5 String Matching

Readings: CLRS 32

Sections not covered in the textbook: suffix trees

String matching, in many of its forms, is one of the most studied and most important problems in computer science. The basic definition of the problem is as follows:

Definition 1 (String matching problem)**.** *Given is a text $T[1 \dots n]$ (usually long), and a pattern $P[1 \dots m]$ (usually short), over a finite alphabet Σ (such as $\Sigma = \{a, b, c, \dots, z\}$).* Pattern P occurs in text T with shift i *if and only if $P[1 \dots m] = T[i+1 \dots i+m]$ (we also say that i is a* valid shift*). The* string matching problem *is to find all valid shifts of pattern P in text T.*

Naive algorithm:

```
for i := 0 to n-m
  valid := true;
  for j := 1 to m
    if P[j] != T[i+j]
      valid := false; break loop

  if (valid) then output shift i
```

Running time: $\Theta(mn)$
Lower bound: Input $P = a^{m-1}b$, $T = a^n$ takes $(n - m + 1).m = \Omega(mn)$ comparisons.

5.1 Rabin-Karp algorithm

Idea: Use a hash function over windows of length m. Compare words only at positions, where the hash matches that of the pattern P. **For this to be efficient, the hash function must support constant-time "shifting" to the right.**

Example:

- $\Sigma = \{0, 1, \dots, 9\}$
- $h(x_1 x_2 \dots x_m) = \underbrace{(x_1 x_2 \dots x_m)}_{(*)} \bmod 13$

  $(*)$: considered as a base 10 number
- Computing initial hash value: (using Horner's formula)

$$h(x_1 x_2 \dots x_m) = x_m + 10(x_{m-1} + 10(\dots + x_1)) \bmod 13$$

- Shifting to the right:

```
x1 x2 ... xm
   x2 ... xm x(m+1)
```

$$h(x_2 x_3 \dots x_{m+1}) = 10(h(x_1 x_2 \dots x_m) - 10^{m-1} x_1) + x_{m+1} \bmod 13$$

- See example of how this works in Figure 32.5 in CLRS

```
Robin-Karp(T[1..n],P[1..m]):
  hashp = hash(P,1)
  hasht = hash(T,1)

  for i:=0 to m-n
  | // inv: hasht = hash(T,i+1)
  | if (hashp = hasht)
  | | // check whether T[i+1..i+m] matches the pattern
  | | valid = true
  | | for j = 1 to m
  | | | if P[j] != T[i+j]
  | | | | valid = false; break loop
  | | if valid then output i
  | hasht = shift_hash(T,i+1,hasht)
```

  – hash function is $(S[i]S[i+1]...S[i+m-1]) \bmod q$
  – precomputed $ttmm1 = 10^{m-1} \bmod q$

```
hash(S,i):
  // compute hash of S[i]S[i+1]...S[i+m-1]
  result = 0
  for j = 0 to m-1
  | result = (10*result + S[i+j]) mod q
  return result

shift_hash(S,i,oldhash):
  // compute hash of S[i+1]S[i+2]...S[i+m]
  // given that oldhash is a hash of S[i]S[i+1]...S[i+m-1]
  return ((oldhash + q - (S[i]*ttmm1 mod q))
          * 10 + S[i+m]) mod q
```

Running time analysis:

- In the worst case: $\Theta(nm)$ (the same as naive algorithm)
- More detailed analysis:
  - $O(m)$ – hash function initialization
  - $O(n)$ – shifting the hash function and comparing to the hash of the pattern
  - $kO(m)$ – checking whether pattern matches (k is number of *hits*: places where hash matches the hash of the pattern, including *spurious hits*: hits that are not real matches of the pattern)

  Total: $O(n+m+km)$

Value of k is hard to estimate, however if we make some assumption about randomness of the string and usual assumptions about uniformity of the hash function, we should expect $k = O(n/q)$, where q is the size of the domain of the hash function (we will not do precise analysis here).

Therefore the **expected running time is** $O(n+m+\frac{nm}{q})$, and in particular, if $q > n$, we can conclude that the expected running time is $O(n+m)$.

5.2 String Matching with Deterministic Finite-State Automata

(parts of this section come from lecture notes by David Eppstein)

Recall the naive algorithm. Here is a typical example of its execution. Each row represents an iteration of the outer loop, with each character in the row representing the result of a comparison (X if the comparison was unequal). Suppose we're looking for pattern "nano" in text "bananobano".

```
      0  1  2  3  4  5  6  7  8  9 10 11
   T: b  a  n  a  n  a  n  o  b  a  n  o

 i=0: X
 i=1:    X
 i=2:       n  a  n  X
 i=3:          X
 i=4:             n  a  n  o
 i=5:                X
 i=6:                   n  X
 i=7:                         X
 i=8:                            X
 i=9:                               n  X
 i=10:                              X
```

Some of these comparisons are wasted work! For instance, after iteration i=2, we know from the comparisons we've done that T[3]="a", so there is no point comparing it to "n" in iteration i=3. And we also know that T[4]="n", so there is no point making the same comparison in iteration i=4.

The main idea of the algorithms in this section is that after we have invested a lot of work making comparisons in the inner loop of the code, we know a lot about what is in the text. Specifically, if we have found a partial match of j characters starting at position i, we know what is in positions $T[i] \dots T[i+j-1]$.

We can use this knowledge to save work in two ways. First, we can skip some iterations for which no match is possible. Consider overlapping the partial match we have found with the new match we want to find:

```
i=2: n  a  n
i=3:    n  a  n  o
```

Here the two placements of the pattern conflict with each other—we know from the i=2 iteration that $T[3]$ and $T[4]$ are "a" and "n", so they can't be the "n" and "a" that the $i = 3$ iteration is looking for. We can keep skipping positions until we find one that doesn't conflict:

```
i=2: n  a  n
i=4:       n  a  n  o
```

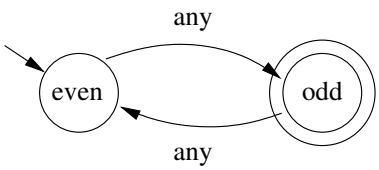
Here the two "n"s coincide.

Deterministic finite-state automata (DFA) (review from CS241)

Deterministic finite-state automaton (DFA) consists of *states* and *transitions*. Each transition is marked with one or several symbols from the alphabet Σ; for a given state there is exactly one outgoing transition for each symbol of the alphabet. One of the states is designated to be *starting state* and one or several states are marked as *accepting state*.

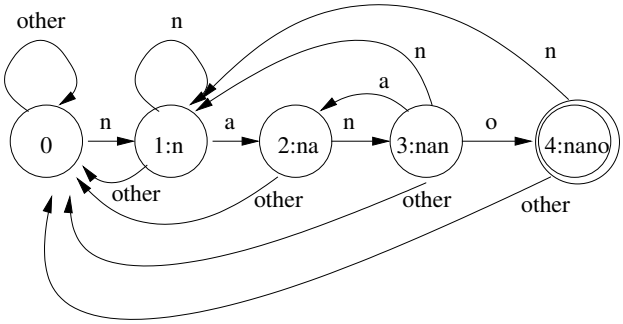
DFA can be used to process text T. We start in the starting state and look at the first letter of the text. According to this first letter, we choose the transition to continue, and change to a new state. This process continues, until we process the whole text. If at position j we are in an accepting state, we say that $T[1 \dots j]$ is accepted by the DFA.

For example, following automaton can be used to accept strings of odd length:



We can represent DFA as their transition function. If the set of states is V and alphabet is Σ, then the transition function is $trans : V \times \Sigma \to V$ (i.e., for state v and symbol s, $trans(v, s)$ is the new state).

DFAs for string matching Consider building a DFA that will accept all the strings that end with a pattern "nano":



The above automaton can be constructed in a straightforward way. Each state corresponds to a prefix of word "nano". Any match will bring us closer towards the accepting state, which corresponds to the complete word "nano". On the other hand, when we observe a mismatch (such as getting character "a" in the state "nan"), we need to go back to the state that represents the longest matching prefix, i.e. to the state which is at the same time suffix of a resulting word (in here "nana") and a prefix of the original pattern ("nano"); in this case, we have to transition to the state "na".

If we number the states with numbers $0 \dots m$, where state i corresponds to the prefix $P[0 \dots i]$ of the pattern P, we set m to be the accepting state, and we compute the transition function *trans* as follows:

```
for i := 0 to m
  prefix := P[1..i];

  for all symbols c from the alphabet
    current := prefix + c;
    for j := i+1 downto 0
      if current[i+1-j+1..i+1] = P[1..j]
        trans[i,c] := j;
        break the loop
```

Running time: $O(m^3|\Sigma|)$

If we are now given text T, we can find all matches of the pattern P in the string T by processing this text by the DFA corresponding to the pattern P. Whenever we reach the accepting state, we have found another occurrence of the pattern P:

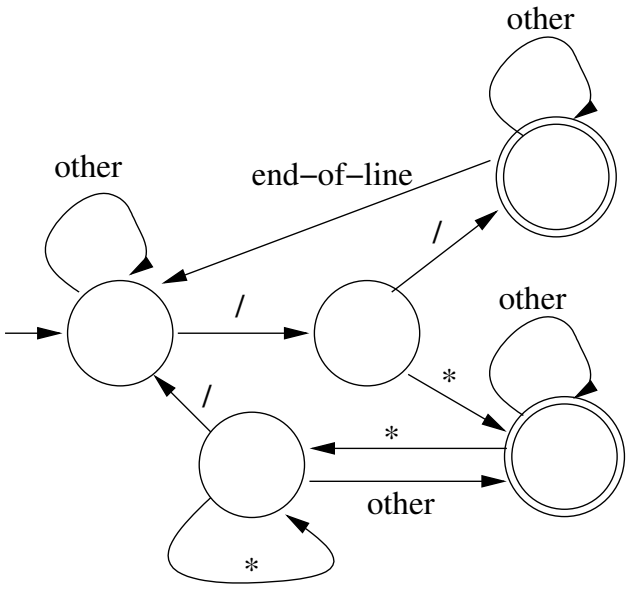
```
DFA_STRING_MATCHING(T[1..n],trans):
  state:=0;
  for i:=1 to n
  | state:=trans[state,T[i]]
  | if (state = m)
  | | output shift i-m
```

Running time: $O(n)$

We have decomposed the problem of string matching into two steps:

- **Step 1:** Preprocessing of pattern P in $O(m^3|\Sigma|)$ time. The output of this step can be reused, if we want to search for the same pattern in multiple texts.
- **Step 2:** String matching in $O(n)$ time.

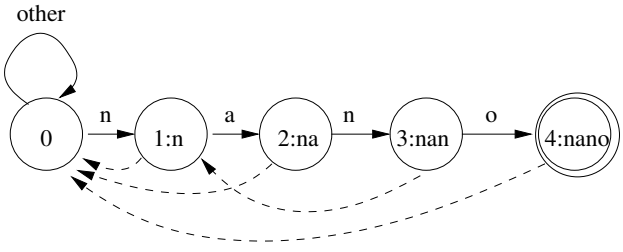
DFAs for other types of patterns. There is no reason, why the DFA must represent exact match of a particular string. We can use the same idea to look for different kinds of patterns, for example, the following DFA looks for the comments in C++ code:



5.3 Knuth-Morris-Pratt algorithm

The algorithm to construct DFA for a given pattern of length m required $O(m^3|\Sigma|)$ time. In this section we will show, how to reduce this running time to $O(m)$.

First of all, we will change the DFA. The transitions that move forward in the DFA will stay the same, however, for a given state we will replace all of its transitions going backwards (corresponding to various mismatches) with a single "mismatch" transition; this transition will be special, because we are allowed to take it without consuming a character from the input string (such transitions are commonly called ε transitions). In the following example, dashed lines correspond to "mismatch" transitions.



When we are matching the text T with this automaton, we would use as many transitions as necessary to consume the next character of the text. For example, to match word `bananannano`, we would use the following transitions:

$$0 \xrightarrow{b} 0 \xrightarrow{a} 0 \xrightarrow{n} 1 \xrightarrow{a} 2 \xrightarrow{n} 3 \to 1 \xrightarrow{a} 2 \xrightarrow{n} 3 \to 1 \to 0 \xrightarrow{n} 1 \xrightarrow{a} 2 \xrightarrow{n} 3 \xrightarrow{o} 4$$

The transition function of this automaton can be represented much more succintly than the transition functions of the DFAs that we designed in the previous section. We don't need to represent "solid" transitions explicitly, because their structure is obvious, and it can be hardcoded in the program doing the matching. This leaves the "dashed" transitions. There is one such transition per state. We will represent these transtions by *prefix function* $\pi(i)$ as follows.

Definition 2 (Suffixo-prefix)**.** *We will call proper substrings (the substrings that are shorter than the original string) of a given string s that are both suffixes and the prefixes of s,* suffixo-prefixes *of s.*

Definition 3 (Prefix function)**.** *Prefix function $\pi(i)$ for every state $1 \leq i \leq m$ gives the target of the "mismatch" transition. In particular, $\pi(i)$ is the length of the longest suffixo-prefix of string $P[1 \dots i]$. (We also define $\pi(0) = 0$.)*

With this representation using the prefix function π (in pseudocodes `pi`), we can now easily write the pseudocode for the string matching algorithm:

```
KMP_STRING_MATCHING(T[1..n],pi):
  // P[m+1] = some character outside alphabet
  state := 0
  for i := 1 to n
  | while state>0 and T[i]<>P[state+1]
  | | state := pi[state]
  | if T[i] = P[state+1]
  | | state := state + 1
  | if state = m
  | | output shift i-m
```

Estimating running time of this algorithm is not as straightforward as before. The transitions using the prefix function π do not advance index i in the text; in fact, in some cases we may have to use as many as $\Theta(m)$ transition before we can advance index i. Therefore it would seem that the running time of the algorithm is $\Theta(nm)$.

However, we can safely say that we will use at most $O(n)$ "solid" transition, because each of these transitions requires a new symbol from the text T. We also note, that each "solid" transition increases the number of the state by at most 1, while each "dashed" transition decreases the same number by at least 1. Since we do not have negative state numbers, **altogether there must be at most as many "dashed" transition as "solid" transitions**, i.e. $O(n)$. Therefore **the whole algorithm takes $O(n)$ time.**

Constructing the prefix function. If we used naive algorithm to construct the prefix function, we would require $O(m^2)$ time for each state. Since there are m states, this would require running time $O(m^3)$. However, the following property of suffixo-prefixes will help us to construct the prefix function faster:

Lemma 1. *If substring $A[1 \dots \ell]$ is a suffixo-prefix of $P[1 \dots i+1]$, then substring $A[1 \dots \ell - 1]$ must be a suffixo-prefix of $P[1 \dots i]$.*

We can use this property to compute the value of $\pi(i)$ as follows:

```
for all values of q such that P[1...q] is suffixo-prefix of P[1...i-1]:
(the values are listed from largest to the smallest)
| if P[q+1]=P[i] then
| | pi[i]:=q+1;
| | break loop;
```

Exercise: Show that the following code lists all suffixo-prefixes of string $P[1 \dots i]$, from the longest to the shortest:

```
q:=i;
while q>0
| q:=pi[q]
| print P[1..q]
```

Hint: You may want to use induction on value of i.

We can combine these two ideas into the following pseudocode that computes the prefix function:

```
CONSTRUCT_PREFIX_FUNCTION(P[1..m]): (version 1)
  pi[0]:=0;
  for i:=1 to m
  | q:=i-1; pi[i]:=0
  | while q > 0
  | | q:=pi[q]
  | | if P[q+1]=P[i] then
  | | | pi[i]:=q+1;
  | | | break loop;
```

Straightforward estimation of the running time suggests that this code runs in $O(m^2)$ time. This estimate can be improved, if we rewrite the code as follows (the two codes are clearly equivalent and run in the same time):

```
CONSTRUCT_PREFIX_FUNCTION(P[1..m]): (version 2)
  pi[0]:=0; pi[1]:=0;
  q:=0;
  for i:=2 to m
  | while q > 0 and P[q+1]<>P[i]
  | | q := pi[q];
  | if P[q+1] = P[i]
  | | q := q+1
  | pi[i] := q
```

To estimate the running time, we will now concentrate on the value of q. Note that it increases at most m times over the course of the whole algorithm (at most once in each iteration of the outermost cycle). In a single iteration of the outermost cycle, it may decrease several times, however, it can never be negative. Therefore it decreases at most as many times, as it increases (i.e., m times). Therefore the **overall running time is $O(m)$.**

(If you think this sounds familiar, you are right: we just used the same trick to estimate the running time of the string matching algorithm.)

Summary. Knuth-Morris-Pratt algorithm is composed of two parts:

- For a given pattern $P[1 \dots m]$, compute the prefix function π in $O(m)$ time.
- For a given text and a prefix function π, find all shifts that match the pattern represented by the prefix function in $O(n)$ time.

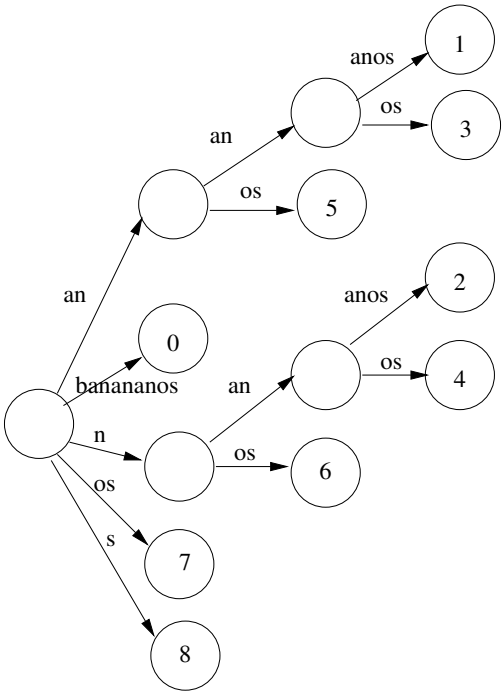
5.4 Suffix trees

In methods using DFAs for string matching, we first preprocessed the pattern and then were were able to search for that pattern in any text faster. In this section, we will show methods, where we first spend some time preprocessing the text, and then we will be able to search for any pattern in this text quickly.

Let $T[1 \dots n]$ be a string of n characters (n can be very large). Let $T^{(i)}$ denote the i-th suffix of string T, i.e. the substring of string T that starts at position i, and extends all the way to the end of the string. In other words, $T^{(i)} = T[i \dots n]$.

If string P occurs as a substring of string T at position i, then the string P is a prefix of string $T^{(i)}$. We will use a data structure, similar to dictionary, that stores *all suffixes of* T. In addition to usual dictionary operations, it supports operation **SearchPrefix** (is P a prefix of any of the strings present in dictionary?). Then we can do string matching in string T simply by asking SearchPrefix(P).

Compressed tries are an ideal data structure for this purpose. We will store each suffix of T in this data structure, with corresponding shift as a value. Here is an example of a suffix tree for $T =$ bananos.



- **Space:** The suffix tree has at most n leaves. Since the tree must branch in every internal node, the total number of internal nodes will be $O(n)$. Each node takes $O(1)$ space (assuming, we store strings corresponding to edges as pointers to the original string).

- **Time for construction:** With algorithms we have seen before, constructing this data structure takes $O(n^2)$ time. However, there are algorithms that can construct the suffix tree in $O(n)$ time. (If you are interested in this algorithm, you can find it in: E. Ukkonen: On-line construction of suffix-trees. Algorithmica 14 (1995), 249-260)

- **Time for string matching:** If the length of the pattern is m, then determining whether the pattern matches the text takes $O(m)$ time. All shifts can be retrieved in $O(m+k)$ time, where k is the number of valid shifts. (Simply traverse the subtree corresponding to the match; size of this subtree is $O(k)$, since it has at most k leaves.)

5.5 Summary

Algorithm	Preprocessing	Matching	
Naive algorithm		$O(mn)$	worst-case
Rabin-Karp		$O(m+n+\frac{mn}{q})$ q =domain size of the hash function	expected
Finite automata	$O(m^3\|\Sigma\|)$	$O(n)$	worst-case
Knuth-Morris-Pratt	$O(m)$	$O(n)$	worst-case
Suffix trees	$O(n)$	$O(m+k)$ k =number of matches	worst-case

Note, that in case of DFA matching and KMP algorithm, once we preprocess a pattern, we do not need to run preprocessing again if we want to search for the same pattern in a different text. Thus these methods are especially advantageous, if we want to search for the same pattern in multiple texts.

On the other hand, suffix trees require $O(n)$ preprocessing of the text, but if we want to search for a different pattern in the same text, we do not need to run this preprocessing again. This is advantageous, if we want to search for multiple patterns in the same text.