PRACTICAL CONSTRAINTS: A TUTORIAL ON MODELLING WITH CONSTRAINTS

ROMAN BARTÁK*

Charles University, Faculty of Mathematics and Physics
Malostranské námestí 2/25, 118 00 Praha 1, Czech Republic
e-mail: bartak@kti.mff.cuni.cz

Abstract: Constraint programming provides a declarative approach to problem solving. The users just state the combinatorial (optimization) problems as constraint satisfaction problems and the underlying solver finds a solution for them. However, in practice, the situation is more complicated as there usually exist various ways how to describe the problem using variables, domains, and constraints. Moreover, the different models may lead to significantly different running times of the solvers. In fact, even a small change in the model may change the efficiency dramatically. This paper describes some known approaches to efficient modelling with constraints in a tutorial-like form.

Keywords: constraint satisfaction, problem modelling, applications

1 INTRODUCTION

"Constraint programming 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 users that designing a constraint model is an easy and straightforward task and that everything stated as a constraint satisfaction problem can be solved 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 modelling is becoming more and more important. Basically, it means that various models of the same problem may lead to different solving times, quite often to significantly different times. Unfortunately, there does not exist (yet) any guide that can steer the user how to design a solvable model. This "feature" of constraint technology might be a bit depressing for novices. Nevertheless, there are many rules of thumb about designing models that would be probably good in the solving times. Moreover, we also believe that it is important to be aware of the insides of constraint satisfaction to understand better behaviour of the solvers and, as a consequence, to design models that exploits the power of the solvers.

The goal of this paper is to provide an overview of modelling techniques used to state problems as constraint satisfaction problems. Respecting what has been said in the previous paragraph, we first survey the mainstream constraint satisfaction technology. Then we make a short view to insides of some interesting constraints to explain their behaviour. In the rest of the paper we demonstrate some modelling techniques using several funny (seesaw), real-life (assignment problem), and hard (Golomb ruler) problems.

2 CONSTRAINT SATISFACTION AT GLANCE

Constraint programming (CP) 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 the variables' values

* Supported by the Grant Agency of the Czech Republic under the contract no. 201/01/0942 and by the project LN00A056 of the Ministry of Education of the Czech Republic.

(Figure 1). Usually, the domains are finite and we are speaking about constraint satisfaction problems (CSP). The task is to find a valuation of the variables satisfying all the constraints, i.e., a feasible valuation. Sometimes, there is also an objective function defined over the problem variables. Then the task is to find a feasible valuation minimizing or maximizing the objective function. Such problems are called constraint satisfaction optimization problems (CSOP).

Note that modelling problems using CS(O)P is natural because the constraints can capture arbitrary relations and various constraints can be easily combined within a single system. Opposite to frameworks like linear and integer programming, the constraints 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. Then, a table is used to describe the feasible tuples. Moreover the constraints can bind variables with different even non-numerical domains, e.g. to restrict the length of a string by a natural number.

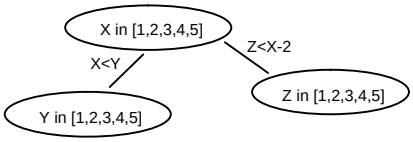

Fig. 1. CSP consists of variables (X,Y,Z), their domains [1,2,3,4,5], and constraints (X<Y, Y<X-2). It can be represented as a constraint (hyper) graph.

Constraint satisfaction technology must take in account the above described generality of the problem specification. Usually, a combination of search (enumeration) with constraint propagation is used; some other techniques, e.g., local search, can also be applied to solve problems with constraints. Despite the fact that many researchers outside CP put equality between constraint satisfaction and simple enumeration, the reality is that the core technology of CP is hidden in constraint propagation combined with sophisticated search techniques.

Constraint propagation is based on the idea of using constraints actively to prune the search space. Each constraint has assigned a filtering algorithm that can reduce domains of variables involved in the constraint by removing the values that cannot take part in any feasible solution. This 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 and so on (Figure 2). Hence the technique is called constraint propagation.

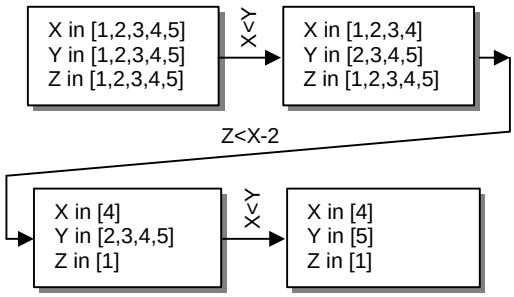

Fig. 2. Constraint propagation does domain reduction by repeated evoking of the filtering algorithms until a fix-point is reached.

Notice that each constraint may have its own filtering algorithm so there is no difficulty to solve the problems with very different constraints. The generic constraint propagation algorithm known under the notion of arc consistency takes care about the correct combination of the local filtering algorithms. On the other hand, this local view of the problem has the disadvantage of incomplete domain reduction. It means that some infeasible values may still sit in the domains of the variables and thus search (with backtracking) is necessary to find a complete feasible valuation of the variables. To reduce the deficiency of local propagation, it is possible to group several constraints and to see this group as a special constraint called a global constraint. Instead of using local propagation over the set of constraints, it is possible to design a special filtering algorithm for the global constraint to achieve more efficient domain filtering (see next section).

As we mentioned above, some search algorithm is usually necessary to find values of the variables. This stage is called labelling as the variables are being labelled there, i.e. the values from respective domains are assigned to variables. After each assignment, the value is propagated via constraints to other variables. If failure is detected then another value is tried. If no value remains in the domain then the algorithm backtracks to the last but one variable and so on. In general labelling adds new constraints to the system to resolve the remaining disjunctions (e.g. $X=5 \vee X\neq5$).

The standard constraint satisfaction technique looking for feasible solutions can be extended to find out an optimal solution. Usually a technique of branch-and-bound is used there. First, some feasible solution is found and then, a next solution that is better than the previous solution is looked for etc. This could be done by posting a new constraint restricting the value of the objective function by the value of the objective function for the so-far best solution.

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

3 INSIDE THE CONSTRAINTS

As mentioned in the introduction, we believe that understanding insides of constraint satisfaction technology improves the modelling skills. In particular, knowledge about how constraint propagation works prevents some surprise effects and as a consequence it leads to better models.

3.1 Disjunction

Assume modelling a simple disjunction X<5 ∨ X>7. There are several ways how to describe such constraint in constraint logic programming languages. The standard method in Prolog is to use two clauses to describe disjunction, in particular[1]:

```
a(X):-X#<5.
a(X):-X#>7.
```

However, this is not a good way of modelling in terms of constraint satisfaction because it leads to alternative constraint models. In particular, the first model contains the constraint X<5 and if we find later that this model has no solution, then the second model with X>7 is tried. The main difficulty of this approach is losing work done when solving the first model because the system must backtrack to introduce the alternative constraint X>7.

An alternative approach is using a disjunctive constraint in the form:

```
a(X):-X#<5 #\/ X#>7.
```

Then the constraint model is deterministic and search is realised within the labelling procedure only. Still, constraint propagation is very week there – the constraint does nothing until all but one components of the disjunction are proved to fail and then the remaining component is activated. In particular, after posting the above disjunctive constraint, the domain of the variable X does not change - we call it a surprise effect because what we expect from the constraint is to change the domain to (inf..4) ∨ (8..sup). As soon as the system finds out (for some reason) that X>4 then X<5 is proved to fail and the constraint X>7 is posted which leads to change of the domain for X.

There exist constructive approaches to disjunction which propagate each component in the disjunction separately and then the resulting domain pruning is a union of the pruned domains in each component. We can model this approach using the following constraint instead of the disjunction:

```
a(X):-X in (inf..4)\/(8..sup).
```

Constructive disjunction is expensive in general but if we are aware about its principles, we can implement them within our constraint models.

[1] We use the notation of `clpfd` library of SICStus Prolog to describe arithmetic constraints [3].

3.2 All-different

Constraint propagation can remove many inconsistent values from variables' domains. However, due to its local character it can hardly detect global inconsistencies. Assume the constraint satisfaction problem from Figure 3. Local propagation via arc consistency deduces not change of the domains because all pairs of values are locally consistent. However, a more global view can discover that values *b* and *c* cannot be assigned to X_3 because they will be used both for X_1 and X_2.

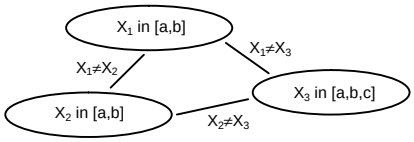

Fig. 3. Locally consistent constraint satisfaction problem that is not globally consistent (*b* and *c* can be removed from the domain of X_3).

Constraint programming provides a mechanism called global constraints to improve propagation in the group of constraints via encapsulating them into a single global constraint with some special filtering algorithm for it. Typically, the constraints in the group are in some sense homogeneous, e.g. it is a set of inequality constraints between every pair of variables. Règin proposed an efficient filtering algorithm for the global constraint called all-different modelling the set of inequalities [8].

The basic idea of Règin's filtering algorithm is to represent the constraint as a bipartite graph with variables on one side and values on the other side - so called value graph (Figure 4). The edges connect the variables with the values in their domains.

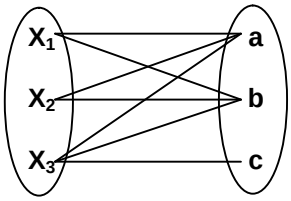

Fig. 4. A value graph for the all-different constraint with three variables.

The filtering algorithm for the all-different constraint is then realised via computing maximal matching in this graph. If an edge is not part of any maximal matching then this edge is removed from the graph. This corresponds to removing the value from the variable's domain.

The advantage of Règin's algorithm is maintaining global consistency over the set of variables while keeping the time efficiency close to local propagation. Therefore it is almost always better to use such global constraints instead of a set of constraints. The details on the Règin's algorithm can be found in [8].

3.3 Scheduling constraints

While some global constraints are more or less generally applicable (like the all-different), many other global constraints were proposed for particular application areas. For example, one of the most popular global constraints in scheduling is edge finding. We describe the version for unary resources but there exist variants for discrete resources as well [1]. As we show later, even if edge finding origins in the scheduling applications it can be applied to other non-scheduling areas.

The scheduling task is to allocate know activities to limited resources. Typically, each activity is described using its start time S and its processing time P. If the resource can process only one activity per time (so called unary resource) then the activities cannot overlap. It means that either one activity precedes the other activity or vice versa. Such constraint can be described as a disjunction:

$$S_1+P_1 \leq S_2 \vee S_2+P_2 \leq S_1$$

For the set of n activities allocated to a single resource we get n^2 binary disjunctive constraints of the above type. We already know that propagating through a disjunctive constraint is rather weak and that a group of similar constraints could be encapsulated in a global constraint. Edge finding is one of the most widely used techniques behind such scheduling global constraint.

The basic idea of edge finding is to identify an "edge" between the activity and the group of activities, in particular to find out if the activity must be processed before the set of activities (or after it). Assume that A is an activity and Ω is a set of activities that does not contain A. In a unary resource the processing time for the set of activities equals to the sum of processing times of these activities:

$$p(\Omega) = \sum_{X \in \Omega} p(X)$$

Assume that processing of the activities from $\Omega \cup \{A\}$ does not start with A. Then processing must start with some activity from Ω so the minimal start time is:

$$\min(start(\Omega)) = \min_{X \in \Omega}\{start(X)\}$$

If we add the processing time of $\Omega \cup \{A\}$ to the minimal start time of Ω and we get the time after the maximal end time of $\Omega \cup \{A\}$ then we know that the activity A can be processed neither inside nor after Ω (Figure 5). Thus, the activity A must start before Ω.

Formally:

$$min(start(W)) + p(W) + p(A) > max(end(W \cup \{A\})) \Rightarrow A << W.$$

$A << \Omega$ means that A must be processed before every activity from Ω so it must be processed before any $\Omega' \subseteq \Omega$. We can use this information to decrease the upper bound for the end time of the activity A using the formula:

$$end(A) \leq min\{ max(end(W')) - p(W') \mid W' \subseteq W\}.$$

A similar rule can be constructed to deduce that A must be processed after Ω:

$$min(start(W \cup \{A\})) + p(W) + p(A) > max(end(W)) \Rightarrow W << A.$$

The above edge finding rules form the core of the filtering algorithm reducing the bounds of the time variables. It may seem that this algorithm must explore all subsets Ω of the set of activities allocated to a given resource. Fortunately we can explore only the sets defined by pairs of activities called tasks intervals [1] so the time complexity of the edge finding filtering algorithm is $O(n^3)$ where n is the number of activities allocated to the resource.

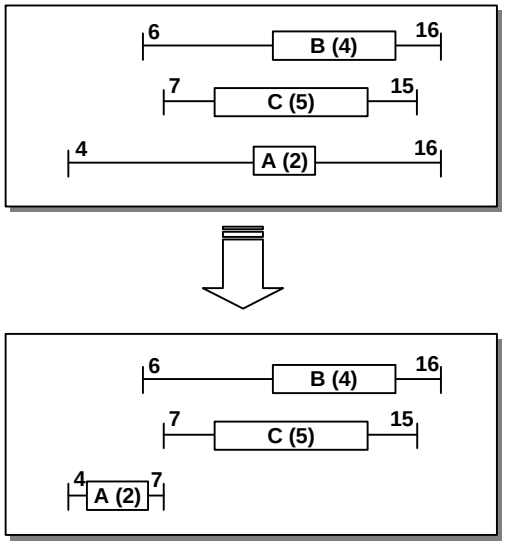

Fig. 5. Edge finding can deduce that the activity A must be processed before the activities B and C (processing time is in parentheses). Notice that binary disjunctive constraints deduce nothing there.

4 MODELLING WITH CONSTRAINTS

In this section, we present several example problems and their constraint models. The main issue behind the presented models is efficiency. We present several techniques how to improve efficiency of the models by adding redundant constraints. To allow immediate testing of the presented ideas, the models are programmed using the `clpfd` library of SICStus Prolog [3,8].

4.1 Seesaw

Let us start our journey with a simple 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 6). 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 seats to all children. Figure 6 shows one of the acceptable solutions to this problem.

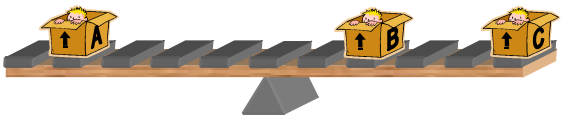

Fig. 6. A seesaw problem and one of its solutions

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, i.e., A for Adam, B for Boris, C for Cecil. If we choose carefully the domain for these variables, i.e. -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 distances, i.e. 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 $(A-B > 2 \vee B-A > 2)$.

The above constraints describe completely the seesaw problem. To get the solution we need to post all these constraints and to do labelling that is a procedure deciding about the variables' values via a depth first search. Figure 7 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).
```

Fig. 7. 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 labelling procedure does the rest of the job.

If we now run the program from Figure 7 we get six different solutions (Figure 8).

```
?- 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. 8. All solutions of the seesaw problem

As the open-eyed reader might notice, only three of these solutions are really different. The remaining three solutions are merely the symmetrical copies of the first three solutions. Thus we can get these solutions easily without wasting time in the general solving mechanism. To remove the symmetrical solutions from the search space one can add so called symmetry breaking constraint. In case of the seesaw problem, it could be a constraint restricting Adam to sit on the seats with non-positive numbers:

$$A \leq 0.$$

It may seem that the goal of the symmetry breaking constraints is to remove the symmetrical solutions only. However, this is useless if we are looking just for one solution satisfying the constraints. In fact, the real role of the symmetry breaking constraints is somewhere else. They remove parts of the search space where no solution exists because the search procedure already explored the symmetrical part of the search space and it found no solution there. For example, if we find that Adam cannot sit on the seat number -5, then we know immediately that he cannot sit on the seat number 5 too. Therefore, the symmetry breaking constraints reduce the search space and thus they increase efficiency of the models (see Section 4.3 for more convincing example). There exist other techniques of symmetry breaking, for details see [10].

If we look at the constraint model for the seesaw problem (Figure 7), we can see there a set of quite similar constraints, namely the distance constraints. Recall, that domain filtering is done independently in these constraints and domain changes are propagated between the constraints using the standard arc consistency technique. As we showed in section 3.2 this may lead to weaker pruning in comparison with some global consistency technique. Figure 9 shows the result of

the initial domain pruning before the start of labelling (the symmetry breaking constraint is included).

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

Fig. 9. Initial domain pruning for the seesaw problem (including symmetry breaking)

As showed in Section 3.2, encapsulating a set of constraints into a global constraint can improve domain pruning while keeping the reasonable efficiency. For some problem areas there are special global constraints designed but after some abstraction they can be used in other problems as well. For example, if we see the boy as a box of width three, then, if the boxes do not overlap, all boys are at least three seats apart (Figure 10).

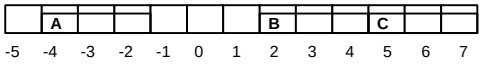

Fig. 10. Allocating boys to seats is similar to scheduling activities to a unary resource.

Thus, we can see the seesaw problem via glasses of scheduling and we can use the edge-finding like technique to model the set of distance constraints. In particular, the following constraint may substitute the set of distance constraints:

```
serialized([A,B,C],[3,3,3],[]).
```

The first argument of the serialized constraint describes the start times of the "activities" while the second argument describes their duration (the last argument is used for options like precedences which are not applied there). The constraint ensures that the activities do not overlap.

Figure 11 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 labelling is smaller.

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

Fig. 11. Initial domain pruning for the seesaw problem with the serialized constraint

4.2 Assignment problems

The second studied problem is more real-life oriented than the seesaw problem. It belongs to the category of assignment problem 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 12).

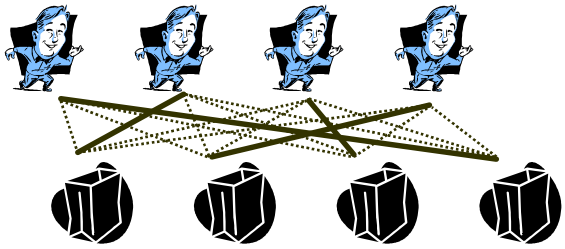

Fig. 12. A simple assignment problem.

The profit made by worker W_i working on product P_j is given by the table in Figure 13.

	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**

Fig. 13. Table describing profit made by workers on particular products

The task is to find a solution to the above 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 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 tabular constraint `element`. Then the sum of the individual profits must be at least 19. Figure 14 shows the constraint model for the assignment problem.

```
assignment(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. 14. A constraint model for the assignment problem

By running the above program we get four different assignments that satisfy the minimal profit

constraint (Figure 15). The first two assignments have profit 19, the third assignment has profit 21 and the last assignment has profit 20.

```
?- assignment(X).

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

Fig. 15. All solutions of the assignment problem

Quite often the task is not to find a feasible solution but to find an optimal solution. Typically, the constraint solvers use branch and bound technique to find optimal solutions. The nice feature is that one does not need to change the constraint model to solve an optimisation problem. Only the standard labelling procedure is substituted by a procedure looking for optimal solutions. In practice, the value of the objective function is encoded into a variable and the system minimises or maximises the value of this variable. Figure 16 shows a change in the code necessary to solve the optimisation problem where the task is to find an assignment with maximal profit.

```
…
EW1+EW2+EW3+EW4 #= E,

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

Fig. 16. A change of the constraint model to solve the optimisation 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 optimisation to the original constraint model. We decided to use variables for workers and values for products. However, it is possible to swap the role of variables and values and to describe products by variables and workers assigned to the products as values for these variables. Figure 17 shows such a dual constraint model.

```
assignment(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. 17. A dual model for the assignment problem

In many problems the role of variables and values can be swapped and it is the model designer who decides which model is more appropriate. In our assignment problem it may seem that both models are fully 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 profit depends more on the product than on the worker. Thus, the profitability constraint propagates more for products than for workers. Figure 18 compares the initial pruning before the start of labelling for both primal and dual models.

```
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. 18. Initial domain pruning for the assignment problem (left-primal model, right-dual model).

Determining the efficiency of different models is a difficult problem. Usually, the best model will be the one in which information is propagated first. To improve propagation, the primal and dual models can be combined into one model. In practice, it means that variables and constraints from both models are used together and special "channelling" constraints interconnect the models (SICStus Prolog provides the `assignment` constraint to interconnect the models). Figure 19 shows a constraint model where both primal and dual models are combined. Thanks to stronger domain pruning this model requires only 9 choices to be explored to find all the solutions of the problem

```
assignment(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. 19. A combined model for the assignment problem

Combining primal and dual models is an easy way how to improve domain pruning. As Figure 20 shows, this combination really helped to prune domains of the variables describing workers.

```
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. 20. Initial domain pruning for the assignment problem with combined primal and dual models

On the other hand, the combined model requires overhead to propagate more constraints so one must be very careful when combining models with many constraints.

4.3 Golomb ruler

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 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 the marks are different. The shortest ruler is the optimal ruler. Figure 21 shows an optimal Golomb ruler of size 5.

Fig. 21. 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 ≥ 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 then described using the all-different constraint applied to all distance variables. Figure 22 shows the above basic constraint model.

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

Fig. 22. A basic constraint model for the Golomb ruler

The basic constraint model already includes several features discussed above. In particular, we use a global constraint all-different instead of the set of binary inequalities. Surprisingly, this decreases slightly efficiency of solving (see Figure 24) probably because domain filtering in other constraints is so weak that the overhead of all-different exceeds its pruning power.

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 symmetry 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 so it can be removed from the solution set. To remove such symmetry we can use a constraint in the following form:

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

As we can see from the table in Figure 24, adding this single constraint decreases significantly the running time. In fact, solving is almost two times faster because the symmetric sub-trees are not explored during search.

We can further improve efficiency of the model by adding some redundant constraints. For example, we can compute 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). 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 $D_{i,j}$:

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

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

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:

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

The above analysis of the problem deduced three additional constraints that can be added to the basic model to improve domain pruning. Figure 23 surveys these 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. 23. An extension of the model for the Golomb ruler

As we can see from the table in Figure 24, the improved model pays off and the running times are significantly smaller. We have also tried the models without the Règin's filtering algorithm for the all-different constraint. In case of the base model, the running times are slightly better (for the reasons see above) but for the extended model, all-different contributes significantly to good efficiency.

	Base model - all_diff	Base model	Base model + symmetry	Base model + symmetry + bounds	Base model + symmetry + bounds - all_diff
7	0	1	0	0	0
8	2	2	1	0	1
9	18	17	8	2	7
10	159	149	76	15	61
11	3327	3455	1811	772	1766

Fig. 24. Running times (in seconds on Mobile Pentium 4-M 1.70 GHz, 768 MB RAM) to find out optimal Golomb rulers using different constraint models. „–all_diff" means simple propagation only (see Appendix).

For comparison with other algorithms we include some optimal Golomb rulers (Figure 25).

1	[0]
2	[0,1]
3	[0,1,3]
4	[0,1,4,6]
5	[0,1,4,9,11]
6	[0,1,4,10,12,17]
7	[0,1,4,10,18,23,25]
8	[0,1,4,9,15,22,32,34]
9	[0,1,5,12,25,27,35,41,44]
10	[0,1,6,10,23,26,34,41,53,55]
11	[0,1,4,13,28,33,47,54,64,70,72]
12	[0,2,6,24,29,40,43,55,68,75,76,85]

Fig. 25. Some optimal Golomb rulers

5 CONCLUSIONS

Determining the efficiency of different 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 insides of some constraints to understand better their behaviour. We have also presented several techniques that usually improve efficiency of the models by following the above rule on propagating earliest.

Encapsulating a set of constraints into a global constraint is always the recommended way of modelling 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.

In the presented models, we use a standard labelling procedure based on the first-fail principle. Another way how to improve efficiency is defining dedicated search procedures but this is a different story.

6 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. Règin J.-Ch.: A filtering algorithm for constraints of difference in CSPs. *Proceedings of 12th National Conference on Artificial Intelligence*, 1994.
9. SICStus Prolog 3.8.7 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 any size M. More precisely, the largest problem that we have solved was of size 13 and it took a couple of days on 1.8 GHz Pentium 4; solving problems of larger size will definitely take much more time. The code was tested in SICStus Prolog 3.8.7 [9] so it follows the syntax of constraints and built-in predicates of SICStus Prolog. For example, SICStus Prolog uses `all_distinct` constraint that implements the Régin's filtering algorithm while `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 labelling procedure requires the domains of the labelled 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):-
     Sol = [0|_],
     UpperBound is M*M,
     ruler(M,-1,UpperBound,Sol),
     last(Sol,XM),
     distances(Sol,1,M,XM,Dist),
     all_distinct(Dist),

     (Dist=[DF,_|_] ->
         last(Dist,DL), DF#<DL
     ;
         true
     ),

     minimize(labeling([ff],Sol),XM).

ruler(0,_,_,[]).
ruler(K,PrevX,UpperBound,[X|Rest]):-
     K>0,
     PrevX#<X, X#=<UpperBound,
     K1 is K-1,!,
     ruler(K1,X,UpperBound,Rest).

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).

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).
```