



Constraint Programming with COMET

Laurent Michel
Pascal Van Hentenryck

ldm@engr.uconn.edu
pvh@cs.brown.edu



Overview

- The COMET Platform
- Core Language
- The CP Solver
 - Declarative Model
 - Search Procedures
- Demo



COMET

- **An optimization platform**

- Constraint-based Local Search (CBLS)
- Constraint Programming (CP)
- Mathematical programming (MP)

- **Availability**

- Windows 32
- MacOS 32/64
- Linux 32/64



Integrating Code with COMET

- **Options available**

- Extend COMET in COMET
 - **User defined constraints** (in CBLS and FD)
- Extend COMET in C++
 - **Call your C++ code from COMET.** Plugin architecture.
- **Embed** COMET in C++
 - **Call COMET from C++**



Integrating Data Sources with COMET

- **Database connectivity**
 - ODBC 2.0 (on all platforms)
- **Data files**
 - XML reading/writing



Solving with COMET

- **Search**

- a unique search language for CBLS, CP, MP

- **Hybridization**

- Solvers are first-class objects



Hybrids 1

- **Two LP/MIP Solvers**
 - Ipsolve
 - coin-Clp
- **Techniques supported through model composition**
 - Model chaining
 - Column generation
 - Benders decomposition



Hybrids 2

- **Combine CP + LS**

- LS for high-quality solutions quickly (and speed up the CP proof)
- CP for optimality proof - completeness

- **Composition?**

- Sequential
- Parallel

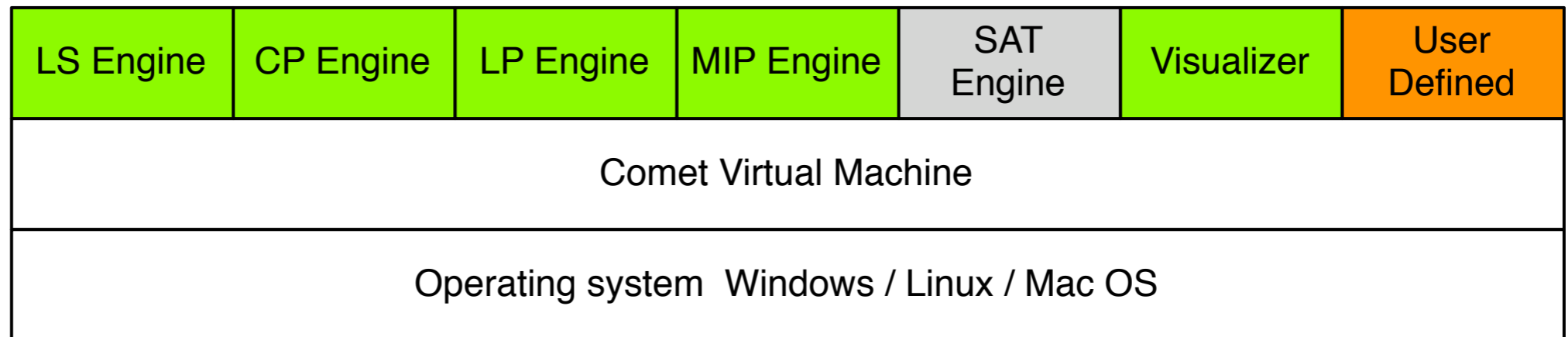
- **Communication?**

- Bounds
- Actual solution, frequencies,



Architecture

Loadable plugins





Basic Language support

- You can define
 - Classes
 - Functions
 - Interfaces
- All the traditional C++/Java - like statements
- Parameter passing is by value
 - Integer, Float, Boolean classes like in Java
- IO
 - stream-based (cin/cout) like in C++



Data support

- **Data support**

- array, matrices, sets, stack, queues, dictionaries

- **Expressions**

- Rich expression language with aggregates for arithmetic and sets

```
int x = sum(i in R) x[i];  
int y = prod(i in R) x[i];  
set{int} a = setof(i in R) (x[i]i%2==0);  
set{int} b = collect(i in R) x[i];
```

- **Slicing**

```
int mx[i in 1..10,j in 1..10] = i * 10 + j;  
int []col3 = all(i in 1..10) mx[i,3];  
int []row4 = all(i in 1..10) mx[4,i];  
int []diag = all(i in 1..10) mx[i,i];
```



Extra Control: Forall Loops

- Basic
- With ordering

```
forall(i in S)  
  BLOCK
```

```
forall(i in S : p(i))  
  BLOCK
```

```
forall(i in S : p(i)) by (f(i))  
  BLOCK
```



Extra Control: Branching - Selectors

- Randomized, Minimum, Maximum
- Semi-greedy

```
select(i in S)  
BLOCK
```

```
selectMin(i in S)(f(i))  
BLOCK
```

```
selectMax(i in S)(f(i))  
BLOCK
```

```
selectMin[k](i in S)(f(i))  
BLOCK
```

```
selectMax[k](i in S)(f(i))  
BLOCK
```

```
select(i in S : p(i))  
BLOCK
```

```
selectMin(i in S : p(i))(f(i))  
BLOCK
```

```
selectMax(i in S : p(i))(f(i))  
BLOCK
```

```
selectMin[k](i in S : p(i))(f(i))  
BLOCK
```

```
selectMax[k](i in S : p(i))(f(i))  
BLOCK
```



Extra Control: Branching - Selectors

- Randomized, Minimum, Maximum
- Semi-greedy

```
select(i in S)  
BLOCK
```

```
select(i in S : p(i))  
BLOCK
```

```
selectMin(i in S)(f(i))  
BLOCK
```

```
selectMin(i in S : p(i))(f(i))
```

Tie-break Broken uniformly at random

Semi-greedy Selectors respect probability distributions

```
select  
BLOCK
```

```
BLOCK
```

```
selectMin[k](i in S)(f(i))  
BLOCK
```

```
selectMin[k](i in S : p(i))(f(i))  
BLOCK
```

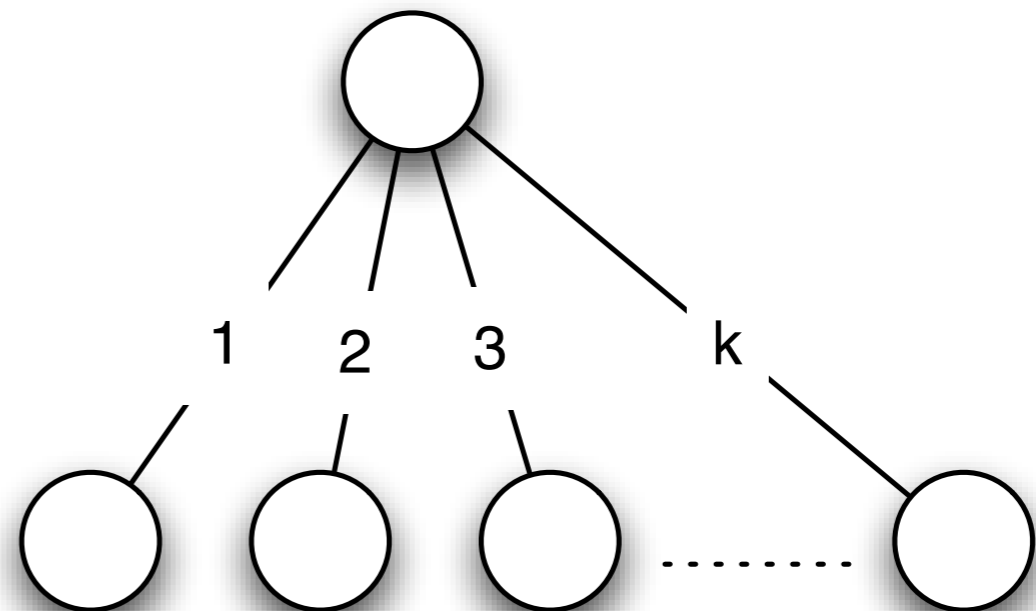
```
selectMax[k](i in S)(f(i))  
BLOCK
```

```
selectMax[k](i in S : p(i))(f(i))  
BLOCK
```

Extra Control: Non-determinism

- Let us express choices
 - Binary

```
try<c>  
  BLOCK1  
|  BLOCK2  
|  BLOCK3  
...  
|  BLOCKk
```





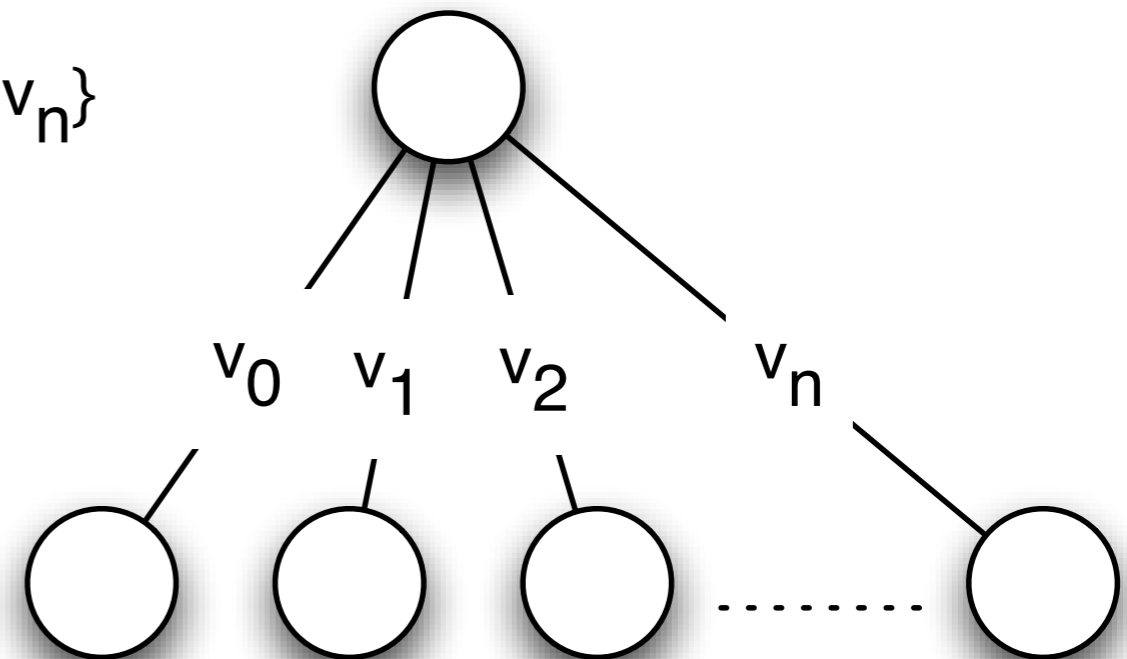
Extra Control: Non-determinism

- Let us express choices

- N-ary
- Branches given by set S

```
tryall<c>(i in S)  
BLOCK
```

$$S = \{v_0, v_1, \dots, v_n\}$$





Extra Control: Non-determinism

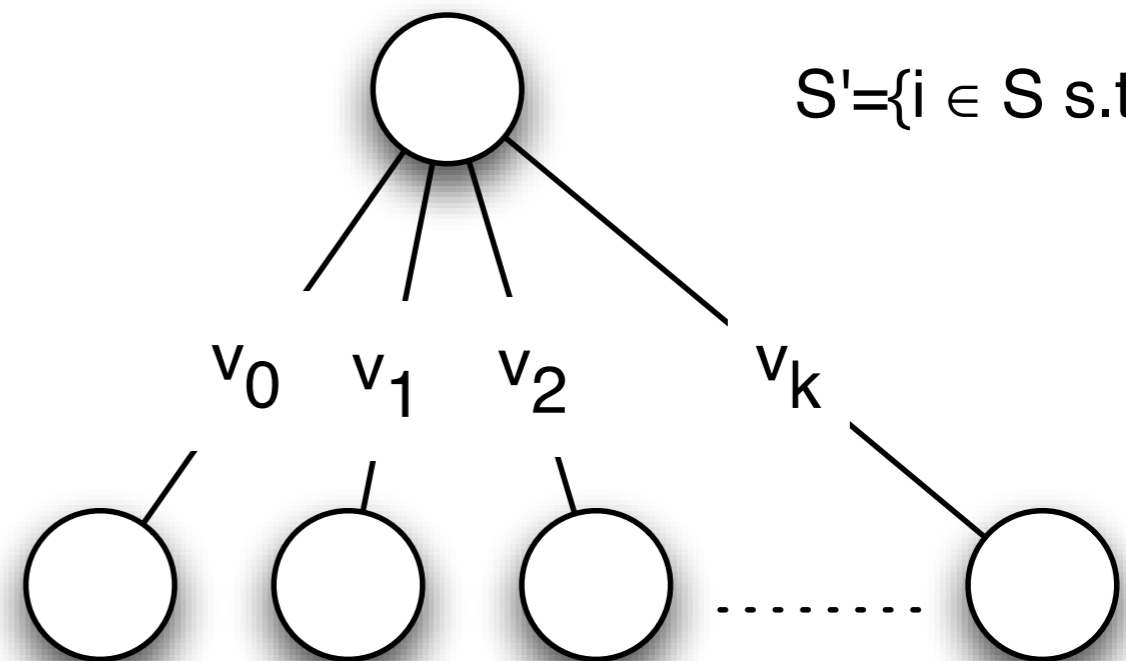
- Let us express choices

- N-ary

- Branches given by subset of S satisfying $p(i)$

```
tryall<c>(i in S : p(i))  
BLOCK
```

$S = \{v_0, v_1, \dots, v_n\}$



$S' = \{i \in S \text{ s.t. } p(i)\}$



Extra Control: Non-determinism

• Let us express choices

- N-ary
- Consider choices in order of increasing $f(i)$

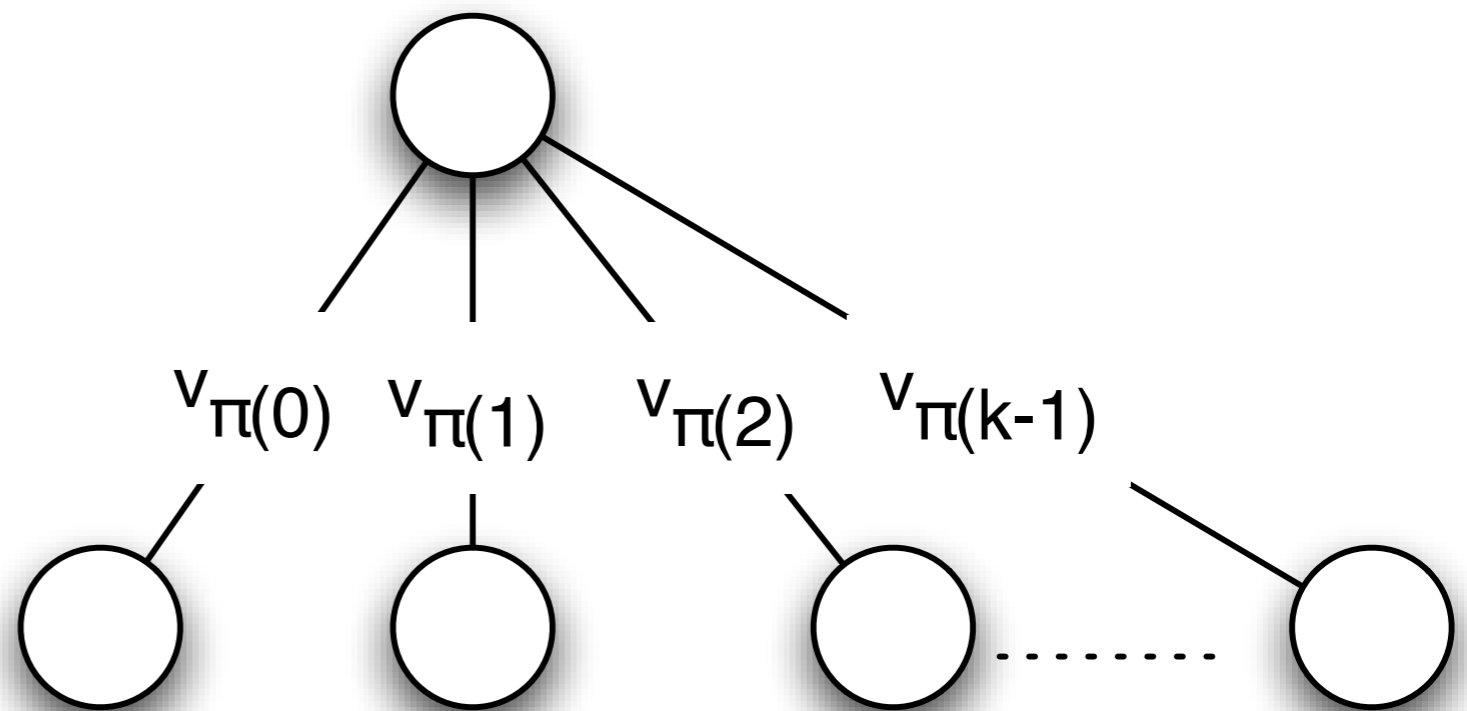
```
tryall<c>(i in S : p(i)) by (f(i))
BLOCK
```

$$S = \{v_0, v_1, \dots, v_n\}$$

$$S' = \{i \in S \text{ s.t. } p(i)\}, |S'| = k$$

$$\pi = \text{permutation}(0..k-1)$$

$$\text{s.t. } i \leq j \Rightarrow f(\pi(i)) \leq f(\pi(j))$$





Extra Control: Non-determinism

- Let us express choices

- N-ary

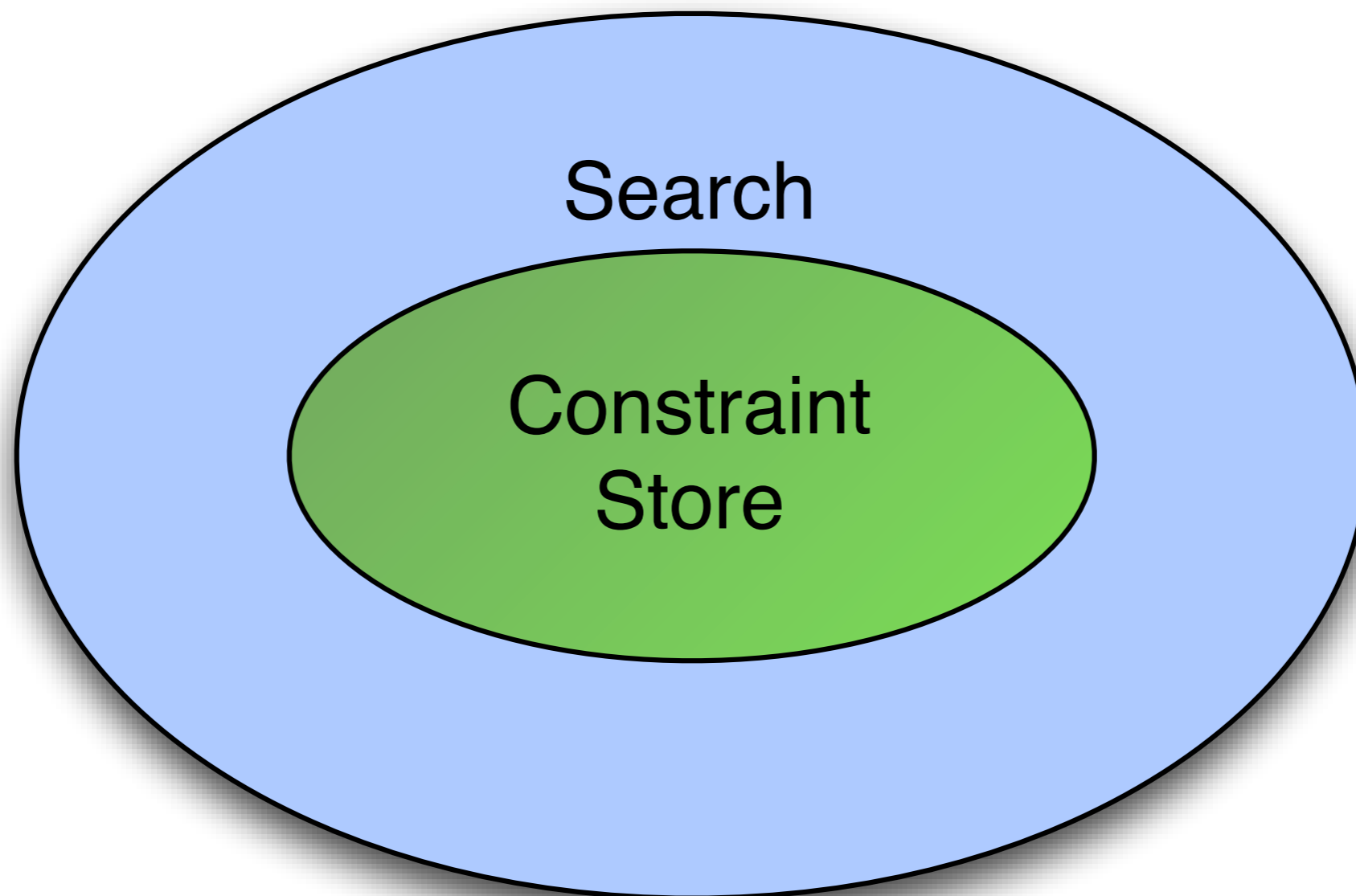
```
tryall<c>(i in S : p(i)) by (f(i))  
  BLOCK  
onFailure BLOCK2
```

- Adds ability to

- Execute BLOCK2 when there is a failure
- Before trying the next choice....

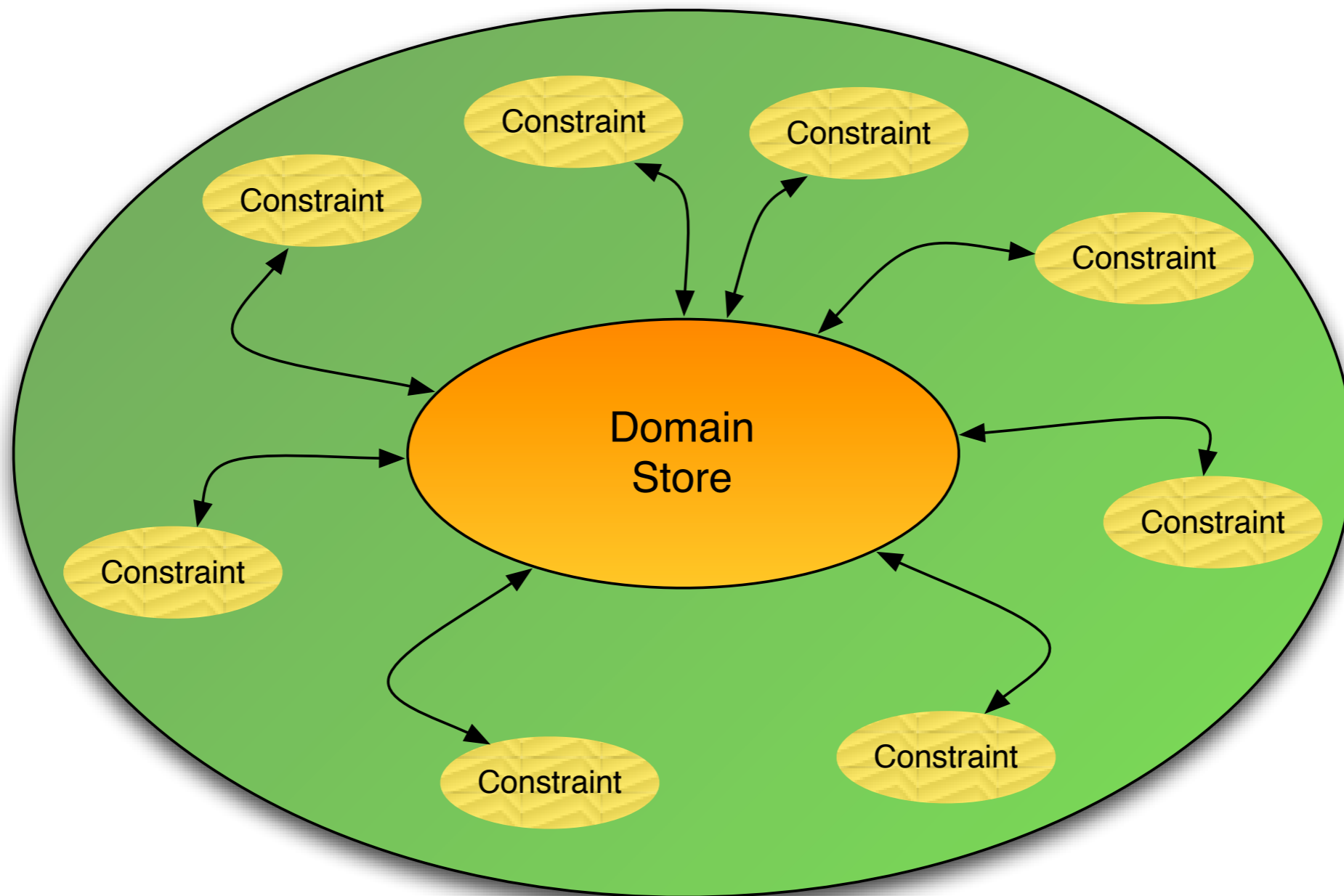


CP Computational Model



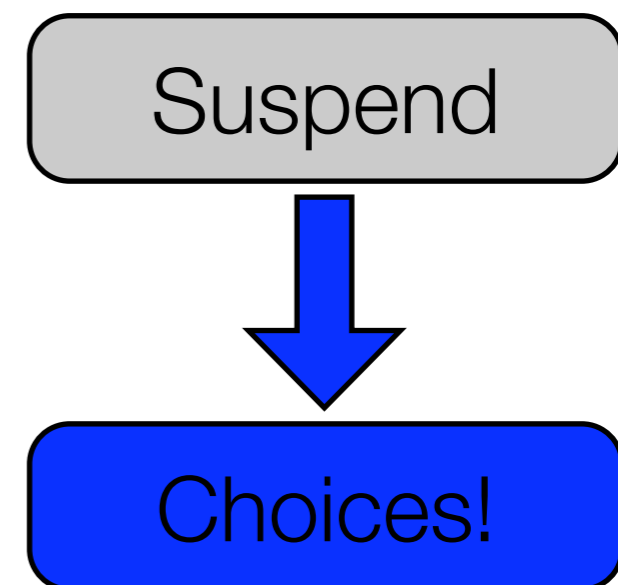
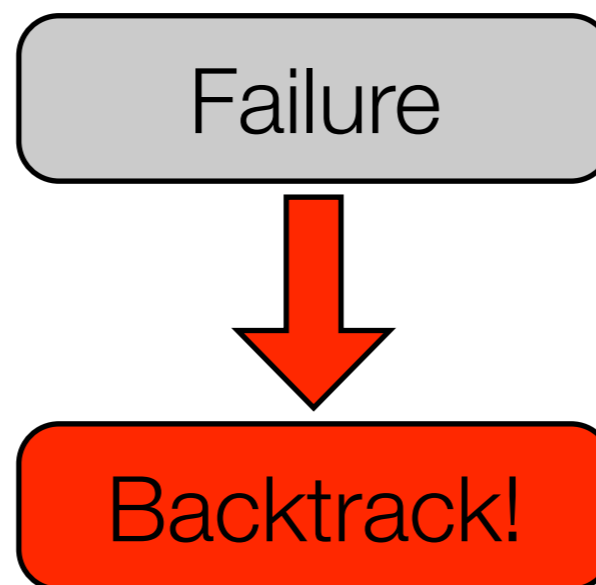
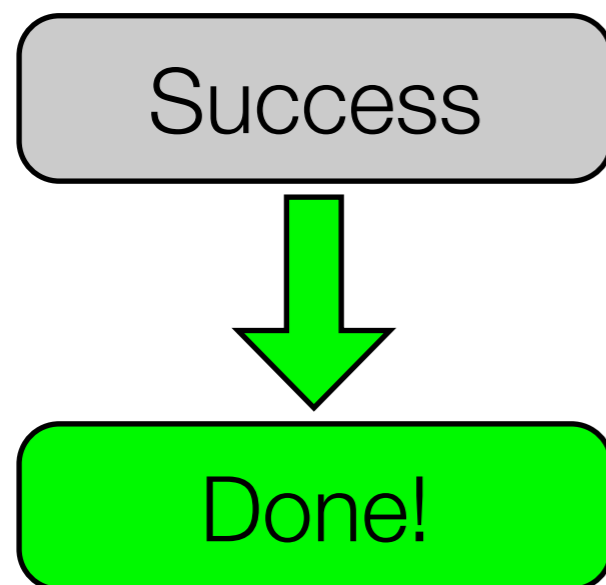


Computational Model



Operationally

- **Compute a fixpoint of the constraint set**
 - Reason on each constraint C *locally*
 - For every variable X appearing in C : prune $D(x)$
 - *Propagate* the impact to other constraints using X
 - Stop when no more changes
- **Outcomes ?**





Solvers

- Computational Model embedded in a solver
- Comet supports several solvers

```
import cotfd;  
Solver<CP> cp();
```

```
import cotln;  
Solver<LP> lp();  
Solver<MIP> ip();
```

```
import cotls;  
Solver<LS> ls();
```

Importing =
Loading a shared library +
defining all the interfaces +
defining all the classes



Solvers

- Computational Model embedded in a solver
- Comet supports several solvers

```
import cotfd;  
Solver<CP> cp();
```

```
import cotln;  
Solver<LP> lp();  
Solver<MIP> ip();
```

```
import cotls;  
Solver<LS> ls();
```

Importing =
Loading a shared library +
defining all the interfaces +
defining all the classes



Variables

- Variables are declared for a specific Solver
- For finite domain
 - Domain can be a range or a set.

```
import cotfd;
Solver<CP>    cp();

var<CP>{int}      x(cp,D);
var<CP>{bool}     y(cp);

var<CP>{set{int}} z(cp,1..10);    // In upcoming v1.3
```



Declarative Model

- **Model states**

- The nature of the problem
 - Constraint Satisfaction Problem
 - Find *one* solution
 - Find *all* solution
 - Constraint Optimization Problem
 - Find one global solution.
 - Prove optimality
- the constraints
 - Arithmetic / Logical / Combinatorial



CSP vs. COP

CSP

```
Solver<CP> m();  
...  
solve<m> {  
    ...  
} [using BLOCK]
```

```
Solver<CP> m();  
...  
solveall<m> {  
    ...  
} [using BLOCK]
```

COP

```
Solver<CP> m();  
...  
minimize<m> obj  
subject to {  
    ...  
} [using BLOCK]
```

```
Solver<CP> m();  
...  
maximize<m> obj  
subject to {  
    ...  
} [using BLOCK]
```



Stating Constraints

- Constraints should be stated *directly* or *indirectly* via one of...

- The “solve” block
- The “subject to” block
- The “using” block

```
solve<m> {  
    m.post(constraint);  
}
```

Auto, onDomains,
onBounds, onValues

- **Rationale...**

- Constraints can *fail* (prove infeasibility)
- Constraints posted inside the block trigger backtracking
- Constraints posted outside these block simply fail
 - [you must check the status manually]



Stating Constraints

- Constraints should be stated *directly* or *indirectly* via one of...

- The “solve” block
- The “subject to” block
- The “using” block

```
solve<m> {  
    m.post(constraint, onDomains);  
}
```

Auto, onDomains,
onBounds, onValues

- Rationale...

- Constraints can *fail* (prove infeasibility)
- Constraints posted inside the block trigger backtracking
- Constraints posted outside these block simply fail
 - [you must check the status manually]



Arithmetic Constraints

- Use all the traditional arithmetic operators

- Binary operators: $+$ $-$ $*$ $/$ $^$ \min \max

- absolute value: $\text{abs}()$

- Use all the relational operators

- $<$ $<=$ $>$ $>=$ $==$ $!=$



Element Constraints

- **Array and matrix indexing**
 - All combinations are allowed
 - Index an array of constants with a variable [ELEMENT]
 - Index a matrix of constants with variable(s) [Matrix ELEMENT]
 - Index an array of variables with a variable
 - Index a matrix of variables with variables(s)



Logical Constraints

- **Negation**

- With the ! operator

```
m.post(!b);
```

- **Conjunction**

- With the && operator

```
m.post((a < b) && (a < d));
```

- **Disjunction**

- With the || operator

```
m.post((a < b) || (a < d));
```

- **Implication**

- With the => operator

```
m.post(a => b);
```




Combinatorial Constraints

- **The “global” constraints**

- alldifferent
- cardinalities (at least, at most, exactly)
- binaryKnapsack, multiKnapsack, binPacking
- spread, deviation
- circuit
- inverse
- lexleq
- table
- sequence
- scheduling constraints...



First Simple Example

- **SEND + MORE = MONEY**

```
import cotfd;

Solver<CP> m();
range Digits = 0..9;

var<CP>{int} x[1..8](m,Digits);
var<CP>{int} S = x[1];
var<CP>{int} E = x[2];
var<CP>{int} N = x[3];
var<CP>{int} D = x[4];
var<CP>{int} M = x[5];
var<CP>{int} O = x[6];
var<CP>{int} R = x[7];
var<CP>{int} Y = x[8];
```



First Simple Example

•SEND + MORE = MONEY

```
import cotfd;

Solver<CP> m();
range Digits = 0..9;

var<CP>{int} x[1..8](m,Digits);
var<CP>{int} S = x[1];
var<CP>{int} E = x[2];
var<CP>{int} N = x[3];
var<CP>{int} D = x[4];
var<CP>{int} M = x[5];
var<CP>{int} O = x[6];
var<CP>{int} R = x[7];
var<CP>{int} Y = x[8];
```

```
solve<m> {
  m.post(alldifferent(x));
  m.post(M != 0);
  m.post(S != 0);
  m.post(
    1000 * S + 100 * E + 10 * N + D +
    1000 * M + 100 * O + 10 * R + E ==
    10000 * M + 1000 * O + 100 * N + 10 * E + Y);
}

cout << x << endl;
```



First Simple Example

• SEND + MORE = MONEY

```
import cotfd;

Solver<CP> m();
range Digits = 0..9;

var<CP>{int} x[1..8](m,Digits);
var<CP>{int} S = x[1];
var<CP>{int} E = x[2];
var<CP>{int} N = x[3];
var<CP>{int} D = x[4];
var<CP>{int} M = x[5];
var<CP>{int} O = x[6];
var<CP>{int} R = x[7];
var<CP>{int} Y = x[8];
```

Notes

1. Solve block
2. Default Search
3. Arithmetic constraint
4. One Combinatorial constraint

```
solve<m> {
  m.post(alldifferent(x));
  m.post(M != 0);
  m.post(S != 0);
  m.post(
    1000 * S + 100 * E + 10 * N + D +
    1000 * M + 100 * O + 10 * R + E ==
    10000 * M + 1000 * O + 100 * N + 10 * E + Y);
}

cout << x << endl;
```



Example

- Magic series

0 1 2 3 4

- A series of length 5

$s[2, 1, 2, 0, 0]$

- Reification (a.k.a. meta-constraint): constraint on constraints

```
import cotfd;
Solver<CP> m();
int n = 20;
range D = 0..n-1;
var<CP>{int} s[D](m,D);
solve<m> {
  forall(k in D)
    m.post(s[k] == sum(i in D) (s[i]==k));
}

cout << s << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail      = " << m.getNFail() << endl;
```



Improving the model : Redundant Constraints

- Add redundant constraint(s)!

$$\sum_{k \in 0..n-1} s[k] = n \quad \sum_{k \in 0..n-1} k \cdot s[k] = n \quad \sum_{k \in 0..n-1} (k - 1) \cdot s[k] = 0$$

```
import cotfd;
Solver<CP> m();
int n = 20;
range D = 0..n-1;
var<CP>{int} s[D](m,D);
solve<m> {
  forall(k in D)
    m.post(s[k] == sum(i in D) (s[i]==k));
  m.post(sum(k in D) (k-1)*s[k]==0);
}

cout << s << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail      = " << m.getNFail() << endl;
```



Searching!

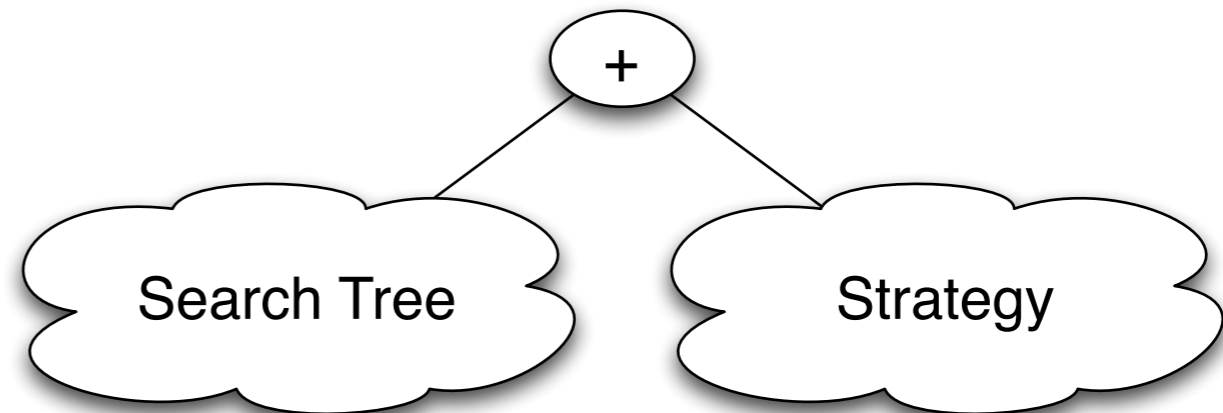
- **Purpose**

- Write your own search procedure
- Exploit problem semantics for...
 - Variables ordering
 - Value ordering
 - Dynamic symmetry breaking
 - Multi-phase searches
 - Dichotomic branching
 -



Search anatomy

- **Two pieces**
 - Specify a search tree
 - What does the tree look like?
 - variable ordering
 - value ordering
 - Specify [optional] a search strategy





Example with Queens

- **Rationale**
 - Simple problem
 - Illustrates the techniques
 - Start off with default strategy (DFS)



The basic model

```
import cotfd;

int t0 = System.getCPUTime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);

solve<m> {
    m.post(alldifferent(all(i in S) q[i] + i));
    m.post(alldifferent(all(i in S) q[i] - i));
    m.post(alldifferent(q));
}

cout << "Time      = " << System.getCPUTime() - t0 << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail    = " << m.getNFail() << endl;
```



Finding all solutions

```
import cotfd;

int t0 = System.getCPUTime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);

solveall<m> {
  m.post(alldifferent(all(i in S) q[i] + i));
  m.post(alldifferent(all(i in S) q[i] - i));
  m.post(alldifferent(q));
}

cout << "Time      = " << System.getCPUTime() - t0 << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail    = " << m.getNFail() << endl;
```



Printing and Counting solutions...

```
import cotfd;

int t0 = System.getCPUtime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);
Integer c(0);
solveall<m> {
    m.post(alldifferent(all(i in S) q[i] + i));
    m.post(alldifferent(all(i in S) q[i] - i));
    m.post(alldifferent(q));
} using {
    labelFF(m);
    cout << q << endl;
    c := c + 1;
}
cout << "Nb          = " << c << endl;
cout << "Time         = " << System.getCPUtime() - t0 << endl;
cout << "#choices    = " << m.getNChoice() << endl;
cout << "#fail       = " << m.getNFail() << endl;
```



What is labelFF ?

- **The default search procedure...**
 - Implements first-fail principle
 - First the variable with the smallest domain
 - Try values in increasing order
- **Can't we write this *ourselves*?**

Sure!
Let's start with a very naive search...
...and build up!



Static Ordering [a.k.a. the label function]

- **Simple idea**

- Label variables in their “natural” order (order of declaration)
- Try values in increasing order

```
...
} using {
  forall(i in S)
    tryall<m>(v in S)
      m.post(q[i] == v);
}
```



Static Ordering 2

- **First improvement**
 - Skip over variables that are already bound!

```
...  
} using {  
  forall(i in S : !q[i].bound())  
    tryall<m>(v in S)  
      m.post(q[i] == v);  
}
```



Static Ordering 3

- **Second improvement**
 - Skip values that are no longer in the domain!

```
...
} using {
  forall(i in S : !q[i].bound())
    tryall<m>(v in S : q[i].memberOf(v))
      m.post(q[i] == v);
}
```




Dynamic Ordering

- **First consider the variables with the smallest domain**
 - Note that this is dynamic, the domain size changes each time!

```
...
} using {
  forall(i in S : !q[i].bound()) by (q[i].getSize())
    tryall<m>(v in S : q[i].memberOf(v))
      m.post(q[i] == v);
}
```



Dynamic Ordering

- **Finally...**

- When we fail, remember that the value is no longer legal!

```
...
} using {
  forall(i in S : !q[i].bound()) by (q[i].getSize())
    tryall<m>(v in S : q[i].memberOf(v))
      m.post(q[i] == v);
    onFailure m.post(q[i] != v);
}
```



Tweaks...

- **Use lighter branching method**

- replace `m.post(x[i] == v)` by `m.label(x[i],v);`

- replace `m.post(x[i] != v)` by `m.diff(x[i],v);`

- **Light api...**



Tweaks...

- **Use lighter branching method**

- replace `m.post(x[i] == v)` by `m.label(x[i],v);`
- replace `m.post(x[i] != v)` by `m.diff(x[i],v);`

- **Light api...**

```
class Solver<CP> {  
    ...  
    Outcome<CP> label(var<CP>{int} x,int v);  
    Outcome<CP> diff(var<CP>{int} x,int v);  
    Outcome<CP> lthen(var<CP>{int} x,int v);  
    Outcome<CP> gthen(var<CP>{int} x,int v);  
    Outcome<CP> inside(var<CP>{int} x,set{int} s);  
    Outcome<CP> outside(var<CP>{int} x,set{int} s);  
    ...  
}
```



Final version

- First-fail principle is 4 lines of code.
- Advantage?
 - You can instrument / modify to your heart's content

```
...
} using {
  forall(i in S : !q[i].bound()) by (q[i].getSize())
    tryall<m>(v in S : q[i].memberOf(v))
      m.label(q[i],v);
    onFailure m.diff(q[i],v);
}
```



Search Strategy

- **What does it do?**
 - Alter the exploration technique
 - DFS
 - LDS
 - BFS
 - ...
- **Other alterations?**
 - Limits
 - Restarting



Example

- Queens DFS style (DFS = Depth First Search – the default –)

```
import cotfd;

int t0 = System.getCPUTime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);
Integer c(0);

solveall<m> {
    m.post(alldifferent(all(i in S) q[i] + i));
    m.post(alldifferent(all(i in S) q[i] - i));
    m.post(alldifferent(q));
} using {
    labelFF(m);
    cout << q << endl;
    c := c + 1;
}
cout << "Nb          = " << c << endl;
```



Example

- Queens BDS style (BDS = Bounded Discrepancy Search)

```
import cotfd;

int t0 = System.getCPUTime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);
Integer c(0);
m.setSearchController(BDSController(m));
solveall<m> {
    m.post(alldifferent(all(i in S) q[i] + i));
    m.post(alldifferent(all(i in S) q[i] - i));
    m.post(alldifferent(q));
} using {
    labelFF(m);
    cout << q << endl;
    c := c + 1;
}
cout << "Nb          = " << c << endl;
```




Formulation & Solvers

- **Consider the classic assignment problem** (http://en.wikipedia.org/wiki/Assignment_problem)

- Given two sets, A and T , of equal size, together with a weight function $C : A \times T \rightarrow \mathbb{R}$. Find a bijection $f : A \rightarrow T$ that minimizes the cost function:

$$\sum_{a \in A} C(a, f(a))$$



Model IP and CP

•IP

minimize $\sum_{i \in A} \sum_{j \in T} C(i, j) * x_{ij}$

subject to

$$\sum_{j \in T} x_{ij} = 1 \quad \forall i \in A$$

$$\sum_{i \in A} x_{ij} = 1 \quad \forall j \in T$$

$$x_{ij} \geq 0 \quad \forall i \in A, j \in T$$

•CP

$$\sum_{a \in A} C(a, f(a))$$



Data Part of the Model

```
int t0 = System.getCPUTime();
int n = 8;
range A = 1..n;
range T = 1..n;
int c[A,T] =
[
  [13, 21, 20, 12, 8, 26, 22, 11],
  [12, 36, 25, 41, 40, 11, 4, 8],
  [35, 32, 13, 36, 26, 21, 13, 37],
  [34, 54, 7, 8, 12, 22, 11, 40],
  [21, 6, 45, 18, 24, 34, 12, 48],
  [42, 19, 39, 15, 14, 16, 28, 46],
  [16, 34, 38, 3, 34, 40, 22, 24],
  [26, 20, 5, 17, 45, 31, 37, 43]
];
```



IP Model

```
import cotln;
Solver<MIP> m();

var<MIP>{int} x[A,T](m, 0..10000); // x[i,j]==1 <=> actor i does task j
var<MIP>{int} z(m, 0..10000);

minimize<m> z subject to {
  forall(i in A)
    m.post(sum(j in T) x[i,j] == 1);
  forall(j in T)
    m.post(sum(i in A) x[i,j] == 1);
  m.post(z == sum(i in A, j in T) c[i,j] * x[i,j]);
}

cout << "z: " << z.getValue() << endl;
forall(i in A)
  cout << sum(j in T) j*x[i,j] << " ";
cout << endl;

cout << "time:      " << (System.getCPUTime()-t0) << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail      = " << m.getNFail() << endl;
```



CP Model v1.0

```
import cotfd;
Solver<CP> m();

var<CP>{int} x[T](m,A); // who does task j ?
var<CP>{int} z(m, 0..10000);

minimize<m> z subject to {
    m.post(alldifferent(x));
    m.post(z == sum(j in T) c[x[j],j]);
}

cout << "z: " << z.getValue() << endl;
cout << "Assigned: " << x << endl;
cout << endl;

cout << "time:      " << (System.getCPUTime()-t0) << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail      = " << m.getNFail() << endl;
```



CP Model v2.0

```
import cotfd;
Solver<CP> m();

var<CP>{int} x[T](m,A); // who does task j ?
var<CP>{int} z(m, 0..10000);

minimize<m> z subject to {
  m.post(alldifferent(x),onDomains);
  m.post(z == sum(j in T) c[x[j],j],onDomains);
}
```



CP Model v3.0

```
import cotfd;
Solver<CP> m();

var<CP>{int} x[T](m,A); // who does task j ?
var<CP>{int} z(m, 0..10000);

minimize<m> z subject to {
  m.post(alldifferent(x),onDomains);
  m.post(z == sum(j in T) c[x[j],j],onDomains);
} using {
  forall(j in T : !x[j].bound())
    tryall<m>(i in A : x[j].memberOf(i)) by (c[i,j])
    m.label(x[j],i);
    onFailure m.diff(x[j],i);
  }
}
```



CP Model v3.0

```
import cotfd;
Solver<CP> m();

var<CP>{int} x[T](m,A); // who does task j ?
var<CP>{int} z(m, 0..10000);

minimize<m> z subject to {
  m.post(alldifferent(x),onDomains);
  m.post(z == sum(j in T) c[x[j],j],onDomains);
} using {
  while(!bound(x)) {
    selectMin(j in T : !x[j].bound(),
              i in x[j].getMin()..x[j].getMax() : x[j].memberOf(i))(c[i,j]) {
      try<m> m.label(x[j],i);
      | m.diff(x[j],i);
    }
  }
}
```




Going Further ?

- More Search Strategies
- Setting Limits
- Restarting
- Exploring the LS engine
- Creating hybrid models
 - LP + CP
 - IP + CP
 - CP + LS
- Parallel Solving on a multi-core machine