Modéliser des problèmes plus complexes d'optimisation sous contraintes

- Différentes approches pour la modélisation
- Modélisation simple en MiniZinc
- Modélisation avancée en MiniZinc
  - Predicats:
    - Contraintes globales
    - Contraintes définies par l'utilisateur et tests
    - Fonctions complexes

  - Gestion de variables locales
  - Négation et fonctions partielles
  - Efficacité
    - Differents modéles possibles
    - Contraintes redondantes

Predicats

- MiniZinc permet de prendre en charge des contraintes complexes via des prédicats qui peuvent être
  - Traités par le solveur ou
  - Définis par l'utilisateur
- Une définition de prédicat est de la forme
  - predicate <pred-name> (<arg-def> … <arg-def>) = <bool-exp>
- Un argument de définition est une déclaration de type MiniZinc
  - int:x, array[1..10] of var int:y, array[int] of bool:b
- Note les tableaux ne sont pas forcément de taille fixe

Contraintes Globales: alldifferent

- alldifferent(array[int] of var int:x)
- Défini un sous problème d’affectation : toutes les vars de x doivent avoir des valeurs différentes

```
include "alldifferent.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 alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;
```

- On doit insérer le include , ou inclure toutes les globales avec include “globals.mzn”

Contraintes Globales: 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})
- Utile pour des problèmes d'affectation qui nécessitent les deux visions
- Exemple avec l'affectation de tâches à des ouvriers.
- array[1..n] of var 1..n: task;

  array[1..n] of var 1..n: worker;

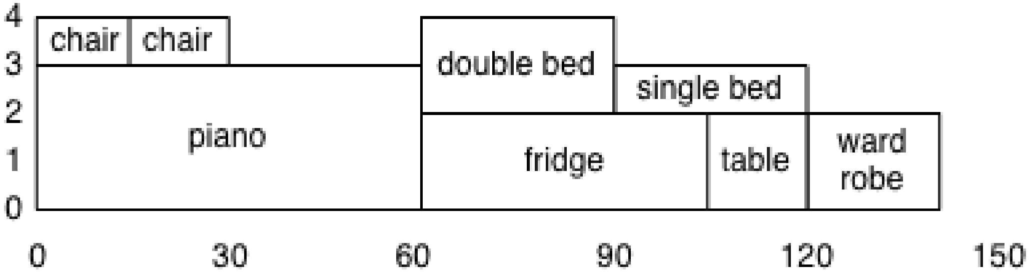
  constraint inverse(task,worker);

Contraintes Globales : cumulative

- Pour l'utilisation cumulative de ressources
- cumulative(array[int] of var int: s, array[int] of var int: d, array[int] of var int: r, var int: b)
  - Un ensemble de tâches avec une date de début s, une durée d et une utilisation de ressource r. On dispose d'au plus b ressources
  - % Déménagement 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;



Contraintes Globales : table

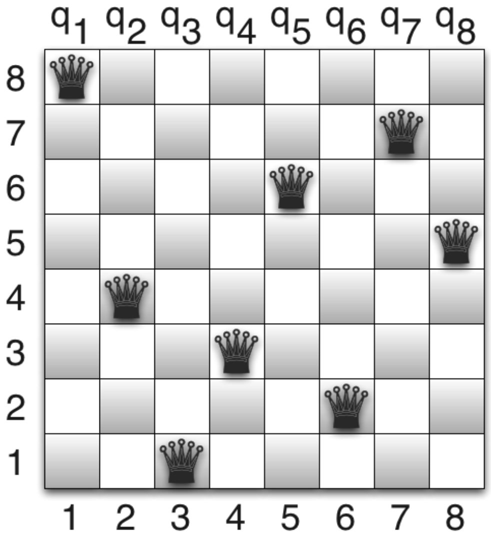
- Contraint un tableau a être une ligne d’une table :
- table(array[int] of var int: x, array[int,int] of int:t)
- Options pour des voitures (car sequencing)
- % doors, sunroof, speakers, satnav, aircon

```
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);
```

Constraintes définies par l’utilisateur

MiniZinc (contrairement à beaucoup de langages de modélisation) autorise l’utilisateur à définir ses propres contraintes :

- predicats (var bool)
- tests (bool)



N-reines :

```
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;
```

Formats de sortie complexes

- Parfois l’affichage est éloigné du problème: Reines
  - Les variables sont les colonnes, sortie par ligne

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

Formats de sortie complexes

- Solution: format plus complexe via une expression d’affichage
-
  ```
  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.....
```

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 alldifferent constraint that takes a 1-D array:

```
alldifferent(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

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* ∨ *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 alldifferent 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 alldifferent(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 =

$$\begin{array}{c|cccc} & p1 & p2 & p3 & p4 \\ \hline w1 & 7 & 1 & 3 & 4 \\ w2 & 8 & 2 & 5 & 1 \\ w3 & 4 & 3 & 7 & 2 \\ w4 & 3 & 1 & 6 & 3 \end{array}$$

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 alldifferent(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 alldifferent(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 alldifferent(task);
constraint alldifferent(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 alldifferent(task);  % redundant
% constraint alldifferent(worker);
constraint inverse(task,worker);
solve maximize sum(t in 1..n) (profit[worker[t],t]);
```

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?