Bit parallel string matching

Alina Gutnova

June 21, 2006

1 Introduction

The *string matching* problem (SMP) consists of finding substring (generally pattern) P in text T. In the basic form both P and T consist of characters in the same alphabet Σ. In practice the text can contain spelling errors. Thus we can extend the problem and try to find all occurrences of P where some characters can be missing, in wrong order, or the text contains additional characters. In the extended pattern matching we can search for more complex patterns which can be described by regular expressions. This problem is described in some applications like word processors, virus scanning, text information retrieval systems, digital libraries, web search engine, etc [HELC01].

There are many approaches to solve SMP. The simplest way to find a substring in the given text is a *brute force algorithm*. It compares all characters of the text with the beginning of the substring. If the characters match, it compares the next characters in the text. The brute force algorithm does not need any additional preparations or space but its time complexity is $O(nm)$ in the worst case and $O(n)$ in the average case, where n is the size of the text and m is the size of the substring [HELC01].

The classical *Boyer-Moore* (BM) [HELC01, RSB77] algorithm is one of the most popular and efficient for common applications (e.g. text editors). BM is like brute force, but it compares characters from right to left starting from the rightmost character that give the opportunity to make larger shifts.

The BM requires preprocessing which takes $O(m + s)$ time, where m is the size of the pattern and s is the size of the alphabet. It also needs $O(m + s)$ additional space where s is the size of the alphabet. Unlike the brute force algorithm, it has a time complexity $O(n+m)$ in the average case. The time required for the algorithm decreases with the increase of the size of the pattern and the alphabet [HELC01].

Another way to solve the SMP is to use finite automata. The *Knuth-Morris-Pratt* (KMP) [HELC01] algorithm is based on representing the pattern by the finite automaton. The states are marked with symbols that should match at the moment. There are two transitions from each state. One of them corresponds to the matching of the characters, while the other one correspond to the mismatching. The automaton moves to the next state if the comparison is successful. Otherwise the automaton moves to the previous state.

KMP needs preprocessing for constructing the automaton (which takes $O(m)$) and additional space ($O(m)$ time). The time complexity of the algorithm is $O(n + m)$ even in the worst case [HELC01].

In general, *bit-parallel string matching* (BPSM) [BYG92, Mye99] algorithm is the most efficient. In contrast to the algorithms considered above, the BPSM algorithm can solve also the extended SMP (described in the first paragraph). The main idea of the bit-parallel algorithms is that they store several data items into a single computer word and then update them in parallel using a single computer operation (e.g. bitwise operation [BYG92]). Bit-parallel algorithms are very efficient for approximate string matching [Mye99].

In the second section of the article we describe briefly some traditional solutions for string matching problem. Then the above mentioned BPSM algorithm is described more carefully in the third section and some empirical results are given in the forth section. The final conclusions are drawn in the fifth section.

2 Traditional solutions

In this section we briefly recall the traditional algorithms for SMP [HELC01, RSB77].

2.1 Boyer-Moore algorithm

When the size of the pattern and the size of the alphabet Σ are sufficiently large, the BM algorithm can be used.

The BM algorithm compares pattern P with text T from right to left. If the compared characters match, then the algorithm compares the next characters. Otherwise the algorithm shifts the pattern according to two heuristics: bad-character heuristic and good-suffix heuristic. The heuristics are independent and they are used simultaneously. The BM algorithm shifts the pattern by the longest of two distances, given by the bad character and

the good suffix heuristics. The tailpiece of the string is called *suffix* and the forepart of the string is called *prefix*.

In the general case, bad-character heuristic works as follows: let us suppose that P is the pattern and T is the text on the same alphabet Σ, $|P| = m$, $|T| = n$. Let us assume that $P[j] \neq T[shift + j]$ is the first mismatch during the comparison of characters from right to left where $1 \leq j \leq m$. Let k be the index of character in the rightmost occurrence $T[shift + j]$ to the pattern P (assume us that $k = 0$ if there is no such occurrence). Then we can increase $shift$ by $j - k$ and without missing any suitable shift. Indeed, if $k = 0$, then stop-symbol $T[shift + j]$ is not met in pattern P. Thus it is possible to shift the pattern to the $j - k = j - 0 = j$ position to the right. If $0 < k < j$, then the pattern can be shifted to the $j - k$ positions to the right, because with smaller shifting stop-symbol and the corresponding character of the pattern would not be equal. Otherwise, if $k > j$, the heuristic shifts the pattern to the left instead of shifting to the right, but the BM algorithm does not consider such cases, because good-suffix heuristic always gives non-null shift values.

Good-suffix heuristic is the following: if $P[j] \neq T[s+j]$ (where $j < m$ and j is the biggest value with such a condition), then the shift can be increased by the minimal distance $\gamma[j] = m - max\{k : 0 \leq k < m \& P[j + 1 \dots m] \sim P_k\}$, where $P_k = P[1, 2, \dots, k]$ from P, such that none of the characters of the suffix $T[shift + j + 1 \dots shift + m]$ would be opposite the different to it character from the pattern. Note, that $\gamma[j] > 0$ for each j. Thus BM algorithm will shift the pattern at least on one position to the right at each step.

2.2 Knuth-Morris-Pratt algorithm

BM is often used in practice, but the KMP algorithm is theoretically more efficient. KMP algorithm is based on finite automaton.

During the construction of a finite automaton for matching the substring in the text it is easy to construct transitions from the beginning state to the finishing state. The transitions are marked by the characters of the substring (Figure 1). The problem is in the attempts to add other characters that do not transfer to the finishing state.

Figure 1: The non-deterministic automaton which recognizes pattern $*automaton*$.

The KMP algorithm is based on the theory of finite automata, but it uses a more simple method for handling the unsuitable characters. The states in the algorithm are marked with symbols that should match at the moment. There are two transitions from each state. One of them corresponds to the matching of the characters, while the other one corresponds to the mismatching. The automaton goes to the next state, if the comparison is successful, otherwise it goes to the previous state.

Figure 2: A KMP-automaton for pattern *ababcb*.

If the comparison was successful, the algorithm compares the next characters. Otherwise it matches the current character again (Figure 2).

Note that when the compared symbols are the same, nothing special needs to be done, only the transition to the next state. On the other hand, transitions that correspond to the mismatch of characters depend on comparing the original pattern with itself. For example, when we match string "*ababcb*" there is no need to go back by four positions, if the next character is not "*c*" is not equal to the character being compared. If we get the fifth character in the pattern, then we know that the first four characters of the text and the pattern are the same. Therefore, characters "*ab*" in the text corresponding to the third and fourth characters of the pattern equal to first and second characters of the pattern.

Prefix-function is used for construction of the KMP automaton. This function is associated with the pattern P and gives the information about the positions of the different prefixes of the string in the pattern P. Prefix-function $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$ that is associated with the pattern $P[1 \ldots m]$ is defined as

$\pi[q] = max\{k : k < q \& P_k \gg P_q\}$.

In other words, $\pi[q]$ is the length of the longest prefix P that is the suffix of P_q (see example in Table 2.2).

P	a	b	a	b	a	b	a	b	c	a
π	0	0	1	2	3	4	5	6	0	1

Table 1: An example of the prefix function for the pattern *abababacba*.

3 Bit-parallel solutions

3.1 Basic algorithm

Before considering the main topic of the article we will briefly review the searching algorithm *Shift-AND* [UM92]. This algorithm works fast, is easy to implement and can be easily generalized to the case of approximate searching.

In a general case we need to find all occurrences of pattern P in text T. Let us generalize the problem and assume that we should to find occurrences of all possible prefixes of pattern P: $P(1) = p_1, P(2) = p_1p_2, \ldots, P(m) = p_1p_2 \ldots p_m$, where p_i is the i-th symbol of pattern P.

For example, if the pattern is $P = cacao$, we should find all occurrences of prefixes $c, ca, cac, caca$ and $cacao$. We need to construct a table that shows whether the current symbol in the text is the last symbol in each of the given prefixes. For each position in the text we will have a five-element bit vector, where the k-th bit equals to 1 if the k-th symbol in the text corresponds to the last symbol in the k-th entry of the prefix. As a result we have a table with m rows and n columns (Table 3.1).

$$R = \begin{array}{c|cccccccccccc} & c & a & c & a & \& & o & c & \& & c & a & c & a & o \\ c & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ a & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ c & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ a & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ o & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Table 2: All occurrences of prefixes $c, ca, cac, caca$ and $cacao$ of pattern $P = cacao$ in text *caca&oc&cacao*, where symbol & means some character except c, a and o.

We are interested in the last row, because it shows us, whether the pattern is present in the text.

Let us describe the performed operations. Let R_j be the j-th column of the table. Then R_j is an m-element bit vector, where $R_j[k] = 1$, if first j char-

acters of the pattern concur exactly with k characters of the text that precede T_j including T_j. In other words, $R_j[k] = 1$, if $p_1, \ldots, p_k = t_{j-k+1}, \ldots, t_j$.

There is a fast method for constructing the table. It can be shown that $(j+1)$-th column of the table depends only on the j-th column, on the pattern and on the character t_{j+1}. For example, the occurrence of *cac* in the $(j+1)$-th position is found only if *ca* occurs in the j-th position and $t_{j+1} = c$. In other words,

$$R_{j+1}[k] = \begin{cases} 1, & \text{if } R_j[k-1] = 1 \; and \; p_k = t_{j+1}, \\ 0, & \text{otherwise.} \end{cases}$$

We suppose that $R_0[k] = 0$ for all k $(1 \leq k \leq m)$ and $R_j[0] = 1$ for all j $(0 \leq j \leq n)$. If we consider, for example, first two columns in the table, $R_2[?] = 1$ only if two following conditions are satisfied: (a) $R_2[i] = 1$ only if $t_2 = p_i$; (b) $R_2[i] = 1$ only if $R_1[?] = 1$, where ? is some position in the column. The condition (a) provides the matching of the last characters of the pattern, while the condition (b) provides the matching of the preceding characters.

To check condition (b), it is enough to shift down the first column. To verify condition (a) fast we need to calculate the characteristic vector of length m for each character. Characteristic vector for character i in our case has 1's in the second and the fourth positions (i.e. in those positions where current character is found) and 0's in all other positions: 01010. Characteristic vector for c is 10100, for o it is 00001 and for all other characters of the alphabet it is 00000.

To verify condition (a), we shift down the first column (first column 10000 becomes $*1000$ after shifting) and compare it with the characteristic vector of i. The new values in the second column will be 1 in those positions, where the corresponding character in the shifted column and the corresponding value in the characteristic vector equal to 1 and 0 otherwise. The only exception is referred to the first position in the column for which the condition (b) is true, since there is no preceding positions. Therefore the value at the first position after shifting is always 1. Thus, by shifting and adding 1 to the third column (10100), we get 11010. After applying the bit operation *AND* for the i's characteristic vector (01010) and the new value of the third column (11010) we get value 01010.

So, first we construct a table which shows the matches of all prefixes of the pattern. Then we calculate recursively the elements of the table. In the end, we consider the method for calculating each column in the table by using a shift of the previous column and an operation of bitwise multiplication. Thus, algorithm *Shift-AND* needs $O(n)$ comparisons [UM92], which are only of bitwise operations.

3.2 Extensions

Let us assume now that in our example we need to find all entries of the pattern *cacao* in text T with at most one mismatch. Let us construct the following Table 3.2:

$$Q = \begin{array}{c|cccccccccccc} & c & a & c & a & \& & o & c & \& & c & a & c & a & o \\ c & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ a & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ c & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ a & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ o & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Table 3: All occurrences of prefixes $c, ca, cac, caca$ and $cacao$ of pattern $P = cacao$ in text $caca\&oc\&cacao$ with mismatches.

The table R is the same as the table that was described above. The second table Q looks like the first table, but it shows not only the exact match, but also the result of comparison, when there is one mismatch (replacement).

Let us consider the fifth column of the table Q. It differs from the fifth column of the table R in the first, third and fifth positions. Indeed, *caca&* matches with the pattern *cacao* with one replacement, *ca&* matches with *cac* with one replacement and & matches with c (first row in the table Q always consists of 1's). The match of the *caca&* and the *cacao* with one replacement is represented in the fourth column of the R as an exact matching with the *caca*. If there is an exact matching, then there is no more than one matching with one replacement. Thus, one of the methods to reconstruct Q from R is to shift the previous column of the table R without doing bitwise multiplication.

Let us consider now the tenth column (in Q it equals to 11010). There is one 1 in the second row (exact matching with *ca*) due to the shift. The fourth row corresponds to the matching of the *c&ca* and *caca*. The matching results are shown in the ninth column of the table Q after comparing the *c&c* and the last character i.

Only two additional arithmetic operations represent all the possibilities of occurrences of pattern P in text T with match and mismatch. Mismatch is replacement (changes), insertion (inserts) or gap (deletes). If there is an exact matching or one replacement for the current character of the text, then the result of the comparison depends on the shift of the previous column of table R. If there was a mismatch before, then the decision depends on the

previous modified column of Q and on the operation AND over the shifted column and the characteristic vector.

Now let us consider insertion and gap. The previous mismatches are shown in the previous column of Q and can be found out by a shift operation and bitwise multiplication as in the case of the replacement. Insertion can be displayed by copying the previous column of R without shifting. Gap can be displayed by shifting the current (new) column of R. For example, the third column of Q for which replacements, insertions and gaps are possible, would be 11110. Here the fourth 1 appears from the matching of *caca* and *cac* with one gap, etc.

If there is a situation with more than one mismatch, the matching can be done by introducing new auxiliary tables for each mismatch. This algorithm can match any regular expression with or without mismatches [Tho68].

Another approach to solve approximate string matching problem is using dynamic programming [Mye99]. The edit distance between two strings S and P is defined as the minimum number of character inserts, deletes and changes needed to convert P to S [HELC01]. Informally, the string edit distance matching problem is to compute the smallest edit distance between P and substrings of T. A well-known dynamic programming algorithm takes time $O(nm)$ to solve this problem [Mye99].

4 Empirical results

In this section we compare different string matching algorithms with the following parameters: the text size n, the pattern length m and the alphabet size $|\Sigma|$. It is known that none of the algorithms are optimal or best in all three cases [BYG92].

In [UM92] an experiment was made for a binary alphabet, an alphabet of size 8 and the English alphabet. The *KMP* algorithm produced in all cases exactly one character comparison. The *BF* algorithm produced approximately the same number of character comparisons for the alphabet of size 8 and for the English alphabet and required more character comparisons for small size alphabet. Empirical results show that for patterns of length greater than 10 the number of comparisons is approximately 2, twice the number required by the *KMP* algorithm for the binary alphabet. The number of comparisons of the BPSM algorithm is generally less than 1 with the exception of the binary alphabet, where the algorithms have on average 1.25 and 1.1 character comparisons. Furthermore, the number of comparisons of the *BPSM* is higher when the binary alphabet is used and decreases as the pattern length increases. Thus, according to the empirical results *BPSM*

is sublinear in the number of character comparisons. The bit-parallel algorithms are more efficient that other string matching algorithms for small and long patterns respectively. Their running time decreases as the pattern length increases and they produce similar running times in all cases with the exception of the binary alphabet.

5 Conclusion

In the article we have considered bit-parallels approaches for solving string matching problem (SMP). String matching is often used in different areas: text editors, virus scanning, digital libraries and web search engines. The paper consists of two parts: in the first part we described two traditional solutions for the problem: Boyer-Moore algorithm and Knuth-Morris-Pratt algorithm. The first one uses the finiteness of the alphabet and another constructs a finite automaton. BM is more used in practice, but theoretically KMP is better, because it solves the SMP in linear time even in the worst case. However, none of the algorithms is suitable for solving extended pattern matching. This problem can be solved by a bit-parallel algorithm described in the second part of the article. The BPSM uses both of the properties of the above mentioned algorithms. It can solve the SMP with or without mismatches and match several patterns the same time. However, in practice bit-parallel algorithm is more efficient than *KMP* and *BF*.

References

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, pages 168–175, 1992.

[HELC01] Cormen Thomas H., Leizerson Charles E., Rivest Ronald L., and Stein Clifford. Introduction to algorithms, second edition. *MIT Press*, 1180:757–785, 2001.

[Mye99] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46:395–415, 1999.

[RSB77] J. S. Moore R. S. Boyer. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[Tho68] K. Thompson. Regular expression search algorithm. *Communication of the ACM*, pages 419–422, 1968.

[UM92] Sun Wu Udi Manber. Fast text searching allowing errors. *Communication of the ACM*, 35:83–91, 1992.