A BRUTE FORCE ALGORITHM FOR "BASKET ANALYSIS" (GCL-BF1)

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

A brute force algorithm (GCL-BF1) is described to identify "interesting data sets" given a standard "basket analysis" database where the columns represent "products"; and the rows "transactions", i.e. groups of products purchased in a single (supermarket) transaction. A typical database of this form is given in Table 1.

The objective of all *basket analysis* algorithms is to firstly identify sets of two or more attributes/products for which there is sufficient support. Some authors refer to such sets as "large" sets. The support for a set is calculated using the identity:

$$\frac{number\ of\ rows\ in\ table\ where\ \alpha\ appears}{Total\ number\ of\ rows\ in\ table} * 100$$

where α represents a set of attributes/products. A set has sufficient support if it is featured in a given percentage of rows in the database, typically a threshold of 20% is used. Thus in the example database given in Table 1 the set $\{A, B\}$ appears in 2 out the 10 given rows, i.e. 20% of the total number of rows include both attributes A and B and thus we say that the set $\{A, B\}$ has a support of 20%. assuming a support threshold of 20% we can now say that the set $\{A, B\}$ has sufficient support.

Given any large set (group of attributes for which there is sufficient support) "association rules" can be derived from the set. An association rule is a relation of the form $\alpha \Rightarrow \beta$ where α and β represent groups of one or more attributes such that the union of α and β represents a large attribute set. Thus, given the database in Table 1 two example candidate association rules are:

$$A \rightarrow B$$
$$B \rightarrow A$$

These can be read as "if A is true then B is true" and "if B is true then A is true" (or "A implies B" and "B implies A"). However, these rule are only candidate association rules. For a rule to be considered valid it must have an appropriate confidence level. Confidence in a rule is again expressed as a percentage; if this percentage is over a certain level (typically 80%) we say that we have sufficient confidence in the rule and therefore the rule is deemed to be valid. The confidence for a rule is calculated using the identity:

$$\frac{support\ for\ A \Rightarrow B}{Support\ for\ A} * 100$$

Thus given the rule $A \rightarrow B$ the confidence in the rule will be equivalent to $\frac{2}{5} * 100 = 40\%$, i.e. less than the suggested 80% threshold, hence the rule has not got sufficient confidence for it to be considered a valid rule. Similarly the confidence in the rule $B \rightarrow A$ will be $\frac{2}{4} * 100 = 50\%$, this is also less than the 80% threshold.

Transaction Number	Attribute identifiers A	B	C	D	E	F	G	H	I	J
1	0	0	1	0	0	0	1	1	0	0
2	1	1	1	1	1	0	0	1	0	1
3	0	0	0	1	1	0	1	1	0	0
4	1	0	0	1	0	0	0	0	0	0
5	0	1	0	1	1	1	0	0	1	0
6	1	0	0	0	0	1	0	0	0	0
7	0	0	1	0	1	0	1	1	1	0
8	0	1	0	1	0	0	0	1	1	0
9	1	1	0	1	1	0	1	0	1	1
10	1	0	0	1	0	0	0	1	0	0

Table 1: Example basket analysis database table

The algorithm described here calculates the support for all possible combinations of one or more attributes and determines whether this support is sufficient assuming a 20% threshold. To determine association rules we are of course only interested in sets of two or more attributes, however to determine rule confidences we will also need to know the support for individual elements. The algorithm also does this. The algorithm does not go on to determine "interesting" association rules. The aim is to establish a bench mark with which other, more computationally efficient, algorithms can be compared. To this end the most naive (and therefore computationally costly) approach possible has been adopted, i.e. a "brute force" approach. With this aim in mind the algorithm is presented in significant detail so that all implementational aspects are covered, as well as the more high level operational details.

This document is organised as follows. Section 2 describes the software purely in terms of input and output including the data structures used. A general overview of the algorithm is then presented in section 3. In section 4 a much more detailed description is given — this section can be omitted by readers who do not wish to be concerned with the implementational detail of the algorithm. An analysis of the algorithm is then presented in section 5, followed by the code in the Appendix.

2 INPUT AND OUTPUT

The brute force program is executed as follows:

```
./bruteForceAlg <INPUTFILE> > <OUTPUTFILE>
```

where the input is in the form of a file of the following form:

```
Number of rows = 10, number of columns = 10
A B C D E F G H I J
==================
0 0 1 0 0 0 1 1 0 0
1 1 1 1 1 0 0 1 0 1
0 0 0 1 1 0 1 1 0 0
1 0 0 1 0 0 0 0 0 0
0 1 0 1 1 1 0 0 1 0
1 0 0 0 0 1 0 0 0 0
0 0 1 0 1 0 1 1 1 0
0 1 0 1 0 0 0 1 1 0
1 1 0 1 1 0 1 0 1 1
```

```
1 0 0 1 0 0 0 1 0 0
```

The above file shows details of 10 attributes (columns) and 10 individual transactions (rows). The nature of the software is such that the maximum possible number of columns/attributes is limited to 26. There is no limit on the number of rows. A "test case" generator program has also been produced to randomly generate basket analysis tables of the above form given a desired number of rows and columns.

From the above it can be seen that the file contains no transaction numbers as these can be deduced from the line number where required. Note also that the first three lines in the input file can safely be ignored. It should also be noted that each row can be interpreted as a binary number which has a decimal equivalent. This feature is used extensively in the algorithm described in that such decimal equivalents are used as codes to describe (a) table rows and (b) each possible attribute set (large or otherwise) that we are interested in. For example, assuming that the least significant bit is represented by column A the first row in Table 1 will be equivalent to the decimal number 196. The remainder of the rows will then have the codes; 671, 216, 9, 314, 33, 468, 394, 859 and 137 respectively.

Output from the algorithm is as follows:

```
Minimum support = 2
A (code = 1) support = 5 (Interesting set)
B (code = 2) support = 4 (Interesting set)
AB (code = 3) support = 2 (Interesting set)
C (code = 4) support = 3 (Interesting set)
AC (code = 5) support = 1
BC (code = 6) support = 1
ABC (code = 7) support = 1
D (code = 8) support = 7 (Interesting set)
AD (code = 9) support = 4 (Interesting set)
BD (code = 10) support = 4 (Interesting set)
ABD (code = 11) support = 2 (Interesting set)
CD (code = 12) support = 1
ACD (code = 13) support = 1
BCD (code = 14) support = 1
ABCD (code = 15) support = 1
....
```

There are four elements to this output (other than the initial `Minimum support = 2` statement):

- A possible subset of the attribute/product set under consideration described as a character string of identifying letters, e.g. A, B, AB, Etc.
- The integer code derived from the bit pattern associated with the attribute; hence A = 1 (0000000001), B = 2 (0000000010), AB = 3 (0000000011) and so on.
- The support for the attribute set in question.
- Whether the set is an "interesting set" or not, i.e. whether the set has a minimum support equivalent to or above a given default threshold (20% in the above case).

The number of possible attribute sets of one element or more that can be described by a database (i.e. the cardinality of the `power set` associated with a given set of attributes) is dependent on the number of columns in the database and can be calculated using the identity:

$$2^n - 1$$

where n is the number of columns. Thus in the above example there will be 1023 possible alternatives, and consequently there will be a 1023 lines of output (one for each possible combination), plus a line stating the minimum support ($Minimumsupport = 2$).

The output (other than the initial `Minimum support = 2` statement) is stored in a linked list of structures defined as follows:

```
typedef struct set {
char name[32];
int code;
int support;
struct set *next;
} SET, *SETPTR;
```

where the first three fields are equivalent to the first three items listed above (whether the support is sufficient or not, is not stored but determined on output).

3 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 a linked list of structures — one structure for each possible combination of one or more attributes.
- Read in the rest of the table starting with the second transaction and continuing until the end of the table is encountered. On each iteration analyse the row and for each possible combination of one or more attributes in the row and increment the support count in the linked list of structures where appropriate. To assist in this use is made of the integer codes described by each row and the Boolean operators available in C so that comparisons can be made using a binary 'and' operation.
- On completion of this iterative process the linked list structure will contain details of all the support that the given table provides for every possible combination of the attributes described by the table columns.
- Output the linked list structure by looping through it and on each iteration determine whether there is adequate support for the attribute set in question according to the given support threshold.

4 DETAILED DESIGN OF ALGORITHM

The algorithm is implemented in C using seven functions/procedures as follows:

1. main
2. processInput
3. generateBitPattern
4. addToSupport
5. createLinkedList
6. createSetStruct

7. outputSetStructLinkedList

The main procedure is used to open and close the input file and instigate the processing. The processing is actually achieved by the *processInput* procedure. The *generateBitPattern* function is used to generate an integer code from a given row in the table. The *addToSupport* procedure is used to update the linked list structure given a particular integer code. The *createLinkedList* procedure and the *createSetStruct* function combine to create the desired linked list structure. Finally the *outputSetStructLinkedList* is used to output the final result.

The code includes four global variables:

NAME	TYPE	DESCRIPTION
startPtr	SETPTR	Global pointer to start of linked list of structures of the form described in section 2.
endPtr	SETPTR	Global pointer to end of linked list of structures of the form described in section 2.
totalRows	Integer	Global variable in which the total number of rows in the input table is stored.
supportThreshold	Integer	Global variable which holds the desired support threshold (20% by default).

In the following sub-sections the detailed design of each of the above functions is presented. The designs are given in the form of Nassi-Shneiderman charts. This part of the document can be omitted by readers who are not concerned with the implementational details of the algorithm.

4.1 MAIN

The main top level procedure takes as input a command line argument which should be the name of an input file. The function then checks for such an argument and if found opens the named input file for "reading". Should no such argument be presented, or if the named file cannot be open (for example because no such file exists), an error message is produced. Once the input file has been opened the *processInput* function is called to carry out the necessary calculations on completion of which the input file is "closed". Finally the *outputSetStructLinkedList* function is called to output the linked list of structures in which the results are contained.

The top level procedure includes only a single data item:

NAME	TYPE	DESCRIPTION
in_file	FILE pointer	Local variable which is assigned the input file name.

The complete design for the *main* procedure is presented in Figure 1.

4.2 PROCESS INPUT

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

- Read and "throw away" first three lines of input (these contain only the headings for the table).
- Read fourth line and from this line determine the number of columns in the table; this is equivalent to the line length divided by two (to allow for spacing). Each input line is temperarily stored as a string — remember that a C string is always terminated by a "null terminator" ('\0'); therefore a string is always one longer than it appears to be.

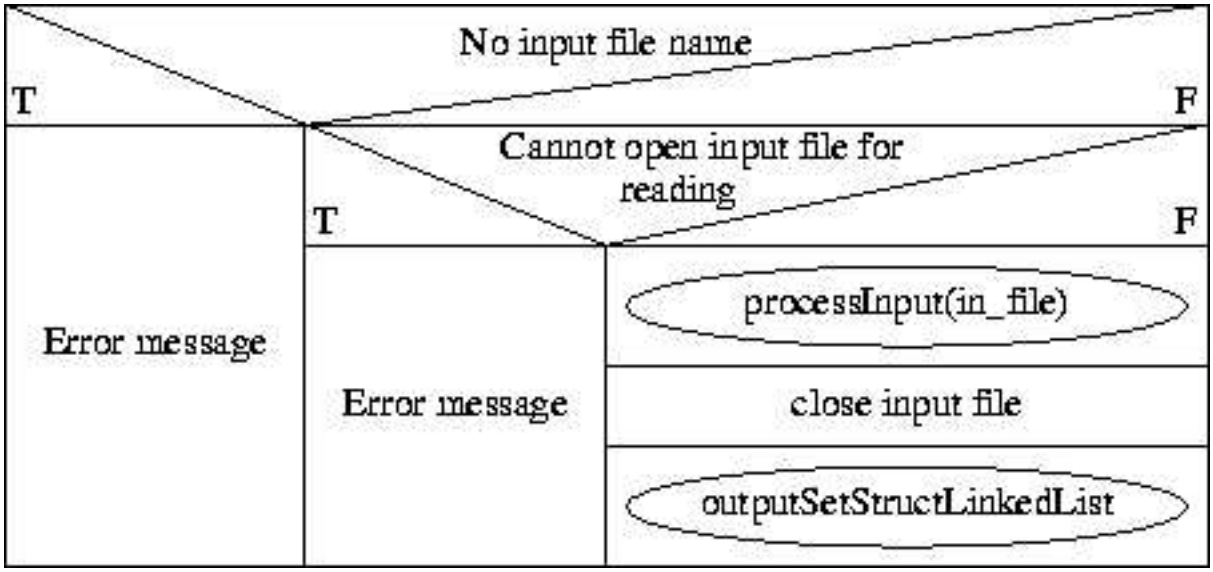

Figure 1: Nassi-Shneiderman chart for *main* procedure

- Call the *createLinkedList* function to produce the linked list of structures for each possible set of one or more attributes according to the number of columns in the table.
- Call the function *generateBitPattern* to convert the current input string into an integer (the string length minus 2 is used for this purpose).
- Call the *addToSupport* function to update the support values for the sets in the linked list of sets according to the nature of the current input string.
- Continue the process of (a) reading rows, (b) generating bit patterns and (c) updating the linked list structures until the end of the file is encountered. At the same time maintain a running total (stored in the global variable *totalRows*) of the number of rows found in the table.

The function includes the following data items:

NAME	TYPE	DESCRIPTION
in_file	FILE pointer	Formal parameter, pointer to current location in input file.
input	Array of 64 characters)	Local variable in which current input line is stored as string (64 is the maximum size of the string).
stringLength	Integer	local variable in which to store the actual length of the input string.
cols	Integer	Local variable containing the number of columns in the input table.
bitPattern	Integer	Local variable describing the bit pattern associated with the binary digits of the input string.

A detailed design for the *processInput* function is presented in Figure 2.

4.3 GENERATE BIT PATTERN

The *generateBitPattern* function returns an integer generated according to the input pattern at the current row in the table (assigned to the input string variable). The left most digit is assumed

to be the least significant digit. Thus, for example, given a bit pattern 11100 this will be equivalent to the integer 7. The function uses a loop construct to step through the input string (a character array) and build up the final number to be returned. Note that the index is incremented in steps of 2 so as to skip over the white space known to separate digits in the input string. The function includes the following data items:

NAME	TYPE	DESCRIPTION
input	Character pointer	Formal parameter pointing to the start of the current input string.
length	Integer	Formal parameter containing the expected length of the input string (minus the null terminator).
index	Integer	Local variable that acts both as the Loop counter and the index for input character array (string).
number	Integer	Local variable to be eventually returned from the function and which will hold decimal number represented by the current input string. Initialised with the value 0.
increment	Integer	The amount that the current value in *number* must be incremented by to take account of the current digit in the input string. This data item is initialised with the value 1.

A detailed design for the *generateBitPattern* function is presented in Figure 3.

Thus given the input string 11100 the procedure will operate as follows:

```
First digit is a '1' thus number = number+increment = 0+1 = 1
increment = increment*2 = 1*2 = 2
Second digit is a '1' thus number = number+increment = 1+2 = 3
increment = increment*2 = 2*2 = 4
Third  digit is a '1' thus number = number+increment = 3+4 = 7
increment = increment*2 = 4*2 = 8
Fourth  digit is a '0'
increment = increment*2 = 8*2 = 16
Fifth  digit is a '0'
increment = increment*2 = 8*2 = 16

index > length exit loop and return number
```

4.4 ADD SUPPORT

The *addToSupport* procedure updates the support totals contained in the linked list of structures. The support total is incremented by 1 if the attribute grouping represented by the binary bit pattern associated with the code is a subset (or equal to) the grouping represented by the bit pattern (code) of the current input line (row). This is determined using a binary 'and' operation between the integer bit code for the set and the bit code for the row. If the result of the 'and' is equivalent to the bit code the support is incremented by 1. The procedure includes the following data items:

NAME	TYPE	DESCRIPTION
bitPattern	Integer	Formal parameter containing the bit pattern associated with the current row in the table.
linkPtr	SETPTR	Local variable instantiated with the global variable *startPtr* which points to the start of the linked list of structures described in section 2.

A detailed design for the *addToSupport* function is presented in Figure 4.

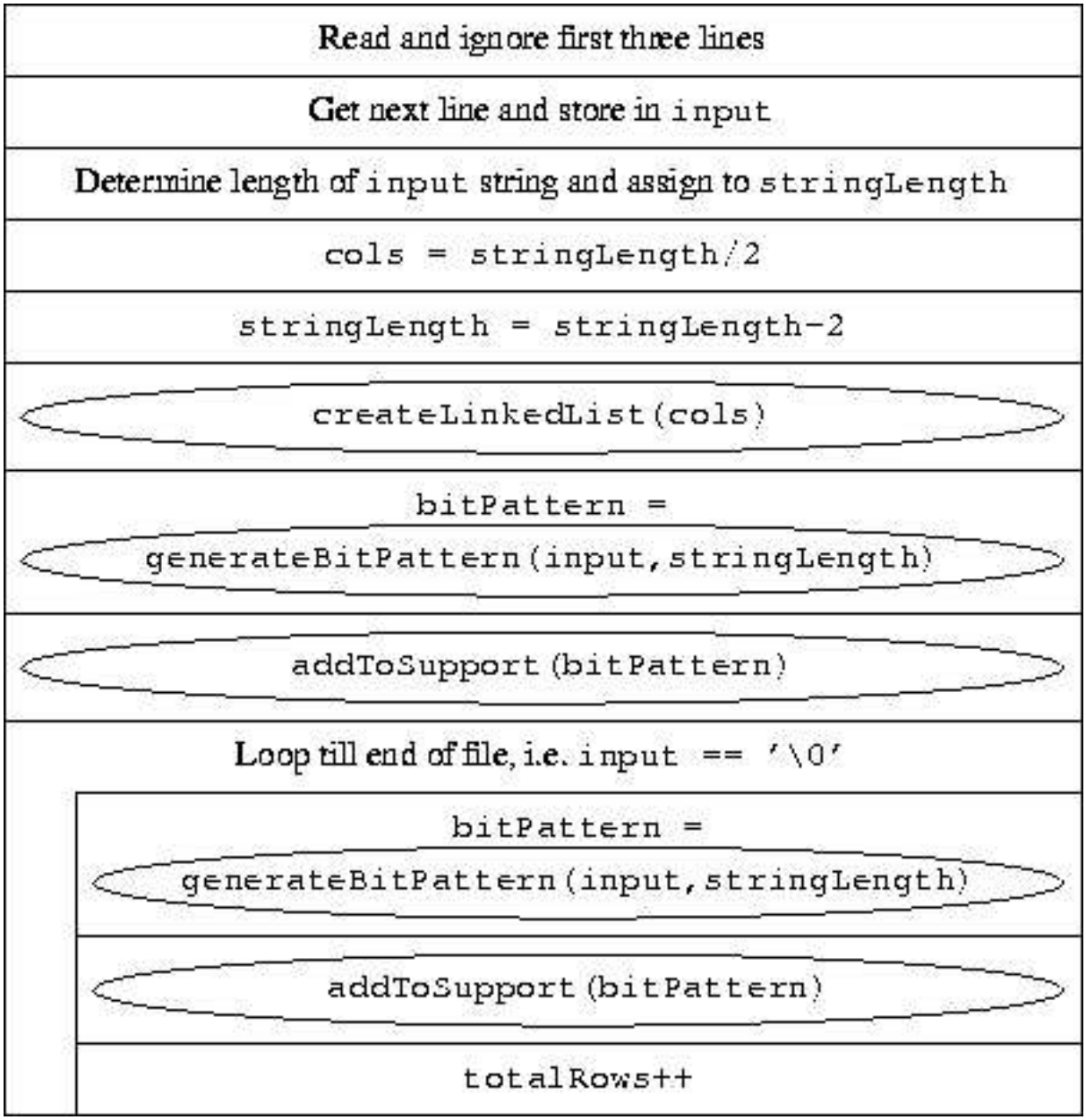

Figure 2: Nassi-Shneiderman chart for *processInput* function

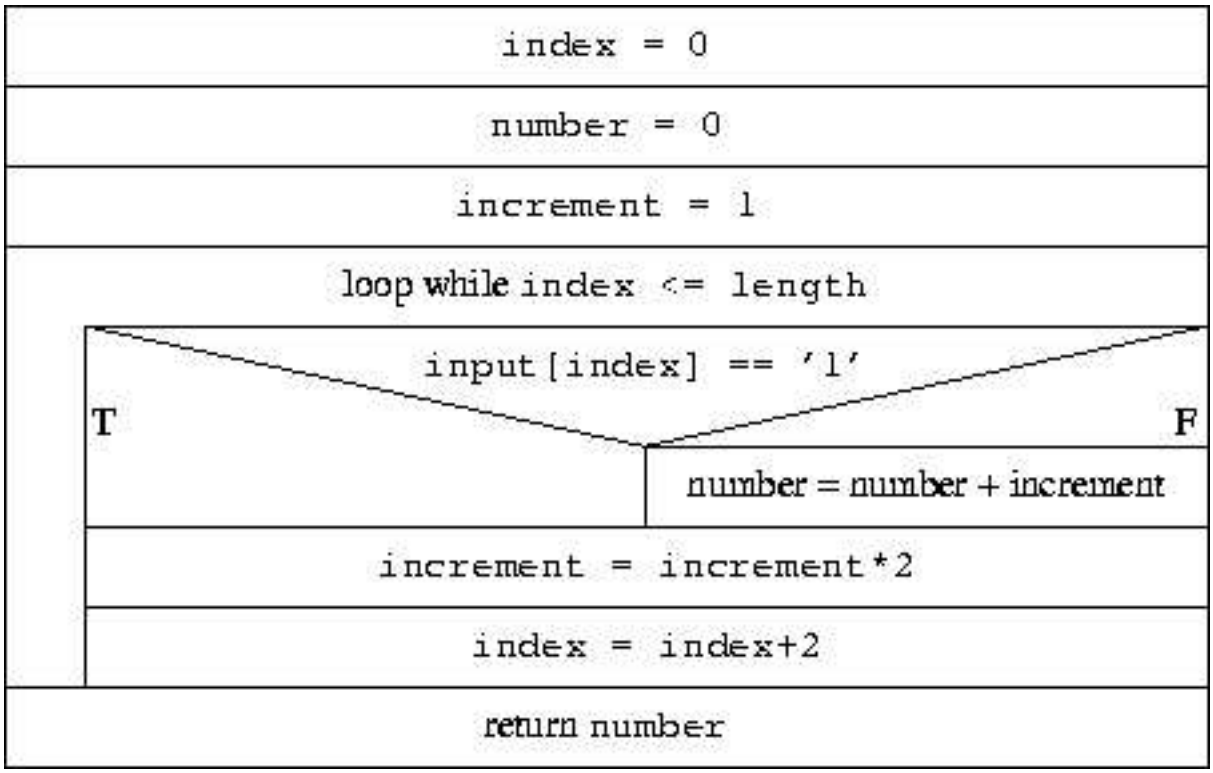

Figure 3: Nassi-Shneiderman chart for *generateBitPattern* function

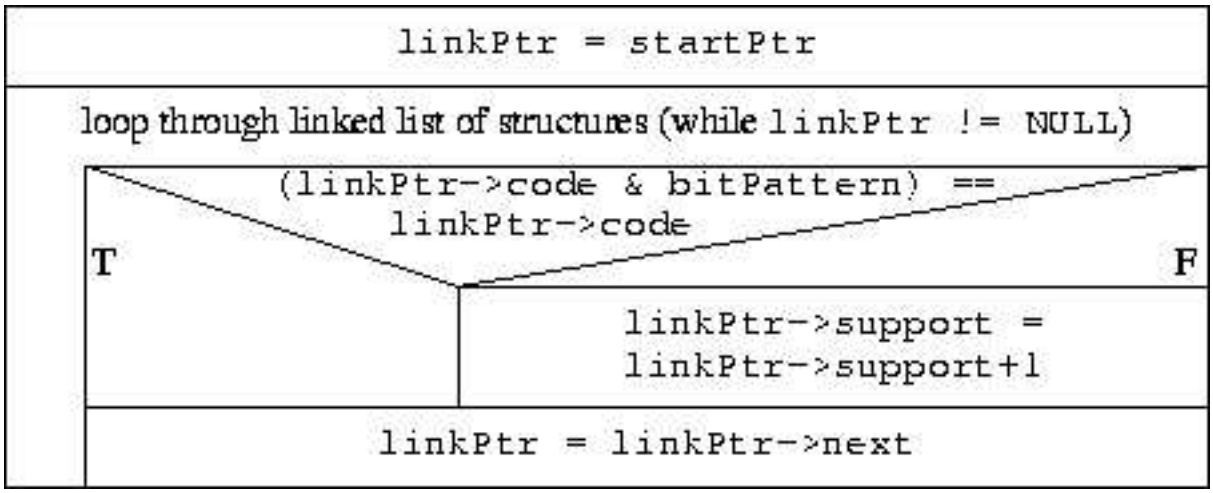

Figure 4: Nassi-Shneiderman chart for *addToSupport* function

4.5 CREATE LINKED LIST

The *createLinkedList* procedure produced a linked list of structures describing the `power set` for the set of attributes in a basket analysis database. This power set then represents all the candidate sets from which association rules may be derived provided that there is sufficient support. The linked list is produced by first creating a single structure to represent the column/attribute A. This initial "linked list" is then appended to by determining and adding new sections of the list in an iterative manner according to the number of columns. Thus the next attribute to considered is the attribute B. This is then used to create the next section of the list by pairing B with the linked list as constructed so far, i.e. the list A. The new section of the linked list will thus comprise AB. These three parts (A and B and AB) are then brought together to create a new "sofar" linked list — A, B, AB. The next column represents the attribute C . When coupled with the list to date we then get AC, BC, ABC. Combining these latest three parts (A, B, AB and C and AC, BC, ABC) we get A, B, AB, C, AC, BC, ABC. And so on until all columns/attributes have been considered.

The procedure has a sizable number of data items as follows:

NAME	TYPE	DESCRIPTION
cols	Integer	Formal parameter giving the number of columns in the table (this in turn defines the number of attributes).
code	Integer	Local variable. The decimal number described by the bit pattern.
count	Integer	Local variable. The loop counter to step through the columns.
newCode	Integer	Local variable. Holds the next code in the sequence calculated from the previous code.
newString	2 Element Character array	Local variable to hold the current attribute/ column letter under consideration (the second element is for the C null terminator).
name	32 Element Character array	Local variable describing an attribute set in string form.
newPtr	SETPTR	Local variable to hold address of start of current newly created structure.
linkPtr	SETPTR	Local variable that act as a loop control variable to step through any linked list developed to date.
tempPtr	SETPTR	Local variable to temporarily hold a pointer to a newly created structure.
newStartPtr	SETPTR	Local variable to point at the start of a new section of the linked list.
markerPtr	SETPTR	Local variable. Points at the current location in a new section of the linked list.

The procedure operates as follows:

- Create the first structure in the linked list (i.e. the start of the linked list) using a call to the *createSetStruct* function which returns a pointer to the start of the newly created structure. Cause the global pointers *startPtr* and *endPtr* to point at this structure.
- Loop for a number of iterations equal to the number of columns (the formal parameter *cols* for the function) minus one because column "A" has already been considered.
- On each iteration:

- Assign the value NULL to *newSetPtr* and assign to *newString* the attribute/column letter under consideration. The "A" column has already been considered so on the first iteration this will assigned the string "B".
- Determine the code for this letter (i.e. the code for B will be $\ldots 00010 = 2$).
- Create a new set structure for this single letter attribute through a call to *createSetStruct*. Assign the newly returned pointer to *newPtr*.
- Now loop through the linked list built up so far (the start of which is pointed to by *startPtr*) and on each iteration (i.e. for each record contained in the linked list built up to date) perform the following:
  * Copy the alphabetic identification for the current structure to the string *name* and catenate this with *newString* (which will contain the letter representing the current column) so that the result is stored in *name*.
  * Calculate a new code (*newCode*) by adding the code for the current column (*code*) to the existing code for the attribute set described by the current structure.
  * Create a new set structure for the attribute set now described by *newString* and *newCode* through a call to *createSetStruct*. Assign the newly returned pointer to *tempPtr*.
  * If there are no structure in the new section of the linked list ($newStartPtr == NULL$) assign *tempPtr* to *newStartPtr*; otherwise $markerPtr->next = tempPtr$.
  * Update *markerPtr* so that it points at the current structure in the new section of the linked list.
- Put the three parts together to determine the new linked list sofar (the variable *endPtr* is used here).

A detailed design for the *createLinkedList* function is presented in Figure 5.

4.6 CREATE SET STRUCTURE

The *createSetStruct* function is a "house keeping" function linked to the foregoing. It is used to create and instantiate a structure of the form defined in section 2 and return a pointer to this structure. The function includes the following data items:

NAME	TYPE	DESCRIPTION
name	Character pointer	Formal parameter that points to start of a character array containing the sequence of letters identifying an attribute set.
code	Integer	Formal parameter. An integer describing the bit pattern associated with the attribute set.
newPtr	SETPTR	Local variable instantiated with the start address on newly created structure which is then returned by the function.

A Nassi-Shneiderman chart for the *createSetStruct* function is given in Figure 6.

4.7 OUTPUT SET STRUCTURE LINKED LIST

The *outputSetStructLinkedList* procedure is used solely to output the final result. It makes use of the global variables *startPtr*, *totalRows* and *supportThreshold* described in section 4. Output is generated simply by iterating through the linked list. On each iteration the contents of the structure is output to the screen. In addition a calculation is made to determine whether the current set is an "interesting set" or not, by comparing the calculated support with the desired minimum support threshold. If the support is sufficient this fact is also output. The procedure uses the following data items:

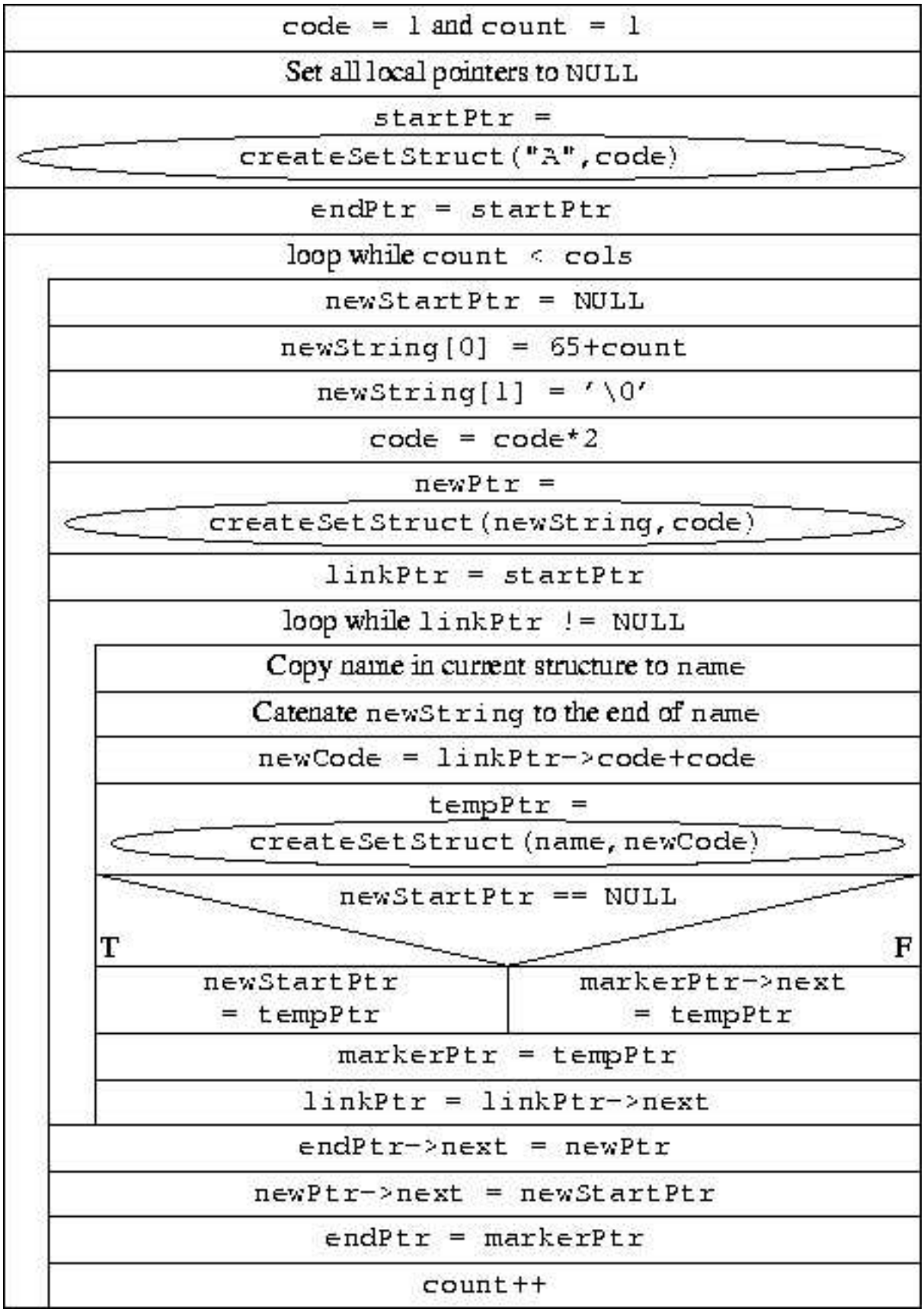

Figure 5: Nassi-Shneiderman chart for *createLinkedList* function

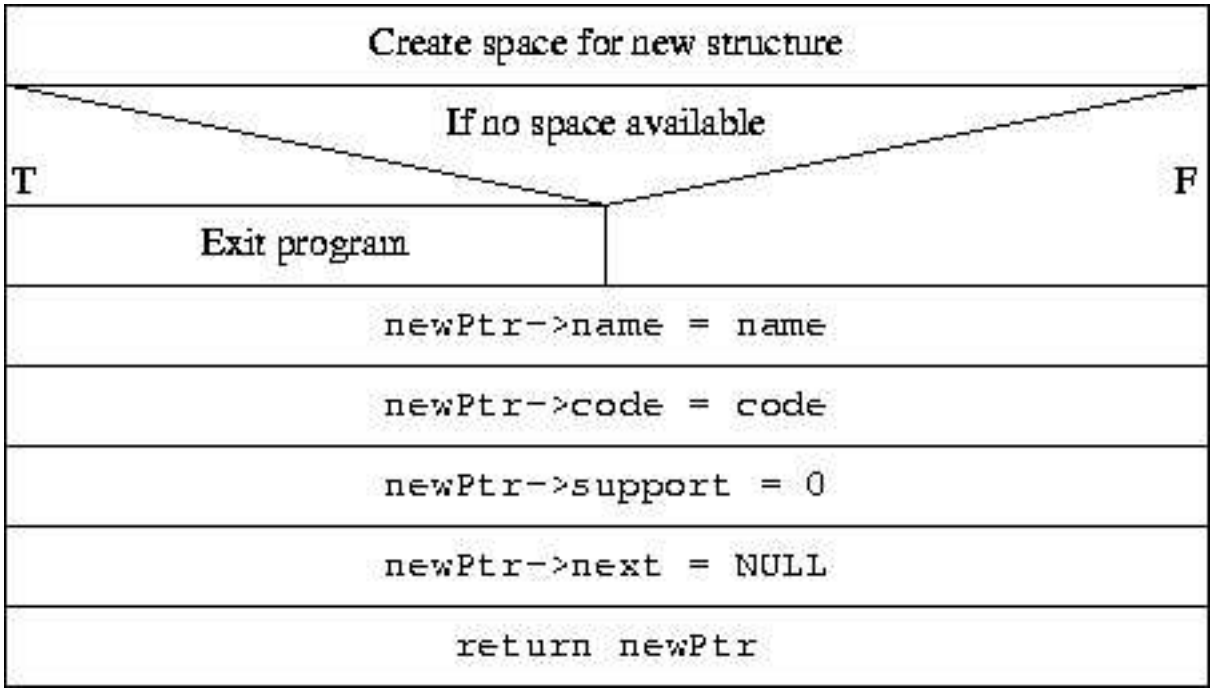

Figure 6: Nassi-Shneiderman chart for *createSetStruct* function

NAME	TYPE	DESCRIPTION
linkPtr	SETPTR	Local variable instantiated with the start address and used as the control variable for the while loop.
minSupport	Integer	Local variable holding the minimum support.

A Nassi-Shneiderman chart for the procedure is given in Figure 7.

5 ANALYSIS

The algorithm is classified as a "single pass" algorithm as only one pass is made through the database table (unlike many other association rule algorithms which make more than one pass). Tests show that the algorithm is quite capable of working with tables comprising 12 columns and 100,000 rows or 16 columns and 10,000 rows in an acceptable time (less than 5 minute to produce a result).

The first significant restriction on the algorithm is the storage space used by the linked list of structures. Each individual structure requires 44 bytes of storage. The total storage required for any given table is thus given by:

$$storage = 44\,(2^n - 1)$$

where n is the number of columns.

The storage requirements can be significantly reduced if the string equivalent of the attribute set is not stored but only the code. Instead, for output, the string can be determined from the code. This would produce a saving of 32 bits per attribute set (73%). It is also worth noting that 9% of the total storage is also used for housekeeping purposes only (i.e. storing the "next" pointer). If the string equivalent is not included in the structure the generation of the linked list (the *createLinkedList* procedure) can be greatly simplified. All that will be required is to produce a linked list $2^n - 1$ structures coded from 1 to $2^n - 1$.

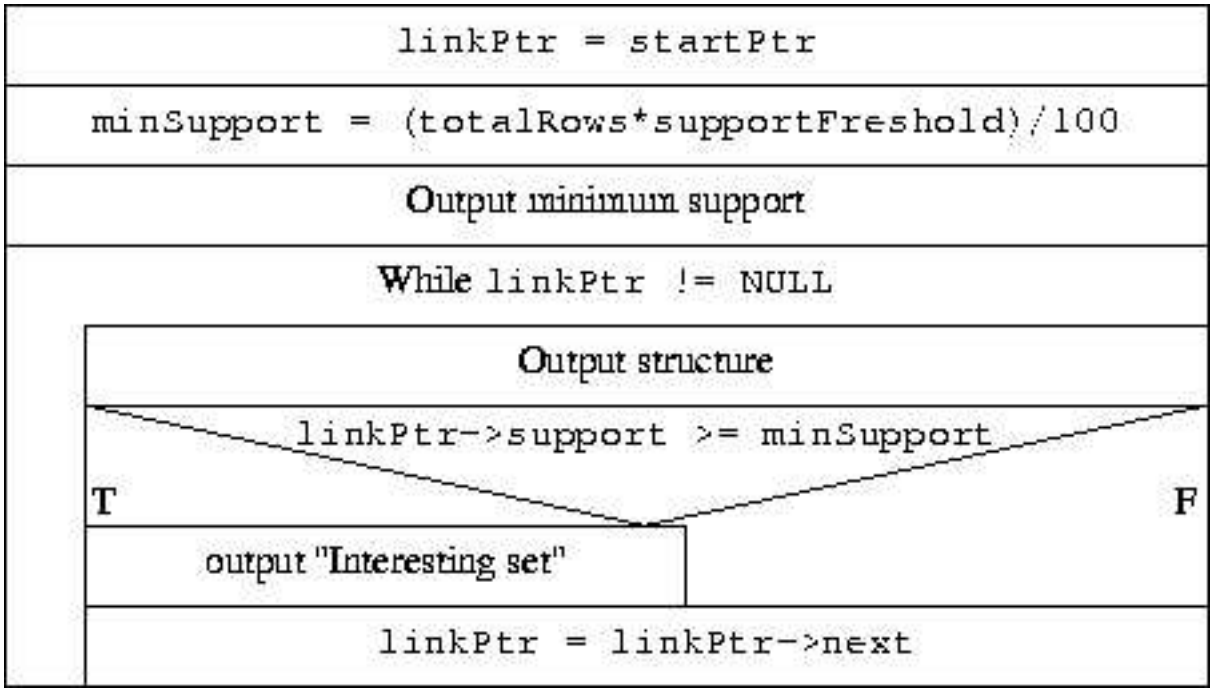

Figure 7: Nassi-Shneiderman chart for *outputSetStructLinkedList* function

The second disadvantage of the brute force approach is that it calculates the support for every possible combination of attributes. Some analysis of a test set comprising 10 columns and 10,000 rows (using a "scarcity" constant 0f 50%) indicates that out of a total of 1023 possible interesting sets only 55 are found to have sufficient support on completion of the algorithm (i.e. only 5% of the sets actually prove to be interesting). Consequently we can conclude that a lot of unnecessary calculation is undertaken. One possible approach is to first calculate all pairs for which there is sufficient support, and then use the knowledge that these are the only attributes that can form part of a "higher level" attribute set. It is also worth noting that in the above experiment there was no combination of attributes that did not appear in the input table at least once.

The number of comparisons that are carried out (in the above) to determine whether to increment the support or not is equivalent to:

$$num\ of\ calculations = m\,(2^n - 1)$$

where m is the number of columns. Thus in this naive approach every row is compared to every column. Any reduction in the number of data sets stored will serve to reduce this figure. There may also be approaches whereby it is possible to avoid the need for this exhaustive comparison. For example if we can determine the attribute sets represented by a particular row and then "find" these sets in the linked list either by (a) making use of the code to (in some sense) "index" into the linked list, or (b) to conduct a binary search to find particular attribute groupings.

The use of the integer codes to represent each row in the table seems useful. However, a standard unsigned integer comprises 32 bits; so using this approach limits the size of the table to 32 columns. Given that, in tests, the brute force algorithm is not capable of handling anything beyond 16 columns this is probably not an immediate concern[1]. However, an alternative coding system may comprise an unsigned integer array where each element represents a block of 32 columns. Thus a table comprising 320 columns would require a 10 element unsigned integer array encoding.

[1] Further the input file is currently limited to 26 columns

The generation of codes from (a) rows input from the table, and (b) to create the initial linked list is carried out in a reasonable computationally efficient manner and (it is felt) unlikely to be improved upon. The number of code generations is equivalent to:

$$num\ code\ generations = 2^n + m - 1$$

one for each possible set combination and one for each row. The use of these integer codes is (to the best of the authors knowledge) unique to the above algorithm and does not feature in existing published alternative algorithms directed at basket analysis.

Overall, although there is much that can be improved upon in the brute force algorithm described, it is felt that these deficiencies render the algorithm to be ideally suited to the role of a bench mark to which future algorithms can be measured/compared.

APPENDIX

```
/* ------------------------------------------------------------------------ */
/*                                                                          */
/*                    B R U T E   F O R C E   A L G O R I T H M             */
/*                                  Frans Coenen                            */
/*                                   2 July 1998                            */
/*                                                                          */
/* ------------------------------------------------------------------------ */

/* Programme to generate interesting sets from a binary database table by "brute
force". */

/* to compile "gcc -o bruteForceAlg bruteForceAlg.c" */

/* ------ INCLUDE STATEMENTS ------ */

#include <stdlib.h>
#include <stdio.h>

/* Example test file:

Number of rows = 10, number of columns = 26
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
==================================================
0 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0
1 1 1 0 1 1 0 1 1 1 0 1 1 0 0 0 0 0 1 1 1 1 1 0 1 1
1 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1 0 1 1
1 0 0 1 0 0 1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 0 1 0 1 1
0 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0 0 1
1 0 1 0 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 1 1 1 0 0
1 1 1 0 0 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 1 1 1
1 0 1 0 0 1 1 1 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 1 1 0
0 0 1 0 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 0 1 1 0 0 1 */

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

/*  Linked list storage structure for sets of attributes. Structure includes:
```

```
1) ta1  = Start address for sequence
2) ta2  = End address for sequence            */

typedef struct set {
char name[32];
int code;
int support;
struct set *next;
} SET, *SETPTR;

SETPTR startPtr = NULL, endPtr = NULL;

/* totalRows       = Total number of rows in table.
   supportThreshold = Support threshold, e.g. 20 (20%). */

int totalRows = 1, supportThreshold = 20;

/* ------ FUNCTION PROTOTYPES ------ */

void processInput(FILE *);
int generateBitPattern(char *, int);
void addToSupport(int), createLinkedList(int);
SETPTR createSetStruct(char *, int);
void outputSetStructLinkedList(void);

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

void main(int argc, char *argv[])
{
FILE *in_file;

/* Check Input. */

if (argc < 2 ) {
printf("INPUT ERROR - no file name.\n");
exit(1);
}

/* Open file and process. */

if ((in_file=fopen(argv[1],"r")) == NULL) {
printf("Unable to open '%s' for reading\n",argv[1]);
exit(1);
}
processInput(in_file);

/* Close file. */

fclose(in_file);
```

```
/* Output result. */

outputSetStructLinkedList();
}

/* ------ 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[64];         /* Maximum input string is 26+25+1 = 52. */
int stringLength, cols, bitPattern;

/* Miss first Three lines. */

fgets(input,64,in_file);
fgets(input,64,in_file);
fgets(input,64,in_file);

/* Get next line, calculate number of columns and produce linked list of
candidate sets.  */

fgets(input,64,in_file);
stringLength = strlen(input);
cols = stringLength/2;
stringLength = stringLength-2;
createLinkedList(cols);

/* process first line. */

bitPattern = generateBitPattern(input,stringLength);
addToSupport(bitPattern);

/* process rest of table */

while (fgets(input,40,in_file) != '\0') {
bitPattern = generateBitPattern(input,stringLength);
addToSupport(bitPattern);
totalRows++;
}
}

/* ------ 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 */

int generateBitPattern(char *input, int length)
{
int index = 0, number = 0, increment = 1;
```

```
while (index <= length) {
if (input[index] == '1') number = number + increment;
increment = increment*2;
index = index+2;
}

return(number);
}

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

/* For each candidate set stored in the linked list determine if current line
supports set. This is achieved using a logical and between the integer bit code
for the set and the bit pattern for the row. If the result of the "and" is
equivalent to the bit code then increment support by 1. */

void addToSupport(int bitPattern)
{
SETPTR linkPtr = startPtr;

while (linkPtr != NULL) {
if ((linkPtr->code & bitPattern) == linkPtr->code)
linkPtr->support = linkPtr->support+1;
linkPtr = linkPtr->next;
}
}

/* ---------------------------------------------------------- */
/*                                                            */
/*          "LINKED LIST OF CANDIDATE SETS" UTILITIES         */
/*                                                            */
/* ---------------------------------------------------------- */

/* ------ CREATE LINKED LIST ------ */

/* Produce the power set for the set of attributes in the database table. This
power set then represents all the candidate sets from which association rules
may be derived provided that there is sufficient support. */

void createLinkedList(int cols)
{
int code = 1, count = 1, newCode;
char newString[2], name[32];
SETPTR newPtr = NULL, linkPtr = NULL, tempPtr = NULL;
SETPTR newStartPtr = NULL, markerPtr = NULL;

/* Create start structure */

startPtr = createSetStruct("A",code);
endPtr = startPtr;

/* Create remainder */

while (count < cols) {
```

```
newStartPtr = NULL;
newString[0] = 65+count;
newString[1] = '\0';
code = code*2;
newPtr = createSetStruct(newString,code);
linkPtr = startPtr;
while (linkPtr != NULL) {
strcpy(name,linkPtr->name);
strcat(name,newString);
newCode = linkPtr->code+code;
tempPtr = createSetStruct(name,newCode);
if (newStartPtr == NULL) newStartPtr = tempPtr;
else markerPtr->next = tempPtr;
markerPtr = tempPtr;
linkPtr = linkPtr->next;
}
endPtr->next = newPtr;
newPtr->next = newStartPtr;
endPtr = markerPtr;
count++;
}
}

/* ------ CREATE SET STRUCTURE ------ */

/* Create a single set structure with given arguments, and return a pointer to
it. */

SETPTR createSetStruct(char *name, int code)
{
SETPTR newPtr = NULL;

if ((newPtr = (SETPTR)(malloc(sizeof(SET))))==NULL) {
printf("Insufficient storage space\n");
exit(1);
}

strcpy(newPtr->name,name);
newPtr->code = code;
newPtr->support = 0;
newPtr->next = NULL;

return(newPtr);
}

/* ------------------------ */
/*                          */
/*          OUTPUT          */
/*                          */
/* ------------------------ */

/* ------ OUTPUT SET STRUCTURE LINKED LIST ------ */

void outputSetStructLinkedList(void)
```

```
{
SETPTR linkPtr = startPtr;
int minSupport = (totalRows*supportThreshold)/100;

printf("Minimum support = %d\n",minSupport);
while (linkPtr != NULL) {
printf("%s (code = %d) support = %d ",linkPtr->name,linkPtr->code,
linkPtr->support);
if (linkPtr->support >= minSupport) printf("(Interesting set)");
printf("\n");
linkPtr = linkPtr->next;
}
}
```