1989

Computing techniques for the enumeration of cyclic Steiner systems

Timothy Frenz Carl

Rochester Institute of Technology
School of Computer Science

Computing Techniques for the Enumeration of Cyclic Steiner Systems

by

Timothy Carl Frenz

April 20, 1989

A thesis, submitted to
The Faculty of the School of Computer Science,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Dr. Donald L. Kreher (Advisor)

Dr. Stanislaw P. Radziszowski

Dr. Charles J. Colbourn

Dr. Peter G. Anderson

Rochester Institute of Technology
School of Computer Science
Masters Thesis

Computing Techniques for the Enumeration of Cyclic Steiner Systems

I Timothy C. Frenz hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date May 19, 1989

ABSTRACT

In this thesis a powerful algorithm is developed for finding cyclic Steiner systems.

A cyclic Steiner system with parameters $S(t,k,v)$ is a pair (V,B), where B is a collection of subsets all of size k (called blocks) and V is a v element set of points, such that each t-subset of V is contained in precisely one block of B. A Steiner system is called cyclic if it has an automorphism carrying the points in a v-cycle.

The results obtained so far with this algorithm are given in Table VII of chapter 5. Among the values reported there, are the number of distinct cyclic solutions to S(2,3,55), S(2,3,57), S(2,3,61) and S(2,3,63) which are 121,098,240, 84,672,512, 2,542,203,904 and 1,782,918,144 respectively. These values were apparently unknown previous to this work.

TABLE OF CONTENTS

CHAPTER I

Introduction and Background

1. Overview

Steiner systems are a special case of t-designs. A $t-(v, k, \lambda)$ design is a pair (V,B), where V is a v element set of points, and B is a collection of k-subsets of V called blocks, such that each t-subset of V appears in precisely λ blocks of B. Here λ is a positive integer, and $1 \leq t \leq k \leq v$, where if any equality exists, the design is said to be trivial. A $t-(v, k, \lambda)$ design is said to be cyclic if $V=\{0,1,2,...,v-1\}$ and whenever K is a block $K+1 = \{x+1 : x \in K\}$ is also a block, addition performed modulo v. When $\lambda=1$, the t-design is called a Steiner system and is denoted by $S(t,k,v)$. For example if $V = \{0,1,2,...,6\}$, and $B = \{ \{0,1,3\}, \{1,2,4\}, \{2,3,5\}, \{3,4,6\}, \{4,5,0\}, \{5,6,1\}, \{6,0,2\} \}$, then (V,B) is a cyclic $S(2,3,7)$.

Let $K = \{ x_1, x_2, x_3, ..., x_k \}$ be any k-element subset of V, where $x_1 < x_2 < x_3 < \cdots < x_k$. Let $d_i = x_{i+1} - x_i$ for $1 \leq i \leq k-1$ and $d_k = v - x_k + x_1$. Define $\delta(K)$ to be the set of all cyclic shifts of $(d_1, d_2, ..., d_k)$. That is $\delta(K) = \{ (d_{\pi(1)}, d_{\pi(2)}, ..., d_{\pi(k)})^i, 0 \leq i \leq k \}$. Then it is easy to see that $\delta(K_1) = \delta(K_2)$ if and only if $K_1 = K_2+i$, for some i, $0 \leq i \leq v-1$, addition performed modulo v. Denote by $<d_1, d_2, ..., d_k>$ the set of all k-element subsets K for which $\delta(K) = (d_1, d_2, ..., d_k)$, that is the cyclic orbit containing K.

One of the main problems with t-designs is the question of their existence. It is easy to see that if (V, B) is a cyclic design and $K \in B$, then $<\delta(K)> \subseteq B$. Thus in particular a cyclic Steiner system can be thought of as a disjoint union of cyclic orbits.

One example of t-designs are the much studied projective planes. It can be shown that a projective plane of order n is an $S(2, n+1, n^2+n+1)$. For example, an $S(2,3,7)$ is a projective plane of order 2. It is easy to construct a projective plane of order p^α by using the finite field of order p^α. Projective planes of order 6 were first shown not to exist by Tarry in 1900 [Tarr]. A more elegant proof was given by Doug Stinson in 1984 [Stin]. The Bruck-Ryser theorem shows that a projective plane of order n does not exist when $n \equiv 1$ or 2 (mod 4) and the square free part of n contains a prime factor $p \equiv 3$ (mod 4) [Bruc]. Hence, in particular there is no plane of order 14. The nonexistence of projective planes of order 10 was only just recently announced by Brenden McKay at the 20^{th} Southeastern Conference on Combinatorics, Computing, and Graph Theory, February 1989, Boca Raton, Florida. It's nonexistence was established by a computer search on a Cray computer that took several years.

Table I

Existence of projective planes of small orders															
order	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
exists	Y	Y	Y	Y	N	Y	Y	Y	N	Y	?	Y	N	?	Y

When t is large, $t > 4$, there is not that much known about the existence or non-existence of $t-(v, k, \lambda)$ designs. For t values greater than 6, the only known t-designs are the ones found by Teirlinck in [Tier87]. In this paper he constructs $t-(v, t+1, (t+1)!^{(2t+1)})$ designs for each $v \equiv t$ ($modulo (t+1)!^{(2t+1)}$) *and* $v \geq t+1$. The only small 6-designs that are known are the 6-(14,7,4) designs found by Kreher and Radziszowski [Kreh86b], the 6-(33,8,36) and 6-(20,9,112) designs

found by Kramer, Leavitt and Magliveras [Magl84,Kram85] and the infinite family $6-(8u+6,7,4u)$, for $u \geq 2$ found by Teirlinck [Tier88].

It is easy to see that the number of blocks b in a Steiner system is

$$b = \frac{\binom{v}{t}}{\binom{k}{t}} \tag{1.1}$$

If $b = m * v$ for some $m \geq 1$, it is possible that m orbits of k-sets under a cyclic group could be used to construct the design, and thus the design would be cyclic. However, it is also possible for a design to be cyclic when b is not a multiple of v; this can occur when the residue r of b modulo v is a sum of factors of v, $f_1 + f_2 + \cdots + f_n$, where each quotient of $\frac{v}{f_i}$ is also a factor of k. In this case the number of orbits needed to generate the Steiner system is at least $m+n$. The orbits of size f_i properly dividing v are called short orbits. Note that if the parameters of a design meet either of these criteria that this does not necessarily prove that such a cyclic design exists, but in order for a cyclic design to exist, it must meet one of these criteria.

Table II

Criteria for Cyclic Steiner Design	
i	b $\equiv$ 0 modulo v
ii	b $\equiv$ r modulo v, where r = $f_1 + f_2 + \cdots + f_n$ and v $\equiv$ 0 modulo f_i for all $i = 1,2, \cdots n$ and k $\equiv$ 0 modulo $\frac{v}{f_i}$ for all $i = 1,2, \cdots n$

2. History

Recent research has found that the first work done in the area of Steiner systems was done by Plucker in 1829. The problem apparently was not well defined at that time. It was in 1844, when Woolhouse asked for the first time for which integers t, k, v does an $S(t, k, v)$ exist? Several partial answers were given only three years later in 1847 by Kirkman who showed that a design $S(2, 3, v)$ exists if and only if $v \equiv 1$ or 3 (mod 6) and he constructed designs $S(3, 4, 2^n)$ for every n [Lind78]. Also Hanani has shown that $S(3, 4, v)$ exist when $v \equiv 2$ or 4 (mod 6) [Hana60].

Actually Steiner's name was not invloved with the problem until 1853, and because the other papers were not well read, his name became associated with the problem. For the reader interested in the early beginnings of the problem and some early solutions see [Lind78].

CHAPTER II

Project Description

1. Constructing Cyclic Steiner Systems

1.1. Matrix Generation

First the A_{tk} matrix, which is defined later, is constructed. The problem is then to find a given set of columns such that the sum of all cells in a given row from all of the given columns equals λ. For t-designs in general this is a difficult problem. However, for Steiner systems the problem can be reduced to doing bit operations on the columns.

There are three steps involved in generating the matrix. The first step consists of generating all of the t-orbits over v points. At first glance it may appear that this is simply enumerating all sequences $d_1, d_2, ..., d_t$ with $d_1, + d_2, + ..., + d_t = v$, but this is not the case. For example, let t be 3 and v be 10, then one of the orbits is <1,4,5>. Note that this is the same orbit as both <4,5,1> and <5,1,4>, because all three of these can be rotated into the same orbit. However, these are not the same as <1,5,4>, <5,4,1>, or <4,1,5>. If t and v are relatively prime, then the number of blocks represented by an orbit will be v. Thus it follows that the number of t-orbits N_t can be calculated with the following formula:

$$N_t = \frac{\binom{v}{t}}{v} \tag{2.1}$$

The second step is to generate all k-orbits over v points. This step is the same as the first step, except different parameters are used. Letting k be 4 and v be 10 again, one orbit is <1,4,1,4>. As above, the number of k-orbits N_k when k and v

are relatively prime is given by

$$N_k = \frac{\binom{v}{k}}{v} \tag{2.2}$$

The formula for counting orbits of an arbitrary automorphism group is the Cauchy-Frobenius-Burnside Lemma and appears in almost any algebra or elementary group text. A particularly nice proof can be found in D.R. Hughes and F.C. Piper's book *Projective Planes*, Springer-Verlag (1973) on page 11. If G is the supposed automorphism group and Fix(g) equals the number of k-subsets of V fixed by $g \in G$, then the general formula is

$$N_k = \frac{1}{|G|} \sum_{g \in G} Fix(G) \tag{2.3}$$

The third step consists of using the t-orbits and k-orbits generated from the first two steps and actually generating the A_{tk} matrix. For each i, $1 \leq i \leq N_t$ and j, $1 \leq j \leq N_k$ one must calculate $A_{tk}[i,j]$ the number of times a representative of the i^{th} t-orbit is contained in members of the j^{th} k-orbit. For the cyclic group this can be calculated by just considering orbit labels and not looking at the blocks in a given orbit. First you calculate how many times you can sum consecutive distances in the j^{th} k-orbit label to make it look like the i^{th} t-orbit label. Then count the number of times the j^{th} k-orbit label can be cyclically rotated to the same label. The quotient of these two operations is $A_{tk}[i,j]$. If $A_{tk}[i,j] > \lambda$, which for Steiner systems is 1, then, the j^{th} k-orbit can not be used in the solution because the sum of all the cells in a given row must equal λ. Consequently in this case the j^{th} column of the A_{tk} matrix need not be stored.

Using the above examples for the t-orbit and k-orbit, <1,4,5> has <1,4,1,4> two times. For example if <1,4,5> is the i^{th} t-orbit label and <1,4,1,4> is the j^{th} k-orbit label, then, $A_{tk}[i,j] = 2/2 = 1$.

Most of the operations we wish to perform on the A_{tk} matrix are column operations. Thus we think of a column as a single entity and refer to them as incidence vectors. When all of the t-orbits incident to a given k-orbit is λ or less, then this incidence vector $\bar{I}$ of length N_t, is saved for inclusion in the search space. The matrix is complete when all k-orbits have been examined and the relevant incident vectors and their corresponding k-orbits are saved for later usage. Thus the matrix is just the collection of these incident vectors of length N_t.

1.2. Generation of Distinct Cyclic Solutions

Once the matrix is generated, the problem is to find a solution for the given design. For a solution to exist among the N_k incident vectors $\bar{I}_1, \bar{I}_2, ..., \bar{I}_{N_k}$, a (0,1)-vector $\bar{x} = (x_1, x_2, ..., x_{N_k})$ must exist such that

$$\sum_{i=1}^{N_k} (\bar{I}_i),j * x_i = \lambda, \text{ for all } j = 1, 2, \cdots N_t \qquad (2.4)$$

Note that this problem is similar to the subset sum problem with the added restriction that each row in the matrix from the designated columns must sum to exactly λ.

When the (0,1)-vector $\bar{x}$ is found, a distinct cyclic solution has been found and the k-orbits from the columns where $x_i = 1$ are saved. These orbits can then be used to generate the design.

1.3. Generation of non-isomorphic Cyclic Solutions

With the given distinct cyclic (dc) solutions to a design, it is then necessary to determine which of these solutions are isomorphic to each other. One can determine the non-ismorphic cyclic (nc) solutions in a variety of ways.

One way of determining isomorphism between two designs, provided the parameters of a design meet the criteria of the theorem below, is to show that the designs are multiplier isomorphic. This method is straight forward and was used in the programs.

To determine which designs are multiplier isomorphic to others, you must go through the list of dc-solutions one at a time and multiply the k-orbits by the numbers relatively prime to v. If for any of these relatively prime numbers a dc-solution is mapped onto a nc-solution, the dc-solution is multiplier isomorphic to the nc-solution and the dc-solution can be rejected. However, if for some dc-solution none of the numbers relatively prime to v can map this design into some nc-solution, the dc-solution is not multiplier isomorphic to any solution and is added to the list of nc-solutions.

Theorem [Phel87b]

(i) Lemma 4.2: For primes p, q, $q \mid p-1$, any cyclic 2-design isomorphic to $(Z_p \times Z_q, B)$ must be multiplier equivalent to it.

(ii) Corollary 4.3: Isomorphic cyclic $2-(pq,q,1)$ designs are always multiplier equivalent for primes p, q with $p > q$.

(iii) Lemma 4.4: If a cyclic $2-(m,k,1)$ design does not exist for either $m = p$ or $m = q$, p, q primes, then any two isomorphic cyclic $2-(pq,k,1)$ designs will

be multiplier equivalent.

2. Functional Specifications

2.1. Design Criteria

The primary aim is to develop programs to generate the non-isomorphic cyclic solutions for cyclic Steiner Systems. The programs are implemented as robust as possible so that non-cyclic Steiner Systems can also be searched for, but non-isomorphic solutions will not be computed for such designs as this would require another algorithm. Also, it is possible with modifications to search for other cyclic t-designs where $\lambda \neq 1$. However, since the primary aim is Steiner systems, the current programs use some programming tricks with bit manipulation to speed up computing times, and are consequently restricted to $\lambda = 1$. Of course, the programs are well documented and thus can easily be modified to search for other designs.

Secondarily, I believed that the system should be as efficient as possible. This is a critical goal of implementation as the search space for even moderate sized v is very large. Even for a cyclic $S(2,3,43)$ the number of k-orbits is 287, and only 7 of these are used to make a solution, thus there are 287 choose 7 ways of picking these 7 incidence vectors, a rather large number of possibilities. I chose to use C as the language for programming for its efficiency and also for its portability, as I in fact have done the programming on more than one machine.

The third criteria is that the user should be able to use the system with ease. I have placed this criteria after the others as the primary user is myself, but I am also making its inputs simple enough for any user with some knowledge of what a

Steiner system is.

Table III

Design Criteria	
Rank	Criteria
1	Robust
2	Well Documented
3	Efficient
4	User Friendly

2.2. Functions Performed

There are six functions performed by this system, three of which correspond to the three phases of generating the cyclic Steiner system: (1) matrix generation, (2) distinct cyclic solution generation, and (3) non-isomorphic solution generation. The other three functions are used before and after the cyclic Steiner system is run: (4) generating a list of parameters which are possible candidates for Steiner systems, (5) determining if a given design could be cyclic, and (6) displaying the solutions after steps 2 and 3.

The first function produces the matrix to be used as input to the second. The second function inputs this matrix and outputs k-orbits which can be displayed with the sixth function. The third function accepts the k-orbits produced by the second function and produces k-orbits which again can be displayed by the sixth function. The fourth function is used before any of the others to test to see if some given parameters meet the criteria for a Steiner system to exist. The fifth function is used before the first three to test to see if a given set of parameters could produce a cyclic design. Finally, the sixth function displays results. Actually, results are

displayed by the same program for all four types of results: dc-solutions, nc-solutions, non-cyclic solutions and non-cyclic non-isomorphic solutions.

2.3. Limitations and Restrictions

One major limitation is disk space. To generate all of the dc-solutions to $S(2,3,49)$ would require 158MB of disk space. A second major limitation is that of time. On the larger cyclic Steiner systems the time to find distinct solutions is quite long, hours and possibly days for some, this is one reason why efficiency is very important to this system.

Also, as currently implemented the largest t, k, v or λ value permitted is 255. This is becasue I have implemented almost everything as char types; actually all of these are of type PT, which is currently an unsigned char. This could be changed to unsigned short, if the values need to exceed 255. I implemented it using char because less space would be required, and since every parameter for possible designs that I considered has argument values less than 100, the limitation of 255 did not seem restrictive.

In the current implementation of determining the isomorphism of the orbits of a design, an orbit is stored as an unsigned long integer. Since I store a k-orbit as an integer this limits the size of v. The following table shows this limitation.

Table IV

Limitations on v for various k												
k	3	4	5	6	7	8	9	10	11	12	13	14
v	65534	1629	260	90	47	31	25	21	20	19	19	19

As k gets large, v approaches $k + 1$. Note that for the k values of 3, 4 and 5 the v value limitation does not matter since I have implemented v as an unsigned char, thus limiting its size to 255. I doubt for the other values of k if I would search for a design with v greater than this limit. For example, with k=6 and v=90 there would be 6,917,940 k-orbits. With k=7 and v=47 there would be 1,338,120 k-orbits. Thus, a large number of orbits would need to be searched through to find a design. If for some reason the representation of orbits seems too restricting, one may want the orbits represented by character strings.

2.4. User Inputs

cyclic is started by the program name followed by the parameters t, k, and v, and optionally λ. Or one may optionally use one parameter, a file name, which contains lines of t, k and v and optionally λ.

gmatc and **gmat** both require t, k, v and a matrix file name.

find, **iso**, **res**, and **resn** only require the matrix file name used by **gmatc** or **gmat**.

The three main functions of the system - **gmatc**, **find**, and iso - also allow an optional parameter which shows its status as it progresses, -s. **find** also has another optional parameter which allows the program to run without saving dc-solutions to the disk, -n. This may be useful in calculating large designs without having to use up storage space. This is what I used in finding the number of dc-solutions for large Steiner triple systems.

All of the programs display what parameters are expected if just the filename is entered without any parameters. This conforms to the normal UNIX standard of

a usage prompt. Syntax summaries for all of the functions for this system are found in Appendix C.

2.5. User Outputs

The three main programs' screen output is the number of solutions found and the time it took to find them. The other outputs are saved in files. **gmatc** saves the t-orbits in $t.v$ and the k-orbits in $k.v$. Both of these files can be displayed with **show** filename t or k. **gmatc** also outputs another k-orbit file named n$k.v$, which contains only the k-orbits to be considered in the search space. Thus these k-orbits intersected all t-orbits λ or fewer times, and their corresponding incidence vectors are the columns of the A_{tk} matrix. The A_{tk} matrix produced by **gmatc** is saved in the matrix file the user picked. This matrix file also contains header information neccessary for **find** to operate, the values of t, k, v, λ a flag as to whether the design under consideration is cyclic or not, and the new k-orbit file name, n$k.v$. After **gmatc** terminates the $t.v$ and $k.v$ files are no longer needed and they may be removed. The n$k.v$ file can not be deleted yet because it is used by **find**.

find reads in the matrix file produced by **gmatc** and searches for solutions to the design according to the parameters in the header. When solutions are found it reads in the k-orbits from the n$k.v$ file and saves them in a file with the prefix of s++ and with a postfix of the matrix file name. The +'s designate alphabetic characters a, b, ... z; thus, **find** can store its output in up to 676 different files. This output file also has header information in it, along with t, k, v, λ, and the cyclic flag, the number of orbits required is saved in the header. **res** will display all of the solutions found by **find**. When find is completed, the n$k.v$ file can be removed.

iso reads in the s++mfn files produced by find and outputs its solutions into n++mfn files, where the +'s represent the same thing as in the find function. These files can then be displayed by **resn**.

cyclic prints to stdout the number of orbits needed to generate the cyclic design and whether the short orbit is required, or it tells the user that it is not cyclic.

show prints to stdout the n-orbits or n-subsets generated by **gmatc** or **gmat** respectively. Actually, this can be used to dump the contents of most files as it just prints n values to a line for the entire contents of the file.

res and **resn** display on stdout the number of solutions found by **find** and **iso** respectively. The postfix of n represents non-isomorphic solutions.

pos displays to stdout what values of t, k, and v allow for a possible Steiner system to exist.

CHAPTER III

Implementation Details

1. Architectural Design

The software design phase of a computer project is one of the most important aspects of programming. With this idea in mind I have coded this project in a modular fashion. I have chosen to use C as my programming language because it is widely used, supports modular program design, and because of its efficiency. By disciplined programming the source files can act as modules, thus a single program is allowed to reside in several source files. The UNIX *make* utility greatly enhances a programmers ability to program in a modular fashion; hence, I have included the *makefile* for my system in Appendix A. In this chapter I show how the various modules fit together to make a complete system which exhaustively constructs non-isomorphic cyclic Steiner systems.

There are six primary executable commands in this system:

(1) **pos**, give possible values for t, k and v;
(2) **cyclic**, determine if a design could be cyclic;
(3) **gmatc**, generate a binary matrix;
(4) **find**, find distinct cyclic solutions;
(5) **iso**, generate non-isomorphic solutions;
(6) **res**, display solutions.

Each of these commands has a source file of the same name with a suffix of ".c". These files consist of routines which are used solely by this command in addition to accessing some shared routines. These shared routines are stored in ".h" files and are included into any programs that need these functions. These shared routines are in three modules:

(1) funct.h, common functions to some commands
(2) globals.h, global constants and macros
(3) mytime.h, time related functions

There are also some other routines which are used in conjunction with some main command, but because the code is self-contained it deserved to be in a separate module from the main source code file. There are five of these modules containing there own related routines:

(1) blocks.c, generates blocks
(2) orbits.c, generates orbits
(3) heap.c, performs a heapsort on the A_{tk} matrix
(4) mat.c, generates incidence vectors from blocks
(5) matc.c, generates incidence vectors from orbits

Isolating these modules from their main module makes enhancements, replacement, or transfer to another application much easier. The programming that was required to code these modules will be discussed at length in chapter 4. We will look at how data structures are named and how they are implemented.

2. Naming Conventions

To facilitate making enhancements in the future as easy as possible, I have used a few guidelines in choosing my identifiers. First, if a variable is to be directly associated with a certain parameter in a design, I named the variable according to what is the most commonly used label for that parameter in a design. Furthermore, if Steiner systems use a different label, then this is used as the variable name, since this project deals with Steiner systems this seems like a logical conclusion. For example, anywhere t, k, v, or lambda is used you can be sure that it represents that particular parameter in a design. Secondly, all functions that work with arrays as opposed to single items have a prefix of "a_" on them. This makes the code

more understandable if you know a certain routine is performing an operation on a whole array rather than just single entities. Finally, all other identifiers are named to represent their purpose in the program.

3. Data Structures and Global Variables

All throughout the programs the parameters t, k, v and λ are of the data type PT. This is defined as an unsigned char in "globals.h", in this way the range of these parameters can be changed by making one change in the "globals.h" file.

gmatc really does not require much in terms of data structures. The current orbit is needed when generating the t-orbits and k-orbits, and you must keep track of the number of permutations to make sure duplicates are not being constructed. When generating the incidence vectors all of the t-orbits are stored internally and each k-orbit is processed one at a time. If the incidence vector contains λ or less in all cells, then the incidence vector and the k-orbit are saved in external files.

find uses a variety of arrays to find solutions to a Steiner system in the most efficient way possible. The A_{tk} matrix is stored in a one dimensional array, so my own addressing is performed. I also have an array used to point into this matrix, *ptr*, which when used as an index for the columns - incidence vectors - makes the columns appear in numerically descending order. My heapsort procedure creates this array. Using this array two other arrays are created removing all duplicate columns from the ptr array. *unique* contains the subscripts of each unique column in descending order in the A_{tk} matrix. *dup* contains subscripts into the *ptr* array as to where the starting and ending locations of the duplicate columns are.

Duplicate columns arise often for Steiner triple systems since every $<d_1,d_2,d_3>$ orbit creates the same incidence vector as $< d_1,d_3,d_2 >$. In fact, every column of the A_{tk} matrix has a duplicate column in a STS, except for the short orbit. Thus, it is easy to see that the number of dc-solutions to a STS(v) must be a multiple of $2^{\frac{v}{6}}$, where $\frac{v}{6}$ is the number of full orbits in the design. Actually, for all $S(2,k,v)$ designs, there will be duplicate columns, and by creating these two arrays, *unique* and *dup*, when a solution is found that consists of one incidence vector, the other solution is immediately saved instead of searching for it.

Two other arrays are then created in find which allow faster manipulation of the A_{tk} matrix. *list* will contain the indices into the *ptr* array of where the highest bit in the incidence vectors changes. In this way once one incidence vector is used from one group, it can jump to the beginning of the next group by accessing the *list* array. I also create an array *next* which points to the next possible incidence vector that could be used with the current incidence vector.

Once all of these arrays are created, I can begin searching for a solution. I also create three more arrays for this purpose: one to hold the indices of the *unique* array of the columns used in the solution, aptly named *ans*; one to indicate what rows still do not have a bit in them, named *to_go*, and one to assist in saving the solutions, named *tmp*.

iso uses a variety of internal data structures also. I store the integer representations of all the orbits from all of the nc-solutions in a binary tree. Each node in the tree contains a pointer to a list of pointers to the nc-solutions that have that orbit as one of their orbits. I have implemented the list of pointers as a

dynamically linked array of pointers, because this saves on the amount of memory required as long as the first array is at least one more than half full. Each nc-solution contains the integer representation of all of its orbits in an array. This array is always sorted when stored, using the heapsort of course, to facilitate comparing a current design to this one.

I also create an array of numbers that are relatively prime to v. These are then used when checking for multiplier isomorphism between designs. When a design is isomorphic to one of the nc-solutions, the number that was used as the multiplier is bubbled to the beginning of the array. Thus, this will be the first number used the next time a design is being checked to see if it is isomorphic to a nc-solution. By observing several designs, it could be seen that if a multiplier m was used to show a design was isomorphic to another design, that the same multiplier m was used again in one of the next designs.

4. Algorithms

The following is the pseudocode for find. Note that bit operations are performed because I assume Steiner systems are being searched for. Thus, this is one segment of the code that must be rewritten if other t-designs are to be found. However, this same code works if you are searching for non-cyclic Steiner systems where the k-sets would be designated by the columns and t-sets by the rows. A solution from this would then be all the blocks that constitute a Steiner system. gmat does produce the required data for find to operate and find solutions to a given Steiner system. Now for the pseudocode.

```
while ( a solution is still possible ) do
  while (( a solution not found ) AND ( a solution is still possible )) do

    (1) find the first column in the current group that does not have a
        BITWISE AND with any of the previously selected columns for the
        design.

    (2) if ( a column was found in this group )
          (a) add this column to the list of columns used so far
          (b) if ( a solution was found )
                set the flag denoting this
              else
                determine the position of the next possible column and
                what group it is in

    (3) else if ( a column has already been picked )
          (a) remove the last column picked from the list of partial
              solutions
          (b) set the column to start searching at to be the next column
              in the same group as this

    (4) else
          set the flag that another solution is not possible

  end while

  if ( a solution was found )
    (1) save the solution found, taking into consideration duplicate
        columns
    (2) remove last column selected and continue searching at that
        point
    (3) reset the flag that a solution was not found

end while
```

The data structures defined in the last section must be created before this algorithm begins. This backtracking algorithm uses these data structures to find all solutions to a given design. One could easily modify the code to find the first N solutions and then stop. Thus, existence of a cyclic $S(t,k,v)$ could be determined.

The code for **iso** is easier to understand. Currently the only form of isomorphism being checked for is multiplier isomorphism. To make this program really

useful other types of computing isomorphism must be added. Again, the data structures from the previous section must be defined and the numbers relatively prime to v computed.

```
while ( more distinct cyclic solutions to check )

  while (( isomorphism not determined yet ) AND
        ( there are more numbers relatively prime to v )) do

    (1) for all orbits of a design
       (a) generate the starter block from the orbit multiplied by some number
       (b) calculate the orbit from this starter block
       (c) calculate the minimum orbit
       (d) generate the integer for this orbit

    (2) determine if this design exists

    (3) if ( the design exists )
          (a) set the flag denoting that isomorphism was determined
          (b) put the mulitplier used at the beginning of the array of
              multipliers

  end while

  if ( the design was not isomorphic )
     (1) add the design to the tree of non-isomorphic cyclic solutions
     (2) save the answer

end while
```

CHAPTER IV

Module Designs and Specifications

In this chapter the primary algorithms used in the implementation of this project are examined along with their corresponding data structures. I will be examining them in the sequence that they are used in finding non-isomorphic cyclic solutions for Steiner systems.

1. Matrix Generation

The matrix generation part of the system consists of two distinct phases: orbit generation and matrix generation. These two phases correspond to two modules of code, orbits.c and matc.c respectively. These are both called from the controlling main program gmatc. Both of these phases are described below in more detail, but here is a summary of each. During the orbit generation phase, all orbits of t distances in a field of size v are calculated and stored in a file called $t.v$. Let the total number of orbits found be N_t, this will be the number of rows in the A_{tk} matrix. This same procedure is then followed for k and v resulting in a $k.v$ file. Here the number of orbits found will be the number of columns in the matrix, and hence denoted by N_k. The A_{tk} matrix is then generated with these two files as input, and the matrix as output. While it is doing this it is eliminating any k-orbits that have a element value greater than λ in any row, thus the number of k-orbits, N_k, could be decreasing. Since we are dealing with Steiner systems the size of the resulting A_{tk} matrix will be $N_t * N_k$ bits.

The orbit generation phase must be able to produce all possible orbits over any distance for a given number of distances. This task is very easy when the number

of distances is 2. There will only be $(v / 2)$ such orbits.

To generate the orbits for a given pair of numbers (n, v), where n is either t or k, I create an array of length n where the sum of these n distances will be v. Then using an iterative process, I generate all possible simple orbits over v points. Here a simple orbit is one in which the distances are in strictly increasing order. With each simple orbit, I recursively generate permutations so that all possible orbits of n distances over v points are generated.

During generation of the incidence vectors, all of the t-orbits generated during the orbit generation phase are read back into internal memory into a one-dimensional array. Then each of the k-orbits is read in and used in producing the incidence vector for that k-orbit. In doing this I use the method described in chapter 2.

To make my programs more adaptable to finding other designs I have also coded a similar command for generating a matrix where the rows and columns are based on blocks rather than orbits. Thus these could be used when searching for non-cyclic designs. The corresponding files for non-cyclic designs are: generation of blocks in blocks.c, matrix generation in mat.c and the main program **gmat** in gmat.c. The same **find** algorithm can be used to enumerate the designs produced by this method, however, **iso** will need drastic revision as this part currently assumes cyclic designs are being compared for isomorphism. Note that the size of the A_{tk} matrix produced using this algorithm will be larger than the A_{tk} matrix produced using **gmatc** by v^2. Thus, this is really not an option for many designs under consideration.

2. Solution Generation

Distinct cyclic (or distinct non-cyclic) solutions are generated by **find**. The only files used in this phase of the project are find.c and heap.c. I perform a heap-sort on the incidence vectors to speed up the search process in find.

Using the data structures defined in the last chapter, I search for the solution using an iterative algorithm rather than a recursive one, this should also help in the overall performance of the algorithm. My algorithm can best be described as an exhaustive backtracking algorithm. I start with the first incidence vector in ptr then jump to the next group defined by list, or the incidence vector defined by next. Then this is applied until no incidence vector in a group defined by list can be found with the current incidence vectors already used, at this point I remove the last incidence vector added to the solution and keep searching in the same group for another incidence vector. If no other incidence vector is found, I remove the last incidence vector added again. This search process goes back and forth and exhaustively finds all distinct cyclic solutions, or distinct non-cyclic solutions.

Although it may seem like wasting a lot of space, and hence memory, to have all of the arrays that were discussed in the previous chapter, the improvement in time is so significant that it is worth it. When not saving the solutions for the large Steiner triple systems that I calculated, over 5000 solutions were found per second on the average.

3. Non-isomorphic Solution Generation

iso is a stand alone source file, other than included header files. Several steps are required to create an orbit that is multiplier isomorphic to the current orbit.

First the starter block S is generated from the current orbit, which is a trivial task. Given the starter block S, a new starter block S_{new} is created by multiplying each element in S by the multiplier m (modulo v). I have an array of multipliers stored, where the multipliers are relatively prime to v. Also the multipliers are stored in the array in the order that they were last used. S_{new} must be sorted such that the new orbit O_{new} can be created, which is the next step, again a trivial task. Given O_{new}, I must ensure that this is the minimum orbit representation for the given distances that can be represented by this orbit. For example, <1,2,4>, <2,4,1> and <4,1,2> all generate the same set of blocks, however my algorithm must use the <1,2,4> orbit when checking for isomorphism.

For each design in question, first all orbits are converted to their integer representations for a given multiplier m. Then the first number in the orbit representation array is used in accessing the binary tree to return a list of all nc-solutions that have that number as one of their orbit representations. The orbit representation array for the design under question is then sorted, and using the list returned by the binary tree traversal, the orbit representation arrays are compared for equality. If there is equality, the designs are isomorphic. If there is no equality, the next multiplier is checked until there are no more multipliers which means the design is nonisomorphic to any of the nc-solutions.

When all dc-solutions have been checked, all of the nc-solutions have been calculated and the task is completed. Again, this algorithm may use a lot of space, but the time improvement was very substantial. I increased the speed from over five days of CPU time on a VAX 8200 to just under four hours of CPU time when finding the number of nc-solutions to STS(43) and STS(45).

CHAPTER V

Results Obtained

The results obtained by the algorithms presented in this thesis are summarized in table VII. In this table the entry under orbits, dc and nc are the number of cyclic orbits, the number of distinct cyclic solutions and the number of non-isomorphic solutions found respectively. The entries for S(2,3,55), S(2,3,57), S(2,3,61) and S(2,3,63) are apparently new. A disscussion of this table and its connection with previous reported results now follows.

1. Colbourn

Peltesohn has shown that there exists cyclic S(2,3,v) for all $v \equiv 1,3$ modulo 6, where $v > 6$ and $v \neq 9$. Colbourn in [Colb82] has enumerated all the dc-solutions for Steiner triple systems for $v \leq 51$ and I have duplicated his results with the exception of $v = 37$. I calculated 28480 distinct cyclic solutions, and Colbourn had 28420 distinct cyclic solutions. By private communication with Colbourn I discovered that the 28480 is indeed the correct answer. The following table shows the results obtained by Colbourn as they were in [Colb82c]. The number of distinct cyclic solutions for Steiner triple systems is represented by dc STS(v), and the number of non-isomorphic cyclic solutions for Steiner triple systems is represented by nc STS(v).

Table V

Colbourn's results for S(2,3,v)		
v	dc STS(v)	nc STS(v)
7	2	1
9	0	0
13	4	1
15	4	2
19	32	4
21	32	7
25	240	12
27	144	8
31	2048	80
33	1600	84
37	28420	820
39	18048	798
43	395648	9508
47	278784	11616
49	6594560	?
51	4474112	?

I have also included other results of the work of Colbourn and Colbourn [Colb80e]. This work was with S(2,4,v) and S(2,5,v) which is again summarized in the table below.

Table VI

S(2,4,v)		S(2,5,v)	
v	nc	v	nc
13	1	21	1
16	-	25	-
25	-	41	1
28	-	45	-
37	2	61	10
40	10	65	2
49	224		
52	206		
61	18132		
64	12048		

In this same paper, [Colb80e] several ma-cyclic designs were also found, ma-cyclic designs are those that have a multiplier automorphism. S(2,3,v) designs were examined up to $v = 85$, except where v was a prime, S(2,4,v) designs to 88, and S(2,5,v) designs to 85. They also generated a S(2,6,91).

In [Colb80b] two new cyclic Steiner quadruple system's were found, SQS(38) and SQS(40). Also, the enumeration of all 29 cyclic SQS(20) was performed by Phelps in [Phel80b].

2. Grannell and Griggs

In a paper by Grannell and Griggs [Gran82] the existence of a SQS(32) was found. They also make a conjecture that cyclic Steiner quadruple systems exist for all v except 8, 14, and 16.

3. Cho

In [Cho80] it was proven that if a cyclic SQS(v) exists, for $v \equiv 2,10$ modulo 12, then a cyclic SQS($2v$) also exists. Thus several existence questions for $v < 100$ were obtained, as well as several large designs, the interested reader may want to read this paper. To summarize some results, new cyclic Steiner quadruple systems were found for several orders of v for which the existence of a cyclic SQS was previously unknown. Here is a list of these new orders: 52, 68, 100, 116, 148, 164, 196, 212, 244, 356, 404, 452, 548, and 592. The smallest orders of v for which existence of a cyclic Steiner quadruple system is still unkown at the time of this paper are: 32, 44, 46, 56, 62, 64, 70, 80, 86, 88, and 94. Note that the SQS(32) was found by Grannell and Griggs since then as mentioned in the previous section. Note also that SQS(44)

4. Gibbons

In Gibbons' work [Gibb76] several results were listed for Steiner systems and t-designs in general. Please refer to his work for his results.

5. Frenz

The following table shows the results that were obtained from the algorithms that were described in this thesis. Non-isomorphism was established by only checking multipliers and making liberal use of the theorem of Kevin Phelps as given in section 1.3 of chapter 2. The only exceptions are for S(3,4,20) and S(3,4,22). The 29 non-isomorphic solutions for S(3,4,20) was established by Phelphs in 1980, see [Phel80b]. The algorithms described in chapter 3 only establish that there are 148 non-multiplier isomorphic S(3,4,20) designs. The entry of 1140 for S(3,4,22) reflects the number of dc-solutions my algorithm reports. The number of non-multiplier isomorphic solutions it finds is 1017. According to a paper by Diener [Dien80], there are however 210 dc-solutions and 21 nc-solutions. I know that the figure of 1017 is incorrect, but I believe my figure of 1140 is correct. Unfortunately, my algorithm does not determine any form of isomorphism other than multiplier isomorphism.

Table VII

v	S(t,k,v)	Orbits	dc	nc
7	S(2,3,7)	1	2	1
10	S(3,4,10)	3	1	1
11	S(4,5,11)	6	2	1
13	S(2,3,13)	2	4	1
	S(2,4,13)	1	4	1
15	S(2,3,15)	3	4	2
17	S(3,5,17)	4	2	1
19	S(2,3,19)	3	32	4
20	S(3,4,20)	15	152	29
21	S(2,3,21)	4	32	7
	S(2,5,21)	1	2	1
22	S(3,4,22)	18	1,140	1,017
25	S(2,3,25)	4	240	12
26	S(3,5,26)	10	1	1
27	S(2,3,27)	5	144	8
31	S(2,3,31)	5	2,048	80
	S(2,6,31)	1	10	1
33	S(2,3,33)	6	1,600	84
37	S(2,3,37)	6	28,480	820
	S(2,4,37)	3	48	2
39	S(2,3,39)	7	18,048	798
40	S(2,4,40)	4	96	10
41	S(2,5,41)	2	8	1
43	S(2,3,43)	7	395,648	9,508
45	S(2,3,45)	8	278,784	11,616
49	S(2,3,49)	8	6,594,560	?
	S(2,4,49)	4	9,184	224
51	S(2,3,51)	9	4,474,112	?
52	S(2,4,52)	5	4,768	206
55	S(2,3,55)	9	121,098,240	?
57	S(2,3,57)	10	84,672,512	?
61	S(2,3,61)	10	2,542,203,904	?
	S(2,4,61)	5	1,087,552	18,132
	S(2,5,61)	3	416	10
63	S(2,3,63)	11	1,782,918,144	?
64	S(2,4,64)	6	385,536	12,048
65	S(2,5,65)	4	16	2

CHAPTER VI

Conclusions

This was a very time consuming project both in terms of research and in programming time. But, I feel that I have gained a lot of knowledge in the area of Steiner systems and in the broader area of t-designs. I would like to pursue research on cyclic t-designs where $\lambda \neq 1$. Changing the necessary modules in my code should be fairly easy, and doing this further work on t-designs would be interesting to me.

The current programs obtain the answers for small designs relatively quickly, but on large designs the size of the matrix is too large to search within a specified time frame. I would also try to change the implementation of some code to try to speed up the run time even more.

One optimization to iso that I would like to make, is to precompute the multiplication of the orbits. Thus, determining multiplier isomorphism would then be reduced to a traversal of a graph.

BIBLIOGRAPHY

[Anej80] Aneja, Y P. "An integer linear programming approach to the Steiner problem in graphs." Networks, Vol. 10 (1980), no. 2, 167-178.

[Blak76] Blake, Ian F. and Rahman, Mushfiqur. "A note on generalized Steiner systems." Utilitas Math., Vol. 9 (1976), 339-346.

[Boyc77] Boyce, William M. "An improved program for the full Steiner tree problem." ACM Trans. Math. Software, Vol. 3 (1977), no. 4, 359-385.

[Bruc] Bruck, R. H., and Ryser, H. J. "The nonexistence of certain finite projective planes." Canad. Jour. Math. Vol. 12,(1960), p 189-203.

[Chen72] Chen, Yi. "The Steiner system S(3, 6, 26)." Journal Geometry, Vol. 2 (1972), 7-28.

[Cho80] Cho, Chung Je. "On cyclic Steiner quadruple systems." Ars Combinatoria, Vol. 10 (1980), 123-130.

[Cho82] Cho, Chung Je. "Rotational Steiner triple systems." Discrete Math., Vol. 42 (1982), no. 2-3, 153-159.

[Colb80a] Colbourn, Charles J., Colbourn, Marlene J. and Phelps, Kevin T. "Combinatorial algorithms for generating cyclic Steiner quadruple systems." Univ. New Brunswick, Fredericton, N.B., 1980.

[Colb80b] Colbourn, Charles J. and Phelps, Kevin T. "Three new Steiner quadruple systems." Utilitas Math., Vol. 18 (1980), 35-40.

[Colb80c] Colbourn, Marlene J. "The analysis of Steiner systems." Utilitas Math., Winnipeg, Man., 1980.

[Colb80d] Colbourn, Charles J., and Colbourn, Marlene J. "A Recursive Construction for Infinite Families of Cyclic SQS", ARS Combinatoria, Vol. 10 (1980), 95-102.

[Colb80e] Colbourn, Marlene J., and Colbourn, Charles J. "Cyclic Steiner Systems having Multiplier Automorphisms", Utilitas Mathematica, Vol. 17 (1980), 127-149.

[Colb81b] Colbourn, Charles J. "Disjoint Cyclic Steiner Triple Systems", Congressus Numeratium, Vol. 32 (1981), 205- 212.

[Colb82b] Colbourn, Marlene J., Colbourn, Charles J., and Rosenbaum, Wilf L. "Trains: An Invariant for Steiner Triple Systems", ARS Combinatoria, Vol. 13 (1982), 149-162.

[Colb82c] Colbourn, Charles J. "Distinct Cyclic Steiner Triple Systems", Utilitas Mathematica, Vol. 22 (1982), 103-126.

[Colb82e] Colbourn, Charles J. "Computing the chromatic index of Steiner triple systems." Comput. Journal, Vol. 25 (1982), no. 3, 338-339.

[Colb88] Colbourn, Charles J., and Van Oorschot, Paul C. "Applications of Combinatorial Designs in Computer Science". Institute for Mathematics and its Applications, March 1988.

[Dien80] Diener, Immo. "On Cyclic Steiner Systems S(3,4,22)." Annals of Discrete Mathematics, Vol. 7 (1980), 301-313.

[Drie76] Driessen, L. M. H. E., Frederix, G. H. M. and van Lint, J. H. "Linear codes supported by Steiner triple systems." Ars Combinatoria, Vol. 1 (1976), no. 1, 33-42.

[Doye78] Doyen, Jean, and Rosa, Alexander. "An updated bibliography of Steiner systems." Annals of Discrete Math., Vol. 7 (1980) 317-349. This was updated again in 1989.

[Gibb76] Gibbons, Peter B. "Computing Techniques for the Construction and Analysis of Block Designs", University of Toronto, doctoral thesis (1976).

[Gion83] Gionfriddo, Mario. "Some results on partial Steiner quadruple systems." North-Holland, Amsterdam-New York, 1983.

[Gran82] Grannell, M. J. and Griggs, T. S. "A cyclic Steiner quadruple system of order 32." Discrete Math., Vol. 38 (1982), no. 1, 109-111.

[Hana60] Hanani, H. "On quadruple systems." Can. J. Math., Vol. 12 (1960), 145-157.

[Kram75] Kramer, Earl S. and Mesner, Dale M. "Admissible parameters for Steiner systems S(t, k, v) with a table for all (v-t) <= 498." Utilitas Math., Vol. 7 (1975), 211-222.

[Kram85] Kramer, E. S., Leavitt, D. W. and Magliveras, S. S. "Construction procedures for t-designs and the existence of new simple and 6-designs", Ann. of Discrete Math. Vol. 26 (1985), 247-274.

[Kreh86a] Kreher, Donald L. and Radziszowski, Stanislaw P. "Finding Simple t-Designs by using Basis Reduction", Congressus Numeratium, Vol. 55 (1986), 235-244.

[Kreh86b] Kreher, D. L. and Radziszowski, S. P. "The existence of simple 6-(14,7,4) designs", Journal of Combinatorial Theory, Vol. 43 (1986), 237-243.

[Lind78] Lindner, Charles C. and Rosa, Alexander. "Steiner quadruple systems – a survey." Discrete Math., Vol. 22 (1978), no. 2, 147-181.

[Magl84] Magliveras, S. S. and Leavitt, D. W. "Simple 6-(33,8,36) designs from $P\Gamma L_2(32)$", Computational Group Theory, Proceedings, London Math. Soc. Sympos. Comput. Group Theory, Academic Press, New York (1984) 337-352.

[Maso77] Mason, David R. "On the construction of the Steiner system S(5,8,24)." Journal of Algebra, Vol. 47 (1977), no. 1, 77-79.

[Phel80a] Phelps, K. T. "Infinite classes of cyclic Steiner quadruple systems." Ann. Discrete Math., Vol. 8 (1980), 177-181.

[Phel80b] Phelps, K. T. "On cyclic Steiner Systems S(3,4,20)." Annals of Discrete Mathematics, Vol. 7 (1980), 277-300.

[Phel87a] Phelps, K. T. "Isomorphism of Circulant Combinatorial Structures." ARS Combinatoria, Vol. 24A (1987), 195-210.

[Phel87b] Phelps, K. T. "Isomorphism Problems for Cyclic Block Designs." Annals of Discrete Mathematics, Vol. 34 (1987), 385-391.

[Poll78] Pollak, H O. "Some remarks on the Steiner problem." Journal of Combinatorial Theory Ser. A, Vol. 24 (1978), no. 3, 278-295.

[Radz88a] Radziszowski, Stanislaw P. and Kreher, Donald L. "Search Algorithm for Ramsey Graphs by Union of Group Orbits". Journal of Graph Theory, Vol. 12 (1988) 59-72.

[Radz88b] Radziszowski, Stanislaw P. and Kreher, Donald P. "Solving Subset Sum Problems with the L3 Algorithm". Journal of Combinatorial Mathematics and Combinatorial Computing, Vol. 3 (1988) 49-63.

[Stin] Stinson, D. R. "A short proof of the Nonexistence of a pair of Orthonogal Latin Squares of Order Six." Jour. Comb. Theory Series A Vol. 36 (1984), p. 373-376.

[Tarr] Tarry, G. "Le probleme de 36 officieurs." Compte Render de l'Association Francais pour l'Advancement de Science Natural, Vol. 1 (1900), p 122-123; Vol. 2 (1901) p. 170-203.

[Teir87] Teirlinck, L. "Non-trivial t-Designs without repeated blocks exist for all t". Discrete Math., Vol. 65 (1987) 301-311.

[Teir88] Teirlinck, L. "Locally trivial t-designs and t-designs without repeated blocks", preprint (1988).

[Trub82] Trubin, V A. "An algorithm for the solution of the Steiner problem on graphs." Mat. Issled. No. 68 (1982), 151-157, 185.

APPENDIX A

Makefile for the System

```
#       Author:         Timothy C. Frenz (tcf2154)
#       File Name:      makefile
#
#
# dependency section
#
# You must comment out the proper FLAGS line so the programs
# will compile for a specific system.
#
#FLAGS  = -D_3B2_310
FLAGS   = -D_3B2_600
#FLAGS  = -D_DEC
#FLAGS  = -DTurboC

#
#       make sure the user enters an option
#
make:
   @echo 'Make what? '
   @echo '  all, gmat, gmatc, find, iso, res, show, pos, poss, cyclic'

#
#       targets of the make command
#
all:
   make gmat
   make gmatc
   make find
   make iso
   make res
   make show
   make pos
   make poss
   make cyclic

gmat: gmat.o blocks.o mat.o
   cc -o gmat gmat.o blocks.o mat.o

gmatc: gmatc.o orbits.o matc.o
   cc -o gmatc gmatc.o orbits.o matc.o
```

```
find: find.o heap.o
   cc -o find find.o heap.o

iso:    iso.c globals.h mytime.h
   cc -O iso.c -o iso  $(FLAGS)

res:    res.c globals.h
   cc -O res.c -o res $(FLAGS)

show:        show.c globals.h
   cc -O show.c -o show $(FLAGS)

pos:    pos.c funct.h
   cc -O pos.c -o pos

poss:   poss.c
   cc -O poss.c -o poss

cyclic: cyclic.c funct.h
   cc -O cyclic.c -o cyclic

#
# Object file dependencies
#
gmat.o: gmat.c globals.h mytime.h
   cc -c -O gmat.c $(FLAGS)

blocks.o: blocks.c globals.h
   cc -c -O blocks.c $(FLAGS)

mat.o: mat.c globals.h
   cc -c -O mat.c $(FLAGS)

gmatc.o: gmatc.c globals.h mytime.h
   cc -c -O gmatc.c $(FLAGS)

orbits.o: orbits.c globals.h
   cc -c -O orbits.c $(FLAGS)

matc.o: matc.c globals.h
   cc -c -O matc.c $(FLAGS)

find.o: find.c globals.h mytime.h funct.h
   cc -c -O find.c $(FLAGS)

heap.o: heap.c
   cc -c -O heap.c
```

APPENDIX B

C Source code for System

I am only including the most important files used in finding Steiner systems. The files will be arranged in alphabetical order as follows.

1) find.c
2) funct.h
3) globals.h
4) gmatc.c
5) heap.c
6) iso.c
7) matc.c
8) orbits.c
9) res.c

```
/*------------------------------------------------------------------------
*
*       Author:         Timothy C. Frenz (tcf2154)
*       Date Written:   30-Jan-1988
*       Last Modified:  05-Apr-1989
*       File Name:      find.c
*       Computer:       AT&T 3B2/310
*       Op. System:     UNIX SYS 5 Rel 3.1
*       Purpose:
*               This code searches a binary Atk matrix for a solution to
*               a steiner system.  Under the current implementation this
*               finds both cyclic and noncylic Steiner systems that were
*               set up with the GMATC or GMAT program respectively.
*
*               The number of columns will be approximately (v choose k)/v
*               and the number of rows (v choose t)/v.  The number of bits
*               on in a column will be (k choose t) and hence the number
*               of bits on in a row will be (v-t choose k-t).
*
*       Subroutines:
*               pcmln()         - process command line arguments
*               rmdup()         - remove duplicates columns from matrix
*               build_list()    - build search list structure
*               dump_sol()      - dump solutions
*               GT()            - Greater than function
*               find()          - finds solutions in Atk matrix
*               main()          - controlling function
*/
#include        "funct.h"
#include        "mytime.h"
#ifdef TurboC
#include        <fcntl.h>
#include        <sys\stat.h>
#endif


/*      E X T E R N A L   F U N C T I O N S
*/
extern heapsort();      /*  This procedure performs a heapsort on the   */
                        /*  binary matrix; however, the data is not     */
                        /*  moved only the pointers into the data are   */
                        /*  changed.                                    */

/* global variables */

  PT    *A;
  unsigned long SOLS, MAX_DUMPS;
  unsigned long *ap, i;         /* pointer into Atk matrix, a counter */

/*------------------------------------------------------------------------
*
*       pcmln()
*
*       This subroutine processes the command line as input by the user.
*
*       Parameters:
*               argv    : command line argument vector
*               status  : flag for whether status information wanted
*               no_dump : flag for whether solutions should be saved
*               solfn   : file name to save the solutions in
*/
pcmln( argv, status, no_dump, solfn )
  char  *status, *no_dump, *argv[], solfn[];
{
  int   i, j;

  *status = *no_dump = 0;
  for ( i=2; argv[i]; i++ )
      for ( j=1; argv[i][j]; j++ )
          if ( argv[i][j] == 's' )
             *status = 1;
          else if ( argv[i][j] == 'n' )
             *no_dump = 1;

  (void) strcpy( solfn, "saa" );        /* Solution ## */
  (void) strcat( solfn, argv[1] );

} /* end of pcmln() */

/*------------------------------------------------------------------------
*
*       rmdup()
*
*       this procedure removes duplicate columns from the matrix Atk[].
*       unique[] is then used in place of ptr[] when searching for a
*       solution.  With a solution found, dup[] can be used in
*       conjunction with ptr to print all "duplicate" columns.
*
*       Parameters:
*               Atk     : The Atk matrix
*               ptr     : array of pointers into the matrix to make the
*                         matrix appear in descending order by column value
*               unique  : array which will not have duplicate column values
*               dup     : array consisting of group starters
*               rows    : number of (unsigned long) rows in the binary matrix
*               cols    : number of columns in the matrix
*/
unsigned long rmdup( Atk, ptr, unique, dup, rows, cols )
  unsigned long Atk[], ptr[], unique[], dup[], rows, cols;
{
  unsigned long r, i, j, cur, used=0;

  unique[0] = ptr[0];
  dup[0] = 0;
  for ( cur=1; cur < cols; cur++ )
    {
    i = unique[used] * rows;
    j = ptr[cur] * rows;
    for ( r=0; (r<rows) && (Atk[i]==Atk[j]); r++, i++, j++ );
    if ( r != rows )
       {
       unique[++used] = ptr[cur];
       dup[used] = cur;
       }
    } /* end of for */

  dup[ ++used ] = cols;

  return( used );
```

```
) /* end of rmdup */

/*-------------------------------------------------------------------------
*
*       build_list()
*
*       Using Atk[] and ptr[] I build a list[] which will point to the
*       next group of columns in Atk[] with the same high bit on.  next[]
*       will point to the next possible column that does not have any
*       bits in common with each of the columns.
*
*       Parameters:
*               Atk     : the Atk matrix
*               ptr     : array of pointers into Atk matrix (duplicates
*                         are removed at this point)
*               bits    : the actual number of rows (hence bits) in the
*                         binary Atk matrix
*               rows    : the number of (unsigned long) rows in the matrix
*               cols    : the number of columns
*               list    : array which will group together all columns that
*                         have the same high bit on that have not been
*                         previously used in another group
*               next    : array which for each column will point to the next
*                         possible column that could be included in the same
*                         solution space
*/
build_list( Atk, ptr, bits, rows, cols, list, next )
  unsigned long Atk[], bits, rows, cols, list[], ptr[], next[];
{
  unsigned long BIT, HIGHBIT, b=0,r=0,c=0,i,j, cpos, ipos, t;
                              /* bit, row, col */

  if ( bits < NUMBITS )
     BIT = (1 << (bits - 1));
  else
     BIT = HIGHBIT = (1 << (NUMBITS - 1));

  while (( r < rows ) && ( c < cols ))
    {
    for ( ; (c < cols) && (BIT & ( Atk[ ptr[c] * rows + r ] )); c++ );
    list[b++] = c;      /* first one in next group */
    BIT >>= 1;
    if ( ! BIT )
       if ( ++r != (rows-1) )
          BIT = HIGHBIT;
       else
          BIT = (1 << (bits%NUMBITS - 1));
    ) /* end of while */

  /* find the next column that does not have any bit in common */
  for ( c=0; c<cols; c++ )
    {
    cpos = ptr[c] * rows;
    for ( t=0, i=c+1; (i<cols) && (!t); i++ )
        {
        ipos = ptr[i] * rows;
        for ( r=0; ((r<rows) && (!(Atk[cpos+r] & Atk[ipos+r]))); r++ );
        if ( r == rows ) t = i;
        )
    next[c] = t;
    )
  )/* end of build_list */

/*-------------------------------------------------------------------------
*
*       dump_sol()
*
*       This procedure dumps out the solutions.
*
*       Parameters:
*               df      : data file, created by GMATC or GMAT, which
*                         is used to print out the solutions found
*               sf      : the solution file, where the answers are stored
*               solfn   : the solution file name
*               k       : the normal meaning in a steiner system S(t,k,v)
*               used    : the number of columns used in the solution
*               ans     : array of pointers that consist of the solution
*               oldptr  : the output of heapsort()
*               dup     : array of duplicate column values in ptr array
*               index   : used to print out all duplicate columns, the current
*                         index that is being processed in the ans array
*               tmp     : temporary answer array, recursively filled to print
*                         out the solutions
*               nd      : the no dump flag
*/
dump_sol( df, sf, solfn, k, used, ans, oldptr, dup, index, tmp, nd )
  unsigned long ans[], oldptr[], dup[], index, tmp[];
  FILE *df, *sf;
  char nd, solfn[];
  short used;
  PT   k;
{
  unsigned long i;
  int  sfh;

  if ( index < used )
     {
     for ( i=dup[ans[index]]; i<dup[ans[index]+1]; i++ )
         {
         tmp[index] = oldptr[i];
         dump_sol( df, sf, solfn, k, used, ans, oldptr, dup, index+1, tmp, nd );
         )
     ) /* end of then */
  else
     {
     SOLS++;
     fwrite( &used, sizeof( used ), 1, sf );
     if ( !nd )
        {
        for (i=0; i<used ; i++)
            {
            fseek( df, (long) (k * tmp[i]), 0 );
            fread( A, k * sizeof(PT), 1, df );
            fwrite( A, k * sizeof(PT), 1, sf );
            )

        if ( ! ( SOLS % MAX_DUMPS ))
           {
           fclose( sf );
```

```
            nth_file( solfn, SOLS/MAX_DUMPS );
#ifdef TurboC
            sfh = open( solfn, O_WRONLY|O_BINARY|O_CREAT|O_TRUNC, S_IWRITE );
            sf = fdopen( sfh, "w" );
#else
            sf = fopen( solfn, "w" );
#endif
            }
        } /* end of not no_dump, thus print answers */

    } /* end of else */

} /* end of dump_sol() */

/*-------------------------------------------------------------------------
*
*       GT()
*
*       This function returns true when the highest bit in the jth column
*       of the Atk matrix is greater than the highest bit in what is left
*       to get.
*
*       Parameters:
*               left    - what is left to get, initially this is an all
*                         1 column, for a solution, this is all 0
*               Atk     - the Atk matrix mapped into one dimension
*               j       - the column we are examining
*               rows    - the number of (unsigned long) rows in the Atk matrix
*/
short GT( left, Atk, j, rows )
  unsigned long Atk[], left[], j, rows;
{
  ap = &Atk[j * rows];
  for ( i=0; !(*ap | left[i]); i++, ap++ );

  return( (*ap ^ left[i]) > left[i] );

} /* end of GT() */

/*-------------------------------------------------------------------------
*
*       find()
*
*       This function finds a number (sol or MAXINT) in the matrix m by
*       combining various members of m to be of the specified size.
*
*       Parameters:
*               sol     : what is being solved for in the LAST row, all
*                         other rows are solved for MAXINT.
*               Atk     : the Atk matrix to search for a solution in
*               ptr     : a sorted pointer array into Atk, duplicates removed
*               list    : an array of columns with common high bits
*               rows    : the number of (unsigned long) row
*               cols    : the number of columns
*               fn      : the file name of the column orbits/blocks
*               c       : the number of entries in a column orbit/block
*               orbs    : the number of orbits/blocks which will probably be
*                         used to generate a solution
*               solfn   : the file to store the solutions in
*               nd      : to dump or not dump the solutions
*               oldptr  : has duplicate entries
*               dup     : has positions of duplicate entries in oldptr
*               next    : next possible position within ptr[] to find
*                         a column with no bitwise AND.
*/
find( sol, Atk, ptr, list, rows, cols, fn, c, orbs, solfn, nd,
      oldptr, dup, next )
  PT    c;
  unsigned long orbs, Atk[], sol, ptr[], list[], rows, cols,
                oldptr[], dup[], next[];
  char  fn[], solfn[], nd;
{
  int   dfh, sfh;
  short done=0, Possible=1, bitand, used=0;
  unsigned long i, j=0, pos, col=0, I,
                *to_go, *ans, *tmp, MAXINT=0, *Ap;
  FILE  *df, *sf, *rf;          /* data, solution and restore files */

  /* initialize globals */
  a_alloc( A, c+1, PT );
  SOLS = 0;

#ifdef TurboC
  if ( (dfh = open( fn, O_RDONLY | O_BINARY )) == -1 )
#else
  if ( (df = fopen( fn, "r" )) == NULL )
#endif
     printf("%d not found\n", fn ), exit(1);

#ifdef TurboC
  df = fdopen( dfh, "r" );
  sfh = open( solfn, O_RDWR | O_BINARY | O_APPEND );
  sf = fdopen( sfh, "a" );
#else
  sf = fopen( solfn, "a+" );
#endif

  for ( i=0; i<NUMBITS; i++ )
     MAXINT = (MAXINT<<1) + 1;

  a_alloc( to_go, rows+1, unsigned long );
  a_alloc( ans, orbs+1, unsigned long );
  a_alloc( tmp, orbs+1, unsigned long );
  for ( i=0; i<(rows-1); i++ )          /* initialize to wanted answer */
    to_go[i] = MAXINT;
  to_go[rows-1] = sol;


  while ( Possible )
    {
    while (( ! done ) && ( Possible ))
      {
      /*
      *   The following code determines if there is a bitwise AND between
      *   Atk and the complement of to_go.  to_go is what is left to get, so
      *   the bitwise negation of to_go is what has already been gotten.  Thus,
      *   this checks to see if the colth column of the Atk matrix has any
      *   bits in common with any of the columns already in the partial
      *   solution.  If there are, the column can not be used because this
      *   would mean that some t-set would intersect some k-set more than
```

```
*   one time.
*/
bitand = 1;
while (( col < list[j] ) && bitand )
    {
    Ap = &Atk[ptr[col] * rows];
    for ( I=0; ((I<rows) && (!(*Ap & (-to_go[I])))); I++, Ap++ );
    if ( bitand = ( I != rows )) col++;
    )

if ( col < list[j] )
    {
    /* a candidate for inclusion into the partial solution, so add it
    *  to the list of what was used so far, then determine what the next
    *  column to start searching at should be.
    */
    done=1;
    pos = ptr[col] * rows;
    for ( i=0; i<rows; i++ )
      if ( to_go[i]  ^= Atk[pos++] ) done=0;          /* subtract */
    ans[used++] = col;
    while ((list[j] < cols) && GT( to_go, Atk, ptr[list[j]], rows )) j++;
    while ( list[j] && (list[j] == list[j+1]) ) j++;
    j++;
    col = ( list[j-1] < next[col] ? next[col] : list[j-1] );
    )
 else if ( used )
    {
    /* last column added as part of the solution can not be part of the
    *  solution, so remove it and start searching where I left off
    */
    col = ans[--used] ;
    pos = ptr[col++] * rows;
    for ( i=0; i<rows; i++ )
        to_go[i]  |= Atk[pos++] ;                     /* add */
    for ( ; j && (list[j-1] >= col); j-- );
    )
 else
    Possible = 0;

 ) /* end of WHILE */

if ( done ) /* a solution was found */
   {
   if ( SOLS == 0 )
      MAX_DUMPS = (MAX_FILE_SIZE - 25) /
                  ( c * used * sizeof( PT ) + sizeof(used) );
   dump_sol( df, sf, solfn, c, used, ans, oldptr, dup,
             (unsigned long) 0, tmp, nd );

   /* remove last column selected and continue searching by resetting
   *  the done flag.
   */
   col = ans[--used] ;
   pos = ptr[col++] * rows;
   for ( i=0; i<rows; i++ )
       to_go[i]  |= Atk[pos++] ;
   for ( ; j && (list[j-1] >= col); j-- );
   done = 0;
   ) /* end of if ( done ) */

    ) /* end of while Possible */

  fclose( df );
  fclose( sf );

  free( A );
  free( to_go );
  free( ans );

  if ( SOLS )
     printf("%u SOLUTIONS FOUND\n\n", SOLS );
  else
     printf("NO SOLUTIONS FOUND\n\n");

) /* end of find() */

/*---------------------------------------------------------------------------
*
*       main()
*
*       This makes sure the proper number of arguments were entered, then
*       calls pcmln(), then processing continues.
*
*/
main( argc, argv )
  int  argc    ;                          /*  Number of command agruments */
  char *argv[] ;                          /*  Pointers to each argument   */
{
  char  kfn[FNL], solfn[FNL], cyclic, status, no_dump;
  PT    t, k, v, lambda;
  int   mfh;
  unsigned long blocks, orbits, rows, cols, ncols, req_rows, *ptr, *list, i,
                *mat, sol, *unique, *dup, *next;
  long  j, start_time, cur_time, last_time;
  FILE  *mf;

  get_time( &start_time );
  if ( argc < 2 )
     fprintf( stderr, "Usage: %s mfn [-srn]\n", argv[0] ), exit(1);

  /* check for matrix file and get header info */
#ifdef TurboC
  if (( mfh = open( argv[1], O_RDONLY | O_BINARY )) == -1 )
#else
  if (( mf = fopen( argv[1], "r" )) == NULL )
#endif
     fprintf( stderr, "ERROR - %s not found\n", argv[1] ), exit(1);
#ifdef TurboC
  mf = fdopen( mfh, "r" );
#endif
  fread( &t, sizeof(t), 1, mf );
  fread( &k, sizeof(k), 1, mf );
  fread( &v, sizeof(v), 1, mf );
  fread( &lambda, sizeof(lambda), 1, mf );
  fread( &cyclic, sizeof(cyclic), 1, mf );
  fread( &rows, sizeof(rows), 1, mf );
  fread( &cols, sizeof(cols), 1, mf );
  fread( kfn, FNL, 1, mf );
```

```
/* check for other paramters, and set up the file names */
pcmln( argv, &status, &no_dump, solfn );

/* calculate number of blocks and orbits */
blocks = num_blk( t, k, v, lambda );
orbits = num_orb( k, v, blocks ) + v;

req_rows = ( rows + NUMBITS - 1 ) / NUMBITS;
                    /* result of compressing */

/* REQUESTING MEMORY */
if ( status ) printf("requesting memory : %d bytes\n",
                  (req_rows*cols+cols+rows)*sizeof(unsigned long) );
a_alloc( mat, req_rows * cols+1, unsigned long);
a_alloc( ptr,    cols+1, unsigned long );
a_alloc( next,   cols+1, unsigned long );
a_alloc( list,   rows+1, unsigned long );
a_alloc( unique, cols+1, unsigned long );
a_alloc( dup,    cols+1, unsigned long );

/* READING IN compressed binary MATRIX from matfn */
if ( status ) printf("reading in matrix\n");
fread( mat, sizeof(unsigned long) * cols * req_rows, 1, mf );
(void) fclose( mf );

/* CREATING SOLUTION VECTOR */
if (( i = rows % NUMBITS ) == 0 ) i = NUMBITS;
sol = (((1 << (i-1)) - 1) << 1) | 1; /* sol has lower i bits on */

if ( status ) printf("PERFORMING HEAPSORT ON MATRIX, ");
for ( i=0; i<cols; i++ )
  ptr[i] = i;
heapsort( mat, ptr, req_rows, cols );
if ( status ) get_time( &cur_time ) ,
              print_time( cur_time-start_time ),

              printf("REMOVING DUPLICATES, had %lu, ", cols ),
              last_time = cur_time;
ncols = rmdup( mat, ptr, unique, dup, req_rows, cols );
if ( status ) get_time( &cur_time ),
              printf("%lu left, ", ncols ),
              print_time( cur_time-last_time ),

              printf("BUILDING SEARCH LIST STRUCTURE, "),
              last_time = cur_time;
build_list( mat, unique, rows, req_rows, ncols, list, next );
if ( status ) get_time( &cur_time ) ,
              print_time( cur_time-last_time ),
              printf("Total time so far was "),
              print_time( cur_time-start_time ),

              printf("SEARCHING FOR ANSWER\n");

/* save header stuff in solution file */
#ifdef TurboC
  mfh = open( solfn, O_WRONLY | O_BINARY | O_CREAT | O_TRUNC, S_IWRITE );
  mf = fdopen( mfh, "w" );
#else
  mf = fopen( solfn, "w" );
#endif
  fwrite( &t, sizeof(t), 1, mf );
  fwrite( &k, sizeof(k), 1, mf );
  fwrite( &v, sizeof(v), 1, mf );
  fwrite( &lambda, sizeof(lambda), 1, mf );
  fwrite( &cyclic, sizeof(cyclic), 1, mf );
  fclose( mf );

  /* go find the solutions */
  if ( status ) printf("beginning find\n");
  if ( cyclic )
     find( sol, mat, unique, list, req_rows, ncols, kfn, k,
           orbits, solfn, no_dump, ptr, dup, next );
  else
     find( sol, mat, unique, list, req_rows, ncols, kfn, k,
           blocks, solfn, no_dump, ptr, dup, next );

  /* free memory */
  free( mat );
  free( ptr );
  free( list );

  /* display total time used */
  printf("Total time consumed for S(%d,%d,%d) was ", t, k, v );
  get_time( &cur_time );
  print_time( cur_time-start_time );

) /* end of main() */
/*-------------------------------------------------------------------------
*
*    Author:    Timothy C Frenz (tcf2154)
*    Language:  C
*    Purpose:   To provide some common functions to various programs.
*    File Name: funct.h
*
*       Functions:
*              choose  - given parameters x and y, returns x choose y.
*              num_blk - given t, k, v, and lambda returns the number of
*                        blocks thaty would be required in a t-design of
*                        the given parameters.
*              num_orb - given k, v and the number of blocks, this
*                        function returns the minimum number of orbits
*                        required in the design.
*/

#include        "globals.h"

char    EXACT;   /* used exclusively by cyclic */


/*-------------------------------------------------------------------------
*
*       choose
*/
unsigned long choose( X, Y )
  PT X, Y;
{
  unsigned long ans, i;

  if ( Y > X/2 ) Y = X - Y;
  ans = X;
```

```
  for ( i=2; i<=Y; i++ )
    {
    ans *= --X;
    ans /= i;
    }

  return( ans );

) /* end of choose */


/*---------------------------------------------------------------------
*
*       num_blk
*/
unsigned long num_blk( t, k, v, lambda )
  PT    t, k, v, lambda;
(
  unsigned long num, den;

  num = lambda * choose( v, t );
  den = choose( k, t );

  if ( num % den )
     return( 0 );
  else
     return( num/den );

) /* end num_blk() */


/*---------------------------------------------------------------------
*
*       num_orb
*/
unsigned long num_orb( k, v, blocks )
  PT    k, v;
  unsigned long blocks;
(
  unsigned long i, j, rem, orbits;

  EXACT=0;
  orbits = blocks / v;
  rem    = blocks % v;

  i = 1;
  while ( rem && (i++ < k))
    {
    if ( i == k )
       {
       if ( (!(v % k)) && ( rem == (v/k) ))
          orbits++, EXACT++, rem=0;
       )
    else if ( (!(k % i)) && (!(v % i)) )
       for ( EXACT=2, j=v/i; rem >= j; orbits++, rem -= j );
    )

  return( ( rem ? 0 : orbits ) );

) /* end of num_orb() */


/*---------------------------------------------------------------------
*
*    Author:    Timothy C Frenz (tcf2154)
*    Language:  C
*    Purpose:   This is the declaration file used by all source files
*               in this project.
*    File Name: globals.h
*
*/

/*      R U N - T I M E   L I B R A R I E S
*/
#include        <stdio.h>               /*  I/O Functions               */
#include        <string.h>              /*  String Functions            */

/*      C O N S T A N T S
*/
#define         NUMBITS         (sizeof(unsigned long) * 8)
#define         FNL             12      /* maximum File Name Length     */

#ifdef   _3B2_310
#define         MAX_FILE_SIZE   1048576
#else    /* _3B2_600    or    _DEC    or    TurboC */
#define         MAX_FILE_SIZE   4194304
#endif

/*      T Y P E ' S
*/
typedef unsigned char PT;


/*      M A C R O ' S
*/

/*      nth_file() gets the next file name for files ?xx*
*                       where the xx is two characters
*/
#define         nth_file( fn, n )                                       \
                (                                                       \
                fn[1] = (n)/26 + 'a';                                   \
                fn[2] = (n)%26 + 'a';                                   \
                )


/*      a_alloc() allocates space for an array
*/
#define         a_alloc( a, s, type )                                   \
                (                                                       \
                a = (type *) calloc( (unsigned) s, sizeof(type) );      \
                if ( a == NULL )                                        \
                   printf("ERROR -- out of memory\n"), exit(1);         \
                )


/*      new() allocates space for one data type
*/
```

```
#define         new( type )     ((type *) malloc( sizeof(type) ))

/*      swap() swaps two somethings
*/
#define         swap( a, b, type )              \
                {                               \
                  type t = a;                   \
                                                \
                  a = b;                        \
                  b = t;                        \
                }
/*--------------------------------------------------------------------------
*
*       Author:         Timothy C. Frenz (tcf2154)
*       Language:       C
*       Date Written:   30-Jan-1988
*       Last Modified:  13-Dec-1988
*       File Name:      gmatc.c
*       Computer:       AT&T 3B2/310
*       Op. System:     UNIX SYS 5 Rel 3.1
*       Purpose:
*               This program sets up the matrix according to the parameters
*               passed in.  This program always creates the matrix for
*               cyclic steiner designs.
*
*       Subroutines:
*               main()          - The main procedure of this project
*               pcmln()         - Proces the command line arguments
*
*/

#include        "funct.h"
#include        "mytime.h"              /* use my time file             */
#ifdef TurboC
#include        <fcntl.h>
#include        <sys\stat.h>
#endif

/*      E X T E R N A L   F U N C T I O N S
*/
extern gen_orb();       /*  This procedure generates orbits for a given */
                        /*  steiner system.                             */
extern gen_matc();      /*  This procedure generates the binary matrix  */
                        /*  for cyclic steiner systems                  */
extern char EXACT;

/*--------------------------------------------------------------------------
*
*       pcmln()
*
*       This subroutine processes the command line as input by the user.
*
*       Parameters:
*               argv    - command line arguments
*               t,k,v   - normal meaning in an S(t,k,v) design
*               status  - flag for printing status information
*               tfn     - file name to store the t-orbits in
*               kfn     - file name to store the k-orbits in
*               new_kfn - file name to store the k-orbits that are used in
*                         the Atk matrix in
*/
pcmln( argv, t, k, v, status, tfn, kfn, new_kfn )
  PT    *t, *k, *v;
  char  *status, *argv[], tfn[], kfn[], new_kfn[];
{
  PT    i, tmp;

  for ( i=0, tmp=0; argv[1][i] != '\0'; i++ )
      tmp = 10 * tmp + argv[1][i] - '0';
  *t = tmp;
  for ( i=0, tmp=0; argv[2][i] != '\0'; i++ )
      tmp = 10 * tmp + argv[2][i] - '0';
  *k = tmp;
  for ( i=0, tmp=0; argv[3][i] != '\0'; i++ )
      tmp = 10 * tmp + argv[3][i] - '0';
  *v = tmp;

  *status = 0;
  if ( argv[5] )
     for ( i=0; argv[5][i]; i++ )
         if ( argv[5][i] == 's' )
            *status = 1;

  /* In the next statements, let t=3, k=5, v=41
  */
  (void) strcpy( tfn, argv[1] );
  (void) strcat( tfn, "." );
  (void) strcat( tfn, argv[3] );      /* tfn="3.41" */

  (void) strcpy( kfn, argv[2] );
  (void) strcat( kfn, "." );
  (void) strcat( kfn, argv[3] );      /* kfn="5.41" */

  (void) strcpy( new_kfn, "n" );
  (void) strcat( new_kfn, kfn );     /* new_kfn="n5.41" */
  for (i=0; new_kfn[i]; i++ );
  for ( ; i<FNL; new_kfn[i++]=0 );

) /* end of pcmln() */

/*--------------------------------------------------------------------------
*
*       main()
*
*       This makes sure the proper number of arguments were entered, then
*       calls pcmln(), then processing continues.  This assumes that the
*       parameters allow a cyclic design to exist.
*
*       Note: I include lambda as part of the parameters stored in the header
*       of the output file to allow for easily modifiying the code to look
*       for t-designs
*/
main( argc, argv )
  int  argc    ;                      /*  Number of command agruments */
  char *argv[] ;                      /*  Pointers to each argument   */
{
  char  tfn[FNL], kfn[FNL], new_kfn[FNL], status, cyclic=1;
  PT    t,k,v, lambda=1;
  int   mfh;
```

```
 long  j;
 unsigned long t_orbs, k_orbs, start_time, cur_time, last_time, offset;
 FILE  *mf;

 get_time( &start_time );
 if (( argc != 5 ) && ( argc != 6 ))
    printf("Usage: %s t k v mfn [-s]\n", argv[0] ), exit(1);

 pcmln( argv, &t, &k, &v, &status, tfn, kfn, new_kfn );

 if ( status ) printf("GENERATING %d-orbits in %d points, ", t, v );
 t_orbs = gen_orb( t, v, tfn, 0 );
 if ( status ) get_time( &cur_time ),
               printf("%lu %d-orbits found in ", t_orbs, t ),
               print_time( cur_time-start_time ),
               last_time = cur_time;

 if ( status ) printf("GENERATING %d-orbits in %d points, ", k, v );
 k_orbs = gen_orb( k, v, kfn, ((t==2) && (EXACT & 1)) );
 if ( status ) get_time( &cur_time ),
               printf("%lu %d-orbits found in ", k_orbs, k ),
               print_time( cur_time-last_time ),
               last_time = cur_time;

 /* save header stuff for find */
#ifdef TurboC
 mfh = open( argv[4], O_WRONLY | O_BINARY | O_CREAT | O_TRUNC, S_IWRITE );
 mf = fdopen( mfh, "w+" );
#else
 mf = fopen( argv[4], "w+" );
#endif
 fwrite( &t, sizeof(t), 1, mf );
 fwrite( &k, sizeof(k), 1, mf );
 fwrite( &v, sizeof(v), 1, mf );
 fwrite( &lambda, sizeof(lambda), 1, mf );
 fwrite( &cyclic, sizeof(cyclic), 1, mf );
 fwrite( &t_orbs, sizeof(t_orbs), 1, mf );
 offset = ftell( mf );
 fwrite( &k_orbs, sizeof(k_orbs), 1, mf );
 fwrite( new_kfn, FNL, 1, mf );

 if ( status ) printf("GENERATING the matrix\n");
 k_orbs = gen_matc( t, k, t_orbs, k_orbs, tfn, kfn, new_kfn, mf );
 if ( status ) get_time( &cur_time ) ,
               printf("%d k-orbits left, matrix gen took ", k_orbs ),
               print_time( cur_time-last_time ),
               last_time = cur_time;

 /* change k_orbs in matrix file, as that might have changed */
 fseek( mf, offset, 0 );
 fwrite( &k_orbs, sizeof(k_orbs), 1, mf );
 fclose( mf );

 /* display total time used */
 printf("%d intersection vectors left in search space\n", k_orbs );
 get_time( &cur_time );
 printf("Time consumed for matrix generation of S(%d,%d,%d) was ", t,k,v );
 print_time( cur_time-start_time );

) /* end of main() */
```

```
/*-------------------------------------------------------------------------
*
*       Author:         Timothy C. Frenz (tcf2154)
*       Language:       C
*       Date Written:   31-Jan-1988
*       Last Modified:  17-Mar-1988
*       File Name:      heap.c
*       Computer:       AT&T 3B2/310
*       Op. System:     UNIX SYS 5 Rel 3.1
*       Purpose:
*               This section of the program performs a heapsort (via
*               cursors) on the columns of the Atk matrix.
*
*       Subroutines:
*               heapsort()      - Performs the sort on the array
*/

#include        "globals.h"             /*  Use my declaration file     */


/*-------------------------------------------------------------------------
*
*       MIN()
*
*       Find the minimum value at the subscripts i1 and i2 if they are
*       within the range of the array (matrix).  If not return zero.
*
*       Parameters:
*               min     - contains the subscript whose value in the Atk
*                         array has the lowest value.
*               Atk     - the Atk matrix (in one dimension)
*               ptr     - the array of cursors into the Atk matrix
*               i1,i2   - two subscripts under consideration
*               size    - the number of columns in the matrix
*               rows    - the number of (unsigned long) rows
*/
MIN( min, Atk, ptr, i1, i2, size, rows )
  unsigned long *min, Atk[], ptr[], i1, i2, size, rows;
{
  unsigned long j, p1, p2;

  if ( i2 < size )
     {
     p1 = ptr[i1] * rows;    /* offset */
     p2 = ptr[i2] * rows;
     for (j=0; (j<rows-1) && (Atk[p1+j] == Atk[p2+j]); j++ );
     if ( Atk[p1+j] < Atk[p2+j] )
        *min = i1;
     else
        *min = i2;
     )
  else
     if ( i1 < size )
        *min = i1;
     else
        *min = 0;

) /* end of MIN() */
```

```
/*-------------------------------------------------------------------------
*
*       LESS
*
*       This function returns true if the element pointed to by p1 is less
*       than the second element pointed to by p2.
*/
int LESS( Atk, p1, p2, rows )
  unsigned long Atk[], p1, p2, rows;
{
  unsigned long j;

  p1 *= rows;
  p2 *= rows;
  for (j=0; (j<rows-1) && (Atk[p1+j] == Atk[p2+j]); j++ );

  return( Atk[p1+j] < Atk[p2+j] );

) /* end of LESS() */

/*-------------------------------------------------------------------------
*
*       heapsort()
*
*       Given an array Atk with size elements, where each element could be an
*       array of integers,
*               Atk[0], Atk[1], Atk[2], ... Atk[size-1]
*       sort it by the heapsort algorithm so that the descending sorted
*       array will be
*               Atk[ptr[0]], Atk[ptr[1]], . . . Atk[ptr[size-1]]
*       ptr[] is initially sent in so that ptr[i] = i.
*/
heapsort(Atk, ptr, rows, size)
  unsigned long Atk[], ptr[], size, rows;
{
  unsigned long tsize=size, par, cur=1, node, c1, c2, min, i;

  while ( cur < size )
    (
    node = cur++;
    par = (node-1) / 2;
    while (( node ) && ( LESS( Atk, ptr[node], ptr[par], rows )))
      (
      swap( ptr[node], ptr[par], unsigned long );
      node = par;
      par = (node-1) / 2;
      ) /* bubble up from a node */
    ) /* end of while, puts in HEAP format, smallest at front */

  while ( tsize > 1 )
    (
    --tsize;
    swap( ptr[0], ptr[tsize], unsigned long );
    node = 0;
    c1 = 1;
    c2 = 2;
    MIN( &min, Atk, ptr, c1, c2, tsize, rows );
    while (( min ) && ( LESS( Atk, ptr[min], ptr[node], rows )))
      (
      swap( ptr[node], ptr[min], unsigned long );
      node = min;
      c1 = node * 2 + 1;
      c2 = c1 + 1;
      MIN( &min, Atk, ptr, c1, c2, tsize, rows );
      ) /* end of bubble down */
    ) /* end of while, transforming from HEAP into sorted array */

) /* end of heapsort() */
/*-------------------------------------------------------------------------
*
*       Author:         Timothy C. Frenz (tcf2154)
*       Language:       C
*       Date Written:   15-Jul-1988
*       Last Modified:  05-Apr-1989
*       File Name:      iso.c
*       Computer:       AT&T 3B2/310
*       Op. System:     UNIX SYS 5 Rel 3.1
*       Purpose:
*               This code calculates which solutions that were produced by
*               FIND are isomorphic to each other and saves one design from
*               each of these groups in the solution file.
*
*       Subroutines:
*               pcmln()         - Process the command line arguments
*               factors()       - creates an array of numbers that are
*                                 relatively prime to v
*               insert()        - insert ni-solutions into a tree and a
*                                 linked list
*               find()          - find integer representation of orbit in
*                                 tree and return an array of pointers
*                                 into the linked list
*               same()          - determines if a design is the same
*                                 (isomorphic) to one of the ni-solutions
*               checkc()        - builds the array of integer representations
*                                 of the orbits to determine if it is
*                                 isomorphic to another
*               main()          - The main procedure of this project
*
*       Data Structures:
*               I store the orbits as unsigned long integers in a binary
*               tree where each node of the tree points to a linked list
*               of pointers to the ni-solutions containing that orbit.
*               Since I am storing the orbits of length k as an integer this
*               limits the size of v.  The following table shows this
*               limitation:
*               k     3      4    5   6   7   8   9  10  11  12  13  14  15  16
*               v  65534   1629  260  90  47  31  25  21  20  19  19  19  19  20
*
*               As k gets large, v = k + 1.
*
*               Note that for the k values of 3, 4 and 5 the v value limitation
*               does not matter since I have implemented v as an unsigned char,
*               thus limiting its size to 255.  I doubt for the other values of
*               k if I would search for a design with v greater than this limit.
*               For example, with k=6 and v=90 there would be 6,917,940
*               k-orbits.  With k=7 and v=47 there would be 1,338,120 k-orbits.
*               Thus as you can see this is quite a large number of orbits
*               to search through.
*
```

```
*               If for some reason the representation of orbits seems to
*               constricting, or a super fast 32 bit machine is developed,
*               you may leave the orbits represented by character strings
*               in both the binary tree and the linked list.
*/

#include        "globals.h"             /*  Use my declaration file     */
#include        "mytime.h"              /* use my time file             */
#ifdef TurboC
#include        <fcntl.h>
#include        <sys\stat.h>
#endif
int     T, I, J;
PT      *Blk, *Blk2, *Fact, C;
unsigned long *Num;

/*      The following record is used for the ni-solutions.  All
*       ni-solutions for a Steiner system are stored in this record.
*/
typedef struct ni_sol (
        unsigned long *orb;
        ) NI_SOL;

/*      The following record is the linked list structure off of each node
*       in the binary tree.  Each node contains a pointer to the next node
*       and an array of pointers to ni-solutions.
*/
#define LSIZE   16
typedef struct lnode (
        struct lnode *next;
        NI_SOL *sol[LSIZE];
        ) LNODE;

/*      The following record is used for the nodes in the binary tree.
*       value is the integer representation of the orbit that is currently
*       stored at this location.  cnt is the number of ni-solutions with
*       this orbit in the first array.
*/
typedef struct tnode (
        struct tnode *left, *right;
        unsigned long value, cnt;
        LNODE *list;
        /* unsigned long *mult;
        *       at some point I would precompute the multipliers of each
        *       orbit and store this number here.  This would be done
        *       when a new ni-solution is added to the tree.  Then to
        *       determine whether two designs are isomorphic, one would
        *       just get the numbers from this list and traverse the
        *       tree.  This should greatly improve on the time it takes
        *       to compute the number of ni-solutions to a given design.
        */
        ) TNODE;
/*-------------------------------------------------------------------------
*
*       pcmln()
*
*       This subroutine processes the command line as input by the user.
*
*       Parameters:
*               argv    : command line arguments
*               status  : flag for status information
*               solfn   : FIND solutions file, input for this program
*               nisofn  : solution file for ISO (this program)
*/
pcmln( argv, status, solfn, nisofn )
  char  *status, *argv[], solfn[], nisofn[];
(
  int   i, j;

  *status = 0;
  for ( i=2; argv[i]; i++ )
      for ( j=1; argv[i][j]; j++ )
          if ( argv[i][j] == 's' )
             *status = 1;

  (void) strcpy( solfn, "saa" );        /* Solution ## */
  (void) strcat( solfn, argv[1] );

  (void) strcpy( nisofn, "naa" );       /* Non-isomorphic solution ## */
  (void) strcat( nisofn, argv[1] );

) /* end of pcmln() */


/*-------------------------------------------------------------------------
*
*       factors()
*
*       This procedure sets all numbers 2, 3, .. (v-1) on, if they have
*       no common factors with v.  This is used for the cyclic designs,
*       where only the ones on are possible candidates for mappings.
*
*       Parameter:
*               v       : usual meaning in S(t,k,v) system
*/
factors( v )
  PT    v;
(
  PT    i,j,h=v/2+1;

  for ( i=2; i<v; i++ )
      Fact[i]=1;

  for ( i=2; i<h; i++ )
      if ( ! ( v%i ))
         for ( j=i; j<v; j+=i )
             Fact[j]=0;

  for ( j=0, i=2; i<v; i++ )
    if ( Fact[i] ) Fact[j++] = i;

  for ( ; j<v; Fact[j++] = 0 );

) /* end of factors() */

/*-------------------------------------------------------------------------
*
*       sort()
*
```

```
*/
sort( A, size )
  unsigned long A[], size;
{
  unsigned long tsize=size, par, cur=1, node, c1, c2, max;

  while ( cur < size )
    {
    node = cur++;
    par = (node-1) / 2;
    while (( node ) && ( A[node] > A[par] ))
      {
      swap( A[node], A[par], unsigned long );
      node = par;
      par = (node-1) / 2;
      ) /* bubble up from a node */
    ) /* end of while, puts in HEAP format, largest at front */

  while ( tsize > 1 )
    {
    --tsize;
    swap( A[0], A[tsize], unsigned long );
    node = 0;
    c1 = 1;
    c2 = 2;
    max = ( c2 < tsize ? ( A[c2] > A[c1] ? c2 : c1 ):( c1 < tsize ? c1 : 0 ));
    while (( max ) && ( A[max] > A[node] ))
      {
      swap( A[node], A[max], unsigned long );
      node = max;
      c1 = node * 2 + 1;
      c2 = c1 + 1;
      max = ( c2 < tsize ? ( A[c2] > A[c1] ? c2 : c1 ):( c1 < tsize ? c1 : 0 ));
      ) /* end of bubble down */
    ) /* end of while, transforming from HEAP into ascending sequence */

) /* end of sort() */

/*-------------------------------------------------------------------------
*
*       insert()
*
*       This procedure inserts the orbits from a given solution into a tree
*       and a linked list.
*
*       Parameters:
*               tptr    : binary tree pointer
*               Num     : the integers representing the orbits of a design
*               orbits  : the number of orbits in a solution
*/
insert( tptr, Num, orbits )
  TNODE  **tptr;
  unsigned long *Num, orbits;
{
  TNODE  *curt, *prev;
  NI_SOL *curni;
  LNODE  *curl;
  unsigned long i, j;

  /* create space for the new solution, sort it, and insert it */
  curni = new( NI_SOL );
  a_alloc( curni->orb, orbits, unsigned long );
  sort( Num, orbits );
  for ( i=0; i<orbits; i++ )
      curni->orb[i] = Num[i];

  /* now insert each of the values from the Num array into the tree */
  for ( i=0; i<orbits; i++ ) if ( Num[i] )
      {
      curt = *tptr;
      prev = 0;
      while ( curt && ( curt->value != Num[i] ))
        {
        prev = curt;
        curt = ( Num[i] < curt->value ? curt->left : curt->right );
        }

      if ( !curt )
         {
         curt = new( TNODE );
         curt->left = curt->right = curt->list = NULL;
         curt->cnt = 0;
         curt->value = Num[i];
         if ( ! prev )
            *tptr = curt;
         else
            if ( Num[i] < prev->value )
               prev->left = curt;
            else
               prev->right = curt;
         } /* a node with value did not exist */

      /* add the solution to the list */
      if ( !(curt->cnt) )
         {
         curl = new( LNODE );
         for ( j=0; j<LSIZE; curl->sol[j++] = NULL );
         curl->next = curt->list;
         curt->list = curl;
         curt->cnt = LSIZE;
         }
      curt-> list-> sol[ --(curt->cnt) ] = curni;

      ) /* end of for loop */

) /* end of insert() */
/*-------------------------------------------------------------------------
*
*       find()
*
*       This procedure finds the orbit in the tree and returns a
*       pointer into the linked list of pointers to ni-solutions.
*
*       Parameters:
*               tptr    : tree pointer
*               num     : the integer representing an orbit of a design
*               sol     : pointer to linked list of these cnt solutions
*/
find( tptr, num, sol )
  TNODE  *tptr;
```

```
  unsigned long num;
  LNODE  **sol;
{
  TNODE  *cur;

  /* first find the number in the tree */
  cur = tptr;
  while ( cur && ( cur->value != num ))
     cur = ( num < cur->value ? cur->left : cur->right );

  if ( !cur )
     *sol = NULL;
  else
     *sol = cur->list;

} /* end of find() */
/*-------------------------------------------------------------------------
*
*       same()
*
*       This function returns 1 if a design in the current list of
*       ni-solutions is isomorphic to the current one.  Otherwise, this
*       returns 0.
*
*       Parameters:
*              tptr    - tree pointer
*              Num     - array of integer orbit representations
*              orbits  - the number of orbits in the design
*/
int same( tptr, Num, orbits )
  TNODE *tptr;
  unsigned long Num[], orbits;
{
  LNODE *cur;
  unsigned long i, j;

  find( tptr, Num[0], &cur );
  sort( Num, orbits );

  while ( cur )
    {
    for ( i=0; i<LSIZE; i++ )
      if ( cur->sol[i] )
         {
         for ( j=0; (j<orbits) && (cur->sol[i]->orb[j] == Num[j]); j++ );
         if ( j == orbits ) return( 1 ); /* an isomorphic design found */
         } /* end of if */

    cur = cur->next;
    } /* end of while */

  return( 0 ); /* no designs are isomorphic to the current one */

} /* end of same() */
/*-------------------------------------------------------------------------
*
*       checkc()
*
*       This function returns non_zero if the cyclic design D is
*       isomorphic to one of the designs in tptr.
*
*       Parameters
*              tptr    : binary tree of integer representations of orbits
*              D       : has orbits for design under consideration
*              k       : number of distances constituting an orbit
*              v       : the sum of the distances in an orbit
*              orbits  : the number of orbits in a design
*/
checkc( tptr, D, k, v, orbits )
  PT    k, v, D[];
  TNODE *tptr;
  unsigned long orbits;
{
  int    i, j, m, orb, base=v-k, num, isom=0;

  m = -1;
  while ( (!isom) && Fact[++m] )
       {
       for ( orb=0; orb<orbits; orb++ )
         {
         /* generate the block */
         C = Blk[0] = 0;
         for ( i=1; i<k; i++ )
             {
             C = (((long) D[orb*k+i] * Fact[m]) + C ) % v;
             for ( j=i; j && (C < Blk[j-1]); j-- )
                 Blk[j] = Blk[j-1];
             Blk[j] = C;
             }

         /* calculate the orbit */
         for ( i=1; i<k; i++ )
             Blk2[i-1] = Blk[i-1] = Blk[i] - Blk[i-1];
         Blk2[k-1] = Blk[k-1] = v - Blk[k-1];

         /* find the minimum one */
         for ( i=1; i<k; i++ )
             if ( Blk2[i] < Blk[0] )
                for ( j=0; j<k; Blk[j] = Blk2[(j+i)%k], j++ );
             else if ( Blk2[i] == Blk[0] )
                {
                for ( j=1; (( j<k ) && (Blk[j] == Blk2[(j+i)%k] )); j++ );
                if (( j<k ) && (Blk[j] > Blk2[(j+i)%k]))
                   for ( j=0; j<k; Blk[j] = Blk2[(j+i)%k], j++ );
                }

         /* generate the integer corresponding to this orbit */
         num = 0;
         for ( i=1; i<k; i++ )
             num = num * base + Blk[i] - Blk[0];

         Num[orb] = num;

         ) /* end of for (orb<orbits) */

       isom = same( tptr, Num, orbits );
       ) /* end of while */

  if ( isom )
     (
```

```
    for ( num=Fact[m]; m > 0; m-- ) Fact[m] = Fact[m-1];
    Fact[0] = num;
    )

  return( isom );

} /* end of checkc() */
/*-------------------------------------------------------------------------
*
*       main()
*
*       This makes sure the proper number of arguments were entered,
*       calls pcmln(), and processing continues.
*/
main( argc, argv )
  int   argc    ;                       /*  Number of command agruments */
  char  *argv[] ;                       /*  Pointers to each argument   */
{
  char  solfn[FNL], nisofn[FNL], cyclic, status;
  PT    t,k,v,lambda, *Orb;
#ifdef TurboC
  int   sfh, nisfh;
#endif
  FILE  *sf, *nisf, *rf;
  unsigned long  cur_time, start_time, max_dumps, i, j,
                 sol=0, non_isom=0, isom, base;
  short orbits;
  TNODE  *root=0;

  get_time( &start_time );
  if ( argc < 2 )
     printf("Usage: %s mfn [-sr]\n", argv[0] ), exit(1);

  /* check for other parameters, and set up the file names */
  pcmln( argv, &status, solfn, nisofn );

  /* check for solution file and get header info */
#ifdef TurboC
  if (( sfh = open( solfn, O_RDONLY | O_BINARY )) == -1 )
#else
  if (( sf = fopen( solfn, "r" )) == NULL )
#endif
     fprintf( stderr, "ERROR - %s not found\n", solfn ), exit(1);
#ifdef TurboC
  sf = fdopen( sfh, "r" );
#endif
  fread( &t, sizeof(t), 1, sf );
  fread( &k, sizeof(k), 1, sf );
  fread( &v, sizeof(v), 1, sf );
  fread( &lambda, sizeof(lambda), 1, sf );
  fread( &cyclic, sizeof(cyclic), 1, sf );
  fread( &orbits, sizeof(orbits), 1, sf );
  fseek( sf, 4 * sizeof(t) + sizeof(cyclic), 0 );

  /* if they are testing for a non-cyclic design, quit */
  if ( ! cyclic )
     fprintf( stderr, "non-cyclic designs can not be tested for ",
                      "non-isomorphic solutions currently\n" ), exit(1);

  max_dumps = (MAX_FILE_SIZE - 25) / (k * orbits * sizeof(PT) + sizeof(orbits));

  /* allocate the space */
  a_alloc( Blk, k+1, PT );
  a_alloc( Blk2, k+1, PT );
  a_alloc( Orb, 2*orbits*k+1, PT );
  a_alloc( Fact, v+1, PT );
  a_alloc( Num, 2*orbits, unsigned long );
  /*  hopefully the maximum number of orbits will not exceed twice the
  *   number of orbits of the first solution
  */

  factors( v );
  base = v - k;

#ifdef TurboC
     nisfh = open( nisofn, O_WRONLY | O_BINARY | O_CREAT | O_TRUNC, S_IWRITE );
     nisf = fdopen( nisfh, "w" );
#else
     nisf = fopen( nisofn, "w" );
#endif
     fwrite( &t, sizeof(t), 1, nisf );
     fwrite( &k, sizeof(k), 1, nisf );
     fwrite( &v, sizeof(v), 1, nisf );
     fwrite( &lambda, sizeof(lambda), 1, nisf );
     fwrite( &cyclic, sizeof(cyclic), 1, nisf );


  /* start processing */
  while ( ( C = getc( sf ) ), !feof( sf ) )
    {
    ungetc( C, sf );

    fread( &orbits, sizeof(orbits), 1, sf );
    fread( Orb, orbits*k*sizeof(PT), 1, sf );
    if ( root )
       isom = checkc( root, Orb, k, v, orbits );
    else
       isom = 0;

    if ( !isom )
       {
       fwrite( &orbits, sizeof(orbits), 1, nisf );
       fwrite( Orb, orbits*k*sizeof(PT), 1, nisf );
       if ( !( ++non_isom % max_dumps ))
          {
          fclose( nisf );
          nth_file( nisofn, non_isom / max_dumps );
#ifdef TurboC
          nisfh = open( nisofn, O_WRONLY | O_BINARY | O_CREAT | O_TRUNC, S_IWRIT
          nisf = fdopen( nisfh, "w" );
#else
          nisf = fopen( nisofn, "w" );
#endif
          )

       /* calculate the integer representations of orbits */
       for ( i=0; i<orbits; i++ )
           {
           Num[i]=0;
           for ( j=1; j<k; j++ )
```

```
            Num[i] = Num[i] * base + Orb[i*k+j] - Orb[i*k];
         )

      /* now insert the new solution into the tree */
      insert( &root, Num, orbits );

      if ( status && ( !(non_isom % 20)))
         printf("%d non-isomorphic solutions found so far\n", non_isom );
      ) /* a non-isomorphic solution was found */

   if ( !( ++sol % max_dumps ))
      (
      fclose( sf );
      nth_file( solfn, sol / max_dumps );
#ifdef TurboC
      sfh = open( solfn, O_RDONLY | O_BINARY );
      sf = fdopen( sfh, "r" );
#else
      sf = fopen( solfn, "r" );
#endif
      )
   if ( status && ( !(sol % 100)))
      printf("%d distinct cyclic solutions processed so far\n", sol );

   ) /* end of while !feof */

  fclose( nisf );
  printf("\n%ld non-isomorphic solutions to S(%d,%d,%d)\n",
         non_isom, t, k, v );

  /* free the memory used */
  free( Blk );
  free( Blk2 );
  free( Orb );
  free( Fact );

  /* display total time used */
  get_time( &cur_time );
  printf("Total time consumed was ");
  print_time( cur_time-start_time );

) /* end of main */
/*-------------------------------------------------------------------------
*
*       Author:         Timothy C. Frenz (tcf2154)
*       Language:       C
*       Date Written:   30-Jan-1988
*       Last Modified:  16-Jun-1988
*       File Name:      matc.c
*       Computer:       AT&T 3B2/310
*       Op. System:     UNIX SYS 5 Rel 3.1
*       Purpose:
*               To generate the binary matrix from the t-orbits and k-orbits.
*
*               EX:     t-orb\k-orb  (1,1,5) (1,2,4) (1,4,2) (1,3,3) (2,2,3)
*                       (1,6)           2       1       1       1       0
*                       (2,5)           1       1       1       0       2
*                       (3,4)           0       1       1       2       1
*
*       Subroutines:
*               next_found()
*               occurences()
*               gen_matc()
*
*/

#define         lambda          1       /* Steiner system               */
#include        "globals.h"             /* Use my declaration file      */

#ifdef TurboC
#include        <fcntl.h>
#include        <sys\stat.h>
#endif

PT      *TEMP;

/*-------------------------------------------------------------------------
*
*       next_found
*
*       This function determines if the next consecutive positions in array
*       k_orbit starting at position pos can sum up to sum.  If so, return 1,
*       otherwise return 0.
*
*       Parameters:
*               sum     - the sum to try to achieve
*               pos     - the position to start looking at
*               k       - the number of distances in the k-orbit
*               k_orbit - the array of distances
*/
char next_found( sum, pos, k, k_orbit )
  PT    sum, *pos, k, k_orbit[];
{
  PT    i, val=sum;

  i = *pos;
  while ( val >= k_orbit[i] )
    (
    val -= k_orbit[i];
    i = (++i) % k;
    )

  *pos = i;
  return ( val == 0 );

) /* end of next_found() */

/*-------------------------------------------------------------------------
*
*       occurences
*
*       This function calculates the number of times the t-orbit
*       lies in the orbit of the k-orbit.
*
*       Parameters:
*               pos     - position to start looking in the t_orbit
*               t       - length of t_orbit
*               t_orbit - the array of t distances
*               k       - length of k_orbit
*               k_orbit - the array of k distances
```

```
*/
int occurences( pos, t, t_orbit, k, k_orbit )
  int   pos;
  PT    t, t_orbit[], k, k_orbit[];
{
  int   intersects=0;
  PT    i, t_pos, k_pos;

  for ( i=0; i<k; i++ )
    {
    t_pos = 0;
    k_pos = i;
    while (( t_pos < t ) &&
            next_found( t_orbit[t_pos+pos], &k_pos, k, k_orbit )) t_pos++;
    if ( t_pos == t ) intersects++;
    }

#ifdef DEBUG
    printf("\t");
    for ( i=0; i<t; i++ )
        printf("%4d", t_orbit[pos+i] );
    printf("   intersects %d times\n", intersects );
#endif
  return( intersects );

} /* end of occurences */

/*-------------------------------------------------------------------------
*
*       repeats
*
*       Parameters:
*               k       - length of k_orbit
*               k_orbit - the array of k distances
*/
PT repeats( k, k_orbit )
  PT    k, k_orbit[];
{
  PT    temp, i, j, same=1;

  for ( i=0; i<k; i++ )
      TEMP[i] = k_orbit[i];
  for ( i=1; i<k; i++ )         /* rotate k-1 times */
      {
      for ( temp=TEMP[0], j=1; j<k; j++ )
          TEMP[j-1] = TEMP[j];
      TEMP[k-1] = temp;
      for ( j=0; (j<k) && (TEMP[j] == k_orbit[j]); j++ );
      if ( j == k ) same++;
      }

  return( same );

} /* end of repeats() */
/*-------------------------------------------------------------------------
*
*       gen_matc
*
*       Returns the number of k-orbits actually used.  The ones deleted are
*       ones that had more than lambda occurences of a t-orbit within it.
*
*/
unsigned long gen_matc( t, k, t_orbs, k_orbs, tfn, kfn, nkfn, mf )
  PT    t, k;
  unsigned long t_orbs, k_orbs;
  char  tfn[], kfn[], nkfn[];
  FILE  *mf;
{
  PT    *t_orbits, *k_orbit, *IV;  /* the incidence vector */
  unsigned long tmp, New_k_orbs=0;
  FILE  *tf, *kf, *nkf;
  int   stop, i, bits, shifts, MAX_SHIFTS, cur_k, rep, tfh, kfh, nkfh;

  for ( i=2, bits=1; i<=lambda; i<<=1, bits++ );
  MAX_SHIFTS = NUMBITS / bits;

#ifdef TurboC
  if (( tfh = open( tfn, O_RDONLY | O_BINARY )) == -1 )
#else
  if ( (tf  = fopen( tfn,  "r" )) == NULL )
#endif
     printf("%s not found\n", tfn ), exit( 1 );

#ifdef TurboC
  if ((kfh  = open( kfn,  O_RDONLY | O_BINARY )) == -1 )
#else
  if ( (kf  = fopen( kfn,  "r" )) == NULL )
#endif
     printf("%s not found\n", kfn ), exit( 1 );

#ifdef TurboC
  tf = fdopen( tfh, "r" );
  kf = fdopen( kfh, "r" );
  nkfh = open( nkfn, O_WRONLY | O_BINARY | O_CREAT, S_IWRITE );
  nkf = fdopen( nkfh, "w" );
#else
  nkf = fopen( nkfn, "w" );
#endif

  a_alloc( t_orbits, t_orbs * t, PT );
  a_alloc( k_orbit, k, PT );
  a_alloc( TEMP, k, PT );
  a_alloc( IV, t_orbs, PT );

  fread( t_orbits, t_orbs * t * sizeof(PT), 1, tf );
  (void) fclose( tf );

  for ( cur_k=0; cur_k < k_orbs; cur_k++ )
    {
    fread( k_orbit, k * sizeof(PT), 1, kf );
    /*
    *  note that the lambda in the following for loop is currently set for
    *  1, since I am only considering steiner systems.  If this were changed
    *  to a variable, then other t-designs could be considered as well.
    *  Changes would be needed elsewhere also.
    */
    for ( i=0; i < t_orbs; IV[i++]=0 )    /* zero the array */

    rep = repeats( k, k_orbit );
#ifdef DEBUG
```

```
    for ( i=0; i<k; i++ )
        printf("%4d", k_orbit[i] );
    printf("   repeats %d times\n", rep );
#endif
    for ( stop=i=0; ((i < t_orbs) && (!stop)); i++ )
        stop = ( (IV[i]=occurences(i*t,t,t_orbits,k,k_orbit)/rep) > lambda );
#ifdef DEBUG
    if ( stop )
       printf("\tREJECTED\n");
    else
       printf("\tACCEPTED\n");
#endif
    if (!stop)
       {
       fwrite( k_orbit, k*sizeof(PT), 1, nkf );
       for ( i=0; i < t_orbs; )
         {
         for ( tmp=shifts=0; ((shifts<MAX_SHIFTS) && (i<t_orbs)); shifts++ )
           tmp = (tmp << 1) + IV[i++];
         fwrite( &tmp, sizeof(tmp), 1, mf );
         }
       New_k_orbs++;
       ) /* end of if !stop */
    ) /* end of for cur_k */

  (void) fclose( kf );
  (void) fclose( nkf );

  free( t_orbits );
  free( k_orbit );
  free( TEMP );
  free( IV );

  return( New_k_orbs );

) /* end of gen_matc() */
/*-------------------------------------------------------------------------
*
*       Author:         Timothy C. Frenz (tcf2154)
*       Language:       C
*       Date Written:   30-Jan-1988
*       Last Modified:  08-Dec-1988
*       File Name:      orbits.c
*       Computer:       AT&T 3B2/310
*       Op. System:     UNIX SYS 5 Rel 3.1
*       Purpose:
*               This section is going to find all of the orbits of X
*               distances in a total distance of Y.
*               Here are two examples:
*
*               EX 1:   X=2, Y=7
*                       1,6; 2,5; 3,4
*               Notice that the size of this is (Y choose X) divided by Y.
*               Thus in this example, (7 choose 2) = 21, 21 / 7 = 3.
*
*               EX 2:   X=3, Y=11
*                       1,1,9; 1,2,8; 1,8,2; 1,3,7; 1,7,3; 1,4,6; 1,6,4;
*                       1,5,5; 2,2,7; 2,3,6; 2,6,3; 2,4,5; 2,5,4; 3,3,5;
*                       3,4,4
*               (11 choose 3)/11 = ((11*10*9) / (3*2*1))/11 = 15.
*
*       Subroutines:
*               perm2()
*               perm_n()
*               perm()
*               gen_orb()
*               print_it_c()
*/

#include        "globals.h"             /* Use my declaration file      */

#ifdef TurboC
#include        <fcntl.h>
#include        <sys\stat.h>
#endif

/*      G L O B A L S
*/
unsigned long   NUM_ORBIT;
PT              *CUR_ORB;

/*-------------------------------------------------------------------------
*
*       print_it_c()
*
*       Parameters:
*               fd      - file descriptor to save the array to
*               a       - the array of distances
*               size    - the size of the array, t or k
*/
print_it_c( fd, a, size )
  FILE  *fd;
  PT    a[], size;
(
  PT    i, shifts=0, temp;
  char  works=1;

  /* initialize the data */
  for (i=0; i<size; i++)
      CUR_ORB[i] = a[i];

  /* shift the data, checking for less than */
  while (( ++shifts < size ) && works )
    {
    temp = CUR_ORB[0];
    for (i=1; i<size; i++)
        CUR_ORB[i-1] = CUR_ORB[i];
    CUR_ORB[size-1] = temp;
    if ( CUR_ORB[0] == a[0] )
       {
       for ( i=1; (i<size) && (a[i]==CUR_ORB[i]); i++ );
       if ( i<size )
          works = (a[i] <= CUR_ORB[i]);
       )
    ) /* end of while */

  if ( works )
     (
     fwrite( a, size * sizeof(PT), 1, fd );
     NUM_ORBIT++;
```

```
    }
) /* end of print_it_c() */

/*-------------------------------------------------------------------------
*
*       perm2
*
*       This procedure prints out all permutations of the array, permuting
*       only on two elements in the array.
*
*       Parameters:
*               l & h   - the two positions in the array to swap
*               size    - the size of the array
*               a       - the array of distances
*               fd      - the file descriptor
*/
perm2( l, h, size, a, fd )
  PT    l, h, a[], size;
  FILE  *fd;
(
  print_it_c( fd, a, size );

  if (( size - 2 ) && ( a[l] != a[h] ))
     (
     swap( a[l], a[h], PT );
     print_it_c( fd, a, size );
     swap( a[l], a[h], PT );
     )

) /* end of perm2() */

/*-------------------------------------------------------------------------
*
*       perm_n
*
*       Parameters:
*               a       - the array of distances to permute on
*               length  - the length of the array a
*               pos     - the current position in array a
*               fd      - the file descriptor
*/
perm_n( a, length, pos, fd )
  PT    a[], length, pos;
  FILE  *fd;
(
  PT    i;

  if (( length - pos ) == 2 )
     perm2( pos, pos+1, length, a, fd );
  else
     (
     perm_n( a, length, pos+1, fd );
     for ( i=pos+1; i<length; i++ )
         if (( a[pos] != a[i] ) && ( a[i] != a[i-1] ))
            (
            swap( a[pos], a[i], PT );
            perm_n( a, length, pos+1, fd );
            swap( a[pos], a[i], PT );
            )
     ) /* end of else */

) /* end of perm_n() */

/*-------------------------------------------------------------------------
*
*       perm
*
*       This procedure tries to perform all permutations on an array such
*       that none of the permuatations could be cyclicly revolved into
*       a previously made one.
*
*       Parameters:
*               size    - the size of array a
*               a       - the array of distances
*               fd      - the file descriptor
*/
perm( size, a, fd )
  PT    a[], size;
  FILE  *fd;
(
  PT    i, same, diff;

  for ( i=1; a[i] == a[0]; i++ );
  same = i;
  diff = size - same;
  switch (diff)
    (
    case 0  :
    case 1  : print_it_c( fd, a, size );
              break;
    case 2  : (
              PT  f=size-2, s=size-1, swaps=(same-1)/2;

              perm2( f, s, size, a, fd );
              for ( i=0; i<swaps; i++ )
                  (
                  swap( a[f], a[f-1], PT );
                  f--;
                  perm2( f, s, size, a, fd );
                  )
              if (!( same & 1 )) /* same is even */
                 (
                 swap( a[f], a[f-1], PT );
                 f--;
                 print_it_c( fd, a, size );
                 )
              swap( a[f], a[s-1], PT );
              break;
              )
    default : perm_n( a, size, 1, fd );

    ) /* end of switch */

) /* end of perm() */

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

```
*       This is the procedure called from main() to get all the cyclic
*       permutations of an array of size num, where the sum of the elements
*       in the array is range.
*
*       Parameters:
*               num     - the length of the array, t or k
*               v       - the sum of the distances in the array
*               fn      - the file name to store the results in
*/
unsigned long gen_orb( num, v, fn, ELIM )
  PT    num, v;
  char  fn[], ELIM;
{
  PT    *val, VAL=1, i, j, rot, last=1, sum, ELIM_VAL=v/num;
  FILE  *fd;
  int   fdh;
  char  found=0;

  a_alloc( CUR_ORB, num, PT );
  a_alloc( val, num, PT );

  val[0] = 0;
  NUM_ORBIT = 0;
#ifdef TurboC
  fdh = open( fn, O_WRONLY | O_BINARY | O_CREAT, S_IWRITE );
  fd  = fdopen( fdh, "w" );
#else
  fd = fopen( fn, "w" );
#endif

  while ( last )
    {
    last--;
    VAL = ++val[last];
    /*
    *  the following sets the array to all the same number from the
    *  position last to the second from the end, this end element is
    *  then assigned v minus the sum of the rest of the numbers.
    */
    for (i=last+1; i<num-1; val[i++] = VAL );
    for (i=0,sum=0; i<num-1; sum += val[i++] );
    val[num-1] = v - sum;

    if (( val[num-1] >= val[num-2] ) && ( val[num-1] <= v ))
       {
       if ( ELIM )
          for ( found=j=0; (!found) && (j<num); j++ )
              if ( val[j] == ELIM_VAL ) found = 1;
       if ( !found ) perm( num, val, fd );
       else
          {
          for ( j=0; (ELIM_VAL == val[j]) && (j<num); j++ );
          if ( j == num ) print_it_c( fd, val, num );
          }
       rot = (val[num-1] - val[num-2]) / 2 ;
       for ( i=0; i<rot; i++ )
         {
         val[num-1]--;
         val[num-2]++;
         if ( ELIM )
            for ( found=j=0; j<num; j++ )
                if ( val[j] == ELIM_VAL ) found = 1;
          if ( !found ) perm( num, val, fd );
          ) /* end of for */
        last = num - 2;
        ) /* if there are rotations at current level */

    ) /* while more possiblities */

  fclose( fd );
  free( CUR_ORB );
  free( val );

  return( NUM_ORBIT );

} /* end of gen_orb() */
/*-------------------------------------------------------------------------
*
*       Author:         Timothy C. Frenz (tcf2154)
*       Language:       C
*       Date Written:   15-Jul-1988
*       Last Modified:  05-Apr-1989
*       File Name:      res.c
*       Computer:       AT&T 3B2/310
*       Op. System:     UNIX SYS 5 Rel 3.1
*       Purpose:        This program prints out the results after running
*                       find or iso.
*
*       Subroutines:
*
*               main()          - The main procedure of this project
*               results()       - actually prints out the results
*/

#include        "globals.h"             /*  Use my declaration file     */
#ifdef TurboC
#include        <fcntl.h>
#include        <sys\stat.h>
#endif

/*-------------------------------------------------------------------------
*       results
*/
results( df, fn, k, parts )
  FILE  *df;
  char  fn[];
  PT    k;
  short parts;
{
  PT    *A;
  unsigned long sol=0, max_dumps;
  int   i, j, p_line, dfh;

  a_alloc( A, k, PT );

  p_line = 85/(k*3+2);
  if ( ! p_line ) p_line = 1;
  max_dumps = (MAX_FILE_SIZE - 25) /
              ( k * parts * sizeof( PT ) + sizeof( parts ) );
```

```
 while ( fread( A, k * sizeof(PT), 1, df ) )
    (
    printf("%5d  ", ++sol );
    for ( j=0; j<k; j++ )
        printf("%3d", A[j] );
    for (i=1; i<parts ; i++)
        (
        if ( (i%p_line) == 0 ) printf("\n     ");
        fread( A, k * sizeof(PT), 1, df );
        printf("  ");
        for ( j=0; j<k; j++ )
          printf("%3d", A[j] );
        ) /* end of for */

    printf("\n");

    if ( ! ( sol % max_dumps ))
       (
       fclose( df );
       nth_file( fn, sol/max_dumps );
#ifdef TurboC
       dfh = open( fn, O_RDONLY | O_BINARY );
       df = fdopen( dfh, "r" );
#else
       df = fopen( fn, "r" );
#endif
       )

    fread( &parts, sizeof(parts), 1, df );
    ) /* while more material */

  printf("\n%d ", sol );
  fclose( df );
  free( A );

} /* end of results() */

/*-------------------------------------------------------------------------
*
*       main()
*/
main( argc, argv )
  int   argc    ;                       /*  Number of command agruments */
  char  *argv[] ;                       /*  Pointers to each argument   */
(
  char  solfn[FNL], cyclic, isom=0;
  PT    t,k,v, lambda;
  unsigned long i;
  short parts;
  int   sfh;
  FILE  *sf;

  if ( argc != 2 )
     fprintf( stderr, "Usage: %s mfn \n", argv[0] ), exit(1);

  for ( i=0; argv[0][i]; i++ )
#ifndef TurboC
     if ( argv[0][i] == 'n' ) isom = 1;
#else
     if ( argv[0][i] == 'N' ) isom = 1;
#endif

  (void) strcpy( solfn, "saa" );
  if ( isom ) solfn[0] = 'n';
  (void) strcat( solfn, argv[1] );

#ifdef TurboC
  if (( sfh = open( solfn, O_RDONLY | O_BINARY )) == -1 )
#else
  if (( sf = fopen( solfn, "r" )) == NULL )
#endif
     fprintf( stderr, "ERROR - file %s not found\n", solfn ), exit(1);
#ifdef TurboC
  sf = fdopen( sfh, "r" );
#endif
  fread( &t, sizeof(t), 1, sf );
  fread( &k, sizeof(k), 1, sf );
  fread( &v, sizeof(v), 1, sf );
  fread( &lambda, sizeof(lambda), 1, sf );
  fread( &cyclic, sizeof(cyclic), 1, sf );
  fread( &parts, sizeof(parts), 1, sf );

  if ( cyclic )
     printf("\nSol#   Orbits\n");
  else
     printf("\nSol#   Blocks\n");

  results( sf, solfn, k, parts );

  if ( isom ) printf("non-isomorphic ");
  if ( cyclic ) printf("cyclic ");
  printf("solutions to S(%d,%d,%d)", t, k, v );
  if ( lambda == 1 ) printf("\n");
  else printf("-%d\n", lambda );

} /* end of main */
```

APPENDIX C

User's Manual

cyclic : Description

Syntax Summary: **cyclic** t k v [lambda]
where: t, k, and v are arguments to test

Description:

cyclic will calculate the number of blocks that would be in a design of the given parameters and then determine if such a design could be cyclic. If the design could be cyclic the program outputs the number of orbits that would be required to generate the steiner design. If the design could not be cyclic, there is a message which tells you so.

Option:

λ signifies that we are interested in looking at a t-design other than a steiner design. The default is 1, hence a steiner design.

gmatc : Description

Syntax Summary: **gmatc** t k v mfn [-s]
where: t, k, and v are numbers representing their usual meaning in a design
mfn is the matrix file name to output the results to

Description:

gmatc generates *t*-orbits and *k*-orbits and produces a matrix which is stored in mfn. The matrix represents the intersection of every *t*-orbit and every *k*-orbit. This file is stored with a header representing the parameters used: t, k, v, λ, cyclic design, and the size of the matrix by rows, and columns, and the filename storing the *k*-orbits so that these can ce retrieved when a solution is derived.

gmat generates *t*-subsets and *k*-subsets and produces a matrix which is stored in mfn. The major difference between **gmatc** and **gmat** is that the first deals with orbits and the second with blocks. Thus the size of the matrix produced by **gmat** is much greater given the same parameters t, k, and v. Actually the size will be the order of v^2 more. The header produced is the same except this is for a non-cyclic design, and the file name stored has *k*-subsets.

Option:

-s prints out status reports of what is happening and how long things are taking to perform.

find : Description

Syntax Summary: **find** mfn [-srn]
where: mfn is the same name used by **gmatc**

Description:

find reads in the paramters that are stored in the header of the matrix file. If the design is cyclic it searches through the matrix for a solution to a cyclic design. If the design is not cyclic, the search space is increased by an order of v in both dimensions. find outputs its results into a file(s) with a prefix of s## and a postfix of mfn. The #'s are replaced with the alphabetic characters aa, ab, ... az, ba, ... bz, ca, ... zz. Thus there can be up to 676 files used as the output from *find*. The first of these output files has a header containing 6 vital pieces of information: the values of t, k, v, and λ, whether the design is cyclic or not, and the number of orbits or blocks required in the design.

Option:

-s prints out status reports of what is happening and how long things are taking to perform.

-r restarts the program from when it last dumped out its location and relevant variables. This is useful if the program is termininated unintentionally.

-n this stops the program from actually storing the solutions found. This may be useful in searching for the number of solutions to some designs; however, it is then not possible to find the number of non-isomorphic solutions to the given design.

iso : Description

Syntax Summary: **iso** mfn [-sr]
where: mfn is the same name used by **gmatc**

Description:

iso begins by reading in the parameters stored in the file(s) produced by find and if the design is cyclic it starts searching through the list of solutions produced by find and calculates which of these solutions are isomorphic to each other and saves only one solution from each of these into a file named n## with a postfix of mfn. If the design is not cyclic, the program ends as I have not implemented that part of the program.

Option:

-s prints out status reports of what is happening and how long things are taking to perform.

-r restarts the program from when it last dumped out its location and relevant variables. This is useful if the program is termininated unintentionally.

poss : Description

Syntax Summary: poss t k [n]
where: t and k represent there usual meaning in a design
n is the maximum size of v to print out values for

Description:

poss calculates what the values of v should be modulo some number.

Option:

n calulates the actual values of v for you up to and including the value of n. These will be printed in ascending order starting with the first value greater than v.

res : Description

Syntax Summary: res mfn
where: mfn is the same name used by gmatc

Description:

res reads in the parameters stored in the s##mfn file and displays the blocks used in the design, or the orbits needed to generate the design, depending on whether the design is cyclic or not.

resn displays the orbits needed to generate the non-isomorphic cyclic designs.

show : Description

Syntax Summary: show fn n
where: fn is a filename
n is a number

Description:

show displays on stdout the contents of the file fn, n points to a line. This can be used to display the orbits generated by **gmatc** or the blocks generated by gmat by using the filenames of t.v or k.v where n would then be t or k.

Summary of Commands and Arguments		
cyclic	t k v [lambda]	tells whether the given parameters allow a cyclic design to exist
gmatc	t k v mfn [-s]	creates the matrix for a cyclic design of the given parameters
gmat	t k v mfn [-s]	creates the matrix for a non-cyclic design of the given parameters
find	mfn [-s] [-r] [-n]	searches the matrix for solutions to the design
iso	mfn [-s] [-r]	calculates which solutions are non-isomorphic
poss	t k [n]	lists the values of v for possible designs
res	mfn	displays solutions of find
resn	mfn	displays solutions of iso
show	fn n	displays the tuples in a t.v file

Table I

APPENDIX D

Orbit Listing

The following listing gives the orbits for all solutions listed in Table VII where the number of nc solutions is less than 100. Using these orbits one may construct the given Steiner system by generating all blocks for a given orbit. All of the blocks together form the Steiner system.

If there is a '+' following the S(t,k,v), this means that the orbit after this is a short orbit that is to be used in the construction of the design. For example S(2,3,15) has a short orbit of <5,5,5>. Thus there are 5 blocks associated with this short orbit, whereas full orbits generate v blocks.

S(2,3,7)

Solution	Orbits
1	1 2 4

S(3,4,10)

Solution	Orbits
1	1 1 4 4 1 2 1 6 2 2 3 3

S(4,5,11)

Solution	Orbits
1	1 1 1 2 6 1 1 4 3 2 1 2 1 3 4
	1 1 5 1 3 1 4 2 2 2 1 2 3 2 3

S(2,3,13)

Solution	Orbits
1	1 3 9 2 5 6

S(2,4,13)

Solution	Orbits
1	1 7 2 3

S(2,3,15) + 5 5 5

Solution	Orbits
1	1 3 11 2 6 7
2	1 3 11 2 7 6

S(3,5,17)

Solution	Orbits
1	1 1 4 7 4 1 2 6 6 2 1 3 3 1 9
	2 2 5 3 5

S(2,3,19)

Solution	Orbits
1	1 3 15 2 7 10 5 6 8
2	1 3 15 2 10 7 5 8 6
3	1 7 11 2 3 14 4 6 9
4	1 7 11 2 3 14 4 9 6

S(2,3,21) + 7 7 7

Solution	Orbits			
1	1 2 18	4 8 9	5 6 10	
2	1 2 18	4 8 9	5 10 6	
3	1 4 16	2 8 11	3 6 12	
4	1 4 16	2 8 11	3 12 6	
5	1 4 16	2 11 8	3 6 12	
6	1 8 12	2 3 16	4 6 11	
7	1 8 12	2 3 16	4 11 6	

S(2,5,21)

Solution	Orbits
1	1 5 2 10 3

S(2,3,25)

Solution	Orbits			
1	1 2 22	4 7 14	5 8 12	6 9 10
2	1 2 22	4 7 14	5 8 12	6 10 9
3	1 2 22	4 7 14	5 12 8	6 9 10
4	1 2 22	4 7 14	5 12 8	6 10 9
5	1 2 22	4 9 12	5 6 14	7 8 10
6	1 2 22	4 9 12	5 6 14	7 10 8
7	1 2 22	4 9 12	5 14 6	7 8 10
8	1 2 22	4 9 12	5 14 6	7 10 8
9	1 2 22	4 12 9	5 6 14	7 8 10
10	1 2 22	4 12 9	5 6 14	7 10 8
11	1 2 22	4 12 9	5 14 6	7 8 10
12	1 2 22	4 12 9	5 14 6	7 10 8

S(3,5,26)

Solution	Orbits		
1	1 1 3 18 3	1 2 6 4 13	1 13 4 6 2
	1 5 5 1 14	1 6 1 9 9	2 2 5 12 5
	2 4 7 5 8	2 8 5 7 4	2 9 3 3 9
	3 4 3 8 8		

S(2,3,27) + 9 9 9

Solution	Orbits			
1	1 2 24	4 7 16	5 10 12	6 8 13
2	1 2 24	4 7 16	5 10 12	6 13 8
3	1 2 24	4 7 16	5 12 10	6 8 13
4	1 2 24	4 7 16	5 12 10	6 13 8
5	1 2 24	4 16 7	5 10 12	6 8 13
6	1 2 24	4 16 7	5 10 12	6 13 8

7	1 2 24	4 16 7	5 12 10	6 8 13	
8	1 2 24	4 16 7	5 12 10	6 13 8	

S(2,3,31)

Solution	Orbits				
1	1 2 28	4 7 20	5 10 16	6 12 13	8 9 14
2	1 2 28	4 7 20	5 10 16	6 12 13	8 14 9
3	1 2 28	4 7 20	5 10 16	6 13 12	8 9 14
4	1 2 28	4 7 20	5 10 16	6 13 12	8 14 9
5	1 2 28	4 7 20	5 16 10	6 12 13	8 9 14
6	1 2 28	4 7 20	5 16 10	6 12 13	8 14 9
7	1 2 28	4 20 7	5 10 16	6 13 12	8 9 14
8	1 2 28	4 20 7	5 10 16	6 13 12	8 14 9
9	1 2 28	4 7 20	5 12 14	6 9 16	8 10 13
10	1 2 28	4 7 20	5 12 14	6 9 16	8 13 10
11	1 2 28	4 7 20	5 12 14	6 16 9	8 10 13
12	1 2 28	4 7 20	5 12 14	6 16 9	8 13 10
13	1 2 28	4 7 20	5 14 12	6 9 16	8 10 13
14	1 2 28	4 7 20	5 14 12	6 9 16	8 13 10
15	1 2 28	4 7 20	5 14 12	6 16 9	8 10 13
16	1 2 28	4 7 20	5 14 12	6 16 9	8 13 10
17	1 2 28	4 20 7	5 12 14	6 9 16	8 10 13
18	1 2 28	4 20 7	5 12 14	6 9 16	8 13 10
19	1 2 28	4 20 7	5 12 14	6 16 9	8 10 13
20	1 2 28	4 20 7	5 12 14	6 16 9	8 13 10
21	1 2 28	4 20 7	5 14 12	6 9 16	8 10 13
22	1 2 28	4 20 7	5 14 12	6 9 16	8 13 10
23	1 2 28	4 20 7	5 14 12	6 16 9	8 10 13
24	1 2 28	4 20 7	5 14 12	6 16 9	8 13 10
25	1 2 28	4 8 19	5 9 17	6 10 15	7 11 13
26	1 2 28	4 8 19	5 9 17	6 10 15	7 13 11
27	1 2 28	4 8 19	5 9 17	6 15 10	7 11 13
28	1 2 28	4 8 19	5 9 17	6 15 10	7 13 11
29	1 2 28	4 8 19	5 17 9	6 10 15	7 11 13
30	1 2 28	4 8 19	5 17 9	6 10 15	7 13 11
31	1 2 28	4 8 19	5 17 9	6 15 10	7 11 13
32	1 2 28	4 8 19	5 17 9	6 15 10	7 13 11
33	1 2 28	4 19 8	5 9 17	6 10 15	7 11 13
34	1 2 28	4 19 8	5 9 17	6 10 15	7 13 11
35	1 2 28	4 19 8	5 9 17	6 15 10	7 11 13
36	1 2 28	4 19 8	5 9 17	6 15 10	7 13 11
37	1 2 28	4 19 8	5 17 9	6 10 15	7 11 13
38	1 2 28	4 19 8	5 17 9	6 10 15	7 13 11
39	1 2 28	4 19 8	5 17 9	6 15 10	7 11 13
40	1 2 28	4 19 8	5 17 9	6 15 10	7 13 11
41	1 3 27	2 8 21	5 12 14	6 9 16	7 11 13

42	1 3 27	2 8 21	5 12 14	6 9 16	7 13 11
43	1 3 27	2 8 21	5 12 14	6 16 9	7 11 13
44	1 3 27	2 8 21	5 12 14	6 16 9	7 13 11
45	1 3 27	2 8 21	5 14 12	6 9 16	7 11 13
46	1 3 27	2 8 21	5 14 12	6 9 16	7 13 11
47	1 3 27	2 8 21	5 14 12	6 16 9	7 11 13
48	1 3 27	2 8 21	5 14 12	6 16 9	7 13 11
49	1 3 27	2 21 8	5 12 14	6 9 16	7 11 13
50	1 3 27	2 21 8	5 12 14	6 9 16	7 13 11
51	1 3 27	2 21 8	5 12 14	6 16 9	7 11 13
52	1 3 27	2 21 8	5 12 14	6 16 9	7 13 11
53	1 3 27	2 21 8	5 14 12	6 9 16	7 11 13
54	1 3 27	2 21 8	5 14 12	6 9 16	7 13 11
55	1 3 27	2 21 8	5 14 12	6 16 9	7 11 13
56	1 3 27	2 21 8	5 14 12	6 16 9	7 13 11
57	1 3 27	2 10 19	5 11 15	6 7 18	8 9 14
58	1 3 27	2 10 19	5 11 15	6 7 18	8 14 9
59	1 3 27	2 10 19	5 11 15	6 18 7	8 9 14
60	1 3 27	2 10 19	5 11 15	6 18 7	8 14 9
61	1 3 27	2 10 19	5 15 11	6 7 18	8 9 14
62	1 3 27	2 10 19	5 15 11	6 7 18	8 14 9
63	1 3 27	2 19 10	5 15 11	6 7 18	8 9 14
64	1 3 27	2 19 10	5 15 11	6 7 18	8 14 9
65	1 5 25	2 7 22	3 13 15	4 10 17	8 11 12
66	1 5 25	2 7 22	3 13 15	4 10 17	8 12 11
67	1 5 25	2 7 22	3 13 15	4 17 10	8 11 12
68	1 5 25	2 7 22	3 15 13	4 10 17	8 11 12
69	1 5 25	2 7 22	3 15 13	4 10 17	8 12 11
70	1 5 25	2 7 22	3 15 13	4 17 10	8 11 12
71	1 5 25	2 22 7	3 13 15	4 10 17	8 12 11
72	1 5 25	2 22 7	3 15 13	4 10 17	8 12 11
73	1 5 25	2 10 19	3 13 15	4 7 20	8 9 14
74	1 5 25	2 10 19	3 13 15	4 7 20	8 14 9
75	1 5 25	2 10 19	3 13 15	4 20 7	8 9 14
76	1 5 25	2 10 19	3 15 13	4 7 20	8 9 14
77	1 11 19	2 7 22	3 5 23	4 13 14	6 10 15
78	1 11 19	2 7 22	3 5 23	4 13 14	6 15 10
79	1 11 19	2 7 22	3 5 23	4 14 13	6 10 15
80	1 11 19	2 22 7	3 5 23	4 13 14	6 10 15

S(2,6,31)

Solution	Orbits
1	1 14 4 2 3 7

S(2,3,33) + 11 11 11

Solution	Orbits				
1	1 2 30	4 6 23	5 13 15	7 12 14	8 9 16
2	1 2 30	4 6 23	5 13 15	7 12 14	8 16 9
3	1 2 30	4 6 23	5 13 15	7 14 12	8 9 16
4	1 2 30	4 6 23	5 13 15	7 14 12	8 16 9
5	1 2 30	4 6 23	5 15 13	7 12 14	8 9 16
6	1 2 30	4 6 23	5 15 13	7 12 14	8 16 9
7	1 2 30	4 6 23	5 15 13	7 14 12	8 9 16
8	1 2 30	4 6 23	5 15 13	7 14 12	8 16 9
9	1 2 30	4 23 6	5 13 15	7 12 14	8 9 16
10	1 2 30	4 23 6	5 13 15	7 12 14	8 16 9
11	1 2 30	4 23 6	5 13 15	7 14 12	8 9 16
12	1 2 30	4 23 6	5 13 15	7 14 12	8 16 9
13	1 2 30	4 23 6	5 15 13	7 12 14	8 9 16
14	1 2 30	4 23 6	5 15 13	7 12 14	8 16 9
15	1 2 30	4 23 6	5 15 13	7 14 12	8 9 16
16	1 2 30	4 23 6	5 15 13	7 14 12	8 16 9
17	1 2 30	4 10 19	5 8 20	6 12 15	7 9 17
18	1 2 30	4 10 19	5 8 20	6 12 15	7 17 9
19	1 2 30	4 10 19	5 8 20	6 15 12	7 9 17
20	1 2 30	4 10 19	5 8 20	6 15 12	7 17 9
21	1 2 30	4 10 19	5 20 8	6 12 15	7 9 17
22	1 2 30	4 10 19	5 20 8	6 12 15	7 17 9
23	1 2 30	4 10 19	5 20 8	6 15 12	7 9 17
24	1 2 30	4 10 19	5 20 8	6 15 12	7 17 9
25	1 2 30	4 19 10	5 8 20	6 12 15	7 9 17
26	1 2 30	4 19 10	5 8 20	6 12 15	7 17 9
27	1 2 30	4 19 10	5 8 20	6 15 12	7 9 17
28	1 2 30	4 19 10	5 8 20	6 15 12	7 17 9
29	1 2 30	4 19 10	5 20 8	6 12 15	7 9 17
30	1 2 30	4 19 10	5 20 8	6 12 15	7 17 9
31	1 2 30	4 19 10	5 20 8	6 15 12	7 9 17
32	1 2 30	4 19 10	5 20 8	6 15 12	7 17 9
33	1 3 29	2 7 24	5 12 16	6 13 14	8 10 15
34	1 3 29	2 7 24	5 12 16	6 13 14	8 15 10
35	1 3 29	2 7 24	5 12 16	6 14 13	8 10 15
36	1 3 29	2 7 24	5 12 16	6 14 13	8 15 10
37	1 3 29	2 7 24	5 16 12	6 13 14	8 10 15
38	1 3 29	2 7 24	5 16 12	6 13 14	8 15 10
39	1 3 29	2 7 24	5 16 12	6 14 13	8 10 15
40	1 3 29	2 7 24	5 16 12	6 14 13	8 15 10
41	1 3 29	2 24 7	5 12 16	6 13 14	8 10 15
42	1 3 29	2 24 7	5 12 16	6 13 14	8 15 10
43	1 3 29	2 24 7	5 12 16	6 14 13	8 10 15
44	1 3 29	2 24 7	5 12 16	6 14 13	8 15 10
45	1 3 29	2 24 7	5 16 12	6 13 14	8 10 15
46	1 3 29	2 24 7	5 16 12	6 13 14	8 15 10

47	1 3 29	2 24 7	5 16 12	6 14 13	8 10 15
48	1 3 29	2 24 7	5 16 12	6 14 13	8 15 10
49	1 3 29	2 12 19	5 8 20	6 9 18	7 10 16
50	1 3 29	2 12 19	5 8 20	6 9 18	7 16 10
51	1 3 29	2 12 19	5 8 20	6 18 9	7 10 16
52	1 3 29	2 12 19	5 8 20	6 18 9	7 16 10
53	1 3 29	2 12 19	5 20 8	6 9 18	7 10 16
54	1 3 29	2 12 19	5 20 8	6 9 18	7 16 10
55	1 3 29	2 12 19	5 20 8	6 18 9	7 10 16
56	1 3 29	2 12 19	5 20 8	6 18 9	7 16 10
57	1 3 29	2 19 12	5 8 20	6 9 18	7 10 16
58	1 3 29	2 19 12	5 8 20	6 9 18	7 16 10
59	1 3 29	2 19 12	5 8 20	6 18 9	7 10 16
60	1 3 29	2 19 12	5 8 20	6 18 9	7 16 10
61	1 3 29	2 19 12	5 20 8	6 9 18	7 10 16
62	1 3 29	2 19 12	5 20 8	6 9 18	7 16 10
63	1 3 29	2 19 12	5 20 8	6 18 9	7 10 16
64	1 3 29	2 19 12	5 20 8	6 18 9	7 16 10
65	1 4 28	2 13 18	3 9 21	6 8 19	7 10 16
66	1 4 28	2 13 18	3 9 21	6 8 19	7 16 10
67	1 4 28	2 13 18	3 9 21	6 19 8	7 10 16
68	1 4 28	2 13 18	3 9 21	6 19 8	7 16 10
69	1 4 28	2 13 18	3 21 9	6 8 19	7 10 16
70	1 4 28	2 13 18	3 21 9	6 19 8	7 10 16
71	1 4 28	2 18 13	3 9 21	6 8 19	7 16 10
72	1 4 28	2 18 13	3 9 21	6 19 8	7 16 10
73	1 5 27	2 10 21	3 14 16	4 9 20	7 8 18
74	1 5 27	2 10 21	3 14 16	4 9 20	7 18 8
75	1 5 27	2 10 21	3 16 14	4 9 20	7 8 18
76	1 5 27	2 21 10	3 16 14	4 20 9	7 8 18
77	1 6 26	2 12 19	3 13 17	4 5 24	8 10 15
78	1 6 26	2 12 19	3 13 17	4 5 24	8 15 10
79	1 6 26	2 12 19	3 17 13	4 5 24	8 10 15
80	1 6 26	2 19 12	3 17 13	4 24 5	8 10 15
81	1 9 23	2 13 18	3 4 26	5 12 16	6 8 19
82	1 9 23	2 13 18	3 4 26	5 12 16	6 19 8
83	1 9 23	2 13 18	3 4 26	5 16 12	6 8 19
84	1 9 23	2 13 18	3 26 4	5 16 12	6 8 19

S(2,4,37)

Solution	Orbits			
1	1 2 21 13	4 22 6 5	7 12 8 10	
2	1 2 21 13	4 22 6 5	7 10 8 12	

S(2,4,40) + 10 10 10 10

Solution	Orbits		
1	1 6 2 31	3 18 5 14	4 12 13 11
2	1 6 2 31	3 18 5 14	4 11 13 12
3	1 6 2 31	3 14 5 18	4 12 13 11
4	1 6 2 31	3 14 5 18	4 11 13 12
5	1 3 9 27	2 5 17 16	6 15 11 8
6	1 3 9 27	2 5 17 16	6 8 11 15
7	1 3 9 27	2 16 17 5	6 8 11 15
8	1 3 9 27	2 16 7 15	5 21 8 6
9	1 3 9 27	2 16 7 15	5 6 8 21
10	1 3 9 27	2 15 7 16	5 21 8 6

S(2,5,41)

Solution	Orbits	
1	1 3 7 18 12	2 19 5 9 6

S(2,5,61)

Solution	Orbits		
1	1 2 18 34 6	4 8 11 14 24	5 17 13 16 10
2	1 2 18 34 6	4 24 14 11 8	5 17 13 16 10
3	1 27 21 10 2	4 5 14 22 16	6 29 8 7 11
4	1 27 21 10 2	4 5 14 22 16	6 11 7 8 29
5	1 27 21 10 2	4 16 22 14 5	6 29 8 7 11
6	1 27 21 10 2	4 16 22 14 5	6 11 7 8 29
7	1 42 2 5 11	3 10 21 4 23	6 14 8 24 9
8	1 42 2 5 11	3 23 4 21 10	6 14 8 24 9
9	1 22 2 27 9	3 28 12 4 14	5 8 7 35 6
10	1 22 2 27 9	3 14 4 12 28	5 8 7 35 6

S(2,5,65) + 13 13 13 13 13

Solution	Orbits		
1	1 2 28 14 20	4 8 38 9 6	5 17 7 25 11
2	1 2 28 14 20	4 6 9 38 8	5 11 25 7 17