NOPT042 Constraint programming: Tutorial 3 – Improving your model

What was in Lecture 3

Look-back search algorithms (tree search)

- backtracking (what order of variables and values?)
- backjumping (to the source of conflict)
- graph-directed backjumping (driven by constraint network, several jumps in sequence)
- Gaschnig backjumping (considers violated constraints, only one jump, to highest-level)
- conflict-driven: combines both, carry the source of conflict through backjumps
- dynamic backtracking: change order of variables (don't rework easy parts outside of conflict)
- backmarking (avoid repeated constraint checks: remember results of tests)

Tips & Tricks

Debuging

- Try very small instances where you understand the solution set.
- Unit-test constraints one by one.
- Start with a model that is as simple as possible.
- For advanced debuging options you can use `debug`, see "How to Use the Debugger" from the Picat Guide.

How to import a file

Use `cl(instance)` to compile (to bytecode `instance.qi`) & load the file `instance.pi` (anywhere in $PATH).

```
main =>
    cl(instance),
    puzzle(Vars),
    solve(Vars).
```

See also Modules (`import`).

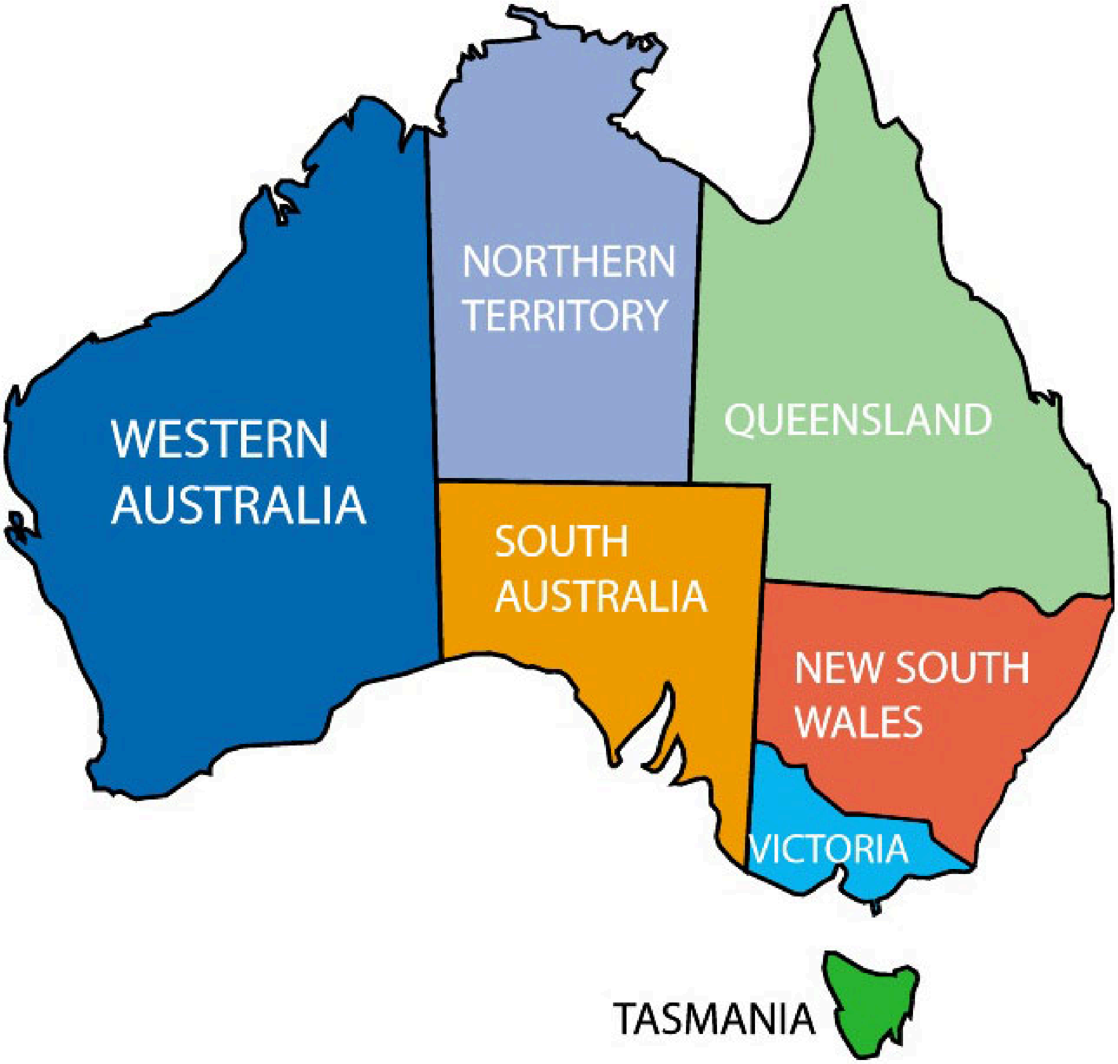
Improving the model

It is a good practice to first create a baseline model, and then try to improve. Ways to create more efficient model include:

- **global constraints**: e.g. `all_different`
- **symmetry breaking**: if there is a symmetry in the search space, e.g. in variables or in values, we can fix one element of the orbit and thus limit the part of the space that needs to be explored
- choosing the best solver for your model (or the best model for your solver)
- choosing a good solver configuration (e.g. search strategy---see the next tutorial)

Example: Map coloring

Create a model to color the map of Australian states and territories 7 with four colors (cf. The 4-color Theorem). (We exclude the Australian Capital Territory, the Jervis Bay Territory, and the external territories. Map coloring is a special case of graph coloring, see this map.



Let's use the following decision variables:

```
Territories = [WA, NT, SA, Q, NSW, V, T]
```

In [1]:
```
!picat map-coloring/map-coloring-baseline
```

```
[1,2,3,1,2,1,1]
```

In [2]:
```
!cat map-coloring/map-coloring-baseline.pi
```

```
import cp.

color_map(Territories) =>
    % variables
    Territories = [WA, NT, SA, Q, NSW, V, T],
    Territories :: 1..4,

    % constraints
    WA #!= NT,
    WA #!= SA,
    NT #!= SA,
    NT #!= Q,
    SA #!= Q,
    SA #!= NSW,
    SA #!= V,
    Q  #!= NSW,
    V  #!= NSW.


main =>
    color_map(Territories),
    solve(Territories),
    println(Territories).
```

In [ ]:

```
```

How can we improve the model?

In [3]:

```
!picat map-coloring/map-coloring-improved
```

```
Western Australia is red.
Northern Territory is green.
South Australia is blue.
Queensland is red.
New South Wales is green.
Victoria is red.
Tasmania is red.
```

In [4]:

```
!cat map-coloring/map-coloring-improved.pi
```

```
import cp.

color_map(Territories) =>
    % variables
    Territories = [WA, NT, SA, Q, NSW, V, T],
    Territories :: 1..8,

    % constraints
    Edges = [
        {WA,NT},
        {WA,SA},
        {NT,SA},
        {NT,Q},
        {SA,Q},
        {SA,NSW},
        {SA,V},
        {Q ,NSW},
        {V ,NSW}
    ],
    foreach(E in Edges)
        E[1] #!= E[2]
    end.

% symmetry breaking constraints
precolor(Territories) =>
    WA #= 1,
    NT #= 2,
    SA #= 3.

% better output than `println(Territories)` (we could also use a map, i.e. a
dictionary)
output(Territories) =>
    Color_names = ["red", "green", "blue", "yellow"],
    Territory_names = ["Western Australia", "Northern Territory", "South Aus
tralia", "Queensland", "New South Wales", "Victoria", "Tasmania"],
    foreach(I in 1..Territories.length)
        writef("%s is %s.\n", Territory_names[I], Color_names[Territories
[I]])
    end.


main =>
    color_map(Territories),
    % precolor(Territories),
    solve(Territories),
    output(Territories).
```

What else is wrong with this model? We always want to separate the model from the data. (See the exercise Graph-coloring below.)

Choosing a solver

Picat provides the following four solvers (implemented as modules):

- cp
- sat
- smt
- mip

What are the differences?

Example: Balanced diet (optimization)

This is (one of?) the first optimization problem for which Linear programming was used. Given a list of food items together with their nutritional values and prices, the goal is to choose a balanced diet---one that contains required minimal amounts of nutrients---while minimizing total price.

Note how we pass options to the solver: `solve($[min(XSum)],Xs)` The $ sign tells the solver to interpret the following as a term, rather than evaluating it as a function.

We will use the `mip` solver. It requires an external MIP solver. Here we use the Computational Infrastructure for Operations Research (COIN-OR)'s Cbc (branch and cut).

First, we need to install the Cbc package.

```
sudo apt-get install  coinor-cbc coinor-libcbc-dev
```

Or without root privileges:

```
wget https://raw.githubusercontent.com/coin-
or/coinbrew/master/coinbrew
chmod u+x coinbrew
./coinbrew fetch Cbc@master
./coinbrew build Cbc
export PATH=$PATH:~/coinbrew/dist/bin
```

In [5]:
```
!picat balanced-diet/balanced-diet
```

```
*** error(existence_error(mip_solver),solve)
```

In [6]:
```
!cat balanced-diet/balanced-diet.pi
```

```
% adapted from Constraint Solving and Planning with Picat, Springer
% by Neng-Fa Zhou, Hakan Kjellerstrand, and Jonathan Fruhman

import mip.

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).

% 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
```

In [7]:

```
!cat __tmp.lp
```

```
Minimize
 obj: X0
Subject To
 -X0 >= -1800
 X0 >= 0
 -X1 >= -10
 X1 >= 0
 -2 X1 - 2 X2 - 4 X3 - 4 X4 + X6 = 0
 -X2 >= -10
 X2 >= 0
 -2 X2 - 3 X1 + X7 = 0
 -20 X2 - 30 X3 - 50 X1 - 80 X4 + X0 = 0
 -X3 >= -10
 X3 >= 0
 -150 X3 - 200 X2 - 400 X1 - 500 X4 + X8 = 0
 -X3 - 2 X1 - 4 X2 - 5 X4 + X5 = 0
 -X4 >= -10
 X4 >= 0
 -X5 >= -120
 X5 >= 8
 -X6 >= -120
 X6 >= 10
 -X7 >= -50
 X7 >= 6
 -X8 >= -12500
 X8 >= 500
Bounds
 0 <= X0 <= 1800
 0 <= X1 <= 10
 0 <= X2 <= 10
 0 <= X3 <= 10
 0 <= X4 <= 10
 8 <= X5 <= 120
 10 <= X6 <= 120
 6 <= X7 <= 50
 500 <= X8 <= 12500
Integers
 X0
 X1
 X2
 X3
 X4
 X5
 X6
 X7
 X8
End
```

In [8]:

```
!cat __tmp.sol
```

```
Problem:
Rows:       23
Columns:    9 (9 integer, 0 binary)
Non-zeros:  41
Status:     INTEGER OPTIMAL
Objective:  obj = 90 (MINimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 r.4                       -90         -1800
     2 r.5                        90             0
     3 r.6                         0           -10
     4 r.7                         0             0
     5 r.8                         0             0             =
     6 r.9                        -3           -10
     7 r.10                        3             0
     8 r.11                        0             0             =
     9 r.12                        0             0             =
    10 r.13                       -1           -10
    11 r.14                        1             0
    12 r.15                        0             0             =
    13 r.16                        0             0             =
    14 r.17                        0           -10
    15 r.18                        0             0
    16 r.19                      -13          -120
    17 r.20                       13             8
    18 r.21                      -10          -120
    19 r.22                       10            10
    20 r.23                       -6           -50
    21 r.24                        6             6
    22 r.25                     -750        -12500
    23 r.26                      750           500

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 X0           *             90             0          1800
     2 X1           *              0             0            10
     3 X2           *              3             0            10
     4 X3           *              1             0            10
     5 X4           *              0             0            10
     6 X6           *             10            10           120
     7 X7           *              6             6            50
     8 X8           *            750           500         12500
     9 X5           *             13             8           120

Integer feasibility conditions:

KKT.PE: max.abs.err = 0.00e+00 on row 0
        max.rel.err = 0.00e+00 on row 0
        High quality

KKT.PB: max.abs.err = 0.00e+00 on row 0
        max.rel.err = 0.00e+00 on row 0
        High quality

End of output
```

Exercises

Exercise: Coins grid

Place coins on a 31×31 board such that each row and each column contain exactly 14 coins, minimize the sum of quadratic horizontal distances of all coins from the main diagonal. (Source: Tony Hurlimann, "A coin puzzle - SVOR-contest 2007")

Exercise: Sudoku

A traditional constraint satisfaction example: solve an $n \times n$ sudoku puzzle. Try the following simple instance (from the book):

```
Board = {
    {4, _,  _, _},
    {_, 1,  _, _},
    {_, _,  _, 1},
    {_, _,  _, 2}
}.
```

Exercise: Graph-coloring

1. Write a program that solves the (directed) graph 3-coloring problem with a given number of colors and a given graph. The graph is given by a list of edges, each edge is a 2-element array. We assume that vertices of the graph are $1, \ldots, n$ where n is the maximum number appearing in the list.
2. Generalize your program to graph k-coloring where k is a positive integer given on the input.
3. Modify your program to accept the incidence matrix (a 2D array) instead of the list of edges.
4. Add the flag `-n` to output the minimum number of colors (the chromatic number) of a given graph. For example:

```
picat graph-coloring "[{1,2},{2,3},{3,4},{4,1}]"
picat graph-coloring "[{1,2},{2,3},{3,1}]" 4
picat graph-coloring "{{0,1,1},{1,0,1},{1,1,0}}" 4
picat graph-coloring -n "[{1,2},{2,3},{3,4},{4,1}]"
```