Topic 4: Modelling (for CP and LCG)¹
(Version of 23rd October 2023)

Pierre Flener and Gustav Björdal

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL442:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

¹Many thanks to Guido Tack for feedback
Outline

1. Viewpoints & Dummy Values

2. Implied Constraints

3. Redundant Variables & Channelling Constraints

4. Pre-Computation
Outline

1. Viewpoints & Dummy Values

2. Implied Constraints

3. Redundant Variables & Channelling Constraints

4. Pre-Computation
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)

Given:

- **nStudents** students,
- **nPgms** study programmes
- **nChairs** chairs around **nTables** tables, and
- **Chairs[t]** as the set of chairs of table **t**, find a seating arrangement such that:

1. each table has students of distinct study programmes;
2. each table has either at least half or none of its chairs occupied;
3. a maximum number of student preferences on being seated at the same table are satisfied.

What are suitable decision variables for this problem?
A viewpoint is a choice of decision variables.

**Example (Student Seating Problem)**

1. **Viewpoint 1:** Which chair does each student sit on?

   ```
   % Chair[s] = the chair of student s:
   array[1..nStudents] of var 1..nChairs: Chair;
   constraint all_different(Chair); % max 1 student per chair
   ```

2. **Viewpoint 2:** Which student, if any, sits on each chair?

   ```
   int: dummyS = 0; % Advice: also experiment with nStudents+1
   set of int: StudentsAndDummy = 1..nStudents union {dummyS};
   % Student[c] = the student, possibly dummy, sitting on chair c:
   array[1..nChairs] of var StudentsAndDummy: Student;
   constraint global_cardinality_closed(Student, [dummyS]+[i| i in 1..nStudents],
   [nChairs - nStudents] ++ [1 | i in 1..nStudents]);
   %all_different(Student) if nStudents+1..nChairs are dummy students
   ```

We revisit this problem at slide 19 and the choice of dummy values in Topic 5: Symmetry, as well as in Topic 8: Inference & Search in CP & LCG.

Let us see how viewpoints differ when stating constraints.
Example (Objects, Shapes, and Colours)

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

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

```plaintext
1 int: n; % number of objects
2 int: s; % number of shapes
3 constraint assert(s >= n, "Not enough shapes");
4 int: c; % number of colours
5 int: dummyColour = 0; % Advice: also experiment with c+1
6 set of int: ColoursAndDummy = 1..c union {dummyColour};
7 % Colour[i] = the colour, possibly dummy, of the object of shape i:
8 array[1..s] of var ColoursAndDummy: Colour;
9 % There are n objects:
10 constraint count(Colour,dummyColour) = s - n;
11 % The numbers of objects of the actually used colours are distinct:
12 constraint all_different_except(global_cardinality(Colour,1..c),{0});
13 % The objects have distinct shapes:
14 % implied by lines 6 and 8!
15 % ... state here the other constraints ...
16 solve satisfy;
```

So what are the shape and colour of a particular object?!
☞ Map the objects onto the shapes with non-dummy colour!
Example (Objects, Shapes, and Colours)

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

1. `int: n; % number of objects`
2. `int: s; % number of shapes`
3. `constraint assert(s >= n, "Not enough shapes");`
4. `int: c; % number of colours`
5. `%`
6. `%`
7. `% Shapes[i] = the set of shapes of colour i:`
8. `array[1..c] of var set of 1..s: Shapes;`
9. `% There are n objects:`
10. `% implied by line 14 below!`
11. `% The numbers of objects of the actually used colours are distinct:`
12. `constraint all_different_except([card(Shapes[colour]) | colour in 1..c],{0});`
13. `% The objects have distinct shapes:`
14. `constraint n = card(array_union(Shapes));`
15. `% ... state here the other constraints ...`
16. `solve satisfy;`

Post-process: map the objects onto actually used shapes.
Can we also model this viewpoint without set variables? ☞ Yes, see next slide!
Example (Objects, Shapes, and Colours)

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

1 int: n; % number of objects
2 int: s; % number of shapes
3 constraint assert(s >= n, "Not enough shapes");
4 int: c; % number of colours
5 
6 %
7 % NbrObj[i,j] = the number of objects of colour i and shape j:
8 array[1..c,1..s] of var 0..1: NbrObj;
9 % There are n objects:
10 constraint n = sum(NbrObj);
11 % The numbers of objects of the actually used colours are distinct:
12 constraint all_different_except([sum(NbrObj[colour,..]) | colour in 1..c],{0});
13 % The objects have distinct shapes:
14 constraint forall(shape in 1..s) (sum(NbrObj[..,shape]) <=1 );
15 % ... state here the other constraints ...
16 solve satisfy;

Which model for viewpoint 2 is clearer or better? ☞ Ask others and try!
Example (Objects, Shapes, and Colours)

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

```plaintext
1 int: n; % number of objects
2 int: s; % number of shapes
3 constraint assert(s >= n, "Not enough shapes");
4 int: c; % number of colours
5 % Shape[i] = the shape of object i:
6 array[1..n] of var 1..s: Shape;
7 % Colour[i] = the colour of object i:
8 array[1..n] of var 1..c: Colour;
9 % There are n objects:
10 % implied by lines 6 and 8!
11 % The numbers of objects of the actually used colours are distinct:
12 constraint all_different_except(global_cardinality_closed(Colour,1..c),{0});
13 % The objects have distinct shapes:
14 constraint all_different(Shape);
15 % ... state here the other constraints ...
16 solve satisfy;
```

We needed to use two parallel arrays in lines 6 and 8 with the same index set but different domains in order to mimic records of two decision variables.
Which viewpoint is better in terms of:

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

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 constraints 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

1. Viewpoints & Dummy Values
2. Implied Constraints
3. Redundant Variables & Channelling Constraints
4. Pre-Computation
Example (Magic Series of length n: model)

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

Decision variables: \( \text{Magic} = \begin{array}{cccc} 0 & 1 & \cdots & n-1 \\ \in I & \in I & \cdots & \in I \end{array} \)

Problem Constraint:

\[
\text{forall}(i \text{ in } I)(\text{Magic}[i] = \text{sum}(j \text{ in } I)(\text{Magic}[j] = i))
\]
or, logically equivalently but better:

\[
\text{forall}(i \text{ in } I)(\text{Magic}[i] = \text{count} (\text{Magic},i))
\]
or, logically equivalently and even better:

\[
\text{global_cardinality_closed} (\text{Magic, array1d}(I, [i \mid i \text{ in } I]), \text{Magic})
\]

Implied Constraints:

\[
\text{sum} (\text{Magic}) = n \\land \text{sum}(i \text{ in } I)(i \times \text{Magic}[i]) = n
\]

Depending on the formulation above of the problem constraint, the implied constraints accelerate a CP solver up to 100 times for \( n=150 \).
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 `implied_constraint`. This allows backends to handle them differently, if wanted (see Topic 9: Modelling for CBLS):

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

Example

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

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

1. Viewpoints & Dummy Values

2. Implied Constraints

3. Redundant Variables & Channelling Constraints

4. Pre-Computation
Example (\(n\)-queens)

Use both the \(n^2\) decision variables \(\text{Queen}[r,c] \text{ in } 0..1\) and the \(n\) decision variables \(\text{Row}[c] \text{ in } 1..n\).

Definition

A redundant decision variable denotes information already denoted by other variables: mutual redundancy (same information) vs non-mutual redundancy.

Benefit: Easier modelling, or faster solving, or both. Careful, the terminology differs: derived parameters vs redundant variables.

Examples (see Topic 6: Case Studies)

- Each \(\text{Queen}[..,c]\) slice is mutually redundant with the variable \(\text{Row}[c]\).
- Best model of Black-Hole Patience: mutual redundancy.
- Models 1 and 3 of Warehouse Location: non-mutual redundancy.
- Sport Scheduling: mutual redundancy.
Example (n-queens)

One-way channelling from each decision variable Row[c] to one of its mutually redundant decision variables of the slice Queen[..,c]:

\[
\text{constraint } \forall (c \in 1..n) (\text{Queen}[\text{Row}[c],c] = 1);
\]

What sets the other decision variables of the slice Queen[..,c]?

Definition

A channelling constraint fixes the value of either some (1-way channelling) or all (2-way channelling) decision variables when the values of the decision variables they are redundant with are fixed.

This applies to both sets of decision variables.

Examples (see Topic 6: Case Studies)

- Best model of Black-Hole Patience: 2-way channelling.
- Models 1 and 3 of Warehouse Location: 1-way channelling.
- Sport Scheduling: 2-way channelling.
Example (Student Seating, viewpoint 2 revisited)

1. `int: dummyS = 0; % Advice: also experiment with nStudents+1`
2. `set of int: StudentsAndDummy = 1..nStudents union {dummyS};`
3. `% Student[c] = the student, possibly dummy, sitting on chair c:`
4. `array[1..nChairs] of var StudentsAndDummy: Student;`
5. `constraint global_cardinality_closed(Student, [dummyS]++[i|i in 1..nStudents], [nChairs - nStudents] ++ [1 | i in 1..nStudents]);`
6. `int: dummyP = 0; % Advice: also experiment with nPgms+1`
7. `set of int: PgmsAndDummy = 1..nPgms union {dummyP};`
8. `% Pgm[s] = the given study programme of student s:`
9. `array[1..nStudents] of 1..nPgms: Pgm;`
10. `% Programme[c] = the programme of the student on chair c:`
11. `array[1..nChairs] of var PgmsAndDummy: Programme; % non-mut. red. w/ Student`
12. `% 1-way channelling from Student to Programme, in case dummyS = 0:`
13. `constraint forall(c in 1..nChairs) (Programme[c] = array1d(StudentsAndDummy, [dummyP] ++ Pgm)[Student[c]]);`
14. `% (1) Each table has students of distinct study programmes:`
15. `constraint forall(T in Chairs) (all_different_except([Programme[c] | c in T]), {dummyP});`
16. `... % constraint (2) and objective (3) of slide 5`

Note that Student uniquely determines Programme via Pgm, but not vice-versa: one can also formulate (1) directly with Student via Pgm.
Outline

1. Viewpoints & Dummy Values
2. Implied Constraints
3. Redundant Variables & Channelling Constraints
4. Pre-Computation
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:

```plaintext
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
constraint ... x ... nbrWinners ...;
solve maximize Pools[x] div nbrWinners; % implicit: element!
```

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

- It yields weak inference, at least in CP and LCG solvers.
- Its inference takes unnecessary time and memory.

Idea: We can precompute all possible objective values, as derived parameters.
Example (Prize-Pool Division, revisited)

Precompute a 2d array of derived parameters, indexed by 1..5 and 1..500, for each possible value pair of \( x \) and \( \text{nbrWinners} \):

2. \( \text{array}[1..5] \) of \( \text{int} \): \( \text{Pools} = [1000, 5000, 15000, 20000, 25000] \);
3. \( \text{var 1..5: x}; \) % index of the actual prize pool within \( \text{Pools} \)
4. \( \text{var 1..500: nbrWinners}; \) % the number of winners
5. \( \text{constraint ... x ... nbrWinners ...}; \)
6. \( \text{array}[1..5,1..500] \) of \( \text{int} \): \( \text{ObjVal} = \text{array2d}(1..5, 1..500, \text{[Pools[p] div n | p in 1..5, n in 1..500]}); \) % div on par!
7. \( \text{solve maximize ObjVal[x,nbrWinners]}; \) % implicit: 2d-element!

Example (Kakuro Puzzle, reminder from Topic 3: Constraint Predicates)

We precomputed \text{all_different_sum}(X, \sigma) \text{ for } |X| \in 2..7 \text{ and } \sigma \in 3..35,\text{ say table}([x,y],[|1,3|3,1|]) \text{ for all_different_sum}([x,y],4)\text{ and table}([y,z],[|1,2|2,1|]) \text{ for all_different_sum}([y,z],3), \text{ because MiniZinc has no all_different_sum predicate and its definition by a conjunction of all_different and sum has too poor inference.}