On the Designing of Popular Packages

Yangjun Chen[1], Wei Shi[2]
Dept. Applied Computer Science
University of Winnipeg, Canada
[1]y.chen@uwinnipeg.ca, [2]shiwei8980923@gmail.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 problem and propose two algorithms to find most popular packages based on the signature techniques. One is for finding single packages while the second is devoted to the design of multiple packages, by which a binary tree for a query log is constructed in a way as a signature tree for a signature file. Extensive experiments have also been conducted, which show that our method for this problem is promising.**

Key words: *Data mining; Single package design; multiple package designs; Signature files; Signature trees*

I. Introduction

Over the last decade, with the rapid development of science and technology, economy and society have greatly progressed. In the meantime, it produces large amounts of data in various fields, such as the human exploration of space, the daily bank transaction data, just to name a few. People were beneficial from exploration and analysis of these data to get useful information. However, although the development of the database technology makes quick data process possible, the face of ever-increasing flood of data is no longer satisfied with its query capabilities. There is a deep-seated problem: whether we can improve the efficiency of extracting information or knowledge over data, which is critical to decision making by managers.

In this paper, we discuss two interesting problems: *Single Package Design* problem (*SPD* for short) and *Multiple Package Design* problem (*MPD* for short), both related to the information extraction [13 – 20]. We will be given a set of activities or items $A = \{a_1, ..., a_m\}$, like *hot spring*, *riding horse*, *glacier*, *hiking*, *airlines*, *boating* and so on (by a travel agency), referred to as an attribute, an elements, or a feature; and a *query log* $Q = \{q_1 \ldots q_n\}$ with each q_i ($i = 1, \ldots, n$) being a 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'. By the *SPD*, 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 *new package*. Thus, what we want is to ensure that the new package satisfies as many queries as possible [6]. For example, for the above vacation package, clients give their preferences by specifying *yes*, *no*, or '*don't care*' for each element to form a query log. The design of a new package is to pick a sub-set of these elements to meet as many queries' requirements as possible.

By the multiple package design (MPD) problem, we will find a minimum set of packages to meet all the queries' needs. That is, instead of a single package, we will find a minimal number of packages to satisfy all the queries in a query log.

In this paper, we address this issue, and propose two new algorithms, based on the *signature trees* discussed in [7] to evaluate the *SPD* and *MPD*, respectively. Concretely, the main idea of our methods can be summarised as follows.

- Signature trees are extended to handle three values: 0, 1, and ? (don't care).
- The search of a signature tree is integrated into the construction of the signature tree. In this way, not only much space (for storing a signature tree), but also much running time can be saved by the pruning of search space.

A lot of experiments have been conducted, which show that our methods are much better than the existing method for this problem..

The remainder of the paper is organized as follows. In Section II, we briefly describe what is a signature file and a signature tree, based on which our methods are established. Then, in Section III and IV, the algorithms for evaluating the *SPD* and the *MPD* will be discussed, respectively. Section V is devoted to the test results. Finally, a short conclusion is set forth in Section VI.

II. Signature files and signature trees

In the advent of WWW, large sets of data need to be manipulated efficiently. Especially, we need to handle complex data structures with set-valued attributes which can be represented as bit strings called *signatures* in some areas, such as digital libraries, data-mining, and hypertext and multimedia systems. A group of signatures can be stored in a file called a *signature file* [25 – 30], as illustrated in Fig. 1(a).

s_1:	0	0	1	0	1	1	1	1	0
s_2:	1	0	0	0	1	1	1	1	0
s_3:	0	1	0	1	0	0	0	1	1
s_4:	1	0	0	0	0	1	1	0	1
s_5:	1	0	0	1	1	0	0	0	1
s_6:	0	0	1	0	0	1	1	1	0
s_7:	1	0	0	1	0	0	0	1	1

(a)

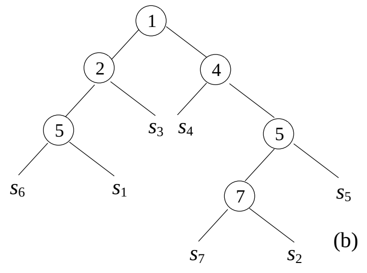


Fig. 1: A Signature file and a signature tree

A signature tree for a signature file $S = s_1. s_2 \ldots s_n$, where $s_i \neq s_j$ for $i \neq j$ and $|s_k| = m$ for $k = 1, \ldots, n$, is a binary tree T such that [7 - 12]

1. For each internal node of T, the left edge leaving it is always labeled with 0 and the right edge is always labeled with 1.
2. T has n leaves labeled 1, 2, ... , n, used as pointers to n different signatures: $s_1, s_2, \ldots, s_n$ in S. Let v be a leaf node. Denote $p(v)$ the pointer to the corresponding signature.
3. Each internal node v is associated with a number, denoted by $k(v)$, to tell which bit will be checked.
4. Let $j_1, \ldots, j_h$ be the numbers associated with the nodes on a path from the root to a leaf v labeled i (then, this leaf node is a pointer to the ith signature in S, i.e., $p(v) = i$). Let $p_1, \ldots, p_h$ be the sequence of labels of edges on this path. Then, $(j_1, p_1) \ldots. (j_h, p_h)$ makes up a signature identifier for s_i; $s_i(j_1, \ldots, j_h)$ since it must be unique for s_i.

As an example, see the signature tree shown in Fig. 1(b), which is constructed over the signature file shown in Fig. 1(a). In the signature tree, each root-to-leaf path is an identifier for a signature in the signature file shown in Fig. 1(a).

III. ALGORITHM FOR THE SPD

Consider a vacation package recommended by a tourist agent, which may contain a set of activities, like *hot spring*, *riding horse*, *glacier*, *hiking*, *airlines*, *boating* and so on. Such a package can be represented as a tuple of the form: $A = \{a_1 \ldots a_m\}$, where each a_i ($i = 1, \ldots, m$) stands for an activity. Accordingly, a query q_j is a string of the form: $q_{j1}q_{j2} \ldots q_{jm}$ with each $q_{jk} \in \{0, 1, ?\}$, where $q_{jk} = 1$ indicates that a_k is selected, and $q_{jk} = 0$ indicates that a_k is not selected while '?' means 'don't care'.

The general package design problem can then be described as follows. Given a query log (or say, a query set or a workload) Q, 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. t is referred to as a new package. Thus, what we want is to ensure that the new package satisfies as many customers as possible. For example, for the above vacation package, clients give their preferences by specifying yes, no, or 'don't care' for each element to form a query log. t should be a sub-set of activities to meet as many customers' needs as possible.

For example, Table 1 shows a query log for a vacation package application. It contains $S = 6$ queries, $M = 6$ attributes (activities), and each query represents a user's favourites. For instance, the query q_1 = (1, ?, 0, ?, 1 ,?), indicates that *hot spring* and *airlines* are q_1's favourites, but glacier is 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

Furthermore, q_1 does not care about whether *riding*, *hiking* or *boating* is available or not.

A. Basic algorithm

Our basic method works in two steps. In the first step, we construct a signature-tree-like structure, called a *SPD-tree*. Then, in the second step, we search the SPD-tree to find a best popular package.

Let $Q = \{q_1, \ldots, q_m\}$ be a query log. We use $q_i[j]$ to represent the value of the jth attribute in q_i ($i = 1, \ldots, m$). Starting from the first attribute value, we divide all queries in Q into two branches. For query q_i ($1 \leq j \leq M$), 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].

In a next step, we will split both left and right branches in the same way as above.

Repeating this process until all the attributes are checked, we will establish a binary tree, as shown in Fig. 2.

The following is a formal description of this working process (Algorithm 1 and Algorithm 2).

ALGORITHM 1. *SPD-tree Construction*

```
Input: a query log $Q = \{q_1, \ldots, q_m\}$, represented as a matrix.
Output: an SPD-tree.
begin
1. create the root node $v$;
2. call method ConstructSPDTree($v$, $Q$, 1);
end
```

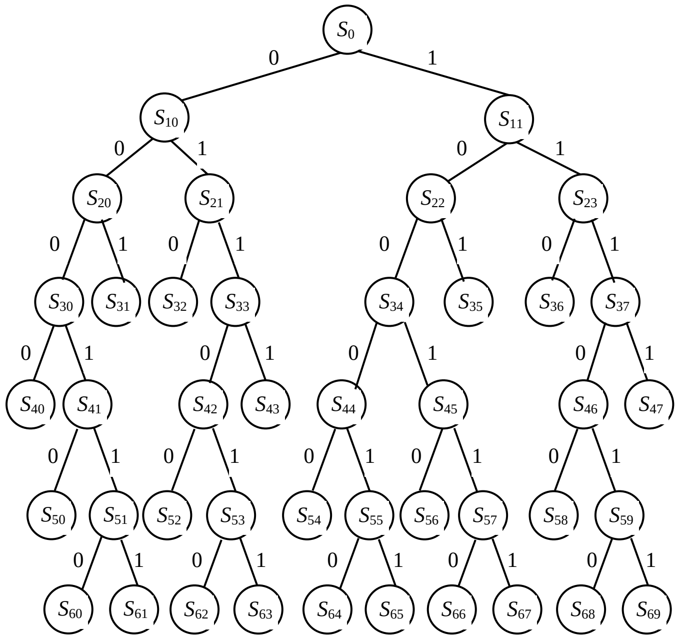
ALGORITHM 2. *ConstrucSPDTree*(p, Q, j)

```
Input: current parent node $p$, query log $Q$, and an integer $j$ to indicate the $j$th attribute.
Output: an SPD-tree.
begin
1.  $m \leftarrow Q.size$;
2.  for $i \leftarrow 1$ to $m$
3.  {
4.    if $Q[i][j]$ = '?' then add $Q[i]$ into $l$ and $r$
5.    else
6.      if $Q[i][j] = 0$ then add $Q[i]$ into $l$ else add $Q[i]$ into $r$;
7.  }
8.  if $l.size > 0$ then
9.  {create a new node $v$; $p.LeftChild \leftarrow v$; $v.ParentNode \leftarrow p$;
10.   if $j + 1 < m$ then call ConstructSPDTree($v, l, j + 1$);
11.   else { $v.LeftChild \leftarrow \emptyset$; $v.RightChild \leftarrow \emptyset$; $v.sig \leftarrow l$; }
12. }
13. if $r.size > 0$ then
14. {Create a new node $w$; $p.RightChild \leftarrow w$; $w.ParentNode$
            $\leftarrow p$;
15.   if $j + 1 < m$ then call ConstructSPDTree($w, r, j+1$);
16.   else {$w.LeftChild \leftarrow \emptyset$; $w.RightChild \leftarrow \emptyset$; $w.sig \leftarrow r$;}
17. }
end
```

In the above algorithm, p, l, r represent the parent, left child, and right child of the current node v, respectively. Once a SPD-tree is constructed, the best popular package can be easily found by checking all the leaf nodes. During the process, for each encountered leaf node v we compute how many queries are satisfied by the node and return the node with the maximum count, denoted as $v.C$. We can establish the corresponding best popular package by traversing the path from v up to the root and assigning attribute values accordingly (see Algorithm 3).

```
ALGORITHM 3. packageSearch
Input: the root node $p$ of a SPD-tree.
Output: the best popular package $P$.
begin
1. let $v_1, v_2, \ldots, v_k$ be all the leaf nodes;
2. find all those leaf nodes $v_j$ such that $v_j.C$ is maximum;
3. for each $v_j$ return the labels on the path from $v_j$ to the root;
end
```

In Fig. 2, we demonstrate the construction of the SPD tree over the query log shown in Table 1.



$S_0 = \{q_1, q_2, q_3, q_4, q_5, q_6\}$ $S_{10} = \{q_3, q_4, q_5, q_6\}$ $S_{11} = \{q_1, q_2, q_3, q_5, q_6\}$

$S_{20} = \{q_3, q_4, q_5\}$ $S_{21} = \{q_4, q_5\}$ $S_{22} = \{q_1, q_2, q_3, q_5\}$ $S_{23} = \{q_1, q_5\}$

$S_{30} = \{q_3, q_5\}$ $S_{31} = \{q_4\}$ $S_{32} = \{q_6\}$ $S_{33} = \{q_4, q_6\}$ $S_{34} = \{q_1, q_3, q_5\}$ $S_{35} = \{q_2\}$ $S_{36} = \{q_1, q_6\}$ $S_{37} = \{q_6\}$

$S_{40} = \{q_3\}$ $S_{41} = \{q_4, q_5\}$ $S_{42} = \{q_4, q_6\}$ $S_{43} = \{q_4\}$ $S_{44} = \{q_1, q_5\}$ $S_{45} = \{q_1, q_3, q_5\}$ $S_{46} = \{q_1, q_6\}$ $S_{47} = \{q_1\}$

$S_{50} = \{q_3\}$ $S_{51} = \{q_3, q_5\}$ $S_{52} = \{q_6\}$ $S_{53} = \{q_4, q_6\}$ $S_{54} = \{q_5\}$ $S_{55} = \{q_1, q_5\}$ $S_{56} = \{q_5\}$ $S_{57} = \{q_1, q_3, q_5\}$
$S_{58} = \{q_1\}$ $S_{59} = \{q_1, q_6\}$

$S_{60} = \{q_3, q_5\}$ $S_{61} = \{q_3\}$ $S_{62} = \{q_4\}$ $S_{63} = \{q_4, q_6\}$ $S_{64} = \{q_1, q_5\}$ $S_{65} = \{q_1\}$ $S_{66} = \{q_1, q_3, q_5\}$ $S_{67} = \{q_1, q_3\}$
$S_{68} = \{q_1\}$ $S_{69} = \{q_1, q_6\}$

Fig. 2: An SPD tree

From this figure, we can see that the leaf node S_{66} has the largest size and the path from the root to it shows a best package: {*hot spring*, *hiking*, *airlines*}.

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

First, we notice that an SPD-tree will have exactly m levels, where m is the length of each query in Q. So, in the worst case, the number of nodes in an SPD-tree is bounded by $O(2^{m-1})$. Since each node represents a subset of Q, the whole size of it is bounded by $O(2^{m-1}|Q|) = O(2^{m-1}n)$. Accordingly, the time for constructing an SPD-tree and the time for searching an SPD-tree are both bounded by $O(2^{m-1}n)$. It is because in the worst case each node in an SPD-tree will be accessed.

B. Approximate algorithm and heuristics

According to [6, 21], the package design problem is *NP*-hard and its running time is exponential in the number of attributes in a query log Q. Thus, when the number of attributes is large, to find an optimal solution to the problem is not feasible. We have to resort to approximation. For this purpose, we integrate the SPD-tree construction with the SPD-tree search to have a heuristic algorithm. By doing this, we can achieve the following advantages.

1. By recording the best results up to now, we need only to dynamically maintain a single path currently being explored.
2. Lots of subtrees can be cut off by checking whether a subtree possibly contains a best popular package or not.
3. We can establish some kinds of heuristics to guide us to explore the most promising paths or reduce the whole searching time.

If we want to find only one most popular package, at any point of time, we need only to record a path corresponding to the best result up to now, and maintain a current path to explore. So the space requirement is reduced to $O(mn)$. However, if we need to keep all the possible most popular packages, in the worst case, we have to use $O(\binom{m}{m/2})$ space to record them.

In order to prune all the possible futile subtrees, we use the following general rule:

If the number of queries in any branch in a subtree is smaller than the number of queries in the candidate result, the subtree can be simply pruned.

From this general rule, a set of rules can be derived to handle different specific cases.

i) If for all the queries represented by a node v, the attribute to be checked contains only '0', or '?', the subtree rooted at the right child of v can be pruned.

ii) If for all the queries represented by a node v, the attribute to be checked does not contains '0', and at least one of them contains '1', the subtree rooted at the left child of v can be cut off.

iii) If for all the queries represented by a node v, the attribute to be checked contains only '?', we cut off the subtree rooted at the right child of v. (Notice that we can also prune the subtree rooted at the left child of v. But the result will be same.)

Finally, identifying a good splitting attribute (node) is important as this can make the tree either balanced or skewed. For this, our heuristic is:

- Choose the attribute with the minimum number of "?" values to minimize the selection for *don't care*.
- If more than one column contains the same number of '?', we continually calculate the number of 1s and the number of 0s in them. We select the column in which the number of 1s and the number of 0s are mostly closed to each other to keep the tree balanced.

Example 1 Generating the SPD tree for the query log shown in Table 1, but integrated with the search for a best package, we will get a set of attributes {*hot spring*, *hiking*, *airline*}, satisfying three queries: q_1, q_3, and q_5. In Fig. 3, we show the whole working process with the pruning rule being applied.

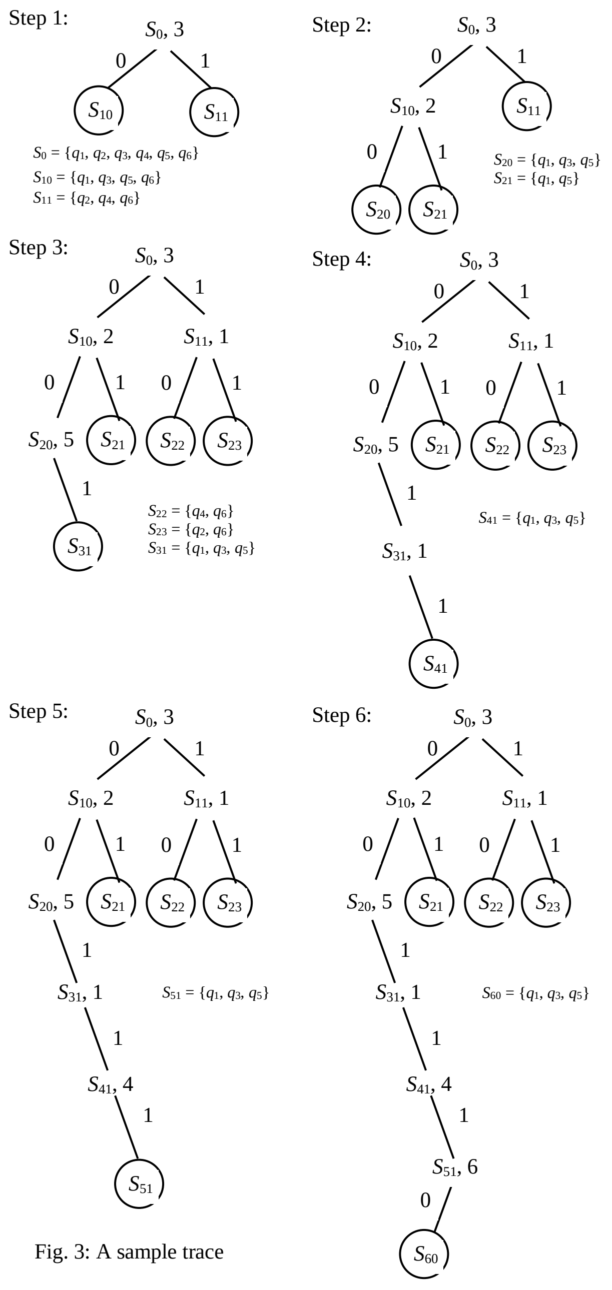


Fig. 3: A sample trace

In the first step, we find the third column which contains the minimum number of '?'. Set this column to be the dividing position, and create a new node v, marked with $<S_0, 3>$. At the same time, all those queries q_i ($1 \le j \le n$) with $q_i[3]$ = '?' or $q_i[3]$ = '0' will be put in its left child. They are $S_{10} = \{q_1, q_3, q_5, q_6\}$ while all those with $q_i[3]$ = '?' or $q_i[3]$ = '1', i.e., $S_{11} = \{q_2, q_4, q_6\}$, will be put in its right child. We notice that $S_{10} \cap S_{11} \neq \Phi$ (empty set).

In the next step, because the number of the queries in the right node v_2 is not greater than that in the left node v_1, the whole subtree rooted at v_2 can directly be pruned. But the left child v_1 representing S_{10} can further be split according to the second column in Q (query log), generating two children of v_1: v_{11} and v_{12}, representing $S_{20} = \{q_1, q_3, q_5\}$ and $S_{21} = \{q_1, q_6\}$, respectively. Again, the subtree rooted at v_{12}, the node representing S_{21} can be completely pruned since S_{21} is smaller than S_{20}. In step 3, we will decompose S_{20} in terms of the fifth column in Q, which contains minimal number of '?'. Since this column also contains no '0', v_{11} has only a right child, representing $S_{31} = \{q_1, q_3, q_5\}$. Continuing this process, we will explore a path as shown in Fig. 2, reaching a node representing $S_{51} = \{q_1, q_3, q_5\}$. Splitting S_{51} in terms of the sixth column, we will create only the left child of this node since this column in S_{51} does not contain '1'. Finally, we get the package result when the SPD tree is constructed. Along the path generated, we have $a_3 = 0$, $a_2 = 0$, $a_5 = 1$, $a_1 = 1$, $a_4 = 1$, $a_6 = 0$, showing a popular package p = [1 0 0 1 1 0].

The time complexity of this modified SPD tree generating algorithm can be analyzed as follows. At each tree level, the number of checked bits is $O(nm)$, where n, m are the queries and attributes, respectively. The SPD tree is of m levels. So the time complexity is bounded by $O(nm^2)$.

IV. ALGORITHM FOR THE MPD

By the *multiple package design* (*MPD*) problem, we will find a minimum set of packages to meet all the customers' needs. That is, instead of a single package, we will find a minimal number of packages to satisfy all the queries in a query log. Below is the definition of the problem:

Multiple Package Design Problem (*MPD*): Given a query log Q with a set of attributes (activities), design a minimal number of packages such that for each query in Q there exists a package satisfying it.

An Algorithm for the *MPD* includes a multiple application of the algorithm for *the SPD*. Two methods for the MPD will be presented. One is based on the signature tree method for the *SPD* while the other is based on the heuristic signature tree method for the *SPD*.

For both of them, the following process will be carried out:

(1) We execute the algorithms for the *SPD* and add the found most popular package P' to the result.

(2) Then, we delete all those queries Q' from the query log, which are satisfied by the package found in the first step.

(3) Repeat the above two steps until no further queries or attributes are left.

Below is the first algorithm for the MPD, designed according to the above general strategy.

ALGORITHM 4. *MPD based on signature trees*

```
Input: a set of queries $Q$.
Output: a set of most popular packages $P$.
begin
while ($Q \neq 0$)
1.  { create the root node $v$;
2.     call method $ConstructSPDTree(v, Q, 1)$;
3.     $\{P', Q'\} \leftarrow searchPackage\ (v)$;
4.     $Q \leftarrow Q \backslash Q'$;
5.     $P = P \cup P'$;
6.  }
end
```

It is an approximate algorithm and in the set of packages we may have some packages with uncertain values (for attributes).

The second algorithm is an approximate algorithm, based on the heuristic method discussed in Subsection *B* of Section III..

ALGORITHM 5. *MPD based on modified signature trees*

```
Input: a set of queries $Q$.
Output: a set of most popular packages $P$.
begin
while ($Q \neq 0$)
1.  { create the root node $v$;
2.     $\{P', Q'\} \leftarrow ConstructSPD(v, Q, 1)$;
3.     $Q \leftarrow Q \backslash Q'$;
4.     $P = P \cup P'$;
5.  }
end
```

V. EXPERIMENTS

In this section, we report the experiment results. We have mainly implemented three different methods: the signature-based method, its modified version, and the method proposed in [6].

The performance measurement mainly focuses on the response time and the output quality. For the SPD, the output quality is measured by the number of queries satisfying the corresponding created package. For the MPD, the output quality is measured by the number of packages in the set found by the algorithms.

The experiments are organized in two categories. The first category is to evaluate the algorithms for finding an optimal single package design while the second category is for finding an optimal multiple package design.

All the experiments are performed on a Sony notebook with a 2.53Ghz Inter Core i3 CPU, with 300 GB hard disk and 8.0GB of memory. The code is written in C++ and run on Windows 7 professional with 32-bit operating system.

- *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' favorites 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.

- *Tested methods*

In the experiment, we have tested three methods:

- **Signature tree for SPD -** It works in two steps. In the first step, we construct a signature-tree-like structure, call a SPD-tree. Then, in the second step, we search the SPD-tree to find the best popular package.
- **Heuristic signature tree for SPD -** The basic algorithm presented in Subsection A of Section III can be dramatically improved by integrating the SPD-tree construction and the SPD-tree search into a single process. By doing this, we can achieve an optimization in both response time and package quality.
- **Heuristic SPD -** This algorithm was proposed by Miah [6]. This is in fact an algorithm to find an approximate solution to an NP-complete problem, the so-called MINSAT problem: Given a set *U* of Boolean variables and a collection of disjunctive clauses over *U*, a truth assignment was found that minimizes the number of satisfied disjunctive clauses. In [6], this algorithm is referred to as *MINSAT HeuristicPD*.
- In addition, all the above three algorithms are extended in a way described in Section IV to solve the *MPD* problem.

A. *Experiments on SPD*

- *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. Obviously, the signature tree SPD is much slower than the other two algorithms as shown in Fig. 3. However, the signature tree SPD and the heuristic signature tree SPD have the same optimal quality (see Fig. 3), better than the Heuristic SPD. The reason for this is that the Heuristic SPD is just an approximate algorithm. The heuristic signature tree SPD requires much less time than the signature tree SPD since the number of the nodes generated by it is significantly less than the signature tree SPD as shown in Fig. 5, which also shows that much less memory space is required. So, the heuristic signature tree SPD has a better overall performance than the other two methods.

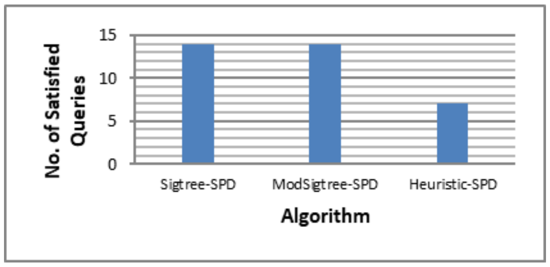


Fig. 3: Quality for Real data sets

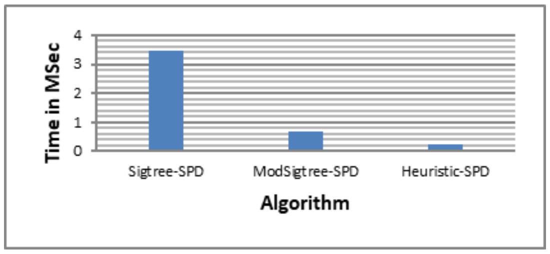


Fig. 4: Time cost for Real data sets

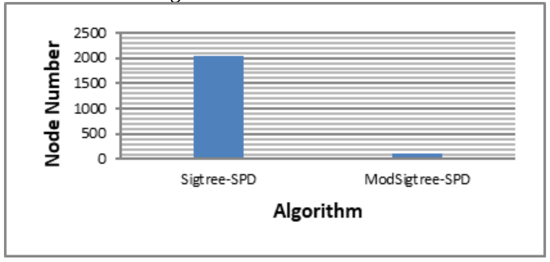


Fig. 5: Space requirement for Real data sets

- *Test results for varying attributes on SPD*

In Figures 6 to 8, we show the results of this test with varying numbers of attributes. From these figures, we see that both the signature tree SPD and the heuristic signature tree SPD again have higher quality than the heuristic SPD. The Fig. 5 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 signature tree SPD needs too much time while the heuristic SPD has too low quality to be used in practice although the time required by heuristic SPD is the best among the three methods. As we can see from Fig. 7, the space requirement of our heuristic signature tree method is also very low.

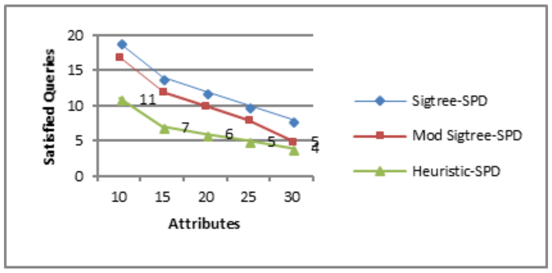


Fig. 6: Quality for varying of attributes for 100 queries

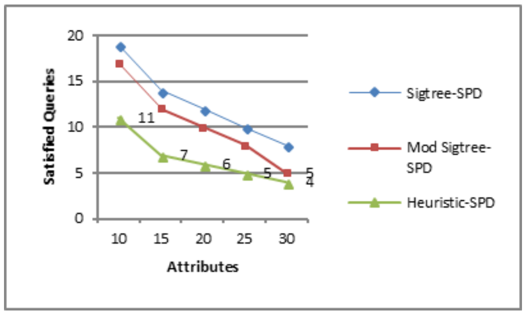


Fig. 7: Time cost for varying attributes for 100 queries

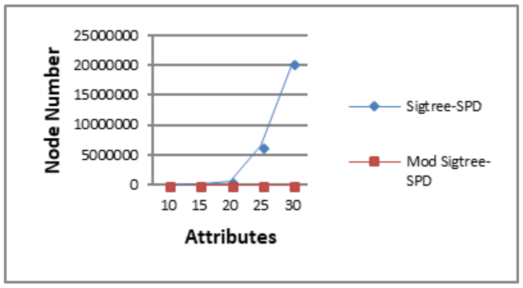


Fig. 8: Space requirement for varying attributes for 100 queries

- *Test results for varying query log size on SPD*

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

Figure 9, 10, and 11 show the results with varying query log sizes. From these figures, we see that the heuristic signature tree SPD is still more efficient than the other two methods. Similar to the previous experiment, the heuristic SPD method uses the least time, but has the worst quality. The heuristic signature tree SPD performs better overall.

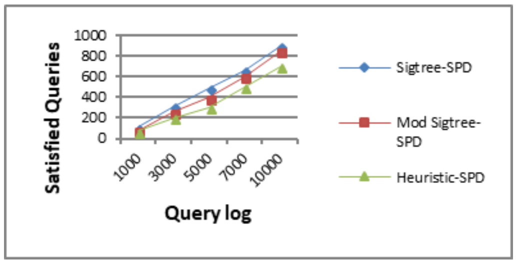


Fig. 9: Quality for varying of query log sizes with 15 attributes

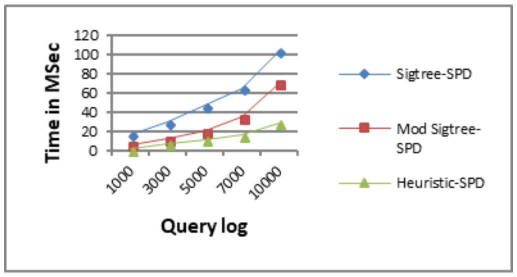


Fig.10 : Time cost for varying of query log sizes with 15 attributes4

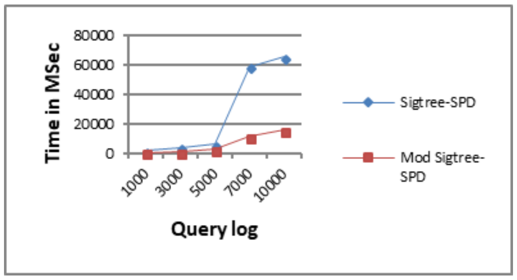


Fig. 11: Space requirement for varying of query log sizes with 15 attributes

B. *Experiments for MPD*

The MPD problem is an *NP*-complete problem. Therefore, it is not feasible to try to find an optimal solution to it. First, the number of possible candidate packages is exponential in the number of attributes. Secondly, the number of different combinations of packages is also exponential in the number of packages. So we will only provide the experiment results of some approximate algorithms for MPD. For the MPD problem, instead of designing a single package as in SPD, we need to find a minimum set of packages to satisfy all the customers' needs. Therefore, a better quality of MPD will have a smaller number of package counts. As shown in Fig. 5, the number of satisfied queries decreases as the total number of attributes increases since more attributes in a query makes the query more selective and difficult to be satisfied.

- *Test results on real data sets for MPD*

In this subsection, we report the test results on the real data for MPD.

Fig. 12 and 13 show the performance and the quality of the three algorithms for the real data set, respectively. Again, the signature tree MPD is much slower than the other two algorithms as shown in Fig.13. However, the signature tree MPD and the heuristic signature tree MPD are relatively similar in quality. The heuristic MPD requires the least time, but it has the worst quality.

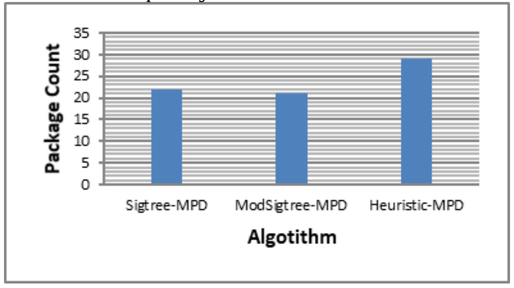


Fig. 12: Quality for Real data sets for MPD

- *Test results on varying attributes for MPD*

Now we show the test results on varying attributes for MPD.

From Fig.14 and 15, we can see almost the same results as the previous experiment. That is, the signature tree MPD and the heuristic signature tree MPD are relatively similar in quality, better than the Heuristic MPD. Fig. 13 shows that the number of packages counts increases as the total number of attributes increases, and Fig. 15 shows that the time used by the heuristic SPD is just a little bit better than the heuristic signature tree MPD.

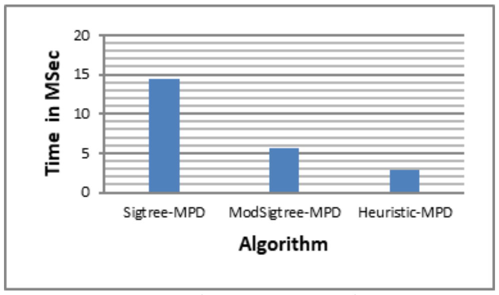


Fig. 13: Time for Real data sets for MPD

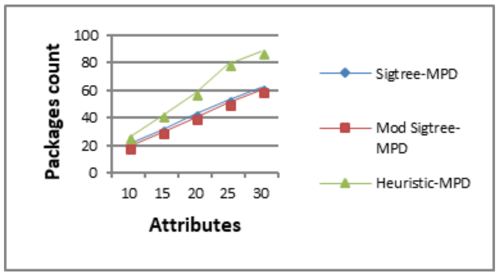


Fig. 14: Quality on varying of attributes for 100 queries for MPD

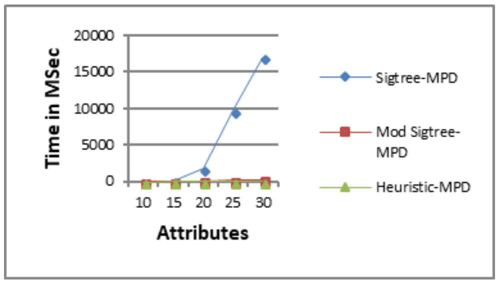


Fig. 15: Time on varying attributes for 100 queries for MPD

- *Test results on varying query log sizes for MPD*

In this subsection, we show the test results on varying query log sizes for the MPD.

Fig. 16 and 17 show the results of this rest. Unlike the above two tests, the signature tree MPD has the best quality and less time than the heuristic signature tree method. That is because each time the heuristic signature tree MPD completes a search, only part of the queries from the query log is deleted, and for the remaining queries a new tree needs to be reconstructed. So, more time is required. However, the signature tree MPD method needs to build only one tree, and then complete the search. It does not need to construct the tree

for several times. Similar to the previous tests the heuristic MPD still takes less time, but has the worst quality.

VI. Conclusion

In this paper, we presented two new methods to solve the Package Design (*SPD*) problem and the Multiple Package Design (*MPD*) problem, respectively, based on the signature tree techniques. 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.

Our work mainly comprises two parts. The first part is for the SPD, by which we will find a single package which can satisfy a maximum group of customers. The second part is for the MPD, by which we try to figure out a minimum set of packages to satisfy all the customers' needs. We have proposed two algorithms to solve each of them. One is based on the traditional signature tree algorithm and the other is based on a modified signature tree algorithm.

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.

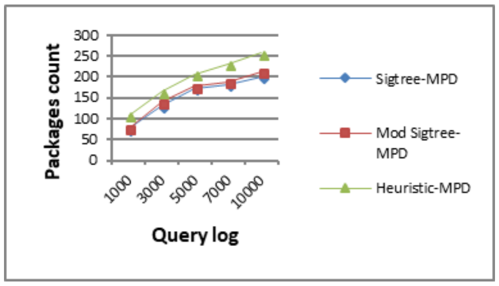


Fig. 16: Quality on varying of query log sizes with 15 attributes

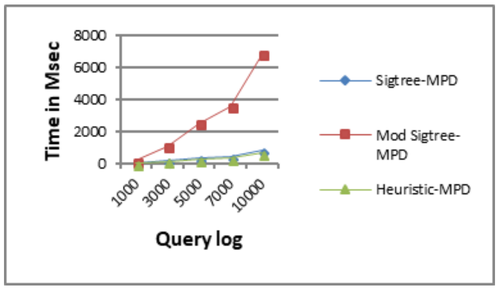


Fig.17: Time for varying of query log size for 15 attributes

REFERENCES

[1] M. G Ceruti, B. Thuraisingham. Dependable objects for databases, middleware and methodologies, in *Proc. Fifth International Workshop on Object-Oriented Real-Time Dependable Systems*, 1999, IEEE, 1999: 75-78.

[2] D C. Hand, J. Lamartine, G. M Essenfelder, Z.Kibar, et al., Mutations in GJB6 cause hidrotic ectodermal dysplasia, *Nature genetics*, 2000, 26(2): 142-144.

[3] W. Han, Campbell, Data Mining: Concepts and Techniques: *Mechanical Engineering*, I Press, Beijing, 2001 : 232-233 .

[4] T. M. Connolly and C. E. Begg, *Database Systems*, Addison Wesley, 233 Spring Verlag, New York, 2002.

[5] M. Liu, *Data mining technology and its application*, Defence Industry Press , 2001.

[6] M. Miah. Most Popular Package Design, in Proc. *Conference for Information Systems Applied Research*, 2011 Conisar Proceedings.

[7] Y. Chen and Yibin Chen, On the Signature Tree Construction and Analysis, *IEEE Trans. Knowl. Data Eng.* 18(9), 2006.

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

[9] Y. Chen and Y.B. Chen, Signature File Hierarchies and Signature Graphs: A New Indexing Method for Object-Oriented Databases, *Proc. ACM Symp. Applied Computing (SAC '04)*, pp. 724-728, 2004.

[10] 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.

[11] Y. Chen, On the general signature trees, in *Proc. Database and Expert Systems Applications*, Springer Verlag, Berlin Heidelberg, 2005: 207-219.

[12] Y. Chen. On the cost of searching signature trees, *Inf. Process. Lett*. 99(1): 19-26 (2006).

[13] A. D. Shocker and V. Shrinivasan, A consumer-based methodology for the identification of new product ideas, *Management Science*, 20, 6 (Feb 1974), 921-937.

[14] S. Albers and K. Brockhoff, A procedure for new product positioning in an attribute space, *European Journal of Operational Research*, 1, 4 (Jul 1977), 230-238.

[15] S. Albers and K. Brockhoff, Optimal Product Attributes in Single Choice Models, *Journal of the Operational Research Society*, (1980) 31, 647–655.

[16] S. Albers, and K. Brockhoff, A procedure for new product positioning in an attribute space. European Journal of Operational Re-search. 1, 4 (Jul 1977), 230-238.

[17] D. M. Albritton and P. R. McMullen, Optimal product design using a colony of virtual ants, *European Journal of Operational Research*, 176, 1 (Jan 2007), 498-520.

[18] B. Gavish, D. Horsky, and K. Srikanth, An Approach to the Optimal Positioning of a New Product, *Management Science*, 29, 11 (Nov 1983), 1277-1297.

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

[20] R. Kohli and R. Krishnamurti, Optimal product design using conjoint analysis: Computational complexity and algorithms, *European Journal of Operational Research*, 40.2, 1989.

[21] R. Kohli, R. Krishnamurti, and P. Mirchandani, The Minimum Satisfiability Problem, *SIAM J. Discrete Math*, 1994.

[22] C. Li, B. C. Ooi, A. K. H. Tung, and H. Wang, DADA: a Data Cube for Dominant Relationship Analysis, *SIGMOD* 2006.

[23] C. Li, A. K. H. Tung, W. Jin, and M. Ester, On Dominating Your Neighborhood Profitably, *VLDB 2007*, 818-829.

[24] M. Miah, G. Das, V. Hristidis, and H. Mannila, Standing Out in a Crowd: Selecting Attributes for Maximum Visibility, *ICDE 2008*: 356-365.

[25] J. K. Kim and J. W. Chang, A new parallel signature file method for efficient information retrieval. In *Proceedings of the 1995 International Conference on Information and Knowledge Management (CIKM '95)*, Baltimore, USA.

[26] E. Tousidou, A. Nanopoulos, and Y. Manolopoulos, Signature-Based Structures for Objects with Set-Values Attributes, *Information Systems*, vol. 27, no. 2, pp. 93-121, 2002.

[27] 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.

[28] 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.

[29] Z. Lin and C. Faloutsos. Frame-sliced signature files. IEEE Transactions on Knowledge and Data Engineering, 4(3):281{289, 1992.

[30] D. L. Lee and C. W. Leng, Partitioned signature files: Design issues and performance evaluation, *ACM Transaction on Information Systems*, 7:158-180, 1989.