Algorithmic Design / Algorithmic Data Mining

Data Mining - Lecture 6

Tommaso Padoan <tommaso.padoan@units.it>

Pattern matching

Given two strings, the *text* and the *pattern*, the problem of finding all occurrences of the pattern inside the text is referred to as *string matching problem* or simply *pattern matching*. Formally, a *string* is a sequence of symbols or characters from a finite *alphabet* Σ, that is the set of all possible symbols/characters. The *empty string*, which has length 0, is denoted by ε. The set of all possible finite-length strings formed by characters from the alphabet is indicated by Σ^*, which includes the empty string ε. For what concerns pattern matching, we assume that both the text T and the pattern P belongs to Σ^*, that is, they are finite strings formed by characters from the same alphabet, and the pattern is smaller than (or equally long to) the text: $|P| \leq |T|$.

First, we introduce some notation. The *concatenation* xy of two string x and y consists of the characters of x followed by those of y in the original order, whose total length is $|x| + |y|$. We say that a string y is a *prefix* of a string x if there exists a string w such that $x = yw$. Similarly, we say that a string y is a *suffix* of x if there exists a string w such that $x = wy$. A prefix/suffix is called *proper* when the padding string in the definition $w \neq \varepsilon$, thus $|y| < |x|$. Note that the empty string ε is both a prefix and a suffix of every string, and so is the whole string itself, since $x = x\varepsilon = \varepsilon x$. For brevity, we indicate the first k characters (i.e. the k-long prefix) of a string x by $x_k = x[0, \dots, k-1]$, given a $k \leq |x|$. Therefore, $x_0 = \varepsilon$ for every string x. We say that a string y is a *substring* of x if $x = vyw$ for some strings v and w, possibly empty. So every prefix or suffix of a string is also a substring of it.

A string y, of length $|y| = m$, *occurs at position* i in a string x when $x[i, \dots, i+m-1] = y$ or, equivalently, y is a suffix of x_{i+m}. Then, pattern matching aims at finding all positions i at which the pattern P occurs in the text T.

Naive algorithm

Letting $|T| = n$ and $|P| = m$, the easiest way to find the position of every occurrence of P in T, is to traverse the whole text sequence, comparing the pattern with every possible m-long substring of the text. Every time a complete match is found, the starting position of the current substring is recorded, to be returned at the end. It is easy to see that this procedure will correctly return the positions of all

occurrences of the pattern in the text. However, the process will, in the worst case, take multiplicative time in the length of both strings $\Theta(nm)$, while linear $\Theta(n)$ in the best case. In particular, the worst-case complexity is quadratic $\Theta(n^2)$ if the pattern is a constant fraction of the text, that is, $m = n/c$.

```
def Naive(T, P):
  n = len(T)
  m = len(P)
  Occ = []
  i = 0
  while i <= n - m:
    if P == T[i : i+m]:   # compare P with T[i,...,i+m-1]
      Occ.append(i)
    i = i + 1
  return Occ
```

Knuth-Morris-Pratt algorithm

The naive algorithm is inefficient because the information gained by comparing the pattern with a substring of the text is never re-used for other substrings, even though they may overlap and share characters, and the pattern to match is always the same. Indeed, the information about one match or mismatch can be quite valuable also for some of the following ones, up to some length.

The algorithm designed by Knuth, Morris and Pratt (KMP) exploits the information gained by every comparison to avoid performing useless comparisons afterwards. To do this, however, it uses an auxiliary data structure (an array) which must be computed in advanced based on the pattern. Such data structure depends only on the pattern, so it can be later used for any text to be matched. For this reason it makes sense to divide the pattern matching procedure into two parts: first the *preprocessing* phase which builds the auxiliary array for the given pattern, and then the matching phase which finds every occurrence of the pattern in the text using the result of the previous phase.

The auxiliary data structure is called *prefix array* (or also *prefix table* or *function*) π, since it contains information about prefixes of the pattern, and has size $|\pi| = m$, same as the pattern. For every $1 \leq k \leq m$, $\pi[k-1]$ is the length of the longest prefix that is also a proper suffix of the k-long prefix P_k of the pattern. In other words, for all $0 \leq j < m$, $\pi[j]$ is the greatest value such that $P_{j+1} = P[0, \ldots, j] = wP[0, \ldots, \pi[j]-1] = wP_{\pi[j]}$ for some $w \neq \varepsilon$. In particular, this means that $\pi[0] = 0$ always. An example of pattern and corresponding prefix array is given in the table below. There, for instance, $P_3 = P[0, 1, 2] = aba$ is a proper suffix of $P_5 = P[0, \ldots, 4] = ababa$, and so $\pi[4] = 3$.

i	0	1	2	3	4	5	6
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

The KMP algorithm and the subroutine to compute the prefix array are actually quite similar. Indeed, the latter matches the pattern against itself, to find for every prefix the longest proper suffix that matches another prefix of the pattern. Like the KMP algorithm, also in this case the already discovered information is used to avoid some comparisons. In particular, the subroutine examines the prefixes of the pattern in order of (increasing) length, so that it can immediately re-use the part of the prefix array already built for the information about shorter prefixes.

```
def PrefixArray(P):
  pi = [0]
  q = 0
  i = 1
  while i < len(P):
    while q > 0 and P[q] != P[i]:
      q = pi[q-1]
    if P[q] == P[i]:
      q = q + 1
    pi.append(q)      # this sets pi[i] = q
    i = i + 1
  return pi
```

To analyse the time complexity of the subroutine, first of all observe that the outer loop performs exactly $|P| - 1$ iterations. Initially $q = 0$ and $i = 1$, and in each of those iterations q is increased at most by 1, while i is always increased by 1, thus we always have $q \leq i$. Consequently, since initially $\pi[0] = 0$, the values currently present in π are always bounded by their index itself, so that in the inner loop $\pi[q - 1] \leq q - 1$. Therefore, every iteration performed by the inner loop will always strictly decrease the value of q, but never makes it negative. Thus, the total number of iterations of the inner loop, aggregated along all iterations of the outer one, is at most the highest value that q could ever reach, that is $|P| - 1$, if q got increased at every iteration of the outer loop. So, the total number of loops iterations is at most $2(|P| - 1)$, thus the computation of the prefix array takes $\Theta(|P|)$ time, linear in the size of the pattern.

As already mentioned, the KMP algorithm works in a similar way, although it matches the entire pattern against the text. The information obtained from previous matches, also partial ones, is again exploited, with the help of the pre-computed prefix array, to have an head start on the following matchings. Similarly to before, a variable q is used to record the length of the longest prefix of the pattern that matches the q last characters (a suffix) of the currently read text. Whenever a mismatch is encoutered,

or a full match is found, such a value is decreased by just the necessary amount based on the collected information, to resume the matching process from the second (in general the next) largest prefix of the pattern that matches the previously read characters of the text. Note that clearly when $q = |P|$ the procedure knows that a match has been found.

```
def KMP(T, P):
  pi = PrefixArray(P)
  Occ = []
  q = 0
  i = 0
  while i < len(T):
    while q > 0 and P[q] != T[i]:
      q = pi[q-1]
    if P[q] == T[i]:
      q = q + 1
    if q == len(P):
      Occ.append(i - q + 1)
      q = pi[q-1]
    i = i + 1
  return Occ
```

The complexity analysis is also analogous. The main differences are that the outer loop will perform $|T|$ iterations instead, and the inner loop at most as many, thus at most $2|T|$ in total. Therefore, the search for all occurreces of the pattern will take $\Theta(|T|)$ time. If we add the cost of the preprocessing, we obtain a total time complexity of $\Theta(|P| + |T|)$, which is $\Theta(|T|)$ since the pattern is at most as long as the text.