Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 101–107

ISBN 978-83-60810-51-4

# Improving Efficiency in Constraint Logic Programming Through Constraint Modeling with Rules and Hypergraphs

Antoni Ligęza
AGH University of Science and Technology
al. Mickiewicza 30, 30-059 Kraków, Poland
Email: ligeza@agh.edu.pl

***Abstract*—Constraint Satisfaction Problems typically exhibit very strong combinatorial explosion of exponential nature. This is due to their intrinsic nature: a number of variables have to be assigned values from their domains. This induces a very large number of potential solutions to be explored. Most typical approaches are oriented towards reduction of the inevitable search through advanced constraint propagation methods. In this paper we analyze a possibility of improving efficiency in Constraint Logic Programming. A hypergraph model of constraints is proposed as a base tool for planning approach. Building the partial solution plan in the form of definite sequence of variables is performed a priori. The plan is executed with a classical backtrack search. The whole process is focused on efficient use of variable values propagation rules. Two example cryptoarithmetic problems are explored in order to explain the proposed approach. The reported results are amazing in comparison to contemporary tools.**

## I. Introduction

A *CONSTRAINT Satisfaction Problem* (CSP, for short) [1], [2], [3] is a task consisting in finding an assignment of values to a finite set of predefined variables. The assigned values are restricted to belong to predefined sets—variable domains. Each variable has a predefined domain, which can be finite, infinite but countable (a discrete set) or uncountable. The key issue is that the assignment must satisfy a set of *predefined constraints*. For a given problem there may exist a single solution (this is mostly the case of toy or testbed problems), a number of admissible solutions (where there is no preference relation established), or, there may be no solution at all. In the last case we speak about the so-called *over-constrained* problems.

Despite the conceptually simple statement, Constraint Satisfaction Problems typically exhibit very strong combinatorial explosion—of exponential nature. This is due to their intrinsic nature: a number of variables have to be assigned values from their domains. This induces a very large number of potential solutions to be explored; each of them must be tested for constraint satisfaction.

The basic approach consists in sequential exploration of the search space with use of *backtracking algorithm*. For intuition, variables are selected in turn, and each time a variable has been selected a value assignment is attempted. This leads in a natural way to a tree search procedure. Every time when inconsistency is detected—i.e. some of the constraints are inevitably unsatisfied—backtracking takes place, and remaining branches of the search tree are explored.

In order to make the search as efficient as possible one can consider how to answer the following questions [4]:

1) Which variable should be assigned next?
2) In what order should its values be tried?
3) Can we detect inevitable failure early?
4) Can we take advantage of problem structure?

Most typical approaches are oriented towards reduction of the inevitable search through advanced constraint propagation methods [1], [2], [3], [4]. Some most typical examples include: (i) selection of a Most Constrained Variable, also known as Minimum Remaining Values, choose the Least Constraining Value, propagation with Forward Checking or more complex Arc Consistency. The structure of the overall problem may turn out to be crucial for solution efficiency; especially problems with tree-structured constraint graphs, and ones with nearly-tree structured graphs can be solved in very efficient way.

Constraint processing is often done with logical methods and tools [2], [5], [6]. Practical solutions are based on PROLOG paradigm and notation. Built-in search an backtracking mechanism play important role in search for solution. Combined with efficient constraint propagation mechanisms they take the form of *Constraint Logic Programming*. A modern approach

This work was carried out within AGH Internal Research Project No 11.11.120.859.

and tools have evolved also into the *Answer Set Programming* (ASP, for short) paradigm [7].

There are many toy problems illustrating the ideas of CSP, for example the so-called cryptoarithmetic problems, logical puzzles, or assignment problems, e.g. the Einstein (or Zebra) problem. In fact, CSP is a common model for diversity of practical, theoretical, toy and entertainment problems. 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, although conceptually simple, is in fact 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 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. At least one solution must exist provided that it is not an over-constrained problem. The real problem is that CSP suffers from strong *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 [1].

Depending on the nature of the domains there are CSP formulations for *Finite Domains* (CSP over finite domains; Finite Domains are often shortened to FD) or continuous domains (e.g. over the rational numbers $R$). The finite domains can be binary ones (e.g. as in the SAT problem), composed of a finite subset of integers (e.g. digits) or symbolic ones (composed of symbols). In this paper we are focused on CSP over FD with mostly symbolic and numerical values.

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, (iii) propagation of constraints in order to reduce domains, and (iv) checking if some of the constraints are not satisfied at any stage when it is possible. If the constraints are not satisfied at some stage *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 order to improve efficiency heuristics based on the so-called *fitness function* or the *estimated distance to the goal* can be used if available.

In the literature the search is often organized with use of the *Constraint Graph*, i.e. a graph modeling the structure of the constraints [1]. In such a graph nodes represent the variables and vertexes 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 [8], [9], [10] 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 a further attempt at extending these techniques towards some two classical CSP problems is discussed.

If available, 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. Note that if a set of rules is available, *forward-chaining* can be applied, and if successful—it can sometimes lead to drastic reduction of search.

In this paper we analyze a possibility of improving efficiency in Constraint Logic Programming. A hypergraph model of constraints is proposed as a base tool for *planning approach*. Building the partial solution plan in the form of definite sequence of variables is performed as an a priori step. The plan is executed with a classical backtrack search. The whole process is focused on efficient use of variable values propagation rules. Two example cryptoarithmetic problems are explored in order to explain the proposed approach. The reported results are amazing in comparison to contemporary tools.

The proposed approach is illustrated with practical solving an example of a well-known cryptoarithmetic problem. In [11] we proposed a hypergraph model for modeling constraints and using rules for efficient propagation of variables values. It turned out, that this simple planning model led to fine efficiency of the target program, although the results were obtained for a classical, single cryptoarithmetic problem `SEND+MORE=MONEY`. The size of the space for this problem is $10^8$, and the final program found the solution with only 364 inferences compared to thousands up to millions of inferences in other approaches. In this paper we improve this result and show experiment with a slightly bigger problem of the size $10^{10}$.

## II. Problem statement

Let us briefly recall the basic problem statement [1], [2], [3]. The presentation is borrowed from [11].

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 a 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 used 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 more 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.

## III. Recapitulation of former results

Constraint satisfaction models and techniques present relatively matured level; for the state-of-the-art consult [1], [2] and for a nice review of modern approaches to constraint propagation [4] (Chapter 5). Here we examine some of them in solving a cryptoarithmetic problem.

Consider the following well-known cryptoarithmetic puzzle [2]:

```
  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 have shown in [11] 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:

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

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, additional variables representing the carry signal have to be introduced. We have:

$$\mathbf{M}: \quad M = C4, \tag{1}$$

$$\mathbf{MSO}: \quad S + M + C3 = 10 * C4 + O, \tag{2}$$

$$\mathbf{ONE}: \quad E + O + C2 = 10 * C3 + N, \tag{3}$$

$$\mathbf{NER}: \quad N + R + C1 = 10 * C2 + E \tag{4}$$

$$\mathbf{EDY}: \quad D + E = 10 * C1 + Y, \tag{5}$$

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

In [11] we have proposed to model the constraints with a hypergraph; the basic principles are as follows:

- modeling the constraints with as many as detailed constraints, as possible; in our case we shall use the constraints specified by equations (1), (2), (3) (4) and (5),
- modeling the overall structure of constraints with a hypergraph; in our case this is a special, tripartite 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 constraint as possible are preferred.

In Fig. 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. 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 non-deterministic 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 $0.444444\%$.

Below we present a simple PROLOG program obtained through the proposed planning method (a slightly improved version with respect to [11].

The current results of the program are as follows:

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

i.e. we arrived at 272 inferences instead of 364 reported in [11]. Now, the question is if this approach can work for even bigger spaces. Below, we consider application of the proposed technique to the DONALD+GERALD=ROBERT problem with 10 variables, and the potential space of candidate solutions of the size $10^{10}$.

## IV. ANOTHER CASE STUDY

Consider the following well-known cryptoarithmetic puzzle [2]:

```
   DONALD
 + GERALD
 ------
   ROBERT
```

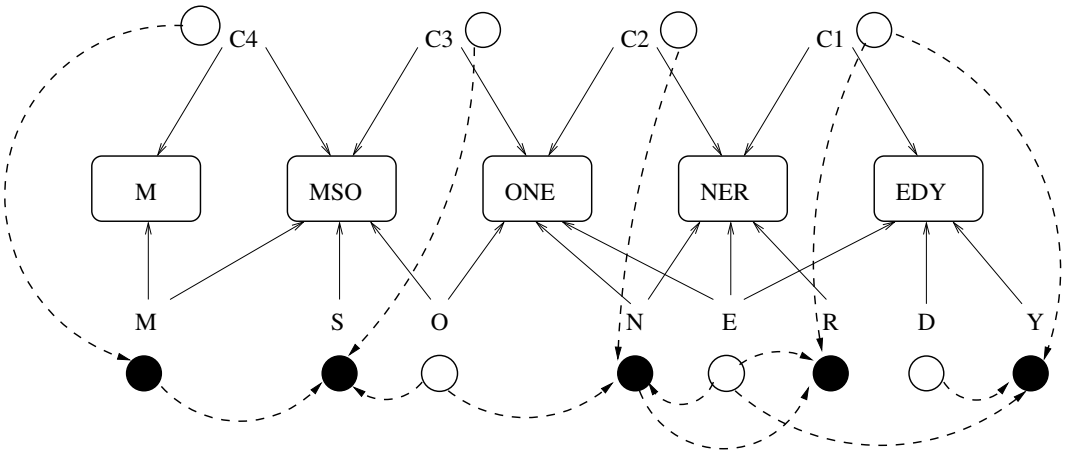


Fig. 1. A hypergraph presenting the structure of local constraints

**Algorithm 1** A PROLOG program with efficient variable ordering and rules for variable values propagation

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

This time we have 10 variables: `D,O,N,A,L G,E,R,B,T`. All of them are to be assigned different digits so that the above constraint is satisfied. Leading variables are different from 0.

We have also the following equations:

$$\mathbf{DGR}: \quad D+G+C5=R \tag{6}$$

$$\mathbf{OEO}: \quad O+E+C4=O+10*C5 \tag{7}$$

$$\mathbf{NRB}: \quad N+R+C3=B+10*C4 \tag{8}$$

$$\mathbf{AAE}: \quad A+A+C2=E+10*C3 \tag{9}$$

$$\mathbf{LLR}: \quad L+L+C1=R+10*C2 \tag{10}$$

$$\mathbf{DDT}: \quad D+D=T+10*C1 \tag{11}$$

These constraints are modeled with a hypergraph shown in Fig. 2. Now, the idea for building an efficient, low-branching plan is as follow:

- we start with selection of C5 and C4; now E can be calculated from equation () and removed from the domain;
- next we select C2 and C3; now A can be calculated from equation () and removed from the domain;
- now we select C1 and L; Le is removed from the domain and R is calculated from equation () and removed from the domain;
- next D is selected and removed from the domain; in consequence T be calculated from equation () and G be calculated from equation (); both the values are removed from the domain;
- next N is selected and removed from the domain; B can be be calculated from equation () and removed from the domain;
- finally, the last remaining values is O.

Let us evaluate the number of possible paths. The branching points are: C5/2, C4/2, C3/2, C2/2, C1/2, L/8, D/6, N/3; after the slash we show the number

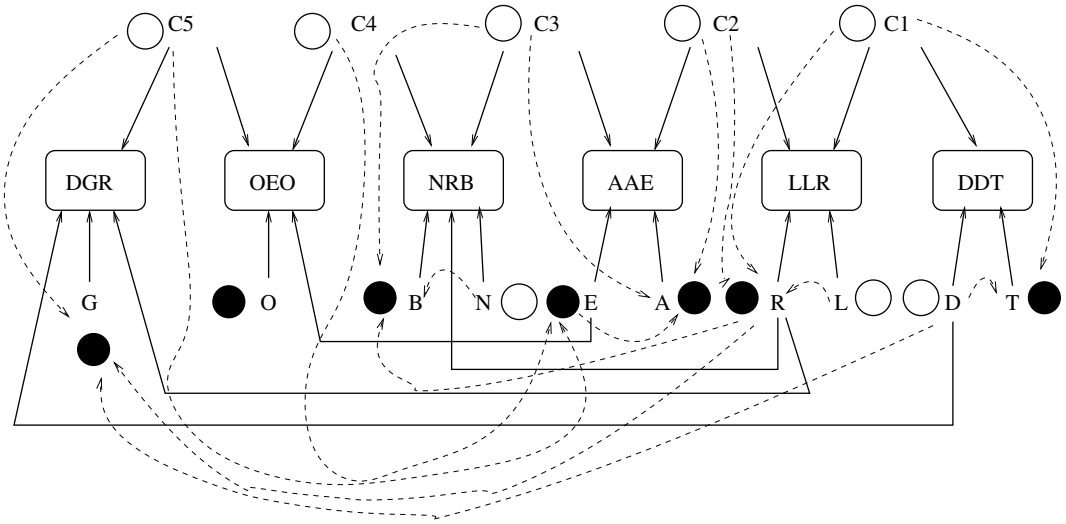


Fig. 2. A hypergraph presenting the structure of local constraints

of possible value choices at the choice-point. So we have 2*2*2*2*2*8*6*3 = 4608. In comparison to the naive approach ($10^{10}$ paths) we have reduction to $0.00004608\%$, and in the case of immediate forward checking ($10*9*8*7*6*5*4*3*2*1 = 3628800$) it is $0.1269841\%$.

Below we present a simple PROLOG program obtained through the proposed planning method. The performance of this program is reported below.

```
?- time(donald_opt).
[5, 2, 6, 4, 8, 1, 9, 7, 3, 0]
[1, 1, 0, 1, 1]
% 1,411 inferences, 0.00 CPU
  in 0.00 seconds
  (0% CPU, Infinite Lips)
true.
```

i.e. we arrived at 1,411 inferences, which seems not to bee so bad. For comparison, a classical but very efficient algorithm presented in [12] needs 17,498 inferences.

For comparison, we present also a solution of the problem with the CLP library of SWI Prolog. The code is listed below. The performance of this program is reported below.

```
?- time(solve_donald(V,C)).
% 502,712 inferences, 0.210 CPU
  in 0.224 seconds
  (94% CPU, 2393867 Lips)
V = [5, 2, 6, 4, 8, 1, 9, 7, 3, 0],
C = [1, 1, 0, 1, 1].
```

It is also worth pointing out that an attempt to solve this problem with the library for CLP of SWI Prolog with the same set of constraints led to around 500,000 inferences.

## V. CONCLUDING REMARKS

This paper reports on some introductory research and preliminary results focused on improving efficiency of constrained problem solving. Two simple, but illustrative examples are explored in-depth. The results are amazing.

The main focus was on providing a hypergraph model for improving efficiency of constraint logic programming with pure PROLOG. The main idea is to build a plan, so that rules for value propagation can be used in a most efficient manner. In other words, we have a two-phase approach: (i) the planning phase with use of the hypergraph model, and (ii) the execution phase. The plan itself can be described as minimum-branching plan assuring a most efficient value propagation.

This approach incorporates a hypergraph for constraint modeling and rules for constraint propagation. A strategy for planning the search should in fact minimize the entropy describing the set of unassigned variables. Although the program incorporates some hand-encoding structure based on intuitions, early results of practical experiments seems promising.

Further research may be oriented towards development of methods for 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.

## ACKNOWLEDGMENT

The research is carried out within AGH University of Science and Technology internal research project No. 11.11.120.859.

**Algorithm 2** Prolog code for the DONALD + GERALD = ROBERT cryptoarithmetic problem

```
donald_opt :-
    Cars = [0,1],
    Digits = [0,1,2,3,4,5,6,7,8,9],
%%% Choice C5/2
    member(C5,Cars),
%%% Choice C4/2
    member(C4,Cars),
    E is 10*C5 - C4,
    select(E,Digits,DE),
%%% Choice C2/2
    member(C2,Cars),
%%% Choice C3/2
    member(C3,Cars),
    A is (E + 10*C3 - C2)/2,
    select(A,DE,DA),
%%% Choice C1/2
    member(C1,Cars),
%%% Choice L/8
    select(L,DA,DL),
    R is 2*L + C1 - 10*C2,
    select(R,DL,DR),
%%% Choice D/6
    select(D,DR,DD),
    T is 2*D - 10*C1,
    select(T,DD,DT),
    G is R - D - C5,
    select(G,DT,DG),
%%% Choice N/3
    select(N,DG,DN),
    B is N + R + C3 - 10*C4,
    select(B,DN,DB),
    DB = [O],
    X = [D,O,N,A,L,G,E,R,B,T],
    write(X), nl,
    C = [C5,C4,C3,C2,C1],
    write(C).
```

**Algorithm 3** Prolog with CLP library code for the DONALD+GERALD=ROBERT cryptoarithmetic problem

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

donald(Vars,Cars) :-
    Cars = [C1,C2,C3,C4,C5],
    Cars ins 0..1,
    Vars = [D,O,N,A,L,G,E,R,B,T],
    Vars ins 0..9,
    all_different(Vars),
    O + E + C4    #= O + 10*C5,
    2*A + C2 #= E + 10*C3,
    2*L + C1 #= R + 10*C2,
    2*D      #= T + 10*C1,
    D + G + C5 #= R,
    N + R + C3 #= B + 10*C4.

solve_donald(Vars,Cars) :-
    Cars = [C1,C2,C3,C4,C5],
    Vars = [D,O,N,A,L,G,E,R,B,T],
    donald([D,O,N,A,L,G,E,R,B,T],
           [C1,C2,C3,C4,C5]),
    label(Vars),
    label(Cars).
```

## REFERENCES


[1] R. Dechter, *Constraint Processing*. San Francisco, CA: Morgan Kaufmann Publishers, 2003.

[2] K. R. Apt, *Principles of Constraint Programming*. Cambridge, UK: Cambridge University Press, 2006.

[3] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming*. Elsevier, 2006.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice-Hall, 2010.

[5] K. R. Apt and M. G. Wallace, *Constraint Programming using ECL$^i$PS$^e$*. Cambridge, UK: Cambridge University Press, 2007.

[6] A. Niederliński, *A Quick and Gentle Guide to Constraint Logic Programming via ECL$^i$PS$^e$*. Gliwice: Jacek Skalmierski Computer Studio, 2011.

[7] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[8] A. Ligęza and J. M. Kościelny, "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.*, vol. 18, no. 4, pp. 465–476, 2008.

[9] A. Ligęza, "A constraint satisfaction framework for diagnostic problems," in *Diagnosis of processes and systems*, ser. Control and Computer Science : information technology, control theory, fault and system diagnosis, Z. Kowalczuk, Ed. Gdańsk, Poland: Pomeranian Science and Technology Publishers PWNT, 2009, vol. 7, pp. 255–262.

[10] ——, "And-or graph with knowledge propagation rules as a model for constraint satisfaction problems," *Automatyka*, vol. 13, no. 2, pp. 411–419, 2009.

[11] ——, "Models and tools for improving efficiency in constraint logic programming," *Decision Making in Manufacturing and Services*, vol. 5, no. 1-2, pp. 68–78, 2011, http://www.dmms.agh.edu.pl.

[12] I. Bratko, *Prolog Programming for Artificial Intelligence*, 3rd ed. Addison Wesley, 2000.