Topic 12: CP and the MiniCP Solver
(Version of 23rd October 2023)

Pierre Flener

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

Reminder from Topic 1: Introduction

A solving technology offers languages, methods, and tools for:

what: **Modelling** constraint problems in a 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 and data as input and tries to solve that problem instance.

Constraint Programming Technology

Constraint programming (CP) offers languages, methods, and tools for:

what: Modelling constraint problems in a high-level declarative 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, called a propagator.

Inference for *One* Constraint: Propagator

Example

Consider the constraint CONNECTED($[C_1, \ldots, C_n]$), which imposes maximum one stretch per colour among the n variables, whose domain is a colour set.
From the following current partial valuation for $n = 6$,
which we assume to be reached upon the search decision $C_4 = \text{red}$:

a propagator of CONNECTED can infer that $C_3 = \text{red}$:

☞ A propagator deletes ~~the~~ impossible values from the current domains of its variables, and thereby accelerates otherwise blind search. It is subsumed when its constraint becomes certainly true under the new current domains.

Roadmap

For CP by systematic search:

- **Consistency:** A consistency (Module 3) is the targeted characterisation of the domain values (Module 2) kept by a propagator (a musician; also known as a 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 at what time (Module 2).
- **Search:** The DFSearch algorithm (of the conductor) calls fixPoint and a branching scheme (Modules 1, 2, and 9).
- **Propagators:** We design propagators for `Sum` and `Element` (Module 3), `Table` (Module 4), `AllDifferent` (Module 5), `Circuit` (Module 6), `Cumulative` (Module 7), and `Disjunctive` (Module 8).

For CP by local search (LS):

- Large-neighbourhood search: hybrid of LS and CP (Module 6).

Outline

Mind the Gap

- MiniCP is a white-box, bottom-up, and 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 declared for each variable.
  In CP solvers, a domain is a dynamically shrinking data structure for a variable, initialised to its declared domain.
- 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 <-> γ(...)` with `var bool: b`, 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 variable `b`, which is a `0 / 1` variable in MiniCP:

- When `b` gets fixed to `1`, post the constraint $\gamma(\dots)$.
- When `b` gets fixed to `0`, post the constraint $\overline{\gamma}(\dots)$.
- When $\gamma(\dots)$ gets subsumed, post the constraint `b=1`.
- When $\overline{\gamma}(\dots)$ gets subsumed, post the constraint `b=0`.

where $\overline{\gamma}(\dots)$ denotes the complement of $\gamma(\dots)$, not code for `not` $\gamma(\dots)$, as CP solvers do not implement `not`.

Propagation may be very poor! Reification may be hard for some predicates!

Constraint combination with reification:

With reification, constraints can be arbitrarily combined with logical connectives: negation ($\neg$), disjunction ($\vee$), conjunction ($\&$), implication ($\Rightarrow$), and equivalence ($\Leftrightarrow$). However, propagation may be very poor!

Example

The composite constraint $(\gamma_1 \ \& \ \gamma_2) \vee \gamma_3$ is modelled as

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

Hence even the constraints γ_1 and γ_2 must be reified.
If γ_1 is $x = y + 1$ and γ_2 is $y = x + 1$, then $\gamma_1 \ \& \ \gamma_2$ is unsatisfiable; however, b is then not fixed to 0 by propagation, as each propagator works individually and there is no communication through their shared variables x and y; hence $b_3 = 1$ is not propagated and γ_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` θ `then` ϕ_1 `else` ϕ_2 `endif`) in MiniZinc often makes the solving slow?

Example

The MiniZinc disjunctive constraint

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

is modelled for MiniCP with reification:

$$(b_0 \Leftrightarrow x = 0) \ \& \ (b_9 \Leftrightarrow x = 9) \ \& \ (b_0 + b_9 \geq 1)$$

But it is logically equivalent to

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

where no reification is involved and no further propagation is needed.

Remember the strong warning in Topic 2: Basic Modelling about a conditional `if` θ `then` ϕ_1 `else` ϕ_2 `endif` or a comprehension, such as `[i | i in` ρ `where` θ`]`, in MiniZinc with a test θ that depends on variables?

Example

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

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

Recall that this is syntactic sugar for

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

This is modelled for MiniCP with reification, as in

```
constraint 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 because both have variables.

Inference: Propagator and Consistency

A MiniZinc inference annotation (recall Topic 8: Inference & Search in CP & LCG) to a constraint, either `value_propagation` or `bounds_propagation` or `domain_propagation`, is prescribed for MiniCP upon designing for the predicate of that constraint a propagator that enforces that consistency.

Example

We design propagators that enforce various consistencies, even others than bounds and domain consistency (Module 3), for `Sum` and `Element` (Module 3), `Table` (Module 4), `AllDifferent` (Module 5), `Circuit` (Module 6), `Cumulative` (Module 7), and `Disjunctive` (Module 8).

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 (Modules 1, 2, and 9) 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

A MiniZinc linear constraint, such as the linear equality `constraint sum(i in 1..n)(A[i] * X[i]) = d`, can be modelled for MiniCP upon designing a propagator for a new `Sum` predicate:

Definition

The $\texttt{Sum}([a_1, \ldots, a_n], [x_1, \ldots, x_n], \sim, d)$ constraint, with

- $[a_1, \ldots, a_n]$ a sequence of non-zero integer parameters,
- $[x_1, \ldots, x_n]$ a sequence of integer variables,
- $\sim$ in $\{<, \leq, =, \neq, \geq, >\}$, and
- d an integer parameter,

holds if and only if the linear relation $\left(\sum_{i=1}^{n} a_i \cdot x_i\right) \sim d$ holds.

It is easy to reify `Sum`.

In Module 3, we design a polynomial-time propagator that enforces bounds consistency for $\sum_{i=1}^{n} x_i = 0$, whose domain consistency is NP-hard to enforce.

Outline

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

Definition (Van Hentenryck and Carillon, 1988)

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

One can generalise `Element` to multi-dimensional arrays. It is hard to reify it.

In Module 3, 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[s]` variables to their non-mutually redundant `Open[w]` variables:

```
constraint forall(s in Shops)
    (Open[Supplier[s]] = 1);
```

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

```
constraint forall(s in Shops)
    (element(Supplier[s], Open, 1));
```

Example (Warehouse Location Problem, a last time)

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

```
solve minimize maintCost * sum(Open)
  + sum(s in Shops)(SupplyCost[s,Supplier[s]]);
```

This must be modelled for MiniCP as in the following MiniZinc reformulation, by declaring `Cost[s]` variables and posting `element` constraints for them:

```
% Cost[s] = the actually incurred supply cost for s:
array[Shops] of var 0..max(SupplyCost): Cost;
constraint forall(s in Shops)
  (element(Supplier[s], SupplyCost[s,..], Cost[s]);
solve minimize maintCost * sum(Open) + sum(Cost);
```

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 array[Apps] of 0..1000: Salary;  % Salary[a] = cost per job to appl. a
2 array[Jobs] of var Apps: Worker; % Worker[j] = appl. allocated job j
3 solve minimize sum(j in Jobs)(Salary[Worker[j]]);
4 constraint ...;        % qualifications, workload, etc
```

Line 3 must be modelled for MiniCP as in the following MiniZinc reformulation, by declaring `Cost[j]` variables and posting `element` constraints for them:

```
array[Jobs] of var 0..max(Salary): Cost; % Cost[j] = salary for job j
constraint forall(j in Jobs)(element(Worker[j],Salary,Cost[j]));
solve minimize sum(Cost);
```

Outline

A MiniZinc constraint on membership of a 1d array among the rows of a 2d array, such as `table(X,T)`, is modelled for MiniCP upon designing a propagator for a new `Table` predicate:

Definition

The `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 Module 4, we design a propagator that enforces domain consistency.

Outline

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

Definition (Laurière, 1978)

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

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

$$\forall i, j \in 1..n \ \textbf{where} \ \ i < j : x_i \neq x_j$$

It is hard to reify `AllDifferent`.

In Module 5, 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

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

Definition (Laurière, 1978)

The `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 Module 6, we design a propagator.

Outline

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

Definition (Aggoun and Beldiceanu, 1993)

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

It is hard to reify `Cumulative`.

In Module 7, we design several propagators that enforce various consistencies on the starting-time variables s_i, when all other arguments are parameters.

Outline

A MiniZinc temporal no-overlap constraint on tasks, such as `disjunctive(S,D)`, can be modelled for MiniCP upon designing a propagator for a new `Disjunctive` predicate:

Definition (Carlier, 1982)

The `Disjunctive`$([s_1, \ldots, s_n], [d_1 \ldots, d_n])$ constraint, where each task T_i has the starting time s_i and 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 Module 8, we design several propagators that enforce various consistencies on the starting-time variables s_i, when the durations d_i are parameters.