NOPT042 Constraint programming: Tutorial 7 – Rostering, table constraint

What was in Lecture 5

Path consistency

- **arc consistency**: never destroys solutions, sometimes can find a solution (without backtracking) -- iff all domains reduced to 1 element
- **path consistency**: for any path in variables, if assignment of start point and end point satisfy all binary constraints between them, then there is a consistent path in the constraint network
- enough to enforce for paths of length two ("every edge [in the constraint network] extends to any triangle")
- PC is stronger, removes pairs of inconsistent values
- but it is more expensive
- algorithms: PC-1, PC-2 [, PC-3, PC-4, PC-5]
- directional path consistency
- drawbacks of PC: memory consumption, bad strength/efficiency ration, modifies the constraint network (adds redundant constraints, changes connectivity, ruins graph-structure-based heuristics), still not complete
- restricted path consistency (AC, only check PC for pairs which are the only support for one of the values)

Exercise:

- Explain why PC is equivalent to path consistency for paths of length two
- Give an example of an instance which is AC but not PC
- Give an example of an instance which is PC (with all domains nonempty) but not solvable

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

```
Picat version 3.7
```

The constraint `regular`

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

> Given a finite automaton (DFA or NFA) of Q states numbered $1, 2, \ldots, Q$ with input from $\{1, \ldots, S\}$, 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 $\{1, \ldots, Q\} \times \{1, \ldots, S\}$ to $\{0, \ldots, Q\}$, 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

Exercise: 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 0011100
!picat global-contiguity/global_contiguity 0110111
```

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

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

```
% Adapted 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,
  InputMax = 2,

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

  % DFA for the regex "0*1*0*"
  Transition = [
    [1,2], % state 1: 0*
    [3,2], % state 2: 1*
    [3,0]  % state 3: 0*
  ],
  NStates = 3,
  InitialState = 1,
  FinalStates = [1,2,3],

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

Exercise: 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
```

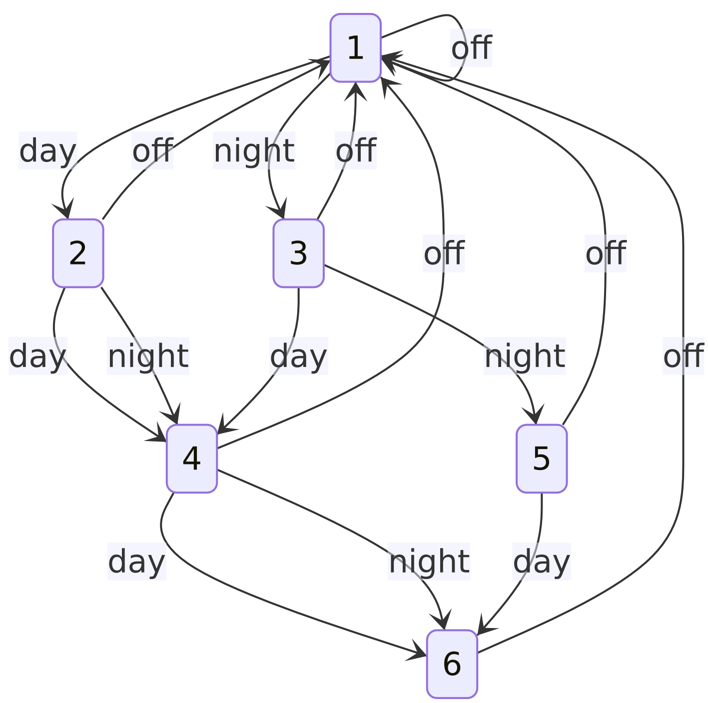
```
instance(NumNurses, NumDays, ReqDay, ReqNight, MinNight) =>
    NumNurses     = 14,
    NumDays       = 7,
    ReqDay        = 3, % minimum number in day shift
    ReqNight      = 2, % minimum number in night shift
    MinNight      = 2. % minimum night shifts for each nurse
```

In [5]:
```
!picat nurse-roster/nurse_roster_regular instance
```

```
CPU time 0.0 seconds. Backtracks: 12

|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  day  |  off  |  day  | night | night |
|  day  |  day  |  off  |  day  | night | night |  off  |
|  day  |  day  |  off  |  day  | night | night |  off  |
| night | night |  off  | night |  off  |  day  |  day  |
| night |  off  | night | night |  off  |  day  |  day  |
|  off  | night | night |  day  |  off  |  day  |  day  |
```

State diagram of the DFA (in mermaid, will not render in the RISE slides), start state is 1, all states are final:



In [6]:
```
!cat nurse-roster/nurse_roster_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(),
  time2(solve(Vars)),
  output(Roster).

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

  % The DFA for the regular constraint.
  Transition = [
    % day-night-off
    [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

  % constraints

  % valid schedule
  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,
    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="|  day  ",2="| night ",3="|  off  "]),
  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 $[a_1, \ldots, a_n]$, 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

Exercise: 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_roster_table instance2
```

```
CPU time 0.037 seconds. Backtracks: 7796

|  day  |  day  |  off  | night | night |  off  |  off  |
|  day  |  day  |  off  | night | night |  off  |  off  |
|  day  |  off  | night | night |  off  |  off  |  day  |
|  day  |  off  | night | night |  off  |  off  |  day  |
| night | night |  off  |  off  |  day  |  day  |  off  |
| night | night |  off  |  off  |  day  |  day  |  off  |
|  off  |  off  |  day  |  day  |  off  | night | night |
|  off  |  off  |  day  |  day  |  off  | night | night |
```

In [9]:
```
!cat nurse-roster/nurse_roster_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(),
  time2(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, D = 1,
  NightShift = 2, N = 2,
  OffShift   = 3, 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="|  day  ",2="| night ",3="|  off  "]),
  foreach(Nurse in Roster)
    foreach(I in 1..Nurse.length)
      print(get(Shifts,Nurse[I]))
    end,
    print("|\n")
  end.
```

Exercise: Graph homomorphism

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

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

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