Effective Modeling with Constraints

Roman Barták

Charles University, Faculty of Mathematics and Physics
Institute for Theoretical Computer Science
Malostranské nám. 2/25, Prague, Czech Republic
roman.bartak@mff.cuni.cz

Abstract. Constraint programming provides a declarative approach to solving combinatorial (optimization) problems. The user just states the problem as a constraint satisfaction problem (CSP) and a generic solver finds a solution without additional programming. However, in practice, the situation is more complicated because there usually exist several ways how to model the problem as a CSP, that is using variables, their domains, and constraints. In fact, different constraint models may lead to significantly different running times of the solver so constraint modeling is a crucial part of problem solving. This paper describes some known approaches to efficient modeling with constraints in a tutorial-like form. The primary audience is practitioners, especially in logic programming, that would like to use constraints in their projects but do not have yet deep knowledge of constraint satisfaction techniques.

1 Introduction

"Constraint programming (CP) represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [4] This nice quotation might convince a reader that designing a constraint model for a particular combinatorial problem is an easy and straightforward task and that everything stated as a constraint satisfaction problem can be solved efficiently by the underlying constraint solver. This holds for simple or small problems but as soon as the problems are more complex, the role of constraint modeling is becoming more and more important. The reason is that different constraint models of the same problem may lead to very different solving times. Unfortunately, there does not exist (yet) any guide that can steer the user how to design an efficiently solvable model. This "feature" of constraint technology might be a bit depressing for novices. Nevertheless, there are many rules of thumb about designing efficient constraint models. We also believe that it is important to be aware of the insides of constraint satisfaction to understand better the behavior of the solvers and, as a consequence, to design models that exploit the power of the solvers.

The goal of this paper is to introduce the basic terminology useful for reading further CP texts and to give an overview of modeling techniques used to state problems as constraint satisfaction problems. The emphasis is put to novice users of the constraint technology, in particular to the current users of logic programming

technology that can be naturally extended by constraints. Respecting what has been said above we will first survey the mainstream constraint satisfaction technology. In the main part of the paper we will demonstrate some modeling techniques namely symmetry breaking, dual models, and redundant constraints, using practical examples.

2 Constraint Satisfaction at Glance

Constraint programming is a framework for solving combinatorial (optimization) problems. The basic idea is to model the problem as a set of variables with domains (the values for the variables) and a set of constraints restricting the possible combinations of variables' values (Figure 1). Usually, the domains are finite and we are speaking about *constraint satisfaction problems* (CSP)[1]. The task is to find an assignment of all the variables satisfying all the constraints – a so called complete *feasible assignment*. Sometimes, there is also an objective function defined over the problem variables. Then the task is to find a complete feasible assignment minimizing or maximizing the objective function – an *optimal assignment*. Such problems are called *constraint optimization problems* (COP).

Note that modeling problems as a CSP or a COP is natural because constraints can describe arbitrary relations and various constraints can be easily combined within a single system. Opposite to frameworks like linear and integer programming, constraints in a CSP/COP are not restricted to linear equalities and inequalities. The constraint can express arbitrary mathematical or logical formula, like ($x^2<y \vee x=y$). The constraint could even be an arbitrary relation that can be hardly expressed in an intentional form, for example restricted resource state transitions in scheduling applications. Then, a table is used to describe tuples satisfying the constraint.

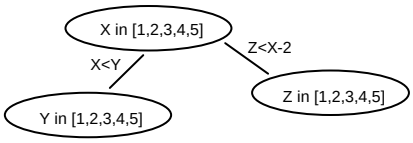


Fig. 1. A CSP consists of variables (X,Y,Z), their domains (in this case identical [1,2,3,4,5]), and constraints (X<Y, Y<X-2). It can be represented as a constraint (hyper) graph

Solving problems using constraints typically involves three steps. First, the problem is expressed in terms of variables, their domains, and constraints – this is called *constraint modeling*. Second, the constraint model is passed to a generic constraint solver that, hopefully, finds a feasible (or optimal) assignment – *constraint satisfactio*n. Finally, this assignment is interpreted as a solution of the original problem. To understand better the important role of constraint modeling let us first look inside the mainstream technique of constraint satisfaction that is a combination of depth-first search (enumeration) with constraint propagation. Despite the fact that

[1] There also exist constraint problems over infinite domains, for example over real numbers, but constraint solving over infinite domains is not covered by this paper.

many researchers outside CP put equality between constraint satisfaction and simple enumeration, the reality is that the real power of CP is hidden in constraint propagation. Note however, that some other techniques, for example local search, can also be applied to solve the problems with constraints.

Constraint propagation is based on the idea of using constraints actively to reduce domains of the variables by removing the values that do not satisfy the constraint. Assume that variables X and Y have an identical domain {1,2,3,4,5} and there is a constraint X<Y between the variables. Visibly, the value 1 can be removed from the domain of Y because there is no value *a* in the domain X such that the pair (*a*,1) satisfies the constraint. Similarly, the value 5 can be removed from the domain of X. This domain filtering is realized by a so called *filtering algorithm* that is attached to each constraint. After applying the filtering algorithm, the constraint becomes (arc) consistent, that is for any value in the domain of a constrained variable there is a compatible value(s) in the domain of the other constrained variable(s) such that the pair (tuple) satisfies the constraint. The filtering algorithm is evoked every time a domain of some variable in the constraint is changed and this change is propagated to domains of the other variables in the constraint and so on (Figure 2).

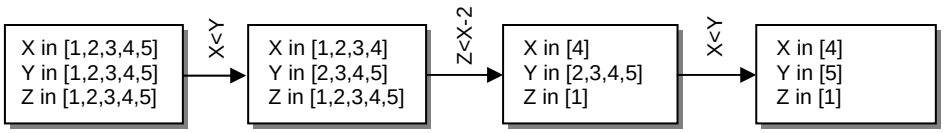
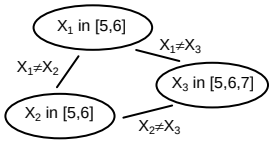


Fig. 2. Constraint propagation does domain reduction by repeated evoking of the filtering algorithms until a fix-point is reached

Notice that a filtering algorithm for a particular constraint may be evoked several times in the propagation loop but because values can only be removed from domains, propagation must finish sometime either by reaching a fix point or by emptying some domain. This basic constraint propagation scheme is called (generalized) *arc consistency* (AC) and it ensures that after finishing the propagation loop each constraint is (arc) consistent. Note however, that arc consistency ensures local consistency only meaning that the values which cannot be part of the complete feasible assignment may still remain in the domains. There exist consistency techniques stronger than AC, like path consistency, but they are rarely used in practice. More frequently, filtering is being strengthened by encapsulating a subset of constraints into a so called *global constraint*. Note finally that the same problem can often be modeled using different sets of constraints leading to different domain pruning and hence the important role of constraint modeling.

Global constraints. Due to its local character arc consistency cannot detect all global inconsistencies. Assume the constraint network on right. All binary constraints are consistent so arc consistency deduces no change of the domains. However, a more global view can discover that values 5 and 6 cannot be assigned to X_3 because they both will be used for X_1 and X_2. If we see the set of primitive constraints as a single n-ary constraint – a global constraint – we can use a special filtering algorithm for this constraint that achieves stronger pruning of domains. Régin proposed an efficient filtering algorithm for such a



constraint called all-different based on finding a maximal matching in the bipartite graph with variables on one side and values on the other side [8].

There exist many other global constraints; some of them were designed for a particular problem area. For example, global constraints in scheduling are used to describe resources to which activities can be allocated. Assume a unary resource that can process only one activity at given time. If we model the position of activity in time using its start time S and processing time P then the main feature of the unary resource, namely that no two activities overlap in time, can be modeled using a set of disjunctions $S_i+P_i \leq S_j \vee S_j+P_j \leq S_i$. Again, domain pruning via AC over such a set of disjunctive constraints is weak and stronger pruning can be achieved for example using a global constraint over all S_i and P_i based on a technique called edge-finding [1].

As we mentioned above, arc consistency is a local consistency technique so some infeasible values may still sit in the domains of the variables and thus search (with backtracking) is necessary to find a complete feasible assignment of the variables. This stage is often called *labeling* because the variables are being labeled there – the values from respective domains are assigned to the variables. After each assignment, the value is propagated to other variables using the above described AC scheme – this is called *maintaining arc consistency* during search. If a failure is detected (any domain becomes empty during AC) then another value is tried and if no value remains in the domain then the algorithm backtracks to the last but one variable and so on. In general, the labeling procedure adds new constraints to the system to resolve the remaining disjunctions until domains of all the variables are singleton. For example, finding a value for the variable X with the domain {5,6,7} is equivalent to resolving the disjunction $X=5 \vee X=6 \vee X=7$ which is called *enumeration* (the constraint X=5 is added first, then X=6 is tried, and finally X=7). Resolving $X=5 \vee X \neq 5$ is called a *step labeling* and resolving $X \leq 6 \vee X>6$ is called *bisection*. By adding the above mentioned constraints, the constraint solver actually builds different constraint models in different branches of the search tree so in some sense choosing the right branching scheme is a part of constraint modeling. We will present later how different branching schemes influence efficiency of constraint satisfaction.

Variable and value selection. The labeling procedure needs to be accompanied by heuristics for choosing the variable to be labeled and for selecting the value to be tried first (or, in general, for selecting the branch). If the variables are labeled in a fix ordered then we are speaking about the *left-most* variable selection. Frequently, *fail-first* variable selection is used where the variable with the smallest domain is labeled first (ties are broken randomly). Value selection heuristics are usually problem dependent, for example the smallest value for the time variables is tried first if we are minimizing the makespan (the end time of the latest activity) in scheduling problems.

The standard constraint satisfaction technique looking for feasible assignments can be extended to find out an optimal assignment. Usually a technique of *branch-and-bound* is used there. First, some feasible assignment is found and then, a next assignment that is better than the previous assignment is looked for and so on. This could be realized by posting a new constraint restricting the value of the objective function by the value of the objective function for the so-far best assignment. This is repeated until no feasible assignment is found and then the last found feasible assignment is the optimal assignment.

A deep and general view of constraint programming can be found in [2,6,7,11].

3 Modeling with Constraints

In this section, we will present several example problems and their constraint models. We will use these problems to demonstrate typical constraint modeling techniques like global constraints, symmetry breaking, dual models, and redundant constraints. The main idea behind these techniques is to improve solving efficiency of the models. To allow immediate testing of the presented ideas, the constraint models are programmed using the `clpfd` library of SICStus Prolog [3,9].

3.1 A Seesaw Problem: Symmetry Breaking and Global Constraints

Let us start our journey with a toy combinatorial problem of placing children to a seesaw [7]. Assume that Adam, Boris, and Cecil want to sit in a seesaw in such a way that the seesaw balances. There are five seats placed uniformly on both arms of the seesaw and one seat is placed in the middle (see Figure 3). Moreover, the boys want to have some space around them. In particular, they require that they are at least three seats apart. The weights of Adam, Boris, and Cecil are respectively 36, 32, and 16 kg. To solve the problem, we need to assign the seats to all children (or vice versa). Figure 3 shows one of the acceptable solutions to this problem.

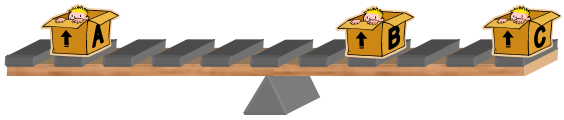


Fig. 3. A seesaw problem and one of its solutions

To model the problem as a constraint satisfaction problem, one needs to decide about the variables, their domains, and the constraints. The natural model for the seesaw problem is using a variable for each boy describing his position on the seesaw: A for Adam, B for Boris, C for Cecil. If we choose carefully the domain for these variables, that is a position on the seesaw: -5,-4,…,+4,+5, then the constraint that the seesaw balances is simply that the moments of inertia sums to 0:

$$36*A + 32*B + 16*C = 0.$$

To restrict the minimal distances between the boys we can use a standard formula for computing the distances that is an absolute value of the difference of the positions. Thus we get the constraints:

$$|A-B| > 2,\ |A-C| > 2,\ |B-C| > 2.$$

Note that $|A-B| > 2$ is a compact representation of the disjunctive constraint ($A-B > 2 \vee B-A > 2$) and that it also ensures that two boys are not sitting on the same seat.

The above constraints describe completely the seesaw problem. To get the solution we need to post all these constraints and to do labeling that is a procedure deciding about the variables' values via depth first search. Figure 4 shows a coding in SICStus Prolog.

```
seesaw(Sol):-
   Sol = [A,B,C],

   domain([A,B,C],-5,5),
   36*A+32*B+16*C #= 0,
   abs(A-B) #> 2,
   abs(A-C) #> 2,
   abs(B-C) #> 2.

   labeling([ff],Sol). % fail-first variable selection
```

Fig. 4. A constraint model for the seesaw problem

Notice that the constraint model for the seesaw problem is fully declarative. So far, we said no single word about how to solve the problem. We merely concentrate on describing the problem in terms of variables, domains, and constraints. The underlying constraint solver that encodes constraint propagation as well as the labeling procedure does the rest of the job.

If we now run the program from Figure 4 we get six different solutions (Figure 5).

```
?- seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;
X = [4,-5,1] ? ;
X = [4,-4,-1] ? ;
X = [4,-2,-5] ? ;
no
```

Fig. 5. All solutions of the seesaw problem

An open-minded reader might notice that only three of the above solutions are really different while the remaining three solutions are merely their symmetrical copies. These symmetrical solutions can be removed from the search space by adding a so called *symmetry breaking constraint*. For example it could be a constraint restricting Adam to sit on the seats marked by the non-positive numbers: $A \leq 0$.

Rule of thumb – symmetry breaking. Some solutions of the problem can be easily deduced from their symmetrical counterparts. These symmetrical solutions should be removed from the search space to decrease its size and thus to improve time efficiency of the search procedure. The easiest way how to realize it is via adding symmetry breaking constraints that avoid symmetrical solutions and reduce the search space. There exist other techniques of symmetry breaking, for details see [10]. For a more convincing example, how the symmetry breaking constraint improves efficiency of the constraint model, see the section on Golomb rulers.

Let us now try to improve further the constraint model from Figure 4. Usually, the constraint model that propagates more is assumed to be better so our goal is to improve the initial domain pruning. Figure 6 shows the result of the initial domain pruning before the start of labeling (the symmetry breaking constraint is included).

```
A in -4..0
B in -1..5
C in -5..5
```

Fig. 6. Initial domain pruning for the seesaw problem (including symmetry breaking)

As we already mentioned a typical way of improving domain pruning is using global constraints that encapsulate a set of primitive constraints. If we look at the constraint model for the seesaw problem (Figure 4), we can identify a set of similar constraints, namely the distance constraints, that is a good candidate to be encapsulated into a global constraint. Notice that these constraints express the same information as non-overlapping constraints in scheduling. In fact, we can see each boy as a box of width three and if these boxes do not overlap then all boys are at least three seats apart (Figure 7). So the seesaw behaves like a unary resource in scheduling problems.

Fig. 7. Allocating boys to seats is similar to scheduling activities to a unary resource

Thus, we can see the seesaw problem via glasses of scheduling and we can use a global constraint modeling the unary resource to substitute the set of distance constraints. For example, the following constraint may by used in SICStus Prolog:

```
serialized([A,B,C],[3,3,3],[ bounds_only(false)])
```

The first argument of the `serialized` constraint describes the start times of the "activities", the second argument describes their duration, and the third argument is used for options of the filtering algorithm (above, we asked to propagate the domains completely, not only their bounds). Figure 8 shows the initial domain pruning when the serialized constraint is used. We can see that more infeasible values are removed from the variables' domains and thus the search space to be explored by the labeling procedure is smaller.

```
A in -4..0
B in -1..5
C in (-5..-3)\/(-1..5)
```

Fig. 8. Initial domain pruning for the seesaw problem with the serialized constraint

Rule of thumb – global constraints. More values pruned from variables' domains mean smaller search space so the constraint models that prune more are assumed better. Most constraint solvers use (generalized) arc consistency so the global constraints are the best way how to improve pruning while keeping good time efficiency. Thus the basic rule in the design of constraint models is using global constraint wherever possible.

Typically, the global constraint substitutes a homogeneous set of primate constraints, like differences or distances between the variables. So if such a subset of constraints appears in the constraint model, it is a good candidate for a global constraint. As we showed in the seesaw problem, it is useful to look for global constraints even in the problem areas which do not seem to be directly related to a given problem.

3.2 An Assignment Problem: Dual Models

The second studied problem is more real-life oriented than the seesaw problem. It belongs to the category of assignment problems like allocating ships to berths, planes to stands, crew to planes etc. In particular, we will describe a problem of assigning workers to products.

Consider the following simple assignment problem [7]. A factory has four workers W_1, W_2, W_3, and W_4, and four products P_1, P_2, P_3, and P_4. The problem is to assign workers to products so that each worker is assigned to one product and each product is assigned to one worker (Figure 9).

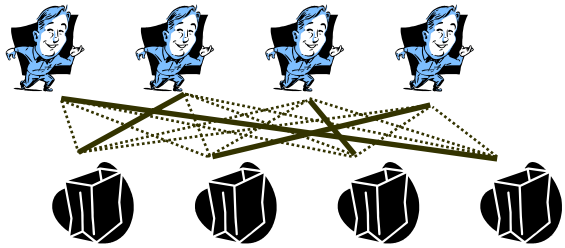


Fig. 9. A simple assignment problem

The profit made by the worker W_i working on the product P_j is given by Table 1.

Table 1. The profit made by workers on particular products

	P_1	P_2	P_3	P_4
W_1	**7**	**1**	**3**	**4**
W_2	**8**	**2**	**5**	**1**
W_3	**4**	**3**	**7**	**2**
W_4	**3**	**1**	**6**	**3**

The task is to find a solution to the above assignment problem such that the total profit is at least 19.

A straightforward constraint model can use a variable for each worker indicating the product on which the worker is working. The domain of such a variable will naturally consists of the products that can be assigned to a given worker. The fact that each worker is working on a different product can be described via a set of binary inequalities or better using the all-different constraint. To describe the profit of the worker, we can use a set of tabular constraints `element` encoding Table 1. The semantics of `element(X,List,Y)` is as follows: `X`-th element of `List` equals `Y`. Then the sum of the individual profits must be at least 19. Figure 10 shows the constraint model for the assignment problem in SICStus Prolog.

```
assignment_primal(Sol):-
    Sol = [W1,W2,W3,W4],

    domain(Sol,1,4),
    all_different(Sol),
    element(W1,[7,1,3,4],EW1),
    element(W2,[8,2,5,1],EW2),
    element(W3,[4,3,7,2],EW3),
    element(W4,[3,1,6,3],EW4),
    EW1+EW2+EW3+EW4 #>= 19,

    labeling([ff],Sol).
```

Fig. 10. A constraint model for the assignment problem

By running the above program we get four different assignments that satisfy the minimal profit constraint (Figure 11). The first two assignments have the profit 19, the third assignment has the profit 21 and the last assignment has the profit 20.

```
?- assignment_primal(X).

X = [1,2,3,4] ? ;
X = [2,1,3,4] ? ;
X = [4,1,2,3] ? ;
X = [4,1,3,2] ? ;
no
```

Fig. 11. All solutions of the assignment problem

Frequently, the assignment problem is formulated as an optimization problem. The nice feature of the CP approach is that one does not need to change a lot the constraint model to solve the optimization problem instead of the feasibility problem. Only the standard labeling procedure is substituted by a procedure looking for optimal assignments. In practice, the value of the objective function is encoded into a variable and the system minimizes or maximizes the value of this variable. Figure 12 shows a change in the code necessary to solve the optimization problem where the task is to find an assignment with the maximal profit.

```
…
EW1+EW2+EW3+EW4 #= E,

maximize(labeling([ff],Sol),E).
```

Fig. 12. A change of the constraint model to solve the optimization problem

The branch and bound technique behind the `maximize` procedure will now find the optimal solution which is `X=[4,1,2,3]`.

Let us now turn our attention back from optimization to the original constraint model. We decided to use the variables for the workers and the values for the products. However, we could also assign workers to products so the variables will correspond to the products while the values will identify the workers. Figure 13 shows such a dual constraint model.

```
assignment_dual(Sol):-
     Sol = [P1,P2,P3,P4],

     domain(Sol,1,4),
     all_different(Sol),
     element(P1,[7,8,4,3],EP1),
     element(P2,[1,2,3,1],EP2),
     element(P3,[3,5,7,6],EP3),
     element(P4,[4,1,2,3],EP4),
     EP1+EP2+EP3+EP4 #>= 19,

     labeling([ff],Sol).
```

Fig. 13. A dual model for the assignment problem

It may seem that both primal (Figure 10) and dual (Figure 13) models are equivalent. However, somehow surprisingly the dual model requires a smaller number of choices to be explored to find all the solutions of the problem (11 vs. 15). The reason is that the profit depends more on the product than on the worker. Thus, the profitability constraint propagates more for the products than for the workers. Figure 14 compares the initial pruning before the start of labeling procedure for both models. It shows that the dual model propagates more so it is a better model for this particular problem.

W1 in 1..4	P1 in 1..2
W2 in 1..4	P2 in 1..4
W3 in 1..4	P3 in 2..4
W4 in 1..4	P4 in 1..4

Fig. 14. Initial domain pruning for the assignment problem (left – primal model, right – dual model)

The propagation power of both primal and dual models can be combined in a single model. In practice, variables and constraints from both models are used together and special "channeling" constraints interconnect the models (SICStus Prolog provides the `assignment` constraint to interconnect the models). Figure 15 shows the combined constraint model. Notice that it is enough to label only the variables from one of the original models. Thanks to stronger domain pruning the combined model requires only 9 choices to be explored to find all the solutions of the problem. Figure 16 shows the initial pruning of the combined model.

Rule of thumb – dual models. In many problems, the role of variables and values can be swapped and a so called dual model to the original primal model can be obtained. Comparing the initial pruning of both primal and dual models could be a good guide for selecting one of them. Usually, the best model will be the one in which information is propagated more.

Sometimes, both primal and dual models can be combined together interconnected via channeling constraints. This combined model exploits the propagation power of both models but it also requires overhead to propagate more constraints. Consequently, one must be very careful when combining models with many constraints. Empirical evaluation of the models could be a good guide there.

```
assignment_combined(Workers):-
    Workers= [W1,W2,W3,W4],

    domain(Workers,1,4),
    all_different(Workers),
    element(W1,[7,1,3,4],EW1),
    element(W2,[8,2,5,1],EW2),
    element(W3,[4,3,7,2],EW3),
    element(W4,[3,1,6,3],EW4),
    EW1+EW2+EW3+EW4 #>= 19,

    Products = [P1,P2,P3,P4],

    domain(Products,1,4),
    all_different(Products),
    element(P1,[7,8,4,3],EP1),
    element(P2,[1,2,3,1],EP2),
    element(P3,[3,5,7,6],EP3),
    element(P4,[4,1,2,3],EP4),
    EP1+EP2+EP3+EP4 #>= 19,

    assignment(Workers,Products),

    labeling([ff],Workers).
```

Fig. 15. A combined model for the assignment problem

W1 in (1..2)\/{4}	P1 in 1..2
W2 in 1..4	P2 in 1..4
W3 in 2..4	P3 in 2..4
W4 in 2..4	P4 in 1..4

Fig. 16. Initial domain pruning for the assignment problem with the combined model

3.3 A Golomb Ruler Problem: Redundant Constraints

Lessons learnt in the previous sections will now be applied to solving a really hard problem of finding an optimal Golomb ruler of given size. In particular, we will show how “small” changes in the constraint model, like adding the symmetry breaking and redundant constraints, may influence dramatically the efficiency of the solver.

Golomb ruler of size M is a ruler with M marks placed in such a way that the distances between any two marks are different. The shortest ruler is the optimal ruler. Figure 17 shows an optimal Golomb ruler of size 5.

Fig. 17. An optimal Golomb ruler of size 5

Finding an optimal Golomb ruler is a hard problem. In fact, there is not known an exact algorithm to find an optimal ruler of size $M \geq 24$ even if there exist some best so far rulers of size up to 150 [5]. Still, these results are not proved yet to be (or not to be) optimal. Golomb ruler is not only a hard theoretical problem but it also has a practical usage in radio-astronomy. Let us now design a constraint model to describe the problem of the Golomb ruler.

A natural way how to model the problem is to describe a position of each mark using a variable. Thus for M marks we have M variables $X_1,\ldots, X_M$. The first mark will be in the position 0 and the position of the remaining marks will be described by a positive integer. Moreover, to prevent exploring all permutations of the marks, we can sort the marks (and hence the variables) from left to right by using constraints in the form $X_i<X_{i+1}$. Finally, we need to describe the difference of distances between the marks. Thus for each pair of marks *i* and *j* (*i*<*j*) we introduce a new distance variable $D_{i,j} = X_j - X_i$. The difference of distances is modeled using the all-different constraint applied to all distance variables. Figure 18 shows this base constraint model.

$$X_1 = 0$$
$$X_1<X_2<\ldots<X_M$$
$$\forall i<j\ D_{i,j} = X_j - X_i$$
$$\text{all\_different}(\{D_{1,2}, D_{1,3}, \ldots, D_{M,M-1}\})$$

Fig. 18. A base constraint model for the Golomb ruler

The base constraint model already includes several features discussed in the previous sections. In particular, we use a global constraint all-different instead of the set of binary inequalities. We already removed many symmetric solutions by using the ordering constraints (permutation can be seen as a special case of symmetry). There is no doubt about a positive effect of this feature. Still, there is one more symmetric solution to be removed and this is mirroring of the ruler. Assume the optimal ruler [0,1,4,9,11], then the ruler [0,2,7,10,11] is a mirror of this ruler that should be removed from the solution space. The mirror solutions can be removed for example by assuming only the solutions where the distance between the first two marks is smaller than the distance between the last two marks. The symmetry breaking constraint has the following form:

$$D_{1,2} < D_{M-1,M}$$

As we can see from Table 2, adding the above symmetry breaking constraint decreases significantly the running time because the symmetric areas of the search tree are not explored during search.

We can further improve efficiency of the model by adding some *redundant constraints*. The redundant constraint is not necessary to define the solution, it can be deduced from the existing constraints, but it can improve the propagation power. In case of the Golomb ruler, we can derive better bounds for the difference variables. $D_{i,j}$ is a distance between the marks *i* and *j*. Notice that this distance consists of the distances (i,i+1), (i+1,i+2) …(j-1,j). Formally,

$$D_{i,j} = D_{i,i+1} + D_{i+1,i+2} + \ldots + D_{j-1,j}$$

Because all distances must be different, we can estimate the minimal sum of distances (i,i+1), (i+1,i+2) …(j-1,j) as a sum of (j-i) different positive numbers. In particular:

$$D_{i,j} \geq \Sigma_{j-i} = (j-i)*(j-i+1)/2$$

Let us now try to estimate the upper bound for $D_{i,j}$ using a similar principle:

$$X_M = X_M - X_1 = D_{1,M} = D_{1,2} + D_{2,3} + \ldots + D_{i-1,I} + D_{i,j} + D_{j,j+1} + \ldots + D_{M-1,M}$$

$$D_{i,j} = X_M - (D_{1,2} + \ldots D_{i-1,i} + D_{j,j+1} + \ldots + D_{M-1,M})$$

Again, all distances must be different so we can estimate the minimal sum of distances (1,2),..,(i-1,i), (j,j+1), …, (M-1,M). There are (M-1-j+i) different numbers so the upper for $D_{i,j}$ can be defined as:

$$D_{i,j} \leq X_M - (M-1-j+i)*(M-j+i)/2$$

The above analysis of the problem deduced three additional redundant constraints that can be added to the base model to improve domain pruning. Figure 19 surveys all the additional constraints.

$$D_{1,2} < D_{M-1,M}$$
$$\forall i<j \; D_{i,j} \geq (j-i)*(j-i+1)/2$$
$$\forall i<j \; D_{i,j} \leq X_M-(M-1-j+i)*(M-j+i)/2$$

Fig. 19. An extension of the model for the Golomb ruler

As we can see from Table 2, the model with redundant constraints pays off and the running times are significantly smaller. For the interested reader, the complete code (SICStus Prolog) of the constraint model with symmetry breaking and redundant constraints is given in Appendix.

Table 2. Running times (in milliseconds on Mobile Pentium 4-M 1.70 GHz, 768 MB RAM) to find out optimal Golomb rulers using different constraint models

size	Base model	Base model + symmetry	Base model + symmetry + redundant constraints
7	220	80	30
8	1 462	611	190
9	13 690	5 438	1 001
10	120 363	49 971	7 011
11	2 480 216	985 237	170 495

Rule of thumb – redundant constraints. From the constraints in the base model we can sometimes deduce some derived constraints that bridge the weak domain pruning. These so called redundant constraints are not necessary to define the solution but if they are added to the base constraint model they can improve domain pruning. On the other hand, propagation via the redundant constraints adds overhead to solving time so one must be careful what and how many redundant constraints are added. A good test for adding redundant constraints can be improved initial pruning. A dual model added to the primal model is a special case of redundant constraints.

As we mentioned in the survey of constraint satisfaction technology, efficiency of problem solving is influenced by the constraint model but also by choosing the right labeling strategy. We achieved the above results (Table 2) by using a labeling strategy which selects the variable with the smallest index for assignment (leftmost variable selection) and which uses the step branching scheme ($X=Value \vee X \neq Value$). We compared two standard variable selection heuristics, namely fail-first and leftmost variable selection, and three branching schemes, namely enumeration, step labeling, and bisection, on the Golomb ruler problem (Table 3). The combination of the leftmost variable selection with the bisection branching scheme seems to be the best option for solving the Golomb ruler problem. Note that different parameters of the labeling strategy may be more appropriate for other problems. Usually, fail-first in the combination with step branching is used as the first choice.

Table 3. Comparison of variable and value selection heuristics for the Golomb ruler problem (runtime is measured in milliseconds on Mobile Pentium 4-M 1.70 GHz, 768 MB RAM)

size	leftmost			fail first		
	enum	step	bisect	enum	step	bisect
7	30	30	30	40	60	40
8	220	190	200	390	370	350
9	1 182	1 001	921	2 664	2 384	2 113
10	8 782	7 011	6 430	20 870	17 545	14 982
11	209 251	170 495	159 559	1 004 515	906 323	779 851

4 Conclusions

Determining the efficiency of different constraint models is a difficult problem and one which relies upon an understanding of the underlying constraint solver. The best model will be the one in which information is propagated earliest [7]. In this paper, we explained the basics of the constraint technology and we presented several techniques that usually improve efficiency of the constraint models by following the above rule on propagating earliest.

Encapsulating a set of constraints into a global constraint is always the recommended way of modeling especially if the appropriate global constraints are implemented in the system. As we showed, sometimes a global constraint intended to a different application area can be applied to the problem so do not be restricted to the subset of the global constraints for your problem area only.

We have also showed that some parts of the solution (search) space can be removed because the solutions from these parts can be easily reconstructed from other solutions. In particular, including so called symmetry breaking constraints always speeds up the solver because they prevent the solver to explore irrelevant (symmetrical) parts of the search space.

Last but not least we presented the idea of redundant constraints. Redundancy means that these constraints are not necessary to define the solution but they can significantly speed up the solver by improving domain pruning (and thus restricting the search space). One example of adding redundancy to the model is combining the

primal model with the dual model where the role of variables and values is swapped. However, redundant constraints add overhead necessary to propagate through them so the user must be careful about using them. Empirical evaluation of the models could be a good guide there.

5 Acknowledgements

The author is supported by the Czech Science Foundation under the contract No. 201/04/1102 and by the project LN00A056 of the Ministry of Education of the Czech Republic. I would like to thank the anonymous reviewers of the preliminary draft for very useful comments and suggestions.

References

1. Baptiste, P. and Le Pape, C.: Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group*, 1996.
2. Barták, R.: *On-line Guide to Constraint Programming*, Prague, 1998. http://kti.mff.cuni.cz/~bartak/constraints/
3. Carlsson M., Ottosson G., Carlsson B.: An Open-Ended Finite Domain Constraint Solver. *Proceedings Programming Languages: Implementations, Logics, and Programs*, Springer-Verlag LNCS 1292, 1997.
4. Freuder, E.C.: In Pursuit of the Holy Grail. *Constraints: An International Journal*, 2, 57-61, Kluwer, 1997.
5. Golomb rulers: some results, 2003. http://www.research.ibm.com/people/s/shearer/grtab.html
6. Kumar, V.: Algorithms for Constraint Satisfaction Problems: A Survey, *AI Magazine* 13(1): 32-44, 1992.
7. Mariot K. and Stuckey P.J.: *Programming with Constraints: An Introduction.* The MIT Press, 1998.
8. Régin J.-Ch.: A filtering algorithm for constraints of difference in CSPs. *Proceedings of 12th National Conference on Artificial Intelligence*, 1994.
9. SICStus Prolog 3.11.2 User's Manual.
10. Smith B.: Reducing Symmetry in a Combinatorial Design Problem. *Proceedings of CP-AI-OR2001*, pp. 351-359, Wye College, UK, 2001.
11. Tsang, E.: *Foundations of Constraint Satisfaction.* Academic Press, London, 1995.

Appendix

The following code describes a complete constraint model to solve the Golomb ruler problem of size M. More precisely, the largest problem that we have solved was of size 13 and it took almost eleven hours on Mobile Pentium 4-M 1.70 GHz. The code follows the syntax of constraints and built-in predicates of SICStus Prolog 3.11.2 [9]. For example, SICStus Prolog uses the `all_distinct` constraint that implements

the Régin's filtering algorithm while the `all_different` constraint implements a simple propagation where the value is removed from domains after its assignment to some variable. The last comment is about the upper bound for the variables describing marks. As the built-in labeling procedure requires the domains of the labeled variables to be finite we decided to use M^2 as the upper bound for these variables.

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

golomb(M,Sol):-
   UpperBound is M*M,
   ruler(M,-1,UpperBound,Sol),
   Sol = [0|_],        % set the first mark to 0
   last(Sol,XM),       % find the last mark
   distances(Sol,1,M,XM,Dist),
   all_distinct(Dist), % distances between marks are different
   (Dist=[DF,_|_] -> last(Dist,DL), DF#<DL ; true),
                       % symmetry breaking
   minimize(labeling([leftmost,bisect],Sol),XM).

% generate variables for marks and post ordering constraints
ruler(0,_,_,[]).
ruler(K,PrevX,UpperBound,[X|Rest]):-
   K>0,
   PrevX#<X, X#=<UpperBound,
   K1 is K-1,!,
   ruler(K1,X,UpperBound,Rest).

% generate distance variables and post redundant constraints
distances([],_,_,_,[]).
distances([X|Rest],I,M,XM,Dist):-
   J is I+1,
   distances_from_x(Rest,X,I,J,M,XM,Dist,RestDist),
   I1 is I+1,!,
   distances(Rest,I1,M,XM,RestDist).

% compute distances between X_I and the rest marks Y_J, I<J
% and post redundant constraints for distances
distances_from_x([],_,_,_,_,_,RestDist,RestDist).
distances_from_x([Y|Rest],X,I,J,M,XM,[DXY|Dist],RestDist):-
   DXY #= Y-X,
   LowerBound is integer(((J-I)*(J-I+1))/2),
   LowerBound #=< DXY,
   UpperBoundP is integer(((M-1-J+I)*(M-J+I))/2),
   DXY #=< XM - UpperBoundP,
   J1 is J+1,!,
   distances_from_x(Rest,X,I,J1,M,XM,Dist,RestDist).
```