Design and Analysis of Algorithms

Lecture 08 – Strings Searching

String Searching

Content produced by persons is most efficiently stored as text.

- Structured (machine-readable) text is good for interacting with automated systems.
- Free text enables personal creativity, yet is open for machine manipulation.
- Sound and video data is analyzed and tagged for efficient access (ie. jump to where he kisses her) and those tags are text.

Therefore finding a particular text pattern in a large text file is one of the core algorithms.

- The need has been around since early 1970s, and most famous algorithms have been developed in late 70s or early 80s.
- Exact matching and similar text matching are different problems but their solution approaches are generally the same.
- The worst case performance of such algorithms is very slow.

String Searching

The Brute Force Approach

- Represent the text file as a very long array of characters.
  - Practical limit is around 2 GB per text file.
  - Works better if your search can be limited to per line, so that your patter does not span multiple lines.
- Among this very long array of characters, find if a shorter array exists (ie. substring search).
  - Have a helper function to check sameness. This function compares aligned characters one by one, and.
  - Align the first character of the text array with the first character of the pattern array. Run the helper function.
  - If there is a mismatch, then shift the pattern by one.

String Searching

The Brute Force Approach

Demonstration

Pass 1 / Comparisons (until mismatch) 2

Text: **A** D F G A B R A C A D A B R A F G J

Pattern: **A B** R A C A D A B R A

Pass 2 / Comparisons (until mismatch) 1

Text: A D F G A B R A C A D A B R A F G J

Pattern: **A** B R A C A D A B R A

Pass 3 / Comparisons (until mismatch) 1

Text: A D F G A B R A C A D A B R A F G J

Pattern: **A** B R A C A D A B R A

String Searching

The Brute Force Approach

- Let the size of the pattern be m, and the size of the text be n.
- The brute force approach makes up to (n-m+1) runs of the helper function. Since we assume m << n, this is roughly equal to n. This means the main algorithm is O(n).
- The helper function is obviously O(m).
- Therefore the brute force is O(mn).

Most “optimized” algorithms have cases in which they degenerate into the brute force approach.

- All popular algorithms have some assumptions that make the search much much faster.

String Searching

Karp-Rabin Algorithm (1987).

- This is an improvement over brute force. We run the O(m) helper function only when the text is similar to the pattern.
- We need a specialized hash function that is **easy to compute for a shifting pattern of values** (rolling window).
  - An example of such a hash function is a **rolling sum**. We subtract the item that is going out of the window, and add the newly added item. This is just 2 simple operations.
  - Polynomial functions used in this type of scheme are called Rabin fingerprints.
- Once we have such a function, we can have a first pass, calculating the hashes of all sub-strings starting with each item in the large array. This is obviously O(n).
- We store these hashes in a hash table for fast access.
- Then we calculate the hash for the pattern. This is O(1).
- Once we have all hashes, we search for hash values that are similar to the hash value we ae searching. We run the one-by-one comparison only on those locations.

String Searching

Karp-Rabin Algorithm (1987).

The algorithm is O(n + km) where k is the number of times we need to run the one-by-one comparison. Note that the hash function may give us false positives, so we cannot safely argue that k=1.

The advantage or Karp-Rabin appears when we do multiple searches on the same text file over time. The hash table is re-used over and over.

For a large number of runs over time (K), we get O ((n+Kkm) / K) average performance. If K approaches n then this is O (n+nkm)/n) = O(1+km) ~ O(m).

This is one of the reasons it is used commonly in **plagiarism detection**.

- Extract patterns in the submitted homework (many short arrays), and then search for those patterns in already hashed reference material (research papers, etc).
- When there are exact matches, mark the pattern in the homework.
- At the end calculate the percentage of text marked in the homework.

String Searching

Boyer-Moore Algorithm (1977).

This is one of the most efficient algorithms for generic use.

The idea is that, once a mismatch is found, we could shift more than one character. We can shift to a **position after the mismatch**. Therefore the helper function starts comparing from the end. If we get mismatches from the end, we shift much more.

There are two types of shifts introduced in the algorithm

- Bad-character shift
- Good-suffix shift

The algorithm is O(n/m) in the best case, and O(mn) in the worst case.

String Searching

Boyer-Moore Algorithm (1977).

Bad-character shift

The bad-character rule considers the character in the text at which the comparison process failed (assuming such a failure occurred).

The next occurrence of that character to the left in pattern is found, and a shift which brings that occurrence in line with the mismatched occurrence in text is proposed.

If the mismatched character does not occur to the left in pattern, a shift is proposed that moves the entirety of pattern past the point of mismatch.

String Searching

Boyer-Moore Algorithm (1977).

Good-suffix shift

Suppose for a given alignment of P and T, a substring t of T matches a suffix of P, but a mismatch occurs at the next comparison to the left.

- Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and the character to the left of t' in P differs from the character to the left of t in P. Shift P to the right so that substring t' in P aligns with substring t in T.
- If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T.
  - If no such shift is possible, then shift P by n places to the right.

If an occurrence of P is found, then shift P by the least amount so that a proper prefix of the shifted P matches a suffix of the occurrence of P in T.

- If no such shift is possible, then shift P by n places, that is, shift P past t.

String Searching

Quick Search (1977).

This is basically Boyer-Moore without the complex good-suffix shift.

The algorithm is O(mn).

However if the alphabet is larger than the pattern (ie. 26 characters in English alphabet and a pattern of 4 letters), then the algorithm is observed to proceed very fast.