Decision Making in Manufacturing and Services
Vol. 5 • 2011 • No. 1–2 • pp. 69–78

Models and Tools for Improving Efficiency in Constraint Logic Programming

Antoni Ligęza*

Abstract. Constraint Satisfaction Problems typically exhibit strong combinatorial explosion. In this paper we present some models and techniques aimed at improving efficiency in Constraint Logic Programming. A hypergraph model of constraints is presented and an outline of strategy planning approach focused on entropy minimization is put forward. An example cryptoaritmetic problem is explored in order to explain the proposed approach.

Keywords: Constraint Satisfaction Problem, Constraint Programming, Constraint Logic Programming

Mathematics Subject Classification: 68N17, 68N19, 68N99

Revised: 30 May 2011

1. INTRODUCTION

A *Constraint Satisfaction Problem* (CSP, for short) (Dechter 2003, Apt 2006) is a task consisting in finding an assignment of values to given variables. The assigned values are restricted to belong to predefined sets — variable domains. The key issue is that the assignment must satisfy a set of predefined constraints. Constraint processing is often done with logical methods and tools (Apt 2006). Practical solutions are based on Prolog search an backtracking mechanism and take the form of *Constraint Logic Programming*.

CSP is a common model for diversity of practical, theoretical, toy and entertainment problems, with cryptoarithmetic puzzles and *Sudoku* being some perfect examples. In industrial practice CSP may serve as a generic model for design, planning, scheduling, etc. In theoretical research it is a model for logical formulae satisfaction checking (the so-called SAT problem) and mathematical programming.

Solving a generic CSP is both conceptually simple and intractable. It is conceptually simple since it can be accomplished — well, at least in theory — by consecutive scanning of all the elements of the Cartesian Product of all the domains. Such a Cartesian Product defines the *search space* for the solutions. By checking for each element

* AGH University of Science and Technology, Krakow, Poland

all the constraints, all the potential solutions can be found. If all the domains are finite, so is the Cartesian Product of them. Hence the algorithm for finite domains always finds all the solutions, provided that it is not an over-constrained problem. The real problem is that CSP suffers from *combinatorial explosion* with respect to size of the search-space of potential solutions. Such problems are referred to as *intractable*. More precisely, intractable problems are ones for which it is known that there is no polynomial algorithm for finding the solution (Dechter 2003).

The general idea behind solving CSP consist in (i) ordering the variables along with some criteria of preference, (ii) subsequent selection of values from their domains and assignment of one value to each of the variables at a time, and (iii) checking if some of the constraints are not satisfied at any stage when it is possible. Whenever an inconsistent assignment is found, whether partial or complete one, *backtracking* is enforced, and a subsequent potential solution is explored. This is backtracking which is responsible for inefficiency, but for a number of problems backtrack-free solution does not exist.

In the literature the search is organized with use of the *Constraint Graph*, i.e. a graph modelling the structure of the constraints (Dechter 2003). In such a graph nodes represent the variables and vertices show that two variables are within the scope of the same constraint. A constraint graph can be used to help manage the order of variable assignment but we miss the precise knowledge about the structure of the constraints. For example, if some k variables are bound by *two* or more different constraints it is not visible from the constraint graph.

In some former works (Ligęza & Kościelny 2008, Ligęza 2009b, Ligęza 2009a) we have applied the constraint processing techniques to refining the set of potential diagnoses. An approach incorporating a special AND-OR graph for constraint modeling and rules for modeling constraint related qualitative knowledge was proposed. In this paper an attempt at extending these techniques towards a classical CSP problem is discussed.

The explicit representation of constraints in the form of inference rules has an obvious advantage: once the values assigned to variables occurring in preconditions are known, such a rule can be fired and values of the variables occurring in the consequent part become known. They can be immediately used either in further search or – in case of inconsistency – for enforcing immediate backtracking.

The proposed approach is illustrated with practical solving an example of a well-known cryptoarithmetic problem.

2. CONSTRAINT SATISFACTION PROBLEMS. BASIC FORMULATION

A Constraint Satisfaction Problem is one where the goal consists in finding a legal assignment of values to a finite set of predefined variables so that a set of given constraints is satisfied.

More formally, after (Dechter 2003), let $X = \{X_1, X_2, \ldots, X_n\}$ denote a set of variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of domains for the variables in X and C is a set of constraints. Each constraint is given by a pair (S_i, R_i), where S_i is referred to as the scope (or scheme) and consists of a selection of variables from X while R_i is

a relation defined over the Cartesian Product of domains appropriate for the variables in the scope. The relation R_i can be defined *explicitly*, i.e. by listing all its tuples, but more frequently it is defined *implicitly* by use of logical or algebraic constraints. The *Constraint Satisfaction Problem (CSP)* is given by the triple (X, D, C).

A solution to a CSP given by (X, D, C) is any assignment of values to variables of X of the form $\{X_1 = d_1, X_2 = d_2, \ldots, X_n = d_n\}$, such that $d_i \in D_i$, $i \in \{1, 2, \ldots, n\}$ and for any constraint $(S_i, R_i) \in C$, R_i is satisfied by the appropriate projection of the solution vector $(d_1, d_2, \ldots, d_n)$ over variables of S_i. Obviously, all the constraints must be satisfied, and there can be one or many solutions; no solution may exist for an over-constrained problem.

The basic technique for solving a CSP given by (X, D, C) consists in subsequent assignment of admissible values to variables of X; the order is chosen in an arbitrary way, and it can influence how fast a solution is found.

In order to improve search efficiency both heuristics and strategies are used. The most typical strategies are based on (i) variable ordering, (ii) values ordering, and (iii) *look-ahead* techniques. Especially the look-ahead strategies can improve search efficiency. The principal idea is that the algorithm looks how current decisions will affect the future search.

3. EXAMPLE PROBLEM STATEMENT AND ITS MODELS

Constraint satisfaction models and techniques present relatively matured level; for the state-of-the-art consult (Dechter 2003, Apt 2006) and for a nice review of modern approaches to constraint propagation (Russell & Norvig 2003) (Chapter 5). Here we examine some of them in solving a cryptoarithmetic problem.

Consider the following well-known cryptoarithmetic puzzle (Apt 2006):

```
  SEND
+ MORE
------
 MONEY
```

The variables – `S, E, N, D, M, O, R, Y` – are to be assigned digits so that the above constraint is satisfied. Different variables are to be assigned different digits. Leading digits (in our example `S` and `M`) are different from 0.

The simplest model for solution of this problem can be as follows:

- successively assign all the variables some digits,
- check if all the variables are assigned different digits; if no – backtrack,
- perform the final summation test.

Such an approach is called *generate-and test*. It is intuitive and very simple to implement in pure PROLOG. Unfortunately, it is very inefficient. Since there are 8 variables, and each can take 10 values, the overall search space contains 10^8 potential solutions.

A slight modification can consist in immediate checking if different variables are assigned different values, once some two variables are assigned values. This approach

is also referred to as *generate-or-test* and it is much more reasonable. Unfortunately, as we shall see in Section 4 it is still far from being efficient.

Another modification may consists in developing a way of variable assignment so that a value once assigned to a variable is removed from the domains of yet-unassigned variables. This approach is referred to as *forward checking* and improves efficiency in a significant way. However, applied alone, it still exhibits high inefficiency.

A more efficient approach may consist in *constraint propagation*. Consider the puzzle and let us assume that variables D and E have been assigned some values (in fact there are $10 * 9$ of different possibilities). Once this is done, Y can be calculated directly, as well as the value of $C1$ being the carry value for the next column. Now, having $C1$ and selecting values of N and R (and note that there are $8 * 7$ possibilities left) one can calculate E. There are two basic possibilities: the calculated E is consistent with the value assumed in the former step — and so we can proceed, or it is different, and backtracking must take place.

In general, consider a constraint propagation rule of the form:

$$D_1^i = d_1 \wedge D_2^i = d_2 \wedge C^{i-1} = c_{i-1} \longrightarrow D^i = \\ = mod_{10}(d_1 + d_2 + c_{i-1}) \wedge C^i = div((d_1 + d_2 + c_{i-1})/10) \quad (1)$$

The rule has simple meaning: having two digits D_1^i and D_2^i to be summed up and the carry signal from the lower position, the current digits D^i is calculated as the sum of the above modulo 10 and the carry signal to next position is calculated as the appropriate integer division result.

It turns out that this approach can reduce the search space in an efficient manner.

Finally, the summation constraint can be decomposed to the following set of five local constraints; unfortunately, the local constraints are not independent, and, moreover, addition variables representing the carry signal have to be introduced. We have:

$$D + E = 10 * C1 + Y, \quad (2)$$

$$N + R + C1 = 10 * C2 + E \quad (3)$$

$$E + O + C2 = 10 * C3 + N, \quad (4)$$

$$S + M + C3 = 10 * C4 + O, \quad (5)$$

$$M = C4, \quad (6)$$

The above constraints can be explored directly after the appropriate variables are instantiated or as a final summation test.

4. EXAMPLE SOLUTIONS

Let us consider example solutions and their efficiency.

As a reference point lets us start with the naive *generate-or-test* approach. A Prolog code follows:

```
digit(1). digit(2). digit(3). digit(4). digit(5).
digit(6). digit(7). digit(8). digit(9). digit(0).
solve(S,E,N,D,M,O,R,Y) :-
     digit(S),S\=0,
     digit(E),E\=S,
     digit(N),N\=E,N\=S,
     digit(D),D\=N,D\=E,D\=S,
     digit(M),M\=D,M\=N,M\=E,M\=S,M\=0,
     digit(O),O\=M,O\=D,O\=N,O\=E,O\=S,
     digit(R),R\=O,R\=M,R\=D,R\=N,R\=E,R\=S,
     digit(Y),Y\=R,Y\=O,Y\=M,Y\=D,Y\=N,Y\=E,Y\=S,
     N1 is 1000*S+100*E+10*N+D,
     N2 is 1000*M+100*O+10*R+E,
     N3 is 10000*M+1000*O+100*N+10*E+Y,
     N3 =:= N1+N2.
```

The results show inefficiency of this basic approach (but also efficiency of modern PROLOG implementations):

```
?- time(solve(S,E,N,D,M,O,R,Y)).
% 38,730,365 inferences, 9.35 CPU in 9.38 seconds (100% CPU, 4142285 Lips)
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2.
```

Now, consider a simple implementation of *forward-checking* strategy[1]:

```
smm :-
        X = [S,E,N,D,M,O,R,Y],
        Digits = [0,1,2,3,4,5,6,7,8,9],
        assign_digits(X, Digits),
        M > 0,
        S > 0,
                  1000*S + 100*E + 10*N + D +
                  1000*M + 100*O + 10*R + E =:=
        10000*M + 1000*O + 100*N + 10*E + Y,
        write(X).

select(X, [X|R], R).
select(X, [Y|Xs], [Y|Ys]):- select(X, Xs, Ys).

assign_digits([], _List).
assign_digits([D|Ds], List):-
        select(D, List, NewList),
        assign_digits(Ds, NewList).
```

The efficiency is improved in a visible way, we have:

```
?- time(smm).
[9, 5, 6, 7, 1, 0, 8, 2]
% 10,503,156 inferences, 8.48 CPU in 8.55 seconds (99% CPU, 1238580 Lips)
true.
```

[1] Source: `http://clip.dia.fi.upm.es/~vocal/public_info/seminar_notes/node13.html`

Now consider the model incorporating propagation rules. A simple implementation can looks as follow (Bratko 2000):

```
solve(S,E,N,D,M,O,R,Y):- run([O,S,E,N,D],[O,M,O,R,E],[M,O,N,E,Y]), M\=0.

del(A,L,L):- nonvar(A),!.
del(A,[A|L],L).
del(A,[B|L],[B|L1]):- del(A,L,L1).

digitsum(D1,D2,C1,D,C,Digs1,Digs):-
del(D1,Digs1,Digs2),
del(D2,Digs2,Digs3),
del(D,Digs3,Digs),
S is D1+D2+C1,
D is S mod 10,
C is S // 10.

sum1([],[],[],0,0,Digits,Digits).
sum1([D1|N1],[D2|N2],[D|N],C1,C,Digs1,Digs):-
sum1(N1,N2,N,C1,C2,Digs1,Digs2),
digitsum(D1,D2,C2,D,C,Digs2,Digs).
run(N1,N2,N):-
sum1(N1,N2,N,0,0,[0,1,2,3,4,5,6,7,8,9],_).
```

By comparison to the former approaches, application of direct propagation rules turns out to be surprisingly efficient:

```
?- time(solve(S,E,N,D,M,O,R,Y)).
% 45,619 inferences, 0.02 CPU in 0.02 seconds (125% CPU, 2280950 Lips)
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2.
```

A still more efficient approach makes use of advanced constraint propagation techniques. To illustrate the approach we shall use the SWI-Prolog library for constraint logic programming over finite domains. An example code looks as follows[2]:

```
sendmoremoney(Vars) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    S #\= 0,
    M #\= 0,
    all_different(Vars),
                1000*S + 100*E + 10*N + D
    +           1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y.

solve(Vars):- Vars=[S,E,N,D,M,O,R,Y],
    sendmoremoney([S,E,N,D,M,O,R,Y]),label(Vars).
```

The power of these advanced library is impressive; not only the code is comprehensive (smart and readable), but the efficiency is very high:

[2] Source: http://en.wikibooks.org/wiki/Prolog/Constraint_Logic_Programming

```
?- time(solve(Vars)).
% 10,088 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
Vars = [9, 5, 6, 7, 1, 0, 8, 2].
```

Finally, we may check the influence of the decomposition of the summation constraint into five separate constraints presented in Section 3.

```
send5(Vars,Cars) :-
    Cars = [C1,C2,C3,C4],
    Cars ins 0..1,
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    all_different(Vars),
    S #\= 0,
    M #\= 0,
    M #= C4,
    D + E      #= 10*C1 + Y,
    N + R + C1 #= 10*C2 + E,
    E + O + C2 #= 10*C3 + N,
    S + M + C3 #= 10*C4 + O.

solve5(Vars,Cars):- Vars=[S,E,N,D,M,O,R,Y],Cars=[C1,C2,C3,C4],
    send5([S,E,N,D,M,O,R,Y],[C1,C2,C3,C4]),label(Vars),label(Cars).
```

The results, although comparable, are by around 50% worse than the above obtained for single test constraint.

```
?- time(solve5(Vars,Cars)).
% 15,130 inferences, 0.01 CPU in 0.00 seconds (207% CPU, 1513000 Lips)
Vars = [9, 5, 6, 7, 1, 0, 8, 2],
Cars = [1, 1, 0, 1].
```

5. HYPERGRAPHS AND RULES

In this section we put forward a rational proposal for an efficient strategy for efficient solution of the cryptoarithmetic problem under discussion. The main idea consists in combining the following steps:

- modeling the constraints with as many as detailed constraints, as possible; in our case we shall use the constraints specified by equations (6), (5), (4) (3) and (2),
- modeling the overall structure of constraints with a hypergraph; in our case this is a special, tri-partite graph visualizing constraint, decimal variables and binary variables,
- turning constraints into value-propagation rules, and
- finding a *minimal branching plan* with constraint propagation for efficient solution of the problem.

The key issue is to combine constraint structure with value-propagation rules, so that a *minimal branching plan* is obtained. In fact, every plan can be assigned some *entropy value*. For intuition, plan with high entropy is likely to require a lot of search and backtracking. A plan with low entropy is likely to enforce little backtracking. A plan with zero entropy can perhaps be executed in a linear way.

Obviously, calculating precise entropy value would be problematic; not only one has to assign probabilities to all choices of variable values, but the entropy should be considered conditional. For every choice of variable value, restrictions of other values should be propagated, and this is time and calculation costly. Hence, we apply a simplified procedure, when a new variable to be assigned a value is selected according to the following intuitive principles:

- variables allowing for direct calculation of as many other variables as possible are preferred,
- variables with restricted domains are preferred over ones with wide domains,
- variables influencing as many constraints as possible are preferred.

In Figure 1 the overall structure of the constraints is presented. There are five nodes representing five constraints, namely: M, MSO, ONE, NER and EDY. The variables involved in each constraint are linked to these nodes by arrows.

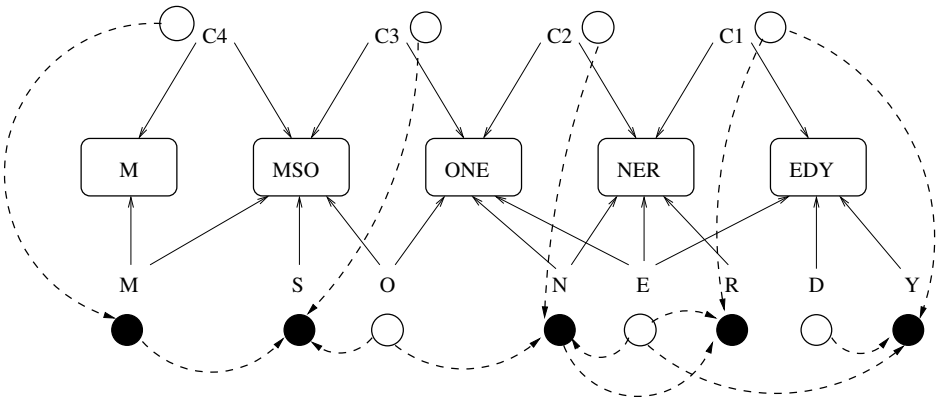

Fig. 1. *A hypergraph presenting the structure of local constraints.*

Now, the idea for building a minimal entropy plan is as follow:

- we start with variable C4; its domain is well-restricted (just 0 or 1), and after a choice it directly determines the value of M. The dashed lines show the plan. An empty circle indicates, that the value of a variable should be selected in a nondeterministic way, and backtracking may take place. The filled-black circles indicate that a variable can be determined in a unique way, if values of preceding variables are known.
- having C4 and M, we choose *O*; its value must be selected, but O is involved in two constraints, namely MSO and ONE. By turning MSO into an inference rule and selecting C3 we determine the value of S.

– the next selected variable is E and C2. Having them, and using constraint ONE turned into a propagation rule we can determine N;
– now we choose the value of D and C1, and using EDY we determine the value of Y.

Let us evaluate the number of possible paths. The branching points are: C4/2, O/9, C3/2, E/8, C2/2, D/7, C1/2; after the slash we show the number of possible value choices. So we have 2*2*2*2*9*8*7=8064.

In comparison to the naive approach (10^8 paths) we have reduction to 0.008064%, and in the case of immediate forward checking ($10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 = 1814400$) it is 0.444444%.

Now look at the program and the results. A simple PROLOG program implementing this plan is given below:

```
smm_rules_opt :-
    Cars = [0,1],
    Digits = [0,1,2,3,4,5,6,7,8,9],
    member(C4,Cars),   %%% Choice C4/2
    M = C4, M>0, select(M,Digits,DM),
    member(C3,Cars),   %%% Choice C3/2
    select(O,DM,DO),   %%% Choice O/9
    S is 10*C4 + O -C3 -M, S>0, select(S,DO,DS),
    member(C2,Cars),   %%% Choice C2/2
    select(E,DS,DE),   %%% Choice E/8
    N is E + O + C2 - 10*C3, select(N,DE,DN),
    member(C1,Cars),   %%% Choice C1/2
    R is E + 10*C2 - C1 - N, select(R,DN,DR),
    select(D,DR,DD),   %%%  Choice D/7
    Y is D + E - 10*C1, select(Y,DD,_),
    X=[S,E,N,D,M,O,R,Y],
    write(X).
```

The results are astonishing:

```
?- time(smm_rules_opt).
[9, 5, 6, 7, 1, 0, 8, 2]
% 364 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
true.
```

Similar results can be expected for plans of similar entropy.

6. CONCLUDING REMARKS

An attempt at providing a logical model for constraint programming is presented. This model incorporates a hypergraph for constraint modelling and rules for constraint propagation. A strategy for planning the search should minimize the entropy function. Although the program incorporates some hand-encoding structure based on intuitions, early results of practical experiments seems encouraging.

Further research may be oriented towards development of methods for more precise evaluation of entropy and semi-automatic or automatic planning for a solution-finding strategy. Moreover, more efficient decomposition of constraints into rules and use of different types of rules (more general rules may cover the case of domain restriction) should be explored.

ACKNOWLEDGMENTS

Research carried out within AGH-UST internal research project No.: 11.11.120.859.

REFERENCES

Apt, K.R., 2006. *Principles of Constraint Programming.* Cambridge University Press, Cambridge, UK.

Bratko, I., 2000. *Prolog Programming for Artificial Intelligence.* Addison Wesley, 3rd edition.

Dechter, R., 2003. *Constraint Processing.* Morgan Kaufmann Publishers, San Francisco, CA.

Ligęza, A. 2009a. And-or graph with knowledge propagation rules as a model for constraint satisfaction problems. *Automatyka*, **13**(2), 411–419.

Ligęza, A. 2009b. A constraint satisfaction framework for diagnostic problems. In Zdzisław Kowalczuk, editor, *Diagnosis of processes and systems*, volume 7 of *Control and Computer Science : information technology, control theory, fault and system diagnosis*, Pomeranian Science and Technology Publishers PWNT, Gdańsk, Poland, pp. 255–262.

Ligęza, A., Kościelny, J.M., 2008. A new approach to multiple fault diagnosis. combination of diagnostic matrices, graphs, algebraic and rule-based models. the case of two-layer models. *Int. J. Appl. Math. Comput. Sci.*, **18**(4), 465–476.

Russell, S., Norvig, P., 2003. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 2nd edition.