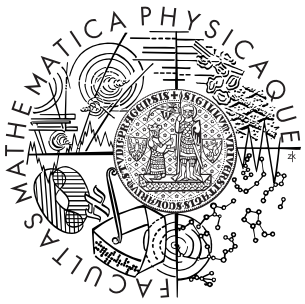
Charles University in Prague
Faculty of Mathematics and Physics

# BACHELOR'S THESIS




Tomáš Haničinec

## Constraint modeling

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Doc. RNDr. Roman Barták, Ph.D.
Field of study: Computer science

2007

I would like to acknowledge Doc. RNDr. Roman Barták, Ph.D for his professional guidance and inspiring advices that allowed me to write this thesis.

I declare that this thesis was written by myself using purely the cited sources. I agree with its' lending and publishing.

In Prague Tomáš Haničinec

# Contents

Title: Constraint modeling
Author: Tomáš Haničinec
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Doc. RNDr. Roman Barták, Ph.D.
Supervisor's e-mail address: Roman.Bartak@mff.cuni.cz

Abstract: Constraint programming is one of the possible ways how to solve complicated combinatorial (and other) problems. We model a problem using variables representing real world objects and constraints representing various relations between the objects. However, there are often many possible ways how to model a problem. And what's more, the choice of a modeling strategy can affect the resulting efficiency dramatically. Unfortunately, there is no general recipe how to model problems efficiently. Nevertheless there are still several modeling techniques, heuristics or advices that could improve the efficiency of models. Some of these techniques are problem dependent, some can be applied only to a certain classes of problems but they still often help. This thesis is trying to give more or less complete list of the most important modeling techniques along with an explanation of why, how and for which classes of problems they work best and also with empirical results underlying the presented facts.
Keywords: constraint, modeling, satisfaction, programming, SICStus, Prolog

# Introduction

We all know the classic imperative programming paradigm used to develop and solve various problems since the beginning of the computer age. All these numerous programming languages like C or Pascal work on the simple principle. When anyone wants to solve a problem he must think about it, find a detailed sequence of elementary steps (called an algorithm) leading to its solution and give it to the computer so it can perform the steps and report the result. The programming language itself provides only some kind of abstraction so one doesn't have to think on the level of machine instructions.

However, what if we want to do it a different way? Wouldn't it be much easier just to state the problem that we want to solve in some well defined form and then let the computer to solve it? Constraint programming is one of the attempts (and possibly the most successful one) to reach this ideal state. In constraint programming we just state the problem by means of variables (typically representing real world objects) and constraints (representing various relations and dependencies among variables - objects). Having the problem stated like this, we can use a generic constraint solver to solve the problem and then to tell us what we wanted to know. It's certainly much more comfortable for the programmer than the classic programming. And there's another advantage. It's also much more general. This way the computer actually *knows* the problem itself so we can ask different questions about the problem and all these questions can be answered using only the problems definition instead of explicitly writing many different programs as it would be inevitable using imperative programming. Indeed, constraint programming is widely used in practice for solving various combinatorial problems. Especially problems like scheduling or timetabling are among the most important applications of constraint programming.

Let's now illustrate constraint programming on a simple example - a so called *n-queens problem*. The setting is as follows:

We are given a standard $n \times n$ chessboard and our task is to place $n$ queens on the chessboard so that no two queens are threatening each other. The solution for n=4 is showed in Figure 1.

We now try to define the n-queens problem using variables and constraints. The most obvious way to do it would probably be using one variable for each queen (let's mark these variables $x_1, x_2, ..., x_n$). Values of the variables could than be integer numbers from 1 to $n^2$ representing the particular queens' position on the chessboard (we will further assume the fields on the chessboard are numbered horizontally from the top left corner so the top left field has number 1, the rightmost field of the top

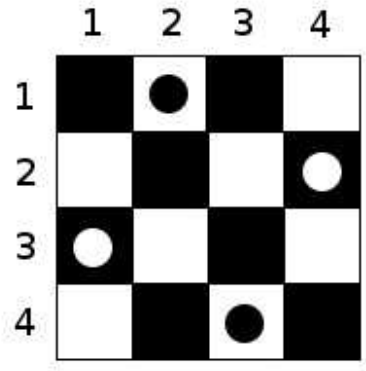



Figure 1: The solution of the n-queens problem for n=4

row has number n and so on). Now we have to express the fact that no queens are threatening each other by constraints. There are three subconditions:

Firstly, no two queens must lie in the same column. We can ensure this by $n(n-1)/2$ constraints

$$(x_i - 1) \circ n \neq (x_j - 1) \circ n$$

where $1 \leq i < j \leq n$ and $\circ$ stands for the operation of modulo (remainder after integer division).

Secondly, no two queens must lie in the same row. The constraints covering this condition could be for example in a form

$$(x_i - 1) \div n \neq (x_j - 1) \div n$$

where $1 \leq i < j \leq n$ and $\div$ stands for the operation of integer division.

Thirdly, no two queens must lie on the same diagonal. This is where we get into problems. There is simply no easy way to determine whether two fields are on the same diagonal or not. Possibly the easiest way is to use $n(n-1)/2$ of rather obscure constraints in the form

$$|((x_i - 1) \div n) - ((x_j - 1) \div n)| \neq |((x_i - 1) \circ n) - ((x_j - 1) \circ n)|$$

where $1 \leq i < j \leq n$, $\div$ is the operation of integer division and $\circ$ is the operation of modulo. This formula may need to be explained. The expression $(x_i - 1) \div n$ stands for the number of the row where queen $x_i$ is located. Similarly the expression $(x_i - 1) \circ n$ stands for the number of the column where queen $x_i$ is located. The formula therefore says that for each two queens $x_i$ and $x_j$, the difference between their rows ant the difference between their columns must not be equal. That means the queens must not lie on the same diagonal.

We have now defined the n-queens problem using variables and constraints. But is the way we just showed (and the reader can see it's a bit awkward way) the only possible way to do so? The answer is of course no.

Let's try a different approach. What if the variables would represent the *rows* of the chessboard instead of the queens? What would change? The variables' values

would be integer numbers from 1 to n (the value of variable $x_i$ would represent the column that the queen in the i-th row is placed in). And what about constraints? The fact that there must be exactly one queen in each row is ensured by the semantics of the second model without any additional constraints (exactly one value has to be assigned to each variable). The fact that there must be exactly one queen in each column is easy to express by $n(n-1)/2$ constraints

$$x_i \neq x_j$$

where $1 \leq i < j \leq n$.

And finally the fact that no two queens must lie on the same diagonal which caused so much trouble in the previous model can be easily covered by constraints in the form

$$|x_i - x_j| \neq |i - j|$$

We have obtained another model of the n-queens problem by looking at the problem from a different angle. And what's more, the second model seems to have some advantages compared to the first model. The second model is definitely simpler, more consistent and easier to understand. And what if we try to measure the time a constraint solver needs to solve the n-queens problem? The following table gives the empirical comparison of both the above presented models implemented in SICStus Prolog (version 4.0.1). All the time values are in seconds[1]. The source codes of the programs used can be found in the appendix (programs number 1 and 2).

| N (chessboard size) | 4 | 8 | 12 | 16 | 20 | 30 |
|---|---|---|---|---|---|---|
| first model | 0.010 | 24.806 | 5312.469 | - | - | - |
| second model | 0.000 | 0.010 | 0.030 | 0.050 | 0.140 | 2.503 |

Table 1: Empirical comparison of the two presented models of the n-queens problems. Time values are in seconds

We can see that the second model consumes significantly less time than the first model. So not only the second model is simpler but it is also much more efficient.

We have shown that the choice of the modeling strategy can affect the resulting efficiency dramatically. Unfortunately, there is no general recipe how to model problems efficiently. However, there are several modeling techniques, heuristics and advices that could improve the models and their efficiency. Some of these techniques are problem dependent, some can be applied only to a certain classes of problems. However, they frequently help. This thesis is trying to give more or less complete list of the most important modeling techniques along with an explanation of why, how and for what classes of problems they work best and also with empirical results underlying the presented facts.

[1]Computer configuration - Intel Celeron 1300 MHz, 256 MB RAM, Microsoft Windows XP

## Organization of the thesis

**Chapter 1 - Principles of Constraint Satisfaction** contains the formal introduction into the field. It covers basic definitions and conventions that are used throughout the thesis. It also introduces some of the most important constraint satisfaction techniques so the reader will know how the solvers work inside which is sometimes useful to fully understand why certain decisions lead to more efficient models.

**Chapter 2 - Constraint Modeling** is the core part of the thesis. It contains the actual list of modeling techniques. It's divided into four parts in accordance with the nature of particular techniques. Each technique is explained in detail, empirically demonstrated on at least one example and the results are eventually further discussed.

**Conclusion** contains generalization and summary of the presented techniques, ideas and experimental results.

# Chapter 1

# Principles of Constraint Satisfaction

To be able to fully understand the methods of constraint modeling, we need to know several basic concepts of constraint satisfaction at first. This chapter therefore covers the basic definitions and conventions as well as some of the techniques used by current constraint satisfaction solvers. Section 1.1 introduces formal definitions of the terms we are going to use throughout this thesis. As a matter of fact, the most of it can be thought of as a formal transcription of our simple n-queens example from the introduction. Section 1.2 is focused on constraint propagation, the most powerful technique of constraint solvers. However, in practice we aren't usually able to solve CSPs using constraint propagation alone. Some amount of search is then necessary to find the assignments satisfying all the constraints. The search algorithm along with some techniques used to improve its efficiency is covered in section 1.3. It is useful to familiarize with these issues in order to understand how certain modeling decisions influence the resulting efficiency of the problem solving.

## 1.1 Definitions

First of all - what do we mean by a constraint satisfaction problem? A *constraint satisfaction problem* (CSP) is a triplet $(X, D, C)$. $X = \{x_1, x_2, ..., x_n\}$ is a finite set of *variables*. $D = \{d_1, d_2, ..., d_n\}$ is a finite set of *domains*. A domain is a finite set of *values* (usually integers), that are being assigned to the variables. Finally, $C = \{c_1, c_2, ..., c_m\}$ stands for a finite set of *constraints*. A constraint is basically a relation defined over some subset of the variables limiting the possible combinations of values they can take. It can be expressed as a mathematical formula of some kind or extensionally as a list of acceptable tuples of values. Formally, each constraint can be thought of as a pair $(\sigma, \rho)$ where $\sigma \subseteq X$ is a set of variables involved in the constraint called it's *scope* and $\rho$ is a subset of the cartesian product of the corresponding domains. An *assignment* is a pair $(x_i, a)$, where $x_i \in X$ and $a \in D_i$ meaning that value $a$ was assigned to variable $x_i$. A *complete(partial) assignment* is an assignment to all(certain) variables in $X$. We say that an (partial) assignment

*satisfies* a constraint $c = (\sigma, \rho)$ if all the variables in $\sigma$ have an assigned value and these values form an element of $\rho$. A *solution* to a CSP is a complete assignment satisfying all the constraints (a CSP can therefore have none or more than one solution, too).

A constraint with the scope of just one variable is called a *unary constraint* (it's basically a domain restriction), a constraint with the scope of size of two is called a *binary constraint* and so on. A unary constraint $c_1$ with the scope $x_i \in X$ is said to be *node consistent* if for every value $v_i \in d_i$ ($d_i \in D$) the assignment $(x_i, v_i)$ satisfies the constraint. Let's now have a binary constraint $c_2$ between variables $x_1, x_2$ with domains $d_1, d_2$ respectively. We say that the constraint $c_2$ is *arc consistent* if for every value $u \in d_1$ there exists a value $v \in d_2$ so that the partial assignment $x_1 = u, x_2 = v$ satisfies $c_2$ and vice versa. So it basically means that every value must participate in at least one pair of values satisfying the constraint.

We can extend this concept further to non-binary constraints obtaining a so called generalized arc consistency. So much like above let's have a constraint $c_3$ with variables $x_1, ..., x_n$ and domains $d_1, ..., d_n$ respectively. Now for every value $v_j \in d_j$ there must exist a tuple of values $v_1, ..., v_{j-1}, v_{j+1}, ..., v_n$ from domains $d_1, ..., d_{j-1}, d_{j+1}, ..., d_n$ so that the partial assignment $x_1 = v_1, ..., x_{j-1} = v_{j-1}$, $x_j = v_j, x_{j+1} = v_{j+1}, ..., x_n = v_n$ satisfies $c_3$. If the above written statement holds for every domain $d_j \in \{d_1, ..., d_n\}$, the constraint $c_3$ is said to be *generalized arc consistent*. More informally again - it must be possible to extend every possible value from every possible domain by other values from other domains to satisfy the constraint.

Finally, constraint $c_3$ from the previous example with numerical domains is said to be *bounds consistent* if the generalized arc consistency paradigm holds for at least the boundary values of each domain (the maximal and the minimal value). So in contrast to the generalized arc consistency, it is not required to hold for every possible value of the domain here. There exists many other consistency levels such as path consistency but these levels are not actually used in solvers frequently and their detailed description is well beyond the scope of this thesis. For more details about consistency techniques and their enforcing see for example [9, 3].

A CSP including only unary or binary constraints is called a *binary CSP*. Two CSPs are said to be *equivalent* if they have the same set of solutions. Now let's focus a bit on binary CSPs because of its great theoretical meaning. It appears that any arbitrary CSP can be transferred to an equivalent binary CSP (there are several techniques to do so. Some of them are for example the *hidden variable* translation and the *dual graph* translation (see [3]). Therefore, the binary CSPs are the only problems we could be concerned about while working with constraint satisfaction techniques. Unfortunately, the non-binary to binary CSP translation techniques are computationally quite expensive and are rarely used in practice. The benefit we gain (a simpler binary CSP) isn't usually worth the extra time spent on the translation. However, the binary CSPs have still a privileged position in a theoretical work where we are more interested in formal correctness and simplicity than in actual low level performance.

When we extend the original CSP definition by an *objective function* we obtain a so called *constraint optimization problem* (COP). An objective function is used to

determine a quality of solutions. While solving a COP, we are not interested in just any or all the solutions but we are looking for the optimal solution instead. COPs are usually solved by an instance of *branch and bound* algorithm - in this case it's basically repeated solving of the corresponding CSP with gradually added constraints representing certain bound on its solution quality. Because of this, if there's a good model for a CSP, then the same model is equally good for the corresponding COP and for the purposes of this thesis we can unify COPs and CSPs and concentrate only on the simple CSPs.

We introduced some of the basic concepts of constraint satisfaction in this section. However, we still don't know how the CSPs are actually solved. We will now focus on *constraint solvers*. Constraint solvers are programs that are able to search and find solutions to a specific CSP. This is not an easy task assuming we want to use such a solver to solve large real-life instances of problems in a reasonable time. Great majority of today's constraint solvers use a combination of constraint propagation and search to find the solutions. Both of these techniques will be explained in the following sections

## 1.2 Constraint propagation

We will start with a simple example. Consider a CSP with two variables $x_1, x_2$ and one simple constraint $x_1 > x_2$. The variables domains are $d_1 = \{1, 2, 3, 4\}$ and $d_2 = \{3, 4, 5, 6\}$. The most naive way to solve such a CSP would be to try all the complete assignments sequentially and to test if these assignments violates the constraint. However, even in this simple case we would for example test pairs $\{1, 3\}$, $\{1, 4\}$, ..., $\{2, 3\}$, ..., $\{3, 3\}$, $\{3, 4\}$, $\{3, 5\}$ and $\{3, 6\}$ without finding any solution. It should be clear that such a primitive method (called *generate and test* by the way) is practically unusable. But what would happen if we use the constraint to help us to actually construct a solution by pruning the values that cannot be a part of a solution? When we look at the problem we can see immediately that none of the values $1, 2, 3$ from the domain $d_1$ can be extended by any value from $d_2$ to form a solution (to satisfy the constraint). We can therefore delete these three values from $d_1$ and the resulting CSP would be equivalent to the original CSP and also much simpler. We can continue further and delete the values $4, 5, 6$ from $d_2$ for the same reason. Now both the domains contain only one element ($d_1 = \{4\}, d_2 = \{3\}$) and these elements form the only solution to the above defined CSP. What we did was actually enforcing the above defined arc consistency on constraint $x_1 > x_2$. This was a simple example of the so called *constraint propagation*, the most powerful technique of constraint solvers [28].

The very reason why we are able to solve real-life problems through constraint satisfaction lies in the ability of solvers to use constraints actively to remove conflicting values from variables domains as it was just presented on our simple example. In practice, each constraint has a special algorithm attached to it that is evoked every time some of the domains in the constraint scope changes to propagate these changes to other domains. Such an algorithm is called a *filtering algorithm* because it filters

the domains of variables in a constraints scope. And since the domain changes are propagated further through the constraint by filtering, the whole process is called the *constraint propagation*.

Now we explained what the filtering is and when it is evoked but we didn't yet explained how it works. There are several possibilities but most of the solvers filter the domains by enforcing some of the consistency levels described earlier. This is always a kind of a tradeoff between a time loss caused by a consistency enforcing algorithm and the possible gain caused by smaller domains. The stronger a consistency level is, the more values it could eliminate and the more complex and time consuming also its enforcing algorithm is. Most of the solvers enforce arc consistency on binary constraints and bounds consistency on arithmetical constraints. Bounds consistency can be thought of as a certain compromise between enforcing computationally expensive generalized arc consistency and doing nothing. The algorithm is relatively fast and it propagates at least a domains lower and upper bound (if not all the values like generalized arc consistency does). Some solvers also support the so called global constraints which usually allow us to enforce higher levels of consistency by reasoning on a set of constraints as it is a single constraint. Global constraints as one of the modeling techniques will be widely discussed in section 2.3.1.

We have seen that we can use constraints to reduce the size of domains by enforcing certain level of consistency on these constraints. However, the levels of consistency used by current constraint solvers are not strong enough to solve the problem completely. The solvers are generally not able to find solutions just by enforcing consistency on all the constraints of a CSP. It is therefore necessary to combine constraint propagation with some kind of an additional search algorithm. The next section describes the main stream search algorithm as well as some of the techniques and heuristics used to improve its efficiency.

## 1.3 Search

Let's have three variables $x_1, x_2, x_3$, all of them with domain $\{1, 2\}$ and three binary constraints $x_1 \neq x_2$, $x_2 \neq x_3$ and $x_3 \neq x_1$. The reader can easily see that all three constraints are arc consistent. And still - it's clear that the specified CSP has no solution and all the values could have been removed from the domains by filtering algorithms.

We have just seen that constraint propagation alone may be insufficient to solve an arbitrary CSP. To be able to find a solution we often have to search for it. Current constraint solvers usually interleave constraint propagation with an enhanced backtracking algorithm. In section 1.3.1 we will describe how the search actually works in constraint solvers while section 1.3.2 introduces some of the techniques used to enhance the efficiency of the search.

### 1.3.1 Labeling

The main stream algorithm for constraint satisfaction is actually very simple. In each step the algorithm selects some yet uninstantiated variable and assigns it a value from the corresponding domain. The assignment of a value to a variable is expressed by an extra constraint that is added to the original CSP (so the solver basically solves just one automatically modified CSP). This additional constraint is then used to reduce some of the domains via constraint propagation. If any domain becomes empty (meaning that the current CSP doesn't have a solution), the algorithm backtracks canceling the last assignment (by deleting the last added constraint from the current CSP). The algorithm terminates when a solution to the original CSP is found or all the possible combinations of assignments are tried without finding a solution. The whole process is often called *labeling* because the variables are consequently labeled with values.

Let's now present the above described algorithm's work on a simple example. Let's have variable $x_1$ with domain $d_1 = \{1, 2, 3, 4, 5\}$. Sometimes during the run of the algorithm value 3 is selected for an assignment. Constraint $x_1 = 3$ is therefore added to the CSP we are actually solving. Since $x_1 = 3$ is an unary constraint, the node consistency will be enforced on it. So the filtering algorithm will delete all the values but 3 from the domain $d_1$. The change of the domain $d_1$ will also cause invoking of filtering algorithms of all the constraints with the variable $x_1$ in their scopes. This way the change in $d_1$ is propagated into other domains too. Then the process is repeated until a solution is found or some domain becomes empty.

In the example we used $x_1 = 3$ as an additional constraint. However, there are other choices possible for an additional constraint representing an assignment. We can for example consequently split the domain along the middle value instead of assigning just one value at a time. And this strategy may well prove to yield better results in certain classes of problems. We will now present three of the labeling strategies using the above defined variable $x_1$.

*Enumeration* is the most obvious strategy. It simply tries to assign one value after another till there are no values left. The constraints consequently added to a CSP would be $x_1 = 1, x_1 = 2, x_1 = 3, ...$ and so on until all values in the domain are tried. The constraint propagation would reduce the domain $d_1$ to just one value.

*Bisection* gradually splits the domain along the middle value as was already mentioned above. The additional constraints would be $x_1 \leq 3, x_1 > 3$ in this case. The propagation would first delete the values $4, 5$ leaving the domain $d_1$ equal to $\{1, 2, 3\}$. If the search failed, the other half of the domain would be used instead.

*Step labeling* works much like the enumeration. The difference is that it doesn't try to assign the next value after the search fails immediately. It does only note that the currently assigned value was unsuccessful. The constraints would be for example $x = 1, x \neq 1$. The propagation would eliminate all the values but 1 from $d_1$. If the search failed, value 1 would be eliminated leaving domain $d_1$ equal to $\{2, 3, 4, 5\}$.

### 1.3.2 Variable and value selection

Let's have a more detailed look on the phase where the algorithm proceeds forward by choosing a variable and a value from its domain for an assignment. First we have to choose a variable. The first choice used as a default is just to select the next variable in some initial fixed ordering. But this way we cannot take advantage of eventual additional information discovered during the search. Wouldn't it be better to order the variables dynamically as the search continues and in each step to choose the variable we think (for some reason) will lead to the smallest remaining search space? The ordering of variables for an assignment may have a significant effect on the running time of the whole algorithm. Unfortunately it appears that there doesn't exist any universally best variable selection scheme and mostly we have just to try it to know which one is the best for our particular problem. There are two main variable selection schemes:

The *left-most* variable selection simply chooses variables according to some initial fixed ordering. It doesn't use additionally discovered information but on the other hand, we can directly influence the performance of the algorithm by ordering the variables ourselves. Thus our knowledge of the particular problem can be used to build a more efficient model.

The *fail-first* variable selection always chooses the variable that most likely causes the search to fail [14]. The fail-first strategy is based on an idea that whenever the search gets into a dead-end branch of a search tree, it's good to realize it as soon as possible so the algorithm can backtrack and it doesn't waste time on searching the rest of the branch. The question remains how to choose such a variable. There are several heuristics that are used to determine which variable will probably cause the search to fail. We will present some of them. Probably the most obvious one is the *dom* heuristic [14]. It simply selects the variable with the smallest domain. It should be clear that the smaller domain of the variable is, the sooner we try all the values and the search fails. But what if there are two or more variables with the smallest domain? The dom heuristic chooses arbitrarily. Another possibility, known as the *dom+deg* heuristic chooses the most-constrained variable instead (the more constraints contain the variable in their scopes the greater the probability that some of these constraints will be violated is) [7, 24]. It also might be useful to combine the "smallest domain" and the "most constrained" approach. The resulting heuristic is called *dom/deg* [5, 25]. For each variable the quotient of the variables' domain size and the number of constraints that contain the variable in their scopes is computed. The heuristic then chooses the variable with the small value of the quotient (small value means that the variable has a small domain and it's participating in many constraints). There are also other heuristics based on the fail-first strategy but it is not necessary to explain them for purposes of this thesis.

We have covered some of the variable selection principles but what about the value selection? The selection of a value for an assignment can influence the performance of the search just as the selection of a variable does. By selecting a value for an assignment we are actually selecting a branch of the search tree the search algorithm is going to visit next. It is usually a good idea to pick the most promising values

first. One of the possible heuristics for choosing the most promising value is the *min-conflict* heuristic which chooses the value with the minimal number of conflicts with the unlabeled variables [17].

More about variable and value selection principles and heuristics can be found for example in [29].

# Chapter 2

# Constraint Modeling

Constraint modeling is a crucial part of constraint programming. Since the solvers themselves usually use fixed algorithms (for example some instance of the above described scheme), the constraint modeling is often our only chance to influence the final performance of problem solving. Unfortunately, there are nearly no general rules telling us how a good and efficient model should look like. The final efficiency of the model closely depends on the inner behavior of the constraint solving algorithm. Unfortunately this behavior is often too complex for us to fully understand all the dependencies and interactions influencing the resulting runtime. The only way to determine the efficiency of a model for sure is usually just to try it empirically. However, there are still some suggestions we should keep in mind while modeling a problem. Firstly - a good and efficient model should contain as much information about the problem as possible. The more information we are able to encode into constraints the better the chance of removing more values from domains via constraint propagation we usually have. It is also a good idea to keep the number of variables in a model as small as possible. Additional variables often slow down the search phase because we have to do more choices and the search tree is deeper. These suggestions are of course not guaranteed to work in every possible case. There are many cases where an additional information about a problem included in a model doesn't help (for example because the constraints used to express the information are cumbersome and don't propagate well). Or we can also easily find cases where a model with more variables is more efficient than a model with less variables (for example because the additional variables allow us to express certain constraints more consistently and improve their propagation). The practical usage of the above explained suggestions will be closely described in some of the later sections.

All the techniques in this chapter are presented in a more or less uniform manner. At first a technique itself is explained, it is widely discussed how and mainly why it works (possibly also for what classes of problems it works best). Then a simple problem is defined and two models of the problem are suggested. The first model is usually the most obvious one, without any enhancements. The second model makes use of the presented modeling technique. The two models are then implemented in SICStus Prolog, version 4.0.1 (see [22]) and their performance is compared[1]. The

[1]Computer configuration - Intel Celeron 1300 MHz, 256 MB RAM, Microsoft Windows XP

runtime is measured with the SICStus Prolog predicate `statistics`. Times over 500 seconds are usually replaced by "-". Several test runs are performed for the purposes of the comparison, each run with different combination of labeling and variable selection scheme. The results of the comparison are then presented and discussed.

The constraint modeling techniques included in this chapter are grouped into four categories according to the nature of the technique. **Section 2.1** describes techniques based on addition of new constraints to an existing model. These techniques make a direct use of the above mentioned suggestion about an additional information in a model. **Section 2.2** is focused on combinations of different models. If we have two or more efficient models of a problem, it might be a good idea to use all of them simultaneously. That way the information would often be propagated earlier by making use of the advantages of all the participating models (at the expense of an additional time spent on the propagation, of course). Finally **section 2.3** is focused on techniques that don't fit to any of the two previous categories. Dealing with global constraints and using auxiliary variables are probably the main techniques presented in this section.

## 2.1 Adding constraints

Techniques presented in this section involve an addition of new constraints into an existing model. We already mentioned that an additional information in the model can help. When we know something about the problem that is not yet included in the model and we are able to encode this information into constraints, it might be a good idea to add these new constraints into the model in order to improve propagation. The main assumption of this technique is that the new constraints must propagate well so the time savings caused by an additional constraint propagation outweighs the additional time consumption caused by the filtering algorithm that is running on the new constraints.

**Section 2.1.1** describes so called implied constraints. Implied constraints don't change the set of solutions to a CSP (they are implied by the original constraints), but they carry additional information that can help to eliminate more values out of the domains via constraint propagation. **Section 2.1.2** is focused on symmetry breaking. Many of the problems we are trying to solve through constraint satisfaction are somehow symmetrical. If we enhance the model with constraints that forbid the symmetrical situations or solutions, we can reduce the search space significantly, while we can still easily reconstruct all the solutions after the search is done.

### 2.1.1 Implied Constraints

Implied constraints (sometimes the term redundant constraints is being used instead) are used to express additional information about the problem that we are actually modeling [26]. Implied constraints are not necessary to find a solution and they don't change the set of solutions to the problem any way. Any model would still work fine

without them. The reason for constructing a model with more constraints than necessary lies in the ability of constraint solvers to use all the constraints to reduce the domains via constraint propagation. Constraint propagation is a very powerful tool and it can improve the time efficiency of the problem solving dramatically. By addition of some new constraints we are actually trying to incorporate more of the constraint propagations power into the problem solving.

So when it could be an appropriate time for utilization of the implied constraints technique? Implied constraints are worth considering whenever we have modeled a problem and we have discovered some additional information about this problem that is not directly expressed in the present model. If this information can then be easily and consistently encoded into constraints, the model with these implied constraints added will probably be more efficient than the original model (the one without the implied constraints). The problem occurs when the additional information is hard to express by means of constraints and when the resulting constraints are difficult to propagate. New constraints always add some kind of overhead (caused by runs of their filtering algorithms). This overhead, if the constraints don't propagate well, might be unproductive. If we want the implied constraints technique to be productive and worthwhile, the propagation of the additional constraints must always compensate and even outweigh the overhead.

Now consider the problem called a *Golomb ruler*: A Golomb ruler is defined as a set of $n$ integers $0 = a_1 < a_2 < ... < a_n$ representing marks in a way that all the $n(n-1)/2$ differences between the marks $(a_j - a_i, 1 \leq i < j \leq n)$ are distinct. The ruler is then said to have length $a_n$. The objective is to find an optimal (a minimum length) ruler [13]. The optimal ruler with five marks is illustrated in Figure 2.1.

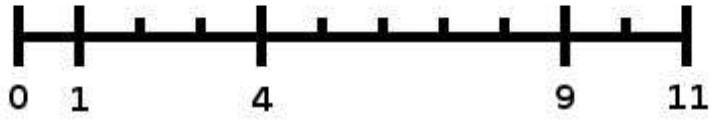



Figure 2.1: The optimal Golomb ruler with five marks

Let's now model the Golomb ruler problem as a CSP. Probably the most straightforward way to do so is to use $n$ variables $x_1, x_2, ..., x_n$ representing the positions of the marks on the ruler. The constraints ensuring different distances between marks would then be in the form

$$|x_j - x_i| \neq |x_l - x_k|$$

where $1 \leq i, j, k, l \leq n, i < j, k < l, i \leq k$ and $i \neq k$ or $j \neq l$. The implementation of this model can be found under number 3 in the appendix. Figure 2.2 shows the summary of the basic model

Now the question arises. What else do we know about the problem that can be used to improve the efficiency of our simple model? We can for example try to estimate bounds of individual distances between marks more accurately. To be able

**Variables**:

$x_1, x_2, ..., x_n$

**Constraints**:

$|x_j - x_i| \neq |x_l - x_k|$

Figure 2.2: Model **Golomb basic**

to do it, we must first add the set of constraints in the form

$$x_k < x_{k+1}$$

$(1 \leq k < n)$ ensuring that the marks will be sorted in ascending order. The reader can find more about such constraints and their impact on the efficiency of the model in the next section. Now we are ready to infer the lower and upper bounds of individual distances.

Firstly the lower bound. We know that each distance $|x_j - x_i|$ is assembled of $(j - i)$ primitive distances $|x_{k+1} - x_k|$ $(i \leq k < n$, note that this is only true when the marks are sorted). Each primitive distance is at least 1 so we can state

$$|x_j - x_i| \geq (j - i)$$

Moreover, the primitive distances must be pairwise different too so the minimal sum of $(j - i)$ primitive distances is $1 + 2 + ... + (j - i) = (j - i)(j - i + 1)/2$ and we can further refine our lower bound to

$$|x_j - x_i| \geq (j - i)(j - i + 1)/2$$

Now what about the upper bound? Each distance $|x_j - x_i|$ is assembled of $(j - i)$ primitive distances so the rest of the ruler consists of $n-1-(j-i)$ primitive distances. As we know, each primitive distance is at least 1 and the distances must be different. The minimal length of the rest of the ruler (the ruler without $|x_j - x_i|$) is therefore $1 + 2 + ... + (n - 1 - j + i) = (n - 1 - j + i)(n - j + i)/2$ and we can state the upper bound on $|x_j - x_i|$ too:

$$|x_j - x_i| \leq x_M - (n - 1 - j + i)(n - j + i)/2$$

where $x_M$ is the total length of the ruler (maximum of $\{x_1, x_2, ..., x_n\}$).

We can now enrich our initial model of the Golomb ruler problem with the above stated lower and upper bound implied constraints and compare the performance of both the models. The implementation of the model with implied constraints in SICStus Prolog can be found as the program number 4 in the appendix. The summary of this model can be found in figure 2.3

The following tables show us the comparison of both models.

Table 2.1 is focused on the direct comparison of the models on the problems of different sizes. The default settings of labeling strategy and variable selection heuristic were used (in SICStus Prolog it means the step labeling strategy an the left-most variable selection).

**Variables**:

$x_1, x_2, ..., x_n$

**Constraints**:

$|x_j - x_i| \neq |x_l - x_k|$

$x_k < x_{k+1}$

$|x_j - x_i| \geq (j - i)(j - i + 1)/2$

$|x_j - x_i| \leq x_M - (n - 1 - j + i)(n - j + i)/2$

Figure 2.3: Model **Golomb implied**

| marks | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|
| optimal length | 6 | 11 | 17 | 25 | 34 |
| Golomb basic | 0.020 | 1.712 | 149.434 | - | - |
| Golomb implied | 0.010 | 0.020 | 0.180 | 3.394 | 56.671 |

Table 2.1: Golomb ruler - empirical comparison of the two presented models. Time values are in seconds

We can see that the difference between the basic model and the model enhanced with the implied constraints is really huge. All the implied constraints that we used in the model were simple, binary and easy to propagate constraints. And these constraints expressed the facts that helped the constraint solver to prune the search space significantly. The gain caused by the smaller search space therefore significantly outweighed the loss caused by handling of additional constraints and that's why the technique was so successful. Implied constraints heavily payed off in this case.

Table 2.2 shows us the dependency between the performance of the models and the choice of the labeling strategy and the variable selection heuristic.

| labeling strategy | step labeling | | enumeration | | bisection | |
|---|---|---|---|---|---|---|
| variable selection | left-most | fail-first | left-most | fail-first | left-most | fail-first |
| Golomb basic (n=5) | 1.712 | 1.712 | 1.372 | 1.372 | 1.672 | 1.683 |
| Golomb implied (n=7) | 3.394 | 6.880 | 2.925 | 5.759 | 3.334 | 6.859 |

Table 2.2: Golomb ruler - dependency between labeling settings and the final performance of a model. Time values are in seconds

We can see that the enumeration labeling strategy has proved to be the best choice for both models. Moreover, in the case of the model enhanced with the implied constraints there is a significant difference between the run times when the left-most variable selection was used and the run times when the fail-first variable selection was used. The fail-first variable selection has proved to be significantly less efficient than the left-most variable selection in this case.

At the end of this section we should also mention the concept of the so called *dominance rules*. The concept originates from other areas of research and its application in constraint satisfaction is not very frequent yet. Dominance rules are by its nature very similar to implied constraints. They also extensively express some additional information that helps to prune the search space. But unlike the implied constraints, dominance rules actually can reduce the set of solutions to a problem. The important thing is that there must exist at least one solution satisfying dominance rules. This concept is therefore suitable for problems where we are satisfied with just one arbitrary solution. In this case dominance rules may bring more efficient model than implied constraints because they are allowed to prune greater parts of the search space (mainly the parts including some of the solutions). However, there is not many problems that can make use of dominance rules and that is why this technique is only shortly mentioned here. More about dominance rules and examples of their application in constraint satisfaction can be found for example in [18] and [1].

### 2.1.2 Symmetry breaking

Many of the problems we are trying to model and solve through constraint satisfaction are somehow symmetrical. In this section we will show and break two types of symmetries.

The first type of symmetry (let's call it the *model symmetry*) occurs whenever the variables in a model represent identical objects. Original objects are not distinguishable in a problem while the variables representing them always are. For example recall the Golomb ruler problem from the previous section. We modeled this problem using $n$ variables representing the marks on a ruler. But the marks on a real world ruler are uniform and identical. It doesn't matter which particular mark is placed as the first on a ruler. The variables on the other hand are distinguishable. Variable $x_1$ is something completely different from variable $x_2$ for the solver. Without breaking this symmetry, the solver would uselessly search through all the possible permutations of the variables and each solution would be re-found $n!$ times.

The variable symmetry occurs when a problem itself is symmetrical. Knowing that, we can fully reconstruct the symmetrical solutions without actually having to find them or even search for them. We will explain this concept using the example of the Golomb ruler problem again. Whenever we find a solution to the Golomb ruler problem, we can obtain another solution immediately by just "turning the ruler round by 180 degrees". For example the solution $(0, 1, 4, 6)$ (the optimal solution for the Golomb ruler with four marks) yields also the (symmetrical) solution $(0, 2, 5, 6)$. Without breaking this type of symmetry the search space would be significantly bigger. There would exist a symmetrical alternative to every (partial) assignment that would have to be searched independently and the only result of such an additional search would eventually be the discovery of a symmetrical solution that we do know anyway.

We have already explained the types of symmetries as well as how they influence the efficiency of the model. Now we will try to break these symmetries in order to

improve the model. Both types of symmetries can be broken by addition of *symmetry breaking constraints* to the original model of the problem. This is very similar to the implied constraints concept described in the previous section. The symmetry breaking constraints also express some additional information about a problem that is added into model to improve constraint propagation. However, unlike the implied constraints the symmetry breaking constraints actually can reduce the set of solutions to the problem by deleting symmetrical solutions out of it.

Recall once again the Golomb ruler problem and its basic model from the previous section. We used $n$ variables $x_1, x_2, ..., x_n$ representing marks on the ruler and constraints in the form

$$|x_j - x_i| \neq |x_l - x_k|$$

(where $1 \leq i, j, k, l \leq n, i < j, k < l$ and $i \neq k$ or $j \neq l$) representing different distances between marks. The source code of such a model can be found in the appendix as the program number 3.

It should be clear that there is nothing preventing the search for symmetrical solutions in this model. For example for $n = 3$ we obtain 12 "optimal" solutions of the length 3: $(0, 1, 3)$, $(0, 3, 1)$, $(1, 0, 3)$, $(1, 3, 0)$, $(3, 0, 1)$, $(3, 1, 0)$ and $(0, 2, 3)$, $(0, 3, 2)$, $(2, 0, 3)$, $(2, 3, 0)$, $(3, 0, 2)$, $(3, 2, 0)$. After looking at these solutions we can see immediately that all we actually need to know is just the first solution $(0, 1, 3)$. All the other solutions are symmetrical to the first one in the means of the two types of symmetry described earlier in this section. Fortunately both types of symmetry are very easy to break in this case.

First the model symmetry. We want to prevent an occurrence of different permutations of the same values among the solutions (such as for example the solutions $(0, 1, 3)$ and $(0, 3, 1)$ from the previous example). This could be easily achieved by ordering the variables in ascending order, that means by addition of symmetry breaking constraints in the form

$$x_i < x_{i+1}$$

(where $1 \leq i < n$) into the model (the reader may remember that we already used such constraints in previous section). The source code of such an improved model can be found in the appendix under number 5 an the summary is in Figure 2.4.

**Variables**:
$x_1, x_2, ..., x_n$
**Constraints**:
$(x_j - x_i) \neq (x_l - x_k)$
$x_k < x_{k+1}$

Figure 2.4: Model **Golomb symmetry 1**

Now what about the variable symmetry? We don't want the solver to search for solutions that are merely turned round by 180 degrees from each other (such as for example $(0, 1, 3)$ and $(0, 2, 3)$). Since the distances between marks must be different,

we can break the variable symmetry by addition of just one constraint

$$x_2 - x_1 < x_n - x_{n-1}$$

meaning that the first difference is less than the last. The model of Golomb ruler problem with both types of symmetry broken implemented in SICStus Prolog can be found in the appendix under number 6. The summary of such a model can be found in figure 2.5.

**Variables**:
$x_1, x_2, ..., x_n$
**Constraints**:
$(x_j - x_i) \neq (x_l - x_k)$
$x_k < x_{k+1}$
$x_2 - x_1 < x_n - x_{n-1}$

Figure 2.5: Model **Golomb symmetry 2**

The empirical comparison of all the models discussed in this section is summarized in Table 2.3. The labeling settings were left to default (the step labeling strategy and the left-most variable selection)

| marks | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|
| length | 11 | 17 | 25 | 34 | 44 | 55 |
| Golomb basic | 1.712 | 149.434 | - | - | - | - |
| Golomb symmetry 1 | 0.000 | 0.020 | 0.241 | 3.545 | 54.278 | - |
| Golomb symmetry 2 | 0.000 | 0.010 | 0.110 | 1.653 | 25.877 | 368.460 |

Table 2.3: Golomb ruler - empirical comparison of models from section 2.1.2. Time values are in seconds

From the comparison we can see that the most successful has proved to be the final model with both the symmetries broken (as we would altogether expect). Nevertheless, the biggest improvement of efficiency of the model was undoubtedly caused by breaking the model symmetry. This is completely in accordance with the fact that for a problem of the size n there are n! solutions that are value symmetrical to each other while for the variable symmetry there are only 2 symmetrical solution. The model symmetry breaking therefore cuts off significantly bigger parts of the search space than the variable symmetry breaking.

Table 2.4 summarizes the dependency between the selection of labeling settings and the performance of the models presented in this section.

The table shows us that there is nearly no difference between different labeling settings this time (except the basic model that was discussed earlier). The values stayed more or less the same no matter what labeling strategy or variable selection we used.

| labeling strategy | step labeling | | enumeration | | bisection | |
|---|---|---|---|---|---|---|
| variable selection | left-most | fail-first | left-most | fail-first | left-most | fail-first |
| Golomb basic (n=5) | 1.712 | 1.712 | 1.372 | 1.372 | 1.672 | 1.683 |
| Golomb symmetry 1 (n=8) | 3.545 | 3.495 | 3.445 | 3.455 | 3.425 | 3.465 |
| Golomb symmetry 2 (n=9) | 25.877 | 26.598 | 26.098 | 26.318 | 25.727 | 25.837 |

Table 2.4: Golomb ruler - dependency between labeling settings and the final performance of the model. Time values are in seconds

## 2.2 Combining different models

Sometimes we can encounter a situation that we managed to create two different models of the problem that both propagate well. We can use either one of them to solve the problem. But if the models are both good and if they use a different perspective to see the problem, it might be beneficial to combine them into one compound model. That way the solver can fully exploit the advantages of both models at the same time and find a solution more quickly than if we use the models separately.

**Section 2.2.1** presents the idea of a dual model. A dual model can be constructed from the model of a so called permutation problem (a problem where the search space is formed by the permutations of variables) by switching the role of values and variables. Dual model is not a combination of models in any way but it is closely related and we are going to need it in the following sections anyway. **Section 2.2.2** describes the most straightforward case of the model combination, the combination of two different models into one. The models are used simultaneously and the dependencies between their individual variables are expressed by the so called channeling constraints. That way, any changes in domains are propagated between the models immediately through the channeling constraints propagation. **Section 2.2.3** extends this concept by omission of certain constraints from the compound model. Since the compound model always contains many redundant constraints, it could be convenient to remove those of them that don't propagate well. Not only that we use the advantages of both the models but we also avoid some of their disadvantages this way.

### 2.2.1 Dual model

This technique is not really a combination of models but it is closely related. Models that are mutually dual are very often used in a combination with each other. The dual model technique is very frequently used on a class of the so called *permutation problems* [23]. A permutation problem is a problem equivalent to finding a certain subset of permutations of the given set of values. That means the model of a permutation problem has to have the same domain for each variable and the number

of values in this domain has to be the same as the total number of variables. We have already seen an example of the permutation problem before. Remember the n-queens problem from the introduction? We showed it could be modeled with n variables representing rows, each one with n possible values representing column and the solutions could then be thought of as a permutation of columns.

Now we know what the permutation problem is. Let's have some permutation problem modeled as a CSP (we will mark the model $M_p$). That means that $M_p$ contains n variables $x_1, x_2, ..., x_n$, each one with the same domain $\{v_1, v_2, ..., v_n\}$, and some constraints $c_1, c_2, ....$ A *dual model* (we will mark it $M_d$) to model $M_p$ can be created by just flipping the role of values and variables in $M_p$. $M_d$ therefore contains n variables $v_1, v_2, ..., v_n$, each one with the same domain $\{x_1, x_2, ..., x_n\}$ and the appropriate set of constraints [12]. In the model of the permutation problem we are assigning an individual value to the specified variable while in its dual model we are choosing an individual variable to have the specified value. It sounds almost the same and one could say there is no practical difference between the basic and the dual model. Well, this is often not true as we will see in a short while.

When we want to model a problem that could be thought of as a permutation problem, it might be a good idea to try both the models (basic and dual model). There can be a significant difference between efficiencies of the models. Usually it's hard to decide which one of the two models will yield better results. It is important to realize what the solution depends on more. If it's on variables or on values. The better model is then usually the model where the solution depends more on variables than on values.

We will now demonstrate the presented facts on the example of finding a normal magic square of order n. A *magic square of order n* is an arrangement of $n^2$ numbers, usually distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant. We say a magic square of order n is *normal* if it contains all the integers from 1 to $n^2$. An example of the normal magic square of order 4 is presented in Figure 2.6 [2].

Let's now think about modeling the problem of finding a normal magic square of order n as a CSP. The most straightforward way is to use $n^2$ variables representing the values in particular fields of the square. Since we want the square to be normal, each variable will have domain $\{1, 2, 3, ..., n^2\}$. We know the sum of each row, column and diagonal must be the same number. This number called the *magic constant* can be easily computed (since it depends only on $n$) and it is equal to $(n^3 + n)/2$. The constraints would therefore be in the form

$$x_{1,j} + x_{2,j} + ... + x_{n,j} = \frac{n^3 + n}{2}$$

[2]It is the famous Dürer's magic square. It was constructed in 1514 (the two numbers in the middle of the bottom row) by Albrecht Dürer in his copper engraving Melancholia. All rows, all columns and both diagonals sum to 34 as well as all the four quadrants (for example 16,3,5,10), the middle of the square (10,11,6,7) and the corners (16,13,4,1). Also the four outer numbers clockwise and counter-clockwise from the corners (3,8,14,9 and 2,12,15,5), the two sets of four symmetrical numbers (2,8,9,15 and 3,5,12,14) and the sum of the middle two entries of the two outer columns and rows (5,9,8,12 and 3,2,15,14) give the sum of 34. Finally any pair of numbers symmetrically placed about the center of the square sums to 17 (for example 3,14 or 11,6)[30, 15]

| | | | |
|---|---|---|---|
| 16 | 3 | 2 | 13 |
| 5 | 10 | 11 | 8 |
| 9 | 6 | 7 | 12 |
| 4 | 15 | 14 | 1 |

Figure 2.6: An example of the normal magic square of order 4. The sum of each row, column and diagonal is 34.

for rows (where $1 \leq j \leq n$),

$$x_{i,1} + x_{i,2} + ... + x_{i,n} = \frac{n^3 + n}{2}$$

for columns (where $1 \leq i \leq n$),

$$x_{1,1} + x_{2,2} + ... + x_{n,n} = \frac{n^3 + n}{2}$$

for the main diagonal and

$$x_{1,n} + x_{2,n-1} + ... + x_{n,1} = \frac{n^3 + n}{2}$$

for the minor diagonal.

The problem of finding a normal magic square of order n can be thought of as a permutation problem. We just showed the model with $n^2$ variables, each variable with the same domain with $n^2$ values. That is why a dual model to the above presented model has to exist. In the dual model the variables will represent individual *numbers* instead of the fields of the square. Now we will assign a field of the square to particular number instead of assigning the number to the particular field of the square. However, we can see immediately that the dual model will probably not be more efficient than the original model. The definition of the dual model is awkward and the formulation of the constraints very problematic (one could even say it's almost impossible). We have to express the fact that whenever the *values* of a n-element subset of variables form a complete row, column or diagonal, the *variables* contained in this subset must sum to the magic constant. The reader can see immediately that this is not a good approach when we want to construct a good and efficient model. The difference is so obvious here that we won't demonstrate it

practically. We will rather present the second example where the differences between basic and dual models are more visible.

We will model an *assignment problem*. A factory has $n$ workers and $n$ products. There is a profit table given. The profit table contains the profits per time unit for each possible combination of worker and product. The goal is to find an assignment of products to workers with the maximal profit [16].

We can see that the above defined assignment problem is a permutation problem. The most obvious model would contain a variable for every worker (we will mark such a variable $x_i$) with domain $\{1, 2, ..., n\}$ (where the numbers from the domain stand for individual products). An assignment $x_i = j$ means that j-th product was assigned to the i-th worker. We can further assume that the profits are given as a $n \times n$ table. An element $p_{i,j}$ of this table contains the profit made when j-th product is assigned to i-th worker. Now the constraints. First we have to ensure that no product can be assigned to two or more workers. This can be easily done by constraints in the form

$$x_i \neq x_j$$

where $1 \leq i < j \leq n$. The next thing to do is to express the total profit of an assignment. We will mark the total profit $P_n$. The constraint would then be

$$P_n = p_{1,x_1} + p_{2,x_2} + ... + p_{n,x_n}$$

The entire model in SICStus Prolog can be found in the appendix under number 7 and the summary is in Figure 2.7.

**Variables**:
$x_1, x_2, ..., x_n$
**Constraints**:
$x_i \neq x_j$
$P_n = p_{1,x_1} + p_{2,x_2} + ... + p_{n,x_n}$

Figure 2.7: Model **Assignment basic**

Now what about the dual model? The variables and their domains are the same. The difference is in semantics. The variables now stand for products and the values stand for workers. The assignment $x_i = j$ therefore means that j-th worker was assigned to i-th product. We will assume the table of profits stays the same as in the previous example (that means value $p_{i,j}$ still represents the profit made when j-th product is assigned to i-th worker). The constraints will then be in the form

$$x_i \neq x_j$$

(where $1 \leq i < j \leq n$) meaning that no worker can be assigned to two or more products and

$$P_n = p_{x_1,1} + p_{x_2,2} + ... + p_{x_n,n}$$

for the value of the total profit. We can see that the dual model is almost the same as the basic model. The only difference can be found in the semantics of the model

and in the constraint for the total profit. In SICStus Prolog this can be handled by the transposition of the table of profits so the code of the dual model itself is the same as the basic models code which can be found in the appendix under number 7. The summary of the dual model is in Figure 2.8.

**Variables**:
$x_1, x_2, ..., x_n$
**Constraints**:
$x_i \neq x_j$
$P_n = p_{x_1,1} + p_{x_2,2} + ... + p_{x_n,n}$

Figure 2.8: Model **Assignment dual**

The table of profits was in both cases generated according to the formula

$$p_{i,j} = \lfloor 2(sin(i)) + 2 + 5(sin(j)) + 5 \rfloor$$

where $1 \leq i, j \leq n$. The reader may ask why we used such a complicated formula when there are many simpler possibilities. Firstly we don't want the values in rows or in columns to form monotonic sequences. If the values in rows or in columns formed monotonic sequences, the solution of the corresponding assignment problem probably wouldn't be suitable for the purposes of our empirical comparison. For example consider the table of profits generated according to the simple formula $p_{i,j} = i + 2j$. The solution of the corresponding assignment problem would probably be in the form $\{x_1 = 1, x_2 = 2, ..., x_n = n\}$. Such a solution is trivial. It would be found very quickly by most of the solvers and it is practically unusable for the comparison. Formula $p_{i,j} = \lfloor 2(sin(i)) + 2 + 5(sin(j)) + 5 \rfloor$ is basically the sum of two simple periodic functions (in the form $c.sin(x) + c$). The coefficients 2 and 5 were picked so that the number in a field of the table of profits depends more on the column than on the row.

Table 2.5 gives the comparison of the basic and dual models of the assignment problem. The time values are in seconds and the step labeling strategy with the left-most variable selection were used.

| workers / products | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|
| profit | 47 | 58 | 72 | 82 | 85 |
| basic model | 0.050 | 0.441 | 3.555 | 31.916 | 359.457 |
| dual model | 0.030 | 0.230 | 1.492 | 11.407 | 126.982 |

Table 2.5: Assignment problem - empirical comparison of the basic and dual s. Time values are in seconds

The second (dual) model has proved to be slightly better than the basic model in this case. It's important to realize that the presented results depend a lot on the table of profits. If we transposed the table (that means if we used the formula

$p_{i,j} = \lfloor 2(sin(j)) + 2 + 5(sin(i)) + 5 \rfloor$), the results would be exactly opposite and the basic model would be better than the dual model. Also if we used for example the formula $p_{i,j} = i + j$ the results of both the models would be exactly the same. The point is that the basic and the dual model of the permutation problem can have different efficiencies. And since it is not always easy to determine which one of the models will be better for the particular problem, it is often a good idea to try both of them in order to identify and use the most efficient one.

### 2.2.2 Union of models

We have ended the previous section about dual models with the statement that we should always try both models (basic and dual) while modeling a permutation problem in order to identify and use the most efficient one of them. However, wouldn't it be better to use both models *together*? Even if one of the models is significantly worse than the other? When we join the two models together, they may form a very efficient model that will use only the best properties of both participating models.

The principle is simple. Let's have a model of the permutation problem (we will mark it $M_1$). $M_1$ consists of $n$ variables $x_1, x_2, ..., x_n$ and $m$ constraints $c_1, c_2, ..., c_m$. Now let's have a dual model of the same permutation problem marked $M_2$. $M_2$ has $n$ variables $y_1, y_2, ..., y_n$ and $l$ constraints $d_1, d_2, ..., d_l$. Consider a union of $M_1$ and $M_2$. We will mark such a union $M$ and call it a *compound model*. $M$ will contain $2n$ variables $x_1, x_2, ..., x_n, y_1, y_2, ..., y_n$ with appropriate domains and $m + l$ constraints $c_1, c_2, ..., c_m, d_1, d_2, ..., d_l$. Such a model would still be quite inefficient. We more or less doubled both the number of variables and constraints with a little or no gain in the propagation power. Both models contained in $M$ propagate their constraints independently. We have to extend the simple union with the set of $n$ so called *channeling constraints* to obtain a model that is able to use the best properties of $M_1$ and $M_2$ at once [8]. Channeling constraints express relations between corresponding variables of both models contained in $M$ and their function is literally binding these two models together. So an assignment of a value to a variable in one simple model ($M_1$ or $M_2$) would be propagated through channeling constraints to the corresponding variable in the other simple model ($M_2$ or $M_1$) immediately. This way we don't even have to use both sets of variables ($\{x_1, x_2, ..., x_n\}$ and $\{y_1, y_2, ..., y_n\}$) for labeling. We can choose just one of the sets (usually the one belonging to the better of both the models) and channeling constraints provide the propagation of all the assignments to the other set.

We have presented the concept of compound model as a union of the basic and the dual models of the permutation problem. This is the easiest and also the most frequent case. However, we can actually construct compound models from any two different models of any problem. The only condition is the existence of channeling constraints between the corresponding variables of both models. We will now show an example of such a case when other models than a basic and a dual model of the permutation problem are used.

Recall once again the setting of the Golomb ruler problem from section 2.1.1 and consider the following two models of the Golomb ruler problem.

The first model contains $n$ variables $x_1, x_2, ..., x_n$ with integer domains representing the *marks* on a ruler. There are $n-1$ model symmetry breaking constraints in the form

$$x_i < x_{i+1}$$

where $1 \leq i < j \leq n$ (we used the same symmetry breaking constraints in the first model in section 2.1.2). The mutual inequality of individual distances between marks is expressed by constraints in the form

$$(x_j - x_i) \neq (x_l - x_k)$$

where $1 \leq i, j, k, l \leq n$, $i < j$, $i \leq k$, $k < l$ and $i \neq k$ or $j \neq l$. The implementation of such a model in SICStus Prolog can be found in the appendix under number 8 and the summary is in Figure 2.9.

**Variables**:
$x_1, x_2, ..., x_n$
**Constraints**:
$(x_j - x_i) \neq (x_l - x_k)$
$x_i < x_{i+1}$

Figure 2.9: Model **Golomb marks**

The second model contains $n(n-1)/2$ variables $d_{i,j}$ $(1 \leq i < j \leq n)$ representing the *distances* between the marks on a ruler (the variable $d_{i,j}$ represents the distance between i-th and j-th mark). There are inequality constraints in the form

$$d_{i,j} \neq d_{k,l}$$

$(1 \leq i, j, k, l \leq n$, $i < j$, $i \leq k$, $k < l$ and $i \neq k$ or $j \neq l)$ ensuring the distances stay pairwise different, the variable symmetry breaking constraint in the form

$$d_{1,2} < d_{n-1,n}$$

(we used the same constraint in the second model in section 2.1.2) and finally constraints in the form

$$d_{i,k} = d_{i,j} + d_{j,k}$$

(where $1 \leq i < j < k \leq n$) expressing the relations between individual distances. This model can be found in the appendix as the program number 9. The summary of the model is shown in Figure 2.10.

We will now try to combine the two above presented models of the Golomb ruler problem into one. The compound model would contain both sets of constraints and both sets of variables (even if we will choose only one set of variables for labeling). The last thing left to do is to express the relations between the corresponding variables of both the models by channeling constraints. In this case it means addition of $n(n-1)/2$ simple constraints in the form

$$d_{i,j} = x_j - x_i$$

**Variables**:
$d_{1,2}, d_{1,3}, ..., d_{1,n}, d_{2,3}, d_{2,4}, ..., d_{n-1,n}$
**Constraints**:
$d_{i,j} \neq d_{k,l}$
$d_{1,2} < d_{n-1,n}$
$d_{i,k} = d_{i,j} + d_{j,k}$

Figure 2.10: Model **Golomb distances**

**Variables**:
$x_1, x_2, ..., x_n$
$d_{1,2}, d_{1,3}, ..., d_{1,n}, d_{2,3}, d_{2,4}, ..., d_{n-1,n}$
**Constraints**:
$(x_j - x_i) \neq (x_l - x_k)$
$x_i < x_{i+1} \quad d_{i,j} \neq d_{k,l}$
$d_{1,2} < d_{n-1,n}$
$d_{i,k} = d_{i,j} + d_{j,k}$
$d_{i,j} = x_j - x_i$

Figure 2.11: Model **Golomb compound**

(where $1 \leq i < j \leq n$). The complete compound model can be found in the appendix under number 10 and its summary is in figure 2.11.

Table 2.6 shows the comparison of performance of the first simple model, the second simple model and the compound model. The results for compound model are presented for both the possible choices of variable sets used for labeling. The time values are in seconds and the step labeling strategy and the left-most variable selection were used.

| marks | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| length | 17 | 25 | 34 | 44 | 55 |
| Golomb marks | 0.020 | 0.240 | 3.535 | 56.080 | - |
| Golomb distances | 0.000 | 0.040 | 0.370 | 3.635 | 36.763 |
| Golomb compound (variables from the "Golomb marks" model used for labeling) | 0.010 | 0.080 | 0.791 | 10.925 | 108.846 |
| Golomb compound (variables from the "Golomb distances" model used for labeling) | 0.020 | 0.090 | 0.871 | 10.445 | 107.194 |

Table 2.6: Golomb ruler - empirical comparison of the two different models and their compound model. Time values are in seconds

We can see from the comparison that the compound model has not proved to be

the best choice in this case. It was outperformed by the second model. That means the constraints from the first model introduced too much overhead to the compound model and their propagation power was not big enough to compensate this overhead sufficiently. In the next section we will show the way how to overcome this difficulty.

Table 2.7 shows how the choice of labeling strategy influence the final performance of the presented models.

| labeling strategy | step labeling | | enumeration | | bisection | |
|---|---|---|---|---|---|---|
| variable selection | left-most | fail-first | left-most | fail-first | left-most | fail-first |
| Golomb marks, n=8 | 3.535 | 3.585 | 3.545 | 3.565 | 3.495 | 3.535 |
| Golomb distances, n=9 | 3.635 | 4.176 | 4.036 | 4.046 | 3.595 | 3.455 |
| Golomb compound (variables from the "Golomb marks" model used for labeling), n=9 | 10.925 | 11.417 | 11.847 | 12.037 | 10.786 | 10.825 |
| Golomb compound (variables from the "Golomb distances" model used for labeling), n=9 | 10.445 | 9.834 | 11.296 | 10.926 | 10.626 | 8.382 |

Table 2.7: Golomb ruler - dependency between labeling settings and the final performance of a model. Time values are in seconds

We can see again there is nearly no correlation between the choice of the labeling strategy and the performance of the model. The reader may however notice one exception from this fact: there appears to be an apparent decrease of run time in the case of the last model (the compound model with the second set of variables used for labeling) and the last labeling settings configuration (the bisection labeling strategy and the fail first variable selection). This fact is even more interesting considering that in the case of the previous model (which is basically exactly the same model only with a different set of variables used for labeling) this anomaly didn't appear and the measured values were almost the same no matter what labeling settings we used.

We have seen the example of the compound model created from two different models of the Golomb ruler problem. However, in rare cases it may be a good idea to combine more than two models. Channeling constraints though must be defined over each pair of such models so the additional gain is not often worth the expenses. Still there may exist a problem where the combination of three or even more models

can outperform other techniques. Since the principle is exactly the same no matter how many models we are using, we will not discuss unions of more models in depth in this thesis. An example of a compound model composed of three different models can be found for example in [11]

### 2.2.3 Combination of models

We have shown the concept of a compound model in the previous section. We can further extend this concept by omitting some of the inefficient constraints of the compound model. We know that we can use only one set of variables for labeling in the compound model. Any changes in this set are propagated through the channeling constraints to the second set of variables where the second set of constraints is used to propagate these changes even further. The second set of constraints is therefore not necessary for the model and the model will work properly without the second set of constraints too. If the constraints from the second set are all simple and propagating well, they will probably help to prune the search space just like the implied constraints from section 2.1.1 did. However, since the second set of constraints is formed by the complete model of the problem, we cannot guarantee that all the constraints from the second model will be propagating well and will help in the compound model too. Whenever we construct the compound model, it's usually a good idea to think more about individual constraints and to realize which constraints may be difficult to propagate. We can then try to remove such constraints from the compound model in order to improve its efficiency (given the constraints we are trying to delete are implied by other constraints of the compound model, of course). Note that by deleting the constraints from the compound model we are actually approaching the implied constraints concept. We can see the compound model with certain constraints *deleted* as a simple model with certain (implied) constraints *added* instead.

The reader may remember the example of the compound model of the Golomb ruler problem from the previous section. The compound model didn't prove to be the best choice and was outperformed by the second simple model. That means that an addition of the constraints from the first simple model to the second simple model meant decrease of efficiency. Let's think about what happened. Firstly, there are two sets of constraints expressing exactly the same things in the compound model. The fact that the distances must be pairwise different is expressed either by binary constraints from the second simple model ($d_{i,j} \neq d_{k,l}$) or by quaternary constraints from the first model ($(x_j - x_i) \neq (x_l - x_k)$, in both cases $1 \leq i, j, k, l \leq n$, $i < j$, $i \leq k$,$k < l$ and $i \neq k$ or $j \neq l$). One would think the binary constraints will propagate better so we can try to delete the set of quaternary constraints out of the compound model. As we can see from the comparison table bellow, this step really improved the efficiency of the model. However, even with the quaternary constraints deleted the compound model it is not efficient enough to outperform the second simple model from the previous section. We can therefore try to delete more constraints and see if it helps. The next candidate for deletion is the set of difference building constraints in the form $d_{i,k} = d_{i,j} + d_{j,k}$ (where $1 \leq i < j < k \leq n$).

When we look at the compound model implementation in the appendix (program number 10), we can see that the set of difference building constraints is relatively complicated and the relations expressed by these constraints are already implied by the channeling constraints that are much easier. We cannot say for sure the deletion of the difference building constraints will improve the models efficiency. The difference building constraints can still be able to prune the search space sufficiently to pay off (in spite of their rather complicated nature). Nevertheless when we try to delete the difference building constraints, we will see the new model really performs better without them. So the propagation power of the difference building constraints was not big enough.

The comparison of all the models discussed in this section along with the models from the previous section (which are closely related) is given in Table 2.8. Time values are in seconds and the step labeling along with the left most variable selection was used.

| marks | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| length | 17 | 25 | 34 | 44 | 55 | 72 |
| Golomb marks | 0.020 | 0.240 | 3.535 | 56.080 | - | - |
| Golomb distances | 0.000 | 0.040 | 0.370 | 3.635 | 36.763 | 883.581 |
| Golomb compound (variables from the "Golomb marks" model used for labeling, no constraints deleted) | 0.010 | 0.080 | 0.791 | 10.925 | 108.846 | - |
| Golomb compound (variables from the "Golomb marks" model used for labeling, quaternary constraints deleted) | 0.010 | 0.050 | 0.461 | 4.637 | 49.942 | 1026.086 |
| Golomb compound (variables from the "Golomb marks" used for labeling, quaternary and building constraints deleted) | 0.000 | 0.030 | 0.341 | 3.405 | 34.029 | 772.651 |

Table 2.8: Golomb ruler - empirical comparison of the two different models and their compound model with certain constraints deleted. Time values are in seconds

The results of the comparison are more or less in accordance with what we would expect. We have already seen that the compound model of the Golomb ruler problem from the previous section did not outperform the second simple model with variables for distances between marks. It was caused by too many constraints with insufficient propagation power that were present in the compound model. The solution to this problem would than be to delete those of the constraints that prove to be inefficient. We tried to delete one set of such constraints (quaternary inequality constraints) and

it improved the model slightly. But the performance of the new model was still a bit worse than the performance of the original simple model with variables for distances between marks. Another potentially problematic set of constraints was then deleted (the set of building constraints representing relations between individual distances). This deletion also improved the performance of the model a bit and the last model with both the above mentioned sets of constraints deleted finally outperformed the original simple model too.

## 2.3 Other techniques

We have already described several modeling techniques based on addition of new constraints into a model and a combination of more models. However, there are many other techniques that doesn't fit to any of the previously covered categories. This section introduces and describes such techniques.

**Section 2.3.1** is focused on global constraints. There are many types of variable relations that are occurring frequently in various models of various problems. We can then implement such a relation as a special constraint and find an efficient filtering algorithm just for it. That way we can reach even higher levels of consistency (and better propagation too) in reasonable time. The classical example is the alldifferent constraint [20] meaning that any two variables from a given set must have different values. **Section 2.3.2** describes the concept of auxiliary variables. In section 2.1 we have seen that we can often improve a model by addition of new constraints into it. Sometimes we can also improve the model by addition of new variables. The additional variables should allow us to express the constraints of the model more compactly. Constraint propagation would then be faster and more efficient and it would outweigh the overhead introduced into the model by the additional variables.

### 2.3.1 Global constraints

In this section we are going to introduce the very important concept of so called *global constraints*. Global constraints are mostly conjunctions of several constraints representing a certain fact that is better to be looked globally as one complex constraint instead of many primitive ones. The widely used example is a set of inequalities versus the alldifferent global constraint [20, 19].

Consider the CSP with three variables $x_1, x_2, x_3$, with domains $\{1, 2\}$, $\{1, 2\}$ and $\{1, 2, 3\}$ respectively and with three binary constraints $x_1 \neq x_2$, $x_2 \neq x_3$ and $x_3 \neq x_1$. The reader may remember we used similar CSP as a demonstration that arc consistency is not strong enough to find a solution without the search in section 1.3. If we tried to solve the CSP, the solver would probably enforce arc consistency on the three inequality constraints and since the CSP is arc consistent already, the arc consistency enforcing wouldn't change any of the domains. But still we can see that values 1 and 2 clearly cannot be assigned to variable $x_3$ and could have been removed from the domain of $x_3$. So why the solver wouldn't recognize it and wouldn't remove the values? The answer is simple: the solver cannot see the interactions between

individual constraints. It processes the constraints one after one as independent entities. So in our example the solver would consequently process the constraints $x_3 \neq x_1$ and $x_3 \neq x_2$ (which are both arc consistent so no change in domains is needed). But the solver wouldn't see the third constraint $x_1 \neq x_2$ and its impact on the domain of the variable $x_3$ at the same time.

To be able to see it, the solver needs a more global view of the problem. Instead of the three inequality constraints we can express the fact that the variables are mutually different by just one constraint (which can be thought of as a conjunction of the inequality constraints). The only thing left to do is to find some reasonably efficient filtering algorithm for such a global constraint. The task of finding such an algorithm is of course problem dependent and we have to find different filtering algorithms for different global constraints. But still, the effort spent on finding filtering algorithms for global constraints and implementing them in constraint solvers usually pays off, especially in cases that occur frequently in constraint modeling (such as the above presented set of inequalities for which the global constraint named alldifferent exists with the efficient filtering algorithm based on maximal bipartite matching that is able to enforce generalized arc consistency [20]). For more information about alldifferent and other global constraints see for example [4] or [21].

There are many standard global constraints with their own filtering algorithms implemented in constraint solvers. We have already met the alldifferent global constraint. The *cumulative* or the *serialized* constraints are some of the other examples of global constraints (both are used in scheduling and timetabling applications). Since the alldifferent constraint is easy to imagine and understand, we will use it in our practical demonstration once again.

Consider the structure called a Latin square. *A Latin square of size n* is an $n \times n$ table filled with n different symbols (usually numbers) in such a way that each symbol occurs exactly once in each row and exactly once in each column. A Latin square is said to be *normalized* if the first row and the first column of such a square are sorted in ascending order. An example of the normalized Latin square of size 4 is shown on Figure 2.12.

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 2 | 1 | 4 | 3 |
| 3 | 4 | 1 | 2 |
| 4 | 3 | 2 | 1 |

Figure 2.12: normalized Latin square of size 4

Now we try to model the problem of finding a Latin square of size $n$ as a CSP.

We should note that the problem of finding a Latin square of size $n$ is trivial and can be solved in linear time. However there is a (computationally much harder) variant of this problem when some fields of the table are already filled (see for example [13]). Since we are interested in simplicity, we will use the simple case (when no fields are filled) as the example. We can assume the symbols in the table are integer numbers from 1 to $n$. A natural choice would be to use $n^2$ variables $x_{i,j}, 1 \leq i, j \leq n$ representing individual numbers in the table (more precisely, the variable $x_{i,j}$ represents the number in i-th row and j-th column of the table). The domains would than be $\{1, 2, ..., n\}$ for each variable. To express that each number occurs exactly once in each row we have to add $n^2(n-1)/2$ binary constraints in the form

$$x_{i,j} \neq x_{i,k}$$

where $1 \leq i, j, k \leq n, k > j$. Similarly, the constraints expressing that each number occurs exactly once in each column would be in a form

$$x_{j,i} \neq x_{k,i}$$

where $1 \leq i, j, k \leq n, k > j$. Source code of the complete model in SICStus Prolog can be found in the appendix under number 11. The sumary of the model is shown in Figure 2.13.

**Variables**:
$x_{1,1}, x_{1,2}, ..., x_{1,n}, x_{2,1}, ..., x_{n,n}$
**Constraints**:
$x_{i,j} \neq x_{i,k}$
$x_{j,i} \neq x_{k,i}$

Figure 2.13: Model **Latin basic**

Looking at the previous model we can find out there are altogether $2 \times n^2(n-1)/2 = n^2(n-1)$ binary inequality constraints. Since the constraints are only telling us the elements of the same row or column are pairwise different, we can replace the $n^2(n-1)$ inequality constraints with only $2n$ global alldifferent constraints in order to get a more efficient model. Formally the constraints would then be in the form

$$alldifferent(x_{i,1}, x_{i,2}, ..., x_{i,n})$$

for rows and

$$alldifferent(x_{1,i}, x_{2,i}, ..., x_{n,i})$$

for columns (in both cases $1 \leq i \leq n$). This model implemented in SICStus Prolog can be found in the appendix under number 12 and the summary is in Figure 2.14.

We can now proceed to the comparison of both models presented in this section. Table 2.9 summarizes the results. Time values are in seconds and labeling strategy and variable selection settings were left to default again (step labeling and left-most variable selection).

**Variables**:

$x_{1,1}, x_{1,2}, ..., x_{1,n}, x_{2,1}, ..., x_{n,n}$

**Constraints**:

$alldifferent(x_{i,1}, x_{i,2}, ..., x_{i,n})$

$alldifferent(x_{1,i}, x_{2,i}, ..., x_{n,i}$

Figure 2.14: Model **Latin global**

| size | 13 | 14 | 15 | 20 | 40 | 60 | 100 |
|---|---|---|---|---|---|---|---|
| Latin basic | 0.241 | 6.219 | 90.480 | - | - | - | - |
| Latin global | 0.010 | 0.020 | 0.030 | 0.040 | 1.212 | 8.363 | 93.835 |

Table 2.9: Latin square - empirical comparison of models from section 2.3.1. Time values are in seconds

We can see the huge difference again. Application of the set of alldifferent global constraints instead of a handful of basic inequality constraints improved efficiency of the model significantly. Global constraints practically allowed us to solve problems of higher orders.

Table 2.10 gives a review of a dependence between the efficiency of the models and the choice of labeling strategy and variable selection heuristics.

| labeling strategy | step labeling | | enumeration | | bisection | |
|---|---|---|---|---|---|---|
| variable selection | left-most | fail-first | left-most | fail-first | left-most | fail-first |
| Latin basic (n=14) | 6.219 | 0.020 | 6.068 | 0.020 | 6.229 | 0.020 |
| Latin global (n=60) | 8.363 | 12.338 | 8.142 | 12.348 | 8.122 | 15.843 |

Table 2.10: Latin square - dependency between labeling settings and the final performance of a model. Time values are in seconds

The reader can see from the table that the basic model performs substantially better with the fail first variable selection heuristic while the differences between individual labeling strategies are insignificant. However, once we introduce global constraints into the model, the situation will change dramatically and the leftmost variable selection heuristic outperforms the fail first variable selection heuristic. One of the possible explanations is that the propagation of constraints in the basic model is very weak and the solver therefore has to rely more on search. The fail-first variable selection heuristic is more complicated than the left-most variable selection heuristic but it also usually speeds the search more. However, if we use the alldifferent global constraints (that are propagating well), the situation will change. The search is no longer that important and the left-most variable selection heuristic (that is simpler) outperforms the fail-first variable selection heuristic.

### 2.3.2 Auxiliary variables

We should already know how important the form of constraints in a model is. A model where all the constraints are short, simple and compact is usually more efficient than a model with complicated and obscure set of constraints. It also holds that certain types of constraints (for example arithmetic constraints, low arity constraints etc.) propagate better than the rest. These are all reasons for us to try to construct our models with simple and compact sets of constraints. To be able to achieve it, we can try to add some redundant variables into the model. Such variables are often called *auxiliary variables* [10].

Auxiliary variables are variables that are not necessary to model a problem and their assigned value doesn't matter as a solution. The reason why it is worth including them into the model lies in the fact that additional variables can help us to express some complicated constraints more compactly and to speed up the propagation process. We will show this concept on the well known Golomb ruler problem again.

Our first model of the Golomb ruler problem from section 2.1.1 (program number 3 in the appendix) contained $n$ variables representing marks on the ruler. The problem with this approach is that the constraints expressing the fact that the distances between different marks had to be pairwise different are too complicated. The reader may remember that we used the set of quaternary constraints in the form

$$|x_j - x_i| \neq |x_l - x_k|$$

where $1 \leq i, j, k, l \leq n$, $i < j$, $i \leq k$, $k < l$ and $i \neq k$ or $j \neq l$. We might expect such constraints won't be very easy to propagate. Indeed, in section 2.2.3 when we were investigating compound models with certain constraints deleted we showed that the set of quaternary inequality constraints, when deleted, improved the overall efficiency of the compound model.

What if we introduce a set of auxiliary variables representing individual distances between the marks of the ruler? For the distance between marks $x_i$ and $x_j$ we can introduce variable $d_{i,j}$. The constraints expressing the fact the distances between different marks have to be pairwise different would than simplify to the binary constraints in the form

$$d_{i,j} \neq d_{k,l}$$

(where $1 \leq i, j, k, l \leq n$, $i < j$, $i \leq k$, $k < l$ and $i \neq k$ or $j \neq l$). The implementation of such a model can be found in the appendix under number 13 and the summary is in figure 2.15.

**Variables**:
$d_{1,2}, d_{1,3}, ..., d_{1,n}, d_{2,3}, d_{2,4}, ..., d_{n-1,n}$
**Constraints**:
$d_{i,j} \neq d_{k,l}$

Figure 2.15: Model **Golomb auxiliary**

The following table gives the complete comparison of both models of the Golomb ruler problem mentioned in this section. Time values are in seconds and the step labeling strategy along with the left most variable selection were used.

| marks | 4 | 5 | 6 | 7 |
|---|---|---|---|---|
| length | 6 | 11 | 17 | 25 |
| Golomb basic | 0.020 | 1.712 | 149.434 | - |
| Golomb auxiliary | 0.010 | 0.200 | 9.824 | 695.661 |

Table 2.11: Golomb ruler - empirical comparison of models from section 2.3.2. Time values are in seconds

We can see that the simpler inequality constraints expressed through the auxiliary distance variables improved the model significantly. It is true that the model improved with auxiliary variables is not as efficient as other improved models we showed earlier (for example the model with implied constraints or the models with symmetry breaking). But the difference between the basic model and the model with auxiliary variables is still substantial. Also auxiliary variables can be easily combined with other techniques presented earlier to form even more efficient models.

The next table shows the dependency between the choice of the labeling strategy with the variable selection heuristic and the performance of the model.

| labeling strategy | step labeling | | enumeration | | bisection | |
|---|---|---|---|---|---|---|
| variable selection | left-most | fail-first | left-most | fail-first | left-most | fail-first |
| Golomb basic (n=5) | 1.712 | 1.712 | 1.372 | 1.372 | 1.672 | 1.683 |
| Golomb auxiliary (n=6) | 9.824 | 9.894 | 8.943 | 8.983 | 9.583 | 9.664 |

Table 2.12: Golomb ruler - dependency between labeling settings and the final performance of a model. Time values are in seconds

We can see from the table that the best option is to use the enumeration labeling strategy. The performance on the other hand doesn't seem to depend on the choice of the variable selection heuristic.

# Conclusion

In this thesis we tried to introduce the most important techniques used to improve the final efficiency of constraint models. Let's now summarize rules that we should follow while modeling a problem as a CSP.

Firstly - a good and efficient model should contain as much information about the problem as possible. The more information we are able to encode into constraints the better chance of removing more values from domains via constraint propagation we usually have. It is however necessary that this information is expressed by simple, consistent and good propagating constraints. We showed that the addition of new information into an existing model can improve the efficiency of the model dramatically. Implied constraints from section 2.1.1 were used to introduced new information into the model. Compound models from section 2.2.2 and 2.2.3 also introduced more information into the model and proved to be more efficient than simple models. Symmetry breaking from section 2.1.2 is another example of how additional information improved the efficiency of the model.

Secondly - the constraints in the model should be as simple as possible. Complicated high arity constraints are not likely to propagate well. There are of course exceptions (for example global constraints from section 2.3.1) but generally it's a good idea to try the simple constraints first. Auxiliary variables from section 2.3.1 allowed us to express the constraints in a more compact form and we saw that the efficiency of the model was significantly improved

Thirdly - it is also a good idea to keep the number of search variables (variables used for labeling) in a model as small as possible. Additional variables often slow down the search phase because we have to do more choices and the search tree is deeper. In this thesis we didn't show directly what happen when we introduce new search variables into an existing model. However, if we recall the n-queens problem from introduction we can see that the second model (with $n$ variables, each variable with $n$ possible values) performed much better than the first model (with $n$ variables too, but each variable with $n^2$ possible values).

We have also showed comparison of performances of models with various labeling settings (combinations of labeling strategy and variable selection heuristic). We have seen that in most of the cases the choice of labeling settings didn't matter. However, there are cases where the differences between individual labeling settings are substantial (recall for example Table 2.10 in the section about global constraints). If we want to identify the most successful combination of labeling strategy and variable selection heuristic for a particular model, the best approach would probably still be

to try various combinations on problems of smaller sizes and pick the combination showing the best results.

# Bibliography


[1] Ph. Baptiste, L. Peridy, E. Pinson: A Branch and Bound to Minimize the Number of Late Jobs on a Single Machine with Release Time Constraints, European Journal of Operational Research, 2003, p. 1–11

[2] R. Barták: Effective Modeling with Constraints. In Applications of Declarative Programming and Knowledge Management. Springer Verlag, LNCS/LNAI 3392, 2005

[3] R. Barták: On-line guide to constraint programming, http://kti.mff.cuni.cz/ bartak/constraints/index.html

[4] N. Beldiceanu, M. Carlsson, J.-X. Rampon: Global Constraint Catalog, Swedish Institute of Computer Science, technical report no. T2005-08, 2005

[5] C. Bessiére, J. C. Régin: MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems, Proceedings CP'96, 1996, p. 61–75

[6] I. Bratko: Prolog Programming for Artificial Intelligence, Addison-Wesley, 1986

[7] D. Brélaz: New methods to color the vertices of a graph, Communications of the ACM, 22, 1979, p. 251–256

[8] B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, J. C. K. Wu: Increasing constraint propagation by redundant modeling: an experience report, Constraints 4, 1999, p. 167–192

[9] R. Dechter: Constraint Processing, Morgan Kaufmann Publishers (Elsevier Science), 2003

[10] M. Dincbas, H. Simonis, P. van Hentenryck: Solving the car-sequencing problem in constraint logic programming, Y. Kodratoff (Ed.), Proceedings ECAI-88, 1988, p. 290–295

[11] I. Dotú, A. del Val, M. Cebrián: Redundant Modeling for the QuasiGroup Completion Problem, F. Rossi (Ed.), Principles and Practice of Constraint Programming - CP 2003, LNCS 2833, Springer, 2003, p. 288–302

[12] P. A. Geelen: Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems, B. Neumann (Editor), Proceedings ECAI'92, 1992, p. 31–35.

[13] I. P. Gent, T. Walsh: CSPLib: a benchmark library for constraints, Technical report APES-09-1999, 1999

[14] R. M. Haralick, G. L. Elliot: Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence, Vol.14, 1980, p. 263-313

[15] J. A. H. Hunter, J. S. Madachy: Mathematical Diversions, Dover, 1975

[16] K. Marriott, P. Stuckey: Programming with Constraints: An Introduction, The MIT Press, 1998

[17] S. Minton, M. D. Johnston, A. B. Philips, P. Laird: Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method, Proceedings National Conference on Artificial Intelligence (AAAI), 1990, p. 17-24

[18] S. D. Prestwich, J.C. Beck: Exploiting Dominance in Three Symmetric Problems, Proceedings of the 4th International Workshop on Symmetry and Constraint Satisfaction Problems, 2004, p. 63–70

[19] J. F. Puget, M. Leconte: Beyond the black box: Constraints as objects, Logic programming: Proceedings of the 1995 International Symposium, The MIT Press, 1995, p. 513-527

[20] J. C. Régin: A filtering algorithm for constraints of difference in CSPs, Proceedings of the National Conference on Artificial Intelligence, 1994, p. 362-367

[21] J. C. Régin: Global Constraints and Filtering Algorithms, in Constraints and Integer Programming Combined, Kluwer, 2003

[22] Programming Systems Group: SICStus Prolog User's Manual, release 3, 5 edition, 1996.

[23] B. Smith: Modelling for Constraint Programming, CP Summer School, 2005.

[24] B. Smith: The Brélaz heuristic and optimal static orderings, Proceedings CP'99, 1999, p. 405–418

[25] B. Smith, S. A. Grant: Trying harder to fail first, Proceedings ECAI'98, 1998, p. 249–253

[26] B. Smith, K. Stergiou, T. Walsh: Modelling the golomb ruler problem, Proceedings IJCAI'99 workshop on non-binary constraints, 1999

[27] L. Sterling, E. Shapiro: The art of Prolog : advanced programming techniques, 2nd Edition, The MIT Press, 1994

[28] G. Sussman, G. Steele: CONSTRAINTS - a language for expressing almost hierarchical descriptions, Artificial Intelligence, Vol.14, 1980, p. 1-39

[29] E. Tsang: Foundations of Constraint Satisfaction, Academic Press, 1993

[30] E. W. Weisstein: Dürer's Magic Square, MathWorld - A Wolfram Web Resource, http://mathworld.wolfram.com/DuerersMagicSquare.html

# Appendix A

# Source Codes

In this section the complete source codes of all the programs used to generate experimental results are presented. The source codes are written in SICStus Prolog version 4.0.1 [22]. Readers unskilled with Prolog programming can find lots of useful information about this language for example in [27] and [6]. Also the parts of code that are more difficult to understand or that could cause misunderstandings are extra commented. The source codes within this section are ordered according to the order of appearance in the thesis.

## 1. N-queens problem - basic model

This is the implementation of the first basic model from the introduction. We used $n$ variables representing positions of individual queens, each variable had a domain $\{1, 2, ..., n^2\}$.

```
:-use_module(library(clpfd)).

%main procedure used to compute the solution
n_queens_1(Solution, N):-
    length(Solution, N),
    Fields is N*N,
    domain(Solution, 1, Fields),
    get_constraints(Solution, N),
    labeling([],Solution).

%for cycle over the solution
get_constraints([], _).
get_constraints([Xi|Tail], N) :-
    get_constraints_head(Xi, Tail, N),
    get_constraints(Tail, N).

%posts constraints to the given variable
get_constraints_head(_, [], _).
get_constraints_head(Xi,[Xj|Tail],N) :-
```

```
    (Xi-1) mod N #\= (Xj-1) mod N, %not in the same column
    (Xi-1) / N #\= (Xj-1) / N, %not in the same row
    %not on the same diagonal
    abs(((Xi-1)/N)-((Xj-1)/N)) #\= abs(((Xi-1)mod N)-((Xj-1)mod N)),
    get_constraints_head(Xi, Tail, N).
```

## 2. N-queens problem - improved model

This is the implementation of the second model from the introduction. We used $n$ variables representing the number of the column the queen placed in a particular row is located. The variables had domains $\{1, 2, ..., n\}$

```
:-use_module(library(clpfd)).

%main procedure used to compute the solution
n_queens_2(Solution, N):-
    length(Solution, N),
    domain(Solution, 1, N),
    get_constraints(Solution, 1),
    labeling([],Solution).

%for cycle over the solution
get_constraints([], _).
get_constraints([Xi|Tail], I) :-
    J is I+1,
    get_constraints_head(Xi, Tail, I, J),
    get_constraints(Tail, J).

%posts constraints to the given variable
get_constraints_head(_, [], _, _).
get_constraints_head(Xi,[Xj|Tail], I, J) :-
    Xi #\= Xj, %not in the same column
    abs(Xi-Xj) #\= abs(I-J), %not on the same diagonal
    Jn is J+1,
    get_constraints_head(Xi, Tail, I, Jn).
```

## 3. Golomb ruler - basic model

This is the implementation of the basic model of the Golomb ruler problem we showed in section 2.1.1. We used $n$ variables representing the marks on the ruler and the set of quaternary inequality constraints ensuring the distances between marks stayed distinct.

```
:-use_module(library(clpfd)).

%main procedure used to compute the solution
```

```
golomb_basic(Solution, N) :-
    length(Solution, N),
    %domains must be finite so we must state an artificial
    %upper bound on the length of the ruler
    Upper_bound is N*N,
    domain(Solution, 0, Upper_bound),
    get_constraints_Xi(Solution, 1),
    maximum(Length, Solution),
    minimize(labeling([],Solution),Length).

% for each Xi (i in [1,n])
get_constraints_Xi([], _).
get_constraints_Xi([Xi|Xi_tail], I) :-
    In is I+1,
    get_constraints_Xj(Xi, Xi_tail, [Xi|Xi_tail], I, In),
    get_constraints_Xi(Xi_tail, In).

% for each Xj (j in [i+1, n])
get_constraints_Xj(_, [], _, _, _).
get_constraints_Xj(Xi,[Xj|Xj_tail], Xk_list, I, J) :-
    Jn is J+1,
    get_constraints_Xk(Xi, Xj, Xk_list, I, J, I),
    get_constraints_Xj(Xi, Xj_tail, Xk_list, I, Jn).

% for each Xk (k in [i, n])
get_constraints_Xk(_, _, [], _, _, _).
get_constraints_Xk(Xi,Xj,[Xk|Xk_tail], I, J, K) :-
    Kn is K+1,
    get_constraints_Xl(Xi, Xj, Xk, Xk_tail, I, J, K, Kn),
    get_constraints_Xk(Xi, Xj, Xk_tail, I, J, Kn).

% for each Xl (l in [k+1, n])
get_constraints_Xl(_, _, _, [], _, _, _, _).
% the same distances cannot be different -> skip
get_constraints_Xl(Xi, Xj, Xi,[Xj|Tail], I, J, I, J) :-
    Ln is J+1,!,
    get_constraints_Xl(Xi, Xj, Xi, Tail, I, J, I, Ln).
get_constraints_Xl(Xi, Xj, Xk, [Xl|Xl_tail], I, J, K, L) :-
    %post inequality constraint
    abs(Xj-Xi) #\= abs(Xl-Xk),
    Ln is L+1,
    get_constraints_Xl(Xi, Xj, Xk, Xl_tail, I, J, K, Ln).
```

## 4. Golomb ruler - model with implied constraints

This is the implementation of the model of the Golomb ruler problem enhanced with implied constraints from section 2.1.1. We used $n$ ordered variables representing the marks on the ruler, the set of quaternary inequality constraints ensuring the distances between marks stayed distinct and implied constraints representing lower and upper bounds of individual distances.

```
:-use_module(library(clpfd)).

%main procedure used to compute the solution
golomb_implied(Solution, N) :-
    length(Solution, N),
    %domains must be finite so we must state an artificial
    %upper bound on the length of the ruler
    Upper_bound is N*N,
    Solution = [0|Solution_tail],%first mark must be zero
    get_marks(0, Solution_tail, Upper_bound),%get ordered marks
    maximum(Length, Solution),%get the length of the ruler
    get_constraints_Xi(Solution, 1, N, Length),
    minimize(labeling([],Solution),Length).

%posts constraints ensuring that the marks will be ordered
get_marks(_, [], _).
get_marks(PrevX, [Xi|Tail], Upper_bound) :-
    Xi #< Upper_bound,
    Xi #> PrevX,
    get_marks(Xi, Tail, Upper_bound).

% for each Xi (i in [1,n])
get_constraints_Xi([], _, _, _).
get_constraints_Xi([Xi|Xi_tail],I, N, Length) :-
    In is I+1,
    get_constraints_Xj(Xi, Xi_tail, [Xi|Xi_tail], I, In, N, Length),
    get_constraints_Xi(Xi_tail, In, N, Length).

% for each Xj (j in [i+1, n])
get_constraints_Xj(_, [], _, _, _, _, _).
get_constraints_Xj(Xi,[Xj|Xj_tail], Xk_list, I, J, N, Length) :-
    Jn is J+1,
    get_constraints_Xk(Xi, Xj, Xk_list, I, J, I, N, Length),
    get_constraints_Xj(Xi, Xj_tail, Xk_list, I, Jn, N, Length).

% for each Xk (k in [i, n])
get_constraints_Xk(_, _, [], _, _, _, _, _).
get_constraints_Xk(Xi, Xj, [Xk|Xk_tail], I, J, K, N, Length) :-
```

```
    Kn is K+1,
    get_constraints_Xl(Xi, Xj, Xk, Xk_tail, I, J, K, Kn, N, Length),
    get_constraints_Xk(Xi, Xj, Xk_tail, I, J, Kn, N, Length).

% for each Xl (l in [k+1, n])
get_constraints_Xl(_, _, _, [], _, _, _, _, _, _).
% the same distances cannot be different -> skip
get_constraints_Xl(Xi, Xj, Xi,[Xj|Tail], I, J, I, J, N, Length) :-
    L is J+1,!,
    get_constraints_Xl(Xi, Xj, Xi, Tail, I, J, I, L, N, Length).
get_constraints_Xl(Xi, Xj, Xk, [Xl|Xl_tail], I, J, K,L,N,Length):-
    %post inequality constraint
    abs(Xj-Xi) #\= abs(Xl-Xk),

    %lower bound implied constraint
    Lower_bound is integer(abs((J-I)*(J-I+1)/2)),
    abs(Xj-Xi) #>= Lower_bound,

    %upper bound implied constraint
    Upper_bound_part is integer(abs((N-1-J+I)*(N-J+I)/2)),
        abs(Xj-Xi) #=< Length-Upper_bound_part,

    Ln is L+1,
    get_constraints_Xl(Xi, Xj, Xk, Xl_tail, I, J, K, Ln, N, Length).
```

## 5. Golomb ruler - model with model symmetry broken

This is the implementation of the model of the Golomb ruler problem with model symmetry broken from section 2.1.2. We used $n$ variables representing the marks on the ruler that were sorted to break the symmetry and the set of quaternary inequality constraints ensuring the distances between marks stayed distinct.

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).

%main procedure used to compute the solution
golomb_symmetry_1(Solution, N) :-
    length(Solution, N),
    %domains must be finite so we must state an artificial
    %upper bound on the length of the ruler
    Upper_bound is N*N,
    Solution = [0|Solution_tail],%first mark must be zero
    get_marks(0, Solution_tail, Upper_bound),%get ordered marks
    get_constraints_Xi(Solution, 1),
    last(Solution, Length),
```

```
    minimize(labeling([],Solution),Length).

%posts constraints ensuring that the marks will be ordered
get_marks(_, [], _).
get_marks(PrevX, [Xi|Tail], Upper_bound) :-
    Xi #< Upper_bound,
    Xi #> PrevX,
    get_marks(Xi, Tail, Upper_bound).

% for each Xi (i in [1,n])
get_constraints_Xi([], _).
get_constraints_Xi([Xi|Xi_tail], I) :-
    In is I+1,
    get_constraints_Xj(Xi, Xi_tail, [Xi|Xi_tail], I, In),
    get_constraints_Xi(Xi_tail, In).

% for each Xj (j in [i+1, n])
get_constraints_Xj(_, [], _, _, _).
get_constraints_Xj(Xi,[Xj|Xj_tail], Xk_list, I, J) :-
    Jn is J+1,
    get_constraints_Xk(Xi, Xj, Xk_list, I, J, I),
    get_constraints_Xj(Xi, Xj_tail, Xk_list, I, Jn).

% for each Xk (k in [i, n])
get_constraints_Xk(_, _, [], _, _, _).
get_constraints_Xk(Xi,Xj,[Xk|Xk_tail], I, J, K) :-
    Kn is K+1,
    get_constraints_Xl(Xi, Xj, Xk, Xk_tail, I, J, K, Kn),
    get_constraints_Xk(Xi, Xj, Xk_tail, I, J, Kn).

% for each Xl (l in [k+1, n])
get_constraints_Xl(_, _, _, [], _, _, _, _).
% the same distances cannot be different -> skip
get_constraints_Xl(Xi, Xj, Xi,[Xj|Tail], I, J, I, J) :-
    L is J+1,!,
    get_constraints_Xl(Xi, Xj, Xi, Tail, I, J, I, L).
get_constraints_Xl(Xi, Xj, Xk, [Xl|Xl_tail], I, J, K, L) :-
    %post inequality constraint
    Xj-Xi #\= Xl-Xk,
    Ln is L+1,
    get_constraints_Xl(Xi, Xj, Xk, Xl_tail, I, J, K, Ln).
```

## 6. Golomb ruler - model with both symmetries broken

This is the implementation of the model of the Golomb ruler problem from section 2.1.2 where both the types of symmetry were broken. We used $n$ variables representing the marks on the ruler that were sorted to break the model symmetry, additional constraint to break the variable symmetry and the set of quaternary inequality constraints ensuring the distances between marks stayed distinct.

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).

%main procedure used to compute the solution
golomb_symmetry_2(Solution, N) :-
    length(Solution, N),
    %domains must be finite so we must state an artificial
    %upper bound on the length of the ruler
    Upper_bound is N*N,
    Solution = [0|Solution_tail],%first mark must be zero
    get_marks(0, Solution_tail, Upper_bound),%get ordered marks
    Solution = [X1,X2|_],
    last(Solution, Xn),
    Prev is N-1,
    element(Prev, Solution, Xprev),
    X2-X1 #< Xn - Xprev,%variable symmetry breaking constraint
    get_constraints_Xi(Solution, 1),
    last(Solution, Length),
    minimize(labeling([],Solution),Length).

%posts constraints ensuring that the marks will be ordered
get_marks(_, [], _).
get_marks(PrevX, [Xi|Tail], Upper_bound) :-
    Xi #< Upper_bound,
    Xi #> PrevX,
    get_marks(Xi, Tail, Upper_bound).

% for each Xi (i in [1,n])
get_constraints_Xi([], _).
get_constraints_Xi([Xi|Xi_tail], I) :-
    In is I+1,
    get_constraints_Xj(Xi, Xi_tail, [Xi|Xi_tail], I, In),
    get_constraints_Xi(Xi_tail, In).

% for each Xj (j in [i+1, n])
get_constraints_Xj(_, [], _, _, _).
get_constraints_Xj(Xi,[Xj|Xj_tail], Xk_list, I, J) :-
    Jn is J+1,
```

```
    get_constraints_Xk(Xi, Xj, Xk_list, I, J, I),
    get_constraints_Xj(Xi, Xj_tail, Xk_list, I, Jn).

% for each Xk (k in [i, n])
get_constraints_Xk(_, _, [], _, _, _).
get_constraints_Xk(Xi,Xj,[Xk|Xk_tail], I, J, K) :-
    Kn is K+1,
    get_constraints_Xl(Xi, Xj, Xk, Xk_tail, I, J, K, Kn),
    get_constraints_Xk(Xi, Xj, Xk_tail, I, J, Kn).

% for each Xl (l in [k+1, n])
get_constraints_Xl(_, _, _, [], _, _, _, _).
% the same distances cannot be different -> skip
get_constraints_Xl(Xi, Xj, Xi,[Xj|Tail], I, J, I, J) :-
    L is J+1,!,
    get_constraints_Xl(Xi, Xj, Xi, Tail, I, J, I, L).
get_constraints_Xl(Xi, Xj, Xk, [Xl|Xl_tail], I, J, K, L) :-
    %post inequality constraint
    Xj-Xi #\= Xl-Xk,
    Ln is L+1,
    get_constraints_Xl(Xi, Xj, Xk, Xl_tail, I, J, K, Ln).
```

## 7. Assignment problem - the basic and the dual model

This is the implementation of the model of the assignment problem from section 2.2.1. We used $n$ variables representing workers in the basic model and products in the dual model. We also used basic inequality constraints ensuring mutually unique assignments and the constraints for an expression of the total profit of an assignment. The basic and the dual models of the assignment problem differs only in the approach to the table of profits and the code of the model itself stays the same.

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).

%main procedure used to compute the solution
assignment(Solution, Profit_table, N):-
    length(Solution, N),
    domain(Solution, 1, N),
    get_constraints(Solution),
    %compute the total profit of an assignment in Solution
    get_profit(Solution, Profit, Profit_table),
    maximize(labeling([],Solution), Profit).

%for each Xi
get_constraints([]).
```

```
get_constraints([Xi|Tail]) :-
    get_constraints_head(Xi, Tail),
    get_constraints(Tail).

%posts inequality constraints
get_constraints_head(_, []).
get_constraints_head(Xi, [Xj|Tail]):-
    Xi #\= Xj, %all the values of Solution must be mutually different
    get_constraints_head(Xi, Tail).

%posts the profit constraint
get_profit([], 0, _).
get_profit([Xi|Tail],Profit,[Profit_table_row|Profit_table_tail]):-
    %get a profit of single worker
    element(Xi, Profit_table_row, Profit_part),
    get_profit(Tail, Profit_rest, Profit_table_tail),
    %add a profit of single worker to the total sum
    Profit #= Profit_part+Profit_rest.
```

## 8. Golomb ruler - the first model from section 2.2.2

This is the implementation of the first model of the Golomb ruler problem from section 2.2.2. We used $n$ variables representing marks with model symmetry breaking constraints.

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).

%main procedure used to compute the solution
golomb_marks(Solution, N) :-
    length(Solution, N),
    %domains must be finite so we must state an artificial
    %upper bound on the length of the ruler
    Upper_bound is N*N,
    Solution = [0|Solution_tail],%first mark must be zero
    get_marks(0, Solution_tail, Upper_bound),%get ordered marks
    get_constraints_Xi(Solution, 1),
    last(Solution, Length),
    minimize(labeling([ff, step],Solution),Length).

%posts constraints ensuring that the marks will be ordered
get_marks(_, [], _).
get_marks(PrevX, [Xi|Tail], Upper_bound) :-
    Xi #< Upper_bound,
    Xi #> PrevX,
```

```
    get_marks(Xi, Tail, Upper_bound).

% for each Xi (i in [1,n])
get_constraints_Xi([], _).
get_constraints_Xi([Xi|Xi_tail], I) :-
    In is I+1,
    get_constraints_Xj(Xi, Xi_tail, [Xi|Xi_tail], I, In),
    get_constraints_Xi(Xi_tail, In).

% for each Xj (j in [i+1, n])
get_constraints_Xj(_, [], _, _, _).
get_constraints_Xj(Xi,[Xj|Xj_tail], Xk_list, I, J) :-
    Jn is J+1,
    get_constraints_Xk(Xi, Xj, Xk_list, I, J, I),
    get_constraints_Xj(Xi, Xj_tail, Xk_list, I, Jn).

% for each Xk (k in [i, n])
get_constraints_Xk(_, _, [], _, _, _).
get_constraints_Xk(Xi,Xj,[Xk|Xk_tail], I, J, K) :-
    Kn is K+1,
    get_constraints_Xl(Xi, Xj, Xk, Xk_tail, I, J, K, Kn),
    get_constraints_Xk(Xi, Xj, Xk_tail, I, J, Kn).

% for each Xl (l in [k+1, n])
get_constraints_Xl(_, _, _, [], _, _, _, _).
% the same distances cannot be different -> skip
get_constraints_Xl(Xi, Xj, Xi,[Xj|Tail], I, J, I, J) :-
    L is J+1,!,
    get_constraints_Xl(Xi, Xj, Xi, Tail, I, J, I, L).
get_constraints_Xl(Xi, Xj, Xk, [Xl|Xl_tail], I, J, K, L) :-
    %post inequality constraint
    Xj-Xi #\= Xl-Xk,
    Ln is L+1,
    get_constraints_Xl(Xi, Xj, Xk, Xl_tail, I, J, K, Ln).
```

## 9. Golomb ruler - the second model from section 2.2.2

This is the implementation of the second model of the Golomb ruler problem from section 2.2.2. We used $n(n-1)/2$ variables representing distances between marks. The requirement that the differences are pairwise different was expressed by the basic set of inequality constraints and the variable symmetry breaking constraint was added.

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).
```

```
%main procedure used to compute the solution
golomb_distances(Distances, N) :-
    Distances_count is integer(N*(N-1)/2),
    length(Distances, Distances_count),
    %domains must be finite so we must state an artificial
    %upper bound on the length of the ruler
    Upper_bound is N*N,
    domain(Distances, 1, Upper_bound),
    get_inequality_constraints(Distances),
    EndI is N-1,
    get_building_constraints_I(1, EndI, N, Distances),
    Distances = [First|_],
    last(Distances, Last),
    First #< Last,%variable symmetry breaking constraint
    Length_index is N-1,
    element(Length_index, Distances, Length),
    minimize(labeling([],Distances),Length).

%for each I in (1, N-2)
get_building_constraints_I(EndI, EndI, _, _).
get_building_constraints_I(I, EndI, N, Distances) :-
    I < EndI,
    FirstK is I+2,
    EndK is N+1,
    get_building_constraints_K(FirstK, EndK ,I, N, Distances),
    In is I+1,
    get_building_constraints_I(In, EndI, N, Distances).

%for each K in (I+2, N)
get_building_constraints_K(EndK, EndK, _, _, _).
get_building_constraints_K(K, EndK, I, N, Distances) :-
    K < EndK,
    FirstJ is I+1,
    EndJ is K,
    get_building_constraints_J(FirstJ, EndJ, I, K, N, Distances),
    Kn is K+1,
    get_building_constraints_K(Kn, EndK, I, N, Distances).

%for each J in (I+1,K-1)
get_building_constraints_J(EndJ, EndJ, _, _, _, _).
get_building_constraints_J(J, EndJ, I, K, N, Distances) :-
    J < EndJ,
    %get indexes of distances
    get_index(IndexIJ, I, J, N),
```

```
    get_index(IndexJK, J, K, N),
    get_index(IndexIK, I, K, N),
    %get distances
    element(IndexIJ, Distances, Dij),
    element(IndexJK, Distances, Djk),
    element(IndexIK, Distances, Dik),
    %post constraint
    Dik #= Dij+Djk,
    Jn is J+1,
    get_building_constraints_J(Jn, EndJ, I, K, N, Distances).

%help procedure computing index of the distance
get_index(Index, I, J, N) :-
    Index is integer(N*(I-1) - (I*(I+1)/2) + J).

%for each distance
get_inequality_constraints([]).
get_inequality_constraints([Head|Tail]) :-
    get_inequality_constraints_head(Head, Tail),
    get_inequality_constraints(Tail).

%posts inequality constraints to a given distance
get_inequality_constraints_head(_, []).
get_inequality_constraints_head(DistanceI, [DistanceJ|Tail]) :-
    DistanceI #\= DistanceJ, %distances are distinct
    get_inequality_constraints_head(DistanceI, Tail).
```

## 10. Golomb ruler - the compound model

This is the implementation of the compound model of the Golomb ruler problem from section 2.2.2. We used constraints from both the models presented earlier, set of channeling constraints to bind the two models together and only one set of variables for labeling.

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).

%main procedure used to compute the solution
golomb_compound(Solution, Distances, N) :-
    %set lengths
    length(Solution, N),
    Distances_count is integer(N*(N-1)/2),
    length(Distances, Distances_count),

    %set domains
```

```
    %domains must be finite so we must state an artificial
    %upper bound on the length of the ruler
    Upper_bound is N*N,
    Solution = [0|Solution_tail],%first mark must be zero
    get_marks(0, Solution_tail, Upper_bound),%get ordered marks
    domain(Distances, 1, Upper_bound),

    %post constraints
    Distances = [First|_],
    last(Distances, Last),
    First #< Last,%variable symmetry breaking constraint
    get_constraints_Xi(Solution, 1),
    get_inequality_constraints(Distances),
    EndI is N-1,
    get_building_constraints_I(1, EndI, N, Distances),
    get_channeling_constraints(Solution, Distances),

    %labeling according to the first model
    last(Solution, Length),
    minimize(labeling([],Solution),Length).

    %labeling according to the second model
    %Length_index is N-1,
    %element(Length_index, Distances, Length),
    %minimize(labeling([],Distances),Length).

%%%                          FIRST MODEL                          %%%
%posts constraints ensuring that the marks will be ordered
get_marks(_, [], _).
get_marks(PrevX, [Xi|Tail], Upper_bound) :-
    Xi #< Upper_bound,
    Xi #> PrevX,
    get_marks(Xi, Tail, Upper_bound).

% for each Xi (i in [1,n])
get_constraints_Xi([], _).
get_constraints_Xi([Xi|Xi_tail], I) :-
    In is I+1,
    get_constraints_Xj(Xi, Xi_tail, [Xi|Xi_tail], I, In),
    get_constraints_Xi(Xi_tail, In).

% for each Xj (j in [i+1, n])
get_constraints_Xj(_, [], _, _, _).
get_constraints_Xj(Xi,[Xj|Xj_tail], Xk_list, I, J) :-
    Jn is J+1,
```

```
    get_constraints_Xk(Xi, Xj, Xk_list, I, J, I),
    get_constraints_Xj(Xi, Xj_tail, Xk_list, I, Jn).

% for each Xk (k in [i, n])
get_constraints_Xk(_, _, [], _, _, _).
get_constraints_Xk(Xi,Xj,[Xk|Xk_tail], I, J, K) :-
    Kn is K+1,
    get_constraints_Xl(Xi, Xj, Xk, Xk_tail, I, J, K, Kn),
    get_constraints_Xk(Xi, Xj, Xk_tail, I, J, Kn).

% for each Xl (l in [k+1, n])
get_constraints_Xl(_, _, _, [], _, _, _, _).
% the same distances cannot be different -> skip
get_constraints_Xl(Xi, Xj, Xi,[Xj|Tail], I, J, I, J) :-
    L is J+1,!,
    get_constraints_Xl(Xi, Xj, Xi, Tail, I, J, I, L).
get_constraints_Xl(Xi, Xj, Xk, [Xl|Xl_tail], I, J, K, L) :-
    %post inequality constraint
    Xj-Xi #\= Xl-Xk,
    Ln is L+1,
    get_constraints_Xl(Xi, Xj, Xk, Xl_tail, I, J, K, Ln).

%%%                          SECOND MODEL                          %%%
%for each I in (1, N-2)
get_building_constraints_I(EndI, EndI, _, _).
get_building_constraints_I(I, EndI, N, Distances) :-
    I < EndI,
    FirstK is I+2,
    EndK is N+1,
    get_building_constraints_K(FirstK, EndK ,I, N, Distances),
    In is I+1,
    get_building_constraints_I(In, EndI, N, Distances).

%for each K in (I+2, N)
get_building_constraints_K(EndK, EndK, _, _, _).
get_building_constraints_K(K, EndK, I, N, Distances) :-
    K < EndK,
    FirstJ is I+1,
    EndJ is K,
    get_building_constraints_J(FirstJ, EndJ, I, K, N, Distances),
    Kn is K+1,
    get_building_constraints_K(Kn, EndK, I, N, Distances).

%for each J in (I+1,K-1)
get_building_constraints_J(EndJ, EndJ, _, _, _, _).
```

```
get_building_constraints_J(J, EndJ, I, K, N, Distances) :-
    J < EndJ,
    %get indexes of distances
    get_index(IndexIJ, I, J, N),
    get_index(IndexJK, J, K, N),
    get_index(IndexIK, I, K, N),
    %get distances
    element(IndexIJ, Distances, Dij),
    element(IndexJK, Distances, Djk),
    element(IndexIK, Distances, Dik),
    %post constraint
    Dik #= Dij+Djk,
    Jn is J+1,
    get_building_constraints_J(Jn, EndJ, I, K, N, Distances).

%help procedure computing index of the distance
get_index(Index, I, J, N) :-
    Index is integer(N*(I-1) - (I*(I+1)/2) + J).

%for each distance
get_inequality_constraints([]).
get_inequality_constraints([Head|Tail]) :-
    get_inequality_constraints_head(Head, Tail),
    get_inequality_constraints(Tail).

%posts inequality constraints to a given distance
get_inequality_constraints_head(_, []).
get_inequality_constraints_head(DistanceI, [DistanceJ|Tail]) :-
    DistanceI #\= DistanceJ, %distances are distinct
    get_inequality_constraints_head(DistanceI, Tail).

%%%                    CHANNELING CONSTRAINTS                    %%%
%posts channeling constraints
get_channeling_constraints([], _).
get_channeling_constraints([Xi|Tail], Distances) :-
    get_channeling_constraints_head(Xi, Tail, Distances, Dist_rest),
    get_channeling_constraints(Tail, Dist_rest).

%posts channeling constraints for a given initial mark
get_channeling_constraints_head(_, [], Dist_tail, Dist_tail).
get_channeling_constraints_head(Xi,[Xj|Tail],[Dij|Dist_tail],Dist_rest):-
    Dij #= Xj-Xi,
    get_channeling_constraints_head(Xi, Tail, Dist_tail, Dist_rest).
```

## 11. Latin square - the basic model

This is the implementation of the basic model of the problem of finding a latin square of size $n$ from section 2.3.1. We used $n^2$ variables representing the fields of the square and basic inequality constraints representing the fact that each number occurs exactly once in each row and exactly once in each column.

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).

%main procedure used to compute the solution
latin_basic(Solution, N):-
    length(Solution, N),
    get_rows(Solution, N),
    get_constraints_rows(Solution),
    get_constraints_columns(Solution, N),
    flatten(Solution, Solution_flattened),
    labeling([], Solution_flattened).

%creates the matrix
get_rows([], _).
get_rows([Row|Tail], N) :-
    length(Row, N),
    domain(Row, 1, N),
    get_rows(Tail, N).

%for each row
get_constraints_rows([]).
get_constraints_rows([Row|Tail]) :-
    get_constraints_one_row(Row),
    get_constraints_rows(Tail).

%posts inequality constraints to given row
get_constraints_one_row([]).
get_constraints_one_row([X|Tail]) :-
    get_constraints_one_row_head(X, Tail),
    get_constraints_one_row(Tail).

%posts inequality constraints to given variable
get_constraints_one_row_head(_, []).
get_constraints_one_row_head(X, [Y|Tail]) :-
    X #\= Y,%post constraint
    get_constraints_one_row_head(X, Tail).

%for i from n to 1
get_constraints_columns(_, 0).
```

```
get_constraints_columns(Solution,I):-
    get_column(Column, Solution, I),
    get_constraints_one_column(Column),
    In is I-1,
    get_constraints_columns(Solution, In).

%separates one column from the matrix
get_column([], [], _).
get_column([X|Rest], [Row|Tail], I) :-
    nth1(I, Row, X),
    get_column(Rest, Tail, I).

%posts inequality constraints to given column
get_constraints_one_column([]).
get_constraints_one_column([X|Tail]) :-
    get_constraints_one_column_head(X, Tail),
    get_constraints_one_column(Tail).

%posts inequality constraints to given variable
get_constraints_one_column_head(_, []).
get_constraints_one_column_head(X, [Y|Tail]) :-
    X #\= Y,%post constraint
    get_constraints_one_column_head(X, Tail).

%flattens a two dimensional array to a one dimensional array
flatten([], []).
flatten([Row|Tail], Result) :-
    append(Row, Result_old, Result),
    flatten(Tail, Result_old).
```

## 12. Latin square - the model with global constraints

This is the implementation of the second model of the problem of finding a latin square of size $n$ from section 2.3.1. We used $n^2$ variables representing the fields of the square. The basic inequality constraints representing the fact that each number occurs exactly once in each row and exactly once in each column were replaced by the set of alldifferent global constraints.

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).

%main procedure used to compute the solution
latin_global(Solution, N):-
    length(Solution, N),
    get_rows(Solution, N),
```

```
    get_constraints_rows(Solution),
    get_constraints_columns(Solution, N),
    flatten(Solution, Solution_flattened),
    labeling([], Solution_flattened).

%creates the matrix
get_rows([], _).
get_rows([Row|Tail], N) :-
    length(Row, N),
    domain(Row, 1, N),
    get_rows(Tail, N).

%posts constraints for rows
get_constraints_rows([]).
get_constraints_rows([Row|Tail]) :-
    all_distinct(Row),
    get_constraints_rows(Tail).

%posts constraints for columns
get_constraints_columns(_, 0).
get_constraints_columns(Solution,I):-
    get_column(Column, Solution, I),
    all_distinct(Column),
    In is I-1,
    get_constraints_columns(Solution, In).

%separates one column from the matrix
get_column([], [], _).
get_column([X|Rest], [Row|Tail], I) :-
    nth1(I, Row, X),
    get_column(Rest, Tail, I).

%flattens a two dimensional array to a one dimensional array
flatten([], []).
flatten([Row|Tail], Result) :-
    append(Row, Result_old, Result),
    flatten(Tail, Result_old).
```

## 13. Golomb ruler - the model with auxiliary variables

This is the implementation of the second model of the Golomb ruler problem from section 2.3.2. We used $n$ variables representing marks on the ruler and $n(n-1)/2$ auxiliary variables representing individual distances between marks.

```
:-use_module(library(clpfd)).
```

```
%main procedure used to compute the solution
golomb_auxiliary(Solution, N) :-
    length(Solution, N),
    %domains must be finite so we must state an artificial
    %upper bound on the length of the ruler
    Upper_bound is N*N,
    domain(Solution, 0, Upper_bound),
    get_distances(Solution, Distances),%get auxiliary variables
    get_constraints(Distances),
    maximum(Length, Solution),
    minimize(labeling([ff, step],Solution),Length).

%for each solution
get_distances([], []).
get_distances([Head|Tail], Distances) :-
    get_distances_head(Head, Tail, Distances_head),
    append(Distances_head, Distances_tail, Distances),
    get_distances(Tail, Distances_tail).

%posts constraints between marks and distances
get_distances_head(_, [], []).
get_distances_head(Xi,[Xj|Tail],[Distance|Distances]) :-
    Distance #= abs(Xj-Xi),
    get_distances_head(Xi, Tail, Distances).

%for each distance
get_constraints([]).
get_constraints([Head|Tail]) :-
    get_constraints_head(Head, Tail),
    get_constraints(Tail).

%posts inequality constraints (to given distance)
get_constraints_head(_, []).
get_constraints_head(DistanceI,[DistanceJ|Tail]) :-
    DistanceI #\= DistanceJ, %distances are distinct
    get_constraints_head(DistanceI, Tail).
```