



***Biosequence Algorithms, Spring 2005***  
***Lecture 4: Set Matching and***  
***Aho-Corasick Algorithm***

Pekka Kilpeläinen

University of Kuopio

Department of Computer Science

# Exact Set Matching Problem

In the **exact set matching problem** we locate occurrences of any pattern of a set  $\mathcal{P} = \{P_1, \dots, P_k\}$ , in target  $T[1 \dots m]$

Let  $n = \sum_{i=1}^k |P_i|$ . Exact set matching can be solved in time

$$O(|P_1| + m + \dots + |P_k| + m) = O(n + km)$$

by applying any linear-time exact matching  $k$  times

**Aho-Corasick algorithm** (AC) is a classic solution to exact set matching. It works in time  $O(n + m + z)$ , where  $z$  is number of pattern occurrences in  $T$

(Main reference here [Aho and Corasick, 1975])

AC is based on a refinement of a **keyword tree**

# Keyword Trees

A **keyword tree** (or a **trie**) for a set of patterns  $\mathcal{P}$  is a rooted tree  $\mathcal{K}$  such that

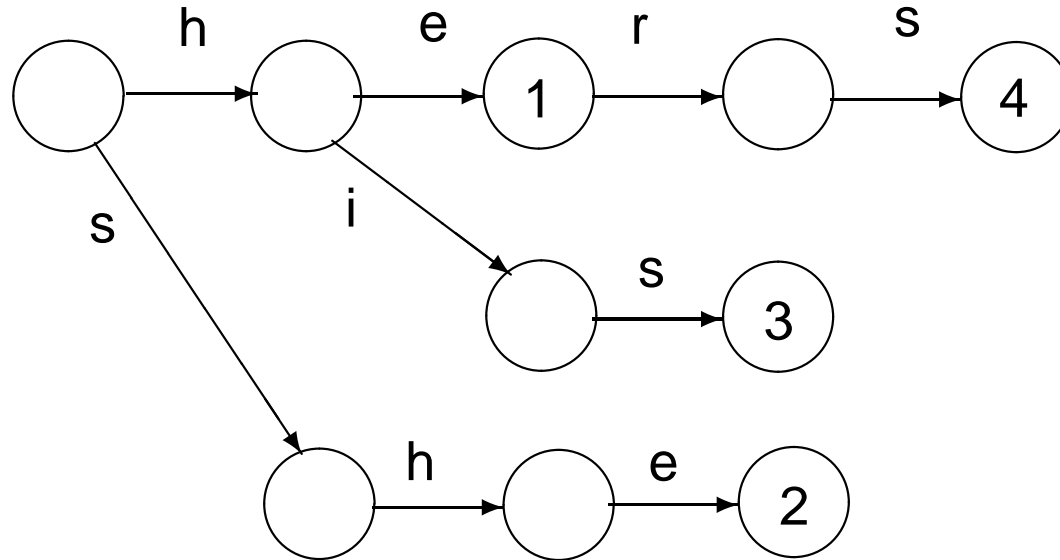
1. each edge of  $\mathcal{K}$  is labeled by a character
2. any two edges out of a node have different labels

Define the **label of a node**  $v$  as the concatenation of edge labels on the path from the root to  $v$ , and denote it by  $\mathcal{L}(v)$

3. for each  $P \in \mathcal{P}$  there's a node  $v$  with  $\mathcal{L}(v) = P$ , and
4. the label  $\mathcal{L}(v)$  of any *leaf*  $v$  equals some  $P \in \mathcal{P}$

# Example of a Keyword Tree

A keyword tree for  $\mathcal{P} = \{\text{he, she, his, hers}\}$ :



A keyword tree is an efficient implementation of a **dictionary** of strings

# Keyword Tree: Construction

**Construction** for  $\mathcal{P} = \{P_1, \dots, P_k\}$ :

Begin with a root node only;

Insert each pattern  $P_i$ , one after the other, as follows:

Starting at the root, follow the path labeled by chars of  $P_i$ ;

- ⑥ If the path ends before  $P_i$ , continue it by adding new edges and nodes for the remaining characters of  $P_i$
- ⑥ Store identifier  $i$  of  $P_i$  at the terminal node of the path

This takes clearly  $O(|P_1| + \dots + |P_k|) = O(n)$  time

# Keyword Tree: Lookup

**Lookup** of a string  $P$ : Starting at root, follow the path labeled by characters of  $P$  as long as possible;

- ⑥ If the path leads to a node with an identifier,  $P$  is a keyword in the dictionary
- ⑥ If the path terminates before  $P$ , the string is not in the dictionary

Takes clearly  $O(|P|)$  time — An efficient look-up method!

Naive application to pattern matching would lead to  $\Theta(nm)$  time

Next we extend a keyword tree into an **automaton**, to support *linear-time* matching

# Aho-Corasick Automaton (1)

**States:** nodes of the keyword tree

**initial state:**  $0 =$  the root

Actions are determined by three functions:

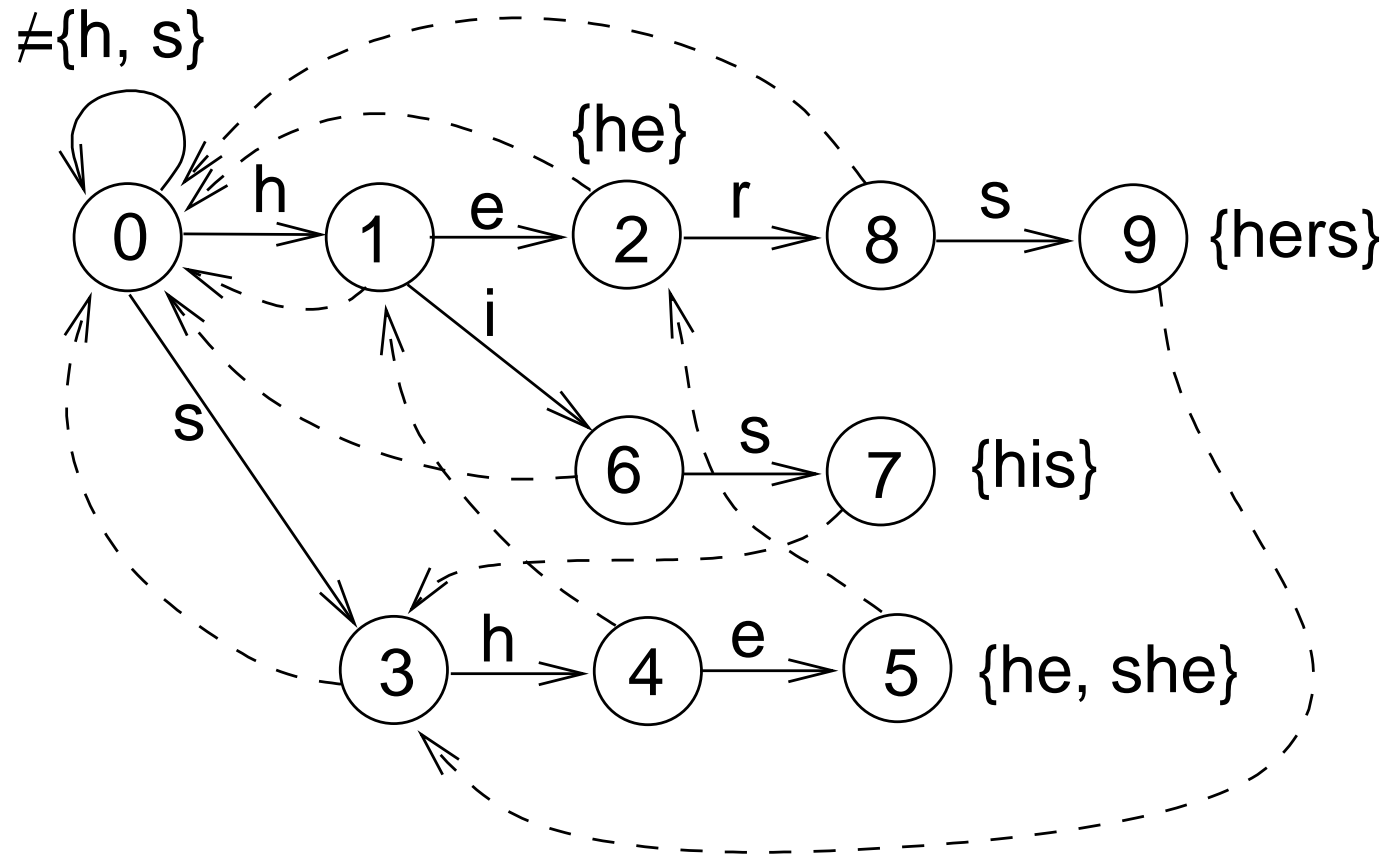
1. the **goto function**  $g(q, a)$  gives the state entered from current state  $q$  by matching target char  $a$ 
  - ⌚ if edge  $(q, v)$  is labeled by  $a$ , then  $g(q, a) = v$ ;
  - ⌚  $g(0, a) = 0$  for each  $a$  that does not label an edge out of the root  
↪ the automaton stays at the initial state while scanning non-matching characters
  - ⌚ Otherwise  $g(q, a) = \emptyset$

# Aho-Corasick Automaton (2)

2. the **failure function**  $f(q)$  for  $q \neq 0$  gives the state entered at a mismatch
  - ⑥  $f(q)$  is the node labeled by the *longest proper suffix*  $w$  of  $\mathcal{L}(q)$  s.t.  $w$  is a prefix of some pattern  
→ a fail transition does not miss any potential occurrences  
**NB:**  $f(q)$  is always defined, since  $\mathcal{L}(0) = \epsilon$  is a prefix of any pattern
3. the **output function**  $\text{out}(q)$  gives the set of patterns recognized when entering state  $q$



# Example of an AC Automaton



Dashed arrows are fail transitions

# AC Search of Target $T[1 \dots m]$

```
 $q := 0$ ; // initial state (root)
for  $i := 1$  to  $m$  do
    while  $g(q, T[i]) = \emptyset$  do
         $q := f(q)$ ; // follow a fail
     $q := g(q, T[i])$ ; // follow a goto
    if  $\text{out}(q) \neq \emptyset$  then print  $i, \text{out}(q)$ ;
endfor;
```

## Example:

Search text “ushers” with the preceding automaton

# Complexity of AC Search

**Theorem** Searching target  $T[1 \dots m]$  with an AC automaton takes time  $O(m + z)$ , where  $z$  is the number of pattern occurrences

**Proof.** For each target character, the automaton performs 0 or more *fail* transitions, followed by a *goto*.

Each *goto* either stays at the root, or increases the depth of  $q$  by 1  $\Rightarrow$  the depth of  $q$  is increased  $\leq m$  times

Each *fail* moves  $q$  closer to the root

$\Rightarrow$  the total number of fail transitions is  $\leq m$

The  $z$  occurrences can be reported in  $z \times O(1) = O(z)$  time (say, as pattern identifiers and start positions of occurrences) □

# Constructing an AC Automaton

The AC automaton can be constructed in two phases

## Phase I:

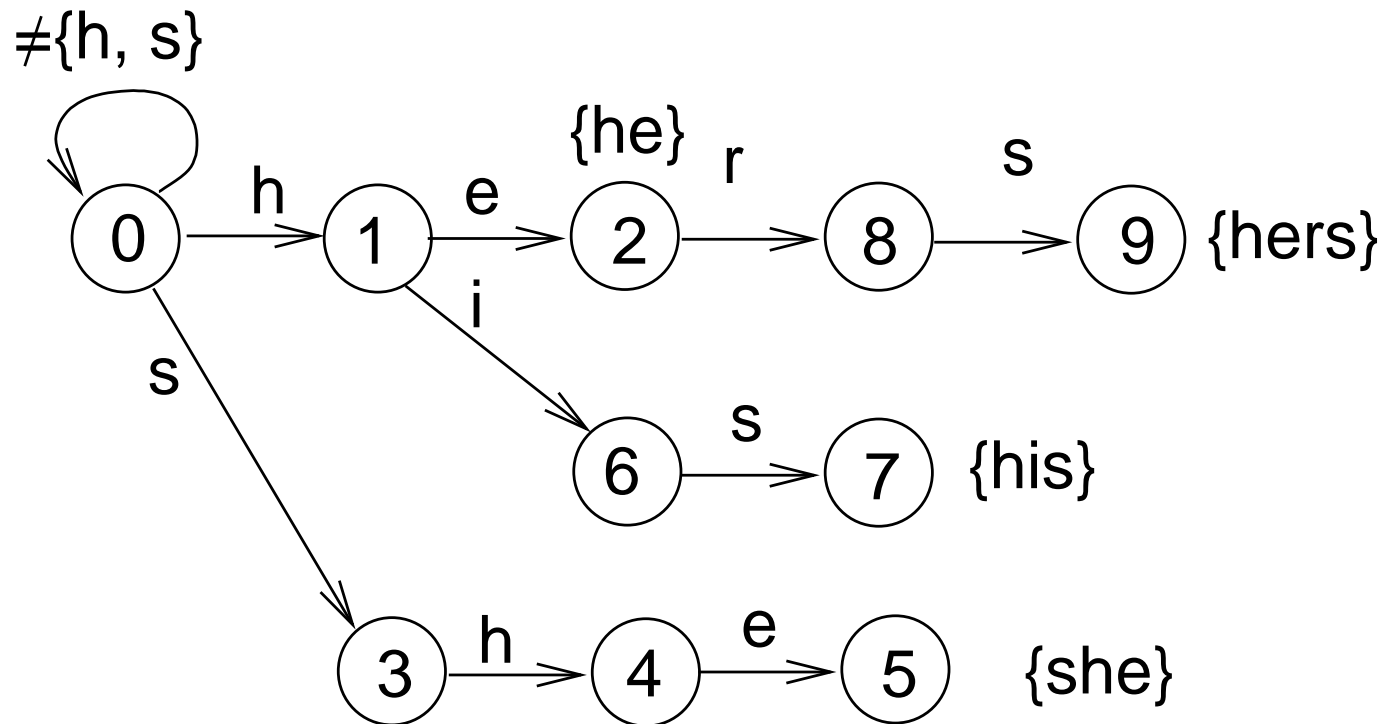
1. Construct the keyword tree for  $\mathcal{P}$ 
  - ⑥ for each  $P \in \mathcal{P}$  added to the tree, set  $\text{out}(v) := \{P\}$  for the node  $v$  labeled by  $P$
2. complete the *goto* function for the root by setting

$$g(0, a) := 0$$

for each  $a \in \Sigma$  that doesn't label an edge out of the root

If the alphabet  $\Sigma$  is fixed, Phase I takes time  $O(n)$

# Result of Phase I



# Phase II of the AC Construction

```
Q := emptyQueue();
for a ∈ Σ do
  if q(0, a) = q ≠ 0 then
    f(q) := 0; enqueue(q, Q);
while not isEmpty(Q) do
  r := dequeue(Q);
  for a ∈ Σ do
    if g(r, a) = u ≠ ∅ then
      enqueue(u, Q); v := f(r);
      while g(v, a) = ∅ do v := f(v); // (*)
      f(u) := g(v, a);
      out(u) := out(u) ∪ out(f(u));
```

What does this do?

# Explanation of Phase II

Functions *fail* and *output* are computed for the nodes of the trie in a breadth-first order

↪ nodes closer to the root have already been processed

Consider nodes  $r$  and  $u = g(r, a)$ , that is,  $r$  is the parent of  $u$  and  $\mathcal{L}(u) = \mathcal{L}(r)a$

Now what should  $f(u)$  be?

**A:** The deepest node labeled by a proper suffix of  $\mathcal{L}(u)$ .

The executions of line (\*) find this, by locating the deepest node  $v$  s.t.  $\mathcal{L}(v)$  is a proper suffix of  $\mathcal{L}(r)$  and  $g(v, a)$  ( $= f(u)$ ) is defined.

(Notice that  $v$  and  $g(v, a)$  may both be the root.)

# Completing the Output Functions

What about

$$\text{out}(u) := \text{out}(u) \cup \text{out}(f(u)); ?$$

This is done because the patterns recognized at  $f(u)$  (if any), and only those, are proper suffixes of  $\mathcal{L}(u)$ , and shall thus be recognized at state  $u$  also.



# Complexity of the AC Construction

Phase II can be implemented to run in time  $O(n)$ , too:

The breadth-first traversal alone takes time proportional to the size of the tree, which is  $O(n)$ ;

OK; ...

Is there also an  $O(n)$  bound for the number of times that the  $f$  transitions are followed (on line (\*))?

**A:** Yes! See next

# AC Construction: Number of fail transitions

Consider the nodes  $u_1, \dots, u_l$  on a path created by entering a pattern  $a_1 \dots a_l$  to the tree, and the depth of their  $f$  nodes, denoted by  $df(u_1), \dots, df(u_l)$  (all  $\geq 0$ )

Now  $df(u_{i+1}) \leq df(u_i) + 1 \Rightarrow$  the  $df$  values increase at most  $l$  times along the path. When locating  $f(u_{i+1})$ , each execution of line (\*) takes  $v$  closer to the root, and thus makes value of  $df(u_{i+1})$  *smaller* than  $df(u_i) + 1$  *by one at least*

$\rightsquigarrow$  line (\*) is executed in total  $\leq l$  times (for a pattern of length  $l$ )

$\rightsquigarrow$  line (\*) is executed in total, for all patterns,  $\leq n$  times

# AC Construction: Unions of output functions

Is it costly to perform

$$\text{out}(u) := \text{out}(u) \cup \text{out}(f(u)); ?$$

**No:** Before the assignment,  $\text{out}(u) = \emptyset$  or  $\text{out}(u) = \{\mathcal{L}(u)\}$ .

Any patterns in  $\text{out}(f(u))$  are shorter than  $\mathcal{L}(u)$

$\Rightarrow$  the sets are disjoint

$\rightarrow$  Output sets can be implemented as linked lists, and united in constant time

# Biological Applications

## 1. Matching against a library of known patterns

A **Sequence-tagged-site** (STS) is, roughly, a DNA string of 200–300 bases whose left and right ends occur only once in the entire genome

**ESTs** (expressed sequence tags) are STSs that participate in gene expression, and thus belong to genes

Hundreds of thousands of STSs and tens of thousands of ESTs (by mid-90's) are stored in databases, and used to compare against new DNA sequences

~> Ability to search for occurrences of patterns in time that is *independent of their number* is very useful

## 2. Matching with Wild Cards

Let  $\phi$  be a **wild card** that matches any *single* character

For example,  $ab\phi\phi c\phi$  occurs at positions 2 and 7 of

1234567890123  
xabvccababcax

A **transcription factor** is a protein that binds to specific locations of DNA and regulates its transcription to RNA

Many transcription factors are separated into families characterized by substrings with wild cards

**Example:** Transcription factor *Zinc Finger* has signature

$C\phi\phi C\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi H\phi\phi H$

( $C$  = cysteine,  $H$  = histidine; amino acids)

## Matching with Wild Cards (2)

If the number of wild cards is bounded by a constant, patterns with wild-cards can be matched in linear time, by counting occurrences of non-wild-card substrings of  $P$ :

Let  $\mathcal{P} = \{P_1, \dots, P_k\}$  be the substrings of  $P$  separated by wild-cards, and let  $l_1, \dots, l_k$  be their end positions in  $P$

**Preprocess:** Build an AC automaton for  $\mathcal{P}$ ;

Initiate occurrence counts: **for**  $i := 1$  **to**  $|T|$  **do**  $C[i] := 0$ ;

**Search** target  $T$  with the AC automaton

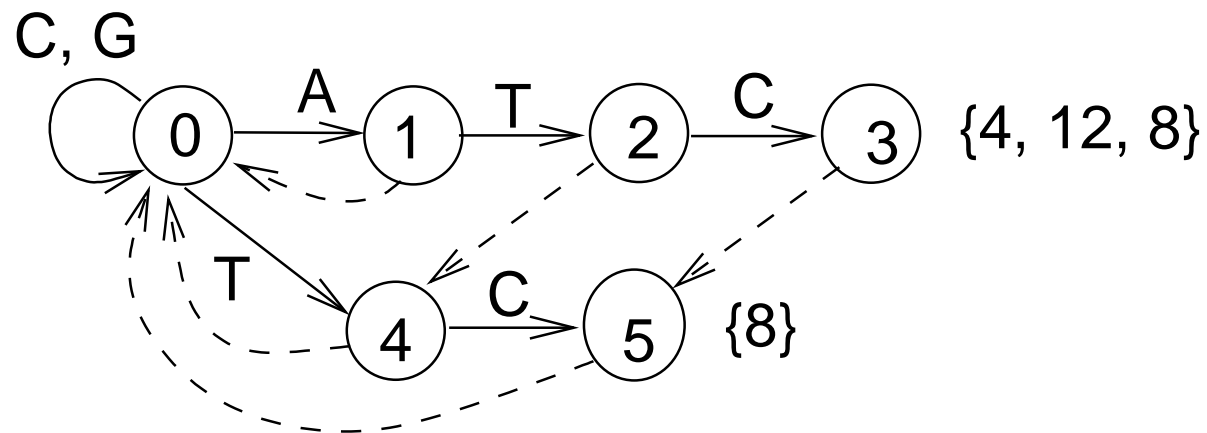
When pattern  $P_j$  is found to end at position  $i \geq l_j$  of  $T$ , increment  $C[i - l_j + 1]$  by one;

Any  $i$  with  $C[i] = k$  is the start position of an occurrence

# Example

Let  $P = \phi ATC \phi \phi TC \phi ATC$

Then  $\mathcal{P} = \{ATC, TC, ATC\}$  with  $l_1 = 4$ ,  $l_2 = 8$  and  $l_3 = 12$



Search on

$i$ : 12345678901234...

$T$ : ACGATCTCTCGATC...

$\rightsquigarrow C[1] = C[7] = C[11] = 1$  and  $C[3] = 3$  ( $\sim$  occurrence)

# Complexity of AC Wild-Card Matching

Let  $|P| = n$  and  $|T| = m$

Preprocessing:  $O(n + m)$  ( $\leftarrow \sum_{i=1}^k |P_i| \leq n$ )

Search:  $O(m + z)$ , where  $z$  is the number of occurrences

Each occurrence increments a cell of  $C$  by one, and each cell  $C[1], \dots, C[m]$  is incremented at most  $k$  times  
 $\Rightarrow z \leq km (= O(m))$  if  $k$  is bounded by a constant)

We have derived the following result:

**Theorem 3.5.1** If the number of wild-cards in pattern  $P$  is bounded by a constant, exact matching with wild-cards can be performed in time  $O(|P| + |T|)$  □