Data Mining

- What is data mining?
- Mining association rules
  - Apriori algorithm
  - Algorithm based FP-trees (frequent pattern trees)
- Finding most popular packages
  - Algorithm based on priority queues

On the Data Mining

- Data mining refers to the mining or discovery of new information in terms of patterns or rules from vast amounts of data.
- Data mining as part of the knowledge discovery process
  - Association rules
  - Classification
  - Sequential patterns

Data mining as part of the knowledge discovery process

- **Association rules**

  For example, whenever a customer buys video equipment, he or she also buy another electronic gadgets.

- **Classification**

  For example, customers may be classified by frequency of visits, types of financing used, amounts of purchases, or affinity for types of items. Some revealing statistics may be generated for such classes.

Data mining as part of the knowledge discovery process

- **Sequential patterns**

  For example, suppose a customer buys a camera, and within three months he or she buys photographic supplies, then within six months he is likely to buy an accessory item. This defines a sequential pattern of transactions. A customer who buys more than twice in lean periods may be likely to buy at least once during the Christmas period.

Goals of Data Mining and Knowledge Discovery

- **Identifying patterns and trends:**

  Data mining enables businesses to identify patterns and trends in large datasets, helping them understand customer behavior, preferences, and market trends.

- **Customer segmentation:**

  By segmenting customers based on their attributes and behaviors, businesses can tailor their marketing strategies, personalize customer experiences, and improve customer satisfaction.

- **Forecasting and prediction:**

  Data mining techniques enable accurate forecasting and prediction, helping businesses make informed decisions, plan for the future, and identify potential risks and opportunities.

Types of Knowledge Discovered

Classification of knowledge

By knowledge creation

- **Deductive knowledge:**
  Deduces new information based on applying pre-specified logic rules of deduction on the given data.
- **Inductive knowledge:**
  discovers new rules and patterns from the supplied data.

By knowledge representation

- Knowledge can be represented in many forms.
- In an unstructured sense, it can be represented as rules or propositional logic.
- In a structured form, it may be represented in decision trees, neural networks, semantic networks, hierarchies of classes.

Types of Knowledge Discovered

- Association rules
- Classification hierarchies
- Inductive knowledge
- Sequential patterns
- Pattern within time series
- Clustering

Association Rules

Market-Basket model, support, and confidence

- Here, the market-basket corresponds to the sets of items a customer buys in a supermarket during one visit.

Transaction_id	Time	Items_bought
101	6:35	milk, bread, cookies, juice
792	7:38	milk, juice
1130	8:05	milk, eggs
1735	8:40	bread, cookies, coffee

Association Rules

Market-Basket model, support, and confidence

- An association rule is of the form $X \Rightarrow Y$, where $X = \{x_1, x_2, ..., x_n\}$, and $Y = \{y_1, y_2, ..., y_m\}$ are sets of items, with x_i and y_j being distinct items for all *i* and *j*. This rule states that if a customer buys *X*, he or she is also likely to buy *y*.

X – LHS (left hand side)
Y – RHS (right hand side)

LHS $\cup$ RHS – itemset

Association Rules

- **The support for a rule LHS ⇒ RHS refers to how frequently a itemset occurs in the database. That is, the support is the percentage of transactions that contain all of the items in the itemset.**

Example. Rule milk ⇒ juice

Support(milk ⇒ juice) = 0.50

Example. Rule bread ⇒ juice

Support(bread ⇒ juice) = 0.25

prevalence of the rule

Association Rules

- The confidence for a rule LHS ⇒ RHS is with respect to the implication shown in the rule, computed as

  confidence(LHS ⇒ RHS)
  = support(LHS ⇒ RHS)/support(LHS)

  Example. Rule milk ⇒ juice

  confidence(milk ⇒ juice) = 0.667 0.5/0.75 = 0.667

  Example. Rule bread ⇒ juice

  confidence(bread ⇒ juice) = 0.50 0.25/0.5 = 0.5

  strength of the rule

Association Rules

- The goal of mining association rules is to generate all possible rules that exceed some minimum user-specified support and confidence threshold.
- The problem is decomposed into two subproblems.

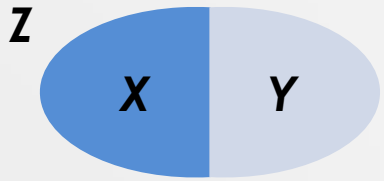
Association Rules

Two subproblems:

1. Generate all itemsets that have a support that exceeds the threshold. These sets of items are called large (or frequent) itemset. Here, 'large' means large support.
2. For each large itemset, all the rules that have a minimum confidence are generated as follows. For a large itemset Z and $Y \subset Z$, let $X = Z - Y$; then if support(Z)/support(X) > minimum confidence, the rule

$$X \Rightarrow Y$$

is a valid rule.



Association Rules

Major problem:

Discovering all large itemsets together with the value for their supports is time-consuming If the cardinality of the set of items is very high.

Assume that the number of items is n. Then, the number of distinct itemsets is 2^n. (In a typical supermarket, we may have thousands of items.)

To reduce the combinatorial search space, two properties are used to reduce the number of sets to be checked:

* Downward closure (of large itemsets)
* Antimonotonicity (of small itemsets)

Association Rules

* Downward closure

  A subset of a large itemset (items in it with large support) must be large

  (that is, each subset of a large itemset exceeds the minimum support.)

* Antimonotonicity

  Conversely, a superset of a small itemset (items in it with low support) is also small (implying that it does not have enough support)

Apriori Algorithm

Input: Database of *m* transactions, *D*, and a minimum support, *mins*, represented as a fraction of *m*.
Output: Frequent itemsets: L_1, L_2, …, L_k
Beging

1. **Compute support(i_j) = count(i_j)/*m* for each individual item i_1, i_2, …, i_n by scanning the database once and counting the number of transactions that i_j appears in (that is, count(i_j));**
2. **The candidate frequent 1-itemset, C_1, will be the set of items i_1, i_2, …, i_n;**
3. **The subset of items containing i_j from C_1 where support(i_j) ≥ *mins* becomes the frequent 1-itemset, L_1;**
4. ***K* = 1;**
   termination = false;
   repeat
5. **L_{k+1} = Ø;**
6. **Create the candidate frequent (*k* + 1)-itemset, C_{k+1}, by combining members of L_k that have *k – 1* elements in common; (this forms candidate frequent (k+1)-itemsets by selectively extending frequent *k*-itemset by one item);**

Apriori Algorithm

7. In addition, only consider as elements of C_{k+1} those $k + 1$ items such that every subset of size k appears in L_k.
8. Scan the database once and compute the support for each member of C_{k+1}; if the support for a member of $C_{k+1} \geq$ mins then add that member of L_{k+1};
9. If L_{k+1} is empty then termination = true
   else $k := k + 1$;

until termination;

End

Sample Trace of the Apriori Algorithm

Transaction_id	Time	Items_bought
101	6:35	milk, bread, cookies, juice
792	7:38	milk, juice
1130	8:05	milk, eggs
1735	8:40	bread, cookies, coffee

Sample Trace of the Apriori Algorithm

$m = 4$, $n = 6$, minsupport = 0.5,

C_1 = {milk (0.75), bread (0.5), juice (0.5), cookies (0.5), eggs (0.25), coffee (0.25)}

L_1 = {milk (0.75), bread (0.5), juice (0.5), cookies (0.5)}

C_2 = {(m, b) (0.25), (m, j) (0.5), (b, j) (0.25), (m, c) (0.25), (b, c) (0.5), (j, c) (0.25)}

L_2 = {(m, j)(0.5), (b, c) (0.5)}

Note that any pair *(x,* e), *(x,* co) $\notin C_2$ because both e and co are small.

In terms of antimonotonicity, such pairs needn't be checked.

Sample Trace of the Apriori Algorithm

$C_3 = \emptyset$

Note that from L_2 = {(m, j)(0.5), (b, c) (0.5)} we cannot find any triplet with support $\geq$ 0.5.

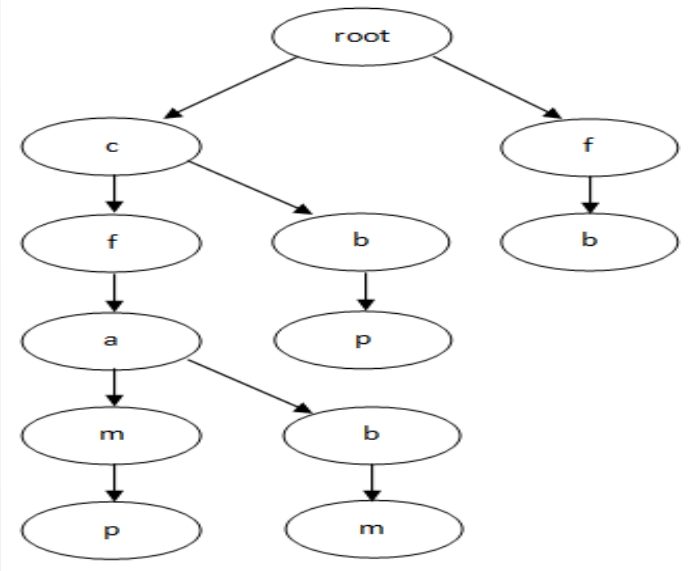
For example, (m, j, b) $\notin C_3$. This can be determined in terms of (j, b) $\notin L_2$ (according to downward closure).

Frequency-Pattern Tree Algorithm

Trie

- A trie is a multiway tree, in which each path corresponds to a string, and common prefixes in strings to common prefix paths.
- Leaf nodes include either the documents themselves, or links to the documents containing the string that corresponds to the path.

Example:



A trie constructed for
The following strings:

s1: cfamp
s2: cbp
s3: cfabm
s4: fb

Frequency-Pattern Tree Algorithm

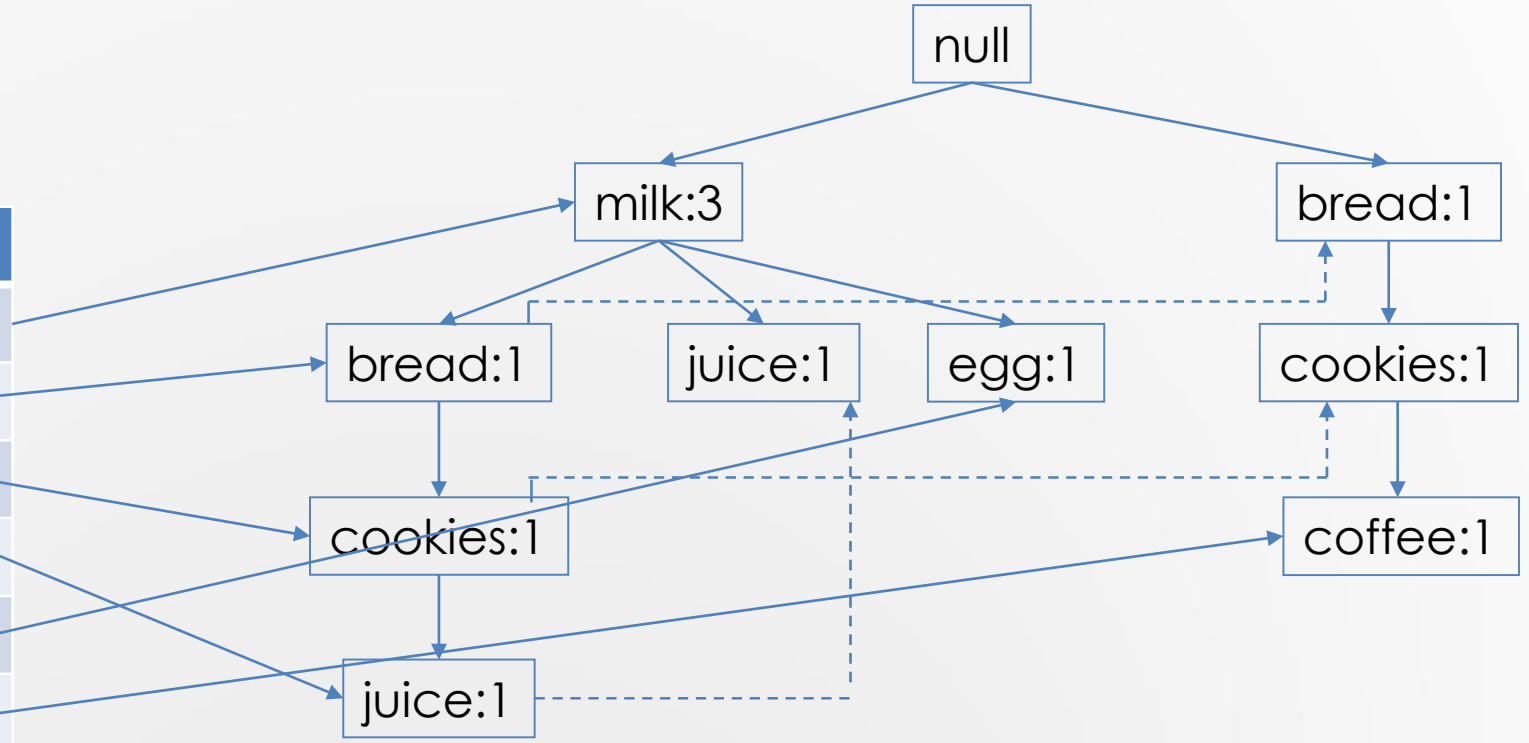
Represent a transaction DB as a trie

- Sort each sequence of items (in a transaction) (decreasingly) by their supports.
- View each sorted item sequence as a string
- Construct a trie over them.

Transaction_id	Items_bought	Sorted sequence
101	milk, bread, cookies, juice	milk, bread, cookies, juice
792	juice, milk	milk, juice
1130	milk, eggs	milk, eggs
1735	bread, coffee, cookies	bread, cookies, coffee

Frequency-Pattern Tree Algorithm

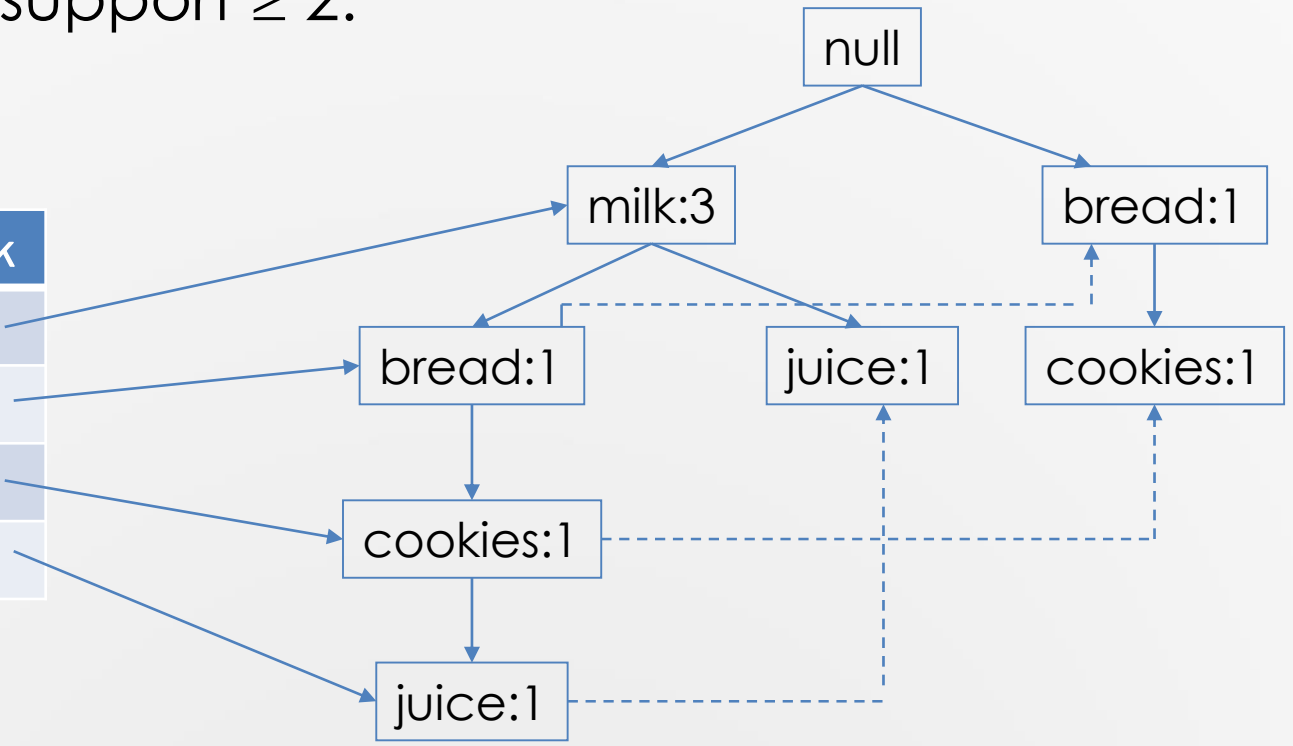
items	support	link
milk	3	
bread	2	
cookies	2	
juice	2	
Egg	1	
coffee	1	



Frequency-Pattern Tree Algorithm

For itemsets with support ≥ 2:

items	support	link
milk	3	
bread	2	
cookies	2	
juice	2	



Frequency-Pattern Tree Construction

- Each sorted sequence of items is represented as a pair: (*head*, *tail*), where *head* stands for the first item, and *tail* for the remaining items.
- Example: {milk, bread, cookies, juice} is represented as {milk, {bread, cookies, juice}}
- A FP-tree can be constructed by inserting transaction sequences one by one.

1. Initially, FP-tree *T* contains only the root *null*.
2. Search *T*. If the current node, *N*, of *T* has a child *N'* with an item name = *head*, then increment the count(*N'*) by 1, else create a new node, *N''*, with count(*N''*) = 1, link *N ''* to its parent and also with the item header table (used for efficient tree traversal).
3. If *tail* is nonempty, then repeat step (2) using *tail* as the sorted list, that is, the old head is removed and the new head is the first item from *tail* and the remaining items become the new tail.

Mining FP-trees for Frequent Patterns

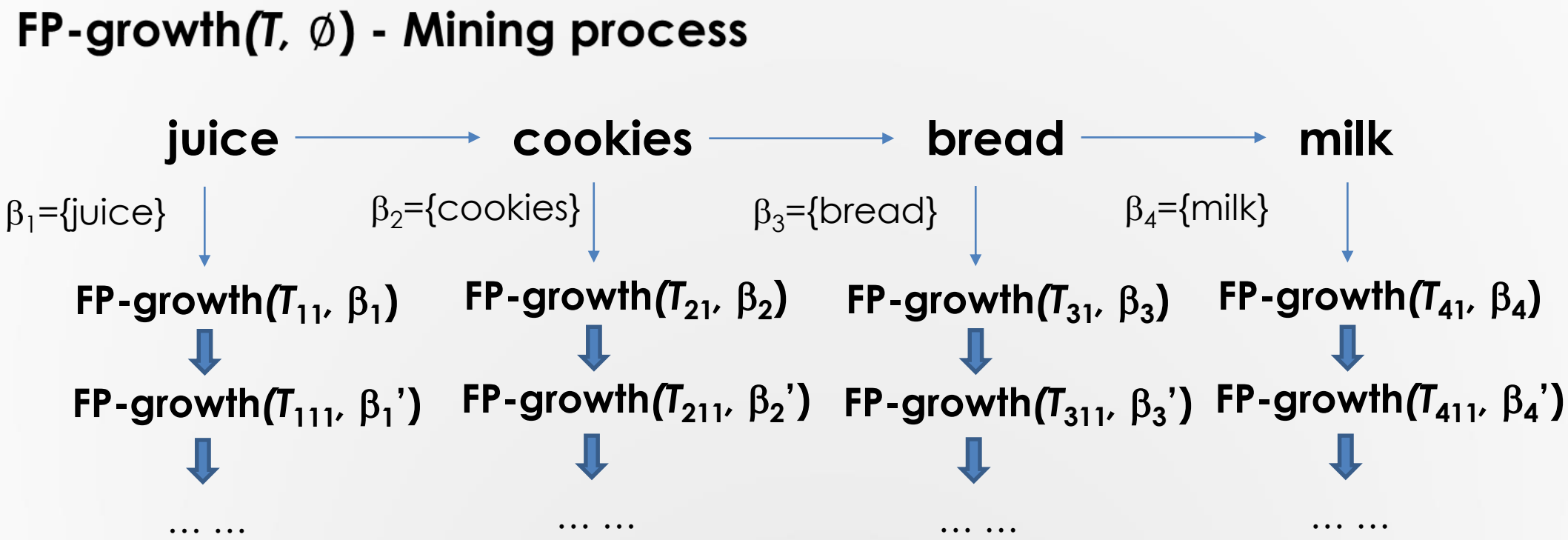
- Finding all frequent patterns with support larger than a certain value

Input: FP-tree and a minimum support – mins
Output: frequent patterns – itemsets
Procedure FP-growth(*tree*, α) (initially, $\alpha = \emptyset$) (*α is a pattern.*)

Begin
    if *tree* is a single path *p* **then**
    for each combination β of the nodes in *p* **do**
        generate pattern $\{\beta \cup \alpha\}$
        with support = minimum support of nodes in β
    else for each item, *i* in the header table, in the order of increasing supports **do**
        generate pattern $\beta = \{i \cup \alpha\}$
        construct β's conditional pattern base
        construct β's conditional FP-tree, β_tree
        if β_tree is not empty **then** FP-growth(β_tree, β)
End

Mining FP-trees for Frequent Patterns

- β's conditional pattern base (β-base) – all the paths found along *i*-links
- β's conditional FP-tree – the trie constructed for all the paths in β-base



Sample Trace of FP-growth Algorithm

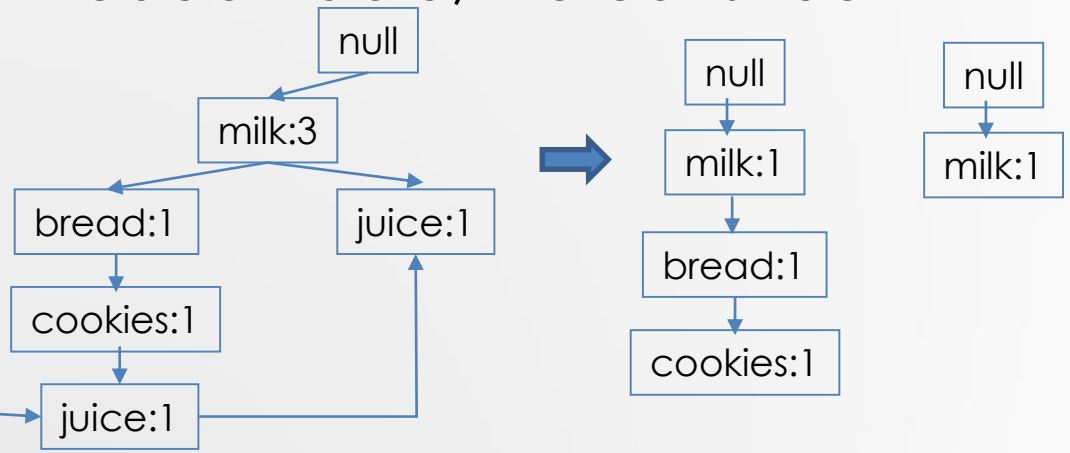
Find itemsets with mins = 2:

- FP-growth(*T*, null)
  - Since the original FP-tree *T* has more than a single path, we execute the **else** part of outer **if** statement.
  - We will examine the frequent items in order of increasing supports (that is, from the last entry in the table to the first).
  - β_1 = {juice:2} with support equal to 2.
  - Following the node link in the term header table, we construct

conditional pattern base:

{(milk, bread, cookies: 1), (milk: 1)}

items	support	link
...	...	
juice	2	



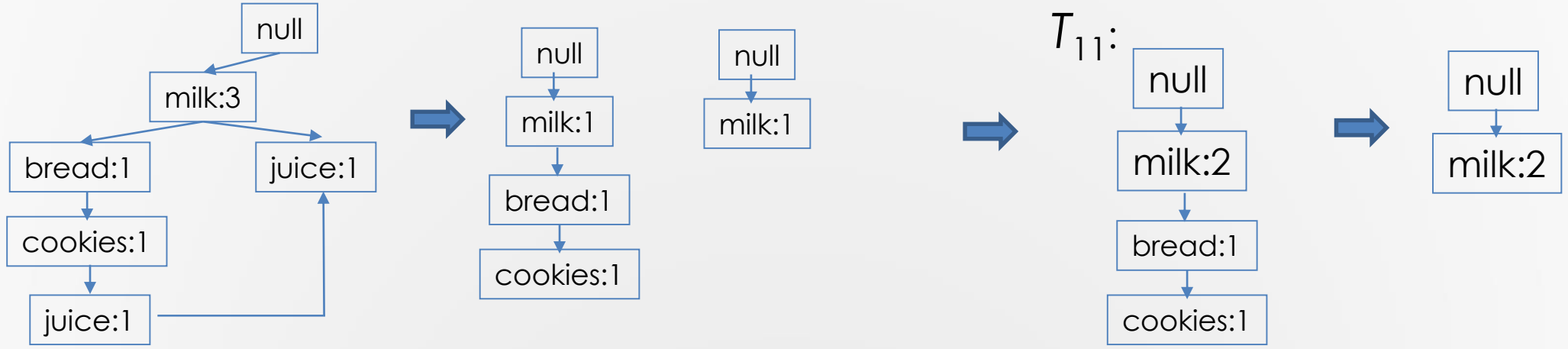
Sample Trace of FP-growth Algorithm

Find itemsets with mins = 2:

- Following the node link in the term header table, we construct conditional pattern base:

  {(milk, bread, cookies: 1), (milk: 1)}

  conditional FP-tree:



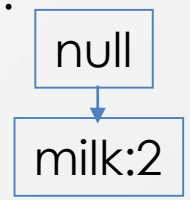
Sample Trace of FP-growth Algorithm

Find itemsets with mins = 2:

- FP-growth(T_{11} , {juice:2}) (*recursive call*)

  Since T_{11} has only one path, all combination β of nodes in the path are generated (that is, {milk, juice:2}) with support equal to 2.

T_{11}:



Sample Trace of FP-growth Algorithm

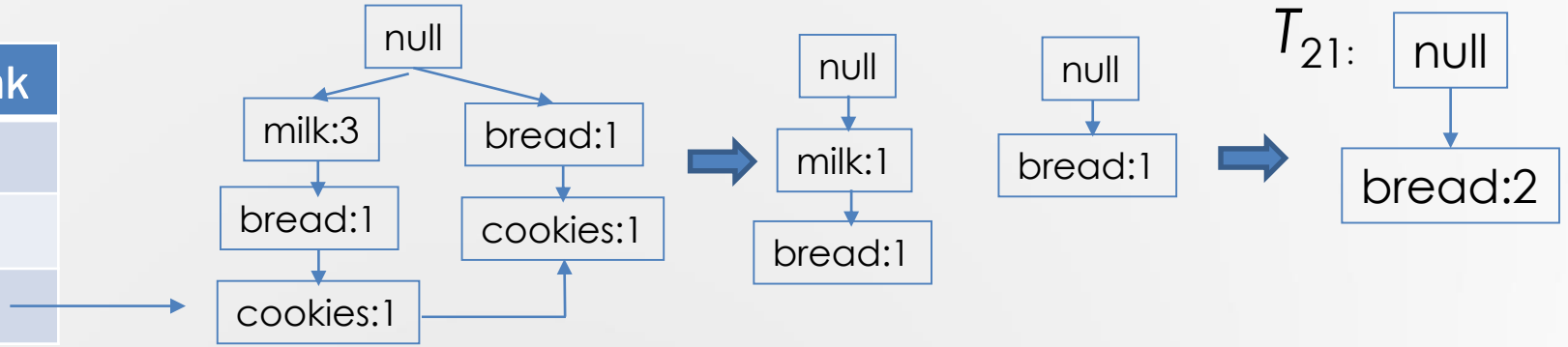
Find itemsets with mins = 2:

- Next, the frequent item, cookies is checked.
  FP-growth(*T*, {cookies:2})

  β_2 = {cookies:2}

  Following the node link in the term header table, we construct
  conditional pattern base = {(milk, bread: 1), (bread: 1)}
  conditional FP-tree:

items	support	link
milk	3	
bread	2	
cookies	2	

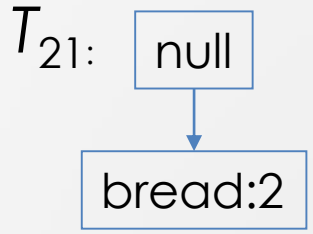


Sample Trace of FP-growth Algorithm

Find itemsets with mins = 2:

- FP-growth(T_{21}, {cookies:2}) (*recursive call*)

  Since T_{21} has only one path, all combination β of nodes in the path are generated (that is, {bread, cookies:2}) with support equal to 2.



Sample Trace of FP-growth Algorithm

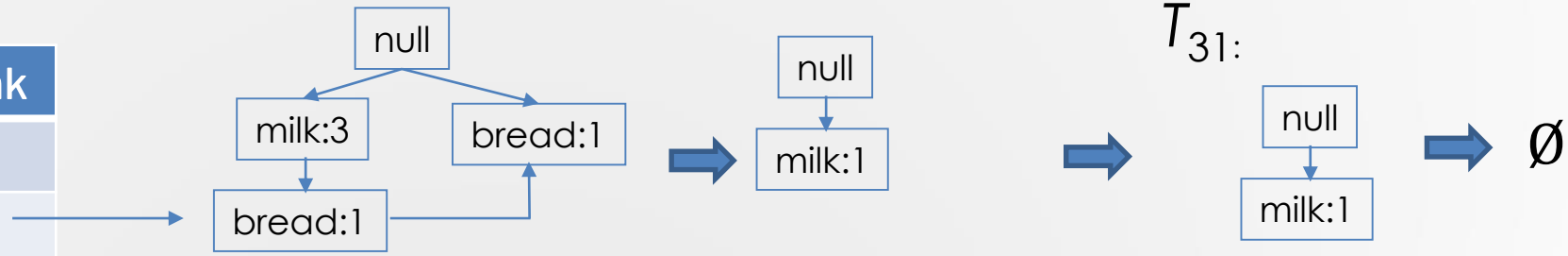
Find itemsets with mins = 2:

- Next, the frequent item, bread is checked.
  FP-growth(*T*, {bread:2})

  β_3 = {bread:2}

  Following the node link in the term header table, we construct
  conditional pattern base = {(milk: 1)}
  conditional FP-tree is empty.
  No frequent patterns will be generated.

items	support	link
milk	3	
bread	2	



Sample Trace of FP-growth Algorithm

Find itemsets with mins = 2:

- Last, the frequent item, milk is checked.
  FP-growth(*T*, {milk:3})

  β_4 = {milk:3}

  Following the node link in the term header table, we construct
  conditional pattern base = empty
  conditional FP-tree = empty.

  No frequent patterns will be generated.

Sample Trace of FP-growth Algorithm

The found frequent patterns:

{milk:3}
{bread:2}
{cookies:2}
{juice:2}
{milk, juice:2}
{bread, cookies:2}

Find the most popular packages

Outline

- Motivation
  - Data mining
  - Most popular packages
- Basic algorithm
- Algorithm based on priority queue searching
  - Priority queue
  - Heuristic for choosing next attribute
- Graph-search based method
- - *p*-graphs and *p**-graphs
- - Trie-like graphs

Motivation

- Given a query log concerning the customers' preference on items or activities, design a package which satisfies as many customers as possible. (a package – a subset of items)

A query log by an travel agency:

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	0	?	1	?	1	?
Q_5	?	0	0	?	?	0
Q_6	?	1	?	0	?	1

Most popular packge:

Hot spring

Hiking

airlines

It satisfies three queries: Q_1, Q_3, Q_5

Basic algorithm based on modified signature tree search

Construction of a modified signature tree:

- 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.

Single Package Design Tree (SPD)

Hot spring

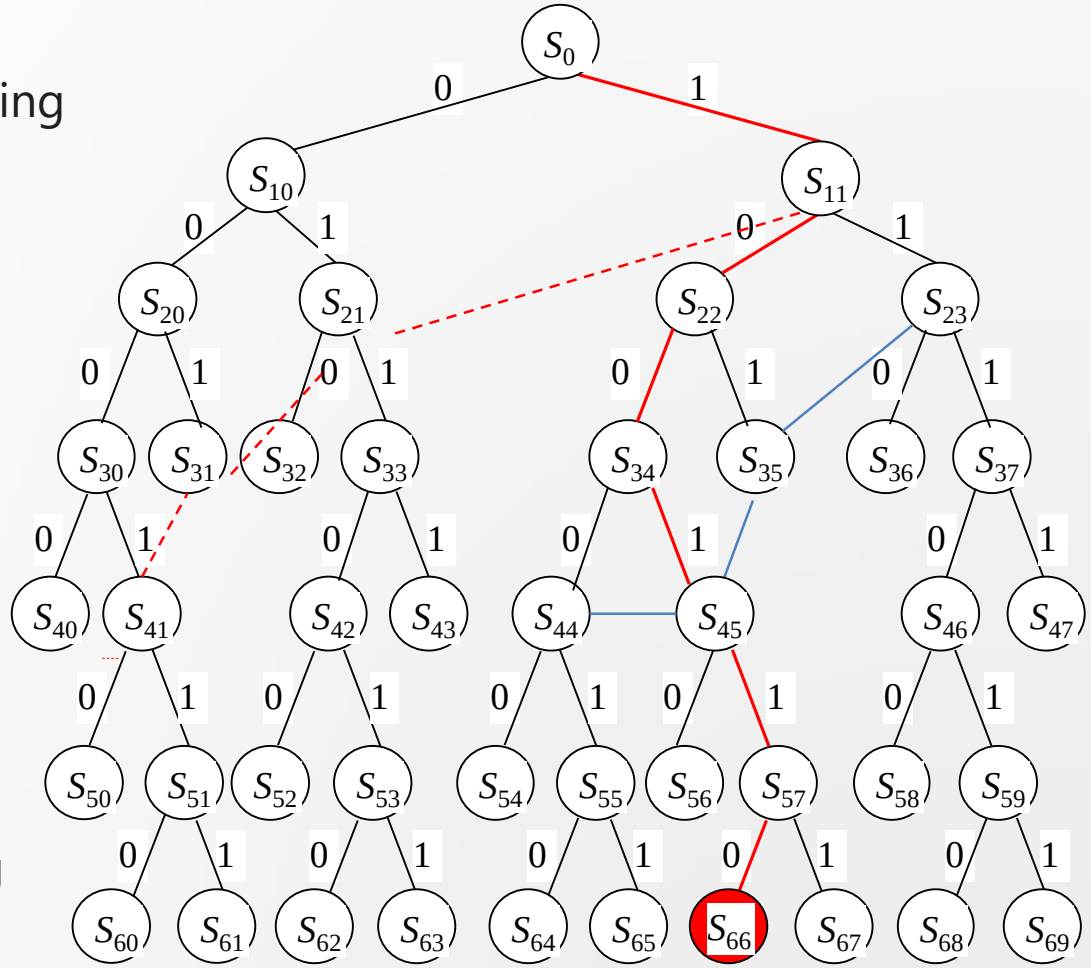
Ride

Glacier

Hiking

Airline

Boating



$S_0 = \{q_1, q_2, q_3, q_4, q_5, q_6\}$ $S_{10} = \{q_3, q_4, q_5, q_6\}$ $S_{11} = \{q_1, q_2, q_3, q_5, q_6\}$
$S_{20} = \{q_3, q_4, q_5\}$ $S_{21} = \{q_4, q_5\}$ $S_{22} = \{q_1, q_2, q_3, q_5\}$ $S_{23} = \{q_1, q_5\}$

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

$S_{40} = \{q_3\}$ $S_{41} = \{q_4, q_5\}$ $S_{42} = \{q_4, q_6\}$ $S_{43} = \{q_4\}$ $S_{44} = \{q_1, q_5\}$
$S_{45} = \{q_1, q_3, q_5\}$ $S_{46} = \{q_1, q_6\}$ $S_{47} = \{q_1\}$
$S_{50} = \{q_3\}$ $S_{51} = \{q_3, q_5\}$ $S_{52} = \{q_6\}$ $S_{53} = \{q_4, q_6\}$ $S_{54} = \{q_5\}$
$S_{55} = \{q_1, q_5\}$ $S_{56} = \{q_5\}$ $S_{57} = \{q_1, q_3, q_5\}$
$S_{58} = \{q_1\}$ $S_{59} = \{q_1, q_6\}$

$S_{60} = \{q_3, q5\}$ $S_{61} = \{q_3\}$ $S_{62} = \{q_4\}$ $S_{63} = \{q_4, q_6\}$ $S_{64} = \{q_1, q_5\}$
$S_{65} = \{q_1\}$ $S_{66} = \{q_1, q_3, q_5\}$ $S_{67} = \{q_1, q_3\}$
$S_{68} = \{q_1\}$ $S_{69} = \{q_1, q_6\}$

Computational complexity of basic algorithm

- Computational complexities of basic algorithm:

  time complexity: $O(mn2^m)$

  space complexity: $O(m^2n^2)$

  where m = number of attributes in Q,

  n = number of queries in Q.

In the worst case, for each level two nodes are kept in the stack. So the space overhead is bounded by $2mn$ since the size of a node is bounded by $O(mn)$.

Priority-first searching algorithm

- A priority queue S is used to control the tree search.
- The key of a node v in S is a pair $(|s(v)|, L)$, where $s(v)$ is the subset of queries represented by v, and L is the level of v.
- a pair $(|s(v)|, L)$ is larger than another pair $(|s(v')|, L')$ iff

  $|s(v)| > |s(v')|$, or

  $|s(v)| = |s(v')|$, but $L > L'$.

Priority-first searching algorithm

- **Two operations over S:**
  - ***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 $S := S \cup \{v\}$.**
- **At each step:**

  $v := extractMax(S)$

  for each child u node of v, **do** $insert(S, u)$

Priority-first searching algorithm

- **Heuristics for choosing a next attribute a (to generate the children of the currently encountered node v):**

  (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.)

  $s(v)[a]$ – all those queries in $s(v)$ with attribute a set to 1

  $s(v)[\neg a]$ - all those queries in $s(v)$ with attribute a set to 0

Algorithm description

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

Input: a query log Q containing n queries with m attributes.
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** ($i \leq n$) **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); $i := L + 1$;

end

Priority-first searching algorithm

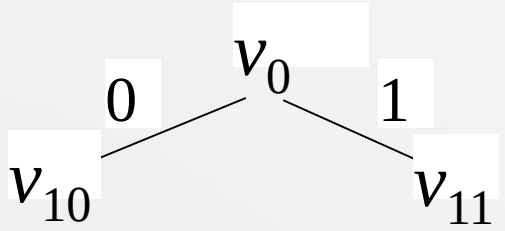
- Example:

Step 1:

S – priority queue:

v_0 (6, 0)

T – search tree:



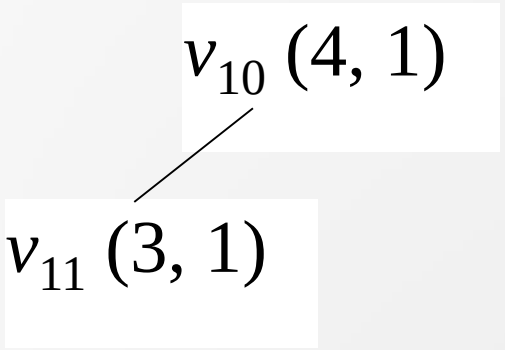
glacier

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

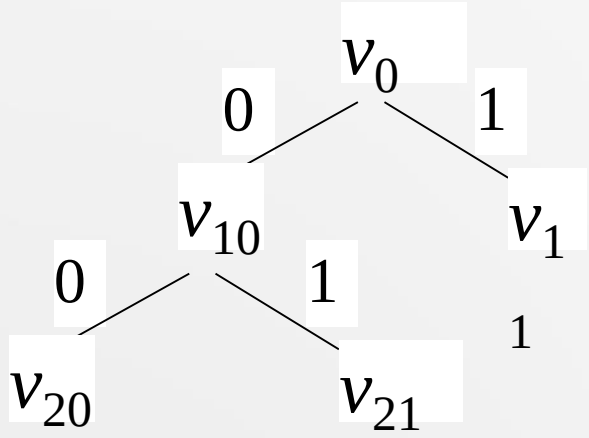
Priority-first searching algorithm

Step 2:

S – priority queue:



T – search tree:



glacier

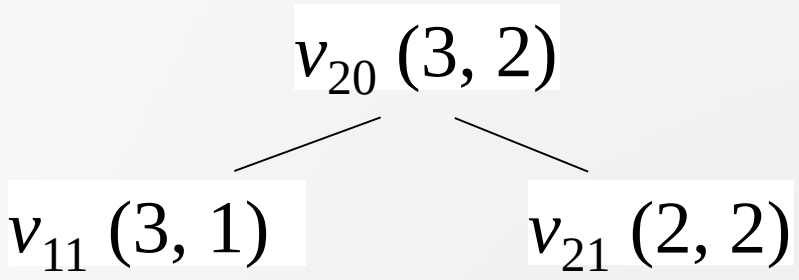
ride

$s(v_{20}) = \{q_1, q_3, q_5\}$ $s(v_{21}) = \{q_1, q_5\}$

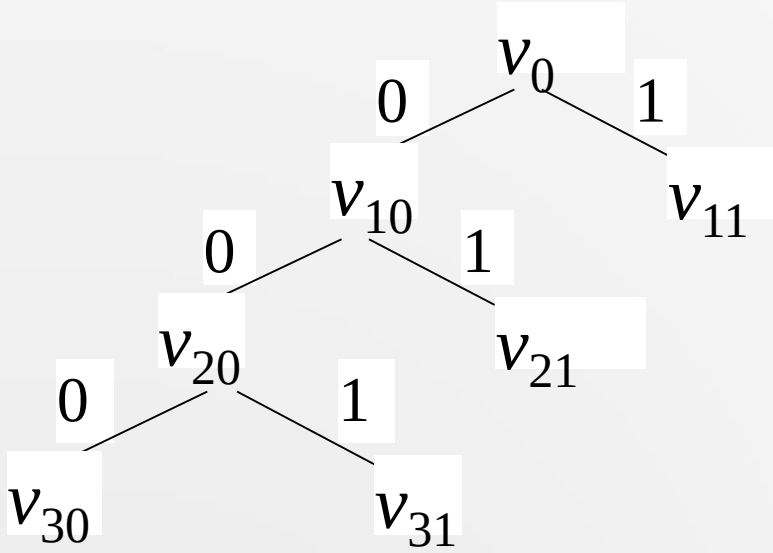
Priority-first searching algorithm

Step 3:

S – priority queue:



T – search tree:



glacier

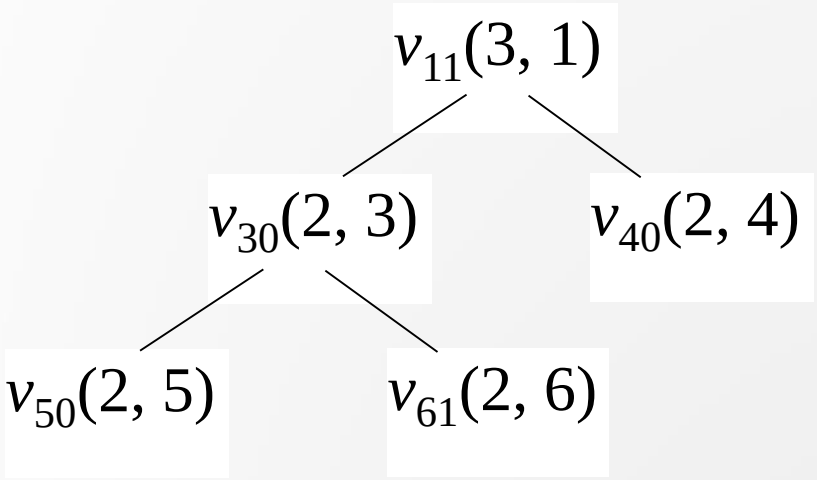
ride

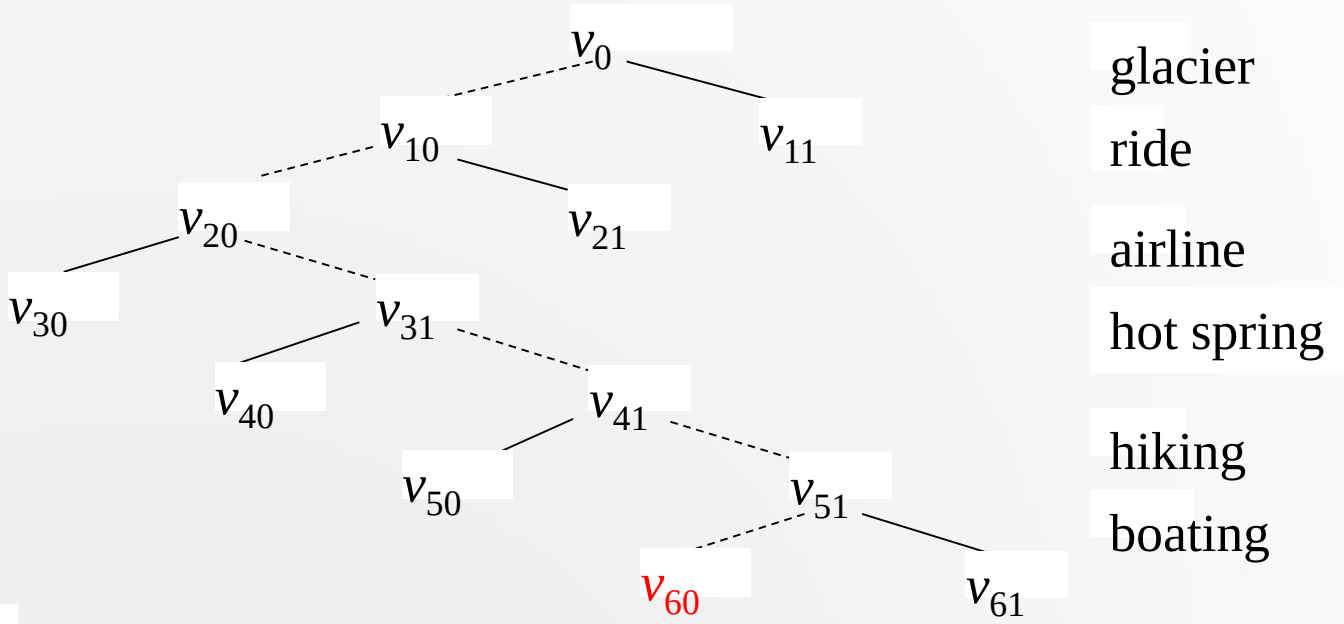
hot spring

$s(v_{30}) = \{q_1, q_5\}$ $s(v_{31}) = \{q_1, q_3, q_5\}$

Priority-first searching algorithm

Last step:





$s(v_{40}) = \{q_3, q_5\}$ $s(v_{41}) = \{q_1, q_3, q_5\}$
$s(v_{50}) = \{q_1, q_5\}$ $s(v_{51}) = \{q_1, q_3, q_5\}$
$s(v_{60}) = \{q_1, q_3, q_5\}$ $s(v_{61}) = \{q_1, q_3\}$

Most popular package: airline, hot spring, hiking

Priority-first searching algorithm

A probabilistic analysis shows that the average time complexity of the algorithm is bounded by

$$O(mn2^{m/2}),$$

much better than the basic algorithm.

Priority Queue

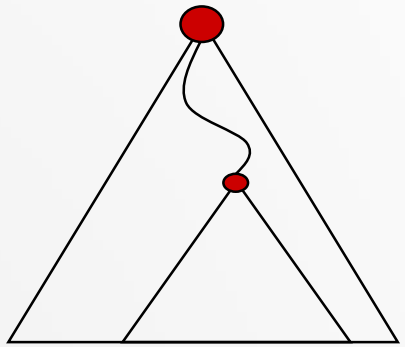
- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set S of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficiently.
- Applications:
  - Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

Basic Operations

- Operations on a max-priority queue:
  - Insert(S, x) - inserts the element x into the queue S
    - $S \leftarrow S \cup \{x\}$.
  - Maximum(S) - returns the element of S with the largest key.
  - Extract-Max(S) - removes and returns the element of S with the largest key.
  - Increase-Key(S, x, k) – increases the value of element x's key to the new value k.
- Min-priority queue supports Insert, Minimum, Extract-Min, and Decrease-Key.
- Heap gives a good compromise between fast insertion but slow extraction and vice versa.

Priority Queue Implemented by Using Heaps

- Max-Heap
  - For every node excluding the root, the value stored in that node is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.



- Min-Heap
  - For every node excluding the root, the value stored in that node is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root

Heap-Extract-Max(*A*)

Implements the Extract-Max operation.

Heap-Extract-Max(*A*)
1. if *heap-size*[*A*] < 1
2. then error “heap underflow”
3. *max* ← *A*[1]
4. *A*[1] ← *A*[*heap-size*[*A*]]
5. *heap-size*[*A*] ← *heap-size*[*A*] - 1
6. MaxHeapify(*A*, 1)
7. return *max*

Running time : Dominated by the running time of MaxHeapify
= $O(\lg n)$

Heap-Insert(*A*, *key*)

Heap-Insert(*A*, *key*)

1. *heap-size*[*A*] ← *heap-size*[*A*] + 1
2. *i* ← *heap-size*[*A*]
4. **while** *i* > 1 **and** *A*[Parent(*i*)] < *key*
5. **do** *A*[*i*] ← *A*[Parent(*i*)]
6. *i* ← Parent(*i*)
7. *A*[*i*] ← *key*

Running time is $O(\lg n)$.

The path traced from the new leaf to the root has length $O(\lg n)$.

Examples

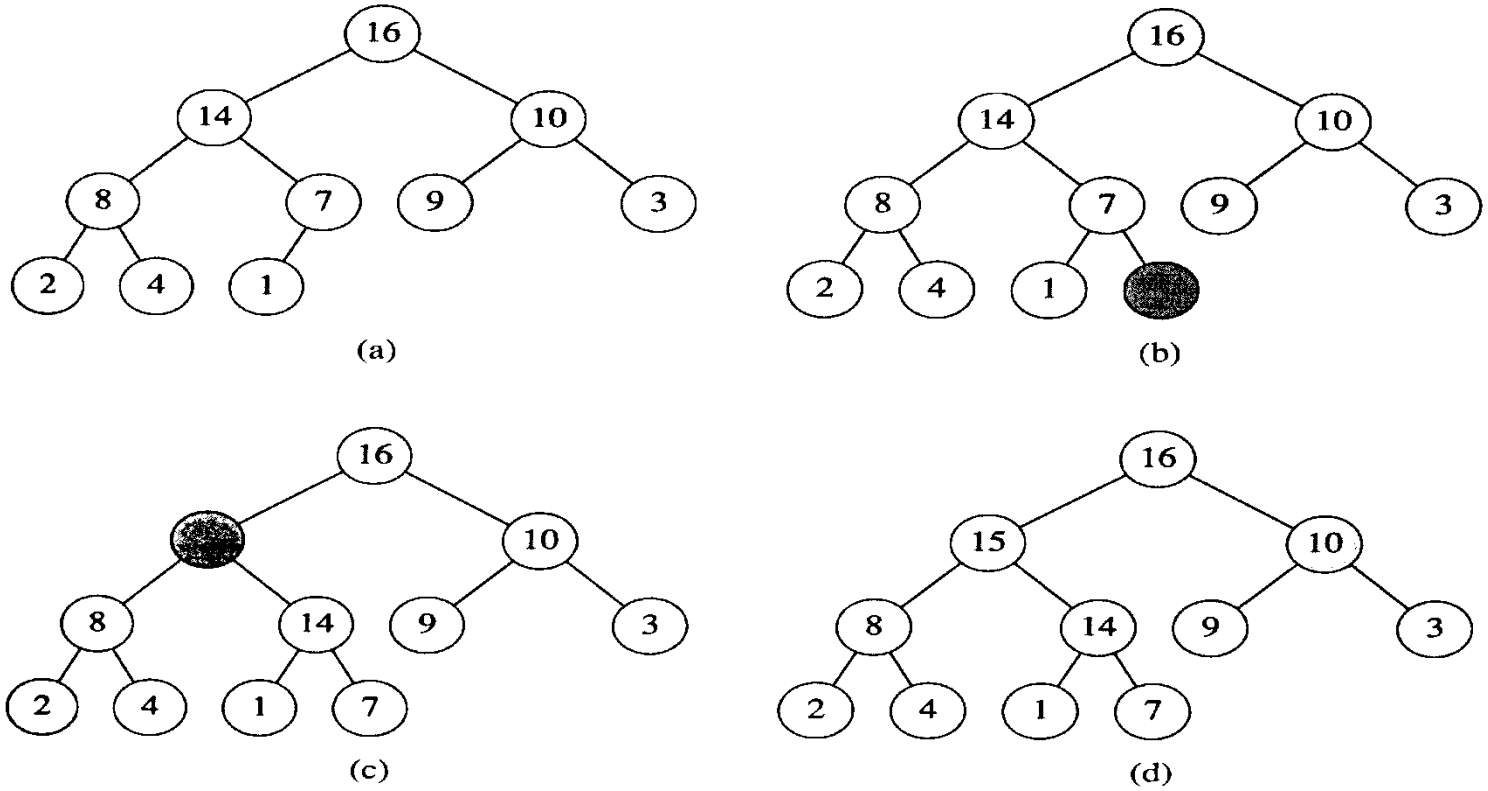


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

Heap-Increase-Key(*A*, *i*, *key*)

```
Heap-Increase-Key(A, i, key)
1    If key < A[i]
2       then error "new key is smaller than the current key"
3    A[i] ← key
4    while i > 1 and  A[Parent[i]] < A[i]
5          do exchange A[i] ↔ A[Parent[i]]
6             i ← Parent[i]
```

```
Heap-Insert(A, key)
1    heap-size[A] ← heap-size[A] + 1
2    A[heap-size[A]] ← – ∞
3    Heap-Increase-Key(A, heap-size[A], key)
```

Experiments

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

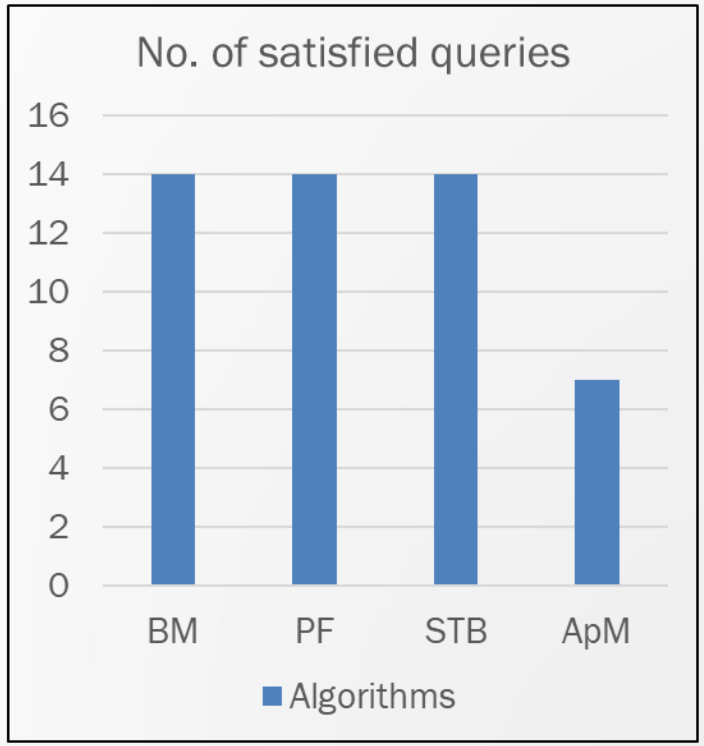
1) Basic method (described in this paper, BM for short),
2) Priority-first search (discussed in this paper, PF for short),
3) Signature tree based [5] (*STB* for short),
4) Approximation method [20] (*ApM* for short).

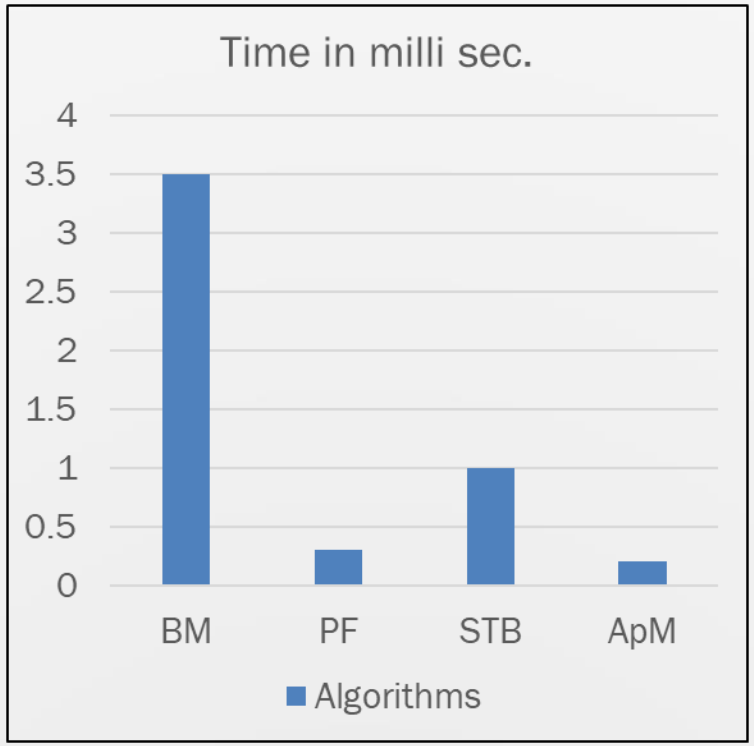
EXperiments

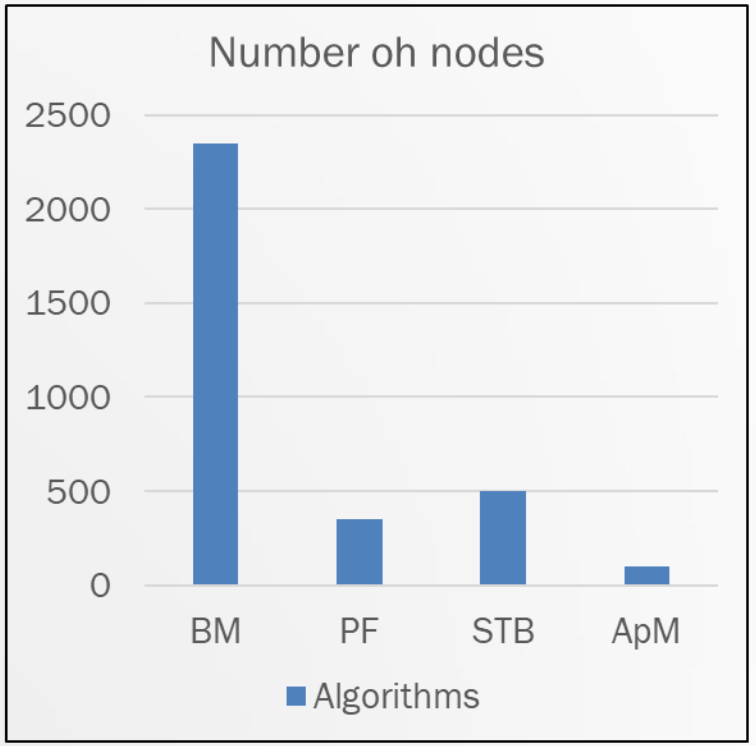
- All the four methods are implemented by ourselves. The code is written in in C++, running on a Linux machine with 32GB of memory and a 2.9GHz 64-core processor.
- Real data: 100 customers' favourites at a Chinese restaurant and surveyed over a long time period. The investigation was designed with 10 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.
- Synthetic data: 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.

Experiments (SPD on real data)

- *Test results on real data sets for SPD*

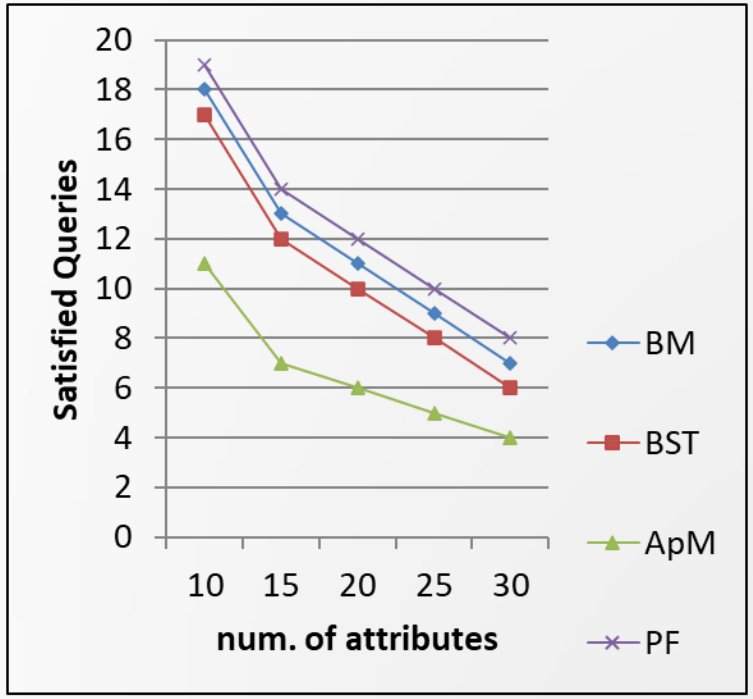


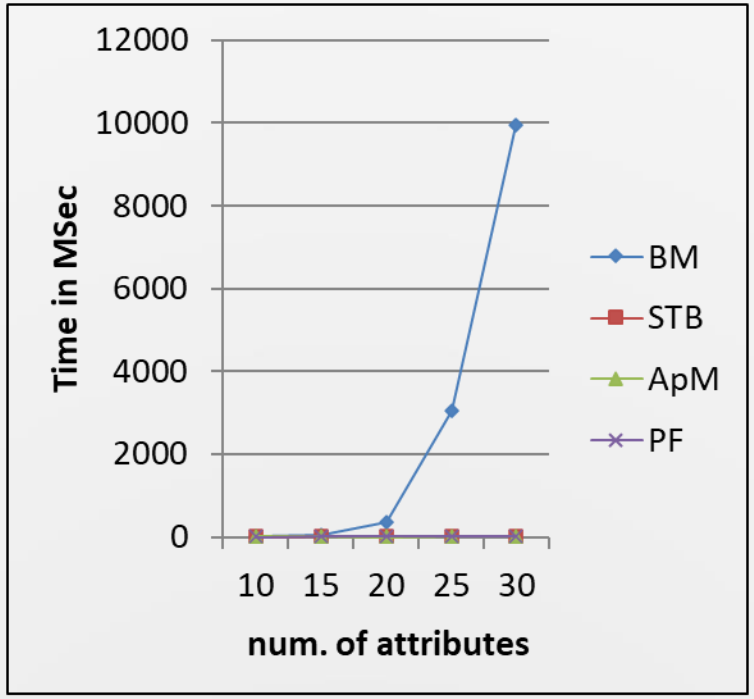


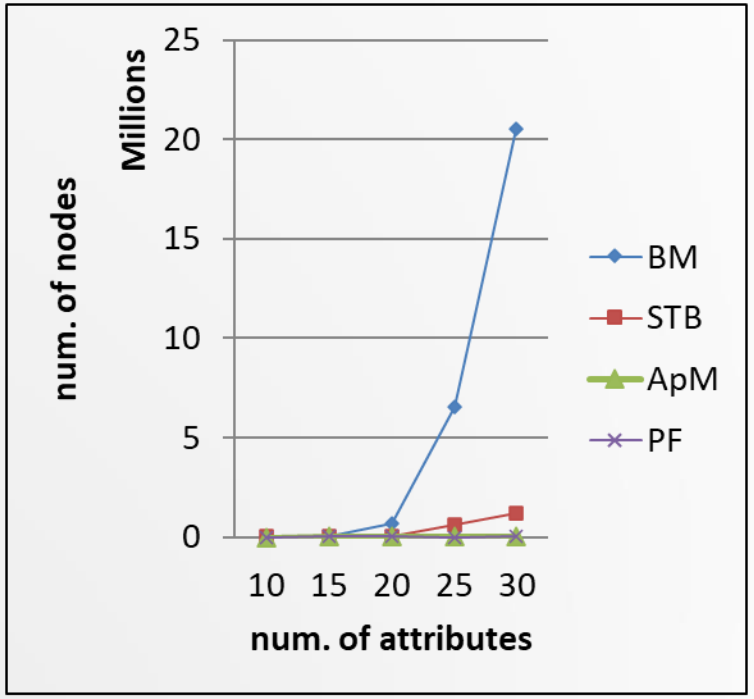


Experiments (SPD on synthetic data)

- *Test results for varying attributes on SPD*

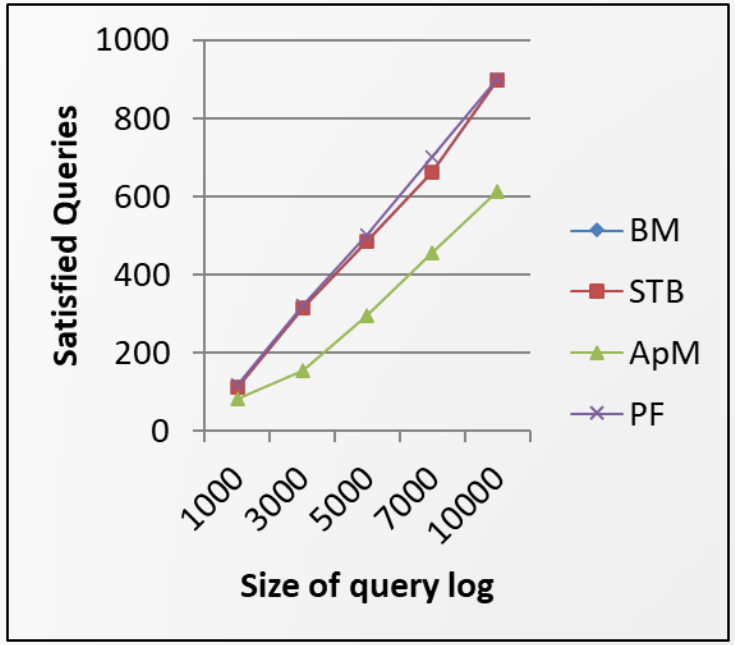


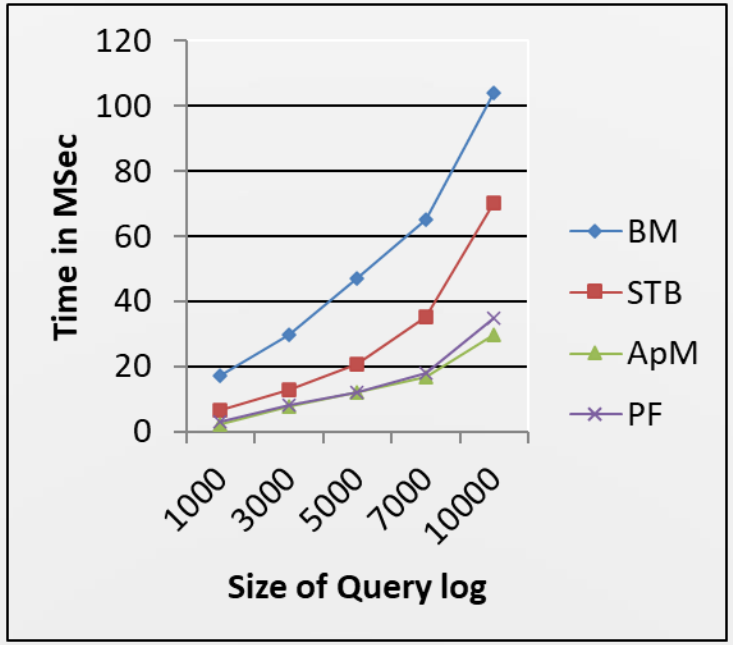


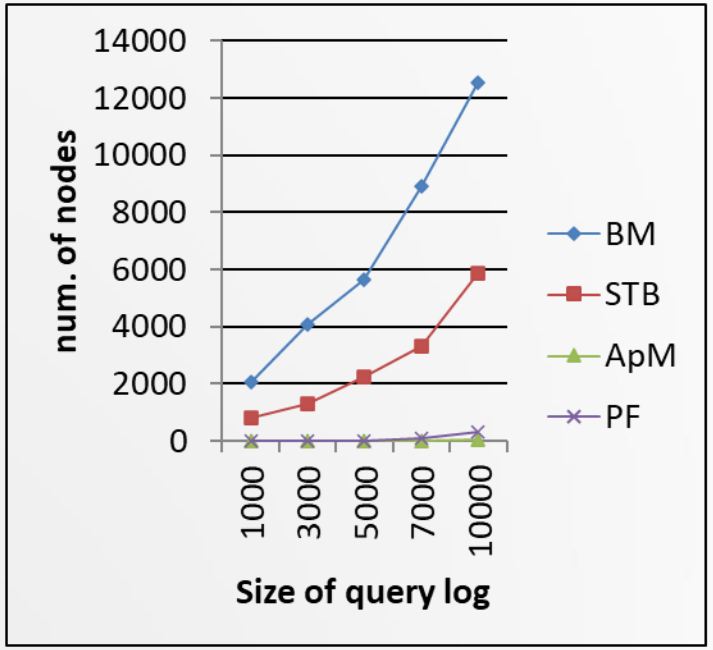


Experiments (SPD on synthetic data)

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







Graph-search based method

Representing a query as an attribute sequence:

- Let $Q = \{q_1, ..., q_m\}$ be a query log. We use $q_i[j]$ to represent the value of the jth attribute in q_i ($i = 1, ..., m$).
- For each $q_i = c_{i1}c_{i2} \ldots c_{im}$ ($c_{ij} \in \{0, 1. *\}$, $j = 1, ..., m$), we will create another sequence: $r_i = d_{i_1} \ldots d_{i_k} (k \leq m)$, where $d_{j_l} = a_{j_l}$ if $c_{ij_l} = q_i[j_l] = 1$, or $d_{j_l} = (a_{j_l}, *)$ if $c_{ij_l} = q_i[j_l] = *$. If $c_{ij_l} = q_i[j_l] = 1$, a_{j_l} will not appear in r_i at all.

$$q_1 = (1, *, 0, *, 1, *)$$

q_1 = hot-spring. (ride, *), (hike, *). airline.(boating, *)

Sorted Attribute Sequences

Sorted attribute sequence:

- Compute the appearance frequencies of attributes in a query log

attribute	Hot spring	ride	glacier	hike	airline	boating
frequency	5/6	3/6	3/6	5/6	6/6	5/6

- Global ordering of attribute such that the most frequent attribute appears first:

airline → Hot spring → hike → boating → ride → glacier

Sorted Attribute Sequences

Sorted attribute sequence:

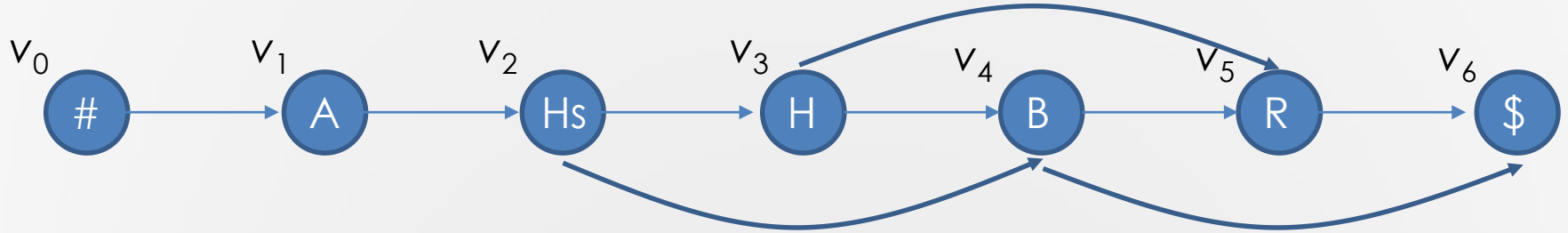
Query ID	Attribute sequence	Sorted attribute sequence
q1	Hs.(R, *).(H, *).A.(B, *)	#.A.Hs.(H, *).(B, *).(R, *).$
q2	Hs.G.(H, *).(A, *).(B, *)	#.(A,*).Hs.(H, *).(B, *).G.$
q3	(Hs, *).H.A.(B, *)	#.A.(Hs, *).H.(B, *).$
q4	(Hs, *).(R, *).G.(H, *).A.(B, *)	#.A.(Hs, *).(H, *).(B, *).(R, *).G.$
q5	(Hs, *).(H, *).(A, *)	#.(A, *).(Hs, *).(H, *).$
q6	(Hs, *).R.(G, *).B	#.(Hs, *).B.R.(G, *).$

Legend: Hs – hot spring, R – ride, G – glacier, H – hike, A – airline, B - boating

p-Graphs

Definition Let $q = d_0 d_1 \ldots d_k d_{k+1}$ 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 ($j = 0, \ldots, k + 1$); and an edge for (d_j, d_{j+1}) for each $j \in \{0, \ldots, k\}$. In addition, there may be an edge from d_j to d_{j+2} for each $j \in \{0, \ldots, k - 1\}$ if d_{j+1} is a pair of the form $(a, *)$, where a is an attribute. Each off-line edge is called a span.

q_1 = (1, *, 0, *, 1, *) = #.A.Hs.(H, *).(B, *).(R, *).$



Each span is represented by a sub-path covered by it: (H, *) = <v_2, v_3, v_4>

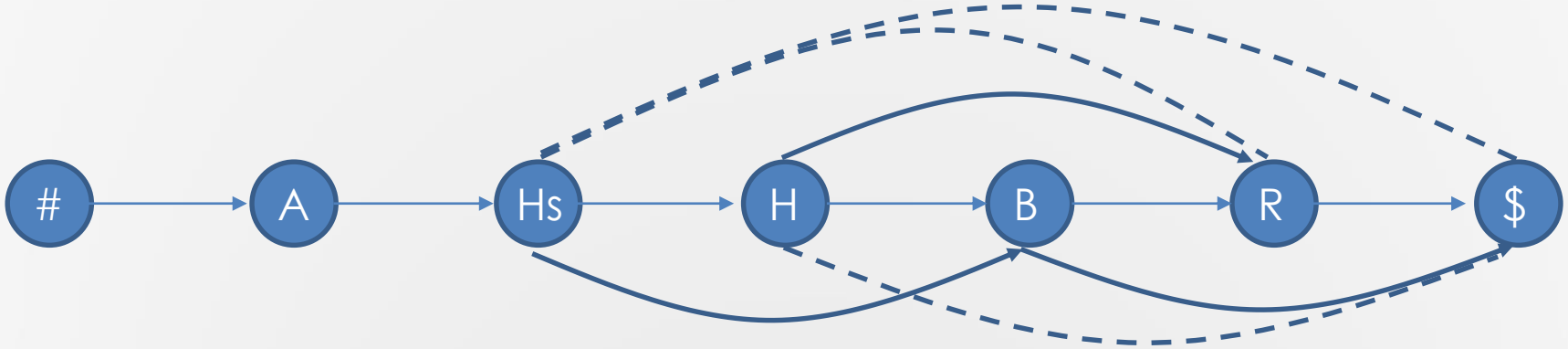
P*-Graphs

- Let $s_1 = \langle v_1, ..., v_k \rangle$ and $s_2 = \langle u_1, ..., u_l \rangle$ be two spans attached on a same path. We say, s_1 and s_2 are overlapped, if $u_1 = v_j$ for some $v_j \in \{v_2, ..., v_{k-1}\}$, or if $v_1 = u_{j'}$ for some $u_{j'} \in \{u_2, ..., u_{l-1}\}$.
- For example, in the above figure, $\langle v_2, v_3, v_4 \rangle$ and $\langle v_3, v_4, v_5 \rangle$ are overlapped. $\langle v_3, v_4, v_5 \rangle$ and $\langle v_4, v_5, v_6 \rangle$ are also overlapped. But $\langle v_2, v_3, v_4 \rangle$ and $\langle v_4, v_5, v_6 \rangle$ 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: (H, *) and (B, *).
- The overlapped spans exhibit 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.

*P**-Graph

Definition 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 $P^* = p \cup S^*$.

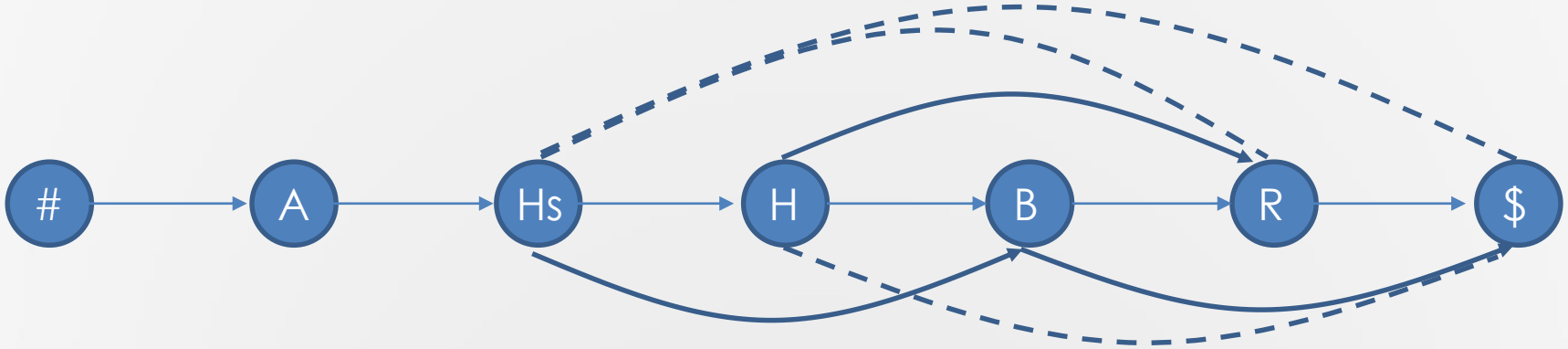
q_1 = (1, *, 0, *, 1, *) = #.A.Hs.(H, *).(B, *).(R, *).$



*P**-Graph

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

q_1 = (1, *, 0, *, 1, *) = #.A.Hs.(H, *).(B, *).(R, *).$

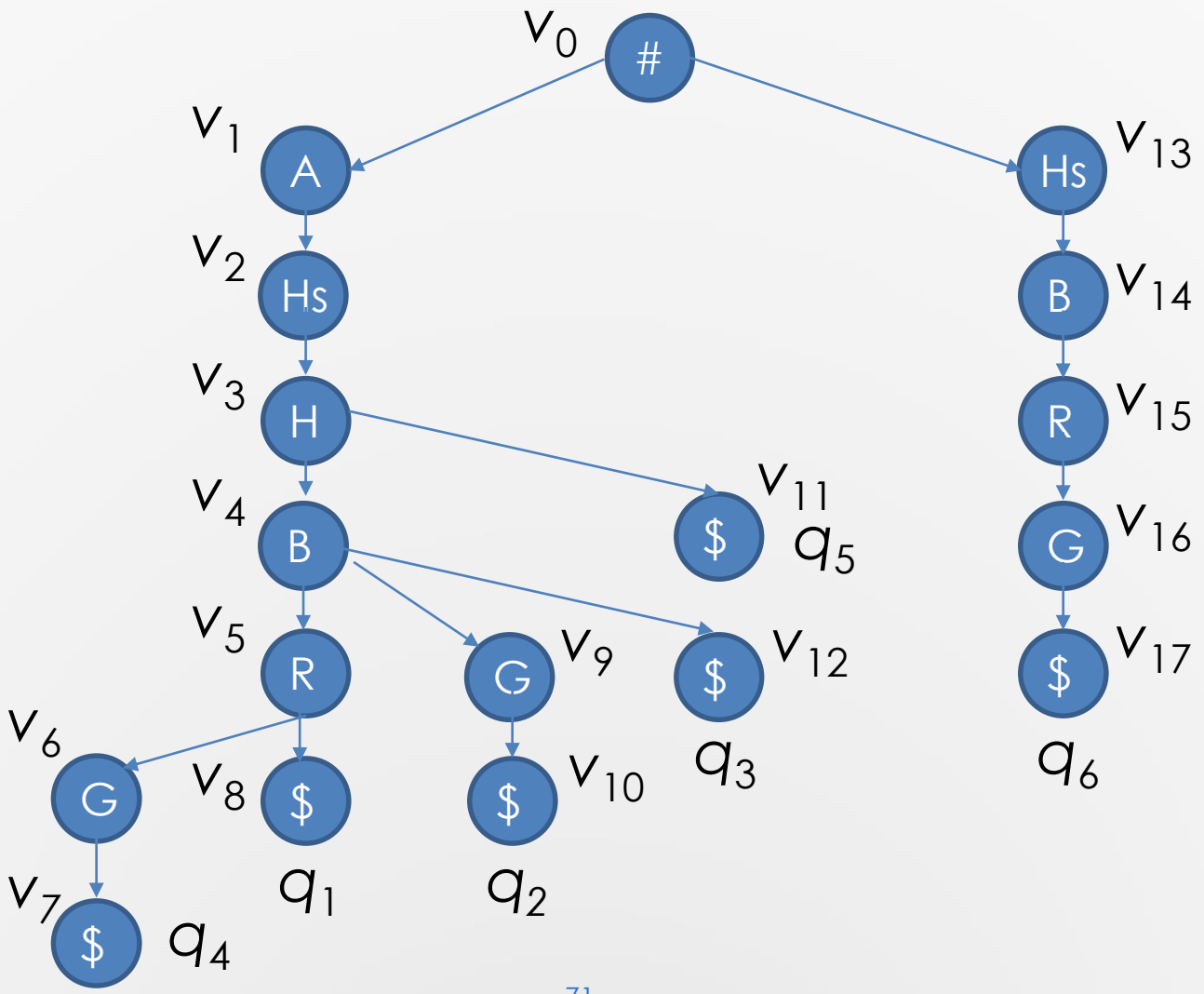


Trie over Main Paths

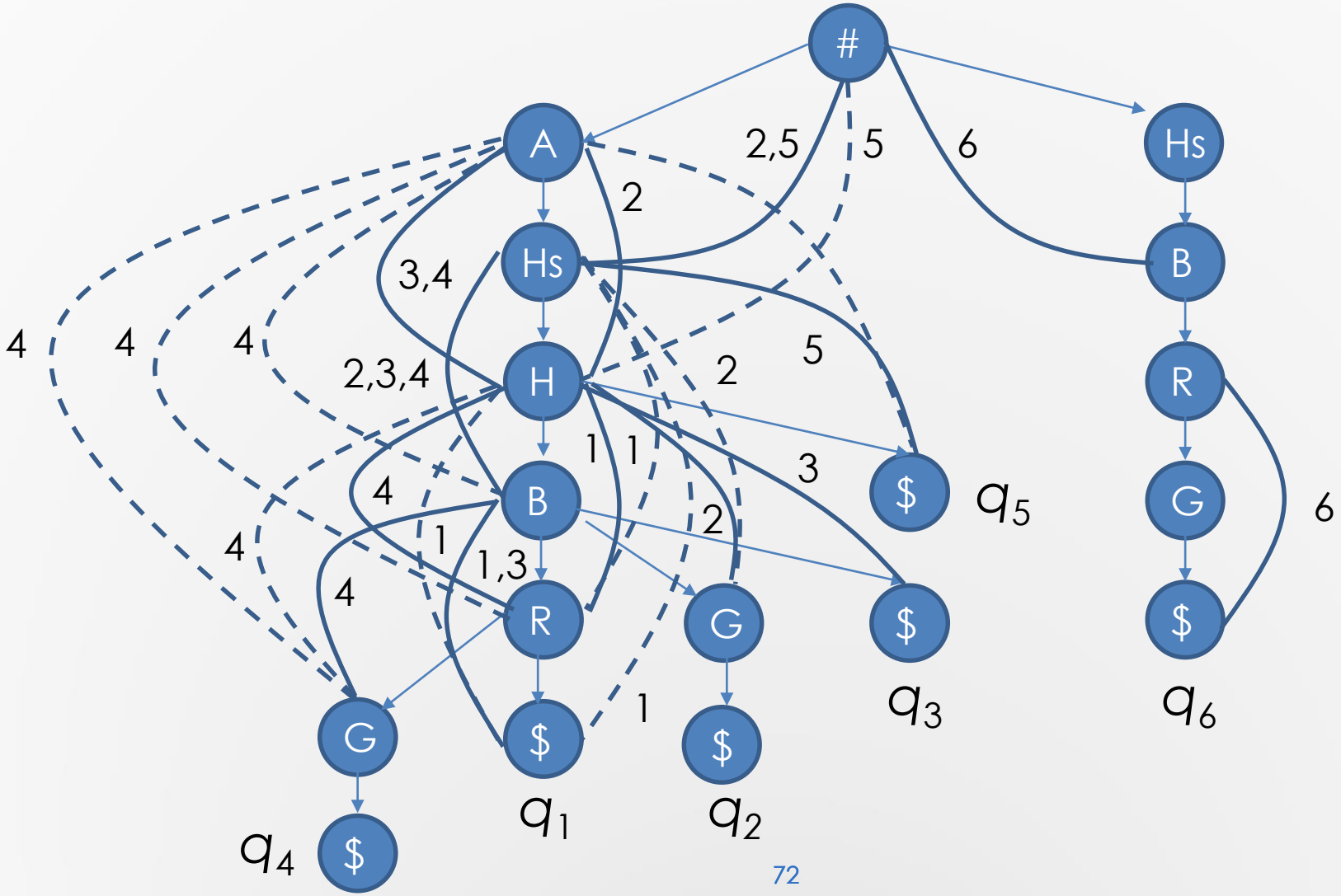
- Let $p_1, p_2, \ldots, p_n$ be the main paths in $P_1^*, P_2^*, \ldots, P_n^*$. A trie over $R = \{p_1, p_2, \ldots, p_n\}$, denoted as $T = trie(R)$, is defined as follows.

- 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_j ($j = 1, \ldots, m$) contains all those sequences with the same first attribute name. The tries: $trie(R_1), trie(R_2), \ldots, trie(R_m)$ are constructed in the same way except that at the kth step, the splitting of sets is based on the kth attribute (along the global ordering of attributes). They are then connected from their respective roots to a single node to create $trie(R)$.

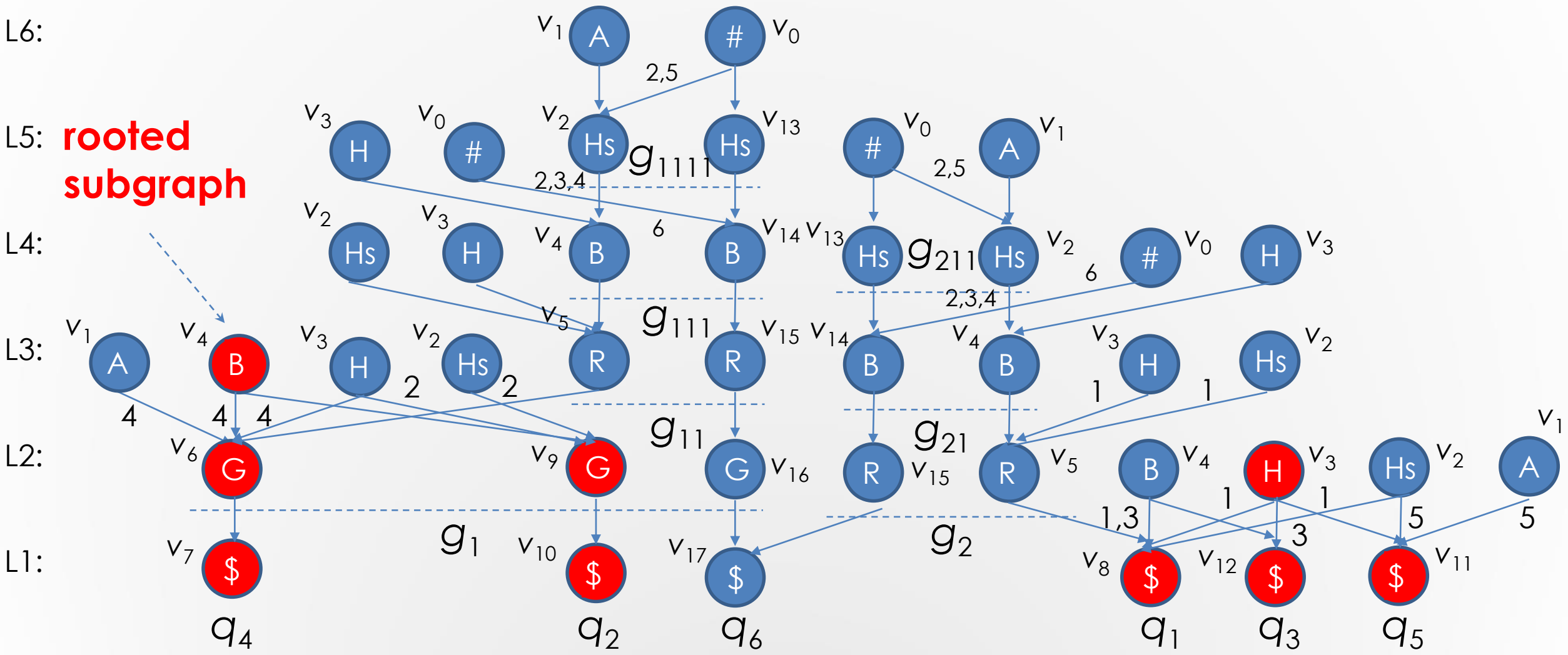
Trie over Main Paths



Trie-like Graphs



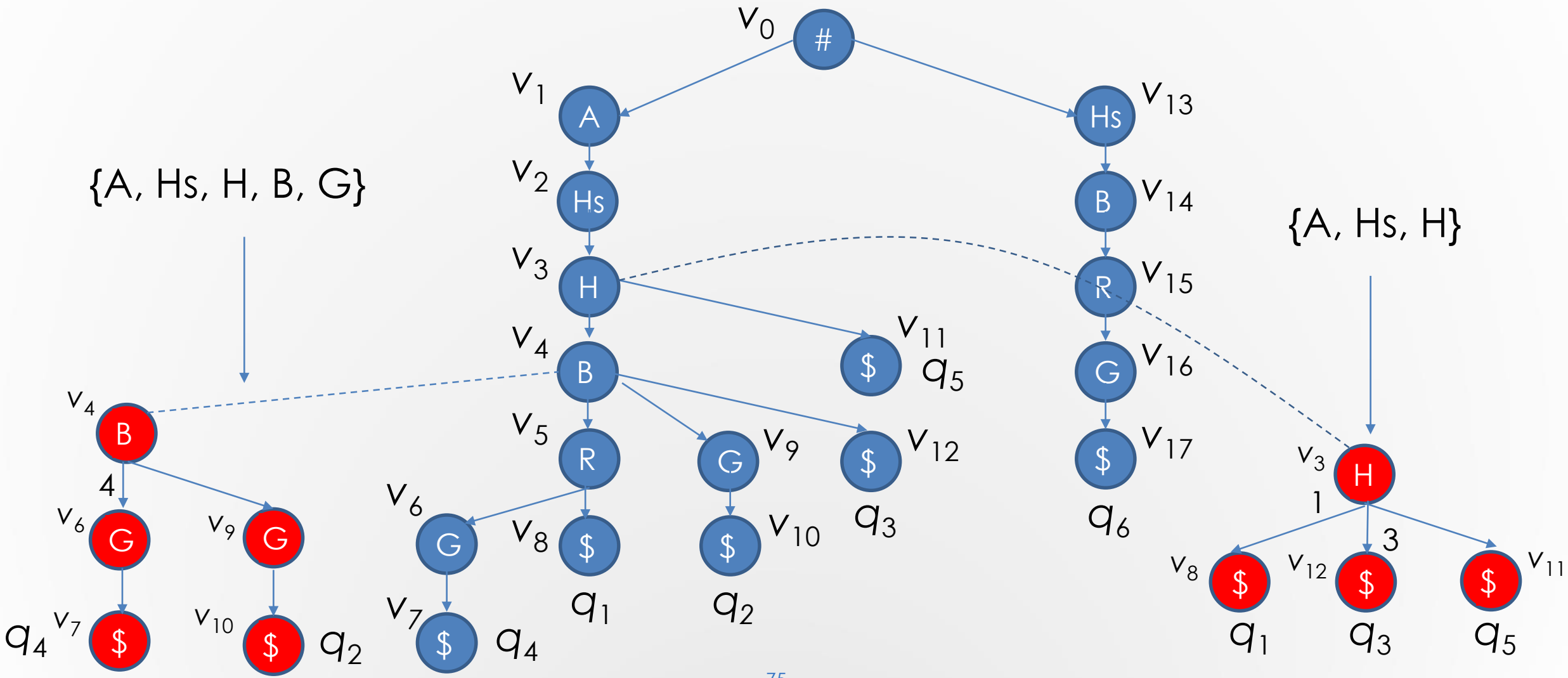
Layered Graphs



How to Find Answers

- Each node without parents in a layered graph is called a root.
- All the nodes reachable from a root make up a rooted subgraph.
- Each rooted subgraph corresponds to a subset of queries satisfied by a certain package.

How to Find Answers



Algorithm

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. push(*S*, *g*);
3. **while** *S* is not empty **do**
4. *g'* := pop(*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 that all the nodes in a group with the same label;
7. **for** each $j \in \{1, ..., k\}$ **do**
8. **if** $|g_j| > 1$ **then**
9. push(*S*, g_j);
10. return *findPackage*(*G′*);

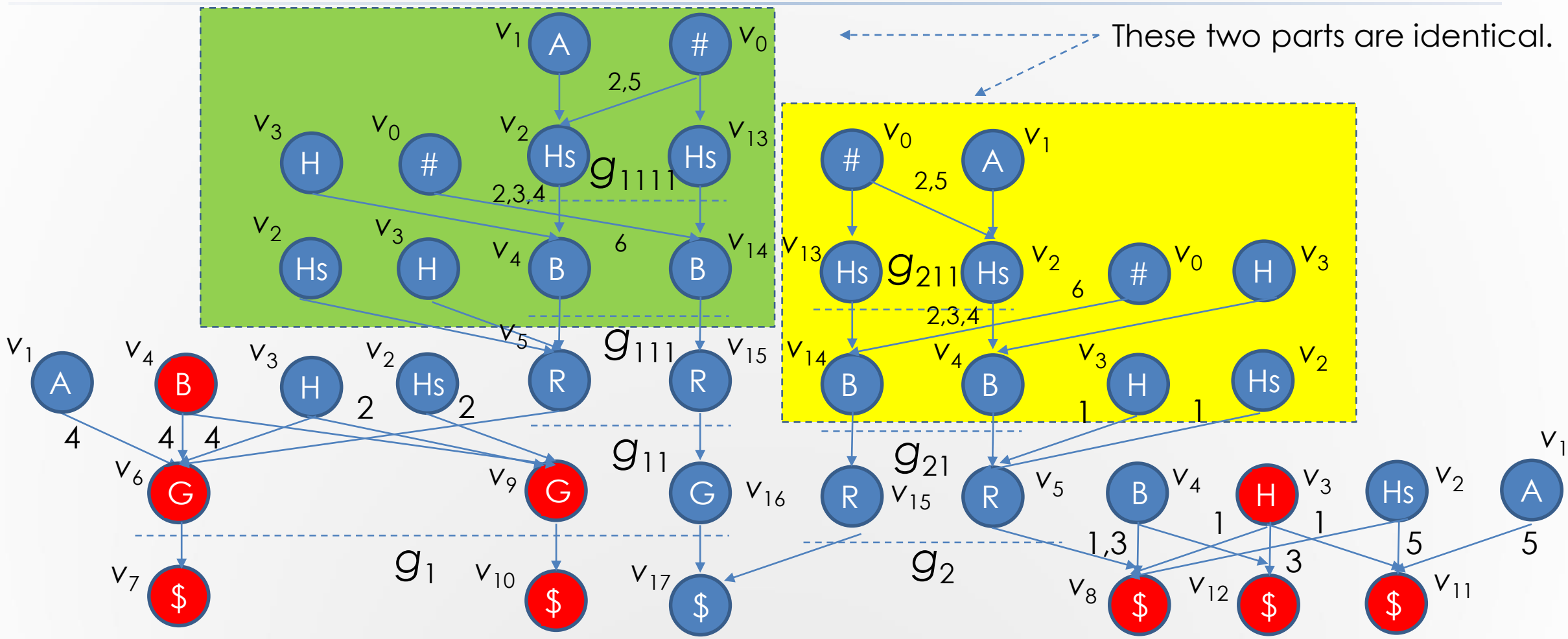
Algorithm

Algorithm 2: *findPackage*(*G'*)

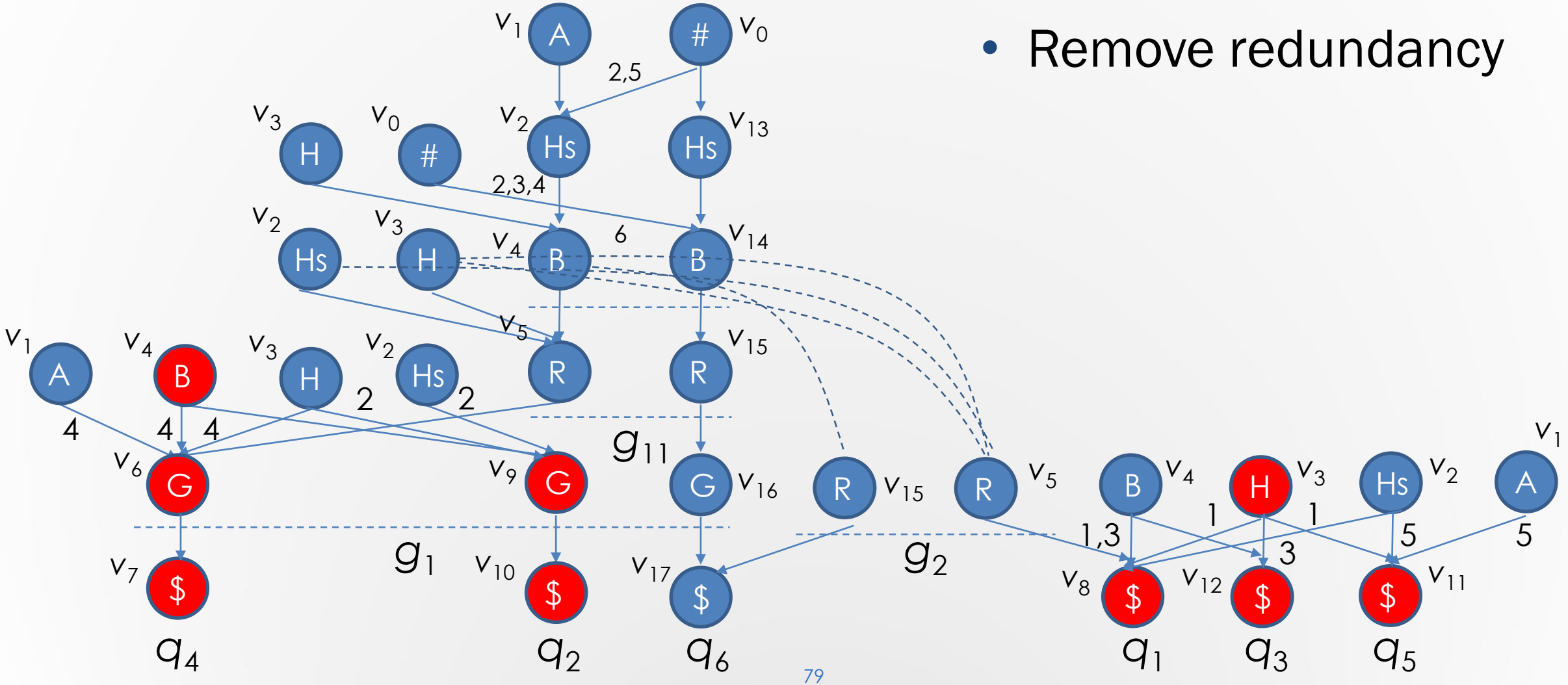
Input: a layered graph *G'*
Output: a most popular package

1. (*u*, s, *f*) := (*null*, 0, *Φ*); (* find a package for a maximum subset of queries. *)
2. **for** each rooted subgraph G_v **do**
3. determine the subset *Q'* of satisfied queries in G_v;
4. **if** |*Q'*| > s **then**
5. *u* := *v*; s := |*Q'*|; *f* := *Q'*;
6. return (*u*, *s*, *f*);

Further Improvement



Further Improvements



- Remove redundancy

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 frequencies of attribute appearances in queries. Since in this process each attribute in a query 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 queries. It is obviously bounded by $O(nm \log m)$.
  - τ_{22}: Time for constructing p^*-graphs for each of queries. Since for each query 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 $O(nm^2)$.

Time Complexity Analysis

- The third part τ_3 is the time for searching G to generate its layered representation. 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 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 rooted subgraph G' is bounded by $O(nm)$, the cost of this part of computation is bounded by $O(n^2m^3)$.
- Thus, the total running time of our algorithm is

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