PARTIAL SUPPORT USING A “PARTIAL SUPPORT TREE”

Frans Coenen

Department of Computer Science
The University of Liverpool
Liverpool, L69 ZF, UK.
email: frans@csc.liv.ac.uk
tel: +44 (0)151 794 3698
fax: +44 (0)151 794 3715

1 INTRODUCTION

In [4] an algorithm for producing a linked list of structures each of which contained the code and support for unique rows in a transaction table was described. The suggested advantage was that this was a way of determining association rules using only one pass of the database but without requiring the entire database to be stored in memory. In this paper we present a second algorithm which has a similar aim except in this case the partial supports are contained in a tree structure. As before the nodes in this tree represent unique rows in the table, however in this case the nodes are arranged lexically and according to whether one is a subset of another. A child node is a superset of its parent node while a sibling nodes are ordered lexically. Examples are given later in the text.

The tree commences with a single node which has a sibling branch and a child branch. New nodes are placed into the tree by traversing the tree until an appropriate location is found. This is achieved using the following “rules” (where $R$ represents the bit pattern of the row in the table currently under consideration and $B$ a bit pattern attached to an existing node in the tree):

**Rule 1:** If $R = B$ (i.e. an identical node is found) simply increment the support associated with the node and return.

**Rule 2:** If $R < B$ and $R \subseteq B$ create a new node for $R$ and place the existing node associated with $B$ on the new node’s child branch. The new node is then placed either as a new root node, or is added to the child or sibling branch of a previously investigated node (see examples given below).

**Rule 3:** If $R < B$ and $R \not\subseteq B$ create a new node for $R$ and place the existing node associated with $B$ on the new node’s sibling branch. The new node is then placed either as a new root node, or is added to the child or sibling branch of a previously investigated node (again, see examples given below).

**Rule 4:** If $R > B$ and $R \supset B$ then:

- If node associated with $B$ is a leaf node create a new node for $R$ and add this to the existing node’s child branch.
- Otherwise proceed down child branch and repeat.

**Rule 5:** If $R > B$ and $R \perp \supset B$ then:

- If node associated with $B$ is a leaf node create a new node for $R$ and add this to the existing node’s sibling branch.
- Otherwise proceed down sibling branch and repeat.
Thus given a bit Pattern $R$ and a bit pattern $B$ associated with an existing node their are five possible outcomes. The possible relationships between $R$ and $B$ are given in Figure 1. Rules 4 and 5 above are fairly straightforward, however to achieve a better understanding of rules 2 to 3, and to supply some additional considerations, it is useful to give some illustrations of their operation. This is done in the following two sub-sections.

1.1 **Rule 2:** $R < B$ and $R \subset B$

The rule is entered either with the start node or as part of a search down a child or sibling branch. Consequently we have three possible outcomes (Figure 2).

However, there is an additional test we must make. Consider the tree given in Figure 3(a). If we wish to add node $B$ to the tree, applying the above rules gives the result shown in Figure 3(b). This is the desired result. However if we now consider the tree given in Figure 3(c) and add node $B$ to this tree, through the application of the above rules we get the result shown in Figure 3(d). This is not the desired result. To resolve this situation we need to check the sibling branch of $BC$ and if this is not a superset of the new node move it up so that it becomes a sibling of the new node.

This still does not necessarily resolve the situation entirely because not all node in the sibling branch are necessarily all supersets or not supersets of the node $R$. For example in Figure 4(a) we commence with a five node tree structure, we than add the node $A$ with the result as shown in Figure 4(b). Node $AC$ is in the right place, however nodes $B$ and $C$ need to be moved up so that they become siblings of $A$. This task is not too computationally expensive in that (where some nodes are to be moved up and others should remain) we only need to find the 3 point in the sibling branch where to “split the tree”.

![Figure 1: Possible relationships between $R$ and $B$](image-url)
Figure 2: Possible outcomes from the application of Rule 2 (R single circle, B double circle, other nodes dashed circle)

Figure 3: Additional considerations when applying Rule 2 (R single circle, B double circle, correct position indicated for C, other nodes dashed circle)
1.2 **Rule 3: $R < B$ and $R \not\subset B$**

As with Rule 2 described above this rule is entered either with the start node or as part of a search down a child or sibling branch. Consequently we have three possible outcomes as shown in Figure 5. In this case there are no special considerations that need to be taken into account when applying the rule (unlike Rule 2).

2 **THE DATA STRUCTURE**

As in early work described in [2, 3] use is again made of bit codes to represent the bit patterns, in particular we use the array encoding described in [3]. The data structure used is therefore defined (in C) as follows:

```c
typedef struct partSupport{
    int code*;
    int support;
```
3 INPUT

The algorithm is designed to be linked in to a “main” program. For example:

```c
#include "support.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* ------ FUNCTION PROTOTYPES ------ */
extern void partialSupport(char *);
extern void outputPartSupportTree(void);

/* ------------------------------- */
/*    MAIN    */
/* ------------------------------- */

void main(int argc, char *argv[])
{
    /* Check Input. */
    if (argc < 2) {
        printf("INPUT ERROR - no file name.
");  
        exit(1);
    }

    /* Create partial support file. */
    partialSupport(argv[1]);

    /* Output result (testing only). */
    outputPartSupportTree();
}
```

Where the function `partialSupport` instigates the calculation and storage of the partial supports, and the `outputPartSupportLinkedList` procedure outputs the result (the latter used for testing purposes only).

The above is invoked as follows:

```
./mainpro <INPUTFILE>
```

where the input (as before) is in the form of a file, for example (\(n = 8\) and \(m = 10\)):

<table>
<thead>
<tr>
<th>Row</th>
<th>Interpretation in DB</th>
<th>Bit Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>B</td>
<td>6</td>
</tr>
<tr>
<td>110</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>011</td>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>001</td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>001</td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>111</td>
<td>ABC</td>
<td>7</td>
</tr>
<tr>
<td>101</td>
<td>AC</td>
<td>5</td>
</tr>
</tbody>
</table>
Given this input we can expect (using the output routines used for testing) output of the form:

(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.2) 5 support = 2
(2) 2 support = 1
(2.1) 6 support = 2
(3) 4 support = 2

where the bracketed sequence of integers represents a node identifier and the following integer the bit code associated with each row. This would produce a tree structure a shown in Figure 6. Note the node numbering convention used (again for testing purposes only).

4 ALGORITHM OVERVIEW

Broadly the algorithm operates as follows:

- Read first line in the table describing a transaction.
- From this line determine the number of columns/attributes in the table and then use this information to generate the first structure in the linked list.
- Read the remainder of the table in an iterative manner. On each iteration:
  - If the row is already contained in the tree structure increment the support variable in the appropriate structure.
  - Otherwise create a new node in the structure (according to the rules identified above).

5 DETAILED DESIGN OF ALGORITHM

The partial support algorithm is implemented in C using the following functions/procedures:

1. partialSupport
2. processInput
3. addSupport
4. addSupport2
5. checkSiblingBranch
6. checkCodes
7. equalityCheck
8. subsetCheck
9. beforeCheck
10. generateBitPattern
11. createStructure
12. outputPartSupportLinkedList
13. outputPartSupport1
14. outputPartSupport2

The partialSupport procedure is used to open and close the input file and instigate the processing and is identical to that described in [4]. The processing is actually carried out by the processInput procedure which loops through the table. The addSupport and addSupport2 procedures are used to instigate the calculation of the bit patterns (codes) associated with each line in the table and either (a) create a new structure in the tree or (b) update the support for an existing structure. Code comparisons are carried out by the checkCodes function which in turn accesses the equalityCheck, subsetCheck and beforeCheck functions. Bit patterns are generated by the generateBitPattern function which is roughly the same as the function of the same name described in [4]. Where necessary new structures are created by the createStructure function. Finally the outputPartSupportTreet, outputPartSupport1 and outputPartSupport2 procedures are used to output the resulting linked list of partial supports and are used for testing purposes only.

The code includes four global variables:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>startPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Global pointer to start of linked list structure (the data type PARTSUPPORTPTR is defined in support.h).</td>
</tr>
<tr>
<td>arraySize</td>
<td>Integer</td>
<td>Global variable indicating size of bit code storage array.</td>
</tr>
<tr>
<td>totalRows</td>
<td>Integer</td>
<td>Global variable in which the total number of rows in the input table is stored.</td>
</tr>
<tr>
<td>totalCols</td>
<td>Integer</td>
<td>Global variable in which the total number of columns in the input table is stored.</td>
</tr>
</tbody>
</table>

In the following sub-sections the detailed design of the addSupport, addSupport2, checkSiblingBranch, checkCodes, equalityCheck, subsetCheck, beforeCheck and createStructure functions/procedures are described. Details concerning the processInput procedure and generateBitPattern functions can be found in [4]. The output procedures are not described as these are not an integral part of the algorithm. The designs are given in the form of Nassi-Shneiderman charts. Readers who are not interested in the implementational details of the algorithm can skip this section and go straight to Section 5 — “Analysis”. All code is listed in Appendix A and test scenarios in Appendix B.
5.1 PROCESS INPUT

The processInput procedure is the principal procedure for calculating support values. The procedure operates as follows:

- Read first line in table and from this line determine the number of columns in the table and consequently the size of the unsigned integer array required to represent the bit codes.
- Create a structure to represent the first line (the initial root node for the tree).
- Process the rest of the table.

The procedure includes the following data items:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_file</td>
<td>FILE pointer</td>
<td>Formal parameter, pointer to current location in input file.</td>
</tr>
<tr>
<td>input</td>
<td>512 character string</td>
<td>Local variable in which current input line is stored as a string (32 is the maximum size of the string).</td>
</tr>
<tr>
<td>cols</td>
<td>Integer</td>
<td>Local variable containing the number of columns in the input table.</td>
</tr>
<tr>
<td>code</td>
<td>Unsigned integer array pointer</td>
<td>Local variable to hold bit codes.</td>
</tr>
<tr>
<td>oldPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Local variable, pointer to temporarily hold previous location in data structure when calling addSupport procedure. This points to NULL initially.</td>
</tr>
</tbody>
</table>

A detailed design for the processInput procedure is presented in Figure 7.
5.2 ADD SUPPORT

The addSupport procedure is called for each line in the table. This is then processed according to the rules presented in Section 1 above with respect to the nodes in the tree structure as created “sofar”. There are five alternatives:

- The current node and the row are the same therefore increment the support associated with the current node (Rule 1).
- The new node is a parent node of the current node (Rule 2).
- The new node is an elder sibling of the current node (Rule 3).
- The new node is located somewhere on the child branch of the current node (Rule 4).
- The new node is located somewhere on the sibling branch of the current node (Rule 5).

Selections are made according to the result returned by the checkCodes function (see below). The procedure includes the following data items:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag</td>
<td>Integer</td>
<td>Formal parameter describing the type of branch currently under consideration: 0 = root, 1 - child, 2 = sibling.</td>
</tr>
<tr>
<td>linkPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Formal parameter to control loop through linked list structure.</td>
</tr>
<tr>
<td>oldPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Formal parameter to hold temporary pointers to structures.</td>
</tr>
<tr>
<td>rowCode</td>
<td>Unsigned integer array Pointer</td>
<td>Formal parameter describing the code/bit pattern for the current line in the table (inputString)</td>
</tr>
<tr>
<td>newPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Local variable to hold pointer to newly created structure (if any).</td>
</tr>
<tr>
<td>newSiblingPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Local variable to hold pointer to new sibling pointer where required.</td>
</tr>
</tbody>
</table>

A detailed design for the addSupport procedure is presented in Figure 8.

5.3 ADD SUPPORT 2

The addSupport procedure is called where Rule 2 or Rule 3 is implemented, i.e. where the code associated with a row in the table under consideration is “before” the node in the tree currently under consideration. In this case the row must be inserted into the tree so that the tree node is a sibling or a child of the row as appropriate, and the row itself represents either a new root node or it is attached as a child or sibling to a previous node. The latter is implemented according to a flag set to 0 for a new root node, 1 for a child branch and 2 for a sibling branch. The procedure includes the following data items:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag</td>
<td>Integer</td>
<td>Formal parameter describing the type of branch currently under consideration: 0 = root, 1 - child, 2 = sibling.</td>
</tr>
<tr>
<td>newPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Formal parameter to hold pointer to newly created structure (if any).</td>
</tr>
<tr>
<td>oldPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Formal parameter to hold temporary pointers to structures.</td>
</tr>
</tbody>
</table>

A detailed design for the addSupport2 procedure is presented in Figure 9.
Figure 8: Nassi-Shneiderman chart for addSupport procedure \((LP = \text{linkPtr}, \ NP = \text{newPtr}, \ CP = \text{childPtr}, \ SP = \text{siblingPtr} \text{ and } \ NSP = \text{newSiblingPtr})\) (e.g. \(OP- > N\) \text{ and } \ LP = \text{linkPtr})
5.4 CHECK SIBLING BRANCH

The *checkSiblingBranch* function is called in the special case where Rule 2 has been applied (new node inserted as a parent to an existing node) and it is possible that some nodes in the sibling branch of the existing node in the tree may have to be moved to become siblings of the newly inserted node. The function returns 1 if adjustment is required, and 0 otherwise. Only those nodes in the sibling branch (as illustrated in Sub-section 1.1) which are not supersets of the newly inserted node need to be moved. As the nodes are ordered any nodes that do need to moved will be stored in sequence after those nodes that do not need to be moved. There are four possibilities:

1. Sibling branch is empty therefore no nodes can be move (return 0).
2. The first node in the sibling branch is not a superset of new node therefore move the entire branch to become a sibling branch of the new node (return 1).
3. All nodes in sibling branch are supersets of row there for move none of them (return 0).
4. First section of the sibling branch contains one or more nodes that need to be moved up, will the remainder comprises nodes which are supersets of the new node. Therefore move those that are not supersets up to become siblings of the new node (return 1).

Selections are made using a called to the *subsetCheck* function described below. Where some nodes are to be moved up and some are not we need to find the “cut-off” point in the branch where the “change-over” takes place, i.e. we need to iterate along the branch. The function includes the following data items:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Formal parameter to control loop through linked list structure.</td>
</tr>
<tr>
<td>rowCode</td>
<td>Unsigned integer array Pointer</td>
<td>Formal parameter describing the code/bit pattern for the current line in the table (<em>inputString</em>).</td>
</tr>
<tr>
<td>newSiblingPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Formal parameter to hold pointer to new sibling pointer where required.</td>
</tr>
<tr>
<td>localLinkPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Local variable to hold pointer to location in tree structure during processing.</td>
</tr>
<tr>
<td>markergPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Local variable to hold pointer to act as a place marker in tree as it is processed.</td>
</tr>
</tbody>
</table>

A detailed design for the *checkSiblingBranch* procedure is presented in Figure 10.
Figure 10: Nassi-Shneiderman chart for checkSiblingBranch function
5.5 CHECK CODES

The `checkCodes` function is used to compare a current bit pattern/code to that held in a particular structure contained in the tree. As described in section 1 there are five possibilities:

- Same (return 1).
- Before and subset (return 2).
- Before and not subset (return 3).
- After and superset (return 4).
- After and not superset (return 5).

Specific checks are carried out by the `equalityCheck`, `beforeCheck` and `subsetCheck` functions described below. The `checkCodes` function uses two data items as follows:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowCode</td>
<td>Unsigned integer pointer</td>
<td>Formal parameter giving bit code for current line in table.</td>
</tr>
<tr>
<td>currentCode</td>
<td>Unsigned integer pointer</td>
<td>Formal parameter giving bit code for current structure.</td>
</tr>
</tbody>
</table>

A detailed design for the `checkCodes` procedure is presented in Figure 11.

5.6 EQUALITY CHECK

Function to determine whether to codes are identical or not. Returns 1 if so, and 0 otherwise. The function uses three data items:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>code1</td>
<td>Unsigned integer pointer</td>
<td>Formal parameter for first bit code.</td>
</tr>
<tr>
<td>code2</td>
<td>Unsigned integer pointer</td>
<td>Formal parameter for second bit code.</td>
</tr>
<tr>
<td>index</td>
<td>Integer</td>
<td>Local variable index with which to step through codes.</td>
</tr>
</tbody>
</table>

A detailed design for the `equalityCheck` procedure is presented in Figure 12.

5.7 BEFORE CHECK

Similar function to the `equalityCheck` function described above, but returns 1 if row is “before” current node, and 0 otherwise. There is however an added complication in that by “before” we mean lexiconically before and not numerically before. For example `AB` is before `B`, although the code associated with `AB` (3) is numerically after the code associated with `B` (2). Processing is carried out using a case statement the selector for which is made up of the remainder of two divisions by 2 (the second multiplied by 2). Thus:

\[
\text{selector} = (\text{number1 rem 2}) + 2(\text{number2 rem 2})
\]

Thus if `number1 = 3` and `number2 = 2` then:

\[
\text{selector} = (3 \text{ rem 2}) + 2(2 \text{ rem 2}) = 1 + 2(0) = 1
\]

This is then used to test the least significant bit of each code in an iterative manner during which the numbers are adjusted so that a new most “significant” bit is tested on each loop. As a result of each test there are four possibilities:

- `selector = 0` Both codes have a 0 bit n therefore continue.
Figure 11: Nassi-Shneiderman chart for checkCodes procedure

Figure 12: Nassi-Shneiderman chart for equalityCheck procedure
Figure 13: Nassi-Shneiderman chart for beforeCheck procedure

selector = 1  Bit n in first number is 1 and in second number is 0, therefore before.

selector = 2  Bit n in second number is 1 and in first number is 0, therefore after.

selector = 3  Both codes have a 1 bit n therefore continue.

The function uses six data items:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>code1</td>
<td>Unsigned integer pointer</td>
<td>Formal parameter for first bit code.</td>
</tr>
<tr>
<td>code2</td>
<td>Unsigned integer pointer</td>
<td>Formal parameter for second bit code.</td>
</tr>
<tr>
<td>index</td>
<td>Integer</td>
<td>Local variable index with which to step through codes.</td>
</tr>
<tr>
<td>number1</td>
<td>Integer</td>
<td>Local variable in which to store elements of code1.</td>
</tr>
<tr>
<td>number2</td>
<td>Integer</td>
<td>Local variable in which to store elements of code2.</td>
</tr>
<tr>
<td>selector</td>
<td>Integer</td>
<td>Selector for case statement.</td>
</tr>
</tbody>
</table>

A detailed design for the beforeCheck procedure is presented in Figure 13.

5.8 SUBSET CHECK

The subsetCheck function, in form, is very similar to the equalityCheck function described above. However, comparisons are made using the “and” logical operator. Given two codes, (say) \( X = 0011 \)
and \( Y = 1011 \), and we wish to determine whether the \( X \) is a subset of \( Y \) we determine the result of a logical “and” operation between the two codes:

\[
0011 \land 1011 = 0011
\]

and if the result is equal to \( X \) (which it is in the above example) then \( X \subseteq Y \) is true. By reversing the arguments we can of course determine if one is a superset of the other.

As with the \textit{equalityCheck} function the function uses three data items:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>code1</td>
<td>Unsigned integer pointer</td>
<td>Formal parameter for first bit code.</td>
</tr>
<tr>
<td>code2</td>
<td>Unsigned integer pointer</td>
<td>Formal parameter for second bit code.</td>
</tr>
<tr>
<td>index</td>
<td>Integer</td>
<td>Local variable index with which to step through codes.</td>
</tr>
</tbody>
</table>

A detailed design for the \textit{subsetCheck} procedure is presented in Figure 14.

### 5.9 CREATE TREE STRUCTURE

The \textit{createStructure} function simply creates a structure for a node in the tree given an appropriate bit code. Assuming that the structure is successfully created the function then returns a pointer to that structure. The function includes the following data items:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>Unsigned Integer Pointer</td>
<td>Formal parameter giving the bit code for the current in the table</td>
</tr>
<tr>
<td>newPtr</td>
<td>PARTSUPPORTPTR</td>
<td>Local variable pointing to start of the newly created structure.</td>
</tr>
</tbody>
</table>

A detailed design for the \textit{createStructure} procedure is presented in Figure 15.

### 6 ANALYSIS

The storage required for the algorithm is as follows:

\[
K(12 + \text{number of array elements}) \text{ bytes}
\]

where \( K \) is the number of unique rows in the table, and \text{number elements} is calculated from:

\[
\text{number of array elements} = \frac{n}{32} + \frac{31}{32} \text{(rounded up to the nearest integer)}
\]
Figure 15: Nassi-Shneiderman chart for `createStructure` procedure

In the worst case $K$ will be equal to $m$ (the number of rows in the table), in the best case $K$ will equal 1 (all rows are the same). If we assume a normal distribution then we can be sure of duplication if:

$$m \Rightarrow 2^n$$

The number of support increments is dependent on the number of unique rows in the database $(K)$ and can be calculated using the identity:

$$support\ increments = m - k$$

The number of comparisons required is largely dependent on the number of duplicates in the table which in turn is dependent on the ratio of $M$ to $N$. Table 1 gives some statistical data comparing the linked list partial support algorithm described in [4] with that described here. The table assumes that $M$ is a constant 20000 rows. Both algorithms benefit from advantages gained by storing duplicates. However, this advantage is lost after $M$ overtakes $2^N$ (at about $N = 17$). When $M$ reaches 22 and beyond the number of duplicates and subsets is such that we can say that the number of Nodes is equivalent to the number of rows (i.e. $k = M$).

A graph indicating the storage requirements for the two algorithms using the data presented in Table 1 is given in Figure 16. From the graph (and Table 1) it can be seen that the tree algorithm requires more storage than the linked list. This is not surprising because each linked list node has one branch, while each tree node has two branches; consequently each tree node will require four more bytes of storage than the equivalent linked list node.

A second graph is presented in Figure 17 which compares the number of “tests” associated with each of the algorithms. From this graph we can see that the tree algorithm works well while the likelihood of subsets is high ($N \leq 15$ where $M = 20000$), but as this likelihood decreases the tree becomes “flat”, i.e. it degenerates into what is effectively a linked list. However, for each row, checks are still made for the possibility of adding child branches; i.e. three checks are made using the tree algorithm (i) an equals check, (ii) a lexicronically "before/after" check and (iii) a subset/superset check, corresponding to only one check in the linked list algorithm.

Overall the advantages offered by the tree algorithm is dependent on the “added value” offered by the tree organisation — the precise nature of this added value has yet to be determined. It may be that using a simple binary tree is the most appropriate middle ground to adopt for the two approaches.
<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Permutations</th>
<th>Number Duplicates</th>
<th>Number Subsets</th>
<th>Linked List Partial Support Alg.</th>
<th>Tree Partial Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Storage</td>
<td>Number Comparisons</td>
</tr>
<tr>
<td>20000</td>
<td>10</td>
<td>1024</td>
<td>18976</td>
<td>512</td>
<td>12288</td>
<td>9729164</td>
</tr>
<tr>
<td>20000</td>
<td>12</td>
<td>4096</td>
<td>15986</td>
<td>1986</td>
<td>48168</td>
<td>31591916</td>
</tr>
<tr>
<td>20000</td>
<td>14</td>
<td>16384</td>
<td>9696</td>
<td>4153</td>
<td>123648</td>
<td>62966092</td>
</tr>
<tr>
<td>20000</td>
<td>16</td>
<td>65536</td>
<td>6210</td>
<td>3288</td>
<td>165480</td>
<td>84694435</td>
</tr>
<tr>
<td>20000</td>
<td>18</td>
<td>262144</td>
<td>1469</td>
<td>2741</td>
<td>222372</td>
<td>94845802</td>
</tr>
<tr>
<td>20000</td>
<td>20</td>
<td>1048576</td>
<td>522</td>
<td>1291</td>
<td>233736</td>
<td>97839324</td>
</tr>
<tr>
<td>20000</td>
<td>22</td>
<td>4194304</td>
<td>144</td>
<td>540</td>
<td>238272</td>
<td>99720345</td>
</tr>
<tr>
<td>20000</td>
<td>24</td>
<td>1677216</td>
<td>71</td>
<td>193</td>
<td>239148</td>
<td>100254196</td>
</tr>
<tr>
<td>20000</td>
<td>26</td>
<td>67108864</td>
<td>13</td>
<td>74</td>
<td>239844</td>
<td>100223287</td>
</tr>
<tr>
<td>20000</td>
<td>28</td>
<td>268435456</td>
<td>5</td>
<td>27</td>
<td>239940</td>
<td>100113417</td>
</tr>
<tr>
<td>20000</td>
<td>31</td>
<td>2147483648</td>
<td>0</td>
<td>4</td>
<td>240000</td>
<td>99820147</td>
</tr>
<tr>
<td>20000</td>
<td>33</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>320000</td>
<td>100728688</td>
</tr>
<tr>
<td>20000</td>
<td>63</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>320000</td>
<td>100648950</td>
</tr>
<tr>
<td>20000</td>
<td>65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>400000</td>
<td>99830442</td>
</tr>
<tr>
<td>20000</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>400000</td>
<td>100207943</td>
</tr>
<tr>
<td>20000</td>
<td>95</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>400000</td>
<td>99546592</td>
</tr>
<tr>
<td>20000</td>
<td>97</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>480000</td>
<td>100583022</td>
</tr>
</tbody>
</table>

Table 1: Comparison of “linked-list” partial support algorithm and “tree” partial support algorithm

Figure 16: Comparison of storage requirements for linked list and tree algorithms ($M = 20000$)
Figure 17: Comparison of number of “test” for linked list and tree algorithms ($M = 20000$)

References


APPENDIX A — IMPLEMENTATION

/ * -------- GLOBAL VARIABLES -------- */

/* Example test file: */
00100011001111100101000010 11001010001100010110010100 00010000010101110010001 101101101011100100010010 10100001101100100011000 0101011110110101111101 0000100100100000100010 010000101011000101111 0010010001011010000010 110011111011001001011111

/* Linked list storage structure to store partial supports. Structure includes: */
1) code = Array of unsigned integers describing the row.
2) support = Partial support for the rows.
3) childPtr = Pointer to child structure.
4) siblingPtr = Pointer to sibling structure.

defined in support.h. */
PARTSUPPORTPTR startPtr = NULL;

/* -------- FUNCTION PROTOTYPES -------- */
void partialSupport(char *);
void processInput(FILE *);
void addSupport(int, PARTSUPPORTPTR, PARTSUPPORTPTR, unsigned int *);
void addSupport2(int, PARTSUPPORTPTR, PARTSUPPORTPTR);
int checkSiblingBranch(PARTSUPPORTPTR, int *, PARTSUPPORTPTR *);
int checkCodes(unsigned int *, unsigned int *);
int equalityCheck(unsigned int *, unsigned int *);
int subsetCheck(unsigned int *, unsigned int *);
int beforeCheck(unsigned int *, unsigned int *);
unsigned int* generateBitPattern(char *);
PARTSUPPORTPTR createStructure(unsigned int *code);

/* Next three prototypes for testing only. */

void outputPartSupportTree(void);
void outputPartSupport1(PARTSUPPORTPTR, char *, int);
void outputPartSupport2(PARTSUPPORTPTR);

/* -------------------------- */
/* */
/* PARTIAL SUPPORT */
/* */
/* -------------------------- */

void partialSupport(char *filename)
{
  FILE *in_file;

  /* Open file and process. */
  if ((in_file=fopen(filename,"r")) == NULL) {
    printf("Unable to open '"%s' for reading\n",filename);
    exit(1);
  }
  processInput(in_file);

  /* Close file. */
  fclose(in_file);
}

/* ------ PROCESS INPUT ------ */

/* Note: fgets(stringname,n,filename) - function to read from a file until either n-1 characters or a newline character is encountered, and store the characters in the given stringname. */

void processInput(FILE *in_file)
{
  char input[512];  /* Largest row has 512 columns! */
  unsigned int *code = NULL;
  PARTSUPPORTPTR oldPtr = NULL;

  /* Get first row, calculate number of columns. */
  fgets(input,512,in_file);
  totalCols = strlen(input)-1;
  arraySize = (int) (totalCols/32.0 + 31.0/32.0);

  /* process first line in table. */
  code = generateBitPattern(input);
  startPtr = createStructure(code);

  /* process rest of table */
  while (fgets(input,512,in_file) != '\0') {
    code = generateBitPattern(input);
    addSupport(0,startPtr,oldPtr,code);
    totalRows++;
  }
}

/* ------ ADD SUPPORT ------ */
/* Add current row to linked list structure as follows: */

code = 1, Increment support
code = 2, Row before and subset of current node (parent)
code = 3, Row before and not subset of current node (elder sibling)
code = 4, Row after and superset of current node (child)
code = 5, Row before and not superset of current node (younger sibling)

Codes generated by call the checkCode function. */

void addSupport(int flag, PARTSUPPORTPTR linkPtr, PARTSUPPORTPTR oldPtr, 
    unsigned int *rowCode)
{
    PARTSUPPORTPTR newPtr = NULL, newSiblingPtr = NULL;

    /* Process comparison code */
    switch (checkCodes(rowCode,linkPtr->code)) {
        case 1: /* Rule: Same */
            linkPtr->support = linkPtr->support+1;
            return;
        case 2: /* Rule 2: Before and subset */
            newPtr = createStructure(rowCode);
            newPtr->childPtr = linkPtr;
            addSupport2(flag,newPtr,oldPtr);
            if (checkSiblingBranch(linkPtr,rowCode,&newSiblingPtr) == 1)
                newPtr->siblingPtr = newSiblingPtr;
            return;
        case 3: /* Rule 3: Before and not subset */
            newPtr = createStructure(rowCode);
            newPtr->siblingPtr = linkPtr;
            addSupport2(flag,newPtr,oldPtr);
            return;
        case 4: /* Rule 4: After and superset */
            if (linkPtr->childPtr == NULL) {
                newPtr = createStructure(rowCode);
                linkPtr->childPtr = newPtr;
            }
            else addSupport(1,linkPtr->childPtr,linkPtr,rowCode);
            return;
        case 5: /* Rule 5: After and not superset */
            if (linkPtr->siblingPtr == NULL) {
                newPtr = createStructure(rowCode);
                linkPtr->siblingPtr = newPtr;
            }
            else addSupport(2,linkPtr->siblingPtr,linkPtr,rowCode);
            return;
        default: /* Default: Error */
            printf("ERROR: Unidentified bit comparison code in addSupport\n");
            exit(1);
    }
}

/* ------ ADD SUPPORT 2 ------ */

/* Add new node where "before and subset". The flag argument indicates which 
type of branch is currently under consideration: = = root, 1 = child, 
2 = sibling. */

void addSupport2(int flag, PARTSUPPORTPTR newPtr, PARTSUPPORTPTR oldPtr) 
{
    switch (flag) {
        case 0: 
            startPtr = newPtr;
            return;
        case 1: 
            }
oldPtr->childPtr = newPtr;
return;

}  
case 2: 
oldPtr->siblingPtr = newPtr;
return;

default:
    printf("ERROR: Unidentified flag in addSupport\n");
}

/* ------ CHECK SIBLING BRANCH ------ */

/* Check sibling branch to determine whether it is superset of the current row or not. If so return 0 (do nothing), and 1 otherwise (in which case some of the sibling nodes must be moved up. There are four possibilities:

1. Sibling branch is MT (return 0).
2. No nodes in sibling branch are superset of row there for move them all up (return 1).
3. All nodes in sibling branch are superset of row there for move none of them (return 0).
4. Some nodes in sibling branch are superset of row others are not therefore move those that are not (return 1).

int checkSiblingBranch(PARTSUPP0RTPTR linkPtr, int *rowCode, PARTSUPP0RTPTR *newSiblingPtr)
{
PARTSUPP0RTPTR localLinkPtr = NULL, markerPtr = NULL;
/* Check for empty sibling branch, if so return 0 (do nothing). */
if (linkPtr->siblingPtr == NULL) return(0);
/* Check if first node in sibling branch points to a superset of row. If not move the entire branch up, return 1. */
if (subsetCheck(rowCode,linkPtr->siblingPtr->code) == 0) {
    *newSiblingPtr = linkPtr->siblingPtr;
    linkPtr->siblingPtr = NULL;
    return(1);
}
/* Check rest. Branch starts of with superset of row (which are OK where they are), but we must find the point where this is no longer the case, i.e. the part of the branch that needs to be moved up (if any). */
markerPtr = linkPtr->siblingPtr;
localLinkPtr = linkPtr->siblingPtr->siblingPtr;
while (localLinkPtr != NULL) {
    if (subsetCheck(rowCode,localLinkPtr->code) == 0) {
        *newSiblingPtr = localLinkPtr;
        markerPtr->siblingPtr = NULL;
        return(1);
    }
    markerPtr = localLinkPtr;
    localLinkPtr = localLinkPtr->siblingPtr;
}
/* No nodes to be moved up. Return 0 (do nothing). */
return(0);
}

/* ----------------------------- */
/* */
/* CODE COMPARISON Routines */
/* */
/* ------------------------------ */

/* ------ CHECK CODE ------ */

/* Check row code with current node code: 1 = same, 2 = parent, 3 = before, 4 = child, and 5 = after. */

int checkCodes(unsigned int rowCode, unsigned int currentCode)
{
    /* Check if same */
    if (equalityCheck(rowCode, currentCode) == 1) return(1);
    /* Check whether before or after and subset/superset. */
    if (beforeCheck(rowCode, currentCode) == 1) {
        if (subsetCheck(rowCode, currentCode) == 1) return(2);
        else return(3);
    }
    if (subsetCheck(currentCode, rowCode) == 1) return(4);
    return(5);
}

/* ------ EQUALITY CHECK ------ */

/* Return 1 if row code is equal to current code, and 0 otherwise. */

int equalityCheck(unsigned int *code1, unsigned int *code2)
{
    int index = 0;
    while (index < arraySize) {
        if (code1[index] != code2[index]) return(0);
        else index++;
    }
    /* Codes the same. */
    return(1);
}

/* ------ BEFORE CHECK ------ */

/* Returns 1 if first codes is less than (before) second code and 0 otherwise. Note that before here is not numerical but lexical, i.e. AB is before B although the code associated with AB (3) is numerically after the code associated with B (2). The selector used in the case statement is made up of the remainder of two divisions by 2 (the second multiplied by 2). Thus:

0 = Both codes have a 0 bit n therefore continue.
1 = Bit n in first number is 1 and in second number is 0, therefore before.
2 = Bit n in second number is 1 and in first number is 0, therefore after.
3 = Both codes have a 1 bit n therefore continue. */

int beforeCheck(unsigned int *code1, unsigned int *code2)
{
    int index = 0, selector;
    unsigned int number1, number2;
    while (index < arraySize) {
        number1 = code1[index];
        number2 = code2[index];
        while (number1 != 0 && number2 != 0) {
            selector = (number1 % 2) + (number2 % 2)*2;
            switch(selector) {
                case 0:
                    break;
case 1:
    return(1);
case 2:
    return(0);
default:
    }
    number1 = number1/2;
    number2 = number2/2;
    }
    if (code1[index] >= code2[index]) return(0);
    else index++;
}

/* Codes the same. */
return(1);
}

/* ------ SUBSET CHECK ------ */
/* Return 1 if row code is a subset of current code, and 0 otherwise. */
int subsetCheck(unsigned int *code1, unsigned int *code2)
{
    int index = 0;
    while (index < arraySize) {
        if (((code1[index] & code2[index]) != code1[index]) return(0);
        else index++;
    }

    /* Row code is a subset of the given code. */
    return(1);
}

/* -------------------------- */
/* */
/* UTILITIES */
/* */
/* -------------------------- */

/* ------ GENERATE BIT PATTERN ------ */
/* Generate an integer bit pattern according to 0 and 1 in table row (least significant bit to the left). e.g. 1 1 1 0 0 = 7 */
unsigned int* generateBitPattern(char *input)
{
    int count1 = 0, count2 = 0, stringIndex=0, arrayIndex=0;
    unsigned int *arrayPtr;
    unsigned int number, increment;

    /* Create space (malloc returns NULL if no space available). */
    arrayPtr = (unsigned int *) malloc(sizeof(unsigned int)*arraySize);

    /* Add values to array. */
    while (arrayIndex < arraySize) {
        number = 0;
        increment = 1;
        count1 = 1;
        while (count1 < 32 && count2 < totalCols) {
            if (input[stringIndex] == '1') number = number + increment;
            increment = increment*2;
            stringIndex++;
        }
        arrayPtr[arrayIndex] = number;
        arrayIndex++;
    }
}

25
count1++;
count2++; }
arrayPtr[arrayIndex] = number;
arrayIndex++;
}
return(arrayPtr);
}
/* --------------------------------------------------------------- */
/* */
/* "LINKED LIST OF PARTIAL SUPPORT" UTILITIES */
/* */
/* --------------------------------------------------------------- */

/* ------ CREATE TREE STRUCTURE------ */
/* Create a set structure of the type PARTIAL SUPPORT given the code for the */
item. */
PARTSUPPORTPTR createStructure(unsigned int *code)
{
PARTSUPPORTPTR newPtr = NULL;
if ((newPtr = (PARTSUPPORTPTR)(malloc(sizeof(PARTSUPPORT)))) == NULL) {
printf("Insufficient storage space\n");
exit(1);
}
newPtr->code = code;
newPtr->support = 1;
newPtr->childPtr = NULL;
newPtr->siblingPtr = NULL;
return(newPtr);
}
/* --------------------------------------------------------------- */
/* */
/* OUTPUT */
/* */
/* --------------------------------------------------------------- */

/* ------ OUTPUT PARTIAL SUPPORT STRUCTURE LINKED LIST ------ */
/* If more than 8 columns it becomes difficult to output node identifiers. For */
8 columns, in the worst case, we need a character string of 256 elements. */
void outputPartSupportTree(void)
{
if (totalCols <= 8) outputPartSupport1(startPtr,"start",1);
else outputPartSupport2(startPtr);
}
/* ------ OUTPUT PARTIAL SUPPORT 1 ------ */
void outputPartSupport1(PARTSUPPORTPTR linkPtr,char *node, int counter)
{
int index=0;
char newXNode[256];
if (linkPtr != NULL) {
  if (strcmp(node,"start") == 0) sprintf(newXNode,"%d",counter);
  else sprintf(newXNode,"%s.%d",node,counter);
  printf("%s",newXNode);
  while (index < arraySize) {
    index++;
    printf("\n");
    }
printf("%d ",linkPtr->code[index]);
index++;
}
printf("support = %d\n",linkPtr->support);
outputPartSupport1(linkPtr->childPtr,newNode,1);
counter = counter+1;
outputPartSupport1(linkPtr->siblingPtr,node,counter);
}

/* ----- OUTPUT PARTIAL SUPPORT 2 ------ */
/* No node identifiers. */

void outputPartSupport2(PARTSUPPORTPTR linkPtr)
{
int index=0;
if (linkPtr != NULL) {
    while (index < arraySize) {
        printf("%d ",linkPtr->code[index]);
index++;
    }
    printf("support = %d\n",linkPtr->support);
    outputPartSupport2(linkPtr->childPtr);
    outputPartSupport2(linkPtr->siblingPtr);
}
}
APPENDIX B - TESTING

Here follows a set of “basket analysis” tables designed to execute all aspects of the code wherever a new node is to be inserted into the tree structure.

**Rule 2:** Before and subset — new root node (start) to tree.

- No siblings to check. Input (add $A$ to \{AB, ABC\}):
  
  110
  111
  100

  Output:
  
  (1) 1 support = 1
  (1.1) 3 support = 1
  (1.1.1) 7 support = 1

- All siblings to be moved up to become siblings of new node (root). Input add $A$ to \{AB, C, ABC, BC, CD\}:
  
  1100
  0100
  0010
  1110
  0110
  0011
  1000

  Output:
  
  (1) 1 support = 1
  (1.1) 3 support = 1
  (1.1.1) 7 support = 1
  (2) 2 support = 1
  (2.1) 6 support = 1
  (3) 4 support = 1
  (3.1) 12 support = 1

- No siblings to be moved up. Input (add $A$ to \{AB, AC, AD, ABC, ACD, ADE\}):
  
  11000
  10100
  10010
  11100
  10110
  10011
  10000

  Output:
  
  (1) 1 support = 1
  (1.1) 3 support = 1
  (1.1.1) 7 support = 1
  (2) 2 support = 1
  (2.1) 6 support = 1
  (3) 4 support = 1
  (3.1) 12 support = 1

- Some siblings to be moved up. Input (add $A$ to \{AB, AC, B, ABC, ACD, BC\}):
  
  1100
  1010
  0100
  1110
  1011
  0110
  1000

  Output:
  
  (1) 1 support = 1
  (1.1) 3 support = 1
  (1.1.1) 7 support = 1
  (1.2) 5 support = 1
  (1.2.1) 13 support = 1
  (2) 2 support = 1
  (2.1) 6 support = 1

**Rule 2:** Before and subset — new node on child branch.

- No siblings to check. Input (add $AB$ to \{A, ABC, ABCD\}):
  
  1000
  1110
  1111
  1100

  Output:
  
  (1) 1 support = 1
  (1.1) 3 support = 1
  (1.1.1) 7 support = 1
  (1.2) 5 support = 1
  (1.2.1) 13 support = 1

- All siblings to be moved up to become siblings of new node (root). Input (add $AB$ to \{A, ABC, AC, AD, ABCD, ACD, ADE\}):
  
  10000
  11100
  10100
  10010
  11110
  10110
  10011
  11000

  Output:
  
  (1) 1 support = 1
  (1.1) 3 support = 1
  (1.1.1) 7 support = 1
  (1.1.1.1) 15 support = 1

- No siblings to be moved up. Input (add $AB$ to \{A, ABC, ABD, ABE, ABCD, ABD E, ABE F\}):
  
  100000
  111000
  110100
  110010
  111100
  110110
  110011
  110000

  Output:
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.1.1.1) 15 support = 1
(1.1.2) 11 support = 1
(1.1.2.1) 27 support = 1
(1.1.3) 19 support = 1
(1.1.3.1) 51 support = 1

- Some siblings to be moved up. Input (add $AB$ to $\{A, ABC, ABD, AD, A BCD, ABDE, ADE\}$):

```
10000
11100
11010
10010
11110
11011
10011
11000
```

Output:
```
(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(1.1.1.1) 15 support = 1
(1.1.2) 11 support = 1
(1.1.2.1) 27 support = 1
(1.2) 9 support = 1
(1.2.1) 25 support = 1
```

**Rule 2:** Before and subset — new node on sibling branch.

- No siblings to check. Input (add $B$ to $\{A, BC, BCD\}$):

```
1000
0110
0111
0100
```

Output:
```
(1) 1 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(2.1.1) 14 support = 1
```

- All siblings to be moved up to become siblings of new node (root). Input (add $B$ to $\{A, BC, BD, BE, BCD, BDE, BEF\}$):

```
100000
011000
010100
010010
011100
010110
010011
010000
```

Output:
```
(1) 1 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(2.1.1) 14 support = 1
(2.2) 10 support = 1
(2.2.1) 26 support = 1
(3) 8 support = 1
(3.1) 24 support = 1
```

- No siblings to be moved up. Input (add $B$ to $\{A, BC, C, D, BCD, CD, DE\}$):

```
10000
01100
00100
00010
00110
00011
00100
```

Output:
```
(1) 1 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(2.1.1) 14 support = 1
(3) 4 support = 1
(3.1) 12 support = 1
(4) 8 support = 1
(4.1) 24 support = 1
```

- Some siblings to be moved up. Input (add $B$ to $\{A, BC, BD, D, BCD, BDE, DE\}$):

```
10000
01100
01000
00000
01110
01011
00011
01000
```

Output:
```
(1) 1 support = 1
(2) 2 support = 1
(2.1) 6 support = 1
(2.1.1) 14 support = 1
(2.2) 10 support = 1
(2.2.1) 26 support = 1
(3) 8 support = 1
(3.1) 24 support = 1
```

**Rule 3:** Before and not subset — new root node (start) to tree. Input (add $A$ to $\{B\}$):

```
01
10
```

Output:
```
(1) 1 support = 1
(2) 2 support = 1
```

**Rule 3:** Before and not subset — new node on child branch. Input (add $A$ to $\{AB, BC\}$):

```
110
011
100
```

Output:
```
(1) 1 support = 1
(1.1) 3 support = 1
(2) 6 support = 1
```

**Rule 3:** Before and subset — new node on sibling branch. Input (add $A$ to $\{B, C\}$):
010
001
100

Output:

(1) 1 support = 1
(2) 2 support = 1
(3) 4 support = 1

Rule 4: After and superset — empty child branch. Input (add C to \{A\}):  
100
001

Output:

(1) 1 support = 1
(2) 4 support = 1

Rule 4: After and superset — proceed down child branch. Input (add C to \{A, B, AB\}):  
100
010
110
001

Output:

(1) 1 support = 1
(1.1) 3 support = 1
(2) 2 support = 1

Rule 5: After and not superset — empty sibling branch. Input (add ABC to \{A\}):  
100
111

Output:

(1) 1 support = 1
(1.1) 7 support = 1

Rule 5: After and not superset — proceed down sibling branch. Input (add ABC to \{A, B, AB\}):  
100
010
110
111

Output:

(1) 1 support = 1
(1.1) 3 support = 1
(1.1.1) 7 support = 1
(2) 2 support = 1

30