Solving Frequency Assignment Problems with Constraint Programming

Diplomarbeit
bei Prof. Dr. M. Grötschel

vorgelegt von
Mathias Schulz

am Institut für Mathematik der
Technischen Universität Berlin

betreut von
Dr. Andreas Eisenblätter

Berlin, Februar 2003

Abstract

The topic of this diploma thesis is the solving of a Frequency Assignment Problem (FAP) in GSM radio networks by means of Constraint Programming. The task of frequency planning is the assignment of frequencies to base stations w.r.t. various constraints such that interference, which may have a substantial impact on the quality of the received signals, is avoided as far as possible.

Constraint Programming is employed since currently applied heuristics aimed at interference minimization often fail for problems where it is problematic to obtain any feasible assignment. First, the focus is on feasibility problems only. Having determined valid solutions, it is also tried to explicitly minimize total interference.

After the introduction of the underlying mathematical model, it is shown that the discussed version of FAP is strongly $\mathcal{NP}$-hard. An overview on the theory of Constraint Programming and a proof that Constraint Satisfaction Problems are strongly $\mathcal{NP}$-complete are provided as well.

ILOG OPL Studio is employed for solving FAP, where the "Optimization Programming Language" (OPL) allows to state mathematical models using an own modeling language. It is investigated to what extent the additional elements of this language compared to Mixed Integer Programming really enable to express more conditions in OPL models than in Mixed Integer Programs.

In this work, various OPL models for FAP feasibility problems are presented. Different modeling alternatives are analyzed and compared with each other. By means of OPL feasibility models, it is possible to obtain valid assignments for large instances as well as for instances which are difficult to solve. As the quality of the determined solutions is often not satisfactory, approaches on explicitly minimizing total interference are introduced. A minimization framework by means of OPL is developed, a construction heuristic is applied, and OPL feasibility models are combined with an improvement heuristics. Furthermore, the results for all these minimization techniques are compared with each other.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

The most common technology for mobile telecommunication is nowadays the General System for Mobile Communications *GSM*. Currently, more than half a billion customers in more than 150 countries worldwide are using the GSM standard.

A high connection quality is a major competitive edge for telecommunication providers. One of the most important problems for radio network operators is the limited availability of frequencies. A high level of interference, particularly due to an unavoidable reuse of frequencies, can have a substantial impact on the quality of the received signals; sometimes a proper reception may even be impossible. Frequency planning deals with assigning frequencies to base stations such that interference is avoided as far as possible. In this thesis, we deal with frequency planning problems especially arising in GSM networks.

A communication link between a mobile and other parties reachable through a public telecommunication network is established by means of a radio link to some stationary antenna which is part of a large infrastructure. Each radio network provider acquires a certain frequency *spectrum* from a national regulation authority. This spectrum is slotted equidistantly into so-called *channels* which are the available channels of the company. For each stationary antenna, a certain number of transmitter/receiver units *(TRXs)* is installed, where one channels has to be assigned to each of these TRXs. Since the number of employed TRXs is typically much larger than the number of available channels, the reuse of frequencies cannot be avoided. It is not uncommon that the same channel is operated by several hundred TRXs in the network.

The reuse of frequencies is limited by interference and separation requirements. Significant interference may occur if the same channel (co-channel) or if directly neighboring channels (adjacent channels) are assigned to different TRXs. The level of interference depends on a lot of factors as different as the distance between both transmitters, the power of the signals, geographical and vegetational aspects, and weather, to name only a few. Separation requirements are defined for pairs of transmitters and enforce a minimum separation in the electromagnetic spectrum between the channels assigned to both TRXs. Furthermore, not each channel of the spectrum may be available for every TRX.

The problem of assigning channels to TRXs while taking additional requirements into account is called the *Frequency Assignment Problem (FAP)*. Our optimization goal is to minimize the sum of all co- and adjacent channel interferences, but other objectives are possible.

In the early nineties, frequency planning in GSM networks was often performed manually due to a lack of adequate, commercially available software tools. Because of the enduring growth of network installations in the middle of the nineties, this practice soon reached its limits. New software for automatic frequency planning has been developed, and since the end of the nineties, significantly improved planning tools are successfully employed. However, further capacity extensions and particularly the introduction of new technologies like GPRS or HSCSD make frequency planning be more and more important again. Thereby, even determining any feasible solution of the Frequency Assignment Problem becomes problematic.

Currently employed heuristics like improvement methods often fail to solve problems when it is difficult to calculate any valid solution since these heuristics rely on the ability of easily obtaining feasible assignments. *Constraint Programming* is said to be well suited especially for solving problems where feasibility is a limiting factor.

Among others, Constraint Programming has been applied to assignment problems, i.e., problems where one kind of resources has to be assigned to another kind of resources while respecting additional constraints. For instance, Rossi [25] reports that the Hong Kong container harbor employed Constraint Programming to allocate berths for container ships. A promising article by Voudouris and Tsang [26] deals with solving the Radio Link Frequency Assignment Problem, which is a problem related to our version of FAP, in military telecommunication networks. They report on obtaining very good results concerning both performance and quality of the determined solutions.

Hence, we decided to employ Constraint Programming for the described Frequency Assignment Problem in GSM networks, while we are especially interested in obtaining solutions for problems where it is difficult to determine any feasible solution. First, we consider feasibility problems only. Subsequently, we also try to explicitly minimize total interference.

Many different constraint solvers are available, including ILOG Solver which is an object-oriented C++ library offering Constraint Programming algorithms. We use ILOG OPL Studio which is built on top of ILOG Solver (among others). The "Optimization Programming Language" (OPL) allows to formulate mathematical models by means of an own modeling language, while ILOG OPL Studio provides possibilities to solve models stated in OPL.

We investigate whether the additional features of the modeling language of OPL compared to Mixed Integer Programming, such as logical conditions or

minimum and maximum functions, provide more possibilities to formulate constraints in OPL models than in Mixed Integer Programs (MIPs). It is shown that under some restrictions, many of the additional elements of OPL can be expressed in MIPs as well, at the expense of additional variables and constraints.

Several OPL models for FAP feasibility problems are introduced. They allow to solve many instances, but the quality of the obtained solutions is not satisfactory. To overcome this, we develop a framework which uses OPL to explicitly minimize total interference. However, the quality of the assignments determined this way is often not much convincing. But a successful procedure proved to be to obtain any solution by means of an OPL feasibility model and to subsequently apply an improvement heuristic to the assignment.

The remaining of this document is organized as follows. Chapter 2 gives some background information on GSM networks and introduces our mathematical model for FAP. Furthermore, the complexity of FAP is studied and it is shown that our version of FAP is strongly $\mathcal{NP}$-hard. Chapter 3 deals with the theory of Constraint Programming. In particular, we show that Constraint Satisfaction Problems are strongly $\mathcal{NP}$-complete. In Chapter 4, an introduction to OPL and a comparison of the modeling capabilities of OPL models and MIPs are given. Chapter 5 presents various OPL models for FAP. Besides feasibility models, our approaches on minimizing total interference are introduced. Computational studies on feasibility as well as on minimization problems are reported in Chapter 6, while conclusions of our investigations are drawn in Chapter 7.

Chapter 2

The Frequency Assignment Problem (FAP)

In this chapter, we discuss the mathematical model of the Frequency Assignment Problem. Section 2.1 provides background information, Section 2.2 presents the model for FAP, while the complexity of FAP is investigated in Section 2.3.

2.1 Background

The topic of this section is the structure of GSM networks. Only the parts relevant for frequency planning and for our model are explained. For further information, in particular for information on the fixed part of the network, see [23].

Signal, speech, and data traffic between mobiles and the system is received and transmitted by means of Base Transceiver Stations (BTSs). One BTS typically operates three antennas. Several *transmitter/receiver units (TRXs)* can be installed for each antenna, see Figure 2.1. Each TRX offers capacity for about six to eight parallel (full rate voice) connections. This is done by means of time divisioning, where each subscriber gets only one out of the available eight time slots. A *site* is where a Base Transceiver Station is installed, the area that can be served by one antenna is called a *cell*. Some of the time slots of the TRXs within each cell are needed to broadcast protocol information. One TRX operates the *broadcast control channel (BCCH)*. The TRX itself rather than its assigned channel is also often called BCCH. Thus, in the rest of this thesis we refer to the TRX instead to its assigned channel as BCCH. Additionally installed TRXs within a cell are called *traffic channels (TCHs)*.

The reuse of frequencies within the network is unavoidable since the number of channels available for a network operator is significantly smaller than the number of installed TRXs (each requiring one channel). When assigning channels to TRXs, interference and separation requirements have to be taken into account. Two kinds of interference have to be considered between geographically close TRXs: *Co-channel interference* may occur if the same channel is

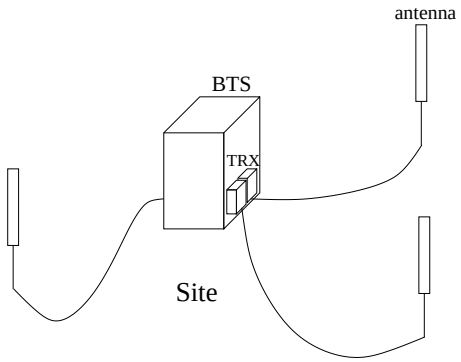


Figure 2.1: Sketch of a site within a GSM network

assigned to different TRXs, *adjacent channel interference* may occur if directly neighboring channels are assigned. Separation requirements impose a minimum separation between assigned channels in the spectrum. In particular, it is used to protect TRXs from interference: Co-channel interference is ruled out with a minimum separation of one, co- and adjacent channel interference are prevented with a minimum separation of two. There are several types of separation requirements: *Co-site separation* imposes restrictions on the TRXs within one site, *co-cell separation* restricts the assignment of TRXs within one cell, and *hand-over* separation is used to ease hand-overs of active calls between two cells. Co-channel interference, adjacent channel interference, and minimum separation are given for each pair of TRXs and need not be symmetric. In addition, it is possible that not each globally available channel is available within each cell. Such *locally blocked channels* may, for example, appear along national borders due to international agreements. Hence, for each cell a (possibly empty) list of locally blocked channels is specified.

One channel has to be assigned to each TRX such that no channel is locally blocked and all separation requirements are met. The optimization goal is to find a frequency assignment with the minimal possible interference. Several objectives stating what to minimize in detail are possible. Our objective is to minimize the sum over all occurring interferences. Further ones are conceivable since the sum over all interferences does not tell much about the interference level between two cells. Even if the overall interference is small, there can be a small number of cells suffering from a high level of interference.

Obtaining interference values between cells is complex [7, Section 2.3.2]. There are several approaches how to predict co-channel interference as well as adjacent channel interference. Consider on the one hand, two cells overlapping on a small region with a high level of interference and on the other hand, two cells overlapping on a large region with little interference. There is no general rule how to express this by means of a single value. Furthermore, interference often occurs between several cells rather than between pairs of cells only.

2.2 Our Model of FAP

In the following we restate the GSM frequency assignment problem described informally in the previous section, cf. [7, Section 3]:

> Given are a list of TRXs, a range of channels, a list of locally blocked channels for each TRX, as well as the minimum separation, the co-channel interference, and the adjacent channel interference matrices.
>
> Assign to every TRX one channel from the spectrum which is not locally blocked such that all separation requirements are met and such that the sum over all interferences occurring between pairs of TRXs is minimized.

For details on the transformation of the problem into a mathematical model, see [7, Section 3.1]; a sketch is given in the following. Let $G = (V, E)$ be an undirected graph. The nodes of G represent the TRXs and are also called *carriers*, the edges of G represent relations between pairs of TRXs. The spectrum C of available channels is a finite interval in $\mathbf{Z}_+$. For each carrier $v \in V$, a set $B_v \subset C$ of locally blocked channels is specified. The set $C \setminus B_v$ is called *the set of available channels* of v. Moreover, three functions $c^{co} : E \to [0, 1]$, $c^{adj} : E \to [0, 1]$, and $d : E \to \mathbf{Z}_+$ are given. They define the co-channel interference, the adjacent channel interference, and the minimum required separation.

The 7-tuple $N = (V, E, C, \{B_v\}_{v \in V}, c^{co}, c^{adj}, d)$ is referred to as *carrier network*. A *frequency assignment* is a function $y : V \to C$ that maps every carrier to a channel. As mentioned above, an assignment is feasible if and only if:

$$y(v) \in C \setminus B_v \quad \forall\, v \in V \tag{2.1}$$

$$|y(v) - y(w)| \geq d(v, w) \quad \forall\, (v, w) \in E \tag{2.2}$$

Constraint (2.1) enforces that only available channels are assigned to carriers, and requirement (2.2) imposes that channels assigned to neighboring carriers meet the minimum required separation. The objective is to minimize the sum over all co-channel and adjacent channel interferences:

$$\min_{y \text{ feasible}} \sum_{\substack{(v,w) \in E: \\ y(v) = y(w)}} c^{co}(v, w) + \sum_{\substack{(v,w) \in E: \\ |y(v) - y(w)| = 1}} c^{adj}(v, w) \tag{2.3}$$

2.3 Complexity of FAP

In this section, we focus on the computational complexity of FAP and show that FAP is strongly $\mathcal{NP}$-hard. Thus, unless $\mathcal{P} = \mathcal{NP}$, there is no polynomial time algorithm that solves this problem.

Theorem 2.1: FAP is strongly $\mathcal{NP}$-hard.

Proof. After restating the definition of strong $\mathcal{NP}$-hardness, we show that FAP is $\mathcal{NP}$-hard by means of a reduction from k-Colorability for undirected graphs to FAP. Then, we explain why FAP is strongly $\mathcal{NP}$-hard.

For a problem instance x let $max(x)$ be the value of the largest number occurring in x. In addition, for a given problem P and a polynomial p let $P^{max,p}$ denote the problem obtained by restricting P to only those instances x for which $max(x) \leq p(|x|)$, where $|x|$ is the coding length of x. A problem P is said to be strongly $\mathcal{NP}$-hard if a polynomial p exists such that $P^{max,p}$ is $\mathcal{NP}$-hard [1].

The problem k-Colorability for an undirected graph $G = (V, E)$ is to decide whether a function $c : V \to \{1, \ldots, k\}$ exists such that $c(u) \neq c(v)$ for each edge $e = (u, v) \in E$. Testing whether some graph is k-colorable is $\mathcal{NP}$-complete for any fixed $k \geq 3$, see for instance Garey and Johnson [11].

Given an arbitrary instance of k-Colorability, an instance of FAP can be constructed as follows. Without loss of generality it is assumed that $k \leq |V|$ since at most one color per node is needed. The graph for FAP is the same as for k-Colorability, the spectrum C is the set $\{1, \ldots, k\}$. There are no locally blocked channels, i.e., $B_v = \emptyset$ for all $v \in V$. For each edge $e \in E$, the value for co-channel interference $c^{co}(e)$ is set to 1, adjacent channel interference, and separation are set to zero.

An optimal solution $y : V \to C$ of FAP can be used to decide the problem k-Colorability: If the optimal solution value is equal to zero, then the solution of FAP is a solution for k-Colorability, too. The given graph is not k-colorable if the optimal solution value is greater than zero. This transformation can be done in polynomial time in the number of nodes and edges of G. Thus, FAP is $\mathcal{NP}$-hard.

FAP is strongly $\mathcal{NP}$-hard since it is $\mathcal{NP}$-hard even if $k \leq |V|$, $C = \{1, \ldots, k\}$, and if all edge weights are in $\{0, 1\}$. The largest occurring number in this instance is k and is less than the coding length of the instance. Hence, the polynomial $p(n) = n$ satisfies the above condition, which proves that FAP is strongly $\mathcal{NP}$-hard.

□

Chapter 3

Constraint Programming: Algorithms and Techniques

This chapter is about the theoretical aspects of Constraint Programming. Some basic definitions are provided in Section 3.1 and it is shown that Constraint Satisfaction Problems are strongly $\mathcal{NP}$-complete. Thus, (unless $\mathcal{P} = \mathcal{NP}$) one must resort to heuristics to solve these kind problems if one wants to obtain polynomial running times. Two popular approaches exist for solving a CSP: Systematic Search investigated in Section 3.2 and Consistency Techniques discussed in Section 3.3. Since both of them are often not well suited for solving CSPs when applied on their own, there is Constraint Propagation studied in Section 3.4, which is a combination of both of them. Finally, some hints for the construction of the search tree are given in Section 3.5.

3.1 Introduction

Constraint Programming (CP) deals with computational systems based on constraints [3]. Like Mathematical Programming, Constraint Programming handles solving problems whose variables are restricted.

Constraint Programming is also often called *Constraint Logic Programming (CLP).*

A class of problems with many practical applications are the so-called *Constraint Satisfaction Problems (CSPs).*

Definition 3.1: (CSP)
A *CSP* is given by:

- a finite set of variables $X = \{x_1, \ldots, x_n\}$,
- a finite set D_i of explicitly given, possible values (domain) for each variable x_i,
- a finite set of restrictions (constraints), each defined over some subset of the given variables.

A solution of a CSP is an assignment of one value from its domain to every variable satisfying each constraint on the variables. It is possible to search for one solution, for all solutions or for a specific solution minimizing or maximizing a given objective function.

As an example for a CSP, consider the variables $v_1, \ldots, v_3$ with the corresponding domains $D_1 = \{1\}$, $D_2 = \{1, 2\}$, $D_3 = \{4\}$, together the constraints $v_i \neq v_j$ for $i \neq j$. The solution of this CSP is $v_1 = 1$, $v_2 = 2$, and $v_3 = 4$.

In Constraint Programming the focus is on feasibility problems. However, minimizing or maximizing of an additionally given objective function is possible. This topic is called *Constraint Optimization*. There are two popular methods [19]. For *Standard Search*, an arbitrary feasible solution is computed in the beginning of the procedure. Iteratively, it is tried to obtain a better solution. Therefore a bound on the objective value is tightened in each step until no new solution is found for the first time. The last detected solution this way is then the optimal one. To apply *Dichotomic Search*, a lower bound on the objective function is needed (if considering minimization problems). Again, in the beginning some solution is computed. This one provides an upper bound on the objective. Roughly speaking, this technique is a binary search on the objective value whereby it is stressed to find feasible solutions.

Theorem 3.2: CSP is strongly $\mathcal{NP}$-complete.

Proof. After showing that solving CSP is in $\mathcal{NP}$, we prove that it is $\mathcal{NP}$-complete. This is done by means of a reduction from SAT to CSP. Solving CSP is $\mathcal{NP}$-complete since it is in $\mathcal{NP}$ and SAT, which is also $\mathcal{NP}$-complete (see for instances Garey and Johnson [11]), can be reduced to CSP. Finally, we demonstrate that solving CSP is strongly $\mathcal{NP}$-complete.

Any solution of a CSP is a certificate which can be used to verify that the CSP is satisfiable. The size of this certificate is polynomial in the size of the problem, and verification can be done in polynomial time in the number of constraints since it consists simply of assigning all variables and checking each constraint. Hence, solving CSP is in $\mathcal{NP}$.

An arbitrary instance of SAT is given as n binary variables and a conjunction of m terms. Each term is a disjunction of a number of literals, where a literals is either one of the n variables itself or the negation of one of the variables. The task is to determine an assignment of the variables such that all terms are satisfied.

Let l_{ij} denote the i-th literal of term j and m_j denote the number of literals of term j.

The CSP corresponding to the given SAT instance can be constructed straightforwardly. For each variable $v_1, \ldots, v_n$ of the SAT problem, a binary variable $V_1, \ldots, V_n$ for the CSP is introduced. Each term in SAT corresponds to one constraint in the CSP as follows:

$$\sum_{i=1}^{m_j} L_{ij} \geq 1, \quad j = 1, \ldots, m, \quad L_{ij} = \begin{cases} V_k & l_{ij} = v_k \\ 1 - V_k & l_{ij} = \overline{v}_k \end{cases}$$

In case the CSP has a solution, this is a solution for the given SAT problem, too. In case the CSP is not satisfiable, the SAT problem is not satisfiable as well. Without loss of generality it is assumed that each variable occurs only once in each term. In case a variable exists within one term once negated and once not negated, this term is always satisfied and need therefore not further be investigated. The truth of a term does not depend on whether the same literal occurs only once or several times within this term. Clearly, this transformation can be done in polynomial time in the number of terms and variables. Thus, CSP is $\mathcal{NP}$-complete.

Solving CSP is strongly $\mathcal{NP}$-complete since CSP is also $\mathcal{NP}$-complete even if in all constraints all occurring coefficients are at most 1.

□

A CSP is called *binary* if each constraint is unary or binary. This means that at most two variables are involved in any constraint. A binary CSP can be depicted as a *constraint graph* $G = (V, E)$. A node $v \in V$ is introduced for each given variable and an edge $e \in E$ is introduced for each given constraint. An edge e connects the variables involved in the constraint corresponding to e and simply means that both variables are restricted in any way. A label on e depicts the specific condition. Unary constraints are represented by loops within G.

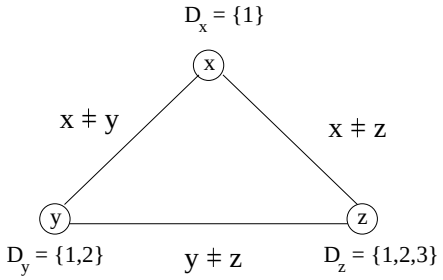


Figure 3.1: A simple constraint graph

Two methods are reported to convert a non-binary CSP into an equivalent binary problem: the *dual graph translation* and the *hidden variable translation*. Both methods introduce for each non-binary constraint one variable, where the former approach adds constraints between newly introduced variables only and the latter procedure introduces constraint which connect both original and new variables. These schemes are described, for instance, in [2]. However, there are also generalizations for the case of non-binary CSPs of the algorithms presented in the following which are based on binary constraints. For example, an adaption of the algorithm Forward Checking, which we discuss in Section 3.4.1, is presented in [2] as well. Since any non-binary problem can be converted into an equivalent binary problem, we examine binary CSPs for the rest of this chapter only.

3.2 Systematic Search

3.2.1 Techniques

A very simple approach is called *Generate and Test (GT)*. Each possible combination of variable assignments is consecutively generated and its validity is tested. This is obviously impractical for large instances.

Backtracking tries to overcome the simplicity of Generate and Test. A partial solution consistent with the constraints is stepwise extended by choosing a value for an unassigned variable that is consistent with the previous partial assignment. In case no consistent value can be found, backtracking is performed on the variable assigned last, i.e., an alternative value from the domain is assigned to the variable. For details, see [18, 3, 4].

3.2.2 Drawbacks

The disadvantage of Generate and Test is its enormous computational effort. Thus, it is mostly not useful for practical applications at all. Already obtained information on the structure of the problem are not exploited.

Backtracking is a very elementary paradigm, too. It suffers from thrashing, i.e., repeated failure induced by the same reason. Furthermore, much redundant work is performed. Inconsistencies are not remembered and the same inconsistencies are detected again and again. Moreover, conflicts are detected too late and are not avoided at the outset.

Example 3.3: See Figure 3.2.

- $X = \{A, B, C, D\}$
- $D_i = \{1, 2, 3\} \quad \forall i \in \{A, B, C, D\}$
- $D < A, \quad B - C \geq 2$

In each branch of the search tree each combination of possible values of the variables B and C is tested for consistency. It is always determined that there is only one possible combination ($B = 3$, $C = 1$).

If $A = 1$, there is no solution at all. Indeed, this is detected only as late as in the very end of the search tree.

Thrashing can be avoided by means of *Intelligent Backtracking*, i.e., backtracking is directly done to the variable that is causing the conflict. However, this approach does not solve the problem of redundant work.

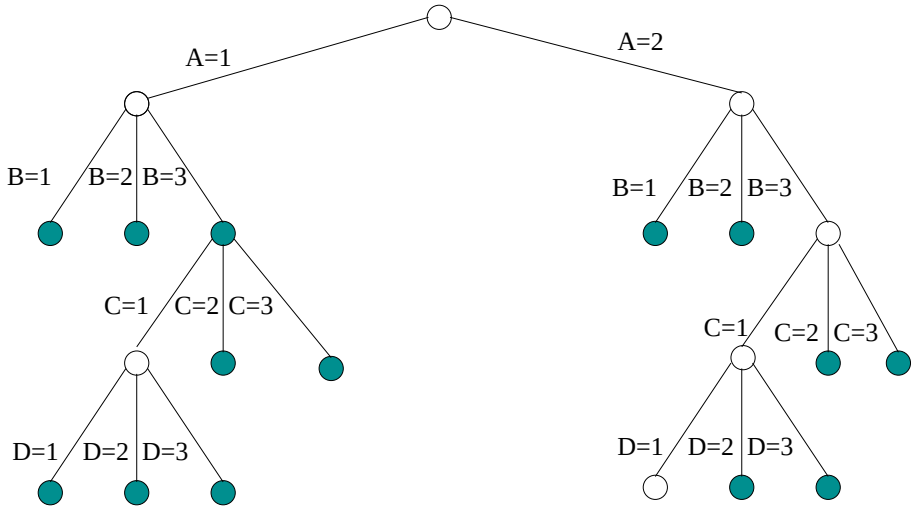


Figure 3.2: A search tree for backtracking

3.3 Consistency

The idea behind consistency techniques is to ease solving a CSP by removing values from the variables domains that cannot be part of any solution. Hence, the domains shrink in the course of the algorithms. Clearly, no solution gets lost by deleting inconsistent values from domains, but, however, it is very unlikely to solve a CSP by means of this approach. The entire CSP is solved if the cardinality of each domain is one.

3.3.1 Node Consistency

Definition 3.4: (node consistency)
A node $v \in V$ of a constraint graph $G = (V, E)$ is called *node consistent*, if each value in the domain of v satisfies all unary constraint on v.

A node $v \in V$ can be made node consistent by simply deleting each inconsistent value from its domain. If a constraint graph $G = (V, E)$ is node consistent, i.e., all nodes $v \in V$ are consistent, every unary constraint can be removed because it is already fulfilled.

3.3.2 Arc Consistency

Definition 3.5: (arc consistency)
An arc (v_i, v_j) of a constraint graph is called *arc consistent* if for each value x in the domain of v_i there is a value y in the domain of v_j such that the assignment $v_i = x,\ v_j = y$ is consistent with all binary constraints involving v_i and v_j. A constraint graph $G = (V, E)$ is arc consistent if each arc $e \in E$ is consistent.

Definition 3.6: (Support)
Considering an arc (v_i, v_j), the *support* of a value $x \in D_i$ is the set of values $y \in D_j$ such that (x, y) is consistent.

The concept of arc consistency is directed, i.e., the arc (v_i, v_j) being consistent does not tell anything about the consistency of (v_j, v_i). As an example, see Figure 3.3: The arc (v_1, v_3) is consistent, but the arc (v_3, v_1) is not consistent.

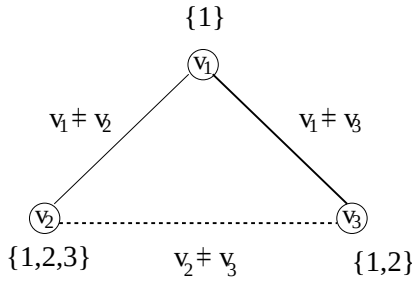


Figure 3.3: Arc consistency is directed

To make an arc (v_i, v_j) consistent it is only necessary to remove all values without support from the domain of v_i. However, to make an entire constraint graph arc consistent, it is not sufficient to examine each arc only once. In case an inconsistent value is removed from an arbitrary domain, it may happen that a previously consistent arc becomes inconsistent [18, 3]. Considering Figure 3.3, the arc (v_2, v_3) is consistent but becomes inconsistent after deleting the value 1 from domain of v_3.

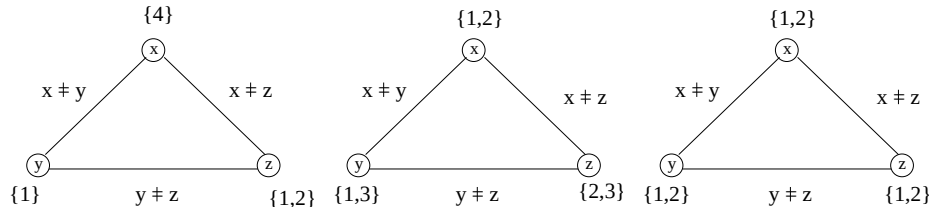
Let D_i denote the domain of variable v_i. If any value is removed from D_i while examining (v_i, v_j), it is not necessary to reconsider the arc (v_j, v_i). Let x be an arbitrary value that is removed from D_i while inspecting the arc (v_i, v_j). The value x is removed since there is no $y \in D_j$ such that $v_i = x,\ v_j = y$ is consistent. Notice that for no $y \in D_j$ the value $x \in D_i$ is in the support of y. Thus, the deletion of x does not make the arc (v_j, v_i) inconsistent.

In case any value is removed from D_i while investigating the arc (v_i, v_j), it is sufficient to reconsider the arcs $(v_k, v_i),\ k \neq i, j$, because they are the only ones which could have become inconsistent.

Let d be the cardinality of the largest domain and let $|E|$ denote the number of arcs of a constraint graph G. It takes $\mathcal{O}(|E|d^2)$ steps to make G arc consistent. Each arc must be considered at least once and it takes d^2 steps to verify the consistency of one arc.

An arc consistent graph may have none, one or more than one solution, examples are given in Figure 3.4.

There are several arc consistency algorithms that are denoted according to an unified numbering scheme. These algorithms are called AC-i, where i is an integer or fractional number. Integer values of i represent a full arc consistency algorithm, i.e., an algorithm that makes the entire graph arc consistent. Fractional values of i denote a partial arc consistency algorithm that ensures arc consistency only on a subgraph of the constraint graph.



(a) Exactly one (b) More than one (c) None

Figure 3.4: Number of possible solutions of an arc consistent constraint graph

Revise: Algorithm 1 is used to make an arc (v_i, v_j) consistent and is the basis for the arc consistency algorithms presented in the following.

Algorithm 1: `Revise`

Data : An arc (v_i, v_j) of the constraint graph

Result: The arc (v_i, v_j) is arc consistent, returns *true* iff at least one value has been removed from D_i

```
begin
    delete ← false;
    foreach x ∈ D_i do
        if there is no y ∈ D_j such that (x, y) is consistent then
            Delete x from D_i;
            delete ← true;
    return delete;
end
```

AC-1 The most simple approach to make a constraint graph $G = (V, E)$ arc consistent is called AC-1, see Algorithm 2. All the arcs of G are tested for consistency by means of the procedure Revise as long as at least one value is removed from any domain. It has already been pointed out that this approach is much too expensive, because each arc of G is reconsidered even if only one value has been deleted from one domain. Let n be the number of variables of a CSP and d be the size of the largest domain, the complexity of AC-1 is $\mathcal{O}(n^3 d^3)$ [20].

AC-3: The idea behind Algorithm AC-3, see Algorithm 3, is that after deleting values from the domain of v_i only the arcs (v_k, v_i) need to be checked again. Let $|E|$ be the number of arcs in the constraint graph and d be defined as above, the complexity of AC-3 is $O(|E|d^3)$ [20].

AC-4: Algorithm AC-3 can be improved further because many pairs of values are over and over checked for consistency for which already has been determined that they are consistent. Algorithm AC-4, see Algorithm 5, uses special data structures for this purpose which are initialized in the beginning of AC-4 by means of algorithm `Initialize`, see Algorithm 4 [3].

Algorithm 2: `AC-1`

Data : A constraint graph G
Result: The graph G is arc consistent
begin
  $Q \leftarrow \{(v_i, v_j) \in arcs(G),\ i \neq j\}$;
  repeat
    $change \leftarrow false$;
    foreach $(v_i, v_j) \in Q$ **do**
      $change \leftarrow$ `Revise(`v_i`,`v_j`)` or $change$;
  until not $change$;
end

Algorithm 3: `AC-3`

Data : A constraint graph G
Result: The graph G is arc consistent
begin
  $Q \leftarrow \{(v_i, v_j) \in arcs(G),\ i \neq j\}$;
  while not $Q.empty()$ **do**
    Choose and delete any arc (v_i, v_j) of Q;
    if `Revise(`v_i`,`v_j`)` **then**
      $Q \leftarrow Q \cup \{(v_k, v_i) \in arcs(G),\ k \neq i,\ k \neq j\}$;
end

- For each variable v_j and for each value $y \in D_j$ a structure $S_{v_j,y}$ is used to remember for which $x \in D_i$ the value y is in the support of x, if $(v_i, v_j) \in arcs(G)$.
- For each arc $(v_i, v_j) \in arcs(G)$ and for each value $x \in D_i$ it is memorized how many values in D_j are in the support of x.
- If this number is zero, then x can be deleted from D_i since it is an inconsistent value. To verify whether any other variable-value pairing has lost its support by this deletion, the pair $(v_i,\ x)$ is inserted into a queue Q to be investigated later.
- To determine the variable-value pairings $(v_k,\ z)$ which need to be checked because of the deletion of x from D_i, the structure $S_{v_i,x}$ is used. For each of the values $z \in D_k$, it is furthermore memorized how many values in D_i are in the support of x. If the value z has lost its support by the deletion of x from D_i, i.e., there was only one supporting value which now is removed, z is deleted from D_k. Because of this deletion, further variable-value pairings must be checked for consistency. Hence, the pair $(v_k,\ z)$ is inserted in Q.

The worst case running time of algorithm AC-4 is $\mathcal{O}(|E|d^2)$ which equals the lower bound to make an entire graph arc consistent, but it won't be better in the average case since the algorithm `Initialize` checks for each

Algorithm 4: `Initialize`

```
Result: Internal data structures used by algorithm AC-4 are initialized
begin
    Q ← {};
    S ← {};
    foreach (v_i, v_j) ∈ arcs(G) do
        foreach x ∈ D_i do
            total ← 0;
            foreach y ∈ D_j do
                if v_i = x, v_j = y  is consistent with the constraints on v_i
                and v_j then
                    total ← total + 1;
                    S_{v_j,y} ← S_{v_j,y} ∪ {< v_i, x >};
            counter[(v_i, v_j), x] ← total;
            if counter[(v_i, v_j), x] = 0 then
                Delete x from D_i;
                Q ← Q ∪ {< v_i, x >};
    return Q;
end
```

arc (v_i, v_j) each pair of values in the domains of v_i and v_j for consistency.

Further algorithms called AC-2, AC-5, AC-6, AC-7, ... exist. They are more or less refinements of AC-3 or AC-4. For additional information, see [18, 3].

3.3.3 k-Consistency and Path Consistency

This is a generalization of arc consistency since the latter one is not sufficient to eliminate all inconsistencies within a constraint graph, cf. Figure 3.4. Path consistency enforces consistency on paths of the constraint graph.

Definition 3.7: (path consistency)
A path $(v_1, \ldots, v_k)$ is *consistent* if for all pairs of values in the domains of v_1 and v_k there exist values within the domains of $v_2, \ldots, v_{k-1}$ such that each arc (v_i, v_{i+1}) is consistent.

Montanary [22] showed that a constraint graph is path consistent if and only if all paths of length 2 are consistent. Hence, it suffices to deal with triples of variables to make an entire constraint graph path consistent [4]. Notice, that even path consistency does not remove all inconsistencies. There are algorithms to make a constraint graph path consistent, but they are rarely used in practice because of their extensive costs. In particular, they consume much memory. Furthermore, theses algorithms add additional edges to the constraint graph. Let n be the number of variables, path consistency can be achieved in $\mathcal{O}(n^3)$ [20].

Algorithm 5: AC-4

Data : A constraint graph G
Result: The graph G is arc consistent
begin
  $Q \leftarrow$ `Initialize`;
  while not $Q.empty()$ **do**
    Select and delete any pair $< v_i, x >$ from Q;
    foreach $< v_k, z > \in S_{v_i,x}$ **do**
      $counter[(v_k, v_i), z] \leftarrow counter[(v_k, v_i), z] - 1$;
      if $counter[(v_k, v_i), z] = 0$ && z is still in D_k **then**
        Delete z from D_k;
        $Q \leftarrow Q \cup \{< v_k, z >\}$;
end

Definition 3.8: (k-consistency)
A constraint graph is *k-consistent* if the following holds: Choose for any $k-1$ variables values from the corresponding domains such that each constraint among these variables is fulfilled. Choose furthermore any k-th variable. There exists a value in the domain of this k-th variable such that each constraint among all k variables is satisfied.

Definition 3.9: (strong k-consistency)
A constraint graph is *strongly k-consistent* if it is j-consistent for each $1 \leq j \leq k$.

Strong k-consistency and k-consistency are generalizations of node, arc, and path consistency. Node consistency is equivalent to strong 1-consistency, arc consistency to strong 2-consistency, and path consistency to strong 3-consistency.

Figure 3.5 presents an example of a 3-consistent constraint graph which is not strongly 3-consistent. This graph is not 1-consistent since the node x is not consistent, but the graph is 2-consistent as well as 3-consistent.

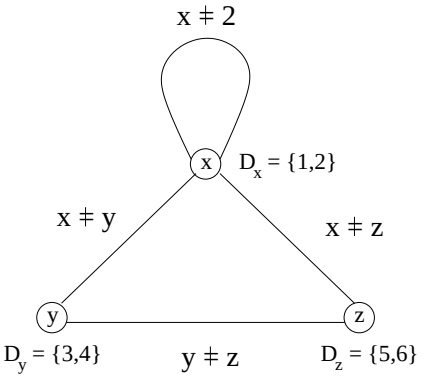


Figure 3.5: A 3-consistent but not strongly 3-consistent constraint graph

Clearly, if a constraint graph with n nodes is strongly n-consistent, a solution can be obtained without backtracking. The following questions arises: Under which circumstances is it possible to determine a solution without backtracking in a k-consistent constraint graph for $k < n$?

Definition 3.10: (ordered constraint graph)
An *ordered constraint graph* is a constraint graph whose nodes have been linearly ordered.

As an example, see Figure 3.6 [18]. The order of the nodes from top to bottom gives the order according to the linear ordering.

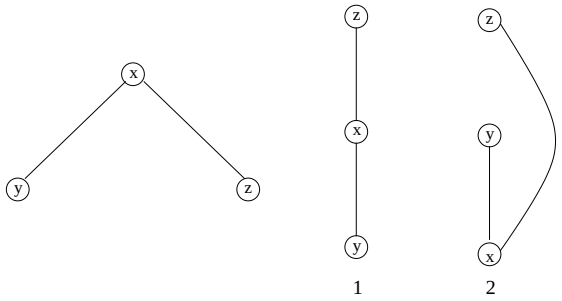


Figure 3.6: A constraint graph and two of its orderings

Definition 3.11: (width)
The *width of a node* in an ordered constraint graph is the number of edges leading to nodes with smaller values w.r.t. the linear ordering. The *width of an ordered constraint graph* is the maximum of the widths of all its nodes, the *width of a constraint graph* is the minimum of the widths of all its ordered constraint graphs.

Theorem 3.12: If a constraint graph G of width ω is strongly k-consistent and $k > \omega$, then there exists a search order within G that is backtrack free, i.e., it is possible to assign all the nodes without performing backtracking.

Proof.

- There exists an ordering of the constraint graph such that for all nodes the number of arcs leading to already instantiated variables is at most ω.
- Each newly assigned value must be consistent with at most ω values.
- It is always possible to determine such a value, because G is at least strongly $(\omega + 1)$-consistent.

□

The previous theorem leads to the following observation:

Each tree has width 1. If this graph is furthermore arc consistent, then there exists a search order that is backtrack free.

3.4 Trimming the Search Tree: Constraint Propagation

We already pointed out that it is often too expensive to apply search strategies on their own. Furthermore, it is often not possible to determine solutions of CSPs only by means of consistency techniques.

Constraint Propagation is a combination of both of them. Again, a backtracking algorithm is underlying. Additionally, consistency algorithms are called in each node of the search tree enforcing a certain degree of consistency within the constraint graph. Various full and partial arc consistency algorithms are employed.

3.4.1 Underlying Ideas of the Algorithms

The following algorithms can be denoted schematically as TS+AC-i, where TS stands for tree search and AC-i is a full or partial arc consistency algorithm. Fractional values for i denote a partial arc consistency algorithm, and integer values for i denote a full arc consistency algorithm, see Table 3.1 [24]. For partial arc consistency algorithms holds that the larger the particular fractional value of i is, the more inconsistencies are eliminated by this algorithm. Each full arc consistency algorithm enforces the same level of consistency in the graph, only computational costs differ.

algorithm	scheme
BT	TS + AC-$\frac{1}{5}$
FC	TS + AC-$\frac{1}{4}$
PL	FC + AC-$\frac{1}{3}$ = TS + AC-$\frac{1}{4}$ + AC-$\frac{1}{3}$
FL	FC + AC-$\frac{1}{2}$ = TS + AC-$\frac{1}{4}$ + AC-$\frac{1}{2}$
RFL1	FC + AC-1 = TS + AC-$\frac{1}{4}$ + AC-1
RFL2	FC + AC-2 = TS + AC-$\frac{1}{4}$ + AC-2
RFL3	FC + AC-3 = TS + AC-$\frac{1}{4}$ + AC-3

Table 3.1: Overview on constraint propagation algorithms

The challenge is to determine an appropriate degree of consistency. If the algorithms eliminate too few inconsistencies, much effort must be spent on searching. In contrast, when investing too much in consistency algorithms the saved costs for searching may not justify occurred additional costs.

The higher the degree of consistency gained, the smaller the number of nodes of the search tree, however, much more tests are needed. Nadel [24] investigated the "n-queens problem" (arrangement of n queens on a $n \times n$ chess board such that no queen is attacked by another one) and the "Confused n-queens problem" (arrangement of the queens such that all queens attack each other) and showed that forward checking needs for these problems the fewest constraint

test altogether. We did not find further observations concerning this topic, but it shows that it is not necessarily useful to gain the highest level of consistency within a constraint graph.

It is of course possible to invoke a different constraint propagation algorithm in each node of the search tree.

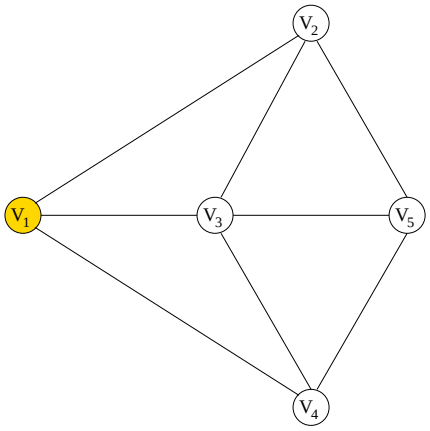


Figure 3.7: Running example

Figure 3.7 is our running example to explain the various algorithms. In the following it is assumed that variable v_1 is already assigned.

Backtracking (BT): Even simple backtracking performs a certain kind of constraint propagation. When assigning the i-th variable, it is assured that the assigned value is consistent with the assignment of the previously considered variables $v_1, \ldots, v_{i-1}$; this is referred to as AC-$\frac{1}{5}$.

Forward Checking (FC): Arc consistency on arcs between not yet assigned variables and already assigned variables is enforced. Before assigning the i-th variable, the arcs (v_j, v_{i-1}), $j = i, \ldots, n$ are made consistent by removing inconsistent values (with respect to the assignment of v_{i-1}) from the domains D_j; this is called AC-$\frac{1}{4}$.

Considering our running example in Figure 3.7, this means that before assigning v_2 the arcs (v_2, v_1), (v_3, v_1) and (v_4, v_1) would be revised.

Partial Lookahead (PL): This algorithm also enforces arc consistency on arcs between not yet assigned variables, but only the arcs (v_j, v_k) with $j < k$ are to be considered. Before assigning the i-th variable, forward checking is invoked first. In addition, the arcs (v_j, v_k), $j = i, \ldots, n-1$, $k = j+1, \ldots, n$ are checked and inconsistent values are removed from the domains D_j; this procedure is named AC-$\frac{1}{3}$.

Considering Figure 3.7, these are the arcs (v_2, v_3), (v_2, v_5), (v_3, v_4), (v_3, v_5) and (v_4, v_5).

Full Lookahead (FL): Arc consistency with respect to the current domains on each arc between not yet assigned variables is enforced. Before assigning the i-th variable, forward checking is invoked first. Furthermore, the arcs $(v_j, v_k),\ j = i, \ldots, n,\ k = i, \ldots, n,\ j \neq k$ are checked and inconsistent values are removed from the domains D_j; this is referred to as AC-$\frac{1}{2}$.

Considering Figure 3.7, this are all the arcs of the subgraph induced by the nodes $v_i,\ i = 2, \ldots, 5$. This algorithm is only a partial arc consistency algorithm, because each arc is considered only once.

Really Full Lookahead 1 (RFL1): Before assigning the i-th variable, forward checking is invoked first. Furthermore, all arcs $(v_j, v_k),\ j = i, \ldots, n,$ $k = i, \ldots, n,\ j \neq k$ are revised as long as in at least one domain are values removed; this is AC-1.

Really Full Lookahead 3 (RFL3): Before assigning the i-th variable, forward checking is invoked first. The subgraph induced by the unassigned nodes is made arc consistent. AC-3 is called and only the arcs that could have been affected by the domain reductions are reconsidered.

Really Full Lookahead 2 (RFL2): It works like RFL3, but instead of AC-3 the algorithm AC-2 is called.

Within the course of these algorithms except for simple backtracking itself, values are removed temporarily from the domains of the variables. In case backtracking is performed by such an algorithm, removed values have to be reinsert in some domains. In the worst case, many deletions followed by reinsertions of values are thus executed by these algorithms.

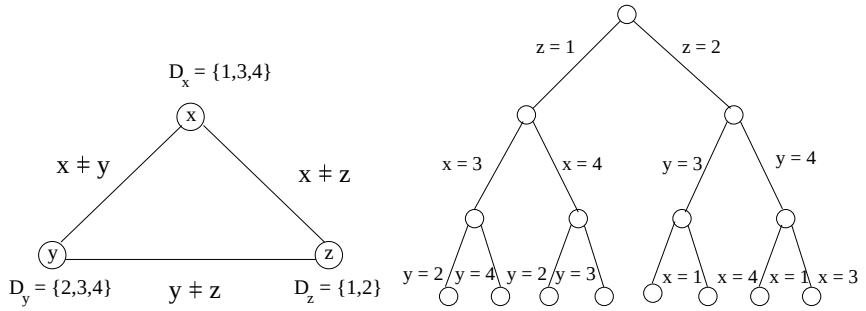
3.5 Constructing the Search Tree

Using backtracking, the order in which variables are to be considered is very important.

A popular heuristic is to choose in each step the variable with the smallest remaining domain. Thereby it may happen that the ordering of the variables differs in various branches of the search tree. Figure 3.8 gives an example. This approach is also called first-fail principle. The algorithm DSATUR proposed by Brélaz [1979] for the node coloring problem can be modeled by this means. DSATUR consists of choosing in each step one of the variables with the maximal number of differently labeled adjacent vertices. Hence, the selected variable has the least number of possible values left.

A further possibility is to choose in each step the variable being mostly bounded by constraints. It is aimed at pruning invalid branches as soon as possible.

An additional heuristic is to determine a maximum stable set in the beginning and to assign these nodes last. Unfortunately, computing a maximum stable set



(a) The problem (b) The corresponding search tree

Figure 3.8: Variable ordering according to the first-fail principle

is $\mathcal{NP}$-hard. In addition, if only a small stable set can be found, the benefit of this approach is minor. The reason for assigning the stable set last is that there are no constraints between these nodes. Hence, after assigning all other nodes one can assign the nodes of the stable set without considering them among themselves. This reduces the size of the search tree [18].

Since it is possible to solve tree structured constraint graphs without backtracking, one further method is to determine within a general graph a cycle-cut set at the beginning, i.e., a set of nodes such that after deleting them the graph is a tree. These variables are assigned first and then deleted from the constraint graph. After enforcing arc consistency, it is possible to solve the problem without backtracking [18]. Let n be the number of nodes of the constraint graph, d be the size of the largest domain, and k be the size of the cycle-cut set. Computational costs of this heuristic are in $\mathcal{O}(d^k * (n-k)d^2)$ since the cycle-cut set must be assigned at most d^k times and after deleting these nodes it takes $\mathcal{O}((n-k)d^2)$ steps to achieve arc consistency on the resulting constraint graph. This approach is only useful if the cycle-cut set is small, otherwise the advantage is negligible.

Furthermore, the ordering of the values to be assigned to the variables is important. Clearly, if a CSP has a solution and always the correct value is chosen, it is possible to solve the CSP without backtracking. If the CSP does not have any solution or all solutions are needed, the value ordering is not relevant.

A possible heuristic for determining a value ordering is to always choose the one that maximizes the number of possible assignments to not yet assigned variables. To count this number, the algorithm AC-4 can be used since this one counts for each possible value the size of its support. Kale [16] adapted by applying this variable ordering a backtracking based algorithm for solving the "n-queens problem". This adaption enabled him to solve the problem for significantly larger numbers of n [18].

Chapter 4

The Optimization Programming Language (OPL)

This chapter gives an introduction to OPL before we present our models for FAP in Chapter 5. A small OPL model is used to discuss some features of the language in Section 4.1, some general modeling techniques are explained in Section 4.2, and MIPs and OPL models are compared in Section 4.3.

4.1 A brief Introduction

The "Optimization Programming Language" (OPL) has been developed by van Hentenryck [13] and combines two different aspects. On the one hand, it is a modeling language that is more expressive than Mixed Integer Programming. It allows to state Linear Programs, Mixed Integer Programs, and Constraint Satisfaction Problems, as well as combinations of all of them. On the other hand, OPL offers the possibility to solve formulated models by providing an interface to underlying solvers. Thereto, in addition to the constraints of the model, a search heuristic is stated that is then executed by a solver. A variety of commercial constraint solvers is available. One of them is ILOG Solver, which is an object-oriented C++-library offering constraint programming algorithms. ILOG OPL Studio is built on top of ILOG Solver (among others) allowing to solve Constraint Satisfaction Problems modeled in OPL by means of the solution framework provided by ILOG Solver.

Figure 4.1 presents a small OPL model for the weighted graph coloring problem. The corresponding graph is shown in Figure 4.2.

An OPL model consists of several parts including the definition of the model data, the declaration of the variables, and the statement of objective and constraints. It is possible to define an own search heuristic. If none is given, a default search procedure is applied. Available data types are strings, discrete

and floating point numbers as well as enumerated data types, which are similar to the ones used in the programming language C.

```
01: enum color {red, blue, green, yellow};
02: int+ number_of_nodes = 4;
03: range node 1..number_of_nodes;
04: struct edge_type { node u; node v; };
05: {edge_type} edges = {<1,2>, <1,3>, <2,3>, <2,4>, <3,4>};
06: int+ costs[color] = [2, 4, 6, 8];
07:
08: predicate is_different( color c1, color c2 )
09:    return c1 <> c2;
10:
11: var color label[node];
12:
13: minimize
14: sum(v in node) costs[ label[v] ]
15: subject to {
16:    forall( e in edges ) is_different(label[e.u], label[e.v]);
17:    label[1] <> green;
18:    sum(v in node) (label[v] = blue) = 2;
19: };
```

Figure 4.1: OPL model for the weighted graph coloring problem

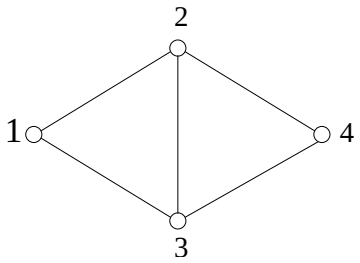


Figure 4.2: Example graph for the weighted graph coloring problem

Row (1) of the model declares an enumerated data type `color`, which can take on one out of four possible values representing the available colors for the nodes. Row (2) defines a non-negative integer, which is the number of nodes of the graph. Row (3) defines the nodes by means of a range of integers from 1 to `number_of_nodes`, and Row (4) declares a structure `edge_type` representing the two end nodes of an undirected edge. Row (5) defines the edge set of the graph. Each set member is of type `edge_type`. Row (6) defines an array `costs` of non-negative integers, one value for each possible color. Array elements can be accessed directly by elements of enumerated data types, i.e., the cost of color red is 2, of blue 4, and so on. Rows (8) and (9) of the model define a `predicate`. `Predicates` can be used to state user-defined constraints. The OPL implementation enforces arc consistency on the conditions defined by `predicates`. The

use of `predicates` may result in a more effective pruning compared to simply stating the constraints, but it is sometimes less efficient.

Row (11) declares an array of variables of type `color`, one variable for each node. Rows (13) and (14) state the objective. Possible objectives are `minimize`, `maximize`, and `solve`. In the latter case the problem is a feasibility problem rather than an optimization problem. Rows (15) to (19) state the constraints. Using the previously defined predicate `is_different`, Row (16) imposes that the label of node u differs from the label of node v for every edge $e = (u, v)$. Row (17) states that the color of node 1 shall not be green. Row (18) is a so-called *Higher-Order Constraint (Meta-Constraint)*. Meta-Constraints are useful for stating complex conditions. It is possible to formulate further conditions within a constraint. One binary variable is associated with each of these embedded conditions that is equal to 1 if and only if the condition is satisfied. The constraint in Row (18) enforces that the number of nodes labeled blue is equal to 2. For each $v \in V$, a binary variable is associated with the condition `label[v]=blue`.

4.2 Modeling Techniques

Given a combinatorial optimization problem, it is often not difficult to devise some OPL model for it. However, to formulate efficient OPL models, a lot of knowledge on Constraint Programming is necessary. The way the constraints or the input data of a problem are formulated may have a substantial impact on the performance of the model. The following guide lines were given in [15] and apply to OPL as well as to Constraint Programming in general. Our tests showed that it is very important to use few variables. Most of our approaches needing a lot of variables were not successful. The formulation of the input data has a major impact on the solution times, too. Especially breaking symmetries (described below) enabled us to solve some instances that we could not solve otherwise. However, if solutions could be obtained easily, the formulation of redundant constraints or breaking symmetries often did not yield any benefit.

Few Variables: One should use as few variables as possible since the introduction of unneeded variables unnecessarily enlarges the search space. The search space of a CSP with variables $v_1, \ldots, v_n$ and corresponding domains $d_1, \ldots, d_n$ is the Cartesian product of the domains $d_1 \times \cdots \times d_n$, whose size is $\prod_{i=1}^{n} |d_i|$.

Symbolic Constraints: An important aspect of constraint programming in general and of OPL in particular are *symbolic constraints*, also referred to as *global constraints*. These are predefined constraints used to state frequently occurring conditions. In addition, extra propagation algorithms and consistency strategies are often applied to symbolic constraints such that a more effective pruning is possible.

A frequently employed global constraint is `alldifferent`. It takes one array as parameter and enforces that all array elements differ from each other. As an example, consider the variables $v_1, \ldots, v_3$ with the corresponding domains $D_1 = D_2 = D_3 = \{1, 2\}$, together with the conditions $v_i \neq v_j$ for $i \neq j$. This problem is arc consistent but has no solution (cf. Figure 3.4 (c)). Modeled by pairwise disequalities and enforcing arc consistency, it is not possible to determine that the problem is infeasible. In contrast, by means of the specialized propagation rules applied to `alldifferent`, it can be determined that there is no solution. A consistency algorithm for `alldifferent` is given in [21, Section 3.5]. Roughly speaking, it first removes the values of all fixed variables, i.e., the variables v_i with $|D_i| = 1$, from the domains of all yet unfixed variables. Next it checks whether the number of unfixed variables exceeds the total number of different values available for all of them together. In the above example, the number of unfixed variables is 3 and the number of different values available for all of them is 2. Hence, the problem is not feasible.

There are a lot of further predefined symbolic constraints, see for instance [13, 14, 15, 21].

Breaking Symmetries: It is also often helpful to break symmetries. Symmetries make the search space contain many different solutions that all provide the same information for us. Breaking symmetries is done by stating additional constraints, e.g., by imposing an ordering on some set of variables. Notice that these additional constraints must not eliminate any relevant solution. As an example, consider the Frequency Assignment Problem presented in Chapter 2. In Section 5.1 it is shown that all TCHs of one cell have to satisfy the same separation requirements. Thus, given the set C of TCHs of a cell and the set of channels assigned to all of them, it does not matter which TCH receives which channel. Let y be the frequency assignment which maps every TRX to a channel, the described symmetries can be broken by stating the requirement $y(v_1) < \cdots < y(v_n)$ for all TCHs $v_1, \ldots, v_n \in C$.

Redundant Constraints: The formulation of redundant constraints, i.e., conditions resulting implicitly from the structure of the problem, may ease finding solutions. Ideally, they provide information that the solvers can (typically) not derive directly from the other given constraints. Considering our running example k-Colorability, let G be the given graph and $C \subseteq G$ be a clique of G. A possible redundant constraint is $c(i) \neq c(j)$ for all $i, j \in C$ with $i \neq j$. The above mentioned `alldifferent` constraint can be used for stating this condition.

4.3 MIPs versus OPL Models

The purpose of this section is to investigate whether the additional features of OPL really provide more possibilites to formulate conditions for a mathematical problem, or if some of these further elements can be expressed in Mixed

Integer Programs (MIPs) as well. Among others, OPL allows to employ logical conditions and functions like absolute value, minimum, maximum, and modulo. Furthermore, strict inequalities and disequalities ($\neq$) on integer variables and symbolic constraints such as `alldifferent` can be stated. These additional features may make OPL models more loosely readable than MIPs, but it is demonstrated that all these mentioned elements of OPL can be expressed in Mixed Integer Programming as well. However, it is not always possible to translate an OPL model into a MIP. Moreover, additional variables and constraints have to be introduced. Hence, the following observations are more of theoretical than of practical interest.

4.3.1 Translating OPL models into MIPs

Important for the transformations given in this section is the capability to express logical conditions in Mixed Integer Programming. Let C be an arbitrary constraint and δ be a binary variable. It is possible to link C and δ by means of additional constraints such that one of the following relations is satisfied:

$$\begin{aligned} (C \text{ holds}) &\Longrightarrow (\delta = 1) \\ (C \text{ holds}) &\Longleftarrow (\delta = 1) \\ (C \text{ holds}) &\Longleftrightarrow (\delta = 1) \end{aligned} \tag{4.1}$$

Binary variables linked with constraints are called indicator variables in the following. They are especially used for formulating logical conditions among constraints in Mixed Integer Programming.

Some of the ideas of the following transformations are summarized in Table 4.2 on page 50.

Restrictions

We have to face some limitations when transforming an OPL model into a MIP. Rational variables may cause some difficulties in strict inequalities. It may even happen that there is no optimal solution to a given problem at all. As an example, consider $\max\{x \in \mathbf{Q} : x < 1\}$, which does not have an optimal solution, but would have an optimal solution when restricted to integer variables.

Strict inequalities and disequalities ($\neq$) on rational variables are not allowed in OPL because of the arising problems. Actually, we do not need to discuss them here. However, in order to perform some of the transformations below, we have to deal with strict inequalities. Thus, we restrict ourselves to a certain precision when considering rational numbers or variables, which enables us to assume all variables and numbers to be integer.

In addition, most transformations cannot be done properly without the existence of some lower and upper bounds on involved terms. If they do not exist, it is shown that these transformations cannot be done without imposing further constraints on some variables. In each case, it is pointed out if lower and upper bounds are needed, and if yes, which terms are affected.

Logical Conditions

The transformations given in this subsection are based on Williams [27]. To express a certain logical relation among a set R of constraints, it is necessary to introduce a binary variable δ_r for all of the $r \in R$ such that one or both of the following relations are fulfilled:

$$\begin{aligned} r &\implies \delta_r = 1 \\ \delta_r = 1 &\implies r. \end{aligned}$$

Moreover, the introduced indicator variables δ_r can be used to ensure that at least k or at most k of the expressions of R are satisfied simultaneously. The following inequality imposes that at least k conditions of R are fulfilled:

$$\sum_{r \in R} \delta_r \geq k.$$

Similarly, the statement $\sum_{r \in R} \delta_r \leq k$ enforces that at most k conditions of R hold.

Now, we show how to link inequalities and binary indicator variables. As a running example, consider the relation:

$$\delta = 1 \Longleftrightarrow \sum_i a_i x_i \leq b.$$

First, we model the direction:

$$\delta = 1 \Longrightarrow \sum_i a_i x_i \leq b. \tag{4.2}$$

An upper bound M on the expression $\sum_i a_i x_i - b$ is needed for this task. As explained below, if this bound does not exist, we cannot do the following transformation without imposing further restrictions on the x_i. Assuming that there is an upper bound M on $\sum_i a_i x_i - b$, Condition (4.2) can be modeled by the constraint:

$$\begin{aligned} & \sum_i a_i x_i + M\delta \leq M + b \\ \Longleftrightarrow & \sum_i a_i x_i - b + M\delta \leq M. \end{aligned} \tag{4.3}$$

In case $\delta = 1$, it is ensured that the original constraint holds. If $\delta = 0$, the term $\sum_i a_i x_i - b$ is constrained to be less than or equal to M. Obviously, unless M is an upper bound on $\sum_i a_i x_i - b$, an additional restriction on the x_i is implied by $\delta = 0$.

Second, we model the direction:

$$\Big(\sum_i a_i x_i \leq b \implies \delta = 1 \Big) \tag{4.4}$$

$$\iff \Big(\delta = 0 \implies \sum_i a_i x_i > b \Big) \tag{4.5}$$

$$\underbrace{\iff}_{(*)} \Big(\delta = 0 \implies \sum_i a_i x_i \geq b + 1 \Big). \tag{4.6}$$

The last equivalence $(*)$ is only valid since we restrict ourselves to integer numbers and variables.

To express Condition (4.6) as an inequality we need a lower bound $m \neq 1$ on $\sum_i a_i x_i - b$. As above, if this bound does not exist, it is not possible to do the given transformation. Given a lower bound $m \neq 1$ on $\sum_i a_i x_i - b$, we can state Condition (4.6) as follows:

$$\sum_i a_i x_i - (m-1)\delta \geq b + 1, \quad m \neq 1 \tag{4.7}$$

$$\iff \sum_i a_i x_i - b - (m-1)\delta \geq 1, \quad m \neq 1.$$

If δ is equal to 0, it is ensured that Constraint (4.6) holds. If $\delta = 1$, the term $\sum_i a_i x_i - b$ is constrained to be greater than or equal to m. There must not be any implication if $\delta = 1$, hence, m has to be a lower bound on this term. This approach cannot be done correctly if this bound does not exist.

It is also possible to indicate whether a constraint $\sum_i a_i x_i \geq b$ holds or not. The constraints corresponding to (4.3) and (4.7) are:

$$\sum_i a_i x_i + m\delta \geq m + b \tag{4.8}$$

$$\sum_i a_i x_i - (M+1)\delta \leq b - 1, \quad M \neq -1. \tag{4.9}$$

Finally, if we want $\delta = 1$ to imply that $\sum_i a_i x_i = b$ holds, we simply state the conditions (4.3) and (4.8) together.

The opposite direction $\sum_i a_i x_i = b \implies \delta = 1$ is slightly more difficult:

$$\Big(\sum_i a_i x_i = b \implies \delta = 1 \Big) \tag{4.10}$$

$$\iff \Big(\delta = 0 \implies \sum_i a_i x_i \neq b \Big) \tag{4.11}$$

$$\iff \Big(\delta = 0 \implies \sum_i a_i x_i < b \ \vee \ \sum_i a_i x_i > b \Big) \tag{4.12}$$

$$\underbrace{\iff}_{(*)} \Big(\delta = 0 \implies \sum_i a_i x_i \leq b - 1 \ \vee \ \sum_i a_i x_i \geq b + 1 \Big). \tag{4.13}$$

$$\iff \Big(\delta = 1 \ \vee \ \sum_i a_i x_i \leq b - 1 \ \vee \ \sum_i a_i x_i \geq b + 1 \Big). \tag{4.14}$$

The transformation $(*)$ can be done since we assumed all a_i and x_i to be integer.

We have to introduce binary variables δ_1 and δ_2 for both terms on the right-hand side of (4.14) that are equal to 1 if and only if their corresponding expressions are fulfilled. Then it is possible to state (4.14) as $\delta + \delta_1 + \delta_2 \geq 1$.

Having introduced binary variables for each constraint that we wish to connect logically, it is possible to represent logical relations between them. Let A and B be two distinct constraints and δ_A and δ_B be the corresponding indicator variables. Table (4.1) gives an overview how to express logical conditions between A and B.

$A \vee B$	$\delta_A + \delta_B \geq 1$
$A \wedge B$	$\delta_A = 1, \delta_B = 1$
$!A$	$\delta_A = 0$
$A \Leftrightarrow B$	$\delta_A - \delta_B = 0$
$A \Rightarrow B$	$\delta_A - \delta_B \leq 0$

Table 4.1: Logical conditions on constraints

Strict Inequalities ($>$, $<$)

Given integer variables and coefficients, it is possible to transform a given strict inequality of the form $\sum_i a_i x_i < b$ into a non-strict inequality $\sum_i a_i x_i \leq b - 1$.

Disequalities ($\neq$)

Disequalities may be expressed by means of strict inequalities using logical conditions:

$$\begin{aligned} & x_i \neq x_j \\ \Longleftrightarrow \quad & (x_j > x_i) \ \vee \ (x_i > x_j) \\ \Longleftrightarrow \quad & (x_i - x_j \leq -1) \ \vee \ (x_j - x_i \leq -1). \end{aligned}$$

We introduce binary indicator variables δ_1 and δ_2 that are equal to 1 if and only if $x_i - x_j \leq -1$ or $x_j - x_i \leq -1$, respectively. It has to be satisfied that:

$$\begin{aligned} m_1 \ \leq \ x_i - x_j + 1 \ \leq \ M_1 \\ m_2 \ \leq \ x_j - x_i + 1 \leq \ M_2. \end{aligned}$$

If any of the lower bounds m_i or upper bounds M_i does not exist, it is not possible to model disequalities by means of indicator variables and logical conditions. The argument is analog to the one given on page 42.

The following constraints enforce that $\delta_1 = 1 \iff x_i < x_j$:

$$x_i - x_j + M_1 \delta_1 \ \leq \ M_1 - 1 \tag{4.15}$$

$$x_i - x_j - (m_1 - 1)\delta_1 \ \geq \ 0, \quad m_1 \neq 1. \tag{4.16}$$

In the same manner it is ensured that $\delta_2 = 1 \iff x_j < x_i$:

$$x_j - x_i + M_2\delta_2 \leq M_2 - 1 \tag{4.17}$$
$$x_j - x_i - (m_2 - 1)\delta_2 \geq 0, \quad m_2 \neq 1. \tag{4.18}$$

Finally, to express the same as the initial disequality we need to connect δ_1 and δ_2 logically by stating $\delta_1 + \delta_2 = 1$.

Altogether, there are two extra variables and five extra constraints needed to express one disequality in MIP.

Absolute Value

There are different approaches how to model absolute values within Mixed Integer Programming. As an example, consider the constraint: $|x_1|+|x_2| \leq 2$. It is possible to express it without using an absolute value by stating the following constraints:

$$\begin{aligned} x + y &\leq 2 \\ x - y &\leq 2 \\ -x + y &\leq 2 \\ -x - y &\leq 2. \end{aligned}$$

Unfortunately, the number of inequalities introduced this way is exponential in the number of occurrences of the `abs`-function in the initial constraint. This means, a constraint $\sum_{i=1}^{n} a_i|x_i| \leq b$ would cause 2^n new inequalities. Clearly, this approach is impractical.

Logical conditions are another possibility to model absolute values in Mixed Integer Programming. We introduce a variable $y \geq 0$ for each variable x whose absolute value is taken and ensure that y is equivalent to $|x|$ by satisfying the following relations:

$$x \geq 0 \implies y = x \tag{4.19}$$
$$x \leq 0 \implies y = -x. \tag{4.20}$$

Binary variables have to be introduced in order to model it. As in the previous subsections, we will need some lower and upper bounds on terms involving x and y. If any of these bounds does not exist, it is not possible to express the absolute value like this (cf. page 42). Assuming the existence of suitable bounds, we introduce binary variables $\delta_1, \ldots, \delta_4$ for the four constraints on both sides of Relations (4.19) and (4.20) as explained in Section "Logical Conditions". This requires six additional constraints. Finally, to express the Relations (4.19) and (4.20), we state:

$$\delta_1 - \delta_2 \leq 0 \tag{4.21}$$
$$\delta_3 - \delta_4 \leq 0. \tag{4.22}$$

Altogether, five additional variables and eight additional constraints are needed for each variable whose absolute value is taken in order to apply this approach.

Minimum Function and Maximum Function

Considering terms of the form $\min(x_i, x_j)$ or $\max(x_i, x_j)$, the simplest way to express minimum or maximum functions in Mixed Integer Programming is to introduce a variable y which is to be maximized or minimized, respectively, in the objective function.

In order to model a minimum function, the introduced variable y has to be maximized and constrained as follows:

$$\begin{aligned} y &\leq x_1 \\ y &\leq x_2. \end{aligned}$$

Analogously, a maximum function is stated by minimizing y and by constraining it by means of:

$$\begin{aligned} y &\geq x_1 \\ y &\geq x_2. \end{aligned}$$

This approach can be used if there are no terms involving the minimum or maximum function within the objective. If there is an objective to be maximized containing a minimum function with a positive sign, or in case the objective is to be minimized and contains a positively signed maximum term, we run into no difficulties as well. Neither lower or upper bounds nor an introduced precision is needed for this transformation. Consider the following example, where the optimal solution value of both problems is 24 with $x_1 = x_2 = 6$:

$$\begin{aligned} \max \;\; & 3\min(x_1, x_2) + x_2 \\ & 2x_1 + x_2 \leq 18 \\ & x_1, x_2 \geq 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} \max \;\; & 3y + x_2 \\ & 2x_1 + x_2 \leq 18 \\ & y \leq x_1 \\ & y \leq x_2 \\ & x_1, x_2, y \geq 0 \end{aligned}$$

However, the above approach cannot be used if the problem is a maximization (minimization) problem and the maximum (minimum) function occurs within the objective with a positive sign. In the following example, the optimal solution value of the original problem is 72 with $x_1 = 0$ and $x_2 = 18$. In contrast, the transformed problem is unbounded:

$$\begin{aligned} \max \;\; & 3\max(x_1, x_2) + x_2 \\ & 2x_1 + x_2 \leq 18 \\ & x_1, x_2 \geq 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} \max \;\; & 3y + x_2 \\ & 2x_1 + x_2 \leq 18 \\ & y \geq x_1 \\ & y \geq x_2 \\ & x_1, x_2, y \geq 0 \end{aligned}$$

Given that we are able to model logical conditions, above example can also be transformed as follows:

$$\begin{aligned} \max \;\; & 3\max(x_1, x_2) + x_2 \\ & 2x_1 + x_2 \leq 18 \\ & x_1, x_2 \geq 0 \end{aligned} \qquad \Longrightarrow \qquad \begin{aligned} \max \;\; & 3y + x_2 \\ & 2x_1 + x_2 \leq 18 \\ & y \leq x_1 \vee y \leq x_2 \\ & x_1, x_2, y \geq 0 \end{aligned}$$

Another way to express minimum or maximum functions is to use implications. Considering the modeling of a minimum function, we introduce a variable y and ensure that:

$$x_i \leq x_j \implies y = x_i \tag{4.23}$$
$$x_j \leq x_i \implies y = x_j. \tag{4.24}$$

This is done by means of binary indicator variables as shown in the previous paragraphs. Maximum functions could be handled in the same manner.

Modulo

Consider an expression of the form x mod k, where x is a discrete or rational variable and k is a non-negative integer number or an integer variable itself. If there is a more complex expression followed by modulo, one can introduce a new variable and associate the expression with it. For this approach, no lower or bounds bounds or any introduced precision is needed.

Modulo can be expressed in Mixed Integer Programming as follows. We have to introduce two new discrete variables y and z and need to state two extra constraints:

$$z - k \leq -1 \tag{4.25}$$
$$y \cdot k + z - x = 0. \tag{4.26}$$

The first requirement models that the result z must be less than or equal to $k - 1$, the second constraint states the modulo condition, i.e., that z is the integer remainder of the division x by k. Then x mod k can be replaced by z.

If k is a variable itself, it is more difficult. The term x_1 mod k can be replaced by z, but, unfortunately, $y \cdot k$ is non-linear. In spite of this, it is possible to express this in Mixed Integer Programming using Separable Programming. For more details see the subsection on modeling non-linear constraints in MIP on page 47.

Non-linear Constraints

As described in Williams [27], it is sometimes possible to express even non-linear models by means of MIPs. We have to distinguish between *convex programming* and *non-convex programming*.

A set $S \subseteq \mathbf{R}^n$ is said to be *convex* if for all $x, y \in S$ and $\lambda \in [0, 1]$:

$$\lambda x + (1 - \lambda)y \in S.$$

A function $f : \mathbf{R}^n \to \mathbf{R}$ is said to be convex if for any $\lambda \in [0, 1]$ and for any $x, y \in \mathbf{R}^n$:

$$\lambda f(x) + (1 - \lambda)f(y) \geq f(\lambda x + (1 - \lambda)y). \tag{4.27}$$

An optimization problem is said to be convex if the objective function as well as the set of feasible points is convex.

The advantage of convex optimization problems is that each local optimum is a global optimum as well. This does not hold for non-convex optimization problems.

A function $f : \mathbf{R}^n \to \mathbf{R}$ is said to be *separable* if it can be expressed as the sum of terms involving only one variable. Hence, the function $f(x, y) = x^2 + y^2$ is separable but the function $g(x, y) = x \cdot y + x^2$ is not. It is useful to consider separable functions in mathematical programming, because they can be approximated by piecewise linear functions. This can be done by means of additional variables and linear constraints. For details, see [27, Chapter 7].

Sometimes it is possible to convert a model in non-separable form into a model in separable form [27, Section 7.4]. Consider the expression $x_1 \cdot x_2$. Two variables y_1 and y_2 are introduced as follows:

$$\begin{aligned} y_1 &= \frac{1}{2}(x_1 + x_2) \\ y_2 &= \frac{1}{2}(x_1 - x_2). \end{aligned}$$

The term $y_1^2 - y_2^2$ is equal to $x_1 \cdot x_2$ and is separable.

Higher-order Constraints

Higher-order constraints like $\sum_{i \in I}(x_i = 3) = 2$ can easily be expressed in MIPs by means of binary indicator variables. We have to introduce one binary variable for each expression embedded within the Higher-order constraint that is equal 1 if and only if the corresponding expression is satisfied.

As an example consider above Higher-order constraint $\sum_{i \in I}(x_i = 3) = 2$. We introduce variables δ_i for all $i \in I$ and ensure that $x_i = 3 \iff \delta_i = 1$. Now, the given constraint could be expressed as $\sum_{i \in I} \delta_i = 2$.

`alldifferent` Constraints

This special kind of constraint is used to ensure that all members of an array get different values. This can be expressed in Mixed Integer Programming by stating pairwise disequalities on all members $x_i \neq x_j$ of the array. Hence, $n(n-1)/2$ disequalities are needed, where n is the number of array elements,

each requiring two introduced variables and five extra inequalities. Altogether, this leads to $n(n-1)$ new variables and $5n(n-1)/2$ new constraints. As described in Section "Disequalities", this approach can only be applied if the lower and upper bounds on the terms $x_i - x_j + 1$ and $x_j - x_i + 1$ exist for all $i \neq j$.

`distribute`, `atleast`, `atmost`, `atleastatmost` Constraints

The constraints `distribute`, `atleast`, `atmost`, and `atleastatmost` are similar to each other and can be used to formulate cardinality conditions on arrays. Let *card*, *value*, *low*, *up*, and *base* be one-dimensional arrays. All of them may have the same index set S with $|S| = n$, except for *base* whose index set is R with $|R| = m$. The requirement `distribute`(*card*, *value*, *base*) holds if *card*[*i*] is the number of occurrences of *value*[*i*] in the array *base*. In the same manner, the constraints `atleast` and `atmost` ensure that the array *base* contains at least or at most times the value *value*[*i*]. The constraint `atleastatmost` is used to state lower and upper bounds on the frequencies of *value*[*i*] within the array *base*.

These constraints can be expressed by means of Higher-order constraints. For `distribute` this is:

$$card[i] = \sum_{j \in R} (base[j] = value[i]) \quad \text{for all } i \in S. \tag{4.28}$$

The other constraints can be expressed in a similar way.

By means of binary indicator variables, these constraints can be expressed in Mixed Integer Programming. We need to introduce for each $i \in S$ and $j \in R$ a binary variable δ_{ij} that equals to 1 if and only if $base[i] = value[j]$. Hence, we can state:

`distribute`$(card, value, base)$:	$card[i]$	$= \sum_{j \in R} \delta_{ij}$	for all $i \in S$
`atleast`$(card, value, base)$:	$card[i]$	$\leq \sum_{j \in R} \delta_{ij}$	for all $i \in S$
`atmost`$(card, value, base)$:	$card[i]$	$\geq \sum_{j \in R} \delta_{ij}$	for all $i \in S$
`atleastatmost`$(low, up, card, base)$:	$low[i]$	$\leq \sum_{j \in R} \delta_{ij} \leq up[i]$	for all $i \in S$.

Altogether, nm binary variables and $2mn$ extra inequalities are needed for the variables δ_{ij} since we have to state two extra conditions for each introduced variable, see page 42. In addition, to state the actual cardinality constraints, n inequalities are needed for `distribute`, `atleast` and `atmost`, and $2n$ are needed for `atleastatmost`.

4.3.2 Translating MIPs into OPL models

Only equalities and inequalities are allowed in Mixed Integer Programming on discrete and rational variables. All these conditions are also valid for OPL. Hence, each MIP can be considered as an OPL model as well.

Constraint	in OPL	in MIP
logical conditions	$A \vee B$	$\delta_A + \delta_B \geq 1$
	$A \wedge B$	$\delta_A = 1, \delta_B = 1$
	$!A$	$\delta_A = 0$
	$A \Leftrightarrow B$	$\delta_A - \delta_B = 0$
	$A \Rightarrow B$	$\delta_A - \delta_B \leq 0$
strict inequalities	$\sum_i a_i x_i < b,\ a_i, x_i \in \mathbf{Z}$	$\sum_i a_i x_i \leq b - 1$
disequalities	$x_i \neq x_j$	$(x_i - x_j \leq -1)\ \vee\ (x_j - x_i \leq -1)$
absolute value	$y = \lvert x \rvert$	$x \geq 0 \Longrightarrow y = x$
		$x \leq 0 \Longrightarrow y = -x$
minimum	$y = \min(x_i, x_j)$	$x_i \leq x_j \Longrightarrow y = x_i$
		$x_i \geq x_j \Longrightarrow y = x_j$
maximum	$y = \max(x_i, x_j)$	$x_i \geq x_j \Longrightarrow y = x_i$
		$x_i \leq x_j \Longrightarrow y = x_j$
modulo	$z = x \mod k$	$z \leq k - 1$
		$ky + z - x = 0$
Higher-order constraints	$x = (C)$	$(\delta = 1) \Longleftrightarrow (C \text{ holds})$
		$x = \delta$
`alldifferent`$(x_1, \ldots, x_n)$		$x_i \neq x_j\ \ \forall\, 1 \leq i < j \leq n$
`distribute`$(card, value, base)$		$card[i] = \sum_{j \in R} \delta_{ij}\ \forall\, i \in S$
`atleast`$(low, value, base)$		$low[i] \leq \sum_{j \in R} \delta_{ij}\ \forall\, i \in S$
`atmost`$(up, value, base)$		$up[i] \geq \sum_{j \in R} \delta_{ij}\ \forall\, i \in S$
`atleastatmost`$(low, up, value, base)$		$low[i] \leq \sum_{j \in R} \delta_{ij} \leq up[i]\ \forall\, i \in S$

Table 4.2: Transformation of OPL models into MIPs

However, it may be useful to examine whether the given constraints can be expressed in OPL more easily and efficiently by means of, for instance, logical conditions, disequalities, symbolic constraints, or functions like minimum, maximum, and absolute value.

Chapter 5

OPL Models for FAP

We try to solve the Frequency Assignment Problem by means of ILOG OPL Studio using different OPL models. Each of these models has its own advantages and disadvantages and is useful to be applied in specific situations. We distinguish between TRX-based and cell-based models. In Section 5.1, a comparison of the cell-based and the TRX-based approach, some general notes, and some important details on our models are given. Sections 5.2 and 5.3 introduce the TRX-based and the cell-based models for feasibility problems, respectively. Our investigations on minimizing total interference are reported in Section 5.4. Using OPL, we are not restricted to linear models. Thus, we also try a non-linear OPL model which is discussed in Section 5.5. The computational results of our tests are reported in Chapter 6. In the following, the models are explained step by step; complete models can be found in the Appendix.

5.1 Introducing TRX-based and cell-based Models

This section gives an overview on our models. The differences between them, their similarities, as well as their advantages and disadvantages are discussed.

5.1.1 Preliminaries

As already mentioned, we distinguish between TRX-based and cell-based models. The input graph to the former ones is the carrier network, while the latter ones work on the cell graph, i.e., the graph where the nodes are the given cells and the edges represent relations among the cells.

All models share the same structure and can be split into the same parts: data definition, variables, objective, constraints, and search heuristic. Furthermore, the definition of constraints is always according to the same scheme. Recall from Section 2.2 that an assignment is feasible if and only if (i) no assigned channel is blocked and if (ii) all separation requirements are met. Hence, each model formulates constraints enforcing these requirements.

Roughly speaking, we have "base models" that can be extended as needed in the TRX-based as well as in the cell-based case. The TRX-based models differ from

each other in the way the input data is specified, which and how constraints are formulated and which search heuristic is applied. The same holds for cell-based models. Each of these models may be extended by some optional constraints.

All models mentioned so far represent feasibility problems rather than optimization problems. In our case, the OPL built-in optimization procedures proved not to be convincing. Hence, we decided to minimize total interference by means of a preprocessing technique called *tightening the separation*, see Eisenblätter [7, Section 4.1.2]. We have a TRX-based and a cell-based model for that purpose; they are considered in Section 5.4.

5.1.2 Differences between cell-based and TRX-based Models

Recall from Section 2.1 that several kinds of separation requirements have to be respected: co-site separation, co-cell separation, and hand-over separation. In addition, a minimum required separation may be specified for each pair of cells individually. The input graph to the TRX-based model is a carrier network, which contains no information on cells and sites. Hence, for TRX-based models, separation values for pairs of TRXs have to be set appropriately in order to satisfy all these requirements. In addition, co-cell separation has to be taken into account for all TRXs of one cell, co-site separation has to be respected for all TRXs of one site, and hand-over relations and their corresponding separation values have to be considered for all involved TRXs. The computation of correct separation values for pairs of TRXs and the transformation of the given cell graph into a carrier network has to be done in preprocessing.

TRX-based models are significantly smaller than their cell-based counterparts since separation values for pairs of TRXs need not be computed within the model. For cell-based models, all separation values are specified for pairs of cells. In fact, co-cell separation and, as described in Section 5.1.3, hand-over separation requirements are constraints on TRXs. For cell-based models, they cannot be taken into account in preprocessing and have to be respected inside the model, which leads to more than twice the number of constraints as for TRX-based models. However, the information on the cell structure allows to state some additional constraints for cell-based models that cannot be formulated for TRX-based models without providing additional information.

5.1.3 The Role of the Preprocessing

OPL does not allow the addition of constraints dynamically during the computation. Hence, we have to add all information to the model in the beginning. The following transformations, including the computation of separation values for pairs of TRXs and the compilation of the carrier graph from the cell graph are done in preprocessing when generating the model data.

Integerization

The values for co- and adjacent channel interference are made integer in preprocessing since all algorithms presented in Chapter 3 are based on discrete variables. This is no problem since all these values can be truncated to at most four decimal places without loosing significant precision.

Input Graphs

Actually, the given problems are directed. In particular, this means that the cell graphs are directed and the values for separation, co- and adjacent channel interference need not be symmetric. In addition, we are given some directed hand-over relations between pairs of cells. In particular, a possible hand-over from cell u to cell v does not tell anything about the possibility of a hand-over from cell v to cell u.

We transform the given directed problems into undirected ones by considering only edges $\{u, v\}$ with $u < v$, after ensuring that for each given edge (u, v), an edge (v, u) exists. The separation value of an undirected edge is defined to be the maximum of the separation values of the underlying directed edges. The values for co- and adjacent channel interference of an undirected edge are the sum of the co- and adjacent channel interferences, respectively, of the underlying directed edges.

Hand-over Separation

For TRX-based problems, the necessary hand-over separations can be taken into account when computing the separation values for pairs of TRXs. For cell-based models, the handling of the hand-over relations is more difficult since the values for required hand-over separation are given for pairs of TRXs and depend on whether the involved TRXs are BCCHs or TCHs. In particular, different values may be required for a hand-over from a BCCH to a TCH and vice versa. To realize the distinction of the hand-over direction, we introduce a hand-over value for each undirected edge $\{u, v\}$ of the cell graph, which can take on one out of four possible values indicating whether (i) no hand-over is possible in either direction, (ii) whether a hand-over is only possible from u to v, (iii) whether a hand-over is only possible from v to u, or (iv) whether a hand-over is possible in both directions. This approach enables us to correctly set hand-over separation values inside cell-based models, but, in the worst case, it doubles the number of formulated separation constraints compared to the TRX-based model.

Available Channels

The available channels of a cell or of a carrier are also determined in preprocessing, where a cell and all contained TRXs have the same set of available channels. We are given the lower and the upper bounds of the spectrum. As the spectrum need not be continuous, we are also given a set of globally blocked

channels that excludes non-available channels from the given interval. Moreover, a set of locally blocked channels may be specified for each cell. For each TRX or for each cell (depending on the model), we compute the list of available channels by taking the lower and upper bounds and excluding globally as well as locally blocked channels. Afterwards, these lists are added to the input data of the models.

Redundant Constraints

A set of redundant constraints for TRX-based as well as for cell-based models is obtained by computing maximum cliques and some further large cliques for the corresponding carrier and cell graphs, and by employing the `alldifferent` constraint to enforce that all TRXs of each clique get different channels. These cliques are also determined in preprocessing.

5.2 TRX-based Feasibility Model

In this section, our TRX-based models are presented. The modeling of the data, the variables, the objective, the constraints, and the search heuristic is discussed in detail. Sections 5.2.1 to 5.2.5 deal with the "base model", while the optional parts of TRX-based models are examined in Section 5.2.6.

5.2.1 Model Data

The data definition of TRX-based feasibility models is shown in Figure 5.1. Rows starting with a "//" are comments, while Row (02) defines a non-negative integer `number_TRXs` representing the number of TRXs of the problem. Thereby, stating the `= ...` makes it possible to employ a data file for the initialization of the variables. The nodes of the graph are defined in Row (03) by means of a range from 0 to `number_TRXs`-1.

Rows (06) to (10) define a record `Edge_type` used to define the undirected edges of the model. Three parameters are given for each edge: both end nodes u and v, and the minimum required separation. This data structure could be used to define directed edges as well, but the convention $u < v$ allows the edges to be considered undirected. The set of edges of the model is defined in Row (13); each edge is of type `Edge_type`. Many separation values for edges in the carrier graph (as well as in the cell graph) are zero, see Table 6.1 on page 77. Even if one or both of the interference values differ from zero, these edges need not be considered since only the required separation is relevant for formulating the constraints. Hence, only such edges with a minimum required separation of at least 1 are given in the model.

Rows (16) and (17) define the lower and upper bounds of the available spectrum, respectively, while Row (18) defines a range `Channel` of generally available channels for the scenario. A set of available channels for each TRX is given in Row (21). This is done as an array of sets of channels, whose name is

```
01: // number of TRXs
02: int+ number_TRXs                    = ...;
03: range TRX 0..number_TRXs-1;
04:
05: // define a record for the set of edges
06: struct Edge_type {
07:    TRX u;
08:    TRX v;
09:    int+ Sep;
10: };
11:
12: // the given set of edges
13: {Edge_type} Edges                    = ...;
14:
15: // the given spectrum
16: int+ first_channel                   = ...;
17: int+ last_channel                    = ...;
18: range Channel [first_channel..last_channel];
19:
20: // the set of available channels for each cell
21: {Channel} Available_channels[TRX]   = ...;
```

Figure 5.1: TRX-based models: data definition

`Available_channels`. Hence, given a TRX t, `Available_channels`$[t]$ returns the set of its available channels.

Even though it is not complicated to state any OPL model of a given discrete optimization problem, one should avoid pitfalls when formulating the models. Recall from Section 2.2 that the values for interference and required separation are given as matrices. A simple approach would be to actually store these values inside OPL as matrices. Because all matrices are symmetric, another possibility could be to only store the upper right-hand sides of the matrices in corresponding vectors. However, in both cases many unnecessary data would be stated since values would be defined for each possible edge in the given graphs, and the graphs are far from being complete. We observed the impact of these ideas in a few test runs. For the instance "siemens3", see Section 6.1, memory consumption rises from 60 MB if using edge sets to 340 MB if applying vectors, and to 547 MB if employing matrices. Moreover, no bradford instance is solvable at all. Another possible pitfall would be to add all edges to the model instead of only those edges with a necessary separation of at least one, and to also state separation constraints for edges with a minimum required separation of zero, which would increase the number of stated constraints by a factor of 50 to 100.

5.2.2 Variables

This section deals with the definition of the variables of the model. The OPL excerpt is given in Figure 5.2. One variable is needed for each TRX to store

its assigned channel. Row (27) of the model defines an array of variables called `chan`, where the variables are of type `Channel`.

```
26: // variables
27: var Channel chan[TRX];
```

Figure 5.2: TRX-based models: variable definition

5.2.3 Objective

Recall that we consider feasibility problems only and do not minimize total interference. The objective is stated simply with the keyword `solve` followed by the formulation of the constraints, see Figure 5.3.

```
29: solve {
30:

   [... definition of the constraints ... ]

38: };
```

Figure 5.3: TRX-based models: stating the objective

5.2.4 Constraints

In order to obtain feasible assignments, two kinds of constraints have to be satisfied. On the one hand, no assigned channel must be locally or globally blocked. On the other hand, all separation requirements have to be met. Recall that for the TRX-based models, all separation requirements such as co-cell, co-site, and hand-over separation are taken into account in the separation values for the given edges. The corresponding OPL excerpt is given in Figure 5.4.

```
31:   // assign only allowed channels
32:   forall( t in TRX )
33:      chan[t] in Available_channels[t];
34:
35:   // respect separation constraints
36:   forall( e in Edges )
37:      abs( chan[ e.u ] - chan[ e.v ] ) >= e.Sep;
```

Figure 5.4: TRX-based models: stating the constraints

Rows (32) and (33) impose that each TRX is assigned one of its available channels, and Rows (36) and (37) ensure that the separation requirements are satisfied for each edge e.

Alternative Formulations

As mentioned in Section 4.1, the use of `predicates` may lead to a more effective pruning compared to simply stating the constraints. The conditions ensuring the minimum required separation between TRXs can be rewritten with `predicates`. The `predicate` definition as well as the rewritten constraints are given in Figure 5.5.

```
25a:   predicate respect_sep(int u, int v, int s)
25b:      return u-v >= s \/ v-u >= s;

   [ ... ]

35':   // respect separation constraints
36':   forall( e in Edges )
37':      respect_sep( chan[ e.u ], chan[ e.v ], e.Sep );
```

Figure 5.5: TRX-based models: alternative formulation of constraints

The "\/" stands for a logical OR in OPL. Rows (25a) and (25b) define the `predicate respect_sep`. It has integers u, v, and s as parameters and ensures that the difference between u and v is at least s. Absolute values could be employed instead of the logical OR, but this would lead to larger solution times in our cases. In Row (37$'$), the `predicate` is applied for each edge e of the problem.

5.2.5 Search Heuristic

The search heuristic in OPL is used to develop the search tree and is a very important aspect of the model. The definition of a search heuristic consists of two parts. First, a rule is defined how to choose the variable which is to be assigned next. This is done with the keyword `forall`. Second, criterions for the choice of the value are specified by means of `try` or `tryall`. Each node in the search tree corresponds to one CSP with some additional conditions. This could be, for example, fixing of variables or domain splittings. The nodes in the search tree are also referred to as choice points. For more information, see [12].

We devise a search heuristic, which we call *Smallest Domain Size*. Unfortunately, the quality of the obtained solutions is not satisfactory. Thus, we also implement variants of the heuristics *T-Coloring* and *DSATUR with Costs*, which proved well-suited for obtaining good results in reasonable time when employed by Eisenblätter while developing a tool for automatic frequency planning in GSM networks [7]. However, the OPL implementations of *T-Coloring* and *DSATUR with Costs* are not successful and outperformed by the heuristic *Smallest Domain Size* with respect to solution times as well as to the quality of the solutions, see Chapter 6.

Smallest Domain Size

Figure 5.6 shows our search heuristic *Smallest Domain Size*. The OPL function `dsize` takes a variable as parameter and returns the current domain size of the variable. Hence, Row (42) selects a TRX whose corresponding variable has the least number of possible values left. The `tryall` statement in Rows (43) and (44) selects the channel to be assigned to the chosen carrier. The function `nbOccur` takes a value c and an array a of variables as parameters and returns the number of occurrences of c in a. Given a TRX t, the value to be assigned to t is an available channel which is most seldomly assigned to any other TRX so far. The idea is to prevent co-channel interferences as far as possible. The actual assignment of the variable is done in Row (45).

```
40: // search heuristic
41: search {
42:    forall( t in TRX ordered by increasing dsize(chan[t]) )
43:       tryall ( c in Available_channels[t] ordered by
44:                increasing nbOccur(c, chan))
45:          chan[t] = c;
46: };
```

Figure 5.6: TRX-based models: search heuristic *Smallest Domain Size*

T-Coloring

Given an undirected graph $G = (V, E)$ and non-empty finite sets $T(u, v)$ of non-negative integers for all $uv \in E$, a T-coloring of G is a labeling f of the nodes of G with non-negative integers such that $|f(u) - f(v)| \notin T(uv)$ for all edges $uv \in E$. An instance of *list coloring* is given as a graph and lists of colors for each node. The task is to find a labeling of the nodes such that each node receives a color from its list and such that no two adjacent nodes are assigned the same color. Frequency assignments have to meet list coloring constraints, because an available channel has to be assigned to each carrier. Moreover, frequency assignments also have to respect T-coloring constraints since the separation requirements have to be met. For more information on these *list T-colorings*, see Eisenblätter [7, Section 4.2.1].

Two values are maintained by the *T-Coloring* heuristic for each carrier: the *saturation degree* and the *spacing degree*. Let $d(vw)$ be the minimum necessary separation for the edge $vw \in E$, the saturation degree *satdeg* of a carrier v is the number of its unavailable channels, whereas the spacing degree *spadeg* of v is $\sum_{vw \in E\,:\,w \text{ unassigned}} d(vw)$. The spacing degree estimates the impact that the assignment of all yet unassigned neighbors of v would have to the assignability of v. The choice of the variable is done by means of both values. The variable v with the maximum saturation degree is chosen. In case of ties, the variable with the maximum spacing degree is taken. Further ties are broken arbitrarily. The channel to be assigned to the chosen carrier v is the available channel of

v with least index. The idea behind this heuristic is the same as for *DSATUR* suggested by Brélaz, namely to choose the variable which is "hardest to deal with" next.

Figure 5.7 presents the OPL implementation of this heuristic. For each TRX u, every neighbor v and the necessary separation between u and v are needed. Since the edge list `Edges` does not provide this information, we have to add a new data member `Neighbors` to the model. Each entry in the adjacency list of a given TRX u is a structure with two members: a neighbor v of u and the required separation between u and v. Row (50) chooses the TRX with the smallest number of values in its domain. This corresponds to choosing the carrier with the maximum number of blocked channels. In case of ties in Row (50), Row (51) first calculates the spacing degree and then takes the carrier with the maximum spacing degree. The OPL function `bound` is used, which test whether a variable has already received a channel. Row (53) selects the available channel with least index for the chosen carrier, while Row (54) assigns this channel.

```
48: search {
49:    forall( t in TRX ordered by increasing
50:            < dsize(chan[t]),
51:              -(sum( v in Neighbors[t]: not bound(chan[v.trx]) ) v.Sep)
52:            > )
53:       tryall( c in Available_channels[t] ordered by increasing c )
54:          chan[t] = c;
55: };
```

Figure 5.7: TRX-based models: search heuristic *T-Coloring*

DSATUR with Costs

DSATUR with Costs is another modification of *DSATUR* and aims at producing a feasible assignment with least possible total interference, see Eisenblätter [7, Section 4.2.2] for details. A matrix `cost_of_channel` is used to record the costs of all carrier/channel combinations. These costs consist of co- and adjacent interference. For each unassigned carrier, a value *key* is defined. Let B_v be the set of locally blocked channels for a carrier v and let M be a suitably chosen, large constant, the *key* of v is defined as

$$key(v) = |B_v|\, M + \sum_{c \in C \setminus B_v} cost\_of\_channel[v][c].$$

The carrier with the maximal key value is taken. The channel to be assigned to the chosen carrier is its available channel with the least value in row `cost_of_channel`$[v]$.

Figure 5.8 shows the implementation of *DSATUR with Costs* in OPL. Rows (58) to (61) select the TRX t with the maximal value $key(t)$. The number of blocked

channels is computed as the number of channels in the spectrum minus the number of channels still contained in the domain of t. Rows (62) and (63) choose the available channel with the least value in row `cost_of_channel`$[t]$, and Row (64) assigns this channel. The matrix `cost_of_channel` is maintained by means of additionally introduced constraints which are based on Higher-order constraints and are very similar to the cost constraints discussed in detail in Section 5.2.6.

```
57: search {
58:    forall( t in TRX ordered by decreasing
59:            (last_channel - first_channel + 1 - dsize(chan[t])) * M +
60:             sum( chan in Available_channels[t])
61:                cost_of_channel[ t, chan ] )
62:       tryall ( c in Available_channels[t] ordered by
63:                increasing cost_of_channel[t,c] )
64:          chan[t] = c;
65: };
```

Figure 5.8: TRX-based models: search heuristic *DSATUR with Costs*

5.2.6 Optional Parts

The presented "base model" can be extended by stating constraints calculating the costs of an assignment (by default, the costs are determined in post processing) and by stating constraints ensuring that all TRXs of each given clique are assigned different channels. Additional data members and variables have to be defined in order to formulate these constraints.

Optional Data Members

Two types of optional data definition are presented in this section: the definition of values for co- and adjacent channel interference, and the definition of cliques in the graph. The adapted OPL excerpt is presented in Figure 5.9.

In order to specify values for co- and adjacent channel interference, two data members `Co_c_i` and `Adj_c_i` have to be added to the definition of the structure `Edge_type` in Rows (06) to (10). Again, edges are added to the input of the model only if at least one of the values for separation, co- or adjacent channel interference differs from zero.

The variables for the set of cliques determined in preprocessing are defined in Row (24). Each clique is a set of TRXs. Since the cliques themselves form a set, too, the data member `Cliques` is of type set of set of TRXs.

Optional Variables

To be able to compute the costs of an assignment, we have to introduce additional variables. The corresponding optional parts of the model are given in Figure 5.10.

```
05 : // define a record for the set of edges
06 : struct Edge_type {
07 :    TRX u;
08 :    TRX v;
09 :    int+ Sep;
09a:    int+ Co_c_i;
09b:    int+ adj_c_i;
10 : };

[...]

23:  // optionally: a set of cliques
24:  {{TRX}} Cliques                    = ...;
```

Figure 5.9: TRX-based models: optional data definition

```
28a: // calculate upper bounds on occuring interferences
28b: int+ max_co_c_i  = sum( e in Edges ) e.Co_c_i;
28c: int+ max_adj_c_i = sum( e in Edges ) e.Adj_c_i;
28d:
28e: var
28f:     int+ co_c_i       in 0..max_co_c_i,
28g:     int+ adj_c_i      in 0..max_adj_c_i,
28h:     int+ total_costs  in 0..max_co_c_i+max_adj_c_i;
```

Figure 5.10: TRX-based models: optional variable definition

All variables have to be restricted to finite intervals. Thus, we have to calculate upper bounds on the total co- and adjacent channel interferences to be able to introduce the new variables properly. The upper bounds are determined in Row (28b) and (28c), where the sum is taken over all edges in the set `Edges` of the values for co- and adjacent channel interference, respectively. Actually, it would suffice to introduce one variable for the total costs of an assignment, but the additional effort spent on distinguishing between co- and adjacent channel interference is negligible. Row (28f) defines the variable `co_c_i` representing the total amount of co-channel interference of the assignment. It ranges in the interval $[0, \texttt{max\_co\_c\_i}]$. In the same manner, the variables `adj_c_i` and `total_costs` are introduced in Row (28g) and Row (28h).

Optional Constraints

The constraints calculating the costs of an assignment and the constraints taking cliques in the given graph into account are discussed in this section. Notice that the cost constraints do not impose any restrictions. They are used to set the values of the optional variables, which are for informational issues only. The optional constraints are presented in Figure 5.11.

The calculation of the costs of an assignment is done by means of Higher-order constraints (as explained in Section 4.1); hence, we refer to these cost constraints

```
37a:
37b:    // calculate co_c_i
37c:    co_c_i  = sum(e in Edges: e.Co_c_i  > 0 )
37d:       ( chan[e.u] = chan[e.v] ) * e.Co_c_i;
37e:
37f:    // calculate adj_c_i
37g:    adj_c_i = sum(e in Edges: e.Adj_c_i > 0 )
37h:       ( abs( chan[e.u] - chan[e.v] ) = 1 ) * e.Adj_c_i;
37i:
37j:    // calculate total costs
37k:    total_costs = co_c_i + adj_c_i;
37l:
37m:    // consider cliques
37n:    forall( clique in Cliques )
37o:       alldifferent( all( trx in clique ) chan[ trx ] );
```

Figure 5.11: TRX-based models: stating optional constraints

as *Higher-order Costs*. The total amount of co-channel interference is determined by the constraints in Rows (37*c*) and (37*d*). For each edge e, a binary variable is associated with the Meta-constraint `( chan[e.u] = chan[e.v] )`, which equals 1 if and only if both end nodes u and v of edge e received the same channel. Thus, the sum takes exactly those edges into account where co-channel interference occurs between both end nodes. However, the value for co-channel interference may be equal to zero for some of the edges. This may happen if at least one of the values for adjacent channel interference or separation on this edge is greater than zero. To prevent unnecessary summations, we exclude such edges with the condition `e.Co_c_i > 0`.

The calculation of total adjacent channel interference in Rows (37*g*) and (37*h*) parallels that of co-channel interference: the binary variables associated with `( abs( chan[e.u] - chan[e.v] ) = 1 )` equal 1 if and only if the channels assigned to u and v differ by 1. The total costs of an assignment are computed by the constraint in Row (37*k*).

The drawback of this approach is that it leads to many new variables. In the worst case, two binary variables are introduced for each edge. This significantly enlarges the search space, and may lead to longer solution times. To overcome this, we apply another way to compute the costs of an assignment, which is called *Implication Costs*. Only one variable `edge_cost` is introduced for each edge, corresponding to the cost of this edge. Afterwards, logical conditions are used to set the variables `edge_cost` appropriately. Even though this alternative needs only half as many additional variables as our first approach, it proved not to be useful in practice. Details on the tests with both kinds of cost constraints are given in Chapter 6.

In the following, we assume that some cliques of TRXs are given in the subgraph of the carrier graph consisting only of those edges with a separation requirement

of at least 1. The `alldifferent` constraint, and therefore extra propagation techniques for symbolic constraints, can be employed to ensure for each clique that a different channel is assigned to all involved TRXs. We call these kind of redundant constraints *Cliques*. For each given clique, Row (37*o*) ensures that the assignments of all involved TRXs differ from each other. It is possible that separation requirements on some edges of a clique are greater than 1. In this case, `alldifferent` enforces less than the actually needed separation, and it is not clear if stating this additional constraint yields any benefit. However, its formulation seems promising since redundant constraints are combined with the use of symbolic constraints.

5.3 Cell-based Feasibility Model

Our cell-based models are discussed in detail in this section. Most of the ideas explained for TRX-based models also hold for cell-based models. However, additional data and constraints are defined since co-cell and hand-over separation requirements have to be considered within the model. Furthermore, there are more possibilities to state redundant constraints.

5.3.1 Model Data

The first part of our cell-based models is the data definition. The corresponding excerpts of our OPL models are shown in Figures 5.12 and 5.13. Row (02) declares a non-negative integer, which is the number of cells of the problem. The nodes of the graph are defined in Row (03) using a range from 0 to `number_cells`-1. Row (07) declares an array of integers to store the number of TRXs in each cell.

Like in our TRX-based models, we define a structure `Edge_type` in Rows (10) through (17), which is used to model the undirected edges of the problem. For each edge, both end nodes, the required separation, and the hand-over value indicating the directions of possible hand-overs are given. The set of edges is defined in Row (20). As above, only those edges $\{u, v\}$ are given where at least one of the values for required separation or possible hand-over is greater than zero.

Rows (23) to (26) declare a structure used to define the TRXs. Two attributes are given for each TRX: the cell containing it, and its number in this cell. The following convention is made: the first TRX in each cell is the BCCH, whereas TRXs with $n \geq 2$ are TCHs. The set of TRXs is computed in Row (29). Each set member is a structure of type `TRX_type`. A structure in OPL is accessed by means of the brackets "<" and ">". As an example, `<3,1>` denotes the BCCH ($n = 1$) of the third cell ($cell = 3$).

The upper and lower bounds of the spectrum and the range of channels are defined as in the TRX-based models in Rows (32) to (34). The set of available

```
01: // number of cells
02: int+ number_cells                    = ...;
03: range Cell 0..number_cells-1;
04:
05: // number of TRXs per cell: BCCH + TCH's
06: // 1: BCCH, 2..n: TCH's
07: int+ number_TRXs[Cell]               = ...;
08:
09: // define a record for the set of edges
10: struct Edge_type {
11:    Cell u;
12:    Cell v;
13:    int+  Sep;
14:    // hand-over relations between cells
15:    // 0: no HO, 1: HO(u->v), 2: HO(v->u), 3: HO(u<->v)
16:    int+  Ho;
17: };
18:
19: // the given set of edges
20: {Edge_type} Edges                    = ...;
21:
22: // define a record for the set of TRXs
23: struct TRX_type {
24:    Cell cell;
25:    int+ n;
26: };
27:
28: // build the set of all TRXs
29: {TRX_type} TRX = { <c,t> | c in Cell & t in 1..number_TRXs[c] };
```

Figure 5.12: Cell-based models: data definition (1/2)

channels for each cell is given in Row (37) by means of an array of sets of channels.

The value for required co-cell separation may differ from cell to cell. Hence, Row (40) defines the necessary co-cell separation for each cell by means of an array. The values of required hand-over separation are given in Rows (43) to (46). These values may depend on whether the involved TRXs are BCCHs or TCHs. Thus, we are given four different values. Row (47) computes the maximum of two of the required hand-over separation values. The OPL function `maxl` is employed, which returns the maximum of a given list of parameters.

5.3.2 Variables

The definition of the variables is the same as for TRX-based models, even though the data member `TRXs` is defined as a set of structures in cell-based models and as a range of integers in TRX-based models. Figure 5.14 shows the OPL excerpt.

```
31: // the given spectrum
32: int+ first_channel                  = ...;
33: int+ last_channel                   = ...;
34: range Channel [first_channel..last_channel];
35:
36: // the set of available channels for each cell
37: {Channel} Available_channels[Cell]  = ...;
38:
39: // co-cell separation
40: int+ Co_cell_separation[Cell]       = ...;
41:
42: // hand-over separation
43: int+ Ho_bcch_to_bcch       = ...;
44: int+ Ho_bcch_to_tch        = ...;
45: int+ Ho_tch_to_bcch        = ...;
46: int+ Ho_tch_to_tch         = ...;
47: int+ Ho_max_bcch_tch       = maxl( Ho_bcch_to_tch, Ho_tch_to_bcch );
```

Figure 5.13: Cell-based models: data definition (2/2)

```
49: // variables
50: var Channel chan[TRX];
```

Figure 5.14: Cell-based models: variable definition

5.3.3 Objective

The objective is again simply stated by the keyword `solve` followed by the definition of the constraints since we only consider feasibility problems here.

5.3.4 Constraints

In addition to the constraints discussed for TRX-based models, co-cell separation and hand-over separation requirements have to be respected inside the cell-based models. Figure 5.15 presents the corresponding OPL excerpt.
Rows (55) and (56) are similar to TRX-based models and ensure that each TRX is assigned one of its available channels.

The constraint in Rows (59) to (61) enforces co-site separation constraints as well as individual separation requirements for pairs of cells. Conditions are formulated for all pairs of TRXs in both end nodes of each edge. The requirement in Rows (64) and (65) imposes co-cell separation constraints for each cell, where the cell specific co-cell separation is enforced for each pair of TRXs i and j with $i < j$.

The handling of hand-over separations is more complex. On the one hand, we have to differentiate between BCCHs and TCHs, on the other hand, the direction of the hand-overs has to be respected. Table 5.1 shows the appropriate hand-over separation values depending on the direction of the hand-over and

```
54:    // assign only allowed channels
55:    forall (t in TRX)
56:       chan[t] in Available_channels[t.cell];
57:
58:    // separation constraints
59:    forall( e in Edges )
60:       forall( t1 in 1..number_TRXs[e.u], t2 in 1..number_TRXs[e.v] )
61:          abs( chan[<e.u,t1>] - chan[<e.v,t2>] ) >= e.Sep;
62:
63:    // respect co-cell separation
64:    forall( c in Cell, ordered i,j in 1..number_TRXs[c] )
65:       abs( chan[<c,i>] - chan[<c,j>] ) >= Co_cell_separation[c];
66:
67:    // hand-over separation u->v, v->u, u<->v
68:    forall( e in Edges: e.Ho >= 1 ) {
69:       // BCCH <-> BCCH
70:       abs( chan[<e.u,1>] - chan[<e.v,1>] ) >= Ho_bcch_to_bcch;
71:       // TCH <-> TCH
72:       forall( t1 in 2..number_TRXs[e.u], t2 in 2..number_TRXs[e.v] )
73:         abs( chan[<e.u,t1>] - chan[<e.v,t2>] ) >= Ho_tch_to_tch;
74:    };
```

Figure 5.15: Cell-based models: stating the constraints

on the type of involved TRXs. A possible hand-over between a TRX of cell $c1$ and a TRX of cell $c2$ with $c1 < c2$ is described. The table is divided into four parts, corresponding to the four possible directions of a hand-over between two cells. We discuss some examples. First, consider the case that a hand-over is only possible from $c1$ to $c2$, which corresponds to a hand-over value of 1. Given a TRX with $n > 1$ in $c1$ and a TRX with $n = 1$ in $c2$, we have to impose between these two the requirement for a hand-over from a TCH to a BCCH. Assuming now, that a hand-over only is possible from $c2$ to $c1$, and considering again a TCH in $c1$ and a BCCH in $c2$, we have to ensure the requirement for a hand-over from a BCCH to a TCH between both TRXs.

Four related constraints are formulated in the cell-based models to ensure all hand-over conditions. One of them is the constraint in Rows (68) to (74) presented on page 66, imposing the conditions that are independent of the direction of a possible hand-over, i.e., the constraints between pairs of BCCHs and pairs of TCHs. Rows (68) selects all edges with a possible hand-over in at least one direction. Given an edge e, Row (70) enforces the minimum required separation between both BCCHs. For each pair of TCHs in both end nodes u and v of e, the required hand-over separation is ensured by the constraint in Rows (72) to (74). The remaining constraints modeling the direction dependent part of the hand-over relations are similar to the discussed one, but not explained in detail here. The complete cell-based model including these constraints can be found in the Appendix on page 97.

[1] maximum of required hand-over separation from a BCCH to a TCH and vice versa.

Cell		$c1$							
Hand-over		no		$c1 \rightarrow c2$		$c1 \leftarrow c2$		$c1 \leftrightarrow c2$	
HO value		0		1		2		3	
	TRX	1	> 1	1	> 1	1	> 1	1	> 1
$c2$	1	0	0	BCCH ↓ BCCH	TCH ↓ BCCH	BCCH ↓ BCCH	BCCH ↓ TCH	BCCH ↓ BCCH	Max [1]
	> 1	0	0	BCCH ↓ TCH	TCH ↓ TCH	TCH ↓ BCCH	TCH ↓ TCH	Max	TCH ↓ TCH

Table 5.1: Separation values for a hand-over of active calls

5.3.5 Search Heuristic

We also implement the search heuristics *Smallest Domain Size*, *T-Coloring*, and *DSATUR with Costs* for cell-based models. However, the syntax slightly differs from the implementation for TRX-based models since the modified data definition have to be taken into account.

Moreover, the heuristic *Smallest Domain Size* is modified in order to use the additional information on cells. The adapted OPL excerpt is presented in Figure 5.16. Again, a TRX is chosen with the least number of possible values left. Unlike for TRX-based models, the TRX contained in the cell with the highest number of TRXs is taken in case of ties. The choice of the channel is the same as for TRX-based models. The additional definition of the tie breaker has a major impact on the performance of this heuristic; without this condition, the heuristic is hardly usable.

```
124: search {
125:    forall( t in TRX ordered by increasing
126:          <dsize(chan[t]), -number_TRXs[t.cell]> )
127:       tryall ( c in Available_channels[t.cell] ordered by
128:                increasing nbOccur(c, chan))
129:          chan[t] = c;
130: };
```

Figure 5.16: Cell-based models: search heuristic *Smallest Domain Size*

5.3.6 Optional Parts

The optional constraints *Higher-order Costs* and *Cliques* for TRX-based models can be formulated for cell-based models as well. In addition, it is possible to state constraints breaking symmetries between the TCHs within each cell and to impose conditions on the minimum value of the sum of all channels assigned to one cell.

Optional Data Members

The definition of optional data members is analogous to the case of TRX-based models. In order to be able to compute the costs of an assignment, we have to add two entries `Co_c_i` and `Adj_c_i` to the definition of the structure `Edge_type`. Moreover, the definition of the data member `Cliques` is also like for TRX-based models, except that the cliques are given as a set of cells rather than as a set of TRXs.

Optional Variables

The definition of the variables used to store the costs of an assignment is the same as in the TRX-based case.

Optional Constraints

The definition of the constraints *Higher-order Costs* and *Cliques* parallels the TRX-based case as well.

All TCHs within one cell can be considered equivalent since all of them have to fulfill the same separation requirements. Hence, given the set of TCHs of one cell and the set of channels to be assigned to them, the specific assignment to all of them is not relevant. These symmetries can be broken by introducing an ordering on the assignments of all TCHs in one cell. We refer to these constraints as *Break Symmetries*.

Furthermore, for each cell, the required co-cell separation has to be taken into account when assigning all contained TRXs. This leads to a redundant constraint on the minimum value of the sum of all channels assigned to one cell. We call these requirements *Sum Channels*. Let lb be the lower bound of the given spectrum and let $d_{co\_cell}(c)$ be the required co-cell separation for a given cell c containing the TRXs $t_1, \ldots, t_n$, any feasible assignment y has to satisfy:

$$\begin{aligned} \sum_{i=1}^{n} y(t_i) \geq & \sum_{i=1}^{n} \Big(lb + (i-1)\, d_{co\_cell}(c) \Big) \\ = & \; n \cdot lb + d_{co\_cell}(c) \sum_{i=1}^{n} (i-1) \\ = & \; n \cdot lb + d_{co\_cell}(c)\, \frac{n(n-1)}{2} \\ = & \; n \left(lb + \frac{n-1}{2}\, d_{co\_cell}(c) \right) \end{aligned} \tag{5.1}$$

The excerpt with both new constraints is shown in Figure 5.17. The constraint in Rows (113) to (115) imposes an ordering on the channels assigned to the TCHs of one cell. Thereby, only cells with at least 3 TRXs need to be considered. The constraint in Rows 118 to 121 ensures Inequality (5.1) for each cell with at least two TRXs.

```
112:    // break symmetries between TCH's within one cell
113:    forall( c in Cells: number_TRXs[c] >= 3 )
114:       forall( i in 2..number_TRXs[c]-1 )
115:          chan[<c,i>] < chan[<c,i+1>];
116:
117:    // constraint on the sum of assigned channels within one cell
118:    forall( c in Cells: number_TRXs[c] >= 2 )
119:       ( number_TRXs[c] * ( first_channel +
120:            (number_TRXs[c]-1) / 2 * Co_cell_separation[c] )
121:       ) <= sum( i in 1..number_TRXs[c] ) chan[<c,i>];
```

Figure 5.17: Cell-based models: stating optional constraints

5.4 Minimizing Total Interference

Up to now, we have considered feasibility problems only. The quality of the solutions obtained with the above models is often poor in comparison to known reference solutions. Hence, we try to explicitly minimize total interference.

As described in Section 3.1, Constraint Programming offers two approaches to minimize or maximize a given objective function: standard search and dichotomic search. Standard search first determines some feasible solution for the problem, and then tries to improve on it step by step, tightening a bound on the objective value. On the contrary, dichotomic search performs a binary search on the objective value. Both methods can be used in OPL but proved not to be successful in our case. Using standard search, the objective value of the initial solution is sometimes about 100 times higher than objective values of known reference solutions, while the attained improvement in each step is minimal. The first solution found by means of dichotomic search is unsatisfactory. This approach fails to determine any further assignment except the first one, even within several hours of continued computation. The reason may be that a lower bound on the objective value is needed in order to apply dichotomic search [19], but that no trivial lower bound greater than zero is available for most instances of the Frequency Assignment Problem, see [10].

To cope with this problem, we employ the preprocessing technique *tightening the separation*, see Eisenblätter [7, Section 4.1.2]. Given a certain threshold, all interference values beyond this value are ruled out by means of additionally introduced separation requirements: co-channel interference is eliminated with a separation of one, while co- and adjacent channel interference are prevented with a separation value of two.

We develop a framework to perform *tightening the separation*, which is explained in the following. Even though the goal is to minimize total interference, feasibility models are used to compute the actual assignments. Minimization is achieved by modifying the input data for these models. For a given scenario, the optimization procedure mainly consists of determining that *tightening the separation* threshold, which yields the least total interference.

Either a TRX-based or a cell-based feasibility model is applied to determine the solutions. As shown in Section 6.2, scenarios often can be solved faster by means of TRX-based models than with cell-based models, but cell-based models enable us to solve some scenarios that cannot be solved with TRX-based models at all. Hence, both kinds of models are employed for our approach. For the TRX-based model, no optional constraints are formulated, while the additional constraints *Cliques*, *Break Symmetries*, and *Sum Channels* are stated for the cell-based model. The search procedure *Smallest Domain Size* is applied in either case.

In order to perform *tightening the separation*, both feasibility models are extended by (*i*) adding the *tightening the separation* threshold to the model data and (*ii*) by defining an initialization procedure which modifies the given separation values in order to prevent interferences beyond the threshold. The corresponding OPL excerpt is presented in Figure 5.18. It applies for the cell-based as well as for the TRX-based model. Row (47*a*) defines a non-negative integer representing the given threshold, while the actual *tightening the separation* procedure is shown in Rows (47*d*) through (47*q*).

```
47a: int+ Threshold = ...;
47b:
47c: // tightening the separation
47d: initialize{
47e:    forall( e in Edges ) {
47f:       if e.Adj_c_i > Threshold & e.Sep < 2 then {
47g:          e.Sep = 2;
47h:       }
47i:       else {
47j:          if e.Co_c_i > Threshold & e.Sep = 0 then {
47k:             e.Sep = 1;
47l:          }
47m:          endif;
47n:       }
47o:       endif;
47p:    }
47q: };
```

Figure 5.18: Minimizing total interference: *tightening the separation*

ILOG OPL Studio offers the possibility to use scripts, similar to shell scripts in UNIX operating systems. These scripts allow solving a specific scenario several times consecutively. In addition, it is possible to modify parts of the input data.

The whole computation is controlled by either an OPL script or by a Perl script. In the beginning, the script determines an initial solution by means of a feasibility model. Subsequently, it computes the costs of this assignment and performs a binary search on the new *tightening the separation* threshold. Whenever a new threshold has been determined, *tightening the separation* with this value is performed and the input data for the feasibility model is altered.

After this, the script uses the feasibility model and the modified data to compute a new assignment.

Our computational tests show that the described costs constraints should not be used for feasibility models, because otherwise most of the calculation time is spent for computing the costs rather than for determining the actual assignments. The costs of an assignment are obtained by employing a second OPL model, which does not search for any solution, but only calculates the total amount of interference for a given assignment. The cost constraints *Higher-order Costs* are used for this purpose. This method reduces the solution times compared to directly stating the cost constraints for the feasibility model, but memory consumption is considerable in both cases. To avoid the calculation of the costs by means of OPL, we employ a Perl script instead of the OPL script, which computes the costs of a given assignment itself. This approach reduces memory consumption by a factor of 2, but the total solution time of the Perl script is in most cases slightly higher than the solution time of the OPL script. The reasons are explained in Section 6.4.1.

A time limit is used to abort the computation for the feasibility models when searching for a solution according to the current *tightening the separation* threshold. If no solution could be found during this time, it is assumed that there is no solution, and a higher threshold is chosen. Obviously, the choice of the time limit is very important for the success of the minimization approach. Setting the limit too high results in unnecessary long solution times. However, some solutions may not be found if the search is aborted too quickly. Our computational tests indicate that the solution times for the scenarios do not depend much on the choice of the current threshold. The time limit is set to 5 times the time needed to solve the scenario with the initial threshold. This proved to be a good compromise.

Altogether, we try three different approaches for minimizing total interference: an OPL script combined with a cell-based feasibility model, an OPL script combined with a TRX-based feasibility model, and a Perl script together with a cell-based feasibility model. The Perl script is not used in conjunction with the TRX-based model since our computational studies show that the cell-based model should be preferred to the TRX-based model when minimizing total interference. For details on the results of all three minimization methods, see Section 6.4.

5.5 Non-linear TRX-based Feasibility Model

As already pointed out, the quality of the solution obtained with the models discussed so far is often unsatisfactory. Since OPL offers the possibility to state and solve non-linear models, we also investigate this approach. After introducing our non-linear mathematical model for FAP, the corresponding OPL model is presented.

Mathematical Model

One binary variable y_{vc} is introduced for each TRX v and for each available channel c of v, which equals 1 if and only if TRX v is assigned the channel c. As in Section 2.2, C denotes the available spectrum, B_v is the set of locally blocked channels for each carrier $v \in V$, and $c^{co} : e \to [0,1]$, $c^{adj} : e \to [0,1]$, $d : e \to \mathbf{Z}_+$ define the co-channel interference, adjacent channel interference, and minimum required separation, respectively. An assignment y is feasible if and only if:

$$\sum_{c \in C \setminus B_v} y_{v,c} = 1 \quad \forall\, v \in V \tag{5.2}$$

$$y_{u,c_1} \cdot y_{v,c_2} = 0 \quad \forall\, (u,v) \in E,\ c_1 \in C \setminus B_u,\ c2 \in \{c \in C \setminus B_v :\ |c_1 - c_2| < d(u,v)\} \tag{5.3}$$

Equation (5.2) ensures that each TRX is assigned exactly one of its available channels, and the separation requirements are respected due to Condition (5.3).

The objective is to minimize the total amount of co- and adjacent channel interference and can be stated as follows:

$$\begin{aligned} \min \quad & \sum_{(u,v) \in E} \sum_{c \in C \setminus (B_u \cup B_v)} y_{uc} \cdot y_{vc} \cdot c^{co}(u,v) + \\ & \sum_{(u,v) \in E} \sum_{\substack{c \in C \setminus B_u: \\ c-1 \in C \setminus B_v}} y_{uc} \cdot y_{v,c-1} \cdot c^{adj}(u,v) + \\ & \sum_{(u,v) \in E} \sum_{\substack{c \in C \setminus B_u: \\ c+1 \in C \setminus B_v}} y_{uc} \cdot y_{v,c+1} \cdot c^{adj}(u,v) \end{aligned}$$

OPL Model

Having introduced our non-linear model for FAP, we now turn to its OPL implementation. For reasons described in Section "Optional Constraints" on page 61, we formulate a model for a feasibility problem rather than for an optimization problem. As for linear models, the parts for data definition, variable definition, stating the objective, the constraints, and the search heuristic are discussed.

The model data is defined as shown in Section 5.2.1 since the non-linear OPL model is TRX-based, too. The variable definition is given in Figure 5.19.

```
26: // variables
27: var int+ y[TRX,Channel] in 0..1,
```

Figure 5.19: Non-linear model: variable definition

Row (27) introduces a binary variable `y` for each TRX and for each channel in the spectrum by means of a 2-dimensional array (sets of variables are not supported in OPL). The objective is stated with the keyword `solve`, whereas

the definition of the constraints is presented in Figure 5.20. Rows (32) and (33) impose that each TRX is assigned one of its available channels, while separation requirements are ensured by the constraints in Rows (36) to (40). Given an edge $e = (u, v)$ and a channel $c1$ assigned to u, it is enforced that no channel $c2$ with $|c1 - c2| < d(u, v)$ is assigned to v.

```
31:    // assign exactly one channel
32:    forall( v in TRX )
33:       sum( c in Available_channels[v] ) y[v,c] = 1;
34:
35:    // separation requirements
36:    forall( e in Edges )
37:       forall( c1 in Available_channels[e.u],
38:               c2 in Available_channels[e.v]:
39:               c1 - e.Sep + 1 <= c2 <= c1 + e.Sep - 1 )
40:          y[e.u,c1] * y[e.v,c2] = 0;
```

Figure 5.20: Non-linear model: stating the constraints

As a search heuristic we employ *Smallest Domain Size*, because this is the most successful one when applied in the previously discussed feasibility models. The OPL excerpt of the adapted heuristic is presented in Figure 5.21.

```
42: // search heuristic
43: search {
44:    forall ( v in TRX ordered by increasing
45:             sum( c in Available_channels[v] ) dsize(y[v,c]) ) {
46:       tryall( c in Available_channels[v] ordered by increasing
47:               sum( w in TRX: bound( y[w,c] ) ) y[w,c] ) {
48:          y[v,c] = 1;
49:          forall( c2 in Channel: c <> c2 ) y[v,c2] = 0;
50:       };
51:    };
52: };
```

Figure 5.21: Non-linear model: search heuristic *Smallest Domain Size*

In Rows (44) and (45), the TRX with the least number of possible values left is chosen. To calculate the number of channels currently available for a TRX, the sum is taken over all available channels of this TRX of the domain sizes of the corresponding binary variables. For a chosen TRX, Rows (46) and (47) select the channel least often assigned to any other TRX so far. For this, the OPL function `bound` is used, which tests whether a variable has already received a channel. Row (48) assigns the selected channel to the chosen TRX, whereas Row (49) ensures that each TRX is assigned only one channel.

In this chapter, we discussed TRX-based and cell-based feasibility models, our approaches for minimizing total interference as well as a non-linear model for FAP. Our tests show that many scenarios are not solvable by means of the

non-linear model, but whenever an assignment is obtained for a given scenario, it is exactly the same as the assignment computed with a linear TRX-based feasibility model (using the search heuristic *Smallest Domain Size*). Hence, we do not further investigate the non-linear approach. In Chapter 6, we present the computational studies made with TRX-based and cell-based feasibility models, compare several feasibility models with each other, and report on the results of the approaches for minimizing total interference.

Chapter 6

Computational Results

In the previous chapter, we introduced different TRX-based and cell-based feasibility models and our investigations on minimizing total interference. In the following, we report on the computational studies of all these approaches.

The tests are performed on a dual Pentium 4 Xeon machine, operating with 2.4 GHz clock speed, equipped with 2 GB system memory, and running a Linux operating system.

Section 6.1 introduces our test instances, while Section 6.2 gives a comparison of cell-based and TRX-based feasibility models. It is shown that TRX-based models often find solutions more quickly than cell-based models, but that TRX-based models fail to solve some scenarios which can be solved by means of cell-based models.

In Section 6.3, a computational comparison of different cell-based feasibility models is presented. TRX-based models are not considered here since cell-based models offer more possibilities to analyze modeling alternatives than TRX-based models, because of the additional constraints for cell-based models that cannot be formulated for TRX-based models without providing additional information in the input data (cf. Section 5.1). Furthermore, the results of the investigations are the same for TRX-based as well as for cell-based feasibility models.

In Section 6.4, the approaches on minimizing total interference are discussed. Our methods using OPL are studied and the obtained results are compared with the solutions computed with a construction heuristic. Furthermore, an improvement heuristic is applied to the solutions computed by means of OPL in order to reduce total interference.

No results on the non-linear approach are reported since on the one hand, most of the instances cannot be solved in reasonable time, and on the other hand, whenever an assignment is found, it is the same as the assignment computed with a linear TRX-based feasibility model.

6.1 Test Instances

Now, we are going to present our test instances. These scenarios have been provided by the E-Plus Mobilfunk GmbH & Co. KG, by the Siemens AG, and by the Swisscom Ltd. Their characteristics are summarized in Table 6.1, a more detailed description and investigation of them can be found in [7]. All these scenarios and corresponding reference solutions can be obtained from FAP web [8].

K: A GSM 1800 network of a dense urban environment with 92 sites, 264 cells, and 267 TRXs. There are 151 interference relations per TRX on the average, e.g., the carrier network contains more than the half of the edges of the complete graph. The spectrum is a contiguous interval of 50 channels.

We also use a modified version of "K" referred to as "K(0.02)", where *tightening the separation* has been performed with a threshold of 0.02. For "K", this value proved to be a good choice to reduce total interference when employing cell-based models.

siemens1: A GSM 900 network with 179 sites, 506 cells, and 930 TRXs. The spectrum is given as two blocks of 20 and 23 channels.

siemens2: A GSM 900 network with 86 sites, 254 cells, and 977 TRXs. The spectrum consists of two contiguous intervals of 4 and 72 channels.

siemens3: A GSM 900 network with 366 sites, 894 cells, and 1623 TRXs. The spectrum is given as 55 contiguous channels.

siemens4: A GSM 900 network with 276 sites, 760 cells, and 2785 TRXs. The available spectrum consists of 39 contiguous frequencies.

Swisscom: A GSM 900 network within a city with 87 sites, 148 cells, and 310 TRXs. The spectrum consists of two contiguous blocks of 3 and 49 channels. There are many local channel blockings: 136 cells are restricted, only 15 channels are available in the worst case, the median value of available channels per cell is 29.

Even though this is a very small network, it is difficult to determine any feasible solution at all because of its many channel blockings. Some example solutions for this scenario can be found at FAP web [8]. However, Eisenblätter et al. report in [10] that they did not succeed in solving the FAP as an Integer Linear Program. In addition, Eisenblätter mentions in [7, Chapter 5] that their heuristics for this scenario failed since they do not work if it is hard to obtain any feasible solutions. Hence, we are especially interested in solving this test instance by means of Constraint Programming.

scenario	# sites	#cells	#TRXs	avg. TRXs/cell	max. TRXs/cell	#edges	$\#\{e \in E : d(e) \geq 1\}$	spectrum size [1]
K	92	264	267	1.01	2	19707	850	50
siemens1	179	506	930	1.84	4	14303	574	20 + 23
siemens2	86	254	977	3.85	6	23216	250	4 + 72
siemens3	366	894	1623	1.82	3	50717	833	55
siemens4	276	760	2785	3.66	5	113336	839	39
Swisscom	87	148	310	2.09	4	846	846	3 + 49
bradford-0	649	1886	1886	1.00	1	241767	5770	75
bradford-1	645	1878	2947	1.57	12	241761	5764	75
bradford-2	644	1876	3406	1.82	12	241760	5763	75
bradford-4	649	1886	3996	2.12	12	241767	5770	75
bradford-10	644	1876	4871	2.60	12	241760	5763	75
bradford_nt-1	649	1886	1971	1.05	3	241767	5770	75
bradford_nt-2	649	1886	2214	1.17	5	241767	5770	75
bradford_nt-4	649	1886	2775	1.47	9	241767	5770	75
bradford_nt-10	649	1886	4145	2.20	12	241767	5770	75

Table 6.1: Overview on scenario characteristics

[1] Non-contiguous spectra are reflected by a '+'.

bradford_nt-*d*-*p*: GSM 1800 networks with (in most cases) 649 sites and 1886 cells. Actually, the same cell graph is underlying all these instances, but three cell graphs differ in their number of nodes and edges. The available spectrum consists of 75 contiguous channels.

On the one hand, these scenarios differ by their number of TRXs, on the other hand, three different interference predictions are available. A basic traffic load has been scaled by the factor d, which can take on values of 0, 1, 2, 4, and 10. The resulting number of TRXs for the scenarios are 1886, 1971, 2214, 2775, and 4145. Interference prediction (p) is done according to a model developed by E-Plus (eplus), using free space propagation (free), and according to a Modified Okumura-Hata model (race). For details, see [7, Section 2.3.2]. There are also variants called bradford-*d*-*p* with a higher number of TRXs compared to their corresponding bradford_nt instances. The number of TRXs for the bradford instances are 1886, 2947, 3406, 3996, and 4871.

In order to increase performance, we compute maximum cliques (and, in addition, a set of further large cliques) for the given graphs and add these information to the input data of the models. For details on the algorithm used, see [6, 7]. We are interested in maximum cliques for the cell graphs as well as for the underlying carrier networks of the scenarios. However, we need cliques for the subgraphs consisting only of those edges $e \in E$ with a minimum required separation $d(e) \geq 1$ rather than cliques for the given graphs themselves. Let G_{cell} and $G_{carrier}$ be the given cell and carrier networks and let H_{cell} and $H_{carrier}$ be their subgraphs containing only those edges e with $d(e) \geq 1$. Table 6.2 lists the sizes of the maximum cliques for all these graphs.

Scenario	G_{cell}	$G_{carrier}$	H_{cell}	$H_{carrier}$
K	68	69	7	7
siemens1	38	75	5	10
siemens2	75	295	6	24
siemens3	51	100	13	23
siemens4	107	443	4	11
Swisscom	9	21	9	21

Table 6.2: Maximum clique sizes within carrier and cell networks

6.2 Overview on Cell-based and TRX-based Models

This section contains a general overview on our experiences made when solving the Frequency Assignment Problem by means of feasibility models and also presents the results for solving all scenarios with one cell-based and one TRX-based feasibility model. These results as well as the values of known reference solutions of the test instances are shown in Table 6.3 on page 80. For both

models, the search heuristic *Smallest Domain Size* is used and no additional constraints are formulated.

The following observations can be made with all our cell-based and TRX-based feasibility models:

- Cell-based models enable us to solve more instances than TRX-based models because of the additional constraints for cell-based models that exploit information on the cell structure.
- Whenever TRX-based models determine a solution for a scenario, the time consumed is less than the time needed by cell-based models.
- TRX-based models require slightly more memory than cell-based models. One reason could be that the carrier graph in TRX-based models is typically larger than the cell graph in cell-based models. TRX-based models consume at most 1.5 times the memory needed by cell-based models, i.e., even for instances with more than 4000 TRXs, only about 250 MB RAM is needed by TRX-based models. Hence, this additional memory consumption does not cause any problem.

  However, as reported in Section 6.3.3, minor changes in the formulation of the constraints may increase memory consumption dramatically, such that most of the instances can hardly be solved at all by the modified model on a regular work station.
- The problem of all feasibility models is the quality of the obtained solutions. As shown in Table 6.3, the costs of the assignment computed for "K" is 85 times the value of the known reference solution if no *tightening the separation* is performed, and 4.4 times the reference value if *tightening the separation* with a threshold of 0.02 is applied. The results for the bradford instances are even less satisfactory. The costs of our assignments exceed the costs of the reference solutions by a factor of 7.5 to 500 ("bradford_nt-10"). For a given scenario, any assignment is computed by means of feasibility models without trying to prevent occurring interferences, which results in this extensive costs. The solution values obtained with cell-based and TRX-based feasibility models differ by at most 10 percent.
- An interesting observation can be made when investigating the solutions of the scenarios determined by means of feasibility models. Given a scenario and applying the search heuristic *Smallest Domain Size*, most of the cell-based models compute exactly the same assignment. The only difference between the models is the performance: solution times differ by a factor of more than 484 ("K"), and memory consumption differs by a factor of 40 ("siemens2"). This holds true for each scenario except for "Swisscom", where different assignments are obtained with different models, and except for the cell-based model with the additional constraint *Implication Costs*, which determines other assignments for two given instances ("siemens1" and "siemens2") than the remaining models.

	cell-based model no additional constr.			TRX-based no additional constr.			
Scenario	RAM [MB]	time [s]	costs	RAM [MB]	time [s]	costs	ref. value
K	18	0.16	41.9	19	0.08	38.3	0.45
K(0.02)	32	0.79	2.0	28	>3600	-	0.45
siemens1	35	1.40	52.0	38	1.26	55.1	2.30
siemens2	49	2.05	56.8	70	1.12	59.9	14.28
siemens3	60	>3600	-	63	>3600	-	5.19
siemens4	78	13.85	297.9	107	6.17	297.2	80.97
Swisscom	11	27565	-	13	20400	-	27.21
bradford-0-eplus	90	4.46	187.5	81	3.84	187.5	0.80
bf-0-free	92	4.40	118.0	83	3.97	118.0	0.00
bf-0-race	87	4.51	82.9	82	3.94	82.9	0.00
bf-1-eplus	120	11.04	484.7	141	5.85	484.2	33.99
bf-1-free	124	10.89	366.9	146	5.29	348.0	0.16
bf-1-race	120	10.66	233.0	141	5.43	231.4	0.03
bf-2-eplus	136	14.85	667.3	167	6.96	688.8	80.03
bf-2-free	140	14.78	488.3	173	7.05	488.9	2.95
bf-2-race	136	14.46	315.5	167	7.17	312.3	0.42
bf-4-eplus	156	20.10	974.4	198	>3600	-	167.70
bf-4-free	165	19.96	770.7	144	>3600	-	22.09
bf-4-race	156	19.39	478.9	201	9.71	436.1	3.04
bf-10-eplus	116	>3600	-	159	>3600	-	400.00
bf-10-free	130	>3600	-	209	>3600	-	117.80
bf-10-race	99	>3600	-	236	>3600	-	30.22
bf_nt-1-eplus	89	4.92	201.0	86	2.29	195.2	0.86
bf_nt-1-free	95	4.84	143.5	87	2.39	135.5	0.00
bf_nt-1-race	123	5.01	96.9	86	2.28	92.3	0.00
bf_nt-2-eplus	129	6.16	267.6	167	7.01	261.8	3.17
bf_nt-2-free	99	6.12	204.1	157	5.25	200.4	0.00
bf_nt-2-race	100	5.97	133.2	165	6.83	144.9	0.00
bf_nt-4-eplus	116	9.75	446.7	125	4.36	430.1	17.73
bf_nt-4-free	117	9.88	326.2	131	4.49	311.5	0.00
bf_nt-4-race	115	9.87	219.9	125	4.46	226.5	0.00
bf_nt-10-eplus	157	20.34	1119.0	199	9.33	1109.7	146.20
bf_nt-10-free	163	20.05	870.2	194	9.76	854.9	5.86
bf_nt-10-race	163	19.76	549.9	199	9.48	555.6	1.07

Table 6.3: Solutions of a cell-based and a TRX-based feasibility model

- The TRX-based case parallels the cell-based one. All TRX-based feasibility models using the search heuristic *Smallest Domain Size* compute the same assignment for a given scenario, but the performance between the TRX-based models varies less drastically than in the cell-based case. Even the non-linear model (which is TRX-based) produces the same assignment (if any) for a given scenario as all the linear TRX-based models.

Considering Table 6.3, every scenario can either be solved within a few seconds, or no solution can be found for this instance within more than one hour. The bradford instances, which have between 2000 and 4000 TRXs, can be solved by means of TRX-based feasibility models in at most 10 seconds, and by means of cell-based models in at most 21 seconds (if any solution is found), while instances with about 1000 TRXs are solved in about 1 to 2 seconds ("siemens1" and "siemens2"). Furthermore, comparing the results for the scenarios "K" and "K(0.02)" computed with the cell-based model, performing *tightening the separation* with a threshold of 0.02 reduces total interference by a factor of 20, whereas the required time remains nearly the same. Moreover, the scenario "K(0.02)" cannot be solved by the TRX-based model, even though a solution is found for this instance by the cell-based model. A *tightening the separation* threshold higher than 0.02 has to be chosen in order to solve the instance "K" by means of a TRX-based feasibility model.

As we have seen, TRX-based models only need about half the time than cell-based models to compute a feasible assignment for a given instance. However, cell-based models enable us to solve some instances that cannot be solve by means of TRX-based models in reasonable time. The differences regarding memory consumption and quality of the solution between both kinds of models are negligible, whereas the quality of the obtained solutions is often not convincing.

6.3 Comparing different Cell-based Models

Computational studies on different cell-based feasibility models are reported in this section. The influence of formulating redundant constraints, of breaking symmetries, of stating cost constraints, and the impact of applying an alternative formulation for the separation constraints are investigated, and different search heuristics are compared.

Among others, the scenario "bradford_nt-10-eplus", which has more than 4000 TRXs and is one of the largest instances we are able to solve, is used for the following tests. The search heuristic *Smallest Domain Size* is employed for all models unless stated otherwise. The results of the following tests are compared with the ones for the model without any further constraint, which are given in Table 6.3.

6.3.1 Redundant Constraints

In this section, the impact of stating redundant constraints and of breaking symmetries is investigated for cell-based feasibility models. It is shown that it is useful to apply these constraints since this may allow solving some instances that are not solvable otherwise. In case no benefit is gained, the effort additionally spent on maintaining the optional constraints is negligible.

In the following, the constraints *Sum Channels*, which impose a condition on the minimum value of the sum of all channels assigned to one cell; *Break Symmetries*, which break the symmetries between the TCHs of one cell; and *Cliques*, which enforce that all TRXs of each given clique are assigned different channels, are investigated. The impact of separately stating these three constraints and the effect of combining the constraints *Break Symmetries* and *Cliques* are observed. Since formulating the constraint *Sum Channels* does not improve performance, it is not combined with any other constraint. The results of these tests are shown in Table 6.4. As a reference for comparison, the results for the model without any redundant constraint are listed as well, cf. Table 6.3.

Comparing the results from Table 6.4 for the models with and without the constraint *Sum Channels*, the solution times are slightly higher if this constraint is formulated.

Stating the constraint *Break Symmetries* enables us to solve the scenarios "siemens3" and "Swisscom", which cannot be solved without this constraint. It is remarkable that "siemens3" can be solved in about six and a half seconds if *Break Symmetries* is formulated, while no solution can be found within one hour if this constraint is omitted. The results for most of the other instances have not changed significantly. This is particularly surprising for "bradford_nt-10-eplus" since this scenario has up to 12 TRXs per cell, and we expected that breaking symmetries reduces the solution time in this case.

The formulation of the redundant constraint *Cliques* allows us to solve "Swisscom" in about three and a half hours. However, the impact of this constraint on the results of the other instances is minor.

When combining the constraints *Break Symmetries* and *Cliques*, it is possible to solve "Swisscom" in less than one hour. This is a considerable success since it is difficult to obtain any feasible solution for this scenario, cf. the notes on page 76.

In summary, stating the additional constraints enables to solve the instances "siemens3" and "Swisscom", while the solution times for the other instances remain nearly the same. Hence, it is useful to apply these optional constraints for the Frequency Assignment Problem.

	with *Sum Channels*		with *Break Symmetries*		with *Cliques*		with *Cliques* and *Break Symmetries*		no additional constraints	
Scenario	time [s]	costs	time [s]	costs	time [s]	costs	time [s]	costs	time [s]	costs
K	0.21	41.9	0.16	41.9	0.15	41.9	0.17	41.9	0.16	41.9
K(0.02)	0.86	2.0	0.79	2.0	0.81	2.0	0.80	2.0	0.79	2.0
siemens1	1.73	52.0	1.36	52.0	1.44	52.0	1.42	52.0	1.40	52.0
siemens2	2.16	56.8	2.37	56.8	2.12	56.8	2.54	56.8	2.05	56.8
siemens3	>3600	-	6.66	55.6	>3600	-	5.63	55.6	>3600	-
siemens4	14.67	297.9	10.37	297.9	13.23	297.9	10.48	297.9	13.85	297.9
Swisscom	23600	-	23694	55.2	12984	54.86	3322	56.6	27565	-
bradford_nt-10-eplus	21.61	1119.0	20.45	1119.0	20.15	1119.0	21.31	1119.0	20.34	1119.0

Table 6.4: Investigating redundant constraints and breaking symmetries

6.3.2 Formulating Cost Constraint

The costs of the assignments are determined in post processing by means of a Perl script. It is also possible to state some constraints in OPL, which allow to calculate the occurring costs. In Section 5.2.6, two different ways for formulating these cost constraints are described. The first approach is by means of Higher-order Constraints and is referred to as *Higher-order Costs*, while the second method uses implications and is called *Implication Costs*.

In the following, the impact of the cost constraints on the performance is investigated and it is shown that both kinds of cost constraints should not be applied. In either case, no redundant constraints are formulated. The obtained results are summarized in Table 6.5 and are compared with the results for the model without any additional constraint from Table 6.3.

	with *Higher-order Costs*			with *Implication Costs*		
Scenario	RAM [MB]	time [s]	costs	RAM [MB]	time [s]	costs
K	67	1.90	41.9	130	77.70	41.9
K(0.02)	60	2.21	2.0	160	64.60	2.0
siemens1	102	5.65	52.0	106	21.53	45.1
siemens2	337	27.20	56.8	287	50.35	44.1
siemens3	252	>3600	-	164	>3600	-
siemens4	547	104.83	297.9	319	>3600	-
Swisscom	14	34014	-	13	29520	-
bradford_nt-10-eplus	1236	1461	1119.0	>2300	>3600	-

Table 6.5: Formulating cost constraints

Considering both tables and comparing the models with and without the constraint *Higher-order Cost*, the formulation of these constraints leads to an increase in memory consumption by a factor of up to 7 and to an increase in the solution times by a factor of up to 10. The computed assignments remain the same. The additional variables implicitly introduced by the Higher-order constraints may cause this deterioration of the performance since two variables are introduced for all those edges, where at least one interference value is greater than zero.

The results for the models with and without the constraint *Implication Costs* can be found in Table 6.5 and 6.3, too. Even though only one variable is needed for each of the previously mentioned edges, the performance deteriorates substantially. As an example, consider the solution time for "K", which increases by a factor of 485 compared to the model without any cost constraints. In addition, the scenarios "siemens3", "siemens4", "Swisscom", and "bradford_nt-10-eplus" cannot be solved within one hour and the amount of consumed memory

increases by a factor between 4 and 15 if the constraint *Implication Costs* is formulated. As mentioned in Section 6.2, this model is the only one that determines solutions for "siemens1" and "siemens2" different than all other models. The costs of these assignments are 10 and 20 percent less, respectively, than the costs of the assignments computed otherwise by the cell-based models, but to obtain these new assignments, the twentyfold of time is needed. However, it remains unclear why this approach has such enormous drawbacks compared to *Implication Costs*.

Summing up, both kinds of cost constraints should not be used for feasibility models since performance decreases significantly otherwise.

6.3.3 Using `predicates`

In Section 5.2.4, we presented an alternative way to state the separation constraints by means of `predicates`. The goal is to enhance finding solutions since arc consistency is enforced on all constraints formulated by means of `predicates`. However, our computational studies show that this alternative formulation should not be applied. The results of the tests are given in Table 6.6, no optional constraints are stated.

	employing `predicates`		
Scenario	RAM [MB]	time [s]	costs
K	119	1.27	41.9
K(0.02)	727	11.11	2.0
siemens1	330	7.82	52.0
siemens2	1900	28.16	56.8
siemens3	1752	>3600	-
siemens4	1156	54.74	297.9
Swisscom	64	14764	-
bradford_nt-10-eplus	>2500	-	-

Table 6.6: Formulating separation constraints with `predicates`

The major problem when employing `predicates` for FAP is the memory consumption. For "siemens2", the amount of needed memory is 1900 MB, while "bradford_nt-10-eplus" cannot be solved at all since more than 2500 MB memory is needed. Even though special propagation techniques may be applied when stating `predicates`, the resulting assignments are identical to the ones obtained by the other models. Hence, we do not employ `predicates` for stating the separation constraints.

6.3.4 Alternative Search Heuristics

All previous tests in this chapter were performed with models employing the search heuristic *Smallest Domain Size*. Since the quality of the obtained solutions is often not convincing, the alternative heuristics *T-Coloring* and *DSATUR with Costs* are investigated and compared to the heuristic *Smallest Domain Size* in this section. No additional constraints are formulated for all these models.

	with the heuristic *T-Coloring*		with the heuristic *Smallest Domain Size*		
Scenario	time [s]	costs	time [s]	costs	ref. value
K	13.57	336.3	0.16	41.9	0.45
K(0.02)	14.97	2.5	0.79	2.0	0.45
siemens1	127.63	290.0	1.40	52.0	2.30
siemens2	751.31	156.3	2.05	56.8	14.28
siemens3	>3600	-	>3600	-	5.19
siemens4	>3600	-	13.85	297.9	80.97
Swisscom	323286	-	27565	-	27.21
bradford_nt-10-eplus	>3600	-	20.34	1119.0	146.20

Table 6.7: Comparing the heuristics *T-Coloring* and *Smallest Domain Size*

The results for the heuristics *T-Coloring* and *Smallest Domain Size* as well as the values of the known reference solutions for the scenarios are given in Table 6.7. It can be seen that the costs of the assignment determined for "K" when applying *T-Coloring* is 8 times the costs when using the heuristic *Smallest Domain Size* and almost 750 times the costs of the known reference solution for this scenario. The second disadvantage of *T-Coloring* implemented in OPL is the solution time. As can also be seen in Table 6.7, solving "siemens2" takes 750 seconds with the heuristic *T-Coloring* compared to about 2 seconds needed by the search heuristic *Smallest Domain Size*. No solution can be found within one hour for the instances "siemens3", "siemens4", and "bradford_nt-10-eplus". The reasons for the failure of this heuristic in our case are not clear.

The results for the search heuristic *DSATUR with Costs* are even more drastic. Not even the instance "K" can be solved within one hour if this heuristic is applied, hence no results on the tests are presented in any table. More than 1300 MB of memory is consumed by this heuristic, which is an increase by a factor of 70 compared to the heuristic *Smallest Domain Size*. The reason for the failure of this heuristic may be the large number of additional variables which are introduced by the Higher-order constraints in order to maintain the matrix `cost_of_channel`, cf. page 59.

Hence, we apply the search heuristic *Smallest Domain Size*. It outperforms the OPL implementations of *T-Coloring* and *DSATUR with Costs* with respect to solution times and the quality of the obtained assignments.

6.4 Minimizing Total Interference

We try to explicitly minimize total interference since the quality of the solutions determined by means of feasibility models is often not convincing with respect to known reference solutions.

Three different approaches to minimize total interference are introduced in Section 5.4: an OPL script combined with a cell-based feasibility model, an OPL script combined with a TRX-based feasibility model, and a Perl script in conjunction with a cell-based feasibility model. In Section 6.4.1, these methods are investigated and it is shown that altogether a minor advantage can be seen for the OPL script combined with the cell-based model.

We implement the heuristic *DSATUR with Costs* in the programming language C++ to compare the results obtained with ILOG OPL Studio with solutions determined by an own computer program. Furthermore, in order to reduce total interference, we apply the improvement heuristic *Variable Depth Search* to the solutions determined by means of OPL. The results of both approaches are discussed in Section 6.4.2 and Section 6.4.3, respectively.

6.4.1 Employing OPL

In the following, we investigate our minimization methods which employ OPL. The computational results are presented in Table 6.8 and are compared with the solutions determined by feasibility models and with the values of known reference solutions for the scenarios, see Table 6.3.

	OPL script with TRX-based model			OPL script with cell-based model			Perl script with cell-based model		
Scenario	RAM [MB]	time [s]	costs	RAM [MB]	time [s]	costs	RAM [MB]	time [s]	costs
K	56	21.56	2.3	60	61.71	1.7	54	84.52	1.7
siemens1	106	65.82	8.8	80	65.00	12.6	86	81.03	12.6
siemens2	515	157.61	40.4	297	223.13	29.3	119	204.02	29.3
siemens3	-	-	-	204	93.53	55.6	237	99.41	55.6
siemens4	885	1472.36	252.8	500	1153.90	206.9	286	1271.11	206.9

Table 6.8: Minimizing total interference by means of OPL

First, we report some general observations on the minimization approaches with OPL. Thereafter, the three applied methods are compared with each other.

Considering Table 6.8 and Table 6.3 and comparing the solutions determined by the minimization approach with the solutions computed with feasibility models, the achieved improvement varies widely among the scenarios. The costs for “K” decrease by a factor of more than 24 (from 41.9 to 1.7), whereas the costs for “siemens3” cannot be reduced by the optimization procedure. The solution

times are considerable for the minimization approach; about 20 minutes are needed to solve the scenario "siemens4". The values of the improved solutions are still between 2 and 10 times the values of known reference solutions. In order to investigate whether it is possible to obtain better results by increasing the search time, we set the time limit for the instance "K" to more than 5 hours (instead of to a few seconds), but the obtained assignments are the same in either case.

Thus, applying the minimization approaches reduces total interference in most cases, but the quality of the obtained solutions is still not much satisfactory.

In the remaining of this section, the three possibilities for minimizing total interference by means of OPL are compared with each other. Considering the results of the OPL script combined with either the cell-based or the TRX-based feasibility model, it can be seen that the cell-based procedure produces in 4 out of 5 cases better solutions than the TRX-based method; recall that no solution at all is found for "siemens3" by means of TRX-based (feasibility) models within more than one hour. In contrast to feasibility models, the quality of the solution obtained with cell-based and with TRX-based models differ by up to 50 percent ("siemens2") when minimizing total interference, because of different final *tightening the separation* thresholds for both kinds of models. Comparing the memory consumption for both methods, the TRX-based approach consumes up to 50 percent more memory than the cell-based method on large instances: for "siemens4" 885 MB is needed instead of 500 MB. This is caused by the variables implicitly introduced by the cost constraints (recall that two variables are added for each edge and that the carrier graph is most often larger than the cell graph). Hence, cell-based models should be preferred to TRX-based models when trying to minimize total interference.

We also employ a Perl script instead of the OPL script in order to prevent calculating the costs by means of OPL, because the latter method consumes much memory. If the same OPL model is used in conjunction with either the OPL script or the Perl script, the obtained assignments are identical since the same algorithm is implemented for the binary search on the final *tightening the separation* threshold in both scripts. As the cell-based model determines better solutions than the TRX-based model in most cases, the Perl script is only used with the cell-based model.

Comparing the amount of memory consumed by the Perl approach and by the OPL script combined with the cell-based model, the Perl variant needs about half the memory than the OPL method on large instances. However, solutions can often be obtained more quickly with the OPL script than with the Perl script, because on the one hand, the whole ILOG OPL Studio has to be started each time the Perl script tries to determine a solution by means of the feasibility model, and on the other hand, the communication between ILOG OPL Studio and our Perl script is realized by files, which slows down the computation. A considerable amount of time is required for starting ILOG OPL Studio when

employing the Perl script. Tests for the instances "K" and "siemens1" show that altogether about 24 and 16 seconds (this are 28 and 20 percent of the total solution times), respectively, are needed for the starts of ILOG OPL Studio.

Considering the discussed methods, the OPL script combined with the cell-based model is preferable if high memory consumption does not cause any problem. The Perl script is useful to be applied in conjunction with the cell-based model if available memory is a bottleneck, while TRX-based models should not be used when minimizing total interference.

However, the approaches using OPL are not much convenient since the quality of the assignments improved this way is still not convincing.

6.4.2 Using a Construction Heuristic

The results of our implementation of the construction heuristic *DSATUR with Costs* are shown in Table 6.9 and are compared with the solutions determined by means of OPL.

If any feasible solution is found, better assignments are obtained in less time by *DSATUR with Costs* compared to our OPL methods. For the scenario "K", we are able to determine an assignment with cost 1.0 within 15 seconds using our C++ program, while only an assignment with cost 1.7 is obtained within 62 seconds by means of OPL. However, the instances "siemens3" and "siemens4" cannot be solved at all by our implementation of *DSATUR with Costs*.

	construction heuristic *DSATUR with Costs*			improvement heuristic *Variable Depth Search*			
Scenario	RAM [MB]	time [s]	costs	start value	final value	time [s]	ref. value
K	20	15.3	1.0	1.7	1.1	4.92	0.45
siemens1	55	77.0	4.5	12.6	4.8	8.03	2.30
siemens2	95	331.3	26.9	29.3	20.4	28.19	14.28
siemens3	50	22.6	-	55.6	11.6	32.80	5.19
siemens4	365	292.0	-	206.9	134.7	51.33	80.97
Swisscom	-	-	-	56.6	36.9	0.60	27.21

Table 6.9: Minimizing interference with C++ programs

6.4.3 Combining OPL and an Improvement Heuristic

The optimization procedures with OPL enable us to solve all considered scenarios, whereas the determined solutions are not good. In contrast, our C++ program produces much better solutions than the OPL methods for some instances, but fails to solve some other scenarios.

In the following, we try to combine OPL models with a C++ program by applying the improvement heuristic *Variable Depth Search* implemented by Eisenblätter [7] to the solutions obtained by means of OPL. The results are presented in Table 6.9, the values of the start and the final solution, the consumed time, and the values of known reference solutions are given. We also try to improve a solution for "Swisscom" in order to investigate whether it is possible to obtain better assignments for an instance where it is difficult to calculate any feasible solution.

As can be seen from Table 6.9, applying *Variable Depth Search* to the assignments determined by means of OPL reduces total interference by a factor of up to 5 ("siemens3") within a few seconds. The costs of the improved solutions are between 1.5 and 2 times the values of the reference solutions. Even the assignment for "Swisscom" is improved this way.

Hence, determining a solution with OPL and improving this assignment by means of an improvement heuristic can be used to obtain solutions with tolerable quality, also for scenarios where feasibility is a limiting factor.

6.5 Summary of the Computational Results

The observations presented in this chapter can be summarized as follows:

Cell-based feasibility models combined with some optional constraints enable us to solve all but three of the given scenarios. In particular, we are able to determine a feasible assignment for "Swisscom", see page 76. Moreover, we solve instances with up to 1000 TRXs within 1 to 2 seconds and instances with up to 4000 TRXs within about 20 seconds with cell-based models.

Using TRX-based feasibility models, it is not possible to solve all instances which can be solved with cell-based feasibility models. But if an assignment is found, the required time is only about half the time as needed by cell-based models.

The differences between the quality of the assignments obtained with cell-based and TRX-based feasibility models are negligible. But for both kinds of models, the quality of the solutions is not sufficient. Comparing the costs of our assignments with the values of known references solutions, our costs exceed the reference costs by a factor between 2 ("Swisscom') and 500 ("bradford_nt-10-race").

Comparing different cell-based feasibility models for FAP, we observe that it is useful to state redundant constraints and constraints breaking symmetries, while cost constraints and `predicates` should be avoided. In addition, it is very important to employ only a few variables since all of our approaches needing many variables failed.

As we succeeded in solving most of the instances, we also try to obtain good assignments. To minimize total interference by means of OPL, we apply the preprocessing technique *tightening the separation* and employ either an OPL script or a Perl script in conjunction with a feasibility model. The benefit of this technique heavily depends on the given scenario, but the quality of the solutions is still not convincing, whereas the required time is considerable. Thus, the OPL approach is not very suitable for minimizing total interference.

Comparing the results obtained by ILOG OPL Studio with the solutions computed with a construction heuristic, the latter one produces better assignments in less time than the OPL approach in three cases, but fails to determine any solution for two out of five investigated instances. Thereby, only those scenarios are solved, where it is not difficult to calculate any feasible assignment.

A successful approach is to combine OPL models and a C++ program by applying the improvement heuristic *Variable Depth Search* to the solutions computed by means of OPL. This significantly reduces total interference, such that the costs of the finally determined assignments are only about 1.5 to 2 times the values of known reference solutions. Compared to the best known solution values, this is not very good, but even such instances where feasibility is problematic can be solved this way.

Chapter 7

Summary and Conclusions

In this thesis, the solving of a version of the Frequency Assignment Problem (FAP) in GSM networks by means of Constraint Programming has been considered. Having introduced our mathematical model, we showed that our version of FAP is strongly $\mathcal{NP}$-hard. Furthermore, the theory of Constraint Programming has been studied. It turned out that Constraint Satisfaction Problems are strongly $\mathcal{NP}$-complete, where the discussed Frequency Assignment Problem can be formulated as a Constraint Satisfaction Problem.

The "Optimization Programming Language" (OPL) has been employed to state and solve various models for FAP. Since the modeling language of OPL is more expressive than Mixed Integer Programming, we analyzed whether the additional features of OPL really provide more possibilities to state constraints than in Mixed Integer Programming. We observed that many of these additional elements can be expressed in Mixed Integer Programs (MIPs) as well, but that the transformation from an OPL model into a MIP is not always possible. Moreover, additional variables and constraints often have to be introduced in order to translate the models. Thus, these investigations are more of theoretical than of practical interest.

On the one hand, we considered feasibility problems only. On the other hand, we tried to explicitly minimize total interference. Two kinds of models have been proposed for feasibility problems: cell-based and TRX-based models. Using a cell-based model, we were particularly able to determine a solution for an instance where it is very difficult to obtain any feasible assignment. TRX-based models did not enable us to solve all those instances we could solve with cell-based models, but if a feasible assignment was found, the solution times of TRX-based models were only about half the solution times of cell-based models. It depends on the scenario, which kind of model should be preferred. However, the quality of the assignments determined with feasibility models has not been satisfactory.

To cope with this, we developed a framework to minimize total interference by means of OPL. The gained benefit varied significantly among the scenarios, but the quality of the solutions still was not much convincing. Alternatively, we

combined our OPL models with a C++ program by applying an improvement heuristic to the solutions determined with OPL, which considerably reduced total interference in reasonable time. Hence, this combination can be used to obtain solutions of moderate quality for many scenarios, even if it is hard to compute any feasible assignment.

Altogether, OPL is suitable for determining feasible assignments even for large instances of FAP or for scenarios where it is difficult to obtain any feasible assignment. However, we could not produce solutions of adequate quality only by means of OPL. In order to compute such assignments, we propose the use of an improvement heuristic in addition to OPL feasibility models.

Appendix A

TRX-based Feasibility Model

```
// number of TRXs
int+ number_TRXs                    = ...;
range TRX 0..number_TRXs-1;

// define a record for the set of edges
struct Edge_type {
   TRX u;
   TRX v;
   int+ Sep;
};

// the given set of edges
{Edge_type} Edges                   = ...;

// the given spectrum
int+ first_channel                  = ...;
int+ last_channel                   = ...;
range Channel [first_channel..last_channel];

// the set of available channels for each cell
{Channel} Available_channels[TRX]   = ...;

// optionally: a set of cliques
{{TRX}} Cliques                     = ...;

// variables
var  Channel chan[TRX];

solve {

   // assign only allowed channels
   forall( t in TRX )
      chan[t] in Available_channels[t];
```

```
   // respect separation constraints
   forall( e in Edges )
      abs( chan[ e.u ] - chan[ e.v ] ) >= e.Sep;

   // optionally: consider cliques
   forall( clique in Cliques )
      alldifferent( all( trx in clique ) chan[ trx ] );
};

// search heuristic
search {
   forall( t in TRX ordered by increasing dsize(chan[t]) )
      tryall ( c in Available_channels[t] ordered by
               increasing nbOccur(c, chan))
         chan[t] = c;
};
```

Appendix B

Cell-based Feasibility Model

```
// number of cells
int+ number_cells                    = ...;
range Cell 0..number_cells-1;

// number of TRXs per cell: BCCH + TCH's
// 1: BCCH, 2..n: TCH's
int+ number_TRXs[Cell]               = ...;

// define a record for the set of edges
struct Edge_type {
   Cell u;
   Cell v;
   int+  Sep;
   // handover relations between cells
   // 0: no HO, 1: HO(u->v), 2: HO(v->u), 3: HO(u<->v)
   int+  Ho;
};

// the given set of edges
{Edge_type} Edges                    = ...;

// define a record for theset of TRXs
struct TRX_type { Cell cell; int+  n; };

// build the set of all TRXs
{TRX_type} TRX = {<c,t> | c in Cell & t in 1..number_TRXs[c] };

// the given spectrum
int+ first_channel                   = ...;
int+ last_channel                    = ...;
range Channel [first_channel..last_channel];

// the set of available channels for each cell
```

```
{Channel} Available_channels[Cell]  = ...;

// separation
int+ Co_cell_separation[Cell]       = ...;

// handover separation
int+ Ho_bcch_to_bcch           = ...;
int+ Ho_bcch_to_tch            = ...;
int+ Ho_tch_to_bcch            = ...;
int+ Ho_tch_to_tch             = ...;
int+ Ho_max_bcch_tch           = maxl( Ho_bcch_to_tch, Ho_tch_to_bcch );

// variables
var  Channel chan[TRX];

solve {

   // assign only allowed channels
   forall (t in TRX)
      chan[t] in Available_channels[t.cell];

   // separation constraints
   forall( e in Edges )
      forall( t1 in 1..number_TRXs[e.u], t2 in 1..number_TRXs[e.v] )
         abs( chan[<e.u,t1>] - chan[<e.v,t2>] ) >= e.Sep;

   // respect co-cell separation
   forall( c in Cell, ordered i,j in 1..number_TRXs[c] )
      abs( chan[<c,i>] - chan[<c,j>] ) >= Co_cell_separation[c];

   // handover separation u->v, v->u, u<->v
   forall( e in Edges: e.Ho >= 1 ) {
      // BCCH <-> BCCH
      abs( chan[<e.u,1>] - chan[<e.v,1>] ) >= Ho_bcch_to_bcch;
      // TCH <-> TCH
      forall( t1 in 2..number_TRXs[e.u], t2 in 2..number_TRXs[e.v] )
        abs( chan[<e.u,t1>] - chan[<e.v,t2>] ) >= Ho_tch_to_tch;
      };

   // handover separation u->v
   forall( e in Edges: e.Ho = 1 ) {
      // BCCH -> TCH
      forall( t2 in 2..number_TRXs[e.v] )
         abs( chan[<e.u,1>] - chan[<e.v,t2>] ) >= Ho_bcch_to_tch;
      // TCH -> BCCH
      forall( t1 in 2..number_TRXs[e.u] )
         abs( chan[<e.u,t1>] - chan[<e.v,1>] ) >= Ho_tch_to_bcch;
```

```
   };

   // handover separation v->u
   forall( e in Edges: e.Ho = 2 ) {
      // BCCH -> TCH
      forall( t2 in 2..number_TRXs[e.v] )
         abs( chan[<e.u,1>] - chan[<e.v,t2>] ) >= Ho_tch_to_bcch;
      // TCH -> BCCH
      forall( t1 in 2..number_TRXs[e.u] )
         abs( chan[<e.u,t1>] - chan[<e.v,1>] ) >= Ho_bcch_to_tch;
   };

   // handover separation u<->v
   forall( e in Edges: e.Ho = 3 ) {
      // BCCH -> TCH
      forall( t2 in 2..number_TRXs[e.v] )
         abs( chan[<e.u,1>] - chan[<e.v,t2>] ) >= Ho_max_bcch_tch;
      // TCH -> BCCH
      forall( t1 in 2..number_TRXs[e.u] )
         abs( chan[<e.u,t1>] - chan[<e.v,1>] ) >= Ho_max_bcch_tch;
   };

   // consider cliques
   forall( clique in Cliques )
      alldifferent( all( cell in clique,
                         t in 1..number_TRXs[cell] ) ass[<cell,t>] );

   // break symmetries within one cell between TCH's
   forall( c in Cells: number_TRXs[c] >= 3 )
      forall( i in 2..number_TRXs[c]-1 )
         ass[<c,i>] < ass[<c,i+1>];

   // redundant constraint on the sum of assigned channels within one cell
   forall( c in Cell: number_TRXs[c] >= 2 )
      ( number_TRXs[c] *
        ( first_channel + (number_TRXs[c]-1) / 2 * Co_cell_separation[c] )
      ) <= sum( i in 1..number_TRXs[c] ) chan[<c,i>];
};

search {
   forall( t in TRX ordered by increasing
         <dsize(chan[t]), -number_TRXs[t.cell]> )
      tryall ( c in Available_channels[t.cell] ordered by
               increasing nbOccur(c, chan))
         chan[t] = c;
};
```

Bibliography

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela and M. Protasi: *Complexity and Approximation: combinatorial optimization problems and their approximability properties*, Springer Verlag, 1999

[2] F. Bacchus, P. van Beek: *On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems*, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98), 26–30, AAAI Press, Menlo Park, ISBN: 0-262-51098-7, pages 311–318, 1998

[3] R. Barták: *Online Guide to Constraint Programming*, Prague, 1998, `http://kti.mff.cuni.cz/~bartak/constraints`

[4] R. Barták: *Constraint Programming: In Pursuit of the Holy Grail*, in: Proceedings of WDS99 (invited talk), Prague, 1999

[5] R. Barták: *Theory and Practice of Constraint Propagation*, in: Proceedings of CPDC2001 Workshop (invited talk), pages 7–14, Gliwice, 2001

[6] R. Carraghan, P. Pardalos: *An exact algorithm for the maximum clique problem*, Operations Research Letters, 9:375-382, 1990

[7] A. Eisenblätter: *Frequency Assignment in GSM Networks: Models, Heuristics and Lower Bounds*, Cuvillier-Verlag, ISBN: 3-89873-213-4, Göttingen, 2001, `ftp://ftp.zib.de/pub/zib-publications/books/PhD_eisenblaetter.ps.Z`

[8] A. Eisenblätter, A. M. C. A. Koster: *FAP web — A website about Frequency Assignment Problems*, 2000, `http://fap.zib.de`

[9] A. Eisenblätter, M. Grötschel, A. M. C. A. Koster: *Frequency Planning and Ramifications of Coloring*, in: Discussiones Mathematicae Graph Theory, 2(1), pages 51–58, 2002

[10] A. Eisenblätter, M. Grötschel, A. M. C. A. Koster: *Frequenzplanung im Mobilfunk*, in: DMV-Mitteilungen 1/2002, pages 18 ff.

[11] M. R. Garey, D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979

[12] P. van Hentenryck: *Search and Strategies in OPL*, ACM Transaction on Computational Logic, 1(2), 2000

[13] P. van Hentenryck: *The OPL Optimization Programming Language*, MIT Press, Cambridge, Massachusetts, 1999

[14] ILOG: *ILOG OPL Studio 3.5.1 Language Manual*, ILOG Mountain View, CA, USA, 2001

[15] ILOG: *ILOG Solver 5.1 User's Manual*, ILOG Mountain View, CA, USA, 2001

[16] L. V. Kale: *A Perfect Heuristic for the N Non-Attacking Queens Problem*, Information Processing Letters, pages 173–178, 1990

[17] T. Kasper, A. Bockmayr: *Branch and Infer: A Unifying Framework for Integer and Finite Domain Constraint Programming*, in: INFORMS Journal on Computing, Vol. 10, No. 3, 1998

[18] V. Kumar: *Algorithms for Constraint Satisfaction Problems: A Survey*, in: AI Magazine 13(1), pages 32–44, 1992

[19] I. J. Lustig, J.-F. Puget: *Program ≠ Program: Constraint Programming and its Relationship to Mathematical Programming*, the Institute for Operations Research and the Management Science (INFORMS), 901 Elkridge Landing Road, Suite 400, Linthicum, Maryland 21090, USA, 2001

[20] A. K. Mackworth, E. C. Freuder: *The Complexity of Constraint Satisfaction Revisited*, 59(1–2), pages 57–62, 1993

[21] K. Marriott, P. Stuckey: *Programming with Constraint: An Introduction*, MIT Press, Cambridge Massachusetts, 1999

[22] U. Montanary: *Networks of constraints fundamental properties and applications to picture processing*, in: Information Sciences 7: 95–132, 1974

[23] M. Mouly, M.-B. Pautet: *The GSM System for Mobile Communications*, Cell & Sys, France, 1992

[24] B. Nadel: *Tree Search and Arc Consistency in Constraint Satisfaction Problems*, in: Search in Artificial Intelligence, Springer-Verlag, New York, 1998

[25] F. Rossi: *Constraint Logic Programming*, Università di Padova, Dipartimento di Matematica Pura ed Applicata, Via Belzoni 7, 35131 Padova, Italy, `frossi@math.unipd.it`

[26] C. Voudouris, E. Tsang: *Solving the Radio Link Frequency Assignment Problem using Guided Local Search*, NATO Symposium on Radio Length Frequency Assignment, Sharing and Conservation Systems (Aerospace), Aalborg, Denmark, October 1998

[27] H. P. Williams: *Model Building in Mathematical Programming*, 3-rd Edition, John Wiley & Sons, ISBN 0-471-94111-5, New York, 1994

Index

Zusammenfassung

Die vorliegende Diplomarbeit beschäftigt sich mit dem Lösen des Frequenzzuweisungsproblems in GSM-Mobilfunknetzen. Ziel der Frequenzplanung ist die Zuweisung von Frequenzen an Basisstationen unter der Berücksichtigung von zahlreichen Nebenbedingungen, so dass Interferenz, welche die Empfangsqualität nachhaltig beeinträchtigen kann, weitestgehend vermieden wird.

Constraint Programming wird für die Suche von Lösungen des Frequenzzuweisungsproblems verwendet, da gegenwärtig eingesetzte Heuristiken für die Interferenzminimierung oft beim Bestimmen von gültigen Zuweisungen für schwer zu lösende Instanzen scheitern. Zuerst werden reine Zulässigkeitsprobleme untersucht. Ist es möglich, Lösungen zu berechnen, so wird auch die explizite Minimierung der Gesamtinterferenz erforscht.

Nach der Formulierung eines geeigneten mathematischen Modells wird die Komplexität des Frequenzzuweisungsproblems untersucht und gezeigt, dass es streng $\mathcal{NP}$-schwer ist. Ein Überblick über die Theorie von *Constraint Programming*, sowie ein Beweis, dass *Constraint Satisfaction Probleme* streng $\mathcal{NP}$-vollständig sind, werden ebenfalls erbracht.

Als Lösungswerkzeug wird ILOG OPL Studio verwendet, wobei OPL („Optimization Programming Language“) die Formulierung von mathematischen Modellen in einer eigenen Modellierungssprache erlaubt. Es wird untersucht, inwiefern die im Vergleich zu *Mixed Integer Progamming* zusätzlichen Elemente dieser Sprache tatsächlich mehr Möglichkeit bieten, Bedingungen in OPL Modellen zu formulieren als in *Mixed Integer Programs*.

In dieser Arbeit werden verschiedene Modelle für Zulässigkeitsprobleme vorgestellt. Unterschiedliche Modellierungsalternativen werden analysiert und miteinander verglichen. Mit Hilfe von OPL Modellen für Zulässigkeitsprobleme können sowohl für große als auch für schwer zu lösende Instanzen gültige Zuweisungen berechnet werden. Da die Qualität der gefundenen Lösungen oft unzureichend ist, werden außerdem Methoden zur expliziten Interferenzminimierung entwickelt. Minimierungsmethoden mit Hilfe von OPL werden präsentiert, eine Konstruktionsheuristik wird angewendet, und OPL Modelle für Zulässigkeitsprobleme werden mit einer Verbesserungsheuristik kombiniert. Weiterhin werden die Resultate all dieser Minimierungsmethoden mit einander verglichen.

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt.

Mathias Schulz
Berlin, 20. Februar 2003