# Modeling Functions

Peter Stuckey

## Deciding Functions

- Many combinatorial problems have the form:
  - assign to each object in one set DOM
  - a value from another set COD
- We can interpret this as
  - Defining a function DOM ➔ COD
  - Or partitioning the set DOM (in sets labelled by COD)
- This function could be
  - injective: assignment problem
  - bijective (DOM = COD): matching problem



## Assignment problems

- Pure Assignment Problem
  - *n* workers
  - and *m* tasks
  - assign each worker to a different task to maximize profit.
- Example problem
  - $n = 4$, $m = 5$, profit matrix is

| | t1 | t2 | t3 | t4 | t5 |
|---|---|---|---|---|---|
| w1 | 7 | 1 | 3 | 4 | 6 |
| w2 | 8 | 2 | 5 | 1 | 4 |
| w3 | 4 | 3 | 7 | 2 | 5 |
| w4 | 3 | 1 | 6 | 3 | 6 |



## Assignment Data and Decisions

- Data

```
int: n;
set of int: DOM = 1..n;
int: m;
set of int: COD = 1..m;
array[DOM,COD] of int: profit;
```

- What are the decisions?

```
array[DOM] of var COD: task;
```

- What is the objective?

```
maximize sum(w in DOM)
             (profit[w,task[w]]);
```

## Assignment Constraints

- Each task is assigned to at most one worker

```
forall(t in COD)
    (sum(w in DOM)
        (bool2int(task[w] = t)) <= 1);
```

- Alternatively
- Each two workers are assigned different tasks

```
forall(w1, w2 in DOM where w1 < w2)
      (task[w1] != task[w2]);
```

- Which is better?



## Global Constraints

- Lets not choose (?)
  - Different solvers will prefer different representations
- Record the structure of the problem
- Let the solver determine the best way it knows how to handle this substructure
- In modelling these substructures are called

**global constraints**



## Alldifferent

- Global constraint version

```
alldifferent(task);
```

- Enforces that each worker is assigned a different task.
- Solvers can make use of the substructure to solve better.
- The first example of a global constraint
- Captures the
  - assignment substructure, or alternatively
  - deciding an injective function



## Include items

- Global constraints are defined in library files.
- We can include files into a MiniZinc model using
  - An inclusion item

    `include` <filename (which is a string literal)>;

- To use `alldifferent` we need to either

```
include "alldifferent.mzn";
```

- or

```
include "globals.mzn";
```

- which includes all globals

## Assignment Data and Decisions

- Data

```
int: n;
set of int: DOM = 1..n;
int: m;
set of int: COD = 1..m;
array[DOM,COD] of int: profit;
```

- Decisions

```
array[DOM] of var COD: task;
```

- Constraints

```
include "alldifferent.mzn";
alldifferent(task);
```

- Objective

```
maximize sum(w in DOM)
            (profit[w,task[w]]);
```



## Assignment Problem

- The pure assignment problem is very well studied
  - specialized polynomial (fast) algorithms
  - maximal weighted matching
- If you have a pure assignment problem
  - use a specialized algorithm
- BUT the real world is never pure
  - add some side constraints and these specialised algorithms almost always break!



## Overview

- Deciding a (finite) function in common
- Deciding an injective function is a
  - assignment (sub)problem
- The global constraint `alldifferent` captures this
- Global constraints
  - names of combinatorial substructures
  - solvers can use their best method for capturing this
  - plenty more to come …



## EOF

# Example Assignment Problem

Peter Stuckey

## CellBlock Question

- Assign `k` prisoners each to a different cell
- Cells are in a grid of size `n * m`
- No prisoner should be in a cell adjacent (north, south, east or west) to a dangerous prisoner (given by set `danger`)
- All female prisoners (given by set `female`) are in rows `1.. (n+1) div 2`
- All male prisoners (remaining prisoners) are in rows `n div 2 + 1 .. n`
- Each cell (i,j) has a `cost[i,j]` for occupation
- Minimize the cost of housing prisoners



## CellBlock Data

```
int: k;
set of int: PRISONER = 1..k;
int: n;
set of int: ROW = 1..n;
int: m;
set of int: COL = 1..m;
array[ROW,COL] of int: cost;
set of PRISONER: danger;
set of PRISONER: female;
set of PRISONER: male
     = PRISONER diff female;
```

dependent parameter declarations



## CellBlock Decisions

- What are the objects of the domain?
  - DOM = PRISONER
- What are the objects of the codomain?
  - COD = ROW x COL
- Representation: two functions

```
array[PRISONER] of var ROW: r;
array[PRISONER] of var COL: c;
```

## CellBlock Constraints

- No two prisoners in the same cell

```
forall(p1, p2 in PRISONER where p1 < p2)
 (abs(r[p1] - r[p2]) + abs(c[p1] - c[p2]) > 0);
```

- Cant we use alldifferent?
- Yes

```
alldifferent([r[p] * m + c[p] | p in PRISONER]);
```

- Mapping each cell block to a unique number
- Noone adjacent to dangerous prisoners

```
forall(p in PRISONER, d in DANGER where p != d)
 (abs(r[p] - r[d]) + abs(c[p] - c[d]) > 1);
```



## CellBlock Constraints + Objective

- Gender constraints

```
forall(p in female)(r[p] <= (n + 1) div 2);
forall(p in male)(r[p] >= n div 2 + 1);
```

- Note the use of `male`
  - clearer than replacing with definition
- Objective function

```
var int: totalcost = sum(p in PRISONER)
                       (cost[r[p],c[p]]);
solve minimize totalcost;
```



## Overview

- Assignment subproblems are common
- We will revisit assignment subproblems where the function is bijective later
  - matching problems



## EOF

# Modeling Partitions

Peter Stuckey

## Partitioning Problems

- Finding a function f: DOM ➔ COD
- Can be considered a partitioning problem
- This may give additional insight if
  - we want to constrain or manipulate the sets
  - $f^{-1}(c) = \{ d \text{ in DOM} \mid f(d) = c \}$



## Rostering Problem

- Given k nurses schedule them for each of the next m days as either day shift, night shift, or day off:
  - There are o nurses on day shift each day
  - There are between l and u nurses on each night shift
  - No nurse has more than two night shifts in a row
  - No nurse has a night shift followed by day shift



## Rostering Objects

- Defining the sets of objects we are reasoning about.
  - Note that we give names to types of shift, to make the model more readable

```
int: k; % number of nurses
set of int: NURSE = 1..k;
int: m; % number of days
set of int: DAY = 1..m;
set of int: SHIFT = 1..3;
int: day = 1; int: night = 2; int: dayoff = 3;
```

## Rostering Decisions

- What are the objects of the domain?
  - DOM = NURSE × DAY
- What are the objects of the codomain?
  - COD = SHIFT
- For each nurse and day choose the shift

```
array[NURSE,DAY] of var SHIFT: x;
```

- Can be considered a partitioning problem
  - partitions nurses by shift type
  - reasons about the sets of nurses on a shift



## Rostering Constraints

- Day Constraints
  - There are `o` nurses on day shift each day

```
forall(d in DAY)
      (sum(n in NURSE)
          (bool2int(x[n,d] = day)) = o);
```

  - There are between `l` and `u` nurses on each night shift

```
forall(d in DAY)
      (sum(n in NURSE)
          (bool2int(x[n,d] = night)) >= l);
forall(d in DAY)
      (sum(n in NURSE)
          (bool2int(x[n,d] = night)) <= u);
```

  - Common subexpressions!



## Intermediate Variables

- Intermediate variables
  - Store values of expressions that are reused
  - Are dependent on decisions
  - (note: intermediate parameters too!)

```
array[DAY] of var 0..k: onnight =
 [sum(n in NURSE)(bool2int(x[n,d] = night))
 | d in  DAY ];
constraint forall(d in DAY)
     (onnight[d] >= l /\ onnight[d] <= u);
```

- Choose bounds for intermediates well
- Or simply

```
array[DAY] of var l..u: onnight =
 [sum(n in NURSE)(bool2int(x[n,d] = night))
 | d in  DAY ];
```



## Nurse Constraints

- Constraints
  - No nurse has more than two night shifts in a row
- How do we express this?

```
forall(n in NURSE, fd in 1..m-2)
   (sum(d in fd..fd+2)(bool2int(x[n,d] = night))
    <= 2);
```

- Yikes not very clear
- Use logical connectives to be explicit

```
forall(n in NURSE, d in 1..m-2)
  (x[n,d] = night /\ x[n,d+1] = night
   -> x[n,d+2] != night);
```

  - No nurse has a night shift followed by day shift

```
forall(n in NURSE, d in 1..m-1)
  (x[n,d] = night -> x[n,d+1] != day);
```

## Logical Connectives

- Boolean expressions
  - `true`
  - `false`
  - `/\` conjunction
  - `\/` disjunction
  - `->` implication
  - `<->` bi-implication or equality
  - `not` negation
- Allow us to combine constraints in powerful ways
  - But beware combining logical constraints and global constraints! More later …



## Nurse Constraints

- Constraints
  - No nurse has more than two night shifts in a row
  - No nurse has a night shift followed by day shift
- After two night shifts what is possible?
  - only a dayoff

```
forall(n in NURSE, d in 1..m-2)
  (x[n,d] = night /\ x[n,d+1] = night
   -> x[n,d+2] = dayoff);
```

- Stronger constraint by combining information



## Partitioning Problems

- Many times when we are partitioning a set we have to partition it with bounds on the size of the partitions
  - e.g. `day = o, l ≤ night ≤ u`
- We have special constraints for partitioning with size bounds
  - `global_cardinality_low_up(x, v, lo, hi)`
  - Constrains $lo_i \leq \Sigma j \text{ in } 1..n \text{ bool2int}(x_j = v_i) \leq hi_i$
  - Bounds the count of the number of occurrences of $v_i$



## Global Cardinality

- Replace

```
forall(d in DAY)
      (sum(n in NURSE)
          (bool2int(x[n,d] = day)) = o);
forall(d in DAY)
      (sum(n in NURSE)
          (bool2int(x[n,d] = night)) >= l);
forall(d in DAY)
      (sum(n in NURSE)
          (bool2int(x[n,d] = night)) <= u);
```

- By

```
forall(d in DAY)
(global_cardinality_low_up([x[n,d]
                           | n in NURSE ],
 [ day, night ], [ o, l ], [o, u]));
```

## Global Cardinality Variants

- There are a number of variants of global cardinality
- Requiring that every value is counted

`global_cardinality_low_up_closed(x,v,l,u)`

  - Additionally forall i. exists j. $x_i = v_j$
- Collecting the counts

`global_cardinality(x, v, c)`

  - Constrains $c_i = \Sigma_{j \text{ in } 1..n}$ bool2int($x_j = v_i$)
- And collecting counts, and requiring every value is counted

`global_cardinality_closed(x, v, c)`



## Overview

- Many discrete optimization problems involve
  - deciding a function f: DOM ➔ COD
  - this can be seen as partitioning DOM
- The partition view is useful when we reason about the sets $f^{-1}(c)$
- Partitioning with cardinality constraints is a substructure captured by the
  - `global_cardinality` family
- Common subexpressions:
  - intermediate variables
- Logical connectives



## EOF



# Pure Partitioning

Peter Stuckey

## Pure Partitioning Problems

- Finding a function f: DOM ➔ COD
- Can be considered a partitioning problem

- But what if we simply want to partition DOM
  - COD is meaningless not given

- Easy case: bounded number of partitions k
  - COD = 1..k;
- Harder case: we dont know how many?



## Pure Partitioning

- Partition DOM into k parts
- Simple array representation

```
int[DOM] of var 1..k: x
```

- Beware multiple representations
- e.g. DOM = 1..4, k = 3
  - x = [1,2,1,3]; x = [3,1,3,2] ; x = [3,2,3,1]
  - all represent partition {1,3} {2} {4}



## Multiple representations

- Add constraints to leave only one representative for each partition

- Value symmetry
  - each of the values 1..k are symmetric
  - we dont care about the names of the groups

- This feature arises in many discrete optimization problems



## Removing value symmetry

- How to we enforce exactly one representation?
- Order the sets of the partition by their least element
  - e.g. {2}, {4}, {1,3} is ordered {1,3}, {2}, {4}
  - unique representation x = [1,2,1,3]
- How do we enforce this constraint
  - the least element in partition i is less than the least element in partition i+1

```
forall(i in 1..k-1)
 (  min([ j | j in DOM where x[j] = i]
  < min([ j | j in DOM where x[j] = i+1]));
```

## Cluster Problem

- Given a set of n points in divide them into at most k clusters so that
  - no two points in the same cluster are more than maxdiam away from each other
  - maximize the minimal distance between any two points in different clusters



## Cluster Problem Data

- Data defining the cluster instance.

```
int: n; % points to be clustered
set of int: POINT = 1..n;
array[POINT,POINT] of int: dist;
   % distance between two points
int: maxdist = max([ dist[i,j] | i,j in POINT]);

int: k; % number of clusters
set of int: CLUSTER = 1..k;

int: maxdiam;
```



## Cluster Problem

- Decisions

```
array[POINT] of var CLUSTER: x;
```

- Value symmetry

```
forall(i in 1..k-1)
   (  min([ j | j in POINT where x[j] = i])
    < min([ j | j in POINT where x[j] = i+1]));
```

- Constraints: max distance within cluster

```
forall(i,j in POINT where i < j /\ x[i] = x[j])
      (dist[i,j] <= maxdiam);
```

- Objective

```
int: obj = min( i,j in POINT where i < j )
     (if x[i]=x[j] then maxdist else dist[i,j]
      endif);
solve maximize obj;
```



## value_precede_chain

- MiniZinc includes a global constraint for removing value symmetry

```
value_precede_chain(array[int] of int: c,
                    array[int] of var int: x)
```

- Enforces that the first occurrence of c[i] in x is before the first occurrence of c[i+1] in x
- We can replace our symmetry eliminating constraint by

```
value_precede_chain([ i | i in 1..k ],x)
```

## Cluster Problem

- This formulation does not require exactly k clusters
  - there can be fewer
- We can easily add a constraint to enforce that each cluster has a member
  - lower bound = 1
  - upper bound = n - k + 1

```
global_cardinality_low_up_closed(x,
  [ i | i in 1..k], [ 1 | i in 1..k],
  [ n - k + 1 | i in 1..k ]);
```



## Pure Partitioning

- What about when we dont know how many clusters
- Simple: there can be no more partitions than n
  - int: k = n;



## Overview

- Pure partitioning with
  - bounded set of clusters k
  - known number of clusters k
  - unknown number of clusters
- Cluster numbers are irrelevant
  - induces a value symmetry
- Remove value symmetry with

`value_precede_chain`



## EOF