1 Notes on pattern matching using a FSM

Given a pattern $P = p_1p_2 \ldots p_m$ we can build an $m + 1$ state FSM to find all occurances of P in a text T. States are numbered 0,1, ..., m. Intuitively, being in state i means that the last i text symbols read match the first i sybmols of P (and for no $j > i$, the last j symbols read match the first j of P).

Let P_i denote the first i symbols of P. We build the FSM for P by first building a machine for the string P_1 (which is easy). We then show how to take an FSM for P_i and extend it to a FSM for P_{i+1}.

The FSM for P_1 has two states. From state 0 or 1 p_1 takes you to state 1. Any other symbol takes you to state 0 (from 0 or 1).

Now assume we have a correct FSM M for P_i. Let $p_{i+1} = X$ and let l be the state you go to if you are in state i and read X.

To create the new FSM M' for P_i we:

1) Add a new state $i + 1$

2) Change the arc from i to l on X to an arc from i to $i + 1$

3) For every symbol c in the alphabet, the arc out of $i + 1$ for c has the same endpoint as the arc out of l for symbol c.

Note that if $l = i$, for symbol X we use the new arc out of l (that is, if $l = i$, we stay in state $i + 1$ when we read an X).

Clearly this can be done in O(mr) time where r is the size of the alphabet: each time we add a new state we have to copy the r links from state l.

2 Converting an FSM to a KMP machine

Our FSM works well for string matching (particularly if the alphabet isn't too large), but it does require storing r pointers for each state if r is the alphabet size. It may also take a bit of work to decide which pointer to take even if our alphabet is of moderate size.

To fix these problems we show how to use an alternative machine which has only two pointers per machine: a success pointer which takes us to the next higher state for a match, and a failure pointer which takes us back to an earlier state when we mismatch with the next character of the pattern. This is the approach used in the KMP algorithm described in section 34.5 of the text. Here we describe a somewhat different way to construct a KMP-like machine.

Suppose we have built an FSM for our pattern P (as described above). Then the failure links we will use have a fairly simple explanation: for a state i let f_i be the non-success link which takes you the least far backwards and let c_i be the character associated with this link. Thus if you are in state i and don't get the next pattern character, this *could* be character c_i. Thus if you went farther back, you might miss a valid match.

Now of course, it may be wrong to go next to state f_i since the current text character might NOT be c_i. Thus we have to check if the current text character matches c_i. To do this we go not to state f_i but to $f_i - 1$. We then check the current text character (again) to see if it matches the character for state $f_i - 1$. If yes, the next character was c_i, so we are now in state f_i just as we would have been in our FSM, and can continue with further text characters. If we do not match, we take the failure link from state $f_i - 1$.

Construction details: As above let f_i be the non-match link from state i in the FSM machine which goes to the highest state (thus f_i could be as large as i but is never $i + 1$). The failure link for state i, F_i, is $f_i - 1$. Note that F_i can be -1. If F_i is -1 this means ALL failure links go to state zero, so we can go next to state zero without checking the current character again.

Code for matching: let t be the current text character and i be our current state. We start by setting i to zero.

REPEAT
i) Get the next text character t
ii) MATCH: IF $i < m$ and $p_{i+1} = t$ THEN $i \leftarrow i + 1$
iii) Otherwise:
$i \leftarrow F_i$
IF $i < 0$ { $i \leftarrow 0$; Go to i)}
ELSE Go to ii) // Continue trying to match the current character t
UNTIL all of T read

2.1 Analysis

A given text character can be compared multiple times, but there is at most one successful comparison per text character. Each failure comparison takes us to an earlier state in the machine (except perhaps the final one). Thus the total number of failure comparisons can be no more than the total number of success compares (to back up, we have to have had a success to go forward first). Thus the total number of comparisons with the text is at most $2n$. This is double the n compares of the FSM, but not too bad.

Note that our method for building the KMP machine is sort of round about. The book gives a direct O(m) method to build the machine (the time is independent of the alphabet size).

In fact you can make the construction more efficient (and in essence duplicate the book's approach in 34.4) if you note that since all we want to keep is the non-success link f_i which goes least far back, we can just copy that when we extend our FSM to the next node. The trick then is when extend from state i to $i+1$ how do we find the state l to be copied (since if the next character is say A, we don't know immediately where A went). The answer is that we take the failure link $F_i = k$ and check to see if the next character expected is A. If so we have found state $l = k + 1$. If not, take the failure link F_k from state k and see if its next character is an A. If so the state we want is $l = F_k + 1$, if not we keep failing back till we find an A or reach state zero.

As in the search algorithm a single check may be slow since we have to follow failure links back. However, as in the analysis of search, the total time spent following links is at most $2m$, and thus the total time to build the KMP machine is $O(m)$.