

# NOPT042 Constraint programming: Tutorial 7 – Rostering, table constraint

```
In [1]: %load_ext ipicat
```

Picat version 3.5#5

## The constraint `regular`

```
regular(L, Q, S, M, Q0, F)
```

Given a finite automaton (DFA or NFA) of  $Q$  states numbered  $1, 2, \dots, Q$  with input from  $\{1, \dots, S\}$ , transition matrix  $M$ , initial state  $Q_0$  ( $1 \leq Q_0 \leq Q$ ), and a list of accepting states  $F$ , this constraint is true if the list  $L$  is accepted by the automaton. The transition matrix  $M$  represents a mapping from  $\{1, \dots, Q\} \times \{1, \dots, S\}$  to  $\{0, \dots, Q\}$ , where 0 denotes the error state. For a DFA, every entry in  $M$  is an integer, and for an NFA, entries can be a list of integers.

---from [the guide](#)

## Example: Global contiguity

Given a 0-1 sequence, express that if there are 1's, they must form a single, contiguous subsequence, e.g. accept `0000` and `0001111100` but not `00111010`. (Problem from [the book](#).)

```
In [2]: !picat global-contiguity/global_contiguity.pi 0011100
!picat global-contiguity/global_contiguity.pi 0110111
```

ok

```
*** error(failed,main/1)
```

```
In [3]: !cat global-contiguity/global_contiguity.pi
```

```

/*****
Adapted from
global_contiguity.pi
from Constraint Solving and Planning with Picat, Springer
by Neng-Fa Zhou, Hakan Kjellerstrand, and Jonathan Fruhman
*****/
import cp.

main([Xstr]) =>
    X = map(to_int,Xstr),
    global_contiguity(X),
    solve(X),
    println("ok").

global_contiguity(X) =>
    N = X.length,

    % This uses the regular expression "0*1*0*" to
    % require that all 1's (if any) in an array
    % appear contiguously.
    Transition = [
        [1,2], % state 1: 0*
        [3,2], % state 2: 1*
        [3,0] % state 3: 0*
    ],
    NStates = 3,
    InputMax = 2,
    InitialState = 1,
    FinalStates = [1,2,3],

    RegInput = new_list(N),
    RegInput :: 1..InputMax, % 1..2

    % Translate X's 0..1 to RegInput's 1..2
    foreach (I in 1..N)
        RegInput[I] #= X[I]+1
    end,

    regular(RegInput,NStates,InputMax,
        Transition,InitialState,FinalStates).

```

## Example: Nurse roster

Schedule the shifts of `NumNurses` nurses over `NumDays` days. Each nurse is scheduled for each day as either: (d) on day shift, (n) on night shift, or (o) off. In each four day period a nurse must have at least one day off, and no nurse can be scheduled for 3 night shifts in a row.

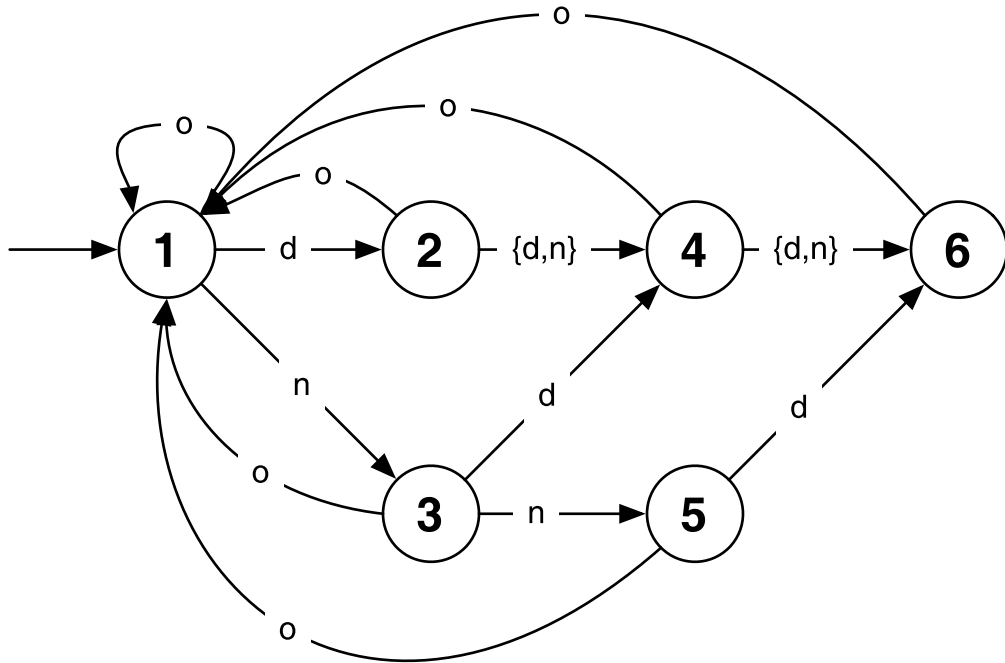
We require `ReqDay` nurses on day shift each day, and `ReqNight` nurses on night shift, and that each nurse takes at least `MinNight` night shifts. (Problem from [the MiniZinc tutorial](#), a similar problem is in [the book](#).)

```
In [4]: !cat nurse-roster/instance.pi
```

```
instance(NumNurses, NumDays, ReqDay, ReqNight, MinNight) =>
    NumNurses      = 14,
    NumDays         = 7,
    ReqDay          = 3, % minimum number in day shift
    ReqNight        = 2, % minimum number in night shift
    MinNight        = 2. % minimum night shifts for each nurse
```

```
In [5]: !picat nurse-roster/nurse_rostering_regular.pi instance.pi
```

dddodnn  
 dddodnn  
 dddodnn  
 dddodnn  
 dddodnn  
 dddodnn  
 dddodnn  
 dddodnn  
 dddodnn  
 ddodnno  
 ddodnno  
 nnonodd  
 nonnodd  
 onndodd



In [6]: `!cat nurse-roster/nurse_rostering_regular.pi`

```

/*****
Adapted from Constraint Solving and Planning with Picat, Springer
by Neng-Fa Zhou, Hakan Kjellerstrand, and Jonathan Fruhman
*****/
import cp.

main([Filename]) =>
    cl(Filename),
    instance(NumNurses, NumDays, ReqDay, ReqNight, MinNight),
    nurse_rostering(NumNurses, NumDays, ReqDay, ReqNight, MinNight, Roster, Stat),
    Vars = Roster.vars() ++ Stat.vars(),
    solve(Vars),
    output(Roster).

nurse_rostering(NumNurses, NumDays, ReqDay, ReqNight, MinNight, Roster, Stat) =>

    % The DFA for the regular constraint.
    Transition = [
    % d n o
    [2,3,1], % state 1
    [4,4,1], % state 2
    [4,5,1], % state 3
    [6,6,1], % state 4
    [6,0,1], % state 5
    [0,0,1] % state 6
    ],
    NStates      = Transition.length, % number of states
    InputMax     = 3,                % 3 states
    InitialState = 1,                % start at state 1
    FinalStates  = 1..6,             % all states are final
    DayShift     = 1,
    NightShift   = 2,
    OffShift     = 3,

    % decision variables
    Roster = new_array(NumNurses, NumDays),
    Roster :: DayShift..OffShift,

    % summary of the shifts: [day,night,off]
    Stat = new_array(NumDays,3),
    Stat :: 0..NumNurses,

    % constraints
    foreach (I in 1..NumNurses)
        regular([Roster[I,J] : J in 1..NumDays],
            NStates,
            InputMax,
            Transition,
            InitialState,
            FinalStates)
    end,
    % statistics for each day
    foreach (Day in 1..NumDays)
        foreach (Type in 1..3)
            Stat[Day,Type] #= sum([Roster[Nurse,Day] #= Type : Nurse in 1..NumNurses])
        end
    end

```

```

end,
sum([Stat[Day,Type] : Type in 1..3]) #= NumNurses,
% For each day the must be at least 3 nurses with
% day shift, and 2 nurses with night shift
Stat[Day,DayShift] #>= ReqDay,
Stat[Day,NightShift] #>= ReqNight
end,

% each nurse gets MinNight shifts
foreach (Nurse in 1..NumNurses)
  sum([Roster[Nurse, Day] #= NightShift : Day in 1..NumDays]) #>= MinNight
end.

output(Roster) =>
Shifts = new_map(3,[1=d,2=n,3=o]),
foreach(Nurse in Roster)
  foreach(I in 1..Nurse.length)
    print(get(Shifts,Nurse[I]))
  end,
  print("\n")
end.

```

## Constraint `sliding_sum` (not available in Picat)

```

sliding_sum(Low, Up, Seq, Variables) =>
  foreach(I in 1..Variables.length-Seq+1)
    Sum #= sum([Variables[J] : J in I..I+Seq-1]),
    Sum #>= Low,
    Sum #=< Up
  end.

```

-- from [Hakank's Picat webpage](#), model `sliding_sum.pi`.

## The table constraint

A *table constraint*, or an *extensional constraint*, over a tuple of variables specifies a set of tuples that are allowed (called positive) or disallowed (called negative) for the variables. A positive constraint takes the form

```
table_in(Vars,R)
```

where `Vars` is either a tuple of variables or a list of tuples of variables, and `R` is a list of tuples in which each tuple takes the form  $[a_1, \dots, a_n]$ , where  $a_i$  is an integer or the don't-care symbol `*`. A negative constraint takes the form:

```
table_notin(Vars, R)
```

```
---from [the guide](http://picat-lang.org/download/picat_guide.pdf)
```

## Example: Nurse roster using `table_in`

Model the above nurse roster problem using the constraint `table_in`. The model is slower, we will need a simpler instance. And for simplicity assume that

```
NumDays = 7.
```

```
In [7]: !cat nurse-roster/instance2.pi
```

```
instance(NumNurses, NumDays, ReqDay, ReqNight, MinNight) =>
    NumNurses      = 8,
    NumDays        = 7,
    ReqDay          = 2, % minimum number in day shift
    ReqNight        = 2, % minimum number in night shift
    MinNight        = 1. % minimum night shifts for each nurse
```

```
In [8]: !picat nurse-roster/nurse_rostering_table.pi instance2.pi
```

```
ddonnoo
ddonnoo
donnood
donnood
nnooddo
nnooddo
ooddonn
ooddonn
```

```
In [9]: !cat nurse-roster/nurse_rostering_table.pi
```

```

/*****
Adapted from Constraint Solving and Planning with Picat, Springer
by Neng-Fa Zhou, Hakan Kjellerstrand, and Jonathan Fruhman
*****/
import cp.

main([Filename]) =>
    cl(Filename),
    instance(NumNurses, NumDays, ReqDay, ReqNight, MinNight),
    nurse_rostering(NumNurses, NumDays, ReqDay, ReqNight, MinNight, Roster, Stat),
    Vars = Roster.vars() ++ Stat.vars(),
    solve(Vars),
    output(Roster).

% rotate valid schedules
rotate_left(L) = rotate_left(L,1).
rotate_left(L,N) = slice(L,N+1,L.length) ++ slice(L,1,N).

nurse_rostering(NumNurses, NumDays, ReqDay, ReqNight, MinNight, Roster, Stat) =>

    % Only works for 7-day rosters!
    NumDays = 7,

    DayShift    = 1,
    NightShift   = 2,
    OffShift     = 3,
    D = 1,
    N = 2,
    O = 3,

    % Valid 7 day schedules:
    % - up to rotation:
    Valid_up_to_rotation = [
        [D,D,D,D,D,O,O],
        [N,O,N,O,D,D,O],
        [N,N,O,O,D,D,O]
    ],
    % - create all rotational variants
    Valid = [],
    foreach (V in Valid_up_to_rotation, R in 0..V.length-1)
        Rot = rotate_left(V,R).to_array(),
        Valid := Valid ++ [Rot]
    end,

    % decision variables:
    % - the roster
    Roster = new_array(NumNurses, NumDays),
    Roster :: DayShift..OffShift,

    % - summary of the shifts: [day,night,off]
    Stat = new_array(NumDays,3),
    Stat :: 0..NumNurses,

    % constraints

```



```

% valid schedule
foreach (Nurse in 1..NumNurses)
    table_in([Roster[Nurse,Day] : Day in 1..NumDays].to_array(), Valid)
end,

% statistics for each day
foreach (Day in 1..NumDays)
    foreach (Type in 1..3)
        Stat[Day,Type] #= sum([Roster[Nurse,Day] #= Type : Nurse in 1..NumNurses])
    end,
    sum([Stat[Day,Type] : Type in 1..3]) #= NumNurses,
    % For each day the must be at least 3 nurses with
    % day shift, and 2 nurses with night shift
    Stat[Day,DayShift] #>= ReqDay,
    Stat[Day,NightShift] #>= ReqNight
end,

% each nurse gets MinNight shifts
foreach (Nurse in 1..NumNurses)
    sum([Roster[Nurse, Day] #= NightShift : Day in 1..NumDays]) #>= MinNight
end.

output(Roster) =>
Shifts = new_map(3,[1=d,2=n,3=o]),
foreach(Nurse in Roster)
    foreach(I in 1..Nurse.length)
        print(get(Shifts,Nurse[I]))
    end,
    print("\n")
end.

```

## Example: Graph homomorphism

Given a pair of graphs  $G, H$ , find all homomorphisms from  $G$  to  $H$ . A *graph homomorphism* is a function  $f : V(G) \rightarrow V(H)$  such that

$$\{u, v\} \in E(G) \implies \{f(u), f(v)\} \in E(H)$$

.

- Generalizes graph  $k$ -coloring ( $c : G \rightarrow K_k$ )
- Easier version: oriented graphs
- How would you model the Graph Isomorphism Problem?