Department of Communications and Networking

Computational Methods in Codes and Games

Esa Seuranen

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 119/2011

Computational Methods in Codes and Games

Esa A. Seuranen

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Electrical Engineering for public examination and debate in Auditorium S4 at the Aalto University School of Electrical Engineering (Espoo, Finland) on the 25th of November 2011 at 12 o'clock.

Aalto University
School of Electrical Engineering
Department of Communications and Networking

Supervisor
Patric R. J. Östergård

Instructor
Patric R. J. Östergård

Preliminary examiners
Kris Coolsaet, University of Ghent, Belgium
Lucia Moura, University of Ottawa, Canada

Opponent
Faina Solov'eva, Sobolev Institute of Mathematics, Russia

Aalto University publication series
DOCTORAL DISSERTATIONS 119/2011

ISBN 978-952-60-4370-8 (pdf)
ISBN 978-952-60-4369-2 (printed)
ISSN-L 1799-4934
ISSN 1799-4942 (pdf)
ISSN 1799-4934 (printed)

Unigrafia Oy
Helsinki 2011

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Esa A. Seuranen

Name of the doctoral dissertation
Computational Methods in Codes and Games

Publisher School of Electrical Engineering

Unit Department of Communications and Networking

Series Aalto University publication series DOCTORAL DISSERTATIONS 119/2011

Field of research Information Theory

Manuscript submitted 30 May 2011 **Manuscript revised** 31 October 2011

Date of the defence 25 November 2011 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

This dissertation discusses exhaustive search algorithms and heuristic search methods in combinatorial optimization, including combinatorial games.

In this work unidirectional covering codes are introduced and some theoretical foundations for them are laid. Exhaustive search is used to construct asymmetric covering codes, unidirectional covering codes and multiple coverings with given parameters---or to show that no such codes exist. Integer programming formulations, bounds on maximal coverages of partial codes and code isomorphisms are used to prune the search space.

Tabu search is used to construct asymmetric and unidirectional covering codes---with several record-breaking codes for the former. A new definition for neighborhood is derived.

The traditional board game of go and computer go results are reviewed. The concept of entropy is introduced into the game context as a metric for complexity and for relevance (of features---like distance to the previous move). Experimental results and questionnaire studies are presented to support the use of entropy.

Keywords covering codes, exhaustive search, go, tabu search

ISBN (printed) 978-952-60-4369-2 **ISBN (pdf)** 978-952-60-4370-8

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2011

Pages 52 **The dissertation can be read at** http://lib.tkk.fi/Diss/

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä Esa A. Seuranen	
Väitöskirjan nimi Laskennalliset menetelmät koodeissa ja peleissä	
Julkaisija Sähkötekniikan korkeakoulu	
Yksikkö Tietoliikenne- ja tietoverkkotekniikan laitos	
Sarja Aalto University publication series DOCTORAL DISSERTATIONS 119/2011	
Tutkimusala Informaatioteoria	
Käsikirjoituksen pvm 30.05.2011	**Korjatun käsikirjoituksen pvm** 31.10.2011
Väitöspäivä 25.11.2011	**Kieli** Englanti
☐ **Monografia**	☒ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Väitöskirjassa käsitellään täydellistä hakua sekä heuristisia hakumenetelmiä kombinatorisessa optimoinnissa, mukaanlukien kombinatoriset pelit.

Tässä väitöskirjatyössä esitellään unidirektionaaliset peittokoodit sekä niiden perusominaisuuksia. Täydellistä hakua käytetään listaamaan annetuilla parametreilla kaikki asymmetriset peittokoodit, unidirektionaaliset peittokoodit ja monipeittokoodit---tai osoittamaan, ettei koodeja ole olemassa kyseisillä parametreilla. Täydellistä hakua rajoitetaan kokonaislukuoptimoinin keinoilla, tarkistamalla koodien isomorfisuutta sekä huomioimalla koodisanojen yhteinen maksimipeittävyys.

Tabuhaun avulla löydetään asymmetrisiä ja unidirektionaalisia peittokoodeja. Hakuun kehitetään uusi määritelmä naapurustolle. Asymmetrisien koodien osalta monia olemassa olevia tuloksia parannetaan huomattavasti.

Tietokone-go:ssa saavutettuja tuloksia käydään lävitse. Entropia esitellään menetelmänä mitata pelien kompleksisuutta sekä (pelillisten piirteiden---kuten edellisen siirron etäisyys) merkitsevyyttä. Sekä laskennallisia tuloksia että kyselytuloksia käytetään motivaationa entropian käyttämiselle.

Avainsanat go, peittokoodit, tabu haku, täydellinen haku		
ISBN (painettu) 978-952-60-4369-2	**ISBN (pdf)** 978-952-60-4370-8	
ISSN-L 1799-4934	**ISSN (painettu)** 1799-4934	**ISSN (pdf)** 1799-4942
Julkaisupaikka Espoo	**Painopaikka** Helsinki	**Vuosi** 2011
Sivumäärä 52	**Luettavissa verkossa osoitteessa** http://lib.tkk.fi/Diss/	

Preface

This research was done at the Communications Laboratory (known as the Department of Communications and Networking from 2008 onward) of Helsinki University of Technology during 2005–2008. The finishing touches were done in 2011.

I am deeply indebted to Prof. Patric Östergård for his supervision of the thesis, for the many helpful comments and ideas—as well as for encouragement to (finally) finish the thesis. I would also like to thank my colleagues for productive (well—if not always, then at least entertaining and delightful) discussions over the years.

The work has been funded by the Academy of Finland under Grants No. 100500, No. 107493 and No. 110196. I also give my thanks to the Graduate School of Electrical and Communications Engineering, Tekniikan edistämissäätiö and Nokia Foundation for their financial support.

Otaniemi, 31.10.2011

Esa Seuranen

List of publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I P. R. J. Östergård and E. A. Seuranen, Constructing asymmetric covering codes by tabu search, Journal of Combinatorial Mathematics and Combinatorial Computing 51 (2004) 165–173.

II P. R. J. Östergård and E. A. Seuranen, Unidirectional covering codes, IEEE Transactions on Information Theory 52 (2006) 336–340.

III E. A. Seuranen and P. R. J. Östergård, New lower bounds on asymmetric covering codes, Congressus Numerantium 178 (2006) 57–63.

IV E. A. Seuranen, New lower bounds for multiple coverings, Designs, Codes and Cryptography 45 (2007) 91–94.

V E. A. Seuranen, Introducing playing style to computer go, in: J. van den Herik, J. Uiterwijk, M. Winands, M. Schadd (eds.), Proceedings of Computer Games Workshop 2007 (CGW 2007), MICC Technical Report Series, Amsterdam, The Netherlands, 2007, pp. 81–91.

VI E. A. Seuranen, Entropy in go, ICGA Journal 32 (2009) 34–40.

Author's contribution

In publications [**I–III**] the author of this thesis has been responsible for the development of the algorithms and for the computation of the results. Of the analytical results the author can be attributed II.A.3) in [**II**] and Theorem 5 in [**III**]. The overall effort of writing the text in publications [**I–III**] has been quite evenly divided between the authors.

The author of this thesis is the sole author of publications [**IV–VI**].

Contents

Abstract **i**

Tiivistelmä **iii**

Preface **v**

List of publications **vii**

Author's contribution **viii**

1 Introduction **1**

2 Covering Codes **3**
- 2.1 Definitions and Terminology 3
- 2.2 Variants 4
- 2.3 Summary of Results 6

3 Computer Search **9**
- 3.1 Exhaustive Search 10
  - 3.1.1 Our Implementation 11
- 3.2 Heuristic Search 12
  - 3.2.1 Overview 12
  - 3.2.2 Examples of Heuristic Search Methods 15
  - 3.2.3 Tabu Search 16

4 Games **19**
- 4.1 Minimax Algorithm and $\alpha\beta$ Pruning 19
- 4.2 Entropy 21
- 4.3 Computer Go 22

5 Conclusions **25**

A Rules of Go **27**

B Source Codes **33**

References **35**

Chapter 1

Introduction

Nowadays computational methods and results are present in a wide variety of areas, making them an interesting and relevant object of research. The increasing importance is due to the growth of knowledge and computational power enabling new viewpoints to old problems. For instance the LDPC (low density parity check) codes were discovered already in the 1960s but only recently have the practical applications for these near optimal performance codes emerged [62]. Also the recent advances in computer go with Monte-Carlo tree search (MCTS) [13] would not have been possible ten years ago.

In games, generally speaking, the most common way of making a computer player is to go exhaustively through all possible moves as far ahead as possible, which is essentially an exhaustive search. With limited computational resources the trick of creating a strong computer player boils down to narrowing the search, that is, pruning away moves of low quality. Even with very effective pruning it is not usually possible to reach the end of the game, hence evaluating a game situation is an important part of a successful computer player. Go [10] as a game is particularly interesting, as it is the only well known traditional game in which humans are still superior to computer players.

In contrast to exhaustive search, the aim in heuristic search methods [1, 49] is to provide a reasonably good solution in a reasonable time without going through the whole search space. This is done by embedding heuristics (rules of thumb, intuitions about good solutions, more or less educated guesses, etc.) into the search. And similarly to game playing, evaluation of solutions plays a vital role as it dictates to a large degree in which way the search proceeds.

In this thesis a heuristic search method called tabu search and exhaustive search are used to tackle the combinatorial problem of minimizing the size of (several types of) covering codes [17]. Tabu search [36] is used to construct explicit codes showing upper bounds and exhaustive search is used to prove non-existence of codes with given parameters. Also the field of computer go is shortly reviewed.

The rest of this thesis is organized as follows. In Chapter 2 covering codes are described. Computational methods (exhaustive search and heuristic search) are considered in Chapter 3. Some discussion on games in general (and go in particular) is provided in Chapter 4. Finally, in Chapter 5 some concluding remarks are drawn.

Chapter 2

Covering Codes

In this chapter we will go over the definitions and terminology related to covering codes. We will consider only binary codes in this thesis as the results in [**I**–**IV**] are for binary cases and omit discussion of q-ary codes (although the generalization of many of the definitions and results to the q-ary case would be rather straightforward). We end the chapter with a summary of the obtained results.

2.1 Definitions and Terminology

Let Z^n, where $Z = \{0,1\}$, be the binary Hamming space of dimension n. A ball $B(x,R)$ of radius R centered at a *word* $x \in Z^n$ consist of all words y that differ from x in at most R coordinates. A *code* $C \subseteq Z^n$ is a *(R-)covering code*, if

$$\bigcup_{c\in C} B(c,R) = Z^n.$$

The smallest R such that C is an R-covering code is called the *covering radius*. The minimal cardinality of an R-covering code is denoted with $K(n,R)$.

Multiple coverings are a generalization of covering codes. A code $C \subseteq Z^n$ is a *(μ-fold) multiple covering*, if for each word $x \in Z^n$ there are at least μ *codewords* $c \in C$ so that $x \in B(c,R)$. The minimum cardinality of a μ-fold multiple R-covering code is denoted with $K(n,R,\mu)$. If multiple occurrences of the same codewords are allowed, the code is called a *multiple covering with repeated codewords.*

The *weight* of a word is the number of nonzero coordinates in it and the *weight distribution* of a code is an $(n+1)$-tuple $(w_0, w_1, \ldots, w_n)$, where w_i is the number of codewords with weight i. For convenience, for some R that is understood, if y is in the ball of radius R centered at x, we say that *x covers y* or that y is *covered* by x. We will also follow the convention of referring to the determination of the values of a function—for example, $K(n,R)$—as a *problem* and determination of the value of such a function for the given set of parameters as an *instance*.

The motivations for studying covering codes include football pools and data compression [17]. For instance, assume one is expected to predict results in n matches (without ties being possible). Obviously one needs 2^n guesses in order to be sure to get each match result correct. But if one settles for getting $n-R$ outcomes correct, then a minimal binary covering code of length n and covering radius R—i.e., $K(n,R)$—tells the necessary amount of guesses. As an example of a binary lossy compression, one can use a codeword in a covering code to represent all the words the codeword covers: in a such scheme one achieves the compression rate $K(n,R)/2^n$ and loses information with at most rate R/n.

Covering codes have received a fair amount of attention in the literature—see [17] and its references for general properties of and results on covering codes. Recent developments regarding lower and upper bounds on the minimal cardinalities of covering codes can be found in [44], and [56] is an extensive bibliography on publications regarding covering radius.

2.2 Variants

Different types of covering codes are obtained by modifying the definition of a ball.

A downward directed ball $B^-(x,R)$ of radius R centered at the word x consist of all words y that can be obtained from x by changing 1s to 0s in at most R coordinates. Similarly an upward directed ball $B^+(x,R)$ is the set of words obtainable from x by changing 0s to 1s in at most R coordinates. A code C is an *asymmetric (R-)covering code*, if

$$\bigcup_{c \in C} B^-(c,R) = Z^n.$$

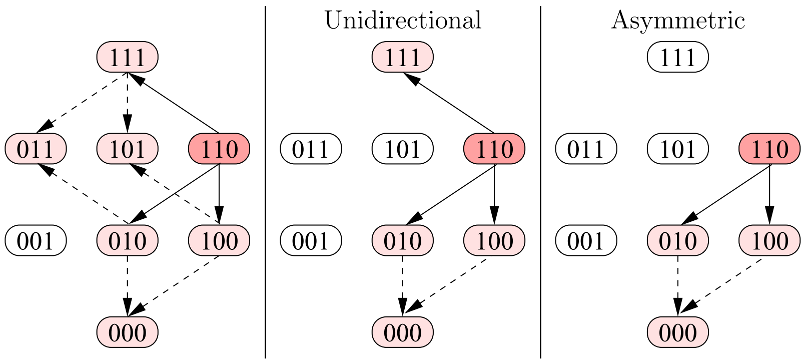

Figure 2.1 Visualization of $B(110,2)$, $B^{-}(110,2) \cup B^{+}(110,2)$ and $B^{-}(110,2)$

Moreover, a code C is a *unidirectional (R-)covering code*, if

$$\bigcup_{c \in C} B^{-}(c,R) \cup B^{+}(c,R) = Z^{n}.$$

Minimum cardinalities of asymmetric and unidirectional covering codes with length n and covering radius R are denoted with $D(n,R)$ and $E(n,R)$, respectively. The concepts of $B(x,R)$, $B^{+}(x,R) \cup B^{-}(x,R)$ and $B^{-}(x,R)$ are visualized in Figure 2.1 for Z^3, $R = 2$ and $x = 110$. Note that the definitions of covering codes and unidirectional covering codes coincide for $R = 1$.

In [**I**–**IV**] variants of covering codes (asymmetric and unidirectional covering codes) are considered along with multiple coverings. Asymmetric covering codes were introduced in [21] and motivated by a data compression application [27]. Afterwards asymmetric covering codes have been studied in [3, 26, 29, 50, 80] and in [**I**, **III**].

The concept of unidirectional covering codes is introduced in [**II**], inspired among other things by studies on unidirectional error-correcting codes. Covering codes and error-correcting codes are dual objects: with a covering code each word is covered by at least one codeword, whereas with an error-correcting code each word is covered by at most one codeword. For certain length n and covering radius R there are codes that are simultaneously covering codes and error-correcting codes—these codes are called perfect codes [55]. For the fundamentals of error-correcting codes the reader is referred to [58].

$n\backslash R$	1	2	3	4	5
1	1				
2	2	1			
3	3	2	1		
4	6	3	2	1	
5	10	5	3	2	1
6	18	8	4	3	2
7	31	14*	7*	4	3
8	58	22*–23	12*	6	4
9	101*–106	35*–40*	18*–19	10*	6
10	179*–196	57*–70	27*–31*	14–15*	8*
11	321*–352	93*–120$^+$	41*–49$^+$	20*–25	12*–13*
12	585*–668	156*–215$^+$	63*–84$^+$	29–40$^+$	17*–21
13	1079*–1253	266*–414$^+$	98*–146$^+$	43–65$^+$	24–33$^+$

$n\backslash R$	6	7	8	9	10
6	1				
7	2	1			
8	3	2	1		
9	4	3	2	1	
10	5	4	3	2	1
11	8*	5	4	3	2
12	11–12*	7	5	4	3
13	15–18*	10*–11	7	5	4

Table 2.1 Bounds on $D(n, R)$ for $n \leq 13$, $R \leq 10$

2.3 Summary of Results

Here we summarize the best known bounds on the minimum size of asymmetric covering codes (Table 2.1), unidirectional covering codes (Table 2.2) and multiple coverings (Table 2.3) for the parameters considered in [**I**–**IV**]. An asterisk in Tables 2.1 and 2.3 is used to denote a bound obtained in [**I**, **III**, **IV**]. A plus sign in Table 2.1 denotes a bound obtained in [80]. In Table 2.2 all the values for $R \geq 2$ are from [**II**].

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

Table 2.2 Bounds on $E(n, R)$ for $n \leq 13$, $R \leq 6$

	$K(n,1,\mu)$		
$n \setminus \mu$	2	3	4
6	20*	30–32	38–40
7	32	48	64
8	58–64	90–94	115*–124
9	104–112	154–160	206–216
10	188*–216	289–316	374–408
11	342–368	512	684–704
12	632*–704	972–1024	1262–1344
13	1172–1280	1758*–1920	2342–2528
14	2187*–2560	3356–3712	4372*–4864
15	4096	6144	8192
16	7713*–8192	11809–12288	15423*–16384

	$K(n,2,\mu)$		
$n \setminus \mu$	2	3	4
6	8*	11*	14*
7	12*	15*–16	19*–20
8	16*–19	22–24	29*–32
9	24–32	35*–44	46–56
10	40–48	56–64	74–88
11	63*–64	93*–100	124*–128
12	108–128	156–192	212–256
13	190–240	269*–336	360–448
14	311*–448	465*–640	620*–768
15	557–768	814–1024	1105–1280
16	1008–1536	1437*–2048	1932–2560

	$K(n,3,\mu)$			$K(n,4,\mu)$		
$n \setminus \mu$	2	3	4	2	3	4
8	8*	11*	12-14	4	6	8
9	10*–12	13*–16	17*–20	4	6	8
10	13–18	19*–24	25*–30	8*	10*–11	12*–14
11	19*–24	28*–36	37*–48	9*–12	12–16	16*–20
12	29–48	43*–60	56–76	12*–18	17*–24	22–30
13	45*–64	67–96	88–112	17*–26	24*–36	32*–48
14	74–120	108–160	141*–192	24*–48	35*–60	46–72
15	116*–160	173*–224	229*–256	36–64	52–96	69*–112
16	197–304	286–448	378*–512	54–112	80–128	106*–188

Table 2.3 Bounds on $K(n,R,\mu)$ for $6 \leq n \leq 13$, $2 \leq \mu \leq 4$, $R \leq 4$

Chapter 3

Computer Search

In coding theory and in discrete mathematics there are numerous open problems, for instance, determining the values of the functions $D(n, R)$, $E(n, R)$ and $K(n, \mu, R)$ described in Chapter 2. Small instances can often be treated exactly by analytical means (with computational methods extending the scope of exact results). But for larger instances one must usually settle for finding lower and upper bounds on the values of such functions.

Traditionally speaking mathematical and analytical results are preferred over computational results, as analytical results can be reapplied more readily to larger instances (or even in different contexts than the original results) whereas the validity of obtained computational results can be hard to verify.

The increase of computational power and the progress in combinatorial optimization [1, 20, 46, 49, 54, 75] have made the computational approach more appealing, especially in practical applications. However, ever-increasing computational power alone does not solve all open problems—as the theory of computation and complexity theory has shown—so there will always be room for novel approaches and ingenuity.

Many problems of combinatorial nature can be stated with a set of linear equations and solved efficiently with any suitable linear programming package[1]. Integer programming [75, 85] steps into the picture when the variables in the equations are constrained to be integers (as the nature of discrete problems usually requires). Unfortunately no efficient algorithms are known for solving integer programming formulations—in practice solv-

[1]Like *glpk* `http://www.gnu.org/software/glpk/` or CPLEX `http://www.ibm.com/software/integration/optimization/cplex-optimizer/`.

ing is done with an exhaustive search (most commonly using a branch-and-bound [52] approach), hence integer programming is in between analytical and computational approaches. Results for various types of covering codes with (integer) linear programming can, for instance, be found in [3, 18, 21, 34, 37, 53, 57, 69, 91, 92]. One should note that a custom exhaustive search approach (even an unoptimized one) tends to be much more efficient than solving a corresponding integer programming formulation with a general solver, as often the problem-specific characteristics aiding a custom exhaustive search cannot be incorporated into the integer programming formulations.

In this chapter we discuss computer search as a way to determine lower and upper bounds on the values of functions $D(n, R)$, $E(n, R)$ and $K(n, R, \mu)$. Exhaustive search as a means of obtaining lower bounds will be discussed in Section 3.1. In Section 3.2 heuristic search is treated as a way to construct explicit codes leading to upper bounds.

Some of the source codes of the computer programs that were used to obtain the results in [**I**–**VI**] are available online, see Appendix B.

3.1 Exhaustive Search

In exhaustive search [49, 79] all possible alternatives are checked and hence an indisputable conclusion on the problem at hand can be reached. The naive approach would be to list all possible solutions and then iterate through the list, but such an approach quickly becomes unfeasible. As an example, for verifying $D(n, R) > A$ one could check that none of all the $\binom{2^n}{A}$ possible codes is an asymmetric R-covering code.

A more practical way of doing exhaustive search is to construct solutions in several steps and in each step prune away those partial solutions that can be proven not to lead to a solution (i.e., all solutions are still processed, but a larger group of solutions are checked at the same time). A general flow of an exhaustive search algorithm is shown in Algorithm 3.1. For computational efficiency good methods for pruning as well as constructing partial solutions (ideally any partial solution would be generated only once) are desired.

One way of reducing the *search space* (the entire set of solutions) is to take symmetries of the search space into account, i.e., consider equivalence of solutions under some notion of equivalence [43]: if there are two or more equivalent solutions, then it is not necessary to go through all of those

```
$\mathcal{P}_{\text{current}} \leftarrow$ setContainingTheStartingSolution
**for** $i = 1$ **to** lastConstructionStep **do**
**begin**
    $\mathcal{P}_{\text{next}} \leftarrow \emptyset$
    **for** $C \in \mathcal{P}_{\text{current}}$ **do**
    **begin**
        **for** $j \in \text{constructionPossibilities}_i(C)$ **do**
        **begin**
            $C' \leftarrow \text{construct}_i(C, j)$
            **if** $\text{canNotPrune}_i(C', \mathcal{P}_{\text{next}})$ **then**
                $\mathcal{P}_{\text{next}} \leftarrow \mathcal{P}_{\text{next}} \bigcup C'$
        **end**
    **end**
    $\mathcal{P}_{\text{current}} \leftarrow \mathcal{P}_{\text{next}}$
**end**
**if** $(\mathcal{P}_{\text{current}} = \emptyset)$ **then**
    **return** “No solution“
**else**
    **return** $\mathcal{P}_{\text{current}}$
```

Algorithm 3.1: The general flow in exhaustive search

solutions—checking just one is sufficient. The real issue is how to determine whether two solutions are equivalent—and how to do it computationally efficiently. An alternative for comparing (partial) solutions is to ensure that no equivalent solutions are constructed in the first place [43, 59, 60, 84].

Exhaustive search has been used with various types of covering codes, for instance, in [3, 7, 8, 69, 70, 72, 74, 92].

3.1.1 Our Implementation

Here we will give an overview (omitting the implementation details) of the exhaustive search algorithms in [**II**–**IV**]. Before proceeding, we point out that we define two codes C_1 and C_2 to be equivalent if and only if $|C_1| = |C_2|$ and there exists a permutation of coordinates σ so that for all $x \in C_1$ we have $\sigma(x) \in C_2$. For general covering codes, which are not considered here, permutations of the symbols are also allowed [43, Definition 2.100]. Our more concise definition follows from the need to preserve weights. We will also use term *coverage* of a code C to mean a $(n+1)$-tuple $(v_0, v_1, \ldots, v_n)$

in which v_w is the number of words with weight w covered by C.

Exhaustive search is used in two ways: (a) for constructing (upper) bounds on coverages of codes with given weight distribution, and (b) and for enumerating all codes with given parameters—or more likely showing the non-existence of such covering codes and therefore obtaining a lower bound on the corresponding instance of $D(n, R)$, $E(n, R)$ or $K(n, \mu, R)$.

In case (a) each construction step consists of adding one codeword into a partial code while pruning away equivalent codes. The actual equivalence checking (i.e., pruning by the symmetries) between two (partial) codes is done by constructing graphs of the codes and using `nauty` [59] to inspect whether the graphs are isomorphic or not. After each construction step the results with the coverages are stored.

In case (b) each construction step again consists of adding one codeword into a partial code, but now pruning is done by both code equivalence and insufficient coverage—in other words, a partial code C is rejected if its coverage together with an upper bound on the coverage of remaining unfixed codewords obtained in (a) show that the covering criterion cannot be fulfilled.

3.2 Heuristic Search

In heuristic search the aim is to find a sufficiently good solution for the instance at hand. The reason for settling for a good enough solution is the simple fact that solving the instance is not possible or that solving would require too much resources. Fortunately, good solutions can often be constructed by exploring only a fraction of the search space. An amount of intuition, academic guesses, rules of thumb and plain common sense are incorporated into a heuristic search (as the word "heuristic" suggests) in order to make it explore the part of the search space in which good solutions are expected to reside.

3.2.1 Overview

We shall begin by describing some central concepts. An *objective function* $z(C)$ tells how good a solution C is—for instance, the length of a route, the cost of a project or the "distance" to a valid solution (like the number of uncovered words with covering codes). A *fitness function* $f(C)$ describes the goodness of a solution C from the viewpoint of the search method. Often the objective function and the fitness function are the one and the same (like in this thesis), but some heuristic search methods—guided local search [87], for instance—modify the fitness function during the search. We will assume (without loss of generality) that the objective and fitness functions are always minimized. A *neighborhood* $\mathcal{N}(C)$ of a solution C is the set of all the solutions that are in some sense near C. The nearness implies that there is an efficient way to obtain $C' \in \mathcal{N}(C)$ from C. It is desired that a global optimum can be reached using the neighborhood (iteratively) while the neighborhood is a compact one (so that it can be iterated through efficiently).

Heuristic search methods can be divided into two main branches: local search methods and evolutionary algorithms [1, 22, 35, 49, 76]. In a *local search method* the current solution is iteratively modified until a sufficiently good solution has been found or the search has proceeded long enough without success. In Algorithm 3.2 the general flow in a local search method is described.

```
 1 : $C_{\texttt{next}} \leftarrow$ initialSolution
 2 : $C_{\texttt{best}} \leftarrow C_{\texttt{next}}$
 3 : **while** $z(C_{\texttt{best}}) >$ desiredValue **and** continueSearching **do**
 4 : **begin**
 5 :     $C \leftarrow C_{\texttt{next}}$
 6 :     **while** $C' \leftarrow$ pickASolutionFrom($\mathcal{N}(C)$) **do**
 7 :     **begin**
 8 :         **if** $z(C') < z(C_{\texttt{best}})$ **then**
 9 :             $C_{\texttt{best}} \leftarrow C'$
10 :         **if** acceptableSolution($C', C, C_{\texttt{next}}$) **then**
11 :         **begin**
12 :             $C_{\texttt{next}} \leftarrow C'$
13 :             breakFromLoopIfDesirable
14 :         **end**
15 :     **end**
16 : **end**
```

Algorithm 3.2: The general flow in a local search

One should take notice that a search method finds a *local optimum* (all the solutions nearby are worse than the solution in question). Hopefully the local optimum found is also a global optimum (or at least nearly as good). A wide variety of search methods arise from different approaches of preventing a search method from getting stuck into a local optimum. Generally speaking only very little can be said about how close the obtained solutions are to a global optimum. See [40, 83] for an approximation point of a view on the subject.

An outline of an *evolutionary algorithm* is described in Algorithm 3.3. The main idea is to have a current population $\mathcal{P}_{\texttt{current}}$ of solutions instead of a single solution. The algorithm proceeds by constructing a new population $\mathcal{P}_{\texttt{next}}$ from $\mathcal{P}_{\texttt{current}}$ with the hope that solutions in $\mathcal{P}_{\texttt{next}}$ will be better than in $\mathcal{P}_{\texttt{current}}$ on average. The algorithm terminates with a similar criteria as in a local search method.

```
1 :  $\mathcal{P}_{\texttt{next}} \leftarrow$ initialPopulationOfSolutions
2 :  $C_{\texttt{best}} \leftarrow$ bestSolution($\mathcal{P}_{\texttt{next}}$)
3 :  while $z(C_{\texttt{best}}) >$ desiredValue and continueSearching do
4 :  begin
5 :      $\mathcal{P}_{\texttt{current}} \leftarrow \mathcal{P}_{\texttt{next}}$
6 :      $\mathcal{P}_{\texttt{next}} \leftarrow$constructNextPopulation($\mathcal{P}_{\texttt{current}}$)
7 :      $C' \leftarrow$ bestSolution($\mathcal{P}_{\texttt{next}}$)
8 :      if $z(C') < z(C_{\texttt{best}})$ then
9 :          $C_{\texttt{best}} \leftarrow C'$
10:  end
```

Algorithm 3.3: The general flow in evolutionary algorithms

For designing a good search method the concepts of intensification and diversification are crucial. *Intensification* means that the part of the search space being explored should be examined thoroughly enough (so that the best solution in the corresponding area is found). The definition of neighborhood plays a vital role in intensification. *Diversification* means that different areas of the search space should be examined. A common approach to address diversification is to run the search method several times, for instance, using randomness—or a more determined way, like greedy randomized adaptive search procedures (GRASP) [32]—in the construction of the initial solutions.

As a final note, by No Free Lunch theorems [90] one should not expect any search method to dominate over other search methods in all problems. Of course, usually one is not trying to get a search method to work extremely well on a wide variety of problems—getting the search method to work well on the problem at hand suffices.

3.2.2 Examples of Heuristic Search Methods

We will now give some examples of established search methods. A *hill-climbing* (or *greedy* local search) method searches the entire neighborhood, picks the best solution in it and never accepts a solution which is worse than the current one. The search can be performed quickly, but usually the obtained solution is not very good (as the search terminates when the first local optimum is encountered). The reader is referred to [20] for a discussion on when a local and global optimum are the one and the same. However, hill-climbing can, for instance, be used in evolutionary algorithms to ensure that the solutions in the population are locally optimal.

Simulated annealing [45, 51] is based on an analogy with a method of cooling metal (known as "annealing"). The approach picks the candidate solution from the neighborhood randomly and accepts it as the next solution if: (a) it is better than the current one or (b) it is worse than the current, with a probability

$$e^{(f(C)-f(C'))/T}, \text{ where } T \text{ is a cooling constant, } \textit{temperature}.$$

The temperature is gradually lowered as the search proceeds. Effectively in the beginning of the search worse solutions are accepted readily, while in the end only improvements are approved. Simulated annealing has been proved to converge into a global optimum in many optimization problems, although in practice the conditions required for such a convergence can be met in a very limited number of problems [1]. Examples of using simulated annealing with codes can be found in [25, 37, 65, 66, 89].

Genetic algorithms [41, 77] are evolutionary algorithms motivated by biology. Two types of operations are defined for solutions: a *mutation* changes a given solution in some way and a *crossover* operation combines two or more solutions into one (or more) solutions. The next population is generated by using different mutation and crossover operations on the solutions of the current population. The probability of a solution C being involved in operations depends on $f(C)$ with respect to the other solutions—i.e., the better the solution, the higher the probability. In other words, the better solutions pass their properties to the next generation more often and by combining several good solutions even better ones are (hopefully) produced. Genetic algorithms have been used to construct codes, for example, in [28, 86].

3.2.3 Tabu Search

Tabu search [30, 34, 36] is another well known local search method. The general flow in tabu search is described in Algorithm 3.4. In tabu search the entire neighborhood is searched and the best solution found is picked as the next solution (even if it is worse than the current one). The distinguishing feature of tabu search is its mechanism for escaping local minima: the search maintains a so-called *tabu list* of recently made moves (or visited solutions) which will be rejected in the future. The tabu list has limited size, so moves (or solutions) which were banned once will be allowed again later. By an *aspiration criteria* it is possible to accept rejected solutions—for instance, if such a solution would be better than $C_{\mathtt{best}}$. In [31] a probabilistic version of tabu search is proposed, which offers some explanation for the observed good performance. Results with tabu search with various types of covering codes can be found, for instance, in [12, 24, 61, 67, 68, 71, 73].

```
$C_{\mathtt{next}} \leftarrow$ initialSolution
$C_{\mathtt{best}} \leftarrow C_{\mathtt{next}}$
**while** $z(C_{\mathtt{best}}) >$ desiredValue **and** continueSearching **do**
**begin**
    $C \leftarrow C_{\mathtt{next}}$
    $C_{\mathtt{next}} \leftarrow \emptyset$
    **for** $C' \in \mathcal{N}(C)$ **do**
        **if** isNotTabu($C'$) **or** aspirationCriteriaApplies($C, C'$) **then**
        **begin**
            **if** $C_{\mathrm{next}} = \emptyset$ **or** $f(C') < f(C_{\mathrm{next}})$ **then**
                $C_{\mathtt{next}} \leftarrow C'$
        **end**
    **if** $z(C_{\mathrm{next}}) < z(C_{\mathrm{best}})$ **then**
        $C_{\mathtt{best}} \leftarrow C_{\mathrm{next}}$
    addToTabuList($C, C_{\mathrm{next}}$)
**end**
```

Algorithm 3.4: A general flow in tabu search

As we use tabu search in [**I**, **II**] to construct explicit codes for obtaining upper bounds on $D(n, R)$ and $E(n, R)$, we will now describe features of our implementations (omitting details, as they can be found in the publications). The cardinality of the solution (*code*) is fixed in the beginning and the current code is modified by changing one codeword at a time. As the fitness function we use the number of uncovered words (ties are broken randomly). The initial code is constructed randomly or by removing a random codeword from a covering code (possibly changing some codewords randomly a bit). The neighborhood is constructed in two ways: by discovering the (lexicographically) next uncovered word and covering it (as in [68]), and by covering an uncovered word by a minimal change in the current code. Two different approaches are also used for the tabu list: one containing the recent changes in codewords (undoing or redoing a change is not allowed), and the other containing recently changed codewords (modifying recently changed codeword is not allowed). As an aspiration criterion we either accept a code if it is a covering code or if it is a best code found so far.

Chapter 4

Games

Combinatorial game theory [5, 19] concentrates on games with perfect information (a player has knowledge of all the possible moves of all players at any given move)—including two-player zero sum games (the sum of players' scores is zero), like chess or go. Analyzing full games (of all but the simplest) with combinatorial game theory is a too daunting task, but in restricted game situations the theory has been applied with success. For instance, the late end game of go can be considered to be solved in [6]. Combinatorial game theory is a part of the larger field of game theory.

In Section 4.1 we will briefly go over the fundamental algorithms of game playing and take notice of their connection to complexity of games. We will extend the discussion to entropy in Section 4.2. The chapter is concluded with an overview of computer go in Section 4.3.

4.1 Minimax Algorithm and $\alpha\beta$ Pruning

Let us consider a perfect information game in which two players—`Max` (having the first move) and `Min`—make their moves in turns. Let score f indicate the result of the game with a higher value for better result for `Max`. Similarly, a smaller value of f is better for `Min`. In other words, `Max` aims to maximize f on her turn while `Min` aims to minimize f on her turn.

In such games (as well as in many others) it is possible to arrange all possible games in the form of a tree, in which nodes represent the options available to players at their turns and leaves have the result of the corresponding game (i.e., the value of f). A simple game of `Max` and `Min` having a single move each is depicted in Figure 4.1.

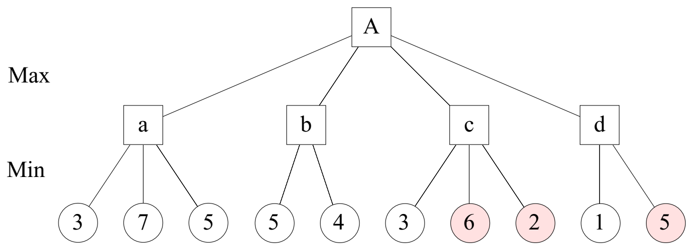

Figure 4.1 A simple game

The minimax algorithm [64] solves such a game by completing a full depth-first search in the tree: i.e., `a` $= \text{Min}(3, 7, 5) = 3$, `b` $= \text{Min}(5, 4) = 4, \ldots,$ and finally `A` $= \text{Max}($`a`$,$ `b`$,$ `c`$,$ `d`$) = \text{Max}(3, 4, 2, 1) = 4$. The minimax algorithm becomes quickly intractable for more interesting games, so instead of going through the entire search tree the tree is cut at a specified depth and the values of the new leaves are calculated with an *evaluation function* $\hat{f}$ of the score f.

The concept of $\alpha\beta$ pruning [47] provides a mechanism for pruning nodes in the search tree. The main idea is to have additional α and β values at each node, representing the best values of f for `Max` and `Min`, respectively. If `Max` encounters a higher value than β when processing the children of the corresponding node, then the remaining unprocessed children can be skipped. The same applies for `Min` with lower values than α.

For an example of $\alpha\beta$ pruning we shall go back to Figure 4.1 (note that nodes and leaves are processed in the order from left to right):

- for solving node `a` the search still needs to go through all the leaves of the node, which ensures that `A` $= \alpha \geq 3$;
- for node `b` both leaves are processed, as they both are larger than the current α—after which `A` $= \alpha \geq 4$;
- with node `c` the first encountered leaf has a smaller value than the current α, hence the other leaves can be pruned (`Min` would not choose a leaf with a higher value than the encountered 3 and `Max` would not choose a node with a worse outcome than with node `b`);
- and similarly with node `d` the latter leaf can be pruned;

- so finally $\texttt{A} = \text{Max}(3, 4, \texttt{c}, \texttt{d}) = 4$, in which $\texttt{c} \leq 3$ and $\texttt{d} \leq 1$. I.e., the three colored leaves in Figure 4.1 could be pruned.

The number of pruned nodes with $\alpha\beta$ pruning depends heavily on the *move ordering*, i.e., the order in which nodes and leaves are processed. An optimal move ordering would halve the search tree (but if such a move ordering would exist, then there would be no need to do the search in the first place) [79]. The complexity of $\alpha\beta$ pruning is considered in [47].

Comparison of different games and discussion on their hardness (for humans and for computers) are conducted in [2, 9, 38, 39] using (among other criteria) a metric called game-tree complexity. The *game-tree complexity* of a game is the number of leaves in the search tree necessary to solve the game. In practice the search tree refers to the search tree of the minimax algorithm—using the search tree with $\alpha\beta$ pruning might be more descriptive of the complexity of the game, but estimating the number of leaves in the pruned tree is harder. Obviously calculating game-tree complexity is possible only for the very smallest games, so usually an estimate s^n is used, in which s is a branching factor (often the average number of legal moves) and n is the average length of the game.

4.2 Entropy

Entropy [81] is a central concept in information theory. The concept of entropy can be used to determine the maximum amount of information which can be reliably transmitted over of a noisy channel. The *entropy* of a discrete random variable X with the probability mass function $p(x)$ is defined as

$$H(X) = -\sum_x p(x) \log_b p(x),$$

where the basis b of the logarithm is determined by the context. The base e is used here.

In [**VI**] the idea of using entropy (motivated by [82]) for measuring complexity (and performance) in go is presented. A *predictor* $\mathcal{P}$ turns a game record g into a sequence of numbers p, in which the ith number p_i is the amount of guesses the predictor needed to guess correctly the ith move g_i in the game record g. With a set of game records $\mathcal{G}$, given predictor $\mathcal{P}$ and frequency

$$q_{\mathcal{P}}^{\mathcal{G}}(j) = \frac{\sum_{p \in \mathcal{P}(\mathcal{G})} \sum_{p_i = j} 1}{\sum_{g \in \mathcal{G}} \sum_i 1},$$

the entropy of a predictor $\mathcal{P}$ for game record set $\mathcal{G}$ is

$$H_{\mathcal{P}}^{\mathcal{G}} = -n \sum_j q_{\mathcal{P}}^{\mathcal{G}}(j) \ln q_{\mathcal{P}}^{\mathcal{G}}(j).$$

A random predictor—which picks a move randomly from the set of legal moves—naturally maximizes the entropy, whereas a predictor with a better understanding of the game (or to be exact, how the players play the game) achieves a considerably smaller entropy. Ultimately, if a game has one outcome (for rational players)—like the one in Figure 4.1—then the entropy is zero.

By using n as the average length of the game and s as the average number of legal moves, the entropy of a game with a random predictor is

$$nH_{\texttt{Random}} = -n \sum_s \frac{1}{s} \ln \frac{1}{s} = -n \ln \frac{1}{s} = \ln(s^n),$$

which provides a connection to the game-tree complexity. As the game-tree complexity measures the complexity of randomly played games, entropy can be used to measure the complexity of more rationally (we shall omit discussion on "rationality" here) played games.

As a final note, entropy has been used in other contexts in games. For instance, in [4, 15] entropy is used to measure distances between distributions.

4.3 Computer Go

The game of go is the last classical board game in which humans are still superior to computers. Many textbooks (including [79]) on artificial intelligence mention go as the next big challenge for artificial intelligence and state further that successful techniques for go should prove widely useful in other applications as well. A rule set for go is provided in Appendix A with some background information on go terminology, conventions and customs.

The history of computer go begins in the 1960s [78, 93]. Like in many other games, computer players have been implemented essentially by designing a high quality evaluation function $\hat{f}$ and then doing a tree search as deep as possible in the style of $\alpha\beta$ pruning. In order for the approach to be successful (for instance, like in chess [42]), a good evaluation function as well as a good move ordering are needed. However, in go this approach has not resulted in a breakthrough, probably because it is notoriously hard to design a good evaluation function for go. A brief survey of the field of computer go can be found in [**V**], whereas more detailed surveys on the applied methods can be found in [9, 10, 16, 63, 88].

A more recent approach for computer go emerged around 2006 and is called Monte-Carlo tree search (MCTS) [13]. The main innovation in MCTS is to run a large number of (pseudo) random games from the current situation and use the results of those random games as an evaluation function. The idea of using simulations as a replacement of an evaluation function is presented in [11] for go, but for the idea to really work a sufficient amount of random games needs to be played in the more interesting branches of the search tree. By "interesting" we mean the nodes corresponding to good moves, as bad moves should be recognized as quickly as possible so that random games can be played in the part of the search tree where they matter the most. Different aspects and developments leading to Monte-Carlo tree search have been studied for instance in [14, 23, 33, 48].

Results with Monte-Carlo tree search have been impressive[1]: for instance in 2007 on a 9×9 board a victory in an even game and in 2009 on a 19×19 board a victory in a 6 stone handicap game were achieved against a professional level player [13]. It should be noted that all the current state-of-the-art computer go players use MCTS in one form or another.

[1] See `http://www.computer-go.info/h-c/`, Human-Computer Go Challenges, N. Wedd.

Chapter 5

Conclusions

As the computational power available has increased, using computational methods have become more appealing—for instance in classification and enumeration of combinatorial objects. The enumeration of objects depends on efficient exhaustive search, so research on efficient generation and pruning techniques is more topical than ever. It is also possible to design an exhaustive search in a depth-first manner so that when the search is adjourned, one can get the best solution found so far as well as bounds for an optimal solution. This possibility makes (partial) exhaustive search also a viable option for obtaining a single solution to the problem instance at hand.

We were able to show with exhaustive search the non-existence of codes with given parameters on several occasions. We assume that some of the techniques used in pruning could be of general use. For further research, concentrating on the order in which the search tree is explored seems like a worthy cause (results obtained from exploring a more promising branch could be used to prune some of the less encouraging branches). Naturally, the idea is not a new one—for example, in the context of game study the idea is known as move ordering (or ranking).

Games in general have connections to many fields, especially if one is speaking of game theory. We introduced the information-theoretic concept of entropy to the game context as a way to measure complexity of games and presented empirical results (computational experiments and a questionary study) to support the relevancy of entropy. Further studies on entropy in other games than go would be interesting, in order to see how the entropy would compare with other complexity metrics for games. Also research on go (being the last traditional game in which humans are superior to computers) would be interesting, especially with respect to Monte-Carlo tree search.

Heuristic search methods are powerful tools for obtaining solutions in problems that seem to be too hard to be tackled by other means. In this thesis we used tabu search to construct several record breaking codes. Without a doubt even better codes than the ones presented in this thesis can in some cases be constructed by simply devoting more computing time for the search. For further research, studying different search heuristics or concentrating efforts on alternative fitness functions could prove fruitful (as this might aid with exhaustive search and branch ordering).

Appendix A

Rules of Go

The Tromp-Taylor[1] rules of go are formulated to be as elegant as possible. We will copy the 10 point rule set here directly, along with some clarifications and background information with terminology of a go player.

1. *Go is played on a* 19×19 *square grid of points, by two players called Black and White.*

   Besides the official 19×19 board, the 13×13 and 9×9 boards are commonly used—especially with beginners. These boards are displayed in Figure A.1.

2. *Each point on the grid may be colored black, white or empty.*

   An intersection can be empty, have a white or have a black stone.

3. *A point P, not colored C, is said to reach C, if there is a path of (vertically or horizontally) adjacent points of P's color from P to a point of color C.*

   Vertically or horizontally adjacent stones of the same color belong into the same *group*. A group has as many *liberties* as it has empty intersections vertically or horizontally adjacent to any stones in the group. In the left diagram in Figure A.2 there are two black groups b and d with 3 and 4 liberties, respectively. Similarly there are three white groups a, c and e with 3, 2 and 4 liberties.

4. *Clearing a color is the process of emptying all points of that color that don't reach empty.*

[1]See `http://homepages.cwi.nl/~tromp/go.html`, J. Tromp and B. Taylor.

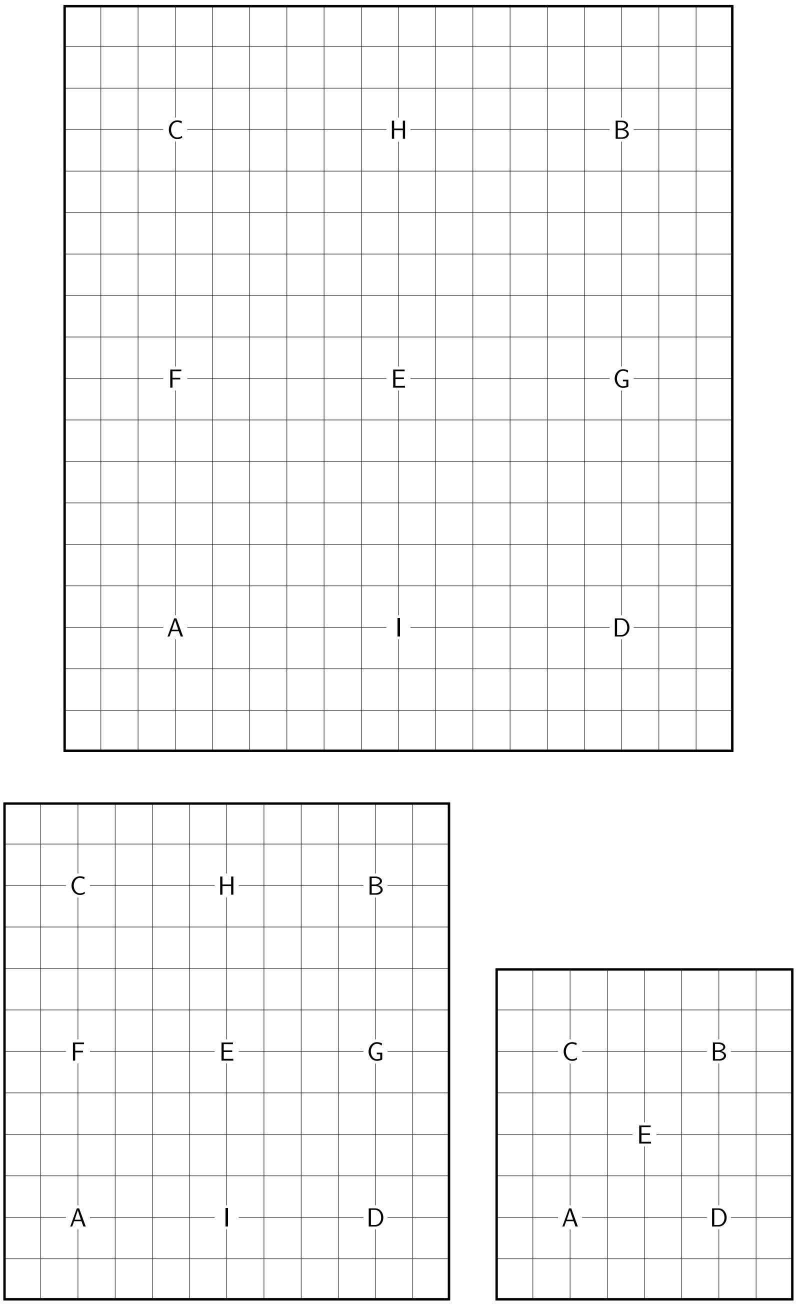

Figure A.1 The most common board sizes for go (19x19, 13x13 and 9x9) and the customary placement of handicap stones (in alphabetical order).

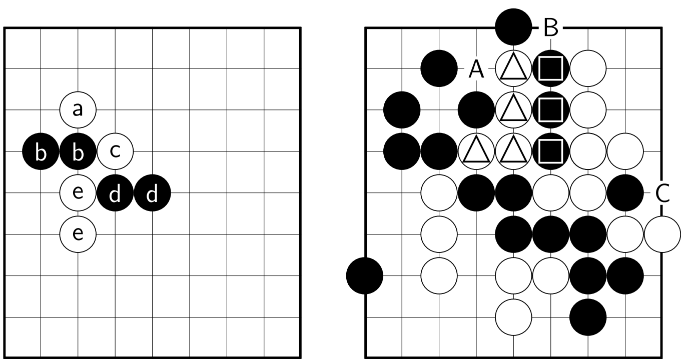

Figure A.2 Examples of liberties and capturing

The removal of stones is referred to as *capturing*, e.g., if a player's move removes the last liberty of some groups, those groups are removed from the board. In the right diagram in Figure A.2 black can capture four △ stones by playing A and white can capture three ■ stones with B. The only other option of capturing stones in this diagram is white playing at C and capturing the single adjacent black stone.

5. *Starting with an empty grid, the players alternate turns, starting with Black.*

   For players with different strength it is common to use handicap stones to make the game more interesting (e.g., both players have equal chances of winning). In other words, the weaker player places as her first move as many handicap stones on the board as is the difference between the players ranks. These stones may be placed freely or (more commonly) in the customary way in the order listed in Figure A.1, except that in 6 and 8 stone handicap games no stone is placed on E. A proper handicap (roughly speaking) for 13×13 is obtained by dividing the rank difference with three and for 9×9 with nine.

   Ranks start with 30 kyu and proceed up to 1 kyu. After kyu ranks dan ranks are used in increasing order from 1 dan to 7 dan. Besides these amateur ranks there are professional dan ranks from 1 dan to 9 dan (with the difference that one handicap stone is considered to equal three ranks). An amateur 7 dan is usually regarded to be equal with a professional 1 dan player. Ranks are honorary titles in the

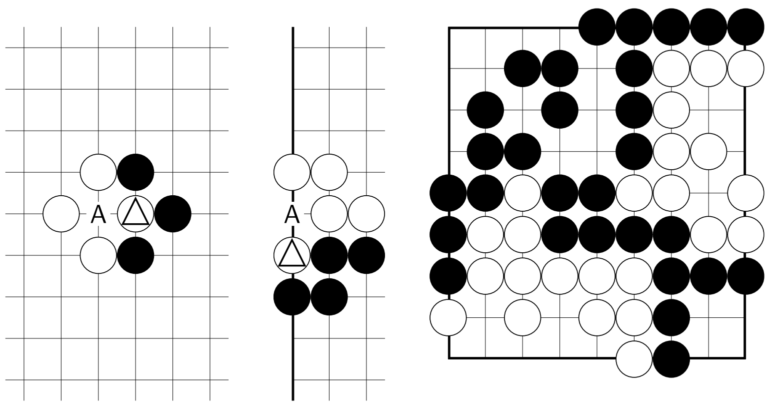

Figure A.3 Examples of ko and scoring.

sense that they are not reduced, hence reflecting more of a player's peak playing performance than their current strength.

6. *A turn is either a pass; or a move that doesn't repeat an earlier grid coloring.*

   The left and center diagrams in Figure A.3 demonstrate the reason of not allowing repetition: Black can capture △ by playing A, but then white could capture black stone at A by playing at △, and so on. In other words, white must move at least once somewhere else. These kinds of repeating situations are called *ko (fights)*, and the moves played elsewhere (in order to break the repetition and win the fight) are called *ko threats*.

   In case the repeating sequence of moves is longer than two, the situation is referred to as *superko*. Superkos are extremely rare in human games, but are somewhat more relevant in computer go as superkos do occur every now and then in the thousands simulations run in the Monte-Carlo tree search approach [13].

7. *A move consists of coloring an empty point one's own color; then clearing the opponent color, and then clearing one's own color.*

   A player either passes or places a stone of her color on board, removes all opponent's groups without liberties and then removes all her groups which still have no liberties.

8. *The game ends after two consecutive passes.*

   All groups that can be captured even if their owner defends them are removed from the board. If the players do not agree about the status of some of the groups, then the play is resumed.

9. *A player's score is the number of points of her color, plus the number of empty points that reach only her color.*

   The score of a player is the sum of her stones on the board and the number of empty intersections surrounded by them. In the right diagram in Figure A.3 white has 24 stones on the board, which enclose 11 empty intersections. Black has 29 stones on the board enclosing 17 intersections. That is, white's score is 35 and black's 46.

   This type of "Stones on board + surrounded intersections" scoring is referred to as *area scoring*. The other general type of scoring is *territory scoring*, in which one's score is the sum of surrounded intersections and captured opponent stones. In practice the area scoring and territory scoring give equivalent scores almost always.

   The area scoring is more convenient in computer go, as the computers can play the game to the very end—i.e., all capturable stones are captured before passing—which makes the scoring trivial.

10. *The player with the higher score at the end of the game is the winner. Equal scores result in a tie.*

    Usually white gets compensation, *komi*, for black having the first move. In 19×19 komi is usually 7.5 points (the non-integer komi is used to avoid ties). For example, with 7.5 point komi black wins with 3.5 points in the right diagram in Figure A.3. In handicap games with players having the rank difference of one stone, the komi is usually 0.5 for the white player.

There are several rule sets[2] for go (Japanese, Chinese, Ing, to name a few) [10]. In practice the rule set used does not affect the game strategy and the effect on the final scoring is minimal.

[2]See `http://home.snafu.de/jasiek/rules.html`, rule discussions, R. Jasiek.

Appendix B

Source Codes

Some of the programs and scripts used in Publications [**I**–**VI**] are available at

`http://esaseuranen.fi/papers/dissertation/`.

Different files are briefly described (and some examples are provided) in README.TXT.

The source codes are provided as they are, with the notice that they were not originally intended to be released (the documentation is rather scarse for the most of the time).

In order to compile programs `gcc` is needed. In order to run the scripts, `perl` should be installed (as well as `sed`).

Some scripts for distributing calculations (when operating with a shared file system) to several computers can be found from the subdirectory `common`.

The exhaustive search algorithm (from [IV])

For compiling the program and using the scripts, `nauty` [59][1] and `glpk`[2] should be installed into the system.

All the related files are in the subdirectory `mcc`. There are scripts (`by*.sh`) for directly obtaining the same results as in [**IV**].

[1]See `http://cs.anu.edu.au/~bdm/nauty/`, The nauty page, B. D. McKay.
[2]See `http://www.gnu.org/software/glpk/`, GNU Linear Programming Kit.

Different versions of the exhaustive search algorithm were used in [**II**–**III**], but it should be possible (by suitable alterations to the scripts) to recreate the results in [**II**–**III**] with respect to the lower bounds.

The tabu search algorithm (from [80])

All the related files are in the subdirectory `tabu`.

Different versions of the tabu search algorithm were used in [**I**–**II**]. For instance, the implementation does not include different definitions for neighborhood (i.e., the one used in [**I**]). The search algorithm is capable of constructing various covering codes, hence it should be possible to construct codes similar to those in [**I**, **II**] (and probably even better ones, given enough time and effort).

Predicting moves in a go game (from [VI])

All the related files are in the subdirectory `go`.

The source codes include functionality for selecting, parsing and cleaning SGF[3] files. The scripts provide a way to query predictions from `gnugo`[4] or from `mogo` [33][5] for the next move in a given game record.

Also the sets of the 19x19 game records used in [**VI**] are included.

[3]Smart Game Format is used for storing records for various board games.
[4]See `http://www.gnu.org/s/gnugo/`, GNU Go
[5]See `http://www.lri.fr/~teytaud/mogo.html`, MoGo.

References

[1] E. Aarts, J. K. Lenstra (eds.), Local Search in Combinatorial Optimization, Wiley, New York, NY, 1997.

[2] L. V. Allis, Searching for solutions in games and artificial intelligence, Ph.D. thesis, Vrije Universiteit, Amsterdam, The Netherlands (1994).

[3] D. Applegate, E. M. Rains, N. J. A. Sloane, On asymmetric coverings and covering numbers, J. Combin. Des. 11 (2003) 218–228.

[4] N. Araki, K. Yoshida, Y. Tsuruoka, J. Tsujii, Move prediction in go with maximum entropy method, in: Proceedings of the 2007 IEEE Symposium on Computer Intelligence and Games, 2007, pp. 189–195.

[5] E. R. Berlekamp, J. H. Conway, R. K. Guy, Winning Ways for Your Mathematical Plays, vol. 1–4, 2nd ed., A. K. Peters, Natick, MA, 2001–2004.

[6] E. R. Berlekamp, D. Wolfe, Mathematical Go: Chilling Gets the Last Point, A. K. Peters, Natick, MA, 1994.

[7] R. Bertolo, P. R. J. Östergård, W. D. Weakley, An updated table of binary/ternary mixed covering codes, J. Combin. Des. 12 (2004) 157–176.

[8] U. Blass, S. Litsyn, The smallest covering code of length 8 and radius 2 has 12 words, Ars Comb. 52 (1999) 309–318.

[9] B. Bouzy, T. Cazenave, Computer go: An AI-oriented survey, Artif. Intell. J. 132 (2001) 39–103.

[10] R. Bozulich, The Go Player's Almanac 2001, Kiseido Publishing Company, Tokyo, 2001.

[11] B. Brügmann, Monte Carlo go, Tech. Rep., Physics Department, Syracuse University, NY (1993).

[12] W. A. Carnielli, E. L. Monte Carmelo, M. V. S. Poggi de Aragão, C. C. de Souza, Upper bounds for minimum covering codes by tabu search, in: M. V. S. Poggi de Aragão, C. C. de Souza (eds.), The Proceedings of the II National Workshop on Combinatorial Problems, 1995, pp. 51–59.

[13] G. M. J.-B. Chaslot, Monte-Carlo tree search, Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands (2010).

[14] G. M. J.-B. Chaslot, J.-T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, J. van den Herik, Monte-Carlo strategies for computer go, in: P.-Y. Schobbens, W. Vanhoof, G. Schwanen (eds.), Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium, 2006, pp. 83–90.

[15] G. M. J.-B. Chaslot, M. H. M. Winands, I. S. van den Herik, Cross-entropy for Monte-Carlo tree search, ICGA J. 31 (2008) 145–156.

[16] K. Chen, Computer go: Knowledge, search, and move decision, ICGA J. 24 (2001) 203–215.

[17] G. Cohen, I. S. Honkala, S. Litsyn, A. C. Lobstein, Covering Codes, North-Holland, Amsterdam, 1997.

[18] G. Cohen, A. C. Lobstein, N. J. A. Sloane, Further results on the covering radius of codes, IEEE Trans. Inform. Theory 32 (1986) 680–694.

[19] J. H. Conway, On Numbers and Games, 2nd ed., A. K. Peters, Natick, MA, 2001.

[20] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, A. Schrijver, Combinatorial Optimization, Wiley, New York, NY, 1998.

[21] J. N. Cooper, R. B. Ellis, A. B. Kahng, Asymmetric binary covering codes, J. Combin. Theory Ser. A 100 (2002) 232–249.

[22] D. Corne, M. Dorigo, F. Glover (eds.), New Ideas in Optimization, McGraw-Hill, London, 1999.

[23] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29–31, 2006. Revised Papers, vol. 4630 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 72–83.

[24] R. Davies, G. F. Royle, Graph domination, tabu search and the football pool problem, Discrete Appl. Math. 74 (1997) 217–228.

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

[26] R. B. Ellis, Density of normal binary covering codes, Discrete Math. 308 (2008) 4446–4459.

[27] R. B. Ellis, A. B. Kahng, Y. Zheng, JBIG compression algorithms for “dummy fill” VLSI layout data, Tech. Rep. CS2002-0709, VLSI CAD Laboratory, UCSD Department of Computer Science and Engineering (2002).

[28] T. Etzion, P. R. J. Östergård, Greedy and heuristic algorithms for codes and colorings, IEEE Trans. Inform. Theory 44 (1998) 382–388.

[29] G. Exoo, Upper bounds for optimal asymmetric covering codes (2003). URL `http://ginger.indstate.edu/ge/COMBIN/ACODES/`

[30] K. Fadlaoui, P. Galinier, A tabu search algorithm for the covering design problem, J. Heuristics, to appear.

[31] U. Faigle, W. Kern, Some convergence results for probabilistic tabu search, ORSA J. Comput. 4 (1992) 32–37.

[32] T. Feo, M. Resende, Greedy randomized adaptive search procedures, J. Global Optim. 6 (1995) 109–133.

[33] S. Gelly, Y. Wang, R. Munos, O. Teytaud, Modification of UCT with patterns in Monte-Carlo go, Tech. Rep. 6062, INRIA (2006).

[34] F. Glover, Future paths for integer programming and links to artificial intelligence, Comput. Oper. Res. 13 (1986) 533–549.

[35] F. Glover, G. A. Kochenberger (eds.), Handbook of Metaheuristics, Kluwer, Boston, MA, 2003.

[36] F. Glover, M. Laguna (eds.), Tabu Search, Kluwer, Dordrecht, The Netherlands, 1997.

[37] H. O. Hämäläinen, I. S. Honkala, M. K. Kaikkonen, S. Litsyn, Bounds for binary multiple covering codes, Des. Codes Cryptogr. 3 (1993) 251–275.

[38] H. J. van den Herik, L. V. Allis, I. S. Herschberg, Which games will survive? in: D. N. L. Levy, D. F. Beal (eds.), Heuristic Programming in Artificial Intelligence: the Second Computer Olympiad, Ellis Horwood, Upper Saddle River, NJ, 1991, pp. 232–243.

[39] H. J. van den Herik, J. W. H. M. Uiterwijk, J. van Rijswicjk, Games solved: Now and in the future, Artif. Intell. 134 (2002) 277–311.

[40] D. S. Hochbaum, Approximation Algorithms for NP-hard Problems, PWS, Boston, MA, 1997.

[41] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, MIT Press, Cambridge, MA, 1992.

[42] F. Hsu, Behind Deep Blue: Building the Computer that Defeated the World Chess Champion, Princeton University Press, Princeton, NJ, 2002.

[43] P. Kaski, P. R. J. Östergård, Classification Algorithms for Codes and Designs, Springer, Berlin, 2006.

[44] G. Kéri, Tables for bounds on covering codes (2011).
URL `http://www.sztaki.hu/~keri/codes/`

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

[46] D. E. Knuth, The Art of Computer Programming Volume 4A, Combinatorial Algorithms, Part 1, Addison Wesley, Upper Saddle River, NJ, 2011.

[47] D. E. Knuth, R. W. Moore, An analysis of alpha-beta pruning, Artif. Intell. 6 (1975) 293–326.

[48] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (eds.), Proceedings of the 17th European Conference on Machine Learning, No. 4212 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2006, pp. 282–293.

[49] D. L. Kreher, D. R. Stinson, Combinatorial Algorithms: Generation, Enumeration, and Search, CRC Press, Boca Raton, FL, 1998.

[50] M. Krivelevich, B. Sudakov, V. H. Vu, Covering codes with improved density, IEEE Trans. Inform. Theory 49 (2003) 1812–1815.

[51] P. J. M. van Laarhoven, E. H. L. Aarts, Simulated Annealing: Theory and Applications, vol. 37 of Mathematics and its Applications, D. Reidel, Dordrecht, 1987.

[52] A. H. Land, A. G. Doig, An automatic method of solving discrete programming problems, Econometrica 28 (1960) 497–520.

[53] W. Lang, J. Quistorff, E. Schneider, Integer programming for covering codes, J. Combin. Math. Combin. Comput. 88 (2008) 279–288.

[54] E. L. Lawler, Combinatorial Optimization: Networks and Matroids, Dover Publications, Mineola, NY, 2001.

[55] J. H. van Lint, A survey of perfect codes, Rocky Mountain J. Math. 5 (1975) 199–224.

[56] A. C. Lobstein, Bibliography – covering radius (2011).
URL `http://www.infres.enst.fr/~lobstein/biblio.html`

[57] A. C. Lobstein, G. J. M. van Wee, On normal and subnormal q-ary codes, IEEE Trans. Inform. Theory 35 (1989) 1291–1295; and 36 (1990) 1498.

[58] F. J. MacWilliams, N. J. A. Sloane, The Theory of Error-Correcting Codes, North-Holland, Amsterdam, 1977.

[59] B. D. McKay, nauty user's guide (version 1.5), Tech. Rep. TR-CS-90-02, Dept. Computer Science, Australian National University (1990).

[60] B. D. McKay, Isomorph-free exhaustive generation, J. Algorithms 26 (1999) 306–324.

[61] C. Mendes, E. L. Monte Carmelo, M. Poggi, Bounds for short covering codes and reactive tabu search, Discrete Appl. Math. 158 (2010) 522–533.

[62] J. C. Moreira, P. G. Farrel, Essentials of Error-Control Coding, Wiley, Chichester, UK, 2006.

[63] M. Müller, Computer go, Artif. Intell. 134 (2002) 145–179.

[64] J. von Neumann, Zur Theorie der Gesellschaftsspiele, Math. Ann. 100 (1928) 295–320.

[65] P. R. J. Östergård, A new binary code of length 10 and covering radius 1, IEEE Trans. Inform. Theory 37 (1991) 179–180.

[66] P. R. J. Östergård, Upper bounds for q-ary covering codes, IEEE Trans. Inform. Theory 37 (1991) 660–664.

[67] P. R. J. Östergård, New multiple covering codes by tabu search, Australas. J. Combin. 12 (1995) 145–155.

[68] P. R. J. Östergård, Constructing covering codes by tabu search, J. Combin. Des. 5 (1997) 71–80.

[69] P. R. J. Östergård, U. Blass, On the size of optimal binary codes of length 9 and covering radius 1, IEEE Trans. Inform. Theory 47 (2001) 2556–2557.

[70] P. R. J. Östergård, H. O. Hämäläinen, A new table of binary/ternary mixed covering codes, Des. Codes Cryptogr. 11 (1997) 151–178.

[71] P. R. J. Östergård, M. K. Kaikkonen, New upper bounds for binary covering codes, Discrete Math. 178 (1998) 165–179.

[72] P. R. J. Östergård, A. Wassermann, A new lower bound for the football pool problem for six matches, J. Combin. Theory Ser. A 99 (2002) 175–179.

[73] P. R. J. Östergård, W. D. Weakley, Constructing covering codes with given automorphisms, J. Combin. Des. 16 (1999) 65–73.

[74] P. R. J. Östergård, W. D. Weakley, Classification of binary covering codes, J. Combin. Des. 8 (2000) 391–401.

[75] C. J. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Dover, Mineola, NY, 1998.

[76] V. J. Rayward-Smith (ed.), Modern Heuristic Search Methods, Wiley, Chichester, UK, 1996.

[77] C. R. Reeves, J. E. Rowe, Genetic Algorithms – Principles and Perspectives: A Guide to GA Theory, Kluwer, Boston, MA, 2003.

[78] H. Remus, Simulation of a learning machine for playing go, in: C. M. Popplewell (ed.), Proceedings of IFIP Congress 1962, North-Holland, Amsterdam, 1962, pp. 428–432.

[79] S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2003.

[80] E. A. Seuranen, Further results on constructing asymmetric covering codes by tabu search, submitted (2011).

[81] C. E. Shannon, A mathematical theory of communication, Bell Syst. Tech. J. 27 (1948) 379–423, 623–656.

[82] C. E. Shannon, Prediction and entropy of printed English, Bell Syst. Tech. J. 30 (1951) 50–64.

[83] J. C. Spall, Introduction to Stochastic Search and Optimization: Estimation, Simulation and Control, Wiley-Interscience, Hoboken, NJ, 2003.

[84] 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), ACM Press, New York, NY, 1996, pp. 576–584.

[85] H. A. Taha, Integer Programming, Academic Press, London, 1975.

[86] R. J. M. Vaessens, E. H. L. Aarts, J. H. van Lint, Genetic algorithms in coding theory—a table for $A_3(n,d)$, Discrete Appl. Math. 45 (1993) 71–87.

[87] C. Voudouris, E. Tsang, Function optimization using guided local search, Tech. Rep. CSM-249, Department of Computer Science, University of Essex (1995).

[88] E. C. D. van der Werf, AI techniques for the game of go, Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands (2004).

[89] L. T. Wille, New binary covering codes obtained by simulated annealing, IEEE Trans. Inform. Theory 42 (1996) 300–302.

[90] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, IEEE Trans. Evolut. Comput. 1 (1997) 67–82.

[91] Z. Zhang, Linear inequalities for covering codes: part I—pair covering inequalities, IEEE Trans. Inform. Theory 37 (1991) 573–582.

[92] Z. Zhang, C. Lo, Linear inequalities for covering codes: part II—triple covering inequalities, IEEE Trans. Inform. Theory 38 (1992) 1648–1662.

[93] A. L. Zobrist, A pattern recognition program which used a geometry-preserving representation of features, Tech. Rep. 85, Computer Sciences Department, University of Wisconsin (1970).

Nowadays computational methods and results are present in a wide variety of areas, making them an interesting and relevant object of research. The increasing importance is due to the growth of knowledge and computational power enabling new viewpoints to old problems. For instance the LDPC (low density parity check) codes were discovered already in the 1960s but only recently have the practical applications for these near optimal performance codes emerged. Also the recent advances with Monte-Carlo tree search (MCTS) in computer go would not have been possible ten years ago. In this thesis a heuristic search method called tabu search and exhaustive search are used to tackle the combinatorial problem of minimizing the size of several types of covering codes. The similarity of exhaustive search and solving/playing combinatorial games is discussed and the information-theoretic concept of entropy as a way to measure complexity of games is introduced.

Aalto-DD 119/2011

Esa A. Seuranen Computational Methods in Codes and Games

ISBN 978-952-60-4369-2
ISBN 978-952-60-4370-8 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Electrical Engineering
Department of Communications and Networking
www.aalto.fi

Aalto University