HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports ISSN 0783–540X
No. **10**; January 1993 ISBN 951–22–1382–6

CONSTRUCTING COVERING DESIGNS BY SIMULATED ANNEALING

Kari J. Nurmela and Patric R. J. Östergård

Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, Finland

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
SF–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B**: Technical Reports | ISSN 0783–540X
No. **10**; January 1993 | ISBN 951–22–1382–6

Constructing Covering Designs by Simulated Annealing

Kari J. Nurmela and Patric R. J. Östergård

Abstract: A $t-(v,m,k,\lambda)$ *covering design* is a pair $(X,\mathcal{A})$, where X is a set of v elements (called *points*) and $\mathcal{A}$ is a multiset of k-subsets of X (called *blocks*), such that every m-subset of X intersects at least λ members of $\mathcal{A}$ in at least t points. It is required that $v \geq k \geq t$ and $v \geq m \geq t$. Such a covering design gives an upper bound on $C_\lambda(v,m,k,t)$, the number of blocks in a minimal $t-(v,m,k,\lambda)$ covering design. In this work it is shown how simulated annealing can be used in searches for covering designs. A cost function and a neighborhood structure needed by the algorithm are presented. The space complexity of the implementation and the time complexity of the cost change calculation are also given. Several options to increase the performance of the algorithm are tested. A performance comparison to local optimization is carried out. Finally, a description how to obtain, compile and use the computer program `cover`, which has been developed during the research, is given.

Keywords: Combinatorial optimization, covering design, simulated annealing.

Printing: TKK Monistamo; Otaniemi 1993

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
SF–02150 ESPOO, FINLAND

Phone: $\dfrac{90}{+358\text{–}0}$ 4511
Telex: 125 161 htkk sf
Telefax: +358–0–465 077
E-mail: lab@hutds.hut.fi

Contents

1 Introduction

Let $v \geq k \geq t$ and $v \geq m \geq t$. A $t-(v, m, k, \lambda)$ *covering design* is a pair $(X, \mathcal{A})$, where X is a set of v elements (called *points*) and $\mathcal{A}$ is a multiset of k-subsets of X (called *blocks*), such that every m-subset of X intersects at least λ members of $\mathcal{A}$ in at least t points. We define

$$C_\lambda(v, m, k, t) = \min\{|\mathcal{A}| \mid \text{there is } t-(v, m, k, \lambda) \text{ covering design } (X, \mathcal{A})\}.$$

Covering designs are here defined in a very general sense. Most of the published results on covering designs have concerned the case $m = t$ (see [14] for an excellent survey). Some results on the case $m > t$ exist [2, 3, 19]; in those papers $\lambda = 1$, and the notation $B(t, k, m, v) = C_1(v, m, k, t)$ is used.

A $t-(v, m, k, \lambda)$ *packing design* can be analogously defined; it is a pair $(X, \mathcal{A})$, where X is a set of v elements and $\mathcal{A}$ is a multiset of k-subsets of X, such that every m-subset of X intersects *at most* λ members of $\mathcal{A}$ in at least t points. Let $D_\lambda(v, m, k, t) = \max\{b \mid \text{there is a } t-(v, m, k, \lambda) \text{ packing design with } |\mathcal{A}| = b\}$.

If a design simultaneously is both a packing and a covering design, it is called a *t-design*. We use t, v, k, λ with the same meaning as above ($m = t$). If a t-design with these parameters exists, it is called a $t-(v, k, \lambda)$ design (sometimes $S_\lambda(t, k, v)$). 2-designs are generally referred to as *balanced incomplete block designs* (BIBDs). In this paper packing designs and t-designs are discussed very briefly.

The problem of finding the exact value of $C_\lambda(v, m, k, t)$ is very difficult and has been solved only for small values of the parameters and in some special cases. Such values have been proved by combinatorial arguments and in some cases by computer search [19]. Due to combinatorial explosion even computers are not of much help in the general case, so for most parameters only upper and lower bounds on $C_\lambda(v, m, k, t)$ are known. Upper bounds are obtained by explicitly constructing covering designs.

In this paper we show how probabilistic methods can be used in the search for covering designs. Many mathematicians react against this approach but as we can see there are many arguments for making use of it. Assume that we for some parameters λ, v, m, k, and t know that $C_\lambda(v, m, k, t) \leq b$. We then try to find a covering design with $b-1$ blocks. If no covering design is found, we can conjecture that no such covering design exists and concentrate on improving the lower bound. If we find a covering design we can try to further improve on the upper bound. When no improvements can be made we can study the record-breaking covering design. Many construction methods have been found by this method, although the role of the computer is seldom mentioned in the published results. Finally, probabilistic methods can be used to find nonisomorphic designs with given parameters [12]. Two designs are said to be isomorphic if one can be transformed into the other by a permutation of the blocks and the points.

In Section 2 we give some basic theoretical results on covering designs. In Section 3 we discuss simulated annealing, the probabilistic method that has been used in the search for covering designs. In Section 4 and 5 the computer implementation and problems and options related to that are treated. In Section 6 the program is evaluated by comparing it to local optimization. Section 7 gives information about how to use the program `cover`. This program is available for all researchers. In Section 8 it is shown how the program can be used to find other designs than pure covering designs, and finally, in Section 9, we conclude the paper.

2 Covering Designs

The definition of general covering designs was given in the Introduction. In the subsequent discussion of such designs we adopt the following conventions. The elements of X (the points) are denoted by the numbers $0, 1, 2, \ldots, v-1$. The letter b is used to denote the number of blocks in a $t-(v, m, k, \lambda)$ covering design. A k-set is said to *cover* an m-set (in t points), if their intersection contains at least t points. If $\lambda = 1$, we sometimes drop the index λ and use $C(v, m, k, t) = C_1(v, m, k, t)$.

Example 1 The family $\{0123, 3456, 5678\}$ is a $2-(9, 4, 4, 1)$ covering design. However, this family of 4-sets is not a minimal design, because $\{0123, 4567\}$ has the same property. A single 4-set will not suffice, so $C(9, 4, 4, 2) = 2$.

A well-known result for covering designs with $m = t$ is the so-called *Schönheim lower bound* [17]:

$$C_\lambda(v, t, k, t) \geq \left\lceil \frac{v}{k} \left\lceil \frac{v-1}{k-1} \cdots \left\lceil \frac{v-t+1}{k-t+1} \lambda \right\rceil \cdots \right\rceil \right\rceil . \tag{1}$$

A generalization of (1) to the case $m \geq t$ does not seem possible, since the proof of (1) is based on a counting argument that does not work for $m > t$. The following lower bound holds in the general case.

Theorem 1 $C_\lambda(v,m,k,t) \geq \left\lceil \dfrac{\lambda \binom{v}{m}}{\sum_{i=t}^{\min(k,m)} \binom{k}{i}\binom{v-k}{m-i}} \right\rceil$.

Proof: The total number of m-sets is $\binom{v}{m}$. Each block in a $t-(v,m,k,\lambda)$ covering design covers $\sum_{i=t}^{\min(k,m)} \binom{k}{i}\binom{v-k}{m-i}$ m-sets; a simple division now gives a lower bound for the number of blocks in the case $\lambda = 1$. The theorem is obtained by multiplying the result by λ. □

Upper bounds for the number of blocks in covering designs can be obtained by explicitly constructing such a design. The method described in the Introduction can then be used. If we are able to construct a covering design with b blocks we try to construct a covering design with $b-1$ blocks and repeat this procedure until we fail at constructing a design.

Very few results on the construction of covering designs by computer have been published. In [14, p. 394] some approaches are mentioned, but there is still a lot to do in this area. We hope that this paper at least partly fills this gap.

2.1 Applications of Covering Designs

In discussions on applications of covering designs, two fields are generally mentioned. They can be used as lottery schemes (see, e.g., [15, 16]) and for error-trapping decoding (see [5]).

Covering designs can be used for example in Finnish lotto. In Finnish lotto (as today) 7 numbers out of 37 are chosen randomly, and if you have guessed at least 4 numbers correctly, you win. Finnish lotto supports the use of lottery schemes. There are six different schemes available (called 'rake systems'), which are $5-(v,5,7,1)$ covering designs, v ranging from 12 to 18 and b (the number of blocks in the designs or, equivalently, the number of rows in the schemes) ranging from 60 to 600. Therefore, it is guaranteed that if you guess 5 numbers correctly, there is at least one row in your scheme that contains those five numbers.

In the case of error-trapping decoding, covering designs with $m = t$, $\lambda = 1$ can be used. The number of blocks in the covering design used gives the complexity of the decoding procedure; this justifies the importance of finding minimal covering designs. We do not further discuss this application here but refer the reader to the aforementioned reference.

3 The Simulated Annealing Algorithm

Simulated annealing, or SA for short, is a combinatorial optimization method. The aim in combinatorial optimization is to find the best solution among a finite number of possible solutions. Simulated annealing is good at finding near-optimal solutions with a reasonable amount of computing. Note that we do not know for sure, if the best solution found is the global optimum. This restricts the use of the algorithm to cases where a good local optimum is also acceptable.

3.1 Local Optimization

The basic concepts of simulated annealing come from local optimization, also called iterative improvement. A combinatorial optimization problem can be characterized by a set of *solutions* and a *cost function*, which attaches a cost to every solution. Our goal is to find the solution with minimum cost. (A maximization problem can be turned into a minimization problem easily by reversing the sign of the cost function.)

In local optimization one must first define a procedure to perturb the solutions in order to get new ones. Perturbation is accomplished by small changes to the current solution. Solution S' is a *neighbor* of another solution S, if S' can be obtained from S by one such change. The *neighborhood* of a given solution S is the set of all neighbors of S.

The local optimization algorithm takes an initial solution, which can be random or based on some heuristic procedure, and then starts to investigate its neighborhood. If it finds a solution in the neighborhood whose cost is lower than the present cost, it moves to that solution. This procedure is iterated until no further improvements can be made.

The solution this algorithm finds is necessarily a *local* optimum, but it is usually not the global optimum. We can of course apply the method many times with different initial solutions and hope that the best solution found is the global optimum.

Physical system	*Optimization problem*
State	Feasible solution
Energy	Cost
Ground state	Optimal solution
Rapid quenching	Local search
Careful annealing	Simulated annealing

Table 1: The analogy [7].

3.2 Simulated Annealing

The term simulated annealing comes from a physical point of view. Annealing is a physical process where a crystal is cooled down from the liquid phase to the solid phase in a heat bath. If the cooling is done carefully enough (instead of rapid quenching), the energy state of the solid at the end of the cooling is at its minimum (or very near to it).

Simulation of the physical cooling can be done with the Metropolis algorithm [13], invented in the early days of computing. It generates sequences of states of the solid in the following way: Given a current state i of the solid with energy E_i, then the next state j is generated by applying a small perturbation to the solid, for instance moving a particle. The energy of the next state is E_j. If the energy difference $E_j - E_i$ is less than or equal to zero, the state j is accepted as the new current state. Otherwise, the state j is accepted with a probability given by

$$\exp\left(-\frac{E_j - E_i}{k_B T}\right),$$

where T is the temperature of the solid and k_B denotes the Boltzmann constant.

If the lowering of the temperature is done slowly enough, the solid reaches thermal equilibrium at each temperature. In the Metropolis algorithm this is achieved by applying a sufficiently large number of perturbations at each temperature.

The simulated annealing algorithm [4, 8] can be viewed as a process analogous to the simulation of cooling down a solid by the Metropolis algorithm. The analogy between SA (combinatorial optimization) and the physical situation (cooling down a solid) is presented in Table 1. Using the analogy we get the simulated annealing algorithm in Figure 1. For a comprehensive treatment of the simulated annealing algorithm, see [1, 10].

SA can be viewed as an extension to local optimization. Local optimization finds new solutions by making small transformations to the current solution

1. Get an initial solution S and an initial temperature T.
2. While *stop criterion* not satisfied do the following:
   (a) While *inner loop criterion* not satisfied do the following:
      i. Select a neigbor S' of S.
      ii. Let $\Delta = \text{cost}(S') - \text{cost}(S)$.
      iii. If $\Delta \leq 0$ (downhill move), set $S = S'$.
      iv. If $\Delta > 0$ (uphill move), set $S = S'$ with probability $e^{-\Delta/T}$.
   (b) Reduce temperature T.
3. Return S.

Figure 1: Simulated annealing.

in such a way that every new solution is better than or equal to the previous solution. In simulated annealing even a move that makes current solution worse is accepted with probability

$$P_{\text{accept}} = \exp\left(\frac{-\Delta}{T}\right) = \exp\left(-\frac{\text{cost}(S') - \text{cost}(S)}{T}\right), \tag{2}$$

where $\text{cost}(S)$ denotes the cost function assigned to the current solution, $\text{cost}(S')$ denotes the cost function assigned to the proposed next solution, Δ is the cost change, and the 'temperature' T is simply a control parameter in the same units as the cost function. Ideally, when local optimization is trapped in a poor local optimum, simulated annealing can 'climb' out of the poor local optimum.

The criterion for stopping the iteration (detecting when the process is 'frozen') in step 2 in Figure 1 varies in the literature. One possible solution is to give a required percentage of accepted moves. When the actual percentage of accepted moves is below this required level for at least a given number of successive executions of steps 2a and 2b and no improvement of the best solution found so far is recorded, the process is considered frozen. Another possible solution is to stop when the cost function at the end of step 2a is the same for a given number of consecutive temperatures.

In step 2(a)i, S' is usually selected randomly among all the neighbors of the current solution, with the same probability for all neighbors. However, with a complicated neighborhood structure, a nonuniformly random selection might be appropriate.

A common inner loop criterion (step 2a), is to perform the loop a constant

(L) number of times. The value of L should be large enough, so that the equilibrium is achieved before reducing the temperature. A rule of thumb is to take L equal to or proportional to the size of the neighborhood [1].

The new temperature in step 2b is usually calculated according to

$$T' = rT, \tag{3}$$

where T' is the new temperature and r is typically in the range $0.95 \ldots 0.99$. Some other *cooling schedules* have also been studied (see for example [7]), but this exponential schedule seems to be not only one of the most efficient, but also very robust.

It can be shown that simulated annealing converges asymptotically to the set of globally optimal solutions, if some conditions are fulfilled [1, Ch. 2–3]. Unfortunately, pushing the probability of finding a global optimum provably to a given level requires very slow cooling schedules and long iterations at each temperature. Hence it can take more time than what it would take to accomplish an exhaustive search in the solution space. Further, we still do not know when we actually have reached one of the global optima. Nevertheless, in practise, SA often finds very good local optima with a reasonable amount of computing.

The main problems while implementing the simulated annealing algorithm are:

Selection of the cost function. Cost functions usually employ some heuristic related to the problem at hand. If there are difficulties while applying the algorithm, it is difficult to say whether the problem is too hard or whether the cost function is poor.

Efficiency. Perturbations and the corresponding cost change calculations should be simple enough to perform, so that the algorithm can perform iterations very fast.

4 Constructing Covering Designs

In the search for covering designs we let a solution S be a multiset of k-subsets of X; the cardinality of the multiset is b. In this section we show how we have chosen problem-specific parameters, like the cost function, and parameters of the simulated annealing algorithm, like the initial temperature. Data structures used are also discussed.

In the rest of this paper `this fontface` is used to denote functions, variables, etc. in our implementation of the algorithm in the C programming language.

4.1 Cost Function

Covering designs cover all m-subsets of v points λ times. One possible cost function for a solution S is the number of m-sets that are covered less than λ times.

$$\mathrm{cost}(S) = \left|\{x \in X_m \mid \ |\{y \in S \mid \ |x \cap y| \geq t\}| < \lambda\}\right|, \tag{4}$$

where X_m denotes the family of all m-subsets of the set X of v points. However, it seems logical to give a higher cost for an m-set that is covered $\lambda - 2$ times than for an m-set that is covered $\lambda - 1$ times. This inspires the following cost function:

$$\mathrm{cost}(S) = \sum_{\{x \in X_m \mid \ |\{y \in S \mid \ |x \cap y| \geq t\}| < \lambda\}} (\lambda - |\{y \in S \mid \ |x \cap y| < t\}|). \tag{5}$$

For $\lambda = 1$, (4) and (5) coincide. We then try to minimize this cost function by using simulated annealing. If the cost ever reaches 0, a covering design has been found and the iteration can be ended.

4.2 Neighborhoods

The main difficulty in implementing the cost function (5) and neighborhood structure to be described here is efficiency. We rely on an extensive tabulation of the neighborhood and covering data. A diagram of the data tables is given in Figure 2. In the sequal the tables are discussed in more detail.

To be able to efficiently store neighbors, etc., we rank individual k-sets and m-sets lexicographically, so that different subsets have different ranks and the ranks vary from 0 to $\binom{v}{k} - 1$ and $\binom{v}{m} - 1$, respectively. Algorithms for this approach has been described for example in [18, pp. 162–163]. The algorithms use binomial coefficients, which have been computed in an array.

The most natural neighborhood structure is as follows. Suppose we have a solution S. Then we can make another solution by selecting one of the k-sets belonging to S. We then replace a point in this k-set by a point not belonging to this k-set. We define *the neighborhood of S* as the set of all solutions obtained from S by one such change. The *neighborhood size* is the cardinality of this set, which is $bk(v - k)$, where b is the number of k-sets in a solution and v and k are as before. It is also possible to consider one k-set at a time and try to make a change to it as described before. This reduces the neighborhood size to $k(v - k)$.

b

1 `kset` `unrankSubset()`

`rankSubset()`

$\binom{v}{k}$

`neighbors`

$k(v-k)$

Family of all k-subsets
of v points

$\binom{v}{k}$

`coverings`

$\mathrm{Cov}(v,m,k,t)+1$

Family of all m-subsets
of v points

1 `covered`

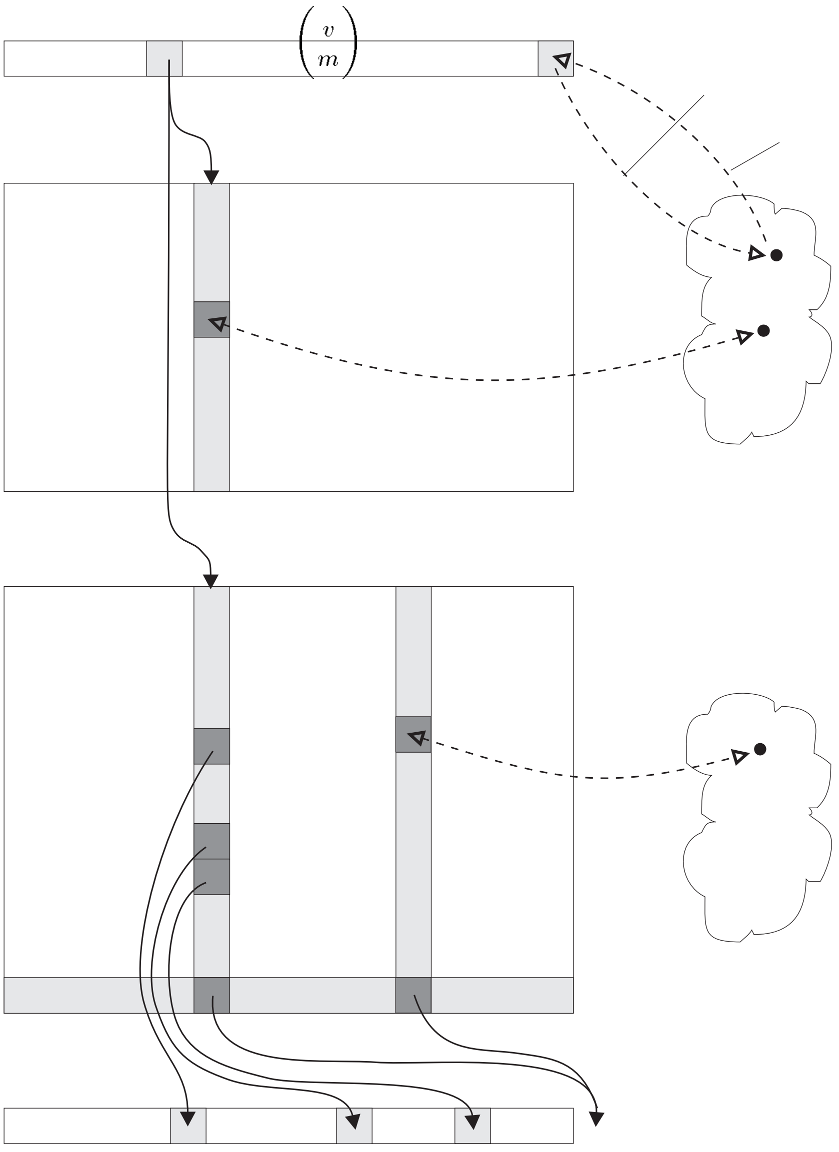

Figure 2: Dynamically allocated data tables.

We compute the neighbors of each k-set in the table `neighbors`. In principle, the table is 2-dimensional: the ranks of the k-sets in the neighborhood of a k-set with rank `Rk` are given in `neighbors[Rk][0]`, `neighbors[Rk][1]`,..., `neighbors[Rk][neighborLen - 1]`, where `neighborLen` equals to $k(v-k)$, because according to the definition of the neighborhood in Section 4.2, there are $k(v-k)$ possible next state sets for each k-set. The variable `neighborLen` should not be confused with the neighborhood size, which is the number of possible next solutions.

In practice, the table is 1-dimensional, because it is allocated dynamically, and the C programming language is not capable of handling 2-dimensional, variable size, dynamically allocated tables (actually, there is no type 'table' in C). So '`neighbors[Rk][x]`' is written '`neighbors[Rk * neighborLen + x]`'.

4.3 Coverings

The calculation of the cost of a given solution is the most critical part in implementing the simulated annealing algorithm. If it cannot be done efficiently, the performance of the algorithm will be poor. Usually, the most convenient (and efficient) way of computing the cost is to compute only the *change* of cost, when the solution is changed in a given way.

In order to compute the change of cost efficiently, we need information to answer the following questions:

- Which m-sets are covered by a given k-set? In the program this information is saved in a table named `coverings`. The ranks of the m-sets covered by a k-set with the rank `Rk` are given by `coverings[Rk][0]`, `coverings[Rk][1]`,..., `coverings[Rk][coverLen - 2]`. (Again, the table is 1-dimensional in practice: '`coverings[Rk][x]`' is written in the program as '`coverings[Rk * coverLen + x]`'.) The variable `coverLen` is the number of m-sets covered by one k-set plus one. The 'plus one' will be explained shortly.
- How many k-sets in the current solution cover a given m-set? This information is kept in a table named `covered`. The number of k-sets covering a m-set with the rank `Rm` is given by `covered[Rm]`.

In the following we consider the case $\lambda = 1$, to make the description clear. The generalization to $\lambda > 1$ is quite straightforward.

Now we can calculate the change of cost as follows. The current solution is S and the proposed new solution is S'. In S there is exactly one k-set A that is changed to another set A' in order to obtain the new solution S'. We take the m-sets covered by the set A and count the 1's in the corresponding places in the table `covered`. Let this count be Δ^-. All these m-sets will no longer be covered, when A is removed from S. Then we remove A from S, that is decrement by 1 each item in `covered` that represents an m-set covered by A. Next we count the 0's in those members of `covered` that represent an m-set covered by A'. Let this count be Δ^+. All these m-sets will be newly covered when A' is added to S. Then the change of cost (denoted `costDelta` in the program) $\Delta = \Delta^+ - \Delta^-$.

This is an inefficient way of implementing the computation, because we have to restore the table `covered`, if the proposed change is rejected. As the computation proceeds, fewer and fewer changes are accepted and we are just changing the table `covered` back and forth most of the time. Fortunately, there is a more efficient way of calculating the change of cost: we increase the size of the table `coverings` so that we can add a sentinel greater than the rank of any possible m-set after each column of the table `coverings` (see Figure 2). Then we sort the ranks of the m-sets covered by each k-set in the table `coverings` in increasing order before starting SA. Now when we want to calculate a cost change, we take two pointers, one pointing to the first cell of the column of the new k-set, the other pointing to the first cell of the column of the current k-set. Let us call the pointers `nextPtr` and `currPtr`, respectively.

Now we set `costDelta` initially to 0 and process the pointers (in the function `computeNeighbor()`) according to the following rules:

`*nextPtr` = `*currPtr`: The m-set rank to which `nextPtr` points equals the rank to which `currPtr` points. That is, the same m-set is covered by both the current and the new solution. Do not change `costDelta`. Increment both pointers to the next row.

`*nextPtr` < `*currPtr`: The m-set rank to which `nextPtr` points is less than the rank to which `currPtr` points. If the m-set with the rank to which `nextPtr` points is currently uncovered (`covered[*nextPtr]` = 0), decrement `costDelta` by 1 (this m-set will become covered, if the new solution is accepted). In any case, increment `nextPtr`.

`*nextPtr` > `*currPtr`: The m-set rank to which `currPtr` points is less than the rank to which `nextPtr` points. If the m-set with the rank to which `currPtr` points is currently covered by 1 k-set (`covered[*nextPtr]` = 1), increment `costDelta` by 1 (this m-set will no longer be covered, if the new solution is accepted). In any case, increment `currPtr`.

The process ends when both the pointers point to the sentinels at the end of each column. Table `covered` is not changed during this computation and it must be updated by the function `acceptNeighbor()` only if the new solution is accepted, in which case the current cost must also be updated by `costDelta`.

The time complexity of the calculation of a cost change is clearly

$$\mathcal{O}(\mathrm{Cov}(v, m, k, t)),$$

where $\mathrm{Cov}(v, m, k, t)$ is given by (9) below.

4.4 Space Complexity

The space demands of our approach set upper limits for the parameters of the designs.

The sizes of the tables in Figure 2 are as follows:

$$\begin{aligned}
\mathrm{size}(\texttt{neighbors}) &= k(v-k)\cdot\binom{v}{k}\cdot\texttt{sizeof(rankType)} && (6)\\
\mathrm{size}(\texttt{coverings}) &= \binom{v}{k}\cdot[\mathrm{Cov}(v,m,k,t)+1]\cdot\texttt{sizeof(rankType)} && (7)\\
\mathrm{size}(\texttt{covered}) &= \binom{v}{m}\cdot\texttt{sizeof(coveredType)} && (8)\\
\mathrm{Cov}(v,m,k,t) &= \sum_{i=t}^{\min(k,m)}\binom{k}{i}\binom{v-k}{m-i}, && (9)
\end{aligned}$$

where $\mathrm{Cov}(v, m, k, t)$ gives the number of different m-subsets out of v points, which are covered by a given k-set in at least t points.

Usually, (7) is the dominating space requirement, and it increases at a rather high rate. Maximum value of (7), denoted by $\mathrm{size}(\texttt{coverings})_{\max}$, for a given v can be obtained by setting $k = m = \lceil v/2 \rceil$, and $t = 1$ and it is $\mathcal{O}(e^v)$. See Table 2 for some numerical values.

The size of the solution space is

$$\mathrm{space}_S = \binom{v}{k}^b. \qquad (10)$$

where b is the number of k-sets in a solution. Equations (7) and (10) suggest that the solution space of a yet space-feasible problem is at its largest when b is large, m is small and t is near m.

v	size(`coverings`)$_{\max}$
3	36
6	1520
9	63504
12	3411408
15	165636900

Table 2: Some maximum values of size(`coverings`) for given v when `rankType` is 4 bytes.

4.5 Initial Temperature

The selection of initial temperature is very important. If we select too high an initial temperature, then the program will spend a lot of time in the beginning of the process without progress of any kind. If we select too low an initial temperature, then the quality of the final solutions will deteriorate.

It is difficult to give the initial temperature directly, because the value of a suitable temperature will depend on the problem instance, the neighborhood structure, the initial solution and the cost function used.

It has been proposed that the algorithm should be given the probability of acceptance of *all moves* (see, e.g., [1]), but in this case it leads to difficulties. If the desired probability is smaller than the relative number of moves that do not increase the cost, then the temperature converges below zero and the iteration formula [1, p. 60] does not work. In this problem there are then sometimes a narrow range of valid initial probabilities.

That is why we here present a more robust and intuitively more appealing initial temperature approximation method. We give the wanted initial probability of acceptance of *cost increasing moves*. Then we approximate the initial temperature according to the formulas below.

If there are n cost increasing moves around the initial solution, then we can approximate the number of moves that would be accepted n_{acc} by

$$n_{\mathrm{acc}} \approx n \cdot \exp\left(\frac{-\overline{\Delta\mathrm{cost}_+}}{T}\right),$$

where $\overline{\Delta\mathrm{cost}_+}$ denotes the average change of cost in cost increasing moves around the current solution. Then the acceptance ratio χ is

$$\chi = \frac{n_{\mathrm{acc}}}{n} \approx \exp\left(\frac{-\overline{\Delta\mathrm{cost}_+}}{T}\right)$$

from which we obtain

$$T \approx \frac{-\overline{\Delta\text{cost}_+}}{\ln \chi}. \tag{11}$$

Now we set χ to the desired initial probability and calculate an estimate for $\overline{\Delta\text{cost}_+}$ by taking randomly `T_ITER` (a suitable value is for example 100) neighbors of the initial solution and calculating the average change of cost of those moves that would increase the cost. An approximation for the initial temperature T is then given by (11). Numerical experiments show satisfying accuracy (an error of 10%–15% in the probability is usual), although the initial temperatures of some designs seem to be a bit more difficult to be estimated this way, especially if the desired initial probability is small. The average change of cost in cost increasing moves also varies among distinct solutions. In this problem the variation is not a serious problem, but one may have to take into account many solutions, not just the initial one, in some other problems.

This approach is intuitively very appealing, because we are mainly interested in how many moves are accepted despite of that they will increase the cost. We are not very interested in how many moves would increase cost and how many would not. If we want that 'practically all moves should be accepted in the beginning', we can set the probability of acceptance for cost increasing moves to, say, 0.8. We do not want to think: 'In this problem instance there are about 60% of moves around the initial solution that will be accepted in any case, so we can set the initial probability between about 60% and 100%, let's say 90%.'

It is also possible to give the initial temperature directly to our program.

4.6 Perturbation Count L

In our program the current solution is perturbed at each temperature L times before moving to the next temperature (L remains constant through the iteration). We give the program a factor `LFact` (defaulting to 1) as a parameter. The value of L is then calculated by multiplying the neighborhood size by `LFact`. It is also possible to give the value of L directly to our program.

4.7 End Condition

We use a simple end condition: the iteration is ended if the cost ever reaches a given limit (usually zero: all m-sets are covered λ times) or if the cost after each temperature doesn't improve during a given number of successive temperatures. A value around 10 usually works well enough if L is equal to the size of the neighborhood.

5 Options

5.1 Different Cost Functions

Only cost function (5) is implemented in our program. It origins from the fact that the more m-sets are covered (λ times) the closer we are to a solution. But is the *resolution* of the cost functions good enough? If there is a large number of neighbors with zero cost difference Δ in the neighborhood of a given solution, then our cost function does not guide the search very well.

Our experiments show, fortunately, that the number of zero difference moves is relatively large only when the parameters and the solution space are small. The average percentage of zero difference moves falls rapidly to 20% and below, when the size of the solution space grows.

In any case, we tried to find an even slightly better cost function by giving weights to the numbers smaller than λ in the table `covered`. The results were not very encouraging: we did not find any weighting that would have given noticeable improvements. Also, as the calculation of the cost becomes more complex, the CPU-time used by the algorithm grows. The option of weighting numbers of the table `covered` was not included in the final version of the program because of the reasons stated above.

5.2 A Different Neighborhood

The two different neighborhoods discussed in Section 4.2 are both implemented in the program. The following approach has been tested but is not implemented.

Smaller neighborhoods can lead to increased performance [7]. The neighborhoods of a given solution can be made smaller by allowing a permutation to change a k-set to a $(k-1)$-set or to a $(k+1)$-set. Then each set in the solution can be permuted by v ways instead of $k(v-k)$, a substantial improvement in this respect.

Allowing *infeasible solutions* (in this case: solutions containing non-k-sets) is compensated by adding a penalty to the cost function for these unbalanced solutions. The solutions tend to converge towards a feasible final solution.

v	size(`coverings`)$_{\max}$ with non-k-sets
3	72
6	4480
9	249984
12	14902272
15	840153600

Table 3: Some maximum values of size(`coverings`) for given v when `rankType` is 4 bytes and non-k-sets are allowed.

However, the final solution can still be slightly infeasible. Therefore, a heuristic fine-tuning is added to make these infeasible solutions feasible, even with some slight increase in the cost.

This approach does not fit very well to this problem. It seems to be difficult to find any fine-tuning algorithm which would turn an infeasible solution into a feasible one (that is, a solution consisting only of k-sets) without increasing the cost remarkably; at least if we want to keep the number of blocks in the solution (b) fixed, as the case usually is.

Another argument for not allowing non-k-sets is the increased space complexity. The space requirement (6) decreases, but the (usually dominating) space requirement (7) becomes

$$\text{size}(\texttt{coverings}) = \sum_{\tilde{k}=t}^{v} \binom{v}{\tilde{k}} \cdot \left\lceil \sum_{i=t}^{min(\tilde{k},m)} \binom{\tilde{k}}{i} \binom{v-\tilde{k}}{m-i} + 1 \right\rceil \tag{12}$$

where we have used the fact that if $\tilde{k} < t$, then none of the corresponding blocks with $\tilde{k}$ points can meet any m-set in at least t points, so they do not have to be stored anywhere. See Table 3 for some maximum numerical values for (12) with a given v. Note that the space requirement (12) does not depend on k, as (7) did. The value of size(`coverings`)$_{\max}$ for a given v is achieved by setting $m = \lceil v/2 \rceil$ and $t = 1$ and it is, as before, $\mathcal{O}(e^v)$.

5.3 Exponentiation

The simulated annealing algorithm calculates the value of $e^{-\Delta/T}$ in the inner loop once on every proposed cost increasing change. Calculating the value of e^x is a costly operation in practically all environments. There are some optimizations that can be done. One approach is to approximate the exponential function by a function, the calculation of which is not so costly.

Another approach is the tabulation of some values of the exponential function. We used this approach, because it offers more accurate approximation with equal savings in computation. The calculations have to be done every time we change the temperature. We give the size of the table (`EXPSIZE`) and use it to calculate a negligible probability `1/EXPSIZE`. This is then used to calculate the corresponding value for the exponent (`negligibleExp`) and the table is filled with exponential values, the exponent ranging from 0 to `negligibleExp`. The value for approximate exponentiation is 0 if the exponent is greater than `negligibleExp`, otherwise it is looked up at the corresponding position in the table.

5.4 On-the-Fly Approach

The performance of simulated annealing in this problem was so good that we decided to try to trade some time to space. If we reject the tables `coverings` and `neighbors` and compute the m-sets covered by a given k-set and the neighbors of a given k-set *on-the-fly*, we can search for designs that are not reachable with our basic approach, because of the space demands. The only big array that we still need in this approach is array `covered` and the space needed by it is shown in (8). Although (8) is $\mathcal{O}(e^v)$, it is usually much smaller than the sizes of tables `coverings` and `neighbors`. We can push the limit set by space demands a bit forward.

6 Comparisons between local optimization and SA

Since very little has been published on computer search for covering designs, it is difficult to find methods to compare SA with and to prove its merits. In the following we have chosen to compare the performance of SA to that of local optimization (Section 3.1). Local optimization is a natural approach to try to solve optimization of this type; although the comparison is not exhaustive, we hope that it gives an indication of how SA work for different parameter sets of the problem.

There were three sets of parameters that we used when comparing simulated annealing to local optimization in the search for covering desings:

1. $5-(11,5,7,1)$, 35 blocks;
2. $2-(12,3,4,1)$, 6 blocks;
3. $5-(12,5,7,1)$, 60 blocks.

Designs of type 2 are easy ones (partly because the solution space is much smaller than in the other two cases), designs of type 3 are the most difficult ones and designs of type 1 are intermediate.

We first tried to search for these designs using local optimization. The only parameter of local optimization in our implementation is the number of random neighbors of the current solution tried before the current solution is taken to be a local optimum. We increased the value of this parameter until no considerable increase in the percentage of successful searches was discovered. With the help of the results obtained by these tests we performed such local optimization runs that the average iterations needed to find one design was as small as possible.

Then we tried to find covering designs using simulated annealing. We used the number of iterations as the measure instead of CPU time, because the iterations in SA and local optimization were similar. The average CPU time is listed for convenience. Local optimization has some slight overhead in our implementation, so the CPU times are not exactly proportional to iterations within a design, as it should be.

It turned out that local optimization is effective in finding only covering designs of type 2. Not many designs of type 1 were found. None of the designs of type 3 were found by local optimization.

SA was more effective in the search for covering desings of types 1 and 3. For designs of type 2 it seems that SA does not increase the efficiency of the search. The nearer we came to local optimization the more efficient the search was. Local optimization works reasonably well, a covering design was found once in approximately 200 runs (one run took only a couple of hundreds of a second of CPU time). Covering designs of type 3 are so difficult that SA found only one such design during the test run, which makes it difficult to estimate the average time for finding one such design.

The results are summarized in Table 4. There is not enough test material here to let us draw any convincing conclusions, but it seems obvious that local optimization is not a good enough optimization method when one has to find any design but the smallest ones. SA performs much better, but even with SA the CPU times grow rapidly as the solution space explodes—not a surprising result.

The fact that local optimization works very well for small values of m, k and t is in accordance with the results in [20], where a greedy algorithm that works well in the search for Steiner triple systems (STS) is presented. An STS is a design with $m = t = 2$, $k = 3$, and $\lambda = 1$ that is perfect: it attains the bound in (1). However, for $k > 3$ the same approach does not perform well.

	solutions		*iterations*		*average iterations*		*average CPU-time*	
design	local	SA	local	SA	local	SA	local	SA
1	20	98	390000	85000	20000	860	0.56	0.027
2	48	†	140	†	2.95	†	$6.3 \cdot 10^{-3}$	†
3	0	1	180000	150000	NaN	150000	NaN	4.7

Iteration counts are expressed in thousands. Average CPU times are in hours. Note that local optimization did not find any designs of type 3.

† In the search for designs of type 2, SA was most efficient when it approached local optimization. The performance was then similar, of course.

Table 4: Comparison results

7 Compiling and Using the Program `cover`

7.1 How to Get the Program

The source code of the program (the files are enumerated in the next subsection) is available via anonymous FTP server `saturn.hut.fi` in directory `pub/cover`. The program can be freely used whenever the following stipulations are followed:

- The program may be used for research only. No commercial use is allowed.
- Whenever the program has been used to obtain results that are published, a reference should be made to this work.
- Bugs in the program should be reported to `Kari.Nurmela@hut.fi` or `Patric.Ostergard@hut.fi`. We also appreciate if improvements and new features are sent to the same e-mail addresses.

7.2 Compiling the Program

The program is written in ANSI C and it should be easily portable to various environments. A makefile is supplied for automated compilation. A couple of things should be noted, however.

The CPU-time the program has used is recorded by system call `getrusage()`. You may have to change this call in `cover.c`, if you want to compile the program in a non-UNIX environment.

The program uses functions `srandom` and `random` as the random number generator. If your system does not have these functions, then you have to rewrite the macros `randomize`, `rnd` and `random01` in `cover.h`. If your system offers several different random number generators, use the best.

The most important data types are defined in `cover.h`. You may wish to change some of them.

The program modules are:

`cover.c`	The main program.
`bincoef.c`	Functions needed to calculate and store binomial coefficients.
`tables.c`	The functions in this file allocate, compute and deallocate the tables in Figure 2.
`setoper.c`	Functions for manipulation of subsets.
`anneal.c`	The simulated annealing algorithm.
`solcheck.c`	This file contains some functions that can be used for debugging purposes.
`exp.c`	Approximate exponentiation.
`arg.c`	Command line option parser.
`readme`	Contains updated information about the program.

7.3 Using the Program

The program is run with the command "`cover [options]`" where options are of the format

`variable=value`

The valid variables and their meanings are

`k`, `t`, `m`, `v`, `l` These variables are the parameters of the design. The default value for `l` (λ) is 1, and if only one of the values `m` and `t` are given it is assumed that $m = t$.

`b` This variable is the number of k-sets (blocks) in the design.

`TestCount` This variable determines, how many annealing runs for the given `b` are accomplished. Variable `TestCount` can be abbreviated '`TC`'. Default 1.

`CoolingFactor` This variable is the value for r in (3). Abbreviation '`CF`'. Default 0.99.

`InitProb` This variable is the value of χ in (11). Abbreviation '`IP`'. Default 0.5.

`frozen` This variable denotes the required number of non-cost-decreasing successive temperatures to indicate that the process if frozen. See Section 4.7. Default 10.

`RestrNeighbor` Set the value of this variable to 1, if you want to use the restricted next solution selection mechanism described in Section 4.2. Abbreviation '`RN`'.

`InitTemp` If you want to give the initial temperature directly instead of giving `InitProb`, then use this variable. Abbreviation '`IT`'.

`L` If you want to give different value for L in the simulated annealing algorithm as described in Section 4.6, then use this variable. See also `LFact` below.

`LFact` This is the primary way of giving the permutation count L. If you do not give the value of L directly (via the `L` parameter), the value of L is calculated by multiplying the neighborhood size by `LFact`. The default value for `LFact` is 1. Abbreviation '`LF`'.

`EndLimit` Sometimes you may want to stop the annealing even before all sets are covered. You can give an limit for an acceptable solution with this variable. Abbreviation '`EL`'.

`local` Set this variable to 1, if you want to perform a local optimization process instead of simulated annealing. Use `L` or `LFact` to determine the number of successive unsuccessful tries that end the search.

`apprexp` If you want to use approximate exponentiation, then set this variable to 1.

`OntheFly` Set this variable to 1, if you want the program not to allocate tables `coverings` and `neighbors`. Abbreviation '`OF`'.

`Pack` If you set this variable to 1, the program searches for packing designs instead of covering designs. Abbreviation '`P`'.

`log` In addition to the output produced to `stdout`, the program writes the parameters, the cost of the best solution found, and the CPU time used to a log file. You can specify the name of the log file by the variable `log`. Default '`cover.log`'.

`result` If the program finds a design satisfying the `EndLimit` requirement, it stores the solution in a result file. You can give the result file a name with this variable. Default '`cover.res`'. The file is overwritten every time a solution is found, so it will contain only the last solution found.

`SolX` Set this variable to 1, if you want the solutions be printed as '`--X-XX`' instead of '`2 4 5`'. Abbreviation '`SX`'.

`verbose` The value of this variable (0, 1, or 2) determines the amount of data presented to the user. Default 2.

Example 2 The command

```
cover t=2 k=3 m=2 v=7 b=7 TC=2
```

tries to find covering design $1-(7,2,3,2)$ with 7 k-sets (which is also the value of $C(7,2,3,2)$). Two annealing runs are accomplished.

8 Constructing Packing Designs and t-Designs

Assume that $\lambda = 1$. Then a design is a $t-(v,m,k,1)$ packing design exactly when for any two blocks there are at least $d = 2k - 4t + 2m + 2$ points that belong to exactly one of the blocks (called the distance between the blocks). The blocks can thus be considered as codewords of a *constant weight code* with minimum distance d. Probabilistic methods such as simulated annealing have successfully been used to construct such codes (designs) [6, 9].

As can be seen in the formula for d, a $t-(v,m,k,1)$ packing design is also a $(t+t')-(v,m+2t',k,1)$ design ($t \in \mathcal{Z}$, such that $t+t' \leq m+2t'$). We can thus restrict our discussion to the case $t = m$. Now each k-set covers $\binom{k}{t}$ t-sets. Then, if the number of uncovered t-sets is $\binom{v}{t} - b\binom{k}{t}$, no t-set can be covered more than once (still, $\lambda = 1$), so we have a packing. Our program can now be used to find such a packing by using the feature that the search can be stopped before all sets are covered (`EndLimit`). Also by setting `Pack` to 1, our program automatically searches for packing designs. If $\lambda > 1$ there is no straightforward way to find packings by considering distances between blocks, however, we can use the set covering approach that is a generalization of the approach in the case $\lambda = 1$.

As t-designs are simultaneously both packing and covering designs, the program can be used directly in the search for them. However, we can do somewhat better by making use of the fact that each point in a t-design occurs in r blocks, where r can be calculated from the parameters of the design. This feature has not been implemented in our program; see [11] for a discussion of how to search for t-designs by simulated annealing.

Construction methods for packing designs and t-desings will be further evaluated in forthcoming studies.

9 Summary and Conclusions

The usage of the simulated annealing algorithm in searches for covering designs has been presented. The cost function in the algorithm was based on the number of uncovered sets. We also tried to increase the performance of the algorithm by considering a slightly more complex cost function, by finding low-cost initial solutions and by restricting the selection of the next solution. None of these approaches outperformed the simplest version of the algorithm (neither did they do much worse), which partly reflects the flexibility and robustness of SA.

Our implementation relies on extensive tabulation of data about covered sets and neighbors. The space complexity of the basic version of the algorithm usually sets the limits of the parameter range for the designs, although with suitable parameter values (for example when b is large) the time demand takes over. We reduced the space complexity by including the possibility of computing the covered sets and neighbors on-the-fly.

The performance tests show that SA performs very well. Local optimization works well for some sets of parameters, but it is generally outperformed by SA. The possibility of using probabilistic methods to find designs and to develop construction methods out of these was mentioned in the Introduction. We can also go in the other direction and evaluate proposed construction methods. For example, in [5] a method is introduced that is used to show that $C_1(15,3,6,3) \leq 45$. We were able to prove that $C_1(15,3,6,3) \leq 31$ by finding a corresponding covering design. The search for this design took only approximately two minutes of CPU time on a SUN SPARCstation 2. This indicates that the method presented in [5] does not work very well, at least not for these parameters.

Although SA works well in the search for designs, there is a point when the space and/or time requirements make the usage of the program intractable or even impossible. This limit could be pushed, e.g., by imposing some structure on the design we search for. However, such considerations are not within the scope of this work.

We finally hope that our program `cover` will be of use for many researchers in design theory.

References

[1] E. Aarts and J. Korst, "Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing," Wiley, Chichester, 1989.

[2] J. A. Bate and R. G. Stanton, Some algorithmic results on $(2,3,3,V)$ designs and $(2,4,4,V)$ designs, *Utilitas Math.* **20** (1981), 195–220.

[3] J. A. Bate and R. G. Stanton, A survey of $(2,2,L,V)$ designs, *Congr. Numer.* **31** (1981), 3–15.

[4] V. Černy, Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm, *J. Opt. Theory Appl.* **45** (1985), 41–51.

[5] A. H. Chan and R. A. Games, (n,k,t)-covering systems and error-trapping decoding, *IEEE Trans. Inform. Theory* **27** (1981), 643–646.

[6] A. A. El Gamal, L. A. Hemachandra, I. Shperling, and V. K. Wei, Using simulated annealing to design good codes, *IEEE Trans. Inform. Theory* **33** (1987), 116–123.

[7] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, Optimization by simulated annealing: An experimental evaluation; Part I, Graph partitioning, *Oper. Res.* **37** (1989), 865–892.

[8] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, Optimization by simulated annealing, *Science* **220** (1983), 671–680.

[9] K.-U. Koschnick, Some new constant weight codes, *IEEE Trans. Inform. Theory* **37** (1991), 370–371.

[10] P. J. M. van Laarhoven and E. H. L. Aarts, "Simulated Annealing: Theory and Applications," Reidel, Dordrecht, 1987.

[11] R. Mathon, Computational Methods in Design Theory, in "Surveys in Combinatorics," Proc. 13th Br. Comb. Conf., Guildford/UK 1991, LMSLNS 166, London, 1991, pp. 101–117.

[12] R. Mathon, Nonisomorphic designs by simulated annealing, submitted for publication.

[13] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, *J. Chem. Phys.* **21** (1953), 1087–1092.

[14] W. H. Mills and R. C. Mullin, Coverings and packings, in J. H. Dinitz and D. R. Stinson (Eds.), "Contemporary Design Theory: A Collection of Surveys," Wiley, New York, 1992, pp. 371–399.

[15] M. Morley and G. H. J. van Rees, Lottery schemes and covers, *Utilitas Math.* **37** (1990), 159–165.

[16] W. Oberschelp, Lotto-Garantiesysteme und Blockpläne, *Math.-Phys. Semesterber.* **19** (1972), 55–67.

[17] J. Schönheim, On coverings, *Pacific J. Math.* **14** (1964), 1405–1411.

[18] D. W. Stanton and D. E. White, "Constructive combinatorics," Springer-Verlag, New York, 1986.

[19] R. G. Stanton and J. A. Bate, A computer search for B-coverings, R. W. Robinson, G. W. Southern, and W. D. Wallis, (Eds.), "Combinatorial Mathematics VII," LNCS 829, Springer-Verlag, New York, 1980, pp. 37–50.

[20] D. R. Stinson, Hill-climbing algorithms for the construction of combinatorial designs, *Ann. Discrete Math.* **26** (1985), 321–334.