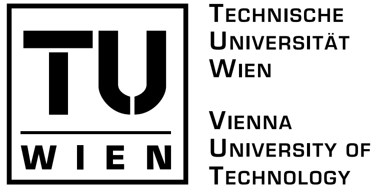


Masterarbeit

Solution Methods for the Social Golfer Problem

Ausgeführt am Institut für

Informationssysteme 184/2
Abteilung für Datenbanken und Artificial Intelligence

der Technischen Universität Wien

unter der Anleitung von Priv.-Doz. Dr. Nysret Musliu

durch

Markus Triska

Kochgasse 19/7, 1080 Wien

Wien, am 20. März 2008

Abstract

The *social golfer problem* (SGP) is a combinatorial optimisation problem. The task is to schedule $g \times p$ golfers in g groups of p players for w weeks such that no two golfers play in the same group more than once. An *instance* of the SGP is denoted by the triple $g{-}p{-}w$. The original problem asks for the maximal w such that the instance $8{-}4{-}w$ can be solved.

In addition to being an interesting puzzle and hard benchmark problem, the SGP and closely related problems arise in many practical applications such as encoding, encryption and covering tasks.

In this thesis, we present and improve upon existing approaches towards solving SGP instances. We prove that the completion problem corresponding to the SGP is NP-complete. We correct several mistakes of an existing SAT encoding for the SGP, and propose a modification that yields considerably improved running times when solving SGP instances with common SAT solvers. We develop a new and freely available finite domain constraint solver, which lets us experiment with an existing constraint-based formulation of the SGP. We use our solver to solve the original problem for 9 weeks, thus matching the best current results of commercial constraint solvers for this instance. We present a new greedy heuristic and a new greedy randomised adaptive search procedure (GRASP) for the SGP. Our method is the first metaheuristic technique that can solve the original problem optimally, and is also highly competitive with existing approaches on many other instances.

Acknowledgements

First and foremost, I thank my advisor Nysret Musliu for supervising this thesis. His encouragement and support were invaluable to me.

Further, I thank Ulrich Neumerkel for introducing me to constraint logic programming, and drawing my attention to PostScript. My gratitude also goes to Mats Carlsson, whose exceptionally elegant CLP(FD) formulation of the SGP spawned my interest in the problem.

Jan Wielemaker provides a robust, feature-rich and free Prolog system, which I used extensively. Thank you!

I also thank my parents for making it all possible in the first place. Last, not least, I thank my girlfriend Beate for her love.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 3
- 1.1 Problem statement . . . 3
- 1.2 Applications of the SGP . . . 3
- 1.3 Computational complexity of the SGP . . . 4
- 1.4 Goals of this thesis . . . 4
- 1.5 Main results of this thesis . . . 4
- 1.6 Further organisation of this thesis . . . 5

2 Design theoretic techniques 6
- 2.1 Introduction . . . 6
- 2.2 Balanced incomplete block designs . . . 6
- 2.3 Steiner triple systems . . . 7
- 2.4 Resolvable designs . . . 8
- 2.5 Kirkman triple systems . . . 10
- 2.6 Finite projective planes . . . 10
- 2.7 Affine planes . . . 11
- 2.8 Latin squares . . . 12
- 2.9 The computational complexity of the SGP . . . 14
- 2.10 Orthogonal Latin squares . . . 15
- 2.11 Isomorphic designs . . . 17
- 2.12 Solving the original SGP instance . . . 18
- 2.13 Conclusion . . . 18

3 SAT formulations 21
- 3.1 Introduction . . . 21
- 3.2 Reasons for SAT formulations . . . 21
- 3.3 The SAT formulation by Gent and Lynce . . . 21
- 3.4 Revisiting the SAT formulation by Gent and Lynce . . . 24
- 3.5 Improving the SAT formulation by Gent and Lynce . . . 26
- 3.6 Experimental results . . . 28
- 3.7 Symmetry breaking . . . 28
- 3.8 Experimental results again . . . 30
- 3.9 More symmetry breaking . . . 31
- 3.10 Working with SAT instances . . . 32
- 3.11 Conclusion . . . 33

4 Constraint programming formulations 37
4.1 Introduction 37
4.2 Constraint logic programming 37
4.3 CLP(FD) 38
4.4 Example: Sudoku 38
4.5 Constraint propagation and search 39
4.6 Selection strategies for variables and values 41
4.7 Visualising the constraint solving process 43
4.8 CLP formulations for the SGP 45
4.8.1 An ECLiPSe formulation of the SGP 45
4.8.2 A SICStus CLP(FD) formulation of the SGP 48
4.9 A new finite domain constraint solver 51
4.10 Experimental results 54
4.11 Conclusion 54

5 A new greedy heuristic for the SGP 57
5.1 An important observation 57
5.2 Freedom of sets of players 58
5.3 A greedy heuristic for the SGP 59

6 Metaheuristic methods 62
6.1 Introduction 62
6.2 Metaheuristic SAT solving 62
6.3 Local search for the SGP 63
6.3.1 The model 63
6.3.2 The neighbourhood 64
6.3.3 The tabu component 64
6.3.4 The tabu search algorithm 65
6.4 Memetic evolutionary programming 65
6.5 A new GRASP for the SGP 66
6.5.1 The greedy heuristic 66
6.5.2 The local search component 67
6.5.3 Experimental results 67
6.5.4 New solutions for the 8−4−10 instance 68
6.6 Conclusion 72

7 Conclusion and future work 73

A Creating portable animations 75

B Bibliography 77

Index 81

1 Introduction

1.1 Problem statement

The *social golfer problem* (SGP) is a combinatorial optimisation problem derived from a question posted to `sci.op-research` in May 1998:

> *32 golfers play golf once a week, and always in groups of 4. For how many weeks can they do this such that no two golfers play in the same group more than once?*

The problem is readily generalised to the following decision problem: Is it possible to schedule $g \times p$ golfers in g groups of p players for w weeks such that no two golfers play in the same group more than once? An *instance* of the SGP is denoted by the triple $g-p-w$. In this thesis, we study the general form of the problem, and solve the original problem instance ($8-4-w$) among others.

Some instances of the SGP have a long history. For example, Euler asked whether two orthogonal Latin squares of order 6 exist, which has become known as "Euler's officer problem" ([Eul82]). In terms of the SGP, this corresponds to solving the $6-6-4$ instance, which is now known to be impossible. As another special case of the SGP, the $5-3-7$ instance also has a long history and is known as Kirkman's schoolgirl problem ([Kir47]).

In the recent past, the SGP has attracted much attention from the constraint programming community. It is problem number 10 in CSPLib, a benchmark library for constraints ([GW99]).

1.2 Applications of the SGP

In addition to being a challenging and interesting puzzle that is obviously useful in golf scheduling, the SGP and closely related problems have many practical applications, such as in encoding, encryption, and covering problems ([HBC70], [Dou94], [GKP95]). We give two simple examples, and more follow in Chapter 2:

Example 1.1. You coordinate a chess tournament for $2n$ players. Each player should play against each other player exactly once, and each player should play exactly once on each day of the tournament. Clearly, this involves designing a schedule for $2n - 1$ days. Further, it is equivalent to solving the SGP instance $n-2-(2n-1)$.

Example 1.2. You communicate with a satellite that can emit n distinct frequencies. You use a code alphabet of m different frequencies for each word. To allow for correction of transmission errors, you require that each pair of frequencies occur *at most once* in the same word. Interpret frequencies as players and words as groups. Clearly, solving the resulting SGP instance yields such a set of code words.

1.3 Computational complexity of the SGP

Little is known about the computational complexity of the SGP. Some instances are easy to solve using deterministic construction methods from *design theory*, a branch of discrete mathematics, and we explain some of them in Chapter 2. It is also clear that not all SGP instances are solvable, and trivial upper bounds are easy to determine. For example, in the original problem statement $(8{-}4{-}w)$, w can be no more than 10.

Proof. Suppose $w \geq 11$, and observe the schedule of an arbitrary but fixed player α. Each week, α plays in a group with 3 *distinct* other players. To play for 11 weeks, α would have to partner $3 \times 11 > 31$ other players. □

1.4 Goals of this thesis

The goals of this thesis are:

1. to present and discuss the most prominent existing approaches towards solving SGP instances, which are:
    - design theoretic techniques
    - SAT encodings
    - constraint-based approaches
    - metaheuristic methods
2. to contribute, as far as possible, to each of these approaches
3. to go, where possible, beyond existing approaches, and obtain new results about the SGP

1.5 Main results of this thesis

We briefly summarise the main results of this thesis:

- We prove that the *completion problem* corresponding to the SGP, i.e., deciding whether a partial schedule can be completed to a valid one, is NP-complete.
- We correct several mistakes in the SAT formulation proposed by Gent and Lynce in [GL05] and [Lyn05]. We then improve upon the corrected formulation, and propose a change in the encoding that significantly reduces the number of variables for all given instances. We show that our formulation can improve running times considerably when solving SGP instances with common SAT solvers.

- We develop a new finite domain constraint solver with the intention to run the SICStus Prolog code published by Carlsson ([Car05]) in a free environment. Guided by customised visualisations of the constraint solving process for the SGP, we use our solver to find solutions for many instances, most notably 8−4−9. This matches the currently best result obtained with commercial constraint solvers for the original problem. Our solver is included in the free Prolog systems SWI-Prolog ([Wie03]) and YAP ([dSC06]).

- We describe a new greedy heuristic based on the notion of *freedom* of sets of players. We use it in a complete backtracking search to solve the instances 8−4−9 and 5−3−7, thus matching constraint-based results on these instances with much simpler methods.

- We use the underlying idea of our greedy heuristic in a new greedy randomised adaptive search procedure (GRASP) for the SGP. Our approach is the first metaheuristic method that can solve the original SGP optimally, and is also highly competitive with existing approaches on many other instances.

1.6 Further organisation of this thesis

In Chapter 2, we introduce terminology from design theory and show deterministic construction methods for SGP instances. We also mention and obtain several existence and inexistence results, and discuss other combinatorial problems that are closely related to the SGP.

In Chapter 3, we correct and improve an existing SAT formulation for the SGP, and solve several SGP instances with common SAT solvers.

We present two existing constraint-based approaches for the SGP in Chapter 4. We use animations of the constraint solving process to obtain valuable suggestions for alternative allocation strategies.

In Chapter 5, we derive a new greedy heuristic for the SGP. We use a variant of this heuristic in Chapter 6, where we discuss existing metaheuristic methods and also present a new GRASP scheme for the SGP.

Chapter 7 concludes and presents opportunities for future research.

2 Design theoretic techniques

2.1 Introduction

Design theory is a subfield of discrete mathematics. Typical applications of design theoretic techniques include statistical designs, and tournaments involving certain balance properties, such as round-robin tournaments.

In this chapter, we introduce several important design theoretic concepts, such as Latin squares, Steiner triple systems and balanced incomplete block designs, and discuss how they relate to the SGP. We also explain a few construction methods, mention and obtain several existence and inexistence results, and establish the computational complexity of the completion problem corresponding to the SGP.

2.2 Balanced incomplete block designs

The statistician F. Yates studied subsets of a set subject to certain balance properties in his 1936 paper [Yat36], with which the modern study of block designs began. We first give the formal definition of what has become known as (v, k, λ) *balanced incomplete block designs* or simply (v, k, λ) *designs*:

Definition 2.1. A (v, k, λ) *balanced incomplete block design* (BIBD) is a collection of k-element subsets (called *blocks*) of a v-element set S $(k < v)$, such that each 2-element subset of S is contained in exactly λ blocks.

BIBDs arise naturally in many statistical experiments, where all comparisons between pairs of elements should typically occur equally often across all possible pairs for fairness. Since $k < v$, no block can contain all elements of S, hence the designation "incomplete". Yates gives the following example of a $(6, 3, 2)$-BIBD in his paper:

Example 2.1. $S = \{a, b, c, d, e, f\}$, with the following 10 3-element blocks: $\{a, b, c\}$, $\{a, b, d\}$, $\{a, c, e\}$, $\{a, d, f\}$, $\{a, e, f\}$, $\{b, c, f\}$, $\{b, d, e\}$, $\{b, e, f\}$, $\{c, d, e\}$, and $\{c, d, f\}$.

In this example, every element occurs in the same number of blocks. Interestingly, this property holds for all BIBDs, and we will later make use of the following well-known theorem ([CD96], [And97]):

Theorem 2.1. In a (v, k, λ) design with b blocks, each element occurs in exactly r blocks, and the following equivalences hold:

$$\lambda(v - 1) = r(k - 1) \tag{2.1}$$

$$bk = vr \tag{2.2}$$

Proof. Take any element $x \in S$, and let r be the number of blocks that contain x. In each of these blocks, x forms a pair with $k-1$ other elements, so there are $r(k-1)$ pairs that involve x. Since x is paired with each of the $v-1$ other elements exactly λ times, it must hold that $r(k-1) = \lambda(v-1)$. This shows that r is uniquely determined by v, k and λ, and must thus be the same for all elements of S. Since each of the v elements appears in r blocks, there are vr appearances of elements in blocks. And since each of the b blocks has k elements, $vr = bk$. □

Specific instances of what has now become known as BIBDs were already studied long before Yates, and often in different contexts. The following example introduces a $(7, 3, 1)$ design called the *seven-point plane* and shows a geometrical interpretation.

Example 2.2. Let $S = \{1, \ldots, 7\}$, with the following seven 3-element blocks: $\{1, 2, 4\}$, $\{2, 3, 5\}$, $\{3, 4, 6\}$, $\{4, 5, 7\}$, $\{5, 6, 1\}$, $\{6, 7, 2\}$, and $\{7, 1, 3\}$. This $(7, 3, 1)$ design is called the *seven-point plane* or *Fano plane*, and it is unique up to renumbering of elements. The 7 elements of S can be considered as *points*, and the blocks as *lines* that connect the elements they contain. A depiction of this design is shown in Fig. 2.1.

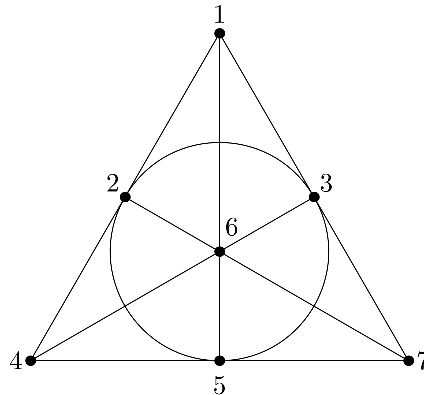


Figure 2.1: The seven-point plane

Notice that in the case of the SGP, a pair of players need not play together in any group at all, in contrast to BIBDs. It is only required that each pair play *at most once* in the same group, i.e., two players can play together either not at all or exactly once. Such designs are therefore also called (0,1)-designs.

2.3 Steiner triple systems

The seven-point plane is an instance of the infinite family of $(v, 3, 1)$ designs, which have become known as *Steiner triple systems* ([LR97], [CD96]):

Definition 2.2. A *Steiner triple system* STS(v) of order v is a $(v,3,1)$ design.

If an STS(v) exists, then by Theorem 2.1 it must hold that $b = \frac{1}{6}v(v-1)$ and $r = \frac{1}{2}(v-1)$. Therefore, $v = 2r+1$, $b = \frac{1}{3}r(2r+1)$, and v must be odd. Continuing, either r or $(2r+1)$ must be divisible by 3. Stating this in another way, we have either $r = 3n$ or $r = 3n+1$, and substituting these two possibilities into the equation for v, we have either $v = 6n+1$ or $v = 6n+3$, justifying the following lemma:

Lemma 2.1. If an STS(v) exists, then $v \equiv 1 \pmod 6$ or $v \equiv 3 \pmod 6$.

The first existence results for Steiner triple systems were obtained by Kirkman, who proved the following in 1847 ([Kir47]): When $v \geq 3$ and either $v \equiv 1 \pmod 6$ or $v \equiv 3 \pmod 6$, an STS(v) exists. Nevertheless, these designs were named after Steiner, who studied them in a geometrical context and posed the problem independently in 1853.

2.4 Resolvable designs

Resolvability is a fundamental concept in combinatorial designs and arises extensively in many practical applications, such as tournament scheduling.

Definition 2.3. A BIBD is *resolvable* if its blocks can be partitioned into c classes such that each element of the design occurs in exactly one of the $v/k = b/c$ groups of each class. The classes are called *parallel classes* or *resolution classes*. The partition into classes is called a *resolution*.

Example 2.3. Consider the following $(9,3,1)$-design consisting of 12 blocks:

$$\begin{array}{cccc} \{0,1,2\} & \{0,3,6\} & \{0,4,8\} & \{0,5,7\} \\ \{3,4,5\} & \{1,4,7\} & \{1,5,6\} & \{1,3,8\} \\ \{6,7,8\} & \{2,5,8\} & \{2,3,7\} & \{2,4,6\} \end{array}$$

Each vertical group of three blocks is a resolution class.

The following theorem shows our first application of design theoretic techniques to the SGP in this chapter:

Theorem 2.2. If D is a resolvable $(v,k,1)$ design, then a resolution of D into w parallel classes is a solution for the SGP instance $\frac{v}{k}$–k–w.

Proof. Enumerate the elements, which correspond to golfers, as $\{1,\dots,v\}$. The design's blocks correspond to groups of golfers, and the w parallel classes are regarded as the weeks. Since they are resolution classes, each golfer plays exactly once in each week. Also, each 2-element subset of $\{1,\dots,v\}$, is contained in exactly one block. Therefore, no pair of golfers occurs more than once in any group. □

As another example, consider the task of constructing a tournament schedule for $2n$ teams in which each team should play against each other team exactly once, and each team should play exactly once on any day. In other words, the task is to find a resolvable $(2n, 2, 1)$ design, which must have $2n - 1$ resolution classes. By Theorem 2.2, such a design would also solve the SGP instance n–2–$(2n - 1)$. One can show:

Theorem 2.3. For each positive integer n, there exists a resolvable $(2n, 2, 1)$ design.

Proof. See [And97]. □

In fact, more holds: There is a well-known construction method for such designs ([And97]), and it is the first construction method for designs that we present in this chapter: Represent the teams by the symbol ∞ and the numbers $1, \ldots, 2n - 1$. Place the numbers equally spaced around a circle, and place the symbol ∞ at the circle's center. The games on day i are built by joining ∞ with i, and the other teams with parallel chords as shown in Fig. 2.2 for the first two days.

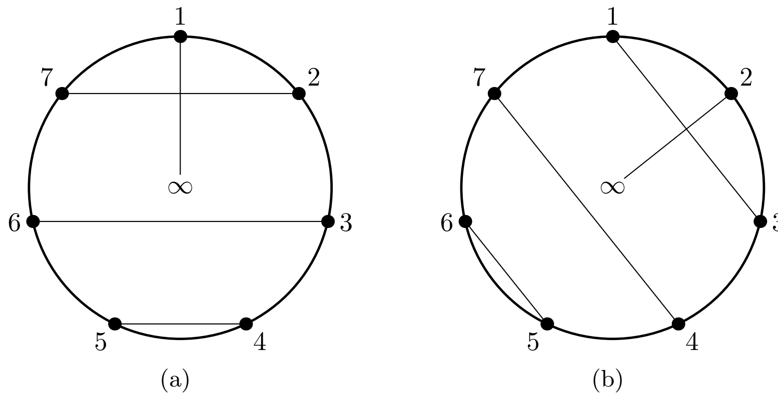


Figure 2.2: Matches between pairs of teams on (a) day 1 and (b) day 2

The construction method justifies the following theorem:

Theorem 2.4. SGP instances of the form n–2–$(2n - 1)$ can be easily solved.

The following result was proved by Kirkman in 1850, and also establishes the existence of corresponding SGP instances:

Proposition 2.1. For each prime p and each positive integer n, a resolvable $(p^n, p, 1)$ design exists.

2.5 Kirkman triple systems

In 1850, Kirkman posed the following problem in the then popular math magazine *Lady's and Gentleman's Diary*:

> *Fifteen young ladies in a school walk out three abreast for seven days in succession: it is required to arrange them daily, so that no two shall walk twice abreast.*

In other words, the task is to find a resolvable (15,3,1) design, i.e., a resolvable STS(15). Such designs are instances of Kirkman triple systems:

Definition 2.4. A *Kirkman triple system* of order v, denoted as KTS(v), is a resolvable STS(v).

Example 2.4. Example 2.3 shows a KTS(9).

Kirkman's original problem, i.e., finding a KTS(15), has become known as *Kirkman's schoolgirl problem*, and Kirkman solved it himself with an interesting construction method. We will later solve it with different methods. The following is immediate and well-known:

Theorem 2.5. A KTS(v) can only exist if $v \equiv 3 \pmod 6$.

Proof. From Lemma 2.1, the case $v \equiv 1 \pmod 6$ can be eliminated, since v must be divisible by 3 for resolvability. □

Using Proposition 2.1, we obtain:

Theorem 2.6. A KTS(3^n) exists for all positive integers n.

Applying Theorem 2.1 yields the following corollary:

Corollary 2.1. SGP instances of the form 3^{n-1}–3–$\frac{3^n-1}{2}$ are solvable.

2.6 Finite projective planes

In addition to being a Steiner triple system, the seven-point plane is also a member of another infinite family of designs which are called *finite projective planes* ([CD96]):

Definition 2.5. A *finite projective plane* of order n is an $(n^2+n+1, n+1, 1)$ design, $n \geq 2$.

Example 2.5. The seven-point plane is a finite projective plane of order 2.

Finite projective planes originate from geometrical contexts and can be equivalently defined in the terminology of axiomatic geometry:

Definition 2.6. A *finite incidence structure* $P = (\mathcal{P}, \mathcal{L}, I)$, also called *finite geometry*, is a finite set of *points* $\mathcal{P}$, a finite set of *lines* $\mathcal{L}$, and an incidence relation I between them.

Definition 2.7. A *finite projective plane* P is a finite incidence structure with the following properties:

1. any two distinct points are incident with exactly one line
2. any two distinct lines are incident with exactly one point
3. there exists a *quadrangle*, i.e., four points such that no line is incident with more than two of them

It can be shown that for any finite projective plane P, there is a positive integer n such that every line of P has exactly $n + 1$ points. The number n is called the *order* of P, and coincides with the order defined from the design theoretic view. Determining which positive integers are orders of finite projective planes is currently an open question in finite geometry. One can show that there is a finite projective plane of order n when n is a prime power, and the orders of all known examples of finite projective planes are prime powers. It is also known that no finite projective plane of order 10 exists. This result was obtained by a backtracking search conducted by Lam, Thiel and Swiercz after more than 800 days of CPU time ([LTS86]). A theorem by Bruck and Ryser ([BR49]) rules out an infinite number of other cases.

2.7 Affine planes

If one interprets a resolvable design geometrically, with the blocks being the lines, then one can regard the blocks within a single resolution class as *parallel lines*, since they have no point, i.e., element, in common. Using again terminology originating from geometry, we define ([CD96]):

Definition 2.8. A *finite affine plane* is a finite incidence structure with the following properties:

1. any two distinct points are incident with exactly one line
2. for any point P outside a line ℓ there is exactly one line through P that has no point in common with ℓ
3. there exist three points that are not incident with a common line

Property 2 is the *Euclidean parallel axiom*, and affine planes are therefore sometimes called *Euclidean planes*. They stand in contrast to projective planes, in which any two lines meet.

From a design theoretic point of view, a finite affine plane is equivalently defined as ([And97]):

Definition 2.9. A finite affine plane of order n is an $(n^2, n, 1)$ design.

A projective plane P can be constructed from an affine plane A as follows (see [LR97]): Let the points of P be the points of A plus one point for each parallel class of A, and let the lines of P be the lines of A, plus one new line ℓ^*. Let ℓ^* be incident with all points corresponding to parallel classes, and add to each line ℓ of P the point corresponding to the parallel class to which ℓ belongs in A. Conversely, given a projective plane, simply remove any line together with all its points to obtain an affine plane. This justifies the following well-known theorem:

Theorem 2.7. An affine plane of order n exists if and only if a finite projective plane of order n exists.

The following important property holds (see [LR97]):

Proposition 2.2. Every affine plane is resolvable.

Every solution for an SGP instance of the form $n{-}n{-}(n+1)$ $(n > 1)$ must thus be an affine plane. Finding a solution for such an instance where n is *not* a prime power would therefore settle an open problem of finite geometry, which is whether a finite projective plane whose order is not a prime power exists. The connection between affine and projective planes also implies the following theorem:

Theorem 2.8. The SGP instance $10{-}10{-}11$ cannot be solved.

Proof. A solution for the SGP instance $10{-}10{-}11$ is a resolvable $(10^2, 10, 1)$ design and thus a finite affine plane of order 10. This would imply the existence of a finite projective plane of order 10, which does not exist. □

The aforementioned Bruck-Ryser theorem rules out additional cases, such as $14{-}14{-}15$ and $22{-}22{-}23$.

2.8 Latin squares

Latin squares play an important rôle in the design of statistical experiments and many related problems ([CD96]):

Definition 2.10. A *Latin square* of order n is an $n \times n$ array in which each cell contains a single symbol from a set S with n elements, such that each symbol occurs exactly once in each row and exactly once in each column.

Example 2.6. Fig. 2.3 (a) shows a Latin square of order 3.

Various experiments can be scheduled with the help of Latin squares. For example, suppose you want to compare 9 laptops, 3 each from 3 different vendors, testing one of each vendor with each of 3 different operating

systems. Let vendors correspond to columns, operating systems to rows, and the numbers in the Latin square of Fig. 2.3 (a) to different testers. If you now select, for each of the 3 different testers, the laptops and operating systems that carry that tester's number, then you obtain a schedule such that one laptop from each vendor, and one of each of the operating systems is tested by each tester. Also, the testers can do their work in parallel.

A single Latin square of order n solves the SGP instance $n{-}n{-}1$: Superimpose n^2 players on the square, with one player for each cell. The element of each cell of the Latin square denotes into which group the corresponding player should be scheduled. For example, in Fig. 2.3, the players whose cells contain the value 2 in the Latin square (i.e., players 1, 5 and 6) play in group 2 in the resulting schedule. Since the Latin square assigns a group to each player, and all elements occur equally often, the resulting schedule is valid. The converse does *not* hold though. As Fig. 2.4 shows, a valid schedule of the SGP instance $n{-}n{-}1$ need not correspond to a Latin square for all superimposed player arrays, since players that occur in the same row of the superimposed array can also play in the same group.

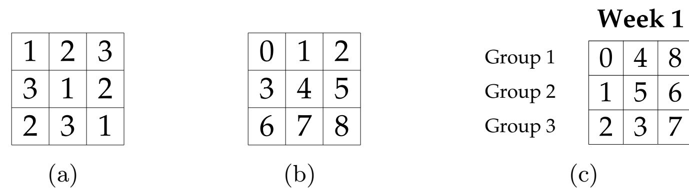


Figure 2.3: (a) Latin square of order 3, (b) superimposed players, (c) induced solution for the SGP instance 3−3−1

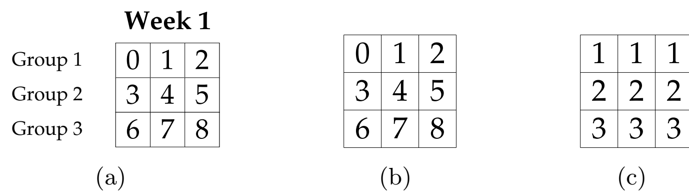


Figure 2.4: (a) A solution for the SGP instance 3−3−1, (b) superimposed players, (c) induced assignment of players to groups

Clearly, it is trivial to construct a Latin square from scratch: For example, start with any permutation of elements in the first row, and cyclically shift all elements once to the right for the following rows. It is much harder to decide whether a partially filled square can be *completed* to a Latin square. This decision problem was proved to be NP-complete by Colbourn ([Col84]). However, this does *not* immediately imply that the completion problem corresponding to the SGP is NP-complete as well, since – as shown above – solutions for the SGP can be constructed that do not yield a Latin square.

2.9 The computational complexity of the SGP

Little is known about the computational complexity of the SGP. Some instances are easily seen to be unsolvable, and some are easily solved with construction methods. Here, we show that the *completion problem* corresponding to the SGP, i.e., deciding whether a partially filled array can be completed to a valid schedule, is NP-complete.

It is clear that the problem is in NP, since the validity of any schedule can be verified in polynomial time in the size of the input. Essentially only all pairs of players have to be checked.

To complete the proof of NP-completeness, we show that the problem is also NP-hard. We reduce the completion problem of Latin squares, which was shown to be NP-complete in [Col84], to the completion problem of SGP instances: To check whether a partially filled $n \times n$ array S is completable to a Latin square of order n, first construct two conflict-free weeks of the SGP instance $n{-}n{-}3$ as follows: Construct the first week arbitrarily, for example, by lining up the players in their natural order. Construct the second week by transposing the first week. Fig. 2.5 shows the first two such weeks of instance $4{-}4{-}3$ built in this way.

	Week 1				Week 2				Week 3			
Group 1	0	1	2	3	0	4	8	12				
Group 2	4	5	6	7	1	5	9	13				
Group 3	8	9	10	11	2	6	10	14				
Group 4	12	13	14	15	3	7	11	15				

Figure 2.5: The first two weeks of instance $4{-}4{-}3$

The third week is reserved for the partially filled square S: Superimpose the player array of the first week onto S, and place the corresponding players into groups of the third week as indicated by the filled cells of S. The resulting partial array of the SGP instance $n{-}n{-}3$ can be completed to a valid schedule iff S can be completed to a Latin square, thus making the corresponding decision problem at least as hard.

Proof. Suppose S can be completed to a Latin square. Then its completion certainly forms a valid week, for any superimposed array of all players. If players are superimposed as in the first week of the schedule, it also contains no conflicts with the first two weeks: Each player partners only players from its own row or column in the first two weeks of the schedule, and a week induced by a Latin square excludes those.

Conversely, suppose the partial array can be completed to a valid schedule. Since each player partners all players from its own row and column in the first two weeks, the induced group assignment of the third week must exclude those and therefore completes S to a Latin square. $\square$

2.10 Orthogonal Latin squares

In order to use Latin squares to construct SGP solutions for several weeks, it is useful to define a relation called *orthogonality* between pairs of Latin squares ([LR97]):

Definition 2.11. Two Latin squares $A = (a_{ij})$ and $B = (b_{ij})$ of order n are *orthogonal* if all ordered pairs (a_{kl}, b_{kl}) are distinct. Stated in another way, if a_{ij} and a_{kl} are the same for two different cells of A, then b_{ij} must differ from b_{kl}.

The concept of orthogonality was first formalised by Euler in [Eul82], where he posed the following question:

> *Can 36 officers be arranged in a 6×6 square such that each of 6 regiments and each of 6 ranks appear in each row and column exactly once?*

This has become known as "Euler's officer problem" and asks essentially for two orthogonal Latin squares of order 6. Euler correctly conjectured that constructing two such Latin squares is impossible, which was shown by Tarry in [Tar00]. Since Euler used Greek and Latin letters for orthogonal Latin squares, they are also known as *Graeco-Latin squares*.

A set of Latin squares is *mutually* orthogonal if they are orthogonal in pairs. Mutually orthogonal Latin squares (MOLS) are used in the design of statistical experiments, as well as for other purposes, such as authentication and encoding ([Dou94], [HBC70]). MOLS can also be interpreted as solutions for SGP instances, by letting each of the squares form a single week as explained in Section 2.8, using the same array of superimposed players for each week. An example is shown in Fig. 2.6, where the players were superimposed as in Fig. 2.4 (b). Orthogonality ensures that two players cannot be scheduled in the same group more than once.

1	2	3
3	1	2
2	3	1

1	2	3
2	3	1
3	1	2

(a)

	Week 1			Week 2		
Group 1	0	4	8	0	5	7
Group 2	1	5	6	1	3	8
Group 3	2	3	7	2	4	6

(b)

Figure 2.6: (a) 2 orthogonal Latin squares of order 3, (b) corresponding solution for the SGP instance 3−3−2, if players are superimposed as in Fig. 2.4 (b)

Importantly, both the reference array and its transpose can be added to a schedule induced by a set of MOLS without causing conflicts, since the players in the same row or column of the reference array cannot play

together in any week stemming from a Latin square. The following property holds ([HW05], [DH05]):

Lemma 2.2. A set of k MOLS of order n is equivalent to a solution of the SGP instance n–n–$(k+2)$.

Applying Tarry's result yields the following well-known theorem:

Theorem 2.9. The SGP instance 6–6–4 cannot be solved.

To further benefit from the connection between the SGP and MOLS, it is useful to know how MOLS can be constructed. Harvey and Winterer have compared several construction methods for MOLS in [HW05], and we now explain one of the methods they cite in more detail. First, we mention an upper bound on the maximal number of MOLS of order n, which is also well-known ([And97], [LR97]):

Proposition 2.3. Let $N(n)$ denote the maximal cardinality of a set of MOLS of order n. Then $N(n) \leq n-1$.

This upper bound motivates to the following definition ([LR97]):

Definition 2.12. A set of $(n-1)$ MOLS of order n is called *complete*.

Applying Lemma 2.2, we obtain:

Lemma 2.3. A complete set of MOLS of order n induces a solution for the SGP instance n–n–$(n+1)$.

The following is also well-known ([LR97]) and follows from our previous observations:

Theorem 2.10. An affine plane of order n (and therefore, by Theorem 2.7, a projective plane of order n) is equivalent to a complete set of MOLS of order n.

We mentioned already that if n is a prime power, a finite projective plane of order n exists, but did not explain how to construct such a plane. In the following, we show how to construct a complete set of MOLS of order n, if n is a prime power. Such a set induces an affine plane of order n and thus a projective plane of order n, and simultaneously solves the SGP instance n–n–$(n+1)$, justifying the following theorem:

Theorem 2.11. If n is a prime power, then the SGP instance n–n–$(n+1)$ can be solved.

The well-known construction ([CD96], [HW05]) is based on finite fields, which are also called Galois fields in honour of Evariste Galois. Let GF(n) be a finite field of order n. Such a field exists if n is a prime power, and there

are several probabilistic and deterministic algorithms to construct such a field (see [Sho94] and the references included therein). For each $\alpha \in \mathrm{GF}(n)$, $\alpha \neq 0$, let $L_\alpha(i,j) = \alpha i + j$ for all $i, j \in \mathrm{GF}(n)$, and with multiplication and addition as defined in $\mathrm{GF}(n)$. Then the set $\{L_1, L_2, \ldots, L_{n-1}\}$ is a complete set of MOLS of order n.

Proof. It is easy to see that each L_i is a Latin square of order n. We prove that these Latin squares are mutually orthogonal: Suppose $L_a(i_1, j_1) = L_a(i_2, j_2)$ and $L_b(i_1, j_1) = L_b(i_2, j_2)$ for $0 < a < b$. Then, by construction of L_a and L_b:

$$ai_1 + j_1 = ai_2 + j_2$$
$$bi_1 + j_1 = bi_2 + j_2$$

Rewriting yields:

$$(i_1 - i_2)a = (i_1 - i_2)b$$

Since $a \neq b$, $i_1 - i_2$ must be 0, thus $i_1 = i_2$ and therefore $j_1 = j_2$. □

2.11 Isomorphic designs

In many cases, one is interested in whether two designs are *isomorphic*, i.e., structurally identical. This means that one can be obtained from the other simply by renaming elements and reordering blocks. The following notion allows to reduce isomorphism testing of designs to isomorphism testing of *graphs*:

Definition 2.13. Let $D = (V, \mathcal{B})$ be a design, where $V = \{x_1, x_2, \ldots, x_v\}$ are the *points* and $\mathcal{B} = \{B_1, B_2, \ldots, B_b\}$ are the *blocks*. Let $G(D)$ be a graph with vertex set $\{x_1, x_2, \ldots, x_v, B_1, B_2, \ldots, B_b\}$, with the element vertices having one colour, and the block vertices having a second colour, and undirected edge set $\{(x_i, B_j) : x_i \in B_j\}$. The graph $G(D)$ is the *Levi graph* of D.

The following holds ([CD96]):

Proposition 2.4. Designs D_1 and D_2 are isomorphic iff graphs $G(D_1)$ and $G(D_2)$ are isomorphic.

We use this property in the next section, where we show that two currently known optimal solutions for the original SGP instance are isomorphic.

2.12 Solving the original SGP instance

After the original SGP instance ($8-4-w$) was first posted to the discussion group `sci.op-research` in 1998, a solution for 9 weeks was soon found. It was also clear that no solution for 11 weeks could exist (see Section 1.3). Whether there exists a solution for 10 weeks was an open question for several years, until Alejandro Aguado constructed an explicit solution for the $8-4-10$ instance in 2004 ([Agu04]), using a result by Colbourn ([Col99]). There, Colbourn uses a combination of backtracking search, instance-specific considerations and design theoretic techniques, and solves the SGP instance $8-4-10$ to obtain a certain different design that is the main subject of his paper. According to Aguado, the connection between Colbourn's result and the SGP was pointed out by Alan C. H. Ling.

Fig. 2.7 shows Aguado's solution for the $8-4-10$ instance ([Agu04]). A different solution for this instance was posted by Andrew John Sadler to the discussion group `comp.constraints`, and we show it in Fig. 2.8. To test whether these designs are isomorphic, we use an obvious extension of the Levi graph and Proposition 2.4: We introduce a third colour for *weeks*, and connect the vertex corresponding to a group g_j to the vertex corresponding to a week w_k iff g_j is a group that occurs in w_k. Using McKay's `dreadnaut` program ([McK90]), we have found that these two designs are isomorphic, and we show a structure-preserving bijection between the 122 vertices of their extended Levi graphs in Fig. 2.9. The vertices are numbered as follows for each of the two designs: Vertices 0–31 represent the players, in their natural order. Notice that while the two solutions use different origins for player numbers (0 and 1, respectively), both Levi graphs start with vertex 0. Vertices 32–111 correspond to groups, lined up week after week in their natural order, and vertices 112–121 are reserved for weeks, also in their natural order. For example, from Fig. 2.9, player number 6 in Aguado's solution corresponds to player number 9 in the other solution, and player number 7 corresponds to player number 10. The group in which they play together is the third group of the first week in both cases, as vertex number 34 is mapped onto itself.

We add to these equivalent (up to isomorphism) solutions two new non-isomorphic ones in Chapter 6.

2.13 Conclusion

When applicable, design theoretic techniques are often among the fastest ways to solve a given SGP instance. However, they also have several drawbacks. First, they are not generally applicable. While construction methods and (in)existence results for many families of SGP instances are known and have been presented, there is no known deterministic method that solves any given SGP instance, or shows that it cannot be solved. Using the

	Week 1	Week 2	Week 3	Week 4	Week 5
Group 1	0 1 2 3	0 4 8 28	0 11 14 21	0 18 24 27	0 6 13 26
Group 2	4 5 22 23	1 6 18 23	1 7 10 28	1 9 19 26	1 4 11 15
Group 3	6 7 20 21	2 7 17 22	2 15 20 25	2 8 11 16	2 9 21 28
Group 4	8 25 26 27	3 5 26 31	3 13 22 24	3 10 17 25	3 8 14 23
Group 5	9 10 11 24	9 13 14 27	4 9 18 31	4 7 12 29	5 12 18 25
Group 6	12 13 15 30	10 15 19 21	5 16 27 30	5 6 14 15	7 19 24 30
Group 7	14 28 29 31	11 25 29 30	6 8 19 29	13 20 23 28	10 16 22 29
Group 8	16 17 18 19	12 16 20 24	12 17 23 26	21 22 30 31	17 20 27 31

	Week 6	Week 7	Week 8	Week 9	Week 10
Group 1	0 7 25 31	0 5 19 20	0 15 17 29	0 9 12 22	0 10 23 30
Group 2	1 5 24 29	1 14 22 25	1 13 16 31	1 8 20 30	1 12 21 27
Group 3	2 12 14 19	2 23 27 29	2 4 26 30	2 5 10 13	2 6 24 31
Group 4	3 18 28 30	3 4 16 21	3 6 11 12	3 7 15 27	3 9 20 29
Group 5	4 6 10 27	6 9 17 30	5 7 8 9	4 14 17 24	4 13 19 25
Group 6	8 13 17 21	7 11 13 18	10 14 18 20	6 16 25 28	5 11 17 28
Group 7	9 15 16 23	8 10 12 31	19 22 27 28	11 19 23 31	7 14 16 26
Group 8	11 20 22 26	15 24 26 28	21 23 24 25	18 21 26 29	8 15 18 22

Figure 2.7: Aguado's solution for the 8−4−10 instance ([Agu04])

	Week 1	Week 2	Week 3	Week 4	Week 5
Group 1	1 2 3 4	1 5 13 26	1 7 17 21	1 6 11 32	1 8 18 24
Group 2	5 6 7 8	2 8 9 31	2 11 13 24	2 7 14 25	2 12 16 21
Group 3	9 10 11 12	3 7 10 30	3 6 18 22	3 8 16 27	3 9 20 28
Group 4	13 14 15 16	4 6 15 28	4 10 16 23	4 5 12 29	4 11 17 27
Group 5	17 18 19 20	11 20 21 29	5 20 25 30	9 17 24 30	5 14 22 32
Group 6	21 22 23 24	12 18 23 32	8 19 28 32	10 19 22 31	6 19 26 30
Group 7	25 26 27 28	14 19 24 27	9 14 26 29	13 18 21 28	7 13 23 31
Group 8	29 30 31 32	16 17 22 25	12 15 27 31	15 20 23 26	10 15 25 29

	Week 6	Week 7	Week 8	Week 9	Week 10
Group 1	1 9 15 22	1 12 19 25	1 10 14 28	1 16 20 31	1 23 27 30
Group 2	2 5 19 23	2 10 18 26	2 6 20 27	2 15 17 32	2 22 28 29
Group 3	3 12 17 26	3 11 14 23	3 21 25 32	3 13 19 29	3 5 15 24
Group 4	4 8 13 25	4 7 20 22	4 24 26 31	4 14 18 30	4 9 19 21
Group 5	6 14 21 31	5 17 28 31	5 9 16 18	5 10 21 27	6 10 13 17
Group 6	7 18 27 29	6 16 24 29	7 11 15 19	6 9 23 25	7 16 26 32
Group 7	10 20 24 32	8 15 21 30	8 17 23 29	7 12 24 28	8 12 14 20
Group 8	11 16 28 30	9 13 27 32	12 13 22 30	8 11 22 26	11 18 25 31

Figure 2.8: A solution for the 8−4−10 instance, taken from `comp.constraints`. It is isomorphic to Aguado's solution.

$6 \leftrightarrow 8$, $7 \leftrightarrow 9$, $8 \leftrightarrow 12$, $9 \leftrightarrow 16$, $10 \leftrightarrow 17$, $11 \leftrightarrow 18$, $12 \leftrightarrow 20$, $13 \leftrightarrow 21$, $14 \leftrightarrow 24$, $15 \leftrightarrow 22$, $16 \leftrightarrow 28$, $17 \leftrightarrow 29$, $18 \leftrightarrow 30$, $19 \leftrightarrow 31$, $20 \leftrightarrow 10$, $21 \leftrightarrow 11$, $22 \leftrightarrow 6$, $23 \leftrightarrow 7$, $24 \leftrightarrow 19$, $25 \leftrightarrow 13$, $26 \leftrightarrow 14$, $27 \leftrightarrow 15$, $28 \leftrightarrow 25$, $29 \leftrightarrow 26$, $30 \leftrightarrow 23$, $31 \leftrightarrow 27$, $44 \leftrightarrow 47$, $47 \leftrightarrow 44$, $48 \leftrightarrow 88$, $49 \leftrightarrow 89$, $50 \leftrightarrow 90$, $51 \leftrightarrow 91$, $52 \leftrightarrow 92$, $53 \leftrightarrow 93$, $54 \leftrightarrow 95$, $55 \leftrightarrow 94$, $56 \leftrightarrow 96$, $57 \leftrightarrow 97$, $58 \leftrightarrow 98$, $59 \leftrightarrow 99$, $60 \leftrightarrow 100$, $61 \leftrightarrow 101$, $62 \leftrightarrow 103$, $63 \leftrightarrow 102$, $64 \leftrightarrow 72$, $65 \leftrightarrow 73$, $66 \leftrightarrow 74$, $67 \leftrightarrow 75$, $68 \leftrightarrow 76$, $69 \leftrightarrow 78$, $70 \leftrightarrow 77$, $71 \leftrightarrow 79$, $72 \leftrightarrow 80$, $73 \leftrightarrow 81$, $74 \leftrightarrow 82$, $75 \leftrightarrow 83$, $76 \leftrightarrow 84$, $77 \leftrightarrow 87$, $78 \leftrightarrow 86$, $79 \leftrightarrow 85$, $80 \leftrightarrow 48$, $81 \leftrightarrow 49$, $82 \leftrightarrow 50$, $83 \leftrightarrow 51$, $84 \leftrightarrow 52$, $85 \leftrightarrow 53$, $86 \leftrightarrow 54$, $87 \leftrightarrow 55$, $88 \leftrightarrow 104$, $89 \leftrightarrow 105$, $90 \leftrightarrow 106$, $91 \leftrightarrow 107$, $92 \leftrightarrow 108$, $93 \leftrightarrow 111$, $94 \leftrightarrow 109$, $95 \leftrightarrow 110$, $96 \leftrightarrow 56$, $97 \leftrightarrow 57$, $98 \leftrightarrow 58$, $99 \leftrightarrow 59$, $100 \leftrightarrow 60$, $101 \leftrightarrow 62$, $102 \leftrightarrow 61$, $103 \leftrightarrow 63$, $104 \leftrightarrow 64$, $105 \leftrightarrow 65$, $106 \leftrightarrow 66$, $107 \leftrightarrow 67$, $108 \leftrightarrow 68$, $109 \leftrightarrow 69$, $110 \leftrightarrow 71$, $111 \leftrightarrow 70$, $114 \leftrightarrow 119$, $115 \leftrightarrow 120$, $116 \leftrightarrow 117$, $117 \leftrightarrow 118$, $118 \leftrightarrow 114$, $119 \leftrightarrow 121$, $120 \leftrightarrow 115$, $121 \leftrightarrow 116$

Figure 2.9: Isomorphism between the 122 vertices of the extended Levi graphs of the two solutions for the $8-4-10$ instance in Fig. 2.7 and 2.8. Omitted vertices are mapped onto themselves.

SGP's relation to Latin squares, we proved that its completion problem is NP-complete. Clearly, an efficient method for deciding whether a partially filled schedule can be completed to a valid one would also entail an efficient method for solving all solvable instances: Simply try all players for one of the free cells, and if the schedule can be completed with any player in that place, fix this choice and apply the same strategy until no free cells remain.

Second, construction methods are generally unable to cope with partially instantiated schedules, or even slight variations of the given constraints. This can be a problem in practical applications, where further restrictions or slightly different constraints will often have to be taken into account.

Finding a solution for an SGP instance n–n–$(n+1)$ $(n > 1)$ where n is *not* a prime power would settle an open question from finite geometry: whether a finite projective plane whose order is not a prime power exists.

3 SAT formulations

3.1 Introduction

Since the SGP is in NP, and the Boolean satisfiability problem (SAT) is complete for that complexity class, any SGP instance can be reduced to a SAT instance. A SAT formulation for the SGP is proposed by Gent and Lynce in [GL05] and [Lyn05].

In this chapter, we first present the SAT formulation for the SGP as it appears in the literature. We find several omissions and mistakes in the existing SAT formulation, which we correct. We then improve upon the corrected formulation, and propose a different encoding that significantly reduces the number of variables for all instances. We present experimental results obtained from using local search and complete backtracking to solve several generated SAT instances that encode SGP instances. We show that our formulation can lead to considerably improved running times when solving SGP instances in this way.

3.2 Reasons for SAT formulations

SAT was the first problem proved to be NP-complete ([Coo71]) and plays an important rôle in complexity theory for this reason. Since all problems in NP can thus be expressed as SAT instances, the implementation of practical SAT solvers has received significant attention from researchers. Several different strategies for solving SAT instances have been proposed, including complete backtracking search ([DLL62]), local search ([SKC93]) and algebraic simplification techniques ([MS00]). Since SAT solvers have become continuously faster as a result of these efforts and are often also freely available, it is attractive to try out SAT formulations where possible.

3.3 The SAT formulation by Gent and Lynce

Consider the general g–p–w instance of the SGP, with $x = g \times p$ the number of golfers. For their SAT formulation in [GL05], Gent and Lynce introduce variables G_{ijkl} ($1 \leq i \leq x$, $1 \leq j \leq p$, $1 \leq k \leq g$ and $1 \leq l \leq w$) denoting whether player i plays in position j in group k and week l. The constraints are then imposed by a set of clauses ensuring that:

- Each player plays exactly once per week, i.e.:
  - Each player plays *at least* once per week
  - Each player plays *at most* once per week
- Each group consists of exactly p players, i.e.:
  - *At least* one player is the jth golfer ($1 \leq j \leq p$)

- *At most* one player is the jth golfer ($1 \leq j \leq p$)

- No two players play in the same group more than once

The following clauses ensure that each player plays *at least* once in each week:

$$\bigwedge_{i=1}^{x} \bigwedge_{l=1}^{w} \bigvee_{j=1}^{p} \bigvee_{k=1}^{g} \mathrm{G}_{ijkl} \tag{3.1}$$

To enforce that each player plays *at most* once each week, it is first ensured that each player plays at most once *per group* in each week:

$$\bigwedge_{i=1}^{x} \bigwedge_{l=1}^{w} \bigwedge_{j=1}^{p} \bigwedge_{k=1}^{g} \bigwedge_{m=j+1}^{p} \neg \mathrm{G}_{ijkl} \vee \neg \mathrm{G}_{imkl} \tag{3.2}$$

A second set of clauses is supposed to guarantee that no player plays in more than one group in any week:

$$\bigwedge_{i=1}^{x} \bigwedge_{l=1}^{w} \bigwedge_{j=1}^{p} \bigwedge_{k=1}^{g} \bigwedge_{m=k+1}^{g} \bigwedge_{n=j+1}^{p} \neg \mathrm{G}_{ijkl} \vee \neg \mathrm{G}_{inml} \tag{3.3}$$

However, clause set (3.3) as proposed in [GL05] is incomplete, and we return to it below. Taking our modification below into account, the clause set (3.1) ∪ (3.2) ∪ (3.3) enforces that each player plays exactly once per week.

A similar set of clauses is introduced for *groups* of golfers:

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{j=1}^{p} \bigvee_{i=1}^{x} \mathrm{G}_{ijkl} \tag{3.4}$$

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{j=1}^{p} \bigwedge_{i=1}^{x} \bigwedge_{m=i+1}^{x} \neg \mathrm{G}_{ijkl} \vee \neg \mathrm{G}_{imkl} \tag{3.5}$$

The set (3.4) ∪ (3.5) is intended to yield valid groups, i.e., groups where exactly one player is in position j for each $1 \leq j \leq p$. We will return to this clause set below as well.

The only constraint left to encode is "socialisation", i.e., no two players can play in the same group more than once. To express this constraint, Gent and Lynce introduce a set of auxiliary variables and a so-called *ladder* matrix. The auxiliary variables G'_{ikl} ($1 \leq i \leq x$, $1 \leq k \leq g$ and $1 \leq l \leq w$) denote whether player i plays in group k and week l. They are related to the variables G_{ijkl} via the equivalence:

$$\mathrm{G}'_{ikl} \leftrightarrow \bigvee_{j=1}^{p} \mathrm{G}_{ijkl} \tag{3.6}$$

posted for all $1 \leq i \leq x$, $1 \leq k \leq g$ and $1 \leq l \leq w$.

The ladder matrix is a $\binom{x}{2} \times (g \times w)$ array of propositional variables denoted by $\textsc{Ladder}_{yz}$. A complete assignment of the ladder variables is said to be *valid* iff every row is a sequence of zero or more TRUE assignments followed by only FALSE assignments. To enforce this, Gent and Lynce propose the set of clauses:

$$\bigwedge_{y=1}^{\binom{x}{2}-1} \bigwedge_{z=1}^{g\times w} \neg\textsc{Ladder}_{yz+1} \vee \textsc{Ladder}_{yz} \tag{3.7}$$

We will return to this clause set below. For now, we illustrate the intention of the ladder matrix with an example given in Fig. 3.1. It is taken from [GL05] and corresponds to the schedule shown in Fig. 3.2. Each row of the matrix corresponds to a pair of golfers. In each row, the column of the rightmost TRUE value, if any, denotes the group in which the respective two golfers play together. We highlight the rightmost TRUE value of each row using a bold **T**. Obviously, at most one TRUE assignment can be the rightmost one in each row. Therefore, each pair of players can occur in at most one group.

	1.1	1.2	2.1	2.2	3.1	3.2
3.4	T	**T**	F	F	F	F
2.3	T	T	T	T	T	**T**
2.4	T	T	T	**T**	F	F
1.2	**T**	F	F	F	F	F
1.3	T	T	**T**	F	F	F
1.4	T	T	T	T	**T**	F

Figure 3.1: Ladder matrix for the schedule of Fig. 3.2

	Week 1		**Week 2**		**Week 3**	
Group 1	1	2	1	3	1	4
Group 2	3	4	2	4	2	3

Figure 3.2: Schedule corresponding to the ladder matrix of Fig. 3.1

The remaining sets of clauses relate the auxiliary variables G'_{ikl} with the ladder variables. If two golfers i and m play in the same group, i.e., $\mathrm{G}'_{ikl} \wedge \mathrm{G}'_{mkl}$ is true for $i < m$, then, according to [GL05], $\textsc{Ladder}_{[\binom{x-i}{2})+m-i](l\times k)}$ must be TRUE and $\textsc{Ladder}_{[\binom{x-i}{2})+m-i](l\times k+1)}$ must be FALSE, and conversely. Formally, they propose the following sets of clauses:

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \neg\mathrm{G}'_{ikl} \vee \neg\mathrm{G}'_{mkl} \vee \textsc{Ladder}_{\binom{x-i}{2})+m-i,l\times k} \tag{3.8}$$

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \neg \mathrm{G}'_{ikl} \vee \neg \mathrm{G}'_{mkl} \vee \neg \textsc{Ladder}_{\binom{x-i}{2}+m-i,l\times k+1} \tag{3.9}$$

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \textsc{Ladder}_{\binom{x-i}{2}+m-i,l\times k+1} \vee \neg \textsc{Ladder}_{\binom{x-i}{2}+m-i,l\times k} \vee \neg \mathrm{G}'_{ikl} \tag{3.10}$$

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \textsc{Ladder}_{\binom{x-i}{2}+m-i,l\times k+1} \vee \neg \textsc{Ladder}_{\binom{x-i}{2}+m-i,l\times k} \vee \neg \mathrm{G}'_{mkl} \tag{3.11}$$

We discuss these clauses in the next section.

3.4 Revisiting the SAT formulation by Gent and Lynce

In this section, we revisit the SAT formulation as proposed by Gent and Lynce and correct some of its clauses, to enforce all desired constraints and thus correctly model SGP instances as SAT instances.

Clauses (3.1) and (3.2) of the SAT formulation by Gent and Lynce are correct. Clause set (3.3) is incomplete: n must range from **1** to p, instead of from $j+1$ to p. This is because a player that plays in week l at position j of group k must not play in *any* other position for further groups of that week l, not just positions *greater* than j. The correct encoding is thus:

$$\bigwedge_{i=1}^{x} \bigwedge_{l=1}^{w} \bigwedge_{j=1}^{p} \bigwedge_{k=1}^{g} \bigwedge_{m=k+1}^{g} \bigwedge_{n=1}^{p} \neg \mathrm{G}_{ijkl} \vee \neg \mathrm{G}_{inml} \tag{3.12}$$

Clause set (3.5) is slightly inaccurate: G_{imkl} must be changed to $\mathrm{G}_{\mathbf{mj}kl}$ to enforce the correct constraint, yielding:

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{j=1}^{p} \bigwedge_{i=1}^{x} \bigwedge_{m=i+1}^{x} \neg \mathrm{G}_{ijkl} \vee \neg \mathrm{G}_{mjkl} \tag{3.13}$$

This is because the other order of these indices would not match the intended usage of these variables. Interestingly, this typographical error could have been found automatically, via a simple reasoning about the known bounds of the variables' indices.

Clause set (3.7) is also slightly inaccurate: y must range from 1 to $\binom{x}{2}$. This is because the constraint must hold for *all* pairs of players, not just every one except the last one, and yields:

$$\bigwedge_{y=1}^{\binom{x}{2}} \bigwedge_{z=1}^{g\times w} \neg\text{LADDER}_{yz+1} \vee \text{LADDER}_{yz} \tag{3.14}$$

It also becomes clear from this clause set that the ladder matrix is in fact a $\binom{x}{2} \times (g \times w + \mathbf{1})$ matrix.

Further, it is easy to see that in the clauses of (3.10), $\neg G'_{ikl}$ must be replaced by its negation, G'_{ikl}. Similarly, $\neg G'_{mkl}$ must be replaced by its negation in the clauses of (3.11), yielding:

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \text{LADDER}_{\binom{x-i}{2}+m-i,l\times k+1} \vee \neg\text{LADDER}_{\binom{x-i}{2}+m-i,l\times k} \vee G'_{ikl} \tag{3.15}$$

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \text{LADDER}_{\binom{x-i}{2}+m-i,l\times k+1} \vee \neg\text{LADDER}_{\binom{x-i}{2}+m-i,l\times k} \vee G'_{mkl} \tag{3.16}$$

This is due to the intention that if adjacent cells of the ladder matrix are TRUE and FALSE in some row, the two players corresponding to that row *should* play together.

However, this is not all that needs to be changed to correctly model the SGP. Consider the "solution" of the SGP instance 8−4−2 shown in Fig. 3.3, in which conflict positions are highlighted. The configuration satisfies all constraints corrected so far, yet still contains conflicts.

	Week 1				Week 2			
Group 1	32	24	16	8	31	23	15	7
Group 2	31	23	15	7	29	21	13	5
Group 3	30	22	14	6	27	19	11	3
Group 4	29	21	13	5	25	17	9	1
Group 5	28	20	12	4	26	22	16	12
Group 6	27	19	11	3	30	18	8	4
Group 7	26	18	10	2	32	28	14	10
Group 8	25	17	9	1	24	20	6	2

Figure 3.3: A "solution" for the 8−4−2 instance, conflicts highlighted

The location of conflict positions in Fig. 3.3 can give valuable indications as to where the model went wrong. In this case, we expect the encoding of *socialisation* to be wrong, since all other constraints are satisfied: Each player plays exactly once each week, and all groups have the correct size. The interesting pattern of conflict positions is due to the following problem in the model: In clause sets (3.8)–(3.11), columns of the ladder matrix are referenced by the terms $(l \times k)$ and $(l \times k + 1)$, with $1 \leq l \leq w$ and

$1 \leq k \leq g$. Clearly, each group should be assigned a distinct column in the ladder matrix, but the way in which the running variables are combined to form a column index does not guarantee that. For example, both the second group of the first week, and the first group of the second week will evaluate to column $1 \times 2 = 2 \times 1 = 2$ of the ladder matrix. As a consequence, the relevant constraints are not imposed on all groups.

To remedy this, we propose that the column index of group k in week l be uniquely determined by the expression $(l-1) \times g + k$. This yields the following revised versions of clause sets (3.8)–(3.11):

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \neg \mathrm{G}'_{ikl} \vee \neg \mathrm{G}'_{mkl} \vee \textsc{Ladder}_{\binom{x-i}{2}+m-i,(l-1)\times g+k} \tag{3.17}$$

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \neg \mathrm{G}'_{ikl} \vee \neg \mathrm{G}'_{mkl} \vee \neg\textsc{Ladder}_{\binom{x-i}{2}+m-i,(l-1)\times g+k+1} \tag{3.18}$$

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \textsc{Ladder}_{\binom{x-i}{2}+m-i,(l-1)\times g+k+1} \vee \neg\textsc{Ladder}_{\binom{x-i}{2}+m-i,(l-1)\times g+k} \vee \mathrm{G}'_{ikl} \tag{3.19}$$

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{i=1}^{x-1} \bigwedge_{m=i+1}^{x} \textsc{Ladder}_{\binom{x-i}{2}+m-i,(l-1)\times g+k+1} \vee \neg\textsc{Ladder}_{\binom{x-i}{2}+m-i,(l-1)\times g+k} \vee \mathrm{G}'_{mkl} \tag{3.20}$$

This completes the corrected formulation.

3.5 Improving the SAT formulation by Gent and Lynce

Having presented a correct SAT formulation for the SGP, we now look for ways to reduce computation time when solving SGP instances in practice. There are various ways to do this, for example, by:

- reducing the number of variables
  This reduces the number of possible assignments and thus leads to a smaller search space.
- reducing the number of clauses
  Fewer clauses means fewer constraints that need to be satisfied, in effect turning more assignments into satisfying ones.

- imposing symmetry breaking constraints
  This removes uninteresting (isomorphic) branches of the search tree and can help a SAT solver to focus on more interesting parts of the search space.

We consider symmetry breaking constraints in Section 3.7, and focus for now on the first two options.

Our first observation regards the number of clauses: The revised versions of clause sets (3.10) and (3.11), namely (3.19) and (3.20), are actually not necessary: They can be omitted from the model without affecting the correctness of the formulation. The sole purpose of the ladder matrix is to reflect the fact that two players play in the same group. This means that *if* two players play in the same group, we want the ladder matrix to reflect that. If, on the other hand, a TRUE and FALSE value in the ladder matrix are adjacent, then the corresponding players need *not* necessarily play together in that group. This is due to the fact that our focus is on the players, and the ladder matrix is only an auxiliary device that we use for the specific purpose of enforcing the socialisation constraint. Other than that, the ladder matrix is of no interest, and remaining cells can assume any values. A similar observation holds for equivalence (3.6), where only the right-to-left implication is of interest, and auxiliary variables can otherwise assume any values. Reducing the number of clauses in this way can result in significantly less computation time, but it can also have adverse effects depending on the method used to solve an instance, and we therefore keep equivalence (3.6).

To reduce the number of variables for each instance, we eliminate the ladder matrix entirely. Instead of equations (3.7)–(3.11), we propose a different way to encode the desired socialisation constraint that lies at the core of the SGP. The set of clauses we propose can be concisely described as:

$$\bigwedge_{l=1}^{w} \bigwedge_{k=1}^{g} \bigwedge_{m=1}^{x} \bigwedge_{n=m+1}^{x} \bigwedge_{k'=1}^{g} \bigwedge_{l'=l+1}^{w} (\neg \mathrm{G}'_{mkl} \vee \neg \mathrm{G}'_{nkl}) \vee (\neg \mathrm{G}'_{mk'l'} \vee \neg \mathrm{G}'_{nk'l'}) \quad (3.21)$$

This states the socialisation constraint in a very straight-forward manner: If two players m and n play in the same group k of a week l, then they cannot play together in any group of further weeks.

This change of the formulation makes all $\textsc{Ladder}_{xy}$ variables unnecessary. For each SGP instance g–p–w, our formulation will therefore always have exactly $\binom{g \times p}{2} \times (g \times w + 1)$ fewer variables (i.e., exactly the number of ladder variables) than the formulation proposed by Gent and Lynce. The number of clauses can be less, equal, or more depending on the instance. In the next section, we assess empirically what can be gained by our change of the formulation.

3.6 Experimental results

We chose the two well-known instances $5-3-w$ and $8-4-w$ for benchmarking, which we also consider quite representative for other instances.

All experiments were conducted on an Apple MacBook with a 2.16 GHz Intel Core 2 Duo CPU and 1GB RAM. We used the two SAT solvers Walksat 4.6 ([SKC93]) and SATO 4.2 ([Zha97]). Walksat uses local search, and SATO uses the Davis-Putnam method. For SATO, we started the solver and waited with a timeout of 20 minutes. For Walksat, we tried many options and chose the cutoff in such a way that the program had about 20 minutes to solve an instance. This is a reasonable time frame to be competitive, as this is also the limit that was chosen in recently reported other approaches, such as in [CDFH06].

Tables 3.1 and 3.2 show benchmark results with the (revised) SAT formulation by Gent and Lynce and our formulation, respectively. For each SGP instance, we show the number of variables and clauses of the generated SAT instance. The "Walksat" column shows the average number of seconds until a satisfying assignment was found in 10 tries. The "SATO" column shows the number of seconds until a single solution was found, averaged over 10 runs to reduce variance. The symbol "–" means that no solution was found within the time limit.

Despite trying many different options with Walksat, we did not find many solutions using the (revised) formulation by Gent and Lynce. In contrast, we could solve all given instances with Walksat's "novelty" option and a cutoff parameter of 10^7 by using our formulation.

It is clear from these figures that our formulation can result in large performance improvements when solving SGP instances in practice. A SAT-based approach towards the SGP thus seems now at least more promising than previously, when results were much further from being competitive with other approaches.

3.7 Symmetry breaking

Another strategy that can reduce computation time when solving SAT instances in practice is to eliminate uninteresting branches of the search tree by breaking some of the symmetries inherent to the underlying problem. Clearly, the SGP is a highly symmetric problem: Players within groups, groups within weeks, and weeks themselves can be reordered arbitrarily. Yet, schedules with different orders of players within the same group lead to distinct solutions in the SAT formulation above. Especially when using SAT solvers that use a complete backtracking algorithm, it can help to eliminate these symmetries and thus force the solver away from regions of the search space that have already been considered in some variant. In [GL05], Gent and Lynce propose the following set of clauses to break the symmetry among

instance	#Vs.	#Cl.	Walksat	SATO
5−3−1	930	6105	0.00s	0.01s
5−3−2	1755	12210	0.02s	0.02s
5−3−3	2580	18315	–	–
5−3−4	3405	24420	–	–
5−3−5	4230	30525	–	–
5−3−6	5055	36630	–	–
8−4−1	5744	52928	–	0.06s
8−4−2	10992	105856	–	0.11s
8−4−3	16240	158784	–	–
8−4−4	21488	211712	–	–
8−4−5	26736	264640	–	–
8−4−6	31984	317568	–	–

Table 3.1: Revised formulation by Gent and Lynce

instance	#Vs.	#Cl.	Walksat	SATO
5−3−1	300	3480	0.00s	0.12s
5−3−2	600	9585	0.00s	0.01s
5−3−3	900	18315	0.00s	0.01s
5−3−4	1200	29670	0.01s	0.03s
5−3−5	1500	43650	2.42s	0.04s
5−3−6	1800	60255	98.94s	–
8−4−1	1280	33088	0.00s	0.03s
8−4−2	2560	97920	0.01s	0.10s
8−4−3	3840	194496	0.05s	1.16s
8−4−4	5120	322816	0.75s	–
8−4−5	6400	482880	0.98s	–
8−4−6	7680	674688	198.92s	–

Table 3.2: Our formulation

players within each group:

$$\bigwedge_{i=1}^{x} \bigwedge_{j=1}^{p} \bigwedge_{k=1}^{g} \bigwedge_{l=1}^{w} \bigwedge_{m=1}^{i-1} \neg \mathrm{G}_{ijkl} \vee \neg \mathrm{G}_{m(j+1)kl} \tag{3.22}$$

Clearly, j must in fact range from 1 to $p - \mathbf{1}$. Also, since the players within each week must be distinct, m can range from 1 to $\mathbf{i}$. With these modifications, the clause set (3.22) ensures that the players within each group are in strictly increasing numerical order.

All groups within a single week can be ordered by their first players, for which Gent and Lynce propose the set of clauses:

$$\bigwedge_{i=1}^{x} \bigwedge_{k=1}^{g} \bigwedge_{l=1}^{w} \bigwedge_{m=1}^{i-1} \neg \mathrm{G}_{i1kl} \vee \neg \mathrm{G}_{m1(k+1)l} \tag{3.23}$$

Again, k must actually range from 1 to $g - \mathbf{1}$, and m can range up to i. From (3.22) and (3.23), it follows that player 1 is the first player of the first group in each week. Using this fact, weeks can be ordered lexicographically by the *second* golfer playing in the first group of each week. Gent and Lynce propose the set of clauses:

$$\bigwedge_{i=1}^{x} \bigwedge_{k=1}^{g} \bigwedge_{l=1}^{w} \bigwedge_{m=1}^{i-1} \neg \mathrm{G}_{i2kl} \vee \neg \mathrm{G}_{m2k(l+1)} \tag{3.24}$$

Again, l must actually range from 1 to $w - \mathbf{1}$. Also, this set of clauses removes more solutions than intended, as it erroneously enforces the constraint for *each* group, instead of only the first group of each week. In addition, we know that the second player in the first group must be distinct for each week, since player 1 also plays in this week, and players cannot meet more than once. We can therefore demand *strictly* ascending second players in the first group of each week, by using instead the following set of clauses:

$$\bigwedge_{i=1}^{x} \bigwedge_{l=1}^{w-1} \bigwedge_{m=1}^{i} \neg \mathrm{G}_{i21l} \vee \neg \mathrm{G}_{m21(l+1)} \tag{3.25}$$

This completes the symmetry breaking constraints. We will encounter them again when discussing a constraint-based approach in Section 4.8.2.

3.8 Experimental results again

It is well-known that symmetry breaking constraints can make it harder for local search methods to find solutions (see [Pre01]), as they reduce the number of solutions and thus the solution density. In contrast, solution methods that involve backtracking search can greatly benefit from symmetry

instance	#Vs.	#Cl.	Walksat	SATO
5−3−6	1800	70935	–	1m25.56s
5−3−7	2100	87885	–	3m5.79s
8−4−4	5120	389872	–	1.58s
8−4−5	6400	566832	–	3.64s
8−4−6	7680	775536	–	12.83s
8−4−7	8690	1015984	–	4m3.85s

Table 3.3: Our formulation, including symmetry breaking constraints

breaking constraints, and we thus expect a positive impact when using these constraints together with the SATO solver.

We added the three symmetry breaking constraints of Section 3.7 to the model we propose in Section 3.5, in the hope to further reduce computation time and solve SGP instances that we previously could not solve within the time limit.

Table 3.3 shows our results for those instances that could not be solved previously. As can be seen, these results largely confirm our expectation: Adding symmetry breaking clauses has a very positive effect for the backtracking solver SATO, with which we can now solve many more instances. Kirkman's schoolgirl problem, instance 5−3−7, can now be solved, and the original SGP can be solved for up to 7 weeks.

When trying to solve instance 5−3−7 with all symmetry breaking constraints, SATO exited with the message "Bus error". We then removed the last two constraints, and broke only the symmetry arising from different player orders within groups to solve this instance. All other instances could be solved with all three symmetry breaking constraints.

As expected, adding symmetry breaking constraints has very adverse effects for Walksat, which is based on local search. None of the instances could be solved within the time limit, not even those that could be solved before adding symmetry breaking clauses.

3.9 More symmetry breaking

The symmetry breaking constraints of Section 3.7 do not break all symmetries of solutions: In addition to the mentioned symmetries, players can be renumbered arbitrarily. Thus, different permutations can still yield distinct but isomorphic solutions. See [BB05] for an example.

In [FHK+02], a very different model for the SGP is proposed, which allows to break much of this symmetry: Let $M_{w\times g, g\times p}$ denote a matrix of Boolean variables. Each column corresponds to a player, and each row corresponds to one group. The Boolean value at position (i, j) denotes whether player j plays in group i. The necessary constraints on M are quite obvious. Much of the symmetry arising from player permutations

can now be broken by imposing a lexicographic “less than” constraint on columns. Similarly, the symmetry among groups can be broken by imposing a lexicographic “less than” constraint on the rows corresponding to the groups of each week. Unfortunately, this still does not remove all isomorphic solutions: As shown in [FFH+02], lexicographically ordering both rows and columns does not break all compositions of row and column symmetries. As this model was also shown to suffer from overheads due to large arising vectors in [FHK+02], we do not consider it further.

It is always possible to break all symmetries of a matrix of decision variables with lexicographic constraints ([CGLR96]). However, since it is in general necessary to add a super-exponential number of constraints, this is often infeasible in practice.

3.10 Working with SAT instances

As demonstrated by the ample need for revision of previous models in this chapter, working with SAT formulations is quite error-prone. This is because the language of propositional logic is comparatively “low-level”: A very limited number of primitive operations must be used to make all desired constraints explicit.

As part of this thesis, we have implemented a domain-specific language that makes working with SAT instances more convenient and less error-prone. SAT instances in DIMACS format can be generated from a much higher-level language that can express ranged disjunctions and conjunctions, and symbolic variables with indices formed by arithmetic expressions. Perhaps most importantly, our tools make it easy to extract all variables that are assigned the value TRUE in an assignment, and to project them back into their symbolic forms. From this symbolic form, figures are readily generated, and assignments can be verified easily.

For example, consider the Walksat output after solving instance 2−2−3, shown in Fig. 3.4. Using our tools, this satisfying assignment for a previously generated SAT instance can be projected back to the symbolic variables of the original formulation. Fig. 3.5 shows the variables that are assigned the value TRUE in the assignment found by Walksat. From these variables, we can easily generate Fig. 3.6, which in turn is readily seen to be a conflict-free schedule for this instance.

Our approach is not limited to satisfying assignments. We can advise Walksat to emit the current assignment every N steps and thus observe the solution process arbitrarily closely. Fig. 3.7 shows the current assignment after every 3000 steps, as Walksat solves instance 5−3−5. From such animations, one can see which clauses are satisfied first, and which are only satisfied towards the end. One can also compare the effects of different solver options and problem formulations, and easily verify a solution’s correctness.

Finally, Fig. 3.8 shows the complete specification of the SGP as pro-

posed in this chapter (including symmetry breaking constraints), using our domain-specific language. The syntax should be clear after comparing it to the clause sets as they appear in this chapter, and we do not discuss it here further. With the necessary additional (Prolog) definitions, which are freely available from the author, this specification can be used to produce SAT instances in DIMACS format.

3.11 Conclusion

The existence of freely available and continuously evolving SAT solvers makes it attractive to try out SAT formulations for combinatorial problems such as the SGP. In this chapter, we discussed an existing SAT formulation for the SGP and revised some of its clauses. It is well known that the choice of encoding can have a significant impact on performance, and we saw this effect for a different encoding of the SGP that we proposed in this chapter. Using our encoding can considerably reduce computation time when solving SGP instances with common SAT solvers in practice.

Working with SAT instances is quite error-prone, as the ample need for revision of an existing SAT encoding has shown. However, the tools we developed as part of this thesis make working with SAT instances more convenient. In particular, they make it easy to generate SAT instances in DIMACS format from a higher-level language. After solving these instances with a SAT solver, True variables can be projected back into their symbolic forms. This makes it easier to verify results, and to generate customised visualisations of solutions, which we highly recommend. These tools are freely available from the author.

By adding symmetry breaking constraints, we solved Kirkman's schoolgirl problem, and the original SGP instance for up to 7 weeks with the freely available complete SAT solver SATO. While this is not yet competitive with other approaches that we discuss in this thesis, a SAT-based approach for solving SGP instances is now at least more promising than previously, when results were much further from being competitive.

```
1  Begin assign with lowest # bad = 0
2   -1 -2 -3 4 -5 6 -7 -8 -9 -10
3   -11 12 -13 14 -15 -16 17 -18 -19 -20
4   -21 22 -23 -24 25 -26 -27 -28 -29 -30
5   -31 32 -33 -34 35 -36 -37 -38 39 -40
6   -41 -42 43 -44 45 -46 -47 -48 -49 50
7   51 -52 -53 54 55 -56 57 -58 59 -60
8   61 -62 -63 64 -65 66 -67 68 -69 70
9   71 -72
10 End assign
```

Figure 3.4: Walksat output after solving instance 2−2−3

```
1 g(1, 2, 1, 1), g(1, 2, 1, 2), g(1, 1, 2, 3),
2 g(2, 2, 2, 1), g(2, 1, 1, 2), g(2, 1, 1, 3),
3 g(3, 1, 1, 1), g(3, 1, 2, 2), g(3, 2, 1, 3),
4 g(4, 1, 2, 1), g(4, 2, 2, 2), g(4, 2, 2, 3),
5 g(1, 1, 1), g(1, 1, 2), g(1, 2, 3),
6 g(2, 2, 1), g(2, 1, 2), g(2, 1, 3),
7 g(3, 1, 1), g(3, 2, 2), g(3, 1, 3),
8 g(4, 2, 1), g(4, 2, 2), g(4, 2, 3)
```

Figure 3.5: Turning TRUE variables of Fig. 3.4 into their symbolic forms

	Week 1		**Week 2**		**Week 3**	
Group 1	3	1	2	1	2	3
Group 2	4	2	3	4	1	4

Figure 3.6: Visualising the solution of Fig. 3.4 and Fig. 3.5

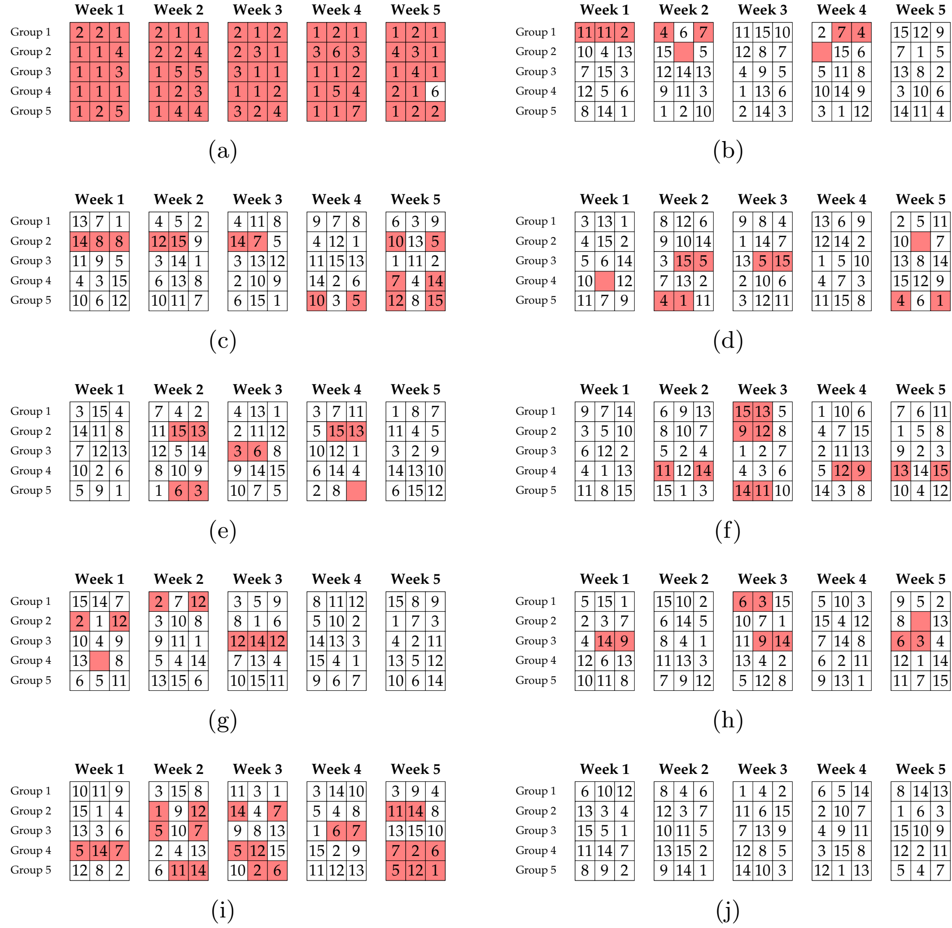


Figure 3.7: Instance 5−3−5, configuration after (a) 0, (b) 3000, (c) 6000, (d) 9000, (e) 12000, (f) 15000, (g) 18000, (h) 21000, (i) 24000, (j) 25539 flips using Walksat, conflicts highlighted

```
golf(G, P, W) :-
        make_lookup(G, P, W, L),
        assoc_to_list(L, Ls),
        length(Ls, Order),
        format("p cnf ~w \n", [Order]),
        X is P * G,
        emit(^(i = 1 to X,
               ^(l = 1 to W,
                 v(j = 1 to P,
                   v(k = 1 to G,
                     g(i,j,k,l))))), L),
        emit(^(i = 1 to X,
               ^(l = 1 to W,
                 ^(j = 1 to P,
                   ^(k = 1 to G,
                     ^(m = j+1 to P,
                       g(i,j,k,l) => \g(i,m,k,l)))))), L),
        emit(^(i = 1 to X,
               ^(l = 1 to W,
                 ^(j = 1 to P,
                   ^(k = 1 to G,
                     ^(m = k+1 to G,
                       ^(n = 1 to P,
                         g(i,j,k,l) => \g(i,n,m,l))))))), L),
        emit(^(l = 1 to W,
               ^(k = 1 to G,
                 ^(j = 1 to P,
                   v(i = 1 to X,
                     g(i,j,k,l))))), L),
        emit(^(l = 1 to W,
               ^(k = 1 to G,
                 ^(j = 1 to P,
                   ^(i = 1 to X,
                     ^(m = i+1 to X,
                       g(i,j,k,l) => \g(m,j,k,l)))))), L),
        emit(^(l = 1 to W,
               ^(k = 1 to G,
                 ^(j = 1 to P,
                   ^(i = 1 to X,
                     g(i,j,k,l) => g(i,k,l))))), L),
        emit(^(l = 1 to W,
               ^(k = 1 to G,
                 ^(i = 1 to X,
                   g(i,k,l) => v(j = 1 to P,
                                 g(i,j,k,l))))), L),
        emit(^(l = 1 to W,
               ^(k = 1 to G,
                 ^(m = 1 to X,
                   ^(n = m+1 to X,
                     ^(kp = 1 to G,
                       ^(lp = l+1 to W,
                         g(m,k,l) ^ g(n,k,l) =>
                           \ (g(m,kp,lp) ^ g(n,kp,lp)))))))), L),
        emit(^(i = 1 to X,
               ^(j = 1 to P-1,
                 ^(k = 1 to G,
                   ^(l = 1 to W,
                     ^(m = 1 to i,
                       g(i,j,k,l) => \g(m,j+1,k,l)))))), L),
        emit(^(i = 1 to X,
               ^(k = 1 to G-1,
                 ^(l = 1 to W,
                   ^(m = 1 to i,
                     g(i,1,k,l) => \g(m,1,k+1,l))))), L),
        emit(^(i = 1 to X,
               ^(l = 1 to W-1,
                 ^(m = 1 to i,
                   g(i,2,1,l) => \g(m,2,1,l+1)))), L).
```

Figure 3.8: A SAT formulation for the SGP in our domain-specific language

4 Constraint programming formulations

4.1 Introduction

Constraint programming (CP) is a declarative formalism that lets users specify conditions a solution should satisfy. Based on that description, a constraint *solver* can then search for solutions.

In this chapter, we discuss two existing constraint-based formulations of the SGP, both of which can solve the original SGP for a maximum of 9 weeks. One of the formulations is written by Stefano Novello, using the free CP platform ECLiPSe ([WNS97]). The other formulation is written by Mats Carlsson and uses the commercial Prolog system SICStus Prolog. We build a new finite domain constraint solver out of the desire to run Carlsson's formulation in a free environment. Guided by animations of the constraint solving process, we are able to solve instance $8-4-9$ with our solver as well. We also present benchmark results for several other instances.

4.2 Constraint logic programming

The first ideas for CP date back to the sixties and seventies ([Sut63]), with the *scene labelling* problem ([Wal75]) being one of the first constraint satisfaction problems (CSPs) that were formalised. A CSP consists of:

- a set X of variables, $X = \{x_1, \ldots, x_n\}$
- for each variable x_i, a set $D(x_i)$ of values that x_i can assume, which is called the *domain* of x_i
- a set of *constraints*, which are simply relations among variables in X, and which can further restrict their domains

One key observation, made by Jaffar, Lassez ([JL87]), Gallaire ([Gal85]) and others, was the insight that pure *logic programming* (LP) can be regarded as an instance of constraint solving, namely as solving constraints over variables whose domains are Herbrand terms. In addition, LP and CP share an important intention, which is to make users less concerned about *how* a problem should be solved, and instead let them focus on a clear description of *what* should be solved. From that description, a logic engine or constraint solver can, in principle, compute a solution without additional instructions. Therefore, logic programming languages like Prolog have become the most important host platforms for constraint solvers, and most Prolog implementations nowadays ship with several libraries for constraint programming. When CP is used with a logic programming language as its host, it is referred to as constraint *logic* programming (CLP). However, constraint programming is not restricted to CLP: It is possible to embed constraint solvers in other host languages, even if they might not blend in as seamlessly as they do with Prolog.

4.3 CLP(FD)

In connection with combinatorial optimisation or completion problems such as the SGP, one of the most frequently used instances of constraint programming is constraint logic programming over finite domains, denoted as CLP(FD). This means that all domains are finite sets of *integers*, and the available constraints include at least the common arithmetic relations between integer expressions.

One advantage when reasoning over integers is that many known laws of arithmetic can be used to further reduce the domains of variables that participate in the provided relations. Another advantage is that there is a predefined total order over the integers, which can often help to eliminate uninteresting symmetries between solutions and reduce the search space.

CLP(FD) can also help to solve problems over rational numbers. The following example is known as the "7-11 problem" ([PG83]):

Example 4.1. The total price of 4 items is €7.11. The product of their prices is €7.11 as well. What are the prices of the 4 items?
Answer. The prices are €3.16, €1.50, €1.25 and €1.20. A CLP(FD) solution for this problem is shown in Fig. 4.1. Line 1 states the domain of all variables, lines 2 and 3 post the two known constraints. The product of all variables equals a quite large constant, which many constraint solvers have problems with. In fact, the problem is already beyond the current capabilities of the SICStus CLP(FD) solver on still common 32-bit platforms, while our solver can easily cope with it. Line 4 imposes additional constraints to break symmetries between values. Finally, line 5 searches for valid ground instantiations of all variables, using a strategy which we explain in Section 4.6. Notice that other constraints could be imposed as well. For example, due to the fundamental theorem of arithmetic, the factorisation of one of the variables must contain one of the prime factors of 711×100^3. However, imposing such a constraint would in general also require weakening the ordering relation, and it is not a priori clear which of the formulations is better.

```
1  ?- Vs = [A,B,C,D], Vs ins 0..711,
2     A * B * C * D #= 711*100^3,
3     A + B + C + D #= 711,
4     A #>= B, B #>= C, C #>= D,
5     labeling([ff], Vs).
```

Figure 4.1: Solving the 7-11 problem with a single query

4.4 Example: Sudoku

In the recent past, a combinatorial number puzzle called *Sudoku* has attracted significant attention. A Sudoku Latin square is a particular kind of

```
1  sudoku(Rows) :-
2          length(Rows, 9), maplist(length_(9), Rows),
3          append(Rows, Vs), Vs ins 1..9,
4          maplist(all_different, Rows),
5          transpose(Rows, Columns), maplist(all_different, Columns),
6          Rows = [A,B,C,D,E,F,G,H,I],
7          blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).
8
9  length_(N, Ls) :- length(Ls, N).
10
11 blocks([], [], []).
12 blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
13         all_different([A,B,C,D,E,F,G,H,I]),
14         blocks(Bs1, Bs2, Bs3).
```

Figure 4.2: A CLP(FD) description of Sudoku Latin squares

Latin square ([CD96], see also Section 2.8):

Definition 4.1. Let a, b and n be positive integers with $a \times b = n$. Partition an $n \times n$ array into $a \times b$ rectangles. An (a, b)-*Sudoku Latin square* is a Latin square on the symbol set $\{1, \dots, n\}$ where each (a, b)-rectangle contains all symbols. A *Sudoku Latin square* is a $(3, 3)$-Sudoku Latin square.

Definition 4.2. An (a, b)-*Sudoku critical set* is a partial Latin square P that is completable in exactly one way to an (a, b)-Sudoku Latin square, and removal of any of the filled cells from P destroys the uniqueness of completion.

Fig. 4.2 shows a CLP(FD) formulation for Sudoku Latin squares. Here, a Sudoku Latin square is modelled as a list of rows, with each row being a list of variables with domain $\{1, \dots, 9\}$. Line 2 ensures the correct list structure, which makes it possible to use the predicate in all directions: One can use the specification to test and complete partially filled squares as well as to enumerate all possible squares. The only constraint used in this formulation is the built-in constraint `all_different`, which imposes pairwise inequalities between all variables occurring in a list. The constraint is imposed for each row (line 4), column (line 5), and 3×3-subsquare (lines 6, 7 and 11–14).

A valid Sudoku puzzle as commonly found in contemporary newspapers and periodicals is a partial Latin square that is completable in exactly one way to a Sudoku Latin square. Fig. 4.3 (a) shows an example of a valid Sudoku puzzle, which is simultaneously a $(3, 3)$-Sudoku critical set. In fact, the figure shows one of the "hardest" currently known Sudoku puzzles: No $(3, 3)$-Sudoku critical set with fewer than 17 given numbers is currently known ([CD96]). Fig. 4.3 (b) shows the Sudoku Latin square that is uniquely determined by this Sudoku critical set.

4.5 Constraint propagation and search

An element v of a domain $D(x)$ is said to be *inconsistent* with respect to a given CSP if there is no solution in which x assumes the value v. A CSP

1								
		2	7	4				
			5					4
	3							
7	5							
					9	6		
	4				6			
							7	1
					1		3	

(a)

1	8	4	9	6	3	7	2	5
5	6	2	7	4	8	3	1	9
3	9	7	5	1	2	8	6	4
2	3	9	6	5	7	1	4	8
7	5	6	1	8	4	2	9	3
4	1	8	2	3	9	6	5	7
9	4	1	3	7	6	5	8	2
6	2	3	8	9	5	4	7	1
8	7	5	4	2	1	9	3	6

(b)

Figure 4.3: (a) A $(3, 3)$-Sudoku critical set, and (b) the induced Sudoku Latin square

is said to be *globally consistent* if none of its domains contains an inconsistent element. Guaranteeing global consistency is computationally infeasible in general, and constraint solvers therefore rely on *local* consistency techniques that are computationally less expensive at the cost of not necessarily reaching a globally consistent state.

Consistency techniques were introduced in [Wal75] and are derived from graph notions (see [Bar99]). The process of deterministically ensuring some form of consistency is called constraint *propagation*. Since propagation alone is typically insufficient to reduce all domains to singleton sets and thus produce concrete solutions, some form of *search* is necessary as well. Systematically trying out values for variables is called *labeling*, and we discuss it in the next section. As soon as a variable is labeled, constraint propagation is used to further prune the search space. Conversely, propagation can in itself yield a singleton set for a variable's domain, thus causing the variable to be instantiated to a ground value. Search and propagation are therefore interleaved when solving a CSP. Clearly, a trade-off must be reached between strong propagation, implying great reduction of the search space for some problems, and computational tractability.

As an example for different consistency notions, consider again Sudoku puzzles. In this case, the search space is often quite large when traversed naively. However, a constraint solver is typically able to delete many values from the domains of those variables that correspond to free cells before the search even begins. To give a visual impression of the values that can be removed from domains, we proceed as follows: First, we subdivide all free cells into 9 small regions as shown in Fig. 4.4. Each region corresponds to the domain element that it contains in this figure. Then, a dot is drawn in those regions that correspond to domain elements which can be excluded

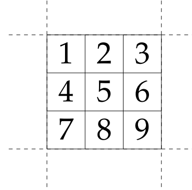
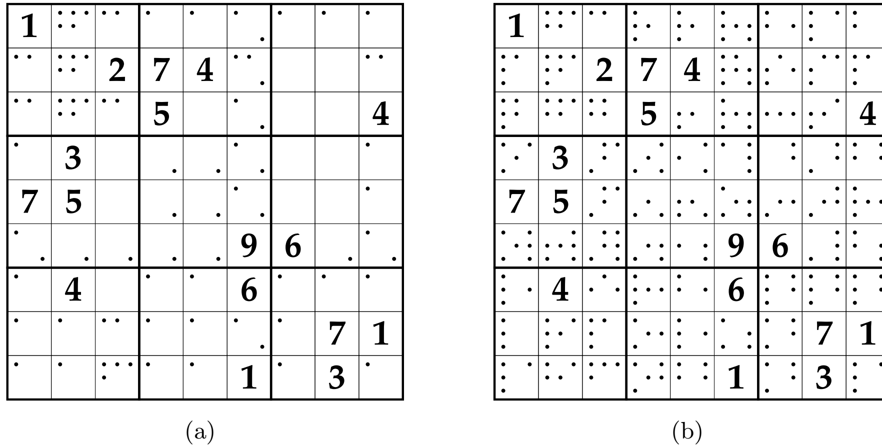


Figure 4.4: Subdivision of a single cell



(a)

(b)

Figure 4.5: Domain elements that can be removed after posting the Sudoku puzzle with (a) a bounds consistent constraint solver and (b) our solver

due to the given constraints. Fig. 4.5 shows which values can be excluded by two different constraint solvers without performing any search. Fig. 4.5 (a) was created with a bounds consistent solver, and Fig. 4.5 (b) was created with our solver, which is arc consistent. A solver with perfect propagation would reduce all domains to singleton sets in this case, making further search unnecessary.

4.6 Selection strategies for variables and values

When searching for solutions of a CSP by trying ground values for variables, there are at least two degrees of freedom: First, the instantiation order of variables. Second, the order in which values are tried for each variable. Choosing good orders can significantly reduce computation time.

We first discuss the impact of variable instantiation orders. Fig. 4.6 depicts two possible search tree shapes arising from complete enumerations of two unconstrained variables, X and Y, with domains of size 2 and 5, respectively. The order or type of actual values that are tried for each variable is currently of no concern, as we focus on the order in which the variables themselves are instantiated. Inner nodes of the search tree, which

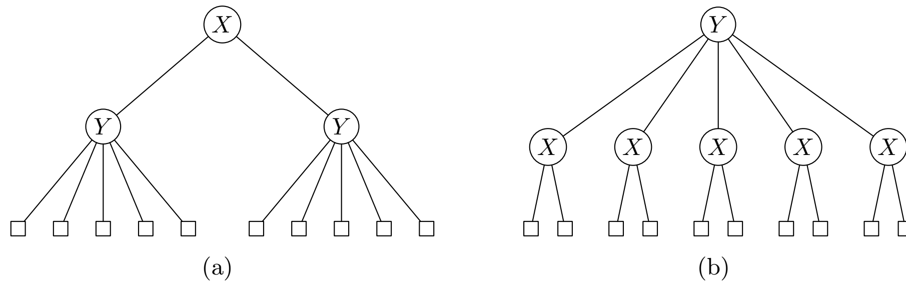


Figure 4.6: Search tree shapes arising from different instantiation orders

are the variables, are shown as circles, and leaves are shown as boxes. When a leaf is reached in the search process, all variables are instantiated. Clearly, the number of leaves must be the same for all possible shapes of the search tree, while the number of inner nodes can obviously differ significantly.

In typical CSPs, many values can turn out to be infeasible. In fact, a significant number of subtrees of the search tree will often turn out to be of no interest at all. We expect the greatest reduction of inner nodes that must still be visited by first trying to instantiate the variable with the *fewest* domain elements left. The strategy of instantiating the variables in order of increasing size of domains is called "first-fail", and often performs very well in practice. The intention here is twofold: First, variables with small domains are likely to run out of domain elements, causing their instantiation to fail. Clearly, it is advantageous to detect inevitable failure as early as possible. Second, instantiating variables can only further constrain the domains of remaining variables. Therefore, we want to instantiate variables with small domains while that is still possible, since the situation can only become worse for them. For a probabilistic analysis of the impact of this strategy, see [HE80].

Constraint solvers typically provide several pre-defined variable selection strategies that users can choose from, and which can influence computation time considerably. Our solver provides the following strategies to instantiate a list of variables (ties are broken by selecting the leftmost variable in the list), which are also available in most other constraint solvers:

- **leftmost**
  Instantiate the variables from left to right in the order they occur in the given list.

- **ff** ("first-fail")
  Instantiate the variable with smallest domain next.

- **ffc**
  Of the variables having smallest domains, the one involved in most constraints is instantiated next.

- **min**
  Instantiate the variable whose lower bound is the lowest next.

- **max**
  Instantiate the variable whose upper bound is the highest next.

For most of these options, it is important to accurately assess a variable's current domain, and thus solvers with different propagation strengths can lead to very different instantiation orders of variables. Somewhat counter-intuitively, stronger propagation can even have an adverse effect in this case. This was first pointed out in [SF94] and can be explained by the fact that stronger propagation can also lead an instantiation strategy *away* from a "good" ordering, since propagation affects the variables' domains and thus the selected variable for many of these options.

After having selected a variable x for instantiation, a constraint solver must choose a value from $D(x)$ that should be assigned to x. A good strategy is often to instantiate x to a value of its domain which constrains the remaining variables *the least*. However, determining which of the values have this property can be costly, and many constraint solvers therefore do not provide this option. Our solver only provides two value selection strategies:

- **up**
  The values of each domain are tried in ascending order.

- **down**
  The values are tried in descending order.

In addition to these pre-defined selection strategies and value ordering options, users are free to implement their own allocation strategies. We regard this as one of the great advantages of constraint-based approaches over other methods: Once all constraints are stated, variables can be instantiated in any order and to any values, and infeasible choices are automatically rejected.

4.7 Visualising the constraint solving process

In many cases, it is very interesting to visualise the constraint solving process graphically. At the very least, one can get an impression of how the search progresses. Based on that observation, one can then try different allocation strategies, which sometimes work much better than others.

Transparent constraint animations have not received much attention in the literature so far: In [NRS97], Neumerkel et al. explain the importance of visualisations in the context of GUPU, a teaching environment for Prolog. However, they do not mention the potential usefulness of visualisations for deducing alternative strategies. Fages et al. present a graphical

user interface for CLP in [FSC04]. Their approach typically requires several changes in the actual program code to obtain visualisations. In addition, it is hard to customise towards problem-specific visualisations. Finally, Ducassé and Langevine present abstract visualisations generated from an automated analysis of execution traces in [DL02]. This requires a rather involved event filtering and transformation scheme.

We now adapt the approach proposed in [NRS97] to the free Prolog system SWI-Prolog and explain it in more detail than the authors themselves. Our intention is to make their very transparent and portable approach more widely accessible and understandable also for casual users of constraint programming systems. We show that obtaining quite accurate and highly customised animations of the constraint solving process need not come at great expense. In addition, we show how valuable suggestions for alternative approaches can be obtained by observing the animated constraint solving process.

To focus on the main points involved when producing animations, we use the so-called N-queens problem as a self-contained and simple example, which is also presented in [NRS97]. The task is to place N queens on an $N \times N$ chess board in such a way that no two queens attack each other, which we call a *consistent* placement. Fig. 4.7 shows a consistent placement of 8 queens.

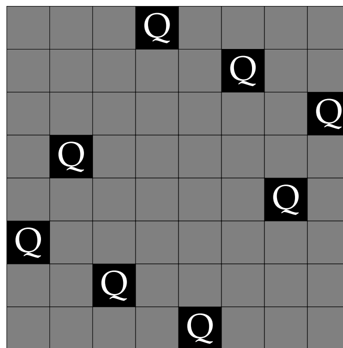


Figure 4.7: A consistent placement of 8 queens

Fig. 4.8 shows a CLP(FD) formulation for the N-queens problem: We use N variables $Q_1, \ldots, Q_N$, where Q_i denotes the row number of the queen in column i. Line 13 imposes the necessary constraints: The queens' rows must be pairwise distinct to forbid horizontal attacks, and diagonal attacks are prohibited as well.

Fig. 4.9 shows how the CLP(FD) formulation can be transparently extended to emit PostScript instructions that visualise the constraint solving process: For each value n_i of the domain of queen Q_j, a so-called *reified* constraint of the form $(Q_j = n_i) \leftrightarrow B_{ij}$ is posted. Constraint reification is a common feature of constraint solvers and lets us reflect the truth value of

many constraints into Boolean variables. When n_i vanishes from the domain of Q_j, B_{ij} becomes 0. In that case, PostScript instructions for graying out the corresponding square are emitted. When B_{ij} becomes 1, the equality holds, and instructions for placing the queen are emitted. On backtracking, the square is cleared in both cases. To make the example completely self-contained, we include the necessary PostScript definitions in Appendix A.

Fig. 4.10 shows an animation for 50 queens. The labeling strategy is *first-fail*, modified as proposed by Ertl in [Ert90]: In case of ties, we try to distribute the queens across the two horizontal halves of the board. In [Ert90], this strategy is proposed without further explanation, and it is not mentioned how this heuristic could be improved for board sizes where it does not perform well. However, when an animation of the process is available, alternative strategies are often apparent. For example, in Fig. 4.10, one can see that the strategy does not distribute the queens as evenly as intended towards the end. We give another example in Section 4.9, where we create similar animations for the SGP.

4.8 CLP formulations for the SGP

Due to its highly constrained and symmetric nature, the SGP has attracted much attention from the constraint programming community. It is problem number 10 in CSPLib, a benchmark library for constraints ([GW99]), and has led to the design of powerful but complex dynamic symmetry breaking schemes, such as [BB05]. Despite these efforts, no constraint solver was so far able to solve the original problem instance, 8−4−10, although a solution is known to exist. Even very advanced constraint solvers, such as the CLP(FD) solver that ships with SICStus Prolog, can currently solve the original problem for a maximum of only 9 weeks. A table of CP-related results for several instances is maintained by Harvey ([Har02]).

In this section, we discuss two different CLP formulations of the SGP. The first one was written by Stefano Novello and uses the freely available ECLiPSe CLP platform ([WNS97]). The second one was written by Mats Carlsson and uses the CLP(FD) solver of the proprietary Prolog system SICStus Prolog.

4.8.1 An ECLiPSe formulation of the SGP

Stefano Novello wrote a widely-cited ECLiPSe program (see [Har02]) that is able to solve the 8−4−9 instance within a few seconds on commodity hardware. His solution uses constraint logic programming over *sets*: In this case, all domain elements are sets, and many common set operations are available as built-in constraints. We have generalised his program to work for arbitrary instances $g{-}p{-}w$, and present it in Fig. 4.11. As can be seen, the built-in backtracking mechanism and powerful constraint libraries of

```
1  n_queens(N, Qs) :-
2          length(Qs, N),
3          Qs ins 1..N,
4          safe_queens(Qs).
5
6  safe_queens([]).
7  safe_queens([Q|Qs]) :-
8          safe_queens(Qs, Q, 1),
9          safe_queens(Qs).
10
11 safe_queens([], _, _).
12 safe_queens([Q|Qs], Q0, D0) :-
13         Q0 #\= Q, abs(Q0 - Q) #\= D0,
14         D1 #= D0 + 1,
15         safe_queens(Qs, Q0, D1).
```

Figure 4.8: A CLP(FD) formulation for the N-queens problem

```
1  animate(Qs) :- length(Qs, N), animate(Qs, N, Qs).
2
3  animate([], _, _).
4  animate([_|Rest], N, Qs) :-
5          animate_(Qs, 1, N),
6          N1 #= N - 1,
7          animate(Rest, N1, Qs).
8
9  animate_([], _, _).
10 animate_([Q|Qs], C, N) :-
11         Q #= N #<==> B,
12         freeze(B, queen_row_truth(C,N,B)),
13         C1 #= C + 1,
14         animate_(Qs, C1, N).
15
16 queen_row_truth(Q, N, 1) :- format("~w ~w q\n", [Q,N]).
17 queen_row_truth(Q, N, 0) :- format("~w ~w i\n", [Q,N]).
18 queen_row_truth(Q, N, _) :- format("~w ~w c\n", [Q,N]), false.
```

Figure 4.9: Observing the constraint solving process for N-queens

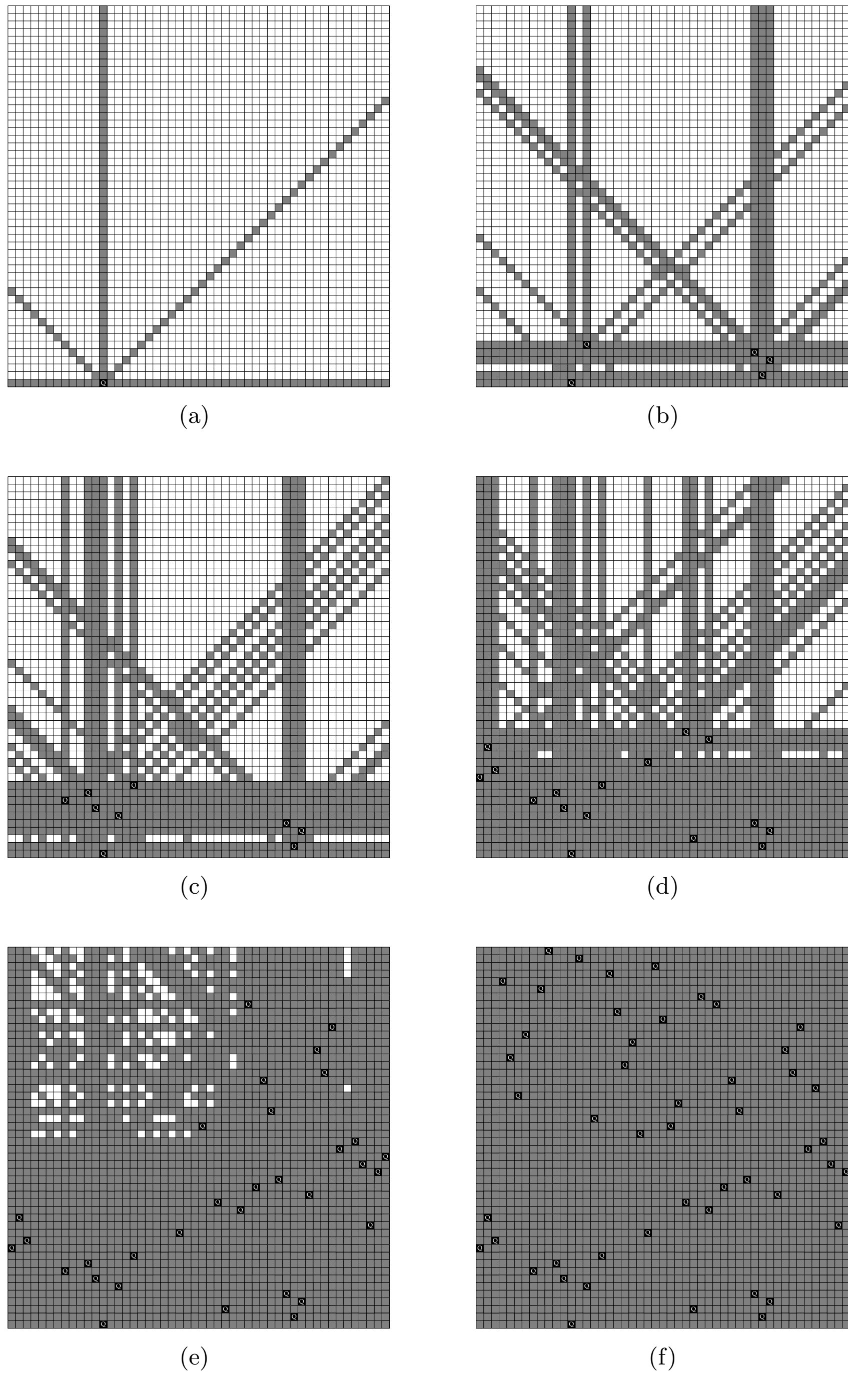


Figure 4.10: 50 queens, strategy *first-fail*, breaking ties as proposed in [Ert90], after: (a) 0.5, (b) 1.0, (c) 1.5, (d) 2.0, (e) 2.5, (f) 2.8 seconds

```
1   golf(G,P,W,Rounds) :-
2           ( for(I, 1, P*G), foreach(I,PlayerList) do true ),
3           ic_sets:(SetUB :: PlayerList..PlayerList),
4           ( count(_,1,W),
5             foreach(GroupsInRound,Rounds),
6             param(SetUB), param(G), param(P) do
7               ( foreach(S,GroupsInRound), count(_,1,G),
8                 param(P), param(SetUB) do
9                     ic_sets:(S :: [] .. SetUB), #(S,P)
10              ),
11              all_disjoint(GroupsInRound)
12          ),
13          ( fromto(Rounds,[R|Rest0],Rest0,[]) do
14                flatten(Rest0,Rest),
15                ( foreach(Group,R), param(Rest) do
16                      ( param(Group),
17                        foreach(Group1,Rest) do
18                            ic:(ISize :: 0..1),
19                            #(Group /\ Group1,ISize)
20                      )
21                )
22          ),
23          ( for(Player,1,P*G), param(Rounds) do
24              ( foreach(R,Rounds), param(Player) do
25                    member(Group,R), Player in Group
26              )
27          ).
```

Figure 4.11: A set-based formulation for the SGP, using ECLiPSe

ECLiPSe allow for a remarkably concise solution, which we briefly explain:

The schedule is represented as a list of weeks, and each week is a list of groups. Each group is a *set* of players, which are represented by the numbers $1, \ldots, g \times p$. Using sets implicitly breaks the symmetry arising from different player orders within groups. The first relevant constraints occur in line 9: This line states that each group S is a subset of the set of players, and that the cardinality of S is p. Line 11 ensures that the groups within each week are disjoint. Line 19 encodes *socialisation* in the language of set theory: The cardinality of the intersection of any two groups must be either 0 or 1. This is equivalent to saying that no two players can play together more than once in the same group. This completes the declarative description of all requirements.

The remaining lines of the program search for solutions: Going through all players in order, we do the following for each player p_i: The weeks are traversed in order, and in each week w_j, p_i is placed into a group of w_j where that is still possible. If no such place can be found, chronological *backtracking* occurs, meaning that the most recent previous assignment is undone, and a different choice is tried. We return to this strategy later.

4.8.2 A SICStus CLP(FD) formulation of the SGP

A CLP(FD) formulation for the SGP was generously posted in 2005 to the discussion group `comp.lang.prolog` by Mats Carlsson, the designer and main implementor of SICStus Prolog and its CLP(FD) solver ([Car05]).

Carlsson's formulation is considerably more verbose than Novello's ECLiPSe version, but also more sophisticated. We explain the main ideas behind Carlsson's formulation without reproducing the complete program:

- The basic data structures are similar to the ECLiPSe formulation: The schedule is a list of weeks, which in turn are lists of groups, and each group consists of initially free variables, which become instantiated to "players" during the search. In slight contrast to Novello's program, players are now represented by the numbers $0, \ldots, g \times p - 1$.
- The first week is initialised by lining up all players in their natural order across the groups. Players $k \times p, \ldots, (k+1) \times p - 1$, $0 \leq k \leq g - 1$, have thus already played together.
- The free variables within each week are constrained to be pairwise distinct, using a built-in constraint like `all_different` as explained in the Sudoku formulation above.
- The most important new idea in Carlsson's formulation is a *multiplication table*, which is used to encode socialisation and is represented as a list of triples. There is one triple (p_i, p_j, m_{ij}) for each pair of players p_i, p_j, $p_i < p_j$ in this list. The value m_{ij} is computed as $p_1 \times (g \times p) + p_2$ and is thus unique for each such pair of players. This multiplication table is used as follows: First, all ordered pairs of distinct variables and players that occur together in any group are collected, in their natural order. For example, for the array shown in Fig. 4.12, all ordered pairs of instantiated and free variables that occur in the same group are: (0,B), (0,C), (B,C), (1,E), (1,F), and (E,F).

	Week 1		
Group 1	0	B	C
Group 2	1	E	F

Figure 4.12: A partially instantiated schedule

Next, these pairs are extended to triples by adding one new variable to each of them. In the previous example, this yields the triples (0,B,x_1), (0,C,x_2), (B,C,x_3), (1,E,x_4), (1,F,x_5) and (E,F,x_6), where x_i denotes a free variable that does not occur anywhere else in the formulation. Then, the variables x_i are constrained to be pairwise different, again using a library constraint like `all_different`. Finally, each of these triples is constrained to be an element of the previously built multiplication table. This is done with a library constraint called `table`. In summary, these steps guarantee that no pair of golfers can occur more than once in any group.

Proof. Suppose p_1 and p_2, $p_1 < p_2$, occur more than once in the same group, and let (p_1, p_2, x_m) and (p_1, p_2, x_n) be the corresponding triples that were built from two groups in which these players play together. Then, by the `table` constraint, $x_m = x_n = p_1 \times (g \times p) + p_2$. But by the `all_different` constraint, $x_m \neq x_n$. Contradiction. □

- Symmetries *within* groups are broken by imposing (implied) "less than" constraints between the variables of each group. Due to the structure of the first week, a "less than" constraint between the integer *quotients* v_i/p and v_j/p can be imposed for each adjacent pair of variables v_i and v_j, since the players up to (but not including) the next multiple of p have already played together in the first week. Symmetries *among* a week's groups are broken by requiring that the list of each group's first variable consist of strictly ascending values. It follows that player 0 must be the first player of the first group in each week. The players that player 0 has partnered in the first week, namely $1, \ldots, p-1$, are assigned to the first positions of the second, third, etc. group of each further week.
- Another set of implied constraints is added: Players that $0, \ldots, p-1$ partner in further weeks must be distinct. Such constraints can yield additional pruning and thus reduce the search space.
- As in the SAT formulation (Section 3.7), symmetries among weeks are broken by imposing "less than" constraints between the second variable of the first group of each week.

Carlsson's formulation also allows to specify labeling options and a parameter that reorders variables before labeling them:

- **byrow**
  The variables of each week occur row after row, from left to right for each row.
- **bycol**
  The variables of each week occur column after column, from top to bottom for each column.
- **byrowall**
  The variables are collected as in **byrow**, and then collated into a single list that contains the ordered variables of each week.
- **bycolall**
  Analogous to **byrowall**, ordering the variables for each week by column before merging them into a single list.

4.9 A new finite domain constraint solver

As part of this thesis, we developed a new finite domain constraint solver out of the desire to run Carlsson's CLP(FD) formulation of the SGP ([Car05]) in a free environment. Our solver is available in the free Prolog systems SWI-Prolog ([Wie03]) and YAP ([dSC06]) as `library(clpfd)`. Thus, Carlsson's formulation can be run with these freely available systems with very few modifications.

Our solver provides several unique features, such as reasoning over arbitrarily large integers and always terminating propagation, but since they are not relevant for solving SGP instances with the present formulation, we do not discuss them here further.

Carlsson reports finding a solution for the 8−4−9 instance within fractions of a second, using SICStus Prolog with the labeling option "min", and ordering "bycolall". The challenge therefore consists in matching this result as closely as possible, using the same problem formulation with our freely available solver.

The two most important constraints of Carlsson's CLP(FD) formulation of the SGP are `all_different` and `table`, which are explained above. For both constraints, quite efficient and strong propagation algorithms are available ([Rég94], [Bar01]), but we did not implement them. Instead, we settled for the simplest implementation of these constraints in both cases.

Our initial results were not competitive. Using an Apple MacBook with a 2.16 GHz Intel Core 2 Duo CPU and the labeling strategy "first-fail" with the ordering "bycolall", we obtained a solution for the 8−4−7 instance within 20 minutes. We found no solution for the 8−4−8 instance within one week of CPU time with the same labeling strategy. This strategy therefore left little hope to solve the yet harder 8−4−9 instance. We then applied the ideas of Section 4.7 to the SGP to get an idea of what the solver was doing. Fig. 4.13 shows various stages of progress when trying to solve the 8−4−9 instance with our constraint solver.

It is apparent from this figure that the weeks are not filled evenly when using the labeling strategy "first-fail": The first few weeks are completely filled after about one minute, and the remaining weeks are all partially filled. This situation seems not to change for quite some time.

We therefore started looking for allocation strategies that did something else. One of the things we tried was to use the allocation strategy of the ECLiPSe formulation: It distributes the players, in their natural order, across all weeks. This ensures that the weeks are filled very uniformly, which at least differs from what we had.

Fig. 4.14 shows the various stages of progress when applying this strategy after posting all constraints of Carlsson's CLP(FD) formulation with our free constraint solver. As can be seen, instance 8−4−9 can thus be solved in reasonable time with our solver as well.

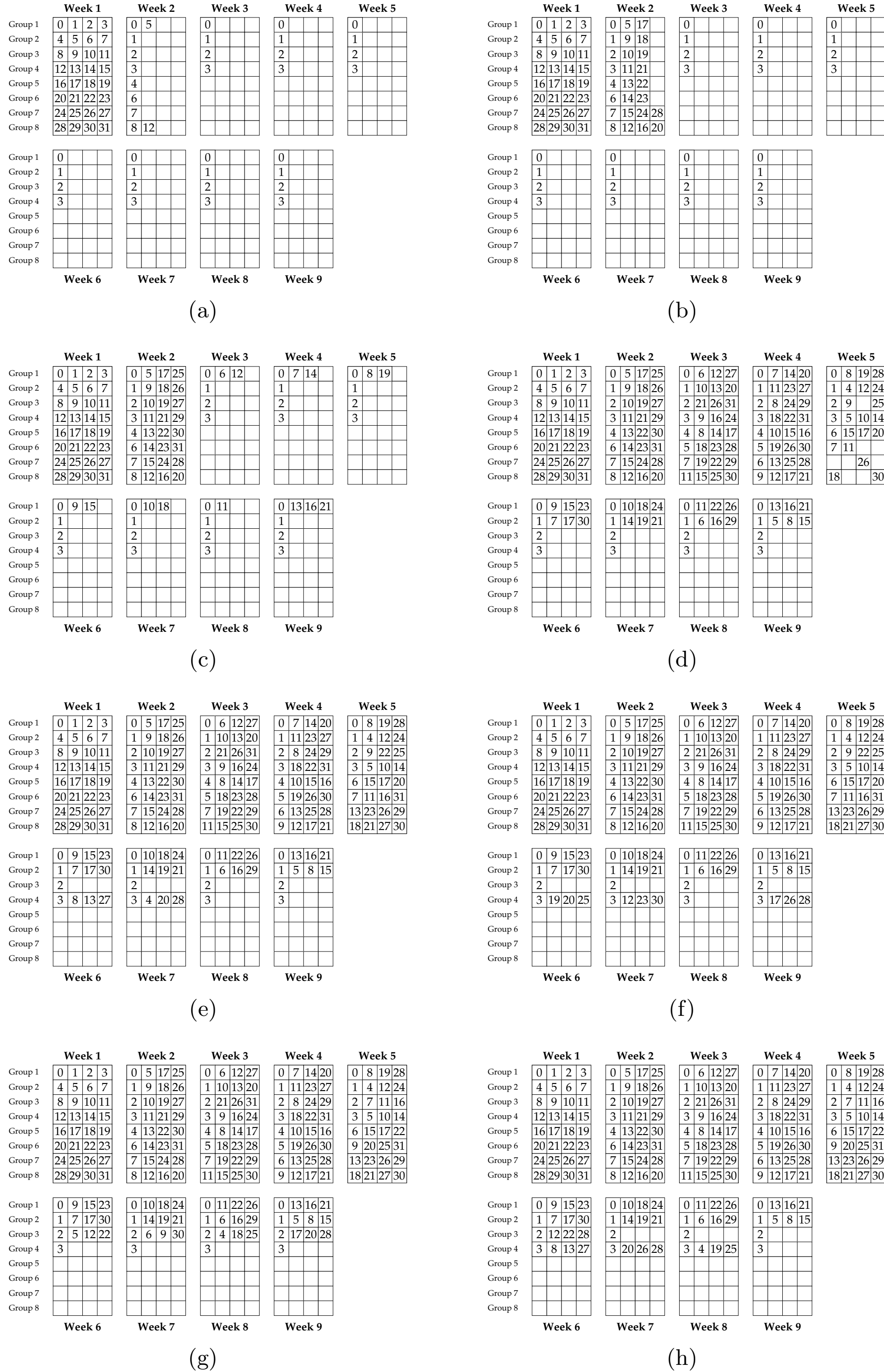


Figure 4.13: Instance 8−4−9, labeling strategy *first-fail*, after: (a) 10, (b) 20, (c) 30, (d) 50, (e) 100, (f) 200, (g) 400, (h) 700 seconds

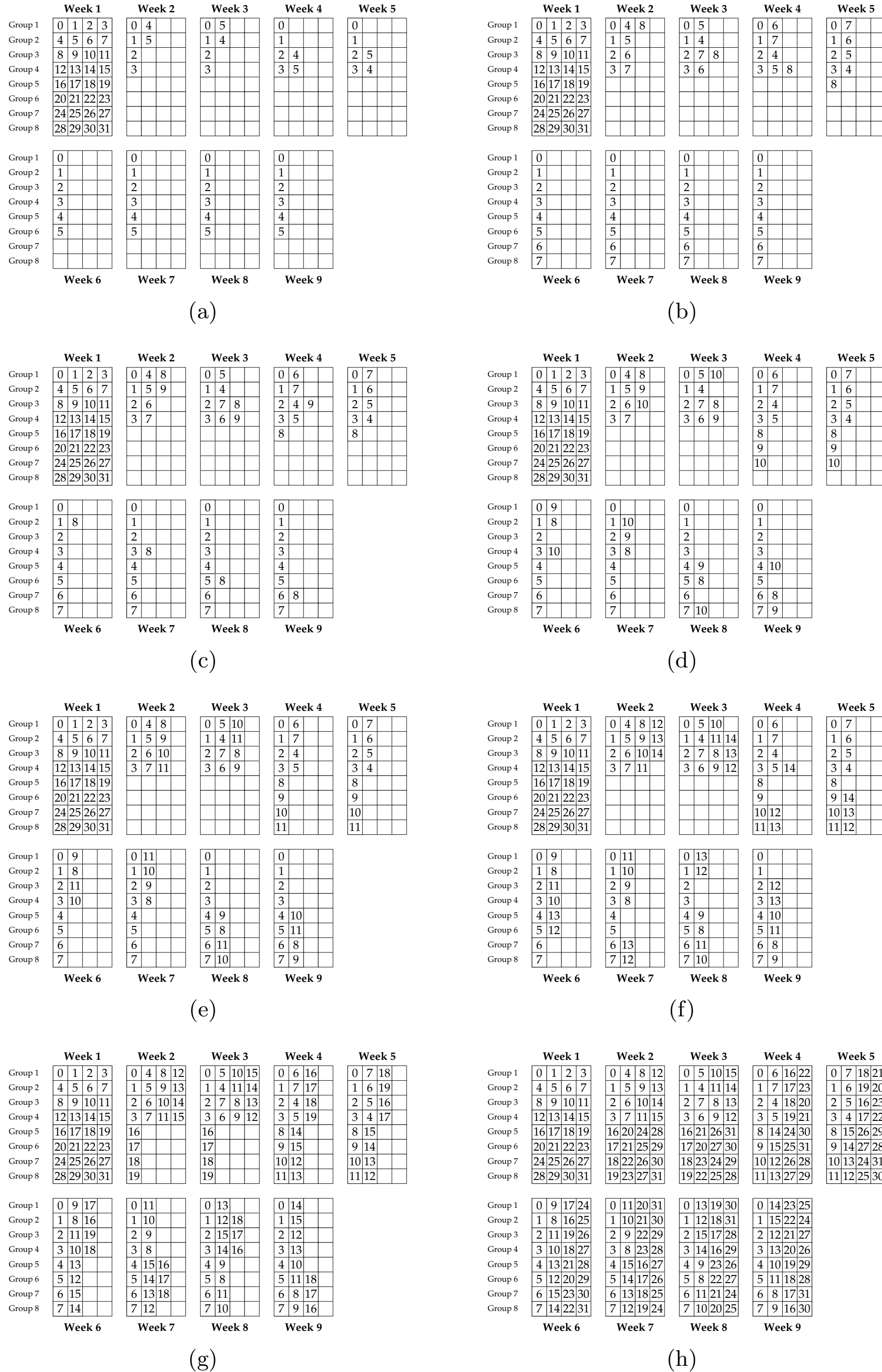


Figure 4.14: Instance 8−4−9, allocation as in the ECLiPSe formulation, after: (a) 10, (b) 12, (c) 13, (d) 14, (e) 15, (f) 16, (g) 17, (h) 18 seconds

4.10 Experimental results

We now have seen two different constraint-based formulations for the SGP, and both can solve the original SGP instance for 9 weeks. The ECLiPSe formulation by Stefano Novello is considerably more concise than Carlsson's formulation, and so it is left to find out whether the higher complexity of Carlsson's CLP(FD) formulation pays off when solving other instances.

Table 4.1 compares the two formulations on several different instances. Each instance was tried with Novello's formulation (using the ECLiPSe CLP platform) and Carlsson's formulation (using our constraint solver with SWI-Prolog) on an Apple MacBook with a 2.16 GHz Intel Core 2 Duo CPU and 1GB RAM, and we show the time it took to find the first solution. The symbol "–" means that no solution was found within 20 minutes. The "strategy" column shows which labeling options and variable order was used with the Prolog formulation. The entry "custom" means that we used the allocation strategy of the ECLiPSe formulation, which we explained above.

From Table 4.1, it is clear that the more sophisticated formulation is very advantageous on many instances. Even with our comparatively simplistic implementations of the most important constraints used in Carlsson's formulation, we are able to solve many instances that cannot be solved with the shorter ECLiPSe formulation within the timeout.

In fact, the ECLiPSe formulation solves many other instances only for one or two weeks, leaving the impression that its allocation strategy works for $8-4-w$ instances largely by coincidence.

It is interesting that our custom allocation solves instance $5-3-7$ faster than instance $5-3-6$, although a solution for the former implies a solution for the latter. We therefore recommend to always try to solve "harder" instances as well, especially when using custom heuristics, and not to stop too early when an instance cannot be solved for a certain number of weeks.

4.11 Conclusion

Constraint-based approaches towards the SGP have many advantages: Advanced constraint solvers allow to compactly state the requirements, and can solve many instances efficiently. Pre-defined variable and value selection strategies allow users to try out different strategies for solving a problem. In addition, it is easy to define custom allocation strategies on top of a constraint-based model.

CLP(FD) formulations can be regarded as a superset of SAT encodings (since the Boolean values TRUE and FALSE can be mapped to integers), and let users express constraints in a more convenient and shorter form that is more easily seen to be correct. Many observations derived from studying SAT encodings (see Chapter 3) also apply to constraint-based approaches, in particular the potential impact of symmetry breaking constraints and

different formulations. Like SAT solvers, constraint solvers have evolved continuously in recent years, and existing formulations automatically benefit from improved solvers.

We have discussed two existing constraint-based approaches for solving SGP instances in this chapter. One uses the freely available CLP platform ECLiPSe, and one was written for the proprietary Prolog system SICStus Prolog. As is often the case for constraint-based approaches, both versions are *complete*: If a solution exists, it will be found. This property can be used to prove by exhaustive search that no solution exists for some instances, such as $6-6-4$. CLP formulations can also be used to decide completion problems, where some values are already given.

We have implemented a new finite domain constraint solver which lets us run Mats Carlsson's CLP(FD) formulation of the SGP with the freely available Prolog systems SWI-Prolog and YAP after marginal modifications. Using our solver and visualisations of the constraint solving process, we could solve the original SGP instance for 9 weeks. This matches the best current result obtained with constraint-based approaches for this instance, but is not optimal. We also presented benchmark results for several other instances and showed that the more sophisticated problem formulation is highly advantageous on many instances.

instance	ECLiPSe	SWI-Prolog	strategy
5–3–1	0.19s	0.01s	custom
5–3–2	2.98s	0.05s	custom
5–3–3	–	0.11s	custom
5–3–4	–	0.20s	custom
5–3–5	–	5.72s	custom
5–3–6	–	156.93s	custom
5–3–7	–	11.88s	custom
6–4–1	0.19s	0.02s	ff, bycolall
6–4–2	–	0.39s	ff, bycolall
6–4–3	–	1.25s	ff, bycolall
6–4–4	–	2.32s	ff, bycolall
6–4–5	–	26.42s	ff, bycolall
7–4–1	0.19s	0.03s	ff, bycolall
7–4–2	–	0.70s	ff, bycolall
7–4–3	–	1.99s	ff, bycolall
7–4–4	–	3.61s	ff, bycolall
7–4–5	–	5.39s	ff, bycolall
8–4–1	0.19s	0.04s	custom
8–4–2	0.22s	1.04s	custom
8–4–3	0.27s	2.22s	custom
8–4–4	0.38s	3.51s	custom
8–4–5	0.49s	4.95s	custom
8–4–6	0.64s	6.93s	custom
8–4–7	0.85s	8.97s	custom
8–4–8	1.18s	12.18s	custom
8–4–9	1.47s	14.64s	custom

Table 4.1: Benchmark results using the two discussed constraint-based approaches for several instances

5 A new greedy heuristic for the SGP

5.1 An important observation

Consider the solution for the instance 8−4−5 shown in Fig. 5.1. We highlight players 0 and 31 in each week.

	Week 1	Week 2	Week 3	Week 4	Week 5
Group 1	0 1 2 3	0 4 8 12	0 5 10 15	0 6 11 13	0 7 9 14
Group 2	4 5 6 7	1 5 9 13	1 4 11 14	1 7 10 12	1 6 8 15
Group 3	8 9 10 11	2 6 10 14	2 7 8 13	2 4 9 15	2 5 11 12
Group 4	12 13 14 15	3 7 11 15	3 6 9 12	3 5 8 14	3 4 10 13
Group 5	16 17 18 19	16 20 24 28	16 21 26 31	16 22 27 29	16 23 25 30
Group 6	20 21 22 23	17 21 25 29	17 20 27 30	17 23 26 28	17 22 24 31
Group 7	24 25 26 27	18 22 26 30	18 23 24 29	18 20 25 31	18 21 27 28
Group 8	28 29 30 31	19 23 27 31	19 22 25 28	19 21 24 30	19 20 26 29

Figure 5.1: A solution for the 8−4−5 instance, with players 0 and 31 highlighted in each week

Consider now the set of players with which player 0 has played in any group in this schedule, which is:

$$\{1, 2, 3\} \cup \{4, 8, 12\} \cup \{5, 10, 15\} \cup \{6, 11, 13\} \cup \{7, 9, 14\} = \{1, \dots, 15\}$$

Analogously, consider the set of players with which player 31 has played in any group, which is:

$$\{28, 29, 30\} \cup \cdots \cup \{17, 22, 24\} = \{16, \dots, 30\}$$

Suppose now that we want to extend this schedule to further weeks, while leaving the existing weeks unchanged. From the previous observations, it follows that player 0 cannot play together with player 31 in any future group, for if these players did play together in any group, they would not find any two partners that are necessary to build a complete group. In fact, there could not even be a further group of size three containing them: Player 0 can only play with {16,...,31}, and player 31 can only play with {0,...,15}, since the other combinations already occurred in previous weeks. This leaves no player that is compatible with both of them.

The same reasoning can be applied to other pairs of players: As soon as *any* two players have partnered complementary subsets of players, they cannot find a third player to play with. An analogous property holds for larger subsets of players. If enough pairs or larger subsets of players run out of compatible partners, it is not possible to build any further group in which they play together. In the next sections, we exploit this observation to develop a new greedy heuristic for the SGP.

5.2 Freedom of sets of players

We first formalise what we observed in the previous section:

Definition 5.1. (*Freedom*) Let C be a partial configuration. For an arbitrary player x, we denote with $P_C(x)$ the *potential partner-set* of x with respect to C, i.e., the set of players that x can still *partner* in any group, assuming C as given. In other words, $P_C(x)$ is the set of all players (except x), minus the players that x has already partnered in any group of C. For any set S of players, we denote with $\varphi_C(S)$ the *freedom* of S with respect to C, and define it as the cardinality of the intersection of the potential partner-sets of all players in S, i.e.:

$$\varphi_C(S) = \left| \bigcap_{x \in S} P_C(x) \right| \tag{5.1}$$

Informally, the freedom of a set of players denotes how many players they can still “partner together”.

For example, with the configuration C of Fig. 5.1, the freedom of the player set $\{0, 31\}$ is:

$$\varphi_C(\{0, 31\}) = |\{16, \ldots, 31\} \cap \{0, \ldots, 15\}| = |\emptyset| = 0$$

As we already observed, these two players therefore cannot play together in any further group. It turns out that the solution of Fig. 5.1 cannot be extended by any further week (in fact, not even by a single *group*) if the given weeks must be kept unchanged. This is because the freedom of *all* pairs of players that have not yet played together is 0 in this case. Clearly, this is not always the case, and different pairs of players also do not necessarily have the same freedom. For example, consider a different solution for the 8−4−5 instance, shown in Fig. 5.2.

	Week 1	**Week 2**	**Week 3**	**Week 4**	**Week 5**
Group 1	0 1 2 3	0 4 8 12	0 5 10 15	0 6 16 22	0 7 18 21
Group 2	4 5 6 7	1 5 9 13	1 4 11 14	1 7 17 23	1 6 19 20
Group 3	8 9 10 11	2 6 10 14	2 7 8 13	2 4 18 20	2 5 16 23
Group 4	12 13 14 15	3 7 11 15	3 6 9 12	3 5 19 21	3 4 17 22
Group 5	16 17 18 19	16 20 24 28	16 21 26 31	8 14 24 30	8 15 26 29
Group 6	20 21 22 23	17 21 25 29	17 20 27 30	9 15 25 31	9 14 27 28
Group 7	24 25 26 27	18 22 26 30	18 23 24 29	10 12 26 28	10 13 24 31
Group 8	28 29 30 31	19 23 27 31	19 22 25 28	11 13 27 29	11 12 25 30

Figure 5.2: A different solution for the 8−4−5 instance

Here are examples computed with the configuration shown in Fig. 5.2:

$$\begin{array}{rcccl} \varphi(\{0,27\}) & = & |\emptyset| & = & 0 \\ \varphi(\{0,24\}) & = & |\{9,11,17,19\}| & = & 4 \\ \varphi(\{0,9\}) & = & |\{17,19,20,23,24,26,29,30\}| & = & 8 \\ \varphi(\{0,9,24\}) & = & |\{17,19\}| & = & 2 \end{array}$$

5.3 A greedy heuristic for the SGP

We now propose a greedy heuristic for the SGP instance $g{-}p{-}w$, which is intended to be used with a complete backtracking search. Let us first suppose that p is even. Then the task of scheduling the players into groups in each week can also be seen as scheduling *pairs* of players into groups.

The backtracking search now proceeds as follows: Visit the weeks one after another, and in each week, traverse the groups in their natural order. For each pair of adjacent positions in a group, we need to select a pair of players still remaining to be scheduled in the current week. Here, we select a pair having *minimal* freedom with respect to the current partial configuration.

The intention behind this choice is that if a pair of players is close to running out of potential partners, then they should be scheduled in the same group while that is still possible at all. This heuristic is similar to the labeling strategy "first-fail" in constraint satisfaction problems (Section 4.6).

If a group is encountered that cannot be completed, *backtracking* occurs: We undo the most recent choice of players, and select a pair with next larger degree of freedom instead.

If p is odd, there are several options. A simple solution is to schedule pairs of players for each group as far as possible, and then to fill the remaining position with any player that is compatible with all other players scheduled in this group.

Another approach is to generalise the heuristic to triples and larger sets of players. Here, a trade-off must be reached between accurate assessment of a scheduling choice and computational tractability.

We show a Prolog specification of this heuristic, tailored for $8{-}4{-}w$ instances (i.e., the original SGP), in Fig. 5.3. We benefit from arbitrary precision arithmetic to represent sets of players as bit vectors. This lets us efficiently intersect sets by using fast bitwise operations. Determining a set's cardinality is thus also very efficient. This program can solve the original SGP for 9 weeks (Fig. 5.4) virtually instantly, although some backtracking is necessary. This matches the best current results of constraint solvers for this instance, while using much simpler methods. After extending the heuristic from pairs to triples, we obtained solutions for Kirkman's schoolgirl problem in 2 seconds, using a 2.16 GHz Apple MacBook (Fig. 5.5). This solves the problem optimally, without taking symmetries into account at all.

While our heuristic does not perform so well on *all* SGP instances, we show in the next chapter that its main idea can be used to solve the original SGP optimally.

```
 1  schedule(N, Weeks) :- init(S0), phrase(weeks(N, S0), Weeks).
 2
 3  init(State) :-
 4      B is (1 << 32) - 1, findall(N-B, between(0,31,N), State0),
 5      list_to_assoc(State0, State).
 6
 7  weeks(0, _)   --> !, [].
 8  weeks(N0, S0) --> { numlist(0, 31, Ps), groups(Ps, S0, S1, Gs, []),
 9      N1 is N0 - 1 }, [Gs], weeks(N1, S1).
10
11  groups([], S, S)   --> !, [].
12  groups(Ps0, S0, S) -->
13      { list_pairs(Ps0, S0, Pairs0, []), keysort(Pairs0, Pairs1),
14        member(_-p(A,B,NA,NB), Pairs1),
15        member(_-p(C,D,NC,ND), Pairs1),
16        A =\= C, A =\= D, B =\= C, B =\= D,
17        Pattern is (1 << A) \/ (1<<B) \/ (1<<C) \/ (1 << D),
18        ( NA /\ NB /\ NC /\ ND) /\ Pattern =:= Pattern,
19        all_delete([A,B,C,D], Ps0, Ps1),
20        eliminate([B,C,D], A, S0, S1), eliminate([A,C,D], B, S1, S2),
21        eliminate([A,B,D], C, S2, S3), eliminate([A,B,C], D, S3, S4) },
22      [[A,B,C,D]], groups(Ps1, S4, S).
23
24  eliminate([A,B,C], P, S0, S) :-
25      get_assoc(P, S0, CP0), put_assoc(P, S0, CP1, S),
26      CP1 is (\ (1<<A) /\ \ (1<<B) /\ \ (1<<C)) /\ CP0.
27
28  all_delete([], Ds, Ds).
29  all_delete([A|As], Ds0, Ds) :- delete(Ds0, A, Ds1),
30      all_delete(As, Ds1, Ds).
31
32  list_pairs([], _)      --> [].
33  list_pairs([L|Ls], S0) --> { get_assoc(L, S0, NL) },
34      pair_up(Ls, S0, NL, L), list_pairs(Ls, S0).
35
36  pair_up([], _, _, _)       --> [].
37  pair_up([B|Bs], S0, NA, A) -->
38      { get_assoc(B, S0, NB), Num is popcount(NA/\NB) },
39      (   {  NA /\ (1<<B) =\= 0, Num >= 4 } ->
40          [Num-p(A,B,NA,NB)]  ; [] ), pair_up(Bs, S0, NA, A).
```

Figure 5.3: Prolog specification of our greedy heuristic for $8{-}4{-}w$ instances

	Week 1	Week 2	Week 3	Week 4	Week 5
Group 1	0 1 2 3	0 4 8 12	0 16 5 21	0 26 6 28	0 11 18 25
Group 2	4 5 6 7	1 5 9 13	1 17 4 20	1 27 7 29	1 10 19 24
Group 3	8 9 10 11	2 6 10 14	2 18 7 23	2 24 4 30	2 9 16 27
Group 4	12 13 14 15	3 7 11 15	3 19 6 22	3 25 5 31	3 8 17 26
Group 5	16 17 18 19	16 20 24 28	8 24 13 29	8 18 14 20	4 15 22 29
Group 6	20 21 22 23	17 21 25 29	9 25 12 28	9 19 15 21	5 14 23 28
Group 7	24 25 26 27	18 22 26 30	10 26 15 31	10 16 12 22	6 13 20 31
Group 8	28 29 30 31	19 23 27 31	11 27 14 30	11 17 13 23	7 12 21 30

	Week 6	Week 7	Week 8	Week 9
Group 1	0 13 7 10	0 19 14 29	0 20 15 27	0 17 9 24
Group 2	1 12 6 11	1 18 15 28	1 21 14 26	1 16 8 25
Group 3	2 15 5 8	2 17 12 31	2 22 13 25	2 19 11 26
Group 4	3 14 4 9	3 16 13 30	3 23 12 24	3 18 10 27
Group 5	16 29 23 26	4 23 10 25	4 16 11 31	4 21 13 28
Group 6	17 28 22 27	5 22 11 24	5 17 10 30	5 20 12 29
Group 7	18 31 21 24	6 21 8 27	6 18 9 29	6 23 15 30
Group 8	19 30 20 25	7 20 9 26	7 19 8 28	7 22 14 31

Figure 5.4: A 9-week solution for the SGP, found by our greedy heuristic

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Group 1	0 1 2	0 3 6	2 5 8	0 5 12	0 7 13	0 8 10	0 9 14
Group 2	3 4 5	1 4 7	0 4 11	1 6 10	1 5 9	1 11 12	1 8 13
Group 3	6 7 8	2 9 12	1 3 14	2 7 14	2 6 11	2 4 13	2 3 10
Group 4	9 10 11	5 10 13	6 9 13	4 8 9	3 8 12	3 7 9	4 6 12
Group 5	12 13 14	8 11 14	7 10 12	3 11 13	4 10 14	5 6 14	5 7 11

Figure 5.5: A solution for Kirkman's schoolgirl problem, found by our greedy heuristic extended to triples

6 Metaheuristic methods

6.1 Introduction

In this chapter, we first present existing metaheuristic approaches for solving SGP instances. We then use a simplified local search in a new greedy randomised adaptive search procedure (GRASP), which we show to be very competitive with existing metaheuristic and constraint-based approaches. One of the instances we solve is $8-4-10$. This makes our method the first metaheuristic technique that solves the original SGP optimally.

6.2 Metaheuristic SAT solving

We already encountered a metaheuristic method in Section 3.6, where we used Walksat to solve generated SAT instances. Walksat is based on GSAT and uses metaheuristic methods like local search, random-walk and random-noise, which were shown to be surprisingly well-suited for solving certain classes of SAT instances ([SKC93]). The basic idea of GSAT is to start with a random assignment of TRUE and FALSE values, and then to repeatedly change ("flip") the assignment of a variable that leads to the largest decrease of unsatisfied clauses. To escape from local optima, this simple local search approach can be mixed with random walk and random noise moves. Random walk was shown to be more effective in [SKC93], and means that with probability p, a variable occurring in an unsatisfied clause is flipped. Algorithm 6.1 shows the basic GSAT algorithm with random walk.

Algorithm 6.1 GSAT with random walk ([SKC93])

```
1: for i ← 1 to MAX-TRIES do
2:   T ← a randomly generated truth assignment
3:   for j ← 1 to MAX-FLIPS do
4:     if T satisfies α then
5:       return T
6:     else
7:       if RANDOM() < p then
8:         Pick a variable occurring in an unsatisfied clause and flip its
           truth assignment
9:       else
10:        Flip any variable in T that results in the greatest decrease (can
           be 0 or negative) in the number of unsatisfied clauses
11:      end if
12:    end if
13:  end for
14: end for
15: return "No satisfying assignment found"
```

We saw in Chapter 3 that Walksat is currently not competitive with other approaches for the tested SGP instances, at least not when using the SAT model that we proposed. However, its main idea can be used to derive a much more effective metaheuristic method for solving SGP instances. It is especially instructive to observe the sequence of intermediate configurations when solving SGP instances with Walksat (Fig. 3.7). As can be seen, several unnecessary steps occur. For example, due to the nature of the SAT encoding, not all positions need to have players assigned to them in the course of the search (Fig. 3.7 (b), (d), and others), and the same player can intermittently occur in several groups of the same week (Fig. 3.7 (a), (b), and others). Such configurations are of course invalid and certainly require additional flips before the instance is solved. Therefore, we should try to prevent these configurations from occurring in the first place, and the method we discuss in the next section does that.

6.3 Local search for the SGP

In [DH05], Dotú and Hentenryck propose a local search approach with tabu lists for the SGP. We briefly describe their main ideas, with some changes in notation for brevity.

6.3.1 The model

Given the SGP instance g–p–w, we introduce a decision variable $x[i,j,k]$ for $1 \leq i \leq g$, $1 \leq j \leq p$, $1 \leq k \leq w$. A schedule σ is an assignment of decision variables to values. The value of a decision variable $x[i,j,k]$ in a schedule σ is denoted as $\sigma(x[i,j,k])$ and states which golfer plays in group i and position j of week k. Golfers are represented by the integers $1,\ldots,g \times p$. A schedule is valid if:

- each golfer plays exactly once in each week
  As we will see, this constraint is ensured by the initial assignment and preserved by local moves. Therefore, it need not be considered explicitly.
- any two golfers play together at most once in the same group.

Dotú and Hentenryck introduce the following notation which is useful to concisely describe their algorithm: The value $\#_\sigma(a,b)$ denotes the number of times golfers a and b play in the same group in schedule σ, and a variable $m[a,b]$ is used to count the number of times that golfers a and b meet in the same group beyond their allowed number of meetings. Since pairs of golfers are allowed to play in the same group at most *once*, its value $\sigma(m[a,b])$ in a schedule σ is defined as:

$$\sigma(m[a,b]) = \max(0, \#_\sigma(a,b) - 1) \quad (6.1)$$

The value $f(\sigma)$ of a schedule σ is defined as as:

$$f(\sigma) = \sum_{a,\,b\,\in\{1,\ldots,g\times p\}} \sigma(m[a,b]) \quad (6.2)$$

Finding a solution for an SGP instance is thus a minimisation problem consisting of finding a valid schedule σ such that $f(\sigma) = 0$.

The triple $\langle i,j,k\rangle$ is a *conflict position* in a schedule σ iff there is a $j' \neq j$ such that there is a week $k' \neq k$ with $\sigma(x[i,j,k]) = \sigma(x[l,m,k'])$ and $\sigma(x[i,j',k]) = \sigma(x[l,n,k'])$ for any l, m and n, i.e., wherever a player is in the same group with another player more than once. We denote the set of conflict positions of a schedule σ with $C(\sigma)$.

6.3.2 The neighbourhood

As their neighbourhood, Dotú and Hentenryck chose swapping two golfers from different groups in the same week. The set S of swaps is defined as:

$$S = \{(\langle g_1,p_1,k\rangle, \langle g_2,p_2,k\rangle) \mid 1 \leq g_1,g_2 \leq g, 1 \leq p_1,p_2 \leq p, g_1 \neq g_2\} \quad (6.3)$$

To focus on swaps that may decrease the objective function, only swaps that involve a conflict position are considered. The reduced set of swaps, denoted as $S^-(\sigma)$, is thus:

$$S^-(\sigma) = \{(s_1,s_2) \in S \mid s_1 \in C(\sigma)\} \quad (6.4)$$

6.3.3 The tabu component

The tabu component in [DH05] is an array *tabu*, which stores a separate tabu list for each week. For each week k, the list $tabu[k]$ contains a triple $\langle a,b,i\rangle$, where i is the first iteration in which golfers a and b can be swapped again in that week. The time a pair of golfers stays in the tabu list is randomly drawn from the interval $[4,100]$. The set $S^t(\sigma,k)$ denotes all swaps s from $S^-(\sigma)$ such that s is not tabu in iteration k.

Aspiration. If a swap improves the best solution found so far, it is also considered, even if it is tabu. The set $\mathcal{S}^*(\sigma,\sigma^*)$ is defined as

$$\mathcal{S}^*(\sigma,\sigma^*) = \{(t_1,t_2) \in S^-(\sigma) \mid f(\sigma[x[t_1] \leftrightarrow x[t_2]]) < f(\sigma^*)\} \quad (6.5)$$

where $\sigma[x_1 \leftrightarrow x_2]$ denotes the schedule σ with the values of variables x_1 and x_2 swapped, $x[\langle i,j,k\rangle]$ is equivalent to $x[i,j,k]$, and σ^* denotes the best solution found so far.

6.3.4 The tabu search algorithm

The tabu search algorithm proposed by Dotú and Hentenryck is summarised in Algorithm 6.2. The initial and new configuration built in lines 2 and 12 must ensure that each golfer plays exactly once each week. One way is to line up all players in their natural order in each week. Alternatively, one can use variants of construction methods from Chapter 2, which can solve several instances deterministically and provide good initial configurations for others ([DH05]).

Algorithm 6.2 SGP-LS ([DH05])

```
1: for i ← 1 to w do tabu[i] ← NIL end for
2: σ* ← σ ← random configuration
3: k ← s ← 0
4: while k ≤ MAX-ITER ∧ f(σ) > 0 do
5:    select (t1, t2) ∈ S^t(σ, k) ∪ S*(σ, σ*) minimising f(σ[x[t1] ↔ x[t2]])
6:    τ ← RANDOM([4,100])
7:    tabu[week(t1)] ← tabu[week(t1)] ∪ {⟨σ(x[t1]), σ(x[t2]), k + τ⟩}
8:    σ ← σ[x[t1] ↔ x[t2]]
9:    if f(σ) < f(σ*) then
10:       σ* ← σ ; s ← 0
11:   else if s > MAX-STABLE then
12:       σ ← random configuration ; s ← 0
13:       for i ← 1 to w do tabu[i] ← NIL end for
14:   else
15:       s++
16:   end if
17:   k++
18: end while
```

The algorithm is notable for not taking symmetries into account at all. Nevertheless, it was shown to be very competitive with constraint-based approaches in [DH05].

6.4 Memetic evolutionary programming

Cotta et al. present a memetic evolutionary programming approach towards the SGP in [CDFH06], where Algorithm 6.2 is used as the local improvement strategy. They report improvements over the results of [DH05] on many instances, using lamarckian learning strategies. This makes their approach the best metaheuristic method for solving SGP instances to date, and the challenge therefore consists in matching their results as closely as possible.

6.5 A new GRASP for the SGP

In this section, we present a new greedy randomised adaptive search procedure (GRASP) for solving SGP instances. The basic structure of a GRASP for a minimisation problem as described by Feo and Resende in [FR95] is as follows:

1. $f^* = \infty$
2. Repeat until stopping criterion:
   (a) Generate a greedy randomised candidate solution x
   (b) Find local optimum x_l with local search starting from x
   (c) If $f(x_l) < f^*$ then
      (i) $f^* = f(x_l)$
      (ii) $x^* = x_l$

In our case, the stopping criterion is either a timeout, or the discovery of a conflict-free schedule.

6.5.1 The greedy heuristic

The greedy heuristic used in our GRASP scheme is based on the observations from Chapter 5. However, scheduling a pair of golfers with *minimal* freedom is exactly the *opposite* of what is appropriate when using the heuristic in combination with a local search method. Instead, we should try to *maximise* the freedom inside groups, to "make room" for good local moves. The intuition is that if freedom among the players of each group is high, the probability that swaps lead to new conflicts decreases.

Our greedy heuristic therefore proceeds as follows: Let us first suppose that p is even. Then the task of scheduling the players into groups in each week can also be seen as scheduling *pairs* of players into groups. To produce an initial configuration, the heuristic visits the weeks one after another. A single week is produced as follows: The week's groups are traversed one after another. For each pair of adjacent positions in a group, the heuristic needs to select a pair of players still remaining to be scheduled in the current week. It selects the pair having *maximal* freedom with respect to the current partial configuration. In addition, there is a parameter γ, with $0 \leq \gamma \leq 1$, that can be used to randomise the heuristic: In the case of ties, a random choice is made among the pairs of players having maximal freedom with probability γ. With probability $1 - \gamma$, pairs are regarded as ordered, with the numerically smaller player first, and the lexicographically smallest pair is selected. After a pair of players was selected and is placed into a group, a large number is subtracted as a penalty from that pair's freedom in further weeks, to discourage that pair from being selected again in a different group.

Other than that, the heuristic pays no attention to potential conflicts in a group, and never undoes a choice of pairs.

The remaining case is when p is odd. Here, the heuristic can still work with pairs of players, except for the last player in each group. With probability γ, that player is randomly selected from all players that are still remaining to be scheduled in that week. With probability $1-\gamma$, the numerically smallest remaining player is selected.

The heuristic is readily generalised from pairs to larger sets of players, although there is a clear trade-off between maximising freedom of groups and efficiency.

6.5.2 The local search component

We aimed to keep the local search component as simple as possible. We chose Algorithm 6.2 as our basis, and then simplified it further. In particular, we eliminated the restart component, since the search is now restarted within the more general GRASP scheme. Also, based on experiments with different lengths of tabu lists, we fixed the length of tabu lists to 10 instead of imposing random limits. We also added random noise: If there was no improvement for 4 iterations, two random swaps are made.

6.5.3 Experimental results

Our implementation consists of two programs: The first one, written in Prolog, generates initial configurations for given g, p, w and γ according to our greedy heuristic. As in Section 5.3, we benefit from arbitrary precision arithmetic to represent sets of players as bit vectors. In fact, we could reuse the existing greedy heuristic from that section with very few modifications.

The second program, written in C++, is the local search component. It is started with parameters g, p, w, and can also read an initial configuration.

We executed 10 runs for each instance $g-p-w$ presented here, using an Apple MacBook with a 2.16 GHz Intel Core 2 Duo CPU and 1GB RAM. Conceptually, a run of a single instance proceeds as follows: First, the greedy heuristic is used to generate five initial configurations with varying γ, including 0, 0.1, 0.2, and two values drawn at random between 0.3 and 1. The time it takes to generate an initial configuration for the benchmark instances is negligible, i.e., at most half a second. Then the local search component is run with each of those starting configurations for at most one minute. If no solution is found in these tries, the search is restarted with that initial configuration that yielded the minimum number of conflicts (and smallest γ, in the case of ties) while it was run, and it is then run to completion or until the timeout is reached.

We chose 20 minutes as the maximum running time of the algorithm, since this is also the limit used in the benchmarks of the memetic algorithm

in [CDFH06], with which we wanted to allow a fair comparison. Many instances took only a few seconds, and all but one of them consistently finished well within the 20 minutes time limit. The only exception was instance 10−6−7, which was solved only in two runs out of ten within the time limit. We considered an instance solved if a solution was found in at least one of the 10 runs we performed.

Our results are shown in Fig. 6.1. For various values of g and p, we show groups of three bars, which denote the maximum w such that the instance could be solved with (from left to right): Our GRASP scheme, the best memetic algorithm introduced in [CDFH06], and local search alone as reported in [DH05]. The latter values are similar to those we obtain if we run the local search component in isolation. In particular, we cannot even solve the 8−4−9 instance, let alone 8−4−10, without our greedy heuristic. For each instance, the thin horizontal lines show the (optimistic) upper bound and the best solution obtained with a mix of constraint-based formulations and basic design-theoretic techniques as collected in [Har02], respectively.

It is clear from these figures that our approach is highly competitive on other instances besides the original problem as well: On all tested instances, it finds solutions for as many weeks as the best variant of the memetic algorithm (surpassing it on 8−4−10 and 8−8−5), and surpasses plain local search and constraint-based solutions in many cases.

6.5.4 New solutions for the 8−4−10 instance

The 8−4−10 instance of the SGP is of particular interest due to two reasons: First, it is the optimal solution for "the" social golfer problem in the original sense, which is problem number 10 in CSPLib, a benchmark library for constraints ([GW99]). Second, being on the verge of solvability, the instance was previously thought to be amenable only to design theoretic techniques and constraint solvers due to its highly constrained nature. However, even the most sophisticated constraint solvers are currently unable to solve the instance. In contrast, by using the GRASP scheme outlined in this section, solutions for this instance are readily generated. Two such solutions are depicted in Fig. 6.2. They were found by varying the randomisation factor of the greedy heuristic. Computation time was 11 and 4 minutes, respectively. We used McKay's `dreadnaut` program ([McK90]) on the extended Levi graphs (see Section 2.11) of the two solutions to verify that they are not isomorphic. The fact that we could obtain structurally different solutions is an indication that the greedy heuristic is meaningful and does not work only by accident. Both solutions are new in the sense that they are not isomorphic to Aguado's solution ([Agu04], see also Section 2.12).

It is left to explain *why* the greedy heuristic works so well on the 8−4−10 instance. While we are currently unable to give an analytical justification, we believe that observing the solution process can give an important indication:

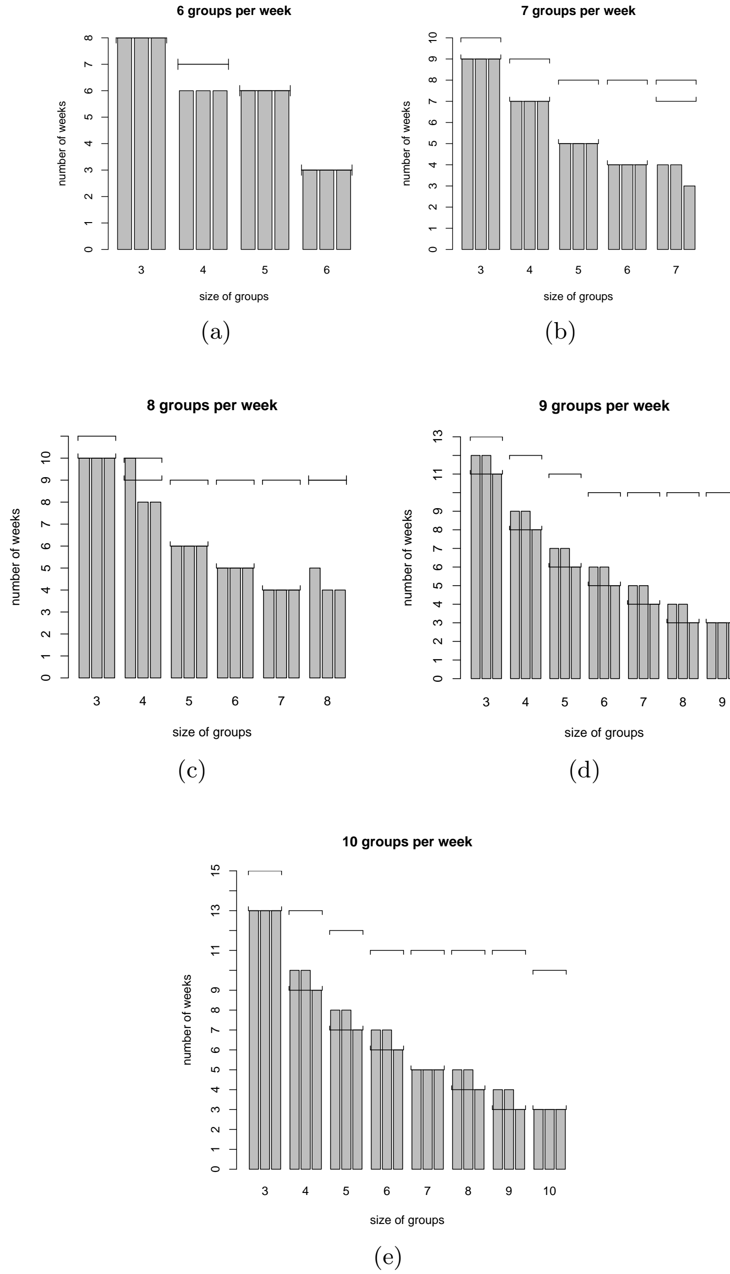


Figure 6.1: Solved number of weeks for g equal to (a) 6, (b) 7, (c) 8, (d) 9 and (e) 10, with various values of p. Each group of three bars represents the maximum w obtained by (from left to right): Our GRASP, the best memetic approach proposed in [CDFH06], and local search alone from [DH05]. The thin horizontal lines show the best w found with constraint solvers ([Har02]) and optimistic upper bounds, respectively.

	Week 1	Week 2	Week 3	Week 4	Week 5
Group 1	3 7 26 1	3 24 0 27	15 24 29 4	22 7 30 15	27 13 6 15
Group 2	15 11 9 18	15 8 20 19	14 8 11 13	23 31 14 25	4 25 18 16
Group 3	20 13 16 22	9 21 17 13	9 19 30 10	18 26 5 13	14 24 26 12
Group 4	23 21 19 0	11 16 23 12	5 3 6 31	4 19 27 12	0 11 29 22
Group 5	28 24 5 30	6 29 30 26	17 20 23 18	1 24 16 9	17 5 7 19
Group 6	27 25 29 17	18 22 10 14	12 1 22 21	8 0 17 6	8 30 3 21
Group 7	14 6 2 4	5 25 1 2	26 28 25 0	2 10 29 21	2 31 9 20
Group 8	31 12 10 8	7 28 4 31	7 16 27 2	28 3 11 20	28 10 1 23

	Week 6	Week 7	Week 8	Week 9	Week 10
Group 1	19 6 1 11	3 16 19 14	22 9 25 6	18 8 1 27	4 20 1 0
Group 2	24 21 31 18	1 13 29 31	28 29 19 18	15 3 25 10	8 2 28 22
Group 3	3 22 4 17	11 7 21 25	21 26 27 20	17 24 2 11	18 6 12 7
Group 4	0 10 7 13	20 24 6 10	0 16 31 15	20 14 29 7	24 19 13 25
Group 5	5 8 29 16	17 15 28 12	1 17 14 30	0 5 9 12	3 23 9 29
Group 6	25 12 20 30	0 30 2 18	4 11 10 5	22 31 19 26	16 17 26 10
Group 7	9 28 14 27	23 22 27 5	13 12 2 3	4 30 23 13	31 27 30 11
Group 8	2 26 15 23	9 4 26 8	23 7 24 8	16 21 28 6	14 5 15 21

	Week 1	Week 2	Week 3	Week 4	Week 5
Group 1	19 9 3 8	24 2 30 4	9 15 0 27	16 0 25 24	15 17 18 12
Group 2	28 25 18 5	18 13 10 8	7 28 4 31	17 23 31 8	19 24 5 14
Group 3	4 1 15 6	20 15 19 21	25 26 2 1	5 26 13 3	7 11 26 8
Group 4	14 11 20 0	1 29 5 7	10 5 12 30	30 6 7 19	25 22 4 21
Group 5	13 23 30 21	23 12 11 16	20 6 3 17	18 22 20 1	3 29 0 30
Group 6	16 10 29 22	26 6 28 0	24 29 23 18	11 2 15 10	13 6 16 27
Group 7	7 2 27 17	3 31 27 25	14 8 16 21	21 12 9 28	28 10 23 1
Group 8	31 26 12 24	14 17 9 22	11 22 19 13	29 4 14 27	2 9 20 31

	Week 6	Week 7	Week 8	Week 9	Week 10
Group 1	0 5 2 8	20 10 27 24	14 18 31 30	17 11 24 28	1 30 11 9
Group 2	11 29 6 31	7 22 12 0	20 23 7 25	9 25 13 29	6 12 14 25
Group 3	28 27 22 30	8 15 30 25	9 4 10 26	30 16 20 26	24 13 7 15
Group 4	15 26 23 14	23 6 9 5	22 6 8 24	27 12 1 8	21 31 0 10
Group 5	19 10 17 25	11 18 3 4	19 12 2 29	31 22 5 15	28 20 29 8
Group 6	12 13 4 20	26 29 21 17	28 15 3 16	2 18 21 6	4 17 16 5
Group 7	18 9 7 16	14 13 28 2	27 11 5 21	19 23 0 4	26 27 19 18
Group 8	1 24 21 3	16 19 31 1	13 0 1 17	14 10 7 3	2 23 22 3

Figure 6.2: Two new non-isomorphic optimal solutions for the original SGP, instance 8−4−10

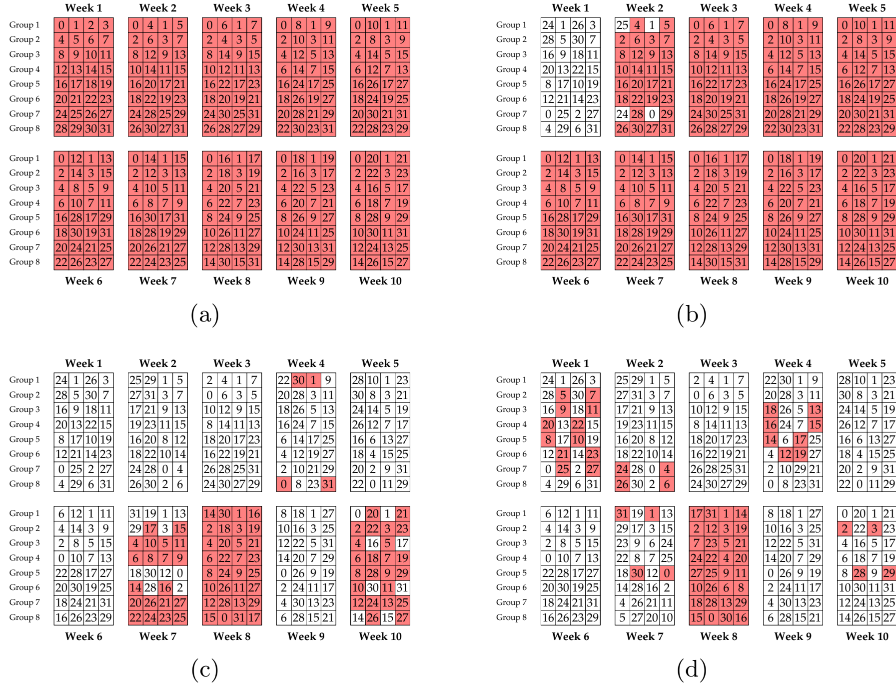


Figure 6.3: Instance 8−4−10, local search with our greedy initial configuration ($\gamma = 0$). Conflicts after (a) 0, (b) 10, (c) 100, (d) 500 iterations.

Fig. 6.3 shows different states of the local search component starting from a greedy initial configuration with $\gamma = 0$, with conflict positions highlighted. Notice that initially, *every* position is a conflict position.

For comparison, Fig. 6.4 shows what happens when the search is started from the trivial initial configuration of simply lining up the players in order for each week, with which we could solve the original SGP instance for a maximum of only 8 weeks. When contrasting the distribution of conflicts in the two figures, one effect of the greedy initial heuristic is apparent: Conflicts become more concentrated, and several weeks become conflict-free very early. In contrast, with a bad initial heuristic (Fig. 6.4), remaining conflicts are dispersed throughout all weeks. We believe that Fig. 6.3 gives a valuable suggestion about how the SGP could be successfully approached with a completely different local search method, which explicitly encodes a behaviour that is similar to the one found in this case. For example, one could build conflict-free groups incrementally, while exchanging players in existing weeks or dropping already built groups on occasion. Another strategy could be to build several conflict-free weeks first, and then to only consider swaps in other weeks for a certain number of iterations.

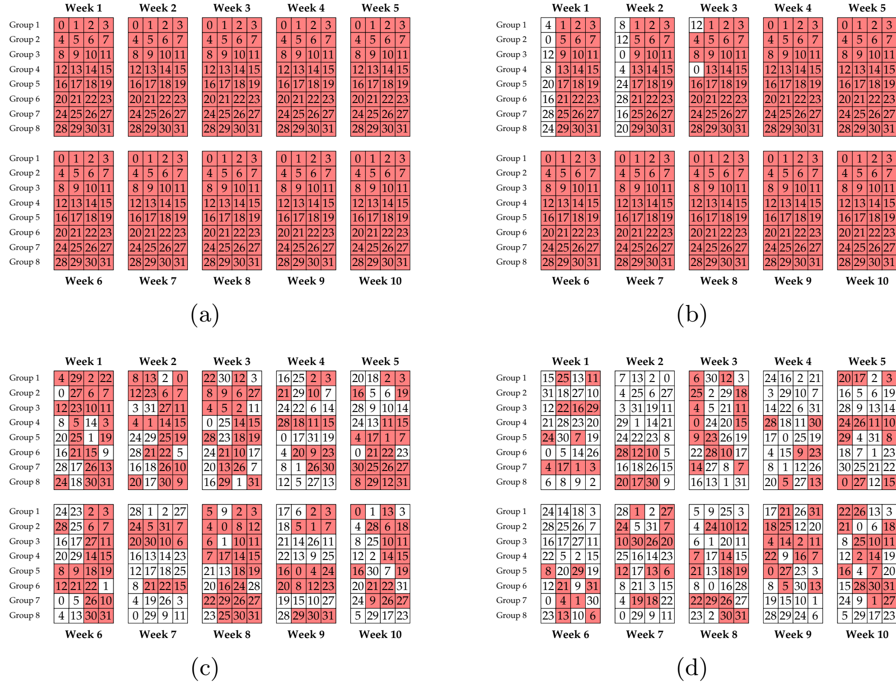


Figure 6.4: Instance 8−4−10, local search using a trivial initial configuration. Conflicts after (a) 0, (b) 10, (c) 100, (d) 500 iterations.

6.6 Conclusion

We have described an existing local search approach for solving SGP instances, which avoids many of the invalid intermediate configurations that can arise when solving SGP instances with Walksat.

We have presented a GRASP scheme for the SGP, in which we used a simplified version of the existing algorithm as the local search component. As greedy heuristic, we used the *opposite* of the greedy heuristic presented in Section 5.3, which is based on the concept of *freedom* of sets of players. Our heuristic is readily randomised and generalised, and was shown to improve results obtained by local search alone. In particular, we have obtained new solutions for the 8−4−10 instance. This makes our approach the first metaheuristic method that solves the original problem optimally, and also surpasses current constraint solvers on this instance. In addition, our approach is among the simplest and was shown to be highly competitive with other metaheuristic and constraint-based techniques on other instances as well, even though it does not take symmetries into account at all.

Metaheuristic approaches can easily cope with constraint variations by changing the objective function, but are typically incomplete and therefore cannot be used to show that an instance cannot be solved.

7 Conclusion and future work

We have presented and discussed the most prominent existing approaches for solving SGP instances, which are design theoretic techniques, SAT encodings, constraint-based approaches and metaheuristics methods. Each of these methods was shown to have advantages and limitations, and several interesting opportunities for future research present themselves.

Design theoretic techniques (Chapter 2) are fast and powerful when applicable, but there is currently no general deterministic method that can solve any given SGP instance or show that it cannot be solved. Also, deterministic construction methods typically cannot handle even slight variations of constraints, or partially instantiated schedules. As the SGP is closely related to finite geometries, Galois fields, and other objects from different branches of discrete mathematics, new results from these areas could also provide further insights into the SGP.

SAT encodings (Chapter 3) are attractive for a number of reasons: SAT solvers are often freely available and have improved continuously in recent years. There are *complete* SAT solvers, such as SATO, which can be used to show that an instance cannot be solved. Using our SAT formulation, we have solved the original SGP instance for 7 weeks. This is not competitive with other approaches that we have presented. However, there is hope that alternative problem formulations or symmetry breaking constraints exist, which could further reduce computation time. In addition, SAT formulations might provide new insights into SGP instances due to their simplicity. By analysing SAT encodings of SGP instances that are known to be unsolvable, one could arrive at better bounds and hardness estimates of other instances, about which only very little is known so far. The tools we developed as part of this thesis make working with SAT encodings less error-prone.

Several similar conclusions can be drawn for constraint programming formulations (Chapter 4): Constraint solvers are becoming increasingly more powerful as new propagation algorithms are discovered, and existing problem formulations automatically benefit from such improvements. Constraint-based solutions are typically *complete* and can thus also be used to show that an instance cannot be solved. The original SGP instance has generated much interest from the CP community, but so far no constraint solver was able to solve the instance optimally. Solving this instance remains a challenging problem for authors of constraint solvers, and further improvements of propagation algorithms and variable selection strategies seem to be necessary to solve the instance optimally with constraint-based methods. A new free constraint solver that we developed lets users experiment with an advanced existing CLP(FD) formulation of the SGP.

Metaheuristic methods (Chapter 6) are typically incomplete and therefore cannot be used to show that an instance cannot be solved. However, they are easily adapted to slight constraint variations and are very competi-

tive with constraint-based approaches on many instances. By combining an existing local search method with a new greedy heuristic (Chapter 5), we have obtained new solutions for the original SGP instance, $8-4-10$. Greedy heuristics for the SGP have not received much attention in the literature so far, and better heuristics might still be discovered.

It is especially interesting to see how various approaches can benefit from each other, of which we saw several examples: Symmetry breaking constraints used in SAT encodings can also be of use in CLP formulations, and constructions from design theory can give useful initial configurations for metaheuristic methods. It would also be interesting to use schedules built with our greedy heuristic as initial configurations for SAT-based approaches.

A Creating portable animations

We include some of the PostScript definitions that we used to create animations and static pictures for presentation purposes. In addition to making our figures and results completely reproducible, these definitions can be very helpful during development. It is our hope that they provide a useful starting point for others too. At the very least, they show that obtaining customised animations of search processes need not come at great expense, and can be done in a highly transparent and portable way. As we have shown on several occasions in this thesis, observing animations of solution processes can give valuable suggestions for alternative strategies in addition to being interesting and useful in its own right.

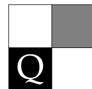
Fig. A.1 shows the definitions used to visualise the constraint solving process for N-queens. Fig. A.2 (a) shows an example of its usage and Fig. A.2 (b) shows the resulting picture.

```
1  /init {  /N exch def 322 N div dup scale -1 -1 translate
2      /Palatino-Roman findfont 0.8 scalefont setfont
3      0 setlinewidth
4      1 1 N { 1 1 N { 1 index c } for pop } for } bind def
5  /showtext { 0.5 0.28 translate
6      dup stringwidth pop -2 div 0 moveto 1 setgray show} bind def
7  /i { gsave translate 0.5 setgray 0 0 1 1 4 copy rectfill 0 setgray
8      rectstroke grestore } bind def
9  /q { gsave translate 0 0 1 1 rectfill (Q) showtext grestore }
10    bind def
11 /c { gsave translate 1 setgray 0 0 1 1 4 copy rectfill 0 setgray
12      rectstroke grestore } bind def
```

Figure A.1: PostScript definitions used to visualise N-queens

```
2 init
1 1 q
1 2 q
1 2 c
2 2 i
```

(a)



(b)

Figure A.2: (a) PostScript instructions and (b) the resulting picture

Fig. A.3 shows the definitions used to visualise the constraint solving and local search process for the SGP. Fig. A.4 shows an example of its usage and the resulting picture.

The intended usage is that such instructions are generated by a program and directly fed to a running PostScript viewer to see the animation in real-time. If PostScript language level 2 is enabled via

```
systemdict /.setlanguagelevel known { 2 .setlanguagelevel} if
```

then `copypage` can be used to update the animation when necessary.

```
1  /maxplayers 21 def
2
3  /showtext { 0.5 0.25 translate
4      dup stringwidth pop -2 div 0 moveto 0 setgray show } bind def
5
6  /groups { gsave lowerleft -3 G 0.35 add moveto
7       1 1 G { 0 -1 rmoveto gsave 0.7 dup scale (Group ) show
8         5 string cvs show grestore } for grestore } bind def
9
10 /init { /W exch def /P exch def /G exch def
11     500 maxplayers P idiv dup P mul add 1 sub div dup scale
12     /Palatino-Roman findfont 0.8 scalefont setfont
13     4 2 translate 0 setlinewidth 1 maxplayers P idiv W { groups } for
14     1 1 W { gsave dup lowerleft P 2 div
15      1 index maxplayers P idiv le { G 0.4 add } { -1 } ifelse translate
16      /Palatino-Bold findfont 0.75 scalefont setfont
17      5 string cvs (Week ) dup stringwidth pop 2 index stringwidth pop
18      add -2 div 0 moveto show show grestore } for
19     1 1 G { 1 1 P { 1 1 W { 3 copy c pop } for pop } for pop } for
20 } bind def
21
22 /lowerleft { dup P mul maxplayers le
23     { G 1 add } { maxplayers P idiv sub 0 } ifelse
24     exch 1 sub dup P mul add exch translate } bind def
25
26 /gcoords { lowerleft 1 sub exch G sub neg translate } bind def
27 /r { 1 setgray 0 0 1 1 4 copy rectfill 0 setgray rectstroke } bind def
28 /g { gsave gcoords r showtext grestore } bind def
29 /c { gsave gcoords r grestore } bind def
30 /p { gsave gcoords 1 0.5 0.5 setrgbcolor 0 0 1 1 4 copy
31      rectfill 0 setgray rectstroke showtext grestore } bind def
```

Figure A.3: PostScript definitions used to visualise the SGP

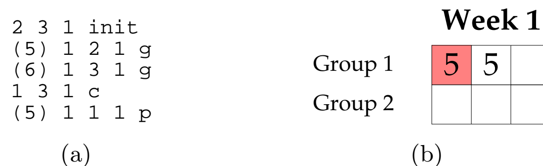


Figure A.4: (a) PostScript instructions and (b) the resulting picture

B Bibliography

References

[Agu04] Alejandro Aguado. A 10 days solution to the social golfer problem. *Manuscript*, 2004.

[And97] Ian Anderson. *Combinatorial designs and tournaments*. Oxford Clarendon Press, 1997.

[Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS), Prague, Czech Republic*, 1999.

[Bar01] Roman Barták. Filtering algorithms for tabular constraints. In *Proceedings of CP2001 Workshop CICLOPS*, 2001.

[BB05] Nicolas Barnier and Pascal Brisset. Solving Kirkman's schoolgirl problem in a few seconds. *Constraints*, 10(1):7–21, 2005.

[BR49] R. H. Bruck and H. J. Ryser. The nonexistence of certain finite projective planes. *Canad. J. Math.*, 1:88–93, 1949.

[Car05] Mats Carlsson. CLP(FD) formulation for the SGP, SICStus demonstration program, 2005.

[CD96] C. H. Colbourn and J. H. Dinitz. *The CRC Handbook of Combinatorial Designs*. CRC Press, 1996.

[CDFH06] Carlos Cotta, Iván Dotú, Antonio J. Fernández, and Pascal Van Hentenryck. Scheduling social golfers with memetic evolutionary programming. In *Hybrid Metaheuristics*, volume 4030 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2006.

[CGLR96] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159, 1996.

[Col84] Charles J. Colbourn. The complexity of completing partial latin squares. *Discrete Appl. Math.*, 8:25–30, 1984.

[Col99] Charles J. Colbourn. A Steiner 2-design with an automorphism fixing exactly $r + 2$ points. *Journal of Combinatorial Designs*, 7:375–380, 1999.

[Coo71] Stephen A. Cook. The complexity of theorem proving procedures. In *STOC71*, pages 151–158, 1971.

[DH05] Iván Dotú and Pascal Van Hentenryck. Scheduling social golfers locally. In *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 155–167. Springer, 2005.

[DL02] Mireille Ducassé and Ludovic Langevine. Automated analysis of CLP(FD) program execution traces. *Lecture Notes in Computer Science*, 2401, 2002.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[Dou94] Douglas R. Stinson. Universal hashing and authentication codes. *Designs, Codes and Cryptography*, 4:369–380, 1994.

[dSC06] Anderson Faustino da Silva and Vítor Santos Costa. The design and implementation of the YAP compiler: An optimizing compiler for logic programming languages. In *ICLP*, volume 4079 of *LNCS*, pages 461–462. Springer, 2006.

[Ert90] M. Anton Ertl. Coroutining und Constraints in der Logik-Programmierung. Diplomarbeit, Technische Universität Wien, Austria, 1990. In German.

[Eul82] Leonhard Euler. Recherches sur une nouvelle espèce de quarrés magiques. *Verh. Zeeuws Gen. Weten. Vlissingen*, 9:85–239, 1782.

[FFH+02] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. *Lecture Notes in Computer Science*, 2470, 2002.

[FHK+02] Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Global constraints for lexicographic orderings. *Lecture Notes in Computer Science*, 2470, 2002.

[FR95] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *J. of Global Optimization*, 6:109–133, 1995.

[FSC04] François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI: A generic graphical user interface for constraint logic programming. *Constraints*, 9(4):241–262, 2004.

[Gal85] Hervé Gallaire. Logic programming: Further developments. In *SLP*, pages 88–96, 1985.

[GKP95] Daniel Gordon, Greg Kuperberg, and Oren Patashnik. New constructions for covering designs. *Journal of Combinatorial Designs*, 4:269–284, 1995.

[GL05] I.P. Gent and I. Lynce. A SAT encoding for the social golfer problem. *IJCAI'05 workshop on Modelling and Solving Problems with Constraints*, 2005.

[GW99] Ian P. Gent and Toby Walsh. CSPLib: A benchmark library for constraints. In *5th Int. Conf. on Principles and Practice of Constraint Programming*, 1999.

[Har02] Warwick Harvey. Warwick's results page for the social golfer problem. *http://www.icparc.ic.ac.uk/∼wh/golf/*, 2002.

[HBC70] M. Y. Hsiao, D. C. Bossen, and R. T. Chien. Orthogonal Latin square codes. *IBM Journal of Research and Development*, 14(4):390–394, 1970.

[HE80] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[HW05] Warwick Harvey and Thorsten Winterer. Solving the MOLR and social golfers problems. In *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 2005.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL*, pages 111–119, 1987.

[Kir47] T.P. Kirkman. On a problem in combinations. *Cambridge and Dublin Math. J.*, 2:109–204, 1847.

[LR97] C. C. Lindner and C. A. Rodger. *Design theory.* CRC Press, 1997.

[LTS86] C. W. H. Lam, L. Thiel, and Stan Swiercz. The nonexistence of code words of weight 16 in a projective plane of order 10. *J. Comb. Theory, Ser. A*, 42(2):207–214, 1986.

[Lyn05] I. Lynce. A SAT encoding for the social golfer problem: Benchmark description. *8th International Conference on Theory and Applications of Satisfiability Testing (SAT0'05)*, 2005.

[McK90] Brendan McKay. Nauty user's guide (version 1.5). Technical report, Dept. Comp. Sci., Australian National University, 1990.

[MS00] Joao P. Marques-Silva. Algebraic simplification techniques for propositional satisfiability. In *Proc. of the 6th Int. Conf. on Principles and Practice of Constraint Programming*, 2000.

[NRS97] Ulrich Neumerkel, Christoph Rettig, and Christian Schallhart. Visualizing solutions with viewers. In *Workshop on Logic Programming Environments*, pages 43–50, 1997.

[PG83] Paul Pritchard and David Gries. The seven-eleven problem. Technical Report TR83-574, Cornell University, Computer Science Department, September 1983.

[Pre01] Steven Prestwich. First-solution search with symmetry breaking and implied constraints. In *Proceedings of the CP'01 Workshop on Modelling and Problem Formulation*, 2001.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, pages 362–367, 1994.

[SF94] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI*, pages 125–129, 1994.

[Sho94] V. Shoup. Fast construction of irreducible polynomials over finite fields. *Journal of Symbolic Computation*, 17(5):371–391, 1994.

[SKC93] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *Second DIMACS Implementation Challenge*, 1993.

[Sut63] I. E. Sutherland. SKETCHPAD: A Man-Machine Graphical Communications System. Technical Report 296, MIT, 1963.

[Tar00] G. Tarry. Le problème de 36 officiers. *Compte Rendu de l'Assoc. française pour l'avanc. des sciences*, 1:122–123, 1900.

[Wal75] D. L. Waltz. Understanding line drawings of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, New York, 1975.

[Wie03] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In *Proc. of the 13th Int. Workshop on Logic Prog. Environments*, pages 1–16, December 2003.

[WNS97] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, London, 1997.

[Yat36] F. Yates. Incomplete randomized blocks. *Ann. Eugenics*, 7:121–140, 1936.

[Zha97] H. Zhang. SATO: An efficient propositional prover. *Lecture Notes in Computer Science*, 1249, 1997.

Index

$C(\sigma)$, 64
$N(n)$, 16
$P_C(x)$, 58
$S^{-}(\sigma)$, 64
$\#_\sigma(a,b)$, 63
γ, 66
$\langle i,j,k\rangle$, 64
$\mathcal{S}^*(\sigma,\sigma^*)$, 64
σ, 63
$\sigma[x_1 \leftrightarrow x_2]$, 64
$\varphi_C(S)$, 58
$f(\sigma)$, 64
$m[a,b]$, 63
(0,1)-design, 7
7-11 problem, 38

affine plane, 11
all_different, 39, 49
animations, 32
aspiration, 64

backtracking, 11, 18, 30, 45, 48, 59
BIBD, 6
block, 6
Bus error, 31

CLP, 37
CLP(FD), 38
comp.constraints, 19
comp.lang.prolog, 48
complete, 16, 55, 73
completion problem, 14
computation time, 41
conflict position, 64
consistency, 40
constraint
  propagation, 40
  solver, 37, 40
constraints, 37
CP, 37, 73
CSP, 37, 41, 42

design, 6
DIMACS, 32
domain, 37
dreadnaut, 18, 68

ECLiPSe, 37, 45, 48, 49
Euclidean
  parallel axiom, 11
  plane, 11
Euler's officer problem, 3, 15

factorisation, 38
Fano plane, *see* seven-point plane
finite
  affine plane, 11
  geometry, 11
  incidence structure, 11
  projective plane, 10, 11
first-fail, 42, 59
flip, 62
freedom, 58, 66

Galois field, 16, 73
GF(n), *see* Galois field
global consistency, 40
Graeco-Latin squares, 15
graph, 17, 40
GRASP, 66
GSAT, 62
GUPU, 43

Herbrand terms, 37

implied constraint, 50
incidence structure, 11
inconsistent, 39
inner node, 41, 42
instantiation order, 41

Kirkman triple system, *see* KTS
Kirkman's schoolgirl problem, 10, 31, 33, 59
KTS, 10

labeling, 40
ladder matrix, 22, 23
lamarckian learning, 65
Latin square, 12
leaf, 42
Levi graph, 17, 68
logic programming, 37

memetic algorithm, 65
MOLS, 15
multiplication table, 49
mutually orthogonal, 15

N-queens, 44, 75
NP, 14, 21
NP-complete, 14, 21

order, 11
orthogonality, 15

parallel
    class, 8
    lines, 11
point, 11
PostScript, 44, 75, 76
prime factor, 38
projective plane, 11
Prolog, 33, 37, 43, 51, 59, 67
propagation, 40, 43

quadrangle, 11
queens, 44, 75

random
    noise, 62, 67
    walk, 62
reified constraint, 44
relation, 37
resolution, 8
resolution class, 8
resolvability, 8
resolvable, 10

SAT, 21, 62
SATO, 28, 31, 73
scene labelling, 37
sci.op-research, 3, 18
seven-eleven problem, 38
seven-point plane, 7, 10
SGP, 3
SICStus Prolog, 45, 48, 51
singleton set, 40
social golfer problem, 3
socialisation, 22, 25, 27, 49
Steiner triple system, 8, 10
STS, 8, 10
subtrees, 42
Sudoku
    critical set, 39
    Latin square, 39
    puzzle, 39
swap, 64
SWI-Prolog, 51
symmetry, 28, 48, 50
symmetry breaking, 27

table, 49
tabu
    list, 64, 67
    search, 65
tournament scheduling, 8

value selection, 43
visualisations, 43

Walksat, 31, 32, 62, 63

YAP, 51