The Implementation of Two Utilities in BDX

Nathaniel Fink

A Major Report
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 1996

ISBN 0-612-18389-0

Abstract

The Implementation of Two Utilities in BDX

Nathaniel Fink

The *BDX* program was developed by Dr. C. Lam and Mr. L. Thiel to generate *Balanced Incomplete Block Designs*, or *BIBDs*. used in combinatorics [6]. In this report, I will be discussing two utilities for *BDX* which I helped to implement.

The first utility consists of Dump and Restore routines which allow the user to halt the generation of block designs and save the state of the program. The user can then continue the run at a later time, using part or all of the generated data. The second utility is a test, based on a theorem by Greig [2], called the Hook-22 test in *BDX*. Its purpose is to reduce the number of cases *BDX* has to consider in generating the special case of a BIBD known as the $(22, 33, 12, 8, 4)$-BIBD.

Chapter 1 gives some background information on block designs and the *BDX* program. Section 1.2 describes block designs in general and Section 1.3 gives a description of *BDX* and some of its capabilities.

Chapter 2 describes the first *BDX* utility I implemented, the Dump/Restore utility. Section 2.1 discusses the nature of the Dump/Restore utility and the types of situations in which it would be needed. Section 2.2 shows how the Dump/Restore utility is used in practice in *BDX* scripts. Section 2.3 shows how the Dump/Restore utility is implemented and describes some of the changes it has gone through since its creation.

Chapter 3 describes the second *BDX* utility, Greig's Test, including Greig's Criterion, and my implementation of the utility as two testing routines. Finally, I show a proof of the redundancy of one of these routines.

Appendix A.1 gives some samples of output from *BDX*. Appendices B and C contain the source listings for the Dump/Restore and Grieg's Test utilities, respectively.

Contents

List of Figures

List of Tables

Chapter 1

Background

The *BDX* program was designed to generate block designs which are *uniform*, *balanced* and *incomplete*, known as BIBDs [1]. This section describes BIBDs and gives a general overview of how *BDX* is used to generate them. This section also describes some utilities in *BDX* that are used to speed up the search.

1.1 Definition of Terms

Some of the terms used in this and later chapters are defined here.

Block Design: A block design is an object used in combinatorial analysis to describe the matching of two sets. One is a set of *varieties* and the other is a set of subsets of the varieties, called the blocks. For a further discussion of block designs, see Section 1.2 below.

Incidence Matrix: An incidence matrix is a rectangular matrix with values 0 and 1. Block designs can be represented by incidence matrices as follows: the entry $A_{ij} = 1$ if and only if *variety* i is in *block* j of the BIBD. This report, as well as the *BDX* program itself, depicts block designs as incidence matrices, and any reference, given here, to an incidence matrix implies a reference to its underlying block design. The incidence matrix is usually referred to as the A matrix. Also, this report makes reference to the rows and columns of A. For consistency, the columns and rows of A will refer respectively to the blocks and varieties of the block design.

$\mathbf{A}_{row\ m}$,$\mathbf{A}_{col\ m}$: $A_{row\ m}$ means the mth row of the A matrix. $A_{col\ m}$ means the mth column of the A matrix.

Degree: We define the degree of a vector to mean the number of ones in the vector. Vectors refer to the rows and columns of the incidence matrix, and so are also restricted to the values 0 and 1.

1.2 Block Designs

The three properties of a block design discussed here are *uniform*, *balanced* and *incomplete*. They can be defined as follows. Let A be the incidence matrix for a block design, with b columns and v rows. If all b columns are of degree exactly k $(0 < k \leq v)$ then A is *uniform*. If k is strictly less than v and all v rows are of degree r where $0 < r < b$ then A is *incomplete*. If all pairs of varieties occur in exactly λ blocks, then the block design is *balanced*. Equivalently, a block design is *balanced* if the inner product of each pair of distinct rows of A is λ. Since each row corresponds to a variety, a pair of varieties occurring in λ columns means that the corresponding rows have exactly λ ones in the same position and so the inner product of the two rows is λ.

Having defined the terms *balanced*, *incomplete* and *uniform*, we can give a formal definition of the *BIBD*. Although the expression BIBD does not explicitly include the term *uniform*, it is included in the definition as follows:

> **Definition:** A *balanced incomplete block design* is an arrangement of v distinct varieties into b blocks such that each block contains exactly k distinct varieties, each object occurs in exactly r different blocks, and every pair of distinct varieties a_i, a_j occurs together in exactly λ blocks. [3, p. 126].

Proposition 1.1 *The relations among the five parameters which describe a BIBD are as follows:*

$$\begin{aligned} bk &= rv, && (1.1) \\ r(k-1) &= \lambda(v-1), \text{ and} && (1.2) \\ \lambda &< r. && (1.3) \end{aligned}$$

Proof: There are k ones in each of b columns. Also, there are r ones in each of v rows. Therefore, the number of ones in A is $bk = rv$, which proves (1.1). Each variety occurs in r columns and is paired with $k-1$ other varieties in that column. Conversely, each variety is paired λ times with all the other $v-1$ varieties. Therefore the total number of pairings for a given variety is $r(k-1) = \lambda(v-1)$, which gives (1.2).

From (1.2) we have

$$r/\lambda = (k-1)/(v-1).$$

Since the block design is *incomplete*,

$$k < v,$$

or,

$$k-1 < v-1.$$

Therefore,

$$\lambda < r.$$

Example 1.1: The concept of the BIBD may be illustrated by means of a class of well-known BIBDs, the projective planes. A projective plane of order n consists of n^2+n+1 lines and n^2+n+1 vertices. Each line joins $n+1$ vertices, and each vertex is the intersection of $n+1$ lines. All pairs of lines intersect exactly once on some vertex and all pairs of vertices lie on exactly one line. This can be represented by a BIBD with n^2+n+1 blocks representing the vertices, and n^2+n+1 varieties representing the lines.

Since each vertex contains $n+1$ lines, the blocks are of degree $n+1$. Additionally, each pair of elements occurs exactly once. This describes a BIBD where $b = v = n^2+n+1$, $r = k = n+1$, and $\lambda = 1$.

(Note: block/var vs col/row)

Although the inner products of the blocks, or columns, are all λ in this example, this is not generally the case. The restriction of λ on the varieties, or rows, does not exist for the columns, whose inner products may vary from 0 to k, where k is the degree of the column. For example, (1.4) is the incidence matrix of a $(9,12,4,3,1)$-BIBD. The inner product of any two distinct rows is 1. However, the inner product of the first and last columns is 0.

$$A = \begin{pmatrix} 1&1&1&1&0&0&0&0&0&0&0&0 \\ 1&0&0&0&1&1&1&0&0&0&0&0 \\ 1&0&0&0&0&0&0&1&1&1&0&0 \\ 0&1&0&0&1&0&0&1&0&0&1&0 \\ 0&1&0&0&0&1&0&0&1&0&0&1 \\ 0&0&1&0&1&0&0&0&0&1&0&1 \\ 0&0&1&0&0&0&1&0&1&0&1&0 \\ 0&0&0&1&0&1&0&0&0&1&1&0 \\ 0&0&0&1&0&0&1&1&0&0&0&1 \end{pmatrix}. \qquad (1.4)$$

The B matrix, which is directly related to A, is the product of A and its transpose.

B is a $b \times b$ matrix and contains elements $B_{ij} = \langle A_{row\ i}, A_{row\ j} \rangle$. B is useful in checking that A forms a BIBD. From the definition of a BIBD, one can show that the matrix $B = AA^T = (r - \lambda)I + \lambda J$; that is, B has r on its diagonal and λ everywhere else. The diagonal contains the inner products of all rows with themselves which are all of degree r. The off-diagonal elements are inner products of distinct rows which all have λ elements in common, since each pair of elements of distinct columns occurs exactly λ times. This is a sufficient test of the requirements of a BIBD, as shown in [3, p. 127].

Another matrix related to A is S. This matrix is defined as $S = A^T A$, and is therefore of size $v \times v$. Since the elements of S are the inner products of the columns of A, we can write this as $S_{ij} = \langle A_{col\ i}, A_{col\ j} \rangle$. S is used in the analysis of A, for the purpose of the generation of A. This will be discussed later in Section 1.3 and in Chapter 3. Note that S is a symmetric matrix, since the inner product operation is commutative.

Definition: If $v = b$ and $r = k$ then A forms a *symmetric* BIBD.

The following theorem was proved in [3, p. 130]:

Theorem 1.1 *If A forms a* symmetric *BIBD then B=S.*

Example 1.2: The projective plane described in Example 1.2 is an example of a symmetric BIBD.

Let $(v, b, r, k, \lambda) = (7, 7, 3, 3, 1)$, which is the projective plane of order 2. That is, A is of size 7×7. It also has columns and rows both of degree 3 and pairs occurring once. Here is an example of this block design given as a set of blocks containing the variety elements $\{1, ..., 7\}$:

$$\begin{aligned}
B_1 &= \{1,2,3\}\\
B_2 &= \{1,4,5\}\\
B_3 &= \{1,6,7\}\\
B_4 &= \{2,4,7\}\\
B_5 &= \{2,5,6\}\\
B_6 &= \{3,4,6\}\\
B_7 &= \{3,5,7\}.
\end{aligned}$$

The incidence matrix A, with columns representing the above blocks, is:

$$\begin{pmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 0\\
1 & 0 & 0 & 1 & 1 & 0 & 0\\
1 & 0 & 0 & 0 & 0 & 1 & 1\\
0 & 1 & 0 & 1 & 0 & 1 & 0\\
0 & 1 & 0 & 0 & 1 & 0 & 1\\
0 & 0 & 1 & 0 & 1 & 1 & 0\\
0 & 0 & 1 & 1 & 0 & 0 & 1
\end{pmatrix},$$

and since A forms a symmetric BIBD, $S = B = AA^T$ is

$$\begin{pmatrix}
3 & 1 & 1 & 1 & 1 & 1 & 1\\
1 & 3 & 1 & 1 & 1 & 1 & 1\\
1 & 1 & 3 & 1 & 1 & 1 & 1\\
1 & 1 & 1 & 3 & 1 & 1 & 1\\
1 & 1 & 1 & 1 & 3 & 1 & 1\\
1 & 1 & 1 & 1 & 1 & 3 & 1\\
1 & 1 & 1 & 1 & 1 & 1 & 3
\end{pmatrix}.$$

1.3 The BDX Program

In this section, we give a brief description of some of the features of *BDX* which are required for the understanding of the report. For a detailed description, see [6].

BDX uses a branch-and-bound algorithm to search for and generate BIBDs. The branch-and-bound algorithm is useful in BIBD generation because it allows tests to be performed on parts of the A matrix to determine whether the partial matrix can lead to a solution, and allows restrictions to be placed on parts of the matrix to shorten the search time.

The branch-and-bound algorithm works by breaking the problem down into search levels. At each level, a fixed number of decisions branch out into the next level, so that the search tree grows geometrically with each level. At a given level, a decision is made using a test function, or predicate. The test "prunes" the tree by determining whether a solution is ultimately possible from the given current partial candidate. When a branch is pruned, all search nodes extending from that partial candidate are skipped. This can significantly reduce the number of cases being tested. In the case of the BIBD, a search of every possible configuration of the incidence matrix would require 2^m candidates to be tested (where m is the number of elements of the matrix). This is far too many for most non-trivial BIBDs. The branch-and-bound algorithm need only test a small fraction of these cases. Additionally, branch-and-bound tests the partial candidate, which is usually faster than testing a complete candidate.

BDX uses a script language to direct the search routine. The language consists of commands to specify the five parameters (v, b, r, k, λ) for the A matrix. *BDX* also references the S matrix, where $S = A^T A$. For both A, and S, *BDX* can restrict elements to specified values. Even though restrictions on S are really restrictions on the blocks of A—since the elements of S are the inner products of the blocks of A—it sometimes easier to dictate restrictions in terms of S. Values can be assigned to A or S, or an area of A can be restricted so that all the ones in that area add up to a given number or range of numbers.

BDX automatically generates all possible solutions of a given BIBD. To reduce the number of cases being searched, cases can be generated with the **isom** option, which causes *BDX* to skip all except non-isomorphic candidates. The generation of the BIBD is done by the **try** command. This command can be used to generate all, or a part, of the matrix. There are several search strategies, each of which restricts the search in a different way. Also, any region of A or S can be printed at any time.

For example, Fig. 1.1 shows a script to generate the projective plane of order 2. The first three statements define the five parameters of the BIBD in the order (v, r, b, k, λ). These are required at the beginning of the script. The fourth line contains a **try** statement which directs *BDX* to perform the actual search of the BIBD on the whole 7×7 matrix. Finally, **exit** ends the run (see [6, Sec. 6]). This script, unfortunately, would generate every possible $(7,7,3,3,1)$-BIBD—over 150,000 of them!

```
7 rows of 3;
7 cols of 3;
lambda=1;
try row[1:7,1:7];
exit
```

Figure 1.1: Script for whole (7,7,3,3,1) with no isomorph rejection

In a single script it is not necessary to generate the entire BIBD. This strategy is useful (and generally necessary) in handling large BIBDs, where the problem must be broken down into steps. See Section 2.1 for more detail.

In this example, we could reduce the number of cases by trying only the first 3 rows, as in Fig. 1.2. This generates less cases because there are less possibilities for the first three rows of the BIBD than for the entire matrix. In this case, there are 5670 solutions.

In another example, we could restrict the search in other ways, such as only allowing solutions which are non-isomorphic, by using the **isom** option of the **try** command, as shown in Fig. 1.3. This eliminates the need to generated versions of solutions attainable by permuting their rows and columns. In this example, only one non-isomorphic solution is generated, as shown in Fig. 1.4. In other words, all of the

```
7 rows of 3;
7 cols of 3;
lambda=1;
try row[1:3,1:7];
exit
```

Figure 1.2: Script for first three rows of (7,7,3,3,1)

solutions that would otherwise be generated are isomorphic.

```
7 rows of 3;
7 cols of 3;
lambda=1;
try row isom [1:7,1:7];
exit
```

Figure 1.3: Script for whole (7,7,3,3,1) with isomorph rejection

```
  1234567

1 1110000
2 1001100
3 1000011
4 0101010
5 0100101
6 0011001
7 0010110
```

Figure 1.4: Solution for (7,7,3,3,1) with isomorph rejection

In Fig. 1.5, we place ones in the first block, restricting their positions. This does not affect whether we get a solution because permutations of columns of the A matrix are isomorphisms and so we still get the same number of non-isomorphic solutions. We also restrict the solutions so that only every 2000th is passed. This means that *BDX* will print the first solution, skip 1999 solutions, print the next one, and so on. Of the 4320 generated solution, 3 are passed, as shown in Fig. 1.6.

```
7 rows of 3;
7 cols of 3;
lambda=1;
A[1:3,1]=1;
try row[1:7,1:7];
pass every 2000 times;
exit
```

Figure 1.5: Script for whole (7,7,3,3,1), fixing the first column of A

```
  1234567

1 1110000
2 1001100
3 1000011
4 0101010
5 0100101
6 0011001
7 0010110

  1234567

1 1010100
2 1000011
3 1101000
4 0011001
5 0100101
6 0110010
7 0001110

  1234567

1 1000101
2 1001010
3 1110000
4 0100110
5 0010011
6 0101001
7 0011100
```

Figure 1.6: Selected solutions for whole (7,7,3,3,1), fixing the first column of A

Chapter 2

The Dump/Restore Utility

This section explains the purpose of the Dump/Restore utility, followed by a description of how this utility is called in a *BDX* script. Finally, there is a description of the implementation of the utility.

2.1 Background

Upon completion of a partial generation of a BIBD, the Dump/Restore utility allows a *BDX* script to dump all the data to a file. The *BDX* program can later restore the data to another script, which continues generating the BIBD. Dump/Restore has two main uses:

1. **Estimation.** When trying to find the most efficient strategy for generating a particular BIBD, several estimates must be made in order to determine whether a given strategy is better. Output data produced before the dump was called can provide information necessary to make estimates in any future runs from that point. Moreover, Dump/Restore allows several tests to immediately start from a single pre-computed point.

2. **Distributed Searches.** A common general strategy in long searches is the distributed search, or a sharing of the workload among many computers. The

Dump/Restore routine allows a single previously generated starting configuration to be restored on any number of machines.

Dump/Restore works especially well for both estimation and distributed searches when the starting configuration takes a long time to generate, as in the case of a starting configuration that was generated by an expensive utility like isomorph rejection. Dump/Restore eliminates the need to generate the configuration for each run, whether for testing, or in a distributed environment.

Estimation

The main goal in developing a script to generate a particular BIBD is to reduce the search time. Estimation is useful in determining which search strategy is fastest. For large problems, starting configurations can be created at various points where it is clear there are several directions the search can take. Based on estimation, decisions can be made on each subsequent part of the search. Many Dump/Restore operations can take place in succession, perhaps creating a distributed search tree.

Rather than a simple sequential search of solutions, *BDX* uses a fast branch-and-bound algorithm. Even so, search time may still be long for many BIBDs. One way to decrease the search time is to reduce the number of cases being searched. This can be done using the following basic methods. These methods can be combined in any variety of search strategies:

1. **Placing restrictions on the entries of the matrices A or S.**

   This method allows the user to restrict values of the entries of these matrices, bypassing cases not having those values.

2. **Reordering of partial steps.** If there is some knowledge of the number of branches of each step, the shape of the search tree can be streamlined by handling certain steps before others.

3. **Isomorph rejection.** Two candidates are isomorphic if they differ only by row or column permutations. Permuting the rows or columns of a solution does not

affect the solution. This means that if one candidate leads to a solution, all candidates that are isomorphic to that candidate (that is, the candidate's orbit) will also lead to solutions. If the user wants to find whether a single solution exists, rather than doing an exhaustive search, the search could be restricted to only non-isomorphic candidates, cutting down the number of cases being searched to one candidate per orbit. However, generating the isomorphism group for isomorph testing can itself be time-consuming.

If the program has run for a while, it would be useful to estimate its completion time. This requires repeated test runs from the same point in the search. If the point of interest is in the middle of the search, rather than starting the search from the beginning, we can create a dump file containing all data generated up to this point, and a test script can restore and test search strategies from this point.

The user can also dump and restore in succession as many times as necessary, exploring many parallel possibilities without having to constantly restart from the beginning.

Distributed Searches

A distributed search is the procedure of continuing a search on more than one machine simultaneously. The user can start any number of parallel runs from the same dump file. For instance, a partial search could be run on one machine, then dumped. It could then be restored on two machines, with two further partial searches performed, and so on. The Restore routine allows some or all of the partial solutions stored in the dump file to be restored to the next script.

For example, a set of dumped partial solutions could be restored in a mutually exclusive manner on each machine in the distributed system, if necessary. In this way, parallel runs could take advantage of the mutually independent way that the branch-and-bound algorithm searches each case. Each machine would restore from a copy of the same dump file, but would be restoring different partial solutions, and would return its own result of the partial search.

2.2 Syntax and Examples

The role of the Dump/Restore utility is to halt the program at a certain point, and to dump all generated data, including all cases of the A matrix and any search restrictions. At a later point, the user can continue the run, using all or some of the previously generated cases.

BDX has the script commands **dump** and **selectimage** to perform dump and restore, and command line option **-r** and **-d** to specify the name of an output file to dump to, and an input file to restore from, respectively. The dump process requires the use of both the **dump** command and the **-d** command line option, and likewise the restore procedure requires **selectimage** and **-r**. The parameter **-r** requires the filename of a file dumped by *BDX*, but the parameter **-d** will default to the stub of the script filename with a **.dmp** extension. For example:

```
bdx run2.dat -r=run1.dmp -d
```

will execute the script in **run2.dat**, restoring the dump file **run1.dmp** and dumping the new state to **run2.dmp**. The command **dump** is usually placed at the end of the program, before **exit**, to indicate the intention to dump the state after everything has been generated. The command **dump** can also be placed in the middle of a program, if there is a need to halt the program before all **try**s are executed.

The command **selectimage** is used to restore a previously dumped image. When an existing file is specified in the *BDX* command with the **-r** option, the file is read in and the lines of script of the dumped image are executed again (ignoring any **try** or other 'work' commands), in order to recreate the state, including internal data on the BIBD as well as the images generated. Examples of internal information are the parameters of the script from which the dump file was created, the parameters of the BIBD being generated, restrictions on A and S and the search mode, whether it be plain, sorted or isomorph rejection. The **selectimage** command accepts most forms of the **<SUBSET>** parameter, as defined in *BDX*, which includes ranges like [7:9&12]

meaning images numbering 7, 8, 9 and 12 and ranges such as [2:11 by 3] meaning images of cases numbering 2, 5, 8 and 11. All other cases are skipped.

Example 2.1: Suppose we want to find a (9, 12, 4, 3, 1)-BIBD. To do this we might generate the first two rows of A and dump the results. Here is a possible script, which we will call **v9b12a.bdx** (see Fig. 2.1). The first three lines give the parameters (9, 12, 4, 3, 1) for the BIBD. These starting lines are necessary for *BDX* to accept the script.

```
9 rows of 4;
12 cols of 3;
lambda=1;
A[1,1:4]=1;
A[2,1]=1;
try row [1:2,1:12];
pass every 4 times;
print row [1:2,1:12];
dump;
exit
```

Figure 2.1: v9b12a.bdx

In order to reduce the size of the output, we reduce the number of solutions. We can guarantee solutions will be found as long as the eliminated cases are isomorphic to some generated cases. Since there are 4 ones in each row and solutions are isomorphic under column permutation, we can fix all 4 ones in the first row, restricting them to the first four positions. Similarly, each column contains 3 ones, so fixing a second one in the first column will not prevent a solution from being generated.

The `try` command instructs *BDX* to generate all possible ways to fill in the parts of the BIBD specified in this command. The `try` command can generate by row or column; in this case we use `try row` to direct *BDX* to fill in a row-by-row manner. The parameters `[1:2,1:12]` will generate the first two complete rows, there being 12 columns. There are still many solutions, so we only pass every fourth solution generated. This is done with the statement `pass every 4 times;`.

To have a visible output to record what has been dumped, we print the first two complete rows, using the same conventions as the `try` command. The dump command then creates a dump file with the name specified on the command line. We execute the script with the following command in the unix shell, which reads the script `v9b12a.bdx`, dumps to the default file and prints to `v9b12a.out`. The command sequence

```
bdx v9b12a.bdx -d > v9b12a.out
```

gives us two files. The first is **v9b12a.out**, as shown in Appendix A.1.1. The second, **v9b12a.dmp**, is a file consisting of the lines of the script from which it was dumped, along with images of the cases as tables of zeros and ones. The output file **v9b12a.out** shows that 56 cases of the first two rows of A have been generated and dumped, of which 14 are output. From the dump file, we can select any case from 1 to 56 to restore in the next script. In the script **v9b12b.bdx** (see Fig. 2.2)

```
9 rows of 4;
12 rows of 3;
lambda=1;
selectimage [1:9 by 4];
try row [3,1:12];
print row [1:3,1:12];
exit
```

Figure 2.2: v9b12b.bdx

we choose cases $1, 5$ and 9 and continue them by generating the third row of A. We execute the script as above, but in this case including the dump file we want to restore from instead of the `-d` option. The command:

```
bdx v9b12b.bdx -r=v9b12a.dmp > v9b12b.out
```

produces the output file `v9b12b.out` (see later in Appendix A.1.2). From the three images restored, case 1 can be extended to the third row in a unique way, and cases

3 and 5 each have three possible extensions.

Example 2.2: Here is a typical situation in which long expensive generation is interrupted by frequent dumps, preserving the partial run for later restore, as well as allowing several possible paths to be tried from the dump points.

In this example a user is building a $(27, 39, 13, 9, 4)$-BIBD. The first script, which the user calls `v27_1.bdx`, is listed in its log file in Appendix A.2.1. In it the user places several restrictions on S and A, requests several fills with `try` statements, and also requests isom testing. The results are then dumped to `v27_1.dmp`; lastly, a **pass every** statement allows only one out of every 50 solutions to be output to the log file. This means that the dump file gets all of the solutions, of which only a fraction is printed. The results, as shown in `v27_1.log`, are that 32 non-isomorphic partial solutions are dumped to `v27_1.dmp` and one solution is output. The ex[illegible]ution time was 25.5 seconds.

The second script, `v27_2.bdx` is listed in `v27_2.log` in Appendix A.2.2. In it the user restores the dump file `v27_1.dmp` from the previous script `v27_1.bdx`. This automatically restores all of statements from the script `v27_1.bdx`. This allows *BDX* to set up the state as it was at the time of the dump, including the parameters of the A matrix and `test isom` option. The statement `selectimage [1:32]` restores all 32 images of the partial solutions from the dump file. The user performs several further fills and dumps again. The results are that 2461 non-isomorphic partial solution are generated and dumped. This takes 37361.4 seconds or about 10 hours and 22 minutes.

The third script, `v27_3.bdx` again restores all 2461 images. Several restrictions on S are made, and then fills are performed. With the final `test isom` statement, 68 partial solutions are generated in 5492.5 seconds or about 1 hour and 31 minutes.

In this way, the user is able to generate a set of partial solutions, stop, study the results, make decisions about how to continue, and then to continue the generation from that point without restarting—certainly a very useful thing to be able to do after a 10 hour run. If necessary, the user could continue the run from **v27_1.dmp** or v27_2.dmp with different scripts than v27_2.bdx or v27_3.bdx.

2.3 Implementation

This section describes the implementation of the Dump/Restore utility. The Dump routine contains two main functions.

The first function, dump_setup(), is listed in its current form in Appendix B.1. This function is called with two parameters: the file name of the source, source_filename, and a string, param, which holds the text of the -d command line option of *BDX*, if any. The function dump_setup() is always called; if the parameter string is empty, the function returns immediately. If the parameter string is not empty, the next test determines whether a filename was specified with the -d option, and if so, copies the filename to the variable dump_filename. If no filename is specified, the dump filename gets the script filename, source_fname, with the rightmost extension replaced with ".dmp". given in the command line, as would be the case if the user were entering lines of script from the console, or if the resulting dump filename already exists, an error occurs. Otherwise, the dump file is opened and a message indicating the source filename and dump filename are output to the dump file. Finally, a function set_lex_echo() instructs *BDX* that each line read from the script file should be echoed to the dump file. This allows the Restore routine to later reconstruct the state of the run from the dump file.

The second Dump function is dump(), a current version of which is given in Appendix B.2. The dump() function is called whenever the dump command is processed by *BDX*. The function first checks whether a dump file was opened, and if not, it immediately returns. If the dump file is open, the current image number, which is held by a static counter initialized to 1, is printed in text form to the dump file. Then the function prints the values of the *A* matrix in text form, whether '0', '1,' or '.' for undefined.

In the current version, *BDX* prints a character to the dump file, indicating whether or not the output is to be compressed, before printing the image number. has two main functions which set up the restore conditions and perform the actual restore. The function `restore_setup()`, listed in its current version in Appendix B.3, accepts the text of the `-r` command line option. As with `dump_setup()`, `restore_setup()` is always called by *BDX*. For this reason the parameter string must be tested to verify whether it is empty.

Next, if the `-r` option was specified without a filename, an error occurs, since there is no default. If a filename is specified, it is stored in the variable `restore_fname`. A small function within the Restore module, `look_for_BEGINIMAGE()` is called to find the first line of first image in the restore file. This line marks the end of the restore file's echoed script, which is not compressed in the restore file. The function `restore_img()`, shown in its current version in Appendix B.4, handles the specifics of restoring an individual image. It has one parameter `img_num` that passes the image number as an integer. The function first tests whether the restore file is open, and otherwise returns immediately. It then scans for the number of the next image to be read from the file. If the image number cannot be found, as would be the case if a specified image number was greater than that of the last image in the restore file, then a warning is printed. The restore file is closed and the restore file pointer given a null value to identify it as closed, before the function returns. If the image number is found, the image is read in, element by element, and stored in the array corresponding to the A matrix. called to handle the restore process. The function `do_restore()` is first called by *BDX* when the command `selectimage` is encountered. It first checks if the restore file is still open. If not, it executes the current script command, and then returns. If the file is still open, it scans for the next image number. The function `restore_testop()` determines whether the next operation called by the restored script should be re-executed. Commands that produce data, such as `try`, or contain important information, such as `rows` and `cols`, return a true value. Operations that produce no data, such as printing, generating cases, dumping, and isomorph testing, return a false value.

2.4 Changes to Dump/Restore

In my original work, the `dump` procedure provided output of readable images, in tables of zeros and ones. This was later found to be space-inefficient. The images are currently compressed using bits to represent the elements of each row, as well as through line-encoding, giving approximately a tenfold reduction in the size of the dump file. If the routine is printing in compressed form, to which *BDX* is currently restricted, the image number is printed in binary, and a special function, `Compress()`, returns the entire matrix in line-encoded compressed form, which is then written to the dump file. For restoring, functions are called to decompress the binary line-encoded image and store it in the array.

Chapter 3

A Test of the (22,33,12,8,4)-BIBD Based on Greig's Criterion

It can be shown that the parameters of the (22, 33, 12, 8, 4)-BIBD are admissible, but it is not known at the time of writing whether this design exists. Although larger BIBDs have been successfully searched—as has, for example, the projective plane of order 10, which corresponds to a (111, 111, 11, 11, 1)-BIBD—the search for the (22, 33, 12, 8, 4)-BIBD is a more difficult problem. Therefore, any method of shortening the search for this BIBD will bring us closer to solving this problem. It was therefore proposed that a test pertaining to this BIBD be included in the *BDX* program. This test is based on Greig's Criterion and takes the form of a predicate test during the search routine in *BDX*.

3.1 Greig's Criterion

An important aspect of branch-and-bound search algorithms is their partial predicate tests. The sooner (that is, the higher in the search tree) that bad partial solutions can be eliminated, the fewer number of cases lower in the search tree that will have to be subsequently searched. Predicates that are fast to calculate, and are largely independent of other predicates in terms of the cases they throw out, are the most useful. A well-defined predicate takes better advantage of the properties of the specific problem to be solved. For the (22, 33, 12, 8, 4)-BIBD, there are several useful properties, including the inequalities shown in theorems below; these are inequalities on the entries of the S matrix.

Theorem 3.1 *For all* 2×2 *submatrices of* S *of the form:*

$$\begin{pmatrix} 8 & u \\ u & 8 \end{pmatrix}$$

$u \leq 4$.

Theorem 3.2 *For all* 3×3 *submatrices of* S *of the form:*

$$\begin{pmatrix} 8 & u & v \\ u & 8 & w \\ v & w & 8 \end{pmatrix}$$

$u + v \geq w$.

Theorem 3.3 *For all* 4×4 *submatrices of* S *of the form*

$$\begin{pmatrix} 8 & c & d & e \\ c & 8 & u & v \\ d & u & 8 & w \\ e & v & w & 8 \end{pmatrix}$$

$c + d + e + u + v + w \geq 10$.

For a proof of these theorems, please see [1, pp. 18-19].

3.2 Implementation

In *BDX*, the test based on Greig's criterion is known as the Hook-22 test. The reason for this name is that it is implemented as an external utility and is meant for BIBDs of size 22. It uses Theorems 3.2 and 3.3. From the completed entries of the S matrix, all principal submatrices of 3×3 and 4×4 elements are tested.

If any of these tests fail, a "BAD" result is returned immediately and the current case of the A matrix is thrown out. Otherwise, "GOOD" is returned, meaning that the test was successful and the current generated case is acceptable.

The main function of the test, `do_hook_hook22()` can be found in `hook.c` listed in Appendix C.1. It uses functions and global variables from `printsl.c` in Appendix C.2. First, the S matrix is generated, ensuring an updated version. This is done by the function `createS()` in `printsl.c`.

Next, the test based on Theorem 3.2 is performed. The array `sortstack`, which holds a list of completed elements of S, is scanned for rows or columns of the S matrix. Since S is symmetric, no distinction is made between rows and columns. Using nested `for` loops, every choice of three rows or columns is generated. To save time, the three rows or columns are generated in a fixed order, and then three tests for the three possible choices of the first row are made together. In other words, no sum of any two variables should be less than the third. If this test fails, a failure counter is incremented and the function returns "BAD". The test based on Theorem 3.3 makes use of the generated data of the previous test. The variable `uvw_sum` holds the sum of the variables u, v and w. The only new variables that have to be assigned are c, d, and e, which are added. If the new sum is less than 10, the test fails in the same way. Larry Thiel later found, however, that this test is not needed. The test based on Theorem 3.3 has therefore been disabled in the current version of *BDX*. Shown here is a proof found by Larry Thiel. It demonstrates that there is no case where a submatrix of S passes the test of Theorem 3.2 but fails the test of Theorem 3.3.

Theorem 3.4 *Let*

$$S_4 = \begin{pmatrix} 8 & c & d & e \\ c & 8 & u & v \\ d & u & 8 & w \\ e & v & w & 8 \end{pmatrix}$$

be a principal submatrix of S. If all of its 3×3 submatrices satisfy the condition of Theorem 3.2 then S_4 satisfies the condition of Theorem 3.3.

Before proving Theorem 3.4, we first establish a corollary of Theorem 3.2:

Corollary 3.1: Let

$$\begin{pmatrix} 8 & u & v \\ u & 8 & w \\ v & w & 8 \end{pmatrix}$$

be a submatrix of S. If $u = 0$ then $v = w$.

Proof: By interchanging columns 1 and 2, as well as rows 1 and 2 to preserve symmetry, v and w trade positions.

So we can say, if $u = 0$ then $w \geq v$. Therefore, $u = 0 \Rightarrow v = w$. **Q.E.D.**

We can now consider the proof of Theorem 3.4. This proof makes reference to the possible Types of each row in the S matrix. These Types are derived in [5, p. 78] and are reproduced in Table 3.1. The Types are defined by the vector $[b_0, \ldots, b_5]$ where b_i is the number of entries in a row of the S matrix with the value i. The values always refer to the off-diagonal entries of the S matrix since by the nature of the $(22, 33, 12, 8, 4)$ the diagonal entries of the S matrix have the value 8. For example, if a row of the S matrix is of Type 1, then in the off-diagonal entries of this row it

Type	b_0	b_1	b_2	b_3	b_4	b_5
1	0	0	12	16	4	0
2	0	1	9	19	3	0
3	0	2	6	22	2	0
4	1	0	6	24	1	0
5	0	3	3	25	1	0
6	0	0	11	19	1	1
7	0	1	8	22	0	1
8	0	4	0	28	0	0
9	1	1	3	27	0	0

Table 3.1: Distribution of values in S

has twelve '2's, sixteen '3's and four '4's. We only need to consider Types 1 to 4, as the others are proved to be invalid in [5, pp. 79-83].

Proof: (of Theorem 3.4)
Let $C_1, \ldots, C_4$ be the columns of A corresponding to the entries of the given 4×4 submatrix of S. Then,

$$\begin{aligned} c &= \langle C_1, C_2 \rangle \\ d &= \langle C_1, C_3 \rangle \\ e &= \langle C_1, C_4 \rangle \\ u &= \langle C_2, C_3 \rangle \\ v &= \langle C_2, C_4 \rangle \\ w &= \langle C_3, C_4 \rangle . \end{aligned}$$

Since Theorem 3.2 applies to all 4 submatrices of the matrix S_4 in Theorem 3.4, most variables in the latter theorem are also interchangeable. Therefore the variable c, which is assigned a value in each case, is arbitrary and is therefore assumed to have the minimum value.

The proof of 3.4 is divided into 6 cases.

Cases 1 and 2 are both for $c = 0$. The only intersection pattern with $b_0 > 0$ is Type 4, which has $b_0 = 1$ and $b_1 = 0$. Since the next nonzero b_i is b_2, all the entries in the row are at least 2. In other words, $d, e, u, v \geq 2$.

Case 1: $c = 0$, and at least one of $d, e, u, v > 2$.
If one of $d, e, u, v > 2$ then Corollary 3.1 implies that another of them is also > 2, since $c = 0$. Therefore $d + e + u + v \geq 10$ and Theorem 3.3 holds.

Case 2: $c = 0$ and $d = e = u = v = 2$.
In other words, $c = \langle C_1, C_2 \rangle = 0$ and $d = \langle C_1, C_3 \rangle = e = \langle C_1, C_4 \rangle = u = \langle C_2, C_3 \rangle = v = \langle C_2, C_4 \rangle = 2$. Of the 22 rows, 8 are filled in C_1, of which 2 are filled in each of C_3 and C_4. This is described in Table 3.2. The remaining 4 *ones* of C_3 and C_4 must occupy the remaining 6 unfilled rows in C_1 and C_2. So $\langle C_3, C_4 \rangle = w \geq 2$. Therefore $c + d + e + u + v + w \geq 10$ and Theorem 3.3 holds.

r_i	# rows	Number of 1s in A			
		C_1	C_2	C_3	C_4
1..8	8	8	0	2	2
9..16	8	0	8	2	2
17..22	6	0	0	4	4

Table 3.2: Case 2 of Proof of Theorem 3.4

Cases 1 and 2 cover all possibilities where $c = 0$. We now assume $c \geq 1$. Cases 3 to 5 are for $c = 1$. Since c is assumed to be the minimal off-diagonal entry in S_4, all of the other off-diagonal entries are ≥ 1. By permuting the rows if necessary, we can assume that d is the smallest among d, e, u and v. Cases 3 and 4 are for $d = 1$ and Case 5 is for $d > 1$. If $d = 1$, then from Theorem 3.2, $c + d \geq u$, so $u = 1$ or 2. Therefore we have the following division with cases 3, 4 and 5:

Case 3: $c = d = 1$ and $u = 1$.

Please refer to Tables 3.3 and 3.4. We have $c = \langle C_1, C_2 \rangle = d = \langle C_1, C_3 \rangle = u =$

$\langle C_2, C_3 \rangle = 1$. Therefore C_1, C_2 and C_3 each share a common *one*. Let C_1 have its 8 *ones* in $r_{1..8}$, let r_1 be the row with the common *one* of C_1 and C_2, let $r_{2..8}$ be the 7 remaining rows holding the 7 remaining *ones* of C_1, let $r_{9..15}$ be the 7 remaining rows holding the 7 remaining *ones* of C_2. If C_3 contains a *one* in r_1 then C_3 will have no *ones* in $r_{2..15}$ where there are *ones* in C_1 and C_2. C_3 will have its remaining 7 *ones* in $r_{16..22}$. The configuration is shown in Table 3.4.

On the other hand, if r_1 has no *one* in C_3, then it must have 1 *one* in each of $r_{2..8}$ and $r_{9..15}$ so that it shares a common *one* with both C_2 and C_3. But then C_3 has only 6 remaining *ones* in $r_{16..21}$, and we will say the last row with no *one* in C_3 is r_{22}. This configuration is shown in Table 3.3.

In either case, the column C_4 must contain e *ones* in $r_{1..8}$ since that is where the *ones* of C_1 are. There are also at most v *ones* in rows $r_{9..15}$ of C_4, since $v = \langle C_2, C_4 \rangle$. There can also be less than v *ones* in this region, since not all *ones* of C_2 must be in $r_{9..15}$.

We call this value $x \leq v$. The same conditions apply to $r_{16..21}$ where C_4 contains $y \leq w$ *ones*. Finally, we have at most 1 remaining *one* in r_{22}, since $e + x + y \leq 8$, so we say $z \leq 1$. From this, we have,

$$e + x + y + z = 8.$$

Since $z = 0$ or 1,

$$e + x + y \geq 7.$$

We defined $x \leq v$ and $y \leq w$ so this gives

$$e + v + w \geq 7.$$

Since $c + d + u = 3$, we have

$$(c + d + u) + (e + v + w) \geq 10.$$

Therefore, Theorem 3.3 holds.

r_i	# rows	Number of 1s in A			
		C_1	C_2	C_3	C_4
1	1	1	1	0	e
2..8	7	7	0	6	
9..15	7	0	7	1	$x \leq v$
16..21	6	0	0	6	$y \leq w$
22	1	0	0	0	$z = 1$

Table 3.3: Case 3a

r_i	# rows	Number of 1s in A			
		C_1	C_2	C_3	C_4
1	1	1	1	1	e
2..8	7	7	0	0	
9..15	7	0	7	0	$x \leq v$
16..22	7	0	0	7	$y \leq w$
	0	0	0	0	$z = 0$

Table 3.4: Case 3b

Case 4: $c = d = 1$ and $u = 2$.

This case is very similar to case 3.2. C_1 and C_2 share common *ones* in 1 row, r_1. If C_3 does not share this common *one*, as in Table 3.5, then it needs 1 *one* in $r_{2..8}$ for $c = 1$ and 2 *ones* in $r_{9..15}$ for $u = 2$. The remaining 5 *ones* of C_3 are in $r_{16..20}$ and $r_{21..22}$ are 0. If C_3 also has a *one* in r_1, as in Table 3.6, then there need be no *ones* in $r_{2..8}$ of C_3. Since $u = \langle C_2, C_3 \rangle = 2$ there is 1 *one* in $r_{9..15}$. The remaining 6 *ones* are in $r_{16..21}$ and r_{22} is 0. Regardless of which subcase we are dealing with, C_4 still has e *ones* in $r_{1..8}$, $x \leq v$ *ones* in $r_{9..15}$, and $y \leq w$ *ones* in $r_{16..20}$ or $r_{16..21}$ for subcases a and b respectively. The remaining region has 1 or 2 rows, depending on the subcase with at most $z = 2$ *ones*. So $z \leq 2$.

Similar to the previous case we have:

$$e + x + y + z \;=\; 8.$$

Since $x \leq v$ and $y \leq w$ and $z = 0$ or 1,

$$e + v + w \geq 6.$$

And since $c + d + u = 4$,

$$c + d + e + u + v + w \geq 10.$$

Therefore, Theorem 3.3 holds.

r_i	# rows	Number of 1s in A			
		C_1	C_2	C_3	C_4
1	1	1	1	0	e
2..8	7	7	0	1	
9..15	7	0	7	2	$x \leq v$
16..20	5	0	0	5	$y \leq w$
21..22	2	0	0	0	$z = 2$

Table 3.5: Case 4a

r_i	# rows	Number of 1s in A			
		C_1	C_2	C_3	C_4
1	1	1	1	1	e
2..8	7	7	0	0	
9..15	7	0	7	1	$x \leq v$
16..21	6	0	0	6	$y \leq w$
22	1	0	0	0	$z = 1$

Table 3.6: Case 4b

Case 5: $c = 1$ and $d, e, u, v \geq 2$.

Then $c + d + e + u + v \geq 9$ and $w \geq 1$ since c is arbitrary and minimal.

Case 6: $c, d, e, u, v, w \geq 2$.

We have $c + d + e + u + v + w \geq 10$ automatically.

Therefore, in all cases, Theorem 3.3 will never fail and is thus redundant. **Q.E.D.**

Chapter 4

Conclusion

In this report, I describe two utilities, which I have implemented, that allow BDX users to generate BIBDs with greater control and efficiency. The first problem was that unless a good search strategy were used, a complete search would take a very long time. A partial search script could give information, but there was no available way of continuing the search once the script ended. The other problem is the search for the (22, 33, 12, 8, 4)-BIBD.

The first utility, *Dump/Restore*, addresses the first problem by allowing a user to run a partial generation, at the end of which one can save all partial solutions for a later restore and continuation. This can help in finding the most time-efficient method for generating the entire BIBD, whether in sequence or by allowing several machines to search, in parallel, subsets of the cases. In the long term, a dump file can be distributed to allow others to study the progress of a particular search, and perhaps to offer better methods for continuing the search.

The second utility, *Greig's Test*, provides a good partial predicate test for the second problem: searching for the (22, 33, 12, 8, 4)-BIBD, which has yet to be found.

Bibliography

[1] Ding, Yuan. *A Study of Balanced Incomplete Block Designs*, thesis, Department of Mathematics and Statistics, Concordia University, 1993.

[2] Greig, M. *An Improvement to Connor's Criterion.* Unpublished.

[3] Hall, M. Jr. *Combinatorial Theory*, 2nd Ed., New York: John Wiley, 1986.

[4] Hall, M. Jr., Roth, R. and Van Rees, J., Vandstone, S.A., On Designs (22,33,12,8,4), *Journal of Conbinatorial Theory, Series A*, **47**, 1988, 157-175.

[5] Hamada, N. and Kobayashi, Y. "On the Block Structure of BIB designs with parameters $v = 22$, $b = 33$, $r = 12$, $k = 8$ and $\lambda = 4$," *Journal of Combinatorical Theory, Series A*, **24**, 1978, 75-83.

[6] Thiel, Larry and Fink, Nathaniel. *BDX Reference Guide.* Department of Computer Science, Concordia University, 1994.

Appendix A

Output of Example of Dump/Restore

A.1 Dump/Restore Example 1

A.1.1 v9b12a.out

```
Data will be dumped to: v9b12a.dmp

       1 9 rows of 4;
       2 12 cols of 3;
       3 lambda=1;
       4 A[1,1:4]=1;
       5 A[2,1]=1;
       6 try row [1:2,1:12];
       7 pass every 8 times;
       8 print row [1:2,1:12];
       9 dump;
      10 exit
End of scan phase
   0 errors and    0 warnings issued.
Scan time was    0.050 Seconds

print request from line 8
              111
     123456789012

   1 111100000000
   2 100011100000

print request from line 8
              111
      123456789012

    1 111100000000
    2 100010100100

print request from line 8
                 111
      123456789012

    1 111100000000
    2 100010001010

print request from line 8
                 111
      123456789012
    1 111100000000
    2 100001100010

print request from line 8
                 111
      123456789012

    1 111100000000
    2 100001001001

print request from line 8
                 111
      123456789012

    1 111100000000
    2 100000101100

print request from line 8
                 111
      123456789012

    1 111100000000
    2 100000011001

Printcount at time =    0.010 seconds
Final printcount
Cmd   6 (          try)               IN:      1/1.0e+00 OUT:     56/5.6e+01.
 ROW       ones             fills       ROW       ones            fills
   1        1/1.0e+00       1/1.0e+00   2      56/5.6e+01       56/5.6e+01
Cmd   7 (   passevery)               IN:     56/5.6e+01 OUT:      7/4.9e+01.
Cmd  10 (        exit)               IN:      7/4.9e+01 OUT:      0/0.0e+00.
 High water mark of stack used 68 out of 300000 bytes
Execute time was    0.010 Seconds
End of bdx.
```

A.1.2 v9b12b.out

```
v9b12a.dmp will be restored.
  1    1 % source read from v9b12a.bdx, dumped to v9b12a.dmp, Tue Aug 29 15
:18:32 1995
  1    2 9 rows of 4;
  1    3 12 cols of 3;
  1    4 lambda=1;
  1    5 A[1,1:4]=1;
  1    6 A[2,1]=1;
  1    7 try row [1:2,1:12];
  1    8 pass every 8 times;
  1    9 print row [1:2,1:12];
  1   10 dump;
  1   11 exit
      12 % 9 rows of 4;
      13 % 12 cols of 3;
      14 % lambda=1;
      15 selectimage [1:5 by 2];
      16 A[3,8:10]=1;
      17 try row [3,1:12];
      18 print row [1:3,1:12];
      19 exit
End of scan phase
   0 errors and    0 warnings issued.
Scan time was    0.030 Seconds

******** Restoring Image #1:

print request from line 18
             111
   123456789012

  1 111100000000
  2 100011100000
  3 100000011100

******** Restoring Image #.

print request from line 18
             111
   123456789012

  1 111100000000
  2 100010001010
  3 010000011100

print request from line 18
             111
   123456789012

  1 111100000000
  2 100010001010
```

```
   3 001000011100

print request from line 18
               111
      123456789012

   1 111100000000
   2 100010001010
   3 000100011100

******** Restoring Image #5:

print request from line 18
               111
      123456789012

   1 111100000000
   2 100001001001
   3 010000011100

print request from line 18
               111
      123456789012

   1 111100000000
   2 100001001001
   3 001000011100

print request from line 18
               111
      123456789012

   1 111100000000
   2 100001001001
   3 000100011100

Printcount at time =    0.000 seconds
Final printcount
Cmd  12 (select image)              IN:      3/3.0e+00 OUT:      3/3.0e+00.
Cmd  16 (         setA)             IN:      3/3.0e+00 OUT:      3/3.0e+00.
Cmd  17 (          try)             IN:      3/3.0e+00 OUT:      7/7.0e+00.
 ROW      ones              fills       ROW      ones              fills

   3       5/5.0e+00       7/7.0e+00
Cmd  19 (         exit)             IN:      7/7.0e+00 OUT:      0/0.0e+00.
 High water mark of stack used 340 out of 300000 bytes
Execute time was    0.010 Seconds
End of bdx.
```

A.2 Dump/Restore Example 2

A.2.1 v27_1.log

```
Data will be dumped to: v27_1.dmp

       1 27 row of 13;
       2 39 col of 9;
        3 lambda = 4;
Region may be printed as ROW[28]
Region may be printed as COL[40]
       4 init S={0&3};
       5 S[1:13,1:13]=3;
       6 under col [1:13];
Region may be printed as ROW[29]
       7 in rows [1:26] type = 4;
       8 end under;
       9 a[27,1:13]=1;
      10 a[1:8, 1] = 1;
      11 %second attempt, first complete top 8 rows, then 6 more,
      12 %then 4 more and then the complete first 13 columns
      13 %
      14 % first fill in row 1
      15 a[1, 2:4] = 1;
      16 % col 2 up to row 8
      17 try col sorting [1:8, 2]; % expect 1 soln
      18 % col 3 and 4 up to row 8
      19 try col sorting [1:8, 3];
      20 try col sorting [1:8, 4];
      21 test isom; % expect  3 soln
      22 %fill in row 5, in cols 5 to 7
      23 a[5, 5:7] = 1;
      24 %fill in 3 extra zero in row 1
      25 a[1, 5:7]= 0;
      26 try col sorting [1:8, 5];
      27 try col sorting [1:8, 6];
      28 test isom;
      29 try col sorting [1:8, 7];
      30 test isom;
      31 try row sorting [8, 1:13];
      32 try row [1&5, 1:13]; % fill in rest of rows 1 and 5 with zeros
      33 try row sorting [2, 1:13];
      34 try row sorting [3, 1:13];
      35 try row sorting [4, 1:13];
      36 try row sorting [6, 1:13];
      37 try row sorting [7, 1:13];
      38 test isom;
      39 dump;
      40 pass every 50 times;
      41 printcol [1:8$&27, 1:13];
      42 exit
End of scan phase
```

```
   0 errors and    0 warnings issued.
 Scan time was    0.258 Seconds
 At start of execute phase bvcount, pvcount, pmcount, bmcount, pgcount
      0    11     0     0    27

 print request from line 41
                  2
      12345678 7

    1 11111111 1
    2 11000000 1
    3 11000000 1
    4 11000000 1
    5 00101000 1
    6 00101000 1
    7 00101000 1
    8 00010001 1
    9 00010001 1
   10 00010001 1
   11 00000110 1
   12 00000110 1
   13 00000110 1

Printcount at time =   25.494 seconds
Final printcount
Cmd  17 (         try)                  IN:      1/1.0e+00 OUT:       1/1.0e+00.
 COL       ones               fills      COL       ones               fills
   2       2/2.0e+00          1/1.0e+00
Cmd  19 (         try)                  IN:      1/1.0e+00 OUT:       2/2.0e+00.
 COL       ones               fills      COL       ones               fills
   3       3/3.0e+00          2/2.0e+00
Cmd  20 (         try)                  IN:      2/2.0e+00 OUT:       5/5.0e+00.
 COL       ones               fills      COL       ones               fills
   4       7/7.0e+00          5/5.0e+00
Cmd  21 (  test isom)                   IN:      5/5.0e+00 OUT:       3/3.0e+00.
Context date=20( 2),    3 certs and     2 non-certs, using   0.175 seconds
Cmd  23 (        setA)                  IN:      3/3.0e+00 OUT:       3/3.0e+00.
Cmd  25 (        setA)                  IN:      3/3.0e+00 OUT:       3/3.0e+00.
Cmd  26 (         try)                  IN:      3/3.0e+00 OUT:       8/8.0e+00.
 COL       ones               fills      COL       ones               fills
   5       8/8.0e+00          8/8.0e+00
Cmd  27 (         try)                  IN:      8/8.0e+00 OUT:      28/2.8e+01.
 COL       ones               fills      COL       ones               fills
   6       28/2.8e+01         28/2.8e+01
Cmd  28 (  test isom)                   IN:     28/2.8e+01 OUT:      13/1.3e+01.
Context date=27( 4),   13 certs and    15 non-certs, using   0.533 seconds
Cmd  29 (         try)                  IN:     13/1.3e+01 OUT:      48/4.8e+01.
 COL       ones               fills      COL       ones               fills
   7       47/4.7e+01         48/4.8e+01
Cmd  30 (  test isom)                   IN:     48/4.8e+01 OUT:      15/1.5e+01.
Context date=29( 7),   15 certs and    33 non-certs, using   0.729 seconds
Cmd  31 (         try)                  IN:     15/1.5e+01 OUT:      15/1.5e+01.
 ROW       ones               fills      ROW       ones               fills
```

```
   8      59/5.9e+01      15/1.5e+01
Cmd  32 (          try)                  IN:     15/1.5e+01 OUT:     15/1.5e+01.
 ROW      ones            fills           ROW      ones              fills
   1      15/1.5e+01      15/1.5e+01      5      15/1.5e+01      15/1.5e+01
Cmd  33 (          try)                  IN:     15/1.5e+01 OUT:     24/2.4e+01.
 ROW      ones            fills           ROW      ones              fills
   2      34/3.4e+01      24/2.4e+01
Cmd  34 (          try)                  IN:     24/2.4e+01 OUT:     70/7.0e+01.
 ROW      ones            fills           ROW      ones              fills
   3     112/1.1e+02      70/7.0e+01
Cmd  35 (          try)                  IN:     70/7.0e+01 OUT:    265/2.6e+02.
 ROW      ones            fills           ROW      ones              fills
   4     358/3.6e+02     265/2.6e+02
Cmd  36 (          try)                  IN:    265/2.6e+02 OUT:    675/6.7e+02.
 ROW      ones            fills           ROW      ones              fills
   6     797/8.0e+02     675/6.7e+02
Cmd  37 (          try)                  IN:    675/6.7e+02 OUT:    675/6.7e+02.
 ROW      ones            fills           ROW      ones              fills
   7     675/6.7e+02     675/6.7e+02
Cmd  38 (    test isom)                  IN:    675/6.7e+02 OUT:     32/3.2e+01.
Context date=37( 9),    32 certs and  643 non-certs, using  20.985 seconds
Cmd  40 (    passevery)                  IN:     32/3.2e+01 OUT:      1/1.0e+00.
Cmd  42 (         exit)                  IN:      1/1.0e+00 OUT:      0/0.0e+00.
 High water mark of stack used 980 out of 300000 bytes
Execute time was    25.502 Seconds
At end of run bvcount, pvcount, pmcount, bmcount, pgcount
   388   386    26    12    39
End of bdx.
```

A.2.2 v27_2.log

```
v27_1.dmp will be restored.


Data will be dumped to: v27_2.dmp

  1    1 % source read from v27_1.bdx, dumped to v27_1.dmp, Mon Mar 28 10:
10:17 1994
  1    2 27 row of 13;
  1    3 39 col of 9;'
  1    4 lambda = 4;
Region may be printed as ROW[28]
Region may be printed as COL[40]
  1    5 init S={0&3};
  1    6 S[1:13,1:13]=3;
  1    7 under col [1:13];
Region may be printed as ROW[29]
  1    8 in rows [1:26] type = 4;
  1    9 end under;
  1   10 a[27,1:13]=1;
  1   11 a[1:8, 1] = 1;
  1   12 %second attempt, first complete top 8 rows, then 6 more,
1   %then 4 more and then the complete first 13 columns
1   %
1   % first fill in row 1
1   a[1, 2:4] = 1;
1   % col 2 up to row 8
1   try col sorting [1:8, 2]; % expect 1 soln
1   % col 3 and 4 up to row 8
1   try col sorting [1:8, 3];
1   try col sorting [1:8, 4];
1   test isom; % expect  3 soln
1   %fill in row 5, in cols 5 to 7
1   a[5, 5:7] = 1;
1   %fill in 3 extra zero in row 1
1   a[1, 5:7]= 0;
1   try col sorting [1:8, 5];
1   try col sorting [1:8, 6];
1   test isom;
1   try col sorting [1:8, 7];
1   test isom;
1   try row sorting [8, 1:13];
1   try row [1&5, 1:13]; % fill in rest of rows 1 and 5 with zeros
1   try row sorting [2, 1:13];
1   try row sorting [3, 1:13];
1   try row sorting [4, 1:13];
1   try row sorting [6, 1:13];
1   try row sorting [7, 1:13];
1   test isom;
1   dump;
1   pass every 50 times;
1   printcol [1:8$&27, 1:13];
1   exit
    selectimage [1:32];
    % put in part 2 (in rows 9 to 26]
    %
    a[9:14, 2] = 1;
    try col [9:26, 1&2]; % fill in col 1 and 2 with zeros
    try col sorting [9:26, 3]; % one choice each
    try col sorting [9:26, 4];
    try col sorting [9:26, 5];
    test isom;
    try col sorting [9:26, 6];
    test isom;
    try col sorting [9:26, 7];
    test isom;
    try col sorting [9:26, 8];
    test isom;
    try col sorting [9:26, 11];
    try col sorting [9:26, 10];
    try col sorting [9:26, 9];
    try col sorting [9:26, 12];
    try col [9:26, 13];
    test isom;
    dump;
```

```
     66 exit
End of scan phase
   0 errors and    0 warnings issued.
Scan time was    0.414 Seconds
At start of execute phase bvcount, pvcount, pmcount, bmcount, pgcount
     0    16     0     0    45

******** Restoring Image #1:


******** Restoring Image #2:

...........................

******** Restoring Image #31:

******** Restoring Image #32:
Printcount at time = 37361.400 seconds
Final printcount
Cmd  43 (select image)                 IN:     32/3.2e+01 OUT:     32/3.2e+01.
Cmd  47 (         setA)                IN:     32/3.2e+01 OUT:     32/3.2e+01.
Cmd  48 (          try)                IN:     32/3.2e+01 OUT:     32/3.2e+01.
 COL      ones            fills      COL      ones            fills
   1       0/0.0e+00     32/3.2e+01   2     32/3.2e+01     32/3.2e+01
Cmd  49 (          try)                IN:     32/3.2e+01 OUT:     32/3.2e+01.
 COL      ones            fills      COL      ones            fills
   3     192/1.9e+02     32/3.2e+01
Cmd  50 (          try)                IN:     32/3.2e+01 OUT:     38/3.8e+01.
 COL      ones            fills      COL      ones            fills
   4     174/1.7e+02     38/3.8e+01
Cmd  51 (          try)                IN:     38/3.8e+01 OUT:     73/7.3e+01.
 COL      ones            fills      COL      ones            fills
   5     362/3.6e+02     73/7.3e+01
Cmd  52 (   test isom)                 IN:     73/7.3e+01 OUT:     66/6.6e+01.
Context date=51( 7),   66 certs and   7 non-certs, using   9.277 seconds
Cmd  53 (          try)                IN:     66/6.6e+01 OUT:    319/3.2e+02.
 COL      ones            fills      COL      ones            fills
   6    1239/1.2e+03    319/3.2e+02
Cmd  54 (   test isom)                 IN:    319/3.2e+02 OUT:    209/2.1e+02.
Context date=53( 9),  209 certs and  110 non-certs, using  17.786 seconds
Cmd  55 (          try)                IN:    209/2.1e+02 OUT:   2281/2.3e+03.
 COL      ones            fills      COL      ones            fills
   7    5135/5.1e+03   2281/2.3e+03
Cmd  56 (   test isom)                 IN:   2281/2.3e+03 OUT:   1269/1.3e+03.
Context date=55(11), 1269 certs and 1012 non-certs, using  65.974 seconds
Cmd  57 (          try)                IN:   1269/1.3e+03 OUT:  14128/1.4e+04.
 COL      ones            fills      COL      ones            fills
   8   28666/2.9e+04  14128/1.4e+04
Cmd  58 (   test isom)                 IN:  14128/1.4e+04 OUT:  13553/1.4e+04.
Context date=57(13), 13553 certs and  575 non-certs, using 224.455 seconds
Cmd  59 (          try)                IN:  13553/1.4e+04 OUT:  40825/4.1e+04.
 COL      ones            fills      COL      ones            fills
  11  109633/1.1e+05  40825/4.1e+04
```

```
Cmd  60 (         try)                IN:  40825/4.1e+04 OUT: 189415/1.9e+05.
 COL       ones              fills         COL       ones               fills
  10  623791/6.2e+05  189415/1.9e+05
Cmd  61 (         try)                IN: 189415/1.9e+05 OUT: 179701/1.8e+05.
 COL       ones              fills         COL       ones               fills
   9  612151/6.1e+05  179701/1.8e+05
Cmd  62 (         try)                IN: 179701/1.8e+05 OUT: 185178/1.9e+05.
 COL       ones              fills         COL       ones               fills
  12  189580/1.9e+05  185178/1.9e+05
Cmd  63 (         try)                IN: 185178/1.9e+05 OUT: 185178/1.9e+05.
 COL       ones              fills         COL       ones               fills
  13   56049/5.6e+04  185178/1.9e+05
Cmd  64 (   test isom)                IN: 185178/1.9e+05 OUT:    2461/2.5e+03.
Context date=63(15), 2461 certs and 182717 non-certs, using 32244.975 seconds
Cmd  66 (        exit)                IN:    2461/2.5e+03 OUT:       0/0.0e+00.
 High water mark of stack used 3784 out of 300000 bytes
Execute time was 37361.408 Seconds
At end of run bvcount, pvcount, pmcount, bmcount, pgcount
   498    390    28    13    59
End of bdx.
```

A.2.3 v27_3.log

```
v27_2.dmp will be restored.

Data will be dumped to: v27_3.dmp

  1    1 % source read from v27_2.bdx, dumped to v27_2.dmp, Mon Mar 28 15:
13:03 1994
  1    2 % source read from v27_1.bdx, dumped to v27_1.dmp, Mon Mar 28 10:
10:17 1994
  1    3 27 row of 13;
  1    4 39 col of 9;'
  1    5 lambda = 4;
Region may be printed as ROW[28]
Region may be printed as COL[40]
  1    6 init S={0&3};
  1    7 S[1:13,1:13]=3;
  1    8 under col [1:13];
Region may be printed as ROW[29]
  1    9 in rows [1:26] type = 4;
  1   10 end under;
  1   11 a[27,1:13]=1;
  1   12 a[1:8, 1] = 1;
  1   13 %second attempt, first complete top 8 rows, then 6 more,
  1   14 %then 4 more and then the complete first 13 columns
  1   15 %
  1   16 % first fill in row 1
  1   17 a[1, 2:4] = 1;
  1   18 % col 2 up to row 8
  1   19 try col sorting [1:8, 2]; % expect 1 soln
  1   20 % col 3 and 4 up to row 8
```

```
1   try col sorting [1:8, 3];
1   try col sorting [1:8, 4];
1   test isom; % expect  3 soln
1   %fill in row 5, in cols 5 to 7
1   a[5, 5:7] = 1;
1   %fill in 3 extra zero in row 1
1   a[1, 5:7]= 0;
1   try col sorting [1:8, 5];
1   try col sorting [1:8, 6];
1   test isom;
1   try col sorting [1:8, 7];
1   test isom;
1   try row sorting [8, 1:13];
1   try row [1&5, 1:13]; % fill in rest of rows 1 and 5 with zeros
1   try row sorting [2, 1:13];
1   try row sorting [3, 1:13];
1   try row sorting [4, 1:13];
1   try row sorting [6, 1:13];
1   try row sorting [7, 1:13];
1   test isom;
1   dump;
1   pass every 50 times;
1   printcol [1:8$&27, 1:13];
1   exit
    selectimage [1:2461];
    try col [1:27, 1:13];
    % continue on the rest of the columns
    % first define the parallel classes
    %
    S[1&14&15, 1&14&15] = 0;
    S[2&16&17, 2&16&17] = 0;
    S[3&18&19, 3&18&19] = 0;
    S[4&20&21, 4&20&21] = 0;
    S[5&22&23, 5&22&23] = 0;
    S[6&24&25, 6&24&25] = 0;
    S[7&26&27, 7&26&27] = 0;
    S[8&28&29, 8&28&29] = 0;
    S[9&30&31, 9&30&31] = 0;
    S[10&32&33, 10&32&33] = 0;
    S[11&34&35, 11&34&35] = 0;
    S[12&36&37, 12&36&37] = 0;
    S[13&38&39, 13&38&39] = 0;
    %now define all the 3's, only need to define upper triangle
    S[ 1, 16:39] = 3;
    S[ 2, 14:15&18:39] = 3;
    S[ 3, 14:17&20:39] = 3;
    S[ 4, 14:19&22:39] = 3;
    S[ 5, 14:21&24:39] = 3;
    S[ 6, 14:23&26:39] = 3;
    S[ 7, 14:25&28:39] = 3;
    S[ 8, 14:27&30:39] = 3;
    S[ 9, 14:29&32:39] = 3;
    S[10, 14:31&34:39] = 3;
```

```
      74 S[11, 14:33&36:39] = 3;
      75 S[12, 14:35&38:39] = 3;
      76 S[13, 14:37] = 3;
      77 S[14&15, 16:39] = 3;
      78 S[16&17, 18:39] = 3;
      79 S[18&19, 20:39] = 3;
      80 S[20&21, 22:39] = 3;
      81 S[22&23, 24:39] = 3;
      82 S[24&25, 26:39] = 3;
      83 S[26&27, 28:39] = 3;
      84 S[28&29, 30:39] = 3;
      85 S[30&31, 32:39] = 3;
      86 S[32&33, 34:39] = 3;
      87 S[34&35, 36:39] = 3;
      88 S[36&37, 38:39] = 3;
      89 %
      90 a[1, 22&24&26&28&30&32&34&36&38] = 1;
      91 try row [27, 1:39]; % expect 1 solution
      92 try col sorting [2:26, 38];
      93 try col sorting [2:26, 36];
      94 test isom;
      95 try col sorting [2:26, 34];
      96 test isom;
      97 try col sorting [2:26, 32];
      98 test isom;
      99 try col sorting [2:26, 30];
     100 test isom;
     101 try col sorting [2:26, 28];
     102 test isom;
     103 try col sorting [2:26, 26];
     104 test isom;
     105 try col sorting [2:26, 24];
     106 test isom;
     107 try col sorting [2:26, 22];
     108 test isom;
     109 try row sorting [8, 20&21];
     110 try col [2:26, 20&21];
     111 try row sorting [8, 18&19];
     112 try col [2:26, 18&19];
     113 try row sorting [8, 16:17];
     114 try col [2:26, 16&17];
     115 try row sorting [9, 14&15];
     116 try col [2:26, 14&15];
     117 try row [1:27, 1:39];
     118 test isom;
     119 dump;
     120 exit
End of scan phase
   0 errors and     0 warnings issued.
Scan time was    0.820 Seconds
At start of execute phase bvcount, pvcount, pmcount, bmcount, pgcount
     0    20     0     0    90
```

```
******** Restoring Image #1:


******** Restoring Image #2:

.........................

******** Restoring Image #2460:


******** Restoring Image #2461:

Printcount at time = 5492.500 seconds
Final printcount
Cmd  44 (select image)                 IN:   2461/2.5e+03 OUT:   2461/2.5e+03.
Cmd  46 (          try)                IN:   2461/2.5e+03 OUT:   2461/2.5e+03.
 COL       ones             fills       COL       ones             fills
   1    2461/2.5e+03    2461/2.5e+03   8        0/0.0e+00    2461/2.5e+03
   2    2461/2.5e+03    2461/2.5e+03   9        0/0.0e+00    2461/2.5e+03
   3    2461/2.5e+03    2461/2.5e+03  10        0/0.0e+00    2461/2.5e+03
   4    2461/2.5e+03    2461/2.5e+03  11        0/0.0e+00    2461/2.5e+03
   5       0/0.0e+00    2461/2.5e+03  12        0/0.0e+00    2461/2.5e+03
   6       0/0.0e+00    2461/2.5e+03  13        0/0.0e+00    2461/2.5e+03
   7       0/0.0e+00    2461/2.5e+03
Cmd  90 (         setA)                IN:   2461/2.5e+03 OUT:   2461/2.5e+03.
Cmd  91 (          try)                IN:   2461/2.5e+03 OUT:   2461/2.5e+03.
 ROW       ones             fills       ROW       ones             fills
  27    2461/2.5e+03    2461/2.5e+03
Cmd  92 (          try)                IN:   2461/2.5e+03 OUT:   2577/2.6e+03.
 COL       ones             fills       COL       ones             fills
  38   43501/4.4e+04    2577/2.6e+03
Cmd  93 (          try)                IN:   2577/2.6e+03 OUT:   2221/2.2e+03.
 COL       ones             fills       COL       ones             fills
  36   25392/2.5e+04    2221/2.2e+03
Cmd  94 (   test isom)                 IN:   2221/2.2e+03 OUT:   1401/1.4e+03.
Context date=93( 7), 1401 certs and  820 non-certs, using 129.671 seconds
Cmd  95 (          try)                IN:   1401/1.4e+03 OUT:   1252/1.3e+03.
 COL       ones             fills       COL       ones             fills
  34    8678/8.7e+03    1252/1.3e+03
Cmd  96 (   test isom)                 IN:   1252/1.3e+03 OUT:   1009/1.0e+03.
Context date=95(10), 1009 certs and  243 non-certs, using  71.603 seconds
Cmd  97 (          try)                IN:   1009/1.0e+03 OUT:    721/7.2e+02.
 COL       ones             fills       COL       ones             fills
  32    4164/4.2e+03     721/7.2e+02
Cmd  98 (   test isom)                 IN:    721/7.2e+02 OUT:    672/6.7e+02.
Context date=97(13),  672 certs and  49 non-certs, using  29.292 seconds
Cmd  99 (          try)                IN:    672/6.7e+02 OUT:    645/6.4e+02.
 COL       ones             fills       COL       ones             fills
  30    1169/1.2e+03     645/6.4e+02
Cmd 100 (   test isom)                 IN:    645/6.4e+02 OUT:    557/5.6e+02.
Context date=99(16),  557 certs and  88 non-certs, using  25.229 seconds
Cmd 101 (          try)                IN:    557/5.6e+02 OUT:    788/7.9e+02.
 COL       ones             fills       COL       ones             fills
```

```
  28      572/5.7e+02      788/7.9e+02
 Cmd 102 (  test isom)                      IN:     788/7.9e+02 OUT:     530/5.3e+02.
 Context date=101(19),  530 certs and   258 non-certs, using  38.358 seconds
 Cmd 103 (          try)                    IN:     530/5.3e+02 OUT:     700/7.0e+02.
  COL       ones               fills         COL       ones               fills
  26      320/3.2e+02      700/7.0e+02
 Cmd 104 (  test isom)                      IN:     700/7.0e+02 OUT:     679/6.8e+02.
 Context date=103(22),  679 certs and    21 non-certs, using  35.945 seconds
 Cmd 105 (          try)                    IN:     679/6.8e+02 OUT:     806/8.1e+02.
  COL       ones               fills         COL       ones               fills
  24      127/1.3e+02      806/8.1e+02
 Cmd 106 (  test isom)                      IN:     806/8.1e+02 OUT:     745/7.4e+02.
 Context date=105(25),  745 certs and    61 non-certs, using  71.473 seconds
 Cmd 107 (          try)                    IN:     745/7.4e+02 OUT:     745/7.4e+02.
  COL       ones               fills         COL       ones               fills
  22        0/0.0e+00      745/7.4e+02
 Cmd 108 (  test isom)                      IN:     745/7.4e+02 OUT:     594/5.9e+02.
 Context date=107(28),  594 certs and   151 non-certs, using 162.643 seconds
 Cmd 109 (          try)                    IN:     594/5.9e+02 OUT:     594/5.9e+02.
  ROW       ones               fills         ROW       ones               fills
   8      594/5.9e+02      594/5.9e+02
 Cmd 110 (          try)                    IN:     594/5.9e+02 OUT:    1000/1.0e+03.
  COL       ones               fills         COL       ones               fills
  20      443/4.4e+02     1000/1.0e+03   21          9/9.0e+00    1000/1.0e+03
 Cmd 111 (          try)                    IN:    1000/1.0e+03 OUT:    1000/1.0e+03.
  ROW       ones               fills         ROW       ones               fills
   8     1000/1.0e+03     1000/1.0e+03
 Cmd 112 (          try)                    IN:    1000/1.0e+03 OUT:    1333/1.3e+03.
  COL       ones               fills         COL       ones               fills
  18      335/3.3e+02     1333/1.3e+03   19          0/0.0e+00    1333/1.3e+03
 Cmd 113 (          try)                    IN:    1333/1.3e+03 OUT:    1333/1.3e+03.
  ROW       ones               fills         ROW       ones               fills
   8     1333/1.3e+03     1333/1.3e+03
 Cmd 114 (          try)                    IN:    1333/1.3e+03 OUT:    1333/1.3e+03.
  COL       ones               fills         COL       ones               fills
  16        0/0.0e+00     1333/1.3e+03   17          0/0.0e+00    1333/1.3e+03
 Cmd 115 (          try)                    IN:    1333/1.3e+03 OUT:    1333/1.3e+03.
  ROW       ones               fills         ROW       ones               fills
   9     1333/1.3e+03     1333/1.3e+03
 Cmd 116 (          try)                    IN:    1333/1.3e+03 OUT:    1333/1.3e+03.
  COL       ones               fills         COL       ones               fills
  14        0/0.0e+00     1333/1.3e+03   15          0/0.0e+00    1333/1.3e+03
 Cmd 117 (          try)                    IN:    1333/1.3e+03 OUT:    1333/1.3e+03.
  ROW       ones               fills         ROW       ones               fills
   1     1333/1.3e+03     1333/1.3e+03   15          0/0.0e+00    1333/1.3e+03
   2     1333/1.3e+03     1333/1.3e+03   16          0/0.0e+00    1333/1.3e+03
   3     1333/1.3e+03     1333/1.3e+03   17          0/0.0e+00    1333/1.3e+03
   4     1333/1.3e+03     1333/1.3e+03   18          0/0.0e+00    1333/1.3e+03
   5     1333/1.3e+03     1333/1.3e+03   19          0/0.0e+00    1333/1.3e+03
   6     1333/1.3e+03     1333/1.3e+03   20          0/0.0e+00    1333/1.3e+03
   7     1333/1.3e+03     1333/1.3e+03   21          0/0.0e+00    1333/1.3e+03
   8     1333/1.3e+03     1333/1.3e+03   22          0/0.0e+00    1333/1.3e+03
   9        0/0.0e+00     1333/1.3e+03   23          0/0.0e+00    1333/1.3e+03
   10       0/0.0e+00     1333/1.3e+03  24        0/0.0e+00     1333/1.3e+03
   11       0/0.0e+00     1333/1.3e+03  25        0/0.0e+00     1333/1.3e+03
   12       0/0.0e+00     1333/1.3e+03  26        0/0.0e+00     1333/1.3e+03
   13       0/0.0e+00     1333/1.3e+03  27     1333/1.3e+03     1333/1.3e+03
   14       0/0.0e+00     1333/1.3e+03
Cmd 118 (   test isom)                  IN:    1333/1.3e+03 OUT:       68/6.8e+01.
Context date=117(30),   68 certs and 1265 non-certs, using 3528.063 seconds
Cmd 120 (        exit)                  IN:      68/6.8e+01 OUT:        0/0.0e+00.
 High water mark of stack used 14792 out of 300000 bytes
Execute time was 5492.516 Seconds
At end of run bvcount, pvcount, pmcount, bmcount, pgcount
  1925  1766    80    39    123
End of bdx.
```

Appendix B

Source Code for Dump/Restore

B.1 dump_setup

```
void dump_setup(char *source_fname, char *param)
{
  int c=0,i=0,j=0,endnum=0;
  char ch,dump_fname[100];
  time_t timer;
  if (!param)
   return;
  if (param[2]=='=') /* dumpfile name specified */
    strcpy(dump_fname,param+3);
  else if (source_fname) /* dumpfile name not specified */
  {
    /* create dumpfile name from source name */
    while (source_fname[i]) /* Find beg. and end of filename */
    {
      if (source_fname[i++]=='/')
        j=i;
    }

    while (source_fname[--i]!='.' && i);

    if (i)
    {
      source_fname[i]=0; /* Truncate source name at last '.' */
      strcpy(dump_fname,source_fname+j);
      source_fname[i]='.';
    }
    else /* No '.' in source filename */
      strcpy(dump_fname,source_fname+j);

    strcat(dump_fname,".dmp");
  }
  else /* no filename to create dumpfilename */
```

```
    {
      fprintf(stderr,"\n *** ERROR dumpfile and sourcefile not specified.\n\n",
              dump_fname);
      exit(1);
    } /* end cond dumfilename not spec. */

    printf("\nData will be dumped to: %s\n\n",dump_fname);

    if ((dumpfile=fopen(dump_fname,"r")))
    {
      fclose(dumpfile);
      fprintf(stderr,"\n *** ERROR dumpfile [%s] exists.\n\n",dump_fname);
      exit(1);
    }
    fclose(dumpfile);
    dumpfile=fopen(dump_fname,"w"); /* Open dumpfile */
    timer=time(NULL);
    fprintf(dumpfile,"%% source read from %s, dumped to %s, %s",
            source_fname,dump_fname,asctime(localtime(&timer)));
    set_lex_echo(dumpfile, NULL);
  } /* dump_setup */
```

B.2 dump()

```
void dump()
{
  static image_num=1;
  int r,c,v;

  if (!dumpfile)
  {
    warning("Dumpfile not open.  Dump skipped.\n");
    return;
  }

  if (image_num==1)
    fprintf(dumpfile, begin_names[1-dbg_flag[DUMP_UNCOMPRESSED]]);

  if (dbg_flag[DUMP_UNCOMPRESSED])
  {
    fprintf(dumpfile,"\n******** Image [%d] ********\n",image_num++);
    for (r=1;r<=NR;r++)
    {
      for (c=1;c<=NC;c++)
        switch(Arow[r][Acol[c+TNR]].val)
        {
          case 0:  putc('0',dumpfile);
                   break;

          case 1:  putc('1',dumpfile);
                   break;
          default: putc('.',dumpfile);
        }

      fprintf(dumpfile,"\n");
    }
  }
  else
  { /* Use compressed form */
    fwrite_int(image_num, dumpfile);
    image_num++;
    cf = Compress(NC, NR, cf);
    WriteCmpForm(cf, dumpfile);
  }
} /* dump */
```

B.3 restore_setup()

```
void restore_setup(char *param)
{
  if (!param)
   return;
  if (param[2]!='=')
  {
    fprintf(stderr,"\n*** ERROR restore filename not specified.\n\n");
    exit(1);
  }

  strcpy(restore_fname,param+3);
  printf("%s will be restored.\n\n",restore_fname);
  restore_source=TRUE;

  restorefile=open_include(restore_fname);
    /* Note that restorefile will stay open after the initial input is read
    /* from it, and the images will be read starting with the next line. */

  eof_string_found = look_for_BEGINIMAGE;

  reset_rcu();  /* for next parse element. Kills rcu_buf. */
} /* restore_setup */
```

B.4 restore_img()

```
void restore_img(int img_num)
{
  int i=0,num=0,r=1,c=1,ch,save;

  if (!restorefile)
  {
    fprintf(stderr,"\n*** ERROR restore file not open.\n\n");
    exit(1);
  }

  save=get_mark();
  if (read_uncompressed)
    num = uncom_num(img_num);
  else
  {
    num = fread_int(restorefile, FALSE);
    while ((num < img_num) and (num >= 0))
    {
      cf = ReadCmpForm(cf, restorefile);
      num = fread_int(restorefile, FALSE);
    }
  }
  if (num!=img_num)
  {
    if (num >= 0)
      printf("Warning: Image %d not found before image %d.\n\n",
      img_num,num);

    fclose(restorefile);
    restorefile=NULL;
    return;
  }

  printf("\n******** Restoring Image #%d:\n\n",img_num);
  fflush(stdout);
  still_good = TRUE;
  if (read_uncompressed)
  {
    while ((ch=getc(restorefile))!=LF); /* Skip over linefeeds */
    ch=getc(restorefile);
    cum_ran_cnt = 1.0;
    while ((ch!='*') and (ch != EOF))
    {
      if (still_good)
      {
        switch(ch)
        {
          case '.': c++;
                    break;

          case '0': still_good &= (RQ_chooseA(0,r,TNR + c++) != 0);
```

```
                    break;

          case '1': still_good &= (RQ_chooseA(1,r,TNR + c++) != 0);
                    break;

          case LF:  if (c>1)
                      r++;c=1;
                    else /* Blank line */
                      ch='*';
        }
      }

      ch=getc(restorefile);
    }
  }
  else
  {
    cf = ReadCmpForm(cf, restorefile);
    Decompress (cf);
  }

  if (still_good)
  {
    do_~md(save_hdr);
  }
  else
  {  /* Count entries here */
    save_hdr->raw_count++;
    save_hdr->estimate++;
  }
  backupto(save);
} /* restore_img */
```

Appendix C

Source Code For Greig (Hook-22) Test

C.1 hook22.c

```
#include "bdx.h"
#include <isominc.h>
#include "printsl.h"
#include "lexinput.h"
#include "opcodes.h"
#include "saverec.h"
#include "hook22.h"

#define GOOD 1
#define BAD  0
#define ABS(x) ((x>0) ? x : -x) /* not used */
#define THM13_LIM 10

int **S;

/* public function for performing test of the S matrix */
boolean do_hook_hook22(save_rec *cmd)
{
  int i,i1,i2,i3,i4,c1,c2,c3,c4;
  int c,d,e,u,v,w,uvw_sum;
  int col;
  static int num=1;
  int tval;


  createS(cmajorp); /* Generate S matrix */

  \* Testing theorems 9 and 13 *\
   for (i1=1; i1<=num_Scols; i1++)
   {
     c1=sortstack[i1];
     for (i2=i1+1; i2<=num_Scols; i2++)
     {
       c2=sortstack[i2];
       for (i3=i2+1; i3<=num_Scols; i3++)
       {
         c3=sortstack[i3];
         u=S[c2][c1];  v=S[c3][c1];  w=S[c3][c2];
         if (u+v<w || u+w<v || v+w<u ) \* Failed theorem 9 with cols c1,c2,c3 *\
         {
           (cmd->p1)++; /* Increment count of failures */
           return(BAD);
         } /* end if u,v,w */

#if 0 /* because theorem 13 is a noop! */
         uvw_sum=u+v+w;
         for (i4=i3+1; i4<=num_Scols; i4++)
         {
           c4=sortstack[i4];
           if (c4==4 && c1==6)
             S[4][6]=1;

           c=S[c4][c1];  d=S[c4][c2];  e=S[c4][c3];
           if (c+d+e+uvw_sum<THM13_LIM) \* Failed theorem 13 with cols c1,c2,c3,c4 *\
           {
             cmd->p1++; /* Increment count of failures */
             return(BAD);
           } /* end if c,d,e,u,v,w */
         } /* end for i4 */
#endif
       } /* end for i3 */
     } /* end for i2 */
   } /* end for i1 */

   return(GOOD); \* Theorems 9 and 13 are successful *\
}  /* do_hook_hook22 */
```

C.2 printsl.c

```
static char rcsid[] = "$Header: /mnt/larry1/home/staffcs/bdx/precomp/RCS/printsl.c,v 3
21:33:06 bdx Exp $";

#include <isominc.h>
#include "bdx.h"
#include "printsl.h"

static int size = 0;  /* Largest rcid in A */
static int *sumstack;  /* Columns in order of decreasing sums */
#define nprintcols 25
  /* Number of columns of S printed per page */

/* Variables exported in printsl.h */
int num_Scols; /* Total number of completed columns of S */
int **S; /* Compute a full or partial S matrix here. */
int *sortstack;  /* Columns in increasing order. */

/* static_init                                               static_init */
static  void static_init (voidplist)
  /* Initial setup. Called only once,  when size == 0 */
{
  int i;

  size = (NC > NR)? NC : NR;
  S = NEWV(int *, size + 1);

  for (i = 1; i <= size; i++)
    S[i] = NEWV(int, size + 1);

  sortstack = NEWV(int, size + 1);
  sumstack = NEWV(int, size + 1);
} /* static_init */

/* createS                                                       createS */
int createS(base_matrix *bmp)
{
  int r,c,i,ri,ci,s;
  A_value *Ar, *Ac;

  if (size == 0)
    static_init();

  num_Scols = 0;
  emptyS();
  for (ri = 1; ri < N_complete_rcs; ri++)
    if ((rc_stack[ri].rcid >= bmp->fstnr) and (rc_stack[ri].rcid <= bmp->lnr))
    {
      S[rcid_to_rc[rc_stack[ri].rcid]][0] = 0;  /* tag active rcs */
      num_Scols++;
    }
  for (ri = 1; ri < N_complete_rcs; ri++)
  {
    r = rc_stack[ri].rcid;
    if ((r >= bmp->fstnr) and (r <= bmp->lnr))
    {
      Ar = Arow[r];
      r = rcid_to_rc[r];
      for (ci = ri + 1; ci < N_complete_rcs; ci++)
      {
        c = rc_stack[ci].rcid;
        if ((c >= bmp->fstnr) and (c <= bmp->lnr))
        {
          Ac = Arow[c];
          c = rcid_to_rc[c];
          s = 0;
          for (i = bmp->fstnc; i <= bmp->lnc; i++)
            s += Ar[Acol[i]].val*Ac[Acol[i]].val;

          S[r][c] = S[c][r] = s;
          S[r][0] += s;
          S[c][0] += s;
        }
      }
    }
  } /* for ri */

  r = s = 0;
  for (c = 1; c <= size; c++)
    if (S[c][0] >= 0)
    {
      sortstack[++r] = c;
      s += S[c][0];
    }

  assert (r==num_Scols);
  return(s);
} /* createS */
```