Topic 12: CP and the MiniCP Solver
(Version of 28th October 2021)

Pierre Flener

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

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Reminder from Topic 1: Introduction

A solving technology offers methods and tools for:

what: **Modelling** constraint problems in **declarative** language.

and / or

how: **Solving** constraint problems **intelligently**:

- **Search**: Explore the space of candidate solutions.
- **Inference**: Reduce the space of candidate solutions.
- **Relaxation**: Exploit solutions to easier problems.

A solver is a software that takes a model & data as input and tries to solve the modelled problem instance.
Constraint Programming Technology

Constraint programming (CP) offers methods and tools for:

what: Modelling constraint problems in a high-level language.

and

how: Solving constraint problems intelligently by:

- either default systematic search upon pushing a button
- or systematic search guided by a user-given strategy
- or local search guided by a user-given strategy

with lots of inference, called propagation in the case of systematic search, but yet little relaxation.

Slogan of CP:

Constraint Program = Model [ + Search ]
CP Solving = Inference + Search

A CP solver conducts search interleaved with inference:

Each constraint has an inference algorithm.
Inference for *One* Constraint: Propagator

Example

Consider the constraint \texttt{CONNECTED}([C_1, \ldots, C_n]), which imposes max one stretch per colour among the \(n\) variables.

From the following current *partial* valuation for \(n = 6\):

\[
\begin{array}{cccc}
\text{black} & \text{red} & C_3 & \text{red} & \text{yellow} & C_6
\end{array}
\]

a propagator (under systematic search) of the \texttt{CONNECTED} predicate can infer that \(C_3 = \text{red}\) and \(C_6 \notin \{\text{red}, \text{black}\}\):

\[
\begin{array}{cccccc}
\text{black} & \text{red} & \text{red} & \text{red} & \text{red} & \text{yellow} & C_6
\end{array}
\]

* A propagator deletes the impossible values from the current domains of the variables, and thereby accelerates otherwise blind search.
Roadmap

For CP by systematic search:

- **Consistency:** A consistency (Part 1) is the targeted characterisation of the domain values (Part 2) kept by a propagator (a musician; aka filtering algorithm) for a constraint, but correctness of the solver (the whole orchestra) must not depend on actually enforcing it.

- **Propagation:** The `fixPoint` algorithm (of the conductor) decides which propagator to run when (Part 1).

- **Search:** The `DFSearch` algorithm (of the conductor) calls `fixPoint` and a branching scheme (Parts 1, 3, 8).

- **Propagators:** We design propagators for `Sum` and `Element` (Part 4), `Circuit` (5), `AllDifferent` (6), `Table` (7), `Cumulative` (9), and `Disjunctive` (10).

For CP by local search (LS):

- Large-neighbourhood search: hybrid of LS and CP (5).
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Mind the Gap

- MiniCP is a white-box, bottom-up, open-source teaching framework for CP, implemented in Java. It is fully functional, but not as engineered and complete as industry-strength CP solvers like Gecode.

- In MiniZinc, a domain is a declared for each variable. In CP solvers, a domain is a *dynamically* shrinking data structure for a variable, initialised to its declared one.

- With CP solvers, one writes an imperative program that states (or: posts) — via any combination of sequential, conditional, iterative, and recursive composition — the declarative constraints, which are given to the solver via propagators enforcing user-chosen consistencies.

- MiniCP does not automatically coerce Booleans (truth is 1, and falsity is 0) into integers, and MiniZinc does.
Reification

A MiniZinc reified constraint, such as \( b \leftrightarrow \gamma(\ldots) \), where \( b \) is a variable of type \texttt{bool}, can be modelled for MiniCP upon naming it and designing a propagator for it. Assume there are search guesses or other constraints on the reifying Boolean (0 / 1 in MiniCP) variable \( b \):

- When \( b \) gets fixed to 1, post the constraint \( \gamma(\ldots) \).
- When \( b \) gets fixed to 0, post the constraint \( \neg\gamma(\ldots) \).
- When \( \gamma(\ldots) \) gets subsumed, post the constraint \( b=1 \).
- When \( \neg\gamma(\ldots) \) gets subsumed, post the constraint \( b=0 \).

where \( \neg\gamma(\ldots) \) denotes the complement of \( \gamma(\ldots) \), not some code for \texttt{not}\( \gamma(\ldots) \), as CP solvers do not implement \texttt{not}. Propagation may be poor! Due to \( \neg \) reification may be hard!
Constraint combination with reification:
With reification, constraints can be arbitrarily combined with logical connectives: negation ($\neg$), disjunction ($\lor$), conjunction ($\land$), implication ($\Rightarrow$), and equivalence ($\leftrightarrow$). However, propagation may be very poor!

Example

The composite constraint $(\gamma_1 \land \gamma_2) \lor \gamma_3$ is modelled as

$$(b_1 \leftrightarrow \gamma_1) \land (b_2 \leftrightarrow \gamma_2) \land (b_3 \leftrightarrow \gamma_3)$$
$$& (b_1 \cdot b_2 = b) \land (b + b_3 \geq 1)$$

Hence even the constraints $\gamma_1$ and $\gamma_2$ must be reified. If $\gamma_1$ is $x = y + 1$ and $\gamma_2$ is $y = x + 1$, then $\gamma_1 \land \gamma_2$ is unsat; however, $b$ is then not fixed to value 0 by propagation, as each propagator works individually and there is no communication through the shared variables $x$ and $y$; hence $b_3 = 1$ is not propagated and $\gamma_3$ is not forced to hold.
Remember the warning in Topic 2: Basic Modelling that the disjunction and negation of constraints (with \/, xor, not, <-, ->, <->, exists, xorall, if \theta then \phi else \psi endif) in MiniZinc often makes the solving slow?

**Example**

The MiniZinc disjunctive constraint

```
constraint x = 0 \/or x = 9;
```

is modelled for MiniCP (and flattened) with reification:

```
(b_0 \iff x = 0) \& (b_9 \iff x = 9) \& (b_0 + b_9 \geq 1)
```

But it is logically equivalent to

```
constraint x in \{0, 9\};
```

where no reification is involved, and no further propagation.
Remember the strong warning in Topic 2: Basic Modelling about a conditional `if θ then φ₁ else φ₂ endif` or a comprehension, say `[i | i in ρ where θ]`, in MiniZinc having a test θ that depends on variables?

**Example**

Consider `var 1..9: x` and `var 1..9: y` for

```
forall(i in 1..9 where i > x)(i > y)
```

Recall that this is syntactic sugar for

```
forall([i > y | i in 1..9 where i > x])
```

This is modelled for MiniCP (and flattened) with reification:

```
forall(i in 1..9)(i > x -> i > y)
```

that is with a logical implication (`->`), hence with a hidden logical disjunction (`\/`): for each i, both sub-constraints are reified as both have variables.
A MiniZinc inference annotation (recall Topic 8: Inference & Search in CP & LCG) to a constraint, \textit{bounds} or \textit{domain}, is prescribed for MiniCP upon designing for the predicate of that constraint a propagator enforcing that consistency.

\textbf{Example}

We design propagators that enforce various consistencies, even others than bounds and domain consistency (Part 1), for \texttt{Sum} and \texttt{Element} (Part 4), \texttt{AllDifferent} (Part 6), \texttt{Circuit} (Part 5), \texttt{Table} (Part 7), \texttt{Cumulative} (Part 9), and \texttt{Disjunctive} (Part 10).
Search: Selection Strategies

A MiniZinc search annotation (recall Topic 8: Inference & Search in CP & LCG) to an objective, such as `int_search(X, first_fail, indomain_min)`, is prescribed for MiniCP by providing a branching scheme, which selects an unfixed variable \( x \) and returns an array (empty if no such \( x \) exists) of branching constraints according to a partition of the current domain of \( x \).

Example

We implement (Parts 1, 3, 8) variable-selection strategies, such as various realisations of the first-fail principle, and value-selection strategies for domain partitioning, such as various realisations of the best-first principle.
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Adding a \texttt{Sum} Predicate

A MiniZinc linear constraint, such as the linear equality 
\begin{equation}
\text{sum}(i \text{ in } 1..n)(A[i] \times X[i]) = d,
\end{equation}
can be modelled for MiniCP upon writing a propagator for a \texttt{Sum} predicate:

\textbf{Definition}

A \texttt{Sum}([a_1, \ldots, a_n], [x_1, \ldots, x_n], R, d) constraint, with

- \([a_1, \ldots, a_n]\) a sequence of non-zero integer constants,
- \([x_1, \ldots, x_n]\) a sequence of integer variables,
- \(R\) in \{\textless, \leq, =, \neq, \geq, \rangle\}, and
- \(d\) an integer constant,

holds iff the linear relation \((\sum_{i=1}^{n} a_i \cdot x_i) \ R \ d\) holds.

It is easy to reify \texttt{Sum}. In Part 4, we write a polynomial-time propagator that enforces bounds consistency on the \(x_i\) for \((\sum_{i=1}^{n} x_i) = 0\), whose domain consistency is NP-hard.
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Adding an \textbf{Element} Predicate

A MiniZinc constraint on an array element at an unknown index $i$, such as $\text{element}(i,X,e)$ or $X[i]=e$ or a constraint on $X[i]$, can be modelled for MiniCP upon designing a propagator for an \textbf{Element} predicate:

\begin{definition} (Van Hentenryck and Carillon, 1988)

An $\text{Element}([x_1, \ldots, x_n], i, e)$ constraint, where the $x_j$ are variables, $i$ is an integer \textit{variable}, and $e$ is a variable, holds if and only if $x_i = e$.

\end{definition}

One can generalise \textbf{Element} to multi-dimensional arrays. It is hard to reify \textbf{Element}. In Part 4, we write propagators that enforce various consistencies on the various variables, depending on the number of dimensions of the array and on whether its elements $x_j$ are variables or parameters.
Example (Warehouse Location Problem)

Recall the one-way channelling constraint of Model 1 (in Topic 6: Case Studies) from the Supplier variables to its non-mutually redundant Open variables:

\[
\text{constraint } \forall (s \text{ in Shops}) \left( \text{Open}[\text{Supplier}[s]] = 1 \right);
\]

This must be modelled for MiniCP as in the following MiniZinc reformulation:

\[
\text{constraint } \forall (s \text{ in Shops}) \left( \text{element} (\text{Supplier}[s], \text{Open}, 1) \right);
\]
Example (Warehouse Location Problem, a last time)

Recall the objective of Model 1 in Topic 6: Case Studies:

\[
\text{solve minimize maintCost} \times \text{sum(Open)} \\
+ \text{sum(s in Shops)} \cdot (\text{SupplyCost}[s,\text{Supplier}[s]])
\]

This must be modelled for MiniCP as in the following MiniZinc reformulation, by explicitly creating a Cost\[s\] variable and an element constraint for each implicit one:

\[
\begin{align*}
\text{Cost}[s] &= \text{actually incurred supply cost for } s: \\
\text{array}[\text{Shops}] \text{ of var } 0..\text{max}(\text{SupplyCost}): \text{Cost}; \\
\text{constraint for all}(s \in \text{Shops}) \\
&\quad \text{element}(\text{Supplier}[s], \text{SupplyCost}[s,..], \text{Cost}[s]); \\
\text{solve minimize maintCost} \times \text{sum(Open)} + \text{sum(Cost)};
\end{align*}
\]

Recall that we actually introduced these Cost\[s\] variables (in Topic 8: Inference & Search in CP & LCG) in order to state a maximal-regret search strategy on those variables.
Example (Job allocation at minimal salary cost)

Remember the model in Topic 3: Constraint Predicates:

1. \( \text{array[Apps] of 0..1000: Salary; } \% \text{ Salary[a]/job by a} \)
2. \( \text{array[Jobs] of var Apps: Worker; } \% \text{ job j by Worker[j]} \)
3. \( \text{solve minimize sum(j in Jobs)(Salary[Worker[j]])}; \)
4. \( \text{constraint ...;} \% \text{ qualifications, workload, etc} \)

Line 3 must be modelled for MiniCP as in the following MiniZinc reformulation, by explicitly creating a \( \text{Cost[j]} \) variable and an element constraint for each implicit one:

\[
\text{array[Jobs] of var 0..max(Salary): Cost; } \% \text{ Cost[j] for job j}
\]
\[
\text{constraint forall(j in Jobs)}
\]
\[
(\text{element(Worker[j],Salary,Cost[j]))});
\]
\[
\text{solve minimize sum(Cost)};
\]
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Adding an AllDifferent Predicate

An MiniZinc constraint of pairwise difference, such as `alldifferent(X)`, can be modelled for MiniCP upon designing a propagator for an AllDifferent predicate:

**Definition (Laurière, 1978)**

An `AllDifferent([x_1, \ldots, x_n])` constraint holds if and only if all the variables `x_i` take different values.

This is logically equivalent to \( \frac{n(n-1)}{2} \) disequality constraints:

\[
\forall i, j \in 1..n \quad \text{where} \quad i < j : x_i \neq x_j
\]

It is hard to reify `AllDifferent`. In Part 6, we write several propagators that enforce various consistencies on the variables, namely a new consistency and domain consistency, which both usually lead to faster solving than the \( \Theta(n^2) \) disequality constraints above.
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Adding a Cumulative Predicate

A MiniZinc constraint on the bounded cumulative resource requirement of tasks, such as `cumulative(S,D,R,u)`, can be modelled for MiniCP upon designing a propagator for a Cumulative predicate:

**Definition (Aggoun and Beldiceanu, 1993)**

A `Cumulative([s_1, \ldots, s_n],[d_1, \ldots, d_n],[r_1, \ldots, r_n],u)` constraint, where each task $T_i$ has a starting time $s_i$, a duration $d_i$, and a resource requirement $r_i$, holds if and only if the resource upper limit $u$ is never exceeded when performing the tasks $T_i$.

It is hard to reify `Cumulative`. In Part 9, we design several propagators that enforce various consistencies on the starting-time variables $s_i$, assuming all other arguments are parameters.
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Adding a **Disjunctive Predicate**

A MiniZinc temporal non-overlap constraint on tasks, such as `disjunctive(S,D)`, can be modelled for MiniCP upon designing a propagator for a `Disjunctive` predicate:

**Definition (Carlier, 1982)**

A `Disjunctive([[s_1, \ldots, s_n], [d_1 \ldots, d_n]])` constraint, where each task \( T_i \) has a starting time \( s_i \) and a duration \( d_i \), holds if and only if no two tasks \( T_i \) and \( T_j \) overlap in time.

It is hard to reify `Disjunctive`. In Part 10, we design several propagators that enforce various consistencies on the starting-time variables \( s_i \), assuming the durations \( d_i \) are parameters.
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
Adding a Circuit Predicate

A MiniZinc constraint on a Hamiltonian circuit, such as `circuit(S)`, can be modelled for MiniCP upon designing a propagator for a `Circuit` predicate:

**Definition (Laurière, 1978)**

A `Circuit([s_1, \ldots, s_n])` constraint holds if and only if the arcs $i \rightarrow s_i$ form a Hamiltonian circuit in the graph defined by the domains of the variables $s_i$: each vertex is visited exactly once.

It is hard to reify `Circuit`. In Part 5, we write a propagator.
Outline

1. Constraint Programming (CP)
2. MiniZinc to MiniCP
3. Sum
4. Element
5. AllDifferent
6. Cumulative
7. Disjunctive
8. Circuit
9. Table
A MiniZinc constraint on membership of a 1d array among the rows of a 2d array, such as $\text{table}(X, T)$, is modelled for MiniCP upon designing a propagator for a $\text{Table}$ predicate:

**Definition**

A Table($[x_1, \ldots, x_n], [[t_{11}, \ldots, t_{1n}], \ldots, [t_{m1}, \ldots, t_{mn}]]$) constraint holds if and only if the values taken by the sequence $[x_1, \ldots, x_n]$ of variables form a row $[t_{i1}, \ldots, t_{in}]$ of the 2d table of parameters given as second argument.

It is easy to reify Table. In Part 7, we design a propagator.