Topic 12: CP and Gecode
(Version of 1st December 2018)

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 Gecode
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. `channel`
12. `precede`

UPPSALA UNIVERSITET

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. `channel`
12. `precede`

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 as input and tries to solve the modelled problem.

UPPSALA UNIVERSITET

Constraint Programming (CP)

MiniZinc to Gecode

`linear`

`element`

MiniModel

`distinct`, `nvalues`, `count`

`binpacking`

`cumulative`, `unary`

`circuit`, `path`

`extensional`

`channel`

`precede`

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 user-given strategies
- or local search guided by user-given (meta-)heuristics
- or hybrid search

plus inference, called propagation, but little relaxation.

Slogan of CP:

Constraint Program = Model [+ Search]

CP Solving = Propagation + Search

A CP solver conducts search interleaved with propagation:

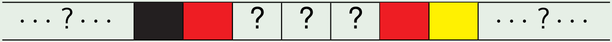
Each constraint has a propagator.

Propagation of one Constraint: Propagator

Example

Consider the constraint CONNECTED([$C_1, \ldots, C_n$]), which enforces max one stretch per colour among the n variables.

From



the CONNECTED([$C_1, \ldots, C_n$]) constraint infers

☞ Propagation is the elimination of ~~the~~ impossible values from the current domains of the variables, and thereby accelerates otherwise blind search.

Outline

1. Constraint Programming (CP)
2. **MiniZinc to Gecode**
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. `channel`
12. `precede`

Procedural vs Declarative

With Gecode, which is a C++ library, one writes an imperative program that states (or: posts) — via sequential, conditional, iterative, or recursive composition — the declarative constraints, which are then given to the solver via propagators achieving chosen consistencies.

Reification

A MiniZinc reified constraint, such as `r <-> x < y`, where `r` is a variable of type `bool`, is modelled in Gecode by appending the reifying variable `r`, of type `Reify`, as an additional argument to the used constraint predicate:

```
rel(x,IRT_LE,y,r)
```

Careful: Not all constraints are reifiable, as in all CP solvers!

We will use the following definition and notation:

Definition

The reification of a constraint $\gamma(\dots)$ is the constraint $r \Leftrightarrow \gamma(\dots)$, where r is a "Boolean" variable, with the truth of $\gamma(\dots)$ represented by 1 and its falsity by 0.

Propagation may be poor: ☞ see Topic 16: Propagators.

Inference: Propagator and Consistency

A MiniZinc inference annotation (recall Topic 8: Inference & Search in CP & LCG) to a constraint, `bounds` or `domain`, is modelled in Gecode by appending the consistency as an additional argument to the used constraint predicate.

The options for integers are value consistency (`IPL_VAL`), bounds consistency (`IPL_BND`), and domain consistency (`IPL_DOM`), consistency being called integer propagation level (IPL), one of them being the default (`IPL_DEF`) in case no consistency is given.

For example:

```
distinct(X,IPL_DOM)
```

☞ For details, see Topic 13: Consistency.

Search: Variable/Value 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 modelled in Gecode by specifying or writing a brancher.

For example:

```
branch(X,INT_VAR_SIZE_MIN(),INT_VAL_MIN())
```

☞ For details, see Topic 15: Search.

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. `channel`
12. `precede`

The `linear` Predicate

A MiniZinc linear constraint, such as the linear equality constraint `sum(i in 1..n)(A[i]*X[i]) = d`, can be modelled in Gecode by using its reifiable `linear` predicate:

Definition

A `linear`$([a_1, \ldots, a_n], [x_1, \ldots, x_n], R, d)$ 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 $\{<, \leq, =, \neq, \geq, >\}$, and
- d an integer constant,

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

Also, `linear`$([x_1, \ldots, x_n], R, d)$ holds iff $\left(\sum_{i=1}^{n} x_i\right) \ R \ d$.

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. linear
4. element
5. MiniModel
6. distinct, nvalues, count
7. binpacking
8. cumulative, unary
9. circuit, path
10. extensional
11. channel
12. precede

The `element` Predicate

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 involving the expression `X[i]`, must be modelled in Gecode by explicitly using its non-reifiable `element` predicate:

Definition (Van Hentenryck and Carillon, 1988)

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

Several variants exist: see the Gecode documentation.

Example (Warehouse Location Problem)

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

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

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

```
array[Shops] of var int: Cost; % Cost[s] to supply s
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 int: Salary;
2 array[Jobs] of var Apps: Worker; % job j by Worker[j]
3 solve minimize sum(j in Jobs)(Salary[Worker[j]]);
4 constraint ...;          % qualifications, workload, etc
```

Line 3 must be modelled in Gecode as in the following MiniZinc reformulation, by explicitly creating a `Cost[j]` variable and an `element` constraint for each implicit one:

```
array[Jobs] of var int: Cost; % job j costs Cost[j]
constraint forall(j in Jobs)
   (element(Worker[j], Salary, Cost[j]));
solve minimize sum(j in Jobs)(Cost[j]); % sum(Cost)
```

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. `linear`
4. `element`
5. **MiniModel**
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. `channel`
12. `precede`

MiniModel

- Using MiniModel, linear constraints can be formulated in Gecode like in MiniZinc: the appropriate `linear` constraints are generated by the Gecode toolchain. Another useful feature will be discussed at page 36.

- Gecode has no constrained functions: everything is modelled relationally, using only constraint predicates.

- Gecode does not eliminate common sub-expressions: a Gecode model automatically generated by the MiniZinc toolchain can outperform a handwritten Gecode model corresponding to the MiniZinc one.

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. linear
4. element
5. MiniModel
6. distinct, nvalues, count
7. binpacking
8. cumulative, unary
9. circuit, path
10. extensional
11. channel
12. precede

The `distinct` Predicate

A MiniZinc constraint of pairwise difference, such as `alldifferent(X)`, can be modelled in Gecode by using its non-reifiable `distinct` predicate:

Definition (Laurière, 1978)

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

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

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

Several variants exist: see the Gecode documentation.

The `nvalues` Predicate

A MiniZinc constraint on the number of distinct values within an array, such as `nvalue(m,X)`, can be modelled in Gecode by using its non-reifiable `nvalues` predicate:

Definition (Pachet and Roy, 1999)

An `nvalues`$([x_1, \ldots, x_n], R, m)$ constraint holds if and only if the number of distinct values taken by the elements of the sequence $[x_1, \ldots, x_n]$ of variables is in relation R with the variable m, where $R \in \{<, \leq, =, \neq, \geq, >\}$:

$$|\{x_1, \ldots, x_n\}| \;\; R \; m$$

Note that R is '$=$' for the `nvalue` predicate of MiniZinc.

Several variants exist: see the Gecode documentation.

The `count` Predicate

A MiniZinc constraint on value counts within an array, such as `global_cardinality(X,V,C)`, can be modelled in Gecode by using its non-reifiable `count` predicate:

Definition (Régin, 1996)

A `count`$([x_1, \ldots, x_n], [v_1, \ldots, v_m], [c_1, \ldots, c_m])$ constraint holds if and only if each variable c_j has the number of variables x_i that take the given value v_j.

Several variants exist: see the Gecode documentation.

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. **`binpacking`**
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. `channel`
12. `precede`

The `binpacking` Predicate

A MiniZinc bin-packing constraint, such as `bin_packing_load(L,B,V)`, can be modelled in Gecode by using its non-reifiable `binpacking` predicate:

Definition

Let item i have the given weight or volume v_i.
Let variable b_i denote the bin into which item i is put.
Let variable ℓ_j denote the load of bin j.
A `binpacking`$([\ell_1, \ldots, \ell_m], [b_1, \ldots, b_n], [v_1, \ldots, v_n])$ constraint holds iff each ℓ_j is the sum of the v_i where $b_i = j$.

There is No Knapsack Predicate in Gecode

A MiniZinc constraint on knapsack packing, such as `knapsack(V,P,X,v,p)`, can be modelled in Gecode by using two `linear` constraints:

$$\texttt{linear}(V, X, =, v)$$

$$\texttt{linear}(P, X, =, p)$$

Recall that `linear` is reifiable.

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. linear
4. element
5. MiniModel
6. distinct, nvalues, count
7. binpacking
8. **cumulative, unary**
9. circuit, path
10. extensional
11. channel
12. precede

The `cumulative` Predicate

A MiniZinc constraint on the bounded cumulative resource requirement of tasks, such as `cumulative(S,D,R,u)`, can be modelled in Gecode by using its non-reifiable `cumulative` predicate:

Definition (Aggoun and Beldiceanu, 1993)

A $\texttt{cumulative}(u, [s_1, \ldots, s_n], [d_1, \ldots, d_n], [r_1, \ldots, r_n])$ 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.

Several variants exist: see the Gecode documentation.

The `unary` Predicate

A MiniZinc temporal non-overlap constraint on tasks, such as `disjunctive(S,D)`, can be modelled in Gecode by using its non-reifiable `unary` predicate, so called because it applies to tasks requiring a unary resource:

Definition (Carlier, 1982)

A `unary`$([s_1, \dots, s_n], [d_1 \dots, d_n])$ 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.

Several variants exist: see the Gecode documentation.

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. `channel`
12. `precede`

The `circuit` Predicate

A MiniZinc constraint on a Hamiltonian circuit, such as `circuit(S)`, can be modelled in Gecode by using its non-reifiable `circuit` predicate:

Definition (Laurière, 1978)

A `circuit`($[s_1, \ldots, s_n]$) constraint holds iff 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.

Several variants exist: see the Gecode documentation.

UPPSALA UNIVERSITET

Constraint Programming (CP)
MiniZinc to Gecode
linear
element
MiniModel
distinct, nvalues, count
binpacking
cumulative, unary
circuit, path
extensional
channel
precede

No Subcircuit, but a Path Predicate

A MiniZinc constraint `subcircuit(S)` can be modelled in Gecode as in the MiniZinc default definition at http://www.minizinc.org/doc-lib/doc-globals.html, which is actually used by the Gecode backend to MiniZinc.

A MiniZinc constraint on a Hamiltonian path, such as `circuit(S) /\ S[t] = f`, can be modelled in Gecode by using its non-reifiable `path` predicate:

Definition

A $\texttt{path}([s_1, \ldots, s_n], f, t)$ constraint holds iff the arcs $i \to s_i$ form a Hamiltonian path from vertex f to vertex t in the graph defined by the domains of the variables s_i: each vertex is visited exactly once.

Several variants exist: see the Gecode documentation.

UPPSALA UNIVERSITET

Constraint Programming (CP)
MiniZinc to Gecode
linear
element
MiniModel
distinct, nvalues, count
binpacking
cumulative, unary
circuit, path
extensional
channel
precede

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. **`extensional`**
11. `channel`
12. `precede`

The `extensional` Predicate

A MiniZinc constraint on membership in a table `T` or regular language, such as `table(X,T)` or `regular(X,R)`, where `R` is a regular expression or a deterministic finite automaton (DFA) defining a regular language, is modelled in Gecode by using its reifiable `extensional` predicate:

Definition

An `extensional`$([x_1, \ldots, x_n], \mathcal{R})$ constraint holds if and only if the values taken by the sequence $[x_1, \ldots, x_n]$ of variables form a row of the 2d table $\mathcal{R}$ of constants or form a string that belongs to the regular language accepted by the regular expression (when using MiniModel) or DFA $\mathcal{R}$.

Several variants exist: see the Gecode documentation.

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. **`channel`**
12. `precede`

The `channel` Predicate

A MiniZinc constraint on two arrays representing a function and its inverse, such as `inverse(X,Y)`, can be modelled in Gecode by using its non-reifiable `channel` predicate:

Definition

A `channel`$([x_1, \ldots, x_n], [y_1, \ldots, y_n])$ constraint holds iff:

$$\forall i, j \in 1..n : x_i = j \Leftrightarrow y_j = i$$

Several variants exist: see the Gecode documentation.

Outline

1. Constraint Programming (CP)
2. MiniZinc to Gecode
3. `linear`
4. `element`
5. MiniModel
6. `distinct`, `nvalues`, `count`
7. `binpacking`
8. `cumulative`, `unary`
9. `circuit`, `path`
10. `extensional`
11. `channel`
12. **`precede`**

The `precede` Predicate

MiniZinc value symmetry-breaking constraints, such as `value_precede(v,w,X)` and its generalisation `value_precede_chain(V,X)`, can be modelled in Gecode by using its non-reifiable `precede` predicate:

Definition

A `precede`$([x_1, \ldots, x_n], v, w)$ constraint holds iff the first occurrence, if any, of value v precedes the first occurrence, if any, of value w among the variables x_i.

Definition

A `precede`$([x_1, \ldots, x_n], [v_1, \ldots, v_m])$ constraint holds iff the first occurrence, if any, of every value v_i precedes the first occurrence, if any, of value v_{i+1} among the variables x_i.