6.046 Recitation 10 Handout

April 25, 2008

1 Pattern Matching with Don't Cares

We are given a text $T[0 \ldots n-1]$ comprised of symbols from the alphabet $\{0, 1\}$ and a pattern $P[0 \ldots m-1]$ comprised of symbols from $\{0, 1, *\}$. The goal is to report all shifts i such that for all $j \in [0, m-1]$, $P[j]$ matches $T[i+j]$, where the $*$ character can match any character.

A naïve algorithm: Test each shift i and check if the pattern matches. Running time: $O(mn)$

1.1 An Efficient Solution [Fischer-Paterson '74]

Idea: Convert this problem into a problem of polynomial multiplication and use FFT to make it fast.

1.1.1 Reduction to Shifted Product Problem

Let's first reduce the pattern matching problem to the shifted product problem:

> Given two binary arrays $A[0 \ldots m-1]$ and $B[0 \ldots n-1]$, calculate all values of $C[i]$ such that
>
> $$C[i] = \sum_j A[j]B[i+j]$$
>
> Example:
>
> $$\begin{array}{lcl} A & = & [\ 0\ \ 1\ \ 1\ \ 0\ \ 1\ ] \\ B & = & [\ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ ] \end{array}$$
>
> $$C[0] = 0+1+0+0+1 = 2$$
> $$C[1] = 0+0+1+0+0 = 1$$
> $$C[2] = \ldots$$

To solve the pattern matching problem, we will first try to detect "Type 1" mismatches of the form where $P[j] = 0$ while $T[i+j] = 1$. (We will repeat the same process later for "Type 2" mismatches where $P[j] = 1$ while $T[i+j] = 0$.) To find Type 1 mismatches, we:

1. Convert P and T into two binary arrays A and B, respectively.
2. Solve the shifted product problem on these arrays and produce the array C.
3. $C[i]$ will contain the number of Type 1 mismatches for shift i.

Finding Type 1 mismatches ($P[j] = 0$ while $T[i+j] = 1$):

Generating A from pattern P:

- Translate 0 into 1

- Translate 1 and $*$ into 0

Generating B from text T:

- Translate 0 into 0
- Translate 1 into 1

"Truth table" for Type 1 mismatches at shift i:

$P[j]$	$T[i+j]$	$A[j]$	$B[i+j]$	$A[j]B[i+j]$
0	0	1	0	0
1 or $*$	0	0	0	0
0	**1**	**1**	**1**	**1**
1 or $*$	1	0	1	0

What changes for finding Type 2 mismatches?

We will want to invert T to get B, and then for P, 0's and $*$'s map to 0, and 1 maps to 1.

1.1.2 Reduction to Polynomial Multiplication

All that's left is solving this shifted product problem. Recall from lecture the problem of multiplying two polynomials $p(x) = \sum_i p_i x^i$ and $q(x) = \sum_i q_i x^i$ to get $u(x) = p(x)q(x)$ such that

$$u_i = \sum_j p_j q_{i-j}$$

If both p and q are of degree n, then we can use FFT to multiply them in $O(n \log n)$ time. The problem now is of converting the pattern matching problem into a problem of polynomial multiplication.

All we have to do is interpret the elements of A and B as coefficients for the polynomials p and q, respectively. The trick is to number the coefficients in B in the reverse order.

$$\begin{array}{cccccccccc} A & = & [& p_0 & p_1 & \cdots & p_{m-2} & p_{m-1} &] \\ B & = & [& q_{n-1} & q_{n-2} & \cdots & \cdots & q_1 & q_0 &] \end{array}$$

The coefficients for $u(x) = p(x)q(x)$ are $u_i = \sum_j p_j q_{i-j}$, which means we can construct the array C by looking for the equivalent ones. In particular,

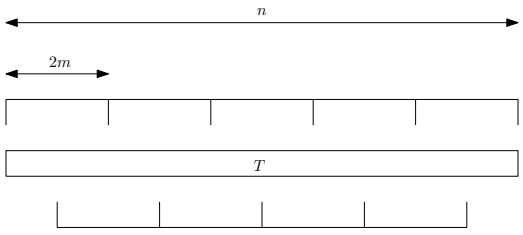
$$\begin{aligned} C[i] &= \sum_j A[j]B[i+j] \\ &= \sum_j p_j q_{n-1-i-j} \\ &= u_{n-1-i} \end{aligned}$$

so $C = [\ u_{n-1} \quad u_{n-2} \quad \cdots \quad u_m \quad u_{m-1}\ ]$

1.2 Running Time

The reduction from the pattern matching problem to the shifted product problem only takes $O(n)$. We then solve two instances of polynomial multiplication where one polynomial is of degree m and the other is of degree n, so we clearly get an upper bound of $O(n \log n)$.

We can improve this running time by breaking up T into $O(n/m)$ overlapping segments of length $2m$. This yields $O(n/m)$ subproblems, each of size $O(m)$ for a subproblem running time of $O(m \log m)$, so we get a total of $O(n \log m)$ running time.



1.3 Other Modifications

The problem as stated has the limitation that the pattern and the text have a very limited alphabet. For an arbitrary alphabet Σ of size s, we can simply represent each symbol in binary (think ASCII). What's the new running time? Does anything work differently?

The string is a factor of $O(\log s)$ longer since it will take $O(\log s)$ bits to uniquely represent a symbol in the alphabet. We no longer get an exact count of mismatched symbols since we actually count all the mismatched bits. We also don't need to and don't want to check offsets that are not on symbol boundaries. Our algorithm will calculate the values for these shifts anyway, but we should ignore them.

For the running time, we still have $O(n/m)$ subproblems. Each subproblem, however, is longer, so it takes $O(m \log s \log(m \log s))$. We can make it so that $\log s = O(m)$ by converting the text string to only have characters that are in the pattern we're trying to match in addition to a single other character that will be used for all characters not in the pattern. We don't care about losing that information because it will never match any part of the pattern. This conversion only takes linear time, so it's essentially free given what we're already doing, and it makes a new alphabet of size $s' = O(m)$. Taking $\log s'$ is also going to be $O(m)$. This means we can simplify the running time for each subproblem to:

$$\begin{aligned} O(m \log s \log(m \log s)) &= O(m \log s \log m + m \log s \log \log s) \\ &= O(m \log s \log m) \end{aligned}$$

Thus each subproblem takes $O(m \log s \log m)$, and the total running time becomes $O(n \log s \log m)$.

1.4 Side Notes

Professor Indyk actually achieved an $O(n \log m)$ running time (no dependence on the alphabet size!) using a randomized algorithm, FOCS '98. A deterministic algorithm with the same running time was later discovered by Cole-Hariharan, STOC '02.

2 Set Sum Problem

Given two binary arrays A and B which each have n integers in the range $[0, 10n]$. We want to find the set $C = \{z = x + y | x \in A, y \in B\}$. We can do this by constructing new arrays

$$A'[i] = \begin{cases} 1 & \text{if } i \in A \\ 0 & \text{otherwise} \end{cases} \text{ and } B'[i] = \begin{cases} 1 & \text{if } i \in B \\ 0 & \text{otherwise} \end{cases}$$

Both these arrays have length $10n = O(n)$. If we use each entry as a coefficient, i.e. $p_i = A'[i]$ and $q_i = B'[i]$, we can use FFT to compute the product of $p(x)q(x)$ and make $C'[i] = \sum_j p_j q_{i-j}$, which will be the number of existing pairs of items in A and B for which their sum is i. We can do the transformations in $O(n)$ time, the polynomial multiplication in $O(n \log n)$ time, and the linear search for all nonzero entries in C' in $O(n)$ time for a total of $O(n \log n)$.