Tutorial on Gecode Constraint Programming

Combinatorial Problem Solving (CPS)

Enric Rodríguez-Carbonell

April 19, 2017

Gecode

- Gecode is environment for developing constraint-programming based progs
  - open: extensible, easily interfaced to other systems
  - free: distributed under MIT license
  - portable: rigidly follows the C++ standard
  - accessible: comes with tutorial + reference book
  - 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, branchers, etc. live
- Models are derived classes from the base class Space.
  The constructor of the derived class implements the model.
- 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 l;

public:                    // *this is called 'home space'

  SendMoreMoney(void) : l(*this, 8, 0, 9) {

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

    rel(*this, s != 0);
    rel(*this, m != 0);
    distinct(*this, l);
    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,l, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
  }
  ...
```

Example

- The model is implemented as class SendMoreMoney, which inherits from the class Space
- Declares an array I 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(bool share, SendMoreMoney& s)
    : Space(share, s) {
    l.update(*this, share, s.l);
  }

  virtual Space* copy(bool share) {
    return new SendMoreMoney(share,*this);
  }

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

}; // end of class SendMoreMoney
```

Example

- The copy constructor should invoke the copy constructor of the parent class, and copy (update) all data structures that contain variables (ignore boolean argument)

  In this example this amounts to
  invoking Space(share,s) and updating the variable array l

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

  Here it uses copy constructor

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

Example

```
int main(int argc, char* argv[]) {

  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. When a search engine returns a model, the user is responsible of deleting it.

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

Example

- Gecode may throw exceptions when creating vars, etc.
- It is a good practice to catch all these exceptions.
  Wrap the entire body of main into a try statement:

```
int main(int argc, char* argv[]) {
  try {

    SendMoreMoney* m = new SendMoreMoney;

    DFS<SendMoreMoney> e(m);
    delete m;

    while (SendMoreMoney* s = e.next()) {
      s->print();
      delete s;
    }
  }
  catch (Exception e) {
    cerr << "Exception: " << e.what() << endl;
    return 1;
  }
}
```

Compiling and Linking

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

```
DIR=/usr/local
LIBS=
    -lgecodeflatzinc  -lgecodedriver \
    -lgecodegist      -lgecodesearch \
    -lgecodeminimodel -lgecodeset    \
    -lgecodefloat     -lgecodeint    \
    -lgecodekernel    -lgecodesupport

p: p.cpp
        g++ -Wall -I$(DIR)/include -c p.cpp
        g++ -Wall -L$(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(l[0]), e(l[1]), n(l[2]), d(l[3]),
       m(l[4]), o(l[5]), t(l[6]), y(l[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(l[1]), n(l[2]), m(l[4]), o(l[5]), y(l[7]);

 IntVar b_e(b.l[1]), b_n(b.l[2]), b_m(b.l[4]),
        b_o(b.l[5]), b_y(b.l[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 integer variables are created with a constructor:

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

This:

- declares a variable x of type IntVar in the space home
- creates a new integer variable implementation with domain $l, l+1, \ldots, u-1, u$
- makes x point to the newly created variable implementation

- Domains can also be specified by an integer set IntSet:

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

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 cannot exceed the limits of the C++ type `int`
- It always holds that $\mathsf{Int::Limits::min} = -\ \mathsf{Int::Limits::max}$
- Typically $\mathsf{Int::Limits::max} = 2147483646\ (= 2^{31} - 2)$
- Example of creation of a Boolean variable

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

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 on stream out: out << x
- A variable must be updated in the space's copy() function.

  E.g. in

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

  variable x is updated from variable y

Accessing the Domain

```
for (IntVarValues i(x); i(); ++i)
cout << i.val() << ' ';
```

uses the value iterator `i` to print all values of the domain of the integer variable `x` (in increasing order)

- `i()` tests whether there are more values to iterate for `i`
- `++i` moves the iterator `i` to the next value
- `i.val()` returns the current value of the iterator `i`

```
for (IntVarRanges i(x); i(); ++i)
cout << i.min() << ".." << i.max() << ' ';
```

uses the range iterator `i` to print all ranges (intervals of the domain) of the integer variable `x`

Arrays of Variables

- Working with an integer variable array IntVarArray is like with integer vars
- For example,

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

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

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

Argument Arrays

- Gecode provides argument arrays
  to be passed as arguments in functions posting constraints
  - IntArgs for integers
  - IntVarArgs for integer variables
  - BoolVarArgs for Boolean variables
  - ...

Argument Arrays

- Integer argument arrays with simple sequences of integers can be generated using IntArgs :: create(n, start , inc)
  - The n parameter gives the length of the array
  - The start parameter gives the starting value
  - The inc determines the increment from one value to the next

```
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:
  - integer values
  - integer/Boolean variables
  - arithmetic operators: $+$, $-$, $*$, $/$, %
  - sum(x): sum of integer/Boolean vars
  - sum(c,x): weighted sum (dot product)
  - min(x), max(x)
  - abs(x): absolute value of x
  - element(x, i): the i-th element of the array x
  - ite(b,x,y): if-then-else expression
  - ...

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
  - !: negation
  - &&: conjunction
  - ||: disjunction
  - ^: exclusive or
  - ==: equivalence
  - =!: non-equivalence
  - >>: implication
  - <<: reverse implication
  - element(x, i): the i-th element of the Boolean array x
  - integer relations

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_XOR: exclusive or
  - BOT_EQV: equivalence
  - BOT_IMP: implication

Relation Constraints

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

- rel (home, x, IRT_LQ, 7): all vars in x are ≤ 7
- rel (home, x, IRT_LQ, z): all vars in x are $\leq$ z
- rel (home, x, IRT_LQ): x is sorted in increasing order
- rel (home, x, IRT_EQ): vars in x are all equal
- rel (home, x, IRT_NQ): the negation of the previous one
- rel (home, x, IRT_LE, y): x is lexicographically smaller than y
- rel (home, x, IRT_EQ, y): if $|x| = |y|$, equality pointwise
- linear (home, a, x,$\bowtie$, z): $a^T x \bowtie z$
- linear (home, x,$\bowtie$, z): $\sum x_i \bowtie z$

Relation Constraints

Here x, y are arrays of Boolean variables, z a Boolean var

- rel (home, BOT_AND, x, z): $\bigwedge_{i=0}^{|x|-1} x_i = z$
- rel (home, BOT_OR, x, z): $\bigvee_{i=0}^{|x|-1} x_i = z$
- clause (home, BOT_AND, x, y, z): $\left(\bigwedge_{i=0}^{|x|-1} x_i\right) \wedge \left(\bigwedge_{j=0}^{|y|-1} \neg y_j\right) = z$
- clause (home, BOT_OR, x, y, z): $\left(\bigvee_{i=0}^{|x|-1} x_i\right) \vee \left(\bigvee_{j=0}^{|y|-1} \neg y_j\right) = z$

Distinct Constraint

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

```
IntVarArray x(10, 1, 10);
distinct(*this, 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 $0 \leq i, j < |x|$ and $i \neq j$

Membership Constraints

- The membership constraint member(home, x, y)
  for an integer variable array x and an integer variable y
  forces that y is included in x

- x and y can also be Boolean variables

Channel Constraints

- Channel constraints link (channel)
  Boolean to integer variables, and integer variables to integer variables.
  For example:
  - For two integer variable arrays x and y of same size, channel(home, x, y) posts $x_i = j \leftrightarrow y_j = i$ for $0 \leq i, j < |x|$
  - For Boolean variable array x and integer variable y, channel(home, x, y) posts $x_i = 1 \leftrightarrow y = i$ 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 levels of consistency
- Post functions take an optional argument that specifies the propagator
- Possible values:
  - ICL_DOM: perform domain propagation.
    Sometimes domain consistency (i.e., arc consistency) is achieved.
  - ICL_BND: perform bounds propagation.
    Sometimes bounds consistency is achieved
  - ICL_VAL: perform value propagation
  - ICL_DEF: default (check reference documentation)
- Different propagators for the same constraint may have different cost.
  In general, ICL_VAL cheapest, ICL_DOM most expensive

Branching

- Gecode offers predefined variable-value branching: when calling branch() to post a branching,
  - 3rd arg defines the variable selected for branching
  - 4th arg defines the values selected for branching
- E.g. for an array of integer vars x the following call

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

  - selects var y with smallest min value (in case of a tie, the first)
  - 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

- The brancher assigns all vars and dies.
  If more branchers exist, the search continues with the next one.

Branching

- Gecode also supports branching on a single variable.
  Then only value selection must be specified.
- E.g., if `x` is an integer variable of type `IntVar`, then

```
branch(home, x, INT_VAL_MAX());
```

  branches on `x` by first trying the largest value of its domain
- In Gecode other branchers can also be programmed

Integer/Boolean Variable Selection

Some of the predefined strategies:

- INT_VAR_NONE(): first unassigned
- INT_VAR_RND(r): randomly
- INT_VAR_DEGREE_MIN(): smallest degree
- INT_VAR_DEGREE_MAX(): largest degree
- INT_VAR_MIN_MIN(): smallest minimum value
- INT_VAR_MAX_MAX(): largest maximum value
- INT_VAR_SIZE_MIN(): smallest domain size
- INT_VAR_SIZE_MAX(): largest domain size
- INT_VAR_DEGREE_SIZE_MIN(): smallest (degree / domain size)
- INT_VAR_DEGREE_SIZE_MAX(): largest (degree / domain size)
- ...

Integer/Boolean Value Selection

Some of the predefined strategies:

- INT_VAL_RND(r): random value
- INT_VAL_MIN(): smallest value
- INT_VAL_MED(): greatest value not greater than the median
- INT_VAL_MAX(): largest value
- INT_VAL_SPLIT_MIN(): values not greater than $\frac{min+max}{2}$
- INT_VAL_SPLIT_MAX(): values greater than $\frac{min+max}{2}$
- INT_VAL_NEAR_MIN(n): values $\approx$ value in array n (ties: smaller value)
- INT_VAL_NEAR_MAX(n): values $\approx$ value in array n (ties: larger value)
- INT_VAL_NEAR_INC(n): values larger than value in array n first
- INT_VAL_NEAR_DEC(n): values smaller than value in array n first
- ...