ASTES

Advances in Science, Technology and Engineering Systems Journal
Vol. 9, No. 4, 60-72 (2024)
www.astesj.com

ASTES Journal
ISSN: 2415-6698

On Mining Most Popular Packages

Yangjun Chen*, Bobin Chen

Department of Applied Computer Science, University of Winnpeg, Manitoba, R3B 2E9, Canada

ARTICLE INFO

Article history:
Received: 22 April, 2024
Revised: 11 June, 2024
Accepted: 31 July, 2024
Online: 07 August, 2024

Keywords:
Data Mining
Most popular packages
NP-complete
Priority-first tree search
Tries
Trie-like graphs

ABSTRACT

In this paper, we will discuss two algorithms to solve the so-called package design problem, by which a set of queries (referred to as a query log) is represented by a collection of bit strings with each indicating the favourite activities or items of customers. For such a query log, we are required to design a package of activities (or items) so that as many customers as possible can be satisfied. It is a typical problem of data mining. For this problem, the existing algorithm requires at least $O(n2^m)$ time, where m is the number of activities (or items) and n is the number of queries. We try to improve this time complexity. The main idea of our first algorithm is to use a new tree search strategy to explore the query log. Its average time complexity is bounded by $O(nm^2 + m2^{m/2})$. By our second algorithm, all query bit strings are organized into a graph, called a trie-like graph. Searching such a graph bottom-up, we can find a most popular package in $O(n^2m^3(log_2\ nm)^{log_2\ nm})$ time. Both of them work much better than any existing strategy for this problem.

1. Introduction

Frequent pattern mining plays an important role in mining associations [1, 2, 3, 4], which are quite useful for decision making. For instance, for the supermarket management, the association rules are used to decide, how to place merchandise on shelves, what to put on sale, as well as how to design coupons to increase the profit, etc.

In general, by the frequent pattern mining [5, 6, 7, 8], we are required to find a frequent pattern, which is in fact a subset of items supported (or say, contained) by most of transactions. Here, by a transaction, we mean a set of attributes or items.

In this paper, we study a more challenging problem, the so-called *single package design* problem (SPD for short [2, 3, 9, 10]), defined below:

- A set of attributes (items, or activities):

  $A = \{a_1, ..., a_m\}$,

- A query log:

  $Q = \{q_1, \ldots\ q_n\}$,

  where each $q_i = c_{i1}c_{i2}\ \ldots\ c_{im}$ with each $c_{ij} \in \{0, 1, *\}$ ($i = 1, \ldots, n$, $j = 1, \ldots, m$).

- In q_i, whether a_j is chosen, depends on the value of c_{ij}. That is, if $c_{ij} = 1$, a_j is selected; if $c_{ij} = 0$, a_j is not selected; otherwise, c_{ij} = '*' means 'don't care'.

Our purpose is to find a bit string $\tau = \tau_1\ ...\ \tau_m$ that satisfies as many queries q_i's in Q as possible. We say, τ satisfies a $q_i = c_{i1}c_{i2}\ \ldots\ c_{im}$ if for each j ($1 \le j \le m$) the following conditions are satisfied:

$c_{ij} = 1 \rightarrow \tau_j = 1$,
$c_{ij} = 0 \rightarrow \tau_j = 0$,
c_{ij} = '*' $\rightarrow \tau_j = 1$ or 0.

A τ is referred to as a package. If it is able to satisfy a maximum subset of queries, we call it a *most popular* package. For instance, for the above vacation package, a query in a log can be created by specifying yes, no, or 'don't care' for each activity by a client. Then, the design of a most popular package is essentially to decide a subset of such activities to satisfy as many queries' requirements (normally according to a questionnaire) as possible. It is a kind of extension to mining association rules in data mining [5], but more general and therefore more useful in practice.

This problem has been investigated by several researchers [9, 10]. The method discussed in [10] is an approximation algorithm, based on the reduction of SPD to MINSAT [11], by which we seek to find a truth assignment of variables in a logic formula (in conjunctive normal form) to minimize the number of satisfied clauses. This is an optimization version of the satisfiability problem [12]. In [9], a kind of binary trees, called *signature trees* [13, 14, 15] for *signature files* [16, 17, 18, 19, 20], is contructed to represent query logs. Its worst-case time complexity is bounded by $O(n2^m)$.

In this paper, we address this issue and discuss two different algorithms to solve the problem. By the first method, we will con-

*Corresponding Author: Department of Applied Computer Science, University of Winnpeg, Manitoba, Canada, R3B 2E9 & y.chen@uwinnipeg.ca

struct a binary tree over a query log in a way similar to [9], but establishing a kind of heuristics to cut off futile branches. Its average time complexity is bounded by $O(nm^2 + m2^{m/2})$. The second algorithm is based on a compact representation of the query log, by which all the query bit strings are organized into a trie-like graph G. Searching G bottom-up recursively, we can find a most popular package in $O(n^2m^3(log_2\ nm)^{log_2\ nm})$ time.

The remainder of the paper is organized as follows. First, we show a simple example of the SPD problem in Section 2. Then, Section 3 is devoted to the discussion of our first algorithm for solving the SPD problem, as well as its time complexity analysis. Next, in Section 4, we discuss our second algorithm in great detail. Finally, we conclude with a summery and a brief discussion on the future work in Section 5.

2. An example of SPD

In this section, we consider a simple SPD shown in Table 1, which contains a query log with $n = 6$ queries, and $m = 6$ attributes (activities), created based on a questionnaire on customers' favourites. For example, the query $q_6 = a_{11}a_{12} \ldots\ a_{16}$ = (*, 1, *, 0, *, 1) in Table 1 shows that *ride* and *boating* are q_6's favourites, but *hike* is not. In addition, q_6 does not care about whether *hot spring*, *glacier* or *airline* is available or not.

Table 1: A query log Q

query	hot spring	ride	glacier	hike	airline	boating
q_1	1	*	0	*	1	*
q_2	1	0	1	*	*	*
q_3	*	0	0	1	1	*
q_4	0	*	1	*	1	*
q_5	*	0	0	*	*	0
q_6	*	1	*	0	*	1

For this small query log, a most popular package can be found, which contains three items: *hot spring*, *hiking*, *airline*, and is able to satisfy a maximum subset of queries: q_1, q_3, q_5.

3. The First Algorithm

In this section, we discuss our first algorithm. First, in Section 3.1, we give a basic algorithm to provide a discussion background. Then, we describe this algorithm in great detail in Section 3.2. The analysis of the average running time is conducted in Section 3.3.

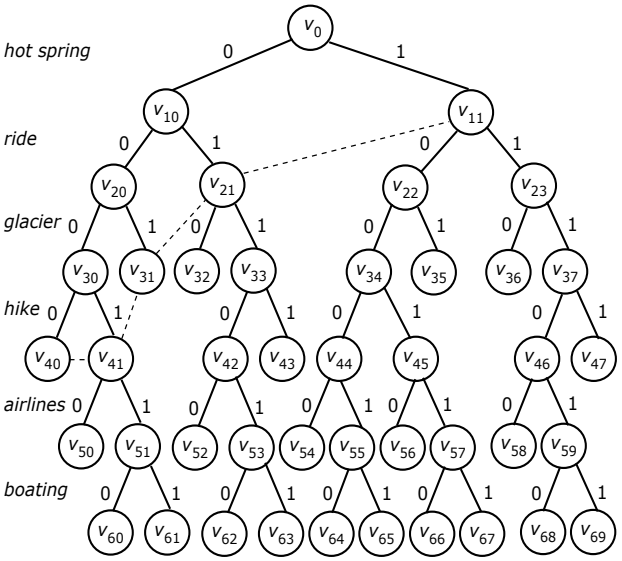
3.1. Basic algorithm

We first describe a basic algorithm to facilitate the subsequent discussion, which is in fact an extension of an algorithm discussed in [15]. The main idea behind it is the construction of a binary tree T over a query log Q. The algorithm works in two steps. In the first step, a signature-tree-like structure is built up, referred to as a *search-tree*. Then, in the second step, the search-tree is explored to find a most popular package.

Given a set of attributes: $A = \{a_1, a_2, \ldots, a_m\}$ and a query log: $Q = \{q_1, \ldots, q_n\}$ over A. Denote by $q_i[j]$ the value of the jth attribute a_j in q_i ($i = 1, \ldots, m$). Then, the binary tree T can be constructed as follows.

1. First, for the whole Q, create *root* of T. $j := 1$.
2. For each leaf node v of the current T, denote by s_v the subset of queries represented by v. For query q_i ($\in s_v$), if $q_i[1]$ = '0', we put q_i into the left branch. If $q_i[1]$ = '1', it is put into the right branch. However, if $q_i[1]$ = '*', we will put it in both left and right branches, showing a quite different behavior from a traditional signature tree construction [15].
3. $j := j + 1$. If j ¿ m, stop; otherwise, go to (2).

For example, for the query log given in Table 1, we will construct a binary search tree as shown in Fig. 1.



$s_0 = \{q_1, q_2, q_3, q_4, q_5, q_6\}$ $s_{01} = \{q_1, q_2, q_5, q_6\}$ $s_{11} = \{q_3, q_4, q_5, q_6\}$
$s_{20} = \{q_1, q_2, q_3, q_5\}$ $s_{21} = \{q_1, q_5\}$ $s_{22} = \{q_3, q_4, q_5\}$ $s_{23} = \{q_4, q_6\}$
$s_{30} = \{q_3, q_5\}$ $s_{31} = \{q_4\}$ $s_{32} = \{q_6\}$ $s_{33} = \{q_4, q_6\}$ $s_{34} = \{q_1, q_3, q_5\}$
$s_{35} = \{q_2\}$ $s_{36} = \{q_1, q_6\}$ $s_{37} = \{q_6\}$
$s_{40} = \{q_1\}$ $s_{41} = \{q_4, q_5\}$ $s_{42} = \{q_4, q_6\}$ $s_{43} = \{q_4\}$ $s_{44} = \{q_1, q_5\}$
$s_{46} = \{q_1, q_6\}$ $s_{45} = \{q_1, q_3, q_5\}$ $s_{47} = \{q_1\}$
$s_{50} = \{q_3\}$ $s_{51} = \{q_3, q_5\}$ $s_{52} = \{q_6\}$ $s_{53} = \{q_4, q_6\}$ $s_{54} = \{q_3\}$ $s_{55} = \{q_1, q_5\}$
$s_{56} = \{q_5\}$ $s_{57} = \{q_1, q_3, q_5\}$ $s_{58} = \{q_1\}$ $s_{59} = \{q_1\}$
$s_{60} = \{q_3, q_5\}$ $s_{61} = \{q_3\}$ $s_{62} = \{q_4\}$ $s_{63} = \{q_4, q_6\}$ $s_{64} = \{q_1, q_3\}$
$s_{65} = \{q_1\}$ $s_{66} = \{q_1, q_3, q_5\}$ $s_{67} = \{q_1, q_3\}$ $s_{68} = \{q_1\}$ $s_{69} = \{q_1, q_6\}$

Figure 1: A search tree.

In Fig. 1, we use s_u to represent the subset of queries associated with u. In terms of the corresponding attribute a, s_u is decomposed into two subsets: $s_u(a)$ and $s_u(\neg a)$, where for each $q \in s_u(a)$ we have $q[a] = 1$ and for each $q' \in s_u(\neg a)$ $q'[a] = 0$. In general, $s_u(\neg a)$ is represented by u's left child while $s_u(a)$ is represented by u's right child.

For example, the subset of queries associated with v_{11} is s_{11} = {q_3, q_4, q_5, q_6}. According to attribute '*ride*', s_{11} is split into two subsets respectively associated with its two children (v_{22} and v_{23}): s_{22} = {q_3, q_4, q_5} and s_{23} = {q_4, q_6}. In addition, we can also see that among all the leaf nodes the subset s_{66} (= {q_1, q_3, q_5}) associated with v_{66} is of the largest size. Then, the labels along the path from the root to it spell out a string 100110, representing a most popular package: {*hot spring*, *hiking*, *airlines*}.

The computational complexity of this process can be analyzed as follows.

First, we notice that in the worst case an search-tree can have O(2^m) nodes. Since each node is associated with a subset of queries, we need O(n) time to determine its two children. So, the time for constructing such a tree is bounded by O($n2^m$). The space requirement can be slightly improved by keeping only part of the search tree in the working process. That is, we need only to maintain the bottom frontier (i.e., the last nodes on each path at any time point during the construction of T.) For example, nodes v_{40}, v_{41}, v_{31}, and v_{21} (see the dashed lines in Fig. 1) make up a bottom frontier at a certain time point. At this point, only these nodes are kept around. However, for each node v on a bottom frontier, we need to keep the bit string along the path from *root* to v to facilitate the recognition of the corresponding best package. In the worst case, the space overhead is still bounded by O($n2^m$).

3.2. Algorithm based on priority-first search

The basic algorithm described in the previous section can be greatly improved by defining a partial order over the nodes in the search tree T to cut off futile paths. For this purpose, we will associate a key with each node v in T, which is made up of two values: <|s_v|, l_v>, where |s_v| is the subset of queries associated with v and l_v is the level of v. (Here, we note that the level of the root is 0, the level of the root's children is 1, and so on.) In general, we say that a pair <|s_v|, l_v> is larger than another pair <|s_u|, l_u> if one of the following two conditions is satisfied:

- |s_v| > |s_u|, or
- |s_v| = |s_u|, but l_v ¿ l_u.

In terms of this partial order, we define a *max-priority queue H* for maintaining the nodes of T to control the tree search, with the following two operations supported:

- *extractMax*(H) removes and returns the node of H with the largest pair.
- *insert*(H, v) inserts the node v into the queue H, which is equivalent to the operation $H := H \cup \{v\}$.

In addition, we utilize a kind of heuristics for efficiency, by which each time we expand a node v, the next attribute a chosen among the remaining attributes should satisfy the following conditions:

1. |$s_v(a)$| - |$s_v(\neg a)$| is maximized.
2. In the case that more than one attributes satisfy condition (1), choose a from them such that the number of queries q in s_v with $q[a]$ = '*' being minimized (the tie is broken arbitrarily.)

Using the above heuristics, we can avoid as many useless branches as possible.

By using the priority queue, the exploration of T is not a *DFS* (depth-first search) any more. That is, the search along a current path can be cut off, but continued along a different path, which may lead to a solution quickly (based on an estimation made according to the pairs associated with the nodes.) This is because by *extractMax*(H) we always choose a node with the largest possibility leading to a most popular package.

Algorithm 1: *PRIORTY-SEARCH*(Q, A)

```
Input  : a query log Q.
Output : a most popular package P.
the pair for root is set to be <|Q|, 0>; i := 0;
inset(H, root); (*root represents the whole Q.*)
while i ≤ m do
    v := extractMax(H); (*recall that the pair associated
      with v is <|s_v|, l_v>.*)
    if i = m then
        return the package represented by the path from
          root to v;
    recognize a next attribute a from A according to
      heuristics;
    generate left child v_l of v, representing s_v(¬a);
    create left child v_r of v, representing s_v(a);
    the pair of v_l is set to be <|s_v(¬a)|, l + 1>;
    the pair of v_r is set to be <|s_v(a)|, l + 1>;
    insert(H, v_l); insert(H, v_r);
    i := l_v + 1;
```

The procedure is given in Algorithm 1. This is in fact a tree search controled by using a priority queue, instead of a stack. First, the *root* is inserted into the priority queue H and its key (a pair of values) is set to be <|Q|, 0>. Then, we will go into a **while**-loop, in each iteration of which we will extract a node v from the priority queue H with the largest key value (line 4), that is, with the largest number of queries represented by v and at the same time on the deepest level (among all the nodes in H). Then, the subset of queries represented by v will be split (see lines 7 and 8) according to a next attribute chosen in terms of the heuristics described above (line 7). Next, two children of v, denoted respectively as v_l and v_r, will be created (see lines 8 and 9). and their keys are calculated (lines 10 and 11). In line 12, these two children are inserted into H. Finally, we notice that i is used to record the level of the currently encountered node. Thus, when $i = m$, we must get a most popular package.

The following example helps for illustration.

Example 1. *1 In this example, we show the first three steps of computation when applying the algorithm PRIORITY-SEARCH( ) to the query log given in Table 1. For simplicity, we show only the nodes in both H and T for each step. In addition, the priority queue is essentially a max-heap structure [21], represented as a binary tree.*

In the first step, (see Fig. 2(a)), the *root* (v_0) of T is inserted into H, whose pair is <6, 0>. It is because the *root* represents the whole query log Q which contains 6 queries and is at level 0. Then, in terms of attribute *glacier* (selected according to the heuristics), Q

is split into two subsets (which may not be disjoint due to possible '*' symbols in queries), which are stored as v_0's two child nodes: v_{10} and v_{11} with $s_{v_{10}} = \{q_1, q_3, q_5, q_6\}$ and $s_{v_{11}} = \{q_2, q_4, q_6\}$. Then, v_{10} with pair <4, 1> and v_{11} with <3, 1> will be inserted into H.

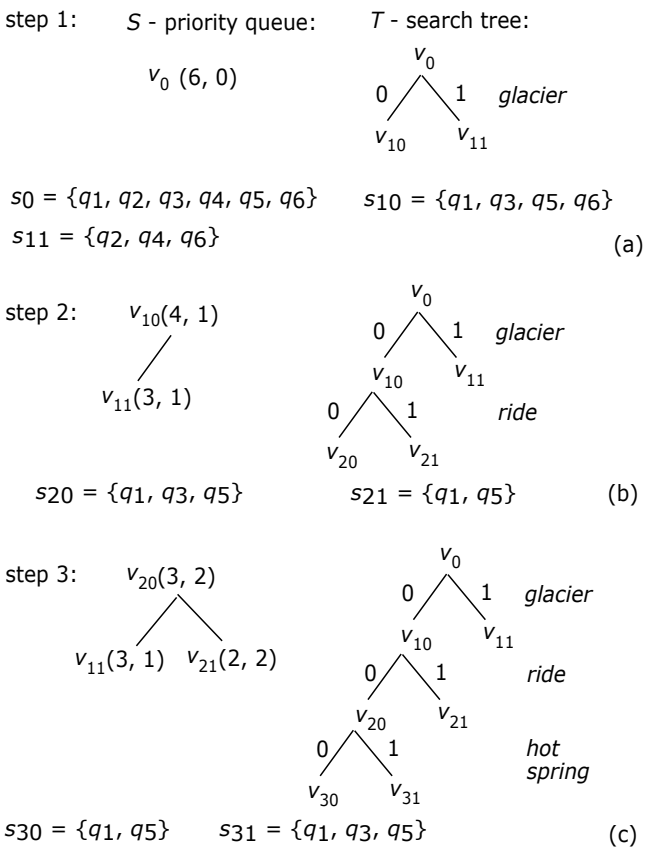


Figure 2: A sample trace.

In the second step (see Fig. 2(b)), v_{10} with pair <4, 1> will be extracted from H. This time, in terms of the heuristics, the selected attribute is *ride*, and $s_{v_{10}}$ will accordingly be further divided into two subsets, represented by its two children: v_{20} with $s_{v_{20}} = \{q_1, q_3, q_5\}$ and v_{21} with $s_{v_{21}} = \{q_1, q_5\}$. Their pairs are respectively <3, 2> and <2, 2>.

In the third step (see Fig. 2(c)), v_{20} with pair <3, 2> will be taken out from H. According to the selected attribute *hot spring*, $s_{v_{20}}$ will be divided into two subsets, represented respectively by its two children: v_{30} with $s_{v_{30}} = \{q_1, q_5\}$ and v_{31} with $s_{v_{31}} = \{q_1, q_3, q_5\}$.

The last step of the computation is illustrated in Fig. 3, where special attention should be paid to node v_{60}, which is associated with a subset of queries: $\{q_1, q_3, q_5\}$, larger than any subset in the current priority queue H. Then, it must be one of the largest subset of queries which can be found since along each path the sizes of subsets of queries must be non-increasingly ordered. Now, by checking the labels along the path from the root to v_{60}, we can easily recognize all the attributes satisfying the queries in this subset. They are {*hot spring*, *hiking*, *airline*}. Since $\{q_1, q_3, q_5\}$ is a maximum subset of satisfiable queries, this subset of attributes must ba a most popular package.

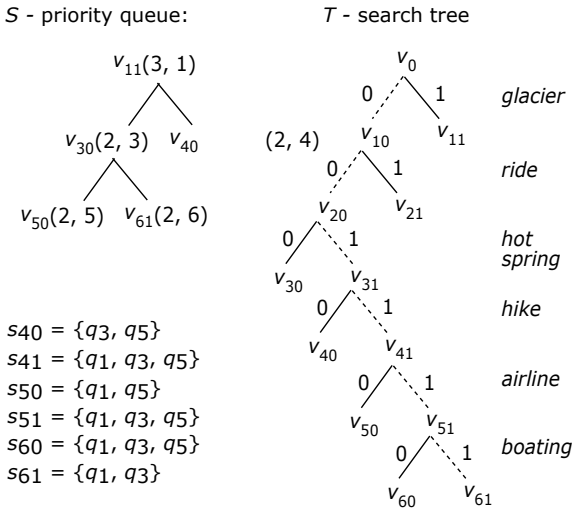


Figure 3: A sample trace.

By this example, a very important property of T can be observed. That is, along each path, the sizes of subsets of queries (represented by the nodes) never increase since the subset of queries represented by a node must be part of the subset represented by its parent. Based on this property, we can easily prove the following proposition.

Proposition 1. *Let Q be a query log. Then, the subset of attributes found in Q by applying the algorithm PRIORITY-SEARCH( ) to Q must be a most popular package.*

Proof. Let v be the last node created (along a certain path). Then, the pair associated with v will have the following properties:

- $l_v = m$, and
- $|s_v|$ is the largest among all the nodes in the current H.

Since the sizes of subsets of queries never increase along each path in T, s_v must be a maximum subset of queries which can be satisfied by a certain group of attributes. One such a group can be simply determined by the labels on the path from *root* to v. This completes the proof. □

The computation shown in Example 1 is super efficient. Instead of searching a binary tree of size $O(2^m)$ (as illustrated in Fig. 1), the algorithm *PRIORITY-SEARCH( )* only explores a single root-to-leaf path, i.e., the path represented by dashed edges in Fig. 3). However, in general, we may still need to create all the nodes in a complete binary tree in the worst case.

Then, we may ask an interesting question: how many nodes need to be generated on average?

In the next subsection, we answer this question by giving a probabilistic analysis of the algorithm.

3.3. Average time complexity

From Fig. 2 and 3, we can see that for each internal node encountered, both of its child nodes will be created. However, only for some of them, both of their children will be explored. For all the others, only one of their children is further explored. For ease of explanation, we call the former 2-nodes while the latter 1-nodes.

We also assume that in T each level corresponds to an attribute. Let $a = a_1a_2 \ldots a_m$ be an attribute sequence, along which the tree T is expanded level-by-level. For instance, in the tree shown in Fig. 3 the nodes are explored along an attribute sequence: *glacier – ride – hot spring – hiking – airline - boating*. For simplicity, we will use a', a'', a''' … to designate the strings obtained by circularly shift the attributes of a. That is,

$a' = a_2 \ldots a_m a_1$,
$a'' = a_3 \ldots a_m a_2 a_1$,
… …
$a^{(m)} = a = a_1 a_2 \ldots a_m$.

In addition, we will use $\aleph_a(T)$ to represent the number of nodes created when applying *PRIORITY-SEARCH*( ) to Q, along a path from top to bottom.

$$\aleph_a(T) = 1 + \aleph_{a'}(T_1) + \aleph_{a'}(T_2) \tag{1}$$

where T_1 and T_2 represent the left and right subtree of *root*, respectively.

However, if the root of T is a 1-node, we have

$$\aleph_a(T) = 1 + \aleph_{a'}(T_1) \quad \text{or} \quad \aleph_a(T) = 1 + \aleph_{a'}(T_2) \tag{2}$$

depending on whether $s_{v_l} \geq s_{v_r}$ or $s_{v_l} < s_{v_r}$, where v_l and v_r stand for the left and right child of the root.

Now we consider the probability that $|T_1| = p$ and $|T_2| = \aleph - p$, where $\aleph$ is the number of all nodes in T. This can be estimated by the *Bernouli probabilities*:

$$\binom{\aleph}{p}\left(\frac{1}{2}\right)^p\left(\frac{1}{2}\right)^{\aleph-p} = \frac{1}{2^\aleph}\binom{\aleph}{p} \tag{3}$$

Let $c_{a,\aleph}$ denote the expected number of nodes created during the execution of *PRIORITY-SEARCH*( ) against Q. In terms of (1), (2) and (3), we have the following recurrences for $\aleph \geq 2$:

$$\text{if root is 2-node, } c_{a,\aleph} = 1 + \frac{2}{2^\aleph}\Sigma_p\binom{\aleph}{p}c_{a',p} \tag{4}$$

$$\text{if root is 1-node, } c_{a,\aleph} = 1 + \frac{1}{2^\aleph}\Sigma_p\binom{\aleph}{p}c_{a',p} \tag{5}$$

Let $\gamma_1 = 1$ if *root* is a 1-node, and $\gamma_1 = 2$ if root is a 2-node. Then, (4) and (5) can be rewritten as follows:

$$c_{a,\aleph} = 1 + \frac{\gamma_1}{2^\aleph}\Sigma_p\binom{\aleph}{p}c_{a',p} - \delta_{\aleph,0} - \delta_{\aleph,1} \tag{6}$$

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

To solve this recursive equation, we consider the following exponential *generating* function of the average number of nodes searched during the execution of *PRIORITY-SEARCH*( ).

$$C_a(z) = \Sigma_{\aleph\geq 0}c_{a,\aleph}\frac{z^\aleph}{2^\aleph} \quad (0 \leq z \leq 1) \tag{7}$$

In the following, we will show that the generating function satisfies a relation given below:

$$C_a(z) = \gamma_1 e^{z/2}C_{a'}(\frac{z}{2}) + e^z - 1 - z. \tag{8}$$

In terms of (6), we rewrite $C_a(z)$ as follows:

$$\begin{aligned} C_a(z) &= \Sigma_{\aleph\geq 0}(1 + \gamma_1(\tfrac{1}{2})^\aleph\Sigma_p\tbinom{\aleph}{p} - \delta_{\aleph,0} - \delta_{\aleph,1})\tfrac{z^\aleph}{2^\aleph} \\ &= \Sigma_{\aleph\geq 0}\tfrac{z^\aleph}{2^\aleph} + \Sigma_p\gamma_1(\tfrac{1}{2})^\aleph\Sigma_{\aleph\geq}\tbinom{\aleph}{p}c_{a',\aleph}\tfrac{z^\aleph}{2^\aleph} \\ &\quad -\Sigma_{\aleph\geq 0}\delta_{\aleph,0}\tfrac{z^\aleph}{2^\aleph} - \Sigma_{\aleph\geq 0}\delta_{\aleph,1}\tfrac{z^\aleph}{2^\aleph} \\ &\leq 2 + \gamma_1\Sigma_p\tfrac{(z/2)^p}{2^p}\Sigma_{\aleph\geq 0}c_{a',p}\tfrac{(z/2)^{\aleph-p}}{2^{\aleph-p}} - 1 - z \\ &= \gamma_1 e^{z/2}C_{a'}(\tfrac{z}{2}) + e^z - 1 - z. \end{aligned} \tag{9}$$

Next, we need to compute $C_{a'}(z)$, $C_{a''}(z)$, …, $C_{a^{(m-1)}}(z)$. To this end, we define γ_i for $i \geq 2$ as follows:

- $\gamma_i = 1$, if all the nodes at level i are 1-nodes.
- $1 < \gamma_i \leq 2$, if at least one node at level i is a 2-node.

Concretely, γ_i is calculated as below:

$$\gamma_i = \frac{2 \times num(\text{2-nodes at level } i) + num(\text{1-nodes at level } i)}{num(\text{nodes at level } i)} \tag{10}$$

where num(node at level i) represents the number of nodes at level i.

In the same way as above, we can get the following equations:

$$\begin{aligned} &C_a(z) = \gamma_1 e^{z/2}C_{a'}(\tfrac{z}{2}) + e^z - 1 - z, \\ &C_{a'}(z) = \gamma_2 e^{z/2}C_{a''}(\tfrac{z}{2}) + e^z - 1 - z, \\ &\quad \ldots\ldots \\ &C_{a^{(m-1)}}(z) = \gamma_m e^{z/2}C_a(\tfrac{z}{2}) + e^z - 1 - z, \end{aligned} \tag{11}$$

These equations can be solved by successive transportation, as done in [22]. For example, when transporting the expression of $C_{a'}(z)$ given by the second equation in (11), we will get

$$C_a(z) = b(z) + \gamma_1 e^{z/2}b(\frac{z}{2}) + \gamma_1\gamma_2 e^{z/2}e^{z/2^2}C_{a''}(\frac{z}{2^2}), \tag{12}$$

where $b(z) = e^z$ - 1 - z.

In a next step, we transport $C_{a''}$ into the equation given in (12). Especially, this equation can be successively transformed this way until the relation is only on $C_a(z)$ itself. (Here, we assume that in this process a is circularly shifted.) Doing this, we will eventually get

$$\begin{aligned} C_a(z) &= \gamma_1 \ldots \gamma_m exp[z(1 - \tfrac{1}{2^m})]C_a(\tfrac{z}{2^m}) + \\ &\quad \Sigma_{j=1}^{m-1}\gamma_1 \ldots \gamma_m exp[z(1 - \tfrac{1}{2^j})](exp(\tfrac{z}{2^j}) - 1 - \tfrac{z}{2^j}) \\ &\leq 2^{m-k}exp[z(1 - \tfrac{1}{2^m})]C_a(\tfrac{z}{2^m}) \\ &\quad + \Sigma_{j=1}^{m-1}\gamma_1 \ldots \gamma_m exp[z(1 - \tfrac{1}{2^j})](exp(\tfrac{z}{2^j}) - 1 - \tfrac{z}{2^j}) \end{aligned} \tag{13}$$

where k is the number of all those levels each containing only 1-nodes.

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

$B(z) = \Sigma_{j=0}^{m-1}\gamma_1\gamma_2 \ldots \gamma_m exp[z(1 - \frac{1}{2^j})](exp(\frac{z}{2^j} - 1 - \frac{z}{2^j})$.

We have

$$C_a(z) = \alpha e^{\beta z} C_a(\gamma z) + B(z). \quad (14)$$

Solving the equation in a way similar to the above, we get

$$\begin{aligned} C_a(z) &= \Sigma_{j=0}^{\infty} \alpha^j exp(\beta \tfrac{1-\gamma^j}{1-\gamma} z) B(\gamma^j z) \\ &= \Sigma_{j=0}^{\infty} 2^{j(m-k)} \Sigma_{h=0} z^{m-1} \gamma_1 \gamma_2 ... \gamma_h [exp(z) \\ &\quad - exp(z(1 - \tfrac{1}{2^h 2^{mj}}))(1 + \tfrac{z}{2^h 2^{mj}})] \end{aligned} \quad (15)$$

Finally, using the *Taylor* formula to expand $exp(z)$ and $exp(z(1-\frac{1}{2^h 2^{mj}}))(1+\frac{z}{2^h 2^m j})$ in the above equation, and then extracting the *Taylor* coefficients, we get

$$C_{a,\aleph} = \Sigma_{h=0}^{m-1} \gamma_1 \gamma_2 ... \gamma_h \Sigma_{j \geq 0} 2^{j(m-k)} D_{jh}(\aleph) \quad (16)$$

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

$$\begin{aligned} D_{jh}(\aleph) &= 1 - (1 - 2^{-mj-h})^{\aleph} \\ &\quad - \aleph 2^{-mj-h} (1 - 2^{-mj-h})^{\aleph - 1}. \end{aligned} \quad (17)$$

$C_{a,\aleph}$ can be estimated by using the Mellin transform [23] for summation of series, as done in [22]. According to [22], $C_{a,\aleph} \sim \aleph^{1-k/m}$. If $k/m \geq 1/2$, $C_{a,\aleph}$ is bounded by $O(\aleph^{0.5})$.

It can also be seen that the priority queue can have up to $O(2^{m/2})$ nodes on average. Therefore, the running time of *extractMax*( ) and *insert*( ) each is bounded by $\log 2^{m/2} = m/2$. Thus, the average cost for generating nodes during the process should be bounded by $O(m\aleph^{0.5}) \leq O(m2^{m/2})$. In addition, the whole cost for selecting an attribute to split s_v for each internal node v into its two child nodes: s_{v_l} and s_{v_r} is bounded by $O(nm^2)$ since the cost for splitting all the nodes at a level is bounded by $O(nm)$ and the height of T is at most $O(m)$. Here, v_l and v_r represent the left and right child nodes of v, respectively.

So we have the following proposition.

Proposition 2. *Let $n = |Q|$ and m be the number of attributes in Q. Then, the average time complexity of Algorithm PRIORITY-SEARCH(Q) is bounded by $O(nm^2 + m2^{m/2})$.*

4. The Second Algorithm

In this section, we discuss our second algorithm. First, we describe the main idea of this algorithm in Section 4.1. Then, in Section 4.2, we discuss the algorithm in great detail. Next, we analyze the algorithm's time complexity in Section 4.3.

4.1. Main idea

Let $Q = \{q_1, ..., q_n\}$ be a query log and $A = \{a_1, ..., a_m\}$ be the corresponding set of attributes. For each $q_i = c_{i1}c_{i2} \ldots c_{im}$ ($c_{ij} \in \{0, 1, *\}$, $j = 1, \ldots, m$), we will create another sequence: $r_i = d_{j_1} ... d_{j_k}$ ($k \leq m$), where $d_{j_l} = a_{j_l}$ if $c_{ij_l} = q_i[j_l] = 1$, or $d_{j_l} = (a_{j_l}, *)$ if $c_{ij_l} = q_i[j_l] = *$ ($l \in \{1, ..., k\}$). If $c_{ij_l} = q_i[j_l] = 0$, the corresponding a_{j_l} will not appear in r_i at all. Let s and t be the numbers of 1s and *s in q_i, respectively. We can then see that $k = s + t$.

For example, for q_1 = (1, *, 0, *, 1, *) in Table 1, we will create a sequence shown below:

r_1 = *hot-spring*. (*ride*, ∗). (*hiking*, ∗).*airline*.(*boating*, ∗).

Next, we will order all the attributes in Q such that the most frequent attribute appears first, but with ties broken arbitrarily. When doing so, (a, *) is counted as an appearance of a. For example, according to the appearance frequencies of attributes in Q (see Table 2), we can define a global ordering for all the attributes in Q as below:

$$A \rightarrow Hs \rightarrow H \rightarrow B \rightarrow R \rightarrow G,$$

where A stands for *airline*, Hs for *hot spring*, H for *hiking*, B for *boating*, R for *ride*, and G for *glacier*.

Following this general ordering, we can represent each query in Table 1 as a sorted attribute sequense as demonstrated in Table 3 (in this table, the second column shows all the attribute sequences while the third column shows their sorted versions). In the following, by an attribute sequence, we always mean a sorted attribute sequence.

In addition, each sorted query sequence in Table 3 is augmented with a start symbol # and an end symbol $, which are used as sentinels for technical convenience.

Finally, for each query sequence q, we will generate a directed graph $\mathcal{G}$ such that each path from the root to a leaf in $\mathcal{G}$ represents a package satisfying q. For this purpose, we first discuss a simpler concept.

Definition 4.1. *(p-graph) Let $q = a_0a_1 \ldots a_ka_{k+1}$ be an attribute sequence representing a query as described above, where a_0 = #, a_{k+1} = $, and each a_i ($1 \leq i \leq k$) is an attribute or a pair of the form (a, *); A p-graph over q is a directed graph, in which there is a node for each a_j ($j = 0, ..., k + 1$); and an edge for (a_j, a_{j+1}) for each $j \in \{0, ..., k\}$. In addition, there may be an edge from a_j to a_{j+2} for each $j \in \{0, ..., k - 1\}$ if a_{j+1} is a pair (a, *), where a is an attribute.*

As an example, consider the p-graph for q_1 = #.A.Hs.(H, *).(B, *).(R, *).$, shown in Fig. 4(a). In this graph, besides a main path going through all the attributes in q_1, we also have three off-line *spans*, respectively corresponding to three pairs: (H, *), (B, *), and (R, *). Each of them represents an option. For example, going through the span for (H, *) indicates that 'H' is not selected while going through 'H' along the main path indicates that 'H' is selected.

In the following, we will represent a span by the sub-path (of the main path) covered by it. Then, the above three spans can be represented as follows:

(H, *) - <v_2, v_3, v_4>

(B, *) - <v_3, v_4, v_5>

(R, *) - <v_4, v_5, v_6>.

Here, each sub-path p is simply represented by a set of contiguous nodes <v_{i_1}, ..., v_{i_j}> ($j \geq 3$) which p goes through. Then, for the graph shown in Fig. 4, <v_2, v_3, v_4> stands for a sub-path from v_2 to v_4, <v_3, v_4, v_5> for a sub-path from v_3 to v_5, and <v_4, v_5, v_6> for a sub-path from v_4 to v_6.

Table 2: Appearance frequencies of attributes.

attributes	*Hs*	*R*	*G*	*H*	*A*	*B*
appearance frequencies	5/6	6/6	5/6	5/6	5/6	5/6

Table 3: Queries represented as sorted attribute sequences.

query ID	attribute sequences*	sorted attribute sequences
q_1	Hs.(R, *).(H, *).A.(B, *).	#.A.Hs.(H, *).(B, *).(R, *).$
q_2	Hs.G.(H, *).(A, *).(B, *).	#.(A, *).Hs.(H, *).(B, *).G.$
q_3	(Hs, *).H.A.(B, *).	#.A.(Hs, *).H.(B, *).$
q_4	(R, *).G.(H, *).A.(B, *).	#.A.(H, *).(B, *).(R, *).G.$
q_5	(Hs, *).(H, *).(A, *).	#.(A, *).(Hs, *).(H, *).$
q_6	(Hs, *).R.(G, *).(A, *).B.	#.(A, *).(Hs, *).B.R.(G, *).$

*Hs: *hot spring*, R: *ride*, G: *glacier*, H: *hiking*, A: *airline*, B: *boating*.

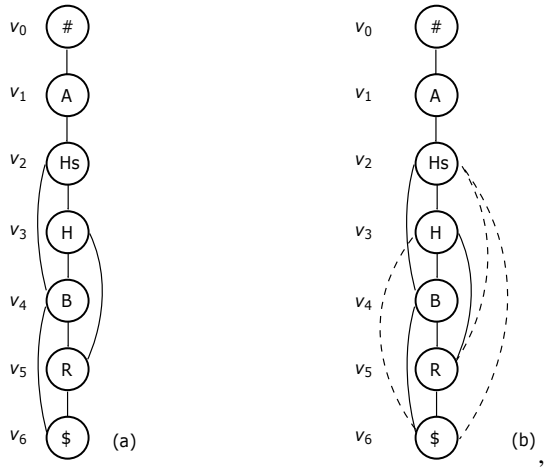


Figure 4: A p-path and a p*-path.

In fact, what we want is to represent each packages for a query q as a root-to-leaf path in such a graph. However, p-graph fails for this purpose. It is because when we go through from a certain node v to another node u through a span, u must be selected. If u itself represents a (c, *) for some variable name c, the meaning of this '*' is not properly rendered. It is because (c, *) indicates that c is optional, but going through a span from v to (c, *) makes c always selected. So, (c, *) is not well interpreted.

For this reason, we will introduce another concept, the so-called p*-graph, to solve this problem.

Let $w_1 = <x_1, ..., x_k>$ and $w_2 = <y_1, ..., y_l>$ be two spans in a same p-graph. We say, w_1 and w_2 are overlapped, if one of the two following conditions is satisfied:

- $y_1 = x_j$ for some $x_j \in \{x_2, ..., x_{k-1}\}$, or
- $x_1 = y_{j'}$ for some $y_{j'} \in \{y_2, ..., y_{l-1}\}$

For example, in Fig. 4(a), $<v_3, v_4, v_5>$ (for (B, *)) are overlapped with both $<v_2, v_3, v_4>$ (for (H, *)) and $<v_4, v_5, v_6>$ (for (R, *)). But $<v_2, v_3, v_4>$ and $<v_4, v_5, v_6>$ are only connected, not overlapped. Being aware of this difference is important since the ovetlapped spans correspond to consecutive *'s while the connected overspans not. More important, the overlapped spans are *transitive*. That is, if w_1 and w_2 are two overlapped spans, the $w_1 \cup w_2$ must be a new, but bigger span. Applying this union operation to all the overlapped spans in a p-graph, we will get their '*transitive closure*'. Based on this discussion, we can now define graph $\mathcal{G}$ mentioned above.

Definition 4.2. *(p*-graph) Let p be the main path in a p-graph P and W be the set of all spans in P. Denote by W* the 'transitive closure' of W. Then, the p*-graph $\mathcal{G}$ with respect to P is defined to be the union of p and W*, denoted as $\mathcal{G} = p \cup W$*.*

See Fig. 4(b) for illustration, in which we show the p*-graph $\mathcal{G}$ of the p-graph P shown in Fig. 4(a). From this, we can see that each root-to-leaf path in $\mathcal{G}$ represents a package satisfying q_1.

In general, in regard to p*-graphs, we can prove the following important property.

Lemma 1. *Let P* be a p*-graph for a query (attribute sequence) q in Q. Then, each path from # to $ in P* represents a package, under which q is satisfied.*

Proof. (1) Corresponding to any package σ, under which q is satisfied, there is definitely a path from # to $. First, we note that under such a package each attribute a_j with $q[j] = 1$ must be selected, but with some don't cares it is selected or not. Especially, we may have more than one consecutive don't cares that are not selected, which are represented by a span equal to the union of the corresponding overlapped spans. Therefore, for σ we must have a path representing it.

(2) Each path from # to $ represents a package, under which q is satisfied. To see this, we observe that each path consists of several edges on the main path and several spans. Especially, any such path must go through every attribute a_j with $q[j] = 1$ since for each of them there is no span covering it. But each span stands for a '*' or more than one successive '*'s. □

4.2. Algorithm based on trie-like graph search

In this subsection, we discuss how to find a packaget that maximizes the number of satisfied queries in Q. To this end, we will first construct a *trie-like* structure G over Q, and design a recursive algorithm to search G bottom-up to get results.

Let $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_n$ be all the p*-graphs constructed for all the queries q_i ($i = 1, ..., n$) in Q, respectively. Denote by p_j and W_j* ($j = 1, ..., n$) the main path of $\mathcal{G}_j$ and the corresponding transitive closure of spans. We will build up G in two steps. In the first step,

we will establish a *trie*, denoted as $T = trie(R)$ over $R = \{p_1, ..., p_n\}$ as follows.

If $|R| = 0$, $trie(R)$ is, of course, empty. For $|R| = 1$, $trie(R)$ is a single node. If $|R| > 1$, R is split into m (possibly empty) subsets $R_1, R_2, \ldots, R_m$ so that each R_i ($i = 1, \ldots, m$) contains all those sequences with the same first attribute name. The tries: $trie(R_1)$, $trie(R_2), \ldots, trie(R_m)$ are constructed in the same way except that at the kth step, the splitting of sets is based on the kth attribute (along the global ordering of attributes). They are then connected from their respective roots to a single node to create $trie(R)$.

In Fig. 5(a), we show the trie constructed for the attribute sequences shown in the third column of Table 2. In such a trie, special attention should be paid to all the leaf nodes each labeled with \$, representing a query (or a subset of queries). Each edge in the trie is referred to as a *tree edge*.

The advantage of tries is to cluster common parts of attribute sequences together to avoid possible repeated checking. (Then, this is the main reason why we sort attribute sequences according to their appearance frequencies.) Especially, this idea can also be applied to the attribute subsequences (as will be seen later), over which some dynamical tries can be recursively constructed, leading to an efficient algorithm.

In the following discussion, the attribute c associated with a node v is referred to as the label of v, denoted as $l(v) = c$.

In addition, we will associate each node v in the trie T with a pair of numbers (*pre*, *post*) to speed up recognizing ancestor/descendant relationships of nodes in T, where *pre* is the order number of v when searching T in preorder and *post* is the order number of v when searching T in postorder.

These two numbers can be used to check the ancestor/descendant relationships in T as follows.

- Let v and v' be two nodes in T. Then, v' is a descendant of v iff $pre(v') > pre(v)$ and $post(v') < post(v)$.

For the proof of this property of any tree, see Exercise 2.3.2-20 in [24].

For instance, by checking the label associated with v_3 against the label for v_{12} in Fig. 5(a), we get to know that v_3 is an ancestor of v_{12} in terms of this property. Particularly, v_3's label is (3, 9) and v_{12}'s label is (11, 6), and we have $3 < 11$ and $9 > 6$. We also see that since the pairs associated with v_{14} and v_6 do not satisfy this property, v_{14} must not be an ancestor of v_6 and *vice versa*.

In the second step, we will add all W_i* ($i = 1, ..., n$) to the trie T to construct a trie-like graph G, as illustrated in Fig. 5(b). This trie-like graph is in fact constructed for all the attribute sequences given in Table 2. In this trie-like graph, each span is associated with a set of numbers used to indicate what queries the span belongs to. For example, the span $<v_2, v_3, v_4>$ is associated with three numbers: 3, 5, 6, indicating that the span belongs to 3 queries: q_3, q_5 and q_6. In the same way, the labels for tree edges can also be determined. However, for simplicity, the tree edge labels are not shown in Fig. 5(b).

From Fig. 5(b), we can see that although the number of satisfying packages for queries in Q is exponential, they can be represented by a graph with polynomial numbers of nodes and edges. In fact, in a single p*-graph, the number of edges is bounded by $O(n^2)$. Thus, a trie-like graph over m p*-graphs has at most $O(n^2m)$ edges.

In a next step, we will search G bottom-up level by level to seek all the possible largest subsets of queries which can be satisfied by a certain package.

First of all, we call a node in T with more than one child a *branching* node. For instance, node v_1 with two children v_2 and v_{13} in G shown in Fig. 5(a) is a branching node. For the same reason, v_3, v_4, and v_5 are another three branching nodes.

Node v_0 is not a branching node since it has one child in T (although it has more than one child in G.)

Around the branching node, we have two very important concepts defined below.

Definition 4.3. *(reachable subsets through spans) Let v be a branching node. Let u be a node on the tree path from root to v in G (not including v itself). A reachable subset of u through spans are all those nodes with a same label c in different subgraphs in G[v] (the subgraph rooted at v) and reachable from u through a span, denoted as $RS_s^{v,u}[c]$, where s is a set containing all the labels associated with the corresponding spans.*

For $RS_s^{v,u}[c]$, node u is called its *anchor* node.

For instance, v_3 in Fig. 5(a) is a branching node (which has two children v_4 and v_{11} in T). With respect to v_3, node v_2 on the tree path from *root* to v_3, has a reachable subset:

- $RS_{\{1,5\}}^{v_3,v_2}[\$] = \{v_8, v_{12}\}$,

We have this *RS* (reachable subset) due to two spans $v_2 \xrightarrow{1} v_8$ and $v_2 \xrightarrow{5} v_{12}$ going out of v_2, respectively reaching v_5 and v_8 on two different p*-graphs in $G[v_3]$ with $l(v_5) = l(v_{12}) =$ '\$'. Then, v_2 is the anchor node of $\{v_8, v_{12}\}$.

In general, we are interested only in those *RS*'s with $|RS| \geq 2$ (since any *RS* with $|RS| = 1$ only leads us to a leaf node in T and no larger subsets of queries can be found.) So, in the subsequent discussion, by an *RS*, we always mean an *RS* with $|RS| \geq 2$.

The definition of this concept for a branching node v itself is a little bit different from any other node on the tree path (from *root* to v). Specifically, each of its *RS*s is defined to be a subset of nodes reachable from a span or from a tree edge. So for v_3 we have:

- $RS_{\{1,3,5\}}^{v_3,v_3}[\$] = \{v_8, v_{11}, v_{11}\}$,

due to two spans $v_3 \xrightarrow{1} v_8$ and $v_3 \xrightarrow{3} v_{11}$, and a tree edge $v_3 \rightarrow v_{12}$, all going out of v_3 with $l(v_8) = l(v_{11}) = l(v_{12}) =$ '\$'. Then, v_3 is the anchor node of $\{v_8, v_{11}, v_{11}\}$.

Based on the concept of reachable subsets through spans, we are able to define another more important concept, *upper boundaries*. This is introduced to recognize all those p*-subgraphs around a branching node, over which a trie-like subgraph needs to be constructed to find some more subsets of queries satisfiable by a certain package.

Definition 4.4. *(upper boundaries) Let v be a branching node. Let $v_1, v_2, ..., v_k$ be all the nodes on the tree path from root to v. An upper boundary (denoted as upBounds) with respect to v is a largest subset of nodes $\{u_1, u_2, ..., u_f\}$ with the following properties satisfied:*

1. Each u_g ($1 \leq g \leq f$) appears in some $RS_{v_i}[c]$ ($1 \leq i \leq k$), where c is a label.

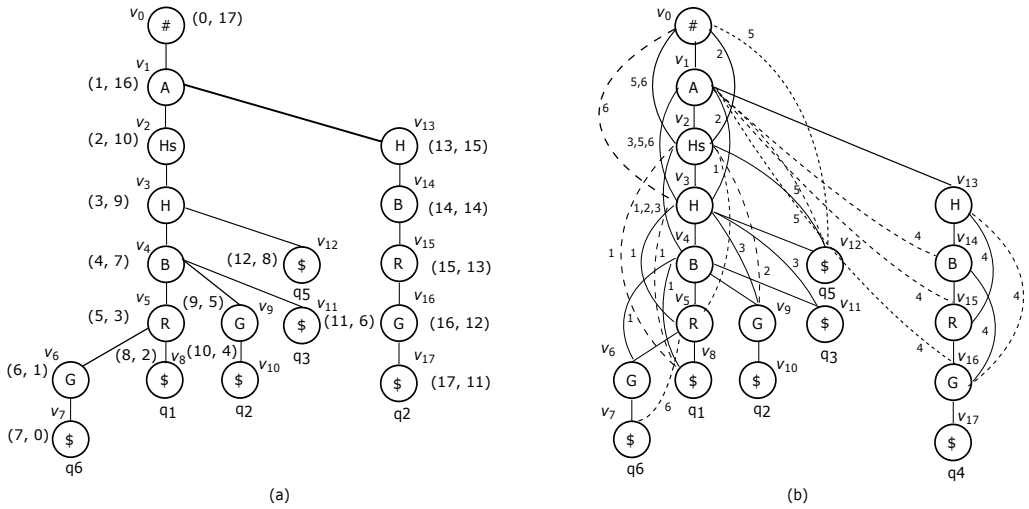


Figure 5: A trie and a trie-like graph.

2. For any two nodes u_g, $u_{g'}$ ($g \neq g'$), they are not related by the ancestor/descendant relationship.

Fig. 6 gives an intuitive illustration of this concept.

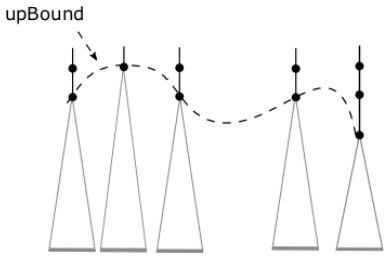


Figure 6: Illustration for upBounds.

As a concrete example, condider branching node v_4 in 5(b). With repect to v_4, we have

- $RS^{v_4,v_3}_{\{1,3\}}[\$] = \{v_8, v_{11}\}$,
- $RS^{v_4,v_4}_{\{1,2\}}[\mathrm{G}] = \{v_6, v_9\}$,

Since all the nodes in these two *RS*s (v_8, v_6, v_9, and v_{11}) are not related by ancstor/descendant relationsh, they make up an upBound with respect to v_4 (a branching node), as illustrated in 7(a).

Then, we will construct a trie-like graph over all the four p*-subgraphs, each starting from a node on the upBound. See Fig. 7(b) for illustration, where v_{6-9} stands for the merging v_6 and v_9, v_{7-10} for the merging v_7 and v_{10}, and v_{8-11} for the merging v_8 and v_{11}.

Obviously, this can be done by a recursive call of the algorithm itself.

In addition, for technical convenience, we will add the corresponding branching node (v_4) to the trie as a virtual *root*, and $v_4 \rightarrow v_{6-9}$ and $v_3 \rightarrow v_{8-11}$ as two virtual edges, each associated with the corresponding *RS*s to facilitate the search of all those packages satisfying corresponding queries. This is because to find such packages we need to travel through this branching node to the *root* of T. See Fig. 9(c) for illustration.

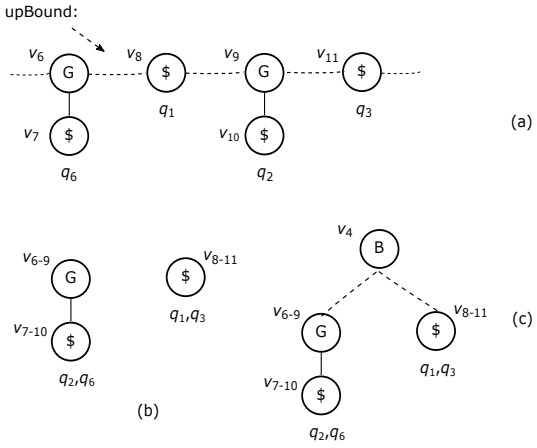


Figure 7: Illustration for upBounds and recursive contruction of trie-like subgraphs.

Specifically, the following operations will be carried out when encountering a branching node v.

- Calculate all *RS*s with respect v.
- Calculate the upBound in terms of *RS*s.
- Make a recursive call of the algorithm over all the subgraphs within $G[v]$ each rooted at a node on the corresponding upBound.

In terms of the above discussion, we design a recursive algorithm to do the task, in which R is used to accommodate the result, represented as a set of triplets of the form:

$<\alpha, \beta, \gamma>$,

where α stands for a subset of conjunctions, β for a truth assignment satisfying the conjunctions in α, and γ is the size of α. Initially, $R = \emptyset$.

Algorithm 2: *popularPack*(Q)

Input : a query log Q.
Output : a most polular package.
let $Q = \{q_1, ..., q_n\}$;
for $i = 1$ *to* n **do**
  construct a p*-graph P_i^* for q_i;
construct a trie-like graph G over $P_1^*, ..., P_n^*$;
return *SEARCH*(G, $\emptyset$);

The input of *popularPack*( ) is a query log $Q = \{q_1, ..., q_n\}$. First, we will build up a p*-graph for each q_i ($i = 1, ..., n$), over which a trie-like graph G will be constructed (see lines 2 - 4). Then, we call a recursive algorithm *SEARCH*( ) to produce the result (see line 5), which is a set of triplets of the form $<\alpha, \beta, \gamma>$ with the same largest γ value. Thus, each β is a popular package.

Algorithm 3: *SEARCH*(G, R)

Input : a trie-like subgraphs G.
Output : a largest subset of conjunctions satisfying a certain truth assignment.
if G *is a single p*-graph* **then**
  R' := subset associated with the leaf node;
  $R := merge(R, R')$;
  return R;
for *each leaf node v in G* **do**
  let R' be the subset associated with v;
  $R := merge(R, R')$;
let $v_1, v_2, ..., v_k$ be all branching nodes in postorder;
for $i = 1$ *to* k **do**
  let P be the tree path from *root* to v_i;
  for *each u on P* **do**
    calculate *RS*s of u with respect to v_i;
  create the corresponding upBound L;
  construct a trie-like subgraph D over all those subgraphs each rooted at a node on L;
  $D' := \{v_i\} \cup D$;
  $R' := SEARCH(D', R)$;
  $R := merge(R, R')$;
return R;

SEARCH( ) works recurcively. Its input is a pair: a trie-like subgraph G' and a set R' of triplets $<\alpha, \beta, \gamma>$ found up to now. Initially, $G' = G$ and $R = \emptyset$.

First, we check whether G is a single p*-graph. If it is the case, we must have found a largest subset of queries associated with the leaf node, satisfiable by a certain package. This subset should be merged into R (see lines 1 - 4).

Otherwise, we will search G bottom up to find all the branching nodes in G. But before that, each subset of queries associated with a leaf node in R will be first merged into R (see line 5 - 7).

For each branching node v encountered, we will check all the nodes u on the tree path from *root* to v and compute their *RS*s (reachable subsets through spans, see lines 8 - 12), based on which we then compute the corresponding upBound with respect to v (see line 13). According to the upBound L, a trie-like graph D will be created over a set of subgraphs each rooted at a node on L (see line 14). Next, v will be added to D as its root (see line 15). Here, we notice that $D' := \{v\} \cup D$ is a simplified representation of an operation, by which we add not only v, but also the corresponding edges to D. Next, a recursive call of the algorithm is made over D' (see linee 16). Finally, the result of the recursive call of the algorithm will be merged into the global answer (see line 17).

Here, the *merge* operation used in line 3, 7, 17 is defined as below.

Let $R = \{r_1, ..., r_t\}$ for some $t \geq 0$ with each $r_i = <\alpha_i, \beta_i, \gamma_i>$. We have $\gamma_1 = \gamma_2 = ... = \gamma_t$. Let $R' = \{r'_1, ..., r'_s\}$ for some $s \geq 0$ with each $r'_i = <\alpha'_i, \beta'_i, \gamma'_i>$. We have $\gamma'_1 = \gamma'_2 = ... = \gamma'_s$. By $merge(R, R')$, we will do the following checks.

- If $\gamma_1 < \gamma'_1$, $R := R'$.
- If $\gamma_1 > \gamma'_1$, R remains unchanged.
- If $\gamma_1 = \gamma'_1$, $R := R \cup R'$.

In the above algorithm, how to figure out β in a triple $<\alpha, \beta, \gamma>$ is not specified. For this, however, some more operations should be performed. Specifically, we need to trace each chain of recursive calls of *SEARCH*( ) for this task.

Let $SEARCH(G_0, R_0) \rightarrow SEARCH(G_1, R_1) \rightarrow ... \; SEARCH(G_k, R_k)$ be a consecutive recursive call process during the execution of *SEARCH*( ), where $G_0 = G$, $R_0 = \emptyset$, and G_i is a trie-like subgraph constructed around a branching node in G_{i-1} and R_i is the result obtained just before $SEARCH(G_i, R_i)$ is invoked ($i = 1, ..., k$).

Assume that during the execution of $SEARCH(G_k, R_k)$ no further recursive call is conducted. Then, R_k can be a single p*-graph, or a trie-like subgraph, for which no *RS*s with $|RS|$ ¿ 1 for any branching node can be found. Denote by r_j the root of G_j and by v_j the branching node around which $SEARCH(G_{j+1}, R_{j+1})$ is invoked ($j = 0, ..., k$ - 1). Denote by T_j the trie in G_j.

Then, the labels on all the paths from r_j in T_j for $j \in \{1, ... \; k\}$ (connected by using the corresponding anchor nodes) consist of β in the corresponding triple $<\alpha, \beta, \gamma>$ with α being a subset of queries associated with a leaf node in G_k. As an example, consider the the trie-like subfraph G' shown in Fig. 7(c) again. In the execution, we will have a chains of recursive calls as below.

$$SEARCH(G, ...) \rightarrow SEARCH(G', ...)$$

Along this chain, we will find two query subset $Q_1 = \{q_2, q_6\}$ (associated with leaf node v_{7-10}) and $Q_2 = \{q_1, q_3\}$ (associated with leaf node v_{7-10}). To find the package for Q_1, we will trace the path in G' bottom from v_{7-10} to v_{6-10}, the reverse edge $v_{6-9} \xrightarrow{1,2} v_4$ (recognized according to $RS^{v_4,v_4}_{\{1,2\}}$[G]), and then the path from $v_4 \rightarrow$ to v_0 in G. Since {1, 2} does not contain 6, q_6 should be removed from Q_1. The package is {A, Hs, H, B, G}. In the same way, we can find the package {A, Hs, H, B} for $Q_2 = \{q_1, q_3\}$.

The following example helps for illustrating the whole working process.

Example 2. *When applying SEARCH( ) to the p*-graphs constructed for all the attribute sequences given in Table 1, we will first construct a trie-like graph G shown in Fig. 5(b). Searching G bottom up, we will encounter four branching nodes: v_5, v_4, v_3 and v_1.*

For each branching node, a recursive call of *SEARCH*( ) will be carried out. But we show here only the recursive call around v_1 for simplicity. With respect to v_1, we have only one *RS* with $|RS| > 1$:

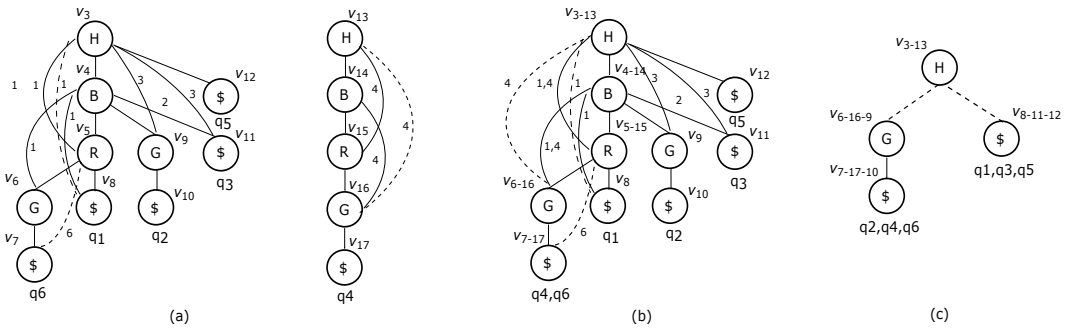


Figure 8: Illustration for Example 2.

- $RS^{v_1,v_1}_{\{1,3,4,5\}}[\mathrm{H}] = \{v_3, v_{13}\}$,

Due to span $v_1 \xrightarrow{3,5,6} v_3$ and tree edge $v_1 \rightarrow v_{13}$.

Therefore, the corresponding upBound is $\{v_3, v_{13}\}$. Then, a new trie-like subgraphs (see Fig. 8(b)) will be constructed by merging two subgraphs shown in Fig. 8(a).

In Fig. 8(b), the node v_{3-13} represents a merging of two nodes v_3 and v_{13} in Fig. 8(a). All the other merging nodes v_{4-14}, v_{5-15}, v_{6-16}, and v_{7-17} are created in the same way.

When applying *SEARCH*( ) to this new trie-like subgraph, we will check all its branching nodes v_{5-15}, v_{4-14}, and v_{3-13} in turn. Especially, with respect to v_{3-13}, we have

- $RS^{v_{3-13},v_{3-13}}_{\{3,4\}}[\mathrm{G}] = \{v_{6-16}, v_9\}$,
- $RS^{v_{3-13},v_{3-13}}_{\{1,3,5\}}[\$] = \{v_8, v_{11}, v_{12}\}$,

According to these *RS*s, we will contruct a trie-like subgraph as shown in Fig. 8(c). From this subgraph, we can find another two query subsets $\{q_2, q_4, q_6\}$ and $\{q_1, q_3, q_5\}$, respectively satisfiable by two packages {A, H, G} and {A, Hs, H}.

In the execution of *SEARCH*( ), much redundancy may be conducted, but can be easily removed. See Fig. 9(a) for illustration.

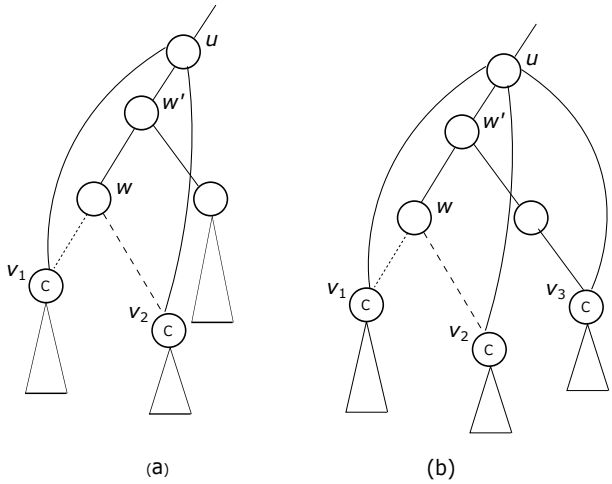


Figure 9: Illustration for redundancy.

In this figure, w and w' are two branching nodes. With respect to w and w', node u will have the same *RS*s. That is, we have

$$RS^{w,u}_s[\mathrm{C}] = RS^{w',u}_s[\mathrm{C}] = \{v_1, v_2\}.$$

Then, in the execution of *SEARCH*( ), the corresponding trie-like subgraph will be created two times, but with the same result produced.

However, this kind of redundancy can be simply removed in two ways.

In the first method, we can examine, by each recursive call, whether the input subgraph has been checked before. If it is the case, the corresponding recursive call will be suppressed.

In the second method, we create *RS*s only for those nodes appearing on the segment (on a tree path) between the current branching node and the lowest ancestor branching node in T. In this way, we may lose some answers. But the most popular package can always be found. See Fig. 9(b) for illustration. In this case, the *RS* of u with respect to w is different from the *RS* with respect to w'. However, when we check the branching node w, $RS^{w,u}_s[\mathrm{C}]$ will not be computed and therefore the corresponding result will not be generated. But $RS^{w',u}_s[\mathrm{C}]$ must cover $RS^{w,u}_s[\mathrm{C}]$. Therefore, a package satisfying a larger, or a same-sized subset of queries will definitely be found.

4.3. Time complexity analysis

The total running time of the algorithm consists of three parts.

The first part, denoted as $\tau_{\cdot}$, is the time for computing the frenquencies of attribute appearances in Q. Since in this process each attribut in a q_i is accessed only once, $\tau_1 = \mathrm{O}(nm)$.

The second part, denoted as τ_2, is the time for constructing a trie-like graph G for Q. This part of time can be further partitioned into three portions.

- τ_{21}: The time for sorting attribute sequences for q_i's. It is obviously bounded by $\mathrm{O}(nm\log_2 m)$.
- τ_{22}: The time for constructing p*-graphs for each q_i (i = 1, ..., n). Since for each variable sequence a transitive closure over its spans should be first created and needs $\mathrm{O}(m^2)$ time, this part of cost is bounded by $\mathrm{O}(nm^2)$.
- τ_{23}: The time for merging all p*-graphs to form a trie-like graph G, which is also bounded by $\mathrm{O}(nm^2)$.

The third part, denoted as τ_3, is the time for searching G to find a maximum subset of conjunctions satisfied by a certain truth assignment. It is a recursive procedure. To analyze its running

time, therefore, a recursive equation should established. Let $l = nm$. Assume that the average outdegree of a node in T is d. Then the average time complexity of τ_4 can be characterized by the following recurrence:

$$\Gamma(l) = \begin{cases} O(1), & \text{if } l \leq \text{a constant}, \\ \sum_{i=1}^{\lceil log_d\ l \rceil} d^i \Gamma(\frac{l}{d^i}) + O(l^2 m), & \text{otherwise.} \end{cases} \quad (18)$$

Here, in the above recursive equation, $O(l^2 m)$ is the cost for generating all the reachable subsets of a node through spans and upper boundries, together with the cost for generating local trie-like subgraphs for each recursive call of the algorithm. We notice that the size of all the *RS*s together is bounded by the number of spans in G, which is $O(lm)$.

From (4), we can get the following inequality:

$$\Gamma(l) \leq d \cdot log_d\ l \cdot \Gamma(\frac{l}{d}) + O(l^2 m). \quad (19)$$

Solving this inequality, we will get

$$\begin{aligned} \Gamma(l) &\leq d \cdot log_d\ l \cdot \Gamma(\tfrac{l}{d}) + O(l^2 m) \\ &\leq d^2 (log_d\ l)(\ log_d \tfrac{l}{d}) \Gamma(\tfrac{l}{d^2}) + \ (log_d\ l)\ l^2 m + \ l^2 m \\ &\leq \ldots\ \ldots \\ &\leq d^{\lceil log_d^l \rceil} (log_d\ l)\ (log_d(\tfrac{l}{d}))\ \ldots\ (\ log_d \tfrac{l}{d^{\lceil log_d^l \rceil}}) \\ &+ l^2 m((log_d\ l)(log_d(\tfrac{l}{d}))\ \ldots\ (log_d \tfrac{l}{d^{\lceil log_d^l \rceil}}) + \ldots + log_d\ l + 1) \\ &\leq O(l(log_d\ l)^{log_d\ l} + \ O(l^2 m(log_d\ l)^{log_d\ l}) \\ &\sim O(l^2 m\ (log_d\ l)^{log_d\ l}). \end{aligned} \quad (20)$$

Thus, the value for τ_3 is $\Gamma(l) \sim O(l^2 m\ (log_d\ l)^{log_d\ l})$.

From the above analysis, we have the following proposition.

Proposition 3. *The average running time of our algorithm is bounded by*

$$\begin{aligned} \Sigma_{i=1}^4 \tau_i &= O(nm) + (O(nm\ log_2\ m) + 2 \times O(nm^2)) \\ &+ O(l^2 m\ (log_d\ l)^{log_d\ l}) \\ &= O(n^2 m^3 (log_d\ nm)^{log_d\ nm}). \end{aligned} \quad (21)$$

But we remark that if the average outdegree of a node in T is < 2, we can use a brute-force method to find the answer in polynomial time. Hence, we claim that the worst case time complexity is bounded by $O(l^2 m(log_2\ l)^{log_2\ l})$ since $(log_d\ l)^{log_d\ l}$ decreases as d increases.

5. Conclusions

In this paper, we have discussed two new method to solve the problem to find a most popular package in terms of a given questionnaire. The first method is based on a kind of tree search, but with the priority queue structure being utilized to control the tree exploration. Together with a powerful heuristics, this approach enables us to cut off a lot of futile branches and find an answer as early as possible in the tree search process. The average time complexity is bounded by $O(nm^2 + m2^{m/2})$, where $n = |Q|$ and m is the number of attributes in the query log $|Q|$. The main idea behind the second method is to construct a graph structure, called p^*-graph. In this way, all the queries in Q can be represented as a trie-like graph. Searching G bottom up, we can find the answer efficiently. the average time complexity of the algorithm is bounded by $O(n^2 m^3 (log_2\ nm)^{log_2\ nm})$.

As a future work, we will make a detailed analysis of the impact of the heuristics discussed in Section 4.2 to avoid any repeated recursive calls. If it is the case, the number of recursive calls for each branching node will be bounded by $O(m)$ since the height of the trie-like graph G is bounded by $O(m)$. Thus, the worst-case time complexity of our algorithm should be bounded by $O(n^2 m^4)$. It is because we have at most $O(nm)$ branching nodes, and for each recursive call we need $O(nm^2)$ time to construct a dynamical trie. So, the total running time will be $O(nm) \times O(m) \times O(nm^2) = O(n^2 m^4)$.

Conflict of Interest The authors declare no conflict of interest.

References

[1] R. Agrawal, T. Imieliński, A. Swami, "Mining association rules between sets of items in large databases," in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 207–216, Association for Computing Machinery, 1993, doi:10.1145/170035.170072.

[2] B. Gavish, D. Horsky, K. Srikanth, "An Approach to the Optimal Positioning of a New Product," Management Science, **29**, 1277–1297, 1983, doi:10.1287/mnsc.29.11.1277.

[3] T. S. Gruca, B. R. Klemz, "Optimal new product positioning: A genetic algorithm approach," European Journal of Operational Research, **146**, 621–633, 2003, doi:https://doi.org/10.1016/S0377-2217(02)00349-1.

[4] J. Resig, A. Teredesai, "A framework for mining instant messaging services," in In Proceedings of the 2004 SIAM DM Conference, 2004.

[5] J. Han, J. Pei, Y. Yin, "Mining frequent patterns without candidate generation," SIGMOD Rec., **29**, 1–12, 2000, doi:10.1145/335191.335372.

[6] M. Miah, G. Das, V. Hristidis, H. Mannila, "Standing Out in a Crowd: Selecting Attributes for Maximum Visibility," in 2008 IEEE 24th International Conference on Data Engineering, 356–365, 2008, doi:10.1109/ICDE.2008.4497444.

[7] J. C.-W. Lin, Y. Li, P. Fournier-Viger, Y. Djenouri, L. S.-L. Wang, "Mining High-Utility Sequential Patterns from Big Datasets," in 2019 IEEE International Conference on Big Data (Big Data), 2674–2680, 2019, doi:10.1109/BigData47090.2019.9005996.

[8] A. Tonon, F. Vandin, "gRosSo: mining statistically robust patterns from a sequence of datasets," Knowledge and Information Systems, **64**, 2329–2359, 2022, doi:10.1007/s10115-022-01689-2.

[9] Y. Chen, W. Shi, "On the Designing of Popular Packages," in 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 937–944, 2018, doi:10.1109/Cybermatics_2018.2018.00180.

[10] M. Miah, "Most popular package design," in 4th Conference on Information Systems Applied Research, 2011.

[11] R. Kohli, R. Krishnamurti, P. Mirchandani, "The Minimum Satisfiability Problem," SIAM Journal on Discrete Mathematics, **7**, 275–283, 1994, doi:10.1137/S0895480191220836.

[12] S. A. Cook, The Complexity of Theorem-Proving Procedures, 143–152, Association for Computing Machinery, 1st edition, 2023.

[13] Y. Chen, "Signature files and signature trees," Information Processing Letters, **82**, 213–221, 2002, doi:https://doi.org/10.1016/S0020-0190(01)00266-6.

[14] Y. Chen, "On the signature trees and balanced signature trees," in 21st International Conference on Data Engineering (ICDE'05), 742–753, 2005, doi:10.1109/ICDE.2005.99.

[15] Y. Chen, Y. Chen, "On the Signature Tree Construction and Analysis," IEEE Transactions on Knowledge and Data Engineering, **18**, 1207–1224, 2006, doi:10.1109/TKDE.2006.146.

[16] F. Grandi, P. Tiberio, P. Zezula, "Frame-sliced partitioned parallel signature files," in Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 286–287, Association for Computing Machinery, 1992, doi:10.1145/133160.133211.

[17] D. L. Lee, Y. M. Kim, G. Patel, "Efficient signature file methods for text retrieval," IEEE Transactions on Knowledge and Data Engineering, **7**, 423–435, 1995, doi:10.1109/69.390248.

[18] D. L. Lee, C.-W. Leng, "Partitioned signature files: design issues and performance evaluation," ACM Trans. Inf. Syst., **7**, 158–180, 1989, doi:10.1145/65935.65937.

[19] Z. Lin, C. Faloutsos, "Frame-sliced signature files," IEEE Transactions on Knowledge and Data Engineering, **4**, 281–289, 1992, doi:10.1109/69.142018.

[20] E. Tousidou, P. Bozanis, Y. Manolopoulos, "Signature-based structures for objects with set-valued attributes," Information Systems, **27**, 93–121, 2002, doi:https://doi.org/10.1016/S0306-4379(01)00047-3.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2022.

[22] P. Flajolet, C. Puech, "Partial match retrieval of multidimensional data," J. ACM, **33**, 371–407, 1986, doi:10.1145/5383.5453.

[23] L. Debnath, D. Bhatta, Integral transforms and their applications, Chapman and Hall/CRC, 2016.

[24] E. K. Donald, "The art of computer programming," Sorting and searching, **3**, 4, 1999.