NOPT042 Constraint programming: Tutorial 4 – Search strategies

What was in Lecture 4

Consistency techniques

- use constraints *actively*
- remove inconsistent values (or combinations thereof) from the domains
- **node consistency**: intersect domain with all unary constraints
- **arc consistency**: ensure that every value of x is compatible with some value y via every binary constraint
- naive implementation: AC-1 (revise all arcs, repeat until no change); improvement AC-2
- **AC-3**: most widely used, keep a queue of arcs to revise (reduction of one domain triggers revision of neighboring domains)
- AC-4 (best worst-case time complexity), AC-5, AC-6, AC-7, AC-3.1, AC-2001
- directional arc consistency: DAC-1

From last week:

- Solution to the Coin grid problem.
- Best model and solver for the problem? MIP, naturally expressed as an integer program
- Unsatisfiable instances - LP works well.
- For sparse solution sets heuristic approaches may be slow.

Today: search strategies

Recall backtracking and friends from the lecture.

- How to explore the search tree?
- E.g., how to select the variable for the next level,
- and the order of values (children nodes)?

The *First Fail* principle: try to prove failure of the subtree as fast as possible, focus on hard variables first.

The predicate `time2` also outputs the number of backtracks during the search - a good measure of complexity.

Example: N-queens

Place n queens on an $n \times n$ board so that none attack another. How to choose the decision variables?

- How large is the search space?
- Can we use symmetry breaking?
- Consider different models.

In [1]:
```
!picat queens/queens-columns 8
```

```
CPU time 0.0 seconds. Backtracks: 24

Q.......
....Q...
.......Q
.....Q..
..Q.....
......Q.
.Q......
...Q....
```

In [2]:
```
!cat queens/queens-columns.pi
```

```
% n-queens, the "columns" model
import cp.

queens_columns(N, Q) =>
    Q = new_array(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]).

main([N]) =>
    N := to_int(N),
    queens_columns(N, Q),
    time2(solve(Q)),
    if N <= 32 then
        output(Q)
    end.

output(Q) =>
    N = Q.length,
    foreach(I in 1..N)
        foreach (J in 1..N)
            if Q[I] = J then
                print("Q")
            else
                print(".")
            end
        end,
        print("\n")
    end.
```

In [3]:
```
!picat queens/queens-board 8
```

```
CPU time 0.037 seconds. Backtracks: 8540

.......Q
...Q....
Q.......
..Q.....
.....Q..
.Q......
......Q.
....Q...
```

In [4]:
```
!cat queens/queens-board.pi
```

```
% n-queens, the "board" model
import cp.

queens_board(N, Board) =>
    Board = new_array(N, N),
    Board :: 0..1,

    sum([Board[I, J] : I in 1..N, J in 1..N]) #= N,

    % rows
    foreach(I in 1..N)
        sum([Board[I, J] : J in 1..N]) #<= 1
    end,
    % cols
    foreach(J in 1..N)
        sum([Board[I, J] : I in 1..N]) #<= 1
    end,
    % diags
    foreach(K in 1-N..N-1)
        sum([Board[I,J] : I in 1..N, J in 1..N, I-J = K ]) #<= 1
    end,
    foreach(K in 2..2*N)
        sum([Board[I,J] : I in 1..N, J in 1..N, I+J = K ]) #<= 1
    end.

main([N]) =>
    N := to_int(N),
    queens_board(N, Board),
    time2(solve(Board)),
    if N <= 32 then
        output(Board)
    end.

output(Board) =>
    N = Board.length,
    foreach(I in 1..N)
        foreach (J in 1..N)
            if Board[I, J] = 1 then
                print("Q")
            else
                print(".")
            end
        end,
        print("\n")
    end.
```

Sometimes it is best to model the problem in both ways and add *channelling constraints*. (Here it does not help.)

In [5]:
```
!picat queens/queens-channeling 8
```

```
CPU time 0.001 seconds. Backtracks: 24

Q.......
....Q...
.......Q
.....Q..
..Q.....
......Q.
.Q......
...Q....
```

In [6]:

```
!cat queens/queens-channeling.pi
```

```
% n-queens, both the "columns" and "board" models with channeling
import cp.

queens(N, Q, Board) =>
    % the two models
    queens_columns(N, Q),
    queens_board(N, Board),

    % channeling
    foreach(I in 1..N, J in 1..N)
        (Board[I,J] #= 1) #<=> (Q[I] #= J)
    end.

main([N]) =>
    N := to_int(N),
    queens(N, Q, Board),
    time2(solve(Q ++ Board)),
    if N <= 32 then
        output(Q)
    end.

queens_columns(N, Q) =>
    Q = new_array(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]).

queens_board(N, Board) =>
    Board = new_array(N, N),
    Board :: 0..1,

    sum([Board[I, J] : I in 1..N, J in 1..N]) #= N,

    % rows
    foreach(I in 1..N)
        sum([Board[I, J] : J in 1..N]) #<= 1
    end,
    % cols
    foreach(J in 1..N)
        sum([Board[I, J] : I in 1..N]) #<= 1
    end,
    % diags
    foreach(K in 1-N..N-1)
        sum([Board[I,J] : I in 1..N, J in 1..N, I-J = K ]) #<= 1
    end,
    foreach(K in 2..2*N)
        sum([Board[I,J] : I in 1..N, J in 1..N, I+J = K ]) #<= 1
    end.

output(Q) =>
    N = Q.length,
    foreach(I in 1..N)
        foreach (J in 1..N)
            if Q[I] = J then
                print("Q")
```

```
            else
                print(".")
            end
        end,
        print("\n")
    end.
```

Can the models be improved using symmetry breaking?

Search strategies

And other solver options: see Picat guide (Section 12.6) and the book (Section 3.5)

In [7]:
```
%load_ext ipicat
```

```
Picat version 3.9
```

In [8]:
```
%%picat -n queens
import cp. % try sat, also try mip with the dual model
queens(N, Q) =>
    Q = new_array(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]).
```

In [9]:
```
%%picat
main =>
    N = 24,
    queens(N, Q),
    time2(solve(Q)).
```

```
CPU time 0.123 seconds. Backtracks: 63778
```

Which search strategy could work well for our model?

Here's how we can test multiple search strategies (code adapted from the book):

In [10]:
```
%%picat

% variable selection strategies
selection(VarSels) => VarSels = [backward,constr,degree,ff,ffc,ffd,forward,inout,le

% value choice strategies
choice(ValChoices) => ValChoices = [down,reverse_split,split,up,updown].

main =>
    selection(VarSels),
    choice(ValChoices),

    Strategies = [[VarSel, ValChoice] : VarSel in VarSels, ValChoice in ValChoices]
```

```
    Timeout = 10000,
    Ns = [100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200],

    println("Successful strategies:"),
    foreach (N in Ns)
        foreach(Strategy in Strategies)
            queens(N,Q),
            time_out(solve(Strategy, Q),Timeout,Status),
            if Status != success then
                Strategies := delete(Strategies, Strategy)
            end
        end,
        printf("N=%d: %w\n", N, Strategies)
    end.
```

```
Successful strategies:
N=100: [[ff,down],[ff,reverse_split],[ff,split],[ff,up],[ff,updown],[ffc,down],[ffc,reverse_split],[ffc,split],[ffc,up],[ffc,updown],[ffd,down],[ffd,reverse_split],[ffd,split],[ffd,up],[ffd,updown]]
N=110: [[ff,updown],[ffc,updown]]
N=120: [[ff,updown]]
N=130: [[ff,updown]]
N=140: []
N=150: []
N=160: []
N=170: []
N=180: []
N=190: []
N=200: []
```

Exercises

Exercise: Magic square

Arrange numbers $1, 2, \ldots, n^2$ in a square such that every row, every column, and the two main diagonals all sum to the same quantity.

- Try to find the best model, solver and search strategy.
- How many magic squares are there for a given n?
- Allow also for a partially filled instance.

Exercise: Minesweeper

Identify the positions of all mines in a given board. Try the following instance (from the book):

```
Board = {
    {_,_,2,_,3,_},
    {2,_,_,_,_,_},
    {_,_,2,4,_,3},
```

```
        {1,_,3,4,_,_},
        {_,_,_,_,_,3},
        {_,3,_,3,_,_}
    }.
```

Knapsack

There are two common versions of the problem: the general **knapsack** problem:

> Given a set of items, each with a weight and a value, determine **how many of each item** to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

And the **0-1 knapsack** problem:

> Given a set of items, each with a weight and a value, determine **which items** to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

(In a general knapsack problem, we can take any number of each item, in the 0-1 version we can take at most one of each.)

Example of an instance:

A thief breaks into a department store (general knapsack) or into a home (0-1 knapsack). They can carry 23kg. Which items (and how many of each, in the general version) should they take to maximize profit? There are the following items:

- a TV (weighs 15kg, costs $500),
- a desktop computer (weighs 11kg, costs $350)
- a laptop (weighs 5kg, costs $230),
- a tablet (weighs 1kg, costs $115),
- an antique vase (weighs 7kg, costs $180),
- a bottle of whisky (weighs 3kg, costs $75), and
- a leather jacket (weighs 4kg, costs $125).

This instance is given in the file `data.pi`.

In [11]:
```
!cat knapsack/data.pi
```

```
instance(Items, Capacity, Values, Weights) =>
    Items = {"tv", "desktop", "laptop", "tablet", "vase", "bottle", "jacket"},
    Capacity = 23,
    Values = {500,350,230,115,180,75,125},
    Weights = {15,11,5,1,7,3,4}.
```

What search strategies could be suitable for Knapsack?