String Matching

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2012 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Introduction

The **string matching** problem is defined as follows. For a text $T[..]$ of length n and a pattern $P[..]$ of length m and the characters of T and P drawn from a finite alphabet Σ, we say that the pattern P starts at position $s+1$ in text T if $T[s+1..s+m]$ is equal to $P[1..m]$ (or to $P[0..m-1]$ if indices start from 0 rather than 1).

A string of length 0 is called the null string and represented by ϵ. A substring of a string S is a string $S[i...j]$, where $j \geq i$. A substring $S[i..j]$, with $i > j$ is considered to be the null string. The length of string S is denoted by $\mid S \mid$. For two strings x and y the concatenation of x and y is string xy of length $\mid x \mid + \mid y \mid$. A string w is a prefix of x if $x = wy$. A string q is a suffix of a string r if $r = tq$.

The naive way for string matching works as follows.

```
  NaiveMatching(P[0..m-1],T[0..n-1])
1. for (i=0;i<=n-m;i++) {// i indicates T[i...] is checked against P[..]
2.  j=0; // runs through the pattern
3.  while ((j<m) && (T[i+j] == P[j])) {
4.   j++;
    }
5.  if (j==m) // a match has been found
6.    return(i); //T[i..i+m-1] is P[0..m-1]
   }
7. reurn(-1); //-1 means no match.
}
```

Namely, in the NaiveMatching algorithm the pattern is overlayed with the text for each starting position of the text (indexed by i). For each such overlay we compare the pattern to the text starting at position i. This is performed through lines 2-4. If a match is found $j == m$ is true and the algorithm returns with the index of the match (starting at i). Otherwise i is incremented by 1 and the algorithm continues. When the for..loop exits no match exists and no match (-1) is returned.

The running time of the algorithm is $O(nm)$ as the loop of line 1 is executed $O(n)$ times and the while loop of lines 3-4 $O(m)$ times.

(c) Copyright A. Gerbessiotis. All rights reserved.

Rabin-Karp algorithm

The Rabin-Karp algorithm is another approach to the matching problem, performs quite well in practice and can also be generalized for two dimensional (i.e. image) matching. Although its worst case running time is $O(nm)$ its average case running time is quite good (Do not forget that in computer science the most practical algorithms are not optimal in terms of worst case performance: hashing and quick-sort are some notorious examples).

The algorithm uses the notion of "hashing" and elementary number-theoretic notions. Suppose that the characters come from an alphabet Σ. We view each character as a digit in radix-d notation where $d =\mid \Sigma \mid$. For example a character from $\Sigma = \{a, b, \ldots, z\}$ can be viewed as a digit of radix-26. Therefore the text T and pattern P can be viewed as an n-digit and m-digit number respectively. For the rest of this discussion we are going to make the assumption that text and pattern are strings of decimal digits.

Therefore for a pattern $P[..]$ of length m we denote by p its decimal value. For a text $T[..]$ we denote by t_i the decimal value of the length m substring starting at i, i.e. of the string $T[i..i+m-1]$. Therefore the pattern matching problem is reduced to deciding whether there exists an index j such that $p = t_j$. If we could compute p in $O(m)$ time and the n numbers t_i in $O(n)$ time, then we could solve the string matching problem in additional $O(n)$ time. We note that

$$p = P[m-1] + 10P[m-2] + \ldots + P[0]10^{m-1}$$

can be computed using Horner's rule in $O(m)$ time as p is the value of the corresponding polynomial at $x = 10$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Rabin-Karp algorithm continued

As soon as t_0 is computed for $T[0..m-1]$ in a similar way the sequence of $t_1, t_2, \ldots$, can be computed in $O(n-m)$ time by the following recurrence relation that computes t_{i+1} from t_i, i.e. the decimal representation of $T[i+1..i+m]$ from $T[i..i+m-1]$.

$$t_{i+1} = 10(t_i - 10^{m-1}T[i]) + T[i+m]$$

Subtraction of the $10^{m-1}T[i]$ term removes the contribution of the $T[i]$ from the computation of the representation $T[i+1..i+m]$, the multiplication by 10 "shifts" the $T[i+1..i+m-1]$ one position to the left and the last term adds to that the contribution of the next term $T[i+m]$ thus giving the decimal representation of $T[i+1..i+m]$ as required. Given that 10^{m-1} can computed once (precomputed) and used forever, the reccurrence for t_{i+1} takes constant time.

The only problem with this approach is that t_i in general can be a very long integer and the assumption of constant time for the computation of t_i may make no sense at all. We can eliminate this problem if we perform all computations modulo q , where q is a very large prime number. In practice we may choose q large enough such that $10q$ can be represented by a single words (for example in C/C++ $10q$ is less than the largest unsigned 32-bit int which is $2^{32}-1$).

If instead of using a decimal alphabet, we have characters in radix-d, the recurrence above is modified accordingly to reflect the cardinality (size) of the alphabet Σ.

$$t_{i+1} = (d(t_i - (d^{m-1} \bmod q)T[i]) + T[i+m]) \bmod q.$$

The problem of using modulo arithmetic is that finding a $t_i \bmod q = (p \bmod q)$ does not mean that $t_i = p$. This may or may not be true. If we have such a "spurious hit" we can then perform the check $T[i..i+m-1] = P[0..m-1]$ to check whether $t_i = p$ indeed.

(c) Copyright A. Gerbessiotis. All rights reserved.

Rabin-Karp algorithm continued

```
   Rabin-Karp (T,P,d,q)
1. h = d**(m-1) mod q;
2. p = 0;
3. t[0] = 0;
4. for(j=0;j<m;j++) {
5.   p= (dp+P[j]) mod q;
6.   t[0]=(dt[0]+T[j])mod q;
7. }
8. for(i=0;i<=n-m;i++){
9.   if (p == t[i]) {
10.    k=0; // runs through the pattern
11.    while ((k<m) && (T[i+k] == P[k])) {
12.      k++;
13.    }
14.    if (k==m) // a match has been found
15.       return(i); //T[i..i+m-1] is P[0..m-1]
16.  }
17.  if (i < n-m)
18.   t[i+1] = (d(t[i]-T[i]h)+T[i+m]) mod q;
19 }
```

In the worst case Rabin-Karp is $\Theta(nm)$. However, on the average as long as $q > m$ we expect the running time of Rabin-Karp to be $O(n + m)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Overviewing Rabin-Karp we point out the main idea of the algorithm is the idea of fingerprinting (hashing).

Problem. In the Rabin-Karp algorithm pattern $P = P[0..m-1]$ is matched to text $T = T[0..n-1]$ by creating a finger print of the pattern P to be denoted by p and fingerprints $t_i = T[i..i+m-1]$ of text T and thus matching first p to t_i for all $i = 0, \ldots n-m$. If $p \neq t_i$ there can be no match of P at position i. If they are equal there may be a match but this requires a comparison of the actual pattern $P[0..m-1]$ to the actual text $T_i = T[i..i+m-1]$; note this latter comparison is not a fingerprint comparison but character to character comparison.

Simple FingerPrints. We describe what the fingerprints look like in Rabin-Karp's algorithm. Let us assume for the sake of simplicity that both the text and the pattern are binary 0-1 strings. The pattern P atains a fingerprint $p = P[m-1] + 2P[m-2] + \ldots + 2^{m-1}P[0]$. Similarly, fingerprint t_i is given by. $t_i = T[i+m-1] + \ldots + 2^k T[i+m-1-k] + \ldots + 2^{m-1}T[i]$. Note that $t_{i+1} = 2(t_i - 2^{m-1}T[i]) + T[i+m]$. Therefore starting with t_0 that can be computed with $O(m)$ computational operations, and progressing inductively, if we know t_i, then t_{i+1} can be found in $O(1)$ time with operations that involve subtraction/addition and binary shifting left/right. Thus establishing all t_i for all i requires $O(n+m)$ time; $O(m)$ is the contribution of finding t_0. In addition $O(m)$ time is required to find p. Total remains $O(n+m)$.

If we compute fingerprints this way, the t_i can grow large if m is large (say larger than 32 bits). Other than that the problem of matching strings has been reduced to the problem of matching fingerprints.

Elaborate Fingerprints. As fingerprints can grow large and can't possibly fit into a single word we deal with this issue by performing all operations mod q where q is a large prime number. Thus the update formula for text fingerprints becomes

$$t_{i+1} = 2(t_i - (2^{m-1} \bmod q) * T[i]) + T[i+m] \bmod q.$$

Note that $(2^{m-1} \bmod q$ is a constant and can be precomputed just once. If q, or $2q$, is kept smaller than the maximum integer available, then the update operations cause no overflows. A problem however that appears with mod$q$ calculations is whether $t_i = p$ really indicates a match or a false match also known as false positive or **spurious match/hit**. For spurious matches/hits we need to go back to the actual text and compare P against $T[i...i+m-1]$ for matching.

(c) Copyright A. Gerbessiotis. All rights reserved.

Non-binary Fingerprints. The discussion of binary fingerprints can be generalized to any alphabet; Page 3 or 4 show this for decimal and general alphabets.

How frequent are false positives? For mod$q$ operations, we have two matches: (a) **real match** when $t_i = p$ and $T_i = P$, and (b) **a spurious match/hit** when $t_i = p$ but $T_i \neq P$. We naturally have misses/mismatches when $t_i \neq p$.

In practice we choose q so that it is a prime and such that $q = Am \ln Am$, for some A. Why q should be so and how one can choose A will become apparent later in this discussion, and in a relevant homework.

$$Pr(\text{spurious match at } i) \quad = \quad Pr(p \bmod q = t_i \bmod q)$$

What is the probability that $p \bmod q = t_i \bmod q$?

(c) Copyright A. Gerbessiotis. All rights reserved.

If the pattern P is relatively long (say more than 4-5 characters) and the alphabet is reasonably large (i.e. strings are non-binary strings), then an algorithm due to R.S. Boyer and J.S.Moore is perhaps the most efficient string matching algorithm.

The Boyer-Moore algorithm differs from the brute-force naive algorithm in that it checks the pattern against the text not left-to-right but right-to-left. If a mismatch is found the action performed on the pattern differs from the approach of the naive algorithm (i.e. shift pattern one position to the right). In the Boyer-Moore algorithm, if the mismatched character $c = T[i]$ of the text that was compared to $P[j]$ does not appear in the pattern P, then the pattern is shifted right j positions, and thus a comparison of $P[..]$ against $T[i+1..i+m]$ will then be performed. This is because c does not appear in P and therefore no character in the portion of the pattern $P[..j-1]$ will ever match $T[i]$ and thus the portion of the text $T[..i]$ can be skipped.

If however, c appears in $P[..j-1]$, we shift P to the right by an appropriate number of positions. Let t be the maximum index in $..j-1$ that a c appears. Then we shift P to the right $j-t$ positions, so that $c = P[t]$ matches $T[i] = c$. If c does not appear in $P[..j-1]$ but it does appear in $P[j+1..]$, then and only then do we shift right one position only (i.e. Boyer-Moore behaves like the naive algorithm).

The actions performed due to the mismatched c describe a method that is known as **bad-character heuristic** or the **character-jump**. How do we compute the last occurrence of a character c in P? We use function BadCharacter (or Compute-Last-Occurrence-Function in CLR, first edition). Array last computes the rightmost occurrence of a character of the alphabet Σ.

```
  BadCharacter(P,m,Alphabet)
1. for(i=0;i<|Alphabet|;i++)
2.  last[i]= Left-1;          //Left=0, Right=m for    P[0..m-1] (C/C++)and
3. for(j=Left;j<Right;j++){   //Left=1, Right=m+1 for  P[1..m]   (Pascal)
4.   last[P[j]]=j;
5. return(last[]);
```

The running time of the algorithm is $O(m+ \mid \Sigma \mid)$ and thus can be performed quite efficiently.

The algorithm

The Boyer-Moore algorithm is described below.

```
   Boyer-Moore(P[0..m-1],T[0..n-1],Alphabet)
0. last[] = BadCharacter(P,m,Alphabet);
1. i=m-1; // runs through the text
2. j=m-1; // runs through the pattern
3. while (i <= n-1) {
4.  if (P[j]==T[i]) {
5.   if (j==0)
6.     return("Match at i");
7.   else {
8.     i--;
9.     j--;
10.  }
11. }
12. else { //mismatch P[j] differs from c=T[i]
13.  i=i+m- MIN(j, 1+last[T[i]]);
14.  j=m-1; //restart from right end of P
15. }
16.}
17.return("no match found");
```

Why step 13 is performed the way shown there becomes clearer by the example cases below

(c) Copyright A. Gerbessiotis. All rights reserved.

```
INITIAL STATE
       T[i-j  ]             c = T[i]                 T[i-j+m-1]
                                !=   =  =  =  =  =  =   =
       P[0]                     P[j]                 P[m-1]
                                 <--Right to Left comparison in BoyerMoore

***********************************
CASE 1.  c=T[i] is nowhere in P[0...m-1]
***********************************

ACTION: Shift P to the right j positions so that

        c=T[i]   T[i+1]            T[i+m]
                   ?   ?  ?   ?  ?   ?      RESTART from right of P.
                 P[0]              P[m-1]

  Last[c] = -1 since c is nowhere in P

  Reset i  :  i = i+m = i+m - MIN(j,0) = i+m -  MIN(j,Last[c]+1)
  Reset j  :  j = m-1
```

```
**********************************
CASE 2. c=T[i] is in P[0..j-1]  and t= Last[c] < j (i.e. P[t]=c)
**********************************
ACTION: Shift P j-t = j-Last[c] positions to the right
        Since j-t>0 => j>t => j >= t+1 => MIN(j,t+1)=t+1=Last[c]+1

        T[i-t]        c=T[i]            T[i-t+m-1]
                       =                          RESTART
        P[0]          c=P[t]            P[m-1]

  Reset i  :  i = i -t +m -1 = i+m - (t+1) = i+m - MIN(j,t+1)
  Reset j  :  j = m-1

**********************************
CASE 3. c=T[i] is in P[j+1...m-1] i.e  and j+1 <= t=last[c]<=m-1
**********************************
ACTION: Shift P to the right one position
        Since t=last[c]>=j+1 => t+1 >= j+2 i.e. MIN(j,t+1)=j.

        T[i-j+1]      c=T[i]            T[i+m-j]
                       =                       RESTART
        P[0]          c=P[j-1]          P[m-1]

  Reset i  :  i = i+m-j= i+m - MIN(j,t+1)
  Reset j  :  j = m-1
```

The best case for Boyer-Moore is an all "a" text and an all "b" pattern. Boyer-Moore will only compare n/m characters, before failure is declared.

What is a worst case text/pattern senario in Boyer-Moore? In the worst-case Boyer-Moore is an $O(nm)$ time algorithm.

Can we improve the performance of Boyer-Moore?

Introduction

The Knuth-Morris-Pratt algorithm is a linear time algorithm (in the worst-case). Note that Boyer-Moore as described before may not be linear in certain cases (e.g. binary stings of short length). For every pattern some precomputation needs to be performed in the KMP algorithm. KMP works left-to-right as opposed to BM that works right to left.

The precomputation involves the computing of the failure function f, that is used when the comparison of $P[j]$ to $T[i]$ fails, and the pattern/text need to be shifted. This means that although $P[..j-1]$ matches $T[..i-1]$, $P[j] \neq T[i]$. Let us call P_k the prefix of P consisting of the first k characters of P and P_{-k} the suffix of P consisting of the last k characters of P.

The question that arises then is by using the lessons learned from the BM algorithm how to shift P to the right by more than one position. If we shift right $P[]$ by say d positions we have the following case.

(c) Copyright A. Gerbessiotis. All rights reserved.

```
(a)                                 i
T:                          abcdefghX
                            0       j     m-1
P:                          ABCDEFGHYIJKLMN
A mismatch occurs at j : This means that a=A b=B   ...  h=H,  and X != Y
-------------------------------------------------------------------------
This is equivalent to having,
(b)                                 i
T:                          ABCDEFGHX
                            0       j     m-1
P:                          ABCDEFGHYIJKLMN
-------------------------------------------------------------------------
If we shift P to the right d=4 positions we have
(c)                                 i
T:                          ABCDEFGHX
P:                              ABCDEFGHYIJKLMN
                                    j
For the shift to be helpful:  E=A, F=B, G=C, H=D, X=E.
```

From cases (a) and (b) a mismatch at position j of the pattern indicates that $P[0..j-1]$ appears in the text just before position i. Therefore a d shift makes sense if a prefix $P[0..k-1]$ of P (like ABCD in Case 3) for some k, is a suffix of $P[1..j-1]$.

Therefore if a mismatch or failed match at position j occurs, the algorithm first reviews the failure function $f[j-1]$ to figure out whether there is a prefix of P that is a suffix of $P[1..j-1]$. If $f[j-1]$ maintains the length of the longest prefix of P that is a suffix of P_{j-1} we are done. We shift P right $j - f[j-1]$ positions, and we continue our string matching at position j as before.

(c) Copyright A. Gerbessiotis. All rights reserved.

```
   KMP(P,T)
1.  f[]=LongestPrefix(P);
2.  i=0;
3.  j=0;
4.  while (i <= n-1) {
5.    if P[j] == T[i] {
6.     if (j==m-1){
7.      return("match at i-m+1");
8.     }
9.     i++; //P[j]==T[i] advance i
10.    j++; //     and j
11.   }
12.   else { // mismatch at j but P[0..j-1] matches T[..i-1]
13.    if (j>0)  // try to advance pattern a little
14.     j=f[j-1];
15.    else     //  no luck; advance text one character to the right.
16.     i++;
17.   }
18.  }
19. return("no match");
```

How do we compute array $f[]$? Element $f[j]$ is the length of the longest prefix $P[0..]$ of P that is a suffix of $P[1..j]$. For the boundary case $f[0] = 0$.

For example let $P = abababacba$. Then $f = [0, 0, 1, 2, 3, 4, 5, 6, 0, 1]$.

We find $f[]$ by using KMP but we match the pattern against a text which is the pattern P! The array f can thus be computed as follows.

Failure Function Computation

```
   LongestPrefix (P);
1.  f[0]=0; // boundary case
2.  i=1; // i-j >=1
3.  j=0;
4.  while (i <= m-1) {
5.    if P[j] == P[i] { //P[0..j] matches P[i-j..i]
6.     f[i]=j+1;        // j+1 matched characters
7.     i++; //P[j]==T[i] advance i and j
8.     j++; //     maintains i-j >=1
9.    }
10.   else { // mismatch at j but P[0..j-1] matches P[i-j..i-1]
11.    if (j>0)
12.     j=f[j-1];
13.    else {    //  no luck; no match
14.     f[i]=0;
15.     i++;
15a.   }
16.   }
17.  }
18. return(f[]);
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Time analysis

The analysis of the Knuth-Morris-Pratt algorithm consistst of examining the three possible cases of matches/mismatches and also examine how the difference $i - j$ behaves. Let us call $D = i - j$, i.e. the difference in shift between the text and the pattern. Naturally, $D \leq n$.

The three cases that we have are as follows.

Case 1. If $T[i] == P[j]$ then, i increases by one and j increases by one and D remains the same $D = (i + 1) - (j + 1) = i - j$.

Case 2. If $T[i] \neq P[j]$ and $j > 0$, then i remains the same, j however changes from j into $f[j - 1]$ a decrease by $j - f[j - 1]$. Then D changes from $i - j$ into $i - f[j - 1]$. Since $f[j - 1] < j$, this means that D is increased by $j - f[j - 1]$.

Case 3. If $T[i] \neq P[j]$ and $j == 0$, then i increases by 1 and j remains the same. This leads to $D = (i + 1) - j$, i.e. an increase by 1 in the value of D.

As a conclusion,

- i increases by at least one in cases 1 and 3.
- D increases by at least one in cases 2 and 3.

Therefore at every iteration of the loop, either i or D increases by at least one. The maximum value for either i or D is n. Therefore the loop of KMP is executed at most $2n$ times.

In order to complete the analysis we need to examine the running time of line 1. By a similar argument it is $O(2m) = O(m)$. Therefore KMP has running time $O(n + m)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Approximate String Matching: Match P with T with at most k errors. The string operations allowed are Insert, Delete, and Replace (a character with another one).

- **Dynamic Programming** Requires $O(nm)$ time (that of brute-force). Build a matrix $C[m, n]$ to record information about a prefix $P[1..i]$ of the pattern to a suffix of the text $T[1..j]$ at position j and record this information at $C[i, j]$. A match is reported if $C[m, j] \leq k$.
  - $C[0, j] = 0$. This is obvious for an empty P.
  - $C[i, 0] = i$. If P has i characters, delete them all to get an empty T.
  - $C[i, j] = C[i-1, j-1]$ if $P_i = T_j$. If the last character of $P[1..i]$ matches the last character of $T[1..j]$ then try to determine the distance between $P[1..i-1]$ and $T[1..j-1]$.
  - $C[i, j] = 1 + \min(C[i-1, j], C[i, j-1], C[i-1, j-1])$ otherwise i.e. $P_i \neq T_j$. Otherwise, try to find the edit distance of $P[1..i-1]$ to $T[1..j]$ and insert $P[i]$ to the former, or find the edit distance of $P[1..i]$ to $T[1..j-1]$ and insert $T[j]$ or find the distance of $P[1..i-1]$ and $T[1..j-1]$ and relabel $P[i]$ to $T[j]$.
- Nondeterministic finite automaton construction. Build a NDFA with $km+k$ states (i.e. k rows of a deterministic automaton that recognizes the input string plus one step per row as an accepting state for a given number of errors).

T=survey P=surgery

m=6 n=7

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Approximate String matching overview

An alternative formulation of a solution.

```
 1. Set n to the length of P                                          T=survey  P=surgery
 2. Set m to the length of T                                            m=6       n=7
 3. If n==0 return m                                                        s  u  r  g  e  r  y
 4. if m==0 return n                                                    0   1  2  3  4  5  6  7
 5. Construct matrix below                                          s   1   0  1  2  3  4  5  6
 6. Initialize row 1 to 0...n-1                                     u   2   1  0  1  2  3  4  5
 7. Initialize column 1 to 0..m-1                                   r   3   2  1  0  1  2  3  4
 8. Examine each character of P                                     v   4   3  2  1  1  2  3  4
 9. Examine each character of T                                     e   5   4  3  2  2  1  2  3
10. if P[i]= T[j]  cost=0                                           y   6   5  4  3  3  2  2  2
      else         cost=1                                                                    ^
11. Set cell C[i,j] as the minimum of                                 m+1 rows, n+1 columns   |
    . cell above plus 1  (i.e. C[i-1,j]+1)                                                  |
    . cell left  plus 1  (i.e. C[i,j-1]+1)                                                  |
    . cell diagonally left up plus cost: (i.e. C[i-1,j-1]+cost)                             |
12. Repeat.                                                                                 |
13  Edit distance is C[m,n] ------------------------------------------------------------------
```

(c) Copyright A. Gerbessiotis. All rights reserved.