ICS 161: Design and Analysis of Algorithms Lecture notes for February 27, 1996

Knuth-Morris-Pratt string matching

The problem: given a (short) pattern and a (long) text, both strings, determine whether the pattern appears somewhere in the text. Last time we saw how to do this with finite automata. This time we'll go through the Knuth-Morris-Pratt (KMP) algorithm, which can be thought of as an efficient way to build these automata. I also have some working C++ source code which might help you understand the algorithm better.

First let's look at a naive solution.
suppose the text is in an array: char T[n]
and the pattern is in another array: char P[m].

One simple method is just to try each possible position the pattern could appear in the text.

Naive string matching:

```
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 a match
}
```

There are two nested loops; the inner one takes O(m) iterations and the outer one takes O(n) iterations so the total time is the product, O(mn). This is slow; we'd like to speed it up.

In practice this works pretty well -- not usually as bad as this O(mn) worst case analysis. This is because the inner loop usually finds a mismatch quickly and move on to the next position without going through all m steps. But this method still can take O(mn) for some inputs. In one bad example, all characters in T[] are "a"s, and P[] is all "a"'s except for one "b" at the end. Then it takes m comparisons each time to discover that you don't have a match, so mn overall.

Here's a more typical example. Each row represents an iteration of the outer loop, with each character in the row representing the result of a comparison (X if the comparison was unequal). Suppose we're looking for pattern "nano" in text "banananobano".

```
      0  1  2  3  4  5  6  7  8  9 10 11
       T: b  a  n  a  n  a  n  o  b  a  n  o

    i=0: X
    i=1:    X
    i=2:       n  a  n  X
    i=3:          X
    i=4:             n  a  n  o
    i=5:                X
    i=6:                   n  X
    i=7:                         X
    i=8:                            X
    i=9:                               n  X
    i=10:                                 X
```

Some of these comparisons are wasted work! For instance, after iteration i=2, we know from the comparisons we've done that T[3]="a", so there is no point comparing it to "n" in iteration i=3. And we also know that T[4]="n", so there is no point making the same comparison in iteration i=4.

Skipping outer iterations

The Knuth-Morris-Pratt idea is, in this sort of situation, after you've invested a lot of work making comparisons in the inner loop of the code, you know a lot about what's in the text. Specifically, if you've found a partial match of j characters starting at position i, you know what's in positions T[i]...T[i+j-1].

You can use this knowledge to save work in two ways. First, you can skip some iterations for which no match is possible. Try overlapping the partial match you've found with the new match you want to find:

```
i=2: n  a  n
i=3:    n  a  n  o
```

Here the two placements of the pattern conflict with each other -- we know from the i=2 iteration that T[3] and T[4] are "a" and "n", so they can't be the "n" and "a" that the i=3 iteration is looking for. We can keep skipping positions until we find one that doesn't conflict:

```
i=2: n  a  n
i=4:       n  a  n  o
```

Here the two "n"'s coincide. Define the *overlap* of two strings x and y to be the longest word that's a suffix of x and a prefix of y. Here the overlap of "nan" and "nano" is just "n". (We don't allow the overlap to be all of x or y, so it's not "nan"). In general the value of i we want to skip to is the one corresponding to the largest overlap with the current partial match:

String matching with skipped iterations:

```
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]));
}
```

Skipping inner iterations

The other optimization that can be done is to skip some iterations in the inner loop. Let's look at the same example, in which we skipped from i=2 to i=4:

```
i=2: n  a  n
i=4:       n  a  n  o
```

In this example, the "n" that overlaps has already been tested by the i=2 iteration. There's no need to test it again in the i=4 iteration. In general, if we have a nontrivial overlap with the last partial match, we can avoid testing a number of characters equal to the length of the overlap.

This change produces (a version of) the KMP algorithm:

KMP, version 1:

```
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);
}
```

The only remaining detail is how to compute the overlap function. This is a function only of j, and not of the characters in T[], so we can compute it once in a *preprocessing* stage before we get to this part of the algorithm. First let's see how fast this algorithm is.

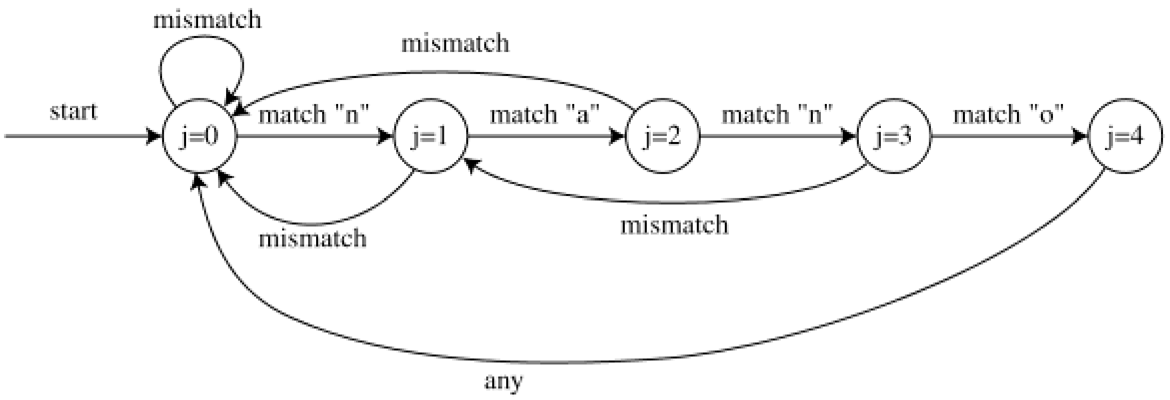
KMP time analysis

We still have an outer loop and an inner loop, so it looks like the time might still be O(mn). But we can count it a different way to see that it's actually always less than that. The idea is that every time through the inner loop, we do one comparison T[i+j]==P[j]. We can count the total time of the algorithm by counting how many comparisons we perform.

We split the comparisons into two groups: those that return true, and those that return false. If a comparison returns true, we've determined the value of T[i+j]. Then 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 with that position again. So each position of T[] is only involved in one true comparison, and there can be n such comparisons total. On the other hand, there is at most one false comparison per iteration of the outer loop, so there can also only be n of those. As a result we see that this part of the KMP algorithm makes at most 2n comparisons and takes time O(n).

KMP and finite automata

If we look just at what happens to j during the algorithm above, it's sort of like a finite automaton. At each step j is set either to j+1 (in the inner loop, after a match) or to the overlap o (after a mismatch). At each step the value of o is just a function of j and doesn't depend on other information like the characters in T[]. So we can draw something like an automaton, with arrows connecting values of j and labeled with matches and mismatches.



The difference between this and the automata we are used to is that it has only two arrows out of each circle, instead of one per character. But we can still simulate it just like any other automaton, by placing a marker on the start state (j=0) and moving it around the arrows. Whenever we get a matching character in T[] we move on to the next character of the text. But whenever we get a mismatch we look at the same character in the next step, except for the case of a mismatch in the state j=0.

So in this example (the same as the one above) the automaton goes through the sequence of states:

```
    j=0
            mismatch T[0] != "n"
    j=0
            mismatch T[1] != "n"
    j=0
            match T[2] == "n"
    j=1
            match T[3] == "a"
    j=2
            match T[4] == "n"
    j=3
            mismatch T[5] != "o"
    j=1
            match T[5] == "a"
    j=2
            match T[6] == "n"
    j=3
            match T[7] == "o"
    j=4
            found match
    j=0
            mismatch T[8] != "n"
    j=0
            mismatch T[9] != "n"
    j=0
            match T[10] == "n"
    j=1
            mismatch T[11] != "a"
    j=0
            mismatch T[11] != "n"
```

This is essentially the same sequence of comparisons done by the KMP pseudocode above. So this automaton provides an equivalent definition of the KMP algorithm.

As one student pointed out in lecture, the one transition in this automaton that may not be clear is the one from j=4 to j=0. In general, there should be a transition from j=m to some smaller value of j, which should happen on any character (there are no more matches to test before making this transition). If we want to find all occurrences of the pattern, we should be able to find an occurrence even if it overlaps another one. So for instance if the pattern were "nana", we should find both occurrences of it in the text "nanana". So the transition from j=m should go to the next longest position that can match, which is simply j=overlap(pattern,pattern). In this case overlap("nano","nano") is empty (all suffixes of "nano" use the letter "o", and no prefix does) so we go to j=0.

Alternate version of KMP

The automaton above can be translated back into pseudo-code, looking a little different from the pseudo-code we saw before but performing the same comparisons.

KMP, version 2:

```
    j = 0;
    for (i = 0; i < n; i++)
    for (;;) {      // loop until break
        if (T[i] == P[j]) { // matches?
        j++;        // yes, move on to next state
        if (j == m) {   // maybe that was the last state
            found a match;
            j = overlap[j];
        }
        break;
        } else if (j == 0) break;   // no match in state j=0, give up
        else j = overlap[j];    // try shorter partial match
    }
```

The code inside each iteration of the outer loop is essentially the same as the function `match` from the C++ implementation I've made available. One advantage of this version of the code is that it tests characters one by one, rather than performing random access in the T[] array, so (as in the implementation) it can be made to work for stream-based input rather than having to read the whole text into memory first.

The overlap[j] array stores the values of overlap(pattern[0..j-1],pattern), which we still need to show how to compute.

Since this algorithm performs the same comparisons as the other version of KMP, it takes the same amount of time, O(n). One way of proving this bound directly is to note, first, that there is one true comparison (in which T[i]==P[j]) per iteration of the outer loop, since we break out of the inner loop when this happens. So there are n of these total. Each of these comparisons results in increasing j by one. Each iteration of the inner loop in which we don't break out of the loop results in executing the statement j=overlap[j], which decreases j. Since j can only decrease as many times as it's increased, the total number of times this happens is also O(n).

Computing the overlap function

Recall that we defined the *overlap* of two strings x and y to be the longest word that's a suffix of x and a prefix of y. The missing component of the KMP algorithm is a computation of this overlap function: we need to know overlap(P[0..j-1],P) for each value of j>0. Once we've computed these values we can store them in an array and look them up when we need them.

To compute these overlap functions, we need to know for strings x and y not just the longest word that's a suffix of x and a prefix of y, but all such words. The key fact to notice here is that if w is a suffix of x and a prefix of y, and it's not the longest such word, then it's also a suffix of overlap(x,y). (This follows simply from the fact that it's a suffix of x that is shorter than overlap(x,y) itself.) So we can list all words that are suffixes of x and prefixes of y by the following loop:

```
while (x != empty) {
x = overlap(x,y);
output x;
}
```

Now let's make another definition: say that shorten(x) is the prefix of x with one fewer character. The next simple observation to make is that shorten(overlap(x,y)) is still a prefix of y, but is also a suffix of shorten(x).

So we can find overlap(x,y) by adding one more character to some word that's a suffix of shorten(x) and a prefix of y. We can just find all such words using the loop above, and return the first one for which adding one more character produces a valid overlap:

Overlap computation:

```
z = overlap(shorten(x),y)
while (last char of x != y[length(z)])
{
if (z = empty) return overlap(x,y) = empty
else z = overlap(z,y)
}
return overlap(x,y) = z
```

So this gives us a recursive algorithm for computing the overlap function in general. If we apply this algorithm for x=some prefix of the pattern, and y=the pattern itself, we see that all recursive calls have similar arguments. So if we store each value as we compute it, we can look it up instead of computing it again. (This simple idea of storing results instead of recomputing them is known as *dynamic programming*; we discussed it somewhat in the first lecture and will see it in more detail next time.)

So replacing x by P[0..j-1] and y by P[0..m-1] in the pseudocode above and replacing recursive calls by lookups of previously computed values gives us a routine for the problem we're trying to solve, of computing these particular overlap values. The following pseudocode is taken (with some names changed) from the initialization code of the C++ implementation I've made available. The value in overlap[0] is just a flag to make the rest of the loop simpler. The code inside the for loop is the part that computes each overlap value.

KMP overlap computation:

```
overlap[0] = -1;
for (int i = 0; pattern[i] != '\0'; i++) {
overlap[i + 1] = overlap[i] + 1;
while (overlap[i + 1] > 0 &&
       pattern[i] != pattern[overlap[i + 1] - 1])
    overlap[i + 1] = overlap[overlap[i + 1] - 1] + 1;
}
return overlap;
```

Let's finish by analyzing the time taken by this part of the KMP algorithm. The outer loop executes m times. Each iteration of the inner loop decreases the value of the formula overlap[i+1], and this formula's value only increases by one when we move from one iteration of the outer loop to the next. Since the number of decreases is at most the number of increases, the inner loop also has at most m iterations, and the total time for the algorithm is O(m).

The entire KMP algorithm consists of this overlap computation followed by the main part of the algorithm in which we scan the text (using the overlap values to speed up the scan). The first part takes O(m) and the second part takes O(n) time, so the total time is O(m+n).
