An Algorithm for Enumerating Distinct Cyclic Steiner Systems

Timothy C. Frenz
School of Computer and Information Science
Center for Science and Technology
Syracuse University
Syracuse, NY 13244-4100
U.S.A.

Donald L. Kreher
Department of Mathematical Science
Michigan Technological University
Houghton, Michigan 49931
U.S.A.

Abstract. An algorithm is presented for finding all (0,1)-solutions to the matrix problem $AX = J$, where A is (0,1)-matrix and J is the all 1's column vector. It is applied to the problem of enumerating distinct cyclic Steiner systems and five new values are obtained. Specifically, the number of distinct solutions to S(2,3,55), S(2,3,57), S(2,3,61), S(2,3,63) and S(3,4,22), are 121,098,240, 84,672,512, 2,542,203,904, 1,782,918,144 and 1140 respectively.

1. Introduction

A *Steiner system* with parameters $S(t,k,v)$ is a pair (V,B) where V is a v-element set of *points*, and B is a collection of k-element subsets of V called *blocks*, such that each t-element subset of V appears in precisely one block of B. Here $1 \leq t \leq k \leq v$, where if any equality exists, the design is said to be *trivial*. An $S(t,k,v)$ design is said to be *cyclic* if $V = \{0,1,2,\ldots,v-1\}$ and whenever K is a block then $K+1 = \{x+1 : x \in K\}$ is also a block, addition performed modulo v. The *number of nonisomorphic* $S(t,k,v)$ *designs* is denoted by $nc(t,k,v)$. We are interested in computing $dc(t,k,v)$ the *number of distinct cyclic* $S(t,k,v)$ *designs*. The values of $dc(t,k,v)$ successfully computed by the algorithm described in this paper is reported in Table I. The values of $dc(2,3,55)$, $dc(2,3,57)$, $dc(2,3,61)$, $dc(2,3,63)$ and $dc(3,4,22)$ are new. It was erroneously reported in [5] that $nc(3,4,22) = 21$ and $dc(3,4,22) = 210$. There are in fact 114 nonisomorphic cyclic solutions and 1140 distinct cyclic solutions. They are given in [8]. The remaining values appear in **1, 2, 3, 4, 6, 9, 10.**

JCMCC 11 (1992), pp. 23-32

Table I

v	$S(t,k,v)$	No. of Orbits	$dc(t,k,v)$	$nc(t,k,v)$
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, 16 or 17	152	29
21	$S(2,3,21)$	4	32	7
	$S(2,5,21)$	1	2	1
22	$S(3,4,22)$	18 or 19	1,140	114
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

2. Solving $AX = J$

The problem of enumerating cyclic $S(t, k, v)$ designs can be reformulated as solving a matrix equation of the form

$$AX = J \tag{1}$$

where A is a special m by n (0,1)-matrix and J is the column vector of all 1's. An algorithm that enumerates all solutions to eqn. (1) with an arbitrary (0,1)-matrix A is given in Section 3. For cyclic $S(t, k, v)$ designs the matrix A has a special form.

Let $K = \{x_1, x_2, x_3, \ldots, x_k\}$ 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 $(d_1, d_2, \ldots, d_k)$. That is $\delta(K) = \{(d_1, d_2, \ldots, d_k), (d_2, \ldots, d_k, d_1), \ldots, (d_k, d_1, \ldots, d_{k-1})\}$. 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 $\langle d_1, d_2, \ldots, d_k \rangle$ the set of all k-element subsets K for which $(d_1, d_2, \ldots, d_k) \in \delta(K)$. Then $\langle d_1, d_2, \ldots, d_k \rangle$ is called the *cyclic orbit* containing K. It is easy to see that if (V, B) is a cyclic design and $K \in B$, then $\langle D \rangle \subseteq B$, for all $D \in \delta(K)$. Thus in particular a cyclic Steiner system is a disjoint union of cyclic orbits.

Let $\Delta_1, \Delta_2, \ldots, \Delta_{N_t}$, and $\Gamma_1, \Gamma_2, \ldots, \Gamma_{N_k}$ be complete lists of cyclic orbits of t and k element subsets of V respectively. Then the matrix $A_{t,k}$ is defined to be the N_t by N_k integer matrix whose $[i, j]$-entry is the number of k-element subsets in Γ_j containing a fixed representative of Δ_i. This matrix can easily be computed using the algorithm described in [7]. Note that for $S(t, k, v)$ designs an orbit Γ_j for which $A_{tk}[i, j] > 1$ for some i cannot be used. Thus these orbits and the corresponding columns of A_{tk} are deleted from consideration. The resulting column reduced matrix is an $m = N_t$ by $n \leq N_k$ matrix A and is the one we use in eqn. (1).

A solution to eqn. (1) is a collection of columns of A that have row sum 1. Consequently, we think of each column as a single entity and refer to it as an incidence vector. Let $I_1, I_2, \ldots, I_n$ be the incidence vectors representing the n columns of A. Then a solution to eqn. (1) is a (0,1)-vector $X = [x_1, x_2, \ldots, x_n]^T$ such that

$$\sum_{i=1}^{n} X[i] I_i = J. \tag{2}$$

Indeed if A is the reduced matrix obtained from A_{tk} and X is a solution, then $\cup\{\Gamma_j : X[j] = 1\}$ will be an $S(t, k, v)$ design.

3. The algorithm find

Our algorithm, which we have called **find**, for finding all solutions to eqn. (1) is given in figure 1. An example of its execution is given in Section 4. It was

implemented in C on different operating systems, primarily UNIX, using bit manipulation programming tricks to speed up computing times. The program uses a variety of arrays to find solutions to a Steiner system in what we believe is a very efficient way. The matrix A is stored as an array of integer vectors whose bits represent the incidence vectors. For an incidence vector I_i we define the *code* of I_i by:

$$\text{code}(I_i) = \sum_{h=1}^{m} I_i[h]2^h.$$

Using this coding function the incidence vectors can be arranged in descending order. This arrangement is maintained by the array *ptr*. Thus

$$\begin{aligned}\text{code}(I_{ptr[0]}) \geq \text{code}(I_{ptr[1]}) &\geq \text{code}(I_{ptr[2]}) \geq \cdots \\ &\geq \text{code}(I_{ptr[n-1]}).\end{aligned}$$

Using this *ptr* array the array *unique* is created which effectively removes all duplicate incidence vectors from the *ptr* array. The array *unique* contains the first subscript of each unique incidence vector in descending order provided by the *ptr* array. Thus, $I_{ptr\,[unique\,[i]]} \neq I_{ptr\,[unique\,[j]]}$ for all $i \neq j$, and $I_{ptr[i]} \neq I_{ptr\,[unique\,[j]]}$ for all i, $ptr[unique[j]] \leq i < ptr[unique[j+1]]$. Thus, when searching for a solution to equation (2), only the *unique* array is searched.

Duplicate columns arise often in the matrix A. In particular for Steiner triple systems every $\langle d_1, d_2, d_3 \rangle$ orbit creates the same incidence vector as $\langle d_1, d_3, d_2 \rangle$. In fact, every column of the A_{tk} matrix has a duplicate column for a cyclic $S(2,3,v)$, except for the short orbit, when $v > 9$ and $v \equiv 3 \pmod 6$. Removing these duplicate columns from the search space reduces the complexity of the search. It is easy to see that the number of distinct cyclic solutions to a $S(2,3,v)$ 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 the array *unique* when a solution is found that consists of a set of of incidence vectors, the other solutions whose incidence vectors are equivalent to the ones found are immediately saved instead of searching for them.

Two other arrays are then created in find which allow faster manipulation of the matrix. The array *bit_change* will contain the indices of the *unique* array where the highest bit in the incidence vectors changes. Thus, $i = unique\ [bit\_change\ [j]\ ]$ is the index i of the first incidence vector such that $I_i[m-j] = 1$ and $I_i[h] = 0$ for $m - j < h \leq m$. If the algorithm decides on using incidence vector i and $unique[bit\_change[j]] \leq i < unique[bit\_change[j+1]]$, then the sum of the incidence vectors choosen so far will have a one in positions $m-i, m-i+1, \ldots m$. Consequently, the next possible vector to consider is in a position at least $unique\ [\ bit\_change[i+1]]$. This avoids a costly linear search of the matrix. Also, an array *next* is created which points to the next possible incidence vector

Algorithm:**find**

Construct and initialize data structures:

ptr, unique, bit_change, bit_category, and next.

Set: *ans* to the empty stack;

current = 0;

$J = \sum_{i=0}^{m-1} 2^i$;

j = 0;

partial = 0;

solution_possible = **TRUE**;

and *solution_found* = **FALSE.**

while(*solution_possible*) do

while(NOT *solution_found*) AND (*solution_possible*) do

(1) while

(i) $j < bit\_change\ [current+1]$, and

(ii)*code* ($I_{ptr[unique[j]]}$) and *partial* $\neq 0$

do $j = j + 1$.

Thus $I_{ptr[unique[j]]}$ is the first incidence vector, if there is one, that is disjoint from the vectors chosen so far.

(2) If ($j \leq bit\_change[\ current+1]$, then

(a)push *j* onto the stack *ans* of incidence vectors used so far;

(b)update *partial* by the code of $I_{ptr[unique[j]]}$, i.e. *partial* = *partial* bitwise-OR *code* ($I_{ptr[unique[j]]}$); and

(c)if (*partial* = *J*)

(d)then set *solution_found* = **TRUE**

(e)otherwise set $j = next\ [j]$ and $current = bit\_category(j)$.

(3) else if (*ans* is not empty) then

(a) pop *j* from the stack *ans*;

(b) remove the code of $I_{ptr[unique[j]]}$ from *partial*, i.e. *partial* = *partial* bitwise-XOR *code* ($I_{ptr[unique[j]]}$); and

(c) set $current = bit\_category(j)$ and $j = j + 1$;

(4) else set *solutions_possible* = **FALSE.**

end while

(5) if (*solution_found*) then

(a) save all solutions found;

(b) set *solution_found* = **FALSE**; and

(c) perform (3.a, 3.b, and 3.c).

end while

Figure 1: Pseudocode for algorithm **find**

in A that could be used with the current incidence vector referenced by *unique*; *next*[*i*] = ∞ if there is no such incidence vector. The construction of these two arrays both require one pass through the *unique* array. We also make use of the function *bit_category*(j) which returns i if *bit_change*[i] $\leq j <$ *bit_change*[$i+1$] and returns 1 plus the number of entries in *bit_change* if there is no such i.

Once all of these arrays are created, the process of searching for a solution begins. Two data structures are used for this purpose: a stack *ans* to hold the indices of the unique array used in the solution (this stack can be thought of as the compressed version of the sparse array X in equation (1)); and, a variable *partial* to indicate the present partial solution of all the incidence vectors selected. Using the data structures defined above, the algorithm searches for the solution using an iterative implementation rather than a recursive one. This improves the overall performance of the algorithm. The algorithm is an exhaustive backtracking algorithm. It starts with the first incidence vector in *ptr*, then jumps to the next group defined by *bit_change*, or the incidence vector defined by *next*. Then this is applied until no incidence vector in a group defined by *bit_change* can be found with the current incidence vectors already used, at this point the last incidence vector added to the solution *ans* is removed, and then the search continues in the same *bit_change* group for another incidence vector. If no other incidence vector is found, the last incidence vector added is removed again. This search process exhaustively finds all distinct solutions to equation (2). Although it may seem like wasting a lot of memory to have all of the arrays that were defined above, the improvement in time is very significant.

4. Example

The following is an example of algorithm **find** with S(2,3,15). The cyclic orbits of 2-element subsets are

$$\langle 1,14\rangle, \langle 2,13\rangle, \langle 3,12\rangle, \langle 4,11\rangle, \langle 5,10\rangle, \langle 6,9\rangle \text{ and } \langle 7,8\rangle.$$

and the cyclic orbits of 3-element subsets are

$\langle 1,1,13\rangle, \langle 1,2,12\rangle, \langle 1,12,2\rangle, \langle 1,3,11\rangle, \langle 1,11,3\rangle, \langle 1,4,10\rangle, \langle 1,10,4\rangle, \langle 1,5,9\rangle$
$\langle 1,9,5\rangle, \langle 1,6,8\rangle, \langle 1,8,6\rangle, \langle 1,7,7\rangle, \langle 2,2,11\rangle, \langle 2,3,10\rangle, \langle 2,10,3\rangle, \langle 2,4,9\rangle$
$\langle 2,9,4\rangle, \langle 2,5,8\rangle, \langle 2,8,5\rangle, \langle 2,6,7\rangle, \langle 2,7,6\rangle, \langle 3,3,9\rangle, \langle 3,4,8\rangle, \langle 3,8,4\rangle,$
$\langle 3,5,7\rangle, \langle 3,7,5\rangle, \langle 3,6,6\rangle, \langle 4,4,7\rangle, \langle 4,5,6\rangle, \langle 4,6,5\rangle$and$\langle 5,5,5\rangle$.

Here is the $A_{2,3}$ matrix with these orbits as labels (note that blanks denote 0).

	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	4	4	4	5
	1	2	12	3	11	4	10	5	9	6	8	7	2	3	10	4	9	5	8	6	7	3	4	8	5	7	6	4	5	6	5
	13	12	2	11	3	10	4	9	5	8	6	7	11	10	3	9	4	8	5	7	6	9	8	4	7	5	6	7	6	5	5
⟨1,14⟩	2	1	1	1	1	1	1	1	1	1	1	1																		
⟨2,13⟩	1	1	1										2	1	1	1	1	1	1	1	1									
⟨3,12⟩		1	1	1	1									1	1						2	1	1	1	1	1				
⟨4,11⟩				1	1	1	1						1			1	1					1	1				2	1	1	
⟨5,10⟩						1	1	1	1					1	1			1	1					1	1			1	1	1
⟨6,9⟩								1	1	1	1					1	1			1	1	1				2		1	1	
⟨7,8⟩										1	1	2						1	1	1	1		1	1	1	1	1			

Now any column having any value other than 0 or 1 is discarded.
Thus, the matrix A that is used to search for a solution is:

	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	3	3	3	3	4	4	5
	2	12	3	11	4	10	5	9	6	8	3	10	4	9	5	8	6	7	4	8	5	7	5	6	5
	12	2	11	3	10	4	9	5	8	6	10	3	9	4	8	5	7	6	8	4	7	5	6	5	5
⟨1,14⟩	1	1	1	1	1	1	1	1	1	1															
⟨2,13⟩	1	1									1	1	1	1	1	1	1								
⟨3,12⟩	1	1	1	1							1	1						1	1	1	1				
⟨4,11⟩			1	1	1	1							1	1				1	1			1	1		
⟨5,10⟩					1	1	1	1			1	1			1	1				1	1	1	1	1	
⟨6,9⟩							1	1	1	1			1	1			1	1					1	1	
⟨7,8⟩									1	1					1	1	1	1	1	1	1				

The matrix is then sorted by the codes of the columns, and the corresponding data structures created. The matrix in this example does not need sorting, but here it is again with the data structures.

indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	4	4	5
	2	12	3	11	4	10	5	9	6	8	3	10	4	9	5	8	6	7	4	8	5	7	5	6	5
	12	2	11	3	10	4	9	5	8	6	10	3	9	4	8	5	7	6	8	4	7	5	6	5	5
⟨1,14⟩	1	1	1	1	1	1	1	1	1	1															
⟨2,13⟩	1	1									1	1	1	1	1	1	1	1							
⟨3,12⟩	1	1	1	1							1	1							1	1	1	1			
⟨4,11⟩			1	1	1	1							1	1					1	1			1	1	
⟨5,10⟩					1	1	1	1			1	1			1	1					1	1	1	1	1
⟨6,9⟩							1	1	1	1			1	1			1	1					1	1	
⟨7,8⟩									1	1					1	1	1	1	1	1	1	1			
code	112	112	88	88	76	76	70	70	67	67	52	52	42	42	37	37	35	35	25	25	21	21	14	14	4
ptr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
unique	0	2	4	6	8	10	12	14	16	18	20	22	24												
bit_category	0	0	0	0	0	1	1	1	1	2	2	3	4	5											
bit_change	0	5	9	11	12	13																			
next	11	7	8	9	5	∞	10	∞	12	12	∞	∞	∞												

The following is a simple example of the algorithm flow. The numbers in parentheses (x) correspond to the statement numbers in the Algorithm **find** presented in this paper. For this example $J = 127$.

	j	*current*	*ans*	*partial*	*solution_possible*	*solution_found*
	0	0	[]	0 = 0000000	TRUE	FALSE
(1)	0					
(2)	11	3	[0]	114 = 1110000		
(1)	11					
(2)	∞	4	[0,11]	126 = 1111110		
(1)	∞					
(3)	12	3	[0]	112 = 1110000		
(1)	3					
(3)	1	0	[]	0 = 000000		
(1)	1					
(2)	7	1	[1]	88 = 1011000		
(1)	7					
(2)	∞	4	[1]	88 = 1011000		
(1)	∞					
(2)	12	4	[1,8]	118 = 1111011		
(1)	12					
(2)			[1,8,12]	127 = 1111111		TRUE
(5)	13	4	[1,8]	118 = 1111011		FALSE
(1)	13					
(3)	9	1	[1]	88 = 1011000		
(1)	9					
(3)	2	0	[]	0 = 0000000		
(1)	2					
(2)	8	0	[2]	76 = 1001100		
(1)	8					
(2)	12	0	[8]	110 = 1101111		
(1)	13					
(3)	9	2	[2]	76 = 1001100		
(1)	9					
(3)	3	0	[]	0 = 0000000		
(1)	3					
(2)	9	2	[3]	70 = 1000110		
(1)	9					
(2)	12	4	[3,9]	95 = 1011111		
(1)	13					
(3)	10	2	[3]	70 = 1000110		
(1)	11					

	j	*current*	*ans*	*partial*	*solution_possible*	*solution_found*
(3)	4	0	[]	0 = 0000000		
(1)	4					
(2)	5	1	[4]	67 = 1000011		
(1)	5					
(2)	∞	5	[4,5]	118 = 1110111		
(1)	∞					
(3)	6	1	[4]	67 = 1000011		
(1)	9					
(3)	5	0	[]	0 = 0000000		
(1)	5					
(3)			Stack	Empty		
(4)					FALSE	

The algorithm finds one solution among the unique incidence vectors. This solution is:

$$I_{ptr[unique[1]]} = [1,0,1,1,0,0,0],$$
$$I_{ptr[unique[8]]} = [0,1,0,0,0,1,1] \text{ and}$$
$$I_{ptr[unique[12]]} = [0,0,0,0,1,0,0].$$

The orbits that have incidence vector $[1,0,1,1,0,0,0]$ are $\langle 1,3,11\rangle$ and $\langle 1,11,3\rangle$. The orbits that have incidence vector $[0,1,0,0,0,1,1]$ are $\langle 2,6,7\rangle$ and $\langle 2,7,6\rangle$. The only orbit that has incidence vector $[0,0,0,0,1,0,0]$ is $\langle 5,5,5\rangle$. Thus, the one solution found among the unique vectors yields the four solutions below:

(1)	$\langle 1,3,11\rangle$,	$\langle 2,6,7\rangle$,	$\langle 5,5,5\rangle$
(2)	$\langle 1,3,11\rangle$,	$\langle 2,7,6\rangle$,	$\langle 5,5,5\rangle$
(3)	$\langle 1,11,3\rangle$,	$\langle 2,6,7\rangle$,	$\langle 5,5,5\rangle$
(4)	$\langle 1,11,3\rangle$,	$\langle 2,7,6\rangle$,	$\langle 5,5,5\rangle$

5. Concluding Remarks

Using the algorithm discussed in this paper, research on other cyclic Steiner systems beyond the range of Table I and on cyclic t–designs with $\lambda > 1$ should be possible. Changing most of the necessary modules in the code to find t–designs should be fairly straightforward. However, another way of comparing incidence vectors that does not require the use of bit operations must be derived, or a different method written to check incidence vectors. 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. A modification to the algorithm would be to change the implementation of some code to try to speed up the run time, adding some heuristics in a few places.

Acknowledgements

This research was conducted at GE Research and Development Center, University of Wyoming, Rochester Institute of Technology and Houghton College. An earlier version of this project was also presented at the Fourth Ontario Combinatorics Workshop held at Queen's University in Kingston, Ontario. The resulting discussions were very helpful. In particular we would like to thank C. J. Colbourn, R. Mathon and A. Rosa for their useful comments. D.L. Kreher was partially supported by NSF Grant CCR-8909894.

References

1. J.C. Cho, *On cyclic Steiner quadruple systems*, Ars Combinatoria **10** (1980), 123–130.
2. C.J. Colbourn, M.J., Colbourn and K.T. Phelps, *Combinatorial algorithms for generating cyclic Steiner quadruple systems*, Univ. New Brunswick, Fredericton, N.B. (1980).
3. C.J. Colbourn and K.T. Phelps, *Three new Steiner quadruple systems*, Utilitas Math. **18** (1980), 35–40.
4. C.J. Colbourn, *Distinct Cyclic Steiner Triple Systems*, Utilitas Mathematica **22** (1982), 103–126.
5. I. Diener, *On Cyclic Stiener Systems* $S(3,4,22)$, Annals of Discrete Mathematics **7** (1980), 301–313.
6. J. Doyen, and A. Rosa, *An updated bibliography of Steiner systems*, Annals of Discrete Math. **7** (1980), 317–349. This was updated again in 1989.
7. D.L. Kreher and S.P. Radziszowski, *Constructing* $6-(14,7,6)$ *designs*, Contemporary Mathematics **111** (1990), 137–151.
8. T.C. Frenz and D.L. Kreher, *The 1140 distinct cyclic* $SQS(22)$*'s.* (technical report).
9. C.C. Lindner and A. Rosa, *Steiner quadruple systems—a survey*, Discrete Math. **22** (1978 no. 2), 147–181.
10. K.T. Phelps, *On cyclic Steiner Systems* $S(3,4,20)$, Annals of Discrete Mathematics **7** (1980), 277–300.