Chapter 5
Print ISBN: 978-93-5547-821-4, eBook ISBN: 978-93-5547-822-1

Most Popular Package Design and NP-Hard Problem

Yangjun Chen [a*] and Bobin Chen [a]

DOI:10.9734/bpi/nramcs/v8/3788E

ABSTRACT

Given a set of items, and a set of user preferences, we investigate the problem of designing a most popular package (or say, a pattern), i.e., a subset of items that maximizes the number of satisfied users. 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 graph structure, called a p^*-graph, used to represent the preference of a user, by which a lot of useless checks can be avoided. The time complexity of the algorithm is bounded by $O(n^2m^3)$, where m is the number of items (or say, attributes) and n is the number of user preferences. Since the problem is essentially NP-hard, the algorithm discussed in this chapter in fact provides a proof of $P = NP$.
CCS Concepts: • Theory of computation → Minimum satisfiability problem.

Keywords: Data mining; single package design; trie; NP-hard; time complexity analysis; MINSAT.

1 Introduction

Data mining, also known as knowledge discovery in data (KDD), is the process of uncovering patterns and other valuable information from large data sets [1-3]. As one of its important problems, the frequent pattern mining [4,5] is to recognize a frequent pattern, in terms of a given set of transactions with each consisting of some items. By a frequent pattern we mean a subset of items which are supported (or say, contained) by most of transactions. In this chapter, we discuss a more challenging problem, the so-called single package design problem (SPD for short [6,7,8,9]), by which we consider a set of activities or items $A = \{a_1, \ldots, a_m\}$, 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' (i.e., a_j can be selected or not). Then, a bit tuple t (or say a bit string with each bit corresponding to an activity) is referred to as a package (or say a truth assignment); and what we want is to ensure that such a package satisfies as many queries as possible. If a package is with this property, it is called 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.

Table 1. A query log Q

queryID	Hot spring	Ride	Glacier	Hiking	Airline	Boating
q_1	1	*	0	*	1	*
q_2	1	0	1	*	*	*
q_3	*	0	0	1	1	*
q_4	*	*	1	*	1	*
q_5	*	0	0	*	*	0
q_6	*	1	*	0	*	1

The design of a most popular package is to pick up a sub-set of these activities to meet as many queries' requiremets as possible.

This problem has been investigated by several researchers [10,9] . In [9], an approximation algorithm was discussed, by which an SPD problem is reduced to a MINSAT problem [11] that is an optimization version of the satisfiability [9], by which we seek to find a truth assignment to minimize the number of satisfied clauses. The method discussed in [10] is in fact based on the construction of a kind of binary trees, called signature trees [12-14] for signature files [15,16,17] . Its worst-case time complexity is bounded by $O(mn2^m)$, where m is the number of items (or say, attributes) and n is the number of queries.

Our method works quite differently, but based on a compact representation of all those truth assignments for each query q, under which q evaluates to true. Organizing all such data structures for all the queries into a trie-like graph G, an efficient algorithm can be designed based on a bottom-up search of G. The time complexity of the algorithm is bounded by $O(n^2m^3)$. As shown in the Appendix, SPD is NP-hard [18,19,20]. Thus, our algorithm is in fact a proof of $P = NP$ [21,22].

The remainder of the chapter is organized as follows. In Section 2, we show a simple example of the SPD problem. Then, in Section 3, the algorithms for evaluating the SPD is discussed in great detail. Section 4 is devoted to the time analysis. Finally, we conclude with a summary in Section 5.

2 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 with $m = 6$ attributes (activities), and each query represents one of user's favourites. For instance, the query $q_1 = c_{11}c_{12} \dots c_{16} = (1,^*, 0,^*, 1,^*)$ 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.
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.

3 Algorithm Description

In this section, we discuss our algorithm. First, we present the main idea of our algorithm in Section 3.1. Then, in Section 3.2, the algorithm is descussed in great detail. Next, we discuss how to improve the algorithm in Section 3.3.

3.1 Main idea

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

For instance, for $q_1 = (1,^*, 0,^*, 1,^*)$ in Table 1 , a sequence:

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

will be generated. Next, we need to compute the frequency of each attribute appearrance in all such sequences in Q, by which $(a, *)$ is counted as an appearance of r. Then, using $F(a)$ to represent the frequency of any attrubute a, we will have $F(\text{hot-spring}) = 5/6, F(\text{ride}) = 3/6, F(\text{glacier}) = 3/6$, $F(\text{hiking}) = 5/6, F(\text{airline}) = 6/6, F(\text{boating}) = 5/6$ for Tabel 1.

In terms of the attribute appearance frequencies, we will impose a global ordering over all attributes such that the most frequent attribute appears first, but with ties broken arbitrarily. For instance, for Q shown in Tabel 1, we can specify a global ordering like this: airline → hot spring → hiking → boating → boating → ride → glacier. In fact, any ordering of attributes works well, based on which a graph representation of true assignments can be established. However, ordering attributes according to their appearance frequencies can greatly improve the efficiency when searching the trie (to be defined in the next subsection) constructed over all the attribute sequences in a query log.

Following this general ordering, each query in Table 1 can be represented as a sorted attribute sequense as demonstrated in Table 2 (see the third column).

In Table 2, each sorted attribute sequence (for a query) is augmented with a start symbol # and an end symbol $ for technical convenience.

For our algorithm, we need to introduce a graph structure to represent all those truth assignments for each attribute sequence (for a q), called a p^*-graph, under which q evaluates to true. For this purpose, we first discuss a simpler concept for ease of explanation.

In the following, by an attribute sequence, we always mean a sorted attribute sequence. We will also use word 'query' and its attribute sequence interchangeably.

Definition 3.1. (*p*-graph) Let $q = d_0 d_1 \dots d_k d_{k+1}$ be an attribute sequence representing a query as described above (with $d_0 = \#$ and $d_{k+1} = \$$). A *p*-graph over q is a directed graph, in which there is a node for each $d_j (j = 0, \dots, k+1)$; and an edge for (d_j, d_{j+1}) for each $j \in \{1, \dots, k\}$. In addition, there may be an edge from d_j to d_{j+2} for each $j \in \{1, \dots, k-1\}$ if d_{j+1} is a pair of the form (*a*, *), where *a* is an attribute.

Table 2 . Queried 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	(Hs, *).(R, *).G.(H, *).A.(B, *).	#.A.(Hs, *).(H, *).(B, *).(R, *).G.$
q_5	(Hs, *).(H, *).(A, *).	#.(A, *) (Hs, *).(H, *).$
q_6	(Hs, *).R.(G, *).B.	#.(Hs, *).B.R.(G, *).$

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

In Fig. 1(a) we show such a p-graph for q_1 = #.A.Hs. $(\mathrm{H},^*).(\mathrm{B},^*).(\mathrm{R},^*).\$$. Beside a main path going through all the items in q_1, there are three off-path edges (edges not on the main path), referred to as spans, corresponding to $(\mathrm{H},^*), (\mathrm{B},^*)$, and $(\mathrm{R},^*)$, respectively. Each span is represented by the sub-path covered by it. For example, we will use the sub-path $< v_2, v_3, v_4 >$ to stand for the span connecting v_2 and v_4; $< v_3, v_4, v_5 >$ for the span connecting v_3 and v_5; and $< v_4, v_5, v_6 >$ for the span connecting v_4 and v_6. By using spans, the meaning of '*'s (it is either 0 or 1) is appropriately represented since along a span we can bypass the corresponding attribute (then it is not selected) while along an edge on the main path we go through the corresponging attribute (then it is selected).

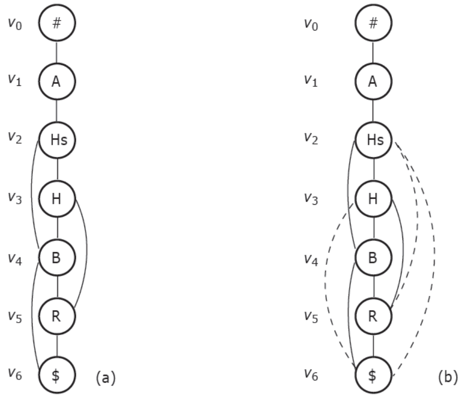

Fig. 1. A *p*-path and a *p* ∗-path

In fact, what we want is to represent all those truth assignments for q in an efficient way, under which q evaluates to true. However, p-graph fails to do so since when we go through from a node v to another node u through a span, u must be selected. If u represents a $(c,^*)$ for some attribute name c, the meaning of this '*' is not properly rendered.

For this reason, the concept of p^*-graph is introduced.

Let $s_1 = \langle v_1, \ldots, v_k \rangle$ and $s_2 = \langle u_1, \ldots, u_l \rangle$ be two spans attached on a same path. We say, s_1 and s_2 are overlapped, if $u_1 = v_j$ for some $j \in \{v_1, \ldots, v_{k-1}\}$, or if $v_1 = u_{j'}$ for some $j' \in \{u_1, \ldots, u_{l-1}\}$. For example, in Fig. 1(a), $< v_2, v_3, v_4 >$ and $< v_3, v_4, v_5 >$ are overlapped. $< v_3 v_4, v_5 >$ and $< v_4, v_5, v_6 >$ are also overlapped. But $< v_2, v_3, v_4 >$ and $< v_4, v_5, v_6 >$ not. Here, we notice that the overlapped spans imply the consecutive 'don't cares', just like $\langle v_2, v_3, v_4 \rangle$ and $\langle v_3, v_4, v_5 \rangle$, which correspond to two consecutive '*'s: $(\text{H},^*)$ and $(\text{B},^*)$. Therefore, the overlapped spans exhibite some kind of transitivity. That is, if s_1 and s_2 are two overlapped spans, the $s_1 \cup s_2$ must be a new, but bigger span. Applying this operation to all the spans over a p-path, we will get a 'transitive closure' of overlapped spans. Based on this observation, we give the following definition.

DEfinition 3.2. (p^*-graph) Let P be a p-graph. Let p be its main path and S be the set of all spans over p. Denote by S^* the 'transitive closure' of S. Then, the p^*-graph with respect to P is the union of p and S^*, denoted as $P^* = p \cup S^*$.

In Fig. 1(b), we show the p^*-graph with respect to the p-graph shown in Fig. 1(a). Concerning p^*-graphs, we have the following lemma.

LEMMA 1. Let P^* be a p*-graph for an attribute sequence (of some query q in Q). Then, each path from # to $ in P^* represents a truth assignment, under which q evaluate to true.

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

(2) Each path from # to $ represents a truth assignment, under which q evaluates to true. 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.

3.2 Algorithm

To find a truth assignment to maximize the number of satisfied queries in Q, we will first construct a trie-like graph G over Q, and then search G bottom-up to find the answer.

Let ${P_1}^*, {P_2}^*, \ldots, {P_n}^*$ be all the p^*-graphs constructed for all the queries $q_1, q_2, \ldots, q_n$ in Q, respectively. Let p_j and $S_j^*(j = 1, \ldots, n)$ be the main path of P_j^* and the transitive closure over its spans, respectively. We will construct G in two phases. In the first phase, we will establish a trie *T*, denoted as $T = trie(R)$ over $R = \{p_1, \ldots, p_n\}$ 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: $trie(R_1)$, $trie(R_2)$, ..., $trie(R_m)$ are constructed in the same way except that at the k th step, the splitting of sets is based on the k-attribute (along the global ordering of atttributes). They are then connected from their respective roots to a single node to create $trie(R)$.

In Fig. 2(a), we show the trie constructed for the sorted attribute sequences shown in 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) in Q. Each edge in the trie is referred to as a *tree edge*.

In the second phase, we will add all ${S_i}^*(i = 1, \ldots, n)$ to the trie T to construct a trie-like graph G, as illustrated in Fig. 2(b), in which we show a trie-like graph that is constructed for all the queries given in Table 1. 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: 2,3,4, indicating that this span belongs to queries: q_2, q_3 and q_4. But no numbers are associated with any tree edges.

We will search G bottom up. First, for each leaf node, we will find all its parents. Then, all such parent nodes will be categorized into different groups such that the nodes in the same group will have the same label (attribute name), which enables us to recognizes all those queries which can be satisfied by a same assignment efficiently. All the groups containing only a single node will not be further explored. (That is, if a group contains only one node *v*, the parent of *v* will not be checked.) Next, all the nodes with more than one node will be explored. We repeat this process until we reach a level at which each group contains only one node. In this way, we will find a set of subgraphs, each rooted at a certain node *v*, in which the nodes at the same level must be labeled with the same attribute name. Then, the path in the trie from the root to v and any path from v to a leaf node in the subgraph correspond to an assignment satisfying all the queries labeling a leaf node in it.

See Fig. 3 for illustration.

In Fig. 3, we show the whole bottom-up searching process of the trie-like graph shown in Fig. 2(b).

- step 1: The leaf nodes of the graph are $v_7, v_8, v_{10}, v_{11}, v_{12}, v_{17}$ (see level 1), representing 6 queries in Q shown in Table 1, respectively. Their parents are $v_1, v_2, v_3, v_4, v_5, v_6, v_9, v_{15}, v_{16}$ (see level 2). Among them, v_6, v_9, v_{16} are all labeled with the same attribute ‘G’ and will be put in a group g_1 while v_5 and v_{15} are both labeled with ‘R’ and put in another group g_2. All the other nodes each are differently labeled and therefore will not be further explored.

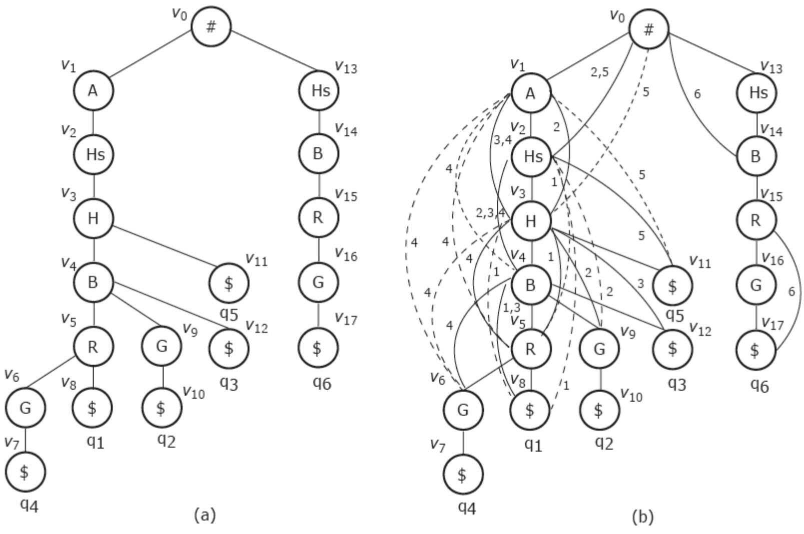

Fig. 2. A trie *T* and a trie-like graph *G*

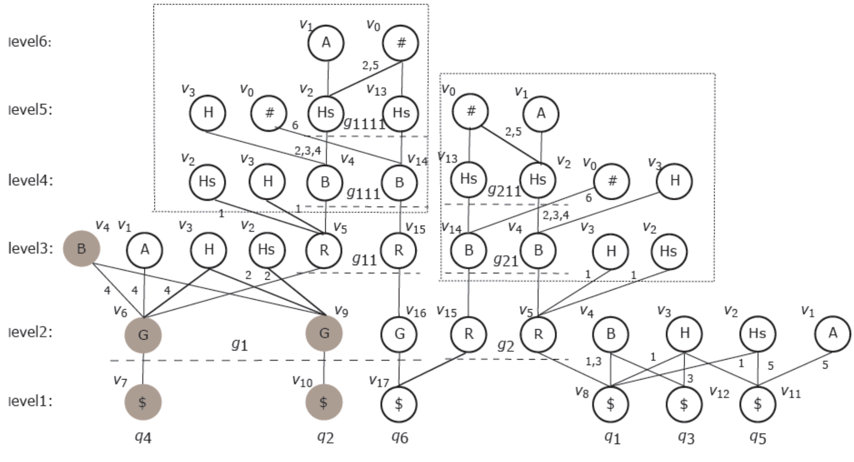

Fig. 3. Illustration for bottom-up search of *G*

- step 2: The parents of the nodes in both g_1 and g_2 will be explored. For g_1, they are v_1, v_2, v_3, v_4, v_5 and v_{15} (see level 3). Among them, both v_5 and v_{15} are labeled with 'R', and then put in a group g_{11}. All the other nodes are differently labeled and will not be further searched. In the same way, the parents of the nodes in g_2 are v_2, v_3, v_4, v_{14}, but only v_4, v_{14} labeled with same 'B' and will be put in a second group g_{21}.

- step 3: The parents of the nodes in g_{11} and g_{21} will be further explored. The parents of the nodes in g_{11} are v_2, v_3, v_4 and v_{14} (see level 4) and v_4, v_{14} have the same label 'B'. Thus, they will be put in group g_{111}. Among the parents of the nodes in g_{21}, v_2 and v_{13} are with the same 'Hs' and will put in a group g_{211}.
- step 4: We continually explore the parents of the nodes in g_{111} and g_{211}. The parents of the nodes in g_{111} are v_0, v_2, v_3 and v_{13} (see level 5). Since v_2 and v_{13} are withe the same label 'Hs', they will be further explored. But all the parents of the nodes in g_{211} are differently labeled and will not be searched.
- step 5: In this step, we will access the parents of v_2 and v_{13}. They are v_1 and v_0 (see level 6), differently labeled. The whole process terminates.

We call the graph illustrated in Fig. 3 a layered representation G' of G. From this, a maximum subset of queries satisfying a certain truth assignment (a subset of attributes) can be efficiently calculated. As mentioned above, each node which is the unique node in a group will have no parents. We refer to such a node as a *s*-root, and the subgraph made up of all nodes reachable from the *s*-root as a rooted subgraph. For example, the subgraph made up of the grey-marked nodes in Fig. 3 is one of such subgraphs.

Concerning rooted subgraphs, we have the following lemma.

LEMMA 2. Let G be a trie-like graph constructed for query log Q and G' its layered representation. Let G_v be a rooted subgraph in G', rooted at v. Then, the labels on each root-to-leaf path in G_v are exactly the same.

Proof. To prove the lemma, we need to show that the labels of the nodes at a same level must be the same. But this can be seen from the construction of G'.

For instance, in the rooted subgraph mentioned above (rooted at v_4, marked grey at level 3 in Fig. 3), we have two paths: $v_4 \rightarrow v_6 \rightarrow v_7$, and $v_4 \rightarrow v_9 \rightarrow v_{10}$. Both are with the same attribute sequence: B.G.$. Here, speciall attention should be paid to the edge $v_4 \rightarrow v_6$, which is associated with a number 4 , indicating that this edge is in fact a span belonging to q_4 (representing (R,*)). Then, the attributes $\{A, Hs, H\}$ represented by the path from v_0 to v_4 in the trie (shown in Fig. 2(a)) plus $\{B, G\}$ form a package satisfying q_2 and q_4. Now we pay attention to another node v_3 at level 2 in Fig. 3 and the corresponding rooted graph, which contains three edges: $v_3 \rightarrow v_8$ (labeled with 1, indicating that it is a span belonging to $q_1, v_3 \rightarrow v_{12}$ (labeled with 3, indicating that it is a span belonging to q_3), and $v_3 \rightarrow v_{11}$ (not labeled, indicating that it is a tree edge in the trie shown in Fig. 2(a)). The attribue subset $\{H\}$ represented by any path in this rooted graph plus the attribute subset $\{A, Hs\}$ represented by the path from the root to v_3 in the trie (shown in Fig. 2(a)) form a package $\{A, Hs, H\}$ satisfying q_1, q_3, q_5. Since this is a maximum subset of queries which can be satifies by a package, $\{A, Hs, H\}$ is a most popular package.

The general rule to determine the subset Q' of queries satisfied by a subset of attributes (or say, a package) for a rooted subgraph G_v is as follow:

- the subset of attribute is: {attributes represented by any path in G_v} ∪ { attributes represented by a path from root to v in the corresponding trie-like graph}
- For any $q_i \in Q'$, there is a path p from the root of G_v to a leaf node representing q_i with one of two conditions satisfied: no edge on p is associted with numbers; or if some edges on p are with numbers, then the set of numbers associated with any such edge on p contains i. (We call this condition the assignment condition.)

In terms of the above discussion, we give the following algorithm. In the algorithm, a queue S is used to explore the layered graph of G. In S, each entry is a subset of nodes labeled with a same attribute name.

Algorithm 1: *SEARCH*(*G*)

```
   Input  : a trie-like graph G.
   Output : a most popular package.
1  G' :={all leaf nodes of G}; g := {all leaf nodes of G};
2  enqueue(S, g);                    (* find the layered graph G' of G *)
3  while S is not empty do
4  |  g := dequeue(S);
5  |  find the parents of each node in g; add them to G';
6  |  divide all such parent nodes into several groups: g_1, g_2, ..., g_k such all the nodes in a
   |    group with the same label;
7  |  for each j ∈ {1, ..., k} do
8  |  |  if |g_j| > 1 then
9  |  |  |_ enqueue(S, g_j);
   |  |_
   |_
10 return findPackage(G');
```

The algorithm can be divided into two parts. In the first part (lines 2 - 12), we will find the layered representation G' of G. In the second part (line 13), we call subprocedure *findPackage*( ), by which we check all the rooted subgraphs to find a package such that the number of satisfied queries is maximized. This is represented by a triplet (u, s, f), corresponding to a rooted subgraph G_u in G. Then, the attributes represented by a path from the root of the trie-like graph to u and the attributes represented by any path in G_u make up a package that satisfies a maximal subset of queries stored in f, whose size is s.

Concerning the correctness of the algorithm, we have the following proposition.

Proposition 1. Let Q be a query log. Let G be a trie-like graph created for Q. Then, the result produced by *SEARCH*(*G*) must be a packages satisfying a maximum subset of queries.

Proof. By the execution of *SEARCH*(*G*), we will first generate the layered reprsentation G' of G. Then, all the rooted subgraphs in G' will be checked. By each of them, we will find a package satisfying a subset of queries, which will be compared with the currently found largest subset of queries. Only the larger between them is kept. Therefore, the result produced by *SEARCH*(*G*) must be correct.

3.3 Improvements

The construction of the layered representation G' of G can be slightly improved by removing any possible redundancy. For example, in Fig. 3, nodes v_5 and v_{15} at level 3 are completely identical to the nodes v_5 and v_{15} at level 2 . Then, the nodes enclosed by the left square are respectively identical to the nodes enclosed by the right square. Thus, the parents of v_5 and v_{15} at level 2 needn't be generated. Instead, we will connect the parents of v_5 and v_{15} at level 3 to v_5 and v_{15} at level 2. That is, connect v_2, v_3, and v_4 at level 4 to v_5 at level 2, and v_{14} at level 4 to v_{15} at level 2, to keep information not lost (see Fig. 4 for illustration).

In this way, a rooted subgraph may contain multiple subsets of queries, each of which can be satisfied by a different package. To see this, pay attention to the subgraph rooted at v_0 at level 6 in Fig. 4.

Algorithm 2: *findPackage*(G')

Input : the layered representation G' of G.
Output : a most popular package.
$(u, s, f) := (null, 0, \Phi)$; (* find a package for a maximum subset of queries. *)
for *each rooted subgraph* G_v **do**
  determine the subset Q' of satisfied queries in G_v;
  if $|Q'| > s$ **then**
    $u := v$; $s := |Q'|$; $f := Q'$;
return (u, s, f);

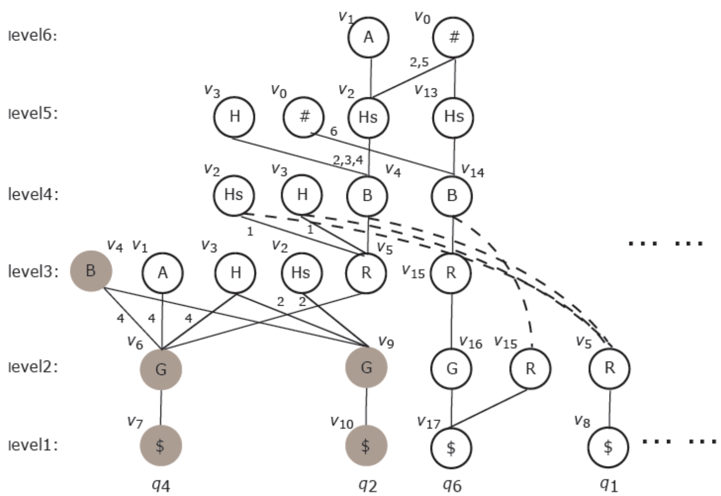

Fig. 4. Illustration for improved G'

Not as with Fig. 3, it now contains two rooted subgraphs as shown in Fig. 5. We call a subgraph like this an extended rooted subgraph.

In Fig. 5(a) and (b) we show the two rooted subgraphs, respectively. We notice that in Fig. 5(a) the path from v_0 to v_7 does not represent a truth assignment under which q_4 evaulates to true. It is because the the set of numbers {2,5} associated with edge $v_0 \rightarrow v_2$ on the path does not contain 4 . But another path from v_0 to v_{17} is a truth assignment for q_6 because each edge on the path is a tree edge. A similar analysis applies to Fig. 5(b).

To distinguish among all the subsets of queries satisfied by different truth assignments represented by an extended rooted sub-graph G_v, we will use a hash function h to create a value for each path p from the root v of G_v to a leaf node representing a certain q_i, which satisfies the assignment condition. Then, all those q_i's with the same $h(p)$ will make up a subset of queries satisfied by a same truth assignment. Let p_1 be the path from v_0 to v_{17} in Fig. 5(a). Let p_2 be the path from v_0 to v_8 in Fig. 5(b). We certainly have $h(p_1) \neq h(p_2)$. In this way, different subsets of queries satisfied by different truth assignments can be differentiated from each other.

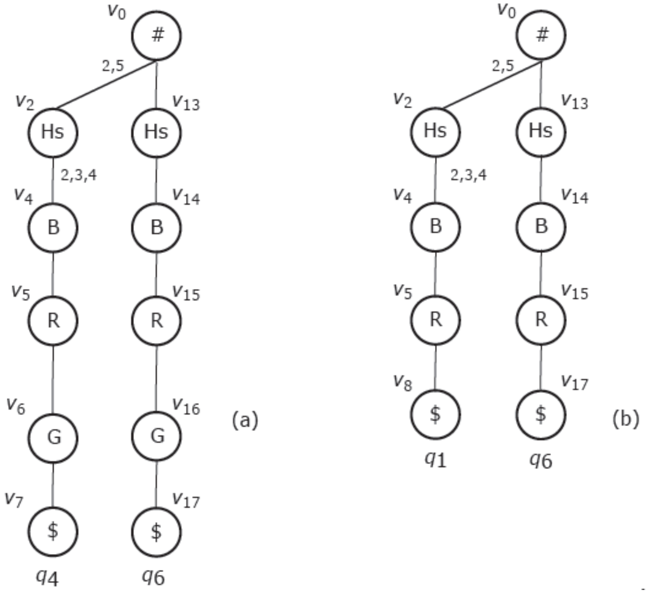

Fig. 5. Illustration for improved G'

4 Time Complexity Analysis

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

The first part τ_1 is the time for computing the frenquencies of attribute appearances in Q. Since in this process each attribute in a q_i is accessed only once, $\tau_1 = O(nm)$.

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

The third part τ_3 is the time for searching G to generate its layered representaion. Since in this process, each edge in G is accessed once and the number of all edges is bounded by $O(nm^2)$, we have $\tau_3 = O(nm^2)$.

The fourth part τ_4 is the time for checking all the extended rooted subgraphs. Since each level in the layered representation G' of G has at most $O(nm)$ nodes, we have $O(nm^2)$ nodes in G' in total. In addition, the number of edges in each extended rooted subgraph is bounded by $O(nm)$, the cost of this part of computation is bounded by $O(n^2m^3)$.

Thus, we have the following proposition

Proposition 2. The total running time of our algorithm is bounded by

$$\sum_{i=1}^{4} \tau_i = O(nm) + (O(nm\log m) + O(nm^2) + O(n^2m^3) = O(n^2m^3).$$

5 Conclusion

In this paper, we have presented a new method to solve the single package design (SPD) problem by representing each query in a query log Q as a compact graph structure, called p^*-graph. Based on this graph structure, all the queries in Q can be organized into a trie-like graph. By searching the trie-like graph bottom-up, a package satisfying a maximum subset of queries of Q can be found efficiently. The time complexity of the algorithm is bounded by $O(n^2m^3)$, where n and m are the number of queries and the number of attributes in Q, respectively. As demonstrated in the Appendix, SPD is in essence a NP-hard problem. Hence, the algorithm discussed in this paper is in fact a proof of $P = NP$.

Competing Interests

Authors have declared that no competing interests exist.

References

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

[2] Resig J, Teredesai A. A Framework for Mining Instant Messaging Services. Retrieved 16 March 2018.

[3] Savasere A, Omiecinski E, Navathe S. An Efficient Algorithm for Mining Association Rules in Large Databases, Proc. of the 21st VLDB Conference Zurich, Swizerland. 1995;432-444.

[4] Available:https://www.jigsawacademy.com/blogs/data-science/frequent-pattern-mining. Retrieved May 2021.

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

[6] Albritton DM, McMullen PR. Optimal product design using a colony of virtual ants, European Journal of Operational Research. 2007;176(1): 498-520.

[7] Gavish B, Horsky D, Srikanth K. An Approach to the Optimal Positioning of a New Product, IManagement Science. 1983;29(11):1277-1297.

[8] Gruca TS, Klemz BR. Optimal new product positioning: A genetic algorithm approach, European Journal of Operational Re-search. 2003;146(3):621-633.

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

[10] Chen Y, Shi W. 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;937-944.

[11] Kohli R, Krishnamurti R, Mirchandani P. The Minimum Satisfiability Problem, SIAM J. Discrete Math. 1994;275-283.

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

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

[14] Chen Y, Chen YB. On the Signature Tree Construction and Analysis, IEEE Transactions on Knowledge and Data Engineering. 2006;18(9):1207-1224.

[15] Grandi F, Tiberio P, Zezula P. Frame-sliced partitioned parallel signature files, In Proc. of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Copenhagen, Denmark. June 1992; 286 – 297.

[16] Kim JK, Chang JW. A new parallel signature file method for efficient information retrieval, In Proc. of the 1995 International Conference on Information and Knowledge Management (CIKM '95), Baltimore, USA.

[17] Lee DL, Kim YM, Patel G. Efficient signature file methods for text retrieval, IEEE Transactions on Knowledge and Data Engineering. 1995;7(3):423-435.

[18] Garey MR, Johson DS. Computers and Intractibility: A Guide to the Theory of NP-Completeness, W. W. Freeman, San Francisco, CA; 1979.

[19] Kesavan V, Kamalakannan R, Sudhakarapandian R, Sivakumar P. Heuristic and meta-heuristic algorithms for solving medium and large scale sized cellular manufacturing system NP-hard problems: A Comprehensive Review, Materials Today: Proceedings. 2020 Jan 1;21:66-72.

[20] da Silveira LA, Soncco-Álvarez JL, de Lima TA, M Ayala-Rincón M. Parallel island model genetic algorithms applied in np-hard problems, 2019 IEEE Congress on Evolutionary Computation (CEC). IEEE. 2019 Jun 10;3262-3269.

[21] Corman TH, Leierson CE, Rivest RL, Stein C. Introduction to Algorithms, McGraw Hill; 2002.

[22] Johnson MS. Approximation Algorithm for Combinatorial Problems, J. Computer System Sci. 1974;9:256-278.

Appendix NP-Hardness of SPD

In this Appendix, we show the NP-hardness of SPD. For this purpose, we view a query log Q as a logic formula in the disjunctive normal form (DNF):

$$D = (c_{11} \wedge \dots \wedge c_{1j_1}) \vee \dots \vee (c_{m1} \wedge \dots \wedge c_{mj_m})$$

For example, the query log given in Table 1 can be represented as a formula in DNF as beow:

$$\begin{aligned} D = \quad & q_1 \vee q_2 \vee q_3 \vee q_4 \vee q_5 \vee q_6 \\ = \quad & (c_1 \wedge \neg c_3 \wedge c_5) \vee \\ & (c_1 \wedge \neg c_2 \wedge c_3) \vee \\ & (\neg c_2 \wedge \neg c_3 \wedge c_4 \wedge \wedge_5) \vee \\ & (c_3 \wedge c_5) \vee \\ & (\neg c_2 \wedge \neg c_3 \wedge \neg c_6) \vee \\ & (c_2 \wedge \neg c_4 \wedge \neg c_5 \wedge c_6) \end{aligned}$$

where c_1 stands for hot spring, c_2 for ride, c_3 for glacier, c_4 for hiking, c_5 for airline, and c_6 for boating.

Then, to find a most popular package is to find a truth assignment that maximizes the number of satisfied conjunctions in D.

Now consider the negation of D :

$$\neg D = (\neg c_{11} \vee \dots \vee \neg c_{1j_1}) \wedge \dots \wedge (\neg c_{m1} \vee \dots \vee \neg c_{mj_m})$$

It is a formula in CNF. To find a truth assignment that maximizes the number of conjunctions in D is equivalent to finding a truth assignment that minimizes the number of clauses in $\neg D$, the so-called MINSAT problem [11], which proves to be NP-hard.

Biography of author(s)

Yangjun Chen
Department of Applied Computer Science, University of Winnpeg, Canada.

Research and Academic Experience: He completed his PhD in Computer Science from the University of Kaiserslautern, Germany, in 1995. He is now a professor in Dept. Applied Computer Science, University of Winnipeg, Canada.

Research Area: His research area is in Algorithm design, and Databases.

Number of Published Papers: He has about 200 publications in Computer Science and Computer engineering.

Bobin Chen
Department of Applied Computer Science, University of Winnpeg, Canada.

Research and Academic Experience: He completed his bachelor degree in Computer Science at University of Toronto, Canada, in 2021. He is now a software engineer.

Research Area: His research area is in Software engineering.

Number of Published Papers: He has published 1 paper.

© Copyright (2022): Author(s). The licensee is the publisher (B P International).