Constraint Programming

Marco Kuhlmann & Guido Tack

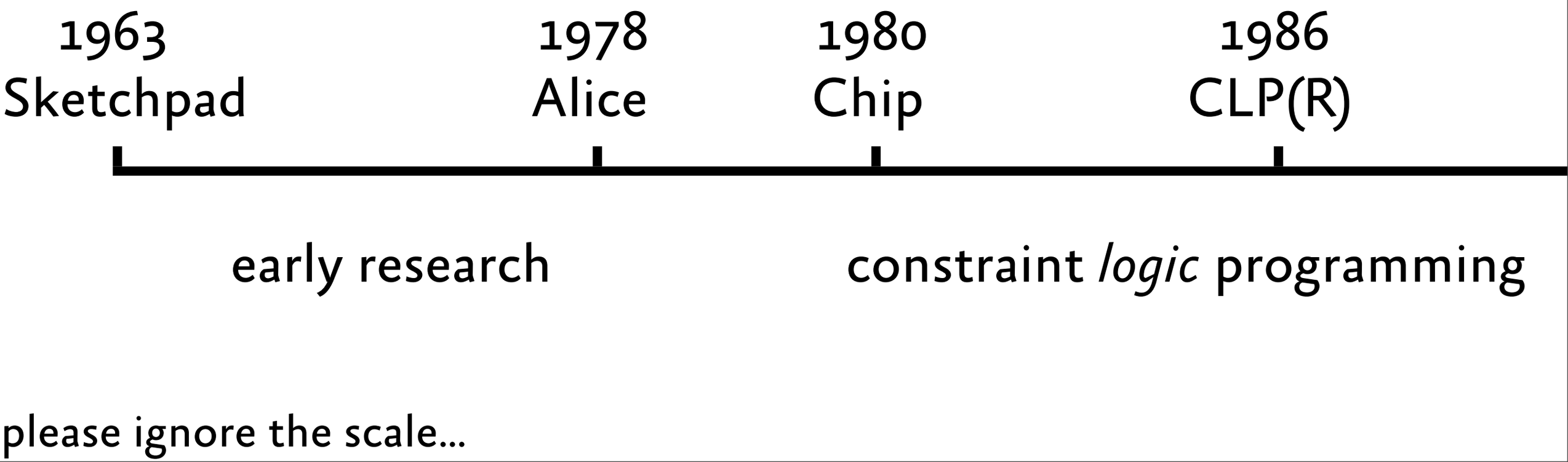
Lecture 2

Today:
History and Practice

History and Practice

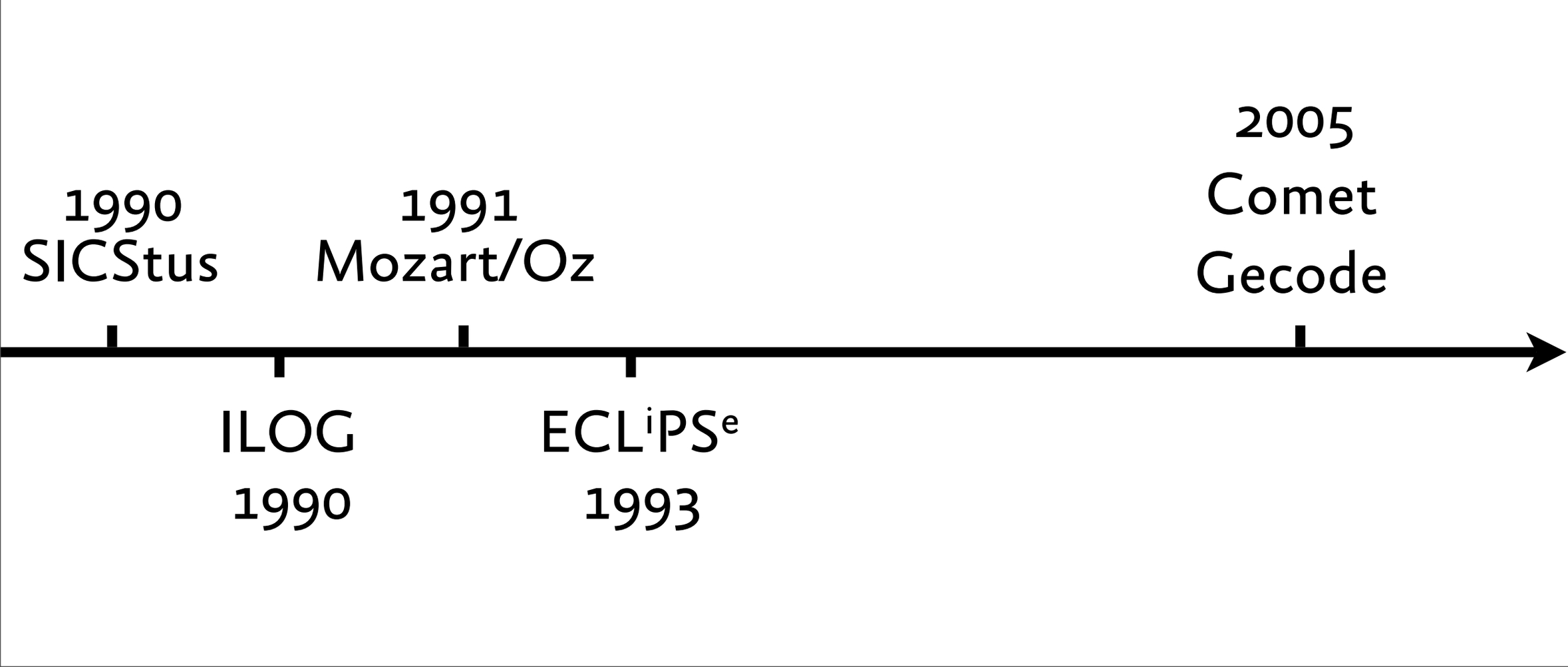
- **Part I: Short historical overview**
  - where does CP come from?
- **Part II: Constraint Programming with Gecode/J**
  - give you intuition about what's under the hood
  - scratch all topics we will discuss in this course

Historical notes

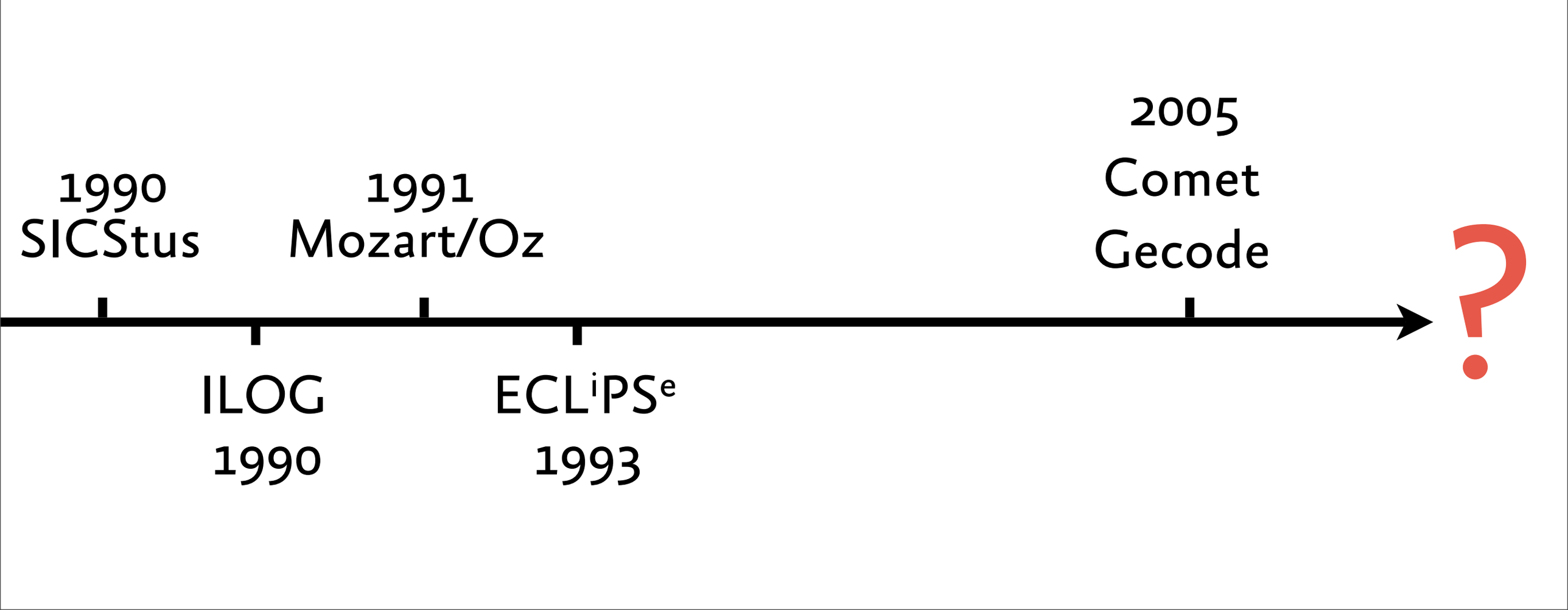


please ignore the scale...

Historical notes



Historical notes



Logic Programming

```
foo(a).
foo(b).
g(X) :- X=[Y, Z], foo(Y), foo(Z).
```

Logic Programming

```
foo(a).
foo(b).
g(X) :- X=[Y, Z], foo(Y), foo(Z).


| ?- g(X).
X = [a,a] ? ;
X = [a,b] ? ;
X = [b,a] ? ;
X = [b,b]
```

Constraint Logic Programming

```
foo(X) :- fd_domain(X, 1, 3).
g(Y,Z) :- foo(Y), foo(Z), Y #< Z,
          fd_labeling([Y,Z]).
```

Constraint Logic Programming

```
foo(X) :- fd_domain(X, 1, 3).
g(Y,Z) :- foo(Y), foo(Z), Y #< Z,
          fd_labeling([Y,Z]).
| ?- g(X,Y).
X = 1
Y = 2 ? ;
X = 1
Y = 3 ? ;
X = 2
Y = 3
```

Constraint Logic Programming

```
foo(X) :- fd_domain(X, 1, 3).
g(Y,Z) :- foo(Y), foo(Z), Y #< Z,
          fd_labeling([Y,Z]).
| ?- g(X,Y).
X = 1
Y = 2 ? ;
X = 1
Y = 3 ? ;
X = 2
Y = 3
```

Model:

constraint program

=

logic program

=

logical formula

Constraint Logic Programming

```
foo(X) :- fd_domain(X, 1, 3).
g(Y,Z) :- foo(Y), foo(Z), Y #< Z,
          fd_labeling([Y,Z]).
| ?- g(X,Y).
X = 1
Y = 2 ? ;
X = 1
Y = 3 ? ;
X = 2
Y = 3
```

Languages/Systems:

GNU Prolog, BProlog, SICStus Prolog, ECL^{i}PS^{e}

Concurrent Constraint Programming

Concurrent Constraint Programming

- Von-Neumann architecture: store *values*
  - operations: `read` and `write`

Concurrent Constraint Programming

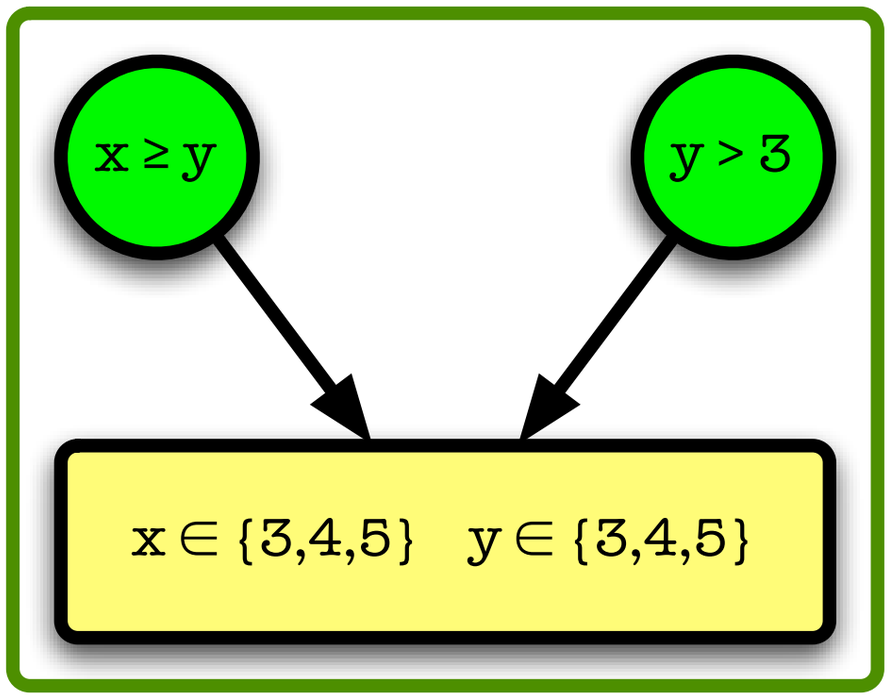
- Von-Neumann architecture: store *values*
  - operations: `read` and `write`
- cc architecture: store *constraints*
  - operations: `ask` and `tell`
  - communication through variables

Concurrent Constraint Programming

- Von-Neumann architecture: store *values*
  - operations: `read` and `write`
- cc architecture: store *constraints*
  - operations: `ask` and `tell`
  - communication through variables
- Languages/systems:
  - cc(FD), AKL, Mozart/Oz

Concurrent Constraint Programming

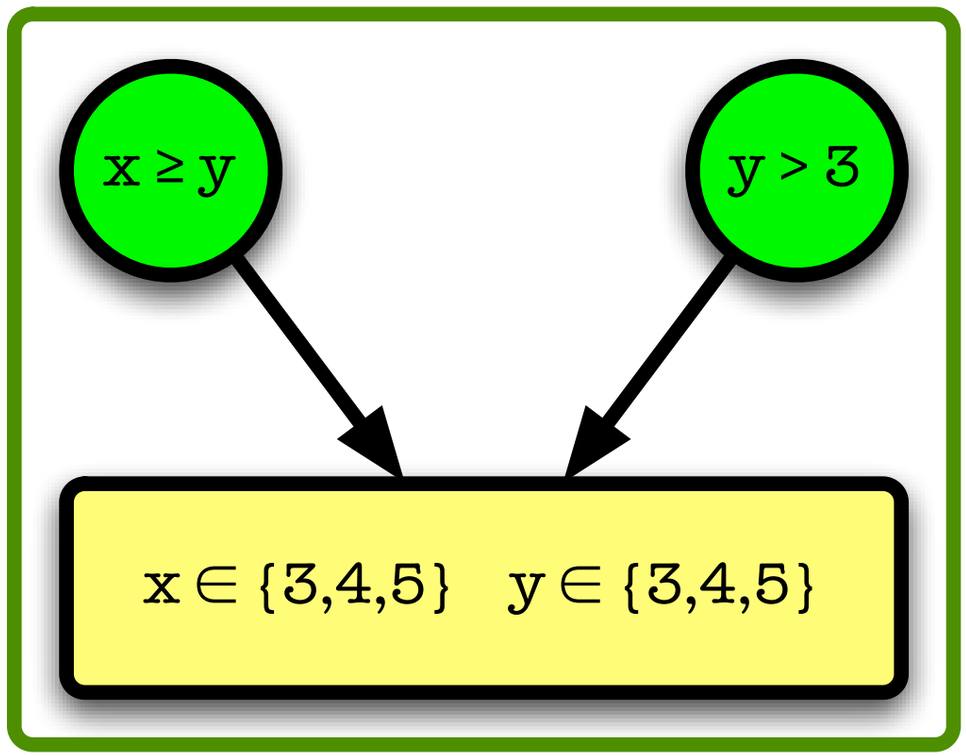
- Von-Neumann architecture: store *values*
  - operations: `read` and `write`
- cc architecture: store *constraints*
  - operations: `ask` and `tell`
  - communication through variables
- Languages/systems:
  - cc(FD), AKL, Mozart/Oz



Constraint Programming with Gecode/J

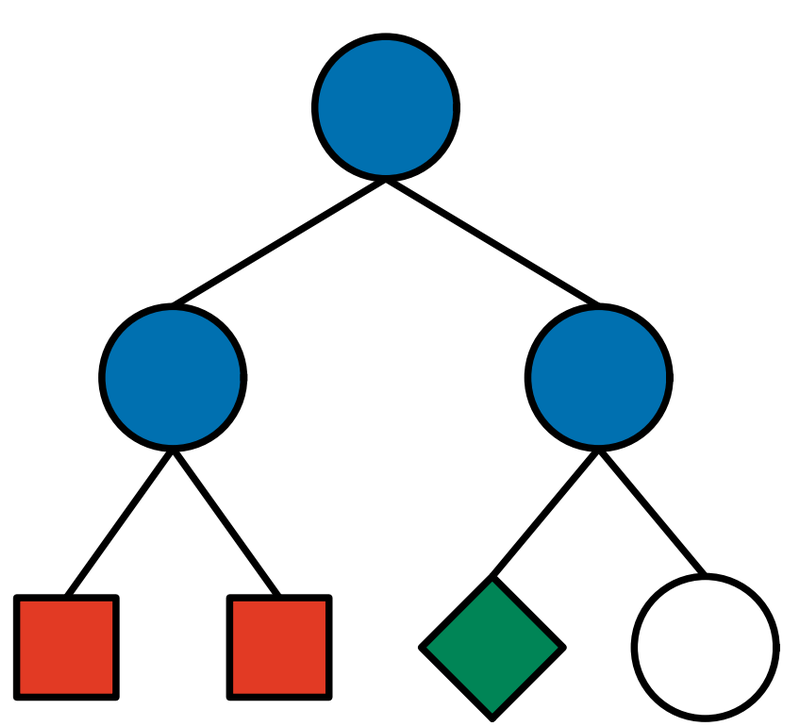
- **Quick reminder of last lecture**
- **Walk-through for *Send More Money***
- **Some modeling techniques**
- **Presentation of first graded lab**

Computation Space

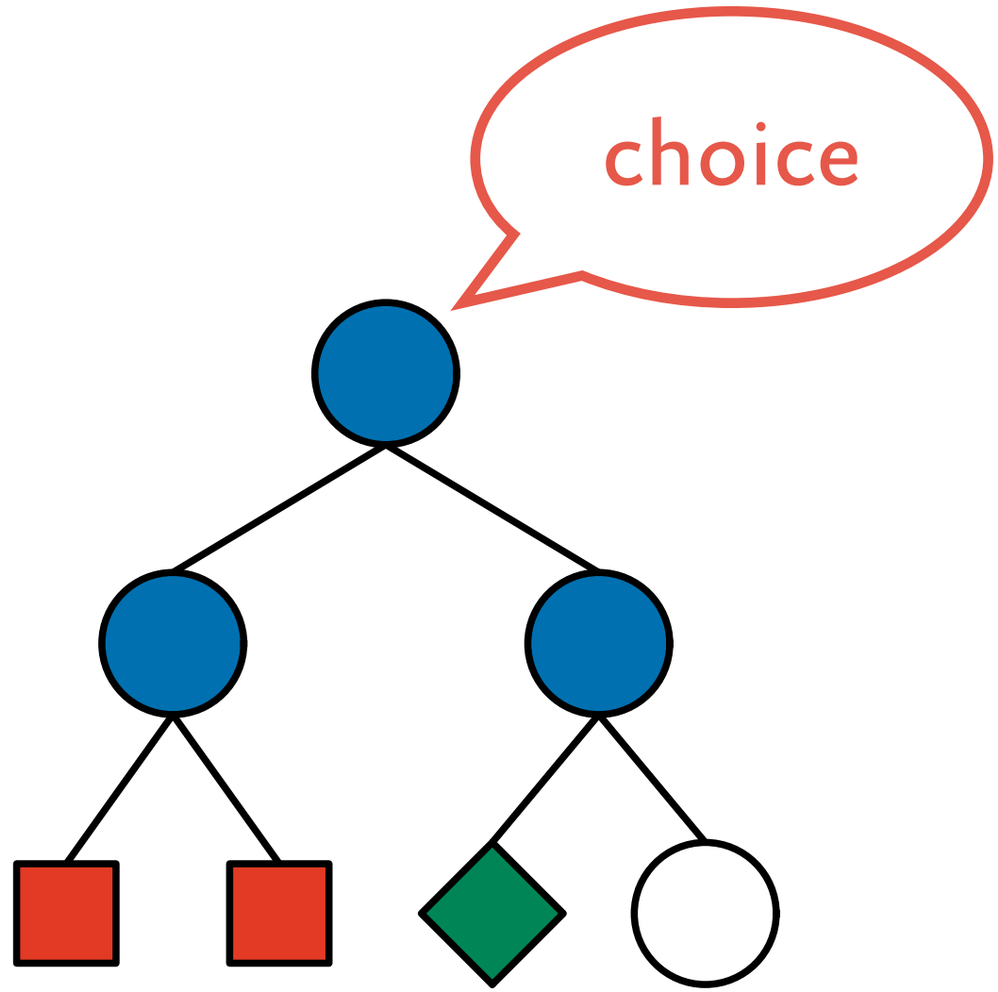


constraint store with connected propagators

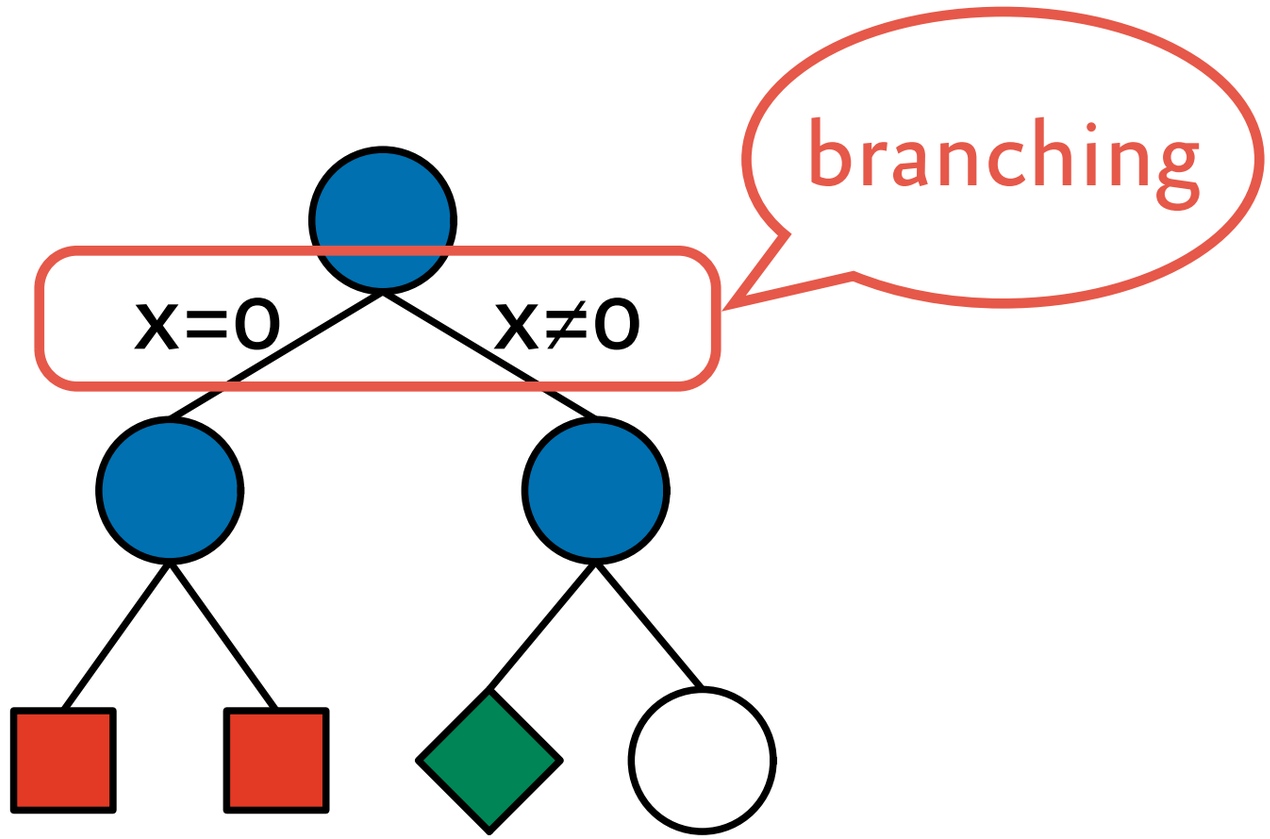
Search



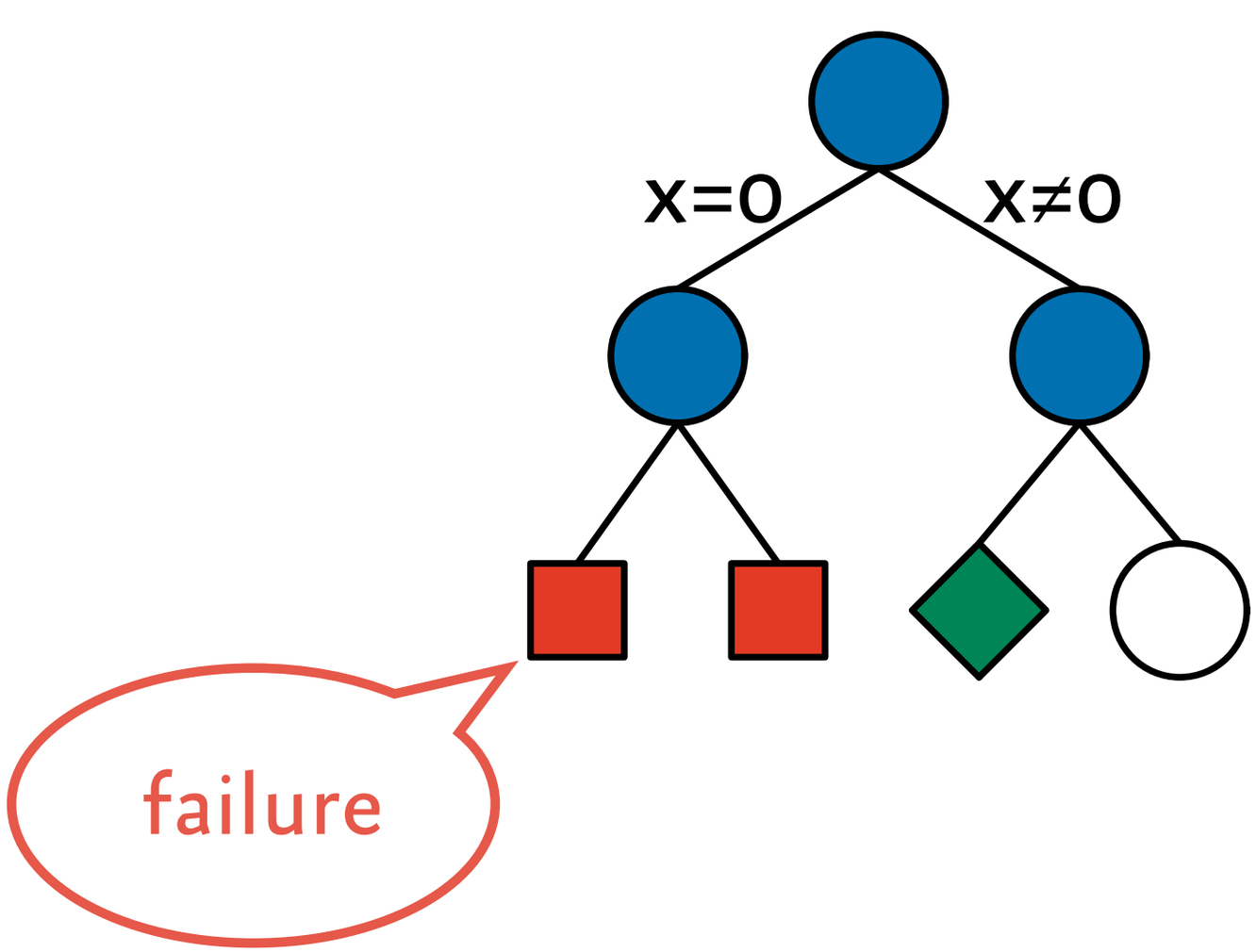
Search



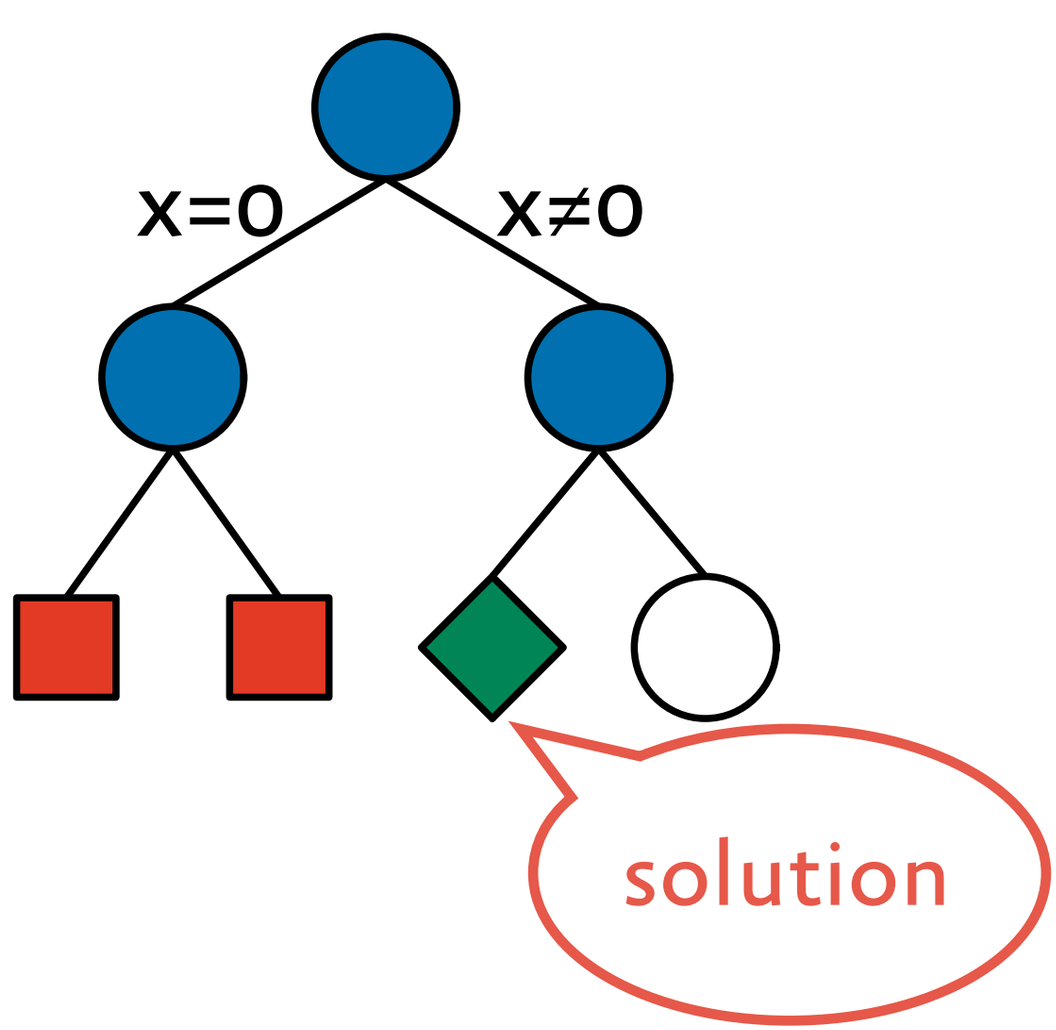
Search



Search



Search



Send More Money

- variables: $S,E,N,D,M,O,R,Y \in \{0,\ldots,9\}$
- constraints: $S \neq 0,\ M \neq 0$

  $\text{distinct}(S,E,N,D,M,O,R,Y)$

$$
\begin{array}{r}
1000\times S + 100\times E + 10\times N + D \\
+\ 1000\times M + 100\times O + 10\times R + E \\
\hline
= 10000\times M + 1000\times O + 100\times N + 10\times E + Y
\end{array}
$$

Modelling in Gecode/J

- **Implement model as a *script***
  - declare variables
  - post constraints (create propagators)
  - define branching
- **Solve script**
  - basic search strategy (DFS)
  - interactive, graphical search tool (Gist)

Script: Overview

- **Inherit from class Space**
- **Constructor**
  - initialize variables
  - post propagators
  - define branching
- **Copy constructor**
  - copy a space
- **Main function**
  - invoke search engine

Script: Structure

```
import static org.gecode.Gecode.*;
import static org.gecode.GecodeEnumConstants.*;
import org.gecode.*;

public class Money extends Space {
  public VarArray<IntVar> letters;
  public Money() {...}
  public Money(Boolean share, Money money) {...}
  public static void main(String[] args) {...}
}
```

Script: Structure

import all we need

```
import static org.gecode.Gecode.*;
import static org.gecode.GecodeEnumConstants.*;
import org.gecode.*;

public class Money extends Space {
  public VarArray<IntVar> letters;
  public Money() {...}
  public Money(Boolean share, Money money) {...}
  public static void main(String[] args) {...}
}
```

Script: Structure

```
import static org.gecode.Gecode.*;
import static org.gecode.GecodeEnumConstants.*;
import org.gecode.*;

public class Money extends Spac
  public VarArray<IntVar> letters;
  public Money() {...}
  public Money(Boolean share, Money money) {...}
  public static void main(String[] args) {...}
}
```

problem variables

Script: Structure

```
import static org.gecode.Gecode.*;
import static org.gecode.GecodeEnumConstants.*;
import org.gecode.*;

public class Money extends S
  public VarArray<IntVa
  public Money() {...}
  public Money(Boolean share, Money money) {...}
  public static void main(String[] args) {...}
}
```

constructor

Script: Structure

```
import static org.gecode.Gecode.*;
import static org.gecode.GecodeEnumConstants.*;
import org.gecode.*;

public class Money extends Space {
  public VarArray<IntVar> lett
  public Money() {...}
  public Money(Boolean share, Money money) {...}
  public static void main(String[] args) {...}
}
```

copy constructor

Script: Structure

```
import static org.gecode.Gecode.*;
import static org.gecode.GecodeEnumConstants.*;
import org.gecode.*;

public class Money extends Space {
  public VarArray<IntVar> letters;
  public Money() {...}
  public Money(Boolean share, Money money) {...}
  public static void main(String[] args) {...}
}
```

Script: Constructor

```
public Money() {
   // Call superclass constructor
   super();
   // Initialize variables
   letters = new VarArray<IntVar>(this,
                                  8,
                                  IntVar.class,
                                  0, 9);
   ...
}
```

Script: Constructor

```
...
// Refer to the letters by name
IntVar s = letters.get(0);
IntVar e = letters.get(1);
IntVar n = letters.get(2);
IntVar d = letters.get(3);
IntVar m = letters.get(4);
IntVar o = letters.get(5);
IntVar r = letters.get(6);
IntVar y = letters.get(7);
...
```

Script: Constructor

```
...
// Initial letters non-zero
rel(this, s, IRT_NQ, 0);
rel(this, m, IRT_NQ, 0);
// IRT: Integer relation type
...
```

Posting Constraints

- **Defined in the class `org.gecode.Gecode`**
  - accessed with `import static`
- **Check the documentation**
- **All constraints take a space as first argument**
  - where is the constraint to be installed?
  - Scripts are subclasses of Space!

Linear equation constraints

- Propagator for equations of the form

$$\sum_{i=1}^{n} a_i x_i = d$$

- Specified as arrays

```
int[] a
```

```
VarArray<IntVar> x
```

- Supported relations:

```
IRT_EQ, IRT_NQ, IRT_LE, IRT_GR, IRT_LQ, IRT_GQ
```

Script: Constructor

```
...
// Post linear equation
int a[]={              1000,    100,     10,       1,
                       1000,    100,     10,       1,
           -10000,  -1000,   -100,    -10,      -1};
VarArray<IntVar> x =
    new VarArray<IntVar>(    s, e, n, d,
                             m, o, r, e,
                          m, o, n, e, y);
linear(this, a, x, IRT_EQ, 0);
...
```

Script: Constructor

```
...
// Letters take distinct values
distinct(this, letters);

// Find values using first-fail
branch(this, letters,
       BVAR_SIZE_MIN,
       BVAL_MIN);

...
```

Branching

- **Choose variable**
  - smallest domain size: `BVAR_SIZE_MIN`
  - smallest minimum: `BVAR_MIN_MIN`
  - given order: `BVAR_NONE`
- **Choose value**
  - try smallest value: `BVAL_MIN`
  - split (lower first) `BVAL_SPLIT_MIN`

Script: Copying

```
public Money(Boolean share, Money m) {
  super(share, m);
  letters = new VarArray<IntVar>(this,
                                 share, m.letters);
}
```

Script: Copying

```
public Money(Boolean share, Mo
  super(share, m);
  letters = new VarArray<IntVar>(this,
                                 share, m.letters);
}
```

copy all variables you need for output!

Script: Copying

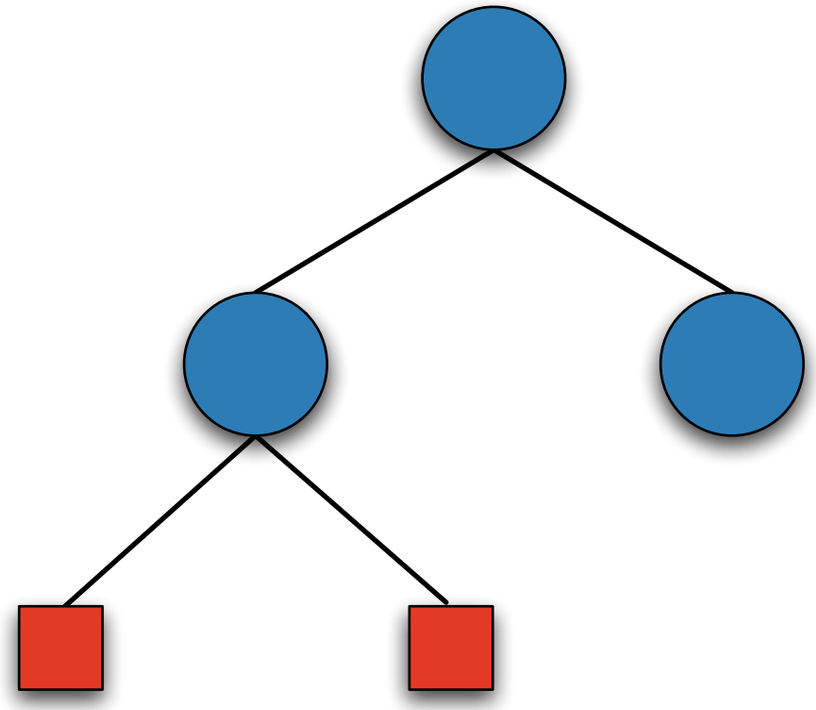
```
public Money(Boolean share, Money m) {
  super(share, m);
  letters = new VarArray<IntVar>(this,
                                 share, m.letters);
}
```

Script: Copying

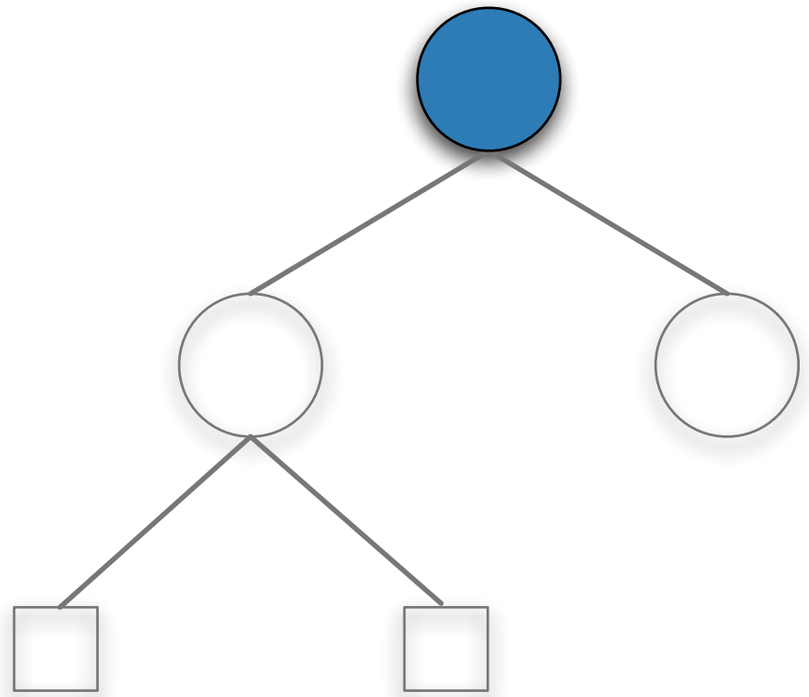
```
public Money(Boolean share, Money m) {
   super(share, m);
   letters = new VarArray<IntVar>(this,
                                  share, m.letters);
}
```

copying of single variables also possible!

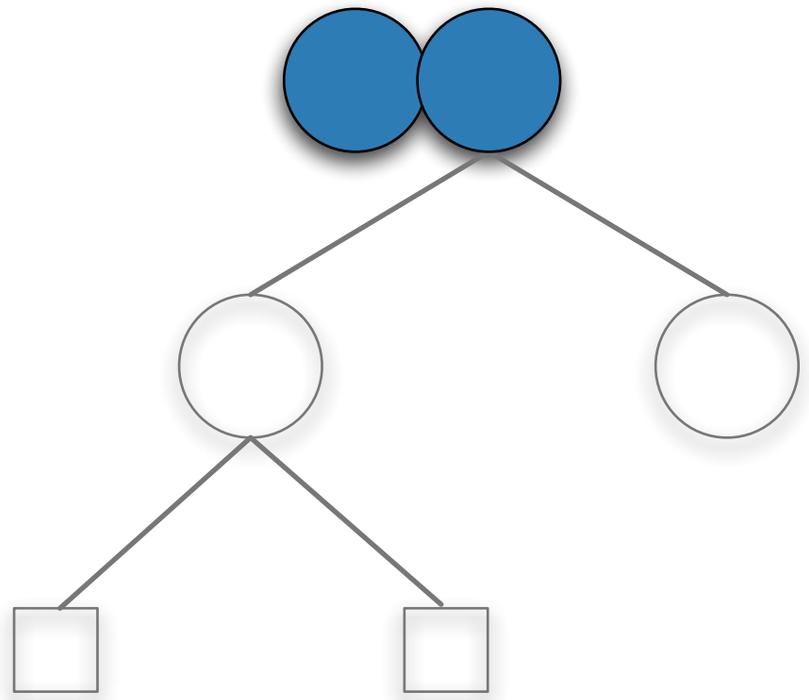
Why copy? Search!



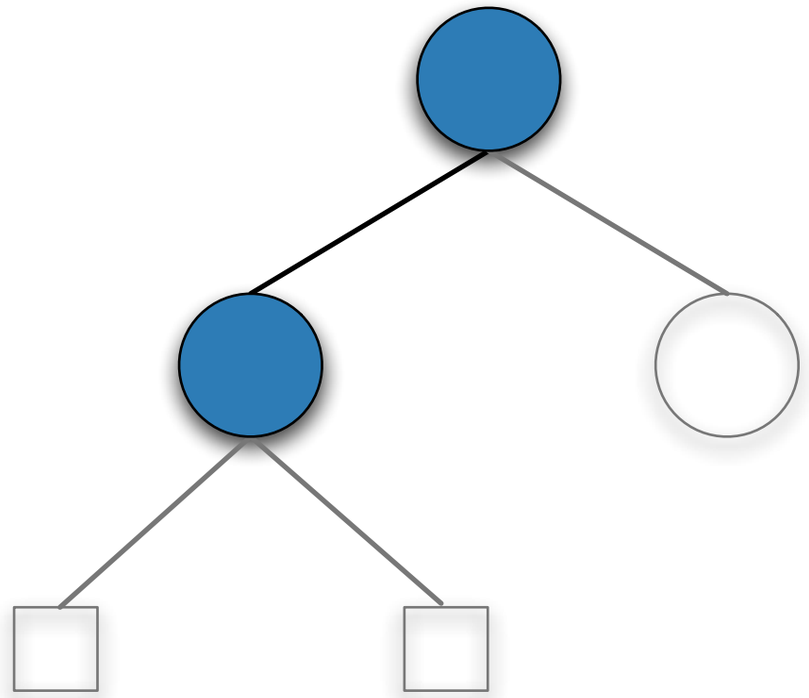
Why copy? Search!



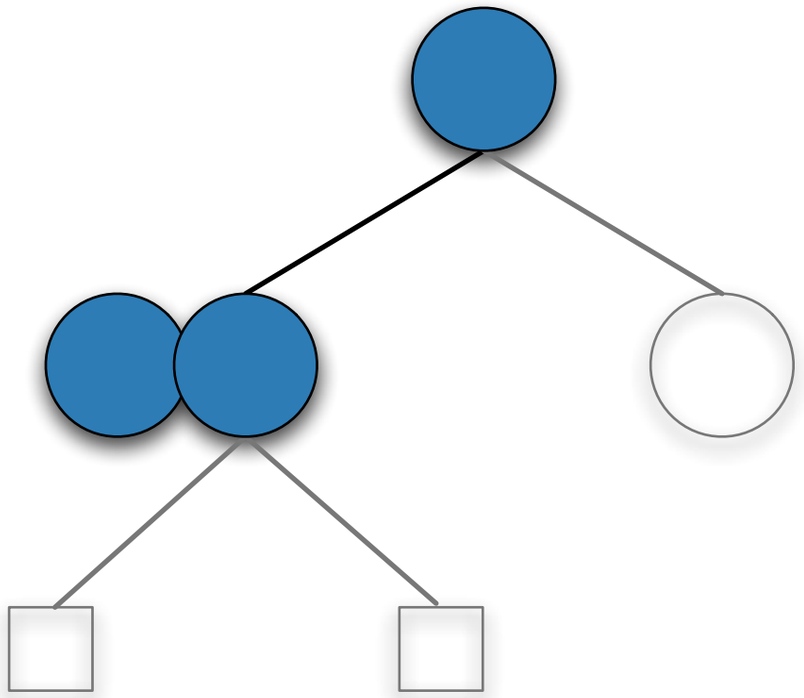
Why copy? Search!



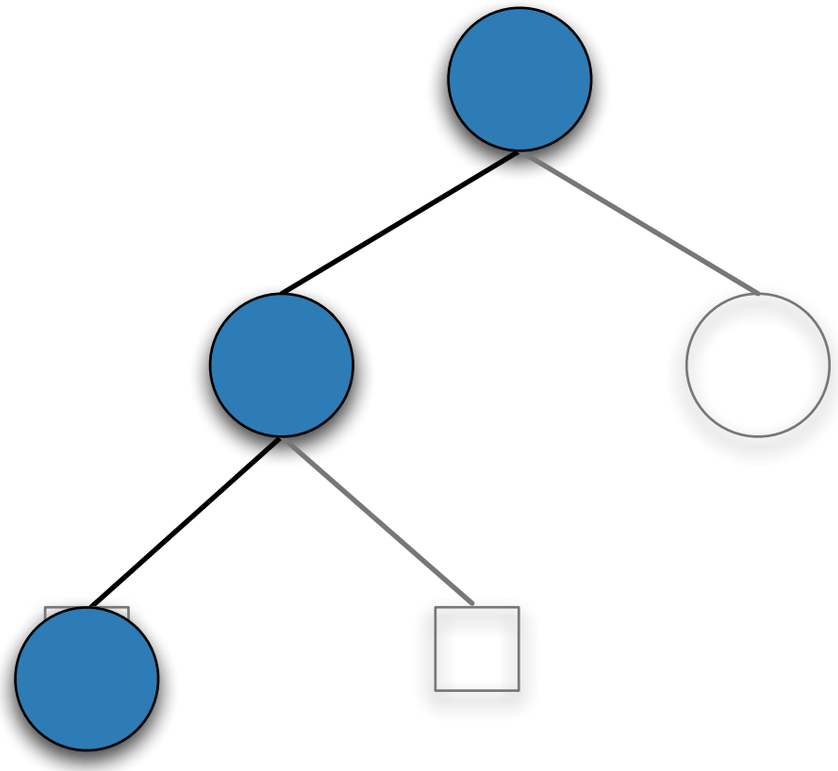
Why copy? Search!



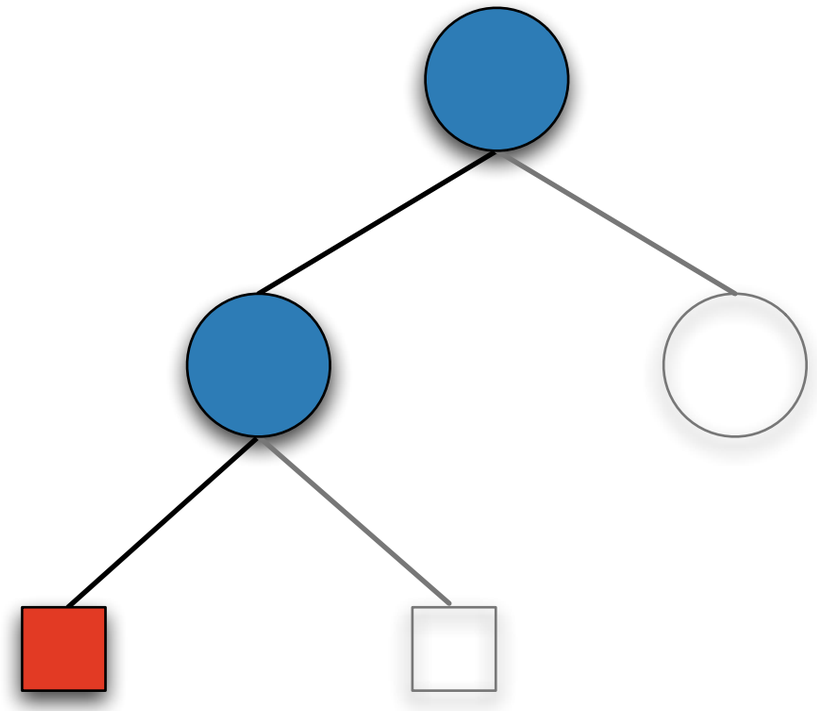
Why copy? Search!



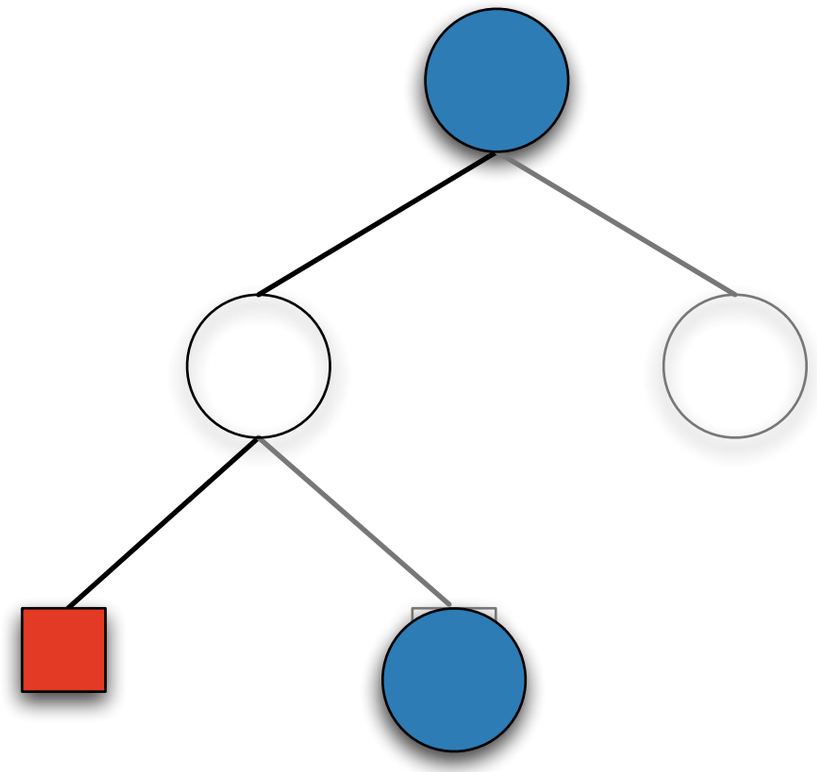
Why copy? Search!



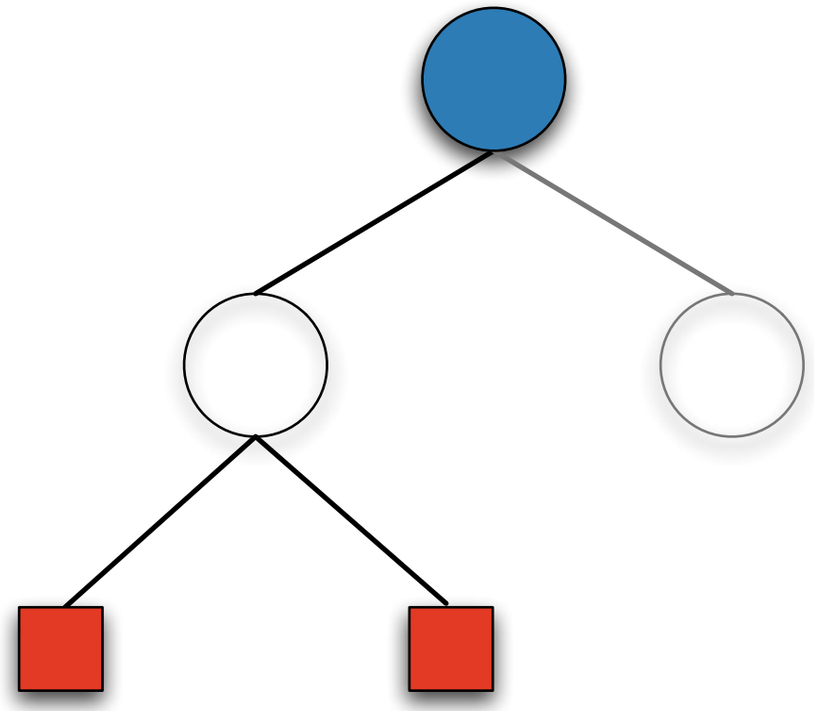
Why copy? Search!



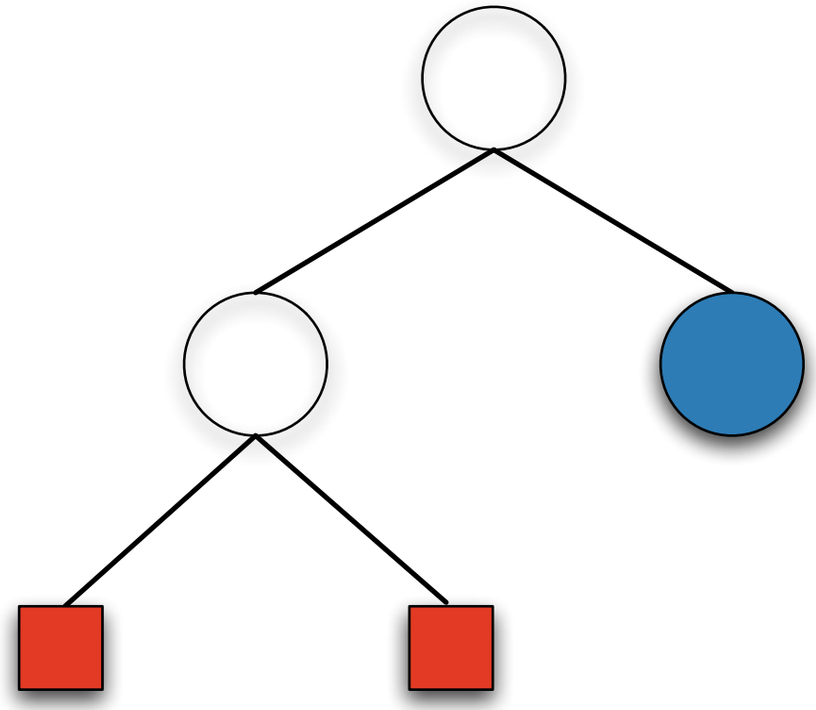
Why copy? Search!



Why copy? Search!



Why copy? Search!



Script

```
import static org.gecode.Gecode.*;
import static org.gecode.GecodeEnumConstants.*;
import org.gecode.*;

public class Money extends Space {
  public VarArray<IntVar> letters;
  public Money() {...}
  public Money(Boolean share, Money money) {...}
  public static void main(String[] args) {...}
}
```

Solving

- **Hidden in `Options` class, but...**
- **Search engines:**
  - `DFSSearch` (depth first search)
  - `BABSearch` (branch-and-bound search)
- **Interactive search tool**
  - Gist

First Solution Search

```
public static void main(String[] args)
{
  Money m = new Money();
  DFSSearch s = new DFSSearch(m);
  Money sol = (Money) s.next();
  if (sol != null) {
    System.out.println(sol.toString());
  }
}
```

First Solution Search

```
public static void main(S
{
  Money m = new Money();
  DFSSearch s = new DFSSearch(m);
  Money sol = (Money) s.next();
  if (sol != null) {
    System.out.println(sol.toString());
  }
}
```

create root space

First Solution Search

```
public static void main(Strin
{
  Money m = new Money();
  DFSSearch s = new DFSSearch(m);
  Money sol = (Money) s.next();
  if (sol != null) {
    System.out.println(sol.toString());
  }
}
```

create
search engine

First Solution Search

```
public static void main(String[] args)
{
  Money m = new Money();
  DFSSearch s = new DFSSearc
  Money sol = (Money) s.next();
  if (sol != null) {
    System.out.println(sol.toString());
  }
}
```

search

All Solution Search

```
public static void main(String[] args)
{
  Money m = new Money();
  DFSSearch s = new DFSSearch(m);
  Money sol = (Money) s.next();
  while (sol != null) {
    System.out.println(sol.toString());
    sol = (Money) s.next();
  }
}
```

Gecode/J Gist

- **Graphical search tree explorer**
  - explore step by step
  - search next solution, search all solutions
  - "inspect" nodes (double-click)
  - hide failed subtrees
  - ...
- **Try it out!**

Invoking Gist

```
import org.gecode.gist.*;

public static void main(String[] args)
{
  Money m = new Money();
  Gist g = new Gist(m);
  g.exploreOne();
  // Also possible: g.exploreAll()
}
```

Best Solution Search

- **Reminder: SMM+**

  ```
    SEND
  +  MOST
  =  MONEY
  ```

- **Constraints:**

  Find distinct numbers such that the equation holds and MONEY is maximal

SMM+: Order

- **Branch-and-bound principle:**

  after each solution, constrain remaining search to find only better solutions

- **Script provides method:**

```
public void constrain(Space s) { ... }
```

- **Invoked on Space object to be constrained**
- **`Space s:` so far best solution**
  - Only use *values* from `s`! Never mix variables!

SMM+: constrain method

```
IntVar money;
public SMM() {
   ...
   int [] c = {10000,1000,100,10,1};
   VarArray<IntVar> l = {m,o,n,e,y}; // pseudocode!
   linear(this, c, l, IRT_EQ, money);
   ...
}

public void constrain(Space bestSol) {
   SMM smm = (SMM) bestSol;
   rel(this, money, IRT_GR, smm.money.val());
}
```

SMM+: main method

```
public static void main(String[] args) {
   SMM m = new SMM();

   BABSearch s = new BABSearch(m);
   SMM best = null;
   SMM sol = (SMM) s.next();
   while (sol != null) {
      best = sol; sol = (SMM) s.next();
   }
   if (best != null) {
      System.out.println(best.toString());
   }
}
```

SMM+: Gist

```
public static void main(String[] args)
{
  SMM m = new SMM();

  Gist g = new Gist(m, true);
  g.exploreAll();
}
```

SMM+: Gist

```
public static void main(String[] args)
{
  SMM m = new SMM();

  Gist g = new Gist(m, true);
  g.exploreAll();
}
```

switch on b&b

Solving: Summary

- **Non-interactive search:**

  `DFSSearch`, `BABSearch`

- **Interactive search:**

  Gist

- **Best solution search:** provide constrain method

- **Search engines independent of script**

Some modeling techniques

- **Global constraints**
- **Symmetry breaking**
- **Reification**

Global constraints

- **Classic example:**

  $x,y,z \in \{1,2\}, \quad x \neq y, x \neq z, y \neq z$

- **No solution!**
- **But: each individual constraint still satisfiable!**

  ⇒ no propagation possible!

- **Solution:** look at several constraints at once

  distinct(x,y,z)

Distinct

- **Remember last exercise for Sudoku?**

  "Experiment with different … propagator strengths."

- `ICL_VAL`: same as x ≠ y, x ≠ z, y ≠ z
- `ICL_BND`: stronger
- `ICL_DOM`: strongest possible propagation

Grocery Puzzle

- A kid buys four items in a grocery store.
- Cashier: "That's €7,11.

  Hold on! I multiplied instead of adding.

  Wow, the sum is also €7,11!"
- What are the individual prices of the four items?

Grocery Puzzle

- **Variables:**

  A,B,C,D ∈ {1, …, 711} (price in cents)

- **Constraints:**
  - A+B+C+D = 711
  - A*B*C*D = 711 * 100 * 100 * 100

Grocery Puzzle

- **Branching:**
  - Bad idea: try values one by one
  - Better: split domains
- **Typically good strategy for problems with arithmetic constraints**
- **Still:**

Grocery Puzzle: Symmetries

- **Many solutions are equivalent!**
  - swap values of A,B,C,D
- **Let's order them:**
  - $A \leq B \leq C \leq D$
- **This is called**

Symmetry Breaking

- **So let's see...**

Grocery Puzzle: Symmetries

- **Symmetry breaking also works on symmetric failure!**
- **Example:**
  - Suppose A,B,C have been assigned 1,2,3 by search.
  - We find no value for D such that we get a solution.
  - No need to try A=2, B=1, C=3!
- **Ruled out by $A \leq B \leq C \leq D$**

Grocery Puzzle: More Symmetries

- **Observation: 711 has prime factor 79**

  711 = 79 × 9

- **So assume**

  A = 79 × X (for a "fresh" variable X)

- **Remove A ≤ B**

- **And now...**

Reified constraints

- **Constraints are in a big conjuntion**
- **How about disjunctive constraints?**

$$A+B=C \lor C=0$$

- **Solution: *reify* the constraints:**

$$(A+B=C \Leftrightarrow b0) \land$$

$$(C=0 \Leftrightarrow b1) \land$$

$$(b0 \lor b1 \Leftrightarrow true)$$

Reified constraints

```
int[] ones = [1,1,-1];
VarArray<IntVar> xs = {a,b,c};
BoolVar b0(this);
BoolVar b1(this);

linear(this, ones, xs, IRT_EQ, 0, b0);
rel(this, c, IRT_EQ, 0, b1);
bool_or(this, b0, b1, true);
```

What we've learned today

What we've learned today

- Use **scripts** to model constraint problems

What we've learned today

- Use **scripts** to model constraint problems
- **Posting a propagator** installs it inside a space

What we've learned today

- Use **scripts** to model constraint problems
- **Posting a propagator** installs it inside a space
- We need **copying** for backtracking during search

What we've learned today

- Use **scripts** to model constraint problems
- **Posting a propagator** installs it inside a space
- We need **copying** for backtracking during search
- **Global constraints** can provide stronger inferences

What we've learned today

- Use **scripts** to model constraint problems
- **Posting a propagator** installs it inside a space
- We need **copying** for backtracking during search
- **Global constraints** can provide stronger inferences
- **Symmetry breaking** is essential for some problems

What we've learned today

- Use **scripts** to model constraint problems
- **Posting a propagator** installs it inside a space
- We need **copying** for backtracking during search
- **Global constraints** can provide stronger inferences
- **Symmetry breaking** is essential for some problems
- **Reification** allows for arbitrary Boolean combinations of constraints

Graded lab: temporal relations

- Given:
  - a number of events
  - constraints between events:
    - *a* ends before *b*
    - *a* and *b* do not overlap
    - *a* happens during *b*
    - ...

Graded lab: temporal relations

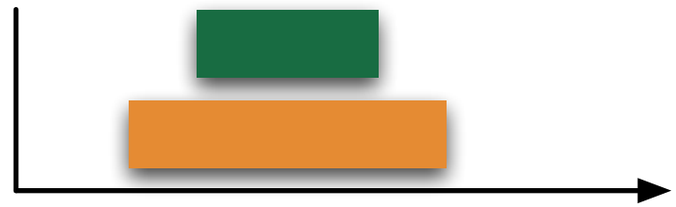
- Given:
  - a number of events
  - constraints between events:
    - *a* ends before *b*
    - *a* and *b* do not overlap
    - *a* happens during *b*
    - ...

Graded lab: temporal relations

- Given:
  - a number of events
  - constraints between events:
    - *a* ends before *b*
    - *a* and *b* do not overlap
    - *a* happens during *b*
    - ...

Graded lab: temporal relations

- Given:
  - a number of events
  - constraints between events:
    - *a* ends before *b*
    - *a* and *b* do not overlap
    - *a* happens during *b*
    - ...



Graded lab: temporal relations

- Data structure:

```
enum Relation { PRECEDES, MEETS, OVERLAPS, DURING,
                STARTS, FINISHES, EQUALS};
class Item {
  String act1; Relation rel; String act2;
}
Vector<Vector<Item>> problem;
```

Graded lab: temporal relations

A precedes **B**

A starts **B**

A meets **B**

A finishes **B**

A overlaps **B**

A equals **B**

A during **B**

Graded lab: temporal relations

- **Your task:**
  - create script that solves temporal relation problems
- **If you have too much time:**
  - create visualization for the solutions
- **Submit**
  - by Monday, April 30, 24:00 MESZ
  - by email to tack@ps.uni-sb.de

Thanks for your attention!
See you on Thursday.