Topic 8: Inference & Search in CP & LCG
(Version of 3rd August 2023)

Pierre Flener and Jean-Noël Monette

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

Outline

Outline

Annotations:

- Annotations provide information to the backend or to the MiniZinc-to-FlatZinc compiler.
- Annotations are optional.
- A backend may ignore any of the annotations.
- The compiler may introduce further annotations.
- Annotations are attached with `::` to model items.
- Annotations do not affect the model semantics.

Annotations to a constraint:

- Annotations can suggest a propagator to use for the constraint by a CP or LCG backend: see slide 8.

Annotations to the objective:

- Annotations can suggest a search strategy to use by a CP or LCG backend: see slide 14.

Outline

Domains (reminder)

Definition

The domain of a decision variable v, denoted here by $\mathrm{dom}(v)$, is the set of values that v can still take during search:

- The domains of the decision variables are reduced by search and by inference (see the next two slides).
- A decision variable is said to be fixed if its domain is a singleton.
- Unsatisfiability occurs if a decision variable domain goes empty.

Note the difference between:

- a domain as a technology-independent declarative entity when modelling;
- a domain as a CP-technology procedural data structure when solving.

CP Solving (reminder)

Tree Search, upon initialising each domain as in the model:

Satisfaction problem:

1. Perform inference (see the next slide).
2. If the domain of some decision variable is empty, then backtrack.
3. If all decision variables are fixed, then we have a solution.
4. Select a non-fixed decision variable v,
   partition its domain into two parts π_1 and π_2, and make two branches:
   one with $v \in \pi_1$, and the other one with $v \in \pi_2$.
5. Recursively explore each of the two branches.

Optimisation problem: when a feasible solution is found at step 3, first add the constraint that the next solution must be better and then backtrack.

CP Inference

Definition

A propagator for a predicate γ deletes from the domains of the variables of a γ-constraint ~~the~~ values that cannot be in a solution to that constraint.
Not all impossible values need to be deleted:

- A domain-consistency (DC) propagator deletes all impossible values from the domains.
- A bounds-consistency (BC) propagator only deletes all impossible minimum and maximum values from the domains.
- A value-consistency (VC) propagator is only awoken when at least one of its decision variables became fixed.

There exist other, unnamed consistencies for propagators.
There is a trade-off between the time & space complexity of a propagator and its achieved deletion of domain values.

Example (Linear equality constraints)

Consider the linear constraint `3 * x + 4 * y = z`
with dom(`x`) = `0..1` = dom(`y`) and dom(`z`) = `0..10`:

- A bounds-consistency propagator reduces dom(`z`) to `0..7`.
- A domain-consistency propagator reduces dom(`z`) to `{0,3,4,7}`.

Time complexity:

- A bounds-consistency propagator for a linear equality constraint can be implemented to run in $\mathcal{O}(n)$ time, where n is the number of decision variables in the constraint.
- A domain-consistency propagator for a linear equality constraint can be implemented to run in $\mathcal{O}(n \cdot d^2)$ time, where n is the number of decision variables in the constraint and d is the sum of their domain sizes, hence in time pseudo-polynomial = exponential in the input magnitude.

Controlling the CP Inference

The choice of propagator for each constraint may be critical for performance. Each CP solver and LCG solver has a default propagator for each available constraint predicate. It is possible to override the defaults with annotations:

- `::` `domain_propagation` asks for a DC propagator.
- `::` `bounds_propagation` asks for a BC propagator.
- `::` `value_propagation` asks for a VC propagator.

Annotations may be ignored, only partially followed, or just approximated: annotations are just suggestions.

Example (Black-Hole Patience)

In Topic 6: Case Studies, the seen inference annotation within the channelling

```
constraint inverse(Card,Pos) :: domain_propagation;
```

has a huge impact with Gecode (CP) + .

Example (n-Queens)

```
array[1..n] of var 1..n: Row;
constraint all_different(Row) :: domain_propagation;
constraint all_different
  ([ Row[c]+c                          | c in 1..n]) ::domain_propagation;
constraint all_different
  ([ Row[c]-c                          | c in 1..n]) ::domain_propagation;
```

Test results with Gecode (CP) to the first solution for `n=101`:

inference	# nodes	seconds
default (no annotation)	348,193	5.5
`bounds_propagation` on `all_different`	348,193	5.5
`domain_propagation` on `all_different`	209,320	3.2

Example (n-Queens)

```
array[1..n] of var 1..n: Row;
constraint all_different(Row) :: domain_propagation;
constraint all_different
  ([(Row[c]+c)::bounds_propagation | c in 1..n]) ::domain_propagation;
constraint all_different
  ([(Row[c]-c)::bounds_propagation | c in 1..n]) ::domain_propagation;
```

Test results with Gecode (CP) to the first solution for `n=101`:

inference	# nodes	seconds
default (no annotation)	348,193	5.5
`bounds_propagation` on `all_different`	348,193	5.5
`domain_propagation` on `all_different`	209,320	3.2
+ `bounds_propagation` on the linear constraints	> 20M	> 600.0
`bounds_propagation` on all the constraints	> 20M	> 600.0

Asking for bounds consistency on the implicit linear equality constraints backfires here, as each is on only 2 decision variables,
but it may pay off upon more decision variables (and be default then).

Outline

Search Strategies

Search Strategies:

- On which decision variable to branch next?
- How to partition the domain of the chosen decision variable?
- Which search (depth-first, breadth-first, . . .) to use?

The search is usually depth-first left-to-right search.

One can suggest to a CP or LCG backend on which decision variable to branch and how, by making an annotation with:

- a variable selection strategy, and
- a domain partitioning strategy.

A search annotation is sometimes exploited for MIP solvers.

Variable Selection Strategy

The variable selection strategy has an impact on the size of the search tree, especially if the constraints are processed with propagation at every node of the search tree, or if the whole search tree is explored: for example, when it is an optimisation problem or when there are no solutions.

Example (Impact of the variable selection strategy)

Consider `var 1..2: x`, `var 1..4: y`, `var 1..6: z`, branching on all domain values, but no constraints:

- If selecting the decision variables in the order `x`, `y`, `z`, then the CP search tree has $1 + 2 + 2 \cdot 4 + 2 \cdot 4 \cdot 6 = 59$ nodes and $2 \cdot 4 \cdot 6 = 48$ leaves.
- If selecting the decision variables in the order `z`, `y`, `x`, then the CP search tree has $1 + 6 + 6 \cdot 4 + 6 \cdot 4 \cdot 2 = 79$ nodes and also $6 \cdot 4 \cdot 2 = 48$ leaves.

Definition (First-Fail Principle)

To succeed, first try where you are most likely to fail. In practice:

- Select a decision variable with the smallest current domain.
- Select a decision variable involved in the largest number of constraints.
- Select a decision variable recently causing the most backtracks.

Example (Impact of the variable selection strategy)

Finding the first solution to `101`-queens with Gecode (CP):

search	# nodes	seconds
default (no annotation)	348,193	5.5
`first_fail`	323,275	5.3
`anti_first_fail`	$>$ 20M	$>$ 600.0
`input_order`	$>$ 13M	$>$ 600.0

(Continued on slide 18)

Domain Partitioning Strategy

The domain partitioning strategy has an impact on the size of the search tree when optimising, when only searching for the first solution, or when performing incomplete search (say by using a time-out).

Example (Impact of the domain partitioning strategy)

Consider `var 1..2: x`, `var 1..4: y`, `var 1..6: z`, domain consistency for `x * y = z`, `x != y`, `x != z`, and `y != z`, smallest-domain variable selection, and depth-first search:

- If the domain is split into singletons by increasing order, then 6 CP nodes are explored before finding the (unique) solution.
- If the domain is split into singletons by decreasing order, then only 2 CP nodes (the root and a leaf) are explored before finding the (unique) solution, without backtracking.

Definition (Best-First Principle)

First try a domain part that is most likely, if not guaranteed, to have values that lead to solutions. This may be like how one would make the greedy choice in a greedy algorithm for the problem at hand, considering its objective function.

Example (Impact of the domain partitioning strategy)

(Continued from slide 16)

Finding the first solution to `101`-queens with Gecode (CP):

search	# nodes	seconds
default (no annotation)	348,193	5.5
`first_fail, indomain_min`	348,193	5.6
`first_fail, indomain`	323,275	5.3
`first_fail, indomain_median`	96	0.1

Motivation for First-Fail and Best-First[1]

	Finding a solution	Detecting unsatisfiability
Variable selection	Must consider all the remaining decision variables	Need not consider all the remaining decision variables: ☞ detect unsatisfiability a.s.a.p.
Domain partitioning	Need not consider all the remaining values: ☞ find a solution a.s.a.p.	Must consider all the remaining values

[1] Based on material by Yves Deville and Pascal Van Hentenryck

Definition (Integer Brancher)

A brancher `int_search(X,`ϕ`,`ψ`)` selects a non-fixed decision variable in the array `X` of integer decision variables, using as variable selection strategy ϕ one of the following:

- `input_order`: select the next decision variable by the order in `X`
- `first_fail`: select a decision variable with the smallest domain
- `smallest`: select a decision variable with smallest minimum in its domain
- `largest`: select a decision variable with largest maximum in its domain
- `occurrence`: select a decision variable involved in the largest number of active propagators
- `most_constrained`: use `first_fail`, break ties with `occurrence`
- `max_regret`: select a decision variable with the largest difference between the two smallest values in its domain
- ... (see Section 4.2.1.2 of the MiniZinc Handbook)

Ties are broken by the order in `X`. (Continued on next slide)

Definition (Integer Brancher, end)

Then, for the chosen decision variable, say v, the brancher selects values in $\mathrm{dom}(v) = \{d_1, \ldots, d_n\}$, with $n \geq 2 \wedge d_1 < \cdots < d_n$, and builds guesses, which are constraints, using as domain partitioning strategy ψ one of the following:

- `indomain`: branch left-to-right on $v = d_1, \ \ldots, \ v = d_n$
- `indomain_min`: branch left on $v = d_1$ and right on $v \neq d_1$
- `indomain_middle`: select d_i nearest the middle $\dot{m} = \lfloor (d_1 + d_n)/2 \rfloor$ so as to branch left on $v = d_i$ and right on $v \neq d_i$
- `indomain_median`: select the median $d_i = d_{\lfloor (n+1)/2 \rfloor}$ so as to branch left on $v = d_i$ and right on $v \neq d_i$
- `indomain_split`: branch left on $v \leq \dot{m}$ and right on $v > \dot{m}$
- `indomain_reverse_split`: branch left on $v > \dot{m}$ and right on $v \leq \dot{m}$
- `outdomain_random`: select a random value d_i so as to branch left on $v \neq d_i$ and right on $v = d_i$
- ... (see Section 4.2.1.2 of the MiniZinc Handbook)

Definition (Boolean Brancher)

A brancher `bool_search(X,`ϕ`,`ψ`)` selects a non-fixed decision variable in the array `X` of Boolean decision variables, using variable selection strategy ϕ and domain partitioning strategy ψ, with the same choices as for integer decision variables, under the convention `false < true`.

Definition (Chaining of Branchers)

A brancher `seq_search([`$\beta_1, \ldots, \beta_n$`])` chains branchers $\beta_1, \ldots, \beta_n$: when some brancher β_i is finished, branch with β_{i+1}.

Careful: A search annotation goes between the `solve` and `satisfy`, `minimize`, or `maximize` keywords, and it is ignored elsewhere.
See the example on slide 37.

The search strategy of Gecode for the decision variables that are not in the search annotation depends also on the `output` statement.

Definition

A set (decision) variable takes a set as value, and has a set of sets as domain. For its domain to be finite, a set decision variable must be a subset of a given finite set (called Σ below).

Integers are totally ordered, but sets are partially ordered: propagation for set decision variables is much harder. Also, set domains can get huge: $\mathcal{O}(2^{|\Sigma|})$.
A trade-off is to over-approximate the domain of a set decision variable S by a pair $\langle \ell, u \rangle$ of finite sets, denoting the set of all sets σ such that $\ell \subseteq \sigma \subseteq u \subseteq \Sigma$:

- ℓ is the current set of mandatory elements of S;
- $u \setminus \ell$ is the current set of optional elements of S.

Example

The domain of a set decision variable represented as $\langle \{1\}, \{1,2,3,4\} \rangle$ has the sets $\{1\}$, $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, $\{1,2,3\}$, $\{1,2,4\}$, $\{1,3,4\}$, and $\{1,2,3,4\}$. Deleting $\{1,2,3\}$ from this domain is impossible!

Definition (Set Brancher)

A brancher `set_search(X,`ϕ`,`ψ`)` selects a non-fixed decision variable $S \doteq \langle \ell, u \rangle$ in the array `X` of set decision variables, using a variable selection strategy ϕ on slide 20:

- `first_fail`: select a decision variable with the smallest $|u \setminus \ell|$
- `smallest`: select a decision variable with the smallest $\min(u \setminus \ell)$
- ... (see Section 4.2.1.2 of the MiniZinc Handbook)

Then, for the chosen decision variable, say $S \doteq \langle \ell, u \rangle$, it selects an element in $u \setminus \ell = \{d_1, \ldots, d_n\}$, with $d_1 < \cdots < d_n$, and builds guesses using a domain partitioning strategy ψ on slide 21 with the following semantics here:

- `indomain_min`: branch left on $d_1 \in S$ and right on $d_1 \notin S$
- `outdomain_max`: branch left on $d_n \notin S$ and right on $d_n \in S$
- `outdomain_median`: select the median $d_i = d_{\lfloor (n+1)/2 \rfloor}$ so as to branch left on $d_i \notin S$ and right on $d_i \in S$
- ... (see Section 4.2.1.2 of the MiniZinc Handbook)

Designing Search Strategies

Problem-specific strategies:
Beside general principles (first-fail and best-first), there are often good strategies that can be designed using problem-specific knowledge.
In MiniZinc, it is often easy to express such strategies in terms of problem-specific concepts.

Interaction with symmetry-breaking constraints:
For higher solving speed, suggest a domain partitioning that drives the search towards solutions satisfied by the symmetry-breaking constraints.

Counter-example

For `a + b + c = 38`, with all decision variables in `1..19`,
and `symmetry_breaking_constraint(a < b /\ b < c)`,
do not use `int_search([a,b,c],input_order,indomain_max)`.

Interaction with the choice of dummy values:

For higher solving speed, suggest a domain partitioning that drives the search towards trying the dummy values (recall the examples of Topic 4: Modelling) at the right moment.

Example (Student Seating, viewpoint 2 revisited again)

```
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]);
...
```

Under Gecode default search, using `dummyS = 0` is a lot slower than using `dummyS = nStudents + 1`, whose speed can however be matched by `dummyS = 0` with `int_search(Student,first_fail,indomain_max)`, for example: search should only try and seat a dummy student on a chair after it turns out that no real student can be seated on it.

Outline

Outline

Agricultural experiment design, AED

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	1	1	1	0	0	0	0
corn	1	0	0	1	1	0	0
millet	1	0	0	0	0	1	1
oats	0	1	0	1	0	1	0
rye	0	1	0	0	1	0	1
spelt	0	0	1	1	0	0	1
wheat	0	0	1	0	1	1	0

Constraints to be satisfied:

1. Equal growth load: Every plot grows 3 grains.
2. Equal sample size: Every grain is grown in 3 plots.
3. Balance: Every grain pair is grown in 1 common plot.

Instance: 7 plots, 7 grains, 3 grains/plot, 3 plots/grain, balance 1.
General term: balanced incomplete block design (BIBD).

The following constraints (of Topic 5: Symmetry) break all the row and column symmetries, but not all their compositions: +

```
constraint symmetry_breaking_constraint(
  forall(v in Varieties diff {max(Varieties)})(
  lex_greater(BIBD[v,..],BIBD[enum_next(v),..])));
constraint symmetry_breaking_constraint(
  forall(b in Blocks diff {max(Blocks)})(
  lex_greatereq(BIBD[..,b],BIBD[..,enum_next(b)])));
```

The use of `lex_greatereq` (as opposed to `lex_lesseq`) is justified by:

- All `BIBD[v,b]` decision variables have the same `0..1` domain, so the first-fail principle cannot distinguish between them: let us fill the `BIBD` incidence matrix in input order (left-to-right in each row, and top-down across the rows).
- Since typically fewer `1`s than `0`s occur in a BIBD, the best-first principle suggests trying `1` before `0`.

```
solve::int_search(BIBD,input_order,indomain_max) satisfy;
```

Outline

The Warehouse Location Problem (WLP)

A company considers opening warehouses at some candidate locations in order to supply its existing shops:

- Each candidate warehouse has the same maintenance cost.
- Each candidate warehouse has a supply capacity, which is the maximum number of shops it can supply.
- The supply cost to a shop depends on the supplying warehouse.

Determine which candidate warehouses actually to open, and which of them supplies which shops, so that:

1. Each shop is supplied by exactly one actually opened warehouse.
2. Each actually opened warehouse supplies a number of shops that is at most equal to its supply capacity.
3. The sum of the actually incurred maintenance costs and supply costs is minimal.

WLP: Sample Instance Data

$$\texttt{Shops} = \{\text{Shop}_1, \text{Shop}_2, \ldots, \text{Shop}_{10}\}$$

$$\texttt{Warehouses} = \{\text{Berlin, London, Ankara, Paris, Rome}\}$$

$$\texttt{maintCost} = 30$$

$\texttt{Capacity} =$

Berlin	London	Ankara	Paris	Rome
1	4	2	1	3

$\texttt{SupplyCost} =$

	Berlin	London	Ankara	Paris	Rome
Shop_1	20	24	11	25	30
Shop_2	28	27	82	83	74
Shop_3	74	97	71	96	70
Shop_4	2	55	73	69	61
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Shop_{10}	47	65	55	71	95

WLP Model 1: Decision Variables (reminder)

Automatic enforcement of the total-function constraint (1):

`Supplier` =

Shop_1	Shop_2	$\cdots$	Shop_{10}
$\in$ `Warehouses`	$\in$ `Warehouses`	$\cdots$	$\in$ `Warehouses`

`Supplier[s]` denotes the supplier warehouse for shop `s`.

Variables redundant with `Supplier`, but not mutually, as less informative:

`Open` =

Berlin	London	Ankara	Paris	Rome
$\in$ `0..1`	$\in$ `0..1`	$\in$ `0..1`	$\in$ `0..1`	$\in$ `0..1`

`Open[w]=1` if and only if warehouse `w` is actually opened.

WLP Model 1: Annotations

The capacity constraint, using `global_cardinality_closed` and boosted with the inference annotation `::domain_propagation`, and the channelling constraint from `Supplier` to `Open` are as in Topic 6: Case Studies.

Let the new decision variable `Cost[s]` denote the actually incurred supply cost for shop `s`. It is non-mutually redundant with `Supplier[s]`, as less informative, and has the following one-way channelling constraint:

```
forall(s in Shops)(Cost[s] = SupplyCost[s,Supplier[s]]);
```

The objective now syntactically simplifies into:

```
solve minimize maintCost * sum(Open) + sum(Cost);
```

For shop `s`, let dom(`Cost[s]`) $= \{d_1, d_2, \ldots, d_n\}$, with $n \geq 2 \wedge d_1 < \cdots < d_n$: the regret of shop `s` is $d_2 - d_1$, that is the difference in supply cost between its currently cheapest and second-cheapest potential supplying warehouses.

The maximal-regret strategy recommends:

- **Variable selection:** Select a decision variable `Cost[s]` such that the shop `s` currently has the maximal regret.
- **Value selection and guesses:** Select value $d = \min(\text{dom}(\texttt{Cost[s]}))$. Branch left on `Cost[s]` $= d$ and right on `Cost[s]` $\neq d$.

The `Supplier[s]` decision variables are then branched on by increasing order of `s` and by increasing value. This brancher accelerates search only if some values in `SupplyCost[s,..]` are equal for some shop `s`.

Upon the first seen one-way channelling from `Supplier` to `Open`, the `Open[w]` decision variables are then branched on by increasing order of `w` and by increasing value, in order to fix any still non-fixed `Open[w]` to `0` faster than by relying upon minimisation (which we did in Topic 6: Case Studies).

This search strategy is expressed in MiniZinc as follows:

```
solve
  :: seq_search([
       int_search(Cost,max_regret,indomain_min),
       int_search(Supplier,input_order,indomain_min),
       int_search(Open,input_order,indomain_min)
  ])
  minimize maintCost * sum(Open) + sum(Cost)
```

Objective values, upon the 3 seen ways of channelling, within 35 seconds by Gecode (CP) on a MacBook-Air laptop, on a hard instance with 16 warehouses of capacity 4 supplying 50 shops, of minimal cost at most 1,190,733:

	Model 1		Model 2
search	1-way(17)	1-way(18)	none
default (no annotation)	none	1,869,494	1,864,913
`first_fail` on `Supplier`	1,520,326	1,524,034	1,524,034
`first_fail` on `Cost`	1,218,079	1,223,704	1,218,079
`max_regret` on `Cost`	1,193,637	1,198,276	1,193,637

Outline

The Sport Scheduling Problem (SSP)

Find a schedule in $\texttt{Periods} \times \texttt{Weeks} \rightarrow \texttt{Teams} \times \texttt{Teams}$ for

- $|\texttt{Teams}| = \texttt{n}$ and $\texttt{n}$ is even (note that only $\texttt{n=4}$ is unsatisfiable)
- $|\texttt{Weeks}| = \texttt{n-1}$
- $|\texttt{Periods}| = \texttt{n/2}$ periods per week

subject to the following constraints:

1. Each possible game is played exactly once.
2. Each team plays exactly once per week.
3. Each team plays at most twice per period.

Idea for a model, and a solution for $\texttt{n=8}$:

	Wk 1	Wk 2	Wk 3	Wk 4	Wk 5	Wk 6	Wk 7
P 1	1 vs 2	1 vs 3	2 vs 6	3 vs 5	4 vs 7	4 vs 8	5 vs 8
P 2	3 vs 4	2 vs 8	1 vs 7	6 vs 7	6 vs 8	2 vs 5	1 vs 4
P 3	5 vs 6	4 vs 6	3 vs 8	1 vs 8	1 vs 5	3 vs 7	2 vs 7
P 4	7 vs 8	5 vs 7	4 vs 5	2 vs 4	2 vs 3	1 vs 6	3 vs 6

The Sport Scheduling Problem (SSP)

Find a schedule in $\texttt{Periods} \times \texttt{Weeks} \rightarrow \texttt{Teams} \times \texttt{Teams}$ for

- $|\texttt{Teams}| = \texttt{n}$ and $\texttt{n}$ is even (note that only $\texttt{n=4}$ is unsatisfiable)
- $|\texttt{Weeks}| = \texttt{n-1}$
- $|\texttt{Periods}| = \texttt{n/2}$ periods per week

subject to the following constraints:

1. Each possible game is played exactly once.
2. Each team plays exactly once per week.
3. Each team plays at most twice per period.

Idea for a model, and a solution for $\texttt{n=8}$, with a dummy week $\texttt{n}$ of duplicates:

	Wk 1	Wk 2	Wk 3	Wk 4	Wk 5	Wk 6	Wk 7	Wk 8
P 1	1 vs 2	1 vs 3	2 vs 6	3 vs 5	4 vs 7	4 vs 8	5 vs 8	6 vs 7
P 2	3 vs 4	2 vs 8	1 vs 7	6 vs 7	6 vs 8	2 vs 5	1 vs 4	3 vs 5
P 3	5 vs 6	4 vs 6	3 vs 8	1 vs 8	1 vs 5	3 vs 7	2 vs 7	2 vs 4
P 4	7 vs 8	5 vs 7	4 vs 5	2 vs 4	2 vs 3	1 vs 6	3 vs 6	1 vs 8

SSP Model 1: Decision Variables (reminder)

Declare a 3d matrix `Team[Periods,ExtendedWeeks,Slots]` of decision variables in `Teams` (denoted `T` below), over a schedule extended by a dummy week where teams play fictitious duplicate games in the period where they would otherwise play only once, thereby strengthening constraint (3) into:

(3') Each team plays exactly twice per period.

Let `Team[p,w,s]` be the team that plays in period `p` of week `w` in game slot `s`:

		Wk 1		⋯		Wk $n-1$		Wk n	
		one	two	⋯	⋯	one	two	one	two
`Team` =	P 1	∈ `T`	∈ `T`	⋯	⋯	∈ `T`	∈ `T`	∈ `T`	∈ `T`
	⋮	⋮	⋮	⋱	⋱	⋮	⋮	⋮	⋮
	P $n/2$	∈ `T`	∈ `T`	⋯	⋯	∈ `T`	∈ `T`	∈ `T`	∈ `T`

SSP Model 1: More Decision Variables (reminder)

Rather declare a 2d matrix `Game[Periods,Weeks]` of decision variables in `Games` over the non-extended weeks.

Let `Game[p,w]` be the game played in period `p` of week `w`:

$$\texttt{Game} = \begin{array}{r|c|c|c|}
 & \text{Week 1} & \cdots & \text{Week } n-1 \\
\hline
\text{Period 1} & \in \texttt{Games} & \cdots & \in \texttt{Games} \\
\hline
\vdots & \vdots & \ddots & \vdots \\
\hline
\text{Period } n/2 & \in \texttt{Games} & \cdots & \in \texttt{Games} \\
\hline
\end{array}$$

The 2d matrix `Game` is mutually redundant with the first $n-1$ 2d columns of the 3d matrix `Team`, which is over the extended weeks.

SSP Model 1: Channelling Constraint

Two-way channelling constraint (reminder):

```
constraint forall(p in Periods, w in Weeks)
  (Team[p,w,one] * n + Team[p,w,two] = Game[p,w]);
```

The game number in `Game` of each period and week corresponds to the teams scheduled at that time in `Team`.

If a CP or LCG solver cannot enforce domain consistency on linear equality, even when `::` `domain_propagation` is used, then precompute a table:

```
constraint forall(p in Periods, w in Weeks)
 (table([Team[p,w,one],Team[p,w,two],Game[p,w]],
 array2d(1..(n*(n-1) div 2), 1..3,
 [[f,s,f*n+s][i] | f,s in Teams where f<s, i in 1..3])));
 % [|1,2,6|1,3,7|1,4,8|2,3,11|2,4,12|3,4,16|] for n=4
```

SSP Model 1: Search Annotation

It suffices to follow the first-fail principle:

- **Variable selection:** Select a decision variable `Game[p,w]` with the currently smallest domain.
- **Value selection and guesses:** Select value $d = \min(\text{dom}(\texttt{Game[p,w]}))$. Branch left on `Game[p,w]` $= d$ and right on `Game[p,w]` $\neq d$.

The `Team[p,w,s]` decision variables need no brancher as they take their values through either the 2-way channelling constraint, especially if propagated to domain consistency, and the `global_cardinality_closed(...)` formulation of constraint (3') in Topic 6: Case Studies.

This search strategy is expressed in MiniZinc as follows:

```
:: int_search(Game,first_fail,indomain_min)
```

SSP Model 2: Smaller Domains for `Game[p,w]` Variables

A round-robin schedule suffices to break many of the remaining symmetries:

- Restrict the games of the first week to the set $\{1 \text{ vs } 2\} \cup \{t+1 \text{ vs } n+2-t \mid 1 < t \leq n/2\}$
- For the remaining weeks, transform each game f vs s of the previous week into a game f' vs s', where

$$f' = \begin{cases} 1 & \text{if } f = 1 \\ 2 & \text{if } f = n \\ f+1 & \text{otherwise} \end{cases}, \text{ and } s' = \begin{cases} 2 & \text{if } s = n \\ s+1 & \text{otherwise} \end{cases}$$

The constraints (1) and (2) are now automatically enforced:
we must only find the period of each game, but not its week.

Search strategy: Choose games for the first period across the weeks, then for the first week across the remaining periods, then for the next period across the remaining weeks, then for the next week across the remaining periods, etc.

Interested in More Details?

For more details on WLP and SSP and their search strategies, see:

Van Hentenryck, Pascal.
The OPL Optimization Programming Language.
The MIT Press, 1999.

Van Hentenryck, Pascal.
Constraint and integer programming in OPL.
INFORMS Journal on Computing, 14(4):345 – 372, 2002.

Van Hentenryck, Pascal; Michel, Laurent; Perron, Laurent; and Régin, Jean-Charles.
Constraint programming in OPL.
PPDP 1999, pages 98 – 116. *Lecture Notes in Computer Science* 1702. Springer-Verlag, 1999.