

Tutorial on Gecode Constraint Programming

Combinatorial Problem Solving (CPS)

Enric Rodríguez-Carbonell

March 10, 2022

Gecode

- Gecode is environment for developing constraint-programming based progs
 - ◆ open source: extensible, easily interfaced to other systems
 - ◆ free: distributed under MIT license
 - ◆ portable: rigidly follows the C++ standard
 - ◆ accessible: comes with a manual and other supplementary materials
 - ◆ efficient: very good results at competitions, e.g. MiniZinc Challenge
- Developed by C. Schulte, G. Tack and M. Lagerkvist
- Available at: <http://www.gecode.org>

Basics

- Gecode is a set of C++ libraries
- Models (= CSP's in this context) are C++ programs that must be compiled with Gecode libraries and executed to get a solution
- Models are implemented using spaces, where variables, constraints, etc. live
- Models are derived classes from the base class Space. The constructor of the derived class
 - ◆ declares the CP variables and their domains,
 - ◆ posts the constraints, and
 - ◆ specifies how the search is to be conducted.
- For the search to work, a model must also implement:
 - ◆ a copy constructor, and
 - ◆ a copy function

Example

- Find different digits for the letters *S, E, N, D, M, O, R, Y* such that equation *SEND+MORE=MONEY* holds and there are no leading 0's
- Code of this example available at <http://www.cs.upc.edu/~erodri/cps.html>

```
// To use integer variables and constraints
#include <gecode/int.hh>

// To make modeling more comfortable
#include <gecode/minimodel.hh>

// To use search engines
#include <gecode/search.hh>

// To avoid typing Gecode:: all the time
using namespace Gecode;
```

Example

```
class SendMoreMoney : public Space {  
protected:  
    IntVarArray x;  
  
public:                                     // *this is called 'home space'  
  
    SendMoreMoney() : x(*this, 8, 0, 9) {  
  
        IntVar s(x[0]), e(x[1]), n(x[2]), d(x[3]),  
                m(x[4]), o(x[5]), r(x[6]), y(x[7]);  
  
        rel(*this, s != 0);  
        rel(*this, m != 0);  
        distinct(*this, x);  
        rel(*this, 1000*s + 100*e + 10*n + d  
                + 1000*m + 100*o + 10*r + e  
        == 10000*m + 1000*o + 100*n + 10*e + y);  
  
        branch(*this, x, INT_VAR_SIZE_MIN(), INT_VAL_MIN());  
    }  
    ...  
}
```

Example

- The model is implemented as class `SendMoreMoney`, which inherits from the class `Space`
- Declares an array `x` of 8 new integer CP variables that can take values from 0 to 9
- To simplify posting the constraints, the constructor defines a variable of type `IntVar` for each letter.
These are **synonyms** of the CP variables, **not new ones!**
- `distinct`: values must be \neq pairwise (aka all-different)
- Variable selection: the one with smallest domain size first (`INT_VAR_SIZE_MIN()`)
- Value selection: the smallest value of the selected variable first (`INT_VAL_MIN()`)

Example

...

```
SendMoreMoney( SendMoreMoney& s )  
: Space(s) {  
    x.update(*this , s.x);  
}
```

```
virtual Space* copy() {  
    return new SendMoreMoney(*this);  
}
```

```
void print() const {  
    std::cout << x << std::endl;  
}
```

```
}; // end of class SendMoreMoney
```

Example

- The copy constructor must call the copy constructor of `Space` and then copy the rest of members (those with CP variables by calling `update`)

In this example this amounts to invoking `Space(s)`
and updating the variable array `x` with `x.update(*this, s.x);`

- A space must implement an additional `copy()` function that is capable of returning a fresh copy of the model during search.

Here it uses copy constructor: `return new SendMoreMoney(*this);`

- We may have other functions (like `print()` in this example)

Example

```
int main() {  
  
    SendMoreMoney* m = new SendMoreMoney;  
  
    DFS<SendMoreMoney> e(m);  
    delete m;  
  
    while (SendMoreMoney* s = e.next()) {  
        s->print();  
        delete s;  
    }  
  
}
```

Example

■ Let us assume that we want to search for all solutions:

1. create a model and a search engine for that model

(a) create an object of class `SendMoreMoney`

(b) create a search engine `DFS<SendMoreMoney>` (depth-first search) and initialize it with a model.

As the engine takes a clone,
we can immediately delete `m` after the initialization

2. use the search engine to find all solutions

The search engine has a `next()` function that returns the next solution, or `NULL` if no more solutions exist

A solution is again a model (in which domains are single values).
When a search engine returns a model, the user must delete it.

■ To search for a single solution: replace while by if

Compiling and Linking

- Template of Makefile for compiling p.cpp and linking:

```
CXX    = g++ -std=c++11
DIR    = /usr/local
LIBS    = -lgecodedriver      -lgecodesearch  \
          -lgecodeminimodel  -lgecodeint     \
          -lgecodekernel     -lgecodesupport

p:  p.cpp
    $(CXX) -I$(DIR)/include -c p.cpp
    $(CXX) -L$(DIR)/lib -o p p.o $(LIBS)
```

Executing

- Gecode is installed as a set of **shared** libraries
- Environment variable `LD_LIBRARY_PATH` has to be set to include `<dir>/lib`, where `<dir>` is installation dir
- E.g., edit file `~/.tcshrc` (create it if needed) and add line

```
setenv LD_LIBRARY_PATH <dir>
```

- In the lab: `<dir>` is `/usr/local/lib`

Optimization Problems

- Find different digits for the letters *S, E, N, D, M, O, T, Y* such that
 - ◆ equation $SEND + MOST = MONEY$ holds
 - ◆ there are no leading 0's
 - ◆ *MONEY* is maximal
- Searching for a best solution requires
 - ◆ a function that **constrains** the search to consider only better solutions
 - ◆ a best solution search engine
- The model differs from *SendMoreMoney* only by:
 - ◆ a new linear equation
 - ◆ an additional **constrain** () function
 - ◆ a different search engine

Optimization Problems

- New linear equation:

```
IntVar s(x[0]), e(x[1]), n(x[2]), d(x[3]),  
        m(x[4]), o(x[5]), t(x[6]), y(x[7]);
```

...

```
rel(*this, 1000*s + 100*e + 10*n + d  
        + 1000*m + 100*o + 10*s + t  
    == 10000*m + 1000*o + 100*n + 10*e + y);
```

Optimization Problems

- `constrain()` function (`_b` is the newly found solution):

```
virtual void constrain(const Space& _b) {  
  
    const SendMostMoney& b =  
        static_cast<const SendMostMoney&>(_b);  
  
    IntVar e(x[1]), n(x[2]), m(x[4]), o(x[5]), y(x[7]);  
  
    IntVar b_e(b.x[1]), b_n(b.x[2]), b_m(b.x[4]),  
           b_o(b.x[5]), b_y(b.x[7]);  
  
    int money = (10000*b_m.val()+1000*b_o.val()  
                +100*b_n.val()+ 10*b_e.val()+b_y.val());  
  
    rel(*this, 10000*m + 1000*o + 100*n + 10*e + y > money);  
}
```

Optimization Problems

- The main function now uses a **branch-and-bound** search engine rather than plain depth-first search:

```
SendMostMoney* m = new SendMostMoney ;  
BAB<SendMostMoney> e(m);  
delete m;
```

- The loop that iterates over the solutions found by the search engine is the same as before:
solutions are found with an increasing value of *MONEY*

Variables

- Integer variables are instances of the class `IntVar`
- Boolean variables are instances of the class `BoolVar`
- There exist also
 - ◆ `FloatVar` for floating-point variables
 - ◆ `SetVar` for integer set variables

(but we will not use them; see the reference documentation for more info)

Creating Variables

- An `IntVar` variable points to a **variable implementation** (= a CP variable). The same CP variable can be referred to by many `IntVar` variables
- New CP integer variables are created with a constructor:

```
IntVar x(home, l, u);
```

This:

- ◆ declares a **program** variable `x` of type `IntVar` in the space `home`
 - ◆ creates a new integer **CP** variable with domain $l, l + 1, \dots, u - 1, u$
 - ◆ makes `x` point to the newly created CP variable
- Domains can also be specified with an integer set `IntSet`:

```
IntVar x(home, IntSet{0, 2, 4});
```

Creating Variables

- The default constructor and the copy constructor of an `IntVar` **do not** create a new variable implementation
- Default constructor:
the variable doesn't refer to any variable implementation (it dangles)
- Copy constructor:
the variable refers to the same variable implementation

```
IntVar x(home, 1, 4);  
IntVar y(x);
```

`x` and `y` refer to the same variable implementation (they are synonyms)

Creating Variables

- Domains of integer vars must be included in
[`Int :: Limits :: min`, `Int :: Limits :: max`] (implementation-dependent constants)
- Typically `Int :: Limits :: max` = 2147483646 ($= 2^{31} - 2$),
`Int :: Limits :: min` = $- \text{Int :: Limits :: max}$
- Example of creation of a Boolean variable:

```
BoolVar x(home, 0, 1);
```

Note that the lower and upper bounds must be passed **even it is Boolean!**

Operations with Variables

- Min/max value in the current domain of a variable `x`: `x.min()` / `x.max()`
- To find out if a variable has been assigned: `x.assigned()`
- Value of the variable, if already assigned: `x.val()`
- To print the domain of a variable: `cout << x`
- To make a copy of a variable (e.g., for the copy constructor of the model):
`update`

E.g. in

```
x.update(home, y);
```

variable `x` is assigned a copy of variable `y`

Arrays of Variables

- Integer variable arrays `IntVarArray` have similar functions to integer vars
- For example,

```
IntVarArray x(home, 4, -10, 10);
```

creates a new array with 4 variables containing newly created CP variables with domain $\{-10, \dots, 10\}$.

- `x.assigned()` returns if all variables in the array are assigned
- `x.size()` returns the size of the array
- For making copies `update` works as with integer variables

Argument Arrays

- Gecode provides **argument arrays** to be passed as arguments in functions that post constraints
 - ◆ **IntArgs** for integers
 - ◆ **IntVarArgs** for integer variables
 - ◆ **BoolVarArgs** for Boolean variables

Argument Arrays

For example:

```
IntVar s(x[0]), e(x[1]), n(x[2]), d(x[3]),  
        m(x[4]), o(x[5]), r(x[6]), y(x[7]);  
...  
IntArgs c(4+4+5); IntVarArgs z(4+4+5);  
c[0]= 1000; c[1]= 100; c[2]= 10; c[3]= 1;  
z[0]= s;    z[1]= e;    z[2]= n;  z[3]= d;  
  
c[4]= 1000; c[5]= 100; c[6]= 10; c[7]= 1;  
z[4]= m;    z[5]= o;    z[6]= r;  z[7]= e;  
  
c[8]= -10000; c[9]= -1000; c[10]= -100; c[11]= -10; c[12]= -1;  
z[8]= m;      z[9]= o;      z[10]= n;    z[11]= e;    z[12]= y;  
  
linear(*this, c, z, IRT_EQ, 0); // c.z = 0, where . is dot product
```


Argument Arrays

Or equivalently:

```
IntVar s(x[0]), e(x[1]), n(x[2]), d(x[3]),  
        m(x[4]), o(x[5]), r(x[6]), y(x[7]);
```

...

```
IntArgs c({  
    1000, 100, 10, 1,  
    1000, 100, 10, 1,  
    -10000, -1000, -100, -10, -1});
```

```
IntVarArgs z({  
    s, e, n, d,  
    m, o, r, e,  
    m, o, n, e, y});
```

```
linear(*this, c, z, IRT_EQ, 0);
```

Argument Arrays

- Integer argument arrays with simple sequences of integers can be generated using `IntArgs::create(n, start, inc)`
 - ◆ `n` is the length of the array
 - ◆ `start` is the starting value
 - ◆ `inc` is the increment from one value to the next (default: 1)

```
IntArgs::create(5,0)      // creates 0,1,2,3,4
IntArgs::create(5,4,-1)   // creates 4,3,2,1,0
IntArgs::create(3,2,0)    // creates 2,2,2
IntArgs::create(6,2,2)    // creates 2,4,6,8,10,12
```

Posting Constraints

- Next: focus on constraints for integer/Boolean variables
- We will see the most basic functions for posting constraints.
(post functions)

Look up the documentation for more info.

Relation Constraints

- **Relation constraints** are of the form $E_1 \bowtie E_2$,
where E_1, E_2 are integer/Boolean expressions, \bowtie is a relation operator
- Integer expressions are built up from:
 - ◆ arithmetic operators: $+$, $-$, $*$, $/$, $\%$
 - ◆ integer values
 - ◆ integer/Boolean variables
 - ◆ `sum(x)`: sum of the array `x`
 - ◆ `sum(c,x)`: weighted sum (dot product)
 - ◆ `min(x)`: min of the array `x`
 - ◆ `max(x)`: max of the array `x`
 - ◆ `element(x, i)`: the `i`-th element of the array `x`
 - ◆ ...

Relation Constraints

- Relations between integer expressions are:
 $==, !=, <=, <, >=, >$
- Relation constraints are posted with function `rel`

```
rel(home, x+2*sum(z) < 4*y);  
rel(home, a+b*(c+d) == 0);
```

Relation Constraints

- Boolean expressions are built up from:
 - ◆ Boolean variables
 - ◆ `element(x, i)`: the `i`-th element of the Boolean array `x`
 - ◆ integer relations
 - ◆ `!:` negation
 - ◆ `&&:` conjunction
 - ◆ `||:` disjunction
 - ◆ `==:` equivalence
 - ◆ `>>:` implication

Relation Constraints

■ Examples:

```
rel(home, x && (y >> z));  
rel(home, !(x && (y >> z)));  
rel(home, (st1+1 <= st2) || (st2+1 <= st1));
```

Relation Constraints

- An alternative less comfortable interface:

`rel (home, E_1 , \bowtie , E_2);` where \bowtie for integer relations may be:

- ◆ `IRT_EQ`: equal
- ◆ `IRT_NQ`: different
- ◆ `IRT_GR`: greater than
- ◆ `IRT_GQ`: greater than or equal
- ◆ `IRT_LE`: less than
- ◆ `IRT_LQ`: less than or equal

and for Boolean relations is one of:

- ◆ `BOT_AND`: conjunction
- ◆ `BOT_OR`: disjunction
- ◆ `BOT_EQV`: equivalence
- ◆ `BOT_IMP`: implication
- ◆ ...

Relation Constraints

Here x , y are arrays of integer variables, z an integer variable

- $\text{rel}(\text{home}, x, \text{IRT_LQ}, z)$: all vars in x are $\leq z$
- $\text{rel}(\text{home}, x, \text{IRT_LE}, y)$: x is lexicographically smaller than y
- $\text{linear}(\text{home}, a, x, \bowtie, z)$: $a^T x \bowtie z$
- $\text{linear}(\text{home}, x, \bowtie, z)$: $\sum x_i \bowtie z$
- ...

Distinct Constraint

- `distinct (home, x)` enforces that integer variables in array `x` take pairwise distinct values (aka `alldifferent`)

```
IntVarArray x(home, 10, 1, 10);  
distinct(home, x);
```

- `distinct (home, c, x)`; for an array `c` of type `IntArgs` and an array of integer variables `x` of same size, constrains the variables in `x` such that

$$x_i + c_i \neq x_j + c_j$$

for $0 \leq i < j < |x|$

Channel Constraints

■ Channel constraints link

integer to Boolean variables, and integer variables to integer variables.

For example:

- ◆ For Boolean variable array x and integer variable y , `channel(home, x, y)` posts $x_i = 1 \leftrightarrow y = i$ for $0 \leq i < |x|$
- ◆ For two integer variable arrays x and y of same size, `channel(home, x, y)` posts $x_i = j \leftrightarrow y_j = i$ for $0 \leq i, j < |x|$

Reified Constraints

- Some constraints have **reified** variants:
satisfaction is monitored by a Boolean variable (**indicator/control variable**)

When allowed, the control variable is passed as a last argument: e.g.,

```
rel(home, x == y, b);
```

posts $b = 1 \Leftrightarrow x = y$,

where x , y are integer variables and b is a Boolean variable

Reified Constraints

- Instead of **full** reification, we can post **half** reification: only one direction of the equivalence
- Functions `eqv`, `imp`, `pmi` take a Boolean variable and return an object that specifies the reification:

```
rel(home, x == y, eqv(b)); // b = 1 ⇔ x = y
rel(home, x == y, imp(b)); // b = 1 ⇒ x = y
rel(home, x == y, pmi(b)); // b = 1 ⇐ x = y
```

Hence passing `eqv(b)` is equivalent to passing `b`

Propagators

- For many constraints, Gecode provides **different propagators** with different pruning power
- Post functions take an optional argument that specifies the propagator
- Possible values:
 - ◆ **IPL_DOM**: perform **domain propagation**.
Sometimes **domain consistency** (i.e., arc consistency) is achieved.
 - ◆ **IPL_BND**: perform **bounds propagation**.
Sometimes bounds consistency is achieved
 - ◆ ...
 - ◆ **IPL_DEF**: default of the constraint (check reference documentation)
- Different propagators have different tradeoffs of cost/pruning power.

Branching

- Gecode offers predefined **variable-value branching**:
when calling `branch(home, x, ?, ?)` for branching on array of integer vars `x`,
 - ◆ 3rd arg defines the heuristic for selecting the **variable**
 - ◆ 4th arg defines the heuristic for selecting the **values**
- E.g. for an array of integer vars `x` the following call

```
branch(home, x, INT_VAR_MIN_MIN(), INT_VAL_SPLIT_MIN());
```

- ◆ selects the var `y` with smallest min value in the domain (if tie, the 1st)
- ◆ creates a choice with two alternatives $y \leq n$ and $y > n$ where

$$n = \frac{\min(y) + \max(y)}{2}$$

and chooses $y \leq n$ first

Integer Variable Selection

Some of the predefined strategies:

- `INT_VAR_NONE()`: first unassigned
- `INT_VAR_RND(r)`: randomly, with random number generator `r`
- `INT_VAR_DEGREE_MIN()`: smallest degree
- `INT_VAR_DEGREE_MAX()`: largest degree
- `INT_VAR_SIZE_MIN()`: smallest domain size
- `INT_VAR_SIZE_MAX()`: largest domain size
- ...

Boolean Variable Selection

Some of the predefined strategies:

- `BOOL_VAR_NONE()`: first unassigned
- `BOOL_VAR_RND(r)`: randomly, with random number generator `r`
- `BOOL_VAR_DEGREE_MIN()`: smallest degree
- `BOOL_VAR_DEGREE_MAX()`: largest degree
- ...

Integer Value Selection

Some of the predefined strategies:

- `INT_VAL_RND(r)`: random value
- `INT_VAL_MIN()`: smallest value
- `INT_VAL_MAX()`: largest value
- `INT_VAL_SPLIT_MIN()`: values not greater than $\frac{min+max}{2}$
- `INT_VAL_SPLIT_MAX()`: values greater than $\frac{min+max}{2}$
- ...

Boolean Value Selection

Some of the predefined strategies:

- `BOOL_VAL_RND(r)`: random value
- `BOOL_VAL_MIN()`: smallest value
- `BOOL_VAL_MAX()`: largest value
- ...