Priority-First Search and Mining Popular Packages

Yangjun Chen[1] and Bobin Chen[2]
Dept. Applied Computer Science, University of Winnipeg, Canada
[1]y.chen@uwinnipeg.ca, [2]BC234kirk@hotmail.com

***Abstract*—By the package design problem we are given a set of queries (referred to as a query log) with each being a bit string indicating the favourite activities or items of customers and required to design a package of activities (or items) to satisfy as many customers as possible. It is a typical problem of data mining. In this paper, we address this issue and propose an efficient algorithm for solving the problem based on a new tree search strategy, the so-called *priority-first search*, by which the tree search is controlled by using a priority queue, instead of a stack or a queue data structure. Extensive experiments have been conducted, which show that our method for this problem is promising.**

keywords—data mining, single package design, priority queues, priority-first search, time complexity analysis

I. Introduction

Frequent pattern mining plays an essential role in mining associations [1, 9, 11], which are important for decision making. For example, for the management of a supermarket, one has to decide, based on the association rules, what to put on sale, how to design coupons, as well as how to place merchandise on shelves in order to maximize the profit, etc.

By the frequent pattern mining [1, 9, 11], we are given a set of transactions with each containing some items, and required to recognize a frequent pattern (a subset of items) which is supported (or say, contained) by most of transactions. In this paper, we discuss a more challenging problem, the so-called *single package design* problem (*SPD* for short [7]), by which we consider a set of activities or items $A = \{a_1, ..., a_m\}$, like *hot spring*, *riding horse* by a travel agency, referred to as an attribute, an item, or a feature; and a *query log* $Q = \{q_1 \ldots q_n\}$ with each q_i ($i = 1, \ldots, n$) being a bit string of length m: $c_{i1}c_{i2} \ldots c_{im}$ ($c_{ij} \in \{0, 1, *\}$, $j = 1, \ldots, m$). Here, $c_{ij} = 1$ indicates that a_j is selected, and $c_{ij} = 0$ indicates that a_j is not selected while '*' means 'don't care'. Then, we will design a *bit tuple t* (or say a bit string with each bit corresponding to an activity) such that the number of satisfied queries is maximized. We will refer to t as a *package*. Thus, what we want is to ensure that the package satisfies as many queries as possible [13]. We call such a package a *most popular package*. For example, for the above vacation package, clients give their preferences by specifying *yes*, *no*, or '*don't care*' for each activity to form a query log. The design of a most popular package is to pick up a sub-set of these activities to meet as many queries' requirements as possible.

This problem has been investigated by several researchers [4, 13]. In [13], an approximation algorithm was discussed, by which an SPD problem is reduced to a MINSAT problem [12] that is an optimization version of the satisfiability [6], by which we seek to find a truth assignment to minimize the number of satisfied clauses. The method discussed in [4] is in fact based on the construction of a kind of binary trees, called *signature trees* [2, 3] for *signature files* [8, 10]. Its worst-case time complexity is bounded by $O(mn2^m)$.

Our method is also based on the construction of a binary tree, but with a new tree search strategy, called the *priority-first search*, being utilized to cut off a lot of branches. Its average time complexity is bounded by $O(nm^2 + mn2^{m/2})$. Extensive experiments have been conducted, which show that our method is much better than all the existing methods for this problem.

The remainder of the paper is organized as follows. In Section II, we show a simple example of the SPD problem. Then, in Section III, the algorithms for evaluating a SPD is discussed in great detail. Section IV is devoted to the test results. Finally, we conclude with a summery in Section V.

II. An example of SPD

As an example of SPD, Table 1 shows a query log for a vacation package application. It contains $n = 6$ queries, $m = 6$ attributes (activities), and each query represents one of user's favourites. For instance, the query $q_1 = c_{11}c_{12} \ldots c_{16} = (1, *, 0, *, 1, *)$ in Table 1, indicates that *hot spring* and *airlines* are q_1's favourites, but glacier is not. Furthermore, q_1 does not care about whether *riding*, *hiking* or *boating* is available or not.

Table 1: A query log Q

QueryId	Hot Spring	Ride	Glacier	Hiking	Airlines	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, we can find a single package: hot spring, hiking, airline, which satisfy a maximum subset of queries: $q_1, q_3, q_5.$

III. Algorithm description

In this section, we discuss our algorithm. First, we give a basic algorithm to provide a discussion background in Subsection A. Then, in Subsection B, we describe our efficient algorithm in great detail. Finally, in Subsection C, we analyse the average time complexity of this algorithm.

The research is supported by NSERC, Canada, DDG-2019-04100.

A. Basic algorithm

In this subsection, we show a basic algorithm to provide a discussion background, which is in fact an extension of an algorithm discussed in [2, 3]. Its main idea is to construct a binary tree T over Q, working in two steps. In the first step, we construct a signature-tree-like structure, called a *search-tree*. Then, in the second step, we search a path in the search-tree to find a most popular package.

Let $A = \{a_1, a_2, \ldots, a_m\}$ be a set of items (or say attributes). Let $Q = \{q_1, \ldots, q_n\}$ be a query log. Denote by $q_i[j]$ the value of the jth attribute a_j in q_i ($i = 1, \ldots, m$). The binary tree will be constructed as follows.

1. Starting from the first attribute value, we divide all queries in Q into two branches. For query q_i ($1 \leq i \leq n$), 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 [6].
2. In a next step, we will split both left and right branches in the same way as (1), but according to a next attribute.
3. Repeating this process until all the attributes are checked, we will establish a binary tree.

In Fig. 1, we show such a binary tree constructed for the query log given in Table 1.

In Fig. 1, for each node v, we use $s(v)$ to refer to the subset of queries represented by v. According to a certain attribute a, $s(v)$ is split into two subsets, denoted as $s(v)[a]$ and $s(v)[\neg a]$ respectively, such that for each $q \in s(v)[a]$ $q[a] = 1$ while for each $q' \in s(v)[\neg a]$ $q'[a] = 0$. $s(v)[\neg a]$ is represented by v's left child while $s(v)[a]$ is represented by v's right child.

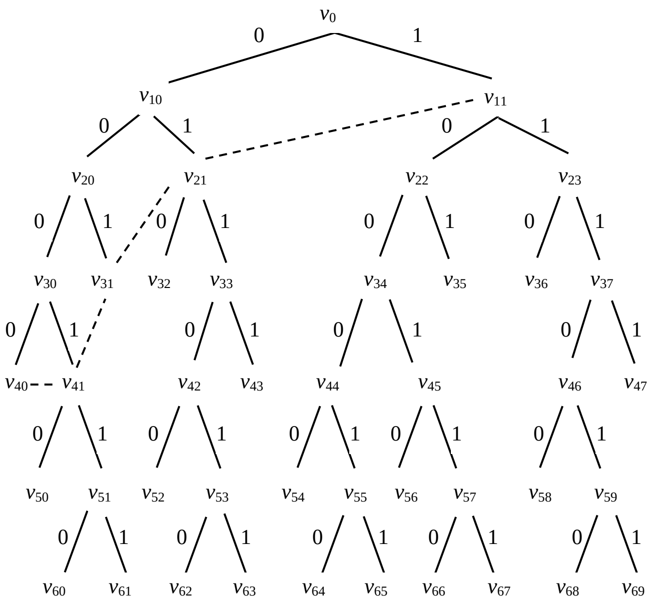
From this figure, we can see that the leaf node v_{66} has the largest size and the labels along the path from the root to it spell out a string 100110, representing a best package: {*hot spring, hiking, airlines*}.

In the following, we analyze the computational complexity of the above algorithm.

First, we notice that an search-tree will have exactly m levels, where m is the number of attributes in Q. So, in the worst case, the number of nodes in a search-tree is bounded by $O(2^m)$. Since each node represents a subset of Q, the time overhead should be bounded by $O(mn2^m)$. But the space requirement is much better since during the construction of a search-tree we need only to maintain a subset of all those nodes on the bottom frontier (i.e., the last nodes on each path at any time point. See the dashed lines in Fig. 1 for illustration, which represent a bottom frontier at a certain point in time: v_{40} - v_{41} – v_{31} - v_{21} – v_{11}.) The number of nodes on a frontier is bounded by $O(2m)$ since in the worst case, a frontier may have 2 nodes for each level. Therefore, the space overhead is bounded by $O(2m^2n)$.

B. Algorithm based on priority-first-searching

The basic algorithm given above is essentially a depth-first search process, controlled by using a stack. If we use a priority queue [5] to control the tree searching, we will get a very efficient strategy since this enables us to cut off a lot of useless branches in T.



$s(v_0) = \{q_1, q_2, q_3, q_4, q_5, q_6\}$ $s(v_{10}) = \{q_3, q_4, q_5, q_6\}$ $s(v_{11}) = \{q_1, q_2, q_3, q_5, q_6\}$

$s(v_{20}) = \{q_3, q_4, q_5\}$ $s(v_{21}) = \{q_4, q_5\}$ $s(v_{22}) = \{q_1, q_2, q_3, q_5\}$ $s(v_{23}) = \{q_1, q_5\}$

$s(v_{30}) = \{q_3, q_5\}$ $s(S_{31}$v $= \{q_4\}$ $s(v_{32}) = \{q_6\}$ $s(v_{33}) = \{q_4, q_6\}$ $s(v_{34}) = \{q_1, q_3, q_5\}$ $s(Sv_{35}) = \{q_2\}$

$s(v_{36}) = \{q_1, q_6\}$ $s(Sv_{37}) = \{q_6$

$s(v_{40}) = \{q_3\}$ $s(v_{41}) = \{q_4, q_5\}$ $s(v_{42}) = \{q_4, q_6\}$ $s(v_{43}) = \{q_4\}$ $s(v_{44}) = \{q_1, q_5\}$ $s(v_{45}) = \{q_1, q_3, q_5\}$

$s(v_{46}) = \{q_1, q_6\}$ $s(v_{47}) = \{q_1\}$

$s(S_{50}) = \{q_3\}$ $S_{51} = \{q_3, q_5\}$ $s(v_{52}) = \{q_6\}$ $s(v_{53}) = \{q_4, q_6\}$ $s(v_{54}) = \{q_5\}$ $s(v_{55}) = \{q_1, q_5\}$

$s(v_{56}) = \{q_5\}$ $s(v_{57}) = \{q_1, q_3, q_5\}$ $s(v_{58}) = \{q_1\}$ $s(v_{59}) = \{q_1, q_6\}$

$s(v_{60}) = \{q_3, q_5\}$ $s(v_{61}) = \{q_3\}$ $s(v_{62}) = \{q_4\}$ $s(v_{63}) = \{q_4, q_6\}$ $s(v_{64}) = \{q_1, q_5\}$ $s(v_{65}) = \{q_1\}$

$s(v_{66}) = \{q_1, q_3, q_5\}$ $s(v_{67}) = \{q_1, q_3\}$ $s(v_{68}) = \{q_1\}$ $s(v_{69}) = \{q_1, q_6\}$

Fig. 1: A search tree

To define a priority queue S to control the tree search to avoid searching futile paths, the key value for each node v in S needs to be defined. It is set to be a pair of the form: ($s(v)$, l), where l 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.

We say that a pair ($s(v)$, L) is larger than another pair ($s(v')$, L') iff $s(v) > s(v')$, or $s(v) = s(v')$, but $L > L'$. Among all the operations defined to manipulate a priority queue [6], the following two operations are very important to our algorithm:

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

In addition, for efficiency, a heuristics is employed to choose a next attribute to explore T:

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) If more than one attributes satisfy condition (1), choose a from them such that the number of queries q in $s(v)$ with $q[a] = *$ is minimized (the tie is broken arbitrarily.)

We use the above heuristics to avoid as many futile branches as possible.

By using the priority queue, we are able to change paths during the searching process so that the current path is always estimated to be with the largest possibility to a leaf node representing a largest subset of satisfied queries.

ALGORITHM 1. *PRIORITY-SEARCH*(Q, A)

```
Input: a set of queries $Q$.
Output: a most popular package $P$.
begin
1.  $i := 0$; the key of root is set to be $(|Q|, 0)$;
2.  $S := insert(root)$;          (*root represents the whole $Q$.*)
3.  while ($i \le n$) do
4.  { $(v, L) := extractMax(S)$;
5.     if $i = n$ then return the package represented by the path
           from root to $v$;
6.     pick up a next attribute $a$ from $A$ according to
           heuristics;
7.     create left child $v_l$ of $v$, representing $s(v)[\neg a]$;
8.     create right child $v_r$ of $v$, representing $s(v)[a]$;
9.     the key of $v_l$ is set to be $(|s(v)[\neg a]|, L + 1)$;
10.    the key of $v_r$ is set to be $(|s(v)[a]|, L + 1)$;
11.    $insert(S, v_l)$; $insert(S, v_r)$;
12.    $i := L + 1$;
13. }
end
```

In the above algorithm, variable i is used to count the level of the current node v. So at the very beginning the current node is *root* and its level is set to be 0 (see line 1).

In each iteration of the **while**-loop, we will extract a node v from the priority queue S with the largest key value (line 4), that is, with the largest number of queries represented by v and at the deepest level (among all the nodes in S). 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 6). Next, the key values for the two children of v will be determined (lines 9 and 10) and inserted into S (line 11). When i becomes equal to m (the number of attributes), we must have found a most popular package.

The following example helps for illustration.

Example 1 In this example, we apply the algorithm *Priority-search*( ) to the query log shown in Table 1, and make a step-by-step trace of the computation for the first three steps. For each step, we show the nodes of both S and T. We also notice that the priority queue is in fact a *max-heap* [5], but rendered as a binary tree.

In Fig. 2, we show the first 3 steps of the computation.

In the first step, the key value of the root (v_0) T is inserted into S. It is equal to (6, 0) since v_0 represents the whole Q that contains 6 queries and is a node at level 0. According to attribute *glacier*, Q is split into two subsets (which may not be disjoint due to possible * in queries), represented by the two child nodes of v_0: 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} (4, 1) and v_{11} (3, 1) are inserted into S.

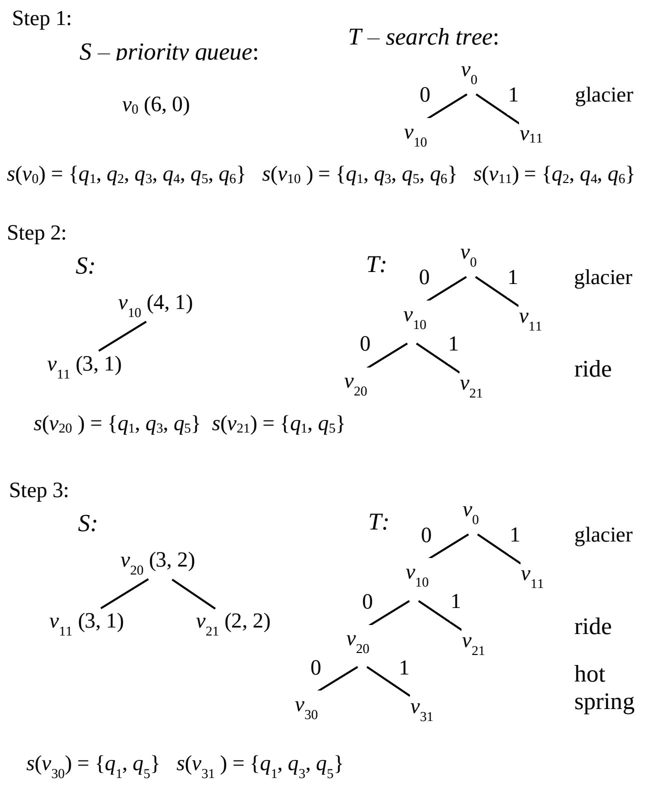


Fig. 2. A sample trace

In the second step, v_{10} (4, 1) will be extracted from S. According to attribute *ride*, $s(v_{10})$ will be further divided into two subsets, represented by its two children: v_{20} and v_{21} with $s(v_{20}) = \{q_1, q_3, q_5\}$ and $s(v_{21}) = \{q_1, q_5\}$.

In the third step, v_{20} (3, 2) will then be extracted from S. This time, according to attribute *hot spring*, $s(v_{20})$ is divided into two subsets, stored its two children respectively: v_{30} and v_{31} with $s(v_{30}) = \{q_1, q_5\}$ and $s(v_{21}) = \{q_1, q_3, q_5\}$.

In Fig. 3, we show the last step of the computation, in which $s(v_{60})$ represents a subset of queries: $\{q_1, q_3, q_5\}$, larger than any subset represented by any node currently in the priority queue S. What is important, it must be one of the largest subset of queries which can be satisfied by any selection of attributes. Then, along the path from the root to v_{60}, we can determine all the attributes by checking the labels on the path. They are {*hot spring*, *hiking*, *airline*}, satisfying q_1, q_3, and q_5. Since this is a maximum subset of satisfied queries, a most popular package is found.

From this example, we can observe a very important property of T. That is, along each path, the sizes of subsets of queries (represented by nodes) never increase since the subset of queries represented by a node is a subset of the queries represented by its parent. We call this property a *size-decreasing property*.

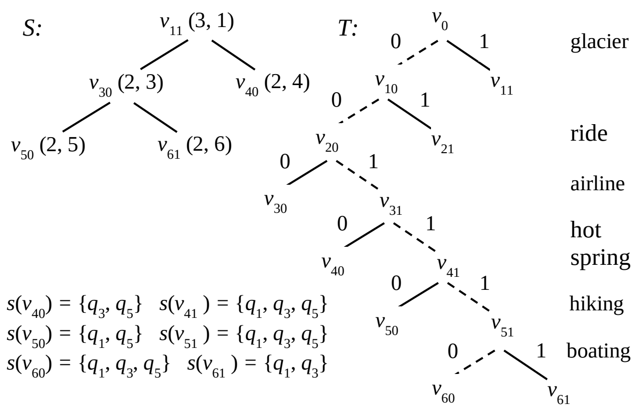


Fig. 3. A sample trace

Concerning the correctness, we have 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. We notice that at any time point, each node in the priority queue is a leaf node in the search tree T currently constructed and on the termination of the algorithm the root of S must be of the largest the key value among all such nodes. That is, this root must represent a largest subset of queries among them. Due to the size-decreasing property, it also implies a largest subset of queries among all the leaf nodes in whole T. Thus, along the corresponding path, we will find a most popular package.

In comparison with Fig. 1, the computation shown in Fig. 2 and Fig. 3 is super efficient. Instead of a binary tree of size O(2^m) created, the algorithm only explores a single root-to-leaf path (see the path represented by dashed edges in Fig. 3). However, in general, we may need to create all the nodes in T 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 give a probabilistic analysis to answer this question.

C. *Average time complexity*

From Example 1, we can see that for each internal node explored, both of its child nodes will be created. For some of them, only one of their children will be further explored as shown in Fig. 3. But for some of them, both of their children will be explored. For convenience, we call the former as 1-nodes while the latter as 2-nodes.

Remember that in T edges are labeled with either 1 or 0. Let $a = a_1a_2 \ldots a_m$ be an attribute sequence, along which the tree T is expanded level-by-level. For example, the tree shown in Fig. 3 is expanded along an attribute sequence: *glacier – ride – airline – hot spring – hiking – boating*. We will use a', a'', a''', … to designate the strings obtained by circularly shift the attributes of a, i.e., $a' = a_2 \ldots a_ma_1$, $a'' = a_3 \ldots a_ma_2a_1$, … $a^{(m)} = a = a_1a_2 \ldots a_m$. In addition, we will use $N_a(T)$ to represent the number of nodes created when applying *PRIORITY-SEARCH*( ) to Q, along a path from top to bottom.

Denote by T_1 the left subtree of T (i.e., the subtree rooted at the left child of the root), and by T_2 the right subtree of T (i.e., the subtree rooted at the right child of the root). Then, if the root of T is a 2-node, we have

$$N_a(T) = 1 + N_{a'}(T_1) + N_{a'}(T_2). \qquad (1)$$

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

$$N_a(T) = 1 + N_{a'}(T_1), \qquad (2)$$

or

$$N_a(T) = 1 + N_{a'}(T_2), \qquad (3)$$

depending on whether $s(r_l) \geq s(r_r)$ or $s(r_l) < s(r_r)$, where r_l and r_r represent the left and right child of the root, respectively.

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

$$\binom{N}{p}\left(\frac{1}{2}\right)^p\left(\frac{1}{2}\right)^{N-p} = \frac{1}{2^N}\binom{N}{p} \qquad (4)$$

Let $c_{a,N}$ 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 $N \geq 2$:

$$\text{If } root \text{ is 2-node, } c_{a,N} = 1 + \frac{2}{2^N}\sum_p \binom{N}{p} c_{a',p} \qquad (5)$$

$$\text{If } root \text{ is 1-node, } c_{a,N} = 1 + \frac{1}{2^N}\sum_p \binom{N}{p} c_{a',p} \qquad (6)$$

Let $\gamma_1 = 1$ if *root* is a 1-node, and $\gamma_1 = 2$ if *root* is a 2-node. Then, (5) and (6) can be rewritten as follows:

$$c_{a,N} = 1 + \frac{\gamma_1}{2^N}\sum_p \binom{N}{p} c_{a',p} - \delta_{N,0} - \delta_{N,1}, \qquad (7)$$

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

$$C_a(z) = \sum_{N\geq 0} c_{a,N}\frac{z^N}{N!} \qquad (8)$$

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

$$C_a(z) = \gamma_1 e^{z/2} C_{a'}\left(\frac{z}{2}\right) + e^z - 1 - z. \qquad (9)$$

According to (7), $C_a(z)$ can be rewritten as follows:

$$C_a(z) = \sum_{N\geq 0}\left(1 + \gamma_1\left(\frac{1}{2}\right)^N \sum_p \binom{N}{p} - \delta_{N,0} - \delta_{N,1}\right)\frac{z^N}{N!}$$

$$= \sum_{N\geq 0}\frac{z^N}{N!} + \sum_p \gamma_1\left(\frac{1}{2}\right)^N \sum_{N\geq 0}\binom{N}{p} c_{a',N}\frac{z^N}{N!} - \sum_{N\geq 0}\delta_{N,0}\frac{z^N}{N!}$$

$$- \sum_{N\geq 0} \delta_{N,1} \frac{z^N}{N!} \quad (10)$$

$$= e^z + \gamma_1 \sum_p \frac{(z/2)^p}{p!} \sum_{N\geq 0} c_{a',p} \frac{(z/2)^{N-p}}{(N-p)!} - 1 - z$$

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

Next, we try to get $C_{a'}(z)$, $C_{a''}(z)$, …, $C_{a^{(m-1)}}(z)$. For this purpose, we define γ_i for $i \geq 2$ as follows:

If all the nodes at level i are 1-nodes, $\gamma_i = 1$. If at least one node at level i is a 2-node, $1 < \gamma_i \leq 2$. Concretely, γ_i is calculated as follows:

$$\gamma_i = \frac{2\times num(2\text{-}nodes\ at\ level\ i) + num(1\text{-}nodes\ at\ level\ i)}{number\ of\ nodes\ at\ level\ i}. \quad (11)$$

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

$$C_a(z) = \gamma_1 e^{z/2} C_{a'}\left(\frac{z}{2}\right) + e^z - 1 - z,$$

$$C_{a'}(z) = \gamma_2 e^{z/2} C_{a''}\left(\frac{z}{2}\right) + e^z - 1 - z,$$

$$\dots \dots \quad (12)$$

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

These equations can be solved by successive transportation. For example, when transporting the expression of $C_{a'}(z)$ given by the second equation in (12), we will get

$$C_a(z) = b(z) + \gamma_1 e^{z/2} b(\tfrac{z}{2}) + \gamma_1\gamma_2 e^{z/2} e^{z/2^2} C_{a''}\left(\frac{z}{2^2}\right), \quad (13)$$

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

In a next step, we transport $C_{a''}$ into the equation given in (13). We can successively transform the equations this way until the relation is only on $C_a(z)$ itself. (Note that in this process a is circularly shifted.) That is, eventually we will get

$$C_a(z) = \gamma_1\gamma_2 \dots \gamma_m exp[z(1 - \tfrac{1}{2^m})]C_a(\tfrac{z}{2^m}) \quad (14)$$

$$+ \sum_{j=0}^{m-1} \gamma_1\gamma_2 \dots \gamma_m exp[z(1 - \tfrac{1}{2^j})](exp(\tfrac{z}{2^j}) - 1 - \tfrac{z}{2^j})$$

$$\leq 2^{m-k} exp[z(1 - \tfrac{1}{2^m})]C_a(\tfrac{z}{2^m})$$

$$+ \sum_{j=0}^{m-1} \gamma_1\gamma_2 \dots \gamma_m exp[z(1 - \tfrac{1}{2^j})](exp(\tfrac{z}{2^j}) - 1 - \tfrac{z}{2^j}),$$

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

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

$$B(z) = \sum_{j=0}^{m-1} \gamma_1\gamma_2 \dots \gamma_m exp[z(1 - \tfrac{1}{2^j})](exp(\tfrac{z}{2^j}) - 1 - \tfrac{z}{2^j}).$$

We have

$$C_a(z) = \alpha e^{\beta z} C_a(\gamma z) + B(z). \quad (15)$$

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

$$C_a(z) = \sum_{j=0}^{\infty} \alpha^j exp(\beta \frac{1-\gamma^j}{1-\gamma} z) B(\gamma^j z)$$

$$= \sum_{j=0}^{\infty} 2^{j(m-k)} \sum_{h=0}^{m-1} \gamma_1\gamma_2 \dots \gamma_h [exp(z)$$

$$- exp(z(1 - \tfrac{1}{2^h 2^{mj}}))(1 + \tfrac{z}{2^h 2^{mj}})] \quad (16)$$

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

$$c_{a,N} = \sum_{h=0}^{m-1} \gamma_1\gamma_2 \dots \gamma_h \sum_{j\geq 0} 2^{j(m-k)} D_{jk}(N), \quad (17)$$

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

$$D_{jh}(N) = 1 - (1 - 2^{-mj-h})^N - N2^{-mj-h}(1 - 2^{-mj-h})^{N-1}. \quad (18)$$

By a complex analysis, we can show that $c_{a,N} \sim N^{1-k/m}$. If $k/m \geq 1/2$, $c_{a,N}$ is bounded by

$$O(N^{0.5}).$$

Since the priority queue can have up to $O(2^{m/2})$ nodes on average, so the time complexity of *extractMax*( ) and *insert*( ) each is bounded log $2^{m/2}$ = m2. So, the average cost for generating all nodes should be bounded by $O(mN^{0.5}) \leq O(m2^{m/2})$. In addition, the whole cost for picking up an attribute to split $s(v)$ for each internal node v into $s(v_l)$ 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, v_r stand for the left and right child nodes of v, respectively. So we have the following proposition.

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

IV. EXPERIMENTS

In order to show that our method does not only have a better theoretical computational complexity than the existing methods for this problem, but is also greatly better than them in practice, we have done a lot of tests on some real data and synthetic data.

In our experiments, we have altogether tested four different methods:

1) Basic method (described in this paper, BM for short),
2) Priority-first search (discussed in this paper, PF for short),
3) Signature tree based [4] (*STB* for short),
4) Approximation method [13] (*ApM* for short).

By the signature tree based method [4], the basic method described in Section III is improved by integrating the search-tree construction and the search-tree searching into a single process. By doing this, the running time can be greatly reduced. By the method discussed in [13], the problem is reduced to the so-called MINSAT problem [12]: Given a set U of Boolean variables and a collection of disjunctive clauses over U, find a truth assignment such that the number of satisfied disjunctive clauses is minimized. Then, an approximation algorithm is applied to solve the MINSAT problem. In [13], this algorithm is referred to as *MINSAT HeuristicPD*.

All the four methods are implemented by ourselves. The code is written in in C++, running on a Linux machine with 32GB of memory and a 2.9GHz 64-core processor.

- *Data sets*

Our experiments are conducted on a real data set and a collection of synthetic data sets (query logs). For the real query log, we collected 100 customers' favourites at a Chinese restaurant and surveyed them during a large party. The investigation was designed with 10 attributes such as lemon chicken, ginger beef, honey garlic shrimp, broccoli with seafood and so on. The customers respond "yes", "no", or "don't care" to each attribute to provide their preferences. Finally, we found that for each attribute the percentage of answers 'yes', 'no', or 'don't care' each is almost 1/3 on average. Because the real query log is very small, the response time and the output quality of the algorithms cannot be really observed. We then generated a collection of larger synthetic data sets (query logs) containing up to 10000 queries with up to 30 attributes. Each query is represented by a string with each position being '0', '1', or '*', evenly populated. We may increase the number of '*' to obtain different experimental results.

- *Test results*

The performance measurement mainly focuses on the response time and the output quality, which is measured by the number of queries satisfying the corresponding found package

Test results for real data sets on SPD

In this subsection, we show the test results for the SPD problem on the real data which contains only 100 queries with 10 attributes.

Figures 3, 4 and 5 show the performance and quality of the algorithms for the real query log. From them, we can see that, the basic methos is much slower than the other three algorithms as shown in Fig. 4. However, the basic method and ours, as well as the signature-tree based method have the same optimal quality (see Fig. 3), better than the method discussed in [13]. The reason for this is that this method is just an approximate algorithm. In addition, our method requires less time than the signature-tree based method. But both of them are much better than the basic method, as shown in Fig. 6, which also shows that the space complexity of the approximation method is best.

From this test, we can see that both the time and space complexities of our method are much than the basic method and the signature-tree based method, but with the same package being obtained. The approximation method has the best time and space overhead, but with the worst quality of answers, not quite useful in practice.

Test results for varying attributes on SPD

In Figures 7, 8 and 9, we show the test results with varying numbers of attributes. From these figures, we see again that the basic method, the signature-tree based, and ours have much higher quality than the approximation method. The Fig. 7 displays that the number of satisfied queries decreases as the total number of attributes increases. The reason for this is that as more attributes are added, the queries become more selective and then more difficult to be satisfied. In general, the basic method needs too much time while the approximation method has too low quality to be used in practice although the time required by this method is the best among all the four methods. As we can see from Fig. 8 and 9, the time and space requirements of our method are also very low, just a little bit worse than the approximation method.

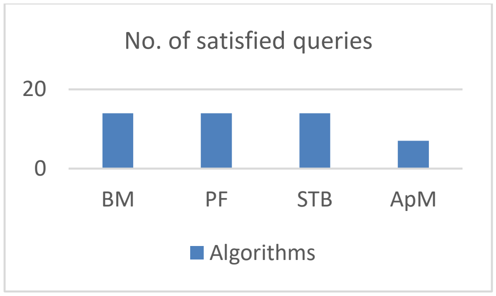


Fig. 4. Quality for Real data sets

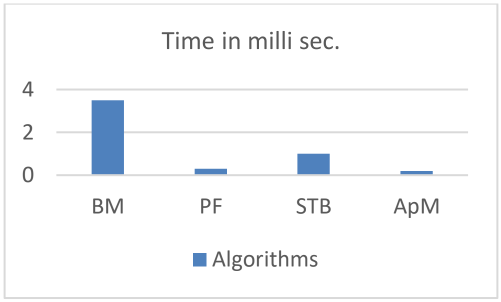


Fig. 5. Time cost for Real data sets

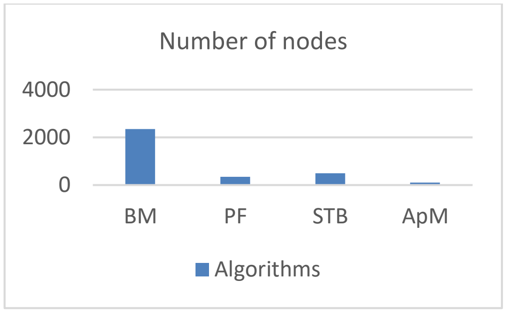


Fig. 6. Space requirement for Real data sets

Test results for varying query log size on SPD

In this subsection, we show the test results for varying sizes of query logs.

Figures 10, 11, and 12 show the results with varying query log sizes. From these figures, we see that the signature-tree based is still more efficient than the basic method and the approximation method. But our is the base. Similar to the previous experiment, the approximation method uses the least time, but has the worst quality; and ours performs better overall.

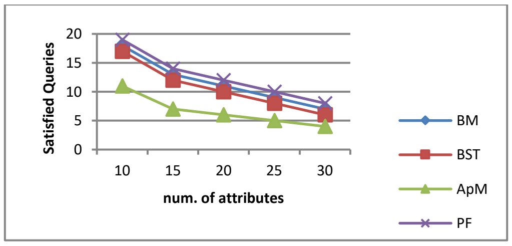


Fig. 7. Quality for varying num. of attributes for 100 queries

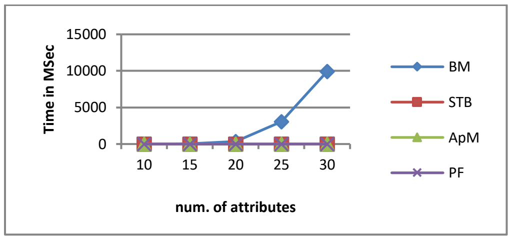


Fig. 8. Time cost for varying attributes for 100 queries

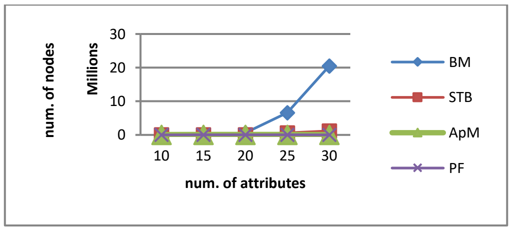


Fig. 9. Space requirement for varying attributes for 100 queries

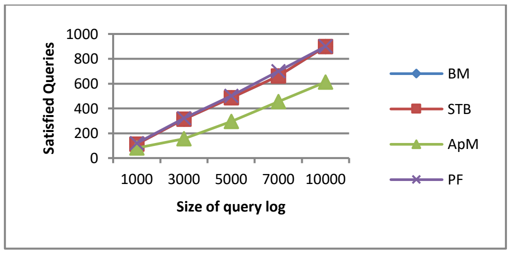


Fig. 10. Quality for varying of query log sizes with 15 attributes

V. Conclusion

In this paper, we presented a new method to solve the single package design (*SPD*) problem by using a new tree search method, by which a priority queue is 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 motivation of this work is to select, from a given set, a subset of the elements according to a query log to satisfy as many customers as possible and to overcome the limitation of the current packages design methods. SPD is a useful extension of the frequent pattern mining problem.

Extensive experiments have been conducted, which show that in general our algorithms are able to find better packages by using almost the same time as the exiting method for this problem.

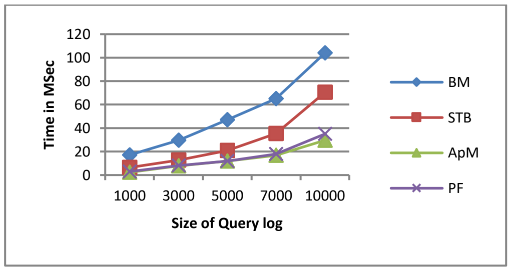


Fig.11. Time cost for varying of query log sizes with 15 attributes

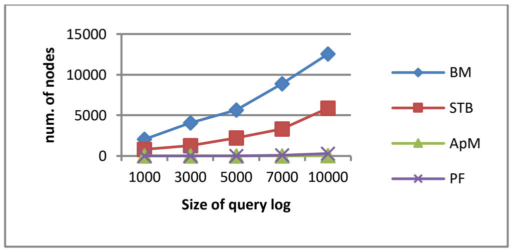


Fig. 12. Space requirement for varying of query log sizes with 15 attributes

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami, Mining Association Rules between Sets of Items in Large Databases, in: *Proc. SIGMOD Conf.*, Washington DC, USA, May 1993, pp. 207-216.

[2] Y. Chen, Signature files and signature trees, *Information Processing Letters*, 82(4):213-221, 2002.

[3] Y. Chen, On the signature trees and balanced signature trees. In *Proc. of 21th Conference on Data Engineering*, pages 742-753, Tokyo, Japan, April 2005.

[4] Y. Chen, and W. Shi, On the Designing of Popular Packages, in *Proc. IEEE Conf. on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics*, Halifax, Canada, 2018, pp. 937-944.

[5] T. H. Corman, C. E. Leierson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, McGraw Hill, 2002.

[6] M. R. Garey and D. S. Johson, Computers and Intractibility: A Guide to the Theory of NP-Completeness, W. H. Freeman, San Francisco, CA, 1979.

[7] T. S. Gruca and B. R. Klemz, Optimal new product positioning: A genetic algorithm approach, *European Journal of Operational Research*, 146, 3, 2003, 621-633.

[8] F. Grandi, P. Tiberio, and 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*, pp. 286-297, Copenhagen, Denmark, June 1992.

[9] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Ebook 2005.

[10] D. L. Lee, Y. M. Kim, and G. Patel, Efficient signature file methods for text retrieval, *IEEE Transactions on Knowledge and Data Engineering*, 7(3):423-435, 1995.

[11] J. Han, J. Pei, and Y. Yin, Mining Frequent Patterns without Candidate Generation, *MOD 2000*, Dallas, TX USA, ACM, pp. 1-12.

[12] R. Kohli, R. Krishnamurti, and P. Mirchandani, The Minimum Satisfiability Problem, *SIAM J. Discrete Math*, 1994, pp. 275-283.

[13] M. Miah, Most Popular Package Design, in Proc. *Conference for Information Systems Applied Research*, 2011 Conisar Proceedings, pp. 1-7.