Topic 4: Modelling (for CP & LCG)
(Version of 17th September 2018)

Pierre Flener and Gustav Björdal

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL441:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

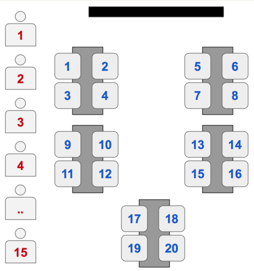
Outline

Outline

Recap

1. Modelling: express problem in terms of
   - parameters,
   - decision variables,
   - constraints, and
   - objective.
2. Solving: solve using a state-of-the-art solver.

Example (Student Seating Problem)



s = 15 students
c = 20 chairs

Given:

- s students, and
- c chairs positioned around tables.

Find a seating arrangement such that:

- Each table has either at least half its chairs occupied, or none.
- Each table has at least as many students as any table behind it.
- A maximum number of student preferences on being seated at the same table are satisfied.

What are suitable decision variables for this problem?

UPPSALA UNIVERSITET

Viewpoints

Implied Constraints

Redundant Variables & Channelling Constraints

Pre-Computation

A viewpoint is a choice of decision variables.

Example (Student Seating Problem)

Viewpoint 1:
For each student, which chair is the student assigned to?

```
% Chair[i] = the chair of student i:
array[1..s] of var 1..c: Chair;
constraint alldifferent(Chair);
```

Viewpoint 2:
For each chair, which student, if any, is seated on it?

```
% Student[i] = the student on chair i:
array[1..c] of var 0..s: Student; % dummy 0
constraint alldifferent_except_0(Student);
```

Let us now look at a generic problem in order to see how viewpoints differ when we start formulating constraints.

Example (Objects, Shapes, and Colours)

There are $\mathtt{n}$ objects, $\mathtt{s}$ shapes, and $\mathtt{c}$ colours, with $\mathtt{s} \geq \mathtt{n}$. Assign a shape and a colour to each object such that:

1. the objects have distinct shapes;
2. the numbers of objects of the used colours are distinct;
3. other constraints, yielding NP-hardness and distinguishing objects and shapes, are satisfied.

This problem can be modelled from different viewpoints:

1. Which colour, if any, does each shape have?
2. Which shapes, if any, does each colour have?
3. Which shape and colour does each object have?
4. ...

Each viewpoint comes with benefits and drawbacks.

Example (Objects, Shapes, and Colours)

Viewpoint 1: Which colour, if any, does each shape have?

```
1  int: n; % number of objects
2  int: s; % number of shapes
3  int: c; % number of colours
4  constraint assert(s >= n, "Not enough shapes");
5  % Colour[i] = the colour of the object of shape i:
6  array[1..s] of var 0..c: Colour; % 0 is a dummy colour
7  % There are n objects:
8  constraint exactly(s-n,Colour,0);
9  % The numbers of objects of the used colours are distinct:
10 constraint
     alldifferent_except_0(global_cardinality(Colour,1..c));
11 % The objects have distinct shapes:
12 %   implied by lines 6 and 8!
13 % ... add here the other constraints ...
14 solve satisfy;
```

Colour `0` is used when there is no object of the given shape.
So what are the shape and colour of a particular object?!
☞ Map the objects onto the shapes with a non-`0` colour!

Example (Objects, Shapes, and Colours)

Viewpoint 2: Which shapes, if any, does each colour have?

```
int: n; % number of objects
int: s; % number of shapes
int: c; % number of colours
constraint assert(s >= n, "Not enough shapes");
% Shapes[i] = the set of shapes of colour i:
array[1..c] of var set of 1..s: Shapes;
% There are n objects:
constraint n = sum(colour in 1..c)(card(Shapes[colour]));
% The numbers of objects of the used colours are distinct:
constraint alldifferent_except_0(colour in 1..c)
  (card(Shapes[colour]));
% The objects have distinct shapes:
constraint n = card(array_union(Shapes));
% ... add here the other constraints ...
solve satisfy;
```

Post-process: map the objects onto actually used shapes.

Can we also model this viewpoint without set variables?

☞ Yes, see the next slide!

Example (Objects, Shapes, and Colours)

Viewpoint 2: Which shapes, if any, does each colour have?

```
int: n; % number of objects
int: s; % number of shapes
int: c; % number of colours
constraint assert(s >= n, "Not enough shapes");
% NbrObj[i,j] = the number of objects of colour i & shape j:
array[1..c,1..s] of var 0..1: NbrObj;
% There are n objects:
constraint n = sum(NbrObj);
% The numbers of objects of the used colours are distinct:
constraint alldifferent_except_0(colour in 1..c)
   (sum(NbrObj[colour,..]));
% The objects have distinct shapes:
constraint forall(shape in 1..s)(sum(NbrObj[..,shape])<=1);
% ... add here the other constraints ...
solve satisfy;
```

Which model for viewpoint 2 is clearer or better?

☞ Ask and try!

Example (Objects, Shapes, and Colours)

Viewpoint 3: Which shape & colour does each object have?

```
1  int: n; % number of objects
2  int: s; % number of shapes
3  int: c; % number of colours
4  constraint assert(s >= n, "Not enough shapes");
5  array[1..n] of var 1..s: Shape; % Shape[i] = shape of obj. i
6  array[1..n] of var 1..c: Colour; % Colour[i] = colour of i
7  % There are n objects:
8  %   implied by lines 5 and 6!
9  % The numbers of objects of the used colours are distinct:
10 constraint alldifferent_except_0
     (global_cardinality_closed(Colour,1..c));
11 % The objects have distinct shapes:
12 constraint alldifferent(Shape);
13 % ... add here the other constraints ...
14 solve satisfy;
```

We have used two parallel arrays with the same index set but different domains in order to represent pair variables.

Which viewpoint is better in terms of:

- Size of the search space
- Ease of formulating the constraints and the objective
- Performance
- Readability

Which viewpoint is better in terms of:

- Size of the search space:
  - Viewpoint 1: $\mathcal{O}((\mathtt{c}+1)^s)$, which is independent of $\mathtt{n}$
  - Viewpoint 2: $\mathcal{O}(2^{\mathtt{s}\cdot\mathtt{c}})$, which is independent of $\mathtt{n}$
  - Viewpoint 3: $\mathcal{O}(\mathtt{s}^{\mathtt{n}} \cdot \mathtt{c}^{\mathtt{n}})$
- Ease of formulating the constraints and the objective
- Performance
- Readability

Viewpoints

Implied Constraints

Redundant Variables & Channelling Constraints

Pre-Computation

Which viewpoint is better in terms of:

- Size of the search space:
  - Viewpoint 1: $\mathcal{O}((\mathtt{c}+1)^s)$, which is independent of $\mathtt{n}$
  - Viewpoint 2: $\mathcal{O}(2^{\mathtt{s}\cdot\mathtt{c}})$, which is independent of $\mathtt{n}$
  - Viewpoint 3: $\mathcal{O}(\mathtt{s}^\mathtt{n}\cdot\mathtt{c}^\mathtt{n})$

  Does this actually matter?
- Ease of formulating the constraints and the objective
- Performance
- Readability

Which viewpoint is better in terms of:

- Size of the search space:
  - Viewpoint 1: $\mathcal{O}((\mathtt{c}+1)^{s})$, which is independent of $\mathtt{n}$
  - Viewpoint 2: $\mathcal{O}(2^{\mathtt{s}\cdot\mathtt{c}})$, which is independent of $\mathtt{n}$
  - Viewpoint 3: $\mathcal{O}(\mathtt{s}^{\mathtt{n}}\cdot\mathtt{c}^{\mathtt{n}})$

  Does this actually matter?
- Ease of formulating the constraints and the objective:
  - It depends on the unstated other constraints.
  - Ideally, we want a viewpoint that allows global-constraint predicates to be used.
- Performance
- Readability

Which viewpoint is better in terms of:

- Size of the search space:
  - Viewpoint 1: $\mathcal{O}((\mathtt{c}+1)^s)$, which is independent of $\mathtt{n}$
  - Viewpoint 2: $\mathcal{O}(2^{\mathtt{s}\cdot\mathtt{c}})$, which is independent of $\mathtt{n}$
  - Viewpoint 3: $\mathcal{O}(\mathtt{s}^{\mathtt{n}}\cdot\mathtt{c}^{\mathtt{n}})$

  Does this actually matter?
- Ease of formulating the constraints and the objective:
  - It depends on the unstated other constraints.
  - Ideally, we want a viewpoint that allows global-constraint predicates to be used.
- Performance:
  - Hard to tell: we have to run experiments!
- Readability

UPPSALA UNIVERSITET

Viewpoints

Implied Constraints

Redundant Variables & Channelling Constraints

Pre-Computation

Which viewpoint is better in terms of:

- Size of the search space:
  - Viewpoint 1: $\mathcal{O}((\mathtt{c}+1)^{\mathtt{s}})$, which is independent of $\mathtt{n}$
  - Viewpoint 2: $\mathcal{O}(2^{\mathtt{s}\cdot\mathtt{c}})$, which is independent of $\mathtt{n}$
  - Viewpoint 3: $\mathcal{O}(\mathtt{s}^{\mathtt{n}}\cdot\mathtt{c}^{\mathtt{n}})$

  Does this actually matter?
- Ease of formulating the constraints and the objective:
  - It depends on the unstated other constraints.
  - Ideally, we want a viewpoint that allows global-constraint predicates to be used.
- Performance:
  - Hard to tell: we have to run experiments!
- Readability:
  - Who is going to read the model?
  - What is their background?

Which viewpoint is better in terms of:

- Size of the search space:
  - Viewpoint 1: $\mathcal{O}((\mathtt{c}+1)^s)$, which is independent of $\mathtt{n}$
  - Viewpoint 2: $\mathcal{O}(2^{\mathtt{s}\cdot\mathtt{c}})$, which is independent of $\mathtt{n}$
  - Viewpoint 3: $\mathcal{O}(\mathtt{s}^{\mathtt{n}}\cdot\mathtt{c}^{\mathtt{n}})$

  Does this actually matter?
- Ease of formulating the constraints and the objective:
  - It depends on the unstated other constraints.
  - Ideally, we want a viewpoint that allows global-constraint predicates to be used.
- Performance:
  - Hard to tell: we have to run experiments!
- Readability:
  - Who is going to read the model?
  - What is their background?

There are no correct answers here:
we actually need to think about this and run experiments.

Outline

Example (The Magic Series Problem)

The element at index `i` in `I = 0..(n-1)` is the number of occurrences of `i`. Solution: `Magic = [1,2,1,0]` for `n=4`.

Variables: `Magic` =

0	1	$\cdots$	n−1
$\in 0..n$	$\in 0..n$	$\cdots$	$\in 0..n$

Constraint:

```
forall(i in I)(Magic[i] = sum(j in I)(bool2int(Magic[j]=i)))
```

or, logically equivalently but better:

```
forall(i in I)(count(Magic,i,Magic[i]))
```

or, logically equivalently and even better:

```
global_cardinality_closed(Magic,I,Magic)
```

Implied Constraint:

```
sum(Magic)=n /\ sum(i in I)(Magic[i]*i)=n
```

For `n=80`, using a CP solver: only 7 search nodes are explored instead of 302; the solving is 1,000 times faster.

Definition

An implied constraint, also called a redundant constraint, is a constraint that logically follows from other constraints.

Benefit:

Solving may be faster, without losing any solutions. However, not all implied constraints accelerate the solving.

Good practice in MiniZinc:

Flag implied constraints using the `implied_constraint` predicate. This allows backends to handle them differently, if wanted (see Topic 9: Modelling for CBLS):

```
predicate implied_constraint(var bool: c) = c;
```
vs
```
predicate implied_constraint(var bool: c) = true;
```

Example

```
constraint implied_constraint(sum(Magic) = n);
```

In Topic 5: Symmetry, we will see the equally recommended `symmetry_breaking_constraint` predicate.

Outline

Redundant Decision Variables

Example (`n`-queens)

Use both the n^2 decision variables `Queen[i,j]` in `0..1` and the `n` decision variables `Row[q]` in `1..n`.

Definition

A redundant decision variable is a decision variable that represents information that is already represented by some other decision variables. It reflects a different viewpoint.

Benefit: Easier modelling of some constraints, or faster solving, or both.

Examples (see Topic 6: Case Studies)

- Model of Black-Hole Patience
- Models 1 & 3 of Warehouse Location Problem

Channelling Constraints

Example (n-queens)

Channelling between the `n` decision variables `Row[i]` in `1..n` and the n^2 decision variables `Queen[i,j]` in `0..1`:

```
forall(i in 1..n)(Row[i] = sum(j in 1..n)(j * Queen[i,j]))
```

Definition

A channelling constraint establishes the coherence of the values of mutually redundant decision variables.

Examples (see Topic 6: Case Studies)

- Model of Black-Hole Patience
- Models 1 & 3 of Warehouse Location Problem
- Experiment with channelling between the viewpoints for the *Objects, Shapes, and Colours* problem (slide 7).

Outline

Example (Prize-Pool Division)

Consider a maximisation problem where the objective function is the division of an unknown prize pool by an unknown number of winners:

```
...
array[1..5] of int: Pools = [1000,5000,15000,20000,25000];
var 1..5: x; % index of the actual prize pool within Pools
var 1..500: nbrWinners; % the number of winners
...
solve maximize Pools[x] div nbrWinners; % implicit: element!
```

Observation: We should avoid using the `div` function on decision variables, because:

- It yields weak inference, at least in CP & LCG solvers.
- Its inference takes unnecessary time and memory.
- It is not supported by all MiniZinc backends.

Idea: We can pre-compute all possible objective values.

Idea: We can pre-compute all possible objective values.

Example (Prize-Pool Division, revisited)

Pre-compute a 2d array, indexed by `1..5` and `1..500`, for each possible value pair of `x` and `nbrWinners`:

```
...
array[1..5] of int: Pools = [1000,5000,15000,20000,25000];
var 1..5: x; % index of the actual prize pool within Pools
var 1..500: nbrWinners; % the number of winners
...
array[1..5,1..500] of int: objVal = array2d(1..5,1..500,
  [Pools[p] div n | p in 1..5, n in 1..500]);
solve maximize objVal[x,nbrWinners]; % implicit: 2d-element!
```