HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports ISSN 0783–5396
No. **27**; November 1993 ISBN 951-22-1857-7

CONSTRUCTING COMBINATORIAL DESIGNS BY LOCAL SEARCH

Kari J. Nurmela

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 **A**: Research Reports
No. **27**; November 1993

ISSN 0783–5396
ISBN 951-22-1857-7

Constructing Combinatorial Designs by Local Search

Kari J. Nurmela

Abstract: In this work probabilistic search heuristics are used in search for covering designs, packing designs and t-designs. Each covering design gives an upper bound for $C_\lambda(v, m, k, t)$, the minimum size of a $t - (v, m, k, \lambda)$ covering design, while each packing design gives a lower bound for $D_\lambda(v, m, k, t)$, the maximum size of a $t - (v, m, k, \lambda)$ packing design, and each t-design establishes the existence of such a design. The designs are constructed using probabilistic combinatorial optimization methods including simulated annealing, tabu search, threshold accepting, great deluge algorithm and record-to-record travel. Implementation issues of these algorithms, such as cost functions and neighborhood structures are discussed. Comparisons of the performance of these algorithms on some designs is carried out. Some elementary search methods including steepest descent algorithm and random walk are included in the comparisons. Finally, the results of the searches are tabulated. The object class hierarchy of the program (written in C++) and brief class descriptions are presented in the appendix.

Keywords: Combinatorial designs, combinatorial optimization, great deluge algorithm, record-to-record travel.

Printing: TKK Monistamo; Otaniemi 1993

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

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

Contents

1 Introduction

There are a lot of important and difficult combinatorial problems that do not have simple closed form analytic solutions. These problems range from theoretical (e.g. determining the values of classical Ramsey numbers or partitioning a graph) to very practical (e.g. optimization of an operational timetable or VLSI design). Solving these problems with usual deterministic algorithms is often hopeless because of the combinatorial explosion which makes it impossible to examine all the possible solutions. During the last few years a new generation of algorithms has emerged. These algorithms use different kinds of local search heuristics and they have had impressive practical success in many NP-complete problems. Due to their probabilistic nature, it is often impossible to say whether the solution given by them is the best possible, but sometimes a solution that is good enough suffices.

In this work we use different probabilistic combinatorial optimization heuristics to find covering designs, packing designs and t-designs. The optimization heuristics used include various iterative improvement algorithms, simulated annealing, tabu search, threshold accepting, the great deluge algorithm and record-to-record travel.

We are interested in finding as small covering designs and as large packing designs as possible. The problem of finding the exact sizes of minimal covering designs and maximal packing designs 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 [47]. Due to the combinatorial explosion even computers are not of much help in the general case, so for most parameters only upper and lower bounds on these sizes are known. Upper bounds on the size of a minimal covering design and lower bounds on the size of a maximal packing design are obtained by explicitly constructing corresponding covering and packing designs. The existence of certain types of t-designs is in many cases an open problem. The existence of a specific t-design may be proved by constructing the design by a computer search. In this work we view the problem of finding a certain design as a combinatorial optimization problem. We define a cost function that is to be minimized. If the cost can be pushed to a global optimum value, then we have the required design.

We also hope that this work will bring us some more knowledge of the performance of the optimization methods mentioned above. Little is known about the performance of these algorithms in general, thus we have to settle on empirical data.

This work is organized in the following way. In Section 2 we give the formal definitions of covering designs, packing designs and t-designs and review some of the known results on them. We also present a new connection between covering and packing designs. In Section 3 we introduce the combinatorial optimization algorithms used. In Section 4 we go into the implementation of these algorithms and consider the implementation options available. In Section 5 we compare the performance of the algorithms in some test cases. In Section 6 we summarize some of the previously known bounds on $C_\lambda(v, m, k, t)$ and $D_\lambda(v, m, k, t)$ and give the new bounds found during this work. In that section we give the results of the t-design searches, too. Finally, in Section 7 we conclude this work. The covering designs giving the upper bounds in Section 6, the object class hierarchy of the program and brief class descriptions are given in the appendixes.

2 Definitions

2.1 Covering Designs

Definition 1 *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. Let the minimum size of a covering design $t-(v,m,k,\lambda)$ be*

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

The index λ in $C_\lambda(v,m,k,t)$ can be omitted if it is 1. We have used the term k-subset of X to denote any set A such that $A \subseteq X$ and $|A| = k$. A m-subset of X is said to *be covered* by a k-subset of X in t points, if the intersection of these two subsets has at least t points. The complement of any subset A of X is defined as usually, $A^* = X \setminus A$.

Definition 2 *Two covering designs $(X, \mathcal{A})$ and $(X', \mathcal{A}')$ are said to be* isomorphic *if there exists a bijection between X and X' such that it maps $\mathcal{A}$ to $\mathcal{A}'$ (the inverse of the bijection then maps $\mathcal{A}'$ to $\mathcal{A}$).*

The isomorphism between two packing designs (Section 2.2) or two t-designs (Section 2.3) is defined in a similar way.

A *binary constant-weight code* is a binary code whose codewords all have the same number of 1's. A $t-(v,m,k,\lambda)$ covering design can be seen as a binary constant weight covering code of length v and weight k such that any binary word of length v and weight m is within Hamming distance $m + k - 2t$ from at least λ codewords.

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 [37] for an excellent survey). Some results on the case $m > t$ exist [5, 6, 47]; in those papers $\lambda = 1$, and the notation $B(t,k,m,v) = C(v,m,k,t)$ is used.

We denote the points in X by the numbers $1, 2, \ldots, v$. The blocks can be denoted by listing the points contained by it (for example '1 4 5 10 12' would be a block with 5 points) or by using a cross-dash notation (`X--XX----X-X`). As covering designs are binary constant weight covering codes, they can also be denoted by binary strings (100110000101).

Trivially, a covering design $t-(v,m,k,\lambda)$ can be formed by taking one block for each possible m-subset of X (there are $\binom{v}{m}$ of them), so we are mainly interested in the value of $C_\lambda(v,m,k,t)$, the minimum number of blocks needed to form the design.

There are several known lower bounds on $C_\lambda(v,m,k,t)$. One of them is the volume bound.

Theorem 1

$$C_\lambda(v,m,k,t) \geq \lambda \frac{\binom{v}{m}}{\sum_{i=t}^{\min(k,m)} \binom{k}{i}\binom{v-k}{m-i}}.$$

Proof: The total number of m-subsets 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-subsets; 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. □

Theorem 2 *If we have a $t-(v,m,k,\lambda)$ covering design, we can always form a $(v-k-m+t)-(v,v-m,v-k,\lambda)$ covering design by complementing the blocks. Thus*

$$C_\lambda(v,m,k,t) = C_\lambda(v,v-m,v-k,v-k-m+t).$$

Proof: Note that complementing a k-subset is a bijective mapping between all the k-subsets and all the $(v-k)$-subsets. An m-subset is covered by a k-subset in t points *if and only if* its complement is covered by the complement of the k-subset in $v-k-m+t$ points. That is, if all the m-subsets are covered by at least λ blocks in at least t points, then all the $(v-m)$-subsets are covered in at least $v-k-m+t$ points by at least λ complements of the blocks. This theorem and its proof are a generalization of Lemmas 2.1 and 2.2 in [47]. □

The following theorem can be used in the case $m=t$.

Theorem 3 ([43])

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

An analogous result for $k = t$ is easily obtained.

Corollary 1

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

Proof: We use Theorems 2 and 3. Let $k = t$.

$$\begin{aligned}
C_\lambda(v, m, t, t) &= C_\lambda(v, v-m, v-t, v-t-m+t) \\
&\geq \left\lceil \frac{v}{v-t} C_\lambda(v-1, v-m-1, v-t-1, v-m-1) \right\rceil \\
&= \left\lceil \frac{v}{v-t} C_\lambda(v-1, m, t, t) \right\rceil .
\end{aligned}$$

□

By iterating the inequalities in Theorem 3 and Corollary 1 and noting that

$$\begin{aligned}
C_\lambda(v, 0, k, 0) &= \lambda \\
C_\lambda(m, m, t, t) &= \lambda
\end{aligned}$$

one obtains

Corollary 2

$$\begin{aligned}
C_\lambda(v, t, k, t) &\geq \left\lceil \frac{v}{k} \left\lceil \frac{v-1}{k-1} \cdots \left\lceil \frac{v-t+1}{k-t+1} \lambda \right\rceil \cdots \right\rceil \right\rceil = L_1(v, k, t, \lambda) \\
C_\lambda(v, m, t, t) &\geq \left\lceil \frac{v}{v-t} \left\lceil \frac{v-1}{v-t-1} \cdots \left\lceil \frac{m+1}{m-t+1} \lambda \right\rceil \cdots \right\rceil \right\rceil \\
&= L_2(v, m, t, \lambda).
\end{aligned}$$

We call the values of $L_1(v, k, t, \lambda)$ and $L_2(v, m, t, \lambda)$ Schönheim lower bounds on the sizes of covering designs. It should be noted that if we can improve any lower bound in the iteration chain, then the resulting lower bound may become better, too.

Further, we can give some elementary results [47], in some of which we require that $\lambda = 1$. Sometimes a single block will form a covering design.

$$C(v, m, k, t) = 1 \textit{ if and only if } v + t \leq k + m.$$

On the other hand, if $t = k = m$, we have a degenerate case and λ copies of each k-subset are needed, hence

$$C_\lambda(v, t, t, t) = \lambda \binom{v}{t}.$$

Further, it can easily be proven that in a minimal covering design with $\lambda = 1$ either every point is contained in some block or no point is contained in two distinct blocks. If $C(v, m, k, t) \geq v/k$ then there exists a minimal covering design containing every point. Otherwise, if $C(v, m, k, t) < v/k$, then there exists a minimal covering design consisting of distinct points, i.e. no point is contained in more than one block. In some cases the following equation gives the value of $C(v, m, k, t)$.

$$\text{If } \frac{v-m}{k-t+1} < \left\lfloor \frac{v}{k} \right\rfloor, \text{ then } C(v, m, k, t) = \left\lceil \frac{v-m+1}{k-t+1} \right\rceil. \tag{1}$$

Sometimes the following upper bound can be used.

$$\text{If } n \geq \frac{v}{k} \text{ and } m > (t-1)n, \text{ then } C(v, m, k, t) \leq n. \tag{2}$$

In Section 6 we give a few more values of $C_\lambda(v, m, k, t)$ and tabulate some distinct known values of $C_\lambda(v, m, k, t)$.

2.1.1 Applications of Covering Designs

In discussions of applications of covering designs, three fields are generally mentioned: lottery schemes (see, e.g., [38, 41]), error-trapping decoding (see [12]) and data compression (see [19]).

For example in Finnish lotto (as today) 7 numbers out of 39 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. The price depends on the number of the rows in the system, so minimal or near-minimal covering designs are cost-effective in this sense.

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.

Covering designs are often needed to achieve optimal compression performance in universal quantizers. Covering designs form the basic building blocks of a universal quantizer for binary sources [19].

2.2 Packing Designs

Packing designs are defined in a way analogous to the definition of covering designs.

Definition 3 *Let $v \geq k \geq t$ and $v \geq m \geq t$. A $t-(v,m,k,\lambda)$* packing 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 most *λ members of $\mathcal{A}$ in t or more points. Let the maximum size of a $t-(v,m,k,\lambda)$ packing design be*

$$D_\lambda(v,m,k,t) = \max\{|\mathcal{A}| : \textit{there is a } t-(v,m,k,\lambda) \textit{ packing design } (X,\mathcal{A})\}.$$

As with covering designs, the index λ in $D_\lambda(v,m,k,t)$ can be omitted if it is 1. A $t-(v,m,k,\lambda)$ packing design can be seen as a binary constant weight code such that any binary word of length v and weight m is within Hamming distance $m+k-2t$ from at most λ codewords.

Trivially, a packing design with $b \leq \lambda$ can always be constructed, so we are mainly interested the maximum size of various packing designs. The simplest upper bound on the size of a packing design is the volume bound analogous to Theorem 1.

Theorem 4

$$D_\lambda(v,m,k,t) \leq \lambda \frac{\binom{v}{m}}{\sum_{i=t}^{min(k,m)} \binom{k}{i}\binom{v-k}{m-i}}.$$

Proof: Analogous to the proof of Theorem 1. $\square$

An analogous result to Theorem 2 is readily obtained.

Theorem 5 *If we have a $t-(v,m,k,\lambda)$ packing design, then a $(v-k-m+t)-(v,v-m,v-k,\lambda)$ packing design can be formed by complementing the blocks. It follows that*

$$D_\lambda(v,m,k,t) = D_\lambda(v,v-m,v-k,v-k-m+t).$$

Proof: Analogous to that of Theorem 2. □

As with covering designs, most existing literature concerns the case $m = t$ or $\lambda = 1$. A well-known upper bound for $D_\lambda(v, m, k, t)$ in the case $m = t$ is Schönheim's upper bound.

Theorem 6 ([44])

$$D_\lambda(v,t,k,t) \le \left\lfloor \frac{v}{k} D_\lambda(v-1, t-1, k-1, t-1) \right\rfloor .$$

Again, the theorem is easily expanded to consider the case $k = t$.

Corollary 3

$$D_\lambda(v,m,t,t) \le \left\lfloor \frac{v}{v-t} D_\lambda(v-1, m, t, t) \right\rfloor .$$

Proof: We use Theorems 5 and 6. Let $k = t$.

$$\begin{aligned} D_\lambda(v,m,t,t) &= D_\lambda(v, v-m, v-t, v-t-m+t) \\ &\le \left\lfloor \frac{v}{v-t} D_\lambda(v-1, v-m-1, v-t-1, v-m-1) \right\rfloor \\ &= \left\lfloor \frac{v}{v-t} D_\lambda(v-1, m, t, t) \right\rfloor . \end{aligned}$$

□

It is clear that

$$\begin{aligned} D_\lambda(v,0,k,0) &= \lambda \\ D_\lambda(v,v,t,t) &= \lambda, \end{aligned}$$

and thus by iterating the inequalities in Theorem 6 and Corollary 3 we obtain

Corollary 4

$$\begin{aligned} D_\lambda(v,t,k,t) &\le \left\lfloor \frac{v}{k} \left\lfloor \frac{v-1}{k-1} \cdots \left\lfloor \frac{v-t+1}{k-t+1} \lambda \right\rfloor \cdots \right\rfloor \right\rfloor = U_1(v,k,t,\lambda) \\ D_\lambda(v,m,t,t) &\le \left\lfloor \frac{v}{v-t} \left\lfloor \frac{v-1}{v-t-1} \cdots \left\lfloor \frac{m+1}{m-t+1} \lambda \right\rfloor \cdots \right\rfloor \right\rfloor \\ &= U_2(v,m,t,\lambda). \end{aligned}$$

We call $U_1(v,k,t,\lambda)$ and $U_2(v,m,t,\lambda)$ the Schönheim upper bounds for packing designs. The following theorem is an elementary result on constant weight error-correcting codes.

Theorem 7 *Let $v > 2k - t$ and $m = t$. Then*

$$D_1(v,t,k,t) = D_1(v, v-2k+t, v-k, v-2k+t).$$

Proof: This proof is from [37]. Let there be a $t-(v,t,k,1)$ packing design $(X,\mathcal{A})$. Because $\lambda = 1$ and $m = t$, the packing condition is equivalent to the condition $|a_1 \cap a_2| < t$ for all distinct blocks $a_1 \in \mathcal{A}$ and $a_2 \in \mathcal{A}$. Now this is equivalent to the condition $|a_1^* \cap b_1^*| < v - 2k + t$, where a^* is the set complement in X, which, in turn, is equivalent to the definition of a $(v-2k+t)-(v, v-2k+t, v-k, 1)$ packing design. That is, the blocks of the new packing design are obtained from the blocks of the original packing design by complementing them in X. □

Another upper bound for packing designs can be obtained by repeatedly applying Theorems 6 and 7. The best bound obtained in this way is called the first Johnson bound [26] and it is sometimes better than the Schönheim bound.

The minimum distance between any two blocks in a $t-(v,m,k,1)$ packing design is

$$d_{\min} = 2\,(k - 2t + m + 1)\,. \tag{3}$$

Note that this minimum distance is always even, as is the case with constant weight codes. It can be seen from (3) that any $t-(v,m,k,1)$ packing design is also a $(t+t')-(v, m+2t', k, 1)$ packing design, where $t' \in \mathbb{Z}$ such that $1 \le t + t' \le m + 2t' \le v$. Thus, any packing design with $\lambda = 1$ can be seen as a packing design with $m = t$.

There is an extensive paper [8] on constant weight error-correcting codes, where they give explicit constructions of several hundred new codes. These codes are packing designs with $\lambda = 1$ and thus we refer to the results in that paper. See also Section 2.2.1.

The notation we use allows us to give two new theorems about the connection between covering and packing designs.

Theorem 8 *Let there be a $t-(v,m,k,\lambda)$ covering design with b blocks. Then there exists a $(m-t+1)-(v,m,v-k,b-\lambda)$ packing design with the same number of blocks.*

Proof: Assume that we have a $t-(v,m,k,\lambda)$ covering design $(X,\mathcal{A})$ with b blocks. Now a $(m-t+1)-(v,m,v-k,b-\lambda)$ packing design can be constructed by complementing the blocks of the covering design, because of the following: Let an m-subset M be covered in t points by λ or more k-subsets. Denote the multiset of these k-subsets by S_1. It is then true that

$$|A \cap M| \geq t \text{ for all } A \in S_1,$$

which is equivalent to

$$|A^* \cap M| \leq m-t \text{ for all } A \in S_1,$$

which implies

$$|A^* \cap M| < m-t+1 \text{ for all } A \in S_1,$$

where A^* is the complement of A in X. That is, the m-subset is *not* covered in $m-t+1$ points by λ or more complements of the blocks of the covering design. Because the same is true for all m-subsets of X, the complements of the blocks of the covering design form a $(m-t+1)-(v,m,v-k,b-\lambda)$ packing design. □

Theorem 9 *Let there be a $t-(v,m,k,\lambda)$ packing design with b blocks. Then there exists a $(m-t+1)-(v,m,v-k,b-\lambda)$ covering design with the same number of blocks.*

Proof: Assume that we have a $t-(v,m,k,\lambda)$ packing design with b blocks. Let an m-subset M be covered in t points by λ or less k-subsets. Denote the complement (in $\mathcal{A}$) of these k-subsets by S_2. It is then true that $|S_2| \geq b-\lambda$ and

$$|A \cap M| < t \text{ for all } A \in S_2,$$

which is equivalent to

$$|A^* \cap M| > m-t \text{ for all } A \in S_2,$$

which implies

$$|A^* \cap M| \geq m-t+1 \text{ for all } A \in S_2.$$

That is, the m-subset is covered in $m-t+1$ points by at least $b-\lambda$ complements of the blocks of the packing design. Because the same is true for all the m-subsets of X, the complements of the blocks of the packing design form a $(m-t+1)-(v,m,v-k,b-\lambda)$ covering design. □

Corollary 5

$$\begin{aligned} D_{C_\lambda(v,m,k,t)-\lambda}(v,m,v-k,m-t+1) &\geq C_\lambda(v,m,k,t) \\ C_{D_\lambda(v,m,k,t)-\lambda}(v,m,v-k,m-t+1) &\leq D_\lambda(v,m,k,t). \end{aligned}$$

2.2.1 Applications of Packing Designs

Perhaps the most important application of packing designs is constant weight error-correcting codes. Given a $t-(v,m,k,\lambda)$ packing design with $\lambda = 1$, it can be seen as a code of length k having b codewords. The minimum Hamming distance between any two codewords (blocks) is given by (3) and thus it is possible to decode any erroneously received word correctly if it has at most $d_{\min}/2-1$ erroneous bits. It is desirable to construct designs having as many blocks as possible, because then we have more codewords in our code while still maintaining the same error-correction capability.

2.3 t-designs

If a $t-(v,t,k,\lambda)$ covering design is also a $t-(v,t,k,\lambda)$ packing design, it is called a t-design. We use t, v, k, λ with the same meaning as above ($m = t$).

Definition 4 *Let $v \geq k \geq t$. A $t-(v,k,\lambda)$* 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 t-subset of X is contained in* exactly *λ members of $\mathcal{A}$.*

Sometimes a t-design is denoted $S_\lambda(t,k,v)$. A t-design with $t = 2$ is called a balanced incomplete block design, or BIBD for short. We use the notation (v,k,λ)-BIBD for a BIBD with parameters v, k, λ. A BIBD with $b = v$ is called a *symmetric* BIBD. A *Steiner triple system* STS(v) is a $(v,3,1)$-BIBD.

If we denote the number of blocks by b and let the number of blocks in which a point occurs be r (it is clear that every point must occur in the same number of blocks), then simple counting arguments give the following two relations that are necessary but not sufficient for the existence of a t-design with these parameters:

$$vr = bk \tag{4}$$

$$\lambda \binom{v}{t} = b \binom{k}{t} \tag{5}$$

A t-design with $t = 1$ exists trivially with all the parameter settings that satisfy (4) and (5), see Figure 12 on page 41. Quite lot is known about the case $t = 2$, but when t grows, fewer and fewer results are available.

BIBDs with $k = 2$ or $k = v$ exist trivially, so we restrict our attention to the cases where $2 < k < v$. An STS(v) is known to exist for all $v \equiv 1$ or 3 (mod 6). In the cases $k = 3$, 4 or 5, all the BIBDs satisfying conditions 4 and 5 exist, with the exception that no (15,5,2)-BIBD exists.

One of the simplest necessary conditions for the existence of BIBDs is the *Fisher's inequality*, which states that a BIBD can exist only if $b \geq v$. A generalization of this condition says that if a $t-(v,k,\lambda)$ design exists and $t = 2s$ is even, then the number of blocks $b \geq \binom{v}{s}$. Further, with symmetric BIBDs we can use the following theorem.

Theorem 10 (Bruck-Ryser-Chowla) *If a symmetric (v,k,λ)-BIBD exists, then:*

1. *If v is even, $k-\lambda$ is a perfect square.*
2. *If v is odd, the Diophantine equation $x^2 = (k-\lambda)y^2 + (-1)^{(v-1)/2}\lambda z^2$ has an integer solution, where at least one of x, y, z is nonzero.*

It should be noted that if a $t-(v,k,\lambda)$ design exists, then the Schönheim lower bound $L_1(v,k,t,\lambda)$ (Corollary 2) for covering designs and the Schönheim upper bound $U_1(v,k,t,\lambda)$ (Corollary 4) for packing designs coincide.

An important existence result for t-designs is Teirlinck's theorem, which states that t-designs exist for arbitrarily large t. Note, however, that the λ-values of these designs are extremely large.

Theorem 11 ([49]) *A $t-(v,t+1,((t+1)!)^{2t+1})$ design exists if $v \geq t+1$ and $v \equiv t \bmod ((t+1)!)^{2t+1}$.*

If $\lambda = 1$, examples of t-designs are known only up to $t \leq 5$. Construction of a $6-(v,k,1)$ design remains an important open question in the study of t-designs. The following theorem shows how we can determine some exact values of $C(v,t,k,t)$ or $D(v,t,k,t)$ if a suitable t-design with $\lambda = 1$ exists.

Theorem 12 ([44]) *If a $t-(v,k,1)$ design exists, then*

$$C(v+1,t,k,t) = L_1(v+1,k,t),$$

where $L_1(v,k,t)$ is the Schönheim lower bound on the size of minimum covering design (Corollary 2). Moreover, if $v > k$, then

$$D(v-1,t,k,t) = U_1(v-1,k,t),$$

where $U_1(v,k,t)$ is the Schönheim upper bound on the size of maximum packing design (Corollary 4).

2.3.1 Applications of t-designs

One of the first applications of t-designs was experiment planning. For example, suppose that there are v different coffee brands and b test persons, who each taste k brands and rank them according to their fancy. Because the sorting of objects according to some criterion can be seen as a repeated comparison of two objects, we would like that any *pair* of two coffee brands is tasted by the same number λ of test persons. We can arrange this by using a (v, k, λ)-BIBD.

BIBDs have also connections to projective and affine planes, Hadamard matrices and Room squares. See [16] for further information and references.

3 Combinatorial Optimization by Local Search

3.1 Combinatorial Optimization Problems

In order to properly define the combinatorial optimization algorithms used we need to formalize the idea of a combinatorial optimization problem. We do it by defining a set of solutions, called solution space and a set of functions (called *moves*) that can be applied to solutions to get new, slightly different solutions.

Definition 5 *Let the set of* feasible solutions *of the problem be S_F and let the* solution space *be S. We require that S is finite. It is the case that $S_F \subseteq S$, but the two sets need not be equal. The objective of combinatorial optimization is to*

$$\text{Minimize } c(s) : s \in S,$$

where $c : S \to \mathbb{R}$ ($\mathbb{R}$ is the set of real numbers) is the cost function *and $c(s)$ gives the cost of the solution s. The set of possible costs is $\mathcal{C}$ and it is assumed that $\mathcal{C} \subset \mathbb{R}$.*

Note that a maximization problem is easily turned into a minimization problem by reversing the sign of the cost function c. In addition to the solution space, the algorithms described in Sections 3.2 through 3.8 make use of the concepts of moves, neighbors and neighborhoods.

Definition 6 *A* move *d is a function from a subset $S(d)$ (the* domain of the move*) of S to S:*

$$d : S(d) \to S.$$

The set of all the moves of the problem is $\mathcal{D}$. We require that the union of the domains of all the moves in $\mathcal{D}$ is the solution set

$$\bigcup_{d \in \mathcal{D}} S(d) = S,$$

so that there are no solutions where all the moves are unapplicable.

We denote the application of a move d to a solution s by $d(s)$, where it is assumed that $s \in S(d)$. If the *inverse* of a move d exists, it is denoted by d^{-1} and it satisfies

$$s = d^{-1}(d(s)) \text{ for any } s \in S(d).$$

Note that a move can be *potentially degenerate*, that is

$$s = d(s) \text{ for some } s \in S(d).$$

If a potentially degenerate move d satisfies

$$s = d(s) \text{ for all } s \in S(d),$$

then the move d is called *degenerate.*

Definition 7 *A solution s' is a* neighbor *of the solution s, if*

$$s' = d(s) \textit{ for some } d \in \mathcal{D}.$$

The neighborhood $N(s)$ *of a solution s is the union of all the neighbors of s:*

$$N(s) = \bigcup_{\{d \in \mathcal{D} | s \in S(d)\}} d(s).$$

If there are potentially degenerate moves in S, then—and only then—a solution can be a neighbor of itself.

Definition 8 *A solution s is a* global optimum, *if there is no other solution s' in the solution space S such that $c(s') < c(s)$. A solution s is a* (weak) local optimum, *if there is no other solution s' in the neighborhood $N(s)$ of s such that $c(s') < c(s)$. If all the neighbors of s have higher cost than $c(s)$, then s is a* strong local optimum.

If $S \neq S_F$, it is assumed that the global optima of S are in the set of the global optima of S_F and thus, if we succeed in finding a global optimum of S, it is guaranteed that it is a global optimum of S_F, too. However, the same does not have to apply for local optima.

The optimization algorithms described below (random walk, simulated annealing, tabu search, steepest descent algorithm, threshold accepting, great deluge algorithm and record-to-record travel) belong to a class of optimization algorithms called *local search algorithms* [2]. An algorithm belongs to this class, if it examines the solutions one solution a time, keeps a single solution in memory (call this solution the *current solution*) and each time the current solution is changed from s to s', it is true that $s' \in N(s)$. Note that in some papers the term local search algorithm is used in a different meaning to denote some of the iterative improvement algorithms in Section 3.2.4.

There are two difficult problems while implementing a local search algorithm.

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.

1. Set $s :=$ random solution in the solution space. Set $s' := s$.
2. While not the *stop criterion* do the following:
   (a) Select a solution s at random in the solution space S.
   (b) If $c(s) < c(s')$ let $s' := s$.
3. Return s'.

Figure 1: Random sampling.

Efficiency. Moving from a solution to one of its neighbors and the corresponding cost change calculations should be simple enough to perform, so that the algorithm can perform iterations fast.

3.2 Elementary Optimization Algorithms

In this section we review some elementary optimization algorithms and define the terminology with the help of the definitions in Section 3.1.

3.2.1 Total Enumeration of Solutions

If the solution space is sufficiently small, we can simply calculate the cost for every solution and return the solution with the smallest cost. However, the combinatorial explosion makes this approach generally unusable. With the solution spaces we have used even the designs with modest size parameters can have 10^{100} or more solutions.

3.2.2 Random Sampling

If we can not explore the whole solution space, we can try perhaps the simplest probabilistic search algorithm, random sampling. The algorithm works simply by selecting random solutions in the solution space and returning the best solution found. The algorithm is summarized in Figure 1. In the rest of this work, it is assumed that each algorithm keeps the best solution encountered during the optimization run in the memory, so that lines like 2(b) and 3 in Figure 1 will not be needed, instead we can just say in the end of the algorithm 'Return the best solution encountered.', as in Figure 2. The stop criterion in step 2 in Figure 1 can simply be a time limit—the optimization run is terminated after the limit is exceeded.

1. Let s be a random solution in the solution space.
2. While *stop criterion* not satisfied do the following:
   (a) Let s be a random solution in the neighborhood $N(s)$ of s.
3. Return the best solution encountered.

Figure 2: Random walking.

1. Select a random solution s from the solution space.
2. Let s' be a solution in $N(s)$ such that $c(s') = \min\{c(s'') \,|s'' \in N(s)\}$.
3. If $c(s') < c(s)$ let $s := s'$ and go to step 2. Otherwise, return s.

Figure 3: Steepest descent algorithm.

3.2.3 Random Walking

Random walking is like random sampling, except that we move from a solution to one of its neighbors instead of selecting a random solution from the whole solution space. See Figure 2. Random sampling can be seen as random walking by setting the neighborhood of a solution to the whole solution space.

3.2.4 Iterative Improvement Algorithms

There are several algorithms that move from the current solution to a new solution only if the new solution is at least as good as the current solution. In this work we call these algorithms *iterative improvement algorithms.*

Steepest descent algorithm is a simple optimization method that can be very efficient in some problems. In steepest descent algorithm the next solution is the most cost-decreasing neighbor of the current solution. If no cost-decreasing neighbors are found, the search is terminated. See Figure 3.

An important property of the steepest descent algorithm is that it always ends in a local optimum. While a local optimum does not have to be a global optimum, in some problems the probability of finding a global optimum in this way is relatively large, so that repeated steepest descent runs give a global optimum at least once with a high probability.

Notice that in step 2 in Figure 3 we have to examine the whole neighborhood of the current solution. In some problems, however, the information giving

1. Select a random solution s from the solution space.
2. Repeat until no improvement in a long time:
   (a) Let s' be a solution in $N(s)$.
   (b) If $c(s') \leq c(s)$, let $s := s'$.
3. Return s.

Figure 4: Local optimization with sideway moves.

the most cost-decreasing move is readily obtainable. Gradient-based searches are examples of this kind of an approach.

It is also possible to examine only a fraction LF of the neighborhood when selecting the next solution. In the performance comparisons this approach is called *probabilistic steepest descent algorithm*. The fraction is usually formed by selecting the wanted number of neighbors 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.

During a probabilistic steepest descent run there will always be a moment, when no cost-decreasing moves were found in $LF \cdot |N(s)|$ solutions. Thus, it is necessary then to perform a complete neighborhood examination so that it can be guaranteed that the search ends in a local optimum.

Another simple optimization algorithm is obtained, if we just examine random neighbors of the current solution and always when we find a better solution than the current one, move to that solution. When there has been more than a given number of steps without any improvement, we can perform a complete neighborhood check as in probabilistic steepest descent algorithm to be sure that the current solution is a local optimum. We call this algorithm *probabilistic local optimization* [25]. Be aware that the term local optimization is used in several different meanings in the literature.

In addition to the cost-decreasing moves, it is sometimes tempting to allow *sideway moves*, i.e. moves that do not affect the cost. However, some special bookkeeping is needed to prevent looping (see Figure 6 on page 23). If the possibility of looping is ignored and we continue the optimization until the cost is not lowered during a given number of trials, then we get a simple but sometimes very useful algorithm in Figure 4.

3.3 Simulated Annealing

Simulated annealing [11, 27] is an optimization method which has had impressive practical success in many NP-complete optimization problems. The simulated annealing algorithm can be viewed as an extension to probabilistic local optimization. If the proposed new solution is equal to or better than the current solution, it is accepted. If the proposed new solution is worse than the current solution, it is even then accepted with probability

$$P_{\text{accept}} = \exp\left(\frac{-\Delta}{T}\right) = \exp\left(-\frac{c(s') - c(s)}{T}\right), \tag{6}$$

where $c(s)$ denotes the cost function assigned to the current solution, $c(s')$ 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. In the beginning the value of T is relatively large so that many cost-increasing moves are accepted in addition to cost-decreasing moves. During the optimization process the temperature is decreased gradually so that fewer and fewer cost-increasing moves are accepted.

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, invented in the early days of computing [33]. 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

$$\exp\left(-\frac{E_j - E_i}{k_B T}\right), \tag{7}$$

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.

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.

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 solution s' such that $s' \in N(s)$.
      ii. Let $\Delta = c(s') - c(s)$.
      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 the best solution encountered.

Figure 5: Simulated annealing.

The simulated annealing algorithm 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 [25].

The simulated annealing algorithm is summarized in Figure 5. For a comprehensive treatment of the simulated annealing algorithm and its analogical physical counterpart, see [1, 11, 27, 30, 33].

The criterion for stopping the iteration (detecting when the process is 'frozen') in step 2 in Figure 5 varies in the literature. One possible approach [25] is to stop when the percentage of accepted moves is below a certain level for a given number of successive executions of steps 2(a) and 2(b), and no improvement of the best solution found so far is recorded. However, if there are many neighbors that have the same cost as the current solution, then the percentage of accepted moves can continuously be rather high even if the cost no more decreases. In our implementation of the simulated annealing algorithm we simply stop the process when the cost at the end of each temperature has not improved during a given number of successive temperatures.

A common inner loop criterion 2(a) 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 [25]. In practice, too high or low value of L can be compensated by lowering the temperature faster or slower, respectively.

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

$$T' = rT, \tag{8}$$

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, [25]), but this *geometric cooling schedule* seems to be not only very robust, but also one of the most efficient.

It can be shown that simulated annealing converges asymptotically to the set of globally optimal solutions if certain 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.

3.3.1 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 the problems in this work there is sometimes a relatively 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. This way we can always set the desired probability between 0 and 1 without the knowledge of the cost distribution around the initial solution.

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

$$n_{\text{acc}} \approx n \cdot \exp\left(\frac{-\overline{\Delta c_+}}{T}\right), \tag{9}$$

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

$$\chi = \frac{n_{\text{acc}}}{n} \approx \exp\left(\frac{-\overline{\Delta c_+}}{T}\right),$$

from which we obtain

$$T \approx \frac{-\overline{\Delta c_+}}{\ln \chi}. \tag{10}$$

Now we set χ to the desired initial probability and calculate an estimate for $\overline{\Delta c_+}$ by taking randomly a suitable number of neighbors of the initial solution and calculating the average change of cost of those moves that would increase the cost. An approximation of the initial temperature T is then given by (10). Of course, (9) is only a rough estimate of the average number of accepted changes, so the initial temperature given by (10) is only an approximation of the right initial temperature. The average change of cost in cost increasing moves also varies among distinct solutions. Sometimes one may have to take into account many solutions, not just the initial one, to get good initial temperatures.

Our 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 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%.'

3.4 Tabu Search

Tabu search [21, 22] is a combinatorial optimization algorithm that can be viewed as an extension to the steepest descent algorithm in Section 3.2.4.

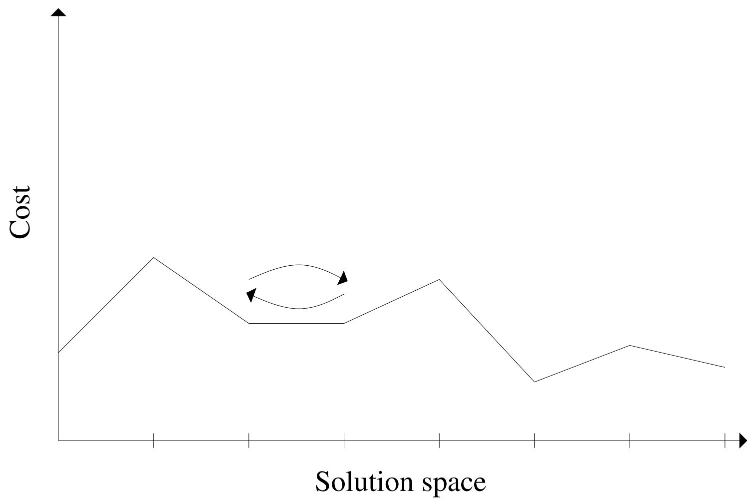

Figure 6: Looping search without a tabu list (in a 1-dimensional solution space).

In the beginning tabu search works exactly the same way as the steepest descent algorithm. When there is no solution s' in the neighborhood of the current solution s such that $c(s') < c(s)$, steepest descent algorithm stops. Tabu search instead continues to move to the best neighbor in the neighborhood of S. This induces a possibility of looping (see Figure 6), because even non-cost-decreasing moves are allowed. Tabu search prevents this kind of looping by introducing a *tabu list.* A tabu list keeps track of the inverses of the last t_l moves and prevents moves that are contained in the list. This prevention is called the *tabu list condition.*

The tabu list T could be implemented as a set of solution pairs:

$$T = \left\{ (s', s) \;\middle|\; \begin{array}{l} \text{a move from solution } s \text{ to solution } s' \text{ has oc-} \\ \text{cured during the last } t_l \text{ moves} \end{array} \right\}$$

However, it is common to put the inverses of the moves themselves on the tabu list:

$$T = \left\{ d^{-1} \mid \text{ the move } d \text{ has been applied during the last } t_l \text{ moves} \right\}. \quad (11)$$

Notice, that as was defined in Section 3.1, a move is a function from a subset of the solution space to the solution space. In practice, the moves must be identified somehow. If the move is identified by a list of integers, we can put that list in the tabu list. However, in some examples of tabu search (see for example [21]), not all the integers in the list characterizing the move are put in the tabu list, that is, the elements of the tabu list are actually sets of moves. This is perfectly acceptable, as long as the inverse of the move that was taken is included in the set. This can be seen as an attempt to improve the diversity of the search.

1. Select an initial solution s.
2. While the stop condition is not met, do the following:
   (a) Let d be the move that leads to the lowest-cost neighbor of s such that
   $$(d \notin T) \vee (d \in T \wedge c(d(s)) < A(d, s))$$
   (b) Let $s := d(s)$. Remove the oldest element in T and add d^{-1} to T.

Figure 7: Tabu search.

Tabu list (11) often restricts the search too much. It may be possible that there are only a few moves in the problem, but each move can be applied to a great number of solutions resulting in a possibly equally great number of neighbor solutions. That is why it is customary to introduce an additional relaxation to the tabu list condition, called the *aspiration level function* $A(d, s)$, which is a two-place function taking as its arguments a move and a solution and returning a value in the same units as the cost function. Aspiration level function need not be time-invariant—indeed, both aspiration level functions described below vary in time as the optimization process proceeds. If the tabu condition would suppress a move d at solution s, it can still be taken, if it satisfies the *aspiration level function criterion*

$$c(d(s)) < A(d, s). \tag{12}$$

We get the tabu search algorithm in Figure 7. We say that an aspiration function is *safe*, if it, combined with a tabu list of form (11), does not permit any loops of length t_l or less.

If the set of possible cost function values is sufficiently small, one safe aspiration function can be obtained as follows [21]. Let the set of possible cost function values be $\mathcal{C}$. We define an initial aspiration level function $A_0(d, s) = c(s)^-$, where $x^-, x \in \mathcal{C}$ denotes a cost value that is less than x such that there exists no other cost value y: $x^- < y < x$. Recall that $\mathcal{C}$ is finite and $\mathcal{C} \subset \mathbb{R}$. If x is the smallest element in $\mathcal{C}$, then $x^- = x$. Now the aspiration level function is

$$A(d, s) = \min \left\{ \{A_0(d, s)\} \cup \left\{ x \;\middle|\; \begin{array}{l} \text{during this optimization process} \\ \text{there has been a move } d' \text{ from} \\ \text{a solution } s' \text{ such that } c(s') = \\ c(s) \text{ and } c(d'(s')) = x \end{array} \right\} \right\} \tag{13}$$

Clearly, this aspiration level function prevents loops of length t_l or less, because if the aspiration level function criterion is fulfilled with a solution s and a move d, we have never during this optimization process moved from any solution with the cost $c(s)$ to a solution that has as low cost as $d(s)$.

It should be noted that during the optimization process the aspiration level moves gradually lower, so that fewer and fewer moves are accepted regardless of the fact that they are on the tabu list.

Another safe aspiration level function is

$$A(d, s) = \begin{array}{l}\text{the cost of the solution that resulted from} \\ \text{the latest application of the move } d.\end{array} \qquad (14)$$

This aspiration level does not move lower during the optimization process as (13), but it is not clear, what the significance of this kind of an aspiration level shift is. We hope that this work can enlighten this aspect of tabu search a little.

Compared to the simulated annealing algorithm in Section 3.3, tabu search does not impose any stop conditions. In our implementations the search is stopped when a given number of iterations has been done or if a global optimum is found.

Note that in step 2(a) in Figure 7 we have to examine the complete neighborhood of the current solution. It is possible to examine only a fraction LF of the neighborhood of the current solution. We call this approach *probabilistic tabu search.*

3.5 Threshold Accepting

The threshold accepting algorithm [17] is an optimization algorithm which is formally very similar to the simulated annealing algorithm. In threshold accepting we accept every move that leads to a new solution *not much worse* than the current solution. To be more specific, if s is the current solution, the proposed next solution $s' \in N(s)$ is accepted as the next solution, if

$$c(s') - c(s) < T_h, \qquad (15)$$

where $T_h \geq 0$ is called the *threshold level.* During the optimization process the threshold level is gradually lowered like the temperature in simulated annealing. In threshold accepting it is guaranteed that a move which would make the current solution much worse is never accepted, unlike in simulated annealing. The convergence of the algorithm has been proved in some cases, see [3].

The threshold accepting algorithm is summarized in Figure 8. The stop cri-

1. Get an initial solution s and an initial threshold $T_h > 0$.
2. While *stop criterion* not satisfied do the following:
   (a) While *inner loop criterion* not satisfied do the following:
      i. Select a solution s' such that $s' \in N(s)$.
      ii. Let $\Delta = c(s') - c(s)$.
      iii. If $\Delta \leq T_h$, set $s = s'$.
   (b) Lower threshold T_h.
3. Return the best solution encountered.

Figure 8: Threshold accepting algorithm.

terion in step 2 and the inner loop criterion in step 2(a) are implemented in our experiments in a way very similar to that of the implementation of the simulated annealing algorithm. The inner loop is performed constant L times and the process is stopped when the cost of the current solution has not improved upon the termination of the inner loop during a given number of threshold levels after the threshold level has reached 0. We accept the proposed new solution even if Δ equals to the threshold level, because then we can lower the threshold down to 0 while keeping the sideway moves allowed. An alternative would be to let the threshold only go down to a small positive constant instead of zero. In any case, the sideway moves in the end of the optimization run seem to improve both the efficiency of the algorithm and the quality of the final solutions in our problems.

It is claimed in [17] that threshold accepting algorithm is rather insensitive to the way the threshold level is lowered. They experimented with Grötschel's Euclidean traveling salesperson problem with 442 cities and concluded that a 'trivial' linear threshold sequence was essentially the best. In linear threshold sequences the next threshold level T_h' is calculated from the current threshold level T_h according to

$$T_h' = \max(0, T_h - r), \tag{16}$$

where $r > 0$ is a constant. However, they suggested that a threshold sequence that lowers the threshold slowlier towards to the end of the optimization run could be better, but they did not have any experimental data supporting this idea. In [17] they used a more adaptive inner loop criterion which ends the inner loop after there has been no improvement in a long time. It may be the case that the threshold accepting algorithm with a linear threshold sequence needs this kind of possibility of adaptation to give higher quality final solutions.

1. Get an initial solution s and an initial water level $W > c(s)$.
2. While *stop criterion* not satisfied do the following:
   (a) While *inner loop criterion* not satisfied do the following:
      i. Select a solution s' such that $s' \in N(s)$.
      ii. If $c(s') \leq W$, set $s = s'$.
   (b) Lower water level W.
3. Return the best solution encountered.

Figure 9: Great deluge algorithm.

An algorithm resembling threshold accepting called *stochastic evolution* was introduced in [42]. Stochastic evolution can be seen as threshold accepting where before every move the threshold is set randomly between 0 and p, where p is regularly updated. They say in [42] that the algorithm works best, if the value of p is initially set near to 0 and incremented only if it seems that the algorithm is stuck in a local optimum. This idea could also be used in simulated annealing, threshold accepting and record-to-record travel in attempt to combine the fast progress of iterative improvement algorithms in the beginning of the optimization run and the high quality final solutions of the more sophisticated algorithms. This, however, is subject to another study.

3.6 Great Deluge Algorithm

The great deluge algorithm [18] simulates the situation where a person is on a hilly island and the sea level suddenly begins to raise. The person walks randomly around and tries keep his or her feet out of the water. In the end the sea level finally reaches the person and the final position of the person is recorded (it is a local optimum). The algorithm for minimization is summarized in Figure 9. The inner loop criterion in step 2(a) is implemented simply by letting the algorithm execute the inner loop L times at each water level, where L is a constant. We accept the proposed new solution even if its cost equals to the water level, because the preliminary tests showed that in these problems the algorithm worked much better that way. The water level is lowered according to

$$W' = \max(c(s), W - r), \tag{17}$$

where W' is the next water level, W is the current water level, $c(s)$ is the cost of the current solution and r is a positive constant. Note that we never lower

1. Get an initial solution s and a record deviation $D > c(s)$. Let $c_{\text{best}} := c(s)$.
2. While *stop criterion* not satisfied do the following:
   (a) Select a solution s' such that $s' \in N(s)$.
   (b) If $c(s') < c_{\text{best}} + D$, set $s = s'$.
   (c) If $c(s') < c_{\text{best}}$, let $c_{\text{best}} := c(s')$.
3. Return the best solution encountered.

Figure 10: Record-to-record travel.

the water level below the current solution, because then it would block the sideway moves from the current solution. The optimization process is stopped when it has not been possible to lower the water level during a given number of water levels.

3.7 Record-to-record Travel

In record-to-record travel [18] a proposed next solution is accepted, if it is not much worse than the best solution seen so far during the optimization run. The algorithm is summarized in Figure 10. The stop criterion in step 2 is implemented simply by ending the run if a global optimum is found or if there has been too many iterations. In this algorithm there is only one parameter, the record deviation, and its use is thus considerably easier than that of e.g. simulated annealing.

3.8 Other Optimization Methods

In addition to the optimization algorithms described in Sections 3.2 through 3.7 we considered genetic algorithms [23]. However, the first experiments showed that a simple genetic algorithm, where each solution is coded into a bitstring and new bitstrings are created through genetic operators (selection, crossover, mutation, inversion), is not competitive with the local search algorithms with the solution coding scheme, cost function and fitness scaling function that we used. It seems that some problem specific heuristic is needed to make genetic algorithms work efficiently in these problems.

According to [24] it is widely recognized that simple genetic algorithms are not good at 'fine tuning' the final solution and a hybrid approach where genetic algorithms are used in the beginning of the optimization run and local search procedures in the end of the run could be used. It turned out, however, that the calculations required by the implemented genetic algorithms to get even near global optima were too time consuming. However, genetic algorithms have been applied successfully to the design of constant weight codes (packing designs with $\lambda = 1$) in [14]. They implemented the cost function, crossover operator, mutation operator and fitness scaling differently from our approach, incorporating more problem-specific knowledge into the algorithm. The difficulty with genetic algorithms as with many other search heuristics is that there are so many implementation options available. This makes it extremely difficult to judge the performance of an algorithm.

The potential of genetic algorithms lies in the fact that they are very easy and straightforward to implement in parallel. Perhaps a massively parallel implementation of a sophisticated genetic algorithm (or more generally—a machine learning based algorithm) could be used to find such designs that are not reachable with the local search strategies.

4 Constructions

In this section we will show how to implement the optimization methods presented in Section 3 so that we can search for covering designs, packing designs and t-designs. Some non-probabilistic construction methods for covering designs are also reviewed.

Many construction methods construct bigger designs from smaller ones. The smaller designs are constructed by hand, or in some cases by methods that work when the parameters satisfy certain conditions. Our approach—viewing the design constructing as an optimization problem—works in principle with any parameter settings. In practice, the space and time demands set the limits within which we can construct good designs. We hope that this approach can be used to form good 'building block' designs that can further be used to construct bigger designs.

4.1 Simple Covering Design Construction Methods

There are several known construction methods that use smaller covering designs to form larger ones. Below we present three such constructions.

4.1.1 Combining Two Covering Designs

This and the next construction method are from [19]. Let G_1 be a $t-(v-1, m, k, \lambda)$ covering design and let G_2 be a $(t-1)-(v-1, m-1, k-1, \lambda)$ covering design. Now a $t-(v, m, k, \lambda)$ covering design can be constructed as follows: take all the blocks of the covering design G_1—they will be part of the resulting covering design. The rest of the blocks are obtained by taking all the blocks of G_2 and by adding the point v to all of them. Hence, we have

$$C_\lambda(v, m, k, t) \leq C_\lambda(v-1, m, k, t) + C_\lambda(v-1, m-1, k-1, t-1). \tag{18}$$

4.1.2 One-bit Splicing

Let $G = (X, \mathcal{A})$ be a $k-(v-1, m, k, 1)$ covering design ($m \geq k$ and $\lambda = 1$). By simple counting arguments, there is at least one point in X that is contained in no more than $\left\lfloor \frac{k|\mathcal{A}|}{v-1} \right\rfloor$ blocks. Without loss of generality, assume that it is the point $v-1$. A new covering design G_1 is now constructed with the following three sets of blocks:

1. All the blocks in G. These blocks cover all the m-subsets that do not contain the point v and some of the blocks which contain the point v but which do not contain the point $v-1$.

2. All the blocks in G which contain the point $v-1$ are included in this new design with the point $v-1$ replaced by the point v. These blocks cover all the m-subsets which contain the point v, which do not contain the point $v-1$ and which were not covered by the blocks in the previous set.

3. All the blocks in a $(k-2)-(v-2, m-2, k-2, 1)$ design with points $v-1$ and v added. These blocks cover all the m-subsets which contain both the points $v-1$ and v.

The constructed covering design is a $k-(v, m, k, 1)$ design and hence

$$\begin{aligned} C_1(v,m,k,k) \;\leq\; & \left\lfloor \frac{v+k-1}{v-1} C_1(v-1,m,k,k) \right\rfloor + \\ & C_1(v-2,m-2,k-2,k-2) \qquad (19) \end{aligned}$$

4.1.3 A Direct Product Type Construction

This construction is a generalization of the method in [38]. Let $G_1 = (X_1, \mathcal{A}_1)$ be a $t_1 - (v, m_1, k, \lambda)$ covering design on x blocks and let $G_2 = (X_2, \mathcal{A}_2)$ be a $t_2 - (v+\delta, m_2, k+\delta, \lambda)$ covering design on y blocks. Then a $(t_1 + t_2 + 1 + \min(m_1 - t_1, m_2 - t_2)) - (2v+\delta, m_1 + m_2 + 1, k + v + \delta, \lambda)$ covering design $G_3 = (X_3, \mathcal{A}_3)$ on $x+y$ blocks can be constructed as follows. Without loss of generality assume that $X_1 = \{1, 2, \ldots, v_1\}$ and $X_2 = \{(v_1+1), (v_1+2), \ldots, (v_1+v_2)\}$. Now take all the blocks of the design G_1 and add all the points $\{(v_1+1), (v_1+2), \ldots, (v_1+v_2)\}$ to them. These blocks form the first part of the resulting design. The second part is obtained by taking all the blocks of the design G_2 and adding the points $\{1, 2, \ldots, v\}$ to them. Hence, we have the following theorem.

Theorem 13

$$\begin{aligned} C_\lambda(2v+\delta, m_1+m_2+1, k+v+\delta, t_1+t_2+1+\min(m_1-t_1, m_2-t_2)) \leq \\ C_\lambda(v, m_1, k, t_1) + C_\lambda(v+\delta, m_2, k+\delta, t_2). \end{aligned}$$

Proof: Clearly, the number of points and sizes of the blocks in the covering design G_3 are correct. Now consider a m_1+m_2+1-subset of X_3 that has r elements belonging to X_1 and m_1+m_2+1-r elements belonging to X_2. Obviously, $r \leq v$ and $m_1+m_2-r \leq v+\delta$. If $r \geq m_1+1$, then the set is covered at least λ times in at least $r + t_2 - (r - (m_1+1)) = t_2 + m_1 + 1$ points in the second part of the design. Correspondingly, if $r \leq m_1$, i.e. $m_1+m_2+1-r \geq m_2+1$ then the set is covered at least λ times in at least $m_1 + m_2 + 1 - r + t_1 - (m_1 - r) = t_1 + m_2 + 1$ points in the first part of the design. So all the $m_1 + m_2 + 1$-subsets are covered λ times in at least $\min(t_2+m_1+1, t_1+m_2+1) = t_1+t_2+1+\min(m_1-t_1, m_2-t_2)$ points. $\square$

4.2 Covering Designs by Local Search

In this section we show how we have chosen problem-specific parameters, like the cost function, solution space, moves and neighborhoods for the optimization algorithms in the search for covering designs. Data structures and other implementation issues are also discussed.

4.2.1 Solution Space and Cost Function

In the search for covering designs $(X, \mathcal{A})$ we let a solution s_{cover} be a list of length b, whose elements are k-subsets of X. We write

$$s_{\text{cover}} = (a_1, a_2, \ldots, a_b)\,,$$

where each a_i is a k-subset of X. The size of the solution space S_{cover} is thus

$$|S_{\text{cover}}| = \binom{v}{k}^b . \tag{20}$$

In the sequel we use the notation $x \in s_{\text{cover}}$ to denote the fact that x is an element in the list $s_{\text{cover}} = (a_1, a_2, \ldots, a_b)$.

Covering designs cover all m-subsets of v points λ times. One possible cost function for a solution s_{cover} is the number of m-subsets that are covered less than λ times, i.e.

$$c_{\text{cover}}(s_{\text{cover}}) = |\{x \in X_m : |\{y \in s_{\text{cover}} : |x \cap y| \geq t\}| < \lambda\}|\,, \tag{21}$$

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

$$c_{\text{cover}}(s_{\text{cover}}) = \sum_{x \in X_m} \max\Big(0,\ \ \lambda - |\{y \in s_{\text{cover}} : |x \cap y| \geq t\}|\Big). \tag{22}$$

For $\lambda = 1$, (21) and (22) coincide. The intuitive argument in favor of (22) has turned out to hold, so this cost function has been used. Both cost functions are zero exactly in the global optima and positive everywhere else.

4.2.2 Moves and Neighborhoods

The most natural neighborhood structure is as follows. A solution s' is a neighbor of another solution s if s' can be obtained from s by changing one point in one block of s. More formally, a move $d_x^{\alpha \leftarrow \beta}$ is a function that maps a solution $s = (a_1, a_2, \ldots, a_x, \ldots, a_b)$ to $s' = (a'_1, a'_2, \ldots, a'_x, \ldots, a'_b)$ so that $a_i = a'_i$ for $i \neq x$, and $a'_x = a_x \setminus \alpha \cup \beta$, where $\alpha \in X$ and $\beta \in X$. We require that $\alpha \neq \beta$, so that degenerate moves are not allowed. The domain $S_{\text{cover}}(d_x^{\alpha \leftarrow \beta})$ of a move $d_x^{\alpha \leftarrow \beta}$ is

$$S_{\text{cover}}(d_x^{\alpha \leftarrow \beta}) = \\ \{s_{\text{cover}} = (a_1, a_2, \ldots, a_x, \ldots, a_b) \mid s_{\text{cover}} \in S_{\text{cover}} \text{ and } \alpha \in a_x \text{ and } \beta \notin a_x\}\,.$$

The set of all moves is

$$\mathcal{D}_{\text{cover}} = \left\{ d_x^{\alpha \leftarrow \beta} \mid 1 \leq x \leq b, 1 \leq \alpha \leq v, 1 \leq \beta \leq v, \alpha \neq \beta \right\}.$$

There are $bv(v-1)$ moves in $\mathcal{D}_{\text{cover}}$. As is apparent from the definition of a move, not all moves can be applied to a solution. The set of moves applicable to a solution s_{cover} is

$$\left\{ d_x^{\alpha \leftarrow \beta} \in \mathcal{D}_{\text{cover}} \mid s_{\text{cover}} \in S_{\text{cover}}(d_x^{\alpha \leftarrow \beta}) \right\}.$$

Recall from Section 2 that the neighborhood $N(s)$ of a solution s is

$$N(s) = \{s' \mid d(s) = s' \text{ for some } d \in \mathcal{D}\}\,.$$

The neighborhood size in this neighborhood implementation is thus

$$|N(s_{\text{cover}})| = bk(v-k) \tag{23}$$

for all $s_{\text{cover}} \in S_{\text{cover}}$. For tabu lists of type (11) an inverse of a move is needed. The inverse of a move $d_x^{\alpha \leftarrow \beta}$ is

$$\left(d_x^{\alpha \leftarrow \beta} \right)^{-1} = d_x^{\beta \leftarrow \alpha} \tag{24}$$

and it can thus be characterized by a 3-tuple (x, α, β). In addition to that, we experimented by putting only one or two of these entities on the list. These 2- or 1-tuples represent sets of moves.

4.2.3 Implementation

The main difficulty in implementing the cost function (22) and the neighborhood structure described in Section 4.2.2 is efficiency. We rely on an extensive tabulation of neighborhood and covering data. We need space for three potentially large tables. Neighborhood information is stored in the table `neighbors`, the information on which sets are covered by a given k-subset is stored in `coverings` and the information on how many times a given m-subset is covered by the current selection is stored in `covered`. A diagram of the data tables is given in Figure 11, where

$$d(v, m, k, t) = \sum_{i=t}^{\min(k,m)} \binom{k}{i} \binom{v-k}{m-i} \tag{25}$$

denotes the number of m-subsets covered by a k-subset in t points when the total number of points is v. In the sequel the tables are discussed in more detail.

To be able to efficiently store neighbors, etc., we rank individual k-subsets and m-subsets 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 [45, pp. 162–163]. The algorithms use binomial coefficients, which have been computed in an array.

The calculation of the cost of a given solution is the most critical part in implementation of the neighborhood structure and the cost function discussed above. In order to compute the cost of a random solution and the change of cost when moving from a solution to one of its neighbors efficiently, we need information to answer the following questions:

- Which m-subsets are covered by a given k-subset? In the program this information is tabulated in the table `coverings`. The nth column of the table gives the ranks of the m-subsets covered by a k-subset with the rank n.

- How many k-subsets in the current solution cover a given m-subset? This information is kept in the table `covered`. The nth entry in this table gives the current number of blocks which cover the m-subset with the rank n.

- Which k-subsets can be obtained from a given k-subset by one element change? This information is in the table `neighbors`. The nth column of the table lists the ranks of those k-subsets that can be obtained by one element change from the k-subset with the rank n.

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-subset A that is changed to another set A' in order to obtain the new solution s'. We take the m-subsets covered by the set A and count the entries less than or equal to λ in the corresponding places in the table `covered`. Let this count be Δ^-. All

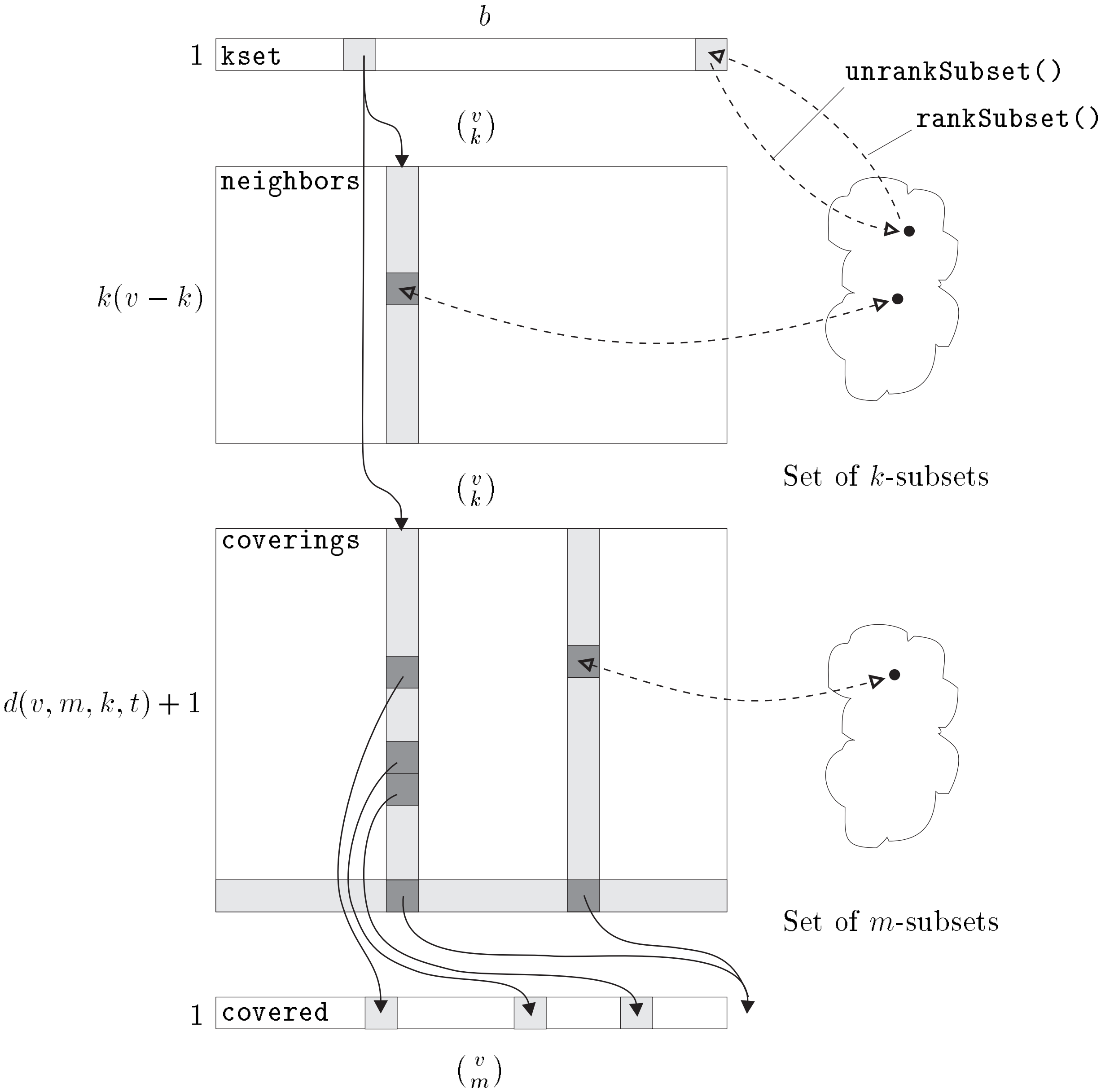

Figure 11: Neighborhood and covering data tables.

these m-subsets will no longer be covered enough, 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-subset covered by A. Next we count the entries less than λ in those members of `covered` that represent an m-subset covered by A'. Let this count be Δ^+. All these m-subsets will be more covered when A' is added to s. Then the change of cost $\Delta = \Delta^+ - \Delta^-$.

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 in most of the optimization algorithms 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-subset after each column of the table `coverings` (see Figure 11). Then we sort the ranks of the m-subsets covered by each k-subset in the table `coverings` in increasing order before starting a local search optimization process. 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-subset, the other pointing to the first cell of the column of the current k-subset. Let us call the pointers `nextPtr` and `currPtr`, respectively. Then we set the cost change `costDelta` initially to 0 and process the pointers according to the following rules:

`*nextPtr` = `*currPtr`: The m-subset rank to which `nextPtr` points equals the rank to which `currPtr` points. That is, the same m-subset 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-subset rank to which `nextPtr` points is less than the rank to which `currPtr` points. If the m-subset with the rank to which `nextPtr` points is currently less than enough covered (`covered[*nextPtr]` $< \lambda$), decrement `costDelta` by 1 (this m-subset will become more covered, if the new solution is accepted). In any case, increment `nextPtr`.

`*nextPtr` > `*currPtr`: The m-subset rank to which `currPtr` points is less than the rank to which `nextPtr` points. If the m-subset with the rank to which `currPtr` points is currently covered by λ or less k-subset(s) (`covered[*nextPtr]` $\leq \lambda$), increment `costDelta` by 1 (this m-subset will be less 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 only if the new solution is accepted, in which case the current cost must also be updated by `costDelta`.

So we traverse once through the m-subsets covered by two k-subsets. According to (25) the number of m-subsets covered by one k-subset is exponential in v in the worst case, so the time required to calculate the cost change grows also exponentially with v in the worst case.

Of course, the cost of the initial solution in local search algorithms must be computed by traversing through all the b blocks. If the time of a cost change calculation is $T_{\Delta\,\mathrm{cost}}$, then the time needed by the cost calculation is

$$T_{\mathrm{cost}} \approx \frac{b}{2} T_{\Delta\,\mathrm{cost}}. \tag{26}$$

This shows one advantage of local search algorithms in combinatorial optimization problems. It is common in combinatorial optimization that a cost change calculation when the solution is altered in a suitable way is easier and faster than calculating the cost of a given solution.

The space complexity of our approach is exponential in v in the worst case, too. Usually the space demands set the limits for the parameters of the design. However, with suitable parameter selection (especially when b is large and m equals t) the time demand can take over.

The number of elements in the tables in Figure 11 are as follows, see (25) for the definition of $d(v, m, k, t)$.

$$\begin{aligned} \mathrm{elem}(\texttt{neighbors}) &= k(v-k)\cdot\binom{v}{k} && (27)\\ \mathrm{elem}(\texttt{coverings}) &= \binom{v}{k}\cdot[d(v,m,k,t)+1] && (28)\\ \mathrm{elem}(\texttt{covered}) &= \binom{v}{m} && (29)\\ & && (30)\end{aligned}$$

The memory requirements of the tables depend on the element sizes, which can be different in different tables depending on the parameter range of interest.

Usually (28) is the dominating space requirement and it increases at a rather high rate. The maximum value of (28) 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.

Fortunately, the case $t = 1$ is rarely encountered. Usually t is near to m, if not equal to it. Equations (28) and (20) 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	Maximum size of `coverings`
3	36
6	1520
9	63504
12	3411408
15	165636900

Table 2: Some maximum sizes (in bytes) of the table `coverings` for given v when each element in the table is 4 bytes.

4.3 Packing Designs by Local Search

In search for packing designs we use a neighborhood structure similar to that in search for covering designs. The set of solutions is $S_{\text{pack}} = S_{\text{cover}}$, the set of moves is $\mathcal{D}_{\text{pack}} = \mathcal{D}_{\text{cover}}$ and we use similar notation for a move: $d_x^{\alpha \leftarrow \beta}$. The inverse of a move is defined in a similar way. Only the cost function is altered, it is now

$$c_{\text{pack}}(s_{\text{pack}}) = \sum_{x \in X_m} \max\Big(0, \quad |\{y \in s_{\text{pack}} : |x \cap y| \geq t\}| - \lambda\Big). \tag{31}$$

That is, a m-subset covered x times adds the cost by $x - \lambda$, if $x > \lambda$.

4.3.1 Implementation

The implementation is almost the same as in Section 4.2.3. The only changes concern the cost and cost change calculation from the values in the table `covered` which tells how many time each m-subset is covered by the current solution. With covering designs, the cost was added by one, if a value of an element in the table `covered` was decreased and the final value of the element was less than λ. Similarly, if the value of an element in the table was increased by one and its start value was below λ, the cost was decreased by one. Now with packing designs, the procedure is inverted. If an m-subset is covered more than λ blocks, the cost is increased when the corresponding entry in the table `covered` is increased. Similarly, if the value of an element in the table `covered` is decreased, the cost is decreased, too, if the value of the element was greater than λ.

Note that in the case $\lambda = 1$, it would be possible to use a cost function based on the Hamming distances between the blocks. Further, we can restrict ourselves to the case $m = t$, see Section 2.2. According to (3) the Hamming distance between any two blocks should be greater than or equal to $2(k - t + 1)$, i.e. their intersection should be smaller than t points. A possible cost function would be

$$c_{\text{pack}}^{\lambda=1}(s) = \sum_{\substack{a \in s, b \in s \\ a \neq b}} \max(0, |a \cap b| - t + 1) . \tag{32}$$

It is be easier to calculate the sizes of these intersections than to count the m-subsets covered by individual blocks. The memory requirements are much lower, too. However, we wanted to keep this work as general as possible and so this approach was not tested. In any case, distance based cost functions have been used in coding theory and they have proven useful, see e.g. [20, 28].

4.4 t-designs by Local Search

Steiner triple systems have been searched in [48]. In their algorithm blocks are added one by one while maintaining the condition that no pair of points is contained in too many blocks. If the negated number of blocks is taken to be the cost function, then their algorithm can be seen as a probabilistic local optimization algorithm where sideway moves are allowed. Their algorithm works very well with Steiner triple systems, but not with any larger t-designs.

We could search for t-designs by searching a covering or packing design with the same parameters (with $m = t$). However, we can try to exploit the fact that each point is contained in $r = bk/v$ blocks (this is the condition (4) necessary for the existence of a t-design).

4.4.1 Solution Space and Cost Function

The solution space $S_{t-\text{des}}$ is a subset of the solution space S_{cover} in Section 4.2.1.

$$S_{t-\text{des}} = \{s_{t-\text{des}} \in S_{\text{cover}} \mid \text{each point occurs in } r = bk/v \text{ blocks of } s\} .$$

The cost function now penalizes ‘overcovering’ and ‘undercovering’, i.e.

$$c_{t-\text{des}}(s_{t-\text{des}}) = \sum_{x \in X_m} \text{abs}\left(\lambda - |\{y \in s_{t-\text{des}} : x \subseteq y\}|\right) , \tag{33}$$

where $\text{abs}(i) = |i|$ denotes the absolute value of an integer. It is easy to see that (33) is zero exactly in the global optima (that is, in solutions that are t-designs). We could have used the cost function (22), too. As a matter of fact, (33) is (22) multiplied by two.

4.4.2 Moves and Neighborhoods

A move now swaps two points between two blocks. More formally, a move $d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta}$ is a function that maps a solution $s_{t-\text{des}} = (a_1, a_2, \ldots, a_x, \ldots, a_y, \ldots, a_b)$ to another solution $s'_{t-\text{des}} = \left(a'_1, a'_2, \ldots, a'_x, \ldots, a'_y, \ldots, a'_b\right)$ so that $a_i = a'_i$ for $x \neq i \neq y$ and $a'_x = a_x \setminus \alpha \cup \beta$ and $a_y = a_y \setminus \beta \cup \alpha$; $\alpha \in X$ and $\beta \in X$. We require that $x < y$ and $\alpha \neq \beta$ so that potentially degenerate moves are not allowed. The domain $S_{t-\text{des}}(d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta})$ of a move $d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta}$ is

$$S_{t-\text{des}}(d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta}) = \left\{ s_{t-\text{des}} = (a_1, \ldots, a_x, \ldots, a_y, \ldots, a_b) \;\middle|\; \begin{array}{l} s_{t-\text{des}} \in S_{t-\text{des}} \text{ and } \alpha \in a_x \text{ and} \\ \beta \notin a_x \text{ and } \alpha \notin a_y \text{ and } \beta \in a_y \end{array} \right\}.$$

The set of all moves is $\mathcal{D} = \left\{ d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta} \mid 1 \leq x < y \leq b; 0 \leq \alpha, \beta \leq v-1, \alpha \neq \beta \right\}$. There is a total of $bv(b-1)(v-1)/2$ moves in $\mathcal{D}$. The inverse of a move $d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta}$ is

$$\left(d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta}\right)^{-1} = d_{x\leftrightarrow y}^{\beta\leftrightarrow\alpha}, \tag{34}$$

so there are four integers that identify the inverse move. A tabu list entry is a 4-tuple (x, y, α, β).

The set of moves applicable to a solution s is

$$\left\{ d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta} \in \mathcal{D}_{t-\text{des}} \mid s_{t-\text{des}} \in S_{t-\text{des}}(d_{x\leftrightarrow y}^{\alpha\leftrightarrow\beta}) \right\}.$$

The neighborhood size $|N(s_{t-\text{des}})|$ in this move and neighborhood implementation is not constant in all the solutions. Simple counting gives

$$0 \leq |N(s_{t-\text{des}})| \leq \frac{k^2 b(b-1)}{2}$$

for all $s_{t-\text{des}} \in S_{t-\text{des}}$. The lower bound zero is reached when all the blocks are identical. This possibility contradicts with the assumption in Section 3.1 that in any solution in the solution space there is at least one move that is applicable in that solution. However, it is easy to prove that if a solution with all blocks identical is in the solution space $S_{t-\text{des}}$, then it is the only solution in the solution space and the lack of applicable moves does no harm.

4.4.3 Implementation

There is a direct connection between the structure of the solution space $S_{t-\text{des}}$ and t-designs. In fact, each solution $s \in S_{t-\text{des}}$ is a $1-(b, r, k)$ design. A 1-design can be easily constructed, see Figure 12. The array in the figure has b columns and k rows. The columns are the blocks of the design. The array is

blocks						
1	2	$\cdots$	λ	$\lambda + 1$	$\cdots$	b
1_1	1_2	$\cdots$	1_λ	2_1	$\cdots$	α_n
α_{n+1} or $(\alpha + 1)_1$		$\cdots$			$\cdots$	
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
		$\cdots$			$\cdots$	v_λ

Figure 12: A construction method for $1 - (v, k, \lambda)$ designs.

filled row by row, each row from left to right. First, λ 1's are inserted, then λ 2's, and so on, until λ v's are inserted to the right of the bottom row. Now each point is included in λ blocks and a $1 - (v, k, \lambda)$ design is complete. Even though it is trivial to form a $1 - (b, r, k)$ design, it is nontrivial to pick up any solution in the solution space $S_{t-\mathrm{des}}$ with equal probability, or even so that the probability of selecting a solution is nonzero with every solution. That is why we form a random initial solution by first constructing a solution with the method in Figure 12 and then by applying a given number of random moves to it.

In addition to the data tables in Figure 11, we keep the binary representations of the blocks (see Section 2.1) in the memory. A random move can now be formed by taking two random blocks (they must not equal) and by calculating the binary AND of their bit strings a and b, let us denote the result by $c = a \wedge b$. The 1's in the result c indicate those elements that are in the both sets—they can not be swapped, because then we would get a degenerate move. Further, no point in a can be swapped with such a point of b that the point in b would coincide with another point in a. Thus the points that can be swapped in a and b are $a_{\mathrm{acc}} = a \wedge \overline{c} \wedge \overline{b} = a \wedge \overline{b}$ and $b_{\mathrm{acc}} = b \wedge \overline{a}$, respectively, where overline denotes the bitwise negation. Because a and b are assumed not to be equal, a_{acc} and b_{acc} both have one or more 1's. Now we choose a 1 from each and swap the corresponding points between these blocks.

This strategy of selecting a random move does not attach an equal probability of selection to each move—the selection distribution is not uniform. However, this strategy is feasible to implement and it works satisfactorily.

A solution change changes two blocks, not one as in covering or packing designs. The cost change calculation is now slightly more complex, because we have to use four pointers instead of two, one is used to browse through the m-subsets covered by the current first block (a_x in the definition of a move in Section 4.4.2), one for the current second block (a_y), one for the next first block (a'_x) and the fourth pointer for to next second block (a'_y). Otherwise, the procedure is a generalization of that in Section 4.2.3.

5 Comparisons of the Performance of the Algorithms

5.1 Time Calculation in Comparisons

In this problem, the cost change calculation time grows exponentially in v in the worst case as can be deduced from (25). On the other hand the overhead in the algorithm implementations depend greatly on the implementation details, and even on the computer architecture. That is why we decide not to use in the comparisons the actual CPU-time used by the algorithms, but instead the number of passes through a column in table `coverings` in Figure 11, i.e. the number of k-subsets, whose coverage is examined. A cost change calculation in the covering or packing design neighborhood structures increments this number by 2 and a cost calculation by b, see (26). Correspondingly, the cost change calculation in the t-design neighborhood structure increments this number by 4 and a cost calculation by b. This number gives a rather accurate measure between the optimum implementations (which are assumed to have little overhead) of the algorithms, although the time values with different neighborhood structures are not exactly comparable, because the cost change calculation in the t-design neighborhood structure is more complex and has more overhead.

5.2 Testcases

In the comparisons of the performance of the algorithms we used the following two testcases:

1. Covering design $5-(10,5,6,1)$ with 50 blocks.

   This is a minimal $5-(10,5,6,1)$ covering design. It was shown in [10] that $C_1(10,5,6,5)=50$ and that this minimal design is unique up to isomorphism. The size of the solution space (20) is $|S| = \binom{10}{6}^{50} \approx 1.3 \cdot 10^{116}$ and the neighborhood size (23) is $50 \cdot 6 \cdot 4 = 1200$.

2. Packing design $5-(14,5,7,1)$ with 42 blocks.

   This is a maximal packing design [8]. It was intented that this testcase would be harder than the previous one. The size of the solution space is $|S| = \binom{14}{7}^{42} \approx 3.1 \cdot 10^{148}$ and the neighborhood size is $42 \cdot 7^2 = 2058$.

The selection of testcases is very difficult, because it is apparent that the mutual superiority between the algorithms depends on the hardness of the problem instance. We would like to choose as hard problem instances as possible, because we are mainly interested in which algorithms perform well in

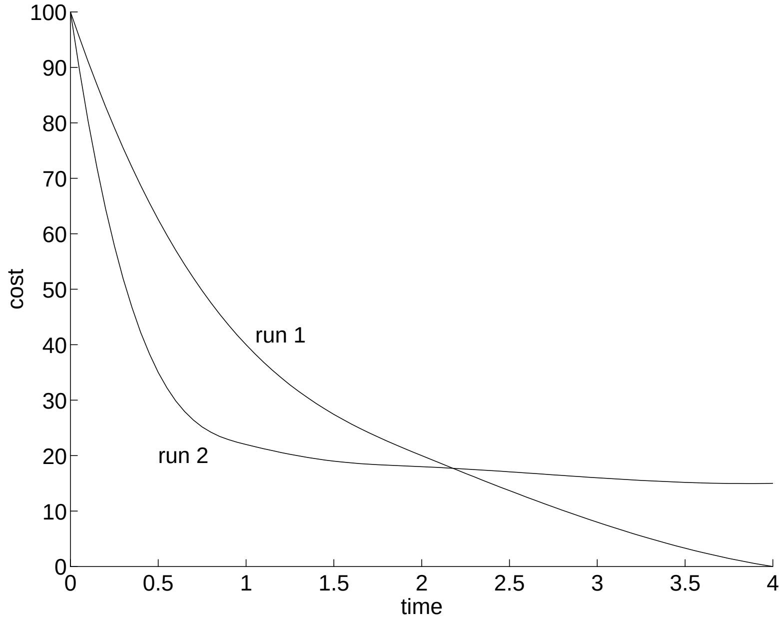

Figure 13: Two different optimization runs.

difficult situations. On the other hand, the computations become infeasible, if the problem is too difficult. It must be remembered that in these comparisons a single run is not sufficient. For example the simulated annealing algorithm has so many parameters that to get a rigorous picture of the optimal performance of the algorithm we need hundreds of thousands of optimization runs with different parameters.

5.3 Comparison Method

When comparing the performance of the algorithms we could simply optimize the average time of finding a global optimum. This is called *meta-optimizing*, because we optimize the parameters of an optimization algorithm. However, in this work, we are interested in the beginning of the optimization runs, too—some optimization parameter settings give very good solutions quickly, but the global optimum is perhaps never obtained. Some other parameter settings can lead almost always to a global optimum, but the progress in the beginning of the optimization run is slow (see Figure 13). In all the performance plots in this work the y-axis is the cost axis with the cost of global optima in the origo and x-axis is the time axis. The time unit is the number of passes through a column in the table `coverings` in Figure 11 as described in Section 5.1.

In order to make the following discussion more precise, we need a few definitions.

Definition 9 *We define a* run r *to be a list of (time,cost) pairs called* points

$$r = ((t_1, c_1), (t_2, c_2), \ldots, (t_n, c_n)),$$

where each pair (t_i, c_i) *indicates that by the time* t_i *the best solution encountered had the cost* c_i*. The length of a run is the number of the pairs in the list.*

A set R of *equally timed runs* is a set of runs which have the same length and which have the property that any two runs $r = ((t_1, c_1), (t_2, c_2), \ldots, (t_n, c_n))$ and $r' = ((t'_1, c'_1), (t'_2, c'_2), \ldots, (t'_n, c'_n))$ are equally timed, i.e. $t_i = t'_i$ for any $1 \leq i \leq n$. If a run r is equally timed with another run r', it is denoted by $r \parallel r'$.

The *mean cost run* mean(R), or mean run for short, of a set R of equally timed runs is a run r that satisfies $r \parallel r'$ for any $r' \in R$ (the run is equally timed with the set) and the cost values of the points in r are the means of the points with the same time in the runs in R. A *median cost run* med(R), or median run for short, is formed similarly, we only use median instead of mean.

We perform n optimization runs with a parameter setting and get a set of equally timed optimization runs. Then we take the median of these runs and get a good picture of the performance of the algorithm with these parameter settings.

We use the median instead of the mean, because median gives us some information of the time when the global optimum is obtained, while mean does not (see Figures 14 and 15). In Figure 14, a set of optimization runs is presented, all of which but one reach the global optimum. The mean and median runs are plotted in Figure 15. Note that the mean run alone in Figure 15 gives us no information, whether or not the global optimum was ever reached, but the median run indicates that more than 50% of the runs did reach a global optimum by the time 6. It is typical in some algorithms that there are some runs that do not reach the global optimum, thus the mean of these runs reaches the global optimum neither, but if more than 50% of the runs reach the global optimum, then so does the median run.

In order to find out the best possible performance of an optimization algorithm we sample extensively the solution space of the meta-optimizing problem. As a result we get a set of median runs (see Figure 16). Note that in this figure there are only 44 median runs, each resulting from 21 optimization runs. In the actual tests we used a substantially greater number of optimization runs.

Now we will select the best points of the runs in this plot. There are, however more than one variable determining the quality of a point. The criterion used here for selection of the best points is *Pareto optimality*.

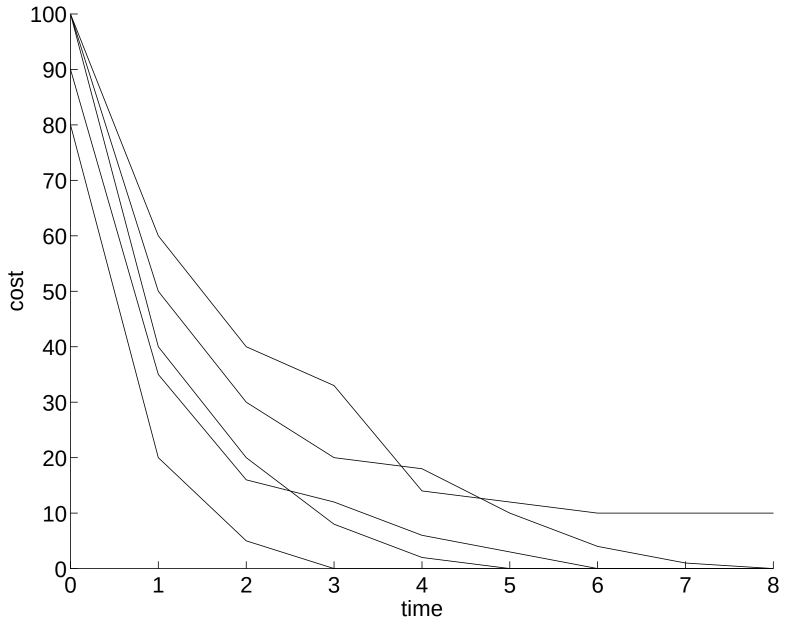

Figure 14: A set of runs.

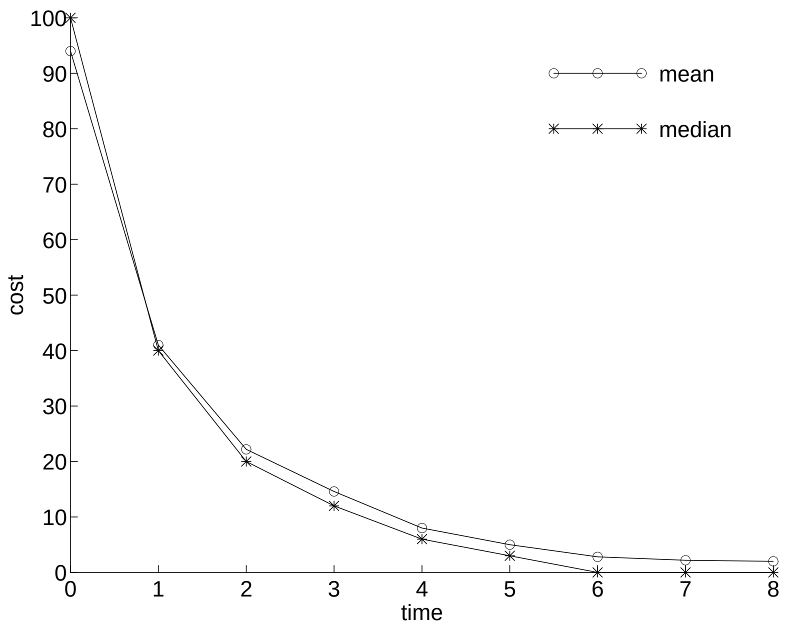

Figure 15: Mean and median runs of the set in Figure 14.

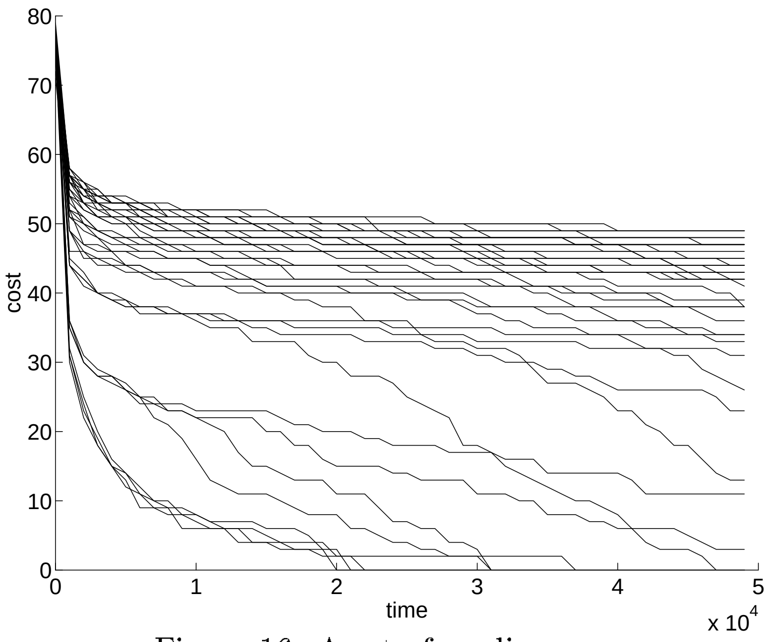

Figure 16: A set of median runs.

Definition 10 *A point* $\mathbf{p} = (p_1, p_2, \ldots, p_n) \in \mathbb{R}^n$ *is* dominated *by* $\mathbf{p}' = (p'_1, p'_2, \ldots, p'_n) \in \mathbb{R}^n$ *iff*

$$p'_i < p_i \textit{ for all } 1 \leq i \leq n.$$

A point $\mathbf{p} \in P \subset \mathbb{R}^n$*, where* P *is finite, is* Pareto-optimal in set P*, if it is not dominated by any other point* $\mathbf{p}'$ *in* P*.*

As a result we obtain a fictitious run shown in Figure 17 by connecting the Pareto-optimal points. This run is fictitious, because it is generally not obtainable with any single parameter setting. However, it describes the best possible performance of the algorithm, because any single point on the run *is* obtainable with some parameter setting. These optimal performance run curves form the basis of our performance comparisons in testcase 1, see Figures 18 through 20.

5.4 Comparison Results

The optimization algorithms in tests include the following algorithms: random sampling (RSAMPLE), random walking (RWALK), steepest descent algorithm (STEEP), probabilistic steepest descent algorithm (PROBSTEEP), probabilistic local optimization (PROBLOC), probabilistic local optimization with sideway moves allowed (SIDELOC), simulated annealing (SA), tabu search (TABU), probabilistic tabu search (PROBTABU), threshold accepting (THRESHOLD), great deluge (DELUGE) and record-to-record travel (REC).

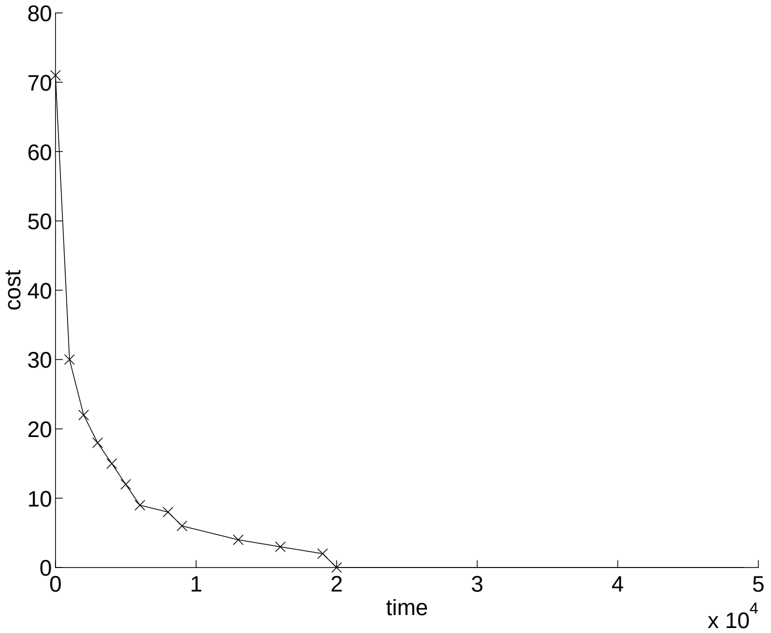

Figure 17: A fictitious run consisting of the Pareto-optimal points.

5.4.1 Testcase 1

The performance curves of the elementary optimization algorithms (RSAMPLE, RWALK, STEEP, PROBSTEEP, PROBLOC and SIDELOC) are plotted in Figure 18. Notice the very good performance of SIDELOC. The difference between RWALK and RSAMPLE is explained by mentioning that the cost change calculation between two neighbors is performed in 1/25th of the time of the cost calculation of a random solution, see (26). STEEP reaches low-cost solutions slowly, because it has to check every solution in the neighborhood of the current solution. PROBSTEEP starts to decrease the cost much more rapidly, because it checks only a fraction of all the neighbors of the current solution. However, also the costs of the final solutions of STEEP are generally higher than those of PROBLOC and PROBSTEEP. It appears that if only the most cost-decreasing moves are accepted, then the performance of the algorithm is not as good as if also some moves that decrease the cost only little were accepted. This explains the better final solutions of PROBSTEEP and PROBLOC. They cannot compete with the performance of SIDELOC, however.

SA, THRESHOLD, DELUGE and REC reduced to SIDELOC in our metaoptimizing tests in this testcase and thus their performance was equal to that of SIDELOC in Figure 18.

The performance curves of the tabu search algorithms are plotted in Figure 19. TABU performs much better than STEEP, even though the complete neighborhood check slows it down when compared to PROBTABU. PROBTABU

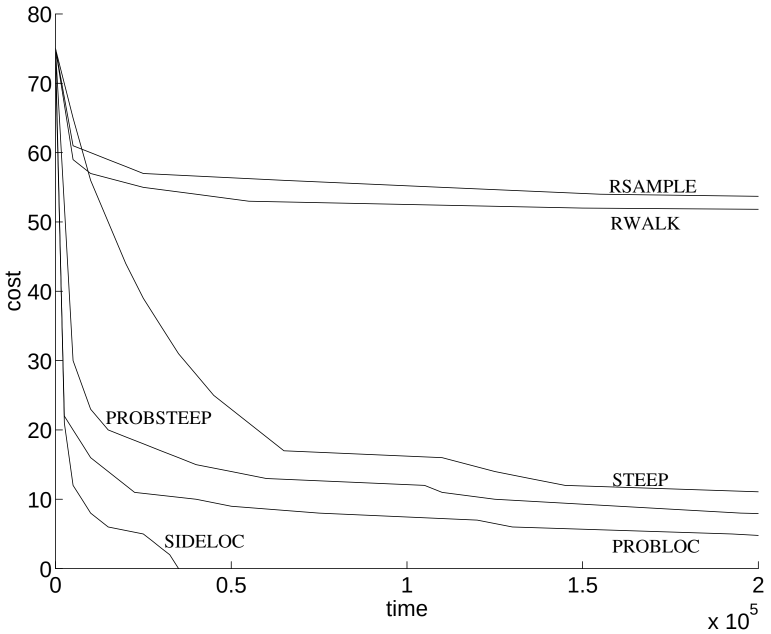

Figure 18: The performance curves of the elementary optimization algorithms.

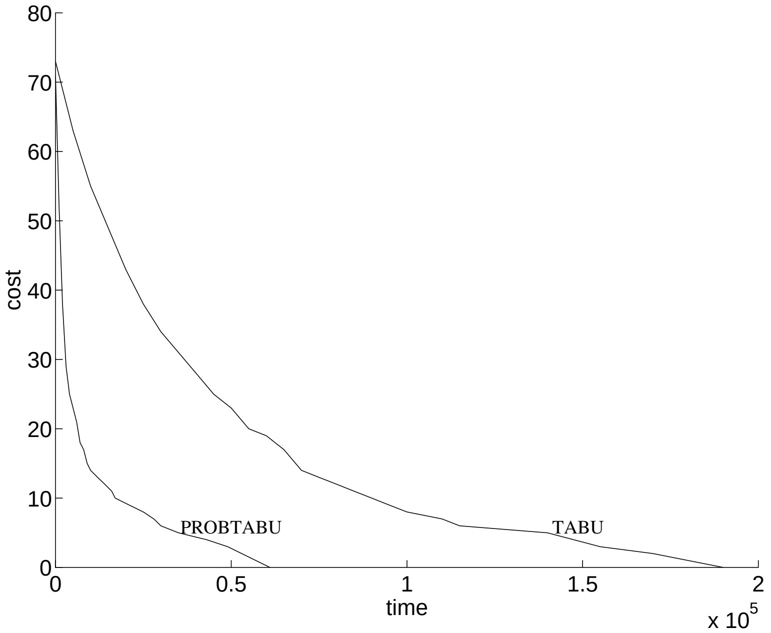

Figure 19: The performance curves of the tabu searches.

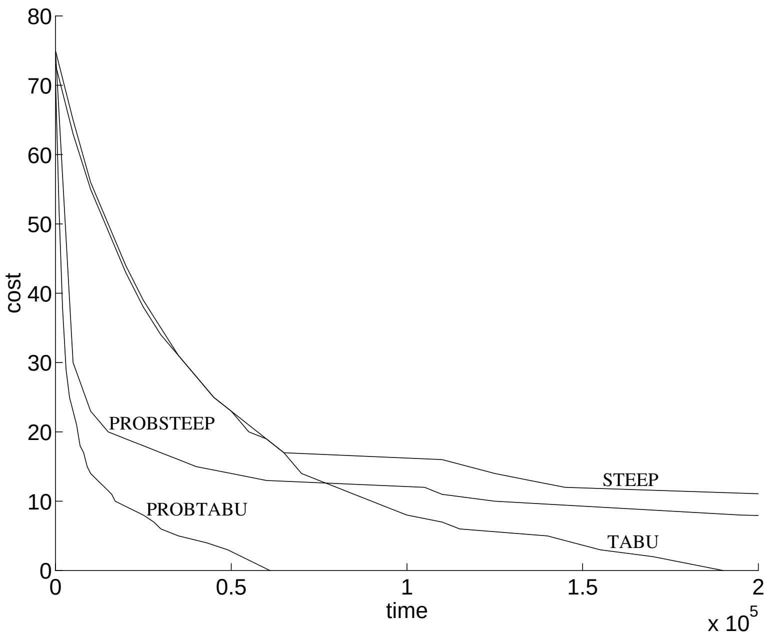

Figure 20: The similarity between tabu searches and steepest descent algorithms.

reduces the cost in the beginning much faster than TABU, but near the global optimum cost it is not much faster than TABU. The experiment data shows that if we examine only a very small fraction of the neighborhood of the current solution, then the performance of the algorithm is poor. It may be the case that in more difficult problems PROBTABU does not work much faster than TABU.

As was mentioned in Section 3.4, there is a direct connection between TABU and STEEP, as well as between PROBTABU and PROBSTEEP. The performance curves of these four algorithms are plotted in Figure 20, where it can be seen how similarly the corresponding algorithms behave in the beginning of the optimization runs.

5.4.2 Testcase 2

The second testcase was computationally so demanding that we could not perform similar extensive tests as with the first testcase. We were limited to perform only a small number of optimization runs and thus we could not use the techniques described in Section 5.3. The plots in this section are plots of individual runs, not plots of Pareto-optimal points of large sets of median runs.

First, we wanted to know, if SIDELOC still works as well as with the first testcase. After running a few long runs, we noticed that SIDELOC never managed to reach the global optimum. We ran SIDELOC again and again, keeping the

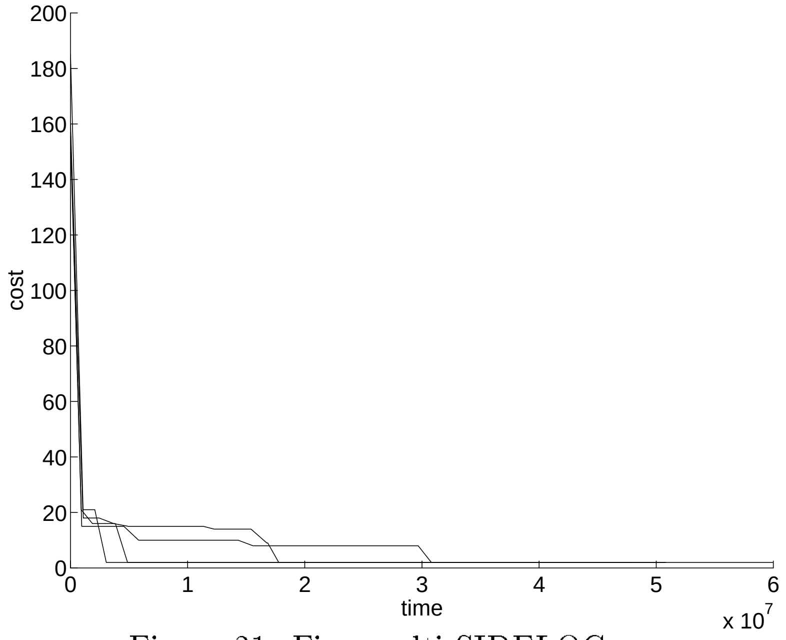

Figure 21: Five multi-SIDELOC runs.

best solution found in memory (call this algorithm multi-SIDELOC). The algorithm gave quite good solutions fast, but because each individual SIDELOC run failed to reach a global optimum, the resulting multi-SIDELOC reached a global optimum neither. In Figure 21 there are five multi-SIDELOC runs, each reaching cost function value 2. Then we tried to show that some of the algorithms SA, THRESHOLD, DELUGE, REC, PROBTABU or TABU *can* reach the global optimum.

The SA runs are plotted in Figure 22. The simulated annealing was run five times with rather high initial temperature and really slow cooling schedule. Notice how fast the runs go to the global optimum after some preliminary iterations. The time of reaching the global optimum might be improved, if the initial temperature were selected lower, but in [25] it was noted that too low initial temperature gives worse final solutions, so this potential improvement may not be very large. All of the five runs except one resulted in better final solutions than multi-SIDELOC in the same time. A global optimum was found in two runs. This shows clearly that in some problem instances SA indeed can give better final solutions than multi-SIDELOC, if there is enough time.

The threshold accepting runs are plotted in Figure 23. They did not perform significantly better than a single SIDELOC run, even though the threshold accepting algorithm was given considerably more time. THRESHOLD did not give as good final costs as multi-SIDELOC.

The the great deluge algorithm runs are plotted in Figure 24. A global optimum was found in three runs and the remaining two runs reached cost

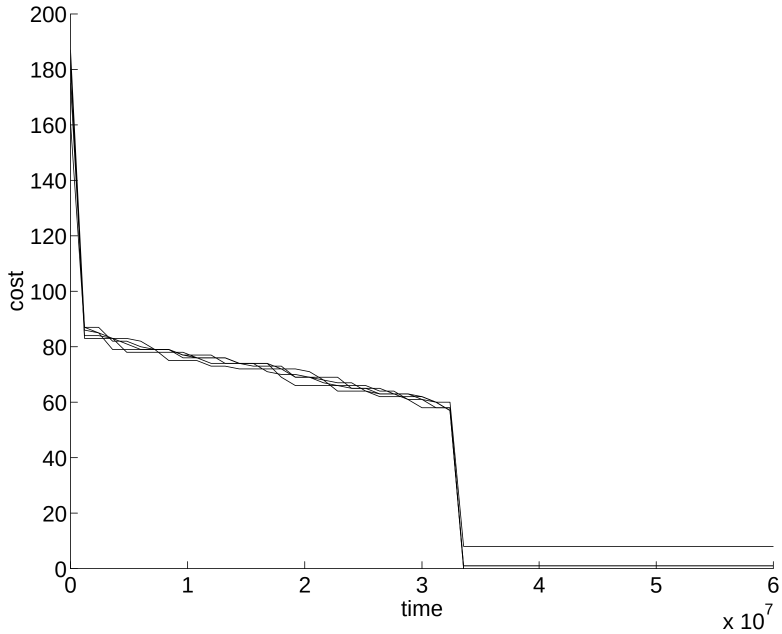

Figure 22: Five SA runs.

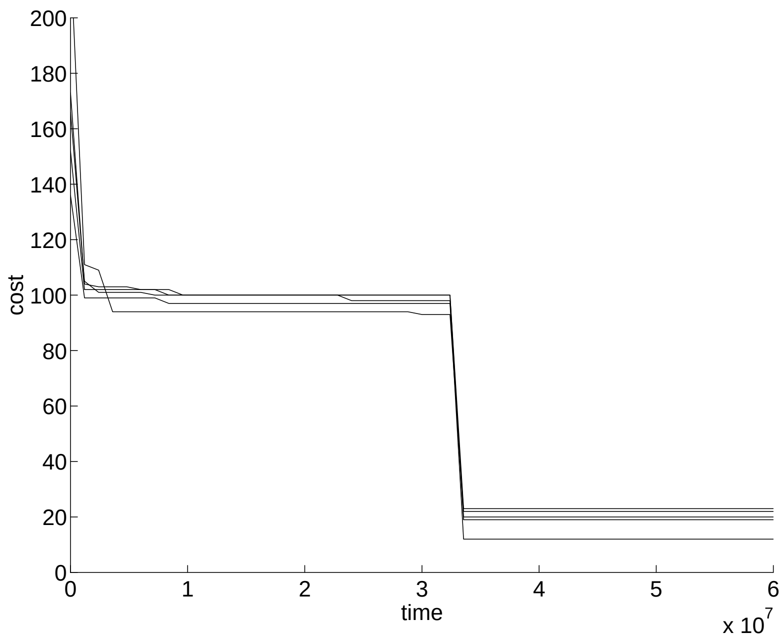

Figure 23: Five THRESHOLD runs.

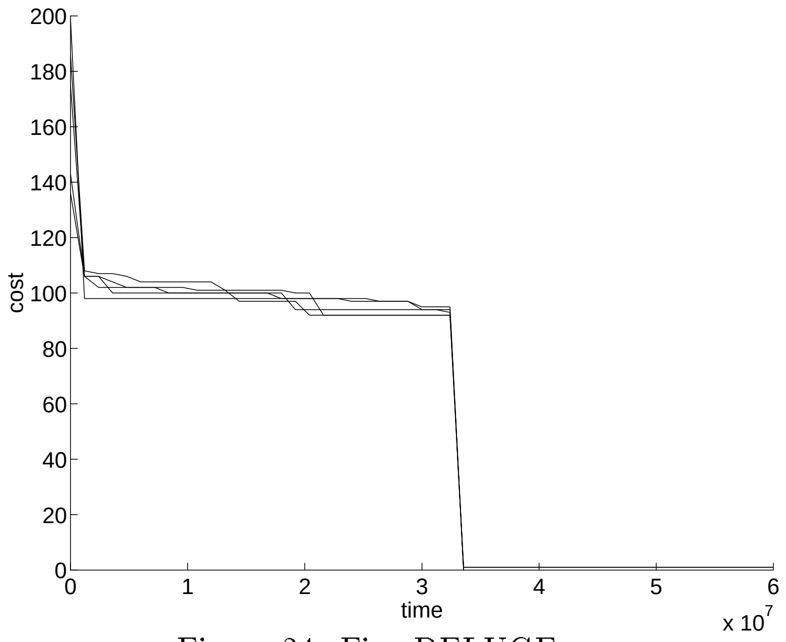

Figure 24: Five DELUGE runs.

value 1, which is quite good performance when compared to that of simulated annealing. DELUGE gave better final costs than multi-SIDELOC.

The record-to-record travel runs are plotted in Figure 25. A global optimum was found with every run much faster than with DELUGE or SA. REC performed much better than multi-SIDELOC. It may be the case that even this testcase was a little too easy, REC did not show similar superiority to SA and DELUGE when trying to find some considerably more difficult designs.

The runs of the probabilistic tabu search are plotted in Figure 26. We experimented with several different tabu list lengths, however the metaoptimizing techniques could not be used because the computational demands were so high. These PROBTABU runs did not give as good final costs as TABU runs, which indicates that the neighborhood fraction LF, see Section 3.4, was too small. No global optima were found, unlike with TABU, which resulted in three runs finding a global optimum. The TABU runs are plotted in Figure 27.

Because this test was not as extensive as with the first testcase, it is not possible to draw any conclusions about the mutual superiority of SA, DELUGE, REC, PROBTABU and TABU. It seems very hard to find a parameter set in the problems in this work such that the problem would be hard enough resulting in some differences between the non-elementary algorithms and yet easy enough so that it would be possible to conclude extensive comparison calculations. However, it was shown that SA, DELUGE, REC and TABU can perform better than multi-SIDELOC, even though SIDELOC was essentially the best optimization algorithm in the first testcase. The performance

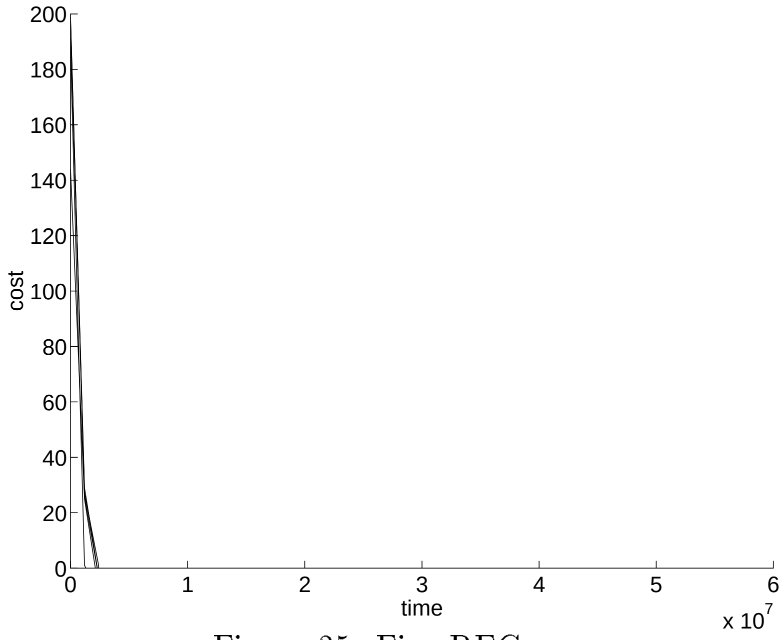

Figure 25: Five REC runs.

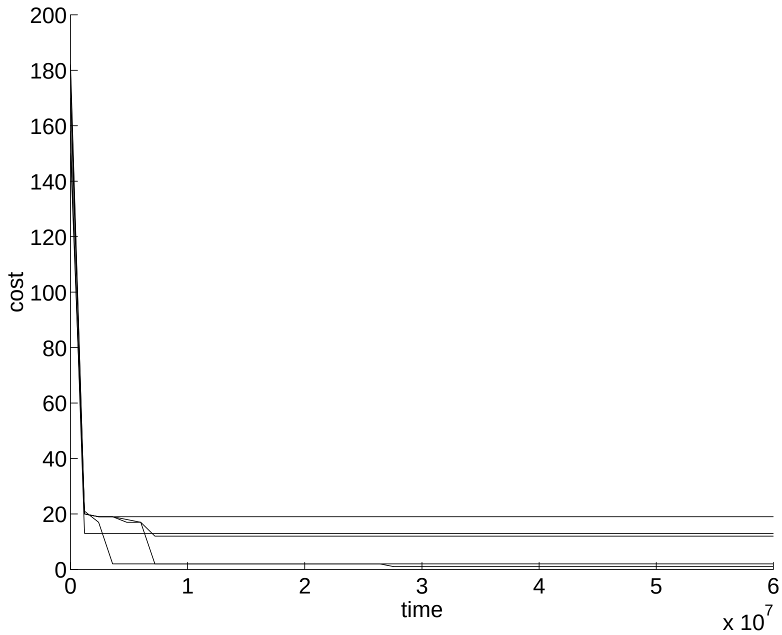

Figure 26: Five PROBTABU runs.

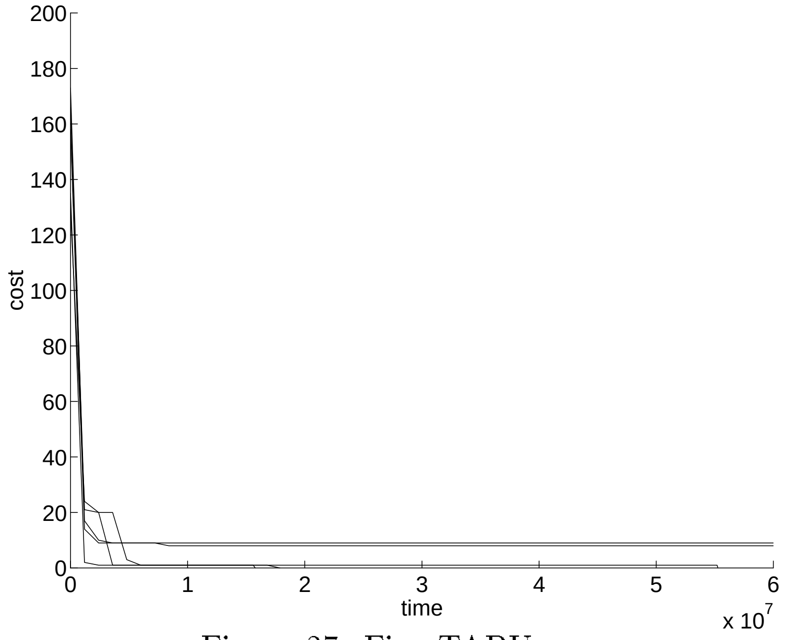

Figure 27: Five TABU runs.

of threshold accepting was a slight disappointment, it did not reach much better final solutions than a single SIDELOC run even when it was given considerably much more time. This may very well have something to do with the problem characteristics. The cost function is highly discontinuos and contains level areas. As was mentioned in Section 3.5, we did not implement an adaptive threshold lowering sequence as in [17], the performance of THRESHOLD would probably be better with such an implementation. This improvement may not be very large, however, because in this test the threshold was lowered very slowly and the final costs were still rather high.

The differences between the performance of various algorithms may seem small, but in this work a minor increase in the final cost indicates that a global optimum (a design) was not found.

6 Tables of Results

In this section we summarize the results of the covering, packing and t-design searches.

6.1 Upper Bounds for Covering Designs

In Tables 3 through 10, upper bounds and exact values for $C(v,t,k,t)$, $k \leq 9$, $v \leq 13$ are given. Upper bounds are preceded by $\leq$. The new upper bounds were determined by explicitly constructing corresponding designs, which are listed in Appendix A. Previously known upper bounds, $C(11,6,7,6) \leq 84$ and $C(12,7,8,7) \leq 126$ [10, 9], are preceded by $\leq$ and typeset in *italic* (we reached these upper bounds in this study, too, but we could not improve them). In addition to these two values, an upper bound $C(12,6,7,6) \leq 176$ can be found in [19], the best bound we reached was 177 (the rest of the upper bounds in their paper are equal to our bounds, except 12 cases, where our bounds are better). Some of the upper bounds for $k=6$, $k=7$, and $k=8$ have also been proved by different constructors of lottery systems. We do not know whether the designs found by us are isomorphic to these systems.

The exact values have been gathered from [37], which summarizes some known exact values for $C(v,t,k,t)$. Some of the exact values listed here are outside the boundaries of the parameters in [37]. These values have been obtained as follows.

The trivial result $C(v,t,v,t) = 1$ gives the upmost diagonal in every table. The result $C(v,t,v-1,t) = t+1$ is easy to prove and that gives us the second diagonals. In 1941 Turán [52] solved a graph theoretic problem which gives us the proof that $C(v,t,v-2,t)$ meets the Schönheim lower bound. That result gives us the third diagonal in our tables. It is trivial that $C(v,1,k,1) = \lceil \frac{v}{k} \rceil$, so we readily have the values for the first columns in our tables. The exact values in the second and third columns follow from the results in [34, 35, 36, 46, 50]; see also [37, Tables 4.1 and 4.2]. The entry $C(11,4,6,4) = 32$ is now known to be exact, because we found a corresponding design and it has been shown that $32 \leq C(11,4,6,4) \leq 33$ [4]. The value $C(11,6,8,6) = 29$ is from [7]. Other exact values are $C(12,6,9,6) = 22$ and $C(13,3,7,3) = 13$, obtained in [38] and [51], respectively.

The designs that give the new upper bounds in these tables are listed in Appendix A in a compressed notion. The designs can be retrieved in the following way: first form all possible k-subsets out of v points and sort them lexicographically in a list. Now let the compressed notion of the design be $s_1, s_2, \ldots, s_b$, the design containing b blocks. The first block of the design is the (s_1+1)th entry in the list. The ith block can be found by taking the position of the $(i-1)$th block in the list and moving forward $s_i + 1$ positions. In this compressed notation, the sequence $a, a, \ldots, a$ (j times) is abbreviated as a^j.

	m,k,t							
v	$1,2,1$	$2,3,2$	$3,4,3$	$4,5,4$	$5,6,5$	$6,7,6$	$7,8,7$	$8,9,8$
2	1							
3	2	1						
4	2	3	1					
5	3	4	4	1				
6	3	6	6	5	1			
7	4	7	12	9	6	1		
8	4	11	14	20	12	7	1	
9	5	12	25	30	30	16	8	1
10	5	17	30	≤ 51	50	45	20	9
11	6	19	47	66	≤ 100	$\leq$ *84*	63	25
12	6	24	57	113	132	$\leq$ *176*	$\leq$ *126*	84
13	7	26	78	≤ 157	245	≤ 264	≤ 297	≤ 185

Table 3: Values of $C(v,k-1,k,k-1)$, $k \leq 9$, $v \leq 13$.

	m,k,t						
v	$1,3,1$	$2,4,2$	$3,5,3$	$4,6,4$	$5,7,5$	$6,8,6$	$7,9,7$
3	1						
4	2	1					
5	2	3	1				
6	2	3	4	1			
7	3	5	5	5	1		
8	3	6	8	7	6	1	
9	3	8	12	12	9	7	1
10	4	9	17	20	20	12	8
11	4	11	≤ 20	32	≤ 34	29	15
12	4	12	≤ 29	≤ 41	≤ 59	≤ 51	≤ 40
13	5	13	≤ 34	≤ 66	≤ 88	≤ 104	≤ 79

Table 4: Values of $C(v,k-2,k,k-2)$, $k \leq 9$, $v \leq 13$.

	m, k, t					
v	$1,4,1$	$2,5,2$	$3,6,3$	$4,7,4$	$5,8,5$	$6,9,6$
4	1					
5	2	1				
6	2	3	1			
7	2	3	4	1		
8	2	4	4	5	1	
9	3	5	7	6	6	1
10	3	6	10	10	8	7
11	3	7	11	≤ 17	16	10
12	3	9	≤ 15	≤ 24	≤ 27	22
13	4	10	≤ 21	≤ 30	≤ 43	≤ 41

Table 5: Values of $C(v, k-3, k, k-3)$, $k \leq 9$, $v \leq 13$.

	m, k, t				
v	$1,5,1$	$2,6,2$	$3,7,3$	$4,8,4$	$5,9,5$
5	1				
6	2	1			
7	2	3	1		
8	2	3	4	1	
9	2	3	4	5	1
10	2	4	6	5	6
11	3	6	8	9	7
12	3	6	≤ 11	12	12
13	3	7	13	≤ 19	≤ 21

Table 6: Values of $C(v, k-4, k, k-4)$, $k \leq 9$, $v \leq 13$.

	m, k, t			
v	$1,6,1$	$2,7,2$	$3,8,3$	$4,9,4$
6	1			
7	2	1		
8	2	3	1	
9	2	3	4	1
10	2	3	4	5
11	2	4	5	5
12	2	5	6	≤ 8
13	3	6	≤ 10	≤ 10

Table 7: Values of $C(v, k-5, k, k-5)$, $k \leq 9$, $v \leq 13$.

	m,k,t		
v	$1,7,1$	$2,8,2$	$3,9,3$
7	1		
8	2	1	
9	2	3	1
10	2	3	4
11	2	3	4
12	2	3	4
13	2	4	6

Table 8: Values of $C(v, k-6, k, k-6)$, $k \leq 9$, $v \leq 13$.

	m,k,t	
v	$1,8,1$	$2,9,2$
8	1	
9	2	1
10	2	3
11	2	3
12	2	3
13	2	3

Table 9: Values of $C(v, k-7, k, k-7)$, $k \leq 9$, $v \leq 13$.

	m,k,t
v	$1,6,1$
9	1
10	2
11	2
12	2
13	2

Table 10: Values of $C(v, k-8, k, k-8)$, $k \leq 9$, $v \leq 13$.

$D(v,m,k,t)$	[8]	our best
$D(12,4,5,4)$	≥ 80	≥ 80
$D(13,4,5,4)$	≥ 123	≥ 123
$D(13,5,6,5)$	≥ 166	≥ 164
$D(14,4,5,4)$	≥ 169	≥ 163
$D(14,5,6,5)$	≥ 278	≥ 278
$D(15,5,7,5)$	≥ 69	≥ 66
$D(16,5,7,5)$	≥ 109	≥ 97

Table 11: Packing designs.

6.2 Lower Bounds for Packing Designs

We are mainly interested in the case $\lambda = 1$ because of its important applications in coding theory. In [37] there are some known exact values of $D(v,t,k,t)$. In addition to that, there are extensive tables of best known constant weight error-correcting codes in [8]. We selected seven packing designs whose exact maximal sizes are not known and tried to break the known records. These seven packing designs are tabulated in Table 11 with the lower bounds obtained from [8] and the best lower bounds that we achieved. We did not manage to improve any of the lower bounds although we reached some of them. It seems that the cost function (31) is not competitive with a Hamming distance based cost function in the case $\lambda = 1$. Our bounds were achieved by searching for designs with one more block than the best known lower bound. A final cost c when the number of blocks is b gives a lower bound $b - c$, because if an m-subset is covered by more than one blocks, we can remove one of the blocks and the cost (31) decreases at least by one.

6.3 t-designs

In search for t-designs we concentrated on the cases in Table 12, which was obtained from [29]. It is not known, if any of the designs in the table exists, so finding one of them would be a remarkable result. The column SIDELOC in Table 12 contains the best cost that was achieved with three runs of the probabilistic local optimization algorithm where sideway moves were allowed (see Figure 4 and Section 5). The cost function and the neighborhood structure were the same as with covering or packing designs.

None of the designs was found with SIDELOC, none of them seemed even very promising. The most promising cases were $4-(15,5,2)$ and $4-(12,6,6)$, but even their final cost was rather high. However, we decided to do a few long runs with DELUGE and REC searching for these two designs because those algorithms had performed quite well in the performance comparisons. The

$t-(v,k,\lambda)$	b	SIDELOC	DELUGE	REC
$3-(16,7,5)$	80	127		
$4-(17,5,1)$	476	133		
$4-(15,5,2)$	546	76	48	50
$5-(16,6,2)$	1456	384		
$6-(15,7,3)$	2145	620		
$7-(16,8,3)$	4290	1782		
$4-(12,6,6)$	198	94	80	71

Table 12: t-design candidates.

final costs achieved that way are in columns DELUGE and REC, respectively. Those values were considerably better than those of SIDELOC, but remained so far away from zero that it seems improbable that these two designs (if they exist) could be find by the methods in this work.

7 Conclusions

In this work we have searched combinatorial designs using local search strategies. The aim was twofold—to find some better designs than the previously known ones and to compare the performance of the search algorithms.

The implementation of the neighborhood structure and cost function used in the algorithms in this work rely on extensive tabulation of data. The space complexity thus sets limits on the parameter range for the designs, although with suitable parameter values (for example when the number of the blocks in the design is large) the time demand takes over. Thus, while we reached many covering designs which are better than the previously known ones, some of the new covering designs with many blocks are probably not minimal covering designs.

In general, the approach we used in this work gave most new results on covering designs. This is perhaps because covering designs have not been researched as extensively as error-correcting codes or t-designs. In addition to that, covering designs tend to have less structure than packing or t-designs, which makes it difficult to develope good non-probabilistic construction methods.

As a comparison, there are 19 entries in the tables in [19] which fall inside the range of Tables 3 through 10 and where the exact size of the corresponding minimal covering design is not known. They used many different construction methods to obtain upper bounds for these designs, but only in one of these 19 cases their bound was better than ours while our bounds were better in as many as 12 cases. This indicates that there are not very good construction methods for covering designs available today and when the parameters of the design are not too large, these existing construction methods can be defeated by probabilistic search methods. The success with packing designs (with $\lambda = 1$ and $m = t$) or t-designs was not as high, because the time and space demands of our approach grow so rapidly. However, our approach can be used in search for packing designs defined in a very general way, it is not limited to the case $\lambda = 1$ and $m = t$.

We managed to give a simple one-to-one correspondence between covering and packing designs (Theorems 8 and 9). We hope that this connection can be used when comparing those known covering design and packing design construction methods that can be generalized into our notation.

It was hoped that the cost function based on the number of covered sets would work in some cases better than Hamming distance based cost functions when searching for packing designs, but it turned out that even the smallest unsettled packing designs with $\lambda = 1$ are so large that this approach cannot compete very well with those distance-based cost functions. However, the good performance of some of the recently invented search algorithms (mainly the great deluge algorithm, record-to-record travel and perhaps the tabu search algorithm), when compared to simulated annealing, give raise to further research in the field of packing designs. The smallest unsettled t-designs tend to be so large that these techniques can be applied only to a few of them. It seems that those few designs (if they exist) are very hard to find with our approach.

It is possible to construct record-breaking combinatorial designs by computer searches in order to find out new construction methods. Many construction methods have been invented in this way, although the role of computer is seldom mentioned. We can also go in the other direction and evaluate proposed construction methods. For example, in [12] a method is introduced that is used to show that $C(15,3,6,3) \leq 45$. We were able to prove that $C(15,3,6,3) \leq 31$ by finding a corresponding covering design. The search for this design took only approximately one minute of CPU time on a SUN SPARCstation 2. This indicates that the method presented in [12] does not work very well, at least not for these parameters.

The performance comparisons indicate that local optimization with sideway moves allowed works best with small parameter values—this also goes in accordance with the results in [48]. However, when parameters of the design grow, then this simple local optimization algorithm fails to give good final solutions. It is then that the more sophisticated algorithms can be applied successfully. Simulated annealing performed quite well in the comparisons, although it seems that in some cases the great deluge algorithm or record-to-record travel can give better final solutions with fewer computations. The difficulty in the comparisons was the great number of computations required when searching for a difficult design. That is why we cannot say which algorithm works best with large designs. However, the very good performance of the record-to-record travel and the great deluge algorithm gives another subject for further study. It is not clear if the good performance of these algorithms is related to the highly discontinuos nature of the cost functions common in coding theoretic problems. It would be very interesting to compare the performance of these algorithms to that of simulated annealing in e.g. a large TSP problem. There are some such comparisons in [18], but they do not indicate what kind of meta-optimizing techniques they used—the methods in Section 5.3 might just be computationally feasible.

As indicated earlier, there is a point when the space and/or time requirements make the usage of our approaches in searches for combinatorial designs 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.

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] E. Aarts and P. J. M. van Laarhoven, Local search in coding theory, *Discrete Math.* **106** (1992), 11–18.

[3] I. Althöfer and K.-U. Koschnick, On the convergence of 'threshold accepting,' *Appl. Math. Optim.* **24** (1991), 183–195.

[4] J. A. Bate and G. H. J. van Rees, Some results on $N(4,6,10)$ and $N(4,6,11)$ and related coverings, *Congr. Numer.* **48** (1985), 25–45.

[5] 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.

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

[7] E. D. Boyer, D. L. Kreher, S. P. Radziszowski, J. Zou and A. Sidorenko, On $(n,5,3)$-Turan systems, preprint.

[8] Shearer, N. J. A. Sloane and W. D. Smith, A new table of constant weight codes, *IEEE Trans. Inform. Theory* **36** (1990), 1334–1380.

[9] D. de Caen, D. L. Kreher and J. Wiseman, On constructive upper bounds for the Turán numbers $T(n,2r+1,2r)$, *Congr. Numer.* **65** (1988), 277–280.

[10] D. de Caen, D. L. Kreher, W. H. Mills and S. P. Radziszowski, On the covering of t-sets with $(t+1)$-sets: $C(9,5,4)$ and $C(10,6,5)$, *Discrete Math.* **92** (1992), 65–77.

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

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

[13] G. D. Cohen, M. G. Karpovsky, H. F. Mattson Jr. and J. R. Schatz, Covering radius—survey and recent results, *IEEE Trans. Inform. Theory* **31** (1985), 328–343.

[14] F. Comellas and R. Roca, Using genetic algorithms to design constant weight codes, preprint.

[15] D. Costa, A tabu search algorithm for computing an operational timetable, to appear in *European J. Oper. Res.*

[16] J. H. Dinitz and D. R. Stinson, A brief introduction to design theory, in J. H. Dinitz and D. R. Stinson (Eds.), "Contemporary Design Theory: A Collection of Surveys," Wiley, New York, 1992, pp. 371–399.

[17] G. Dueck and T. Scheuer, Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing, *J. Comput. Phys.* **90** (1990), 161–175.

[18] G. Dueck, New optimization heuristics: the great deluge algorithm and the record-to-record travel, *J. Comput. Phys.* **104** (1993), 86–92.

[19] T. Etzion, V. Wei and Z. Zhang, Bounds on the sizes of constant weight covering codes, preprint.

[20] 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.

[21] F. Glover, Tabu search—Part I, *ORSA J. on Computing* **1** (1989), 190–206.

[22] F. Glover, Tabu search—Part II, *ORSA J. on Computing* **2** (1990), 4–32.

[23] D. Goldberg, "Genetic Algorithms in Search, Optimization & Machine Learning," Addison-Wesley, Reading, 1989.

[24] J. J. Grefenstette, Incorporating problem specific knowledge into genetic algorithms, in L. Davis (Ed.), "Genetic Algorithms and Simulated Annealing," Pitman, London, 1987, pp. 42–60.

[25] 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.

[26] S. M. Johnson, Upper bounds for constant weight error correcting codes, *Discrete Math.* **3** (1972), 109–124.

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

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

[29] E. S. Kramer, private communication.

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

[31] R. Mathon, Computational Methods in Design Theory, in "Surveys in Combinatorics," Proc. 13th Br. Comb. Conf., Guildford/UK 1991, *London Math. Soc. Lecture Note Ser.* **166**, London, 1991, pp. 101–117.

[32] R. Mathon, Nonisomorphic designs by simulated annealing, preprint.

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

[34] W. H. Mills, On the covering of pairs by quadruples I, *J. Combin. Theory Ser. A* **13** (1972), 55–78.

[35] W. H. Mills, On the covering of pairs by quadruples II, *J. Combin. Theory Ser. A* **15** (1973), 138–166.

[36] W. H. Mills, Covering designs I: Coverings by a small number of subsets, *Ars Combin.* **8** (1979), 199–315.

[37] 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.

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

[39] K. J. Nurmela and P. R. J. Östergård, Constructing covering designs by simulated annealing, Report B10, Digital Systems Laboratory, Helsinki University of Technology, 1993.

[40] K. J. Nurmela and P. R. J. Östergård, Upper bounds for covering designs by simulated annealing, to appear in *Congr. Numer.*

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

[42] Y. G. Saab and V. B. Rao, Combinatorial Optimization by Stochastic Evolution, *IEEE Trans. Comput. Aided Design* **10** (1991), 525–535.

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

[44] J. Schönheim, On maximal systems of k-tuples, *Studia Sci. Math. Hungar.* **1** (1966), 363–368.

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

[46] R. G. Stanton and J. G. Kalbfleisch, Coverings of pairs by k-sets, *Internat. Conf. on Combin. Math. Ann. NY Acad. Sci.* **175**, (1970), 366–369.

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

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

[49] L. Teirlinck, Non-trivial t-designs without repeated blocks exist for all t, *Discrete Math.* **65** (1987), 301–311.

[50] D. T. Todorov, On the covering of pairs by 13 blocks, *C. R. Acad. Bulgare Sci.* **38** (1985), 691–694.

[51] D. T. Todorov and V. D. Tonchev, On some coverings of triples, *C. R. Acad. Bulgare Sci.* **35** (1982), 1209–1211.

[52] P. Turán, Eine Extremalaufgabe aus der Graphentheorie, *Mat. Fiz. Lapok* **48** (1941), 436–452.

Appendix A: Covering Designs

In the following we list the covering designs that give the upper bounds in Tables 3 through 10. The designs are listed in compressed notation that is explained in Section 6.

$C(10, 4, 5, 4) \leq 51$
3, 1, 3, 1, 7, 4, 1, 7, 2, 4, 8, 0, 3, 9, 2, 4, 0, 8, 4, 5, 4, 2, 0, 7^2, 1, 4, 5^2, 2, 7, 2, 3, 6, 3, 2, 7, 2, 5^2, 7, 0, 2, 4, 5, 2^2, 8, 2, 9, 5

$C(11, 3, 5, 3) \leq 20$
15, 5, 37, 25, 11, 23, 21, 36, 2, 22, 17, 34, 39, 16, 23, 1, 43, 23, 8, 38

$C(11, 4, 6, 4) \leq 32$
9, 7, 27, 0, 20, 6, 20, 26, 1, 14, 2, 15, 30, 3, 17^3, 13, 7, 3, 7, 26, 9, 26, 4, 5, 20, 6, 45, 5^2, 17

$C(11, 5, 6, 5) \leq 100$
5, 4, 2, 0, 3, 5, 3, 4, 7, 0^2, 11, 3, 4, 6, 0^2, 9, 1, 0, 5, 11, 3, 0, 8, 1, 12, 0, 1, 3, 6, 12, 0^2, 2, 6^3, 0, 5, 0, 4, 0, 6, 2, 9, 1, 2, 3, 8, 0, 1, 3, 0, 6, 0, 6, 9, 6, 3^2, 2, 9, 3, 0^2, 1, 8, 1, 7, 4, 0, 14, 0, 4, 0^2, 6, 0, 8, 3, 4^2, 11, 0^2, 4, 0, 7, 0, 4, 1, 5, 10, 3, 0, 5, 2^2, 3

$C(11, 4, 7, 4) \leq 17$
5, 7, 23, 28, 25, 21, 26, 7, 31, 2, 41, 12, 11, 30, 9, 10, 19

$C(11, 5, 7, 5) \leq 34$
8, 20, 0, 8^2, 3, 2, 16, 11, 0, 9, 10, 13, 17, 7^2, 11, 4^2, 12, 3, 20, 2, 8, 10, 20, 7, 4, 7, 6, 11, 15, 3, 7

$C(12, 3, 5, 3) \leq 29$
9, 17, 12, 41, 86, 31, 40, 6, 69, 5, 15, 7, 8, 29, 12, 19, 51, 17, 40, 6, 41, 21, 6, 11, 15, 49, 20, 2, 77

$C(12, 3, 6, 3) \leq 15$
70, 24, 78, 98, 73, 32, 60, 51, 17, 55, 22, 76, 64, 77, 84

$C(12, 4, 6, 4) \leq 41$
16, 30, 6, 31, 24, 20, 24, 26, 16, 14, 18, 14, 25, 37, 18, 14, 20, 34, 4, 16, 21, 26, 21, 16, 5, 18, 37, 33, 11, 49, 14, 18, 14, 16, 26, 46, 6, 24, 32, 13, 14

$C(12, 3, 7, 3) \leq 11$
95, 61, 30, 32, 44, 96, 44, 50, 113, 127, 64

$C(12, 4, 7, 4) \leq 24$
32, 4, 75, 19, 38, 61, 24, 13, 1, 56, 26, 58, 29, 6^2, 31, 14, 47, 10, 58, 27, 26, 27, 72

$C(12, 5, 7, 5) \leq 59$
16, 18, 0, 15, 29, 4, 8, 14, 8, 13, 28, 7^2, 26, 36, 9, 5, 17, 6, 4, 9, 10, 21, 14, 4,

12, 11, 15, 1, 26, 8, 15, 2, 13, 8, 14, 6, 20, 3, 11, 15, 7^2, 9, 13, 11, 7, 14, 29, 3, 4, 5, 16, 10, 9, 4, 24, 15, 36

$C(12,6,7,6) \leq 177$
4, 2^2, 3, 1, 0, 5, 7, 4, 3, 5, 2^2, 3^2, 6, 3, 6, 3, 5, 0^2, 8, 2, 6, 0, 3, 2, 3, 7, 1, 7, 5, 0, 1, 12, 0^2, 4, 6, 4, 0, 2, 4, 2, 6, 7^2, 4, 0, 2, 0, 3^2, 1, 6, 0^2, 14, 5, 1, 2^3, 4, 0, 6, 8, 0, 4, 3, 16, 0^2, 1, 5, 0^2, 1, 5, 3, 9, 2, 4, 5, 6, 11, 1, 2^2, 5, 0, 5, 3^2, 2, 0, 10, 5, 0, 5, 2, 5, 6, 0^2, 2, 1, 8, 4, 6, 2^2, 3, 2, 7, 0, 4, 0, 5, 3, 9, 3, 0, 8, 7, 9, 7, 0, 2, 1, 0, 1, 3, 5, 4, 10, 2, 1, 4^2, 1, 4, 3, 2, 3^2, 0, 10, 1^2, 2, 3, 11, 0, 3, 0, 5, 3^2, 2, 4, 1, 2, 7^2, 3, 6, 0, 5, 1, 3, 1, 8, 5, 10, 1

$C(12,5,8,5) \leq 27$
16, 9, 6, 32, 13, 3, 29, 13, 7, 31, 4, 21, 10, 30, 39, 18, 29, 9, 18, 19, 25, 10, 2, 40, 8^2, 10

$C(12,6,8,6) \leq 51$
7, 3, 7^2, 13, 4, 12, 8, 2, 29, 1, 12, 10, 2, 9, 13, 4, 9, 7, 2, 13, 11, 2, 17, 3, 7, 24, 1, 3, 30, 2, 14, 9, 11, 1, 11, 12, 5, 2^2, 10^2, 5, 14, 4, 2, 23, 1, 10, 3, 17

$C(12,4,9,4) \leq 8$
13, 17, 19, 28, 2, 38, 33, 26

$C(12,7,9,7) \leq 40$
1, 6, 4^2, 6, 4, 0, 3, 6, 8, 5, 0, 11, 3, 0, 5, 3, 12, 1, 18, 6, 0, 2, 4, 9, 0, 2, 1, 5, 2, 3, 4^2, 2, 4, 6, 7^2, 4^2

$C(13,3,5,3) \leq 34$
76, 38, 12, 64, 21, 0, 67, 4, 47, 29, 19, 25, 76, 12, 40, 41, 1, 43, 42, 37, 76, 11, 38, 86, 18, 46, 2, 29, 24, 12, 78, 26, 9, 101

$C(13,4,5,4) \leq 157$
2, 3, 12, 10, 4, 5, 7, 9, 17, 3, 10^2, 12, 8, 4, 2, 14, 2, 9, 13, 12, 4, 6, 13, 2^2, 19, 0, 11, 0, 23, 0, 5, 4, 16, 8, 18, 0^2, 3, 6^2, 2, 5, 29, 0, 2, 4, 8, 3, 4, 14, 10, 13, 8, 3, 6, 4, 16, 0, 3, 12, 6, 3, 7^2, 6, 8^2, 6, 2, 5, 3, 16, 10, 6, 8, 0, 10, 5, 7, 10, 7, 6^2, 5, 3, 6^2, 3, 11, 7, 20, 2, 9, 5, 8, 4, 23, 0, 5^2, 13, 15, 7, 5, 11, 13, 6, 17, 6, 0, 7, 12, 11, 5, 16, 1, 0, 13^2, 3, 17, 0^2, 1, 5, 0, 12, 3, 6, 1, 2^2, 21, 5, 10^2, 2^2, 6, 3, 12, 5, 7, 9, 6, 9, 1, 2, 24, 2, 3, 24, 1^2, 6

$C(13,3,6,3) \leq 21$
81, 5, 102, 43, 156, 42, 66, 38, 117, 66, 103, 43, 63, 81, 189, 39, 89, 112, 103, 31, 92

$C(13,4,6,4) \leq 66$
23, 5, 17, 57, 22, 31, 25, 2, 19, 5, 6, 55, 14, 81, 4, 12, 0, 76, 87, 0, 12, 4, 20, 0, 12, 4, 115, 25, 2, 19, 113, 2, 14, 6, 8, 20, 1, 17, 6, 140, 8, 13, 3, 102, 20, 13, 3, 6, 19, 1, 15, 49, 3, 15, 65, 3, 11, 61, 7, 6, 32, 1, 24, 1, 52, 34

$C(13,4,7,4) \leq 30$
22^2, 82, 74, 4, 20, 7, 38, 110, 45, 29, 9, 82, 166, 29, 59, 71, 94, 79, 7, 91, 74,

40, 58, 146, 21, 55, 40, 24, 6

$C(13, 5, 7, 5) \leq 88$
4, 13, 39, 14, 9, 5, 2, 49, 11, 13, 39, 11, 0, 4, 3, 9, 17, 11, 17, 13, 6, 36, 4, 13, 44, 4, 35, 17, 26, 25, 22, 38, 4, 31, 9, 17, 18, 7, 49, 31, 3, 30, 23, 43, 6, 20^2, 26, 49, 13, 12, 8, 13, 31, 50, 2, 14, 7, 3, 51, 35, 11, 33^2, 3, 45, 0^2, 3^2, 21, 11, 8, 4, 24, 7, 28, 14, 17^2, 38, 5, 44, 32, 10, 4, 22, 9

$C(13, 6, 7, 6) \leq 264$
0, 14^2, 4, 11, 3, 2^3, 5, 7, 1, 0, 4, 15, 6^2, 0, 15, 3, 2, 3, 5, 2, 3, 2, 5, 1, 12, 3, 4, 3, 16, 3, 12, 4, 12, 9, 3, 6, 4, 19, 3, 2, 0, 2, 5, 7, 1, 8, 13, 1, 6, 1, 3, 8, 1, 0, 3, 7^2, 4, 3, 6, 17, 3, 5, 6, 4, 20, 6, 2, 12^2, 4, 3, 0, 5^2, 0, 8, 1, 7, 16, 0, 8, 7, 2, 1, 23, 0, 5, 8, 15, 2, 21, 0^4, 5, 1, 2, 5, 1, 4, 2^2, 12, 3, 5, 6, 3, 12, 4, 7, 4, 1, 8, 4, 14, 12, 1, 10, 5^2, 7, 8, 1, 2, 5, 1, 4, 2^2, 5, 0, 4, 9, 2, 1, 8, 14, 2, 7, 13, 4, 8, 12, 0, 3, 12, 9, 4, 5, 0^2, 2, 7, 14, 7^2, 2, 6, 16, 8, 9, 1, 5, 6, 1, 8, 10^2, 2, 14, 7, 0^2, 3, 7, 5, 9, 7, 4, 0, 2^2, 3, 2, 12, 3, 5^2, 4, 0, 5, 3, 2, 23, 1, 6, 10, 14, 0, 7, 6, 7, 4, 3, 6, 11, 8^2, 12, 9, 1, 0, 1, 2, 3, 9, 4, 1^2, 2, 11, 1, 8, 5, 3, 18, 6, 4^3, 2^2, 5, 3, 5, 6^2, 0, 4, 5, 4^2, 3, 9, 0, 16, 4^2, 1, 12, 9, 2, 10, 1, 8, 7, 5, 7

$C(13, 3, 8, 3) \leq 10$
61, 173, 172, 81, 9, 176, 146, 44^2, 286

$C(13, 4, 8, 4) \leq 19$
37, 7, 105, 111, 95, 48, 26, 138, 101, 48, 53, 61, 12, 77, 18, 70, 66, 24, 74

$C(13, 5, 8, 5) \leq 43$
24, 30, 19, 9, 13, 54, 17, 16, 30, 10, 4, 12, 29, 36, 74, 41, 37, 5, 29, 13, 36, 6, 33, 26, 41, 57, 29, 9, 36, 84, 21, 28, 27, 32, 50, 20, 39, 46, 4, 56, 3, 35, 0

$C(13, 6, 8, 6) \leq 104$
12, 14, 12, 14, 11^2, 10, 5, 14, 7, 9, 16, 11, 8, 17, 5, 21, 19, 9, 0, 5, 21, 23, 7, 2, 14, 4, 3, 7, 32, 3, 5, 31, 2, 44, 0, 12, 10, 0, 4, 6, 2, 19, 12, 18, 25, 4, 6, 10, 2, 13, 6, 9, 28, 21, 4^2, 14, 13, 25, 15, 6^2, 8, 14, 8, 10, 23, 8^2, 4^2, 11, 12, 22, 4, 1, 10, 14, 7, 12, 13, 18, 20, 23^2, 3, 4, 8, 0, 7, 2, 10, 15, 37, 14, 5, 8, 7, 24, 8, 23, 1, 3

$C(13, 7, 8, 7) \leq 297$
3, 5, 2, 3, 2, 7, 0, 2, 0, 8, 9, 5, 1^2, 3, 5, 2, 0, 7, 4, 2, 12, 0^2, 4, 3, 2, 3, 1, 3^2, 2^2, 3, 4, 3, 2, 4, 3, 2^3, 0^2, 7, 9, 0, 4, 1, 0, 1, 13, 5, 1, 3, 6, 3, 6, 0^2, 2, 1, 2^2, 4, 0, 1, 15, 0, 4, 3^2, 1^3, 8, 3, 10, 1, 4, 1, 4, 1, 2^2, 3^2, 2, 0, 10, 3, 1, 0, 4, 2, 1, 0, 10, 1, 8, 2, 1^2, 7, 2, 1, 2, 3, 1, 5, 2^2, 4, 10, 6, 8, 1, 2, 8, 1, 3, 1^2, 0, 4, 9, 2, 1^2, 6^2, 7, 1, 6, 2, 1, 5, 4, 3, 7, 5, 6, 0, 1^2, 0, 4, 1, 4, 2^2, 5^2, 4, 3, 4, 3, 4, 6, 0^2, 1, 10, 0^2, 2, 6, 5, 1, 10, 1, 2, 4, 6, 0^2, 1, 0, 5, 4, 7, 0^3, 7, 2, 9, 1, 2, 7, 11, 2, 7, 0^2, 4, 0, 2, 4, 8, 1, 21, 1, 4^2, 6, 1, 2, 1^3, 4, 3, 17, 1, 2, 3, 1, 0, 1, 4^2, 1^2, 6^2, 0, 1, 2, 1, 5, 0, 7, 9, 4, 7, 1, 3, 8, 2, 1, 9, 6, 7, 0, 1, 2, 8, 7, 6, 0^3, 3, 1, 0, 4, 1, 4, 3, 2, 4, 9, 4, 10, 5, 6, 0, 8, 0, 1^2, 2, 5, 2, 1, 12, 5, 1^2, 3, 1, 11, 4, 11, 9, 1, 2, 4, 0^4, 3, 2, 0, 4

$C(13, 4, 9, 4) \leq 10$

106, 14, 15, 39, 85, 97, 117, 27, 34, 84

$\boldsymbol{C(13,5,9,5) \leq 21}$
2, 29, 9, 72, 11, 18, 2, 133, 2, 21, 11, 70, 67, 4, 12, 25, 9, 15, 135, 13, 12

$\boldsymbol{C(13,6,9,6) \leq 41}$
10, 17, 7, 31, 13, 16, 8, 19, 23, 4, 25, 20, 44, 6, 3, 26, 11, 6, 2, 34, 7, 24, 8, 22, 4, 23, 18, 2, 9, 47, 31, 0, 6, 4, 37, 19^2, 4, 5, 46, 5

$\boldsymbol{C(13,7,9,7) \leq 79}$
9, 1, 13, 8, 10, 0, 2, 9, 11, 4, 2, 15, 8, 3, 24, 7, 4, 2, 21, 6, 13, 3, 11, 1, 11, 4, 5^2, 8, 3, 17, 13, 2, 6, 7, 4, 1, 8, 12, 7^2, 15, 3, 7^2, 3, 11, 4, 10, 5, 24, 11, 16, 7, 2, 4, 8^2, 7^2, 13, 9, 5, 4, 17, 2, 17, 0, 12, 15, 9, 3^2, 11, 13, 3, 5, 12, 4

$\boldsymbol{C(13,8,9,8) \leq 185}$
2, 1, 9, 0^3, 2, 5, 1, 10, 5, 0^2, 1, 0, 1, 2, 0, 10, 1, 5^2, 0, 4, 0, 4, 2^2, 0, 14, 4^2, 0^3, 1, 0, 1, 7, 4, 0, 2, 0, 5, 7, 0, 1^2, 10, 1^3, 8, 12, 0^2, 1, 2, 4, 0, 5, 4, 1, 4^2, 2, 3, 0, 9, 3, 5^4, 0^2, 1, 0, 1, 4, 0^2, 5, 0, 1, 2, 4, 9, 2, 7, 5, 1^2, 7, 6, 2, 1^2, 0, 1, 2, 6, 1, 5, 4, 2, 1, 0, 11, 1, 0, 4, 0, 4^2, 1, 3, 4^3, 1, 0, 4, 6, 2, 1, 5, 6, 4, 2^2, 6, 2, 0, 3, 2, 7, 1, 0, 3^2, 0^2, 4, 5, 0, 1, 8, 12, 0^2, 1, 2, 3, 4^2, 0^2, 2, 9, 0, 6, 4, 1, 12, 0^2, 4, 0, 1, 2, 8, 0, 1, 7, 1, 3^2, 8, 0, 3, 2, 1, 2^2

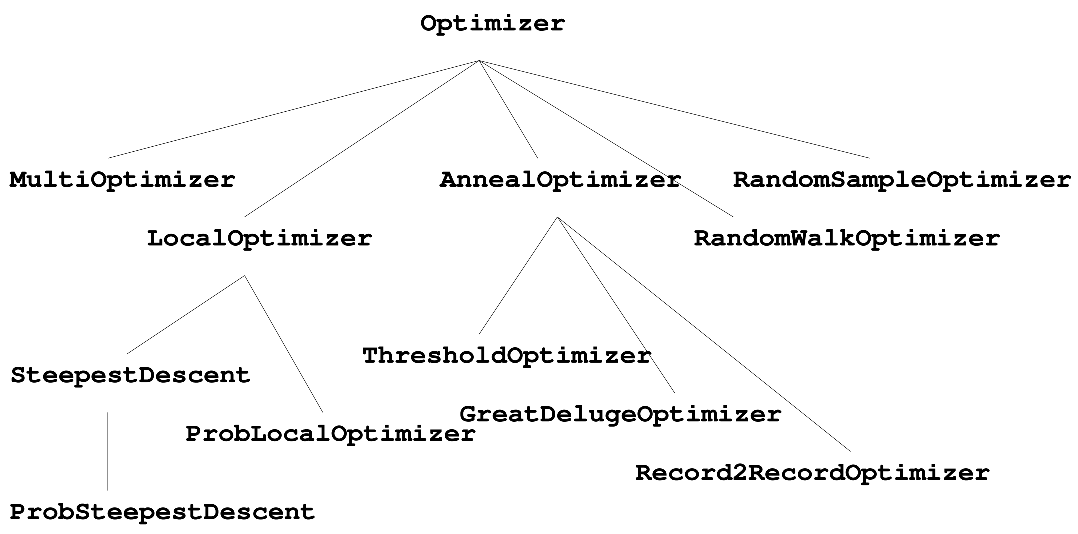

Figure 28: Optimizer class hierarchy.

Appendix B: Program Outline and Class Hierarchy

The program `opt` written in C++ during this research was used in search for the designs in this work. The program separates the optimization algorithms and the problem (search of covering designs, packing designs or t-designs). The optimization algorithms are each a class derived from the `optimizer` class. Each algorithm requires a corresponding problem instance derived from the `solution` class. For the meta-optimizing mentioned in Section 5.3 the class `optimizer` is derived from the `TotalEnumerationSolution` which is, in turn derived from `Solution`.

The reason why we decided to implement the program in this kind of a C++ class library was better data abstraction and module encapsulation. The optimizer algorithms are separated from the specific problems in this work so that adding a new optimization problem in the program causes little overhead.

The class hierarchy is presented in a graphical from in Figures 28 and 29. The derivations of the classes specific to the problems in this work are shown in dotted lines. The derivations of general purpose optimizer and solution classes are shown in solid lines. As is apparent from the class hierarchy graphs, virtual multiple inheritance is needed, because the class hierarchies are not trees, but directed acyclic graphs.

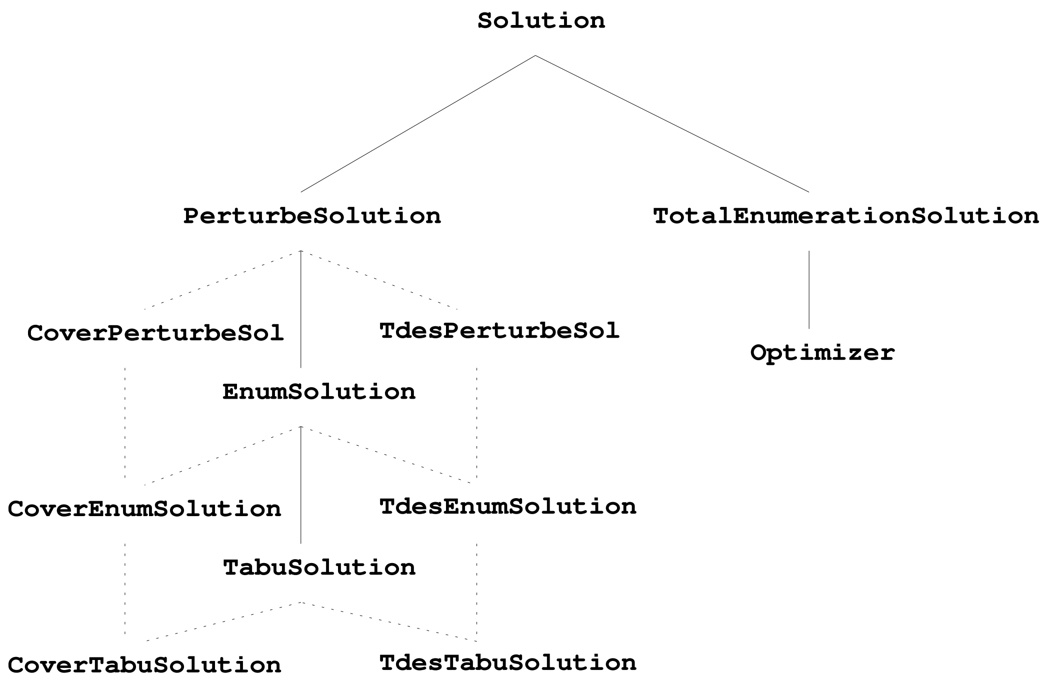

Figure 29: Solution class hierarchy.

B.1 Class Descriptions

In addition to the classes described below there are many other classes in the program that take care of the command line option parsing, the performance data recording, etc.

B.1.1 Optimizer Classes

`Optimizer`

The `Optimizer` class is an abstract base class for all the optimization algorithms. It defines some operators common for all the algorithms such as `GetBestFound` which returns the best solution that was found during the last optimization run and `Optimize` which starts a new optimization run.

`MultiOptimizer`

The `MultiOptimizer` class is an optimizer that takes an optimization algorithm and repeatedly starts new optimization runs until the algorithm has used the maximum amount of resources.

Optimizer implementations

The classes SteepestDescent, ProbSteepestDescent, ProbLocalOptimizer, AnnealOptimizer, TabuOptimizer, ProbTabuOptimizer, RandomSampleOptimizer, RandomWalkOptimizer, ThresholdOptimizer, GreatDelugeOptimizer and Record2RecordOptimizer implement the optimization algorithms in Section 3.

SteepestDescent, ProbSteepestDescent and ProbLocalOptimizer use the EnumSolution solution class, TabuOptimizer and ProbTabuOptimizer use the TabuSolution solution class and all the other optimizer classes use the PerturbeSolution solution class.

B.1.2 Generic Solution Classes

Solution

This is the abstract base class for all the solution classes. The methods in this class are those that are common to all solution classes and include methods like GetCost which returns the solutions current cost, Print which prints the solution, Store and Recall which store and recall the solution in and from the memory, respectively, GlobalOptimum which is true in global optimums and InitializeRandomSol which initializes the solution randomly.

TotalEnumerationSolution

TotalEnumerationSolution contains methods FirstSolution and NextSolution which set the solution to the first solution in the solution space and to the next solution in the solution space, respectively.

PerturbeSolution

PerturbeSolution contains methods PerturbeCost which changes the solution randomly, RealizePerturbation which makes the last perturbated solution into the current solution and NeighborhoodSize which returns the size of the neighborhood of the current solution.

EnumSolution

EnumSolution adds the possibility to check all the neighbors of the current solution via the methods EnumCost and RealizeEnum.

`TabuSolution`

`TabuSolution` adds the possibility to check if the proposed move breaks the tabu list condition.

B.1.3 Problem-specific Solution Classes

`CoverPerturbeSol`, `CoverEnumSolution` and `CoverTabuSolution` implement the neighborhood structure, cost function and tabu lists described in Sections 4.2 and 4.3 while `TdesPerturbeSol`, `TdesEnumSolution` and `TdesTabuSolution` implement the corresponding functions described in Section 4.4. All these classes use the `CoverTables` class, which provides the solution classes with the data tables in Figure 11.