

Constraint Programming with Gecode

Stefano Benedettini

DEIS, *Alma Mater Studiorum* University of Bologna, Campus of Cesena, Italy
s.benedettini@unibo.it

Artificial Intelligence 2011

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- Gecode Architectural Overview

3 Modeling with Gecode

- Integer Variables
- Set Variables
- Branching
- Modeling Support

What is Gecode?

- Efficient C++ library for constraint programming

Gecode Advantages

- Standard C++
 - Can be easily integrated
 - Extensible
- Parallel
- FOSS

Gecode Disadvantages

- It is C++
 - Less expressive than DSLs or CLP (debatable)
 - A lot of syntactic noise
- It's only CP
 - Does not integrate OR techniques

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- [Gecode](#) home page
- This lecture is based on [Modeling and Programming with Gecode](#)
- Gecode [API documentation](#)
 - We are mostly interested in the [Modeling](#) section

Introduction

Gecode Basics

Getting Started
Gecode Architectural
Overview

Modeling with Gecode

Integer Variables
Set Variables
Branching
Modeling Support

Proprietary but Free (of Charge) to an Extent

CPLEX Suite: C++ library. Feature-rich; supported by IBM

Comet: a C++-inspired language for problem solving.
Integrates CP and OR

SICStus Prolog: Feature-rich implementation of Prolog;
supports CLP

FOSS

Ecl'ps^e: CLP solver and Prolog environment

JaCoP: Java CP solver

Many more...

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

Foreword

- Tested configuration:
 - Ubuntu 10.04
 - GCC 4.4.3
- Latest version (3.5.0) of Gecode is required
 - Download it [here](#)

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- Gecode Architectural Overview

3 Modeling with Gecode

- Integer Variables
- Set Variables
- Branching
- Modeling Support

Introduction

Gecode
Basics

Getting Started

Gecode Architectural
Overview

Modeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- Gecode Architectural Overview

3 Modeling with Gecode

- Integer Variables
- Set Variables
- Branching
- Modeling Support

Introduction

Gecode

Basics

Getting Started

Gecode Architectural
Overview

Modeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- SEND + MORE = MONEY example
- SEND + MORE = MONEY reprise
- SEND + MOST = MONEY constraint optimization problem

Introduction

Gecode

Basics

Getting Started

Gecode Architectural
Overview

Modeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- SEND + MORE = MONEY example
- SEND + MORE = MONEY reprise
- SEND + MOST = MONEY constraint optimization problem

Introduction

Gecode

Basics

Getting Started

Gecode Architectural
Overview

Modeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- SEND + MORE = MONEY example
- SEND + MORE = MONEY reprise
- SEND + MOST = MONEY constraint optimization problem

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- **Gecode Architectural Overview**

3 Modeling with Gecode

- Integer Variables
- Set Variables
- Branching
- Modeling Support

Constraints

- Formally a constraint is a subset of all possible variable assignments (mathematical relation)
- Operationally, this definition is almost useless

Local Consistency

- Local consistency conditions are properties of a constraint model that involve a subset of variables
- A model is satisfiable $\Rightarrow$ is consistent
- Different kinds of consistency:
 - Arc consistency, Path consistency, ...
 - Some are stronger than others
 - Eg., a model might be arc-consistent but not path-consistent
 - Stronger consistency properties are more difficult to prove

Constraint Propagation

- Constraint propagation is a way to check/enforce consistency
 - Check if model is consistent
 - If possible, modify model to reach consistency (new model has the same solutions)
 - Otherwise, model is unsatisfiable
 - Eg., **AC-3 algorithm** for Arc consistency
 - Works by:
 - Removing domain values
 - Adding constraints
 - Modifying existing constraints
 - Some constraint allow specialized forms of propagation
 - Eg., `alldifferent`
-
- A constraint is useful/usable if an effective and efficient propagation algorithm is available

Propagators

- Roughly, a propagator is a constraint implementation
- A propagator implements a constraint propagation algorithm
 - In Gecode a propagator IS a constraint
 - A propagator is an *Actor* which is scheduled for execution
 - You can program new propagators

Consistency Level

- Measures propagation strength
 - How much your model is simplified
- Trade-off between effectiveness and computational cost:
 - A strong enough propagation could solve a CSP ($\mathcal{NP-C}$)
 - Weak propagation could leave the model unaltered

Gecode Space

- A Space embodies the **state** of your CP model
 - Roughly, it is the *constraint store*
 - Manages the following data structures (**mediator**):
 - Decision variables (variable implementations)
 - Constraints (propagators)
 - Branchers (more on this later)
- Provides support for search algorithms

Gecode Variables

- A variable (integer or set) is a **proxy** to one **variable implementation**
 - Exposes a reduced but simpler interface
 - Hides the complexities of variable implementations
- A ***Var** object behaves like a pointer
 - Can refer to different Var Imps during its lifetime
 - Var Imps share the lifetime of the Space they belong to

Relation Between Spaces and Variables

Introduction

Gecode
Basics

Getting Started

Gecode Architectural
Overview

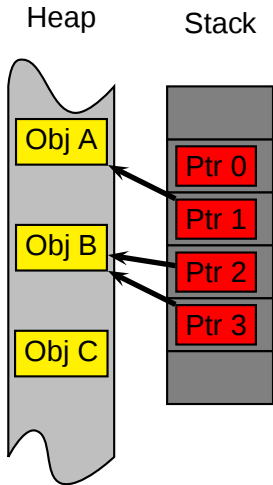
Modeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support



Relation Between Spaces and Variables

Introduction

Gecode
Basics

Getting Started

Gecode Architectural
Overview

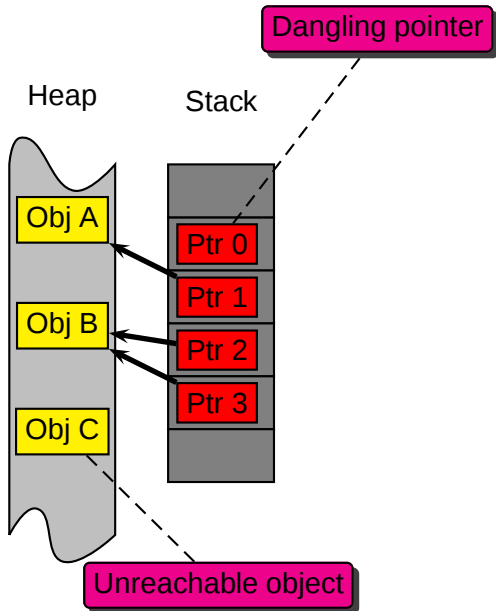
Modeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support



Relation Between Spaces and Variables

Introduction

Gecode
Basics

Getting Started

Gecode Architectural
Overview

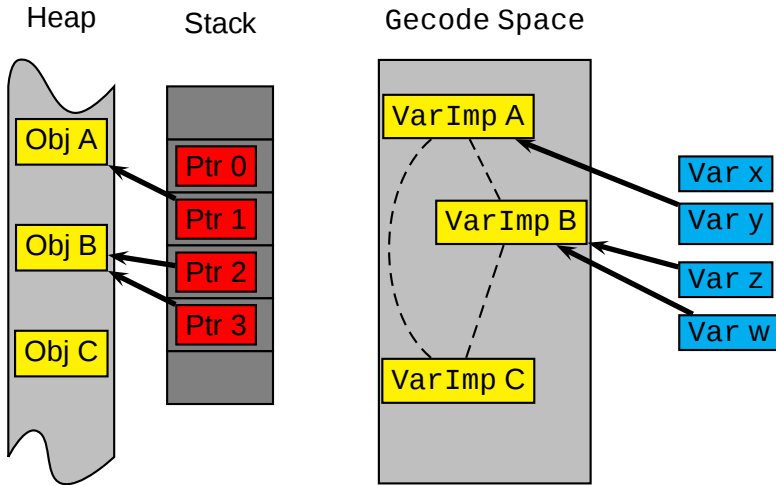
Modeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support



Relation Between Spaces and Variables

Introduction

Gecode
Basics

Getting Started

Gecode Architectural
Overview

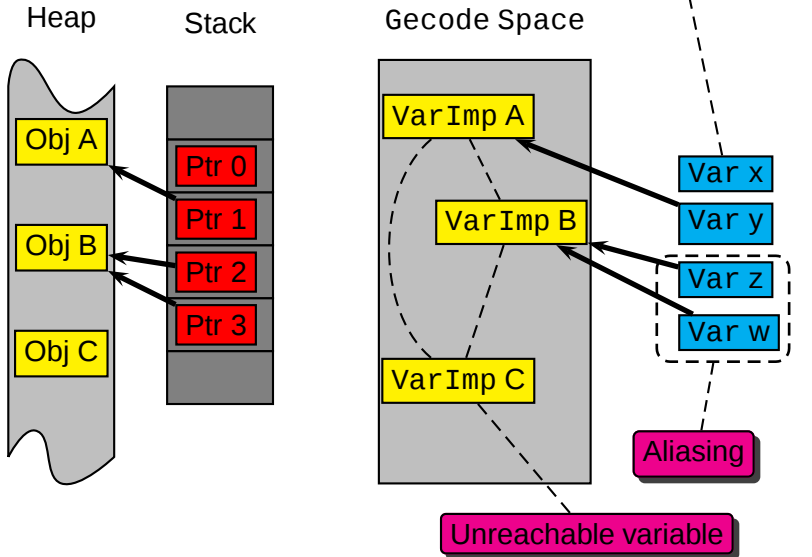
Modeling with
Gecode

Integer Variables

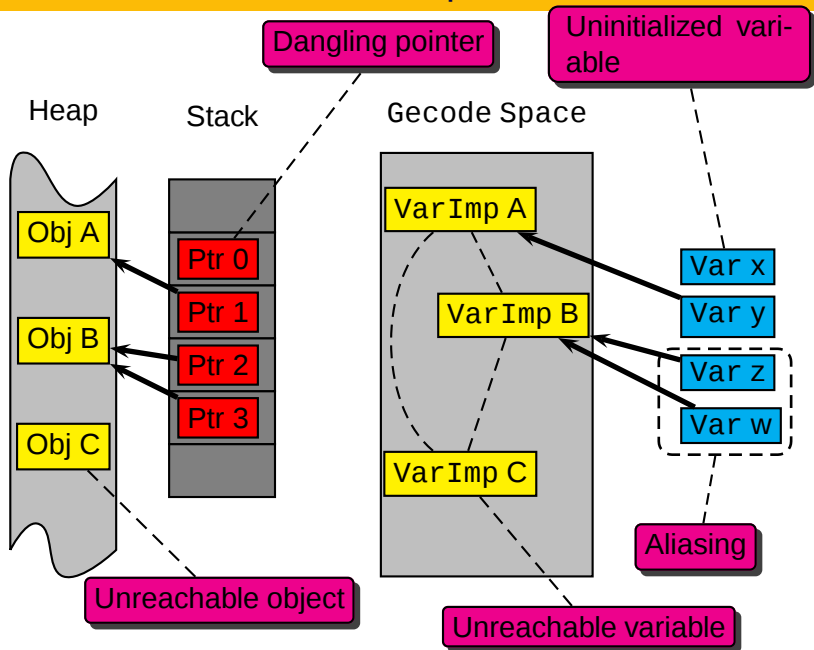
Set Variables

Branching

Modeling Support



Relation Between Spaces and Variables



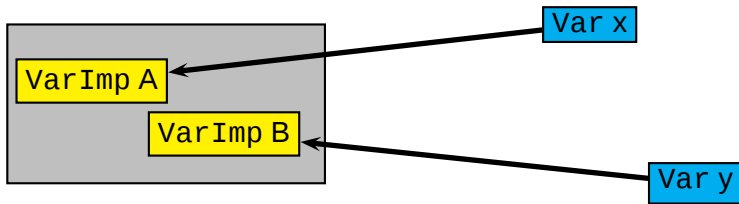
Copying and Updating Variables

- Copying a variable create an alias for a variable implementation
 - You do not instantiate a new variable
 - Just like copying pointers does not copy pointees
- We have to manually invoke update method during copy-construction

Example

```
struct MyModel : Space {  
    IntVar x, y;  
    // ...  
    MyModel(bool share, MyModel& m) : Space(m) {  
        x.update(*this, share, m.x);  
        // y = m.y; // wrong: this creates an alias!  
        y.update(*this, share, m.y);  
    }  
}
```

Original Space



Original setup

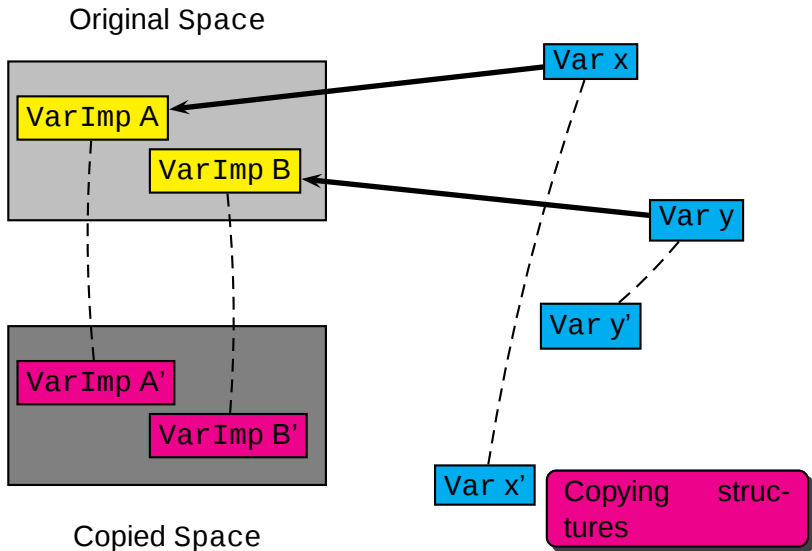
Introduction

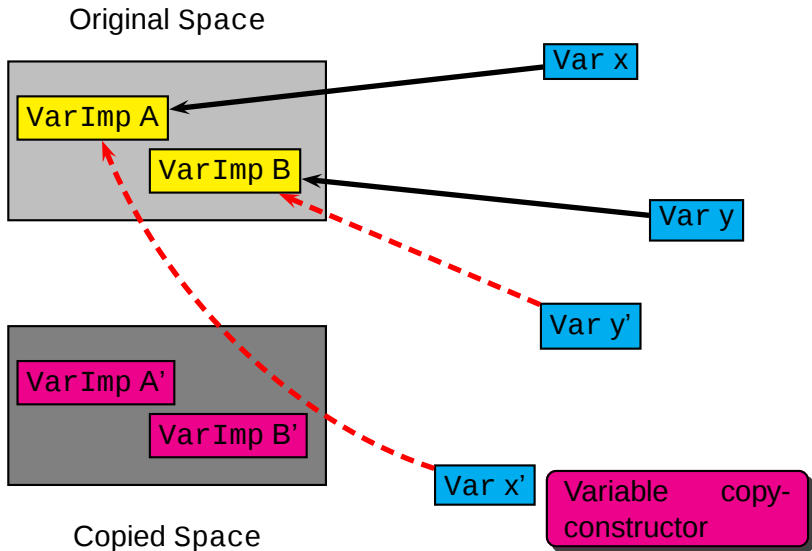
Gecode
Basics

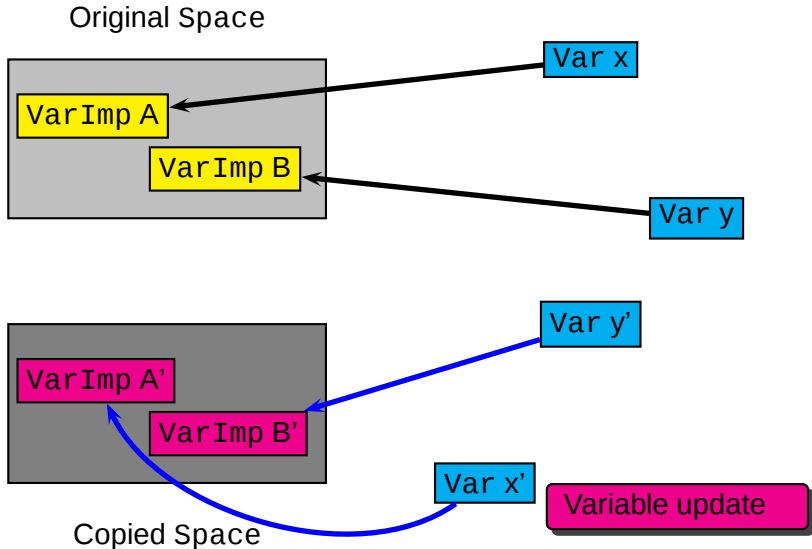
Getting Started
Gecode Architectural
Overview

Modeling with
Gecode

Integer Variables
Set Variables
Branching
Modeling Support







Ingredients for a Search Algorithm

- A search algorithm is characterized by:
 - 1 The **shape** of the search tree
 - 2 The **traversal** strategy
 - Depth First Search
 - Breadth First Search
 - Beam Search
 - Limited Discrepancy
 - 3 For optimization problem, also a node **cost** function
- Branchers “shape” the search tree
 - “Split” nodes in the search tree
 - Formally they define the node successor function
- Operationally, a brancher creates choice points:
 - A variable is assigned
 - A new constraint is posted
 - Logically, a father node (of type Space) is copied and modified
- Branchers operate on decision variables

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- Search algorithms “navigate” the search tree
- Gecode is algorithm-agnostic
 - It's not committed to a particular strategy
- New search strategies can be implemented
- We will settle for default ones:
 - Depth First Search: for CSPs
 - Branch-and-Bound: for COPs; also requires a suitable cost function

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- Gecode Architectural Overview

3 Modeling with Gecode

- Integer Variables
- Set Variables
- Branching
- Modeling Support

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- Overview on Gecode modeling facilities
- Decision variable types:
 - 1 Finite domain integer variables
 - 2 Boolean variables
 - 3 Finite set variables
- A summary on available constraints
- Presentation of Gecode MiniModel
- Useful data structures:
 - Integer arrays
 - Integer sets
 - Matrices
- Branching and search

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- Gecode Architectural Overview

3 Modeling with Gecode

- **Integer Variables**
- Set Variables
- Branching
- Modeling Support

- Domain is a finite subset of $\mathbb{N}$:
 - $\{\text{Int}::\text{Limits}::\text{min}, \text{Int}::\text{Limits}::\text{max}\}$ for integer variables
 - $\{0, 1\}$ for Boolean variables
- They are not interchangeable!
- Domain is never empty:
 - If it is, the space is failed
- Include `int.hh`

Example

From now on, Gecode namespace is implied. `home` represent the current Space.

```
IntVar x(home, 4, 10); //  $x \in \{4, \dots, 10\}$ 
IntVar y(home, IntArgs(5, 1, 5, 3, 7, 6)); //
     $y \in \{1, 3, 5, 6, 7\}$ 
BoolVar b(home, 0, 1), c(home, 0, 0); //
     $b \in \{0, 1\}, c = 0$ 
```

`IntArgs` constructor has variadic argument list

- *Fixed size* array of variables
- Useful to define a number of variables in one go
- *Can* be stored as data members in models

Example

```
IntVarArray v(home, 4, -3, 2); //  
     $v_i \in \{-3, \dots, 2\}, \forall i \in 0, \dots, 3$   
IntVarArray w(home, 3, IntArgs(3, 10, 20, 30)); //  
     $w_i \in \{10, 20, 30\}, \forall i \in 0, \dots, 2$   
IntVarArray row(home, 9, 1, 9); // a Sudoku row of  
    cells  
BoolVarArray bs(home, 100, 0, 1); // variable array  
    for a SAT instance
```

- Gecode offers an array structure much more powerful than plain C++ array
 - Grows on demand
 - Appendable
 - Sliceable
 - Factory method to return integer ranges
- **Cannot be stored** as data members in models
- **IntVarArgs** (**BoolVarArgs**) works on integer (Boolean) variables
- IntArgs works on plain integers

Example

```
IntVarArgs v(home, 4, 1, 7); // can create variables
IntVarArgs l = IntVarArgs(home, 6, -5, 5) + v; //
    can concatenate two or more arrays
IntVarArgs s = l.slice(3, 1, 4); // s = (l3, l4, l5, l6)
s << IntVar(home, 1, 10); // append
IntArgs::create(6, 2, 2); // (2,4,6,8,10,12)
```

Generalities

- Propagators are posted to a Space by calling functions
- Data structures involved in a constraint are always copied when passed as arguments
- Can specify consistency level

Reified Constraints

- Let C be a constraint and b a Boolean (control) variable
- C is reified entails:

$$C \Leftrightarrow b$$

Example

$x \in \{0, \dots, 10\}, y \in \{4, \dots, 6\}, b \in \{0, 1\}, x < y \Leftrightarrow b$

Implies:

$x \in \{0, \dots, 5\}$ if $b = 1$

$x \in \{3, \dots, 10\}$ if $b = 0$

Introduction

Gecode
Basics

Getting Started

Gecode Architectural
OverviewModeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- **Relational constraints** on integer variables or variable arrays altogether
- Constraints on variables

```
rel(home, x, IRT_LE, y); //  $x < y$   
rel(home, x, IRT_NQ, 7); //  $x \neq 7$ 
```

- Constraints on arrays
 - Pairwise relations:

```
rel(home, v, IRT_GQ); //  $v_0 \geq v_1 \geq \dots \geq v_n$ 
```

- Element-by-element:

```
rel(home, v, IRT_LE, w); //  $v_0 < w_0 \wedge v_1 < w_1 \wedge \dots \wedge v_n < w_n$ 
```

- Several **arithmetic functions** available

```
min(home, x, y, z); //  $\min(x, y) = z$   
max(home, v, m); //  $\max(v_0, v_1, \dots, v_n) = m$   
mod(home, x, q, r); //  $x \bmod q = r$ 
```

- **Linear combination** of variables and scalars

```
linear(home, a, v, IRT_LQ, x); //  $\sum_{i=0}^n a_i v_i \leq x$   
linear(home, a, v, IRT_EQ, 3, b); //  $\sum_{i=0}^n a_i v_i = 3 \Leftrightarrow b$ 
```

- Also Boolean variables

```
linear(home, bs, IRT_GR, 10); //  $\sum_{i=0}^n bs_i > 10$ : at  
    least 10 variables set
```

- Alldifferent

```
distinct(home, v); // stronger than  
 $v_i \neq v_j \quad \forall i, j = 0, \dots, n$ 
```

- Array element access

```
element(home, v, i, y); //  $v_i = y, i \in \mathbb{N}$   
element(home, v, x, y); //  $v_x = y, x$  is a variable
```

- Channel

```
channel(home, x, y); //  $x_i = j \Leftrightarrow y_j = i \quad \forall i, j = 0, \dots, n$   
channel(home, x, b); //  $x = b, x \in \mathbb{N}, b \in \{0, 1\}$ 
```

- Sorted

```
sorted(home, v); // array is sorted in increasing  
order
```

- Counting

```
count(home, v, k, IRT_EQ, m); // number of elements  
in v equal to k:  $|\{i \in 0, \dots, n \mid v_i = k\}| = m$ 
```

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- Gecode Architectural Overview

3 Modeling with Gecode

- Integer Variables
- **Set Variables**
- Branching
- Modeling Support

- **Set variables** represent finite sets of integers
- Domain of a set variable is a set of sets of integers
- Domains of that kind grow exponentially with cardinality
 - There are 2^n subset of a n -element set
- Set domains are approximated as *set intervals*

Domain: interval $[l, u]$ represents all sets $\{s \mid l \subseteq s \subseteq u\}$

Greatest Lower Bound l : all elements which *must* be included in the set

Least Upper Bound u : all elements which *may* be included in the set

Cardinality constraint: $\#[m, M]$ bounds on cardinality

Example

$$\begin{aligned} & [\{1, 2\}, \{1, 2, 3, 4\}] \#[2, 4] \\ & \{\{1, 2\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 3, 4\}\} \end{aligned}$$

- Include `set.hh`
- From programmer's point of view, `SetVars` behave just like `IntVars`
- Arrays of set variables are supported
- Remember the `IntSet` class to easily specify literal sets

Example

```
IntSet a(home, 1, 2, 1, 7, 1, 4); // defines
    a ∈ [{1,2},{1,...,7}] #[1,4]
IntSet b(home, IntArgs(2, 1, 2), IntSet(1, 7), 1, 4)
    ; // same as before
IntSet c(home, IntArgs(2, 1, 2), IntSet(1, 7)); //
    same as before but cardinality is bounded in
    [0,Set::Limits::card]
```

Introduction

Gecode
Basics

Getting Started

Gecode Architectural
OverviewModeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- Everything you know about constraints on integers still applies
- **Relational** constraints

```
rel(home, a, SRT_SUB, b); //  $a \subseteq b$   
rel(home, a, SRT_NQ, b, z); //  $a \neq b \Leftrightarrow z$ 
```

- **Operation** constraints

```
rel(home, SOT_UNION, x, y); //  $\bigcup_{x_i \in x} x_i = y$   
rel(home, x, SOT_INTER, y, SRT_EQ, z); //  $x \cap y = z$ 
```

Introduction

Gecode

Basics

Getting Started

Gecode Architectural
OverviewModeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- Minimal and maximal element

```
min(home, s, x); //  $\min s = x \wedge s \neq \emptyset$ 
max(home, s, x, b); //  $(\max s = x \wedge s \neq \emptyset) \Leftrightarrow b$ 
```

- Channeling

```
channel(home, v, s); //  $\{v_i \in v\} = s$ 
channel(home, b, s); //  $b_i \Leftrightarrow i \in s \quad \forall b_i \in b$ 
```

- Cardinality

```
cardinality(home, s, x); //  $|s| = x$ 
```

- Weights

```
weights(home, e, w, s, x); //

$$e, w \in \mathbb{Z}^n \quad s \subseteq e, x = \sum_{e_i, w_i \in e, w} w_i e_i I_{(e_i \in s)}$$

```

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- Gecode Architectural Overview

3 Modeling with Gecode

- Integer Variables
- Set Variables
- **Branching**
- Modeling Support

Introduction

Gecode

Basics

Getting Started

Gecode Architectural
Overview

Modeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- Branchers create choice points in the search tree
- Branchers are declaratively specified
 - You can also program your own brancher
- Branchers are actors
 - A brancher is associated to one or more variables
 - Whenever the propagators reach a fixpoint (no more propagation is possible) a brancher is run and choice points are created
 - When all choice points have been explored, the brancher ceases to exist
 - Control passes to the next brancher
 - Branchers are tried in order
- A brancher can either assign a variable or add a new constraint

Brancher Definition

- A Brancher is defined by three elements:
 - A set of variables
 - A variable selection strategy
 - A value selection strategy

Some Branching Strategies

- Variable selection strategies (for **integers** and **sets**):
 - First unassigned
 - Smallest/largest degree
 - Smallest/largest domain size
- Value selection strategies (for **integers** and **sets**):
 - Smallest/largest/random value
 - Values not greater than mean (domain splitting)
 - All values from smallest/largest
- Note: First unassigned + All values = labeling

```
IntVarArray v(home, 6, 2, 5);  
branch(v, INT_VAR_SIZE_MIN, INT_VAL_MIN);
```

- First it creates $v_0 = 2 \vee v_0 \neq 2$ choice points
- If $v_0 = 2$ is selected (a variable assignment), branching goes on with v_1
- If $v_0 \neq 2$ is selected (a new constraint), branching continues with $v_0 = 3 \vee v_0 \neq 3$

```
branch(v, INT_VAR_SIZE_MIN, INT_VALUES_MAX);
```

- This is labeling starting from the maximum value
- Creates $v_0 = 2 \vee v_0 = 3 \vee v_0 = 4 \vee v_0 = 5$ choice points
- Whichever branch is selected, next choice points would be: $v_1 = 2 \vee v_1 = 3 \vee v_1 = 4 \vee v_1 = 5$

Introduction

Gecode Basics

Getting Started

Gecode Architectural
Overview

Modeling with Gecode

Integer Variables

Set Variables

Branching

Modeling Support

1 Introduction

2 Gecode Basics

- Getting Started
- Gecode Architectural Overview

3 Modeling with Gecode

- Integer Variables
- Set Variables
- Branching
- Modeling Support

Introduction

Gecode
Basics

Getting Started

Gecode Architectural
OverviewModeling with
Gecode

Integer Variables

Set Variables

Branching

Modeling Support

- Gecode uses C++ operator overloading to naturally express constraints
- Almost any legal C++ expression can be posted:
 - All arithmetic operators
 - Relational and Boolean operators
 - Set operations (borrow from Boolean operators)
 - Intermixes integer, set and Boolean variables
- Advantages:
 - Dramatically reduces code size
 - Increases readability
- Include `minimodel.hh`
- Comprehensive documentation is available [here](#)

- Any arithmetic expression:

```
rel(home, 10 * x + 4 * y - 3 * z == -1);  
rel(home, 10 * x + x * (y + 2 * z) == 0);
```

- Array element access:

```
rel(home, element(v, 2 * x + 1) == y); //  $v_{2x+1} = y$ 
```

- Array operations:

```
rel(home, sum(v) + min(w) == x); //  $\sum v_i + \min w = x$ 
```

- Intermixes integer and Boolean variables and relations:

```
rel(home, (x + y == z) >> (a && b)); //  
       $x + y = z \Rightarrow a \wedge b$   
rel(home, (!a | b | !c | d) && (a | b | c | !d)); //  
      two SAT clauses
```

- Set operations:

```
rel(home, (a & b) | c == d); //  $(a \cap b) \cup c = d$ 
```

- Variable binding:

```
IntVar z = expr(home, x + 2 * y - 1);
```

- `Matrix<A>` is a template class
- Provides a **bidimensional view** of an array-like type
 - `IntArgs`, `IntVarArray` or `IntVarArgs`
- Can access rows, columns, single elements or submatrices

Example

```
IntVarArray v(home, 81, 1, 9);  
Matrix<IntVarArray> m(v, 9, 9); // a Sudoku grid  
m.row(0); // first row  
m.col(2); // third column  
IntVar x = m(1, 3); // element access (column, row)  
m.slice(3, 6, 0, 3); // fourth frame:  
     $i \in \{3, 4, 5\}, j \in \{0, 1, 2\}$   
distinct(home, m.row(1)); // posts an alldifferent  
    constraint on the second row
```

- Constraints specified by a deterministic finite state machine
 - Enforce an array of variables to match a regular expression
 - Eg.: $v \in \{0, 1, 2\}^n \sim 0^*(1|2)^+0$
 - In Gecode

```
REG r = *REG(0) + +(REG(1) | REG(2)) + REG(0);
REG r2 = *REG(0) + +REG(IntArgs(2, 1, 2)) + REG(0);
// equivalent
REG r3 = REG(0)(1, 3) + *REG(IntArgs(3, 0, 1, 2)); //
0{1,3}(0|1|2)*
```

- Usage:

```
REG r = /* your regular expression */;
DFA automaton(r);
extensional(home, v, automaton);
```

- Makes writing models for COP easier and cleaner
- Your script should extend **Maximize/MinimizeSpace** instead of Space
- Implement a cost method that computes the cost function $F(\cdot)$

```
struct MyModel : MaximizeSpace {  
    IntVarArray v;  
    IntVar x;  
    // ...  
    IntVar cost() const {  
        return expr(*this, sum(v) + 10 * x);  
    }  
};
```

- Whenever a better solution s^* is found, BAB engine will post the new constraint

$$s < F(s^*)$$

Constraint Programming with Gecode

Stefano Benedettini

DEIS, *Alma Mater Studiorum* University of Bologna, Campus of Cesena, Italy
s.benedettini@unibo.it

Artificial Intelligence 2011