Chapter 4: Basic Constraint Reasoning (SEND+MORE=MONEY)

Helmut Simonis

email: h.simonis@4c.ucc.ie
homepage: http://4c.ucc.ie/~hsimonis

Cork Constraint Computation Centre
Computer Science Department
University College Cork
Ireland

ECLiPSe ELearning Overview

Outline

Contents

What we want to introduce

- Finite Domain Solver in ECLiPSe
- Models and Programs
- Constraint Propagation and Search
- Basic constraints: linear arithmetic, alldifferent, disequality
- Built-in search: Labeling
- Visualizers for variables, constraints and search

Welcome to chapter 4 of the ECliPSe ELearning course. In this chapter, we look at the finite domain solver of the ECLiPSe system. We introduce the basic concepts of domain variables and constraints, propagation and search. As an example, we use the well-known SEND+MORE=MONEY puzzle.

We start with a description of the problem and ways of modelling the problem with a simple ECLiPSe program. We then look in more detail at the constraint setup, how the constraints deduce initial information about the problem. Propagation alone is not enough to solve the puzzle, we show how search interleaved with constraint propagation finds the solution of the puzzle. We conclude with a set of lessons learned, general principles that we can extract from this simple example.

What we want to introduce in this chapter is the finite domain solver in ECLiPSe, provided by the `ic` library. We talk about models and programs, and how they can be used to solve problems. We introduce the concepts of constraint propagation and search, and show some basic constraints for linear arithmetic, the `alldifferent` and disequality constraints. We also use a built-in for search, `labeling`, which assigns values to variables. Throughout the lesson we will use visualizers for variables, constraints and the search tree to understand what is happening inside the ECLiPSe system when solving this problem.

1 Problem

Let's start with the problem.

Figure 1: Problem Definition

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

SEND+MORE=MONEY is a crypt-arithmetic puzzle, where characters encode numbers and we have to deduce which character stands for which digit. It is one of the classical, early examples of constraint programming, showing how the constraint reasoning mimicks the human reasoning when solving this type of problem. The arithmetic problem is usually shown in the form of a handwritten addition. The puzzle is defined by the following rules:

Rules

- Each character stands for a digit from 0 to 9.
- Numbers are built from digits in the usual, positional notation.
- Repeated occurrence of the same character denote the same digit.
- Different characters denote different digits.
- Numbers do not start with a zero.
- The equation must hold.

Listing 1: Program Sendmory

```
1  \index{sendmory}
2  \index{module!sendmory}
3  :-module(sendmory).
4  :-export(sendmory/1).
5  :-lib(ic).
6
7  sendmory(L):-
8        L = [S,E,N,D,M,O,R,Y],
9        L :: 0..9,
10       alldifferent(L),
11       S #\= 0,
12       M #\= 0,
13       1000*S + 100*E + 10*N + D +
14       1000*M + 100*O + 10*R + E #=
15       10000*M + 1000*O + 100*N + 10*E + Y,
16       labeling(L).
```

2 Program

If we want to model this problem as a finite domain constraint problem, we have to define variables and constraints. *Variables* range over *domains*, which describe the possible values they can take. In this section we consider only finite domains, indeed only finite subsets of natural numbers. We will consider constraint problems with non-integral domains in a later chapter.

An *assignment* is a mapping from the variables to values in their respective domains.

A *constraint* ranges over one or multiple variables and expresses a condition which must hold between these variables. Constraints can take many forms, they may be explicit, describing combinations of values that are allowed or excluded, or symbolic, for example in the form of arithmetic constraints. Formally, we can see a constraint as a subset of the Cartesian product of the domains of its variables. An assignment *satisfies* a constraint if its projection on the variables of the constraint belong to this constrained subset.

An assignment of values to variables is a *solution* if all constraints are satisfied.

For many problems, we have considerable choices in selecting the variables and constraints. Often, models differ dramatically in the number of variables needed, the complexity and number of constraints used to describe the problem, and the speed by which a constraint solver can find a solution.

For the small puzzle considered here, the choice of the variables is fairly obvious: Each character is a variable, which ranges over the values 0 to 9.

For the constraints, a natural description would be:

- An *alldifferent* constraint between all variables, which states that two different variables must have different values. This is a very common constraint, which we will encounter in many other problems later on.
- Two *disequality constraints* (variable X must be different from value V) stating that the variables at the beginning of a number can not take the value 0.
- An arithmetic *equality constraint* linking all variables with the proper coefficients and stating that the equation must hold.

Remember that this is not the only model of this problem, we will discuss some alternatives later on.

The program shown in Listing 1 is an example of a finite domain constraint program in ECLiPSe, which implements our model above.

Line 3 shows the module declaration. Each file should be a module, and there should be one module per file. All predicates defined in a module are only visible inside the module, unless they are exported.

Line 4 shows an export directive. It states that the predicate `sendmory` with *arity* one (with one argument) should be exported from this module.

Line 5 contains a library directive. We are loading the `ic` library, which defines the *I*nterval *C*onstraints in the ECLiPSe language.

Lines 7-16 define the `sendmory` predicate which is the core of our model. It consists of a single clause with the clause head `sendmory(L).` The clause has a single argument L, which is a list of variables as defined in line 8. The square brackets denote lists in Prolog.

We next define the domain of the variables in line 9. The infix operator `::` takes a variable or list of variables as the first (left) argument, and a domain (here `0..9`) as the second (right) argument.

We then express the `alldifferent` constraint in line 10. This constraint is a built-in constraint in the `ic` library, and is thus shown in bold.

Lines 11-12 express the two disequality constraints. The `ic` library defines the operator `#\=` for this purpose.

The big linear equality constraint is expressed in line 14. The operator `#=` is used in the `ic` library.

Finally, we call the builtin search routine `labeling` to assign values to the variables.

Choice of Model

- This is *one* model, not *the* model of the problem
- Many possible alternatives
- Choice often depends on your constraint system
  - Constraints available
  - Reasoning attached to constraints
- Not always clear which is the *best* model
- Often: Not clear what is the *problem*

The model we have presented here is *one* model of the problem, it is not *the* model of the puzzle. Quite often there is a natural way of expressing the problem, but typically there are still many possible alternatives for expressing particular aspects of a model.

The choice which model to use often depends on your constraint system. A system may or may not contain particular constraints, some constraint system have constraint that others don't have and which are difficult to implement in another system. Some constraint systems have special reasoning attached to some constraints, so that they are much more powerful solving some problems than other systems.

Unfortunately, it is not always clear which is the *best model*, or if there is indeed a best model for a given problem class. Quite often, we don't even know what exactly the problem is, and we can decide which constraints we want to include and which aspects of a problem we can ignore.

One important aspect of constraint programming is that we want to play with the model, experiment by adding or removing some constraints, to really find out which problem we are going to solve and how we are going to do it.

Indeed, for the puzzle here we have developed two alternative models, which are described at the end of this chapter. There are hyperlinks on the slides to look at these alternatives, or you can use the table of contents of the video to look at them now.

Running the program

- To run the program, we have to enter the query
  - `sendmory:sendmory(L).`
- Result
  - `L = [9, 5, 6, 7, 1, 0, 8, 2]`
  - `yes (0.00s cpu, solution 1, maybe more)`

If we want to run the program, we have to enter a query `sendmory:sendmory(L).` which calls the predicate `sendmory` in the module `sendmory` with a single argument, a free variable L.

The system then returns the answer as a binding for the variable L, and it says `yes`, indicating that it has found a first solution. There may be additional solutions, which can be obtained by backtracking.

Question

- But how did the program come up with this solution?

How did the program actually find this solution, what happened in the constraint system before it printed out this message? That is the question we are going to answer in the following sections.

Usually, we are not going to analyze the behaviour of a program in such great detail, but for an introduction I think it is useful to understand how much work can happen inside the constraint engine and how variables, constraints and search interact to find a solution.

3 Constraint Setup

We will now go through the constraint setup, where the constraints are created and perform their initial propagation before the search is started. We are going to use different visualization tools to understand what is happening, this gives a much better conceptual model than tracing through the execution with the line debugger.

3.1 Domain Definition

Domain Definition

```
L = [S,E,N,D,M,O,R,Y],
L :: 0..9,
```

$$[S, E, N, D, M, O, R, Y] \in \{0..9\}$$

When we first encounter our list of variables, we define them to be domain variables with a domain from 0 to 9.

Figure 2: Domain Visualization

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

We can represent the variables with a *domain visualizer* (Figure 2). It shows the state of the domain variables at this point. Each line shows one variable, each column one value. Initially, all values are allowed for all variables, this is indicated by white cells.

3.2 Alldifferent Constraint

Alldifferent Constraint

```
alldifferent(L),
```

- Built-in of `ic` library
- No initial propagation possible
- *Suspends*, waits until variables are changed
- When variable is fixed, remove value from domain of other variables

- *Forward checking*

The next constraint we encounter is the `aldifferent` constraint. This constraint states that all variables in its argument list must be pairwise different, no two variables can take the same value. This is a built-in constraint of the `ic` library, which is used in many models. We will encounter it again in the next chapter.

When the `alldifferent` constraint is first called, all variables have the same domain, none of the variables are assigned. At this point, the constraint can not do any propagation, it therefore just suspends and waits until one of its variables is changed either by other constraints or as part of the search routine.

When one of the variables becomes assigned, the constraint will wake up and remove this value from the domain of all other variables. This reasoning is called *forward checking* and is the most basic reasoning technique for the `alldifferent` constraint.

Figure 3: `alldifferent` visualizer

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

After having setup the constraint, we can use a *constraint visualizer* to see its status. This looks just like the domain visualizer, but will provide more information as we progress.

3.3 Disequality Constraints

The program now handles the two disequality constraints. As each constraint only deals with a single variable, we can solve them by removing the value from the domain of the variable. This leads to an update in two of the domains:

$$S \in \{1..9\}, M \in \{1..9\}$$

The two disequality constraint are now solved, they can be discarded.

Table 4 shows the state of the variables after these steps, two cell have been colored gray to indicate that the values have been removed.

Figure 4: Domains after Disequality

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

3.4 Equality Constraint

The next constraint encountered is the equality constraint in line 14. The processing of this constraint is more complex, we start with a normalisation of the constraint.

The constraint as stated contains multiple occurrences of the same variable. In a first step, we reduce these multiple occurences by combining their coefficients. For didactic purposes, we avoid negative coefficients by placing terms either on the left or the right hand side of the equation.

	1000*S+	100*E+	10*N+	D
	+1000*M+	100*O+	10*R+	E
10000*M+	1000*O+	100*N+	10*E+	Y

is transformed into

	1000*S+	91*E+		D
		+	10*R	
9000*M+	900*O+	90*N+		Y

Simplified Equation

$$1000 * S + 91 * E + 10 * R + D = 9000 * M + 900 * O + 90 * N + Y$$

We now look at the initial propagation of the equality constraint, while noting that the `alldifferent` constraint is suspended, but will be woken when variables are fixed to values.

Propagation

We consider the two sides of the equation, taking the domains of the variables into account.

$$1000 * S^{1..9} + 91 * E^{0..9} + 10 * R^{0..9} + D^{0..9} =$$
$$9000 * M^{1..9} + 900 * O^{0..9} + 90 * N^{0..9} + Y^{0..9}$$

The left hand side ranges from 1000 to 9918, the right hand side from 9000 to 89919.

$$\underbrace{1000 * S^{1..9} + 91 * E^{0..9} + 10 * R^{0..9} + D^{0..9}}_{1000..9918} =$$
$$\underbrace{9000 * M^{1..9} + 900 * O^{0..9} + 90 * N^{0..9} + Y^{0..9}}_{9000..89919}$$

If the equation holds, then both sides must be equal. We can therefore consider the intersection of the ranges for left and right hand side.

$$\underbrace{1000 * S^{1..9} + 91 * E^{0..9} + 10 * R^{0..9} + D^{0..9}}_{9000..9918} =$$
$$\underbrace{9000 * M^{1..9} + 900 * O^{0..9} + 90 * N^{0..9} + Y^{0..9}}_{9000..9918}$$

From this we can deduce the following domain restrictions:

Deduction:

$$M = 1, S = 9, O \in \{0..1\}$$

Consider lower bound for S

- Lower bound of equation is 9000
- Rest of lhs (left hand side) $(91 * E^{0..9} + 10 * R^{0..9} + D^{0..9})$ is atmost 918
- S must be greater or equal to $\frac{9000-918}{1000} = 8.082$
  - otherwise lower bound of equation not reached by lhs
- S is integer, therefore $S \geq \lceil \frac{9000-918}{1000} \rceil = 9$
- S has upper bound of 9, so $S = 9$

Consider upper bound of M

- Upper bound of equation is 9918
- Rest of rhs (right hand side) $900 * O^{0..9} + 90 * N^{0..9} + Y^{0..9}$ is at least 0
- M must be smaller or equal to $\frac{9918-0}{9000} = 1.102$
- M must be integer, therefore $M \leq \lfloor \frac{9918-0}{9000} \rfloor = 1$
- M has lower bound of 1, so $M = 1$

Consider upper bound of O

- Upper bound of equation is 9918
- Rest of rhs (right hand side) $9000 * 1 + 90 * N^{0..9} + Y^{0..9}$ is at least 9000
- O must be smaller or equal to $\frac{9918-9000}{900} = 1.02$
- O must be integer, therefore $O \leq \lfloor \frac{9918-9000}{900} \rfloor = 1$
- O has lower bound of 0, so $O \in \{0..1\}$

Figure 5: Propagation of equality: Result

	0	1	2	3	4	5	6	7	8	9
S		-	-	-	-	-	-	-	-	✹
E										
N										
D										
M		✹	-	-	-	-	-	-	-	-
O			✖	✖	✖	✖	✖	✖	✖	✖
R										
Y										

So after the initial reasoning of the equality constraint (Figure 5), we have deduced that two variables (S and M) must have fixed values and that another variable O can only range between 0 and 1.

Figure 6: Propagation of `alldifferent`

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

The assignment of the variables will wake up the `alldifferent` constraint. This removes the assigned values 1 and 9 from the domain of the unassigned variables. But variable O could only take values 0 or 1. If we remove value 1, then it must take value 0. This assignment will be propagated through the `alldifferent` constraint and removes the value 0 from the remaining variables (Figure 6.

At this point the equality constraint will be checked again, since some variable bounds have been updated.

For ease of presentation, we first remove the constant parts of the constraint, and obtain a simplified form:

Removal of constants

$$1000 * 9 + 91 * E^{2..8} + 10 * R^{2..8} + D^{2..8} = \\ 9000 * 1 + 900 * 0 + 90 * N^{2..8} + Y^{2..8}$$

$$\mathbf{1000} * \mathbf{9} + 91 * E^{2..8} + 10 * R^{2..8} + D^{2..8} = \\ \mathbf{9000} * \mathbf{1} + \mathbf{900} * \mathbf{0} + 90 * N^{2..8} + Y^{2..8}$$

$$91 * E^{2..8} + 10 * R^{2..8} + D^{2..8} = 90 * N^{2..8} + Y^{2..8}$$

We now go though several steps of updating variable bounds, each triggering a renewed check of the equality constraint.

Propagation of equality (Iteration 1)

$$\underbrace{91 * E^{2..8} + 10 * R^{2..8} + D^{2..8}}_{204..816} = \underbrace{90 * N^{2..8} + Y^{2..8}}_{182..728}$$

$$\underbrace{91 * E^{2..8} + 10 * R^{2..8} + D^{2..8}}_{204..728} = 90 * N^{2..8} + Y^{2..8}$$

$$N \geq 3 = \lceil \frac{204 - 8}{90} \rceil, E \leq 7 = \lfloor \frac{728 - 22}{91} \rfloor$$

Propagation of equality (Iteration 2)

$$91 * E^{2..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{3..8} + Y^{2..8}$$

$$\underbrace{91 * E^{2..7} + 10 * R^{2..8} + D^{2..8}}_{204..725} = \underbrace{90 * N^{3..8} + Y^{2..8}}_{272..728}$$

$$\underbrace{91 * E^{2..7} + 10 * R^{2..8} + D^{2..8}}_{272..725} = 90 * N^{3..8} + Y^{2..8}$$

$$E \geq 3 = \lceil \frac{272 - 88}{91} \rceil$$

Propagation of equality (Iteration 3)

$$91 * E^{3..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{3..8} + Y^{2..8}$$

$$\underbrace{91 * E^{3..7} + 10 * R^{2..8} + D^{2..8}}_{295..725} = \underbrace{90 * N^{3..8} + Y^{2..8}}_{272..728}$$

$$\underbrace{91 * E^{3..7} + 10 * R^{2..8} + D^{2..8}}_{295..725} = 90 * N^{3..8} + Y^{2..8}$$

$$N \geq 4 = \lceil \frac{295 - 8}{90} \rceil$$

Propagation of equality (Iteration 4)

$$91 * E^{3..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{4..8} + Y^{2..8}$$

$$\underbrace{91 * E^{3..7} + 10 * R^{2..8} + D^{2..8}}_{295..725} = \underbrace{90 * N^{4..8} + Y^{2..8}}_{362..728}$$

$$\underbrace{91 * E^{3..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{4..8} + Y^{2..8}}_{362..725}$$

$$E \geq 4 = \lceil \frac{362 - 88}{91} \rceil$$

Propagation of equality (Iteration 5)

$$91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{4..8} + Y^{2..8}$$

$$\underbrace{91 * E^{4..7} + 10 * R^{2..8} + D^{2..8}}_{386..725} = \underbrace{90 * N^{4..8} + Y^{2..8}}_{362..728}$$

$$\underbrace{91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{4..8} + Y^{2..8}}_{386..725}$$

$$N \geq 5 = \lceil \frac{386 - 8}{90} \rceil$$

Propagation of equality (Iteration 6)

$$91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}$$

$$\underbrace{91 * E^{4..7} + 10 * R^{2..8} + D^{2..8}}_{386..725} = \underbrace{90 * N^{5..8} + Y^{2..8}}_{452..728}$$

$$\underbrace{91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}}_{452..725}$$

$$N \geq 5 = \lceil \frac{452 - 8}{90} \rceil, E \geq 4 = \lceil \frac{452 - 88}{91} \rceil$$

No further propagation at this point We have reached a fixed-point of the constraint propagation, all constraints have been checked, and no further domain reduction is possible.

Figure 7 shows the state of the variables after the constraint setup, three variables have been assigned, and the domains of the unassigned variables have been reduced.

Figure 7: Domains after setup

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

4 Search

We now call the `labeling` routine to assign values to the unassigned variables.

The predicate call `labeling([S,E,N,D,M,O,R,Y])` in line 16 on page 3 will enumerate the variables in the given order, each time starting with the smallest value in the domain. If the assignment of that value fails, the next value will be tried, and so on. If no more values remain, the search will backtrack to the last previous choice and try its next alternative. This will continue until either

- a valid assignment is found for all variables, and the search *succeeds*
- no more untested choices remain, the search *fails*

4.1 Step 1

Figure 8: Search Tree Step 1

Figure 8 shows the search tree at this point. The top node shows the variable being assigned (here S), the label on the edge shows the value that is taken (here 9). The link is dotted, as the variable is already assigned, and no new choice is required.

4.2 Step 2

Figure 9 shows the search tree at this point. We are currently assigning variable E to value 4. As this is a proper choice, a solid link is used.

Assigning E to 4 (Figure 10) will wake the equality constraint

Propagation of $E = 4$, equality constraint

$$91 * 4 + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}$$

$$\underbrace{91 * 4 + 10 * R^{2..8} + D^{2..8}}_{386..452} = \underbrace{90 * N^{5..8} + Y^{2..8}}_{452..728}$$

$$\underbrace{91 * 4 + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}}_{452}$$

$$N = 5, Y = 2, R = 8, D = 8$$

Figure 9: Step 2, Alternative $E = 4$

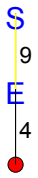


Figure 10: Assignment $E = 4$

	0	1	2	3	4	5	6	7	8	9
S										
E					✹	-	-	-		
N										
D										
M										
O										
R										
Y										

The state after propagation the equality constraint is shown in Figure 11.

The assignments will trigger the `alldifferent` constraint, but the constraint detects that two variables (D and R) have the same value 8. This is not allowed and the `alldifferent` constraint fails (Figure 12).

We backtrack to the last choice (E), and then try the next value ($E = 5$). This situation is depicted in figure 13. The left child of node E is marked as a failure, we are currently testing value 5, the second alternative. The assignment of value ($E = 5$) will trigger both the `alldifferent` and the equality constraint. Assume that the `alldifferent` is woken first[1].

The propagation of the assignment $E = 5$ in the `alldifferent` constraint will remove value 5 from the variable N (Figure 15). At this point, since the lower bound of variable N is updated to 6, the equality constraint is triggered.

Propagation of equality

$$91 * 5 + 10 * R^{2..8} + D^{2..8} = 90 * N^{6..8} + Y^{2..8}$$

$$\underbrace{91 * 5 + 10 * R^{2..8} + D^{2..8}}_{477..543} = \underbrace{90 * N^{6..8} + Y^{2..8}}_{542..728}$$

$$\underbrace{91 * 5 + 10 * R^{2..8} + D^{2..8} = 90 * N^{6..8} + Y^{2..8}}_{542..543}$$

[1] Which of the constraints is woken up first can be difficult to predict without understanding the details of the implementation of the solver

Figure 11: Result of equality propagation

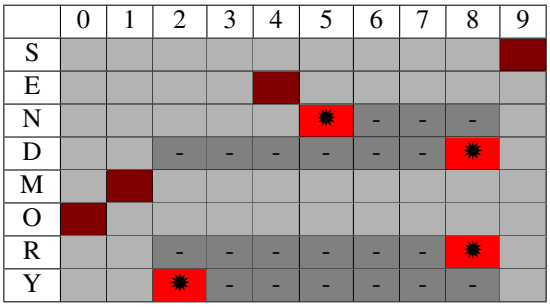
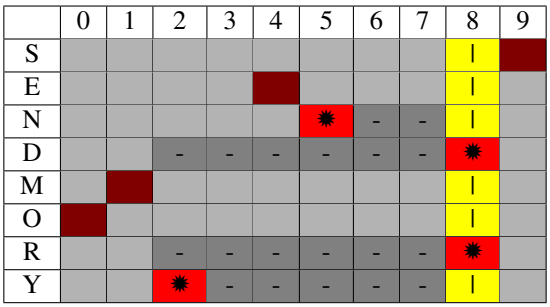


Figure 12: Propagation of `alldifferent` fails!



$$N = 6, Y \in \{2, 3\}, R = 8, D \in \{7..8\}$$

The `alldifferent` constraint (figure 17) has fixed variable D to 7. We again wake the equality constraint.

Propagation of equality

$$91 * 5 + 10 * 8 + 7 = 90 * 6 + Y^{2..3}$$

$$\underbrace{91 * 5 + 10 * 8 + 7}_{542} = \underbrace{90 * 6 + Y^{2..3}}_{542..543}$$

$$\underbrace{91 * 5 + 10 * 8 + 7 = 90 * 6 + Y^{2..3}}_{542}$$

$$Y = 2$$

This finally assigns the last variable Y to 2 (Figure 18).

4.3 Further Steps

All variables have been assigned, the search succeeds. Figure 19 shows the resulting search tree. There is only a single variable (E) for which an actual choice had to be made, all other variables are assigned by constraint propagation. The last node at the bottom of the tree is a green success node, indicating that a solution was found.

4.4 Solution

We finally reached a solution (Figure 20), it is easy to check that this indeed solves the puzzle.

Figure 13: Step 2, Alternative $E = 5$

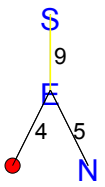


Figure 14: Assignment $E = 5$

	0	1	2	3	4	5	6	7	8	9
S										
E					-	✹	-	-		
N										
D										
M										
O										
R										
Y										

Figure 15: Propagation of `alldifferent`

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

Figure 16: Result of equality propagation

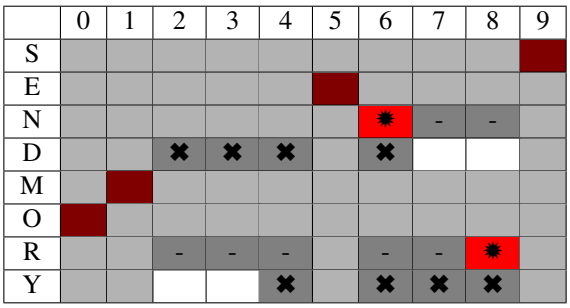


Figure 17: Propagation of `alldifferent`

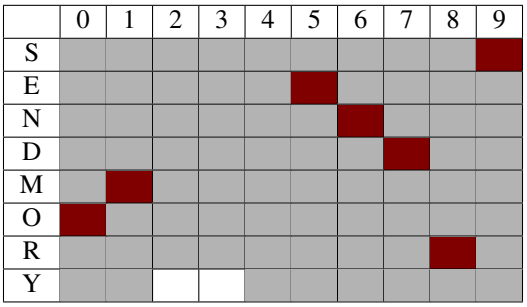


Figure 18: Last propagation step

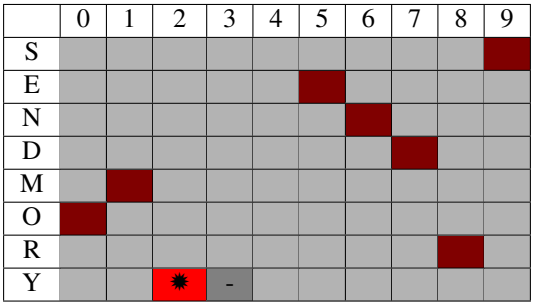


Figure 19: Complete Search Tree

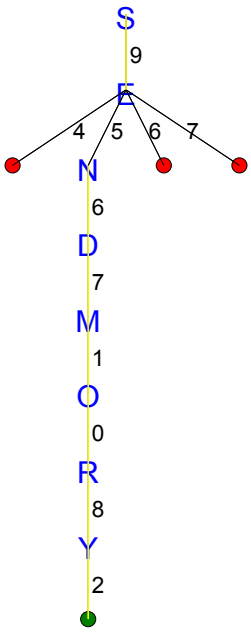


Figure 20: Solution

	9	5	6	7
+	1	0	8	5
1	0	6	5	2

5 Lessons Learned

Topics introduced

- Finite Domain Solver in ECLiPSe, `ic` library
- Models and Programs
- Constraint Propagation and Search
- Basic constraints: linear arithmetic, `alldifferent`, disequality
- Built-in search: `labeling`
- Visualizers for variables, constraints and search

We have introduced the finite domain solver for ECLiPSe, the `ic` library. We have shown a simple model for our example puzzle, and seen how that translates into an ECLiPSe program. The key elements of problem solving with constraint programming are constraint propagation and search, and we have shown some basic constraints, linear arithmetic, the `alldifferent` and disequality constraints. For search we used the `labeling` primitive which offers a simple depth-first search. To understand how the system found the solution we used different visualizers for variables, constraints and the search tree.

Lessons Learned

- Constraint models are expressed by variables and constraints.
- Problems can have many different models, which can behave quite differently. Choosing the best model is an art.
- Constraints can take many different forms.
- Propagation deals with the interaction of variables and constraints.
- It removes some values that are inconsistent with a constraint from the domain of a variable.
- Constraints only communicate via shared variables.

Lessons Learned

- Propagation usually is not sufficient, search may be required to find a solution.
- Propagation is data driven, and can be quite complex even for small examples.
- The default search uses chronological depth-first backtracking, systematically exploring the complete search space.
- The search choices and propagation are interleaved, after every choice some more propagation may further reduce the problem.

6 Alternative Models

We now have reached the end of this chapter, in this section we present two alternative models for the puzzle, and show how the results differ from the model we have used above.

6.1 Model without Disequality

We now consider an alternative model, that we might have used as our expression of the puzzle.

Alternative 1

- Do we need the constraint "Numbers do not begin with a zero"?
- This is not given explicitely in the problem statement
- Remove disequality constraints from program
- Previous solution is still a solution
- Does it change propagation?
- Does it have more solutions?

The model "Alternative1" looks at the question "Do we need to state that numbers do not begin with a zero?". This constraint is not explicitely given in the problem formulation, and we might decide that we do not want to model it. The impact of this change is to remove the disequality constraints from the program. This model then is a *relaxation* of our previous model. The solution we found before will still be a solution, but there may be others. We also can consider if removing the constraints changes the propagation, is it easier or more difficult to solve this relaxed problem?

Program without Disequality

Listing 2: Alternative 1

```
1  :-module(alternative1).
2  :-export(sendmory/1).
3  :-lib(ic).
4
5  sendmory(L):-
6        L = [S,E,N,D,M,O,R,Y],
7        L :: 0..9,
8        alldifferent(L),
9        1000*S + 100*E + 10*N + D +
10       1000*M + 100*O + 10*R + E #=
11       10000*M + 1000*O + 100*N + 10*E + Y,
12       labeling(L).
```

Here we have the program without the disequality constraints.

After the constraint setup, we reach the following state of the problem (Figure 21). We see that the domain reduction from the initial propagation is much weaker, only variable M has a reduced domain.

Figure 22 shows the difference to our initial model.

We see from the search tree (Figure 23) that our modified program finds many solutions, exploring many alternative choices for the variables.

Figure 21: After Setup without Disequality

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

Figure 22: Setup Comparison

original

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

alternative 1

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

Note:

- Not just a different model, solving a different problem!
- Often we can choose which problem we want to solve
  - Which constraints to include
  - What to ignore
- In this case not acceptable

Our modified program really solves a relaxation of the original problem, which has many more solutions. For many problems we can choose which problem we want to solve, and whether a relaxation is a good approximation of the intended result. But in this case, we have to say that the relaxed model is not acceptable.

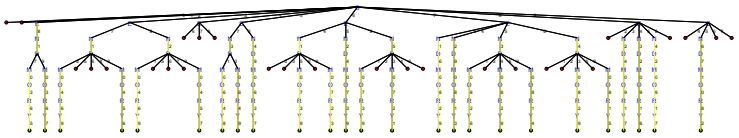
6.2 Multiple Equations

Alternative 2

- Large equality difficult to understand by humans
- Replace with multiple, simpler equations
- Linked by carry variables (0/1)
- Should produce same solutions
- Does it give same propagation?

The second alternative model is motivated in a different way: In our original program we had this one, large equality constraint. Humans do not really reason on such equations: We learned in school to solve additions

Figure 23: Search Tree: Too many solutions



position by position, using 0/1 carries to transfer results from one position to the next. Our modified program will also use this technique.

This should produce the same solution, but might give a different amount of propagation.

Figure 24 shows the modified equation with the carry variables.

Figure 24: Adding Carry Variables

$$\begin{array}{rrrrrr} & S & E & N & D \\ + & M & O & R & E \\ \hline +C5 & C4 & C3 & C2 & \\ \hline M & O & N & E & Y \end{array}$$

We can adapt our program to the new model by replacing the one, large equation with five, smaller equations and introducing new variables C_2, C_3, C_4, C_5 which range over a domain $0..1$.

After the constraint setup, we reach the following domain state (Figure 25), Figure 26 shows the difference to our initial model. We see that the new model is strictly weaker than the original, the domains for E and N have not been reduced as much.

The search tree in Figure 27 shows that we find the same solution, but we explore a few more dead-ends first. For variable E, values 2 and 3 were not removed by the constraint setup, but branching on these values leads to immediate failure.

If we compare the complete search trees, we see that the only difference is in the number of choices explored for variable E. While in the original model we had to explore four branches, we now have to explore seven. Clearly, this does not matter too much, and will not have any real impact on the speed of the program, but for larger problems it is important to remove inconsistent values as early as possible, as long as that can be done without excessive computational effort.

The difference in propagation is caused by treating the positions of the addition as individual, independent

Listing 3: Alternative 2

```
:-module(alternative2).
:-export(sendmory/1).
:-lib(ic).

sendmory(L):-
        L=[S,E,N,D,M,O,R,Y],
        L :: 0..9,
        [C2,C3,C4,C5] :: 0..1,
        alldifferent(L),
        S #\= 0,
        M #\= 0,
        M #= C5,
        S+M+C4 #= 10*C5+O,
        E+O+C3 #= 10*C4+N,
        N+R+C2 #= 10*C3+E,
        D+E    #= 10*C2+Y,
        labeling(L).
```

Figure 25: With Carry Variables: After Setup

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

constraints. As the constraints only communicate via value restrictions of the variables, we deduce less information in the new model.

Observations

- This is solving the original problem
- Search tree slightly bigger
- Caused here by missing interaction of equations
- And repeated variables
- But: Introducing auxiliary variables not always bad!

This second alternative has given us a model which solves the same problem as the original program, but with slightly less propagation. This is causes by missing possible interaction of the constraints and ignoring the repeated occurrence of the same variable in multiple constraints.

In this particular case, introducing new, auxiliary variables did not improve our program. But we will see later on that this is a technique which can sometimes dramatically reduce the search effort.

Figure 26: Setup Comparison

original

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

alternative2

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

Figure 27: Search Tree: First Solution

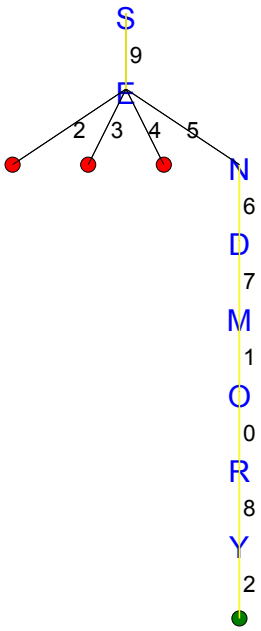
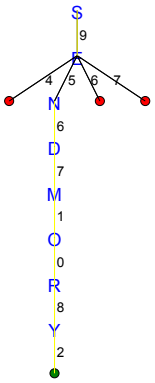
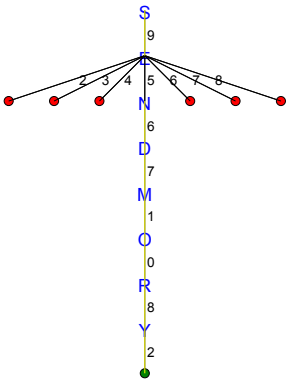


Figure 28: Comparison

Single Equation



Multiple Equations



7 Exercises

Exercises

1. Does the reasoning for the equality constraints that we have presented remove all inconsistent values? Consider the constraint Y=2*X.
2. Why is it important to remove multiple occurences of the same variable from an equality constraint? Give an example!
3. Solve the puzzle DONALD+GERALD=ROBERT. What is the state of the variables before the search, after the initial constraint propagation?
4. Solve the puzzle Y*WORRY = DOOOOD. What is different?
5. (extra credit) How would you design a program that finds new crypt-arithmetic puzzles? What makes a good puzzle?