7. All Keywords in Text

The Aho-Corasick Algorithm

This section studies the problem of finding all keywords in a string of text. To gain a deeper understanding, read the paper (Aho and Corasick, 1975).

Problem 3: All Keywords Problem

Function Problem: Given a set of keywords \( \{k_0, \ldots, k_{n-1}\} \) and a text string \( T \), find all occurrence of the keywords in \( T \).

Example: Find all Keywords in a Text

Find keywords: \( \{\text{he, she, his, hers}\} \) in the text:

\[ T = \text{“the time for this ushers ashes”} \]

The keywords occur at positions

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td>1 20 27</td>
</tr>
<tr>
<td>she</td>
<td>19 26</td>
</tr>
<tr>
<td>his</td>
<td>14</td>
</tr>
<tr>
<td>hers</td>
<td>20</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14</td>
<td></td>
</tr>
<tr>
<td>t h e t i m e f o r t h i s u s h e r s a s h e s</td>
<td></td>
</tr>
<tr>
<td>15 16 17 18 19 20 21 22 23 24 25 26 27 28 29</td>
<td></td>
</tr>
</tbody>
</table>

Given the keywords, the Aho-Corasick algorithm constructs a finite state machine that recognizes each keyword. The machine consists of three functions that Aho and Corasick call \( \text{goto} \), \( \text{failure} \), and \( \text{output} \).

The \( \text{goto} \) function for the keywords, he, she, his, and hers is shown as a finite state machine in Figure 8 below. When a match is found, the \( \text{goto} \) function maps a (state, character) pair forward to a next state. For instance, if the machine is in state 2 and the next character is \( r \), then the machine moves to state 8. When the next character in the text does not move the search forward, the \( \text{failure} \)
function is called. For instance, if the machine is in state 1 and the next character is neither $e$ nor $i$, the machine fails back to the start state 0. Likewise, in states 2, 3, 6, or 8, if the next character is not $r$, $h$, $s$, or $s$, respectively, fail back to the start state 0. In state 4 where an $h$ was just seen, fail to state 1. In state 5 where he was just seen, fail to state 2. And, in state 7 or 9, where an $s$ was just seen, fail to state 3.

Finally, there is an output function that prints keywords and where they were found in the text.

<table>
<thead>
<tr>
<th>State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>he</td>
</tr>
<tr>
<td>5</td>
<td>{she, he}</td>
</tr>
<tr>
<td>7</td>
<td>his</td>
</tr>
<tr>
<td>9</td>
<td>hers</td>
</tr>
</tbody>
</table>
With these functions, the Aho-Corasick algorithm can be written as follows. The code is object-oriented. It assumes a class called State that has auxiliary goto, failure, and output functions.

The general structure of the algorithm is shown in the code below.

**Listing 19: Aho-Corasick All Keywords Search**

```java
89\a
\langle \text{Aho–Corasick Algorithm } 89\a \rangle \equiv
\langle \text{Auxiliary functions } 90\a \rangle

\text{public void ahoCorasick(String text) {}
  State state = start;
  for (int i = 0; i < text.length; i++) {
    \langle \text{While } goto \text{ fails follow the failure function } 89b \rangle
    \langle \text{Otherwise, follow the } goto \text{ function } 89c \rangle
    \langle \text{If in an output state, print information } 89d \rangle
    state = state.goto(text.charAt(i));
  }
}
```

\(89b\) \langle \text{While } goto \text{ fails follow the failure function } 89b \rangle \equiv
while (state.goto(text.charAt(i)) == fail) {
  state = state.failure();
  \langle \text{If in an output state, print information } 89d \rangle
}\)

Once a state where the label on an outgoing edge matches the character in the text, follow that edge to the next state.

\(89c\) \langle \text{Otherwise, follow the } goto \text{ function } 89c \rangle \equiv
state = state.goto(text.charAt(i));

Pretend the class State has an auxiliary function hasOutput that decides if a state has output to be displayed.

\(89d\) \langle \text{If in an output state, print information } 89d \rangle \equiv
if (state.hasOutput()) {
  state.output();
}
```
**Analysis of Aho-Corasick Algorithm**

The ahoCorasick() algorithm makes fewer than \(2n\) state transitions in processing a text string of length \(n\). To understand this note the following:

1. The algorithm makes exactly \(n\) goto (forward) transitions where \(n\) is the length of the text string.
2. For each character in the text the algorithm makes zero or more failure (backward) transitions.
3. To reach a state of depth \(d\) requires \(d\) goto transitions.
4. If we reach a state of depth \(d\), then no more than \(d\) failure transitions can occur afterwards.

Therefore, the number of failure transitions is no more than the number of goto transitions. (The number of gotos is greater than or equal to number of failures) The total the number of state transitions is therefore bound above by

\[
goto's + failure's \leq n + n = 2n
\]

and the algorithm makes \(2n\) or fewer transitions.

The goto and failure functions can be constructed in time \(O(m)\) where \(m\) is the length of the concatenated keywords.

**Aho-Corasick goto Function**

An array of keywords is passed into the buildGoTo function: The goto function is constructed. And the output function is partially defined.

Assume \(output(state)\) is empty when \(state\) is first created. Also, assume \(state.goto(c) = fail\) if \(c\) is undefined or if \(state.goto(c)\) has not yet been defined.

```java
90a Auxiliary functions 90a)
   public void buildGoTo(String[] keyword) {
      State start = new State();
      (Enter each keyword into the goto table 90b)
      (Add self-loops to start for characters not starting keywords 91a)
   }

90b Enter each keyword into the goto table 90b)
   for (int i = 0; i < keyword.length; i++) {
      (Enter keyword[i] 91c)
   }
```

Can you help find all the mistakes in this code? I’m certain there are many.
Add self-loops to start for characters not starting keywords \(91a\) ≡

*For each character c in the alphabet \(91b\)*

\[
\text{if (start.goto(c) == fail) start.goto(c) = start;}
\]

The alphabet and its implementation are not fleshed out here.

\(91b\)  
*For each character c in the alphabet \(91b\)* ≡

To enter a new keyword, first follow its already matched prefix. Then add states for its remaining suffix.

\(91c\)  
*Enter keyword[i] \(91c\)* ≡

\[
\begin{align*}
\text{State state} &= \text{start}; \\
\text{int } j &= 0; \\
\text{Follow existing path prefix of keyword[i] \(91d\)} \\
\text{Construct new path for suffix of keyword[i] \(91e\)} \\
\text{Save keyword[i] as output \(91f\)}
\end{align*}
\]

\(91d\)  
*Follow existing path prefix of keyword[i] \(91d\)* ≡

\[
\begin{align*}
\text{while (state.goto(keyword[i].charAt(j)) != fail) } & \{ \\
\text{state} &= \text{state.goto(keyword[i].charAt(j))}; \\
++j;
\}
\end{align*}
\]

\(91e\)  
*Construct new path for suffix of keyword[i] \(91e\)* ≡

\[
\begin{align*}
\text{for (int } k = j; k < \text{keyword[i].length}; k++) & \{ \\
\text{State newState} &= \text{new State()}; \\
\text{state.setGoto(keyword[i].charAt(k))} &= \text{newState}; \\
\text{state} &= \text{newState};
\}
\end{align*}
\]

\(91f\)  
*Save keyword[i] as output \(91f\)* ≡

\[
\text{state.saveOutput(keyword[i]);}
\]
Aho-Corasick failure Function

Now let’s build the failure function. First, add each state of depth 1 to a queue. Each such state fails back to the start state. Then, use the states stored in the queue to compute failures for states of greater depth.

92a \[ \text{Auxiliary functions } 90a \equiv \]
\[
\text{public void buildFailureTranstions()} {
  \langle \text{Add each state of depth one to a queue } 92b \rangle
  \langle \text{Compute failure for states of depth } d \text{ from those of depth } (d - 1) 92c \rangle
\}
\]

92b \[ \langle \text{Add each state of depth one to a queue } 92b \rangle \equiv \]
\[
\text{Queue queue = new Queue;}
  \langle \text{For each character } c \text{ in the alphabet } 91b \rangle {
    \text{State state = start.goto(c);}
    \text{if (state != start) {}
      \text{queue.enqueue(state);}
      \text{state.setFailure(start);}
    }
  }
\]

Next, get a state from the queue. For every character c that moves this state forward, queue up that nextState. Then, follow failures from state until they end. Set nextState to fail to where the goto function moves c.

Here is an example using figure 8: Pretend you are in state 8, a state of depth 3, having matched her. On the character s, move to nextState = 9. The failure from state 8 has already been computed to be the start state 0. From this start state, the goto function moves to state 3 on s. Therefore, failure in nextState = 9 moves to state 3.

92c \[ \langle \text{Compute failure for states of depth } d \text{ from those of depth } (d - 1) 92c \rangle \equiv \]
\[
\text{while (queue.notEmpty()) {}
  \text{State state = queue.dequeue();}
  \langle \text{For each character } c \text{ in the alphabet } 91b \rangle {
    \text{if (state.goto(c) != fail) {}
      \text{State nextState = state.goto(c);}
      \text{queue.enqueue(nextState);}
      \langle \text{Follow failures from state until they end } 93a \rangle
      \text{nextState.setFailure(failState.goto(c));}
      \text{state.saveOutput(nextState.output());}
    }
  }
\}
\]
Follow failures from state until they end \( \equiv \)
\[
\text{failState} = \text{state.failure()};
\]
\[
\text{while} \ (\text{failState.goto(c)} == \text{fail}) \ {
\text{failState} = \text{failState.failure()};}
\]

\subsection{Aho-Corasick [[output] Function]}

Auxiliary functions \( \equiv \)
\[
\text{public bool hasOutput() {}
\text{// to be determined}
\}
\]
\[
\text{public bool output() {}
\text{// to be determined}
\}
\]

**Exercises**

1. Construct the \textit{goto}, \textit{failure}, and \textit{output} functions for the keywords sofa, soft, take, tame, sort, fast


Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]


# Index

- **Haskell** programming language, 20, 31, 44, 116, 137
- **Java** programming language, 31
- Ackermann function, 174
- Axioms
  - Peano, 192
- Bézout, Étienne, 47
- C programming language, 30, 31, 42, 43, 55, 109, 152, 153
- Calendar, 7
- Cartesian product, 142
- Catalan numbers, 156, 161
- Computability, 191
- Computable function, 28
- Computer Language Benchmark Game, 30
- Contradiction, 193
- Dijkstra's algorithm, 183
- DNA, 146
- Empty
  - String λ, 61
- Fibonacci, 44, 46, 142
- Floating point
  - Machine epsilon, 50
- Function
  - Ackermann, 174
- Functional programming, 31
- Gödel, Kurt, 192
- Golden ratio, 142
- Greatest common divisor, 47
- Gutenberg Project, 84
- Hash Tables, 93
- Hofstadter, Douglas, 25
- Kleene, Stephen, 53
- Knuth, Donald, 30
- Lamé, Gabriel, 44
- λ calculus, 26, 28
- Linear congruence, 47
- Listing
  - C
    - Matrix Chain Multiplication, 158
- Listings
  - C
    - A Minimum Algorithm, 135
    - Aho-Corasick, 89
    - Boyer-Moore Pattern Matching, 72
    - Brute-force Left-to-Right Pattern Matching, 56
    - Bubble Sort, 109
    - Build a Heap, 126
    - Counting Sort, 128
    - Direct Address Table Operations, 94
    - Edit Distance, 152
    - Find Set with Path Compression, 177
    - Greatest Common Divisor, 42
    - Hash Table with Chaining, 95
    - Heap Sort, 127
    - Heapify a List, 126
    - Inner Product, 155
    - Insertion Sort, 112
    - Knuth–Morris–Pratt Pattern Matching, 67
    - Maximum Subsequence Sum (Cubic), 34
    - Maximum Subsequence Sum (Linear), 35
    - Merge Sort, 118
    - Morris–Pratt Pattern Matching, 58
    - Partition about a pivot, 120
    - Quick Sort, 120
    - Radix Exchange Sort, 132
    - Radix Sort, 129
    - Randomized Partition, 136
    - Randomized Selection, 138
    - Rational Knapsack, 167
    - Selection Sort, 114
    - Shell Sort, 116
    - Union by rank, 177
- Haskell
  - Ackermann function, 174
  - Apply a Function Repeatedly, 50
  - Brute-force Left-to-Right Pattern Matching, 57
  - Bubble Function, 108
  - Bubble Sort, 109
  - Extended Euclidean Algorithm, 48
  - Fold from the left, 37
  - Greatest Common Divisor, 44
  - Inner Product, 155
  - Insertion Sort, 112
  - Matrix Multiplication, 155
  - Maximum Subsequence Sum, 37
  - Merge Sort, 117
  - Merging Sorted Lists, 117
  - Newton's Square Root Method, 51
  - Newton's Square Root Recurrence, 50
  - Quicksort, 120
  - Randomized Partition, 137
  - Relative Error Convergence Test, 50
  - Sample Code Header, 31
  - Second of a pair, 36
  - Selection Sort, 114
  - Sorted Decision Problem, 106