7. All Keywords in Text

The Aho-Corasick Algorithm

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

The AhoâĂŞCorasick string matching algorithm formed the basis of the original Unix command fgrep.

Describe applications where an algorithm for this problem could be useful.

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: {`he, she, his, hers`} *in the text:*

T = "the time for this ushers ashes"

The keywords occur at positions

Keyword	***Index***		
`he`	*1*	*20*	*27*
`she`	*19*	*26*	
`his`	*14*		
`hers`	*20*		

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
`t`	`h`	`e`		`t`	`i`	`m`	`e`		`f`	`o`	`r`		`t`	`h`
15	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*
`i`	`s`		`u`	`s`	`h`	`e`	`r`	`s`		`a`	`s`	`h`	`e`	`s`

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 *goto*, *failure*, and *output*.

The *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 *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 `failure`

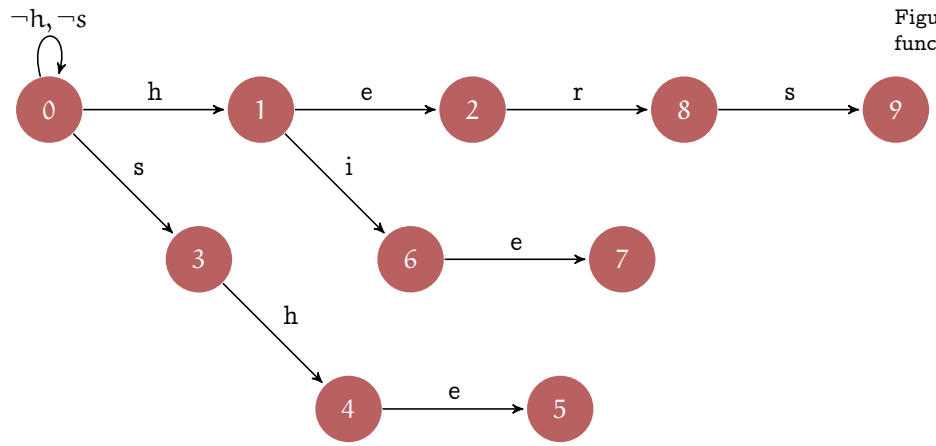

Figure 8: The Aho-Corasick `goto` function

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.

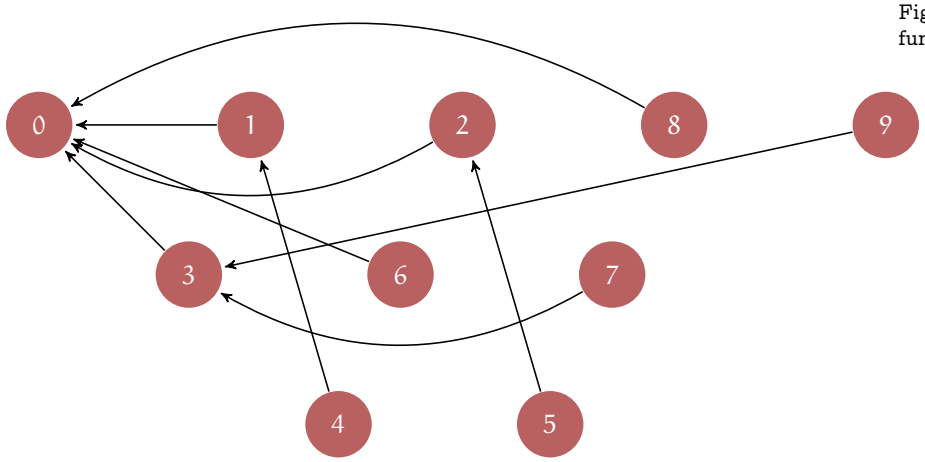

Figure 9: The Aho-Corasick `failure` function

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

State	Output
2	he
5	{she, he}
7	his
9	hers

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

89a ⟨*Aho–Corasick Algorithm* 89a⟩≡

```
⟨Auxiliary functions 90a⟩

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

89b ⟨*While* `goto` *fails follow the* `failure` *function* 89b⟩≡

```
while (state.goto(text.charAt(i)) == fail) {
  state = state.failure();
  ⟨If in an output state, print information 89d⟩
}
```

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

89c ⟨*Otherwise, follow the* `goto` *function* 89c⟩≡

```
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 ⟨*If in an* `output` *state, print information* 89d⟩≡

```
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 greeter than or equal to number of failures) The total the number of state transitions is therefore bound above by

$$\text{goto's} + \text{failure's} \leq n + n = 2n$$

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*

Can you help find all the mistakes in this code? I'm certain there are many.

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.

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⟩
}
```

91a ⟨*Add self-loops to* `start` *for characters not starting keywords* 91a⟩≡

```
⟨For each character c in the alphabet 91b⟩
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⟩≡

```
State state = start;
int j = 0;
⟨Follow existing path prefix of keyword[i] 91d⟩
⟨Construct new path for suffix of keyword[i] 91e⟩
⟨Save keyword[i] as output 91f⟩
```

91d ⟨*Follow existing path prefix of keyword[i]* 91d⟩≡

```
while (state.goto(keyword[i].charAt(j)) != fail) {
  state = state.goto(keyword[i].charAt(j));
  ++j;
}
```

91e ⟨*Construct new path for suffix of keyword[i]* 91e⟩≡

```
for (int k = j; k < keyword[i].length; k++) {
  State newState = new State();
  state.setGoto(keyword[i].charAt(k)) = newState;
  state = newState;
}
```

91f ⟨*Save* `keyword[i]` *as* `output` 91f⟩≡

```
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 ⟨*Auxiliary functions* 90a⟩+≡

```
public void buildFailureTranstions() {
  ⟨Add each state of depth one to a queue 92b⟩
  ⟨Compute failure for states of depth d from those of depth (d − 1) 92c⟩
}
```

92b ⟨*Add each state of depth one to a queue* 92b⟩≡

```
  Queue queue = new Queue;
  ⟨For each character c in the alphabet 91b⟩ {
    State state = start.goto(c);
    if (state != start) {
      queue.enqueue(state);
      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 ⟨*Compute* `failure` *for states of depth* d *from those of depth* (d − 1) 92c⟩≡

```
while (queue.notEmpty()) {
  State state = queue.dequeue();
  ⟨For each character c in the alphabet 91b⟩ {
    if (state.goto(c) != fail) {
      State nextState = state.goto(c);
      queue.enqueue(nextState);
      ⟨Follow failures from state until they end 93a⟩
      nextState.setFailure(failState.goto(c));
      state.saveOutput(nextState.output());
    }
  }
}
```

93a ⟨*Follow failures from* `state` *until they end* 93a⟩≡

```
failState = state.failure();
while (failState.goto(c) == fail) {
  failState = failState.failure();
}

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

93b ⟨*Auxiliary functions* 90a⟩+≡

```
public bool hasOutput() {
  // to be determined
}

public bool output() {
  // to be determined
}
```

Exercises

1. Construct the *goto*, *failure*, and *output* functions for the keywords

   sofa, soft, take, tame, sort, fast

Bibliography

Agrawal, M., Kayal, N., and Saxena, N. (2002). Primes is in P. http://www.cse.iitk.ac.in/news/primality.html. [page 205]

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 87]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 53]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 141]

Bentley, J. (1984a). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 33]

Bentley, J. (1984b). Programming Pearls: How to sort. *Commun. ACM*, 27(4):287–ff. [page 120]

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 71]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 199]

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]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 193], [page 203]

Gödel, K. (1930). Die vollst andigkeit der axiome des logischen funktionenkalk uls. *Monatshefte f ur Mathematik und Physik*, 37:349–360. [page 194]

Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications. [page 194]

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 53]

Hoare, C. A. R. (1961). Quicksort. *Computer Journal*, 5(1):10–15. [page 119]

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition*. Basic Books, New York, New York. [page 25]

Kernighan, B. W. and Ritchie, D. M. (1988). *The* C *Programming Language*. Prentice Hall, second edition. [page 31]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 30]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, ACM Press. [page 147]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 55]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. [page 31]

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley. [page 193], [page 195], [page 206]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., second edition. [page 30]

Polya, G. (1945). *How To Solve It*. Princeton University Press. ISBN 0-692-02356-5. [page 26]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley. [page 31]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 30]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, Reading, Massachusetts. [page 146]

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Informatica*, 7:327–355. [page 119]

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857. [page 119]

Sedgewick, R. (2004). *Algorithms in Java*. Addison-Wesley. [page 31]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 44]

Stansifer, R. (1984). Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department. [page 194]

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 197]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 19]

Index