Journal of Computer Science 3 (12): 924-933, 2007
ISSN 1549-3636

A New Algorithm for Subset Matching Problem

Yangjun Chen
University of Winnipeg, Winnipeg, Manitoba, Canada R3B 2E9

Abstract: The subset matching problem 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 ($1 \le j \le m$). This is a generalization of the ordinary string matching and can be used for finding matching subtree patterns. In this research, we propose a new algorithm that needs $O(n \cdot m)$ time in the worst case. But its average time complexity is $O(n + m \cdot n^{\log 1.5})$.

Key words: String matching, tree pattern matching, subset matching, trie, suffix tree, probabilistic analysis

INTRODUCTION

The subset matching problem is a generalization of the ordinary string matching problem, by which both the pattern and text are sequences of sets (of characters). 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 (see the definition given in[7]). 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 ($1 \le j \le m$). 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 $p[2] \not\subset t[i + 2 - 1]$.

This problem was defined in[5] and is of interest, as it was shown (also in[5] and its improved version[7]) that the well-known tree pattern matching problem can be linearly reduced to this problem. In addition, as shown in[8], this problem can also be used to solve general string matching and counting matching[10,11] and enables us to design efficient algorithms for several geometric pattern matching problems. Up to now, the best way for solving subset matching is based on the construction of superimposed codes (bit strings[3,4]) for the characters in Σ and the convolution operation of vectors[1]. The superimposed codes are generated in such way that no bit string (for a character) is contained in a boolean sum of k other bit strings, where k is the largest size of the sets in both t and p. As indicated in[6],

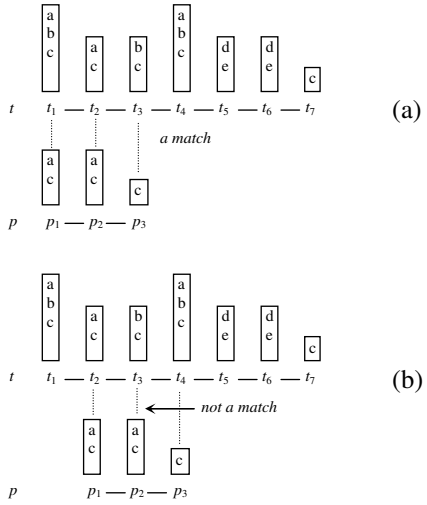


Fig. 1: Example of set match

such superimposed codes can be generated in $O(n \cdot \log^2 m)$ time. In addition, by decomposing a subset matching into several smaller problems[5], the convolution operation can also be done in $O(n \cdot \log^2 m)$ time by using Fourier transformation[1] (if the cardinality of Σ is bounded by a constant). Therefore, the algorithm discussed in[6] needs only $O(n \cdot \log^2 m)$ time.

In this research, 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 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. The algorithm runs in $O(n \cdot m)$ time in the worst case and in $O(n + m \cdot n^{\log 1.5})$ on average.

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 (resp. p). (In the following discussion, we use $b(t_i)$ ($b(p_j)$) to denote the bit string for t_i (resp. 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).

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

Fig. 2: A 0-1 matrix for a text string

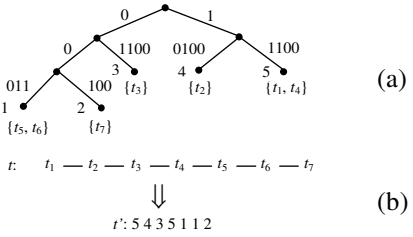


Fig. 3: A trie and set-string transformation

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).
- 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.

- Denote the ith position in $b(p_j)$ by $b(p_j)[i]$.
- 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. We do the following checkings:
  - If $b(p_j)[i] = 1$, we will explore the right outgoing edge e_r of v.
  - If $b(p_j)[i] = 0$, we will explore both e_l and e_r.

In fact, this definition just corresponds to the process of checking whether a set contains another as a subset. That is, if $b(p_j)[i] = 1$, we compare only 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, it does not matter whether the corresponding bit in $l(e_r)$ is 1 or 0. If they match, we move to the right child of v. If $b(p_j)[i] = 0$, we will check both $l(e_l)$ and $l(e_r)$.

For example, to find all the t_i's in the text string t shown in Figure 1(a), which match p_1 in p shown in the same figure, we will search the trie against $b(p_1) = 10100$. For this, part of the trie will be traversed as illustrated by the heavy lines in Figure 4(a).

This shows that in t there are three sets t_1, t_2 and t_4 containing p_1. But in t', t_1 and t_4 are represented by 5 and t_2 is represented by 4. So we associate $\{4, 5\}$ with p_1 and replace p_1 in p with $\{4, 5\}$. In this way, we will transform p into another string p', in which each position remains a set containing some integers that represent all those sets in t, which contain the corresponding set at the same position in p (see Figure 4(b) for illustration.) Each set in p' can be represented by a bit string of length l, where l is the number of different sets (t_i's) in t. If i belongs to the set, the ith position is set to 1; otherwise, it is set to 0.

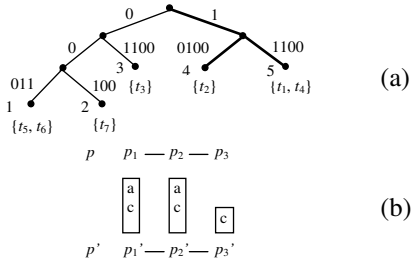
Formally, the above transformation defines two functions as below:

$$f_t: Set_t \rightarrow I,$$

where, Set_t is the set of all t_i's in t and $I = \{1, ..., l\}$; and $f_t(t_i) = a$ if a is the number for a leaf node in $trie(T)$, with which a set containing t_i is associated; and

$$f_p: Set_p \rightarrow 2^I,$$

where, Set_p is the set of all p_j's in p and 2^I is the set containing all the subsets of I, i.e., the power set of I; and $f_t(p_j) = b$ if b is a set of integers each labeling an leaf node in $trie(T)$, which is encountered when searching $trie(T)$ against $b(p_j)$.



(a)

(b)

Fig. 4: Trie searching and set-string transformation

Obviously, these two functions satisfy the following property.

Lemma 1: Let t_i be a set in t and p_j be a set in p. Assume that $f_t(t_i) = a$ and $f_p(p_j) = b$. Then, we have $t_i \supseteq p_j$ if and only if $a \in b$.

Proof: It can be directly derived from the above definition of the string transformations.

In a next step, we construct a suffix tree over $t' = t_1't_2' \ldots t_n'$, the transformed t, using a well-known algorithm such as the algorithms discussed in[12,13]. We remark that the alphabet for t' is $\{1, ..., l\}$ ($l \leq 2^k$) since each $t_i' \in \{1, ..., l\}$. It is a relatively large. But it is a sorted set, which is constructed when we number the leaf nodes of the trie for t. Therefore, the construction of the suffix tree over t' requires only $O(n)$ time. (More exactly, using McCreight's algorithm[12], we need $O(\log l \cdot n)$ time. $\log l = \log 2^k = k$.

For example, for the string shown in Figure 3(b), we will generate a suffix as shown in Figure 5.

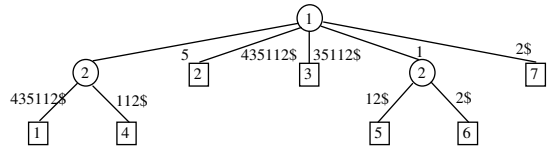


Fig. 5: The suffix tree for t'

In this tree, each internal node v is associated with an integer (denoted as $int(v)$) to indicate the position (in p') to be checked when searching; and each edge is labeled with a substring and all the labels along a path (from the root to a leaf node) form a suffix in t', plus \$, a special symbol, which makes every suffix not a prefix of any other. So a leaf node can be considered as a pointer to a suffix. In order to find all the substrings in t, which match p, we will explore the suffix tree for the transformed string t' against the transformed string $p' = p_1'p_2' \ldots p_m'$ as follows (note that each p_i' ($i \in \{1, ..., m\}$) is a set.)

- Search the suffix tree from the root.
- Let v be the node encountered and p_i' be the set to be checked.
- Let $e_1, ..., e_q$ be v's outgoing edges. Let $l(e_j) = l_{j_1} ... l_{j_h}$ (for some h) be the label of e_j ($1 \leq j \leq q$). Then, for any $e_j = (v_j, u_j)$ ($1 \leq j \leq q$), if $l_{j_1} \in p_{i+int(v)-1}'$, $l_{j2} \in p_{i+int(v)}'$, ... and $l_{j_h} \in p_{i+int(v)+h-1}'$, the subtree rooted at u_j will be continually explored. Otherwise, the subtree will not be searched any more. In addition, we notice that the symbol \$ is always ignored when we check the labels associated with the edges in $trie(T)$

In the above process, if we can find an edge $e = (v, u)$ with $l(e) = l_1 ... l_g ...$ such that l_g is checked against p_m' with $l_g \in p_m'$, any leaf node in the subtree rooted at u indicates a substring in t, which matches p.

The following is the formal description of the whole process.

Algorithm *Subset-Matching*
begin
1. Let $t = t_1t_2 ... t_n$ and $p = p_1p_2 ... p_m$;
2. Transform t to $t' = t_1't_2' ... t_n'$ and p to $p' = p_1' ... p_m'$ by using the trie constructed over t;
3. Construct a suffix tree t_{suffix} over t';
4. Search t_{suffix} against p';
5. **for** any $e = (v, u)$ in t_{suffix} with $l(e) = l_1 ... l_g ...$ such that l_g is checked against p_m' with $l_g \in p_m'$ **do**
6. {return all the leaf nodes in the subtree rooted at u;}

end

Example 1: By applying the above algorithm to the problem shown in Figure 1(a), $trie(T)$ shown in Figure 3(a) will be first generated and t will be transformed to t' as shown in Figure 3(b). Then, by searching $trie(T)$ against each p_i one by one, we will transform p to p' as

shown in Figure 4(b). The suffix tree for *t'* is shown in Figure 5. Finally, we will search the suffix tree against *p'* as shown by the heavy edges in Figure 6.

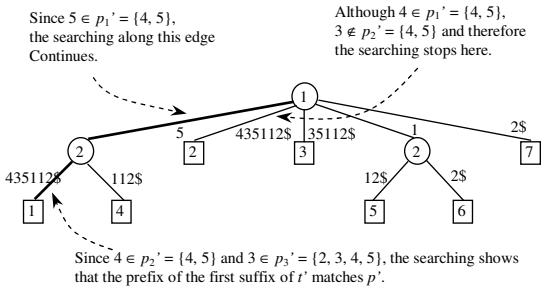


Fig. 6: Illustration for suffix searching

For this simple example, only one path in the suffix tree is explored. But multiple paths may be searched in general.

Proposition 1: Let $t = t_1t_2 \ldots t_n$ and $p = p_1p_2 \ldots p_m$. The algorithm *Subset-Matching* will find all t_i's ($1 \le i \le n - m$) such that $t_{i+1}t_{i+2}\ldots t_{i+m-1}$ matches p.

Proof: Let $n_1 \rightarrow n_2 \ldots \rightarrow n_m \rightarrow n_{m+1}$ be a path that is visited when searching the trie against *p'*. Let $l_i = l(n_i, n_{i+1})$ denote the label associated with the edge (n_i, n_{i+1}) ($1 \le i \le m$). Then, we must have $l_i \in p_i'$ ($1 \le i \le m$). In terms of Lemma 1, the substring in t: $f_t^{-1}(l_1) \ldots f_t^{-1}(l_m)$ definitely matches $f_p^{-1}(p_1') \ldots f_p^{-1}(p_m') = p_1p_2 \ldots p_m$. We remark that all the suffixes represented by the leaf nodes in the subtree rooted at n_{m+1} have $l_1 \ldots l_m$ as the prefix. So, each of these suffixes corresponds to a substring in t, which matches p.

The time complexity of the algorithm consists of four parts: C_1, C_2, C_3 and C_4, which are defined and estimated below.

C_1 is the time used for constructing the trie for t. In the case that Σ is fixed, it needs only $O(n)$ time.

C_2 is the time spent on generating *p'* for p. Let A represent the largest number of the edges visited when searching *trie*(T) against a $b(p_i)$ in p ($1 \le i \le m$). Then, C_2 is bounded by $O(A \cdot m)$.

C_3 is the cost for constructing the suffix tree over *t'*. It is $O(n)$.

C_4 is the cost for searching the suffix tree against *p'*. It is bounded by $O(A' \cdot m)$, where A' is the largest edges explored during the searching of the suffix tree.

Therefore, the whole computation process runs in time $O(n + A \cdot m + A' \cdot m)$. In the worst case, it is $O(m \cdot n)$. Averagely, however, both A and A' are on the order of $O(n^{\log 1.5})$ as shown in the subsequent section.

ANALYSIS OF A AND A'

In this section, we give a simple analysis of the average value of A. A precise probabilistic analysis is given in Section 5.

In order to analyze the average cost of A, we consider a 'worse' case that the trie is not compact, i.e., each edge is labeled with a single bit (instead of a bit string), which makes the analysis easier. In Figure 7(b), we show a non-compact trie for a set of bit strings shown in Figure 7(a).

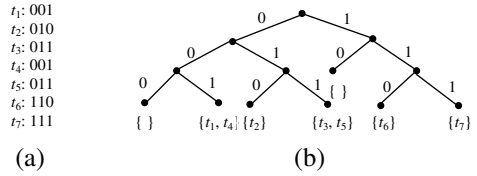


Fig. 7: A non-compact trie

For such a non-compact trie T, the searching of it against a bit string $s = s[1]s[2] \ldots s[k]$ is performed in a similar way to a compact trie, but simpler:

- Let v be the node encountered and $s[i]$ be the position to be checked.
- If $s[i] = 1$, we move to the right child of v.
- If $s[i] = 0$, both the right and left child of v will be visited.

In the following, we use $c_s(T)$ to represent the expected number of the edges visited when searching T against s. In addition, we use *s'*, *s''*, *s'''*, ... to designate the patterns obtained by circularly shifting the bits of s to the left by 1, 2, 3, ... positions.

Obviously, if the first bit of s is 0, we have, for the expected cost of a random string s,

$$c_s(T) = 1 + c_{s'}(T_1) + c_{s'}(T_2) \qquad (1)$$

where, T_1 and T_2 represent the two subtrees of the root of T. See Figure 8 for illustration.

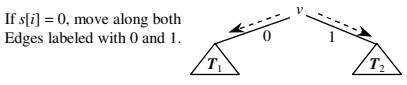


Fig. 8: Illustration for trie searching

It is because in this case, the search has to proceed in parallel along the two subtrees with s changing cyclically to s'.

If the first bit in s is 1, we have

$$c_s(T) = 1 + c_{s'}(T_2) \quad (2)$$

since in this case the search proceeds only in T_2.

In order to get the expectation of $c_s(T)$, we make the following assumption.

For each t_i in t, each element in t_i is taken from Σ with probability $p = 1/2$, independently from all the other t_j's and all the other elements in t_i.

Under this assumption, T_1 and T_2 will have almost the same size $\left\lfloor \frac{N}{2} \right\rfloor$, where N is the number of the nodes in T. So (1) and (2) can be rewritten as follows:

$$c_s(N) = 1 + 2c_{s'}(\left\lfloor \frac{N}{2} \right\rfloor), \quad (3)$$

and

$$c_s(N) = 1 + c_{s'}(\left\lfloor \frac{N}{2} \right\rfloor). \quad (4)$$

From (3) and (4), we get the following recurrence equation:

$$c_s(N) = 1 + \frac{3}{2} c_{s'}(\left\lfloor \frac{N}{2} \right\rfloor). \quad (5)$$

Solving the above recursion, we get

$$c_s(N) = O(1.5^{\log}N) = O(N^{\log 1.5}). \quad (6)$$

In terms of (6), we have the following proposition.

Proposition 2: A is on the order of $O(n^{\log 1.5})$.

Proof: The number of the nodes in *trie*(T) is bounded by $O(kn)$. So the average value of A is $O((kn)^{\log 1.5}) = O(n^{\log 1.5})$.

Since only $O(n^{\log 1.5})$ edges are visited on average when searching *trie*(T) against a $b(p_j)$ in p, the size of the set of all those t_i's that contain p_j is on the order of $O(n^{\log 1.5})$ and so is A.

IMPROVEMENTS

The above process can be significantly improved.

For p, we can also generate a trie over $b(p_j)$'s, denoted by *trie*(P), where P represents the 0-1 matrix for p, which is constructed in the same way as T for t. But for ease of control, we will establish non-compact tries for both t and p as illustrated in Figure 9.

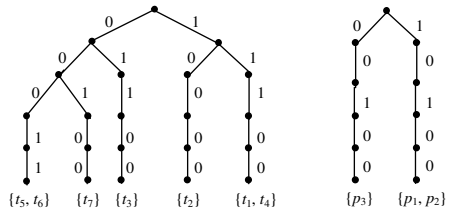


Fig. 9: Two non-compact tries

We will search these two tries simultaneously with the above containment checking simulated.

For this purpose, we will maintain a stack, *stack*, in which each entry is of the form $\{v, u\}$ with $v \in$ *trie*(T) and $u \in$ *trie*(P). During the process, each time we encounter a node v in *trie*(T) and a node u in *trie*(P), we will manipulate *stack* as below.

- Let v_1 and v_2 be two children of v with edge (v, v_1) labeled by 0 and edge (v, v_2) by 1; and u_1 and u_2 be two children of u with edge (u, u_1) labeled by 0 and edge (u, u_2) by 1;
- Push three pairs $\{v_2, u_2\}$, $\{v_2, u_1\}$ and $\{v_1, u_1\}$ (in the order specified) into stack.
- If v is a leaf node, put the number associated with v into a set associated with u to record the fact the sets represented by v contain the sets represented by u.

Below is the formal description of the algorithm. In the algorithm, the following two symbols are used:

- *Num*(v) - a number associated with a leaf node v in *trie*(T).
- *Matching*(u) - a sorted set (of integers) associated with a leaf node u in *trie*(P). Each integer in the set represents one or more sets in t, which contain the sets represented by u.

Algorithm *p-transformation*(*trie*(T), *trie*(P))
begin
1. $v_0 \leftarrow$ root of *trie*(T); $u_0 \leftarrow$ root of *trie*(P);
2. push(*stack*, $\{v_0, u_0\}$);
   (*push (v_0, u_0) into *stack*.*)
3. **while** *stack* not empty **do**
4. $\{\{v, u\} \leftarrow$ pop(*stack*);
5. **if** v is a leaf node **then**
   matching(u) $\leftarrow$ *matching*(u) $\cup$ {*num*(v)};
6. **else** {
7. let v_1 and v_2 be two children of v with (v, v_1) labeled by 0 and (v, v_2) by 1;

8. let u_1 and u_2 be two children of u with (u, u_1) labeled by 0 and (u, u_2) by 1;
9. push(*stack*, $\{v_2, u_2\}$); push(*stack*, $\{v_2, u_1\}$); push(*stack*, $\{v_1, u_1\}$);}
10. }

end

By the above algorithm, each p_j in p will be transformed to a set of integers. Applying this algorithm to the tries shown in Figure 5(a) and (b), we will get the same result as shown in Figure 4(b). But we search *trie*(T) against only two paths instead of three. In addition, p_1 and p_2 are replaced with the same set {4, 5}. So we implement P' as a pointer sequence with each pointer pointing to a set of integers.

In general, for all those p_j's that share the same prefix, the prefix is checked only once, which enables us to save much time.

The worse case time complexity C can be analyzed as follows.

Each pair $\{v, u\}$ generated during the process, v and u must be on the same level in *trie*(T) and *trie*(P), respectively. Let N_t be he numbers of different sets (t_i's) and N_p the numbers of different sets (p_j's) in p. We have

$$C = \sum_{i=1}^{k} num_T(i) \cdot num_P(i)$$
$$= N_T N_P \sum_{i=1}^{k} \frac{num_T(i) \cdot num_P(i)}{N_T \cdot N_P} = O(kN_t \cdot N_p),$$

where $num_T(i)$ ($num_P(i)$) represents the number of the nodes on level i in *trie*(T) (resp. in *trie*(P)).

Now we analyze the average time of this algorithm.

We simply use T and P to represent *trie*(T) and *trie*(P), respectively. Denote $root_T$ the root of T and $root_P$ the root of P. Let T_1 be the left subtree of $root_T$ and T_2 the right subtree of $root_T$. Let P_1 be the left subtree of $root_P$ and P_2 the right subtree of $root_P$. Then, we have the following recurrence equations:

$$C(T, P) = 1 + C(T_1, P_1) + C(T_2, P_1) + C(T_2, P_2), \quad (7)$$

(*$root_P$ has both the left and right child nodes.*)

$$C(T, P) = 1 + C(T_1, P_1) + C(T_2, P_1), \quad (8)$$

(*$root_P$ has only the left node.*)

$$C(T, P) = 1 + C(T_2, P_2), \quad (9)$$

(*$root_P$ has only the right child node.*)

where $C(T, P)$ represents the average number of the pairs (v, u) created during the process with $v \in T$ and $u \in P$.

From the above equations, we get

$$C(n, m) = 1 + 2C(\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{M}{2} \right\rfloor), \quad (10)$$

which leads to the following proposition.

Proposition 3: $C(n, m) \lesssim n^{\log 1.5} m^{\log 1.5}$.

Proof: We prove the proposition by induction on n and m.

Basic step: When $n = 1$ and $m = 1$, the proposition trivially holds.

Induction step: Assume that the propsotion holds for $a < n$ and $b < m$. That is, we have $C(a, b) \lesssim a^{\log 1.5} b^{\log 1.5}$ for any $a < n$ and any $b < m$. Then, in terms of (10) and the induction hypothesis, we have

$$C(n, m) = 1 + 2C(\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{M}{2} \right\rfloor)$$
$$\leq (1/(n^{\log 1.5} m^{\log 1.5}) + 1/(2^{\log 1.5} 2^{\log 1.5})) n^{\log 1.5} m^{\log 1.5}$$
$$= (1/(n^{\log 1.5} m^{\log 1.5}) + 1/2.25)\, n^{\log 1.5} m^{\log 1.5}.$$

For $n \geq 2$ and $m \geq 2$, $1/(n^{\log 1.5} m^{\log 1.5}) < 1/2.25$. So $C(n, m) \lesssim n^{\log 1.5} m^{\log 1.5}$.

Proposition 3 shows that the average cost the algorithm *p-transformation* is on the order of $O(n^{\log 1.5} m^{\log 1.5})$.

PROBABILISTICAL ANALYSIS

In terms of the analysis conducted in Section 3, we have the following two recurrences:

$$c_s(\boldsymbol{T}) = 1 + c_{s'}(\boldsymbol{T}_1) + c_{s'}(\boldsymbol{T}_2) \quad (11)$$

$$c_s(\boldsymbol{T}) = 1 + c_{s'}(\boldsymbol{T}_2) \quad (12)$$

where $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ represent the two subtrees of the root of $\boldsymbol{T}$.

Given N ($N \geq 2$) random nodes in $\boldsymbol{T}$, the probability that

$$|\boldsymbol{T}_1| = q, |\boldsymbol{T}_2| = N - q \quad (13)$$

can be estimated by the *Bernoulli* probabilities

$$\binom{N}{p}\left(\tfrac{1}{2}\right)^{p}\left(\tfrac{1}{2}\right)^{N-p} = \tfrac{1}{2^{N}}\binom{N}{p} \tag{14}$$

Let $c_{s,N}$ denote the expected cost of searching a trie of size N against s. We have the following recurrences

if s starts with 0,

$$c_{s,N} = 1+\frac{2}{2^{N}}\sum_{q}\binom{N}{1}c_{s',q}\ ,\ N \geq 2; \tag{15}$$

if s starts with 1,

$$c_{s,N} = 1+\frac{1}{2^{N}}\sum_{q}\binom{N}{1}c_{s',q}\ , N \geq 2. \tag{16}$$

Let $\lambda_i = 1$ if ith bit in s is 1 and $\lambda_i = 2$ if ith bit in s is 0. The above recurrence can be rewritten as follows

$$c_{s,N} = 1+\frac{\lambda_1}{2^{N}}\sum_{q}\binom{N}{1}c_{s',q} - \delta_{N,0} - \delta_{N,1}, \tag{17}$$

where $\delta_{N,j}$ (j = 0, 1) is equal to 1 if $N = j$; otherwise equal to 0.

Proposition 4: The exponential generating function of the average cost $c_{s,N}$

$$C_s(z) = \sum_{n\geq 0} c_{s,n}\frac{z^{n}}{n!} \tag{18}$$

satisfies the relation

$$C_s(z) = \lambda_1 e^{z/2}C_{s'}\left(\frac{z}{2}\right)+e^{z} - 1 - z. \tag{19}$$

Proof: In terms of equation (17), $C_s(z)$ can be rewritten as follows

$$\begin{aligned}
C_s(z) &= \sum_{n\geq 0}\left(1+\lambda_1\left(\tfrac{1}{2}\right)^{n}\sum_{p}\binom{n}{p}c_{s',p}-\delta_{n,0}-\delta_{n,1}\right)\frac{z^{n}}{n!}\\
&= \sum_{n\geq 0}\frac{z^{n}}{n!}+\sum_{p}\lambda_1\left(\tfrac{1}{2}\right)^{n}\sum_{n\geq 0}\binom{n}{p}c_{s',p}\frac{z^{n}}{n!}\\
&\quad - \sum_{n\geq 0}\delta_{n,0}\frac{z_{n}}{n!} - \sum_{n\geq 0}\delta_{n,1}\frac{z_{n}}{n!}\\
&= e^{z}+\lambda_1\sum_{p}\frac{(z/2)^{p}}{p!}\sum_{n\geq 0}c_{s',p}\frac{(z/2)^{n-p}}{(n-p)!} - 1 - z\\
&= \lambda_1 e^{z/2}C_{s'}\left(\frac{z}{2}\right)+e^{z} - 1 - z
\end{aligned} \tag{20}$$

In the same way, we will get $C_{s'}(z)$, $C_{s''}(z)$, ... and so on. Concretely, we will have the following equations:

$$C_s(z) = \lambda_1 e^{z/2}C_{s'}\left(\frac{z}{2}\right)+e^{z} - 1 - z, \tag{21}$$

$$C_{s'}(z) = \lambda_2 e^{z/2}C_{s''}\left(\frac{z}{2}\right)+e^{z} - 1 - z$$

$$C_{s(m-1)}(z) = \lambda_m e^{z/2}C_{s}\left(\frac{z}{2}\right)+e^{z} - 1 - z.$$

These equations can be solved by successive transportation. For instance, when we transport the expression of $C_{s'}(z)$ given by the second equation in (11), we have

$$C_s(z) = a(z)+\lambda_1 e^{z/2}\,a\left(\tfrac{z}{2}\right)+\lambda_1\lambda_2 e^{z/2}\,e^{z/2^{2}}\ C_{s''}\left(\tfrac{z}{2^{2}}\right) \tag{22}$$

where $a(z) = e^{z} - 1 - z$.

In a next step, we transport $C_{s'''}$ into the equation given in (22). This kind of transformation continues until the relation is only on C_s itself. Then, we have

$$\begin{aligned}
C_s(z) &= \lambda_1\lambda_2 \ldots \lambda_m exp\left[z\left(1-\frac{1}{2^{m}}\right)\right]C_s(\tfrac{z}{2^{m}}) +\\
&\quad \sum_{j=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_j exp\left[z\left(1-\tfrac{1}{2_j}\right)\right]\left(exp\left(\tfrac{z}{2^{j}}\right)-1-\tfrac{z}{2^{j}}\right)\\
&= 2^{m-k}\,exp\left[z\left(1-\frac{1}{2^{m}}\right)\right]C_s(\tfrac{z}{2^{m}}) +\\
&\quad \sum_{j=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_j exp\left[z\left(1-\tfrac{1}{2_j}\right)\right]\left(exp\left(\tfrac{z}{2^{j}}\right)-1-\tfrac{z}{2^{j}}\right)
\end{aligned} \tag{23}$$

where k is the number of 1s in s.

Let $\alpha = 2^{m-k}$, $\beta = 1-\frac{1}{2^{m}}$, $\lambda = \frac{1}{2^{m}}$ and

$$A(z) = \sum_{j=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_j exp\left[z\left(1-\tfrac{1}{2_j}\right)\right]\left(exp\left(\tfrac{z}{2^{j}}\right)-1-\tfrac{z}{2^{j}}\right).$$

We have

$$C_s(z) = \alpha e^{\beta z}C_s(\lambda z)+A(z) \tag{24}$$

This equation can be solved by iteration as discussed above:

$$C_s(z) = \sum_{j=0}^{\infty}\alpha^{j}exp\left(\beta\tfrac{1-\lambda^{j}}{1-\lambda}z\right)A\left(\lambda^{j}z\right) = \tag{25}$$

$$\sum_{j=0}^{\infty}2^{j(m-k)}\sum_{h=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_h\left[exp(z)-exp\left(z\left(1-\tfrac{1}{2^{h}2^{mj}}\right)\right)\left(1+\tfrac{z}{2^{h}2^{mj}}\right)\right]$$

Using Taylor formula to expand $exp(z)$ and $exp\left(z\left(1-\frac{1}{2^h 2^{mj}}\right)\right)\left(1+\frac{1}{2^h 2^{mj}}\right)$ in $C_s(z)$ given by the above sum and then extract the Taylor coefficients, we get

$$c_{s,N} = \sum_{h=0}^{m-1} \lambda_1\lambda_2\cdots\lambda_h \sum_{j\geq 0} 2^{j(m-k)} D_{jh}(n) \qquad (26)$$

where $D_{00}(n) = 1$ and for $j > 0$ and $h > 0$,

$$D_{jh}(n) = 1 - (1 - 2^{-mj-h})^n - n2^{-mj-h}(1 - 2^{-mj-h})^{n-1} \qquad (27)$$

We notice that $N \leq min\{n, 2^k\}$. So $c_{s,N} = O(N^{0.5}) \leq O(n^{0.5})$. This shows that the average time complexity of Algorithm *Set-Matching* is on the order of $O(n+ m\cdot n^{0.5})$. In the following, we show how to evaluate $c_{s,N}$.
First, we define

$$\phi(x) = \sum_{h=0}^{m-1} \lambda_1\lambda_2\cdots\lambda_h \sum_{j\geq 0} 2^{j(m-k)} D_{jh}(n)\ ,\ (x \geq 0) \qquad (28)$$

Then, we perform the following computations to evaluate $\phi(x)$:

1. Define the Mellin transformation of $\phi(x)$ ([8], p. 453):

$$\phi^*(\sigma) = \int_0^\infty \phi(x)x^{\sigma-1}dx \qquad (29)$$

2. Derive an expression for $\phi^*(\sigma)$, which reveals some of its singularities.
3. Evaluate the reversal Mellin transformation

$$\phi(x) = \frac{1}{2i\pi}\int_{c-i\infty}^{c+i\infty} \phi^*(\sigma)x^{-\sigma}d\sigma \quad -1 < c < -(1-\tfrac{k}{m}) \qquad (30)$$

The integral (30) is evaluated by using Cauchy's theorem as a sum of residues to the right of the vertical line $\{c + iy \mid y \in \Re\}$, where $\Re$ represents the set of all real numbers. This compuation method was first proposed in [14]. The following is just an extended explanation of it.

Remember that $D_{jh}(x) = 1 - (1 - 2^{-mj-h})^x - x2^{-mj-h}(1 - 2^{-mj-h})^{x-1}$. We rewrite it under the form

$$D_{jh}(x) = 1 - e^{-x\alpha_{jh}} - \beta_{jh}x e^{-x\alpha_{jh}} \qquad (31)$$

with $\alpha_{jh} = -\log(1 - 2^{-mj-h})$ and $\beta_{jh} = 2^{-mj-h}(1 - 2^{-mj-h})^{-1}$.

Now we consider the following expansion, which is valid for small values of x:

$$(-\log(1 - x))^{-\sigma} = x^{-\sigma}(1 - \frac{x\alpha}{2} + O(|\sigma|^2x^2)) \qquad .(32)$$

Let $x = 2^{-mj-h}$. Then, we have (by using the above expansion)

$$\alpha_{jh} = (-\log(1 - 2^{-mj-h}))^{-(-1)} \sim (2^{mj+h}). \qquad (33)$$

In addition, for small values 2^{-mj-h}, we also have

$$\beta_{jh} = 2^{-mj-h}(1 - 2^{-mj-h})^{-1} = O(2^{-mj}). \qquad (34)$$

Following the classical properties of Mellin transformation, we have the following proposition.

Proposition 5. Denote $D_{jh}^*(\sigma)$ the Mellin transformation of $D_{jh}(x)$. We have

$$D_{jh}^*(\sigma) = \int_0^\infty \phi(x)x^{\sigma-1}dx \qquad (35)$$

$$= -(\alpha_{jh})^{-\sigma}\Gamma(\sigma) - \beta_{jh}(\alpha_{jh})^{-\sigma-1}\sigma\Gamma(\sigma)$$

provided $-1 < Re(\sigma) < 0$, where $\Gamma(\sigma)$ is the *Euler Gamma* function.

Proof. The following formulas are well-known:

$$\int_0^\infty (e^{-x} - 1)x^{\sigma-1}dx = \Gamma(\sigma) \quad -1 < Re(\sigma) < 0 \qquad (36)$$

$$\int_0^\infty (xe^{-x})x^{\sigma-1}dx = \sigma\Gamma(\sigma) \quad -1 < Re(\sigma) \qquad (37)$$

$$\int_0^\infty f(ax)x^{\sigma-1}dx = a^{-\sigma}\int_0^\infty f(x)x^{\sigma-1}dx \text{ for } a > 0 \qquad (38)$$

In terms of these formulas, we have

$$D_{jh}^*(\sigma) = \int_0^\infty D_{jh}(x)x^{\sigma-1}dx \qquad (39)$$

$$= \int_0^\infty (1 - e^{-x\alpha_{jh}})x^{\sigma-1}dx - \int_0^\infty \beta_{jh}xe^{-x\alpha_{jh}}x^{\sigma-1}dx$$

$$= -(\alpha_{jh})^{-\sigma}\Gamma(\sigma) - \beta_{jh}(\alpha_{jh})^{-\sigma-1}\sigma\Gamma(\sigma).$$

Now we try to evaluate the following two sums:

$$\omega_h(\sigma) = \sum_{j\geq 0} 2^{j(m-k)}(\alpha_{jh})^{-\sigma}, \qquad (40)$$

$$\upsilon_h(\sigma) = \sum_{j\geq 0} 2^{j(m-k)}\beta_{jh}(\alpha_{jh})^{-\sigma-1}.$$

From (33) and (34), we can see that the two sums given by (40) are uniformly and absolutely convergent when σ is in the following stripe:

Stripe: $-1 < \mathrm{Re}(\sigma) < -(1 - \frac{k}{m})$. (41)

Furthermore, in terms of (33) and (34), both $\omega_h(\sigma)$ and $\upsilon_h(\sigma)$ can be approximated by the following sum:

$$\varpi_h(\sigma) = \sum_{j\geq 0} 2^{j(m-k)}(2^{mj+h})^{\sigma} \quad (42)$$

When $\mathrm{Re}(\sigma) < \sigma_0 = -(1 - \frac{k}{m})$, this series can be summed exactly:

$$\varpi_h(\sigma) = 2^{h\sigma}\frac{1}{1-2^{m-k+m\sigma}}. \quad (43)$$

Thus, $\phi^*(\sigma)$ is defined in *Stripe* and can be computed as follows

$$\phi^*(\sigma) = \int_0^{\infty}\phi(x)x^{\sigma-1}dx \quad (44)$$

$$= \int_0^{\infty}(\sum_{h=0}^{m-1}\lambda\lambda\cdots\lambda\sum_{j\geq 0}2^{j(m-k)}D_{jh}(x))x^{\sigma-1}dx$$

$$= -\sum_{h=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_h(\omega_h(\sigma)+\sigma\upsilon(\sigma))\Gamma(\sigma)$$

$$= -\Gamma(\sigma)(1+\sigma)\sum_{h=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_h 2^{h\sigma}\frac{1}{1-2^{m-k+m\sigma}}.$$

From this, we can observe all the sigularities (poles), i.e., $\sigma = 0$, at which $\Gamma(\sigma)$ is not defined; and all those values of σ, at which $(1 - 2^{m(\sigma-\sigma_0)})$ becomes 0:

$$\sigma_j = \sigma_0 + \frac{2ij\pi}{m\log 2}, (j = 0, \pm 1, \pm 2, ...) \quad (45)$$

To compute the integral in (21), we consider the following integral

$$\phi_N(x) = \frac{1}{2i\pi}\int_{L_N}\phi^*(\sigma)x^{-\sigma}d\sigma, \quad (46)$$

where L_N is a rectangular contour oriented clockwise as shown in Figure 10.

$$L_N = L_N^1 + L_N^2 + L_N^3 + L_N^4, \quad (47)$$

$$L_N^1 = \left\{c + iu \middle| u| \leq \frac{(2N+1)\pi}{m\log 2}\right\},$$

$$L_N^2 = \left\{v + i\frac{(2N+1)\pi}{m\log 2}\middle| c \leq v \leq \frac{k}{3m}\right\},$$

$$L_N^3 = \left\{\frac{k}{3m} + iu \middle| u| \leq \frac{(2N+1)\pi}{m\log 2}\right\},$$

$$L_N^4 = \left\{v - i\frac{(2N+1)\pi}{m\log 2}\middle| c \leq v \leq \frac{k}{3m}\right\},$$

where N is an integer. This contour is of a similar type used in ([9], p. 132).

Let ϕ_N^i be the integral along L_N^i (i = 1, 2, 3, 4). Then, $\phi_N(x) = \phi_N^1(x) + \phi_N^2(x) + \phi_N^3(x) + \phi_N^4(x)$. Furthermore, we have the following results:

$$\lim_{N\to\infty}\phi_N^1(x) = \phi(x),$$

$$\lim_{N\to\infty}\phi_N^2(x) = \mathrm{O}(1),$$

$$|\phi_N^3(x)| \leq x^{-k/(3m)}\int_{L_\infty}|\phi^*(\sigma)|d\sigma$$

$= \mathrm{O}(x^{-k/(3m)})$, and

$$\lim_{N\to\infty}\phi_N^4(x) = \mathrm{O}(1).$$

Thus, we have

$$\lim_{N\to\infty}\phi_N(x) = \phi(x) + \mathrm{O}(x^{-k/(3m)}). \quad (48)$$

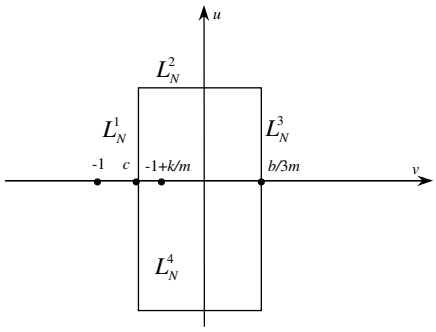


Fig. 10: The rectangular contour L_N

On the other hand, $\lim_{N\to\infty}\phi_N(x)$ can be evaluated as the sum of the residues of the integrand, i.e., $\phi^*(\sigma)x^{-\sigma}$, inside L_N. Concretely, we have

$$\lim_{N\to\infty}\phi_N(x) = -\sum_{\alpha\in Pole(\phi^*(\sigma))}(\phi^*(\sigma)x^{-\sigma}, \sigma = \alpha) \quad (49)$$

$$= -\sum_{\alpha\in Pole(\phi^*(\sigma))}\lim_{\sigma\to\alpha}(\sigma-\alpha)\phi^*(\sigma)x^{-\sigma}.$$

Within L_∞, $\phi^*(\sigma)$ has the following poles:

$\alpha = 0$, and

$\alpha = \sigma_j = \sigma_0 + \frac{2ij\pi}{m\log 2}$ ($j = 0, \pm 1, \pm 2, ...$)

The contribution of the pole $\alpha = 0$ is O(1); and the contribution of $\alpha = \sigma_0$ is

$$\lim_{\sigma\to\sigma_0}(\sigma-\sigma_0)\phi^*(x)x^{-\sigma} \quad (50)$$

$$= x^{-\sigma_0}\frac{(1+\sigma_0)\Gamma(\sigma_0)}{m\log 2}\sum_{h=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_h 2^{h\sigma_0}.$$

Finally, the contribution of each σ_j ($j = 0, \pm 1, \pm 2, ...$)

$$\lim_{\sigma \to \sigma_0} (\sigma - \sigma_0)\phi^*(x)x^{-\sigma} \tag{51}$$

$$= x^{-\sigma_0} exp(-\frac{2ij\pi}{m}\log_2 x)(1+\sigma_j)\Gamma(\sigma_j)\sum_{h=0}^{m-1}\lambda_2\lambda_2\cdots\lambda_h 2^{h\sigma_0} \ .$$

So we have

$$\lim_{N \to \infty} \phi_N(x) \tag{52}$$

$$= x^{-\sigma_0}\frac{(1+\sigma_0)\Gamma(\sigma_0)}{m\log 2}\sum_{h=0}^{m-1}\lambda\lambda\cdots\lambda 2^{h\sigma_0} +$$

$$\sum_{j=-\infty}^{-1} x^{-\sigma_0} exp(-\frac{2ij\pi}{m}\log_2 x)(1+\sigma_j)\Gamma(\sigma_j)\sum_{h=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_h 2^{h\sigma_j}$$

$$+ \sum_{j=1}^{+\infty} x^{-\sigma_0} exp(-\frac{2ij\pi}{m}\log_2 x)(1+\sigma_j)\Gamma(\sigma_j)\sum_{h=0}^{m-1}\lambda_1\lambda_2\cdots\lambda_h 2^{h\sigma_j} =$$

$$x^{-\sigma_0}\frac{(1+\sigma_0)\Gamma(\sigma_0)}{m\log 2}\sum_{h=0}^{m-1}\lambda\lambda\cdots\lambda 2^{h\sigma_0} \ .$$

From this, we know that

$$C_{s,n} = O(n^{-\sigma_0}) = O(n^{1-\frac{k}{m}}).$$

CONCLUSION

In this research, a new algorithm for the subset-matching problem is proposed. The main idea of the algorithm is to represent each set t_i in the text string t as a single integer a and each set p_j in the pattern string p as a set b of integers such that $a \in b$ if and only if $p_j \subseteq t_i$. This is done by constructing a trie structure over t. In this way, we transform the original problem into a different subset matching problem, which can be efficiently solved by generating a suffix tree over the new text string that has an integer at each position. In the worst case, the algorithm runs in $O(n + l \cdot m)$ time, where l is the number of different sets (t_i's) in t. But its average time complexity is $O(n + m \cdot n^{\log 1.5})$.

REFERENCES

1. Aho, A.V., J.E. Hopcroft and J.D. Ullman, 1974. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Com., London.
2. Churchill, R.V., 1958. Operational Mathematics. McGraw-Hill Book Company, NewYork.
3. Faloutsos, C., 1985. Access Methods for Text. ACM Computing Surveys, 17 (1): 49-74.
4. Faloutsos, C., 1992. Signature Files, In: Information Retrieval: Data Structures and Algorithms, Frakes W.B. and R. Baeza-Yates, (Ed.). Prentice Hall, New Jersey, pp: 44-65.
5. Cole, R. and R. Hariharan, 1997. Tree pattern matching and subset matching in randomized O(n log^3 m) time. Proceedings of the Twenty Ninth Annual Symposium on the Theory of Computing, pp: 66-75.
6. Cole, R. and R. Hariharan, 2002. Verifying candidate matches in sparse and wildcard matching, in Proc. of the 34th ACM Symposium on Theory of Computing, Montreal, QC, Canada, pp: 592-601.
7. Cole, R. and R. Hariharan, 2003. Tree pattern matching to subset matching in linear time, SIAM, J. Comput. 2 (4): 1056-1066.
8. Indyk, P., 1997. Deterministic superimposed coding with applications to pattern matching. In Proceeding. 38th Annual Symposium on Foundations of Computer Science, Florida, USA, pp: 127-136.
9. Knuth, D.E., 1973. The Art of Computer Programming: Sorting and Searching, Addison-Wesley Pub. London.
10. Muthukrishan, S. and K. Palem, 1994. Non-standard Stringology: Algorithms and Complexity, in Proceeding 26th ACM Symposium on Theory of Computing, pp: 770-779.
11. Muthukrishan, S., 1995. New results and open problems related to non-standard stringology, proceeding 6th annual symposium on combinatorial pattern matching, pp: 298-317.
12. McCreight, E., 1976. A space-economical suffix tree construction algorithm, J. ACM 23 (2): 262-272.
13. Ukkonen, E., 1995. Constructing suffix tree on-line in linear time, Algorithmica 14 (3): 249-260.
14. Flajolet, P. and C. Puech, 1986. Partial match retrieval of multidimentional data, J. ACM, 33 (2) 371-407.