Chapter 2
Basic Constraint Modeling

Abstract Given a set of variables, each of which has a domain of possible values, and a set of constraints that limit the acceptable set of assignments of values to variables, the goal of a CSP (Constraint Satisfaction Problem) is to find an assignment of values to the variables that satisfies all of the constraints. Picat provides three solver modules, including `cp` (Constraint Programming), `sat` (Satisfiability), and `mip` (Mixed Integer Programming), for modeling and solving CSPs. This chapter provides an introduction to modeling with constraints, with a primary focus on the `cp` module, and a secondary focus on the `sat` and `mip` modules.

2.1 Introduction

In Picat, it is possible to use the same model and syntax for three different solver modules. The `cp` and `sat` modules support integer-domain variables, while the `mip` module supports both integer-domain and real-domain variables. This chapter introduces constraint modeling for solving CSPs (Constraint Satisfaction Problems), in which the possible values of the domains are integers which are often not too large.

Note: The models that use the `sat` solver do not work in the Windows version of Picat, because the Lingeling SAT solver doesn't compile with MVC. Windows users can install Cygwin,[1] and use the Cygwin version instead.

In general, a constraint model consists of a set of *decision variables*, each of which has a specified domain, and a set of *constraints*, each of which restricts the possible combinations of values of the involved decision variables. In order to use a solver, a constraint program must first import the solver module. A constraint program normally poses a problem in three steps: (1) generate variables; (2) generate constraints over the variables; and (3) call `solve` to invoke the solver in order to find a valuation for the variables that satisfies the constraints and possibly optimizes an objective function.

A CSP forms a search space. Solvers have techniques for reducing the search space, and use strategies to search the space for solutions. For example, CP and SAT solvers use *constraint propagation* to prune unfruitful paths in the search space.

[1] https://www.cygwin.com/.

N.-F. Zhou et al., *Constraint Solving and Planning with Picat*,
SpringerBriefs in Intelligent Systems, DOI 10.1007/978-3-319-25883-6_2

A decision variable is also called a *domain variable*. In Picat, the domain constraint `X::D` can be used to restrict the domains of variables. A domain variable is a special kind of a logic variable. General logic variables are assumed to have the set of all possible ground terms, called the *Herbrand base*, as their implicit domains, and operations such as unification can be used to restrict their possible values. In contrast, domain variables have explicit domains, and a rich set of built-in constraints, such as arithmetic constraints and the `all_different` constraint, can be used to restrict their possible values.

The three solver modules in Picat have different strengths and weaknesses. The `cp` module tends to be the best choice for problems in which effective global constraints and/or problem-specific labeling strategies are available. The `sat` module is often well-suited to problems that can be clearly represented as Boolean expressions or have efficient CNF (Conjunctive Normal Form) encodings. The `mip` module is still the best choice for many kinds of Operations Research problems. It is often instructive to test all three solvers on the same problem. The common interface that Picat provides for the solver modules allows seamless switching from one solver to another.

The programs in this chapter and in the next chapter demonstrate an important aspect of constraint modeling, namely *declarative programming*. The ideal is to just state the problem and let the solver do the job, at least in principle. Logic programming also has this feature, but, arguably, constraint modeling takes this one step (or several steps) further. Depending on the specific problem at hand, it might still be necessary to explicitly state certain things, such as for loops and list comprehensions, so please take the notion of "declarativity" with a grain of salt.

2.2 SEND+MORE=MONEY

`SEND+MORE=MONEY` is one of the most commonly-used examples when introducing CSPs. The problem is to substitute each letter (`SENDMORY`) with a distinct digit in the range of `0..9`, such that the equation `SEND + MORE = MONEY` is satisfied. The following gives a model for the problem. Another model, which uses carrying in the same way as multiplication by hand, is left as an exercise.

```
import cp.

main =>
  Digits = [S,E,N,D,M,O,R,Y],
  Digits :: 0..9,
  all_different(Digits),
  S #> 0,
  M #> 0,
               1000*S + 100*E + 10*N + D
  +            1000*M + 100*O + 10*R + E
```

```
    #= 10000*M + 1000*O + 100*N + 10*E + Y,
    solve(Digits),
    println(Digits).
```

Here is a breakdown of the code:

`import cp`: This ensures that the CP solver is used. The module name `cp` can be changed to `sat` or `mip` to use a different solver.

`Digits::0..9`: This defines the domains of the decision variables in the list `Digits`, and thus the single variables `S`, `E`, `N`, `D`, `M`, `O`, `R`, and `Y`. Recall that the range `0..9` denotes the list of integers from `0` through `9`.

`#>` and `#=`: These are *arithmetic constraint operators*. The disequality constraints `S #> 0` and `M #> 0` ensure that the leading digits of `SEND` and `MORE` must both be greater than 0. The equality constraint ensures that `SEND` plus `MORE` equals `MONEY`. Note that all of the arithmetic constraint operators begin with `#`. This special notation distinguishes between the constraint operators and the normal relational operators in Picat.

Functions in arithmetic constraints are treated as data constructors, with the exception of list comprehensions and index notations. In this example, the expressions on the two sides of the equality constraint are constructed as two terms before being passed to the predicate that defines `#=`. Changing `#=` in the program to `=` would result in an evaluation error, since the expressions involve uninstantiated variables, meaning that the functions cannot be evaluated.

`all_different(Digits)`: `all_different` is a *global constraint* which states that all of the decision variables in `Digits` must be distinct. In contrast to unary or binary arithmetic constraints that have one or two arguments, global constraints are often defined to work on a larger collection of decision variables, such as a list. Global constraints are quite unique to CP, compared to other types of CSP solvers, since the CP solver employs special propagation algorithms for global constraints in order to reduce the search space. For this reason, global constraints are often high-level *tools-for-thought* for modeling problems with CP.

Global constraints can be decomposed into smaller constraints. For example, `all_different` (L) can be decomposed into a set of inequality constraints that contains X `#!=` Y for each pair of distinct elements X and Y in L. However, these decompositions are often slower than the specially-crafted global constraints.

`solve(Digits)`: The `solve` predicate finds an assignment of values to the variables that satisfies all of the accumulated constraints. In general, all of the decision variables should be passed to `solve`.

2.3 Sudoku—Constraint Propagation

This section shows how to model the *Sudoku problem*, using a small 4×4 Sudoku instance to demonstrate how constraint propagation changes the domains of the variables during search. Note that the behavior that is described in this section is

specific to the CP solver. The SAT and MIP solvers do not use domain reduction in the same manner.

2.3.1 A Sudoku Program

Given an $N \times N$ board with some squares already filled with hint values, the objective of the Sudoku problem is to fill all the empty squares with values in $1..N$, such that each row, each column, and each of the N blocks are filled with distinct values. Figure 2.1 gives a Picat program for the problem.

The predicate `board(Board)` gives a problem instance. A board configuration is given as a matrix (i.e., a 2-dimensional array), where an anonymous variable, indicated by _ (an underscore), means an unknown, and the numbers are pre-filled hint values. The predicate `sudoku/1` first calculates the dimension `N` and the block size `N1` of the instance. For the given instance, `N` $= 4$ and `N1` $= 2$. The domain constraint `Board :: 1..N` changes all the anonymous variables to domain variables with the domain `1..N`. The predicate `sudoku/1` then generates an `all_different` constraint for each row, each column, and each of the `N` blocks. Finally, it calls the solver to *label* the variables with values, meaning that the solver assigns values to the variables.

Modeling tip: The row and column constraints and the list comprehensions should now be familiar. The third constraint, the list comprehension for the block constraints, might take some time to figure out, both to model and to understand. One tip when modeling this kind of constraint is to first do a "pen and paper version" for finding the proper indices and to then model the indices that are needed for each block.

Note that this problem instance has two solutions. Usually, a Sudoku problem has a single, unique solution.

2.3.2 Constraint Propagation—Concepts

One of the key operations employed in a CP solver is called *constraint propagation*, which actively uses constraints to prune search spaces. Whenever a variable changes, meaning that the variable has been instantiated or a value has been excluded from its domain, the domains of all the remaining variables are filtered to contain only those values that are consistent with the modified variable. Section 2.3.3 provides a concrete example of this domain reduction. There may exist different propagation rules for a constraint, depending on the level of *consistency* to be achieved, i.e., on the efforts to be made to reduce domains.

A *fixpoint* is reached after the propagation phase when no domains can be reduced any further. A fixpoint represents a *failure* if any of the domains becomes empty, and a fixpoint represents a solution if all of the domains become *singletons*, i.e.,

```
import cp.

main =>
  board(Board),
  sudoku(Board),
  println(Board),
  nl.

sudoku(Board) =>
  N = Board.length,      % dimension of Board
  N1 = ceiling(sqrt(N)), % block size
  Board :: 1..N,

  % row constraints
  foreach (I in 1..N)
    all_different([Board[I,J] : J in 1..N])
  end,

  % column constraints
  foreach (J in 1..N)
    all_different([Board[I,J] : I in 1..N])
  end,

  % block constraints
  foreach (I in 1..N1..N, J in 1..N1..N)
    all_different([Board[I+K,J+L] :
                   K in 0..N1-1, L in 0..N1-1])
  end,
  solve(Board).

board(Board) =>
  Board = {{4, _,  _, _},
           {_, 1,  _, _},
           {_, _,  _, 1},
           {_, _,  _, 2}}.
```

Fig. 2.1 A program for Sudoku

single-element domains. In general, the propagation phase may not be able to reach a failure or a solution, in which case the CP solver needs to use another key operation, called *search* or *labeling*, to guess a value or a range of values for a variable. After a guess, constraint propagation reduces the domains of the related variables. When a failure is reached, the CP solver *backtracks* to a previous guess and tries a different value or values for the variable.

The performance of a constraint program is greatly affected by *search strategies*, which decide the variables to select and the values to guess. In Picat, search strategies can be given to the CP solver as options in `solve` (*Options*, *Vars*). The `ff` (*first-fail*) strategy, which selects the leftmost variable with the smallest domain, is often used.

2.3.3 Constraint Propagation—Example

Before reading further, try to solve this Sudoku instance by hand, and study the thinking used to reach the different values. For instance, one may think, "since this is the only possible value in this cell, the cell must be that value." One might also think, "the values in these rows and columns, when taken together, mean that this cell must have that value." Reasoning like this is a way of *reducing the domains* of the different variables, as mentioned in Sect. 2.3.2.

For example, the following shows how constraint propagation changes the domains of the variables after each loop statement in the Sudoku program. Initially, after `Board::1..N`, all of the domains are either a hint value or the full domain (`1..4`).

```
   4       1..4       1..4       1..4
1..4          1       1..4       1..4
1..4       1..4       1..4          1
1..4       1..4       1..4          2
```

After the loop that posts the *row constraints*, the domains change to the following:

```
   4       1..3       1..3       1..3
2..4          1       2..4       2..4
2..4       2..4       2..4          1
1,3,4   1,3,4      1,3,4          2
```

Since the hint value at cell [1,1] is 4, the `all_different` constraint for the first row excludes 4 from the domains at cells [1,2], [1,3], and [1,4]. The domains in the other rows are changed in a similar fashion.

After the loop that posts the *column constraints*, the domains change to the following:

```
   4          2          1          3
 2,3          1        2,3          4
 2,3       3,4       2..4          1
 1,3       3,4        3,4          2
```

The hint value at cell [3,4] is 1, and the value at cell [4,4] is 2. After the column constraint for column 4 is posted, the domain at cell [1,4] changes to 3, and the domain at cell [2,4] changes to 4. After the domain at cell [1,4] changes to 3, the row constraint for the first row is activated to exclude 3 from the domains at cells [1,2] and [1,3]. These changes will trigger the other constraints to reduce the related domains before a fixpoint is reached.

After the loop that posts the *block constraints*, the domains change to the following:

```
4       2       1       3
3       1       2       4
2     3,4     3,4       1
1     3,4     3,4       2
```

The constraint for the upper-left block removes 2 from the domain at cell [2,1], the constraint for the upper-right block removes 3 from the domain at cell [2,3], and the constraint for the lower-right block removes 1 from the domain at cell [3,3]. These changes will trigger the row and column constraints to further reduce the related domains before a fixpoint is reached.

When `solve(Board)` is executed, there are only four uninstantiated variables. After guessing 3 for the domain at cell [3,2], the constraints are triggered to remove 3 from the domains at cell [3,3] and cell [4,2]. After these changes, 4 is removed from the domain at cell [4,3]. For this instance, a single guess is sufficient to find a solution, meaning that backtracking is not needed. Note that this behavior is more of an exception; most interesting problems normally require backtracking.

2.4 Minesweeper—Using SAT

The *Minesweeper problem* in this context is not the full GUI-based program that is normally associated with the name "Minesweeper". Instead, it is a simplified problem, where the goal is to identify the positions of all the mines in a given matrix, with hints that state how many mines there are in the neighboring cells, including diagonal neighbors. A cell in the middle of the matrix has eight neighbors, a cell in the corner has three neighbors, and an edge cell has five neighbors.

Here is a problem instance:

```
. . 2 . 3 .
2 . . . . .
. . 2 4 . 3
1 . 3 4 . .
. . . . . 3
. 3 . 3 . .
```

For this instance, the third cell in the first row has the value 2, which indicates that it has two adjacent mines. The cells marked with "`.`" are unknowns, meaning that the cell may or may not have a mine. One can also observe that if there is a number in a cell, then the cell cannot contain a mine.

```
import sat.

main =>
  % define the problem instance
  problem(Matrix),
  NRows = Matrix.length,
  NCols = Matrix[1].length,

  % decision variables: where are the mines?
  Mines = new_array(NRows,NCols),
  Mines :: 0..1,

  foreach (I in 1..NRows, J in 1..NCols)

    % only check those cells that have hints
    if ground(Matrix[I,J]) then

      % The number of neighboring mines must equal Matrix[I,J].
      Matrix[I,J] #= sum([Mines[I+A,J+B] :
                                  A in -1..1, B in -1..1,
                                  I+A >  0, J+B >  0,
                                  I+A =< NRows, J+B =< NCols]),

      % If there is a hint in a cell, then it cannot be a mine.
      Mines[I,J] #= 0
    end
  end,
  solve(Mines),
  println(Mines).

problem(P) =>
  P = {{_,_,2,_,3,_},
       {2,_,_,_,_,_},
       {_,_,2,4,_,3},
       {1,_,3,4,_,_},
       {_,_,_,_,_,3},
       {_,3,_,3,_,_}}.
```

Fig. 2.2 A program for Minesweeper

Figure 2.2 gives a program for the problem. The predicate `problem/1` specifies an instance matrix in which an integer indicates a hint value and an underscore _ indicates an unknown. For an instance matrix, let `NRows` be the number of rows and `NCols` be the number of columns. The function `new_array(NRows,NCols)` creates a matrix of the same dimension as the instance matrix. The domain constraint `Mines::0..1` restricts every variable to be Boolean: the value 0 indicates that the cell does not contain a mine, and the value 1 indicates that the cell contains a mine. The loop generates constraints to ensure that all of the hint values are respected. For each cell, let `I` be its row number and `J` be its column number. If `Matrix[I,J]` is ground, meaning that the cell has a hint value, then `Mines[I,J]` must be 0, and the sum of the neighboring cells in the matrix `Mines` must be equal to the hint value.

Table 2.1 Minesweeper: CP vs SAT

N	Solutions	CP (s)	SAT (s)	Winner
50	1	0.016	0.072	CP
107	1	0.068	0.208	CP
202	1	0.284	0.548	CP
430	1	1.96	2.19	CP
440	2	3.6	3.08	SAT
450	5	10.1	6.46	SAT
500	3	7.5	5.03	SAT
601	1	4.99	3.65	SAT
1000	1	21.69	9.47	SAT
1500	3	804.6	61.27	SAT
1601	1	143.25	40.79	SAT

Note how this constraint is expressed nicely using a list comprehension. For a cell whose position is `[I,J]`, a neighboring cell must have the position `[I+A,J+B]`, where `A` and `B` are row and column deltas in `-1..1`. In the list comprehension, conditions are tested to ensure that `I+A` is a valid row number and `J+B` is a valid column number.

This program uses the `sat` module. For large instances of the Minesweeper problem, the `sat` module tends to be significantly faster than other solver modules. In general, SAT solvers tends to outperform CP solvers on 0/1 integer programming modules. SAT solvers perform better because the search strategies that CP solvers use for this type of problem are limited, and because SAT solvers are much more intelligent than CP solvers in pruning unfruitful paths from the search space.

The authors did an experiment of generating 20 $N \times N$ random matrices with N random hint values in the range of 1 to 8, including both solvable and unsolvable Minesweeper instances. The timings for selected N with at least one solvable instance are shown in Table 2.1. This suggests that, for this setup, the CP solver is faster than the SAT solver for $N \leq 430$, and that, for larger N, the SAT solver is significantly faster. Note: The exact times are for Picat version 1.3, and might differ in later versions.

2.5 Diet—Mathematical Modeling with the `mip` Module

This section considers the *Diet problem*, a popular linear programming problem in Operations Research. Given a set of foods, each of which has given nutrient values, a cost per serving, and a minimum limit for each nutrient, the objective of the diet problem is to select the number of servings of each food to consume so as to minimize the cost of the food while meeting the nutritional constraints. Table 2.2 gives an example. In this example, a diet is required to contain at least 500 calories, 6 ounces of chocolate, 10 ounces of sugar, and 8 ounces of fat.

Table 2.2 An example of the diet problem

Type of food	Calories	Chocolate (oz.)	Sugar (oz.)	Fat (oz.)	Price (cents)
Chocolate cake (1 slice)	400	3	2	2	50
Chocolate ice cream (1 scoop)	200	2	2	4	20
Cola (1 bottle)	150	0	4	1	30
Pineapple cheesecake (1 piece)	500	0	4	5	80
Limits	500	6	10	8	–

```
import mip.

main =>
  data(Prices,Limits,{Calories,Chocolate,Sugar,Fat}),
  Len = length(Prices),
  Xs = new_array(Len),
  Xs :: 0..10,

  scalar_product(Calories,Xs, #>=,Limits[1]), % 500
  scalar_product(Chocolate,Xs,#>=,Limits[2]), % 6
  scalar_product(Sugar,Xs,    #>=,Limits[3]), % 10
  scalar_product(Fat,Xs,      #>=,Limits[4]), % 8
  scalar_product(Prices,Xs,XSum), % to minimize
  solve($[min(XSum)],Xs),
  writeln(Xs).

% plain scalar product
scalar_product(A,Xs,Product) =>
  Product #= sum([A[I]*Xs[I] : I in 1..A.length]).

scalar_product(A,Xs,Rel,Product) =>
  scalar_product(A,Xs,P),
  call(Rel,P,Product).

data(Prices,Limits,Nutrition) =>
  % prices in cents for each product
  Prices = {50,20,30,80},
  % required amount for each nutrition type
  Limits = {500,6,10,8},

  % nutrition for each product
  Nutrition =
    {{400,200,150,500},  % calories
     {  3,  2,  0,  0},  % chocolate
     {  2,  2,  4,  4},  % sugar
     {  2,  4,  1,  5}}. % fat
```

Fig. 2.3 A program for the diet problem

Figure 2.3 gives a program for the example problem. The predicate

```
data(Prices,Limits,Nutrition}
```

defines a problem instance, where `Prices` is an array of prices of the foods, `Limits` is an array of the minimum numbers of servings for the different nutrients, and `Nutrition` is a matrix that gives the nutrient values of each type of food. The array `Xs` is a vector of decision variables. For each food type numbered `I`, `Xs[I]` denotes the number of servings of the food. The maximum number of servings is set to 10.

This program uses two user-defined constraints, named `scalar_product/3` and `scalar_product/4`, to handle the nutritional constraints. Let `A` be a vector of nutrient values. The `scalar_product(A,Xs,Product)` constraint ensures that `Product` is the total value of the nutrient. The `scalar_product/4` constraints ensure that the total value of each nutrient is no less than the required limit.

The call `scalar_product(Prices,Xs,XSum)` creates a new decision variable, `XSum`, which is equal to the total price of the foods. This is an optimization problem; the option in `solve($[min(XSum)],Xs)` means that `XSum` is minimized.

The program can be shortened by using a loop. The `main` predicate can be rewritten into the following:

```
main =>
  data(Prices,Limits,Nutrients),
  Len = length(Prices),
  Xs = new_array(Len),
  Xs :: 0..10,
  foreach (I in 1..Nutrients.length)
    scalar_product(Nutrients[I],Xs,#>=,Limits[I])
  end,
  scalar_product(Prices,Xs,XSum),
  solve($[min(XSum)],Xs),
  writeln(Xs).
```

This shortened program is more general than the previous version, and works on an input table of any size.

Modeling tip: For beginners, it might be difficult to start by implementing a very general model. Therefore, it is often beneficial to first write a small "explicit" model, which can then be generalized. Also, when writing a new model, it is recommended to start with a small problem instance, where the solutions are known. This has the advantages of being faster to solve, and that one can directly check that the solver has given a correct solution.

In this example, when modeling the diet problem, all three of the constraint modules quickly solve the problem. The next section gives an example for which `mip` is significantly faster than `sat` and `cp`.

2.6 The Coins Grid Problem: Comparing MIP with CP/SAT

As indicated earlier, the same Picat model can often be used for all three solver modules. This section studies the *Coins Grid problem*, an example on which the MIP solver is much faster than the other two solvers.

The problem statement is from Tony Hürlimann's "A coin puzzle: SVOR-contest 2007",[2] and is described below:

> In a quadratic grid (or a larger chessboard) with 31 × 31 cells, one should place coins in such a way that the following conditions are fulfilled:
>
> 1. In each row exactly 14 coins must be placed.
> 2. In each column exactly 14 coins must be placed.
> 3. The sum of the quadratic horizontal distance from the main diagonal of all cells containing a coin must be as small as possible.
> 4. In each cell at most one coin can be placed.

Figure 2.4 gives a program for the problem. The grid is represented as a matrix with 0/1 entries, with 0 indicating that a coin is not placed in the cell, and 1 indicating that a coin is placed in the cell. The row and column constraints are easy to model using `sum`. For a cell in row `I` and column `J`, the "quadratic horizontal distance" of the cell from the main diagonal is `abs(I-J)*abs(I-J)`. `Sum` is the total of the distances between the coins and the grid's main diagonal. The objective of this problem is to minimize `Sum`.

Table 2.3 compares the three solvers using the same model on different grid sizes (`N`) and different numbers of coins (`C`). The MIP solver solves the original problem (`N=31, C=14`) in a few milliseconds, whereas both CP and SAT failed to solve this problem in reasonable times (several hours). When compared to the MIP solver, both the CP and SAT solvers are very slow, even for small values of `N` and `C`.

The reason why the MIP solver is much faster is probably that all of the constraints in the model are linear, and MIP solvers are often very fast for linear models.

Modeling tip: When working with harder optimization problems, it is often useful to see the current value of the objective variable during the solving phase. In Picat, the option `report` (*Call*) can be used for this purpose. For example,

```
solve($[min(Sum),report(printf("Sum: %w\n", Sum))],Vars),
```

tells Picat to print out "`Sum: `" whenever it finds a new and better value of `Sum`.

[2] Taken with permission from www.svor.ch/competitions/competition2007/AsroContestSolution.pdf.

```
import mip.

main =>
  N = 31,
  C = 14,
  time2(coins(N, C)).

coins(N,C) =>
  X = new_array(N,N),
  X :: 0..1,

  foreach (I in 1..N)
    C #= sum([X[I,J] : J in 1..N]), % rows
    C #= sum([X[J,I] : J in 1..N])  % columns
  end,

  % quadratic horizontal distance
  Sum #= sum([(X[I,J] * abs(I-J) * abs(I-J)) :
                I in 1..N, J in 1..N]),
  solve($[min(Sum)],X),
  println(sum=Sum).
```

Fig. 2.4 A program for the coins-grid problem

Table 2.3 Coins grid problem (Picat 1.3)

N	C	Sum	CP (s)	SAT (s)	MIP (s)
8	4	80	2.27	3.48	0.0
8	5	198	72.15	3.76	0.0
9	4	90	12.61	6.48	0.0
10	4	98	97.39	7.51	0.0
31	14	13668	–	–	0.016

2.7 *N*-Queens—Different Modeling Approaches

The *N-queens problem* is another standard CSP problem. The objective is to place N queens on an $N \times N$ chessboard, such that no queen can capture any other queen. Queens on a chessboard can go in three directions: horizontally, vertically, and diagonally. There are several ways of modeling this problem. This section presents three models.

Here is the first, "naive", model:

```
import cp.

queens1(N, Q) =>
  Q = new_list(N),
  Q :: 1..N,
```

```
foreach (I in 1..N, J in 1..N, I < J)
  Q[I] #!= Q[J],          % columns
  Q[I] - I #!= Q[J] - J, % diagonal 1
  Q[I] + I #!= Q[J] + J  % diagonal 2
end,
solve([ff],Q).
```

This model uses a list `Q` of `N` variables, representing the chessboard's N rows, where each variable's value is a column number for each row. For all different pairs of `I` and `J` (where `I` < `J`), this model ensures the following:

1. The columns are distinct: `Q[I] #!= Q[J]`.
2. No two queens are placed the same diagonal: `Q[I] - I #!= Q[J] - J`, and `Q[I] + I #!= Q[J] + J`.

The second model uses the global constraint `all_different/1` for handling the three principal constraints:

```
import cp.

queens2(N, Q) =>
  Q = new_list(N),
  Q :: 1..N,
  all_different(Q),
  all_different([$Q[I]-I : I in 1..N]),
  all_different([$Q[I]+I : I in 1..N]),
  solve([ff],Q).
```

As shown in earlier examples, the arguments of the latter two `all_different` constraints must be "escaped" by `$` in order for them to be treated as terms rather than as function calls.

Recall that the option `[ff]` specifies the *first-fail* labeling strategy. More about labeling strategies will discussed in the next chapter.

A third model is a 0/1 integer programming model that uses an $N \times N$ array. For each cell, a 0/1 domain variable is used, with 0 indicating an empty square, and 1 indicating a queen. Figure 2.5 shows a program based on this model.

The first two foreach loops ensure that there is exactly one queen in each row and each column. The two latter loops ensure the diagonality constraints by restricting each possible diagonal in the grid to contain at most one queen. Note that some diagonals might not contain any queens.

The timings of the three N-queens versions for the three constraint solvers, shown in Tables 2.4, 2.5 and 2.6, indicate some interesting differences. For the CP solver, the `all_different` version is significantly faster than both the SAT and MIP solvers. For the SAT solver, the "naive" version is faster than the `all_different` version, which perhaps indicates that, for this problem, the translation of the constraint `all_different` to the SAT solver's representation is not as good as

```
import sat.

queens3(N, Q) =>
  Q = new_array(N,N),
  Q :: 0..1,
  foreach (I in 1..N)
    % 1 in each row
    sum([Q[I,J] : J in 1..N]) #= 1
  end,
  foreach (J in 1..N)
    % 1 in each column
    sum([Q[I,J] : I in 1..N]) #= 1
  end,
  foreach (K in 1-N..N-1)
    % at most 1 in each \ diagonal
    sum([Q[I,J] : I in 1..N, J in 1..N, I-J==K]) #=< 1
  end,
  foreach (K in 2..2*N)
    % at most 1 in each / diagonal
    sum([Q[I,J] :  I in 1..N, J in 1..N, I+J==K]) #=< 1
  end,
  solve(Q).
```

Fig. 2.5 A 0/1 integer programming model for N-queens

Table 2.4 N-queens: "Naive version", using `solve($[ff],Q)` for CP

N	CP (s)	SAT (s)	MIP (s)
8	0.00	0.08	0.35
10	0.00	0.12	0.50
12	0.00	0.16	5.17
20	0.00	0.80	1557.3
50	0.00	8.74	–
100	0.02	48.10	–
200	4.86	105.61	–
400	0.56	–	–
1000	5.02	–	–

the implemented "naive" version. Note that the MIP solver does not support the `all_different` constraint. For the third model, the SAT solver is by far the fastest among the three solvers. Also, this is the model where the MIP solver has its fastest times.

Comparing all the models and solvers, the fastest combination is the CP solver running the `all_different` model.

Table 2.5 *N*-queens: `all_different` version, using `solve($[ff],Q)` for CP

N	CP (s)	SAT (s)	MIP (s)
8	0.00	0.09	–
10	0.00	0.14	–
12	0.00	0.22	–
20	0.00	0.92	–
50	0.004	8.33	–
100	0.01	473.1	–
200	2.36	782.88	–
400	0.28	–	–
1000	1.8	–	–

Table 2.6 *N*-queens: 0/1 integer programming version, using `solve($[inout],Q)` for CP

N	CP (s)	SAT (s)	MIP (s)
8	0.0	0.0	0.0
10	0.0	0.0	0.0
12	0.0	0.0	0.02
20	0.0	0.05	0.27
50	0.05	0.93	12.38
100	10.17	1.65	704.7
200	>60 min	7.20	–
400	–	48.41	–
1000	–	704.11	–

The timings are for Picat version 1.3 using Linux Ubuntu 12.04, 64-bit, Intel i7-930 CPU (2.8 GHz), 12 Gb RAM. Later versions might give different timings (and even different relations between the solvers' times).

As indicated in this chapter, testing different approaches and solvers can give very different behaviors in terms of solving times. Sometimes, this kind of experimentation is not really needed, since the first variant is "good enough". Nevertheless, trying different variants is a good way of learning constraint modeling, and can also be fun.

2.8 Bibliographical Note

This chapter and Chap. 3 cover a few techniques and applications of constraint programming. A more in-depth discussion of constraint programming, including its history, the algorithms that constraint solvers use, and further techniques and applications of constraint programming, is included in [45].

The literature on constraint programming is abundant, including tutorials [3, 7], surveys [10, 28, 44]) and textbooks ([18, 37, 57, 58]). Picat's CP solver is inherited from B-Prolog [67, 68].

The Diet problem is just one type of Operations Research problem. References [54] and [65] contain examples of how Operations Research can be used to manage businesses.

The description of the Coins Grid problem is taken from [27].

2.9 Exercises

1. Implement the *carry version* of the `SEND + MORE = MONEY` problem from Sect. 2.2. In the carry version, whenever the the sum of the column is greater than ten, the leftmost digit of the sum is explicitly added to the next column on the left. For example,

```
 C4 C3 C2 C1
     S  E  N  D
+    M  O  R  E
---------------
= M  O  N  E  Y
```

   The four variables `C1`, `C2`, `C3`, and `C4` are the carries in the addition.
   Compare the performance of the two models.
2. Implement the `DONALD + ROBERT = GERALD` problem using both the non-carry approach and the carry approach. Compare the performances.
3. Modify the Sudoku program from Sect. 2.3.1 to solve the following 9×9 Sudoku instance:

```
_ _ 2  _ _ 5  _ 7 9
1 _ 5  _ _ 3  _ _ _
_ _ _  _ _ _  6 _ _

_ 1 _  4 _ _  9 _ _
_ 9 _  _ _ _  _ 8 _
_ _ 4  _ _ 9  _ 1 _

_ _ 9  _ _ _  _ _ _
_ _ _  1 _ _  3 _ 6
6 8 _  3 _ _  4 _ _
```

   Ensure that it has only one solution.

4. The output of the `minesweeper` model in Sect. 2.4 is just a list of 0s and 1s. Add a predicate to print the output in the following format:
   The mine cells are represented with `X`, and the non-mine cells are shown as _.
   Check that the minesweeper solution complies with the given hints.
5. Write a constraint model for the *Zebra problem* (sometimes attributed to Lewis Carroll):

   Five men with different nationalities live in the first five houses of a street. They practice five distinct professions, and each of them has a favorite animal and a favorite drink, all of them different. The five houses are painted in different colors.
   The Englishman lives in a red house.
   The Spaniard owns a dog.
   The Japanese is a painter.
   The Italian drinks tea.
   The Norwegian lives in the first house on the left.
   The owner of the green house drinks coffee.
   The green house is on the right of the white one.
   The sculptor breeds snails.
   The diplomat lives in the yellow house.
   Milk is drunk in the middle house.
   The Norwegian's house is next to the blue one.
   The violinist drinks fruit juice.
   The fox is in a house next to that of the doctor.
   The horse is in a house next to that of the diplomat.

   Who owns a zebra, and who drinks water?

   Ensure that the model has a unique solution.
6. *Map coloring*:

   (a) The map coloring problem is to color the countries of a map, ensuring that each country is given a different color than the countries that are its neighbors. Below are some European countries and their neighbors. Implement a map coloring model for these countries.

   Countries: Belgium, Denmark, France, Germany, Luxembourg, Netherlands

   Neighbors:

   - Belgium: France, Germany, Luxembourg, Netherlands
   - Denmark: Germany
   - France: Belgium, Germany, Luxembourg
   - Germany: Belgium, Denmark, France, Netherlands, Luxembourg
   - Luxembourg: Belgium, France, Germany
   - Netherlands: Belgium, Germany

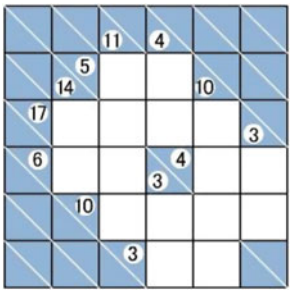

Fig. 2.6 Sample Kakuro puzzle

Print all the possible solutions.
Hint 1: The colors themselves are not very important, and can be represented as integers.
Hint 2: There are many solutions which are the same, except that the numbers are a permutation of another solution. To remove some of these *symmetries* one can assume, for example, that Belgium has the color 1.

(b) Find a map of Europe and implement a map coloring model of all the European countries.

7. Test the models in this chapter with the `cp` and `sat` constraint modules and compare the results.
8. Solve the instance of the *Kakuro puzzle* that is depicted in Fig. 2.6. The objective of the Kakuro puzzle is to fill in the white squares with digits between 1 and 9 such that each set of horizontal digits adds up to the value on its left, and each set of vertical digits adds up to the value above it. No digit can be used more than once in each sum. Note: when one square contains two pre-filled numbers, the top number applies to the horizontal sum, and the bottom number applies to the vertical sum.[3]

[3]This is a relatively small problem instance of Kakuro. For larger and more difficult instances, see the website Kakuro Conquest (http://www.kakuroconquest.com/).

http://www.springer.com/978-3-319-25881-2

Constraint Solving and Planning with Picat

Zhou, N.-F.; Kjellerstrand, H.; Fruhman, J.

2015, XI, 148 p. 40 illus., 31 illus. in color., Softcover
ISBN: 978-3-319-25881-2