Modelling Constrained Optimization Problems

- Different approaches to modelling constrained optimization problems
- Basic modelling with MiniZinc
- Advanced modelling with MiniZinc
  - Predicates:
    - Global constraints
    - User defined constraints & tests
    - Reflection functions
  - Let expressions (local variables)
  - Negation and partial functions
  - Efficiency
    - Different problem models
    - Redundant constraints

Predicates

- MiniZinc allows us to capture complex constraint in a predicate. Predicate may be
  - Supported by the underlying solver, or
  - Defined by the modeller
- A predicate definition has the form
  - predicate <pred-name> (<arg-def> … <arg-def>) = <bool-exp>
- An argument definition is a MiniZinc type declaration e.g.
  - int:x, array[1..10] of var int:y, array[int] of bool:b
- Note arrays do not need to be fixed size

Global constraints: alldifferent

- all_different(array[int] of var int:x)
- Defines an assignment subproblem: all vars in the x array take a different value

```
include "all_different.mzn";
var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint        1000 * S + 100 * E + 10 * N + D
               +  1000 * M + 100 * O + 10 * R + E
  = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint all_different([S,E,N,D,M,O,R,Y]);

solve satisfy;
```

- To use a global we need to include it, or include all globals with include “globals.mzn”

Global Constraints: inverse

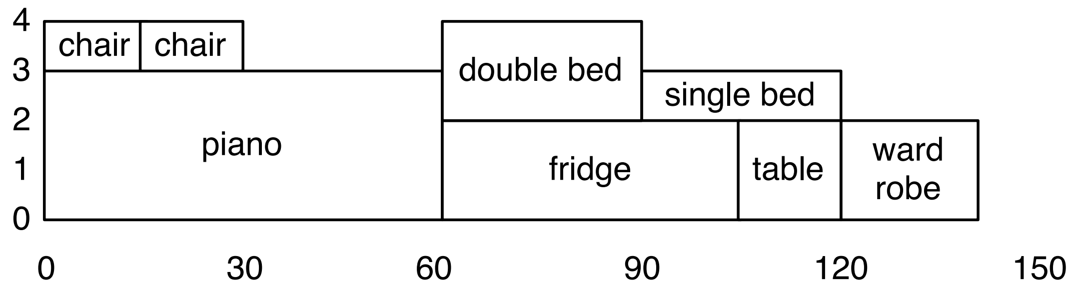
- inverse(array[int] of var int:f, array[int] of var int:if)
  - $f[i] = j \Leftrightarrow if[j] = i$ (*if* is the inverse function f^{-1})
- Helpful for assignment problems where we want both views of the problem
- array[1..n] of var 1..n: task;
  array[1..n] of var 1..n: worker;
  constraint inverse(task,worker);
  - We can express constraints about tasks and workers
    constraint t[1] > t[2] /\ w[3] < w[4];
  - task for worker 1 is numbered after task for worker 2
  - worker of task 3 is numbered less than worker for task 4

Global constraints: cumulative

- A constraint for cumulative resource usage
- cumulative(array[int] of var int: s, array[int] of var int: d, array[int] of var int: r, var int: b)
  - Set of tasks with start times s, and durations d and resource usages r never require more than b resources at any time
- % piano, fridge, double bed, single bed, wardrobe chair, chair, table

  d = [60, 45, 30, 30, 20, 15, 15, 15];

  r = [3, 2, 2, 1, 2, 1, 1, 2]; b = 4;



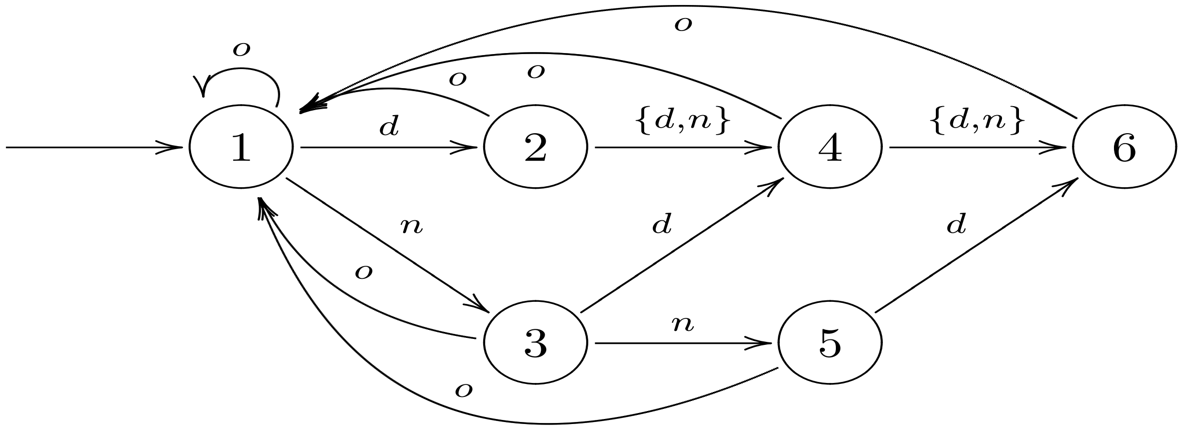
Global constraints: table

- Enforce that array of variables take value from one row in a table:
- table(array[int] of var int: x, array[int,int] of int:t)
- Consider a table of car models
- % doors, sunroof, speakers, satnav, aircon

```
models = [| 5, 0, 0, 0, 0  % budget hatch
          | 4, 1, 2, 0, 0  % standard saloon
          | 3, 1, 2, 0, 1  % standard coupe
          | 2, 1, 4, 1, 1  |]; % sports coupe
constraint table(options, models);
```

Global constraints: regular

- Enforce that sequences of variables form a regular expression, defined by a DFA
- regular(array[int] of var int:x, int:Q, int: S;
  array[int,int] of int:d, int q0: set of int:F)
- Sequence x (taking vals 1..S) accepted by DFA with Q states, start state q0, final states F, and transition function: d
- One day off every 4 days, no 3 nights

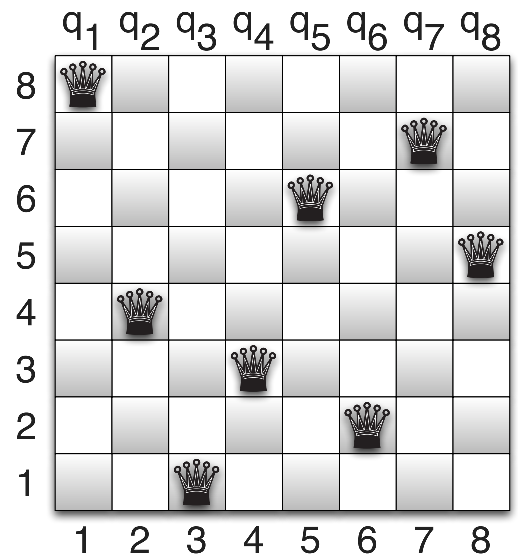


	d 1	n 2	o 3
1	2	3	1
2	4	4	1
3	4	5	1
4	6	6	1
5	6	0	1
6	0	0	1

User Defined Constraints

MiniZinc (unlike most other mathematical modelling languages) allows the modeller to define their own:

– predicates (var bool)
– tests (bool)



N-queens example:

```
int: n;
array [1..n] of var 1..n: q;

predicate
    noattack(int: i, int: j, var int: qi, var int: qj) =
        qi  != qj   /\   qi + i != qj + j  /\   qi - i != qj - j;

constraint
    forall (i in 1..n, j in i+1..n) (noattack(i, j, q[i], q[j]));

solve satisfy;
```

Complex Output (aside)

- Sometimes the output form is not close to the natural model: Queens
  - variables are columns, output by rows
- Solution: complex output expression
- output [if fix(q[j]) == i then "Q" else "." endif ++
  if j == n then "\n" else "" endif | i,j in 1..n];

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

Complex Output (aside)

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

- Sometimes the output form is not close to the natural model: Queens
  - variables are columns, output rows numbers
- Solution: complex output expression
- ```
  output [ if fix(q[j]) == i then show(j) else "" endif ++
           if j == n then "\n" else "" endif | i,j in 1..n];
  ```
- Output

```
1
7
5
8
2
4
6
3
```

- Alternate Solution: add to model
- ```
  array[1..n] of var 1..n: r; % row vars
  constraint inverse(q,r);
  output [ show(r[i]) ++ "\n" | i in 1..n];
  ```

Reflection Functions

- To help write generic tests and predicates, various reflection functions return information about array index sets, var set domains and decision variable ranges:
  - index_set(<1-D array>)
  - index_set_1of2(<2-D array>), index_set_2of2(<2-D array>)
  - ...
  - dom(<arith-dec-var>), lb(<arith-dec-var>), ub(<arith-dec-var>)
  - lb_array(<var-set>), ub_array(<var-set>)
- The latter class give "safe approximations" to the inferred domain, lowerbound and upperbound
  - Currently in mzn2fzn this is the declared or inferred bound

Extending assertions

- For predicates we introduce an extended assertion
  - assert(<bool-exp>, <string>, <bool-exp>)
- If first <bool-exp> evaluates to false prints <string> and aborts otherwise evaluates second <bool-exp>
- Useful to check user-defined predicate is called correctly

Using Reflection

- The disjunctive constraint:
  - cumulative where resource bound is 1 and all tasks require 1 resource.

  -
```
include "cumulative.mzn";
predicate disjunctive(array[int] of var int:s,
                              array[int] of int:d) =
      assert(index_set(s) == index_set(d),
          "disjunctive: first and second arguments “ ++
          “must have the same index set",
        cumulative(s, d, [ 1 | i in index_set(s) ], 1)
     );
```

All_different

- Write a predicate defining the all_different constraint that takes a 1-D array:

```
all_different(array[int] of var int:x)
```

Local Variables

- It is often useful to introduce local variables in a test or predicate
- The let expression allows you to do so

  let { <var-dec>, ... } in <exp>

  (It can also be used in other expressions)
- The var declaration can contain decision variables and parameters
  - Parameters must be initialized
- Example:

```
let {int: l = lb(x), int: u = ub(x) div 2, var l .. u: y} in
     x = 2*y
```

Exercise: Local Variables

```
var -2..2: x1;
var -2..2: x2;
var -2..2: x3;
var int: ll;
var int: uu;
constraint even(2 * x1 - x2 * x3);
predicate even(var int:x) =
     let { int: l = lb(x), int: u = ub(x) div 2, var l..u: y } in
      x = 2 * y /\ l = ll /\ u = uu;
output["l = ",show(ll), " u = ",show(uu), "\n"];
```

What prints out?

Complex use of local variables

```
predicate lex_less_int(array[int] of var int: x,
                       array[int] of var int: y) =
   let { int: lx = min(index_set(x)), int: ux = max(index_set(x)),
         int: ly = min(index_set(y)), int: uy = max(index_set(y)),
         int: size = min(ux - lx, uy - ly),
         array[0..size+1] of var bool: b }
   in
   b[0] /\
   forall(i in 0..size) (
      b[i] = ( x[lx + i] <= y[ly + i] /\
              ( x[lx + i] <  y[ly + i] \/ b[i+1]) )
   )
   /\
   b[size + 1] = (ux - lx < uy - ly);
```

X is lexicographically less than Y

Meaning of Negation

- Local variables cannot appear in a negated context
  - not <bool-exp>
  - <bool-exp> -> <bool-exp>
  - <bool-exp> = <bool-exp> (or <->)
- This is because they wont get the right meaning

```
predicate even(var int:x) = let {var int:y } in x = 2*y;
constraint not even(z);
```

- Translates to (more about translation later)

```
let { var int:y } in not (z = 2 * y)
```

- Solution: z = 2, y = 0 !
- Solvers don't support:
  - forall y. not (z = 2 * y)

Partial functions

- Given declarations

```
var 0..1: x;
var 0..5: i;
array[1..4] of int:a = [1,2,3,4];
```

- What are expected solutions for
  - `constraint 1 != 1 div x;`
  - `constraint not(1 == 1 div x);`
  - `constraint x < 1 \/ 1 div x != 1;`
  - `constraint a[i] >= 3;`
  - `constraint not(a[i] < 2);`
  - `constraint a[i] >= 2 -> a[i] <= 3;`

Relational semantics

- A partial function creates answer false
  - at the nearest enclosing Boolean context
- Examples

– 1 != 1 div 0	*false*
– not(1 == 1 div 0)	not(*false*) = *true*
– 0 < 1 \/ 1 div 0 != 1	*true* $\vee$ *false* = *true*
– a[0] >= 3	*false*
– not(a[0] < 2)	not(*false*) = *true*
– a[0] >= 2 -> a[0] <= 3	*false* -> *false* = *true*

Efficiency in MiniZinc

- Of course as well as correctly modelling our problem we also want our MiniZinc model to be solved efficiently
- Information about efficiency is obtained using the MiniZinc flags
  - solver-statistics [number of choice points]
  - statistics [number of choice points, memory and time usage]
- Extensive experimentation is required to determine relative efficiency

Improving Efficiency in MiniZinc

- Add search annotations to the solve item to control exploration of the search space.
- Use global constraints such as all_different since they have better propagation behaviour.
- Try different models for the problem.
- Add redundant constraints.

And for the expert user:

- Extend the constraint solver to provide a problem specific global constraint.
- Extend the constraint solver to provide a problem specific search routine.

Modelling Effectively

- Modelling is (like) programming
  - You can write efficient and inefficient models
- Take care to avoid some simple traps
  - Bound variables as tightly as possible
    - Avoid `var int` if possible
  - Avoid introducing unnecessary variables
  - Make loops as tight as possible

Bound your variables

```
var int: x;
var int: y;
constraint x <= y /\ x > y;
solve satisfy;
```

- Takes an awful long time to say no answer

```
var -1000..1000: x;
var -1000..1000: y;
```

- Is almost instant

Unconstrained variables

```
include "all_different.mzn";
array[1..15] of var bool: b;
array[1..4] of var 1..10: x;
constraint all_different(x) /\
           sum(i in 1..4)(x[i]) = 9;
solve satisfy;
```

- Takes a long time to say no
- Remove the bool array its instant!
- Sometimes unconstrained vars arise from matrix models where not all vars are used

Efficient loops

- Think about loops, just like in other programs

```
int: count = sum [1 | i, j, k in NODES where i<j
     /\ j<k /\ adj[i,j] /\ adj[i,k] /\ adj[j,k]];
```

- Compare this to

```
int: count = sum( i, j in NODES where
          i < j /\ adj[i,j])(
                 sum([1 | k in NODES where j < k
                        /\ adj[i,k] /\ adj[j,k]]));;
```

Global Constraints

- Where possible you should use global constraints
- MiniZinc provides a standard set of global constraints in the file globals.mzn
- To use these you simply include the file in the model

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

- **Exercise:** Rewrite N-queens to use all_different.
- **Exercise**: Look at globals.mzn

Different Problem Modellings

- Different views of the problem lead to different models
- Depending on solver capabilities one model may require less search to find answer
- Look for model with fewer variables
- Look for more direct mapping to primitive constraints.
- Empirical comparison may be needed

Different Problem Modellings

Simple assignment problem:

- n workers and
- n products

- Assign one worker to each product to maximize profit

Instance:

n=4 & profit matrix =

	$p1$	$p2$	$p3$	$p4$
$w1$	7	1	3	4
$w2$	8	2	5	1
$w3$	4	3	7	2
$w4$	3	1	6	3

Exercise: Model this in MiniZinc

MIP-style model

```
int: n;
array[1..n,1..n] of int: profit;
array[1..n,1..n] of var 0..1: assign;
constraint
    forall(w in 1..n) (
       sum(t in 1..n) (assign[t,w]) = 1 );
constraint
    forall(t in 1..n) (
       sum(w in 1..n) (assign[t,w]) = 1 );
solve maximize
   sum( w in 1..n, t in 1..n) (
      assign[t,w]*profit[t,w]);
```

Assign task to worker

```
include “globals.mzn”;
int: n;
array[1..n,1..n] of int: profit;
array[1..n] of var 1..n: task;

constraint all_different(task);
solve maximize
   sum(w in 1..n) (
      profit[w,task[w]]);
```

Assign worker to task

```
include “globals.mzn”;
int: n;
array[1..n,1..n] of int: profit;
array[1..n] of var 1..n: worker;

constraint all_different(worker);
solve maximize
   sum(t in 1..n) (
      profit[worker[t],t]);
```

Redundant Constraints

- Sometimes solving behaviour can be improved by adding redundant constraints to the model
- The magic series model will run faster with redundant constraints:

```
int: n;
array[0..n-1] of var 0..n: s;

constraint
  forall(i in 0..n-1) (
     s[i] = sum(i in 0..n-1)(bool2int(s[j]=i)));
constraint
   sum(i in 0..n-1) (s[i]) = n;
constraint
   sum(i in 0..n-1) (s[i]*i) = n;
solve satisfy;
```

Redundant Constraints

- An extreme kind of redundancy is to combine different models for a problem using channeling constraints.

```
int: n;
array[1..n,1..n] of int: profit;
array[1..n] of var 1..n: task;
array[1..n] of var 1..n: worker;
constraint all_different(task);
constraint all_different(worker);
constraint
   forall( w in 1..n) (w = worker[task[w]]);
constraint
   forall( t in 1..n) (t = task[worker[t]]);
solve maximize
   sum(t in 1..n) (
      profit[worker[t],t]);
```

Redundant Constraints

- There are globals for channeling constraints.
  - inverse(x,y): $x[i] = j <-> y[j] = i$
- A better combined model

```
int: n;
Include "inverse.mzn";
array[1..n,1..n] of int: profit;
array[1..n] of var 1..n: task;
array[1..n] of var 1..n: worker;
% constraint all_different(task);  % redundant
% constraint all_different(worker);
constraint inverse(task,worker);
solve maximize sum(t in 1..n) (profit[worker[t],t]);
```

Labelling

- Recall that in CP the construction of the search tree can have a huge effect on efficiency
- The search strategy is often called labelling.
- There are two choices made in labelling
  - which variable to label
  - which value in the domain to try
- Default labelling
  - try variables in order of the given list
  - try value in order min to max (returned by dom)
- However we can use different strategies. These can lead to dramatic performance improvement.

First Fail Labelling

- One useful heuristic is the **first-fail principle**

  "To succeed, try first where you are most likely to fail"

- At each step choose the variable with the smallest domain.
- Do this dynamically based on the domain size after propagation.

Annotations

- MiniZinc allows the user to annotate their model with information for the underlying solver to guide how it solves it
- Such annotations do not change the model's meaning but can greatly affect efficiency
- Example annotations are
  - <var-decl> :: is_output

    means that this is an output variable.

    Note that an output item overrides is_output annotations
  - <constraint> :: bounds
  - <constraint> :: domain

    specifies the type of propagation to use with the constraint

Search Annotations

- MiniZinc allows control of search using annotations on the solve item
- For integer variables these have form

  int_search(vars, var_select, choice, explore)

- vars is a 1D array specifying the var int variables affected by the annotation;
- var_select is the variable selection strategy
  - input_order, first_fail, anti_first_fail, smallest, largest, occurrence, most_constrained, max_regret
- choice is the value choice strategy
  - indomain, indomain_min, indomain_max, indomain_split, …
- explore is the search strategy
  - complete, bbs(s), fail, …

See the FlatZinc specification for more details

http://www.g12.csse.unimelb.edu.au/minizinc/downloads/doc-0.10/flatzinc-spec.pdf

Search Annotations

- For the N-queens model you might use

```
solve ::
      int_search(
            q,
            first_fail,
            indomain_min,
            complete
      )
      satisfy;
```

Extending the Constraint Solver

- MiniZinc can be executed using ECLiPSe, Mercury G12 solving platform, or Gecode.
- These allow new global constraints to be added to the solver
- They also allow new search strategies to be added
  - we'll talk about search strategies later

Summary

- Advanced models in MiniZinc use predicates to define complex subproblem constraints
  - Global constraints (give better solving)
  - User defined constraints & tests (Give better readability)
- We need to be careful with negation and local variables
- Efficiency depends on the model formulation
- Developing an efficient decision model requires considerable experimentation

Zinc

- However MiniZinc is not a very powerful modelling language.
- MiniZinc is a subset of Zinc.
- Zinc extends MiniZinc providing
  - Tuples, enumerated constants, records, discriminated union
  - Var sets over arbitrary finite types
  - Arrays can have arbitrary index sets.
  - Overloaded functions and predicates.
  - Constrained types
  - User defined functions.
  - More powerful search parameterized by functions.
- Coming soon…

Exercise 1: Predicates

- Write a predicate definition for
  - near_or_far(var int:x, var int:y, int:d1, int:d2)

    which holds if difference in the value of x and y is either at most d1 or at least d2.
  - Can you optimize its definition for simple cases?
- Write a predicate definition for
  - sum_near_or_far(array[int] of var int:x, int: d1, int:d2)

    which holds if the sum of the x array is at most d1 or at least d2

Exercise 2: Comparing Models

- Try out the different versions of the assignment problem on the problems from examples.pdf (add an extra worker G to the unbalanced example with costs all 30)
  - Compare the number of choices required to solve using
    mzn –statistics
  - Try all five models, which is best?
  - Try different solvers?