Lecture 7

Approximate string matching

We want to find approximate occurrences of the pattern in the text. We need to define what an "approximate" occurrence is. In other words, "errors".

Mismatches

```
though
 |   |   2 mismatches  (Hamming distance)
trougf
```

The *k-mismatches* problem: Given P, T, and k (integer), find all occurrences of P with at most k mismatches in T.

Notice that the naive string matching algorithm works for the k-mismatches problem – $O(mn)$ time.

How about this example?

```
programming
programing
```

If we count only mismatches, there are many. But we know there is one mistake, which is a deletion. So we consider deletion errors and insertion errors, as well. That is, there are three kinds of errors: Mismatch, Insertion, Deletion (these are called "differences").

How many errors? (3)

```
a b c d e f g
|   |  / /  |     (Levenshtein distance)
a h c e f i g
```

The *k-differences* problem: Given P, T, and k (integer), find all occurrences of P with at most k differences in T.

$O(mn)$ algorithm is not obvious. (Our goal is $O(kn)$ algorithm.)

Edit Distance

Before we tackle the k-differences problem, we consider a simpler problem.

The *edit distance* between two strings X and Y is the minimum number of differences between them (or the minimum number of edit operations that transform X into Y). See the previous example.

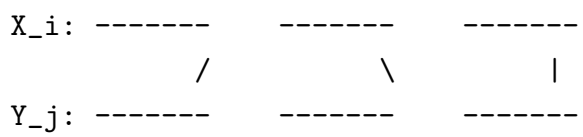
The edit distance problem is to find the edit distance between X and Y.

Let $m = |X|$ and $n = |Y|$. Assume that $X[m+1] = \$$ and $Y[n+1] = \#$ (these characters always cause mismatches).

Let $D[i, j]$ be the edit distance between X_i (prefix of X having i characters) and Y_j.

$D[i, 0] = i$ (i deletions), $D[0, j] = j$ (j insertions)

Three cases for $D[i, j]$, $i, j \geq 1$



$$D[i, j] = \min(D[i, j-1] + 1, D[i-1, j] + 1, D[i-1, j-1] + 0/1)$$

```
    a h c e f i g
  0 1 2 3 4 5 6 7
a 1 0 1 2 3 4 5 6
b 2 1 1 2 3 4 5 6
c 3 2 2 1 2 3 4 5
d 4 3 3 2 2 3 4 5
e 5 4 4 3 2 3 4 5
f 6 5 5 4 3 2 3 4
g 7 6 6 5 4 3 3 3
```

So we find the edit distance $D[m, n]$ in $O(mn)$ time. The path from (m, n) to $(0, 0)$ shows the way $D[m, n]$ is obtained.

The k-Differences Problem

Apply the edit distance algorithm to $n - m + 1$ positions of the text? Text lengths of occurrences are not always m because of insertions and deletions.

For each position i of the text, run the edit distance algorithm with $P[1..m]$ and $T[i..i + m + k]$. Time: $O(m^2 n)$.

Some approximate occurrence are detected several times: if $T[i..j]$ has distance 2 ($< k$), then $T[i - 1..j]$ has distance at most 3 ($\leq k$).

Let $D[i, j]$ be the minimum number of differences between P_i and any suffix of T_j.

$D[i, 0] = i$ (i deletions), $D[0, j] = 0$ (empty prefix of P = empty suffix of T_j)

The same recurrence

$$D[i, j] = \min(D[i, j-1] + 1, D[i-1, j] + 1, D[i-1, j-1] + 0/1)$$

Example: $T = abbdadcbc$, $P = adbbc$, $k = 2$.
P occurs at positions $3, 4, 7, 8, 9$.

		a	b	b	d	a	d	c	b	c
	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	0	1	1	1	1
d	2	1	1	2	1	1	0	1	2	2
b	3	2	1	1	2	2	1	1	1	2
b	4	3	2	1	2	3	2	2	1	2
c	5	4	3	2	2	3	3	2	2	1

Show the path from (5,7) to (0,4).

$D[m, j] \leq k$ iff P occurs at end position j of T with at most k differences. We solved the k-differences problem in $O(mn)$ time.

Ukkonen's Algorithm

Ukkonen's algorithm for the edit distance problem

Look at diagonal $d = j - i$. Entries are increasing by 1.

Lemma 1 *$D[i, j] = D[i-1, j-1]$ or $D[i, j] = D[i-1, j-1] + 1$.*

Idea: go along diagonals in the D array and compute the locations of value 0's, value 1's, ... until (m, n) is reached.

If the given distance k is small, $O(kn)$ time. In the worst case $O(mn)$ time. (There is a lower bound.)

For a diagonal d and a difference e, let $C[e, d]$ be the largest column j such that $D[j - d, j] = e$. In other words, the entries of value e on diagonal d end at column $C[e, d]$.

Previous example.

```
      -4 -3 -2 -1  0  1  2  3  4
-1              -9 -1 -9
 0           -9 -1  1  0 -9
 1        -9 -1  1  3  2  1 -9
 2  -9 -1  1  5  4  4  3  2 -9
 3  -1  1  5  6  7  5  5  4  3
```

The size of C array: $O(k^2)$

There are two problems, depending on whether k (differences) is given or not. First, the case k is given.

```
procedure Ukk(X,Y)
  for d = 0 to k+1 do
    C[d-1,d] = d-1; C[d-2,d] = -infi
  od
  for d = -(k+1) to -1 do
    C[|d|-1,d] = -1; C[|d|-2,d] = -infi
  od
  for e = 0 to k do
    for d = -e to e do
      col = max(C[e-1,d-1]+1, C[e-1,d]+1, C[e-1,d+1])
      while X[col+1-d] == Y[col+1] do
        col = col + 1
      od
      C[e,d] = col
    od
  od
```


Assume that $C[e-1, d-1]$, $C[e-1, d]$, $C[e-1, d+1]$ were computed correctly. It means that entries of value $e-1$ reach column $C[e-1, d-1]$ on diagonal $d-1$, ...

$$col = \max(C[e-1, d-1]+1, C[e-1, d]+1, C[e-1, d+1])$$

```
entries of value e-1: marked by *
automatic values of e: marked by .      . * * * .
                                          . * * * .
                                            . * . * .
                                              . * . * .
                                                . . . * .
                                                      . .
```

$D[col-d, col]$ get value e from one of the last entries of value $e-1$ on diagonals $d-1$, d, and $d+1$ by one of the three types of differences.

- Time $O(k \min(m, n))$: $2k + 1$ diagonals are computed. For each diagonal the number of comparisons is at most its length in table D, which is $\min(m, n)$.
- If the number of consecutive matches in a diagonal can be computed in constant time (suffix tree), it is $O(k^2)$ time – later.

The case k is not given (i.e. the edit distance problem)

- Run procedure Ukk until we reach (m, n). Let t be the edit distance between X and Y. Time $O(t \min(m, n))$.

Modification of Ukk for k-differences problem

- Compute table C corresponding to table D of k-difference problem. The number of diagonals is approximately $n + k$. For each diagonal at most m comparisons. So still $O(mn)$ time.
- If consecutive matches are found in constant time, $O(kn)$ time for the k-differences problem

The k-mismatches problem

- If consecutive matches are found in constant time, for each text position, find consecutive matches k times. $O(kn)$ time.

Summary

	D	C	suffix tree
Edit distance	$O(mn)$	$O(kn)$	$O(k^2)$
k-differences	$O(mn)$	$O(mn)$	$O(kn)$
k-mismatches	naive $O(mn)$		$O(kn)$

The Longest Common Subsequence Problem

- A is a *subsequence* of B if A is obtained by deleting zero or more symbols from B.
- C is a *common subsequence* of A and B if C is a subsequence of A and also a subsequence of B.

Problem: Given two strings X and Y, find a longest common subsequence of X and Y.

Relationship to the edit distance problem: here count matches, in edit distance count differences.

Examples

```
a b c d e f g
|   |   / /   |
a h c e f i g
```

edit distance: 3, lcs: 5 (dual of edit distance)


```
a b c d
d e f g
```

edit distance: 4, lcs: 1 (not dual)

Consider the edit distance problem where insertion and deletion costs are 1, and the mismatch cost is 2 (i.e., delete and insert). Then this problem is dual to LCS.

$$\mathrm{ed}(X, Y) = |X| + |Y| - 2 \cdot \mathrm{lcs}(X, Y)$$

- 1st example: $\mathrm{ed}(X, Y) = 4$, $|X| = |Y| = 7$, $\mathrm{lcs}(X, Y) = 5$.
- 2nd example: $\mathrm{ed}(X, Y) = 6$, $|X| = |Y| = 4$, $\mathrm{lcs}(X, Y) = 1$.

Unix diff command: each line is a symbol. Exactly the LCS problem.

Let $L[i, j]$ be the length of LCS of X_i and Y_j.

$L[i, 0] = 0$ for all i, $L[0, j] = 0$ for all j.

Recurrence:

$$\text{if } X[i] = Y[j] \quad \text{then} \quad L[i, j] = L[i-1, j-1] + 1$$
$$\text{else} \quad L[i, j] = \max(L[i, j-1], L[i-1, j]).$$

		a	b	b	a	b	a
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
a	0	1	1	1	2	2	2
b	0	1	2	2	2	3	3
a	0	1	2	2	3	3	4
b	0	1	2	3	3	4	4

LCS : aaba, abab

An algorithm similar to the edit distance problem – $O(mn)$ time.

Hunt-Szymanski Algorithm

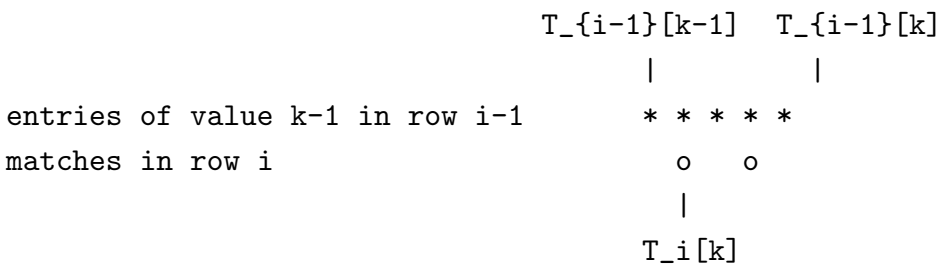
Let r be the total number of matching pairs between X and Y. Note that $0 \leq r \leq mn$. In diff command, $r \leq \min(m, n)$.

Idea: Each row of the L array is increasing by 1. Go down row by row, and compute $T_i[k]$ for row i which is the smallest j such that $X[1..i]$ and $Y[1..j]$ contains a common sequence of length k.

	[0]	[1]	[2]	[3]	[4]
T_0	0				
T_1	0	1			
T_2	0	1	4		
T_3	0	1	2	5	
T_4	0	1	2	4	6
T_5	0	1	2	3	5

The values of T_i for each i are strictly increasing.

Assume that T_{i-1} has been computed. Now compute T_i.



$T_i[k]$ = smallest j such that $X[i] = Y[j]$ and
$T_{i-1}[k-1] < j \leq T_{i-1}[k]$.


```
T[0] = 0
T[1..n] = infi (n+1)
for i = 1 to m do
  for j from largest to smallest such that X[i]=Y[j] do
    find k such that T[k-1] < j <= T[k]
    T[k] = j
  od
od
print largest k such that T[k] != infi
```

The direction of j is important because we use only one array T.

How to implement T

- array: to find k, use binary search – $O(\log n)$ time
- balanced search tree – $O(\log n)$ time
- van Emde Boas data structure when the universe is small integers – $O(\log \log n)$ time

Total time – $O(r \log n)$ if an array is used.

Preprocessing: find all matches

- If we know the alphabet and we can use characters as indices (not true in diff command), create lists by scanning Y in $O(n)$ time.

```
Y : a b b a b a  ->  M[a] : 6 4 1
                     M[b] : 5 3 2
```


- If the alphabet size is infinite (as in diff), we will make the following lists in $O(n \log n)$ time.

```
M[1] : 6 4 1
M[2] = M[1]
M[3] : 5 3 2
M[4] = M[1]
M[5] = M[3]
```

1. For each sequence, make pairs (char,position) and sort.

   ```
   Y : abbaba -> (a,1) (a,4) (a,6) (b,2) (b,3) (b,5)
   X : aabab  -> (a,1) (a,2) (a,4) (b,3) (b,5)
   ```

2. Merge two sorted lists creating the M lists as we go.

To recover LCS in reverse, maintain pointers among matching pairs – $O(r)$ space