

## Chapter 14: Finite Set and Continuous Variables - SONET Design Problem

Helmut Simonis

Cork Constraint Computation Centre  
Computer Science Department  
University College Cork  
Ireland

ECLiPSe ELearning [Overview](#)



## Licence

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License.

To view a copy of this license, visit [http:](http://creativecommons.org/licenses/by-nc-sa/3.0/)

[//creativecommons.org/licenses/by-nc-sa/3.0/](http://creativecommons.org/licenses/by-nc-sa/3.0/) or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.



# Outline

- 1 Problem
- 2 Program
- 3 Search
- 4 Conclusions



# What we want to introduce

- Finite set variables
- Continuous domains
- Optimization from below
- Advanced symmetry breaking
- SONET design problem without inter-ring flows



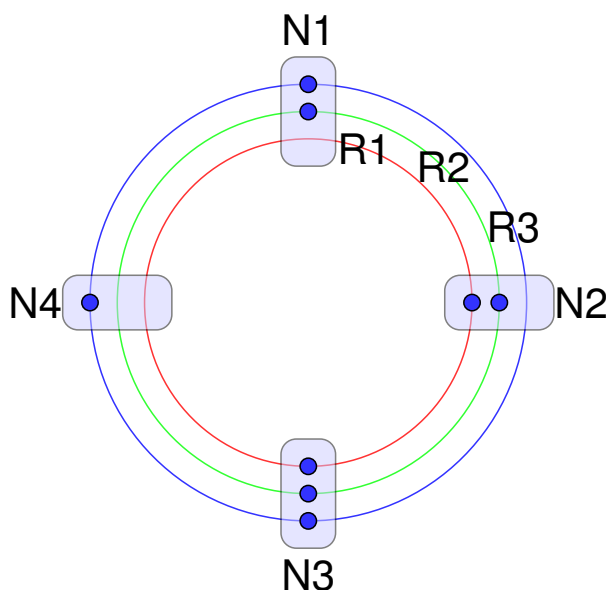
# Problem Definition

## SONET Design Problem

We want to design a network with multiple SONET rings, minimizing ADM equipment. Traffic can only be transported between nodes connected to the same ring, not between rings. Traffic demands between nodes are given. Decide which nodes to place on which ring(s), respecting a maximal number of ADM per ring, and capacity limits on ring traffic. If two nodes are connected on more than one ring, the traffic between them can be split arbitrarily between the rings. The objective is to minimize the overall number of ADM.



# Example



3 rings, 4 nodes, 8 ADM  
 Every node connected to at least one ring  
 On every ring are at least two nodes  
 N1 connected to R2 and R3  
 N4 and N2 can't talk to each other  
 Traffic between N1



R1 or R2 or both

## Data

- Demands  $d \in D$  between nodes  $f_d$  and  $t_d$  of size  $s_d$
- Rings  $R$ , total of  $|R| = r$  rings
- Each ring has capacity  $c$
- Nodes  $N$



## Model

- Primary model integer 0/1 variables  $x_{ik}$ 
  - Node  $i$  has a connection to ring  $k$
  - A node can be connected to more than one ring
- Continuous  $[0..1]$  variables  $f_{dk}$ 
  - Which fraction of total traffic of demand  $d$  is transported on ring  $k$
  - A demand can use a ring only if both end-points are connected to it



# Constraints

$$\min \sum_{i \in N} \sum_{k \in R} x_{ik}$$

s.t.

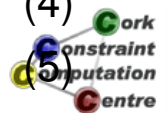
$$\sum_{i \in N} x_{ik} \leq r \quad (1)$$

$$\sum_{k \in R} f_{dk} = 1 \quad (2)$$

$$\sum_{d \in D} s_d * f_{dk} \leq c \quad (3)$$

$$f_{dk} \leq x_{f_d k} \quad (4)$$

$$f_{dk} \leq x_{t_d k} \quad (5)$$



# Dual Models

- Introducing finite set variables
- Range over sets of integers, not just integers
- Most useful when we don't know the number of items involved
- Here: for each node, the rings on which it is placed
- Could be one, could be two, or more
- Hard to express with finite domain variables alone



## Dual Model 1

- Finite set variables  $N_i$ 
  - Which rings node  $i$  is connected to
- Cardinality finite domain variables  $n_i$ 
  - $|N_i| = n_i$



## Dual Model 2

- Finite set variables  $R_k$ 
  - Which nodes ring  $k$  is connected to
- Cardinality finite domain variables  $r_k$ 
  - $|R_k| = r_k$



## Channeling between models

- Use the zero/one model as common ground
- $x_{ik} = 1 \Leftrightarrow k \in N_i$
- $x_{ik} = 1 \Leftrightarrow i \in R_k$



## Constraints in dual models

- For every demand, source and sink must be on (at least one) shared ring
  - $\forall d \in D : |N_{f_d} \cap N_{t_d}| \geq 1$
- Every node must be on a ring
  - $n_i \geq 1$
- A ring can not have a single node connected to it
  - $r_k \neq 1$



## Assignment Strategy

- Cost based decomposition
- Assign total cost first
- Then assign  $n_j$  variables
- Finally, assign  $x_{jk}$  variables
- If required, fix flow  $f_{dk}$  variables
- Might leave flows as bound-consistent continuous domains



## Optimization from below

- Optimization handled by assigning cost first
- Enumerate values increasing from lower bound
- First feasible solution is optimal
- Depends on proving infeasibility rapidly
- Does not provide sub-optimal initial solutions





## Redundant Constraints

- Deduce bounds in  $n_i$  variables
  - Helps with finding  $n_i$  assignment which can be extended
- Symmetry Breaking



## Symmetries

- Typically no symmetries between demands
- Full permutation symmetry on rings
- Gives  $r!$  permutations
- These must be handled somehow
- Further symmetries if capacity seen as discrete channels



## Symmetry Breaking Choices

- As part of assignment routine
  - SBDS (symmetry breaking during search)
  - Define all symmetries as parameter
  - Search routine eliminates symmetric sub-trees
- By stating ordering constraints
  - As shown in the BIBD example
  - Ordering constraints not always compatible with search heuristic
  - Particular problem of dynamic variable ordering



## Defining finite set variables

- Library `ic_sets`
- Domain definition `X :: Low..High`
  - *Low, High* sets of integer values, e.g. `[1, 3, 4]`
- or `intsets(L, N, Min, Max)`
  - `L` is a list of `N` set variables
  - each containing all values between `Min` and `Max`



## Using finite set variables

- Set Expressions:  $A \wedge B, A \vee B$
- Cardinality constraint:  $\#(\text{Set}, \text{Size})$ 
  - *Size* integer or finite domain variable
- `membership_booleans(Set, Booleans)`
  - Channeling between set and 0/1 integer variables



## Using continuous variables

- Library `ic` handles both
  - Finite domain variables
  - Continuous variables
- Use floats as domain bounds, e.g.  $X :: 0.0 .. 1.0$
- Use  $\$=$  etc for constraints instead of  $\#=$
- Bounds reasoning similar to finite case
- But must deal with safe rounding
- Not all constraints deal with continuous variables



## Ambiguous Import

- Multiple solvers define predicates like `::`
- If we load multiple solvers in the same module, we have to tell ECLIPSe which one to use
- Compiler does not deduce this from context!
- So
  - `ic:(X :: 1..3)`
  - `ic_sets:(X :: [] .. [1,2,3])`
- Otherwise, we get loads of error messages
- Happens whenever two modules export same predicate



## Top-level predicate

```
:-module(sonet).
:-export(top/0).
:-lib(ic), lib(ic_global), lib(ic_sets).
```

```
top:-
    problem(NrNodes, NrRings, Demands,
            MaxRingSize, ChannelSize),
    length(Demands, NrDemands),
    ...
```



## Matrix of $x_{ik}$ integer variables

```

...
dim(Matrix, [NrNodes, NrRings]),
ic: (Matrix[1..NrNodes, 1..NrRings] :: 0..1),
...

```



## Node and ring set variables

```

...
dim(Nodes, [NrNodes]),
intsets(Nodes[1..NrNodes], NrNodes, 1, NrRings),
dim(NodeSizes, [NrNodes]),
ic: (NodeSizes[1..NrNodes] :: 1..NrRings),
dim(Rings, [NrRings]),
intsets(Rings[1..NrRings], NrRings, 1, NrNodes),
dim(RingSizes, [NrRings]),
ic: (RingSizes[1..NrRings] :: 0..MaxRingSize),
...

```



## Channeling node set variables

```

...
(for (I, 1, NrNodes),
  param(Matrix, Nodes, NodeSizes, NrRings) do
    subscript (Nodes, [I], Node),
    subscript (NodeSizes, [I], NodeSize),
    # (Node, NodeSize),
    membership_booleans (Node,
                          Matrix[I, 1..NrRings])
  ),
...

```



## Channeling ring set variables

```

...
(for (J, 1, NrRings),
  param(Matrix, Rings, RingSizes, NrNodes) do
    subscript (Rings, [J], Ring),
    subscript (RingSizes, [J], RingSize),
    RingSize #\= 1,
    # (Ring, RingSize),
    membership_booleans (Ring,
                          Matrix[1..NrNodes, J])
  ),
...

```



## Demand ends must be (on atleast one) same ring

```

...
(foreach(demand(I, J, _Size), Demands),
  param(Nodes, NrRings) do
    subscript(Nodes, [I], NI),
    subscript(Nodes, [J], NJ),
    ic:(NonZero :: 1..NrRings),
    #(NI /\ NJ, NonZero)
  ),
...

```



## Flow Variables

```

...
dim(Flow, [NrDemands, NrRings]),
ic:(Flow[1..NrDemands, 1..NrRings]::0.0 .. 1.0),
(for(I, 1, NrDemands),
  param(Flow, NrRings) do
    (for(J, 1, NrRings),
      fromto(0.0, A, A+F, Term),
      param(Flow, I) do
        subscript(Flow, [I, J], F)
      ),
      eval(Term) $= 1.0
    ),
...

```



## Ring Capacity Constraints

```

...
(for (I, 1, NrRings),
  param (Flow, Demands, ChannelSize) do
    (foreach (demand (_, _, Size), Demands),
      count (J, 1, _),
      fromto (0.0, A, A+Size*F, Term),
      param (Flow, I) do
        subscript (Flow, [J, I], F)
      ),
      eval (Term) $=< ChannelSize
    ),
  ...

```



## Linking $x_{ik}$ and $f_{dk}$ variables

```

...
(foreach (demand (From, To, _), Demands),
  count (I, 1, _),
  param (Flow, Matrix, NrRings) do
    (for (K, 1, NrRings),
      param (I, From, To, Flow, Matrix) do
        subscript (Flow, [I, K], F),
        subscript (Matrix, [From, K], X1),
        subscript (Matrix, [To, K], X2),
        F $=< X1,
        F $=< X2
      )
    ),
  ...

```





## Setting up degrees

```

...
dim(Degrees, [NrNodes]),
(for (I, 1, NrNodes),
  param(Degrees) do
    subscript (Degrees, [I], Degree),
    neighbors (I, Neighbors),
    length (Neighbors, Degree)
  ),
...

```



## Defining cost and assigning values

```

...
sumlist (NodeSizesList, Cost),
assign (Cost, Handle, NrNodes, Degrees,
        NodeSizes, Matrix).

```



## Assignment Routines

```

assign (Cost, Handle, NrNodes, Degrees,
        NodeSizes, Matrix) :-
  indomain (Cost),
  order_sizes (NrNodes, Degrees, NodeSizes,
               OrderedSizes),
  search (OrderedSizes, 1, input_order, indomain,
          complete, []),
  order_vars (Degrees, NodeSizes, Matrix,
              VarAssign),
  search (VarAssign, 0, input_order, indomain_max,
          complete, []).

```



## Order ring size variables by increasing degree

```

order_sizes (NrNodes, Degrees, NodeSizes,
             OrderedSizes) :-
  (for (I, 1, NrNodes),
   foreach (t (X, D), Terms),
   param (Degrees, NodeSizes) do
     subscript (Degrees, [I], D),
     subscript (NodeSizes, [I], X)
  ),
  sort (2, =<, Terms, OrderedSizes).

```



## Ordering decision variables

```

order_vars (Degrees, NodeSizes, Matrix, VarAssign) :-
  dim (Matrix, [NrNodes, NrRings]),
  (for (I, 1, NrNodes),
   foreach (t (Size, Y, I), Terms),
   param (Degrees, NodeSizes) do
     subscript (NodeSizes, [I], Size),
     subscript (Degrees, [I], Degree),
     Y is -Degree
  ),
  sort (0, =<, Terms, Sorted),
  ...

```



## Ordering decision variables

```

...
(foreach (t (_, _, I), Sorted),
 fromto (VarAssign, A1, A, []),
 param (NrRings, Matrix) do
  (for (J, 1, NrRings),
   fromto (A1, [X|AA], AA, A),
   param (I, Matrix) do
     subscript (Matrix, [I, J], X)
  )
) .

```



## Data (13 nodes, 7 rings, 24 demands)

```

problem(13, 7,
  [demand(1, 9, 8), demand(1, 11, 2), demand(2, 3, 25),
   demand(2, 5, 5), demand(2, 9, 2), demand(2, 10, 3),
   demand(2, 13, 4), demand(3, 10, 2), demand(4, 5, 4),
   demand(4, 8, 1), demand(4, 11, 5), demand(4, 12, 2),
   demand(5, 6, 5), demand(5, 7, 4), demand(7, 9, 5),
   demand(7, 10, 2), demand(7, 12, 6), demand(8, 10, 1),
   demand(8, 12, 4), demand(8, 13, 1), demand(9, 12, 5),
   demand(10, 13, 9), demand(11, 13, 3),
   demand(12, 13, 2)
], 5, 40).

```



## Neighbors of a node

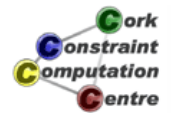
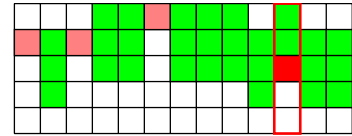
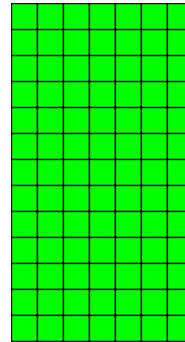
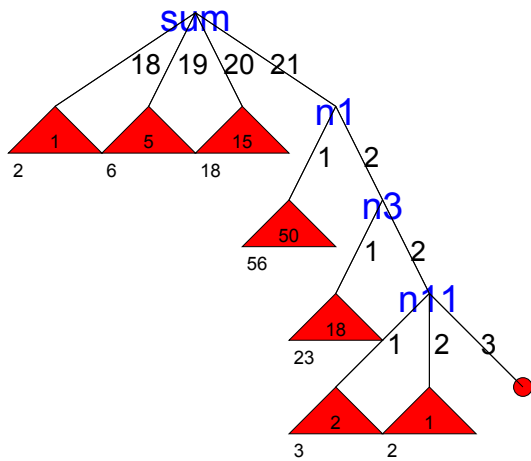
```

neighbors(N, List) :-
  problem(_, _, Demands, _, _),
  (foreach(demand(I, J, _), Demands),
   fromto([], A, A1, List),
   param(N) do
     (N = I ->
      A1 = [J|A]
     ; N = J ->
      A1 = [I|A]
     ;
      A1 = A
    )
  ).

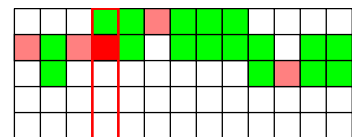
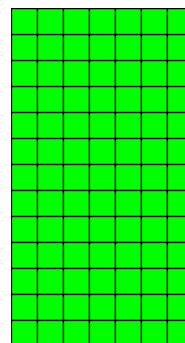
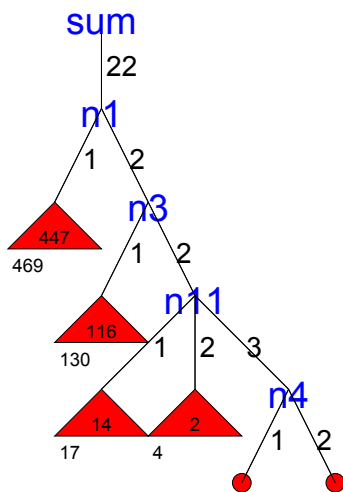
```



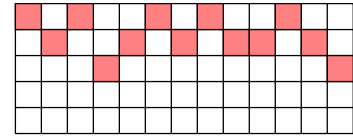
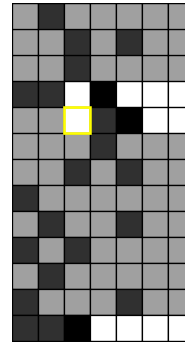
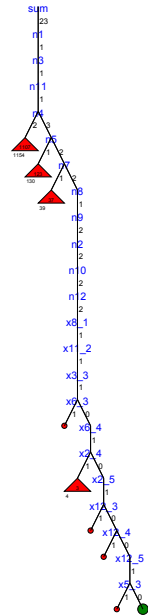
# Search at Cost 18-21



# Search at Cost 22



# Search at Cost 23



# Conclusions

- Introduced finite set and continuous domain solvers
- Finite set variables useful when values are sets of integers
- Useful when number of items assigned are unknown
- Can be linked with finite domains (cardinality) and 0/1 index variables



## Continuous domain variables

- Allow to reason about non-integral values
- Bound propagation similar to bound propagation over integers
- Difficult to enumerate values
- Assignment by domain splitting



## SONET Problem

- Example of optical network problems
- Competitive solution by combination of techniques
- Channeling, redundant constraints, symmetry breaking
- Decomposition by branching on objective value



## More Information



Barbara M. Smith.

Symmetry and search in a network design problem.

In Roman Barták and Michela Milano, editors, *CPAIOR*,  
volume 3524 of *Lecture Notes in Computer Science*, pages  
336–350. Springer, 2005.

