

Pattern matching algorithms

Vineet Bafna *

October 4, 2004

1 Algorithms for keyword search

These are meant to supplement your class notes, but do not substitute for class lectures. Try these algorithms on examples and make sure that you understand how they work. The first algorithm (see Figure 1) describes keyword search without the use of failure nodes, and is therefore slower than the second. The second algorithm (see Figure 2) uses failure function F to ensure a linear running time. These algorithms assume that no pattern is a proper substring of the other. That case needs some special handling but the basic ideas remain the same.

procedure *SearchKeyword*

```
/*=====|
| T[c] is the database character at position c |
| The keyword tree or the trie is described by the following: |
| - A[v,X] is the node that v transitions to upon reading symbol X. |
| - Nodes that appear at the end of a pattern string are labeled with |
| an identifier for that pattern |
|=====*/

l = 1
c = 1
v = root
repeat
  if ((w = A(v,T[c])) ≠ φ)
    v = w
    c = c + 1
    if (v has label i)
      print "Pattern i matches starting at position l"
  else
    c = l + 1
    l = c
    v = root
  end
until (c > n) /* n is the database size*/
```

Figure 1: An $O(l_p n)$ algorithm for keyword search, where l_p is the length of the longest pattern

1.1 Analysis

An informal argument for correctness of Algorithm 2 (Figure 2) is as follows: At each step, we are at some node v in the automaton, and reading the symbol $T[c]$. If there is a valid transition, we simply take it, updating v to $A[v, T[c]]$, and c to $c + 1$. If there isn't one, and v is the root, there isn't any match possible at this position, so we simply increment c , update l to c and start again. On the other hand, if v is not the root, and does have a failure function $F[v]$, we simply change l to start at the new location.

*Computer Science Department, APM 3832, UC San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114. Email: vbafna@cs.ucsd.edu. Ph: 858-822-4978(W), 858-534-7029(F)

procedure *SearchKeyword*

```
/*=====|
| T[c] is the database character at position c |
| The keyword tree or the trie is described by the following: |
| -A[v,X] gives the node the v transitions to upon reading symbol X. |
| -F[v] is the node v transitions to if A[v,X] == NULL |
| -lp[v] is the length of the longest string which is a prefix for some |
| pattern and a proper suffix of the string denoted by node v |
| - Nodes that appear at the end of a pattern string are labeled with |
| an identifier for that pattern |
=====*/

l = 1
c = 1
v = root
repeat
  if ((w = A(v,T[c]) ≠ φ)
    v = w
    c = c + 1
    if (v has label i)
      print "Pattern i matches starting at position l"
  else if (v==root)
    c = c + 1
    l = c
  else
    l = c - lp[v]
    v = F[v]
  end
until (c > n)
```

Figure 2: An $O(n)$ algorithm for keyword search

For the running time, note that in every case either c is incremented, or l is incremented, and so the number of steps is bounded by the sum of the increments of c , and the increments of l . As both l , and c cannot exceed the database size n , and neither is decremented, the total number of steps is bounded by $2n$.

2 Regular Expression based search

In this section, we search a searching a database string T to see if a general regular expression R matches a *substring* of T . First, consider the simpler problem of determining if a *prefix* of T , $T[1 \dots c]$ matches R . As discussed in class, R can be described by an NFA with a start node s , and an end node t such that $T[1 \dots c]$ matches R if and only if there is a path from s to t that generates $T[1 \dots c]$. For each position c , define $N(c)$ as the set of states in R reachable after reading $T[1 \dots c]$. It is sufficient to generate these subsets because $T[1 \dots c]$ matches R if and only if $t \in N(c)$.

In order to compute $N(c)$, observe the following: v is reachable only if there is a state u with one of the following properties.

1. u is reachable after reading $T[1..c]$, and (u, v) is an ϵ -edge in R .
2. u is reachable after reading $T[1..c - 1]$ and (u, v) is an edge in R labeled with $T[c]$.

The algorithm in Figure 3 describes the recurrences corresponding to this. Finally, to see how to use this to check if a substring of T matches R , simply work with the regular expression $R' = \Sigma^*R$.

```

procedure SearchRegularExpression

/*=====|
| T[c] is the database character at position c |
| N(c) is a subset of nodes in regular expression R s.t. |
| - v in N(c) iff there is a path from s to v in R |
|   that generates T[1..c] |
| The Regular expression is described by the following: |
| - A start node s, and end node t |
| - Transition matrix A[v,X] gives the node that v transitions to upon |
|   reading symbol X. |
| - Eps(v) is the set of nodes that are reachable from v using epsilon |
|   transitions. |
=====*/

N(0) = {s} ∪ Eps(s)
for c = 1 to n
  N(c) = ∅
  for all (u ∈ N(c - 1))
    if ((v = A[u, T[c]]) ≠ ∅)
      N(c) = N(c) ∪ {v} ∪ Eps(v)
  end
  if t ∈ N(c)
    print " T matches the regular expression"
end

```

Figure 3: An $O(mn)$ time algorithm that tests if a prefix of T matches regular expression R