DM841

Discrete Optimization

Part II

Lecture 4

Introduction to Gecode

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Christian Schulte, KTH Royal Institute of Technology*]

Outline

1. Constraint Languages

2. Gecode

Resume

- Modelling in CP
  - Examples: graph coloring, Sudoku, crosswords, cryptoarithmetic
- Overview on solving constraint satisfaction problems
  - search = backtracking + branching
  - propagate + filtering
  - level of consistency (arc/generalized + value/bound/domain)

Constraint Programming:
representation (modeling language) + reasoning (search + propagation)

Outline

1. Constraint Languages

2. Gecode

Constraint Programming Systems

Expressiveness language stream
(modeling)
+
(efficient solvers)
Algorithm stream

CP systems typically include

- general purpose algorithms for constraint propagation (arc consistency on finite domains)
- built-in constraint propagation for various constraints (eg, linear, Boolean, global constraints)
- built-in for constructing various forms of search

Logic Programming

Logic programming is the use of mathematical logic for computer programming.

First-order logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver.

Logic programming supports the notion of logical variables

- Syntax – Language
  - Alphabet
  - Well-formed Expressions
    E.g., $4X + 3Y = 10; 2X - Y = 0$
- Semantics – Meaning
  - Interpretation
  - Logical Consequence
- Calculi – Derivation
  - Inference Rule
  - Transition System

Logic Programming

Example: Prolog

> *A logic program is a set of axioms, or rules, defining relationships between objects.*
>
> *A computation of a logic program is a deduction of consequences of the program.*
>
> *A program defines a set of consequences, which is its meaning.*

Sterling and Shapiro: The Art of Prolog, Page 1.
To deal with the other constraints one has to add other constraint solvers to the language. This led to Constraint Logic Programming

Prolog Approach

- Prolog II till Prolog IV [Colmerauer, 1990]
- CHIP V5 [Dincbas, 1988] http://www.cosytec.com (commercial)
- CLP [Van Hentenryck, 1989]
- Ciao Prolog (Free, GPL)
- GNU Prolog (Free, GPL)
- SICStus Prolog
- ECLiPSe [Wallace, Novello, Schimpf, 1997] http://eclipse-clp.org/ (Open Source)
- Mozart programming system based on Oz language (incorporates concurrent constraint programming) http://www.mozart-oz.org/ [Smolka, 1995]

Other Approaches

Libraries:
Constraints are modeled as objects and are manipulated by means of special methods provided by the given class.

- CHOCO (free) http://choco.sourceforge.net/
- Kaolog (commercial) http://www.koalog.com/php/index.php
- ILOG CP Optimizer `www.cpoptimizer.ilog.com` (ILOG, commercial)
- Gecode (free) `www.gecode.org`
  C++, Programming interfaces Java and MiniZinc
- G12 Project
  `http://www.nicta.com.au/research/projects/constraint_programming_platform`

Other Approaches

Modelling languages:

- OPL [Van Hentenryck, 1999] ILOG CP Optimizer
  www.cpoptimizer.ilog.com (ILOG, commercial)
- MiniZinc [] (open source, works for various systems, ECLiPSe, Geocode)
- Comet
- AMPL

- Catalogue of Constraint Programming Tools:
  `http://openjvm.jvmhost.net/CPSolvers/`
- Workshop "CPSOLVERS-2013"
  `http://cp2013.a4cp.org/node/99`

CP Languages

Greater expressive power than mathematical programming

- constraints involving disjunction can be represented directly
- constraints can be encapsulated (as predicates) and used in the definition of further constrains

However, CP models can often be translated into MIP model by

- eliminating disjunctions in favor of auxiliary Boolean variables
- unfolding predicates into their definitions

CP Languages

- Fundamental difference to LP
  - language has structure (global constraints)
  - different solvers support different constraints
- In its infancy
- Key questions:
  - what level of abstraction?
    - solving approach independent: LP, CP, ...?
    - how to map to different systems?
  - Modeling is very difficult for CP
    - requires lots of knowledge and tinkering

Summary

- Model your problem via Constraint Satisfaction Problem
- Declare Constraints + Program Search
- Constraint Propagation
- Languages

Outline

1. Constraint Languages

2. Gecode

Gecode

an open constraint solving library

Christian Schulte
KTH Royal Institute of Technology, Sweden

Gecode People

- Core team
  - Christian Schulte, Guido Tack, Mikael Z. Lagerkvist.
- Code
  - contributions: Christopher Mears, David Rijsman, Denys Duchier, Filip Konvicka, Gabor Szokoli, Gabriel Hjort Blindell, Gregory Crosswhite, Håkan Kjellerstrand, Joseph Scott, Lubomir Moric, Patrick Pekczynski, Raphael Reischuk, Stefano Gualandi, Tias Guns, Vincent Barichard.
  - fixes: Alexander Samoilov, David Rijsman, Geoffrey Chu, Grégoire Dooms, Gustavo Gutierrez, Olof Sivertsson, Zandra Norman.
- Documentation
  - contributions: Christopher Mears.
  - fixes: Seyed Hosein Attarzadeh Niaki, Vincent Barichard, Pavel Bochman, Felix Brandt, Markus Böhm, Roberto Castañeda Lozano, Gregory Crosswhite, Pierre Flener, Gustavo Gutierrez, Gabriel Hjort Blindell, Sverker Janson, Andreas Karlsson, Håkan Kjellerstrand, Chris Mears, Benjamin Negrevergne, Flutra Osmani, Max Ostrowski, David Rijsman, Dan Scott, Kish Shen.

Gecode

Generic Constraint Development Environment

- **open**
  - easy interfacing to other systems
  - supports programming of: constraints, branching strategies, search engines, variable domains
- **comprehensive**
  - constraints over integers, Booleans, sets, and floats
    - different propagation strength, half and full reification, …
  - advanced branching heuristics (accumulated failure count, activity)
  - many search engines (parallel, interactive graphical, restarts)
  - automatic symmetry breaking (LDSB)
  - no-goods from restarts
  - MiniZinc support

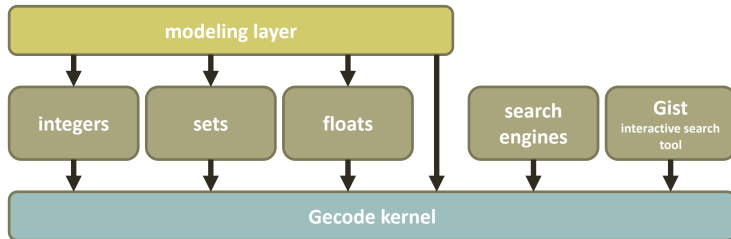
Gecode

Generic Constraint Development Environment

- **efficient**
  - *all* gold medals in *all* categories at *all* MiniZinc Challenges
- **documented**
  - tutorial (> 500 pages) and reference documentation
- **free**
  - MIT license, listed as free software by FSF
- **portable**
  - implemented in C++ that carefully follows the C++ standard
- **parallel**
  - exploits multiple cores of today's hardware for search
- **tested**
  - some 50000 test cases, coverage close to 100%

SOME BASIC FACTS

Architecture



- Small domain-independent kernel
- Modules
  - per variable type: variables, constraint, branchings, …
  - search, FlatZinc support, …
- Modeling layer
  - arithmetic, set, Boolean operators; regular expressions; matrices, …
- All APIs are user-level and documented (tutorial + reference)

Openness

- MIT license permits commercial, closed-source use
  - motivation: public funding, focus on research
  - not a reason: attitude, politics, dogmatism
- More than a license
  - **license** restricts what users **may do**
  - **code and documentation** restrict what users **can do**
- Modular, structured, documented, readable
  - complete tutorial and reference documentation
  - new ideas from Gecode available as scientific publications
- Equal rights: Gecode users are first-class citizens
  - you can do what we can do: APIs
  - you can know what we know: documentation
  - on every level of abstraction

Constraints in Gecode

- Constraint families
  - arithmetics, Boolean, ordering, ….
  - alldifferent, count (global cardinality, …), element, scheduling, table and regular, sorted, sequence, circuit, channel, bin-packing, lex, geometrical packing, nvalue, lex, value precedence, …
- Families
  - many different variants and different propagation strength
- All global constraints from MiniZinc have a native implementation
- Gecode ⇆ Global Constraint Catalogue: > 70 constraints

  abs_value, all_equal, alldifferent, alldifferent_cst, among, among_seq, among_var, and, arith, atleast, atmost, bin_packing, bin_packing_capa, circuit, clause_and, clause_or, count, counts, cumulative, cumulatives, decreasing, diffn, disjunctive, domain, domain_constraint, elem, element, element_matrix, eq, eq_set, equivalent, exactly, geq, global_cardinality, gt, imply, in, in_interval, in_intervals, in_relation, in_set, increasing, int_value_precede, int_value_precede_chain, inverse, inverse_offset, leq, lex, lex_greater, lex_greatereq, lex_less, lex_lesseq, link_set_to_booleans, lt, maximum, minimum, nand, neq, nor, not_all_equal, not_in, nvalue, nvalues, or, roots, scalar_product, set_value_precede, sort, sort_permutation, strictly_decreasing, strictly_increasing, sum_ctr, sum_set, xor

History

- 2002
  - development started
- 1.0.0
  - December 2005
- 2.0.0
  - November 2007
- 3.0.0
  - March 2009
- 4.0.0
  - March 2013
- 4.2.0 (current)
  - July 2013

43 kloc, 21 klod

77 kloc, 41 klod

34 releases

81 kloc, 41 klod

164 kloc, 69 klod

168 kloc, 71 klod

Tutorial Documentation

- 2002
  - development started
- 1.0.0 — 43 kloc, 21 klod
  - December 2005
- 2.0.0 — 77 kloc, 41 klod
  - November 2007
- 3.0.0 — 1 klod
  - March 2009
- 4.0.0 — 164 kloc, 69 klod
  - March 2013
- 4.2.0 (current)
  - July 2013

Modeling with Gecode (98 pages)

Modeling & Programming with Gecode (522 pages)

Future

- Large neighborhood search and other meta-heuristics
  - contribution expected
- Simple temporal networks for scheduling
  - contribution expected
- More expressive modeling layer on top of libmzn
- Grammar constraints
  - contribution expected
- Propagator groups
- …

- Contributions anyone?

Deployment & Distribution

- Open source ≠ Linux only
  - Gecode is native citizen of: Linux, Mac, Windows
- High-quality
  - extensive test infrastructure (around 16% of code base)
- Downloads from Gecode webpage
  - software: between 25 to 125 per day (total > 40000)
  - documentation: between 50 to 300 per day
- Included in
  - Debian, Ubuntu, Fedora, OpenSUSE, Gentoo, FreeBSD, …

Integration & Standardization

- Why C++ as implementation language?
  - good compromise between portability and efficiency
  - good for interfacing

  well demonstrated

- Integration with XYZ…
  - Gecode empowers users to do it
  - no "Jack of all trades, master of none"

  well demonstrated

- Standardization
  - any user can build an interface to whatever standard…
  - systems are the wrong level of abstraction for standardization
  - MiniZinc and AMPL are de-facto standards

Modeling & Programming

Constraint Programming with Gecode

Overview

- Program model as ***script***
  - declare variables
  - post constraints (creates propagators)
  - define branching

- Solve script
  - basic search strategy
  - Gist: interactive visual search

Program Model as Script

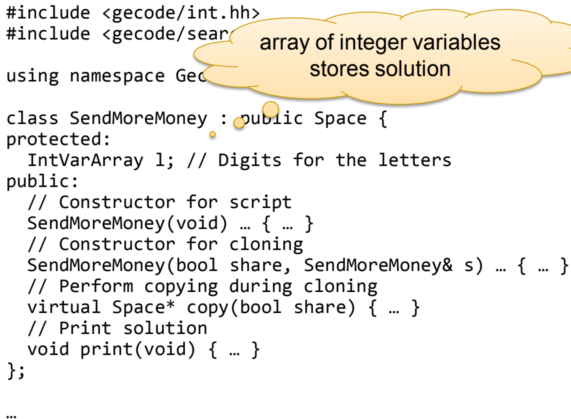
Script: Overview

- Script is class inheriting from class Space
  - members store variables regarded as solution
- Script constructor
  - initialize variables
  - post propagators for constraints
  - define branching
- Copy constructor and copy function
  - copy a Script object during search
- Exploration takes Script object as input
  - returns object representing solution
- Main function
  - invokes search engine

Script for SMM: Structure

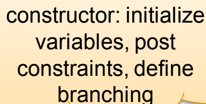
```
#include <gecode/int.hh>
#include <gecode/search.hh>

using namespace Gecode;

class SendMoreMoney : public Space {
protected:
  IntVarArray l; // Digits for the letters
public:
  // Constructor for script
  SendMoreMoney(void) … { … }
  // Constructor for cloning
  SendMoreMoney(bool share, SendMoreMoney& s) … { … }
  // Perform copying during cloning
  virtual Space* copy(bool share) { … }
  // Print solution
  void print(void) { … }
};

…
```

Script for SMM: Structure



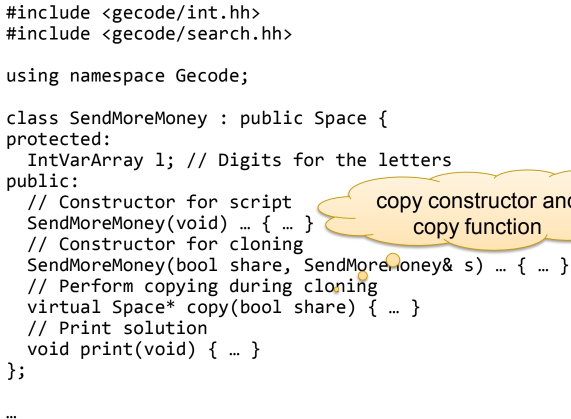

```
#include <gecode/int.hh>
#include <gecode/sear

using namespace Gec

class SendMoreMoney : public Space {
protected:
  IntVarArray l; // Digits for the letters
public:
  // Constructor for script
  SendMoreMoney(void) … { … }
  // Constructor for cloning
  SendMoreMoney(bool share, SendMoreMoney& s) … { … }
  // Perform copying during cloning
  virtual Space* copy(bool share) { … }
  // Print solution
  void print(void) { … }
};

…
```

Script for SMM: Structure




```
#include <gecode/int.hh>
#include <gecode/search.hh>

using namespace Gecode;

class SendMoreMoney : pu
protected:
  IntVarArray l; // Digits fo  the letters
public:
  // Constructor for script
  SendMoreMoney(void) … { … }
  // Constructor for cloning
  SendMoreMoney(bool share, SendMoreMoney& s) … { … }
  // Perform copying during cloning
  virtual Space* copy(bool share) { … }
  // Print solution
  void print(void) { … }
};

…
```

Script for SMM: Structure



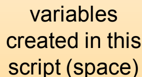

```
#include <gecode/int.hh>
#include <gecode/search.hh>

using namespace Gecode;

class SendMoreMoney : public Space {
protected:
  IntVarArray l; // Digits for the letters
public:
  // Constructor for script
  SendMoreMoney(void) … { … }
  // Constructor for cloning
  SendMoreMoney(bool share, SendMoreMoney& s) … { … }
  // Perform copying during cloning
  virtual Space* copy(bool share) { … }
  // Print solution
  void print(void) { … }
};

…
```

Script for SMM: Constructor

```
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]);
  // Post constraints
  …
  // Post branchings
  …
}
```

Script for SMM: Constructor

```
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]);
  // Post constraints
  …
  // Post branchings
  …
}
```

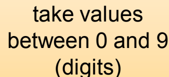


Script for SMM: Constructor

```
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]);
  // Post constraints
  …
  // Post branchings
  …
}
```

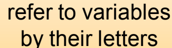
Script for SMM: Constructor

```
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]);
  // Post constraints
  …
  // Post branchings
  …
}
```



Script for SMM: Constructor

```
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]);
  // Post constraints
  …
  // Post branchings
  …
}
```



Script for SMM: Constructor

```
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]);
  // No leading zeros (IRT: integer relation type)
  rel(*this, s, IRT_NQ, 0);
  rel(*this, m, IRT_NQ, 0);
  // All letters must take distinct digits
  distinct(*this, l);
  // The linear equation must hold
  …
  // Branch over the letters
  …
}
```

Posting Constraints

- Defined in namespace `Gecode`
- Check documentation for available constraints
- Take script reference as first argument
  - where is the propagator for the constraint to be posted!
  - script is a subclass of Space (computation space)

Linear Equations and Linear Constraints

- Equations of the form

  $c_1 \cdot x_1 + \ldots + c_n \cdot x_n = d$

  - integer constants: c_i and d
  - integer variables: x_i
- In Gecode specified by arrays
  - integers `(IntArgs)` c_i
  - variables `(IntVarArray, IntVarArgs)` x_i
- Not only equations
  - `IRT_EQ, IRT_NQ, IRT_LE, IRT_GR, IRT_LQ, IRT_GQ`
  - equality, disequality, inequality (less, greater, less or equal, greater or equal)

Script for SMM: Constructor

```
SendMoreMoney(void) : l(*this, 8, 0, 9) {
  …
  // The linear equation must hold
  IntArgs c(4+4+5); IntVarArgs x(4+4+5);
  c[0]=1000; c[1]=100; c[2]=10; c[3]=1;
  x[0]=s;    x[1]=e;   x[2]=n;  x[3]=d;
  c[4]=1000; c[5]=100; c[6]=10; c[7]=1;
  x[4]=m;    x[5]=o;   x[6]=r;  x[7]=e;
  c[8]=-10000; c[9]=-1000; c[10]=-100; c[11]=-10; c[12]=-1;
  x[8]=m;      x[9]=o;     x[10]=n;    x[11]=e;   x[12]=y;
  linear(*this, c, x, IRT_EQ, 0);
  // Branch over the letters
  …
}
```

Linear Expressions

- Other options for posting linear constraints are available: minimodeling support
  - linear expressions
  - Boolean expressions
  - matrix classes
  - …
- See the examples that come with Gecode

Script for SMM: Constructor

```
…
#include <gecode/minimodel.hh>
…

  SendMoreMoney(void) : l(*this, 8, 0, 9) {
    …
    // The linear equation must hold
    post(*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 over the letters
    …
  }
```

Script for SMM: Constructor

```
SendMoreMoney(void) : l(*this, 8, 0, 9) {
  …
  // Branch over the letters
  branch(*this, l, INT_VAR_SIZE_MIN, INT_VAL_MIN);
}
```

Branching

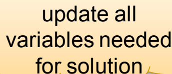
- Which variable to choose
  - given order `INT_VAR_NONE`
  - smallest size `INT_VAR_SIZE_MIN`
  - smallest minimum `INT_VAR_MIN_MIN`
  - …
- How to branch: which value to choose
  - try smallest value `INT_VAL_MIN`
  - split (lower first) `INT_VAL_SPLIT_MIN`
  - …

Script for SMM: Copying

```
// Constructor for cloning
SendMoreMoney(bool share, SendMoreMoney& s) : Space(share, s) {
  l.update(*this, share, s.l);
}
// Perform copying during cloning
virtual Space* copy(bool share) {
  return new SendMoreMoney(share,*this);
}
```

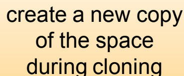
Script for SMM: Copying

```
// Constructor for cloning
SendMoreMoney(bool share, SendMoreMoney& s) : Space(share, s) {
  l.update(*this, share, s.l);
}
// Perform copying during cloning
virtual Space* copy(bool share) {
  return new SendMoreMoney(share,*th
}
```



Script for SMM: Copying

```
// Constructor for cloning
SendMoreMoney(bool share, SendMoreMoney& s) : Space(share, s) {
  l.update(*this, share, s.l);
}
// Perform copying during cloning
virtual Space* copy(bool share) {
  return new SendMoreMoney(share,*this);
}
```



Copying

- Required during exploration
  - before starting to guess: make copy
  - when guess is wrong: use copy
  - discussed later
- Copy constructor and copy function needed
  - copy constructor is specific to script
  - updates (copies) variables in particular

Copy Constructor And Copy Function

- Always same structure

- Important!
  - must update the variables of a script!
  - if you forget: crash, boom, bang, …

Script for SMM: Print Function

```
…
  // Print solution
  void print(void) {
    std::cout << l << std::endl;
  }
```

Summary: Script

- Variables
  - declare as members
  - initialize in constructor
  - update in copy constructor
- Posting constraints
- Create branching
- Provide copy constructor and copy function

Solving Scripts

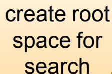
Available Search Engines

- Returning solutions one by one for script
  - `DFS` depth-first search
  - `BAB` branch-and-bound
  - `Restart, LDS`
- Interactive, visual search
  - `Gist`

Main Method: First Solution

```
…

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

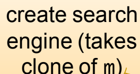
Main Method: First Solution



…

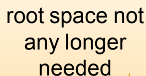
```
int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Main Method: First Solution




```
…

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Main Method: First Solution

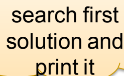


…

```
int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Main Method: First Solution

```
…

int main(int argc, char* argv[
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```



Main Method: All Solutions

```
…

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;
  }
  return 0;
}
```

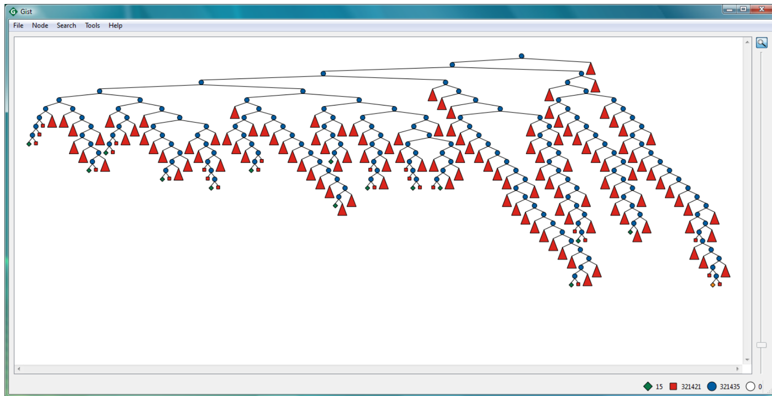
Gecode Gist

- A graphical tool for exploring the search tree
  - explore tree step by step
  - tree can be scaled
  - double-clicking node prints information: inspection
  - search for next solution, all solutions
  - …
- Best to play a little bit by yourself
  - hide and unhide failed subtrees
  - …

Main Function: Gist

```
#include <gecode/gist.hh>

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  Gist::dfs(m);
  delete m;
  return 0;
}
```

Gist Screenshot



Best Solution Search

Reminder: SMM++

- Find distinct digits for letters, such that

```
     SEND
+    MOST
---------
=   MONEY
```

and MONEY maximal

Script for SMM++

- Similar, please try it yourself at home
- In the following, referred to by `SendMostMoney`

Solving SMM++: Order

- Principle
  - for each solution found, constrain remaining search for better solution
- Implemented as additional method

```
virtual void constrain(const Space& b) {
  …
}
```

- Argument b refers to so far best solution
  - only take values from b
  - never mix variables!
- Invoked on object to be constrained

Order for SMM++

```
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[8]);

  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[8]);

  int money = (10000*b_m.val()+1000*b_o.val()+100*b_n.val()+
               10*b_e.val()+b_y.val());
  rel
  post(*this, 10000*m+1000*o+100*n+10*e+y > money);
}
```

value of any next solution

value of current best solution b

Main Method: All Solutions

```
…

int main(int argc, char* argv[]) {
  SendMostMoney* m = new SendMostMoney;
  BAB<SendMostMoney> e(m);
  delete m;
  while (SendMostMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Main Function: Gist

```
#include <gecode/gist.hh>

int main(int argc, char* argv[]) {
  SendMostMoney* m = new SendMostMoney;
  Gist::bab(m);
  delete m;
  return 0;
}
```

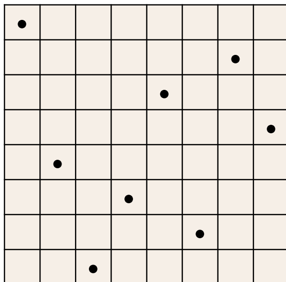
Summary: Solving

- Result-only search engines
  - `DFS, BAB`
- Interactive search engine
  - `Gist`

- Best solution search uses constrain-method for posting constraint
- Search engine independent of script and constrain-method

8-Queens

Problem Statement



- Place 8 queens on a chess board such that the queens do not attack each other
- Straightforward generalizations
  - place an arbitrary number: *n* Queens
  - place as closely together as possible

What Are the Variables?

- Representation of position on board
- First idea: two variables per queen
  - one for row
  - one for column
  - $2 \cdot n$ variables
- Insight: on each column there will be a queen!

Fewer Variables…

- Have a variable for each column
  - value describes row for queen
  - *n* variables

- Variables: $x_0, \ldots, x_7$
  where $x_i \in \{0, \ldots, 7\}$

Other Possibilities

- For each field: number of queen
  - which queen is not interesting, so…
  - n^2 variables
- For each field on board: is there a queen on the field?
  - 8×8 variables
  - variable has value 0: no queen
  - variable has value 1: queen
  - n^2 variables

Constraints: No Attack

- not in same column
  - by choice of variables
- not in same row
  - $x_i \neq x_j$ for $i \neq j$
- not in same diagonal
  - $x_i - i \neq x_j - j$ for $i \neq j$
  - $x_i - j \neq x_j - i$ for $i \neq j$
- $3 \cdot n \cdot (n - 1)$ constraints

Fewer Constraints…

- Sufficient by symmetry

  $i < j$ instead of $i \neq j$

- Constraints
  - $x_i \neq x_j$ for $i < j$
  - $x_i - i \neq x_j - j$ for $i < j$
  - $x_i - j \neq x_j - i$ for $i < j$

- $3/2 \cdot n \cdot (n - 1)$ constraints

Even Fewer Constraints

- Not same row constraint

  $x_i \neq x_j$ for $i < j$

  means: values for variables pairwise distinct
- Constraints
  - distinct(x_0, …, x_7)
  - $x_i - i \neq x_j - j$ for $i < j$
  - $x_i - j \neq x_j - i$ for $i < j$

Pushing it Further…

- Yes, also diagonal constraints can be captured by distinct constraints
  - ~~see assignment~~

```
distinct(x0,     x1,    ...,  x7)
distinct(x0-0,  x1-1, ..., x7-7)
distinct(x0+0, x1+1, ..., x7+7)
```

Script: Variables

```
Queens(void) : q(*this,8,0,7) {
  …
}
```

Script: Constraints

```
Queens(void) : q(*this,8,0,7) {
  distinct(*this, q);
  for (int i=0; i<8; i++)
    for (int j=i+1; j<8; j++) {
  rel post(*this, x[i]-i != x[j]-j);
      post(*this, x[i]-j != x[j]-i);
    }
  …
}
```

(In the code, "post" is struck through in red and "rel" is written in red beside it as the replacement.)

Script: Branching

```
Queens(void) : q(*this,8,0,7) {
  …
  branch(*this, q,
         INT_VAR_NONE,
         INT_VAL_MIN);
}
```

Good Branching?

- Naïve is not a good strategy for branching

- Try the following (see assignment)
  - first fail
  - place queen as much in the middle of a row
  - place queen in knight move fashion

Summary 8 Queens

- Variables
  - model should require few variables
  - good: already impose constraints
- Constraints
  - do not post same constraint twice
  - try to find “big” constraints subsuming many small constraints
    - more efficient
    - often, more propagation (to be discussed)

Grocery

Grocery

- Kid goes to store and buys four items
- Cashier: that makes $7.11
- Kid: pays, about to leave store
- Cashier: hold on, I multiplied!
  let me add!
  wow, sum is also $7.11
- You: prices of the four items?

Model

- Variables
  - for each item A, B, C, D
  - take values between {0, …, 711}
  - compute with cents: allows integers
- Constraints
  - A + B + C + D = 711
  - A * B * C * D = 711 * 100 * 100 * 100

The unique solution (upon the symmetry breaking of slide 87) is:
A=120, B=125, C=150, D=316.

Script

```
class Grocery : public Space {
protected:
    IntVarArray abcd;

    const int s = 711;
    const int p = s * 100 * 100 * 100;
public:
    Grocery(void) … { … }

    …
}
```

Script: Variables

```
Grocery(void) : abcd(*this,4,0,711) {
  …
}
```

Script: Sum

```
…
// Sum of all variables is s
linear(this, abcd, IRT_EQ, s);

IntVar a(abcd[0]), b(abcd[1]),
       c(abcd[2]), d(abcd[3]);
```

Script: Product

```
IntVar t1(*this,1,p);
IntVar t2(*this,1,p);
IntVar t3(*this,p,p);

mult(*this, a, b, t1);
mult(*this, c, d, t2);
mult(*this, t1, t2, t3);
```

Branching

- Bad idea: try values one by one
- Good idea: split variables
  - for variable x
  - with $m = (\min(x) + \max(x)) / 2$
  - branch $x < m$ or $x \geq m$
- Typically good for problems involving arithmetic constraints
  - exact reason needs to be explained later

Script: Branching

```
branch(*this, abcd,
       INT_VAR_NONE,
       INT_VAL_SPLIT_MIN);
```

Search Tree

- 2829 nodes for first solution
- Pretty bad…

Better Heuristic?

- Try branches in different order

  split with larger interval first
  - try: `INT_VAL_SPLIT_MAX`
- Search tree: 2999 nodes
  - worse in this case

Symmetries

- Interested in values for A, B, C, D
- Model admits equivalent solutions
  - interchange values for A, B, C, D
- We can add order A, B, C, D:

  $A \leq B \leq C \leq D$
- Called “symmetry breaking constraint”

Script: Symmetry Breaking

```
…
rel(this, a, IRT_LQ, b);
rel(this, b, IRT_LQ, c);
rel(this, c, IRT_LQ, d);
…
```

Effect of Symmetry Breaking

- Search tree size 308 nodes
- Let us try `INT_VAL_SPLIT_MAX` again
  - tree size 79 nodes!
  - interaction between branching and symmetry breaking
  - other possibility: $A \geq B \geq C \geq D$
  - we need to investigate more (later)!

Any More Symmetries?

- Observe: 711 has prime factor 79
  - that is: $711 = 79 \times 9$
- Assume: A can be divided by 79
  - add: $A = 79 \times X$
    for some finite domain var X
  - remove $A \leq B$
  - the remaining B, C, D of course can still be ordered

Any More Symmetries?

- In Gecode

```
IntVar x(*this,1,p);
IntVar sn(*this,79,79);
mult(*this, x, sn, a);
```

- Search tree 44 nodes!
  - now we are talking!

Summary: Grocery

- Branching: consider also
  - how to partition domain
  - in which order to try alternatives
- Symmetry breaking
  - can reduce search space
  - might interact with branching
  - typical: order variables in solutions
- Try to really understand problem!

Domination Constraints

- In symmetry breaking, prune solutions without interest
- Similarly for best solution search
  - typically, interested in just one best solution
  - impose constraints to prune some solutions with same "cost"

Another Observation

- Multiplication decomposed as

  $A \cdot B = T_1 \quad C \cdot D = T_2 \quad T_1 \cdot T_2 = P$

- What if

  $A \cdot B = T_1 \quad T_1 \cdot C = T_2 \quad T_2 \cdot D = P$

  - propagation changes: 355 nodes
  - propagation is not compositional!
  - another point to investigate

Magic Squares

2	9	4
7	5	3
6	1	8

Unique solution for n=3, upon the symmetry breaking of slide 99.

Magic Squares

- Find an $n \times n$ matrix such that
  - every field is integer between 1 and n^2
  - fields pairwise distinct
  - sums of rows, columns, two main diagonals are equal
- Very hard problem for large n
- Here: we just consider the case $n=3$

Model

- For each matrix field have variable x_{ij}
  - $x_{ij} \in \{1, .., 9\}$
- One additional variable s for sum
  - $s \in \{1, .., 9\times 9\}$
- All fields pairwise distinct
  - distinct(x_{ij})
- For each row i have constraint
  - $x_{i0} + x_{i1} + x_{i2} = s$
  - columns and diagonals similar

Script

- Straightforward
- Branching strategy
  - first-fail
  - split again: arithmetic constraints
  - try to come up with something that is really good!
- Generalize it to arbitrary *n*

Symmetries

- Clearly, we can require for first row that first and last variable must be in order
- Also, for opposing corners
- In all (other combinations possible)
  - $x_{00} < x_{02}$
  - $x_{02} < x_{20}$
  - $x_{00} < x_{22}$

Important Observation

- We know the sum of all fields

  $1 + 2 + \ldots + 9 = 9(9+1)/2 = 45$

- We "know" the sum of one row

  s

- We know that we have three rows

  $3 \times s = 45$

Implied Constraints

- The constraint model already implies

  $3 \times s = 45$

  - implies   solutions are the same
- However, adding a propagator for the constraint drastically improves propagation
- Often also: redundant or implied constraint

Effect

Simple model	92 nodes
Symmetry breaking	29 nodes
Implied constraint	6 nodes

Summary: Magic Squares

- Add implied constraints
  - are implied by model
  - increase constraint propagation
  - reduce search space
  - require problem understanding
- Also as usual
  - break symmetries
  - choose appropriate branching

Outlook…

- Common modeling principles
  - what are the variables
  - finding the constraints
  - finding the propagators
  - implied (redundant) constraints
  - finding the branching
  - symmetry breaking

Modeling Strategy

- Understand problem
  - identify variables
  - identify constraints
  - identify optimality criterion
- Attempt initial model simple?
  - try on examples to assess correctness
- Improve model much harder!
  - scale up to real problem size