CSE 322 - Introduction to Formal Methods in Computer Science
String Matching

Dave Bacon

Department of Computer Science & Engineering, University of Washington

Suppose that you are given a short pattern and a long text and you wish to determine if that pattern appears somewhere in the text. For example, the pattern might be `xyxyyxyxyxx` and the text might be something like `xyxxyxyxyyxyxyxyyxyxyxx`. (What aren't all the things you read made up entirely of xs and ys? Sheesh, you're sure reading some sophisticated literature.) Here we will describe an algorithm (or a version of it) known as the Knuth-Morris-Pratt string matching algorithm.

A natural way to approach this problem is to begin to search through the text, and starting at each location in the text see if the pattern is matched. If, for example, the pattern to be match is in the array `P` of length `T` and the text is in an array `t` of length `n`, then simple pseudocode for doing this might be

```
for (i=0; T[i]!='\0' ; i++)
  {
  for (j=0; T[i+j]!='\0' && P[j]!='\0' && T[i+j]==P[j]; j++);
  if (P[j]=='\0') found match
  }
```

Lets see how this algorithm will perform on the strings listed above. During the first execution of the outer loop, the inner loop finds a partial match

```
xyxxyxyxyyxyxyxyyxyxyxx
xyxm
```

where m means mismatch, before it terminates the inner loop without having found the pattern. It then steps up the beginning of the out loop by one and the match fails after one check:

```
xyxxyxyxyyxyxyxyyxyxyxx
xyxm
 m
```

Okay it doesn't get that far. Continuing, the pattern matching algorithm will act as

```
xyxxyxyxyyxyxyxyyxyxyxx
xyxm
 m
  xm
   xyxym
    m
     xyxyyxyxm     (misses by one)
      m
       xym
        m
         m
          xyxym
            m
             xyxyyxyxyxx (success)
```

Clearly this is not a very good algorithm when you think about it. For example there are many places where knowing that we had failed previously is valuable information that we have just discarded because we are being myopic. In particular a worse case run time for this pattern matching algorithm will be $O(mn)$. So a good question to ask is whether this can be improved upon.

Two optimizations can be identified. One is in the outer loop. Consider, for example the beginning of the algorithm

```
xyxxyxyxyyxyxyxyyxyxyxx
xyxm
 m
```

Clearly we since we found a match of xyx we can skip the the next iteration of the outer loop since we know the second character is a y and not an x This came from knowing that we had the string xyx. Thus we should be executing something like at least

```
xyxxyxyxyyxyxyxyyxyxyxx
xyxm
  xm
```

Skipping the outer loops will lead to pseudo code like

```
i=0;
while(i<n)
  {
  for (j=0; T[i+j]!='\0' && P[j]!='\0' && T[i+j]==P[j]; j++);
  if P[j]=='\0' found a match;
  i=i+max(1,j-overlap(P[0..j-1],P[0...m]);
```

Some definitions to understand this code. First of all a *prefix* of a string $w = w_1w_2 \ldots w_n$ is a substring containing w_1: $w_1w_2 \ldots w_k$. A *suffix* of a string $w = w_1w_2 \ldots w_n$ is a substring containing w_n: $w_kw_{k+1} \ldots w_n$. The overlap of two strings x and y is the longest word that's a suffix of x and a prefix of y The overlap of two strings x and y is the longest word that's a suffix of x and a prefix of y. The function overlap in the above procedure is the length of the overlap of the two strings indicated, with the caveat that we do not allow this overlap to be all of the string x or all of the string y.. Thus, for example, the overlap of xyx and xyxyyxyxyxx is x whose length is 1.

Another optimization which can be performed is in the inner loops. For example above we skipped

```
xyxxyxyxyyxyxyxyyxyxyxx
xyxm
  xm
```

But really we can do better than this. We know that the third character is an x (this was the overlap used in the outer loop.) Thus we really don't need to test whether it matches, but instead can proceed to start testing the next character in the inner loop. In other words the overlap should be used to skip over steps in the inner loop.

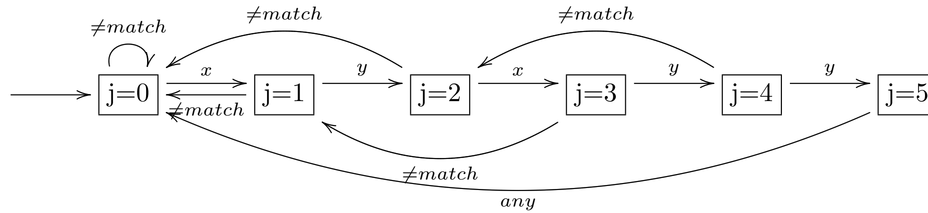
Skipping both outer and inner loops as appropriate leads to the pseudo code

```
i=0;
o=0;
while(i<n)
  {
  for (j=o; T[i+j]!='\0' && P[j]!='\0' && T[i+j]==P[j]; j++);
  if P[j]=='\0' found a match;
  o=overlap(P[0..j-1],P[0...m]);
  i=i+max(1,j-o);
```

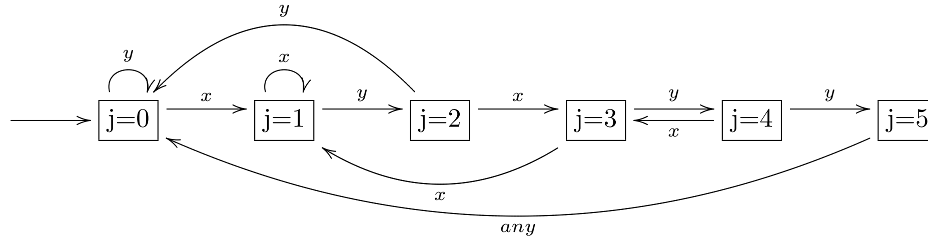
What is the running time of this algorithm? The basic idea is to look at how many times the comparison T[i+j]==P[j], which is in the inner loop, is performed. At first glance you might think this is again $O(mn)$, but this is not so. First of all consider the places where the value of T[i+j]==P[j] is true. In this case we've determined the value of T[i+j]. In future iterations, as long as there is a nontrivial overlap involving T[i+j] we'll skip past that overlap and not make a comparison of that position again. Thus there are at most n such true comparisons. On the other hand, for the false comparisons, the inner loop exits, so there can only be as many of these as there are outer loop executions, which is n. Thus the total running time is at most $2n$ which is $O(n)$.

So what does this all have to do with finite automata? Well we can think of the execution of the above algorithm as a lot like a finite automata. At each step j is kep and either set to j+1 (matching in the inner loop) or to the overlap o (after a mismatch) At each step the value of o is just a function of j. So we think of this as something like an automata with arrows connecting values of j and labeled by matches and mismatches. Now this is note quote the standard DFA since its execution is a bit different: if you get a mismatch, then you traverse the mismatch arrow, and then continue with the character you are currently looking at. Of course, you could convert this to a DFA by following mismatch arrows appropriately.

Let's construct this first "match"/"mismatch" DFA for the pattern: `xyxyy`.



Of course we could always convert this into a DFA since we can figure out what happens when we have a mismatch and read the next character. If we do this we obtain the following DFA:



So far we have talked about the runtime of this algorithm supposing that we have been given the function `overlap`. Of course overlap can be computed from the pattern before hand from `P`, but a question that arises in analyzing this algorithm is how much time it will take to figure out this function. It is helpful to calculate `overlap` for a string like `xyxyy`. First overlap(`x`,`xyxyy`)=ε, since we do not allow entire strings to be overlaps. Second overlap(`xy`,`xyxyy`)=ε, because `xy` ends in `y` and again we cannot return the whole string. Finally something interesting happens at the third step, overlap(`xyx`,`xyxyy`)=`x`. Next overlap(`xyxy`,`xyxyy`)=`xy` and overlap(`xyxyy`,`xyxyy`)=ε. Thus the array for `overlap` has values $0, 0, 1, 2, 0$.

So how do we compute the necessary array `overlap`? Well a naive way is to do exactly what we did before for the brute force algorithm and scan through `P`. This will give a running time of $O(m^2)$. For small m this isn't too bad, but it can be improved. Lets call the overlap array `F` for ease of notation. Then consider the following pseudo code:

```
F[0]=0;
i=1;
j=0;
while (i<m) do
  {
  if (P[i]==P[j])
    {
    // we have matched j+1 characters
    F[i]=j+1;
    i++;
    j++;
    }
  else
    if (j>0)
      {
      // use overlap function to shift P
      j=F[j-1];
      }
    else
      {
      F[i]=0; // no match found
      i++;
      }
  }
```

How does this code work? It is basic dynamic programming. `i` is the `F[i]` we are trying to calculate. `j` is a carrier for the size of the suffix we have so far matched. As we proceed through the algorithm, if you get a match between

P[i] and P[j] then you can calculate F[i] by adding one to j. If there isn't a match, then, if you haven't found a match before, then you need to set F[i] to zero and increment i. If, however, you have found a match before, then j contains the size of the suffix we have so far matched. We can then calculate F[j-1] to check if the prefix of this size could be matched with suffix of our current location, i.e. set j equal to this value. Then we proceed as before checking if P[i] and P[j] with this new j.

The running time for constructing the overlap is $O(m)$. The way to see this is again to analyze the cases where the P[i] == P[j] is evaluated. If it is true, then i is incremented. If it is false, then j can only be decremented. Since j can only be decremented m times, the total run time is bounded by $2m$.

Thus we see that we can calculate the overlap function using $O(m)$ time (and $O(m)$ space.) Thus the total run time of the KMP algorithm is $O(n + m)$.