



TEKNILLINEN KORKEAKOULU  
Department of Computer Science and Engineering

Esa Seuranen

## Asymmetric and Unidirectional Covering Codes

Master's thesis submitted in partial fulfillment of the requirements for the degree  
of Master of Science in Technology

Espoo, 11<sup>TH</sup> April 2005

Supervisor: Professor Markku Syrjänen  
Instructor: Professor Patric Östergård

<b>Tekijä:</b>	Esa Seuranen
<b>Osasto:</b>	Tietotekniikka
<b>Pääaine:</b>	Tietämystekniikka
<b>Sivuaine:</b>	Tiedonhallinta
<b>Työn nimi:</b>	Epäsymmetriset ja yksisuuntaiset peittokoodit
<b>English title:</b>	Asymmetric and Unidirectional Covering Codes
<b>Sivumäärä:</b>	ix + 47
<b>Professori:</b>	T-93 Tietämystekniikka
<b>Työn valvoja:</b>	Professori Markku Syrjänen
<p>Kombinatorinen optimointi on laaja tutkimuskohde, jolla on monia käytännön sovelluksia. Kombinatorista optimointia voi lähestyä monilla tunnetuilla perinteisillä menetelmillä, kuten kokonaislukuohjelmoinnilla, paikallisella haulilla tai täydellisellä haulilla – ratkaisten näin ongelman joko globaaliin optimiin (mikä saattaa olla varsin vaikeaa ja aikaa vievää) tai sitten riittävän hyvään ratkaisuun.</p> <p>Tässä työssä tarkastelemme kahta, epäsymmetristä ja yksisuuntaista, peittokoodiongelman muunnelmaa. Sovellamme kokonaislukuoptimointia, paikallishakua (tabuhaku) ja täydellistä hakua selvittämään (tai ainakin rajaamaan) epäsymmetrisien ja yksisuuntaisten peittokoodien kokoja. Keskustelemme perinteisten ratkaisumenetelmien mahdollisista heikkouksista ja ongelmista – sekä keinoista, joilla ne voisi kenties ratkaista käyttämällä apuna tietoutta käsiteltävänä olevasta ongelmasta. Esittelemme lyhyesti alustavia tuloksia joistakin mainituista keinoista. Lopuksi listaamme parhaimmat tiedossa olevat ala- ja ylärajat epäsymmetrisille ja yksisuuntaisille peittokodeille.</p>	
<b>Avainsanat:</b>	epäsymmetrinen, isomorfismi, kokonaislukuohjelmointi, peittokoodi, tabuhaku, täydellinen haku, yksisuuntainen

<b>Author:</b>	Esa Seuranen
<b>Department:</b>	Department of Computer Science
<b>Major:</b>	Knowledge Engineering
<b>Minor:</b>	Knowledge Management
<b>Title:</b>	Asymmetric and Unidirectional Covering Codes
<b>Number of pages:</b>	ix + 47
<b>Chair:</b>	T-93 Knowledge Engineering
<b>Supervisor:</b>	Professor Markku Syrjänen
<p>Combinatorial optimization is a wide area, which has many practical applications. There are many common approaches, such as integer programming and local search methods, which can be used to solve the combinatorial optimization problem at hand - either to a global optimum (possibly difficult and slow) or to a reasonably good solution.</p> <p>In this thesis we discuss two variants, asymmetric and unidirectional, of the combinatorial optimization problem known as the covering code problem. We apply conventional techniques - that is, integer programming, a local search method (tabu search) and exhaustive search - to solve, or at least to bound, the optimal solutions for asymmetric and unidirectional binary covering codes, and we discuss the weaknesses and problems that may arise when these methods are used. We suggest some approaches, which might be able to overcome these issues by using more knowledge on the problem at hand, and we give some preliminary results with some of these ideas. In the end we summarize and list the best known bounds for these two covering code problem variants.</p>	
<b>Keywords:</b>	asymmetric, covering code, exhaustive search, integer programming, isomorphism, tabu search, unidirectional

## Acknowledgements

The research presented in this thesis has been conducted in the Communications Laboratory in the Department of Electrical and Communications Engineering at the Helsinki University of Technology.

I would like to thank Professor Patric Östergård for the numerous advises during the past two years, and for being the instructor for my Master's thesis. I am grateful for Professor Markku Syrjänen for supervising this work. Finally I deliver my thanks to my workmates for their helpful suggestions and comments (and just generally putting up with me) during the – sometimes hectic – writing process.

Espoo, 11<sup>TH</sup> April 2005

Esa Seuranen

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Covering Code Problem</b>	<b>3</b>
2.1	Covering Codes . . . . .	3
2.2	Asymmetric and Unidirectional Covering Codes . . . . .	5
2.2.1	Asymmetric Covering Codes . . . . .	6
2.2.2	Unidirectional Covering Codes . . . . .	8
2.2.3	Constant Weight Codes . . . . .	9
<b>3</b>	<b>Integer Programming</b>	<b>11</b>
3.1	General Information . . . . .	11
3.2	Integer Programming Problems . . . . .	12
<b>4</b>	<b>Exhaustive Search</b>	<b>17</b>
4.1	Graph Isomorphism . . . . .	17
4.2	Exhaustive Search Implementation . . . . .	18
4.3	Comments on Exhaustive Search . . . . .	23
<b>5</b>	<b>Tabu Search</b>	<b>25</b>
5.1	Local Search . . . . .	25
5.2	Tabu Search . . . . .	27
5.2.1	Fitness Function . . . . .	29
5.2.2	Comments on Fitness Function . . . . .	29
5.2.3	Neighborhood Function . . . . .	30
5.2.4	Comments on Neighborhood Function . . . . .	31
5.2.5	Tabu List . . . . .	31
5.2.6	Ending Condition . . . . .	32
5.2.7	Initial Guess . . . . .	32
5.3	Comments on Implementation . . . . .	33
<b>6</b>	<b>Empirical Tests</b>	<b>34</b>
6.1	Tuning Tabu Search . . . . .	34
6.1.1	Conclusions on Tabu Search . . . . .	37
6.2	Exhaustive Search . . . . .	39

---

6.2.1	Conclusions on Exhaustive Search . . . . .	41
6.3	Comments on the Results . . . . .	41
<b>7</b>	<b>Summary</b>	<b>43</b>

# List of Figures

2.1	A football pool example . . . . .	4
2.2	A lossy compression example . . . . .	5
2.3	A graphical presentation of $Z_2^4$ . . . . .	6
2.4	Differences between $R$ -, $R^\pm$ - and $R^+$ -coverages . . . . .	6
4.1	Samples of graph isomorphism . . . . .	18
4.2	A code isomorphism example . . . . .	19
4.3	An exhaustive search example . . . . .	22
5.1	A tabu search example . . . . .	28
6.1	Tuning tabu search: $(8, K)^+1$ . . . . .	35
6.2	Tuning tabu search: $(9, K)^+2$ . . . . .	36
6.3	Tuning tabu search: $(10, K)^+1$ . . . . .	37
6.4	Tuning tabu search: $(10, K)1$ . . . . .	38
6.5	Tuning tabu search: $(11, K)^{\pm 2}$ . . . . .	39
6.6	The impact of $L$ and $T$ on tabu search . . . . .	40
6.7	All non-isomorphic $(4, 6)^+1$ codes . . . . .	42

# List of Tables

5.1	The impact of initial guess . . . . .	33
6.1	The total number of non-isomorphic $(n, D(n, R))^+R$ codes . . . . .	41
7.1	Lower bounds on $\lceil \frac{1}{n}\phi(n, R) \rceil$ . . . . .	44
7.2	The best known bounds on $D(n, R)$ . . . . .	44
7.3	The best known bounds on $E(n, R)$ . . . . .	45

# List of Algorithms

1	Exhaustive search for solving $v(n, w, k, w')$ . . . . .	20
2	Exhaustive search for non-existence proofs . . . . .	21
3	A generic genetic algorithm . . . . .	26
4	A generic simulated annealing algorithm . . . . .	27
5	A generic tabu search algorithm . . . . .	28

# List of IP Problems

1	$\text{IP}_{\text{exact}}(n, R)$	12
2	$\text{IP}_{\text{sphere}}(n, R)$	13
3	$\text{IP}_{\text{adv}}(n, R)$	14
4	$\text{IP}_{\text{sphere}:\phi}(n, R)^+$	16
5	$\text{IP}_{\text{adv}:\phi}(n, R)^+$	16

# Chapter 1

## Introduction

Combinatorial optimization is a wide area, which has many practical applications. There are many common approaches, such as integer programming and local search methods, which can be used to solve the combinatorial optimization problem at hand – either to a global optimum (which may be time consuming and very difficult) or to a reasonably good solution (which can be attained relatively quickly).

In this thesis we discuss two variants, asymmetric and unidirectional, of the combinatorial optimization problem known as the *covering code problem*. The basic idea with the covering code problem is that we have a space we want to cover, and we can cover it by picking locations for codewords. A codeword covers a certain area around it, i.e. everything which is within the *covering radius* from the codeword. The goal is to minimize the number of codewords (i.e. the *cardinality* of a *covering code*) we need to cover the whole space. For example, consider cellular networks: what the network company wants is that phones can be used everywhere inside the specified area (the space to be covered). For a phone to be usable, it must be close enough (within the covering radius) to a base station (a codeword). Since base stations are not cheap, the network company wants a plan (a covering code) to allocate as few base stations as possible to the specified area, so that phones can still be used everywhere inside the area.

We apply the conventional techniques – that is, integer programming, a local search method (tabu search) and exhaustive search – to solve, or at least to bound, the optimal solutions for asymmetric and unidirectional binary covering codes, and we discuss the weaknesses and problems that may arise when these methods are used. We suggest some approaches, which might be able to overcome these issues by using more knowledge on the problem at hand, and we give some preliminary results with some of these ideas.

This thesis has the following structure. General information regarding covering codes, notations and the definitions for asymmetric and unidirectional covering codes are given in Chapter 2. The possibilities of using integer programming with the problem are discussed in Chapter 3. In Chapter 4 we describe an exhaustive search algorithm, which is used to construct codes with certain properties (or to

---

show that no such codes exist). The local search technique, tabu search, and our implementation of it are discussed in Chapter 5. The empirical tests and their results with tabu search and exhaustive search are presented in Chapter 6, and finally in Chapter 7 we give the concrete results – that is, the best known lower and upper bounds on asymmetric and unidirectional binary covering codes – as well as our conclusions attained during the making of this thesis.

# Chapter 2

## Covering Code Problem

In this chapter we describe covering codes and introduce the main notations used with them. We describe the concepts of asymmetric and unidirectional covering codes and review the published results on them.

### 2.1 Covering Codes

We begin by a few definitions. If a vector is in a code  $C$ , we call it a *codeword*. A word *word* is used of those vectors, which may or may not be in a code. The *weight*,  $w$ , of a word is the number of 1s in the word. A code  $C \subseteq S$  is an  $R$ -covering code on a vector space  $S$ , if for any word  $v \in S$  there is a codeword  $c \in C$  that *covers*  $v$  – that is, the distance from  $c$  to  $v$  is at most  $R$ . The common definition for the *distance* is the Hamming distance  $d_H(c, v)$ , which is the number of coordinates where the vectors  $c$  and  $v$  differ. The objective in a covering code problem is to minimize the cardinality of the covering code.

The vector spaces that have been researched quite well are  $Z_k^n$ , for  $k = 2, 3, 4$  (binary, ternary and quaternary). In addition also *mixed covering codes*, where space  $S$  is spanned by vectors of type  $Z_2^n Z_3^m$ , have been studied to some extent. The study of covering codes have been mainly motivated by *football pools* and *data compression*. For further information about covering codes the reader is referred to [6]. A binary covering code of length  $n$ , cardinality  $K$  and covering radius  $R$  is denoted with  $(n, K)R$ . Minimal cardinalities of  $(n, K)R$  codes are denoted with  $K(n, R)$  (the notation  $K(b, t, R)$  is used for mixed alphabets, where  $b$  is the number of binary and  $t$  ternary coordinates).

**Example** In football pools a player tries to guess the outcomes of football matches (1 = home team wins, X = draw, 2 = guest team wins). In Figure 2.1 we have a player, who is “sure” about the outcomes of eleven matches, but he/she is uncertain about the two remaining matches. So the question is, how should our player purchase game coupons to ensure the result he/she desires – assuming the player really is correct in those eleven matches? To ensure thirteen correct outcomes he/she will

have to buy all coupons ( $3^2 = 9$ ), but if he/she would be satisfied with at least twelve correct outcomes, the less coupons would be necessary. A 1-covering code on  $Z_3^2$  will give him/her the answer about which coupons should be purchased (three coupons are needed, as the reader can verify from the graph).

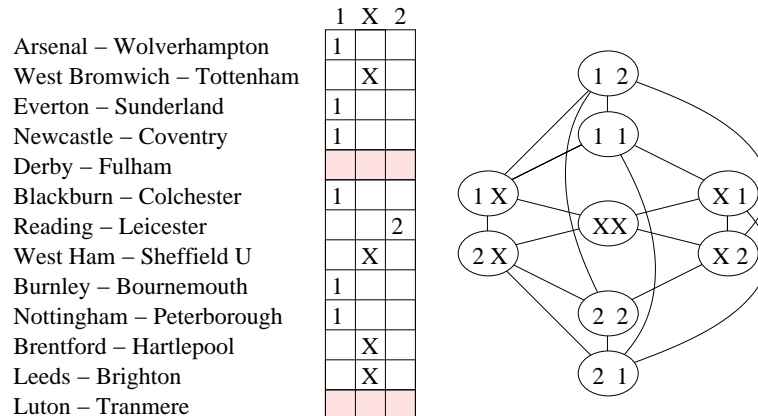


Figure 2.1: A football pool example

**Example** One idea to implement a lossy compression (for binary data) is to use a covering code on  $Z_2^n$  – an original binary vector in  $Z_2^n$  is encoded with a codeword, that covers it. Lets say we would like to compress (lossly) data, which is represented by binary vectors of length seven. Then we could use a minimal 1-covering code on  $Z_2^7$  (i.e.  $(7, 4)$  code) to do the lossy compression – see Figure 2.2, in which codewords are presented by their decimal and binary values, and the covered words are drawn with the same color as the codeword, which covers them. A minimal  $(7, 4)$  code has the cardinality of 16, so the codewords can be encoded with four bits. Therefore we could compress seven bits by four bits, but – of course – there is no way to retrieve the lost information in those three bits.

Error-correcting codes are closely related to covering codes. With covering codes one tries to construct a minimal code covering the whole space, whereas with error-correcting codes one tries to construct a maximal code so that the coverages of individual codewords do not overlap. With error-correcting (and error-detecting) codes the interesting parameter is the minimum distance  $d$ , which is the shortest Hamming distance between any two codewords. The value of  $d$  is interesting, because the code can correct up to  $\lfloor \frac{d-1}{2} \rfloor$  errors (and detect up to  $\lceil \frac{d-1}{2} \rceil$  errors). For odd  $d$  an error-correcting code problem can be considered to be dual to the covering code problem (with  $\frac{d-1}{2} = R$ ). If the solutions of these two problems coincide, then the solution is a *perfect code* (that is, there is a covering code which covers every word in space exactly once). See [13] and its references for more information on error-correcting, error-detecting and perfect codes.

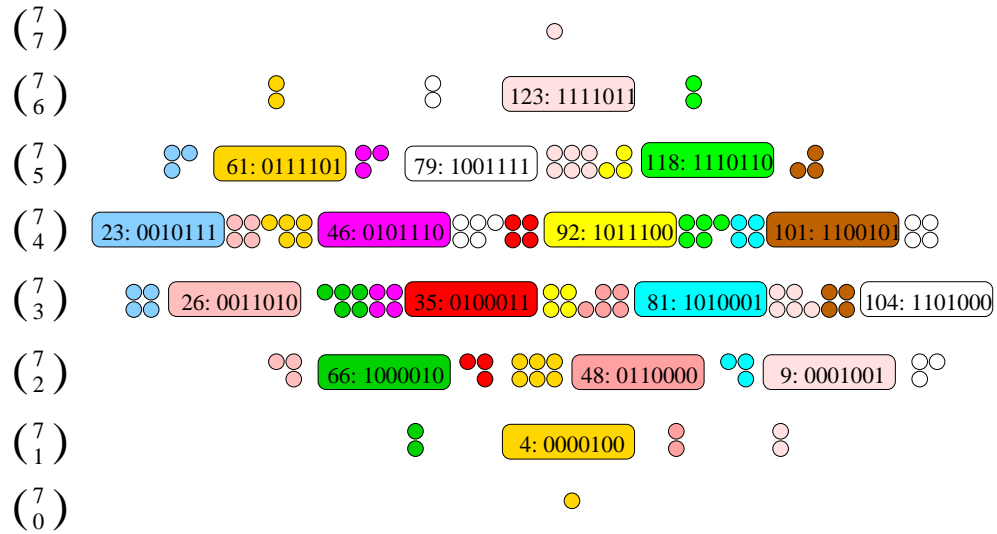


Figure 2.2: A lossy compression example

Asymmetric and unidirectional error-correcting codes have been studied quite extensively, since they are very useful in telecommunication. Asymmetric and unidirectional covering codes have received almost no attention at all, due to the fact that the practical applications for them are scarce or non-existent.

## 2.2 Asymmetric and Unidirectional Covering Codes

We say that a codeword  $c$   $R^+$ -covers all the words that can be obtained from  $c$  by replacing at most  $R$  1s with 0s – and similarly, a codeword  $c$   $R^-$ -covers all the words that can be obtained from  $c$  by replacing at most  $R$  0s with 1s. A code  $C$  is an asymmetric  $R$ -covering code if for every word  $v \in Z_2^n$  there is  $c \in C$ , which  $R^+$ -covers  $v$ .

In a similar way a code  $C$  is a unidirectional  $R$ -covering code if for every word  $v \in Z_2^n$  there is  $c \in C$ , which either  $R^+$  or  $R^-$ -covers (i.e.  $R^\pm$ -covers)  $v$ . We use the notations  $(n, K)^+R$  and  $(n, K)^\pm R$ , respectively, for asymmetric and unidirectional covering codes on  $Z_2^n$  with cardinality  $K$ . The notations  $D(n, R)$  and  $E(n, R)$  are used for the minimal cardinalities of such codes.

To give a clear basis and motivation for some of the results presented shortly we demonstrate a graphical way to approach the problem.

**Example** A graphical representation of  $Z_2^4$  is shown in Figure 2.3. Both the decimal and binary representation of the words are included. The words with the same weight  $w$  are grouped on the same level. In the left there is a binomial showing the total number of words with the corresponding weight. There is an edge between words, if the corresponding words differ exactly at one coordinate.

In addition, the differences between  $R^-$ ,  $R^\pm$ - and  $R^+$ -coverages for  $R = 2$  are demonstrated in Figure 2.4. The codeword (node 13) is indicated with darker color and the words covered by it are indicated with lighter colors and arrows.

Apparently the asymmetric covering codes were first studied in [7]. Their work was motivated by layout of data compression in VLSI designs [8]. In addition asymmetric covering codes have been studied in [3, 17], as well as in [9] (although no paper was published).

The unidirectional covering codes have only been studied in [18]. The practical applications for them are yet to arise.

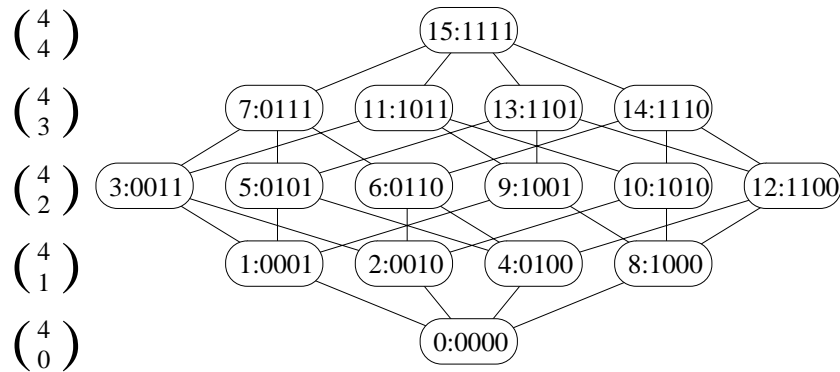


Figure 2.3: A graphical presentation of  $Z_2^4$

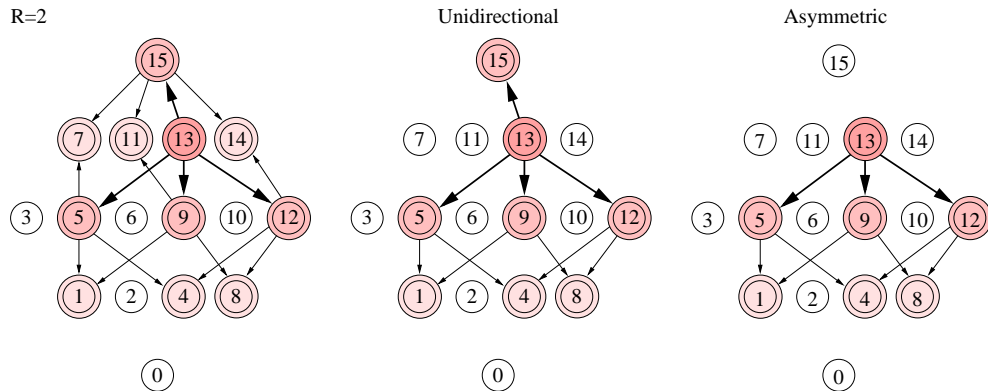


Figure 2.4: Differences between  $R^-$ ,  $R^\pm$ - and  $R^+$ -coverages

### 2.2.1 Asymmetric Covering Codes

We will now review the results presented in [3, 7, 17] for those parts, which still contribute to (note that quite a few results were omitted from this thesis, and

reader should refer to the papers for some more theoretical results) the best known lower and upper bounds on  $D(n, R)$ . Theorems and Corollaries from 2.2.1 to 2.2.6 are from [7] and Theorem 2.2.7 from [3].

**Theorem 2.2.1** *Let  $C$  be a  $(n, K_1)^+R$  code and  $1 \leq i \leq n$  be an arbitrary coordinate. If  $(c_1, c_2, \dots, c_i, \dots, c_n) \in C$  and  $c_i = 1$ , then  $(c_1, c_2, \dots, c_{i-1}, c_{i+1}, \dots, c_n) \in \hat{C}$ . The shortened code  $\hat{C}$  is a  $(n-1, K_2)^+R$  code.*

**Proof** Let  $C_i$  be the set of codewords  $c \in C$ , which have 1 at the coordinate  $i$ . Since  $C$  is a  $(n, K)^+R$  code, all the vectors  $(v_1, v_2, \dots, v_{i-1}, 1, v_{i+1}, \dots, v_n) \in Z_2$  are  $R^+$ -covered by  $C_i$ .  $\square$

**Corollary 2.2.2**  $D(n-1, R) \leq D(n, R)$ .

Let  $\phi(n, R)$  be the maximal number of 0s in a minimal  $(n, K)^+R$  code.

**Theorem 2.2.3**  $D(n-1, R) \leq D(n, R) - \phi(n, R)/n$ .

**Proof** Let  $C$  be a minimal  $(n, K_1)^+R$  code attaining  $D(n, R)$ , which has the maximal number,  $\phi(n, R)$ , of 0s. Let  $i$  be the coordinate, in which the most of the codewords in  $C$  have 0. Then shortening  $C$  at the coordinate  $i$  produces a  $(n-1, K_2)^+R$  code, which has a cardinality  $K_2$  at most  $D(n, R) - \lceil \frac{\phi(n, R)}{n} \rceil$ .  $\square$

The direct sum construction,  $\oplus$ , is defined as follows:  $(c_1, c_2) \in C_1 \oplus C_2$ , where  $C_1$  and  $C_2$  are codes, and  $c_1 \in C_1$ ,  $c_2 \in C_2$ .

**Theorem 2.2.4** *Let  $C_1$  and  $C_2$  be  $(n, K_1)^+R_1$  and  $(m, K_2)^+R_2$  codes, then  $C_1 \oplus C_2$  is a  $(n+m, K_1K_2)^+R_3$  code, where  $R_3 = R_1 + R_2$ .*

**Proof** Let  $v_1 \in Z_2^n$  and  $v_2 \in Z_2^m$ . Since  $C_1$  and  $C_2$  are asymmetric covering codes, there are  $c_1 \in C_1$  and  $c_2 \in C_2$  which  $R_1^+$ - and  $R_2^+$ -cover  $v_1$  and  $v_2$ . According the definition of direct sum  $(c_1, c_2) \in C_1 \oplus C_2$ , and therefore for every  $v = (v_1, v_2) \in Z_2^{n+m}$  there is a  $c = (c_1, c_2) \in C_1 \oplus C_2$  at least  $(R_1 + R_2)^+$ -covering  $v$ .  $\square$

**Corollary 2.2.5**  $D(n, R) \leq D(n_1, R_1) \cdot D(n - n_1, R - R_1)$ .

**Theorem 2.2.6**  $D(n, n - \bar{R}) \geq \bar{R} + 1$  for  $n \geq 1$  and  $\bar{R} \geq 0$ , with equality when  $n \geq n_{\bar{R}} := \frac{\bar{R}(\bar{R}+1)}{2}$ . Further more,  $n_{\bar{R}}$  is the least integer  $n$  for which equality holds.

**Proof** We omit the proof. The reader is referred to [7, Theorem 13].

We do not use the following result, but we find it well worth mentioning, as it might be used for proposing a structure for covering codes and ease – for instance – constructing better covering codes by a local search method.

**Theorem 2.2.7**

$$D(n, 1) \leq \left| \bigcup_{i=0}^{\lfloor n/2 \rfloor} C(n+1, n+1-2i, n-2i) \right|,$$

where  $C(v, k, t)$  is a covering design, where any collection of special  $k$ -subsets  $S$  of a  $v$ -set such that any  $t$ -subset  $T$  is contained in at least one  $S$ .

**Proof** The reader is referred to [3, Theorem 3] for the proof.

**2.2.2 Unidirectional Covering Codes**

We will now review the results presented in [18]. Some of the results are presented in Section 2.2.3, since they can also be used with asymmetric covering code problem.

Unidirectional covering codes are a bit more difficult to be constructed, that is, the construction methods working for covering codes and asymmetric covering codes either does not seem to work or they can be applied only limitedly. For instance, the direct sum construction works for unidirectional covering codes, as long as other code is full (i.e.  $Z_2^n$ ).

**Theorem 2.2.8** *If  $C \subseteq Z_2^n$  has unidirectional covering radius  $R$ , then  $C \oplus Z_2^m$  is a  $(n+m, 2^m|C|)^{\pm R}$  code.*

**Proof** Let  $C$  be a  $(n, K)^{\pm R}$  code. For any  $v = (v_n, v_m) \in Z_2^{n+m}$  (where  $v_n \in Z_2^n$ ,  $v_m \in Z_2^m$ ) there is a  $c = (c_n, c_m) \in C \oplus Z_2^m$  (where  $c_n \in C$ ,  $c_m \in Z_2^m$ )  $R^{\pm}$ -covering  $v$ . Since  $v_n$  is  $R^{\pm}$ -covered by  $c$  and  $v_m$  0-covered by  $c_m$ .  $\square$

**Corollary 2.2.9**  $E(n+1, R) \leq 2E(n, R)$ .

Some trivial results:

**Theorem 2.2.10** 1. The code  $C = \{00 \dots 0\}$  attains  $E(n, R) = 1$ ,  $n \leq R$ .

2. The code  $C = \{00 \dots 0, 11 \dots 1\}$  attains  $E(n, R) = 2$ ,  $\lfloor \frac{n}{2} \rfloor \leq R < n$ .

3.  $E(n, 1) = K(n, 1)$ , where  $K(n, R)$  is the minimal cardinality of a  $R$ -covering code on  $Z_2^n$ .

One more construction related theorem was introduced – which is exactly the same as with “normal” covering codes – but we need an another definition before proceeding.

A code is 2-surjective, if for any two coordinates the all four  $\{00, 01, 10, 11\}$  combinations can be found in the codewords in those two coordinates.

**Theorem 2.2.11** *If there exists a code attaining  $E(n+2, R+1)$  that is not 2-surjective, then  $E(n, R) \leq E(n+2, R+1)$ .*

**Proof** Let  $C$  be a code attaining  $E(n + 2, R + 1)$  and let it be not 2-surjective in the last two coordinates (w.l.o.g). Let  $b \in \mathbb{Z}_2^2$  be the combination not appearing in those two last coordinates. Truncating (removing) last two coordinates of  $C$  gives code  $C'$ . The unidirectional covering radius of  $C'$  must be at most  $R$  (since  $C$  has unidirectional covering radius  $R + 1$  and  $b$  does not occur in the two last coordinates of any  $c \in C$ ).  $\square$

**Theorem 2.2.12**  $E(2R + 3, R) = 8$  for  $R > 1$ .

**Proof** Solving  $\text{IP}_{\text{exact}}(7, 2)^\pm$  – see Section 3 and IP Problem 1 – shows that  $E(7, 2) = 8$ . Results in [12, Table 1] show that there are no 2-surjective  $(2R + 3, 7)R$  codes for  $R > 1$ , and therefore no 2-surjective  $(2R + 3, 7)^\pm R$  codes exist for  $R > 1$  either. If there would be a  $(2R + 3, 7)^\pm R$  code  $C$ , where  $R > 1$  and  $K = 7$ , then Theorem 2.2.11 would give  $E(2(R - 1) + 3, R - 1) \leq 7$ . Applying this recursively one would arrive to  $E(7, 2) \leq 7$ , which is a contradiction.

### 2.2.3 Constant Weight Codes

Let  $v(n, w, k, w')$  be the maximum number of words with weight  $w'$  and length  $n$ , that can be covered with  $k$  codewords of weight  $w$  of length  $n$ . Later on in Sections 3 and 4 we use this function to construct integer programming problems and improve the pruning in exhaustive search. These results were presented in [18].

**Theorem 2.2.13**  $v(n, w, k, w')$  is the same for both asymmetric and unidirectional covering codes (though naturally  $v(n, w, k, w + i) = 0$  in asymmetric context for  $1 \leq i \leq n - w$ ).

**Proof** Follows from the definition of  $v(n, w, k, w')$  as well as the definitions of asymmetric and unidirectional coverage.  $\square$

The values of  $v(n, w, k, w')$  and constant weight codes (or *packing designs*) are closely related – especially the value of  $\mathcal{A}(n, d, w)$ , which is the maximum cardinality of a constant weight code consisting of codewords with weight  $w$ , length  $n$  and minimum distance  $d$  (that is, any two codewords in the code differ at least in  $d$  coordinates). For extensive results on this function the reader is refer to [2, 5, 13].

**Theorem 2.2.14**  $v(n, k, w, w') = k \cdot v(n, 1, w, w')$  for  $k \leq \mathcal{A}(n, 2(|w - w'| + 1), w)$

**Proof** Follows from the definition of  $\mathcal{A}(n, d, w)$ .  $\square$

**Theorem 2.2.15** In unidirectional context  $v(n, k, w, w') = v(n, k, n - w, n - w')$ .

**Proof** By symmetry.  $\square$

**Theorem 2.2.16**  $v(n, k + 1, w, w') \leq \frac{k+1}{k} \cdot v(n, k, w, w')$ .

**Proof** Assume  $v(n, k + 1, w, w') > \frac{k+1}{k} \cdot v(n, k, w, w')$ . Let  $C$  be the code attaining  $v(n, k + 1, w, w')$ . There must be a codeword  $c \in C$ , that solely covers at most  $\frac{1}{k+1}v(n, k + 1, w, w')$  words. Then the code  $C \setminus \{c\}$  would cover

$$v(n, k + 1, w, w') - \frac{1}{k + 1}v(n, k + 1, w, w') > \left(\frac{k + 1}{k} - \frac{k + 1}{k} \frac{1}{k + 1}\right)v(n, k, w, w')$$

words and therefore attain a higher value for  $v(n, k, w, w')$ , which is a contradiction.  $\square$

**Theorem 2.2.17** *Let  $k = \mathcal{A}(n, 2(|w - w'| + 1), w) + i$  with  $i \geq 1$ . Then  $v(n, k, w, w') \leq k \cdot v(n, 1, w, w') - i$ .*

**Proof** For  $i = 1$  the theorem follows from the definition of  $\mathcal{A}(n, d, w)$ . For  $i > 1$  follows from Theorem 2.2.16.  $\square$

# Chapter 3

## Integer Programming

Integer programming is a traditional and mathematical way to approach the combinatorial optimization problem at hand. We are not interested how to actually solve integer programming problems, but how to formulate useful integer programming problems – problems that are solvable in reasonable time and whose solutions would still provide some useful information for improving the lower and upper bounds of  $D(n, R)$  and  $E(n, R)$ . There are many general integer (and linear) programming problem solvers around, from which we used GLPK [10].

### 3.1 General Information

Integer programming (IP) problems, like linear programming (LP) problems, consist of an objective function,  $g_z()$ , which is either minimized or maximized, and a set of constraints,  $g_i()$ , that must be satisfied. The distinctive point of an IP problem is that all the variables  $x_i$  must be integers – if only some variables are required to be integers, then the problem is called mixed integer programming (MIP) problem:

$$\begin{aligned} \text{Min (or Max)} Z &= g_z(x_1, x_2, \dots, x_n) \\ \text{Subject To:} \\ g_i(x_1, x_2, \dots, x_n) &\begin{cases} \leq \\ \equiv \\ \geq \end{cases} b_i, & i \in M \equiv \{1, 2, \dots, m\} \\ x_j &\geq 0, \\ x_j &\text{ is an integer,} & j \in N \equiv \{1, 2, \dots, n\} \end{aligned}$$

Many of the combinatorial optimization problems can be formalized as IP problems and solved with any general IP problem solver. Integer programming is well known and a hard-studied-part of mathematics – partly because the practical applications for efficient IP problem solver are numerous and partly because efficient algorithms for solving LP problems are known (like simplex method). The disappointing thing is, that solving IP problems is hard – much harder than solving LP problems. This seems a bit controversial, since the search space for LP problems

seems to be larger, but the integer constraint breaks down the properties, which enable efficient solving of LP problems. Usually an exhaustive search must be performed to get an optimal solution to an IP problem. For more information on integer programming the reader is referred to [23].

## 3.2 Integer Programming Problems

Integer programming has been used for asymmetric covering codes in [3, 7] and for unidirectional covering codes in [18]. We use superscripts  $+$  and  $\pm$  to distinguish between asymmetric and unidirectional formulations of the integer programming problems.

The covering code problem can be formulated as integer programming problem directly and then solved, but this approach is feasible only for small problems (that is, for  $n \leq 7$  or 8). We refer this direct formulation as  $\text{IP}_{\text{exact}}(n, R)$  – see IP Problem 1, where  $b_i = 1$  only if  $i$  (decimal form) is a codeword and  $S(i)$  is the set of words that cover the word  $i$ .  $\text{IP}_{\text{exact}}(n, R)$  can be relaxed and solved as linear programming problem to get a lower bound on  $D(n, R)$  or  $E(n, R)$ , but the bound obtained by this way is not very good.

---

**IP Problem 1**  $\text{IP}_{\text{exact}}(n, R)$

---

$$\begin{aligned} \text{Min } Z &= \sum_{i=0}^{2^n-1} b_i \\ \text{Subject To:} & \\ 1 &\leq \sum_{j \in S(i)} b_j, \text{ where } 0 \leq i < 2^n \\ b_i &\in \{0, 1\}, \text{ where } 0 \leq i < 2^n, \end{aligned}$$


---

There are more indirect ways to use integer programming to get some useful information of the problem. One such way, used in [7], is to group together the codewords with the same weight (instead of thinking individual codewords) and then consider, how many other words of certain weight can  $k$  codewords with weight  $w$  at most cover: one  $w$ -weighted codeword  $1^+$ -covers  $\binom{w}{0}$  words of weight  $w$  and  $\binom{w}{1}$  words of weight  $w-1$ . If there are  $k$  codewords with weight  $w$ , they cover at most  $k$  times as many words as single codeword – this approach results in  $\text{IP}_{\text{sphere}}(n, R)$ , see IP Problem 2, where  $s_w$  is the number of  $w$ -weighted codewords.

$\text{IP}_{\text{sphere}}(n, R)$  is very compact formulation and very fast to solve (only  $n+1$  variables and  $n+1$  constraints). The lower bound achieved by solving  $\text{IP}_{\text{sphere}}$

---

**IP Problem 2**  $\text{IP}_{\text{sphere}}(n, R)$ 


---

$$\text{Min } Z = \sum_{w=0}^n s_w$$

Subject To:

$$\binom{n}{w} \leq s_w + \sum_{\substack{r=1 \\ 0 \leq w+r \leq n}}^R s_{w+r} \binom{w+r}{r} + \overbrace{\sum_{\substack{r=1 \\ 0 \leq n-w+r \leq n}}^R s_{w-r} \binom{n-w+r}{r}}^{\text{only in unidirectional case}},$$

where  $0 \leq w \leq n$

$$s_w \in \left\{0, 1, 2, \dots, \binom{n}{w}\right\}, \text{ where } 0 \leq w \leq n,$$


---

seems to be better than the one attained by solving LP version of  $\text{IP}_{\text{exact}}$ , but the lower bound achieved this way is not very tight.

However, the formulation ignores the fact, that at some point the coverages of codewords must overlap. By taking this into account improvements can be introduced. One approach, taken in [18], is to study constant weight codes and how much their coverages overlap, and then include the information about the overlaps into the integer programming problem. This approach needs few auxiliary functions,  $\text{last}(w)$  and  $v(n, w, k, w')$ , and it leads in  $\text{IP}_{\text{adv}}(n, R)$  – see IP Problem 3, where  $s_{w,k} = 1$  exactly when there are  $k$  codewords of weight  $w$ ;  $\text{last}(w)$  is the maximal number of codewords with weight  $w$ ; and  $v(n, w, k, w')$  is the maximal number of  $w'$ -weighted words that can be covered with  $k$  codewords of weight  $w$ .

$\text{IP}_{\text{adv}}(n, R)$  produces slightly better lower bounds than  $\text{IP}_{\text{sphere}}(n, R)$  while still being solvable also for larger problem instances. Solving the values (or bounds) of  $v(n, w, k, w')$  is discussed in Section 4.2 (whereas some theoretical results were discussed already in Section 2.2.3).

An upper bound for  $\text{last}(w)$  for specific problem and for the assumed minimal cardinality of a covering code can be attained by modifying  $\text{IP}_{\text{sphere}}$  a little: adding a constraint for specifying the cardinality of the code

$$M = \sum_{w=0}^n s_w,$$

where  $M$  is the lower bound for the problem; and changing the objective function to

$$\text{Max } Z = s_{w'},$$

---

**IP Problem 3**  $\text{IP}_{\text{adv}}(n, R)$ 


---

$$\text{Min } Z = \sum_{w=0}^n \sum_{k=1}^{\text{last}(w)} k \cdot s_{w,k}$$

Subject To:

$$\begin{aligned} \binom{n}{w} &\leq \sum_{k=1}^{\text{last}(w)} k \cdot s_{w,k} \\ &+ \underbrace{\sum_{r=1}^R \sum_{k=1}^{\text{last}(w+r)} v(n, w+r, k, w) \cdot s_{(w+r),k}}_{\text{only in unidirectional case}} \\ &+ \sum_{r=1}^R \sum_{k=1}^{\text{last}(w-r)} v(n, w-r, k, w) \cdot s_{(w-r),k}, \\ &\text{where } 0 \leq w \leq n \end{aligned}$$

$$\sum_{k=1}^{\text{last}(w)} s_{w,k} \leq 1, \text{ where } 0 \leq w \leq n$$

$$s_{w,k} \in \{0, 1\}, \text{ where } 0 \leq w \leq n \text{ and } 1 \leq k \leq \text{last}(w),$$


---

where  $0 \leq w' \leq n$ . This modified (denoted with an additional subscript  $\text{last}$ ) formulation is then solved for each  $0 \leq w' \leq n$  to get upper bounds on  $\text{last}(w)$  for all  $0 \leq w \leq n$ . Note that if the lower bound for the current problem is improved, then the values of  $\text{last}(w)$  must be solved again (since the assumed minimal cardinality  $M$  has changed).

Similar modifications can be done to  $\text{IP}_{\text{adv}}$ , i.e. change the objective function to

$$\text{Max}Z = \sum_{k=1}^{\text{last}(w')} k \cdot s_{w',k}$$

and add the constraint

$$M = \sum_{w=0}^n \sum_{k=1}^{\text{last}(w)} k \cdot s_{w,k},$$

to get a bit tighter upper bounds for  $\text{last}(w)$ .

In Section 2.2.1 we introduced  $\phi(n, R)$  as the maximum total number of 0s in a minimal  $(n, K)^+R$  code, but nothing was mentioned about how to solve it. IP Problem 4 (in which  $s_w$  is the number of  $w$ -weighted codewords) was defined in [7] to solve an lower bound on  $\phi(n, R)$ , but we can do a bit better by using the idea in  $\text{IP}_{\text{adv}}$  and adding a constraint to fix the cardinality of the code to the current lower bound on  $D(n, R)$ . These improvements lead to  $\text{IP}_{\text{adv}:\phi(n, R)^+}$  – see IP Problem 5, where  $s_{w,k} = 1$  exactly when there are  $k$  codewords of weight  $w$ ;  $\text{last}(w)$  is the maximal number of codewords with weight  $w$ ;  $v(n, w, k, w')$  is the maximal number of  $w'$ -weighted words that can be covered with  $k$  codewords of weight  $w$ ; and  $K$  is the cardinality of a minimal  $^+(n, R)$  code (i.e. the lower bound on  $D(n, R)$ ).

$\text{IP}_{\text{adv}}$  can be used with Theorem 2.2.3 to improve several lower bounds on  $D(n, R)$ . The differences are demonstrated later in Section 7 in Table 7.1.

**Theorem 3.2.1**  $\text{IP}_{\text{sphere}:\phi(n, R)^+}$  and  $\text{IP}_{\text{adv}:\phi(n, R)^+}$  give a lower bound on  $\phi(n, R)$ .

**Proof** Since the total number of 0s is minimized in both formulations, the solutions to these formulations are less than or equal to  $\phi(n, R)$ .  $\square$

---

**IP Problem 4**  $\text{IP}_{\text{sphere}}^{\phi}(n, R)^+$ 


---

$$\text{Min } Z = \sum_{w=0}^n s_w(n-w)$$

Subject To:

$$\binom{n}{w} \leq s_w + \sum_{\substack{r=1 \\ 0 \leq w+r \leq n}}^R s_{w+r} \binom{w+r}{r}, \text{ where } 0 \leq w \leq n$$

$$s_w \in \{0, 1, 2, \dots, \binom{n}{w}\}, \text{ where } 0 \leq w \leq n,$$


---

---

**IP Problem 5**  $\text{IP}_{\text{adv}}^{\phi}(n, R)^+$ 


---

$$\text{Min } Z = \sum_{w=0}^n \sum_{k=1}^{\text{last}(w)} k \cdot s_{w,k}(n-w)$$

Subject To:

$$\binom{n}{w} \leq \sum_{k=1}^{\text{last}(w)} k \cdot s_{w,k} + \sum_{r=1}^R \sum_{k=1}^{\text{last}(w+r)} v(n, w+r, k, w) \cdot s_{(w+r),k}$$

where  $0 \leq w \leq n$ 

$$K = \sum_{w=0}^n \sum_{k=1}^{\text{last}(w)} k \cdot s_{w,k}$$

$$\sum_{k=1}^{\text{last}(w)} s_{w,k} \leq 1, \text{ where } 0 \leq w \leq n$$

$$s_{w,k} \in \{0, 1\}, \text{ where } 0 \leq w \leq n \text{ and } 1 \leq k \leq \text{last}(w),$$


---

# Chapter 4

## Exhaustive Search

For some difficult problems integer programming resembles an exhaustive search, which would in the worst case end up to enumerate all possible solutions. The integer programming problem solvers are very general, and all available problem specific information cannot be incorporated into integer programming problem, thus the most efficient way to prune hopeless solutions are not used. So one might consider doing a problem specific exhaustive search algorithm, which takes full advantage of the available information and therefore – hopefully – will be able to solve a bit larger problems. A such exhaustive search was implemented in [18] and will be described in detail here. But before going into exhaustive search, we discuss graph (code) isomorphism.

### 4.1 Graph Isomorphism

One rather powerful method for pruning out codes with equal properties is graph isomorphism and it has been used, for instance, in [19]. A graph isomorphism is a bijection between the vertices of two graphs  $G$  and  $H$ :

$$f : V(G) \rightarrow V(H)$$

with the property that any two vertices  $u$  and  $v$  from  $G$  are adjacent if and only if  $f(u)$  and  $f(v)$  are adjacent in  $H$ . If an isomorphism can be constructed between two graphs, then we say those graphs are isomorphic (i.e. if there exists a way to relabel the vertexes of the other graph, so that the two graphs become identical, then the graphs are isomorphic).

**Example** Some samples of graph isomorphisms are shown in Figure 4.2:

a) All these three graphs are the same complete graph  $K_4$  (complete graph has an edge from every vertex to every other vertex), and they are naturally isomorphic. It does not matter how the graphs are drawn.

b) These two graphs look pretty much alike – and indeed they are isomorphic, since there is a way to relabel them to look identical:  $1 \leftrightarrow 4$  and  $2 \leftrightarrow 3$ .

c) These two graphs represent two asymmetric 1-covering codes on  $Z_2^3$  – namely, codes  $\{2, 5, 7\}$  and  $\{3, 4, 7\}$ . Again these graphs are isomorphic (and equivalent in their covering properties): relabeling  $2 \leftrightarrow 4$  and  $3 \leftrightarrow 5$  make graphs identical.

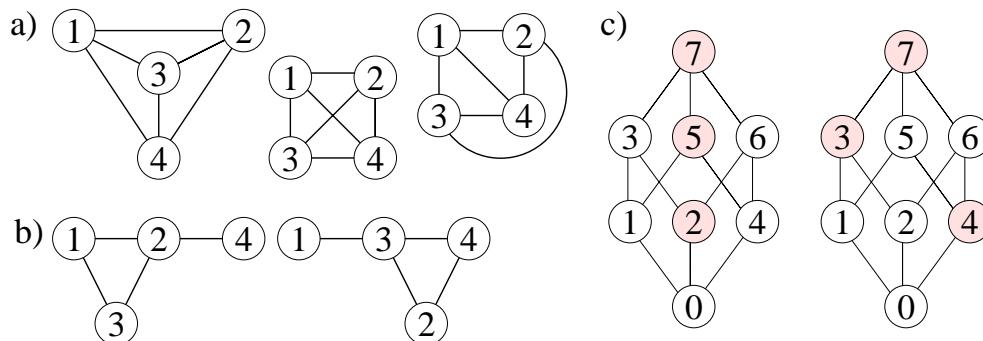


Figure 4.1: Samples of graph isomorphism

The interesting question here is, how to represent the codes as graphs. One could construct graphs from the whole space  $Z_2^n$  with the codewords, but this is rather cumbersome (the graphs and the computational effort for checking isomorphisms become large). A cleverer way is to observe the codewords and their relations to the coordinates, in which they have 1. One (and nearly the standard) way to do this in practice is to construct a bigraph, so that the one side has the coordinates and the other has the codewords. There is an edge between a coordinate and a codeword, if the codeword has 1 in that coordinate.

**Example** In Figure 4.2 we show, that the two asymmetric 1-covering codes on  $Z_2^3$  in previous Figure 4.1 are indeed isomorphic. The coordinate values (1, 2 and 4) as well as the codewords (sum of the coordinate values) are included in the figure for clarity, but they are not necessary for determining the isomorphism. A way to rename the graph vertexes is as follows:

- a) relabeling codewords:  $b \leftrightarrow c$ ,
- b) relabeling coordinates  $2 \leftrightarrow 4$  (and recalculating codewords) and
- c) redrawing the graph gives the wanted outcome  $\rightarrow$  graphs constructed from codes  $\{2, 5, 7\}$  and  $\{3, 4, 7\}$  are isomorphic, and so are the codes.

## 4.2 Exhaustive Search Implementation

Exhaustive search tend to be somewhat looked down by mathematicians and computer scientist alike, for the idea of using brute force to solve the problem instead of developing a mathematically valid and elegant solution to the problem. However, “brute forcing” seems to be the only way for some problems, at least for now

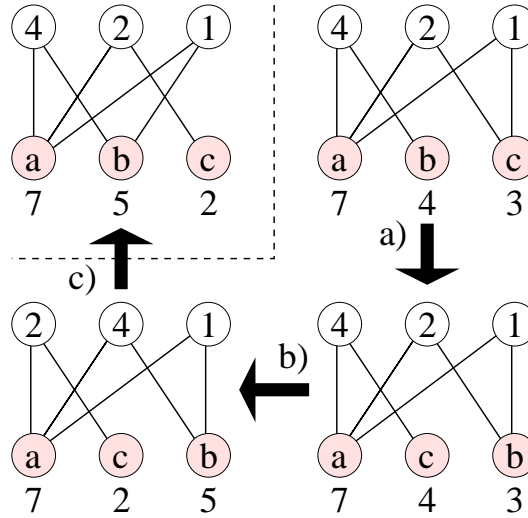


Figure 4.2: A code isomorphism example

– besides, computers become more and more able to do powerful calculations, so why not to use this resource. The philosophy and issues of exhaustive search are discussed in [15].

We use exhaustive search for two different, but related, matters: for solving  $v(n, w, k, w')$ , Algorithm 1, and for proving non-existence of specific code, Algorithm 2. Both of these algorithms are breadth-first searches, which makes them rather simple and fast, although rather memory-consuming (which limits the usage for those problems, in which the pruning works well).

**Example** See Figure 4.3 to follow the exhaustive search process, in which we are solving the lower bound on  $E(7, 2)$ , which can naturally be done also by solving  $\text{IP}_{\text{exact}}(7, 2)^\pm$ .

We begin by solving  $\text{IP}_{\text{sphere}}(7, 2)^\pm$ , which give us a lower bound of  $E(7, 2) > 6$  – next we try to improve it. First we use  $\text{IP}_{\text{sphere:last}}(7, 2)^\pm$  to find out upper bounds on  $\text{last}(w)$  assuming that  $K$  is the current lower bound (i.e.  $K = 7$ ). Next we solve necessary (needed by  $\text{IP}_{\text{adv}}$  and Algorithm 2) values of  $v(7, w, k, w')$  with Algorithm 1 – and if some values could not be solved, then those values are upper bounded with Theorems 2.2.16 and 2.2.17.

Next we solve  $\text{IP}_{\text{adv}}(7, 2)^\pm$  to see, whether the current lower bound could be improved directly (were not so lucky with our example). We continue by (probably) improving the upper bounds on  $\text{last}(w)$  with  $\text{IP}_{\text{adv:last}}(7, 2)^\pm$ . We proceed by generating all possible weight distributions of possible covering codes (with cardinality of the current lower bound) and checking those weight distributions with Algorithm 2, which generates all covering codes of given weight distribution. Since no covering codes are generated, we can conclude that there are none (and therefore  $E(7, 2) > 7$ ).

---

**Algorithm 1** Exhaustive search for solving  $v(n, w, k, w')$  for  $k = 1, 2, \dots, \text{last}(w)$

---

**Require:**  $0 \leq w \leq n$ ,  $0 < n$  and  $0 < \text{last}(w) \leq \binom{n}{w}$

```

1: current  $\leftarrow$  newHashtable()
2: insertInto(current, emptyCode())
3:  $k \leftarrow 0$ 
4: while !endingCriteria() and  $k \leq \text{last}(w)$  do
5:   future  $\leftarrow$  newHashtable()
6:   for all  $C \in \textit{current}$  do
7:     for all  $v \in \text{wordsWithWeight}(w)$  do
8:       skip if alreadyInCode( $C, v$ )
9:        $C' \leftarrow \text{canonize}(C + v)$ 
10:      insertIntoIfNotAlreadyThere(future,  $C'$ )
11:    end for
12:  end for
13:  printMaxCoverages(future)
14:  swap(future, current)
15:   $k \leftarrow k + 1$ 
16: end while

```

**Ensure:** Maximal coverages have been printed

---

This process can be used iteratively to improve the lower bound even further till either the upper bound is met (i.e. covering codes are generated) or (very likely) the exhaustive search becomes too exhaustive for the algorithm to handle.

Few notes about the algorithms:

**Hashtable():** An implementation of an open ended hash table. Usually a good size for a hash table is a prime number not too close to the powers of 2 – we used 9973 as *hashSize*. For the hashing function we used canonized graph ( $G$ ) and the following function (vertexes are 0, 1, 2, ...):

```

hash  $\leftarrow 0$ 
shift  $\leftarrow 0$ 
for all  $vertex \in G$  do
  for all  $e \in \text{endVerticesOfEdges}(vertex)$  do
     $hash \leftarrow hash + (e \ll shift)$ 
     $shift \leftarrow (shift + 1) \bmod 8$ 
  end for
end for
return  $hash \bmod hashSize$ 

```

Reason for using hash table is that hash table is a data structure with (on average) a constant time needed for retrieval, insertion and deletion operations – that is, of course, only if the hash table is large enough and the hashing functions distributes the items evenly all over the hash table.

---

**Algorithm 2** Exhaustive search for non-existence proofs

---

**Require:**  $\sum_{i=0}^n weights[i] = K$ ;  $0 \leq weights[i] \leq \binom{n}{i}$  for  $0 \leq i \leq n$ ;  $0 < n$ ;

```

1: current  $\leftarrow$  newHashtable()
2: insertInto(current,emptyCode())
3: w  $\leftarrow$  n
4: while  $0 \leq w$  do
5:   while  $0 < weights[w]$  do
6:     weights[w]  $\leftarrow$  weights[w] - 1
7:     future  $\leftarrow$  newHashtable()
8:     for all C  $\in$  current do
9:       for all v  $\in$  wordsWithWeight(w) do
10:        skip if alreadyInCode(C,v)
11:        skip if cannotBeCoveringCode(C + v,weights)
12:        C'  $\leftarrow$  canonize(C + v)
13:        insertIntoIfNotAlreadyThere(future,C')
14:      end for
15:    end for
16:    swap(future,current)
17:  end while
18:  w  $\leftarrow$  w - 1
19: end while
20: printCodes(current)

```

**Ensure:**  $weights[i] = 0$  for  $0 \leq i \leq n$ ;

---

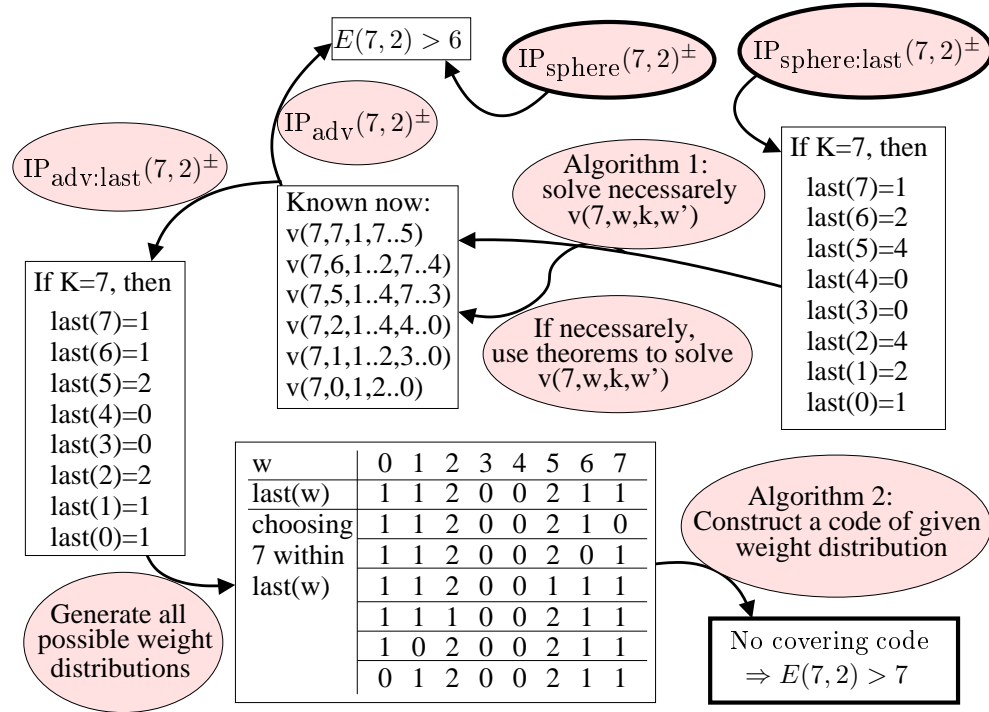


Figure 4.3: An exhaustive search example

We chose this hash function for not any particular reason, it just seemed to produce hash values sufficiently well scattered around the hash table – there probably is still room for improvement for both the efficiency and goodness of the definition (i.e. how fast the hash value can be calculated and how evenly the hash values are scattered around the hash table). Note that the vertexes and edges are always (and should be) gone through in the same order.

**endingCriteria():** We used the total number of generated distinct (non-isomorphic) graphs,  $J$ , as the ending criterion. With a large problem and too large  $J$  the algorithm demands more memory than available, which results in unsuccessful run. The proper value (as large as possible) for  $J$  depends on  $n$  and  $w$  (i.e. the larger  $\binom{n}{w}$  is, the smaller the largest value of  $k$  can be). In practice  $v(n, w, k, w')$  can be solved for  $4 \leq k \leq 16$  with Algorithm 1 (with  $J$  varying between 4000 and 200000).

**canonicalize():** This refers to labeling the graph canonically (canonical labeling is the same for all isomorphic graphs). We used *nauty* [14] do to the canonical labeling – see [14, 22] and their references for more information on the subject.

**printMaxCoverages():** The thing to be noticed here is that the maximal coverage printed is not a coverage of a single code, but the maximum number of words of certain weight that can be covered with a code. It could happen, that there

does not exist a single constant weight code (with weight  $w$  and cardinality  $k$ ) attaining the values of  $v(n, w, k, w')$  for different  $w'$ .

**canNotBeCoveringCode( $C$ , array  $weights$ ):** This method checks the actual coverage – i.e. how many words of different weights are covered – of already assigned codewords ( $C$ ) and sums (over the weights) it up with the upper bound of possible coverage of unassigned codewords ( $weights$ ), which is obtained by using  $v(n, w, k, w')$  values. If the sum shows that a partial code cannot be a covering code, then the partial code can be discarded.

It is not clear, whether it is faster to prune codes first by isomorphism and then by coverage upper bound or in the other way around – it depends on the problem instances. However, the difference between pruning order is not significant (typically the running times differ by few percents).

### 4.3 Comments on Exhaustive Search

The comment is about the order, in which the codewords are fixed by the exhaustive search algorithm. The thing to keep in mind here is, that to make the pruning more efficient we would want to fix codewords close each other so that their coverages overlap as much as possible (better sooner than later). In Algorithm 2 the codewords to be fixed are selected in a predetermined order – i.e. first the codeword with weight  $n$ , then codewords with weight  $n - 1$ , and so on.

This is a good approach with asymmetric covering codes, since the codewords with more weight  $R^+$ -cover more words than codewords with smaller weight and proceeding the constructing in order brings forward the coverage overlaps as soon as possible. With unidirectional covering codes the approach is safe but not the best: for some weight distributions the exhaustive search could be performed a lot faster by fixing the codewords in reverse order (i.e. starting from weight 0). For example, when using exhaustive search to show  $E(10, 3) > 13$  it would take about 3000 seconds to go through all weight distributions for either order – but if a “correct” choice of the order is made for each weight distribution, the time necessary is only about 300 seconds.

Even though the performance of the exhaustive search can be optimized by selecting a correct order for fixing codewords, this ordering does not greatly improve the capability of the exhaustive search algorithm (i.e. correct order does not help to improve the lower bounds, it just makes the algorithm run a bit faster). The observation, that the hardest weight distributions to go through with the exhaustive search algorithm tend to be symmetric weight distributions, is also noteworthy.

The next logical step from performing an exhaustive search is to make a greedy algorithm of it – instead of performing an exhaustive search, we now select  $P$  most promising partial codes for further development and discard the rest (note that if  $P$  is sufficiently large, then this greedy algorithm would perform exactly like an

---

exhaustive search). We tried this approach as well, and we found out that the definition of fitness function (will be defined in Section 5.2.1) did not work well with this greedy approach, which is not that surprising after all. So we decided to look into other properties of the partial codes as well, like the amount of overlaps, average times (and variance) a single word was covered as well as some other more exotic properties. Adding these properties with manually tweaked coefficients to the fitness function we managed to improve the performance (greedy algorithm found a covering code with considerably smaller  $P$ ). However, this approach was tested only on few smaller problem instances and rather briefly, so we can not at this stage provide anything concrete about the subject.

# Chapter 5

## Tabu Search

In this section we give some general information about local search methods and how they can be applied to covering code problems. We concentrate our effort on tabu search, a relatively new local search method, which has been used quite successfully with many combinatorial problems.

### 5.1 Local Search

What we want to do is to improve the upper bounds on  $D(n, R)$  and  $E(n, R)$  – and one way to do it is to construct explicit covering codes by a local search method. A meta heuristic means a method, that can be implemented and used rather easily with different type of problems. Simulated annealing, tabu search and genetic algorithms are considered to be meta heuristics. Some consider neural networks also as a meta heuristic, but implementing an neural network algorithm for a arbitrary problem is not as straightforward as with the (other) meta heuristics. The common functions shared all local search methods are the neighborhood function, that defines which words are close to each other, and the fitness function, that tells how good the codes are. More information about different meta heuristics and local searching in general can be found in [1, 20].

The basic idea in simulated annealing is that there is a current solution, which is altered. The algorithm keeps a global temperature, which decreases with time. The temperature together with the fitness function describes the probability, how likely a solution in the neighborhood is selected. The hope is that the solution stabilizes (with a low temperature the probability for accepting a worse solution is very small) into a very good solution. Mathematically speaking simulated annealing is rather appealing choice, since it has been proven in [24] to converge to global optimum. However, in practice the conditions in which the convergence to a global optimum can be guaranteed to happen, are too strict.

Genetic algorithms uphold a set of solutions, a population, from which parent (usually two) solutions are picked and then combined with crossover operations. Along with crossover operations there are mutation operations, which alter individ-

ual solutions a bit. The idea is to select the parents randomly, but giving good solutions a better chance to become parents. Then the crossover operation produces new solutions, children, which are hopefully better than their parents (thus combining the good parts of their parents). The mutations are necessary to introduce “new genetic material” into the population, so that the population would not stagnate into a local optimum.

There are not many papers (if at all) within coding theory, in which genetic algorithms were applied. The reason is probably the fact, that even though it does intuitively seem a very promising idea to combine two good codes in some meaningful way in order to produce a bit better codes, it seems to be very hard to come up with good crossover operations – the good qualities of codes seem to get broken down rather easily.

We did a simple implementation (see Algorithm 3 for a generic genetic algorithm) of a genetic algorithm with following parameters: the codes were assumed to have a certain weight distribution and cardinality; the crossover operation combined two codes,  $C_1$  and  $C_2$ , producing also two codes by selecting a cut weight  $w_c$ , above which the codewords were picked of  $C_1$  and below which the codewords were taken from  $C_2$  (and vice versa); mutation operation changed a codeword of weight  $w$  into another codeword of the same weight. This simple implementation produced better codes than just constructing codes randomly, but it was no way near the performance of tabu search or simulated annealing.

---

**Algorithm 3** A generic genetic algorithm

---

```

1: pop =initialPopulation()
2: while (!coveringCode(pop) and !endCriterion()) do
3:   newPop ← emptyPopulation()
4:   while !populationFull(newPop) do
5:     parents ← selectParents(pop)
6:     newPop ← newPop + produceChildren(parents)
7:   end while
8: end while

```

---

We also did some preliminary testing with simulated annealing, see Algorithm 4 for a generic simulated annealing algorithm, using the same neighborhood function as tabu search – see Section 5). The difference between the actual results (how small covering codes were found) with simulated annealing and tabu search were not that big. According to these preliminary tests it is impossible to say, whether tabu search is superior to simulated annealing or is it the other way around.

Formal constructions, which were discussed in Sections 2.2.1 and 2.2.2, are more appealing in the sense, that they tell something general about the codes. In contrast finding few separate codes using local search algorithms does not seem to be a solution to the general problem – but codes found this way may prove be rather useful, when designing new better constructions for covering codes.

---

**Algorithm 4** A generic simulated annealing algorithm
 

---

```

1:  $T = \text{initialTemperature}()$ 
2:  $C = \text{initialGuess}()$ 
3: while (!coveringCode( $C$ ) and !endCriterion()) do
4:   Select  $C' \in \mathcal{N}(C)$  randomly
5:    $r \leftarrow \text{random}(0, 1)$ 
6:   if  $r < e^{(\mathcal{F}(C') - \mathcal{F}(C))/T}$ 
7:     then  $C \leftarrow C'$ 
8:    $T \leftarrow \text{updateTemperature}(T)$ 
9: end while

```

---

## 5.2 Tabu Search

Tabu search is a local search method, that can be thought of as a variation on the theme of steepest ascent, i.e. *hill-climbing* algorithm. In a default hill-climbing algorithm the best solution in the neighborhood is chosen as the current solution. The search ends when no more improvements can be made – that is, when a (local) optimum is encountered.

The idea behind the tabu search is to combine a hill-climbing algorithm (which goes towards optima very rapidly) with a local optimum escape mechanism. Still the best solution in the neighborhood is selected, but it may be worse than the current solution. This allows the algorithm to escape from the local optima. However, without any other modifications there is quite a big chance, that the algorithm would next select the local optimum again and end up looping between few solutions. The additional alteration is to ban the algorithm from reentering the (recently) visited solutions, and this mechanism is called *tabu list*.

Tabu search was discovered in 1970s and reached its current form in the late 1980s. The intriguing part of tabu search is, that though no clean proof of its convergence to global is known (there are few proofs for tabu search variants, which might be closer to simulated annealing than tabu search), the algorithm has shown remarkable performance. The reader is referred to [11, 21] and their references for further information on tabu search.

We chose tabu search to be our local search method. The decision was mainly based on earlier results with tabu search [16] as well as some preliminary testing of our own. In our implementation we fix the cardinality of the code in the beginning, i.e. we modify the codewords in order to get a covering code. A code for a generic tabu search algorithm is shown in Algorithm 5.

**Example** A short example about tabu search is shown in Figure 5.1: Initial code was constructed randomly. A step in the algorithm has following structure: a step begins with searching the uncovered word (to be covered) lexicographically – and 6 is the first uncovered word encountered. Next the neighborhood is determined: if any of the words 6, 7 or 14 would be a codeword, then the word 6 would be covered.

**Algorithm 5** A generic tabu search algorithm

- 1:  $C = \text{initialGuess}()$
- 2: **while** (!coveringCode( $C$ ) **and** !endCriterion()) **do**
- 3:   Select  $C' \in \mathcal{N}(C) \setminus \mathcal{T}()$  so that  $\mathcal{F}(C') \leq \mathcal{F}(C'')$  for all  $C'' \in \mathcal{N}(C) \setminus \mathcal{T}()$
- 4:   updateTabuList( $C, C'$ )
- 5:    $C \leftarrow C'$
- 6: **end while**

So the neighborhood is a *Cartesian product* of codewords and 6, 7 and 14 (and the size of the neighborhood is  $6 \cdot 3 = 18$ ). The entire neighborhood is searched through, and there are six different changes that seem equally good (resulting in codes with  $\mathcal{F}() = 3$ ) – one of them is picked randomly (here the choice is  $5 \rightarrow 7$ ). Now one step is completed and the next step can begin (determine the uncovered word, 13; determine the neighborhood,  $\{0, 3, 7, 8, 11, 12\} \times \{13, 15\}$ ; and select the next change from the most promising ones  $\{0 \rightarrow 15, 8 \rightarrow 15\}$ , which is not tabu – unless the perspiration criteria allows the change). Either of the changes result in a covering code, so no need to use the tabu list actually arose during this example.

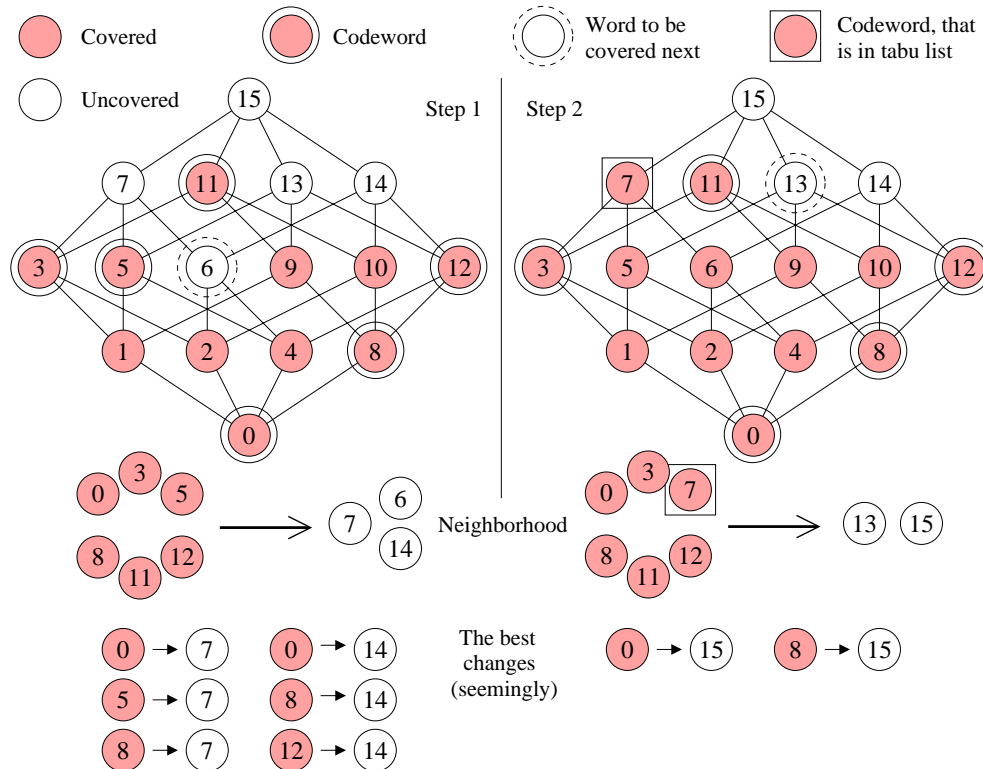


Figure 5.1: A tabu search example

### 5.2.1 Fitness Function

The fitness function,  $\mathcal{F}()$ , is a very crucial part of any local search method. The better the fitness functions is the more it directs the local search to the correct direction, i.e. global optimum (here, towards a covering code). And if the fitness function directs the search towards a valid solution fast, even better. However, defining such a nice fitness function for the global scope is hard, if not impossible, so one has to settle to examine the local scope.

Other nice quality – or maybe even a requirement – for fitness function is that it is computable with as little effort as possible: either the fitness function can be calculated very fast from the scratch, or the current fitness value can be updated quickly according to the changes made.

The fitness function we will use is a rather obvious one, the total number of words not covered by the current solution (code)  $C$ . This definition is rather intuitive and has the nice property, that  $C$  is a covering code only if  $\mathcal{F}(C) = 0$ . Furthermore, when comparing two codes, the one with smaller  $\mathcal{F}()$  value can be considered better, simply because it covers more words and is closer, in some sense, to a covering code. The value of this fitness function can be updated efficiently by keeping track of how many times words are covered (array  $cov[]$ ):

**Require:**  $cov[]$  is array of words, so that  $cov[i]$  tells the number of times the word  $i$  is covered.  $C$  is the current code and the change  $c \rightarrow c'$  ( $c \in C, c' \notin C$ ) is about to take place. The change caused to the current fitness value is  $\Delta\mathcal{F}$

```

1:  $\Delta\mathcal{F} = 0$ 
2:  $S \leftarrow$  set of words  $R$ -covered by  $c$ 
3: for all  $i \in S$  do
4:    $cov[i] \leftarrow cov[i] - 1$ 
5:   if  $cov[i] = 0$  then  $\Delta\mathcal{F} \leftarrow \Delta\mathcal{F} + 1$ 
6: end for
7:  $S \leftarrow$  set of words  $R$ -covered by  $c'$ 
8: for all  $i \in S$  do
9:   if  $cov[i] = 0$  then  $\Delta\mathcal{F} \leftarrow \Delta\mathcal{F} - 1$ 
10:   $cov[i] \leftarrow cov[i] + 1$ 
11: end for
12:  $C \leftarrow (C - c) + c'$ 
13: return  $\Delta\mathcal{F}$ 

```

### 5.2.2 Comments on Fitness Function

When comparing two codes one has to be a bit careful. It would be easy to say that the code with smaller  $\mathcal{F}()$  value is better (it does cover more of the space and hence would be “closer” to a covering code), but is hard to say, which code really is better (i.e. has a better potential to be modified into a covering code). For example a code  $C$  with  $\mathcal{F}(C) = 1$  sure seems to be a promising code (only one uncovered word), but

it does no good if there is no way to make that (apparently so small) change to make  $C$  a covering code (i.e.  $C$  is a local optima for the used local search method, and it is so steep that the local search method can not get out of it). Maybe some other much worse (in  $\mathcal{F}()$  sense) code would have a much better structure and would have the true potential to be transformed into a covering code.

The above mentioned becomes very apparent, if a code is constructed from a scratch, adding one code word at a time and always greedily (minimizing  $\mathcal{F}()$ ). As the construction seems to go smoothly in the beginning, there are only bad left (that do not cover that much) left for the final codewords. It would probably have been better to make some ungreedy choices (or redefine  $\mathcal{F}()$ ) at some point, so that the partial code would have in some sense better structure.

### 5.2.3 Neighborhood Function

The definition of the neighborhood function,  $\mathcal{N}()$ , is the other rather important part of any local search algorithm. The neighborhood defines the solutions (codes) that are, in some sense, near the current solution. What would be desirable is that the neighborhood would contain only the good solutions (solutions that are closer to a global optimum). The local search algorithms tend to take two approaches to the neighborhood: either all solutions in the neighborhood are checked (and the seemingly best is chosen) or a solution is picked randomly until it is accepted as the current solution. For example the latter approach is adopted in simulated annealing whereas the former is (usually) used with tabu search algorithms.

One other thing to notice is that the size of the neighborhood (at least with the former approach) has a quite impact on the running time. So it would be desirable for the neighborhood to be as small as possible – but not too small, since that would restrain the local search algorithm too much and disable the algorithm from escaping a local optimum.

We use the neighborhood that was introduced to covering codes in [16]. First an uncovered word is searched lexicographically, then all changes, which would cover that word, are said to be in the neighborhood. A change is made and the next uncovered word is searched starting from the last one. What is nice with this definition is that size of the neighborhood is almost constant and that the neighborhood necessarily includes changes, which can result into a covering code (every word must be covered by a codeword). In addition implementing this neighborhood in practice is straightforward.

Another idea for neighborhood (mentioned in [16] too) is to use the same definition as above, but apply it to all uncovered words (instead of just one). This neighborhood becomes very large and slows down the algorithm – and the gains are not that great (for some problem instances the chance of finding a covering code increases a bit), but sometimes it might be worth of the extra running time.

One more idea for neighborhood (introduced in [17]) is to consider all uncovered words and all the changes that could cover them, with one additional condition: for

a change  $c \rightarrow c'$  to be in the neighborhood, the Hamming distance between the old and new codeword  $d_H(c, c') = 1$ . Although this definition produced about as good codes the first mentioned, it did it in lesser time. The downside for this definition is that it is a bit more unstable, i.e. it works for asymmetric covering codes, but for unidirectional (and normal) covering codes the first definition seems to perform better.

#### 5.2.4 Comments on Neighborhood Function

Instead of tweaking around the neighborhood, one could also change the problem a bit temporarily, solve the modified problem, and then change the solution to the modified problem back into a solution to the original problem. Such an approach was introduced in [17] for asymmetric covering codes, even though the approach was not investigated fully.

The general idea goes as follows: The aim is to find a new record-breaking code attaining new upper bound on  $D(n, R)$ , and the starting point is the previous record breaking code  $C = (n, D(n, R))^+R$ . First  $i$  coordinates are selected randomly, let us say the last  $i$  coordinates (w.l.o.g.). The codewords having 1 in any of those coordinates are considered to be fixed. Let the  $\hat{C} \subseteq C$  be the set of codewords having 0 in all those coordinates. Next a codeword is removed from  $\hat{C}$  (randomly) and the algorithm tries to change the remaining code into a code covering almost all (some of them are covered by the fixed codewords) words of type  $(v_1, v_2, \dots, v_{n-i}, 0, 0, \dots)$ . If this search is successful and a  $\hat{C}'$  is the new code, then the code  $C - \hat{C} + \hat{C}'$  is a record breaking covering code on the original problem.

Note that the search actually takes place in  $Z_2^{n-i}$  instead of  $Z_2^n$ . The advantage here is, that when the problem becomes smaller, it might be easier to solve it more efficiently than the whole problem (for instance, use a larger neighborhood, that would be too large to be used with the original problem). The disadvantage is, of course, the fact that part of the solution is fixed and the search may be doomed to fail (the fixed part of the code is simply not good enough).

#### 5.2.5 Tabu List

The tabu list is the mechanism, which should enable tabu search to escape from local optima and proceed towards better solutions – hopefully all the way to a global optimum. The idea is simple: keep the visited solutions in the memory and do not visit them again – in practice it is impractical to keep all visited solutions in memory, so one has to settle either to subset of (recently) visited solutions or to some estimation of the visited solutions (i.e. not storing the actual solutions, but the changes that were made recently). Hence the name “tabu list” comes from this list of banned solutions/changes. Sometimes a special condition arise that would make it sensible to ignore the tabu list. For example, if ignoring the tabu list would lead into a better solution than ever before, it might make sense to allow the chance.

These special circumstances are handled with perspiration criteria.

Our approach in this thesis is to keep a list of changes in the solution, that is, once a word is changed to a codeword it will not be changed during the time it is in the tabu list – so this definition of tabu list has one parameter, tabu list size  $T$ , which tells how many steps of the algorithm a codeword remains in the tabu list. We used only one perspiration criterion: a change is made whenever it leads to a covering code. This definition for tabu list was used, for instance, in [16, 18].

One can (and should) always wonder, how good the definition above is. Our own experiments give the feeling, that even though for some problems instances there are better definitions available (resulting in either slightly faster algorithm, slightly more-likely-to-succeed algorithm, or both), this definition is very general and works rather well regardless the problem instance.

For example, one other definition for tabu list was used in [17] (combined with neighborhood mentioned in the previous section). It was defined as follows: A change  $c \rightarrow c'$  is tabu, if  $c$  was changed into  $c'$  (or vice versa) during last  $T$  iterations.

### 5.2.6 Ending Condition

The trivial point to end search is the point when a solution matching some quality criteria is found. With covering codes this would mean that a covering code is found. But such a happy ending is not always possible (either no such code exists, or it is too hard for the algorithm to find), so an another ending mechanism should be included. Here we give the algorithm the number of iterations,  $L$ , it can run without improving the best (during the current search) found solution. The motivation here is to end the unfruitful search as soon as possible. On the other hand, if there seems to be happening even slight improvement, then the search should be allowed to continue longer.

One other approach would be to simply state the maximum number of iterations, during which the algorithm is run (if it does not find a covering code sooner). Whether this approach is inferior to the first mentioned is not that clear: for individual runs this may be so, but when running the algorithm over large number of times and comparing the actual successful runs (a covering code was found) with respect to running times, the results seem rather similar.

### 5.2.7 Initial Guess

The initial guess, the code from which the algorithm starts the search process, has quite an impact on both running time and the probability of successful run when using tabu search (in this respect tabu search differs quite fundamentally – at least in theoretical sense – from simulated annealing, since the initial guess should not really matter with simulated annealing [24]).

There are few ways to construct the initial guess: The simplest way is to create it randomly. The problem with this approach is that the initial code is not very

$R \setminus n$	$D(n, R)$							$E(n, R)$						
	7	8	9	10	11	12	13	7	8	9	10	11	12	13
1	X	X	X	X	X			X	X	X	X	X		
2	X	X	X					X	X	X	X	X		
3	X	X	X	X				X	X	X	X	X		
4	X	X	X	X	X			X	X	X	X	X	X	
5	X	X	X	X	X	X		X	X	X	X	X	X	X
6	X	X	X	X	X	X	X	X	X	X	X	X	X	X
7	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table 5.1: The impact of initial guess

close to a covering code (i.e. neighborhoods based on all uncovered words are large) and with larger codes the searches tend to get stuck into local optima; another way to go is to construct (see Sections 2.2.1 and 2.2.2) a code or pick an existing one, and remove a codeword from it randomly.

In Table 5.1 the “X” marks those problem instances, in which tabu search algorithm can find (in practice, in theoretical sense there is always a chance) a covering code attaining the current best known upper bound on  $D(n, R)$  and  $E(n, R)$  (Tables 7.2 and 7.3) using a randomly constructed initial guess. As can be seen from the table, for the smaller problem instances using a randomly constructed initial guess works well, but for larger problems removing a codeword from existing code works better (the first existing code were either constructed with direct sum construction – Theorem 2.2.4 or 2.2.8 – or with the algorithm itself with random initial code of large cardinality).

### 5.3 Comments on Implementation

While implementing a local search method one has to bear in mind the fact that the total performance of the algorithm is not necessarily the sum of its individual parts. For example, just changing the definition of the tabu list to the one described in the last paragraph of Section 5.2.5 (as in [17]) declines the overall performance of the whole algorithm.

The lesson learned here is that it is dangerous to jump into conclusions (i.e. do some preliminary testing with one implementation and regard the implementation inferior/superior just based on that). The different parts of the algorithm (like neighborhood, fitness function and tabu list) should support each other – and this should be taken into consideration already when doing the design.

It is hard to say anything general about what this should mean in practice, but one should bear in mind that a very good idea (for instance for fitness function) might seem bad just because the other parts of the implementation do not support the idea.

# Chapter 6

## Empirical Tests

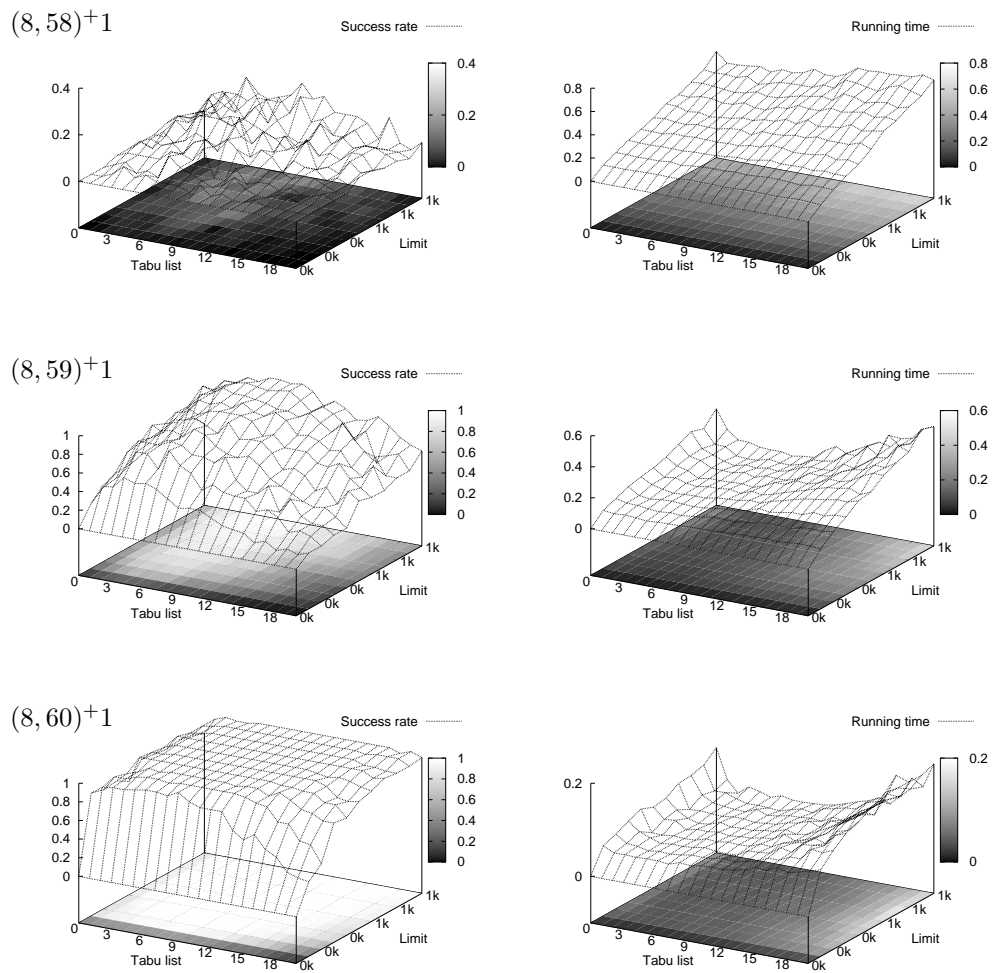
In this chapter the empirical results regarding the tabu search and exhaustive search are examined.

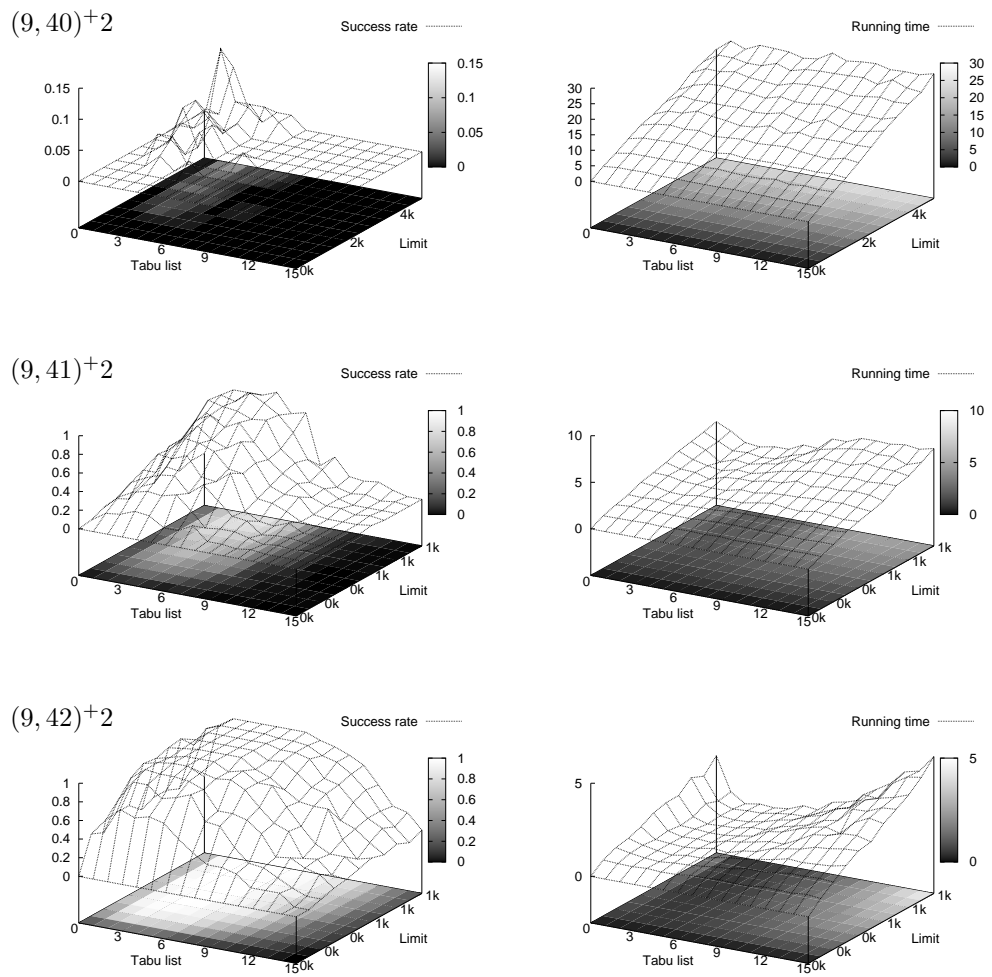
### 6.1 Tuning Tabu Search

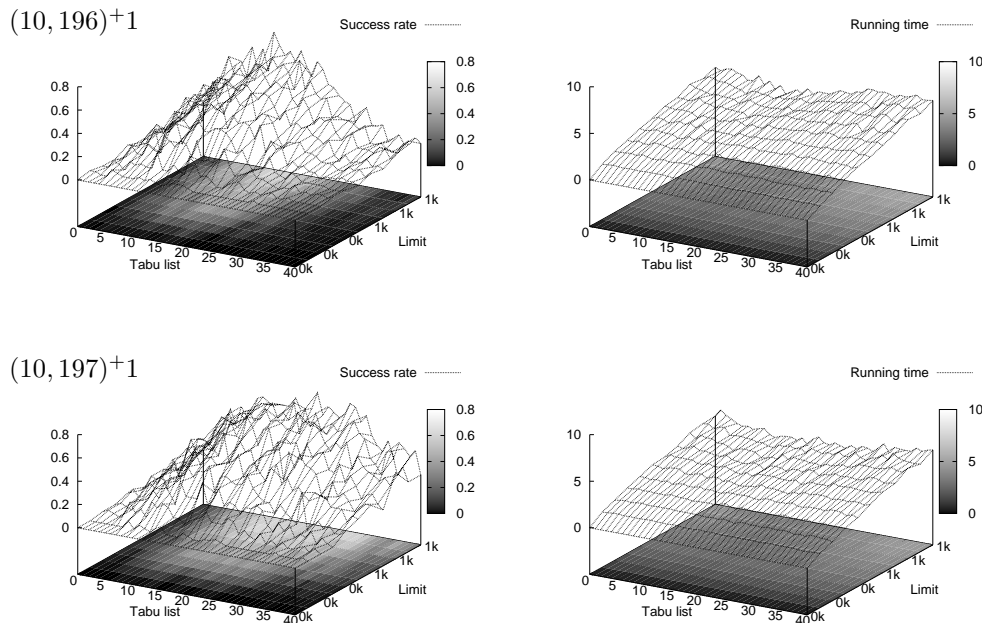
The algorithms were implemented with standard ANSI C. Empirical tests were run on 10 2.4 GHz Celeron Debian Linux machines with 1 GB ram. For each parameter combination the algorithm was run 50 times, and result of the run (whether a covering code was found or not) and the total running time were measured. The time was measured with C function *getrusage()*, which measures the actual resources (such as CPU time) used by the process. Measuring the actual real world time was out of the question, since the tests were run on shared university computers, in which other users might also do something requiring a lot of computing power.

Test runs were carried out for the tabu search algorithm for several problem instances. The instances were selected to be those, which were not too easy (algorithm would fail every now and then) but the algorithm could still found a code attaining the current best known upper bound by starting with a random code as the initial guess. The objective was to find out how the different parameter values (the tabu list size and the allowed iterations,  $L$ ) affect the performance of the algorithm. The results are presented in Figures 6.1, 6.2, 6.3, 6.4 and 6.5, in which success rate refer to the number of successful runs (a covering code was found) versus the total number of runs, and the running time refers to the average duration (in seconds) of a single execution of the algorithm.

One interesting question is, whether it is more efficient to run the algorithm with longer limit,  $L$ , therefore increasing the chance of finding a covering code; or would it be wiser to run the algorithm several times with smaller  $L$  during the same period of time. In Figure 6.6 the average running time is divided with success rate for some of the problem instances. The result of this division can be considered to be the efficiency of algorithm with the corresponding parameter settings – i.e. how much time on average is needed to find a covering code (with a probability near 100%).

Figure 6.1: Tuning tabu search:  $(8, K)^{+1}$

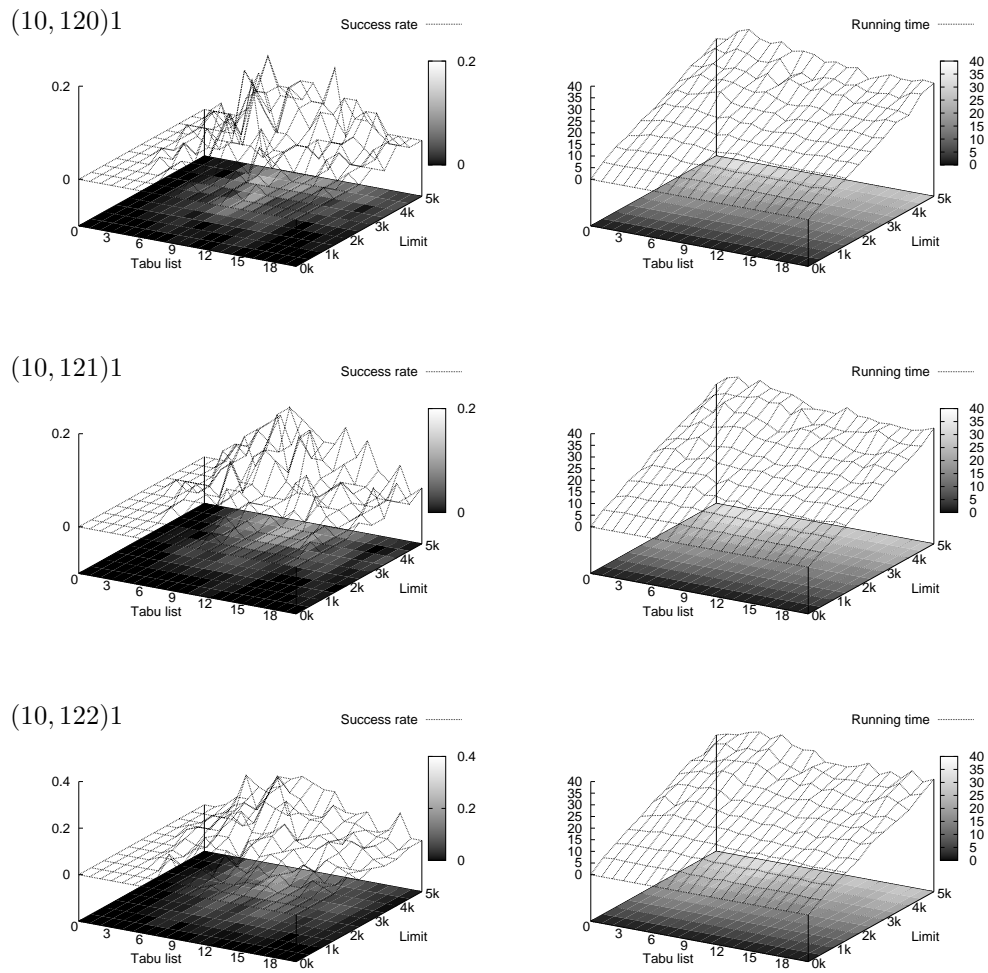
Figure 6.2: Tuning tabu search:  $(9, K)^{+2}$

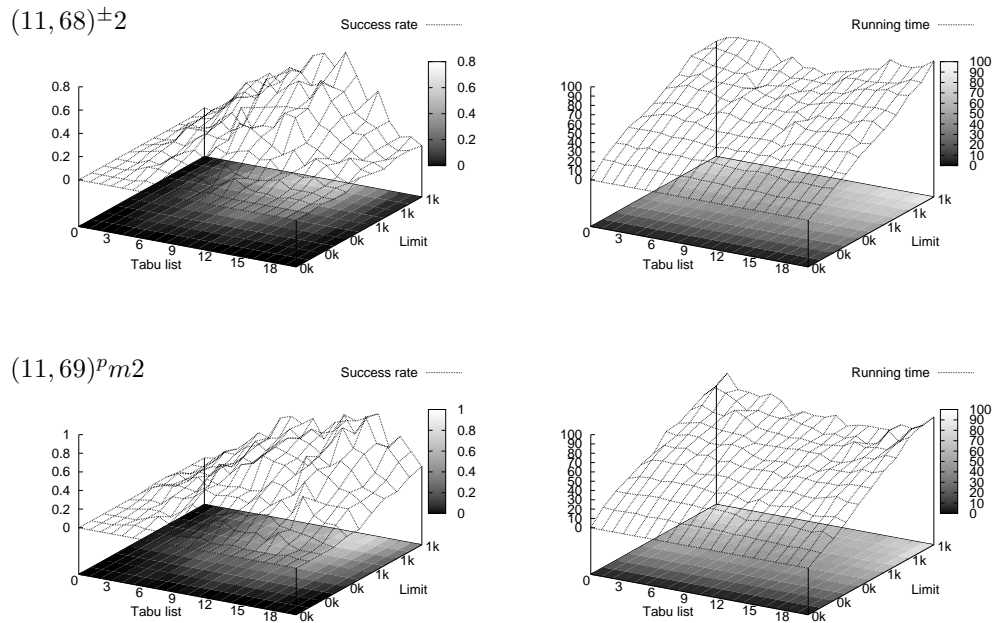
Figure 6.3: Tuning tabu search:  $(10, K)+1$ 

### 6.1.1 Conclusions on Tabu Search

Rather wide test runs were carried out with tabu search in order to find about the behavior of tabu search, and this included test runs in which the algorithm was used to find a bit worse code than it would have been possible: logic behind this was to study, is the seemingly optimal tabu list size same regardless the cardinality of the target covering code (i.e. would it be possible to determine the optimal tabu list size by running the algorithm first with larger – and easy to find – codes). And it would seem that the algorithm behaves so, but only to some extent, because the range of “seemingly optimal” tabu list size widens rather fast – Figures 6.1 and 6.2 – which can not be fully compensated by shortening the search, and for larger and harder problems the tabu list size is not that crucial any more – Figures 6.3, 6.4 and 6.5).

As for the differences between asymmetric and unidirectional problem instances – tabu search seems to perform (with the current neighborhood, fitness function and tabu list) much more nicely with the asymmetric case. For example the running time for successfully searching  $(10, 120)1$  code is about seven times longer (the running time needed to get about 20% success rate) than using the same algorithm for finding  $(10, 196)+1$  codes (Figures 6.3 and 6.4). This phenomenon seems to repeat for every asymmetric-unidirectional problem pair, and this probably happens because the asymmetric covering behaves more nicely than unidirectional covering: with asymmetric coverings a single codeword covers less words – and when a codeword is changed, it does not affect so many words – only the subset of those which weight

Figure 6.4: Tuning tabu search: (10,  $K$ )1

Figure 6.5: Tuning tabu search:  $(11, K)^{\pm 2}$ 

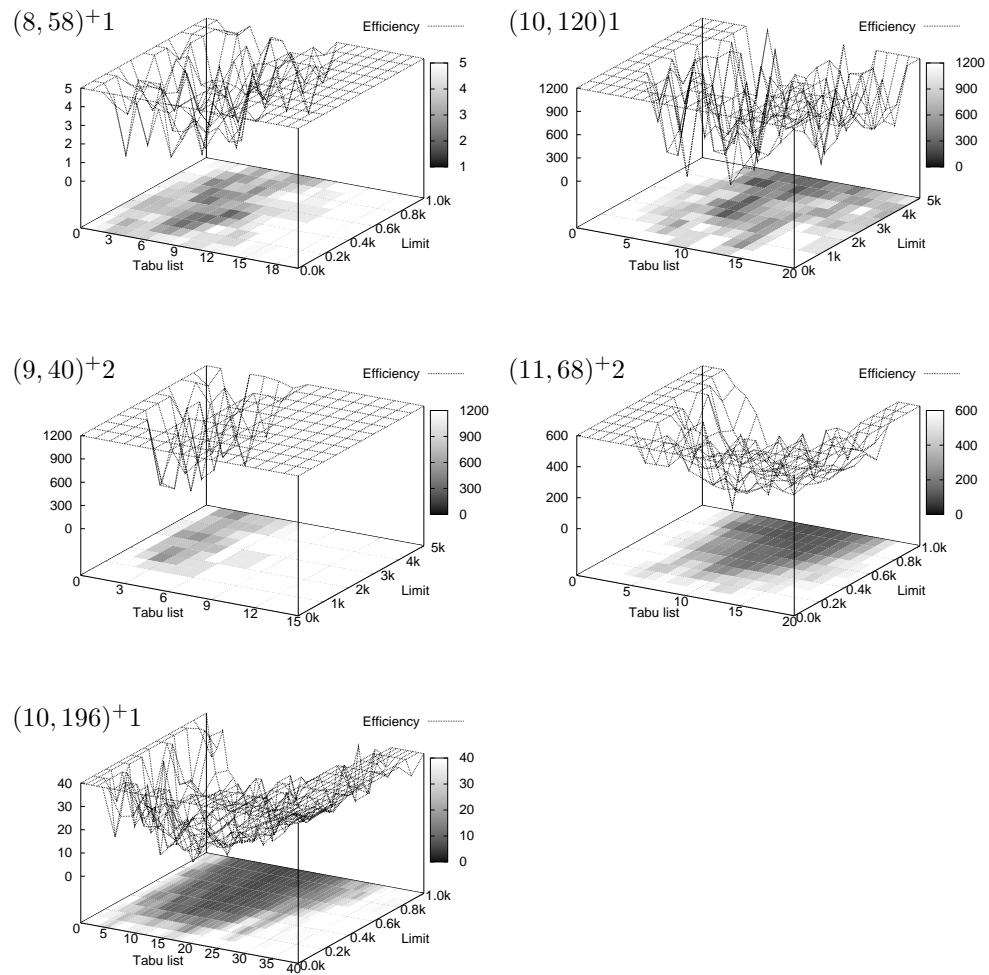
less than the codeword itself, whereas changing a codeword in an unidirectional code has greater impact (so it is “easier” to break the code).

## 6.2 Exhaustive Search

The exhaustive search algorithms were implemented to solve  $v(n, w, k, w')$ , Algorithm 1 and to show non-existence of covering codes of given cardinality, Algorithm 2. With exhaustive search algorithms one must always consider, whether the algorithm really works (e.g. the reason why algorithm does not find codes really is because there are none – and not because there is an error in implementation) and the attained results can be trusted.

We are confident that the implemented algorithms perform correctly. This is because for Algorithm 1 the results coincide with  $\mathcal{A}(n, d, w)$  and with Algorithm 2 the results are correct with smaller problem instances. For the convenience (for others to verify the results presented in this thesis regarding the exhaustive search) the total number of covering codes with cardinality of the exact bound are listed in Table 6.1.

Evaluating these results is difficult, since no such listing were made on earlier research, even though in [3] they state that there are only four non-isomorphic asymmetric  $(4, 6)^{+1}$  codes – which seems to be an error. The four isomorphic codes

Figure 6.6: The impact of  $L$  and  $T$  on tabu search

$n \setminus R$	1	2	3	4	5	6	7	8	9
1	1								
2	2	1							
3	1	3	1						
4	8	3	4	1					
5	3	10	5	5	1				
6	19	4	1	8	6	1			
7	32	402	15	4	11	7	1		
8			9676	4	8	15	8	1	
9				388	78	14	19	9	1

Table 6.1: The total number of non-isomorphic  $(n, D(n, R))^+R$  codes

mentioned in [3] are the upper four codes in Figure 6.7, but there are four additional (the lower four) non-isomorphic codes.

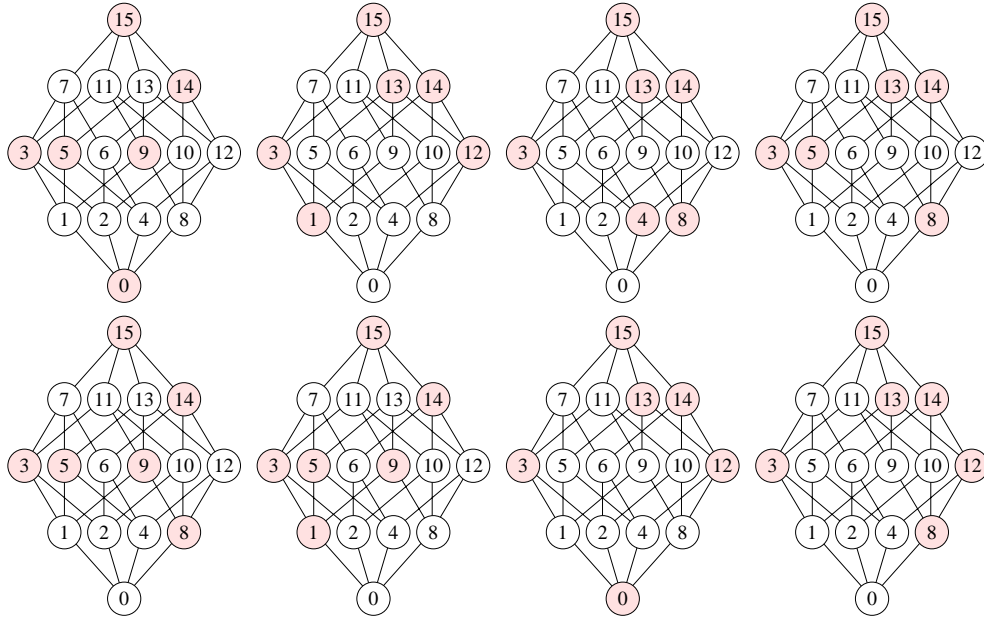
### 6.2.1 Conclusions on Exhaustive Search

The thing to be noted with the exhaustive search is that we were always able to improve a lower bound attained only by using  $\text{IP}_{\text{adv}}$ . The explanation for this is rather simple: since exhaustive search used the same results (that is,  $v(n, w, k, w')$ ) as  $\text{IP}_{\text{adv}}$  for pruning codes, it were able to improve the lower bounds slightly (1–3 codewords). However, there should be a lot of room for improving  $\text{IP}_{\text{adv}}$  or designing a better method for attaining lower bounds – and such an improved method could most likely be used in exhaustive search to prune code more efficiently.

## 6.3 Comments on the Results

A curious question regarding the attained lower and upper bounds on  $D(n, R)$  and  $E(n, R)$  is that how good are the bounds (i.e. the used methods) really are? Results listed in this thesis seem rather impressive (many lower and upper bounds attained), but the research on asymmetric and unidirectional covering codes is scarce. So how to evaluate how good the bounds really are (i.e. how close are they to the actual values and how easy would it be to improve them)?

We did some testing on  $K(n, R)$ , which have been researched widely and its very unlikely to get easily any improvements on the bounds. Idea would be to compare the results with  $K(n, R)$  the known best result – and observe the difference. Then that difference might be used to estimate the results on  $D(n, R)$  and  $E(n, R)$ . Even though the problems differ to some amount, this estimation would give us some kind of feeling about the goodness of the results (feeling that might be proven completely wrong though).

Figure 6.7: All non-isomorphic  $(4, 6)^+1$  codes

The feeling we got with tabu search were, that tabu search performs well (the best known upper bound is found) for at least  $n \leq 10$  and for  $K(n, R) \leq 20$ , but for the larger problems there might be room for improvement.

The exhaustive search algorithm can handle the lower bounds on  $K(n, R)$  for small  $n$  and  $K(n, R)$ . Particularly interesting are the lower bounds  $K(10, 3) \geq 9$  and  $K(12, 4) \geq 8$ , which might be improved even with the current implementation of the exhaustive search, that is if few months of computing time were used (this is just a rough estimate – and the computer used for the calculations should have quite a lot of memory and raw processing power to handle the breadth-first search).

# Chapter 7

## Summary

The results of earlier studies, which still contribute to the best know bounds on  $D(n, R)$  and  $E(n, R)$ , were listed in Chapter 2. Upper bounds were not improved in this thesis, but we managed to improved several lover bounds on  $D(n, R)$  by combining the results of different papers – theoretical part from [7] combined with integer programming and exhaustive search used in [18].

The lower bounds on  $\lceil \frac{1}{n}\phi(n, R) \rceil$  (used with Theorem 2.2.3) are listed in Table 7.1. The unmarked entries are attained with  $\text{IP}_{\text{sphere}:\phi(n, R)^+}$  – as in [7], where the actual values were omitted. For some entries a better result could be achieved with  $\text{IP}_{\text{adv}:\phi(n, R)^+}$ , and they are marked with subscript  $a$ .

The best known lower and upper bounds on  $D(n, R)$  and  $E(n, R)$  are listed in Tables 7.2 and 7.3. The keys to Tables are as follows:

	Lower bounds (subscripts)
<i>in italics</i>	Theorem 2.2.6
<i>a</i>	$\text{IP}_{\text{adv}}$ (IP Problem 3)
<i>e</i>	Exhaustive search (Section 2)
<i>h</i>	Theorem 2.2.3 and results in Table 7.1 (better than in [7])
<i>o</i>	$\text{IP}_{\text{exact}}$ (IP Problem 1)
<i>t</i>	Theorem 2.2.12
	Upper bounds (superscripts)
*	Tabu search (Section 5), as used in [18]
	Both
<b>bold</b>	Bound on $K(n, R)$ [4]
Unmarked	Theorem 2.2.3 and results in Table 7.1
	Theorem 2.2.10, Corollary 2.2.9, lower bound on $K(n, R)$
1	Attained in [7]
2	Attained in [3]
3	Attained in [9] using a local search
4	Attained in [17]

$n \setminus R$	1	2	3	4	5	6	7	8	9
2	1	0							
3	1	1	0						
4	3	1	1	0					
5	4	2	1	1	0				
6	7	3	1	1	1	0			
7	12	5	2	1	1	1	0		
8	22	7	3	2	1	1	1	0	
9	$a43$	$a13$	$a6$	2	2	1	1	1	0
10	$a78$	$a21$	$a9$	4	2	1	1	1	1
11	$a142$	$a36$	$a14$	$a6$	3	2	1	1	1
12	257	61	$a22$	9	$a5$	3	2	1	1
13	483	106	34	14	7	4	2	2	1

Table 7.1: Lower bounds on  $\lceil \frac{1}{n}\phi(n, R) \rceil$ 

$n \setminus R$	1	2	3	4	5
1	$1$				
2	$2$	$1$			
3	$3$	$2$	$1$		
4	$16^1$	$3$	$2$	$1$	
5	$10^1$	$5^1$	$3$	$2$	$1$
6	$118^1$	$18^1$	$4$	$3$	$2$
7	$231^2$	$o14^3$	$o7^1$	$4$	$3$
8	$258^2$	$e22-23^3$	$e12^3$	$6$	$4$
9	$h101-106^2$	$h35-40^4$	$h18-19^3$	$e10$	$6$
10	$h179-196^2$	$h56-70^3$	$h27-31^4$	$14-15^4$	$e8^4$
11	$h321-352^2$	$h92-121^4$	$h41-51^4$	$h20-25^3$	$e12-13^4$
12	$569-668^3$	$151-218^4$	$h63-92^4$	$29-42^4$	$h17-21^3$
13	$1052-1253^3$	$257-421^4$	$92-165^4$	$43-71^4$	$24-35^3$

$n \setminus R$	6	7	8	9	10	11
6	$1$					
7	$2$	$1$				
8	$3$	$2$	$1$			
9	$4$	$3$	$2$	$1$		
10	$5$	$4$	$3$	$2$	$1$	
11	$e8^1$	$5$	$4$	$3$	$2$	$1$
12	$11-12^4$	$7$	$5$	$4$	$3$	$2$
13	$15-18^4$	$e10-11^3$	$7$	$5$	$4$	$3$

Table 7.2: The best known bounds on  $D(n, R)$

$n \setminus R$	1	2	3	4	5	6	7	8	9
1	<b>1</b>								
2	<b>2</b>	1							
3	<b>2</b>	2	1						
4	<b>4</b>	2	2	1					
5	<b>7</b>	2	2	2	1				
6	<b>12</b>	4	2	2	2	1			
7	<b>16</b>	<sub>o</sub> 8	2	2	2	2	1		
8	<b>32</b>	<sub>o</sub> 14*	4	2	2	2	2	1	
9	<b>62</b>	<sub>e</sub> 21–24*	<sub>t</sub> 8	2	2	2	2	2	1
10	<b>107–120</b>	<sub>e</sub> 32–36*	<sub>e</sub> 14–15*	4	2	2	2	2	2
11	<b>180–192</b>	<sub>e</sub> 53–68*	<sub>e</sub> 21–26*	<sub>t</sub> 8	2	2	2	2	2
12	<b>342–380</b>	<sub>e</sub> 91–126*	<sub>e</sub> 32–44*	<sub>e</sub> 14–15*	4	2	2	2	2
13	<b>598–704</b>	<sub>e</sub> 157–240*	<sub>e</sub> 48–74*	<sub>e</sub> 20–26*	<sub>t</sub> 8	2	2	2	2

Table 7.3: The best known bounds on  $E(n, R)$ 

The tabu search seems to be a very nice method for finding covering codes – the search is relatively fast and the algorithm is easy to tune (only two parameters, which do not need to be tuned to the optimality for the algorithm to work). The weak point for the algorithm seems to be the fitness function, since it seemingly does not direct the search quite well towards the global optima and makes local optima too steep.

Our suggestion for the next step would be to include other information – which might seem to be rather irrelevant – to the fitness function so that truly good codes can be recognized. Such information could be the average times (and/or variance) a word is covered, distances between the codewords or the uncovered words. Since interpreting and using this information to our advantage directly is hard (since we do not have a clear picture, what kind of properties a good code has), a learning algorithm – like neural networks – would be called for. A learning algorithm needs some data to learn the good properties from, and this data could be provided by an exhaustive search algorithm. For example, exhaustive search could be used to list all partial codes, from which a covering code can be constructed by adding the right codewords.

# Bibliography

- [1] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley, Chichester, 1997.
- [2] E. Agrell, A. Vardy, and K. Zeger. Upper bounds for constant-weight codes. *IEEE Trans. Inform. Theory*, 46:2373–2395, 2000.
- [3] D. Applegate, E. M. Rains, and N. J. A. Sloane. On asymmetric coverings and covering numbers. *J. Combin. Des.*, 11:218–228, 2003.
- [4] R. Bertolo, P. R. J. Östergård, and W. D. Weakley. An updated table of binary/ternary mixed covering codes. *J. Combin. Des.*, 12:157–176, 2004.
- [5] A. E. Brouwer, J. B. Shearer, N. J. A. Sloane, and W. D. Smith. A new table of constant weight codes. *IEEE Trans. Inform. Theory*, 36:1334–1380, 1990.
- [6] G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein. *Covering Codes*. North-Holland, Amsterdam, 1997.
- [7] J. N. Cooper, R. B. Ellis, and A. B. Kahng. Asymmetric binary covering codes. *J. Combin. Theory Ser. A*, 100:232–249, 2002.
- [8] R. B. Ellis, A. B. Kahng, and Y. Zheng. JBIG compression algorithms for "dummy fill" VLSI layout data. Technical report, VLSI CAD Laboratory, UCSD Department of Computer Science and Engineering, 2002.
- [9] G. Exoo. Upper bounds for optimal asymmetric covering codes. [Electronic]. [Referred April 10, 2005]. <URL:<http://isu.indstate.edu/ge/Acodes/>>.
- [10] GLPK (GNU Linear Programming Kit). <URL:<http://www.gnu.org/software/glpk/glpk.html>>.
- [11] A. Hertz, E. Taillard, and D. de Werra. Tabu search. In *Local Search in Combinatorial Optimization*, Wiley-Intersci. Ser. Discrete Math. Optim., pages 121–136. Wiley, 1997.
- [12] G. Kéri and P. R. J. Östergård. Further results on the covering radius of small codes, submitted for publication.

- 
- [13] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, Amsterdam, 1977.
- [14] B. D. McKay. nauty user's guide (version 1.5). Technical Report TR-CS-90-02, Dept. Computer Science, Australian National University, 1990.
- [15] J. Nievergelt, R. Gasser, F. Mäser, and C. Wirth. All the needles in a haystack: can exhaustive search overcome combinatorial chaos? In *Computer Science Today*, volume 1000 of *Lecture Notes in Comput. Sci.*, pages 254–274. Springer, 1995.
- [16] P. R. J. Östergård. Constructing covering codes by tabu search. *J. Combin. Des.*, 5:71–80, 1997.
- [17] P. R. J. Östergård and E. A. Seuranen. Constructing asymmetric covering codes by tabu search. *J. Combin. Math. Combin. Comput.*, 51:165–173, 2004.
- [18] P. R. J. Östergård and E. A. Seuranen. Unidirectional covering codes. unpublished.
- [19] P. R. J. Östergård and W. D. Weakley. Constructing covering codes with given automorphisms. *J. Combin. Des.*, 16:65–73, 1999.
- [20] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors. *Modern Heuristic Search Methods*. Wiley, Chichester, 1996.
- [21] C. R. Reeves. Modern heuristic techniques. In *Modern Heuristic Search Methods*, pages 1–25. Wiley, 1996.
- [22] D. A. Spielman. Faster isomorphism testing of strongly regular graphs. In *Proceedings of the Twenty-eighth Annual ACM Symposium on the Theory of Computing (Philadelphia, PA, 1996)*, pages 576–584, New York, 1996. ACM.
- [23] H. A. Taha. *Integer Programming*. Academic Press, New York, 1975.
- [24] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*, volume 37 of *Mathematics and its Applications*. D. Reidel Publishing Co., Dordrecht, 1987.