Molecular Sequence Algorithms, Spring 2004

Lecture 4: Set Matching and Aho-Corasick Algorithm

Pekka Kilpeläinen

University of Kuopio
Department of Computer Science

Exact Set Matching Problem

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

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

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

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
(The exposition is mainly based on [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

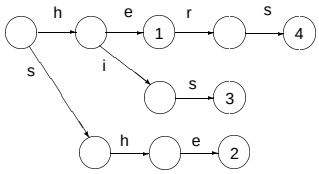
1. each edge e of $\mathcal{K}$ is labeled by a character
2. any two edges out of a node have different labels

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

3. for each $P \in \mathcal{P}$ there is a node v s.t. $\mathcal{L}(v) = P$, and
4. for each *leaf* v we have some $P \in \mathcal{P}$ with $\mathcal{L}(v) = P$

Example of a keyword tree

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



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

Keyword tree: Construction

Construction for $\mathcal{P} = \{P_1, \ldots, P_z\}$:

Begin with a root node only; Insert each pattern P_i, one after the other:

Follow the path labeled by characters of P_i as long as possible.

- If P_i exhausts at node v, store an identifier of P_i at v
- If the path terminates before P_i, continue the path by adding new edges and nodes for the remaining characters of P_i

Takes clearly $O(|P_1| + \cdots + |P_z|) = O(n)$ time

Keyword tree: Lookup

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

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

Takes clearly $O(|P|)$ time; efficient as a 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: root (denoted 0)

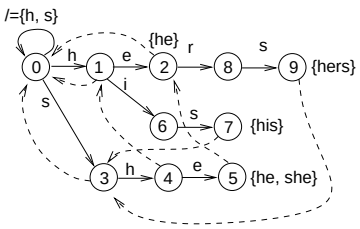
The action of the automaton is determined by three functions defined for the states:

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

Aho-Corasick automaton (2)

2. a **failure function** $f(s)$ gives the state entered at a mismatch
   - When w is the longest *proper suffix* of $\mathcal{L}(s)$ s.t. w is a prefix of some pattern, $f(s)$ is the node labeled by w
     $\rightsquigarrow$ we do not miss any potential occurrence by a fail transition
3. an **output function** out(s) gives the set of patterns recognized when entering state s

Example of an AC automaton



Dashed: fail transitions; those not shown lead to the root

Search using an AC automaton

$q := 0$; // initial state (root)
for $i := 1$ **to** m **do**
  while $g(q, T[i]) = \emptyset$ **do**
    $q := f(q)$;
  $q := g(q, T[i])$;
  if out$(q) \neq \emptyset$ **then print** i, out(q);
endfor;

Example:

Search text " ushers " with the preceding automaton

Efficiency of AC search

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

Proof. For each text char, we perform a *goto*, and possibly a number of *fail* transitions.
Each *goto* either stays at the root, or the depth of the current state (q) increases by 1
$\rightsquigarrow$ the depth of q is increased at most m times

Each *fail* moves q closer to the root $\rightsquigarrow$ the number of them can be at most m

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

Constructing an AC automaton (I)

An AC automaton can be constructed in two phases

Phase I:

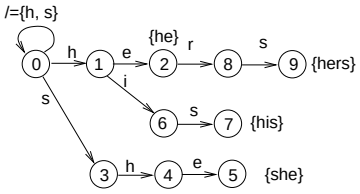
1. Construct the keyword tree for $\mathcal{P}$
   - for each $P \in \mathcal{P}$ added to the tree, if v is the node labeled by P, set out$(v) := \{P\}$
2. complete the *goto* function for the root by setting

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

for each $a \in \Sigma$ not labeling an edge out of the root

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

Result of Phase I



Constructing an AC automaton (II)

$Q :=$ emptyQueue();
for $a \in \Sigma$ **do**
  if $g(0, a) = q \neq 0$ **then**
    $f(q) := 0$; enqueue(q, Q);
while not isEmpty(Q) **do**
  r:= dequeue(Q);
  for $a \in \Sigma$ **do**
    if $g(r, a) = u \neq \emptyset$ **then**
      enqueue(u, Q); $v := f(r)$;
      while $g(v, a) = \emptyset$ **do** $v := f(v)$;
      $f(u) := g(v, a)$;
      out$(u) :=$ out$(u) \cup$ out$(f(u))$;

What does this do?

Idea of AC construction Phase II

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

$\rightsquigarrow$ when considering a node, nodes that are closer to the root have been treated

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 $\mathcal{L}(u)$.
This is found by trying nodes labeled by shorter and shorter suffixes of $\mathcal{L}(r)$, until a node v one is found for which $g(v, a)$ is defined and gets assigned to $f(u)$.
(Note that v and $g(v, a)$ may be the root.)

Completing the output functions

What about out$(u) :=$ out$(u) \cup$ out$(f(u))$?

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

Efficiency of the AC construction (1)

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 $O(n)$

How much work is done for following f transitions (in the inner-most loop)?

AC construction: Number of fail transitions

Consider the nodes $u_1, \ldots, u_l$ on a path created by entering a pattern $a_1 \ldots a_l$ to the tree, and the depths of their f nodes, denoted by $df(u_1), \ldots, df(u_l)$

Now $df(u_{i+1}) \leq df(u_i) + 1$, which means that the df values can increase at most l times along the path. Now each execution of $v := f(v)$ decreases the value of $df(u)$ by one at least

$\rightsquigarrow$ in total, at most l *fail* transitions (for a pattern of length l)

$\rightsquigarrow$ the f links are followed, in total, at most n times

AC construction: Unions of output functions

Is it costly to unite output functions (that is, to perform out(u) := out$(u) \cup$ out$(f(u))$)?

No: The sets can be implemented as linked lists, and a union thus in constant time

(Any patterns in out$(f(u))$ are shorter than $\mathcal{L}(u)$, which is (possibly) the only member of out(u) before the assignment)

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

$\rightsquigarrow$ set matching in time *independent of the number of patterns* is highly useful

2. Matching with wild cards

Let ϕ be a **wild card** that matches any *single* character

For example, $ab\phi\phi c\phi$ occurs at positions 2 and 8 of

$$xabvccababcax$$

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: Signature for a common transcription factor of *Zinc Finger*:

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

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

1. **for** $i := 1$ **to** m **do** $C[i] := 0$;
2. Using AC, locate occurrences of patterns in $\mathcal{P}$; When an occurrence of P_i is found to start at position j of T, increment $C[j - l_i + 1]$ by one;
3. Report any i with $C[i] = k$ as a start of an occurrence

Complexity of AC-based wild-card matching

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

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

Each occurrence increments a cell of C by one, and each cell is incremented at most k times
$\rightsquigarrow$ there can be at most km occurrences
($= O(m)$ if k is bounded by a constant)

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 $O(n + m)$