String Matching Algorithms

By:-
Shiranchal Taneja
sxt9088

Talk about..

1. Introduction
2. Naïve Algorithm
3. Overview of Rabin-Karp
4. Overview of Knuth-Morris-Pratt
5. Running times
6. References

Introduction

- Why do we need string matching?
  - String matching is used in almost all the software applications straddling from simple text editors to the complex NIDS.
  - "Find and replace all" in text editors.
  - Network Intrusion Detection Systems (NIDSs)
    - Main functioning i.e. signature matching is based on string matching.
  - Therefore, efficient string matching algorithms can greatly reduce response time of these applications

String matching

- To find all occurrences of a pattern in a given text.
- We can formalize the above statement by saying: Find a given pattern p[1..m] in text T[1..n] with n>=m.

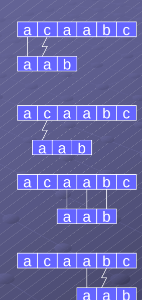
*text is the string that we are searching.
*pattern is the string that we are searching for.
*Shift is an offset into a string.

Naïve algorithm

- Naïve algorithm finds all valid shifts using an iteration that compare pattern for each of the possible n-m+1 values of shift.

Naïve algorithm

It seems like sliding a "template" containing the pattern over the text, and looking for match.



- S = 0
- S = 1
- S = 2
- S = 3

Naïve algorithm

- NaiveEg.c

```
long naivesearch ( unsigned char *P, long M, unsigned char *T,
    long N)
{   long i,j;
    for ( i =0 ; i<= N-M; i++)
    {           for (j=0; j<M; j++)
                {           if (P[j] != T[i + j]  break;   }
                if (j == m ) return (i) ;
    }
    return (-1);
}
```

For each (n-m+1) possible values of shift s, inner loop runs for M times. Thus

<u>Θ ((n-m+1)m)</u> => <u>Θ(n2)</u>

Rabin – Karp algorithm

- String matching algorithm that compares string’s hash values, rather than string themselves.
- Performs well in practice, and generalized to other algorithm for related problems, such as two-dimensional pattern matching.
- Worst case running time is O((n-m+1)*m”)
- For efficiency, the hash value of the next position in the text is easily computed from the hash value of the current position.

Rabin - Karp…Contd

- Example of removing and shifting the elements of the array.
- Let T=“123456” and m =3
- T(0) = 123
- First: remove the first digit: (123 – 100*1) = 23
- Second: Multiply by 10 to shift it: 23*10 = 230
- Third: Add last digit: 230 + 4 = 234, which is T(1)
- The algorithm runs by comparing, t (s) with p.
  - When t(s) = p, then we have found the substring P in T, starting from position s.
  - Problem: t(s) and p may be too large, therefore no built-in data type can fit them.
  - Solution: All t(s) and p be performed in modulo q.

Knuth-Morris-Pratt algorithm

- It ensures that a string search will not require more than N character comparison (once some pre-computation is performed)
- e.g. pattern “AAAB” is searched in “AAAXAAAAA”.

  Naïve algorithm will ok till “B” in the pattern fails to match the 4th char of the text. At this point, it will shift the pattern by one position and start over.

  Looks Inefficient ? Yes

  Because during the mismatch above, we learned some key information about the text- because first three characters match successfully.

  We implicitly know what the first two character are of the next substring are, so we should not explicitly check them.
- KMP uses this information to reduce number of times it compares each character of text with character in pattern.

KMP ….contd

- The difficulty lies in figuring how far to skip when a mismatch occurs. The goal is to skip as far as possible without missing any potential matches.
- Trick here is to pre-compute a table of skips for each prefix ahead of time, so that at running time, we immediately know how many chars to skip.
- Note that this pre-computation is based on entirely upon Pattern P, so the length of the text do not have any role to play in cost of pre-computation.