| | | |
|---|---|---|
| | **6.854 Advanced Algorithms** | |
| Lecture 1: 9/3/2003 | | Lecturer: Erik Demaine, David Karger |
| | Scribes: Jiawen Chen | |

# String Matching

## 1.1 Problem formulation

We consider a basic problem in string matching, and two different approaches to it. The first will be algorithmic, while the second will emphasize data structures. The basic problem we consider is: given a string $T$ and a pattern $P$, determine if $P$ appears as a substring in $T$, i.e.:

**Input:** A text $T$ and a pattern $P$.
**Output:** Is $P$ a substring of $T$? If so, where?

### 1.1.1 Motivation and Background

The string searching problem arises in many fields. An example is searching for common gene sequences in a long DNA sequence. A more familiar example are the type-ahead-find features of modern Internet browsers and text editors.

### 1.1.2 Obvious Algorithm

The obvious algorithm to solve this problem compare each $P$ with $T$ at each position, shifting over $P$ at each iteration. The running time of this algorithm is $O(|T| * |P|)$. The algorithm needs to compare all $|P|$ characters of $P$ at each iteration, and there are at worst $|T|$ iterations (actually, $|T| - |P|$).

### 1.1.3 Rabin-Karp String Matching

The Rabin-Karp string matching algorithm is a randomized algorithm that can solve the string matching problem in $O(|T| + |P|)$. The main technique used here is *fingerprinting*, which we will discuss in detail a bit later. The motivation is as follows: if we can find a way of compare the pattern $P$ with a $|P|$ segment of $T$ in $O(1)$ time, we can search the text in $O(|T|)$ time.

The idea of fingerprinting is as follows: comparing two strings $x, y$ of length $|P|$ takes $|P|$ comparisons. Suppose we had a function $f$ with the following property:

$$f(x) = f(y) \iff x = y$$

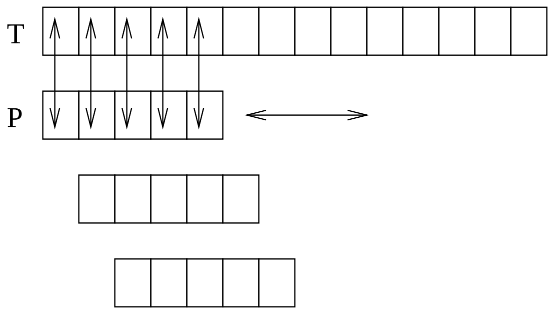

Figure 1.1: Naive String Matching

and checking $f(x) = f(y)$ could be done in constant time. This is not possible, so instead we settle for the following:

**Definition 1** ***fingerprint of a string*** $(f(x))$**:** *The fingerprint of a string $x$ is a mapping from the string $x$ to a set of integers such that if $x = y$ then $f(x) = f(y)$ and if $f(x) = f(y)$ then $x = y$* with high probability.

Assume (without loss of generality) that the input strings are over $\{0, 1\}$. Treat a string $x$ as the binary representation of a number. Define $f(x) =$ (number represented by $x$) mod $p$, where $p$ is a random prime.

To do the comparisons, we need to compute $f(P)$ and $f(T[i : i+|P|-1])$, for every $1 \leq i \leq |T|-|P|$. Can we compute this in $O(|T| + |P|)$ time? Yes. The trick is to compute $f(T[i + 1 : i + |P|])$ from $f(T[i : i + |P| - 1])$ in constant time as follows: compute $T[i + 1 : i + |P| - 1]$ from $T[i : i + |P| - 1]$ by "dropping" $T[i]$: subtract from the current fingerprint $2^{|P|-1} * T[i]$.
Then compute $T[i + 1 : i + |P|]$ from $T[i + 1 : i + |P| - 1]$ by "adding the last letter and shifting": multiply the current fingerprint by 2 and add $T[i + |P|]$.

### Analysis

A *false match* when $f(x) = f(y)$ but $x \neq y$. For our fingerprint, this occurs when $f(x) \equiv f(y)\ (mod\, p)$ which is exactly when $f(x) - f(y)$ is a multiple of $p$. (Slight notational difficulty here: $P$ with a capital letter is the input text pattern we want to find in the text and is a string. $p$ with a lower case is the random prime number that we pick.)

What is the probability that a *false match* occurs? This is exactly the following probability:

$$\Pr\left(n \text{ is a multiple of } p\right)$$

where $n = f(x) - f(y)$. We upper bound this probability as follows: how many prime factors does $n$ have? Clearly, this is at most $\log n$. As long as the prime we choose is not one of these, we will be fine. Suppose we choose $p$ as a random prime from $[2, Q]$. By the Prime Number Theorem, there are roughly $Q/\log Q$ primes in $[2, Q]$. Therefore, we have that:

$$\Pr(n \text{ is a multiple of } p) \leq \frac{\log n}{Q/\log Q}$$

Using the *Union Bound*, which states that:

$$\Pr(E_1 \cup E_2) \leq \Pr(E_1) + \Pr(E_2)$$

We have that:

$$\Pr(\exists \text{ a failing test}) \leq |T| \frac{\log n}{Q/\log Q}.$$

By choosing $Q$ appropriately large, we can make this probability vanishingly small, so that our algorithm will almost always be correct.

#### Machine Model

One aspect that we did ignore in our analysis has been the machine model. We have assumed that we can compare and manipulate the fingerprints in $O(1)$ time. We reasonably assume that our machine can manipulate *machine words* in $O(1)$ time. A 32-bit machine word can address an incredibly large range, so it is reasonable to have our machine can manipulate, in constant time, machine words of size $O(\log n)$, where $n$ is the problem size. This assumption is known as the *Transdichotomous assumption*.

For the Rabin-Karp string matching algorithm, we need to manipulate integers up to size $Q$. For $Q = (|T| * |P|)^2$ so that the probability of a false match is $\leq \frac{1}{|T|*|P|}$, we have $\log Q = 2(\log|T| + \log|P|)$. Therefore, using this machine model, Rabin-Karp can be run in $O(|T| + |P|)$.

#### Comments

Notice that even if Rabin-Karp finds a mismatch, we can easily check the result to see if it really did match. But now we must perform the check every time the fingerprints match, which increases the runtime.

The version of Rabin-Karp as stated above is known as a *Monte Carlo Algorithm*, which is "always fast and probably right."

If we check the results and slow it down, it becomes a *Las Vegas Algorithm*, which is "always right, and probably fast."

Given a Monte Carlo algorithm, it is easy to convert it to a Las Vegas algorithm if we can check our answers. Going the reverse way is more difficult.

### 1.1.4 Suffix Trees

We have discussed the string matching problem from the algorithms perspective and described a randomized algorithm to solve the problem in just $O(|T| + |P|)$ time. Here, we discuss a data structures perspective.

Although Rabin-Karp is "fast," we may want to preserve the work we have done if we have multiple queries. We want to design a data structure that can answer multiple queries efficiently.

Let $n$ be the length of the text and $m$ be the length of the pattern. We want to preprocess the text in $O(n)$ time and produce a data structure that takes $O(n)$ space and can answer queries in $O(m)$ time.

**Trees and Tries**

Here we give a description of various data structures that we might try to use:

1. **Tree Dictionary** Keep a set of $k$ strings in a binary tree dictionary. Check if the pattern is one of the strings in tree. This would take $O(m \log k)$ time to do a search. The problem here is that we would have to store all $n(n+1)/2$ substrings in the dictionary. This is clearly too large a value for $k$.

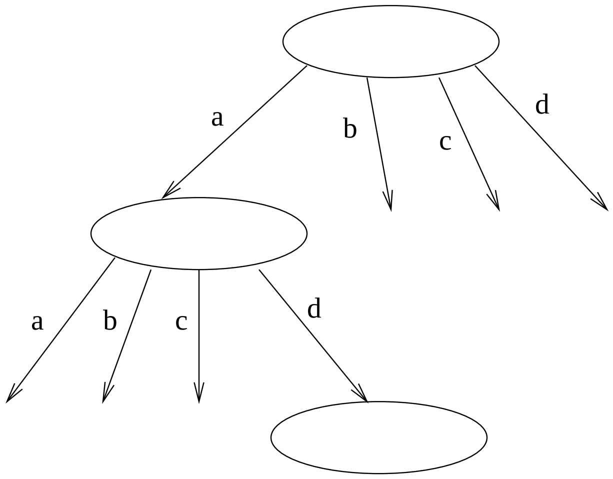


Figure 1.2: An example of a trie on {a,b,c,d}.

2. **Trie** A trie is another type of search tree. The difference between a trie and a regular search tree is that the outgoing edges of a node are the ones that correspond to characters. A walk down the path of a trie corresponds to iterating through the characters of a string. See Figure 1.2 for an example. The nodes store the index information for a search. At each node, the edges are implemented by an array of size $|\Sigma|$, where $\Sigma$ is the alphabet and is thus directly indexable. Therefore, tries can search for a pattern in $O(m)$ time! The problem with a trie is that it also requires the storage of all possible substrings of the text and would take $O(n^2)$ time to build the trie.

3. **Suffix Tree** Since we realize that storing all possible substrings is highly redundent, instead build a trie of only the $n$ *suffixes* of the text. Any substring of the text is a *prefix* to *some suffix* in the text; therefore, it reaches some internal node of the trie. We label each leaf node

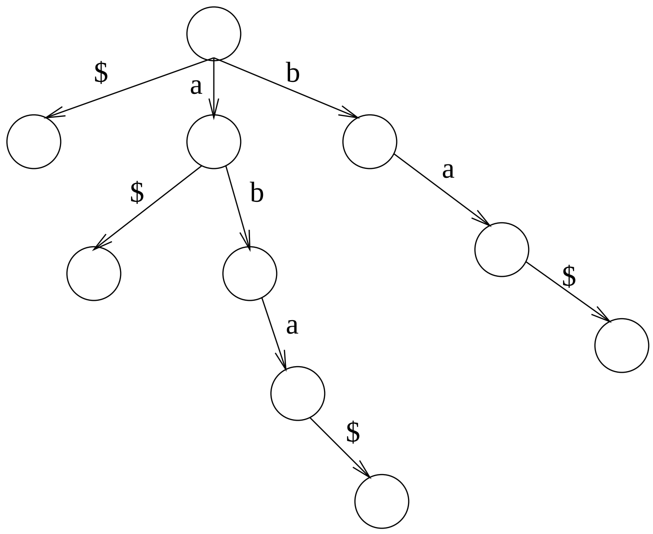


Figure 1.3: A suffix tree on aba$

of the suffix tree with the starting position of the suffix it represents. In order to ensure that every suffix is represented by a leaf node, we append to the input text a special character $. The size of such a suffix tree is still $O(n^2)$. If the suffix tree is constructed in a naive way, for every character in every suffix we have to either perform a search or create a new node, requiring a total construction time of $\Theta(n^2)$.

**Suffix Tree Construction**

The goal will be to try to build suffix trees in linear time and space. We begin an attempt at that here.

*Observation:* Suppose we just inserted $aw$, where $a$ is a character and $w$ is a string, we know that the next insertion is going to be $w$.

Next, we define a "suffix link", which will be central to our linear time construction of suffix trees.

**Definition 2** ***Suffix Links*** *A suffix link is a pointer from the node representing $ax$ to the node representing $x$.*

We build a suffix tree by inserting the longest suffix (the entire word), then the second longest suffix, etc., and inserting suffix links from the word just inserted to the last word inserted. The following makes this precise.

**Insertion Algorithm** Assume $aw$ was just inserted. Keep the pointer at the bottom of the path walked to insert $aw$. Walk up until we find a suffix link or reach the root. If we reached a suffix link, traverse it. If you found the root, stay there. Walk down as much as we walked up, extending the trie as needed. In other words, search/insert the substring of $w$ you just walked up. For each node we walked up, add new suffix links.

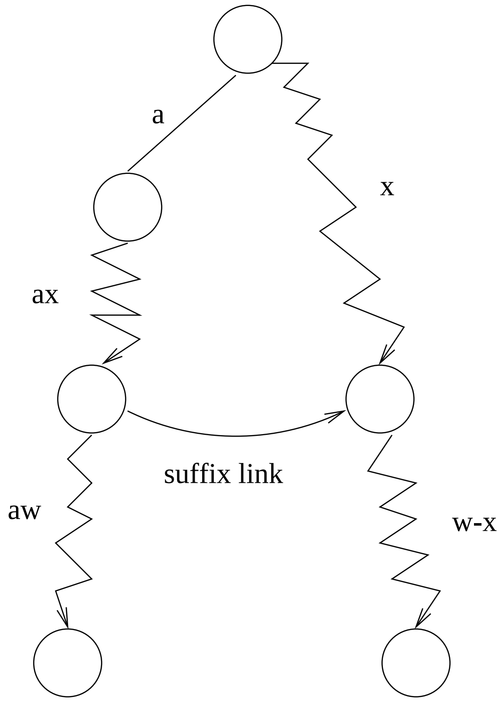


Figure 1.4: Suffix Tree Construction

**Analysis**

Observe that during construction, we never ascend past a node with a suffix link and each node we ascend gets a suffix link after this pass. Therefore, the total ascent work is at most the number of suffix links, which is at most the number of nodes in trie.

Since we descend as much as we ascend, the total work is $\Theta$(size of suffix tree).

Even if we use suffix links, construction time is still not $O(n)$ since $|T| = \Theta(n^2)$. We will see next lecture how to *compress* the tree so that we can achieve $O(n)$ in both space and time.