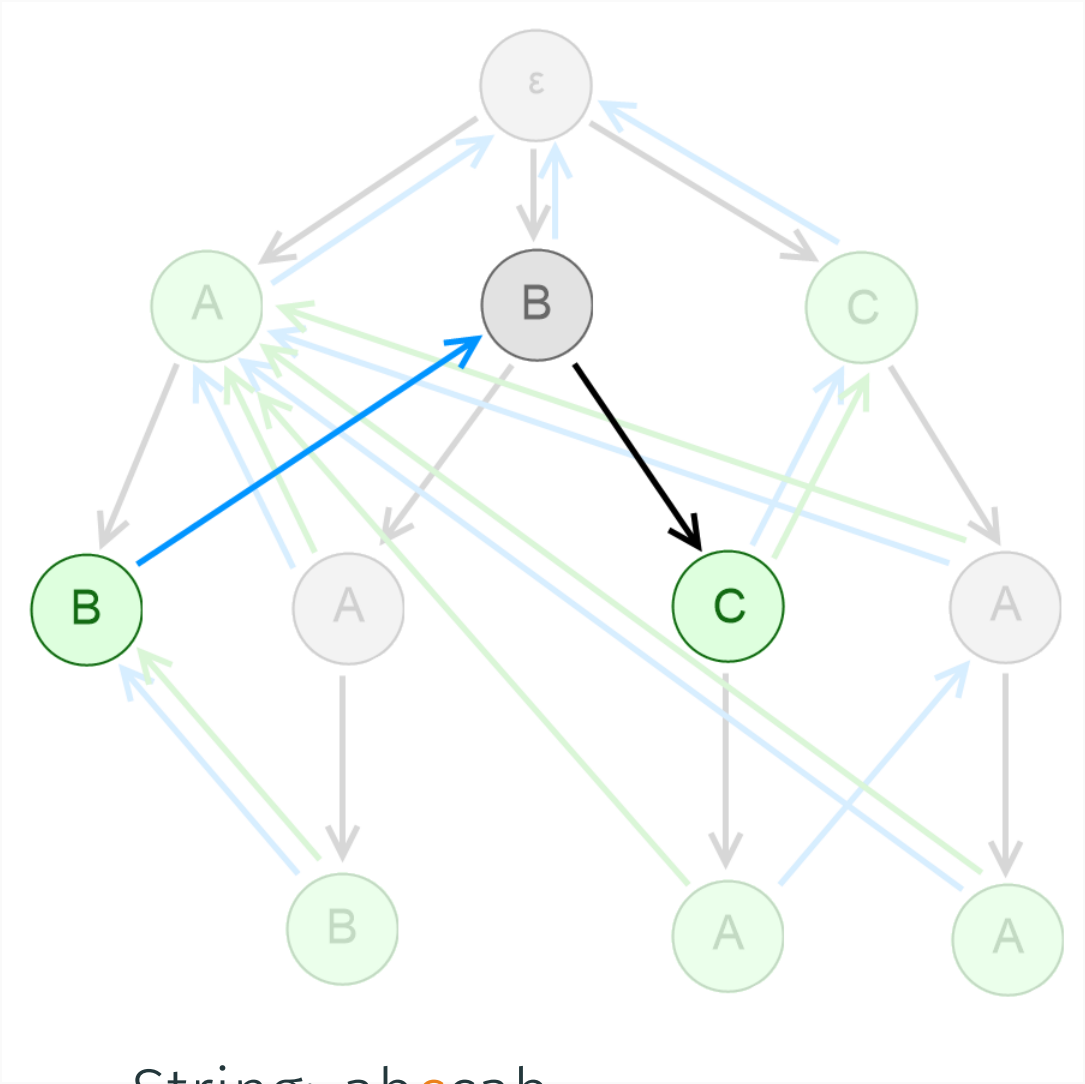
Aho Corasick: Matching Example



String: abccab

Output: a, ab

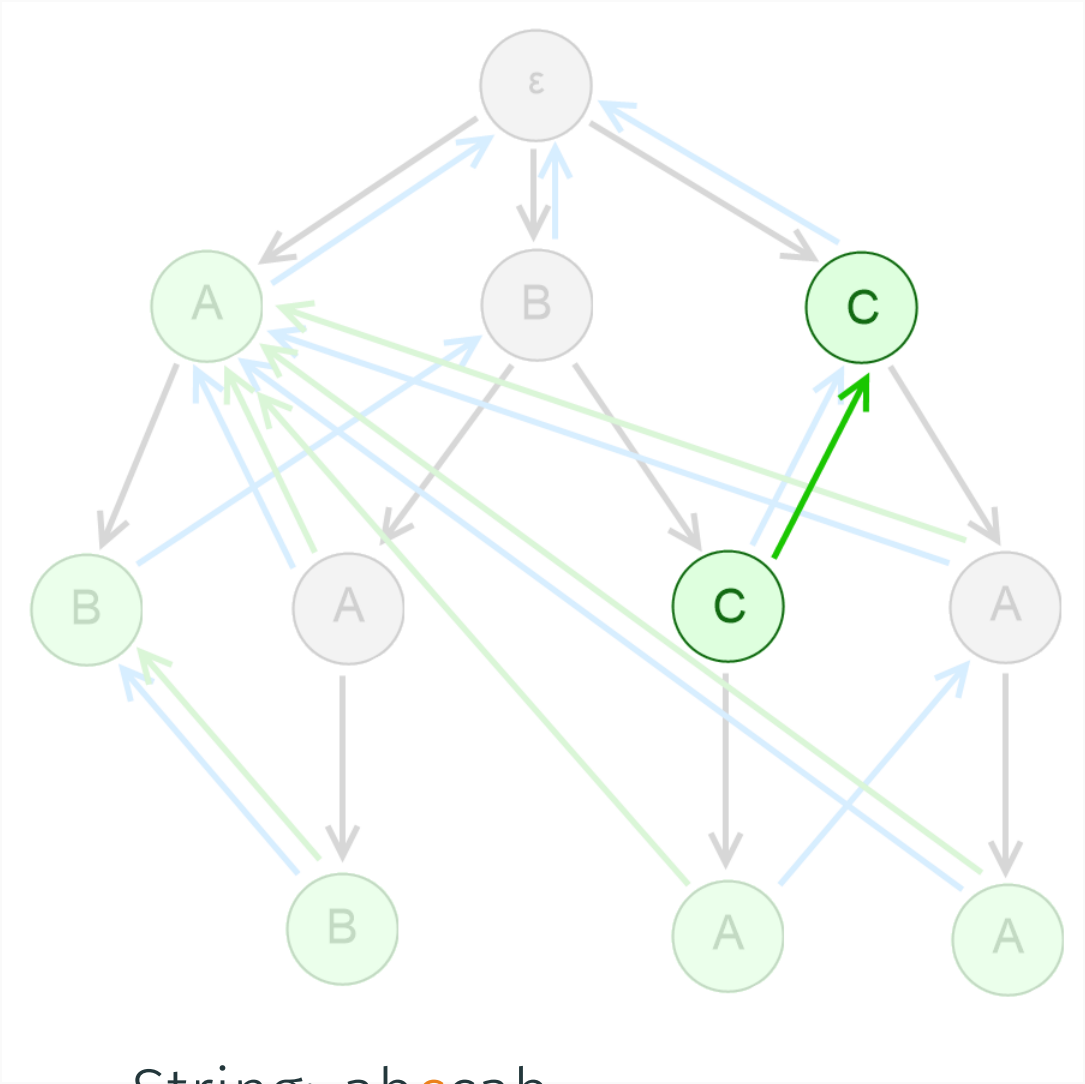
Aho Corasick: Matching Example



String: abccab

Output: a, ab

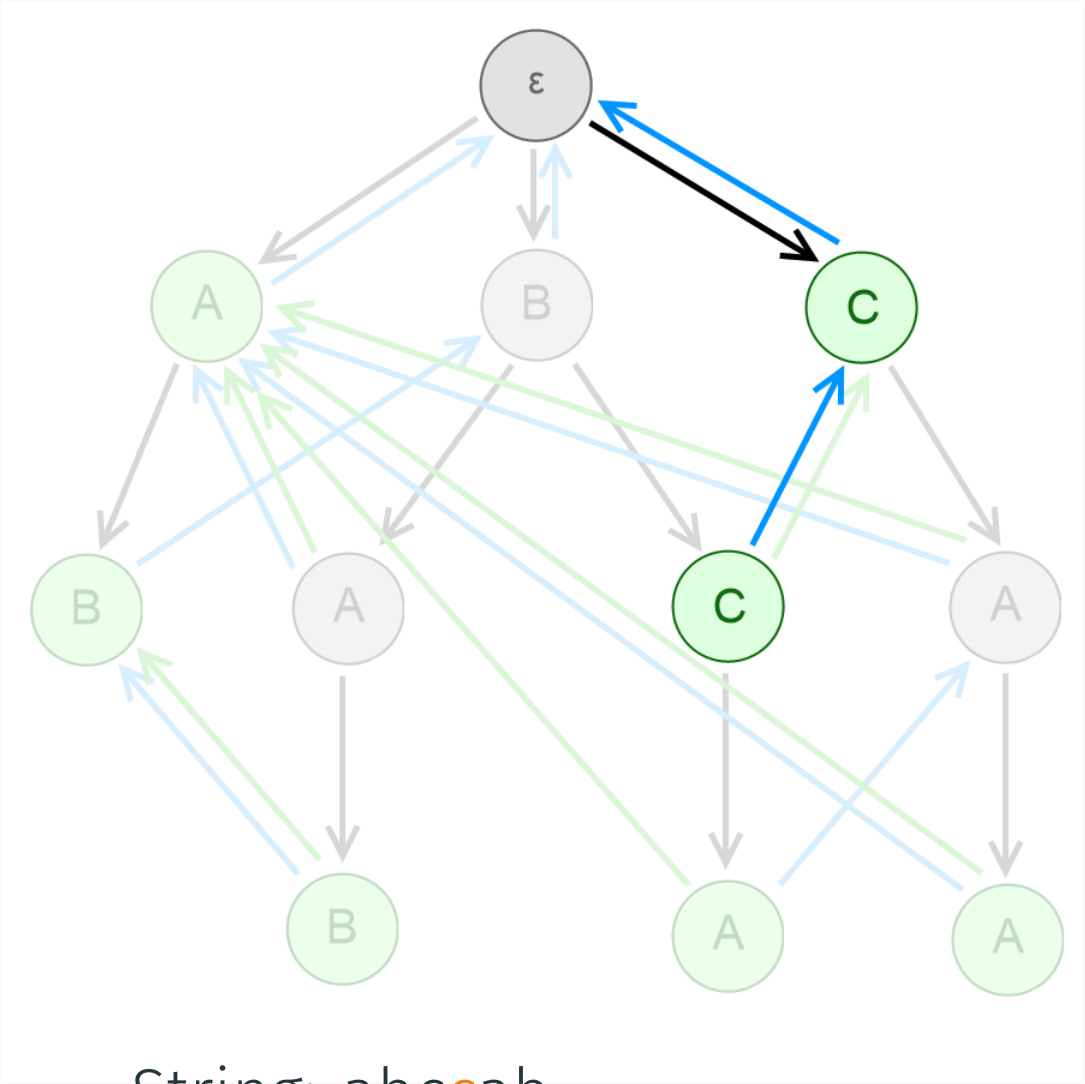
Aho Corasick: Matching Example



String: abccab

Output: a, ab, bc, c

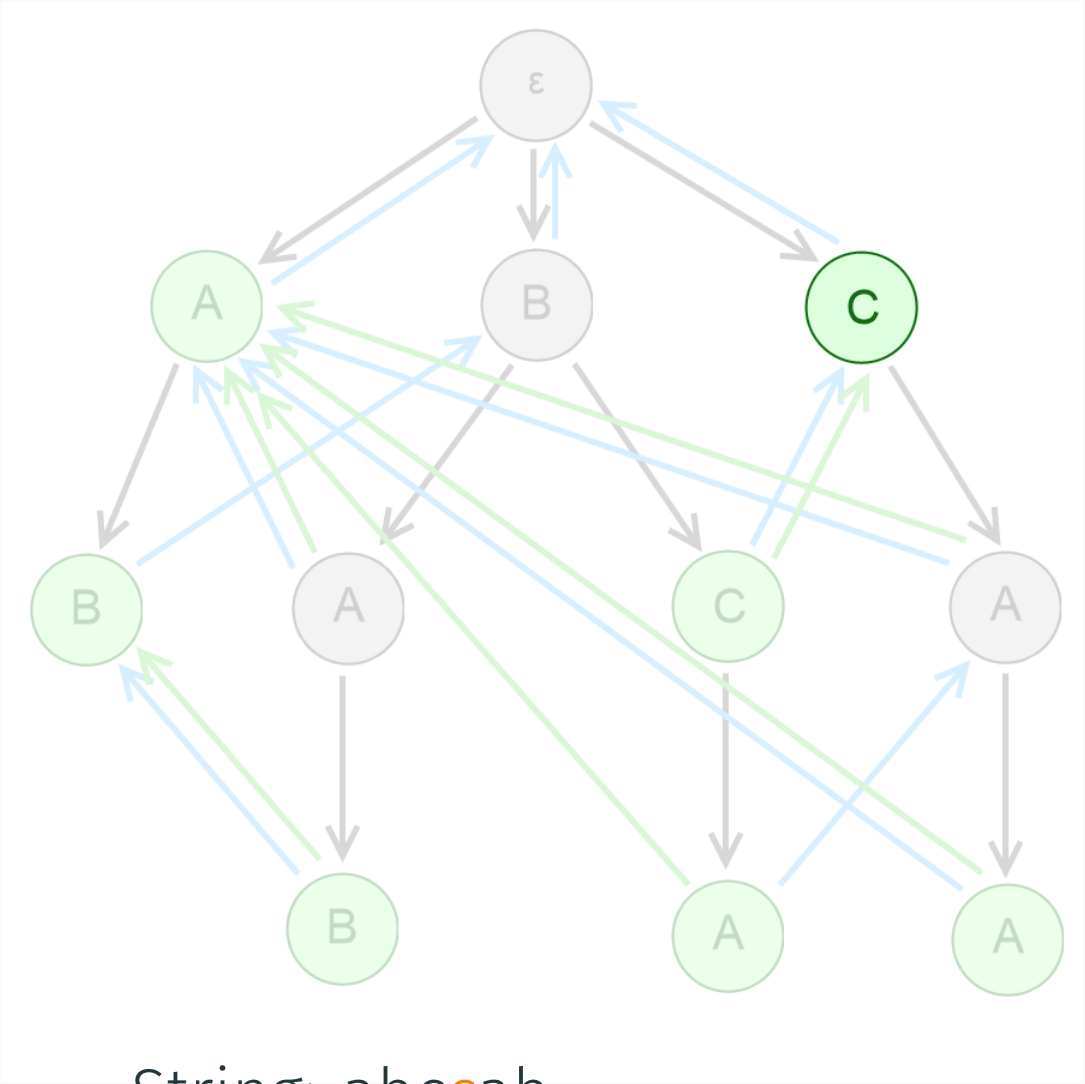
Aho Corasick: Matching Example



String: abccab

Output: a, ab, bc, c

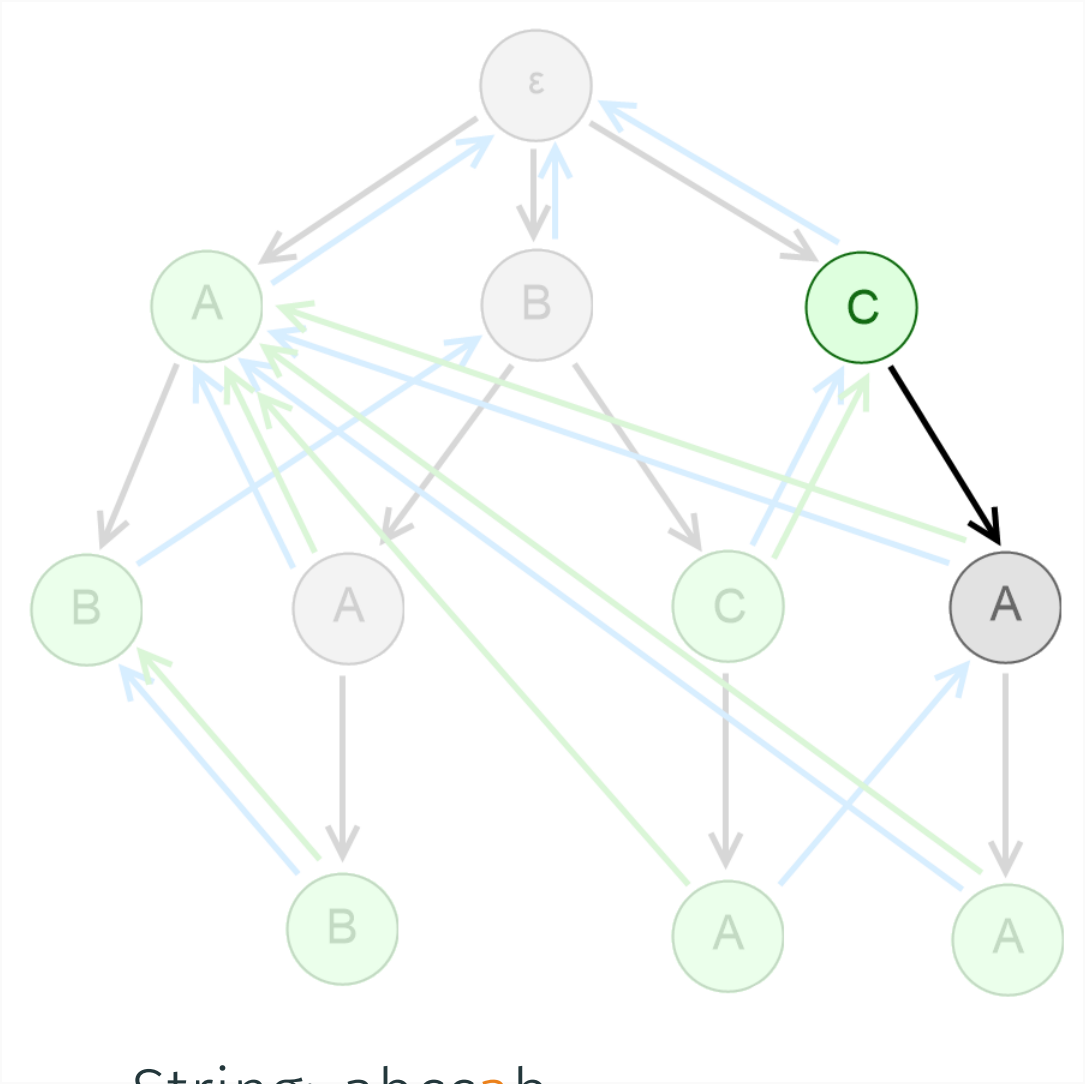
Aho Corasick: Matching Example



String: abccab

Output: a, ab, bc, c, c

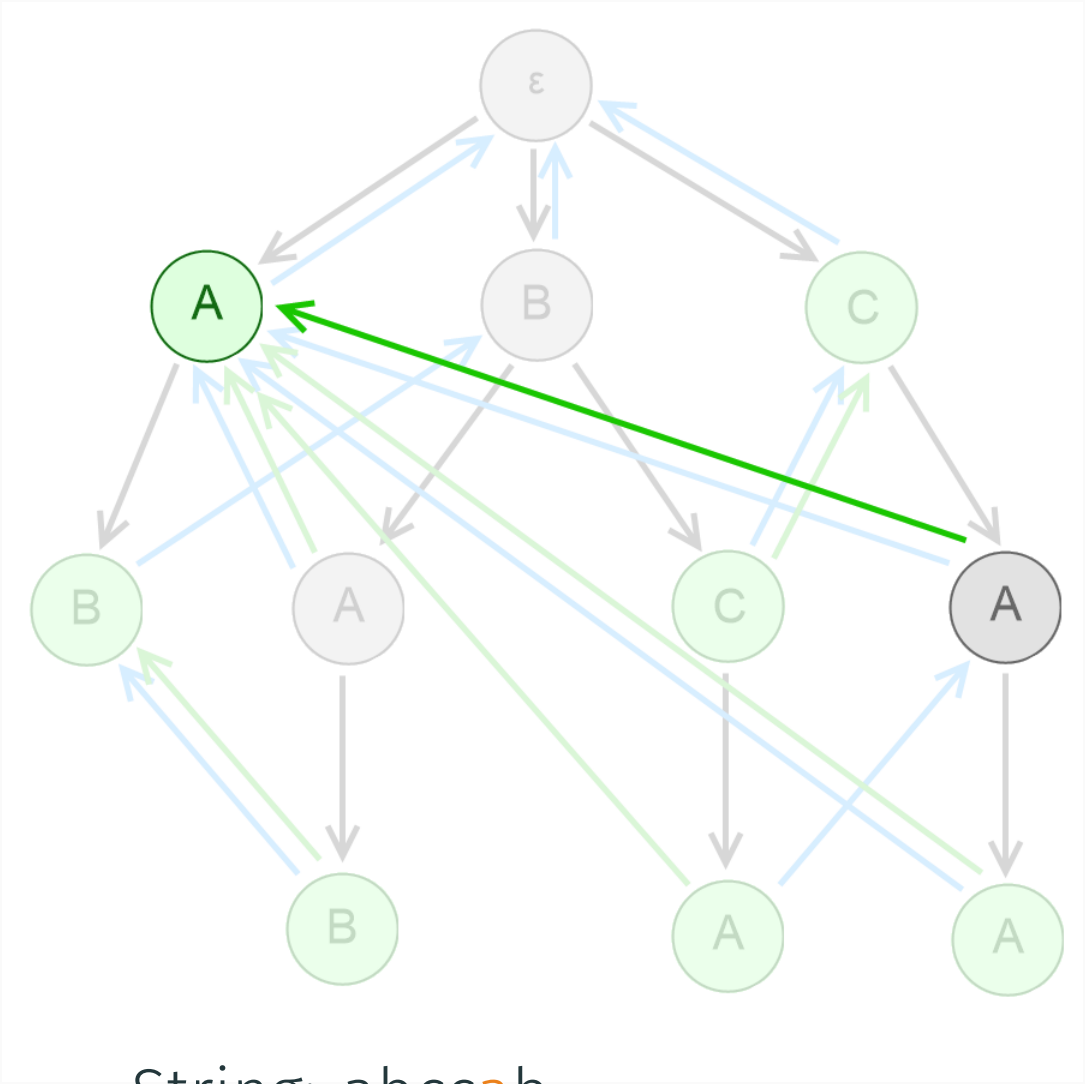
Aho Corasick: Matching Example



String: abccab

Output: a, ab, bc, c, c

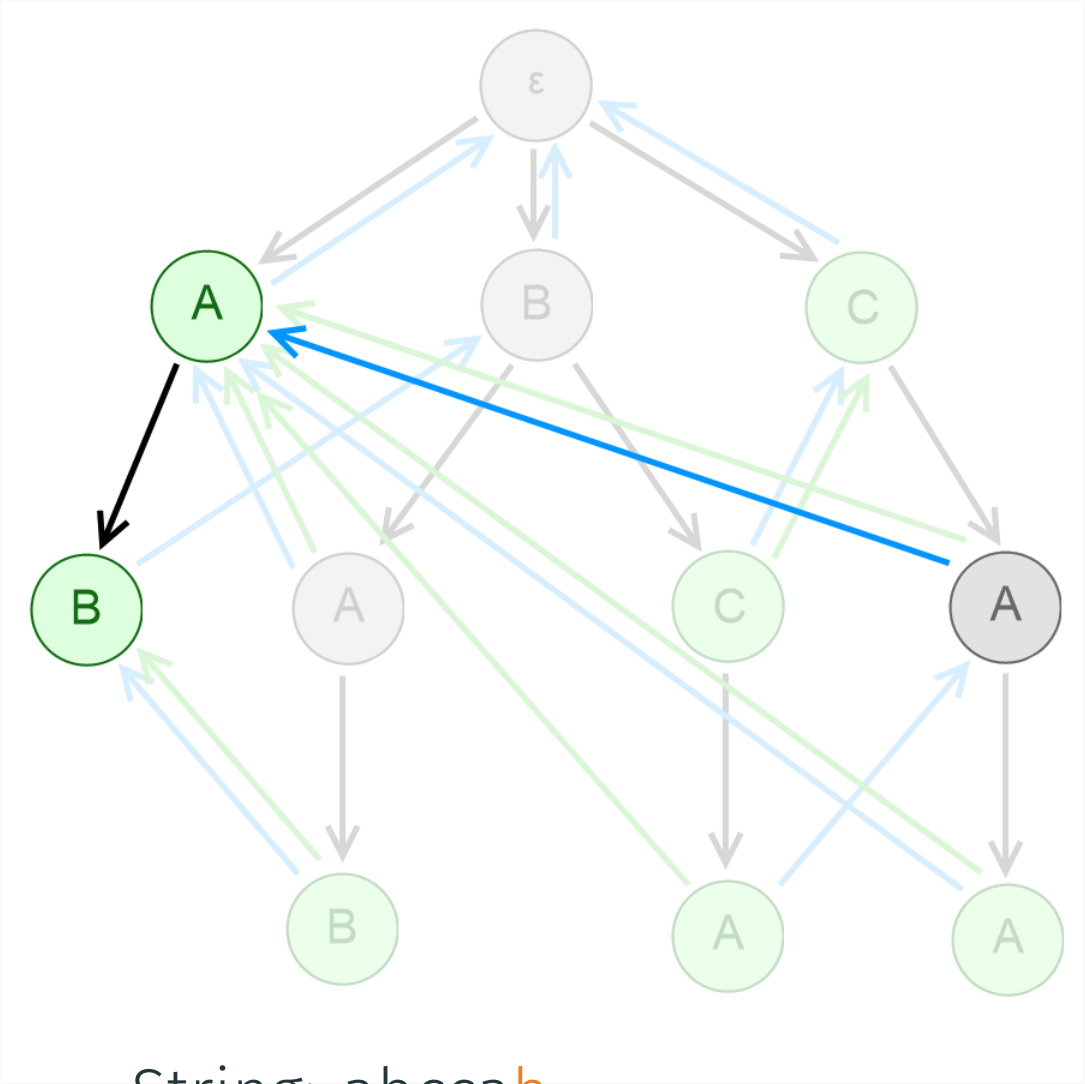
Aho Corasick: Matching Example



String: abccab

Output: a, ab, bc, c, c, a

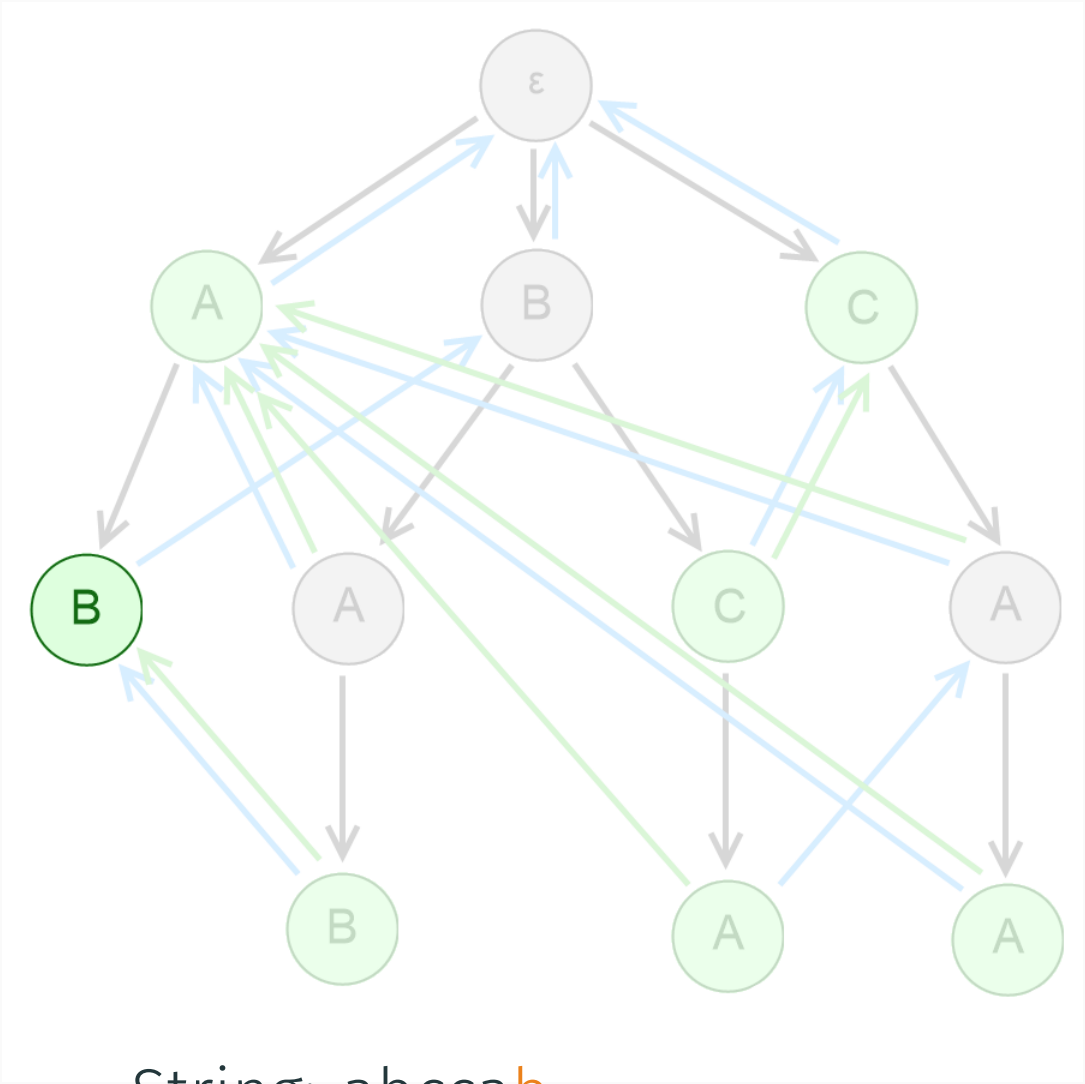
Aho Corasick: Matching Example



String: abccab

Output: a, ab, bc, c, c, a

Aho Corasick: Matching Example



String: abccab

Output: a, ab, bc, c, c, a, ab

Aho Corasick: Summary

Building the DFA

Aho Corasick: Summary

Building the DFA

- Build a trie of all the search strings

Aho Corasick: Summary

Building the DFA

- Build a trie of all the search strings
- Build Fail arrows: BFS through trie, for every node,
  - Follow parent's Fail arrow $\geq$ once until you reach another node with child of same char $\rightarrow$ point arrow to this child
  - If no match, point arrow to root

Aho Corasick: Summary

Building the DFA

- Build a trie of all the search strings
- Build Fail arrows: BFS through trie, for every node,
  - Follow parent's Fail arrow $\geq$ once until you reach another node with child of same char $\rightarrow$ point arrow to this child
  - If no match, point arrow to root
- Build Dictionary arrows: BFS through trie, for every node
  - Follow its Fail arrow once
  - If see a node ending a search string, point arrow to here
  - If see a node with Dictionary arrow, point arrow to its destination
  - Otherwise, point Dictionary arrow to NULL

Aho Corasick: Summary

Finding matches

Aho Corasick: Summary

Finding matches

- Initialize state = root of trie

Aho Corasick: Summary

Finding matches

- Initialize state = root of trie
- Read next character and transition state
  - When character match child, move to child
  - When character does mot match child, follow Fail arrows until you get to a node whose child matches character
  - Go to root if no match along Fail arrows

Aho Corasick: Summary

Finding matches

- Initialize state = root of trie
- Read next character and transition state
  - When character match child, move to child
  - When character does mot match child, follow Fail arrows until you get to a node whose child matches character
  - Go to root if no match along Fail arrows
- Output all matches ending at current character
  - Output match if current node is end of a search string
  - Follow Dictionary arrows to end and output every string seen.

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Building DFA

- For any path from root, depth of Fail arrow destination increases by at most one when you move down one node
- Fail arrow decrease depth by at least one
- $\Rightarrow$ Building Fail arrow for all nodes on a specific path from root takes O(path length) arrow traversals

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Building DFA

- For any path from root, depth of Fail arrow destination increases by at most one when you move down one node
- Fail arrow decrease depth by at least one
- $\Rightarrow$ Building Fail arrow for all nodes on a specific path from root takes O(path length) arrow traversals
- Trie = union of path for each string
- $\Rightarrow$ Fail arrows take $O(M)$ to build

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Building DFA

- For any path from root, depth of Fail arrow destination increases by at most one when you move down one node
- Fail arrow decrease depth by at least one
- $\Rightarrow$ Building Fail arrow for all nodes on a specific path from root takes O(path length) arrow traversals
- Trie = union of path for each string
- $\Rightarrow$ Fail arrows take $O(M)$ to build
- Dictionary arrows built in constant time per node (2 arrows)
- $\Rightarrow$ Dictionary arrows take $O(M)$ to build

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Building DFA

- For any path from root, depth of Fail arrow destination increases by at most one when you move down one node
- Fail arrow decrease depth by at least one
- $\Rightarrow$ Building Fail arrow for all nodes on a specific path from root takes O(path length) arrow traversals
- Trie = union of path for each string
- $\Rightarrow$ Fail arrows take $O(M)$ to build
- Dictionary arrows built in constant time per node (2 arrows)
- $\Rightarrow$ Dictionary arrows take $O(M)$ to build

$\Rightarrow$ Building the DFA takes $O(M)$

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Finding matches

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Finding matches

- Each iteration increase depth by at most one (follow Success arrow once) and each Fail arrow decrease depth by at least one
- $\Rightarrow$ Same as KMP analysis, $O(N)$ Fail/Success arrow traversals

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Finding matches

- Each iteration increase depth by at most one (follow Success arrow once) and each Fail arrow decrease depth by at least one
- $\Rightarrow$ Same as KMP analysis, $O(N)$ Fail/Success arrow traversals
- Each Dictionary arrow traversal = one distinct match
- $\Rightarrow O(Z)$ Dictionary arrow traversals

Aho Corasick: Complexity Analysis

Define

- $M =$ total length of all strings we are trying to find
- $N =$ length of the target string
- $Z =$ total number of matches

Finding matches

- Each iteration increase depth by at most one (follow Success arrow once) and each Fail arrow decrease depth by at least one
- $\Rightarrow$ Same as KMP analysis, $O(N)$ Fail/Success arrow traversals
- Each Dictionary arrow traversal = one distinct match
- $\Rightarrow O(Z)$ Dictionary arrow traversals

$\Rightarrow$ Finding matches takes $O(N + Z)$

$\Rightarrow$ Total time complexity: $O(M + N + Z)$

Problem 1 – Wildcards Yet Again

Find instances of S_1 in S_2, where * can stand for any single char.

S_1	S_2	Match?
c*mp*t*r	a computer	Yes
	comp0ter	Yes
	c0mp8t3r	Yes
	cooomputer	No

Figure 2: Example of single-character wildcard matching

Note: the original KMP method does not work!

Problem 1 – Solution

- Split S_1 by $*$ into pieces $T_1, \ldots, T_k$
- Run Aho Corasick to find all instances of $T_1, \ldots, T_k$ in S_2
- Initialize array A to 0
- For each match of T_j, increment $A[i]$ where i is the start index of S_1 if this occurrence of T_j were part of a complete match of S_1
- Output indices where $A[i] = k$
- Time complexity: $O(n + m \cdot \#\text{wildcards})$

Problem 2 – BFS

Given a list of $n \leq 16$ strings $S_1, S_2, \ldots, S_n$, find the shortest string that contain all of them as substrings, allowing overlap of occurrences!

Problem 2 – Solution

Observation: don't need entire string – only need longest suffix that matches prefix of some string we want.

Run BFS where the state is the pair of:

- Bitmask of which strings we have seen
- The current Aho Corasick state (i.e. trie node)

At each iteration of BFS, try appending all possible characters.

Time complexity: $O(2^n \cdot \sum |S_k|)$

Next Class

Suffix Array