Distributed Isomorph-free Exhaustive Generation of Anti-Pasch Partial Triple Systems

Lucia Moura [*] and Sebastian Raaphorst [†]
School of Information Technology and Engineering
University of Ottawa
e-mail: lucia@site.uottawa.ca, raaphors@site.uottawa.ca

Abstract

Anti-Pasch partial Steiner triple systems (anti-Pasch PSTSs) arise in erasure codes, extremal set systems and combinatorial design theory. Maximal anti-Pasch PSTSs correspond to erasure-resilient codes that are used for handling failures in large disk arrays. These codes support failure of any set of 3 disks and most sets of 4 disks while having the smallest possible update penalty and check-disk overhead. In this article, we apply a general algorithm for isomorph-free exhaustive generation of incidence structures [4, 5] to the specific case of anti-Pasch PSTSs. We develop and implement a distributed version of the algorithm, which is experimentally analysed. Using this implementation, we obtain a complete, isomorph-free catalogue of the maximal anti-Pasch PSTSs of order v, for $v \leq 15$. The enumeration and classification results for $10 \leq v \leq 14$ are new and computationally nontrivial.

1 Introduction

A Steiner triple system (STS) of order v, denoted by $STS(v)$, is a pair $(V, \mathcal{B})$ where V is a v-set and $\mathcal{B}$ is a collection of unordered triples (called blocks) of V, such that for every unordered pair in V, there exists exactly

[*]Supported by an NSERC research grant.
[†]Supported by an NSERC Undergraduate Research Award.

one block in $\mathcal{B}$ that contains it. A partial Steiner triple system (PSTS) of order v, denoted by $PSTS(v)$, is a pair $(V, \mathcal{B})$ with similar properties, in which "exactly one" in the previous definition is substituted by "at most one".

The study of triple systems is an active area of research (see [1]). Avoidance problems in triple systems have been largely studied [6]. One famous problem is Pasch avoidance. A Pasch configuration on an STS (PSTS) is a configuration consisting of 4 triples on the STS (PSTS) spanning 6 points, which can only be of the form: $\{\{a,b,c\},\{a,d,e\},\{f,b,d\},\{f,c,e\}\}$. An STS or PSTS is Pasch-free if it does not contain 4 triples forming a Pasch. The anti-Pasch packing number, denoted here by $D(v)$, is the maximum number of blocks on an anti-Pasch $PSTS(v)$; the anti-Pasch $PSTS(v)$ containing exactly $D(v)$ blocks are said to be *maximal*. When an anti-Pasch $STS(v)$ exists, then a maximal anti-Pasch $PSTS(v)$ must be a Steiner triple system.

Chee et al. [2] have shown that maximal anti-Pasch PSTSs give optimal erasure-resilient codes that tolerate all 3-erasures and most 4-erasures. These codes are used for handling failures in large disk arrays (RAIDs) and have smallest possible update penalty and check-disk overhead among the codes tolerating the same level of failure. The only 4-erasures that cannot be corrected are the unavoidable ones coming from catastrophic failures, i.e. an erasure of a disk plus all of its check disks. This application has been introduced by Hellerstein et al. [8]. The parity check matrix for the code is given by $[A|I]$ where A is the point-by-block incidence matrix of the anti-Pasch PSTSs.

In this article, we classify the anti-Pasch $PSTS(v)$, for $v \leq 15$, by a computational exhaustive generation of the non-trivial cases, namely $v \in [10, 14]$. A catalogue containing all such pairwise non-isomorphic designs, as well as their automorphism group sizes and leave graphs, is provided.

The algorithm employed in the search is an orderly algorithm based on the methods by Denny and Gibbons [4]. The algorithm also falls into the general framework described by McKay [10]: it is an isomorph-free exhaustive generation that guarantees to generate each (pairwise nonisomorph) substructure only once, and it is fully parallelizable. We implement a distributed version of the algorithm, which dynamically assigns the unfinished subtasks whenever a workstation is free. This implementation uses a client-server approach, in which a server generates all small substructures (partially filled incidence matrices) which are independently completed by various client workstations. Such distributed method raises some computational questions that are experimentally investigated, such as: how big should the substructures generated by the server be, how is the running time affected by the addition of more workstations (clients), among others. We believe that our experimental findings can be of value to researchers applying similar distributed exhaustive generation techniques to other combinatorial problems.

2 Previous existence and enumeration results

We now summarize previously known results on the existence and enumeration of STSs and PSTSs, for both general and anti-Pasch cases. An $STS(v)$ exists if and only if $v \equiv 1, 3 \pmod 6$; for these values an optimal $PSTS(v)$ must be an $STS(v)$. For all v, the size of the largest $PSTS(v)$ has been determined, and various constructions are known. An anti-Pasch $STS(v)$ is known to exist for every admissible parameter except for $v = 7, 13$, that is, an anti-Pasch $STS(v)$ exists if and only if $v \equiv 1, 3 \pmod 6$ and $v \neq 7, 13$ (see references in [1] plus the later result by Grannell et al. [7]). This implies that $D(v) = (v^2 - v)/6$ for all $v \equiv 1, 3 \pmod 6$, $v \neq 7, 13$. We observe that the known construction of optimal PSTSs for $v = 0, 2 \pmod 6$ from the deletion of a point on a $PSTS(v + 1)$ can be directly applied to show that $D(v) = (v^2 - 2v)/6$ for all $v \equiv 0, 2 \pmod 6$, $v \neq 6, 12$. As far as we know, $D(v)$ has not been generally determined for $v \equiv 4, 5 \pmod 6$. In [11], the value of $D(v)$ is computationally obtained for $v \in \{6, 7, 10, 11, 12, 13\}$, which completes the determination of $D(v)$ for $v \leq 15$.

Enumeration results of pairwise non-isomorphic STSs for all admissible $v \leq 19$ are known and shown in the next table. The entries for $v = 19$ have been recently completed by Kaski and Ostergard [9]. However, much less is known on the enumeration of anti-Pasch PSTSs. Indeed, the only previously known results seems to be the ones implied by the existence of anti-Pasch STSs.

v	7	9	13	15	19
# $STS(v)$	1	1	2	80	11,084,874,829
# anti-Pasch $STS(v)$	0	1	0	1	2,538

3 Outline of the algorithm

The algorithm employed by us is based on the general method proposed by Denny and Gibbons [4] (see more details in [3]). Their method consists of various improvements on an algorithm first proposed by Gibbons et al. [5].

The algorithm builds the incidence matrix (points by blocks) for each triple system generated. The main strategy is a 2-level backtracking algorithm for constructing the incidence matrix: the first level of the backtracking consists of the generation of the matrix, row by row, and the second level is a backtracking on the possible values of the columns for each row. It has been experimentally shown (see Denny [3]) that it is much more efficient to do the backtracking on points (rows) rather than blocks of the designs.

The columns of the matrix (corresponding to the blocks of the design) are required to be lexicographically ordered. In this way, identical columns of a partially generated matrix can be grouped into `cells`, and new points are added to the leftmost columns in a cell, avoiding the generation of some isomorphic configurations. See [3] for more details on the row cell structure.

Exactly one representative of each isomorphism class of designs of any given order must be generated, this is called a *canonical representative.* The canonical representative selected is the lexicographically smallest design among the members of an isomorphism class (the ordering of designs is based on a special lexicographical ordering of matrices described in Section 3.1). An orderly generation is employed, imposing that the rows of the matrix be sorted in lexicographical order as well. In this way, partial matrices of a canonical representative must be canonical partial designs; therefore, we can backtrack on any row corresponding to a partial matrix that is not in canonical form. We explain the canonicity test algorithm in Section 3.3.

Another important aspect of the algorithm is the checking that the partial matrices satisfy several properties, so that they correspond to partially constructed anti-Pasch PSTSs. Partial matrices satisfying these properties are called *feasible.* These properties are described in Section 3.1 and 3.2.

A rough general description of the algorithm is given next:

```
Input:    $v$; $ub$, $lb$ (upper and lower bounds on $D(v)$)
Output:   isomorph-free list of $PSTS(v)$ with $D(v)$ blocks

Let $M$ be a $v$ by $ub$ matrix, initially full of zeroes.
cr = 1; (stores the current row)
designList = empty-list;
$b_{max} = 0$; (largest # blocks in a design generated so far)
while ($cr > 0$) do
    result = generate-next-row($M$,$cr$);   (row backtracking)
```

```
   if (result is not OK) then    (there is no next row)
      $cr = cr - 1$; (backtrack on rows)
   elseif ($M$ is feasible) [and (M is canonical)] then (*)
         $cr = cr + 1$; (prepare for generation of next row)
   else do nothing ( i.e.  try next possible row content)
   endif;
   if ($cr = v + 1$) then
      Let $b$ be #blocks in the design given by $M$.
      if ($b \geq b_{max}$) and ($b \geq lb$) then
         if ($M$ is canonical) then (**)
            if ($b > b_{max}$) then
               $b_{max}$ = $b$;  (update $b_{max}$)
               designList = empty-list;
            endif;
            add $M$ to designList;
            if ($b > lb$) lb = b;    (update lb)
         endif;
      endif;
      $cr = v$;
   endif;
endwhile;
output designList;
```

The procedure generate-next-row is a second level backtracking algorithm on rows. Figure 1 shows, for row number 4, the feasible rows in the order they are generated by this procedure.

Remark: If we are interested in avoiding other configurations, the only required change in the above algorithm is in the feasibility test for M.

3.1 Restrictions used in row generation

Let v be the order of the designs sought and ub be an upper bound on its number of blocks. Since we don't know the number of columns on the largest incidence matrix (corresponding to the maximal designs), we take M to be a 0-1 matrix with v rows and ub columns such that the sum of the elements in a column are equal to either 3 or 0 (the column either represents a triple or is not used). Given two rows there exists at most one column for which both rows contain number 1 (partial PSTS property), and no 6×4 submatrix of M contains exactly two ones per column (anti-Pasch property).

Let r_i be the number of 1s in row i of the incidence matrix of a design. Knowledge on the structure of the designs sought allows for the determination of lower and upper bounds on $D(v)$ and on r_i, which adds extra restrictions to feasible rows.

Theorem 1. *Let $v \neq 6, 7, 12, 13$. Then, the following are valid lower and upper bounds on $b = D(v)$ and r_i:*

if $v \equiv$	b_{lower}	b_{upper}	r_{lower}	r_{upper}
0 (*mod* 6)	$\frac{v^2-2v}{6}$	$\frac{v^2-2v}{6}$	$\frac{v-2}{2}$	$\frac{v-2}{2}$
1 (*mod* 6)	$\frac{v^2-v}{6}$	$\frac{v^2-v}{6}$	$\frac{v-1}{2}$	$\frac{v-1}{2}$
2 (*mod* 6)	$\frac{v^2-2v}{6}$	$\frac{v^2-2v}{6}$	$\frac{v-2}{2}$	$\frac{v-2}{2}$
3 (*mod* 6)	$\frac{v^2-v}{6}$	$\frac{v^2-v}{6}$	$\frac{v-1}{2}$	$\frac{v-1}{2}$
4 (*mod* 6)	$\frac{(v-1)(v-2)}{6}$	$\frac{v^2-2v-2}{6}$	1	$\frac{v-2}{2}$
5 (*mod* 6)	$\frac{(v-2)(v-3)}{6}$	$\frac{v^2-v-8}{6}$	1	$\frac{v-1}{2}$

Proof. For $v \equiv 0, 1, 2, 3 \pmod 6$, these results follow from the existence of maximal $PSTS(v)$ that are anti-Pasch, for $v \neq 6, 7, 12, 13$. For $v \equiv 4, 5 \pmod 6$, the upper bounds on b and r come from the size of a maximal $PSTS(v)$; the lower bound on v comes from the existence of an anti-Pasch $STS(v-1)$ and an anti-Pasch $STS(v-2)$, respectively. □

Let C_i be the set of columns j of M such that $M_{i,j} = 1$. We employ the following modified lexicographical ordering on the rows of matrix M: we say that $C_{i_1} <_L C_{i_2}$ if and only if $|C_{i_1}| > |C_{i_2}|$ or ($|C_{i_1}| = |C_{i_2}|$ and C_{i_1} is lexicographical smaller than C_{i_2}). This is a modification of the usual lexicographical ordering, such that rows with larger numbers of 1's precede the others. This is chosen in order to give priority to rows that are more likely to give large b, since large b arises from large r_i's; this heuristics attempts to increase the lower bound on b earlier on the generation. This lexicographical ordering imposes restrictions on the contents of the next row to be generated.

3.2 Feasibility testing

The following proposition gives a useful pruning condition which was employed by our algorithm.

Proposition 1. *Let M be a v by b incidence matrix of an anti-Pasch $PSTS(v)$, with rows satisfying the above mentioned modified lexicographical*

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	1	1	1	0	0	0	0	0	0	0	0
2	1	0	0	0	1	1	1	0	0	0	0	0
3	1	0	0	0	0	0	0	1	1	1	0	0

Column (cell) backtracking for row 4:

4	1											

infeasible: more than 3 ones in a column

4	0	1	0	0	1	0	0	1	0	0	1	0

feasible row

4	0	1	0	0	1	0	0	0	0	0	1	1

feasible row

4	0	1	0	0	0	0	0	1	0	0	1	1

feasible row

4	0	0	0	0	1	0	0	1	0	0	1	1

feasible row

4	0	1	0	0	1	0	0	1	0	0	0	0

infeasible: violates Proposition 1

Figure 1: Row generation during `generate-next-row`

ordering. Let M_i be the submatrix consisting of the first i rows of M, let O_i be the total number of ones in M_i and r_i be the number of ones in row i. Let lb be a lower bound on b. Then,

$$O_i + r_i(v - i) \geq 3 \cdot lb. \tag{1}$$

Proof. The left-hand side is an upper bound on the number of ones in M, while the right-hand size is a lower bound on it. □

Whenever (1) is violated, M_i is considered infeasible and the algorithm backtracks on row i.

When the number of rows is v, we also ensure the global feasibility that every column has either three or no 1's.

3.3 Canonicity testing

For each possible row contents for the incidence matrix, associate a rank according to the modified lexicographical ordering described in the previous section. A design B can be represented by the set R_B of ranks for the rows of its incidence matrix. Let C be the set of designs in the same equivalence class of isomorphic designs. We considered a design $B \in C$ to be canonical (i.e. to be the canonical class representative) if the set R_B is lexicographically smaller than $R_{B'}$ for all $B' \in C \setminus \{B\}$. We will say that B is lexicographical smaller than B' ($B <_L B'$) whenever R_B is lexicographically smaller than $R_{B'}$.

Our implementation employs a canonicity test algorithm by Denny and Gibbons [3, 4]. The canonicity test algorithm consists of a backtracking algorithm which goes through the possible permutations on the rows and employs quick tests that permit an early non-canonicity detection and also computes the automorphism group of the canonical designs. We describe this method next.

Let $\mathcal{B}$ be a design on v points; let $k \leq v$ and $\Phi_k = [p_1, p_2, \ldots, p_k]$ represent the point relabeling $(p_1 \to 1), (p_2 \to 2), \ldots, (p_k \to k)$. The canonicity test is a backtracking on Φ_v in strictly increasing lexicographical order. Given a partial relabeling Φ_k, we construct $\mathcal{B}'_k$, the relabeled partial structure on points $\{1, 2, \ldots, k\}$. Let $\mathcal{B}_k$ be the partial structure corresponding to the first k rows of the incidence matrix of $\mathcal{B}$. The algorithm continues in the following way, in each of the cases below:

1. If $\mathcal{B}_k <_L \mathcal{B}'_k$ then backtrack on level k (since any extension of Φ_k maps $\mathcal{B}$ to a lexicographically larger design).
2. If $\mathcal{B}_k >_L \mathcal{B}'_k$ then reject $\mathcal{B}$ and exit canonicity test (since any extension of Φ_k maps $\mathcal{B}$ to a lexicographically smaller design, proving that $\mathcal{B}$ is not canonical).
3. If $\mathcal{B}_k = \mathcal{B}'_k$ then
   if ($k = v$) then record the automorphism Φ_v of $\mathcal{B}$, else extend Φ_k to Φ_{k+1} by setting p_{k+1} to its smallest possible value.

The algorithm continues in this way, until we exit in step 2 (non-canonical $\mathcal{B}$) or the backtracking is finalized (canonical $\mathcal{B}$).

The canonicity test must be invoked whenever a complete anti-Pasch PSTS is generated. Canonicity tests may also be invoked for partially generated designs, since only canonical partial matrices can be extended to canonical matrices, given the ordering imposed on the matrix rows. Our experiments reported in Section 5 show that canonicity testing at each generated row is much more efficient than canonicity testing restricted to complete matrices. These two variations are reflected in the main algorithm by doing the canonicity testing at either the line marked with (*) or the one marked with (**).

Example 1. *Canonicity testing conditions.*
In Figure 2, since $B >_L B'$, we conclude B is not canonical and we abandon the canonicity testing algorithm.
Remark: Φ induces a column permutation on the matrix, since the matrix must remain block ordered.

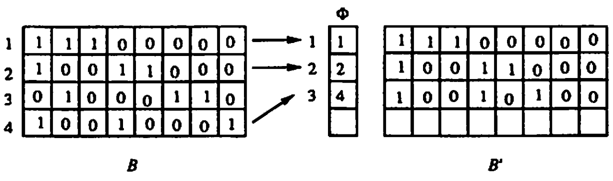


Figure 2: One step of the canonicty testing algorithm.

3.4 The distributed version of the algorithm

Our algorithm likewise the algorithms described in [4, 5, 10] process each design independently, without explicitly examining previously generated designs. Therefore, it can be fully parallelized: different processors may deal with the completion of different partially filled incidence matrices (*starter configurations*) independently.

Gibbons and Denny [4] suggest the following distributed algorithm. Choose a row L such that enumeration up to row L can be done quickly; the partial designs up to row L are the starter configurations. Each processor deals with extending a specified subset of the starter configurations. With this approach, every processor generates *all* starter configurations, and not only the ones it is supposed to process. Lexicographically smaller starter configurations tend to take longer to extend, so one should try to distribute lexicographically consecutive ones evenly among the processors. McKay [10] suggests to split the search among P processors evenly by letting processor i extend all starter configurations labeled j, such that $(j \bmod P) = i$.

Our distributed algorithm uses a different approach. One processor (the sever) is responsible for generating the starter configurations and the other processors (the clients) are responsible for extending them. Disk memory constraints prohibit the storage of starter configurations. The server holds the current starter configurations and waits for a client to request a configuration. When a client requests a configuration, the server delivers it to the client and proceeds to generate the next configuration.

There are two advantages of this approach over the ones described before. First, load balance between clients is more likely achieved by our approach, since clients request tasks whenever they are free, rather than having a share of all tasks pre-assigned to them. Second, the backtracking process done by the server in order to generate the next starter configuration happens in parallel with the processing of configurations by clients. This suggests that we can choose over a wider range of values of L. This choice will be influenced by the relative speed of the server with respect to the clients.

4 The catalogue of anti-Pasch $PSTS(v)$ for $v \leq 15$ with maximum b

The following table shows enumeration results for the maximal anti-Pasch PSTS of order up to 15 obtained by our algorithm. We include information on how many of these designs are also mitre-free. A *mitre* is a configuration of the form $\{\{a,b,c\},\{a,d,e\},\{a,f,g\},\{b,d,f\},\ \{c,e,g\}\}$. STSs (PSTs) that contain no Pasch and no mitre are called 5-sparse [6].

The columns labeled by v and b represent the order of the PSTS and the maximum number of blocks in a $PSTS(v)$, respectively. The third column contains the number of non-isomorphic PSTSs of order v containing exactly b blocks. The fourth column reports on the total number of distinct designs; this value was computed using the automorphism group sizes of the various non-isomorphic designs. For each table entry, the second row indicates how many of the anti-Pasch designs are 5-sparse. Whenever computed values represent new results, they are marked in bold. The algorithm correctly reproduces the known results. The orders of the maximal anti-Pasch PSTS that are Steiner triple systems are underlined.

v	b	#NonIsoDesigns	#DistDesigns
5	2	1	15
5-sparse		1	15
6	3	1	120
5-sparse		1	120
7	5	1	420
5-sparse		0	0
8	8	1	840
5-sparse		0	0
$\underline{9}$	12	1	840
5-sparse		0	0
10	12	**6**	**5,518,800**
5-sparse		**1**	**151,200**
11	15	**14**	**257,425,920**
5-sparse		**5**	**55,218,240**
12	19	**5**	**2,075,673,600**
5-sparse		**1**	**479,001,600**
13	24	**2**	**4,151,347,200**
5-sparse		**0**	**0**
14	28	**6**	**319,653,734,400**
5-sparse		**0**	**0**
$\underline{15}$	35	1	21,794,572,800
5-sparse		0	0

Tables 1, 2, 3, and 4 in the Appendix contain the complete isomorph-free catalog of maximal anti-Pasch PSTS on v points, for $v \leq 15$, found by our algorithm. The column labeled s contains an arbitrary reference number to distinguish designs of the same order; each triple of the design is reported vertically to save space. The 5th column contains the size of the automorphism group of the design; dividing $v!$ by this size, we obtain the number of distinct designs isomorphic to it. Column M reports on the number of mitre configurations found in each design. The last column reports on the *leave graph* of the design, which is the graph with vertex set $[1, v]$ and edge set corresponding to the uncovered pairs in the design.

5 Experimental analysis of the algorithm

The distributed program was run on a system with one server and up to 31 client machines; the sequential program was tested on the server only. The server consisted of a Sun UltraAX-MP with four 400 MHz UltraSPARC-II processors and 1024 MB of RAM. The clients were Sun Ultra 5, with one 360 MHz UltraSPARC-Iii processor and 128 MB of RAM.

We investigated the following questions:

1. How much improvement is obtained when doing canonicity testing at each row rather than at the last row only ?
2. How many rows (parameter L) should be preprocessed by the server?
3. How much improvement on total time can we get by increasing the number of clients?

The answer to the first question is that a dramatic improvement is obtained when doing canonicity testing at each row (rather than at the last row only) for $v \geq 11$. For $v = 11$ and $v = 12$ the algorithm was 8 and 23 times faster, respectively. For $v = 14, 15$, we were unable to complete the classification if the canonicity testing was applied only at the end. After these findings, we run all other experiments doing canonicity tests at every row.

Data for investigating the second question is gathered in Table 5. The number of clients employed was 31. For each v, all possible values of L have been tried. We report the total CPU time (all clients considered), as well as the average ((total CPU time)/31) and the maximum CPU time for any given client. We also report the total real time from beginning to end of the generation. Number of units refer to the total number of starter configurations processed by all clients.

From the data for $v = 12, 13$, we found that a few medium values of L starting at around $v/2$ tended to yield smaller running times. This is illustrated in the graphs of Figure 3. We tried such medium values for $v \geq 14$ and report the results for which running time didn't exceed several days. This approach enabled us to solve these cases.

In order to have some insight on question 3, we did experiments with $v = 13$ combined with a range of preprocessed rows $L \in [6, 12]$. Table 6 shows results when the number of clients is 1, 2, 5, 10, 20, 31. We observed a sharp decline on running time from 1 to 2 and from 2 to 5 clients, and a much less significant decline from 20 to 31. This can be observed in the graphs in Figure 4, where curves for $L = 6, 7, 8, 9$ are plotted showing running times for increasing number of clients.

6 Conclusion

In this article, we have described an algorithm for the isomorph-free generation of maximal anti-Pasch partial Steiner triple systems. A distributed version of the algorithm has been designed, implemented and experimentally analyzed.

The model in which approximately 50-65% of the incidence matrix rows are processed by the server while the rest is completed by the clients was successful for our implementation and hardware specification. This enabled the completion of the classification of these systems for order up to 15.

References

[1] C.J. Colbourn and A. Rosa. Triple Systems. Oxford University Press, 1999.

[2] Y.M. Chee, C.J. Colbourn, and A.C.H. Ling. Asymptotically optimal erasure-resilient codes for large disk arrays. *Discrete Appl. Math.* 102:3–36, 2000.

[3] P.C. Denny. Search and enumeration techniques for incidence structures. *Master's thesis*, University of Auckland, 1998.

[4] P.C. Denny and P.B. Gibbons. Case studies and new results in combinatorial enumeration. *J. Combin. Des.*, 8:239–260, 2000.

[5] P.B. Gibbons, R.A. Mathon and D.G. Corneil. Computing techniques for the construction and analysis of block designs. *Util. Math*, 11:161-192, 1977.

[6] M.J. Grannell and T.S. Griggs. Configurations in Steiner triple systems. *Combinatorial Designs and their applications*, 103–126, Chapman & Hall/CRC Res. Notes Math. 403, Chapman & Hall/CRC, 1999.

[7] M.J. Grannell, T.S. Griggs, and C.A. Whitehead. The resolution of the anti-Pasch conjecture. *J. Combin. Des.*, 8:300–309, 2000.

[8] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz and D.A. Paterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12:182–208, 1994.

[9] P. Kaski and P. R. J. Ostergard. The Steiner triple systems of order 19. *Math. Comp.*, 73:2075–2092, 2004.

[10] B.D. McKay, Isomorph-free exhaustive generation, *J. Algorithms*, 26:306–324, 1998.

[11] L. Moura. Rank Inequalities and Separation Algorithms for Packing Designs and Sparse Triple Systems. *Theoret. Comput. Sci.*, 297:367–384, 2003.

Appendix: Tables and Figures

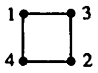
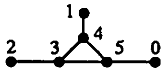
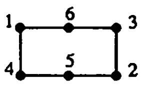
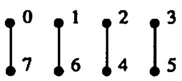
s	Designs	G	M	Leave Graph
$v = 5$, $b = 2$				
1	00 13 24	8	0	
$v = 6$, $b = 3$				
1	001 122 345	6	0	
$v = 7$, $b = 5$				
1	00012 13534 24656	12	1	
$v = 8$, $b = 8$				
1	00011224 13534356 24675677	48	1	
$v = 9$, $b = 12$				
1	000011122234 135734634565 246857886778	432	6	

Table 1: Anti-Pasch PSTS catalogue (part I).

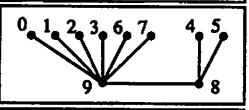
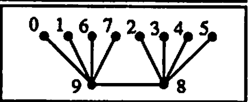
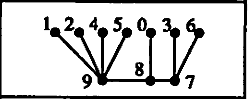
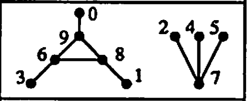
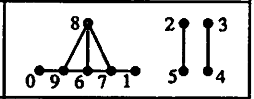
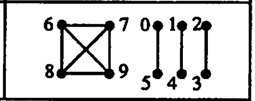
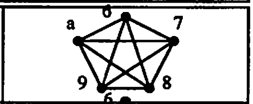
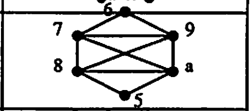
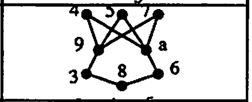
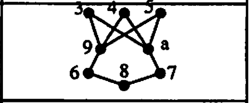
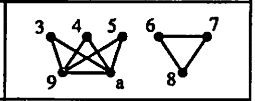
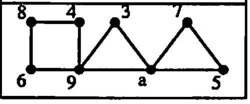
s	Designs	G	M	Leave Graph
$v = 10$, $b = 12$				
1	000011122234 135734634565 246857886779	12	3	
2	000011122234 135734634565 246857896779	16	3	
3	000011122234 135734634565 246957886798	2	1	
4	000011122234 135734634575 246859786998	3	1	
5	000011122234 134534534755 267898686979	2	1	
6	000011122334 123423545455 678989769768	24	0	
$v = 11$, $b = 15$				
1	000001111222334 123452345345455 6789aa978689a76	120	0	
2	000001111222334 123462346345455 5789aa978689a76	4	1	
3	000001111222334 125792468368455 3468a579aa79678	1	2	
4	000001111222334 125782468369455 3469a57a987a678	2	1	
5	000001111222334 125782468367455 3469a57a989a678	6	1	
6	000001111222334 125782468357456 3469aa57969878a	1	1	

Table 2: Anti-Pasch PSTS catalogue (part II).

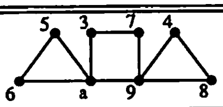
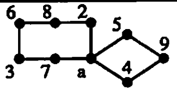
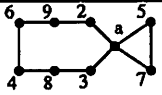
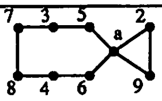
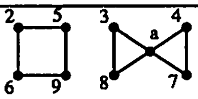
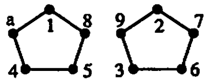
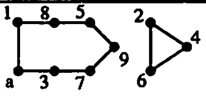
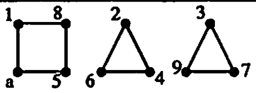
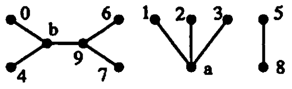
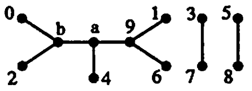
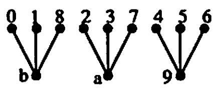
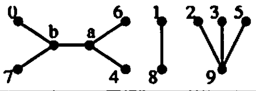
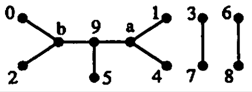
s	Designs	G	M	Leave Graph
$v = 11, b = 15$ (cont.)				
7	000001111222334 124682457356567 3957aa69847889a	1	2	
8	000001111222346 135793468345857 2468a57a9967a89	1	1	
9	000001111223346 124682357574598 a357946896879aa	2	3	
10	000001111223345 124682357564876 a35794689789aa9	2	0	
11	000001111223345 124682357794566 a357946898a978a	8	0	
12	000001112223456 135793463457878 2468a57986aa99a	10	5 5	
13	000001112223446 135793463586587 2468a57997a8a9a	2	0	
14	000001112223446 135793463586587 2468a579a7989aa	4	0	
$v = 12, b = 19$				
1	0000011112222334456 13579346834586758a7 2468a57b9967b8b9aba	1	1	
2	0000011112222334456 1357834683457695877 2469a57aba6988bb9ab	1	1	
3	0000011112222334459 135783468345667567a 2469a57a998b78bab8b	3	3	
4	0000011112222334458 1357834693456675679 2469a57bab8a78ab98b	1	0	
5	0000011112222334457 135783468345668567a 2469a57b998a7abb98b	1	1	

Table 3: Anti-Pasch PSTS catalogue (part III).

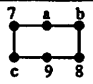
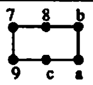
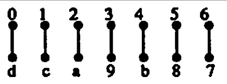
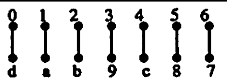
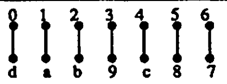
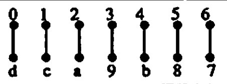
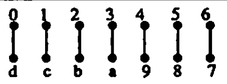
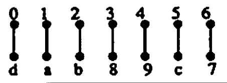
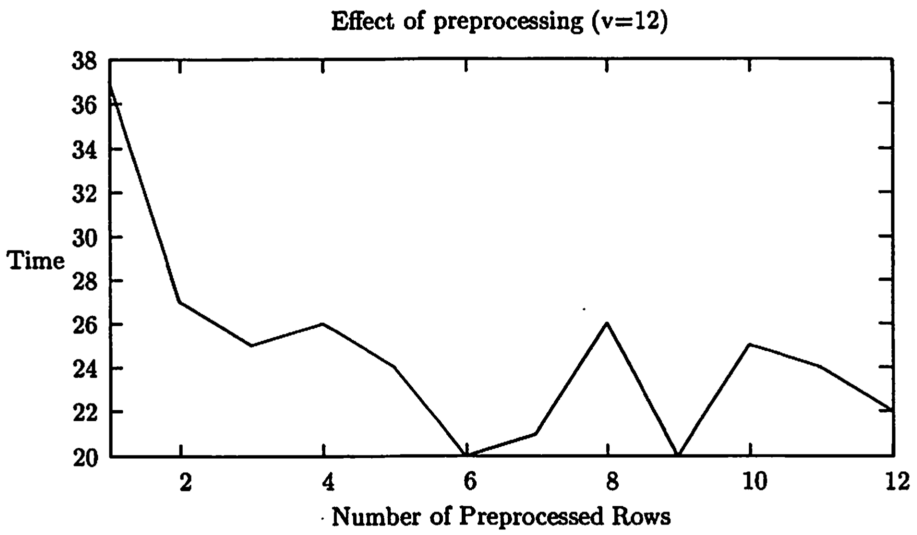
s	Designs	G	M	Leave Graph
$v = 13$, $b = 24$				
1	000000111112222233344456 13579b3456a3456956856787 2468ac79b8ca87cb9bccaba9	6	11	
2	000000111112222233344456 13569b3457834569568567 87 2478ac9b6ac78cabbcaa9c9b	2	5	
$v = 14$, $b = 28$				
1	0000001111122222333444556677 13579b3468a3458b6785899c8a9a 2468ac579dbc679ddbaacdbdbccd	1	7	
2	0000001111122222333444556677 13579b3468c3458a6785899b8a9a 2468ac579bdd679cbcaadbcdcddb	1	2	
3	0000001111122222333444556677 13579b3468c3458a6785899b8a9a 2468ac579bdc679ddbaadbcdcbdc	1	3	
4	0000001111122222333444556677 13579b3468a3458967858cab899a 2468ac57d9bc67bdabd9adcdcbcd	3	4	
5	0000001111122222333444556678 13579b3469a3457867858c7b9a9a 2468ac578dbc69adbd9abdcdcdbc	12	4	
6	0000001111122222333444556678 13579b3469c3457867b58a789a9a 2468ac578bd96adccaddbcb9dbcd	4	1	
$v = 15$, $b = 35$				
1	00000001111112222223333444455566678 13579bd3469ac34578b678a58ab78979c9a 2468ace578bde96aecdbcded9cebecaeddb	60	2	

Table 4: Anti-Pasch PSTS catalogue (part IV).

v	L	Client CPUTime			RealTime	#Units
		Total	Avg	Max	(seconds)	
11	1	17.83	0.58	17.83	39	1
	→ 2	11.72	0.38	11.72	**14**	4
	3	21.04	0.68	12.90	15	23
	4	18.53	0.60	10.94	18	113
	5	18.60	0.60	8.41	19	452
	6	5.17	0.17	0.66	22	2,426
	7	14.99	0.48	0.75	47	7,842
	8	9.80	0.32	0.52	86	13,768
	9	1.92	0.06	0.15	62	8,063
	10	0.17	0.01	0.03	26	877
	11	-	-	-	16	0
12	1	24.15	0.78	24.15	37	1
	2	24.16	0.78	24.16	27	2
	3	23.98	0.77	22.63	25	7
	4	23.93	0.77	22.53	26	32
	5	25.45	0.82	21.81	24	134
	→ 6	23.60	0.76	17.60	**20**	508
	7	21.39	0.69	7.02	21	1,427
	8	16.66	0.54	2.66	26	1,695
	→ 9	12.62	0.41	3.01	**20**	520
	10	3.85	0.12	0.35	25	632
	11	0.14	0.00	0.04	24	270
	12	-	-	-	22	0
13	1	616.31	19.88	616.31	626	1
	2	618.58	19.95	618.58	623	2
	3	617.15	19.91	472.91	476	7
	4	616.46	19.89	471.98	475	33
	5	618.74	19.96	274.70	276	134
	6	615.34	19.85	192.62	193	230
	7	675.63	21.79	82.11	83	499
	→ 8	564.97	18.22	46.02	**79**	3,824
	9	452.05	14.58	22.17	257	16,677
	10	158.81	5.12	6.64	656	51,762
	11	12.81	0.41	1.22	607	19,779
	12	0.81	0.02	0.06	551	365
	13	-	-	-	643	0
14	8	481,520.58	15,532.92	42,038.06	42,294	93,939
	→ 9	436,283.51	14,073.66	16,479.11	**24,870**	853,365
15	→ 9	671,345.23	21,656.30	34,752.76	**34,868**	404,142
	10	635,159.30	20,489.01	20,846.04	62,640	3,045,865

Table 5: The effect of the number of rows preprocessed by the server.



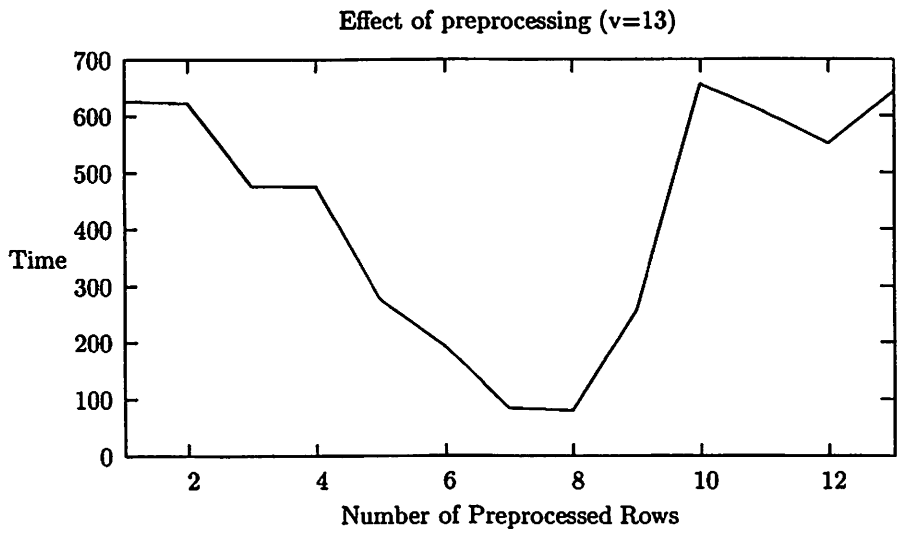


Figure 3: Total running time vs number of preprocessed rows (L).

L	NumClients	Total Client CPUTime	RealTime
6	1	616.53	1,316
	2	616.52	331
	5	617.18	195
	10	616.37	283
	20	616.05	195
	31	615.34	193
7	1	614.42	1,406
	2	612.48	329
	5	617.95	152
	10	676.46	136
	20	615.24	83
	31	675.63	83
8	1	579.31	2,484
	2	563.45	432
	5	578.07	201
	10	666.63	161
	20	572.79	103
	31	564.97	79
9	1	467.33	2,895
	2	470.87	851
	5	457.59	468
	10	548.29	318
	20	612.80	277
	31	452.05	257
10	1	152.06	3,732
	2	156.19	2,074
	5	250.49	1,139
	10	320.47	800
	20	238.90	697
	31	158.81	656
11	1	9.59	1,705
	2	9.45	1,161
	5	10.04	765
	10	11.16	623
	20	11.62	637
	31	12.81	607
12	1	0.09	580
	2	0.04	557
	5	0.09	553
	10	0.18	550
	20	0.12	553
	31	0.18	551

Table 6: The effect of the number of processors (clients) for $v = 13$ and various values of L.

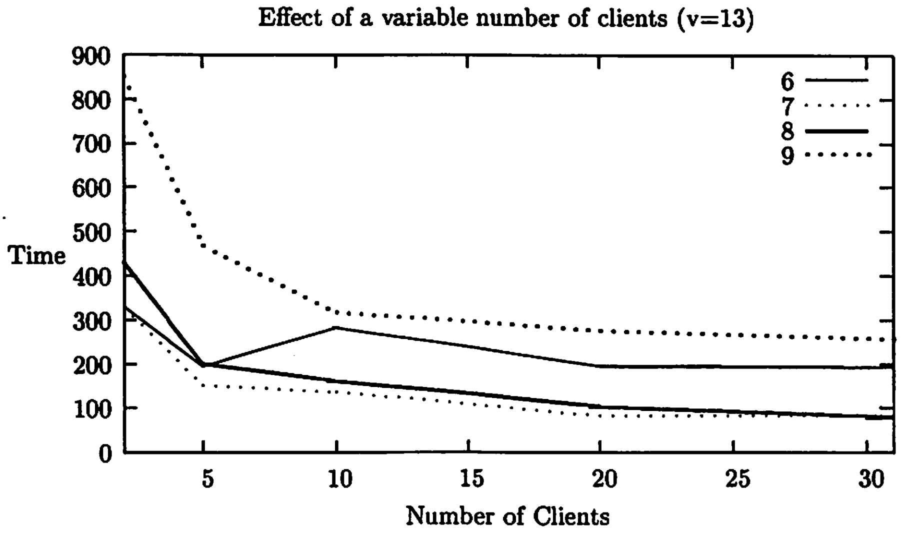


Figure 4: Running time for different number of clients, for $L = 6, 7, 8, 9$.