A New Algorithm for Subset Matching Problem Based on Set-String Transformation

Yangjun Chen
University of Winnipeg, Canada

INTRODUCTION

In computer engineering, a number of programming tasks involve a special problem, the so-called *tree matching* problem (Cole & Hariharan, 1997), as a crucial step, such as the design of interpreters for nonprocedural programming languages, automatic implementation of abstract data types, code optimization in compilers, symbolic computation, context searching in structure editors and automatic theorem proving. Recently, it has been shown that this problem can be transformed in linear time to another problem, the so called *subset matching* problem (Cole & Hariharan, 2002, 2003), which is to find all occurrences of a pattern string p of length m in a text string t of length n, where each pattern and text position is a set of characters drawn from some alphabet Σ. The pattern is said to occur at text position i if the set $p[j]$ is a subset of the set $t[i + j - 1]$, for all j ($1 \leq j \leq m$). This is a generalization of the ordinary string matching and is of interest since an efficient algorithm for this problem implies an efficient solution to the tree matching problem. In addition, as shown in (Indyk, 1997), this problem can also be used to solve general string matching and counting matching (Muthukrishan, 1997; Muthukrishan & Palem, 1994), and enables us to design efficient algorithms for several geometric pattern matching problems. In this article, we propose a new algorithm on this issue, which needs only $O(n + m)$ time in the case that the size of Σ is small and $O(n + m \cdot n^{0.5})$ time on average in general cases.

BACKGROUND

The subset matching problem was defined in Cole and Hariharan (1997) and shown also in Cole and Hariharan (1997) and its improved version (Cole and Hariharan, 2003) that the well-known tree pattern matching problem can be linearly reduced to this problem. Formally, the text t is a string of length n and the pattern p is a string of length m. Each text position $t[i]$ and each pattern position $p[j]$ is a set of characters (not a single character), taken from a certain alphabet Σ. Strings, in which each location is a set of characters, will be called *set-strings* to distinguish them from ordinary strings. Pattern p is said to match text t at position i if $p[j] \subseteq t[i + j - 1]$, for all j ($1 \leq j \leq m$). As an example, consider the set-strings t and p shown in Figure 1.

Figure 1(a) shows a matching case, by which we have $p[j] \subseteq t[i + j - 1]$ for $i = 1$, and $j = 1, 2, 3$, while Figure 1(b) illustrates an unmatching case since for $i = 2$ we have $p[2] \not\subset t[i + 2 - 1]$.

Until now, the best way for solving subset matching is based on the construction of superimposed codes, or bit strings (see Chen, 2006; Faloutsos, 1995), for the characters in Σ and the convolution operation of vectors (Aho, Hopcroft, & Ullman, 1974). The superimposed codes are generated in such a way that no bit string (for a character) is contained in a Boolean sum of l other bit strings, where l is the largest size of the sets in both t and p. As indicated in Cole and Hariharan (2002), such superimposed codes can be generated in $O(n\log^2 m)$ time. In addition, by decomposing a subset matching into several smaller problems (see Cole & Hariharan, 1997), the convolution operation can also be done in $O(n\log^2 m)$ time by using Fourier transformation (Aho et al., 1974) (if the cardinality of Σ is bounded by a constant). Therefore, the algorithm discussed in Cole and Hariharan (2002) needs only $O(n\log^2 m)$ time.

In this article, we explore a quite different way to solve this problem. The main idea of our algorithm is to transform a subset matching problem into another subset matching problem by constructing a trie over the text string. In the new subset matching problem, t is reduced to a different string t', in which each position is an integer (instead of a set of characters); and p is changed to another string p', in which each position remains a set (of integers). This transformation gives us a chance to use the existing technique for string

Figure 1. Example of subset match

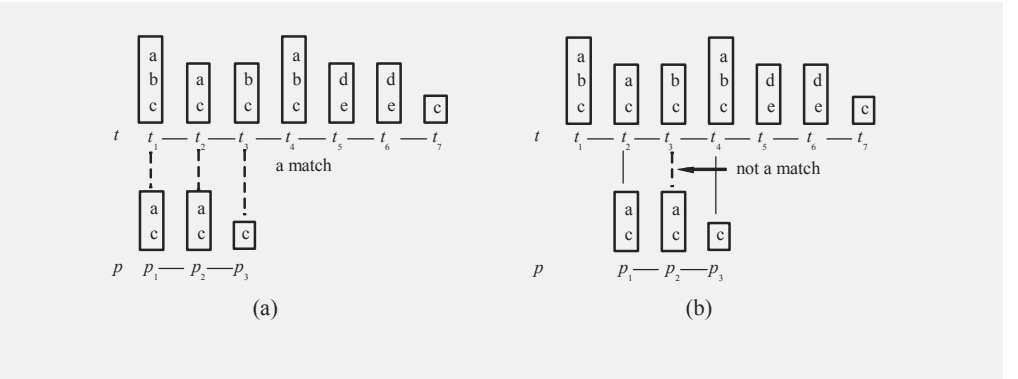

Figure 2. A 0-1 matrix for a text string

	1	2	3	4	5	6	7
a	1	1	0	1	0	0	0
b	1	0	1	1	0	0	0
c	1	1	1	1	0	0	1
d	0	0	0	0	1	1	0
e	0	0	0	0	1	1	0

matching to solve the problem. Concretely, we will generate a suffix tree over t' and search the suffix tree against p' in a way similar to the traditional methods. If the size of the alphabet is small, our method needs only $O(n + m)$ time. But in general cases, it needs $O(n + m \cdot n^{0.5})$ time on average.

The remainder of the article is organized as follows. In the second section, we discuss our algorithm, which is designed based on a transformation of subset matching problems. In the third section, we analyze the average time of a trie searching, which shows the average cost of our method. Finally, the fourth section is a short conclusion.

ALGORITHM DESCRIPTION

Assume that $\Sigma = \{1, ..., k\}$. We construct a 0-1 matrix $T = (a_{ij})$ for $t = t_1 t_2 \ldots t_n$ such that $a_{ij} = 1$ if $i \in t_j$ and $a_{ij} = 0$ if $i \notin t_j$ (see Figure 2 for illustration). In the same way, we construct another 0-1 matrix $P = (b_{ij})$ for $p = p_1 p_2 \ldots p_m$.

Then, each column in T (P) can be considered as a bit string representing a set in t (in p). (In the following discussion, we use $b(t_i)$ ($b(p_j)$) to denote the bit string for t_i (p_j).)

In a next step, we construct a (compact) trie over all $b(t_i)$'s, denoted by *trie*(T), as illustrated in Figure 3(a).

In this trie, for each node, its left outgoing edge is labeled with a string beginning with 0 and its right outgoing edge is labeled with a string beginning with 1; and each path from the root to a leaf node represents a bit string that is different from the others. In addition, each leaf node v in *trie*(T) is associated with a set containing all those t_i's that have the same string represented by the path from the root to v. Then, t can be transformed as follows:

- Number all the leaf nodes of the trie from left to right (see Figure 3 for illustration).
- Replace each t_i in t with an integer that numbers the leaf node, with which a set containing t_i is associated.

For example, the text string t shown in Figure 1(a) will be transformed into a string t' as shown in Figure 3(b), in which each position is an integer. For this example, t_1 and t_4 are replaced by 5, t_2 by 4, t_3 by 3, t_5 and t_6 by 1, and t_7 by 2.

In order to find all the sets in t, which contain a certain p_j, we will search *trie*(T) against $b(p_j)$ as below.

(i) Denote the ith position in $b(p_j)$ by $b(p_j)[i]$.

(ii) Let v (in *trie*(T)) be the node encountered and $b(p_j)[i]$ be the position to be checked. Denote the left and right outgoing edges of v by e_l and e_r, respectively.

- If $b(p_j)[i] = 1$, we will explore the right outgoing edge e_r of v. That is, we will compare the label of e_r, denoted by $l(e_r)$, with the corresponding substring in $b(p_j)$ according to the following criteria: if one bit in $b(p_j)$ is 1, the corresponding bit in $l(e_r)$ must be one; if one bit in $b(p_j)$ is 0,