A SECOND BRUTE FORCE ALGORITHM FOR "BASKET ANALYSIS" (GCL-BF2)

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 second version (GCL-BF2) of the brute force algorithm (GCL-BF1) described in ([1] is described. This version offers the following advantage over the previous version:

- The amount of temporary storage is greatly reduced.
- It is not necessary to compare bit patterns (possible attribute groupings).

The first advantage has been achieved by dispensing with the cumbersome linked list data structure in which results were stored, and replacing this with a 1-D array. Again much use is made of the fact that each possible permutation of attributes is defined by a Binary sequence that translates into an integer. This integer is then used as the index for the array, i.e. each index of the array represents a particular configurations and the content the support. As with the previous brute-force system this approach does limit the system to a maximum of 32 attributes (note that in practice the Brute Force 1 algorithm could not reasonably handle more than 14 attributes and the algorithm described here no more than 20; thus this is not considered to represent a significant problem at present).

The second advantage is a direct result of the above. Using the 1-D array data structure it is no longer necessary to carry out comparisons between lines in the table and elements of the data structure to identify attributes sets whose support requires incrementation. Instead we determine all the possible attribute combinations represented by a table row, each identified by a bit code, and use this code to index into the database and update the support. Of course this saving is offset by the work done to determine all possible permutations in a row, however the work done to determine these permutations is considered to be less than that required previously to carry out all the necessary comparisons.

2 INPUT AND OUTPUT

The brute force 2 program is executed in a similar manner to its predecessor:

```
./bruteForceAlg2 <INPUTFILE>  <OUTPUTFILE>
```

where the input (slightly modified so that there is no preamble or spacing between elements, as is the case of brute force 1) is in the form of a file thus:

```
0010001100
1111100101
0001101100
1001000000
0101110010
1000010000
```

```
0010101110
0101000110
1101101011
1001000100
```

Output from the algorithm has also been modified so that only the “large sets” (interesting sets) are produced:

```
Minimum support = 2
A (code = 1) support = 5
B (code = 2) support = 4
AB (code = 3) support = 2
C (code = 4) support = 3
D (code = 8) support = 7
AD (code = 9) support = 4
BD (code = 10) support = 4
ABD (code = 11) support = 2
....
```

The output (other than the initial `Minimum support = 2` statement) is generated from the 1-D array in which all supports are stored.

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 the 1-D array data structure — one element for each possible combination of one or more attributes. Initilaise each element with the value 0.
- Process the first line by determining every possible pernmutation of attributes describing each permutation using an bit code — use this code to index into the data structure and update the support.
- 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 determine the codes for all the possible attribute combinations that can be generated and use this code to index into the data structure so as to increment the appropriate supports.
- On completion of this iterative process the array will contain details of the support for every possible combination of the attributes described by the table columns.
- Output those attribute sets for which there is sufficient support by looping through the array and determining (on each itteration) whether there is adequate support according to the given support threshold.

4 DETAILED DESIGN OF ALGORITHM

The design of the brute force 2 algorithm is very similar to the brute force 1 version. The algorithm is again implemented in C using the following functions/procedures:

1. main
2. processInput

3. addSupport
4. createArray
5. outputResult
6. outputResult

The main procedure is used to open and close the input file and instigate the processing. The processing is actually carried out by the *processInput* procedure. The *addSupport* procedure is used to update the 1-D array data structure given a particular bit code. The *createArray* procedure creates space for, and instantiates, a 1-D array of the appropriate dimensions. The *outputResult* procedure is used to identify and output the large attribute sets. The *deconstructArrayIndex* is called from the *outputResult* procedure to determine the attribute set represented by a particular index (an integer encoding) associated with a sufficient level of support.

The code includes four global variables:

NAME	TYPE	DESCRIPTION
$arrayPtr$	Integer Pointer	Global pointer to start of arraye.
$arraySize$	Integer	Global variable indicating total number of elements in array.
$totalRows$	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 procedures is presented. 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.

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 procedure 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* procedure is called to carry out the necessary calculations, on completion of which the input file is "closed". Finally the *outputResult* procedure is called to output the array 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 procedure operates as follows:

- Read first line in table and from this line determine the number of columns in the table; this is equivalent to the line length minus one. Each input line is temporarily stored as a string — remember that a C string is always terminated by a "null terminator" ('\0'); therefore a string is always one element longer than it appears to be.

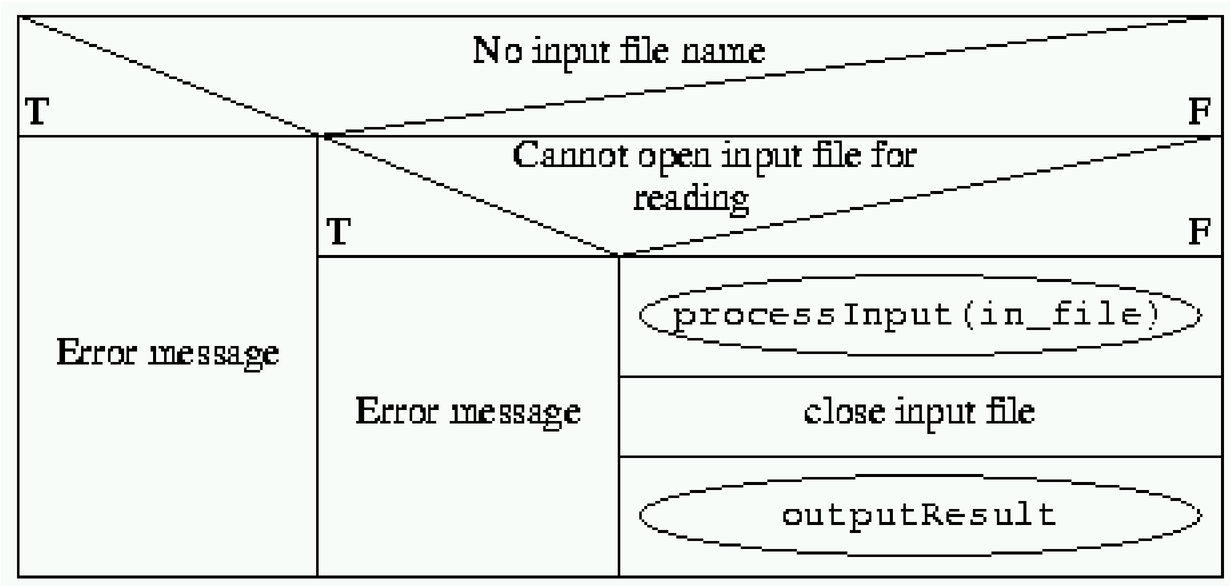

Figure 1: Nassi-Shneiderman chart for *main* procedure

- Call the *createArray* procedure to produce the 1-D array data structures for each possible set combination according to the number of columns in the table.
- Continue the process of (a) reading rows and (b) updating the support 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 procedure 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 (32 is the maximum size of the string).
cols	Integer	Local variable containing the number of columns in the input table.
startSetCode	Integer	Local constant describing the bit pattern associated with no attribute permutation.
startIndex	Integer	Local constant describing the first index in the array
startDigit	Integer	Local constant describing the bit pattern for the first permutation to be considered.

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

4.3 ADD SUPPORT

The *addSupport* procedure is a recursive procedure used to update the support totals contained in the array structures given an input line from the table. This is achieved by determining the bit codes for each permutation of attributes contained in the current input line and using these bit codes to index into the array and increment the support by 1 on each occasion. The bit patterns are calculated and the incrementation undertaken in a recursive manner. The procedure is entered with:

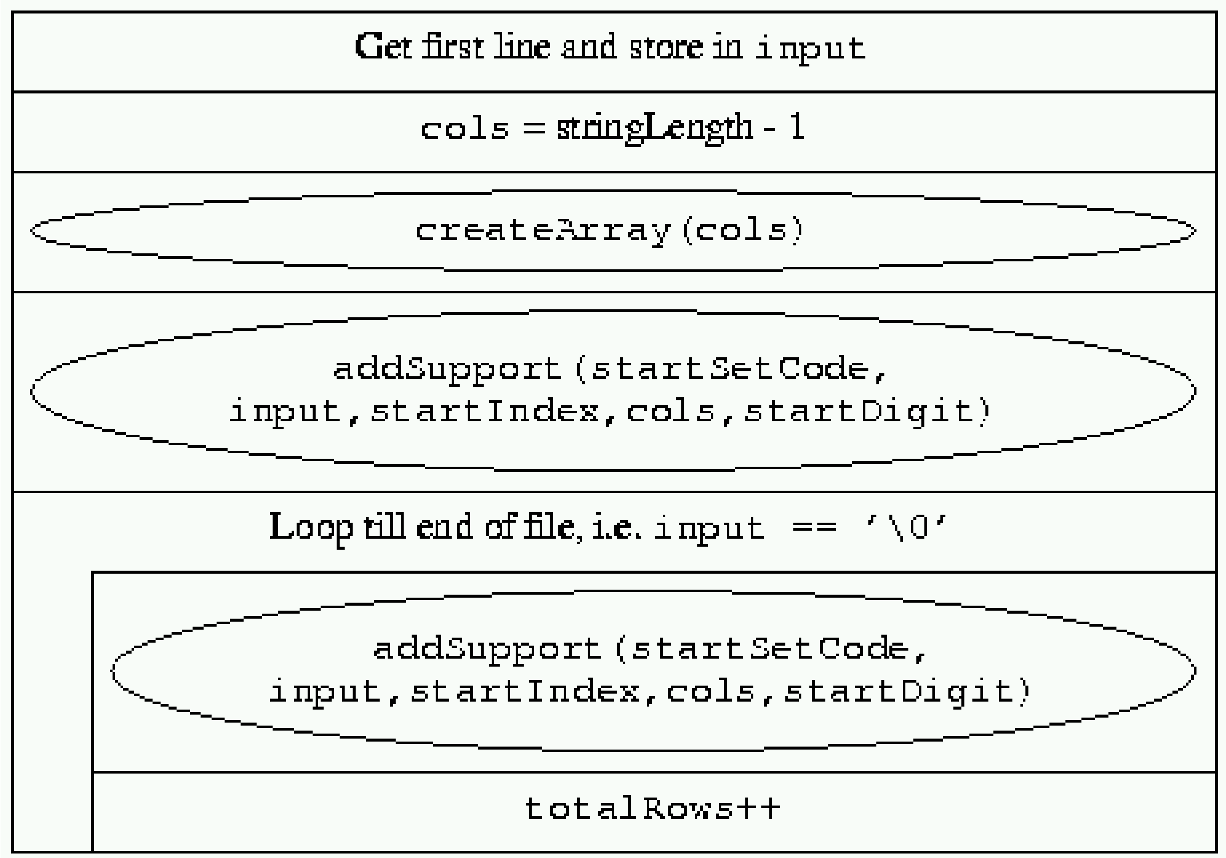

Figure 2: Nassi-Shneiderman chart for *processInput* procedure

- A *setCode* initialised to 0 (no pattern).
- The input string.
- The current *index* initialised to 0.
- The length of the input string.
- The value of the current digit in the binary sequence initialised to 1.

We then proceed as follows:

1. Loop through input string until end string is encountered
2. On each iteration:
   (a) Determine the value of the next digit by doubling the value of the current digit.
   (b) Increment the index
   (c) If the current digit is '1' then; (i) determine the bit pattern derived sofar by adding the current digit value to the current set code, (ii) use the bit pattern to index into the array and increment the support, (iii) recur with the next digit value and the incremented index, (iv) once the recursion has unwound determine the next set code by adding the current set code to the current digit value (not the next digit value calculated in 2(a) above).

For example given an input string of the value 0110 we proceed as follows:

Loop 1, Index 0: The initial values will be: $setCode = 0, index = 0, currentDigit = 1$. On the first iteration $inputString[0]$ will equal 0, therefore repeat.

Loop 1, Index 1: Settings: $setCode = 0, index = 1, currentDigit = 2$. On this iteration $inputString[1]$ will equal 1 therefore calculate the bit pattern/array index:

$$arrayIndex = currentDigit + setCode = 2 + 0 = 2$$

increment the support by using this pattern to index into the array and then recur with a new call to *addSupport* (settings: $setCode = 0, index = 2, currentDigit = 4$).

Loop2, Index 2: Settings: $setCode = 0, index = 2, currentDigit = 4$. $inputString[2]$ is also equivalent to 1, so determine the bit pattern:

$$arrayIndex = currentDigit + setCode = 4 + 0 = 4$$

Update the array and recur again.

Loop 3, Index 3: settings: $etCode = 0, index = 3, currentDigit = 8$. $inputString[3]$ is equivalent to 0, so repeat.

Loop 3, index 4: Settings: $setCode = 0, index = 4, currentDigit = 16$. The index is now greater than the length of the input string therefore jump out of the loop, come out of the second recursion.

On coming out of the second recursion (loop 3), revise the set code by adding the current digit value to the current *setCode* (both from before the commencement of the recursion), Thus: $setCode = setCode + currentDigit = 0 + 4 = 4$ and repeat.

Loop 2, Index 3: Settings: $setCode = 4, index = 3, currentDigit = 8$. $inputString[3]$ is equivalent to 0, so repeat again.

Loop 2, Index 4: Settings: $setCode = 4, index = 4, currentDigit = 16$. The index is again now greater than the length of the input string therefore jump out of the current loop hence terminating the first recursion (loop 2).

The process now continues by calculating a new *setCode* (using values from before the start of the first recursion): $setCode = setCode + currentDigit = 0 + 2 = 2$ and continue the initial loop.

Loop 1, Index 2: Settings: $setCode = 2, index = 2, currentDigit = 4$. $inputString[2]$ is also equivalent to 1, so determine the bit patter:

$$arrayIndex = currentDigit + setCode = 4 + 2 = 6$$

update the array and recur again (third recursion in this example).

Loop 4, Index 3: Settings: $setCode = 2, index = 3, currentDigit = 8$. $inputString[3]$ is equivalent to 0, so repeat.

Loop 4, Index 4: Settings: $setCode = 2, index = 4, currentDigit = 16$. The index is again now greater than the length of the input string therefore jump out of the current loop thus terminating the (third) recursion (loop 4).

Continue the process by calculating a new *setCode* (again using values from before the start of the first recursion): $setCode = setCode + currentDigit = 4 + 2 = 6$ and continue the loop.

Loop 1, Index 3: Settings: $etCode = 6, index = 3, currentDigit = 8$. $inputString[3]$ is equivalent to 0, so repeat.

Loop 1, Index 4: Settings: $setCode = 6, index = 4, currentDigit = 16$. The index is now greater than the input string length thus end the loop, there are no more recursions to unwind, therefore end the procedure.

From the above three patterns have been found: $0100 = 2$, $0010 = 4$ and $0110 = 6$.

The procedure includes the following data items:

NAME	TYPE	DESCRIPTION
setCode	Integer	Formal parameter describing the code/bit pattern as derived sofar, initialised to 0.
inputString	Character Ptr.	Formal parameter, pointer to start of input string.
index	Integer	Formal parameter, the index with which to process the input string, initialised to 0.
length	Integer	Formal parameter indicating the length of the input string.
currentDigit	Integer	Formal parameter representing the value for the current digit under consideration, initialised to 1.
arrayIndex	Integer	Local variable to index into the 1-D array storage structure.

A detailed design for the *addSupport* procedure is presented in Figure 3.

4.4 CREATE ARRAY

The *createArray* procedure dynamically creates a 1-D array whose size is equivalent to the number of possible attribute permutations. The size of the set is calculated using the identity:

$$arraysize = 2^{numberofcolumns}$$

Once created we loop through the array initialising each element (indicating support) to 0. The procedure uses two data items as follows:

NAME	TYPE	DESCRIPTION
numCols	Integer	Formal parameter giving the number of columns in the table (this in turn defines the number of attributes).
index	Integer	Local variable to index into the array when initialising it.

A detailed design for the *createArray* procedure is presented in Figure 4.

4.5 OUTPUT RESULT

The *outputResult* procedure is used solely to output the final result. We loop through the array testing each element. If this is above the minimum support freshold we deconstruct the array index to produce the attribute sequence described by the index/bit pattern and then output this sequence and the support. The deconstruction is carried out through a call the *deconstructArrayIndex* procedure described below. During the processing a count is also maintained of the total number of interesting sets. The procedure uses the following data items:

NAME	TYPE	DESCRIPTION
minSupport	Integer	Local variable holding the minimum support.
index	Integer	Local variable to index into the array.
total	Integer	Local variable in which to maintain a count of the total number of attribute combinations which are supported.

A Nassi-Shneiderman chart for the procedure is given in Figure 5.

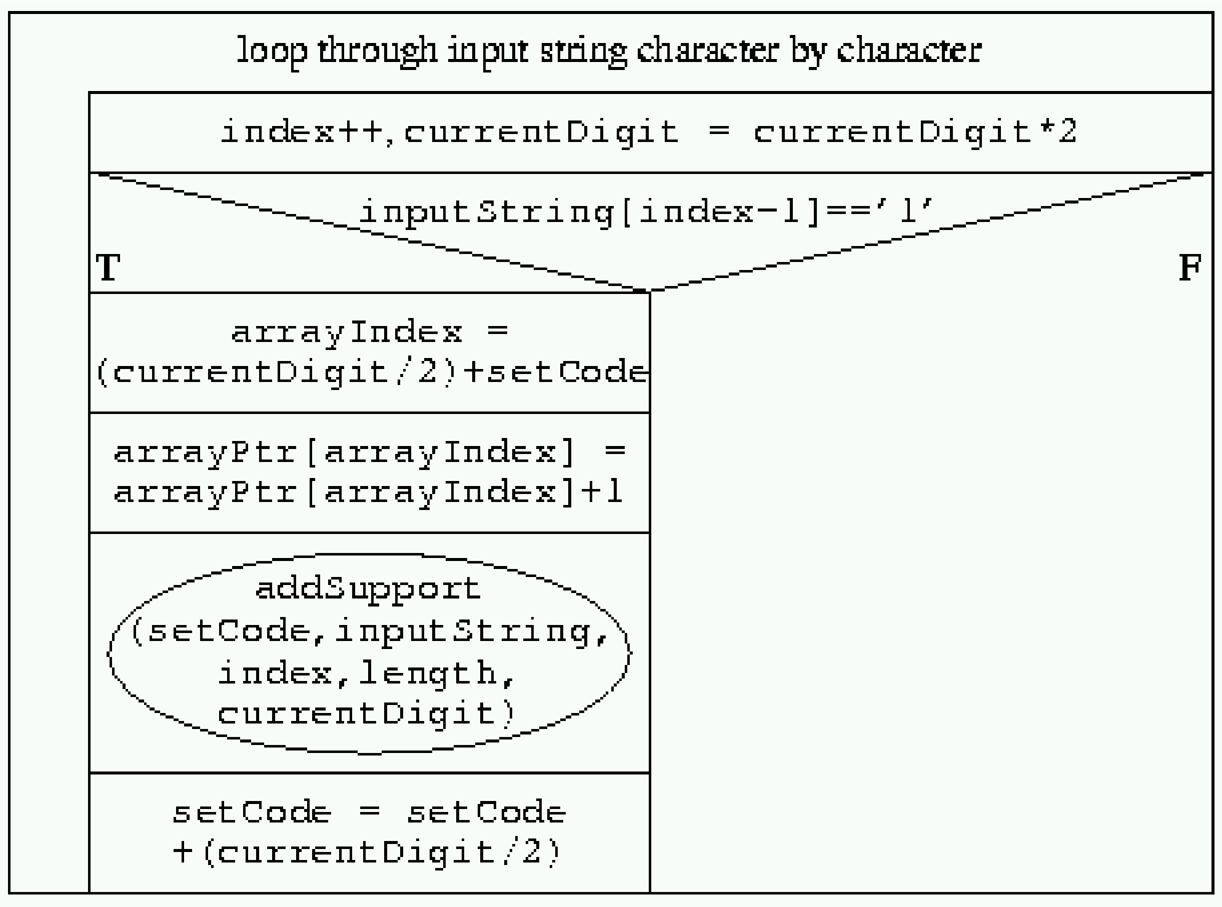

Figure 3: Nassi-Shneiderman chart for *addSupport* procedure

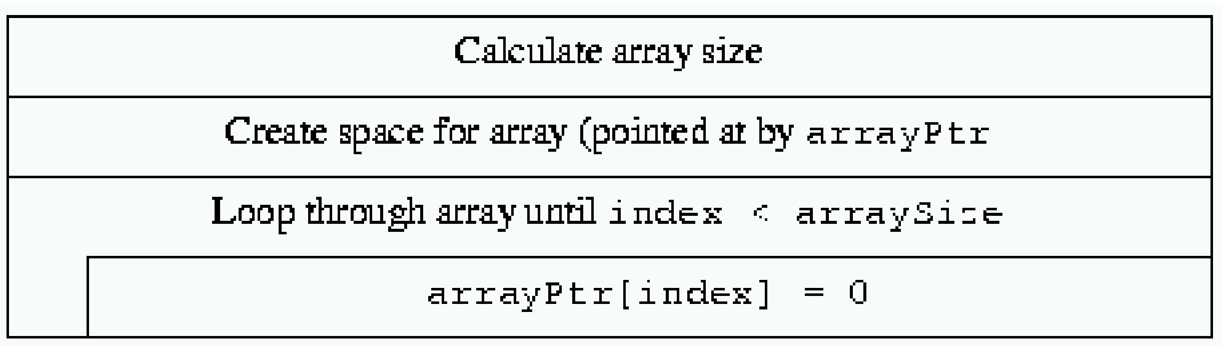

Figure 4: Nassi-Shneiderman chart for *createArray* procedure

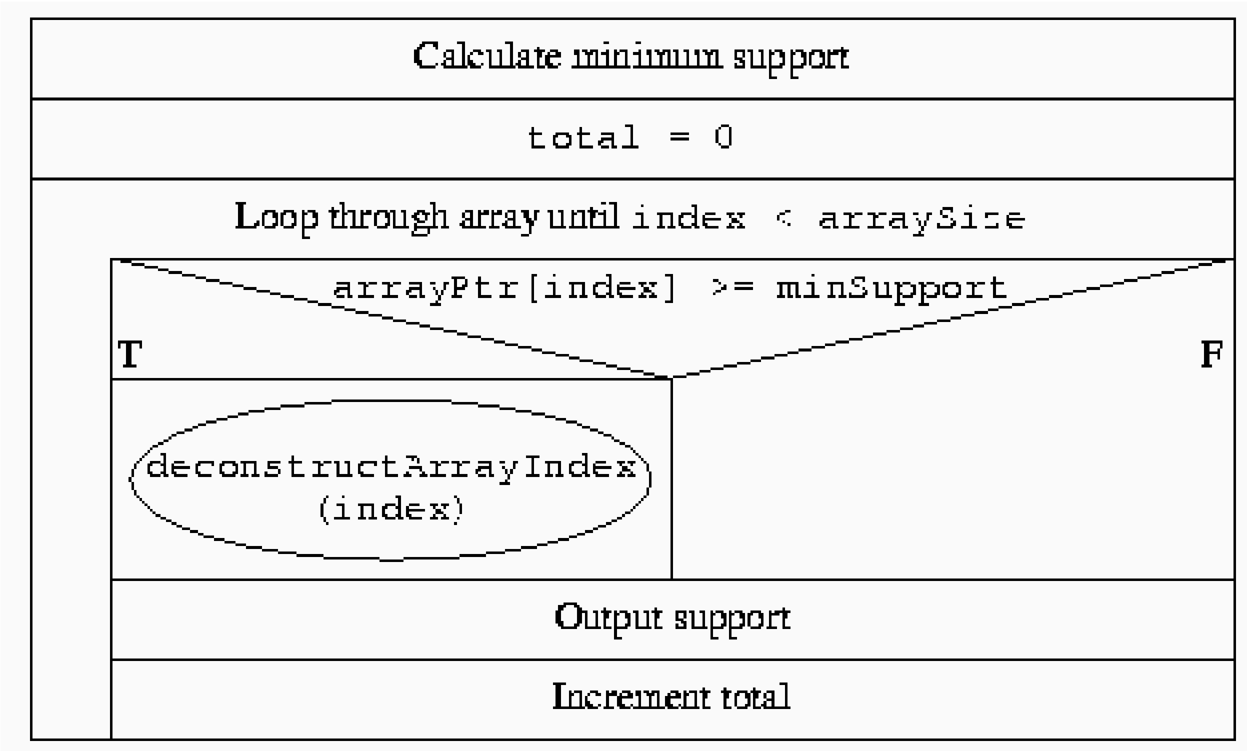

Figure 5: Nassi-Shneiderman chart for *outputResult* procedure

4.6 DECONSTRUCT ARRAY INDEX

The final procedure, *deconstructArrayIndex* is similar to the *generateBitPattern* procedure described in [1]. The *deconstructArrayIndex* loops through the array index and outputs a letter/number pair associated with each digit wherever a '1' is encountered. Attributes are numbered $a1$, $b1 \cdots y1$, $z1$, $a2$, $b2$, and so on. Thus, for example, given a index/bit pattern 11100 this will be equivalent to the attribute sequence $a1, b1, c1$. The procedure includes the following data items:

NAME	TYPE	DESCRIPTION
number	Integer	Formal parameter giving the index to be deconstructed
alpha	character	Local variable in which current letter code is stored, initialised with the letter 'a'.
counter	Integer	Local variable in which current number code is stored, initialised with the number '1'.

A detailed design for the *deconstructArrayIndex* procedure is presented in Figure 6

5 ANALYSIS

The algorithm, like the previous brute force 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 20 columns and 100,000 rows, thus far outperforming the previous algorithm which could only cope reasonable with 12 columns and 100,000 rows.

The first significant advantage of the algorithm is the storage space required. This is given by the identity:

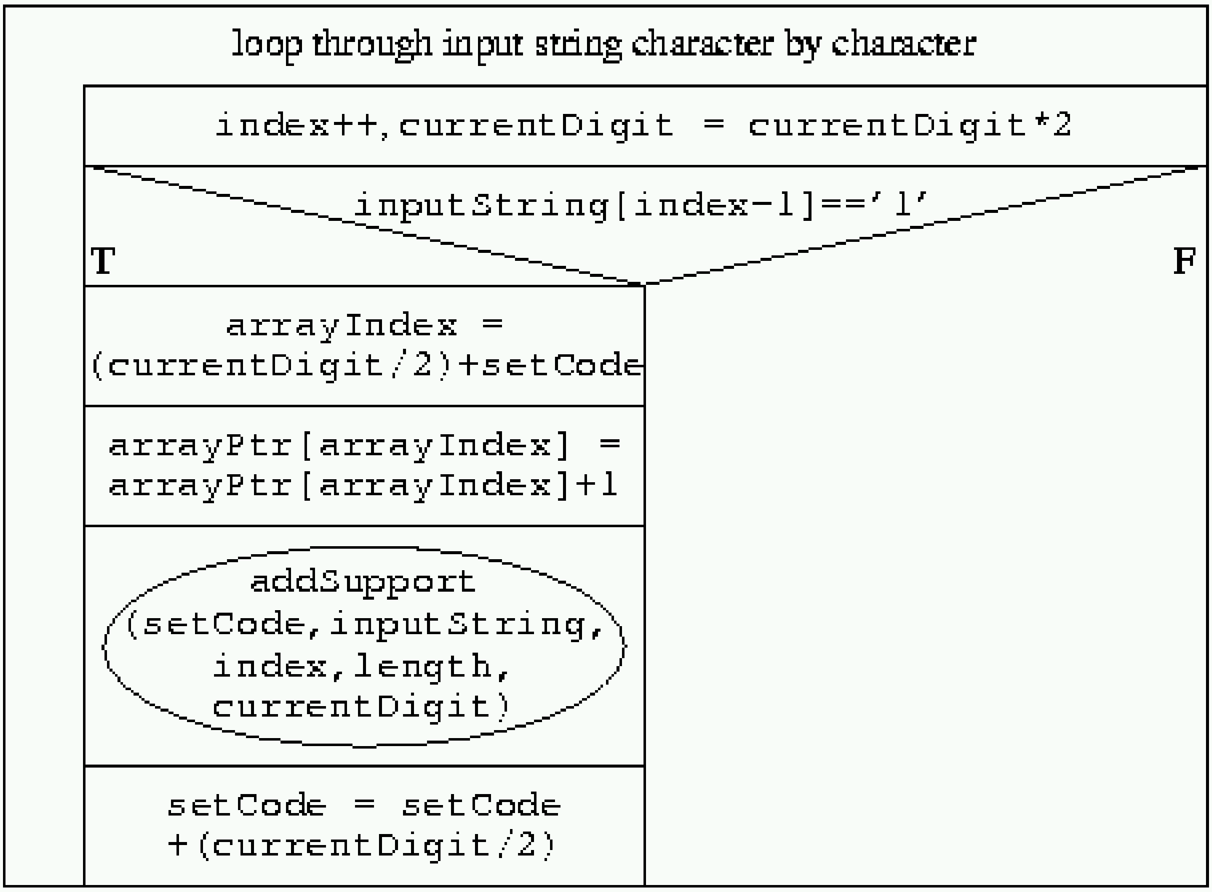

Figure 6: Nassi-Shneiderman chart for *deconstructArrayIndex* procedure

$$storage = 4\left(2^n - 1\right) bites$$

where n is the number of columns. The constant 4 is included because supports are stored as 32 bit integers (4 bites).

The second advantage is that the number of comparisons has been reduced by no longer comparing every row with every possible permutation contained in storage, but instead calculating the possible permutations in any current row and then using the permutation code to index into the array. This results in a saving of 50% assuming a normal distribution, i.e. given any specific row in the table only half of the elements will be instantiated with a 1. Thus the number of updates is calculated from:

$$num\ of\ calculations = m\left(2^{\frac{n}{2}} - 1\right)$$

Where m is the number of rows.

The use of the integer codes to represent each row in the table is again seen as a good idea. However, as before this limits the approach to 32 columns — a standard unsigned integer comprises 32 bits.

The generation of codes which was a feature of the previous algorithm is omitted here (other than when outputting results), instead permutations must be calculated. Assuming a normal distribution there are:

$$permutations = \frac{2^n - 1}{2}$$

permutations per line. Thus the total number of permutations is given by:

$$numpemutationidentifications = m \left(\frac{2^n}{2} \right)$$

Where m is the number of permutations.

The number of pattern generations (for output) is given by:

$$numcodegenerations = n + \frac{(n\,(n-1))}{2}$$

Thus comparing the two brute force algorithms

	GCL-BF1	GCL-BF1
Storage (bites)	$4\,(2^n - 1)$	$44\,(2^n - 1)$
Number of Comparisons	$m\,(2^n - 1)$	Nil
Number of Updates	$\frac{m(2^n-1)}{2}$	$\frac{m(2^n-1)}{2}$
Number permutation identifications	Nil	$m\left(\frac{2^n-1}{2}\right)$
Number code generations	$2^n + m - 1$	$n + \frac{(n(n-1))}{2}$

Thus overall the second brute force algorithm far out performs the first, however, it can hardly be described as a good algorithm — it can only reasonably cope with 20 attributes. We require something that can cope with many more. The algorithm calculates the support for all combinations of attributes although the majority of combinations will not be adequately supported, i.e. there is much wasted work. The route to improvement is thus to reduce to magnitude of this wasted effort.

References

[1] Coenen, F. (1998). A Bruteforce Algorithm for "Basket Analysis" (GCL-BF1). Dept of Computer Science, University of Liverpool. Working Paper 2.

APPENDIX

```
/* ------------------------------------------------------------------------ */
/*                                                                          */
/*                B R U T E  F O R C E   A L G O R I T H M   2              */
/*                                                                          */
/*                              Frans Coenen                                */
/*                                                                          */
/*                             10 January 19989                             */
/*                                                                          */
/* ------------------------------------------------------------------------ */

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

/* to compile "gcc -o bruteForceAlg bruteForceAlg.c -lm -lc" */
```

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

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

/* Example test file:

00100011001111100101000110
11001001000000010111001010
00010000001010111001010001
10110110101110010001001010
10100000110111001000011000
01010111101101110101111101
00010001001010010010010101
01000011101011110100001111
00100100010111010010000010
11001111100111001001011011    */

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

/* arrayPtr        = Pointer to start of array data structure.
   arraySize       = Total number of elements.
   totalRows       = Total number of rows in table.
   supportFreshold = Support freshold, e.g. 20 (20%). */

int *arrayPtr;
int arraySize;
int totalRows = 1, supportFreshold = 20;

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

void processInput(FILE *);
void addSupport(int, char *, int, int, int);
void createArray(int);
void outputResult(void);
void deconstructArrayIndex(int);

/* ---------------------- */
/*                        */
/*          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. */

outputResult();
}

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

/* Note: fgets(stringname,n,filename) - procedure 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[32];
int cols;
const int startSetCode = 0;
const int startIndex = 0;
const int startDigit = 1;

/* Get first row, calculate number of columns and produce 2_D array
representuing candidate sets.  */

fgets(input,64,in_file);
cols = strlen(input)-1;
createArray(cols);

/* process first line. */

addSupport(startSetCode,input,startIndex,cols,startDigit);

/* process rest of table */

while (fgets(input,64,in_file) != '\0') {
        addSupport(startSetCode,input,startIndex,cols,startDigit);
        totalRows++;
        }
}

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

```
/* For each permutation in the current input line determine the index and
update the array*/

void addSupport(int setCode, char *inputString, int index, int length,
                int currentDigit)
{
int arrayIndex;

while (index<length) {
        index++;
        currentDigit = currentDigit*2;
        if (inputString[index-1] == '1') {
                arrayIndex = (currentDigit/2)+setCode;
                arrayPtr[arrayIndex] = arrayPtr[arrayIndex]+1;
                addSupport(setCode,inputString,index,length,currentDigit);
                setCode = setCode+(currentDigit/2);
                }
        }

}

/* ---------------------------------------------- */
/*                                                */
/*          ARRAY DATA STRUCTURE UTILITIES        */
/*                                                */
/* ---------------------------------------------- */

/* ------ CREATE ARRAY ------ */

/* Produce the 1-D integer array data structure representing the power set of the
input attribute set. This power set then represents all the candidate sets from
which association rules may be derived provided that there is sufficient support. */

void createArray(int numCols)
{
int index;

/* Calculate array size. */

arraySize = (int) pow(2.0,(double) numCols);

/* Create space (Malloc returns NULL if no space available). */

arrayPtr = (int *) malloc(sizeof(int)*arraySize);

/* Initialise the array with a sequence of numbers derived from the indexes. */

for(index=0;index < arraySize;index++) arrayPtr[index] = 0;
}

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

```
/* ------------------------ */

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

void outputResult(void)
{
int minSupport = (totalRows*supportFreshold)/100;
int index, total=0;

for (index=0;index<arraySize;index++) {
        if (arrayPtr[index] >= minSupport) {
                deconstructArrayIndex(index);
                printf(" (support = %d)\n",arrayPtr[index]);
                total++;
                }
        }

printf("Total number of interesting sets = %d\n",total);
}

/* ------ DECONSTRUCT ARRAY INDEX ------ */

/* Output array index (the attribute code) as a set of attribute headings. */

void deconstructArrayIndex(int number)
{
char alpha = 'a';
int counter = 1;

while (number != 0) {
        if (number % 2 == 1) {
                printf("%c%d ",alpha,counter);
                }
        if (alpha == 'z') {
                alpha = 'a';
                counter++;
                }
        else alpha = alpha+1;
        number = number/2;
        }
}
```