NOPT042 Constraint programming: Tutorial 4 – Search strategies

What was in the lecture? Nothing, the lecture was canceled.

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 the *dual* model.

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

```
CPU time 0.0 seconds. Backtracks: 24

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

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

```
% n-queens, primal model
import cp.

queens_primal(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_primal(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-dual 8
```

```
CPU time 0.037 seconds. Backtracks: 8540

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

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

```
% n-queens, dual model
import cp.

queens_dual(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_dual(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, primal and dual models with channeling
import cp.

queens(N, Q, Board) =>
    % primal
    queens_primal(N, Q),
    queens_dual(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_primal(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_dual(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 3.7
```

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 0.122 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

selection(VarSels) => VarSels = [backward,constr,degree,ff,ffc,ffd,forward,inout,le
choice(ValSels) => ValSels = [down,reverse_split,split,up,updown].

main =>
    selection(VarSels),
    choice(ValSels),
    Timeout = 2000,
    Ns = [80, 100, 120],

    foreach (N in Ns, VarSel in VarSels, ValSel in ValSels)
        queens(N,Q),
```

```
        time_out(solve([VarSel,ValSel], Q),Timeout,Status),
        if Status = success then
            println([N,VarSel,ValSel])
        end
    end.
```

```
[80,ff,down]
[80,ff,reverse_split]
[80,ff,split]
[80,ff,up]
[80,ff,updown]
[80,ffc,down]
[80,ffc,reverse_split]
[80,ffc,split]
[80,ffc,up]
[80,ffc,updown]
[80,ffd,down]
[80,ffd,reverse_split]
[80,ffd,split]
[80,ffd,up]
[100,ff,down]
[100,ff,reverse_split]
[100,ff,split]
[100,ff,up]
[100,ff,updown]
[100,ffc,down]
[100,ffc,reverse_split]
[100,ffc,split]
[100,ffc,up]
[100,ffc,updown]
[100,ffd,down]
[100,ffd,reverse_split]
[100,ffd,split]
[100,ffd,up]
[100,ffd,updown]
[120,ff,updown]
[120,ffc,updown]
[120,ffd,down]
[120,ffd,reverse_split]
[120,ffd,split]
[120,ffd,up]
[120,ffd,updown]
```

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?