



# Hybrid Techniques

## CP + CBLS

---

Laurent Michel, Pascal Van Hentenryck  
University of Connecticut  
Brown University



# Overview

---

- COMET Introduction
  - CP
  - CBLS
- CP+CBLS Hybridization
  - What to expect
  - Classes of hybrids
  - Implementation challenges
- Sequential composition
- Parallel composition
- Large Neighborhood Search



# Part I

---

## COMET Introduction





# COMET Introduction

---

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



# COMET

---

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



# Integrating Code with COMET

---

- Options available

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



# Modeling with COMET

---

- **Modeling power**
  - High level models for CBLS and CP
  - rich language of constraints and objectives
  - vertical extensions



# Solving with COMET

---

- **Search**

- a unique search language for

- CBLS

- CP

- MP

- **Hybridization**

- Solvers are first-class objects





# Hybrids 1

---

- Three LP/MIP Solvers
  - Ipsolve
  - coin-Clp
  - Gurobi
- 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
- LNS

- Communication?

- Bounds
- Actual solution, frequencies, ....

# Architecture

---



Loadable plugins

LS Engine	CP Engine	LP Engine	MIP Engine	SAT Engine	Visualizer	User Defined
Comet Virtual Machine						
Operating system Windows / Linux / Mac OS						



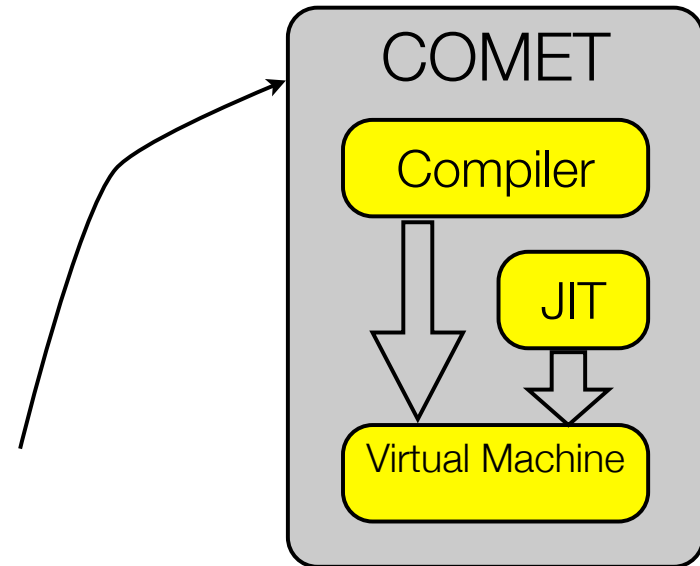
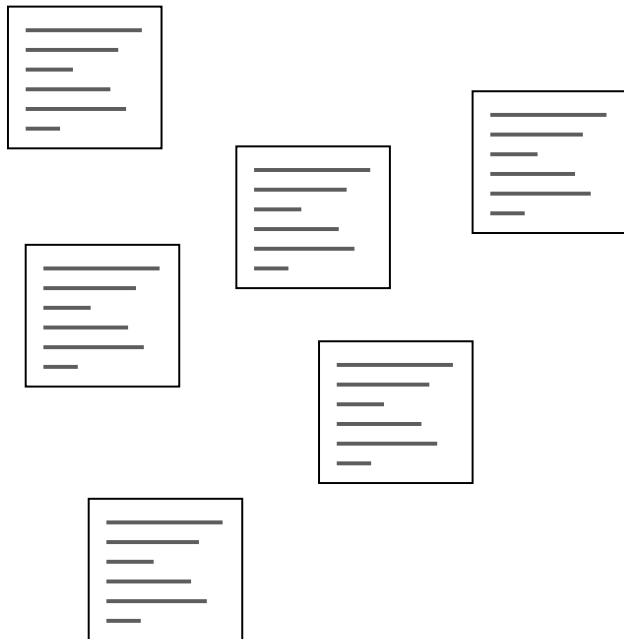
# Core Language

---

- **Similar to C++ or Java**
  - Statically typed
  - Strongly typed
- **Abstractions**
  - Classes
  - Interfaces
- **Control**
  - All the usual gizmos
  - Additional looping / branching construction

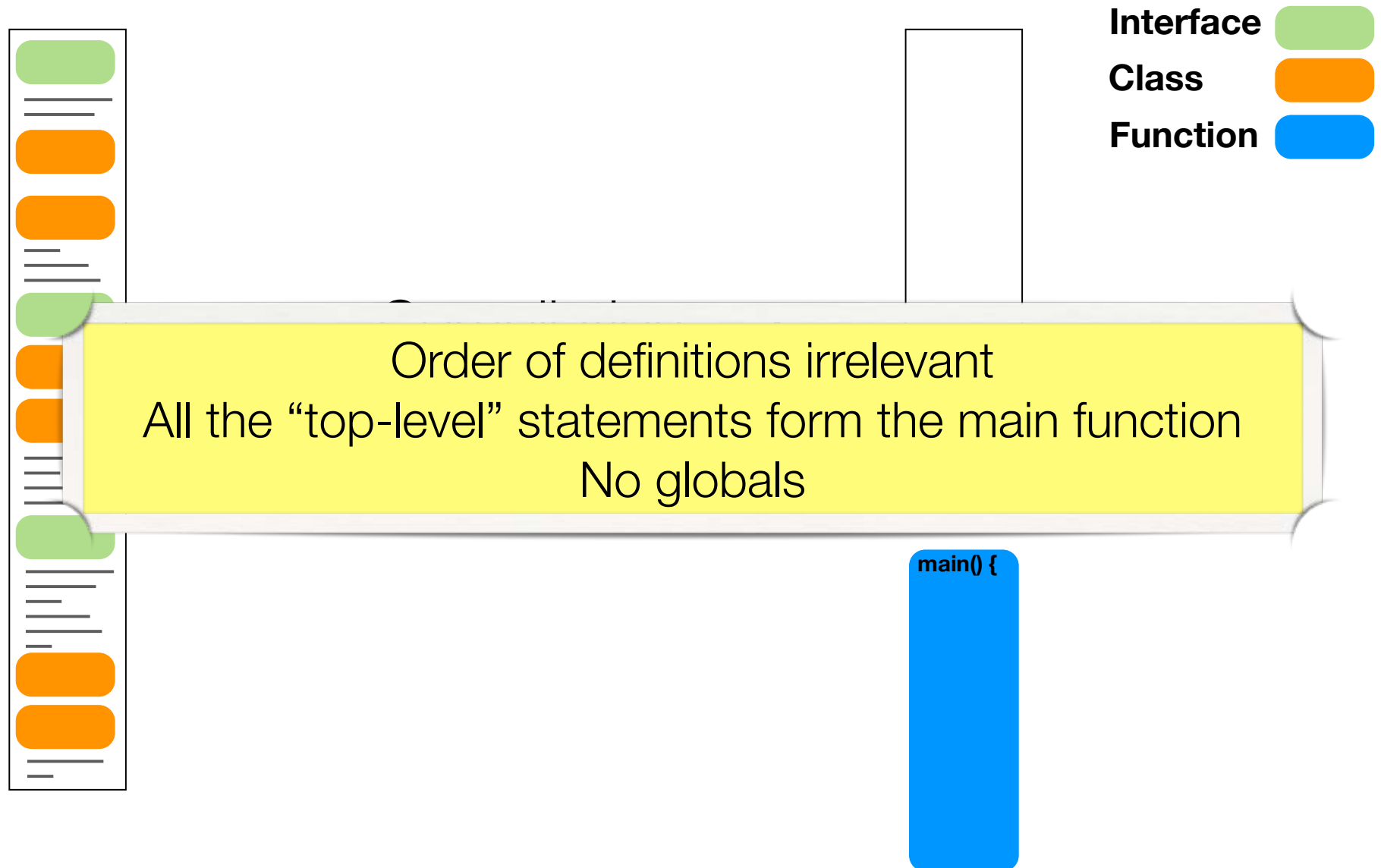
# Workflow

---





# Source Organization





# 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 [doc](#)

- 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
```

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

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

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

```
select  
  BLOCK
```

Tie-break Broken uniformly at random  
Semi-greedy Selectors respect probability distributions

```
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
```

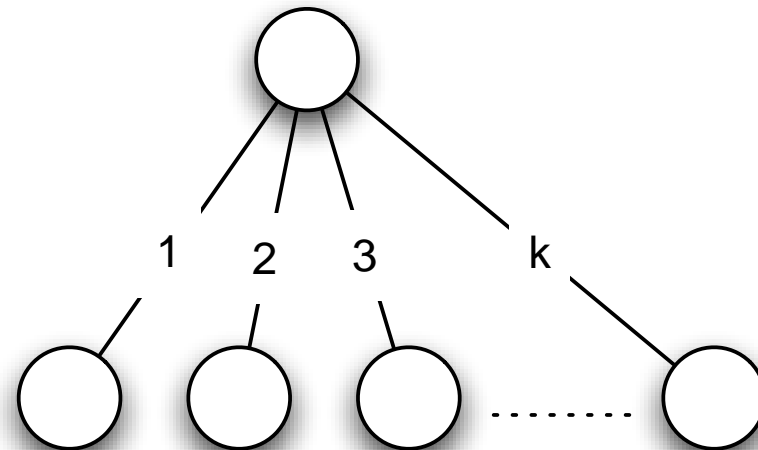
```
selectMax[k](i in S : p(i))(f(i))  
  BLOCK
```

# Extra Control: Non-determinism

- Let us express choices

- Binary

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





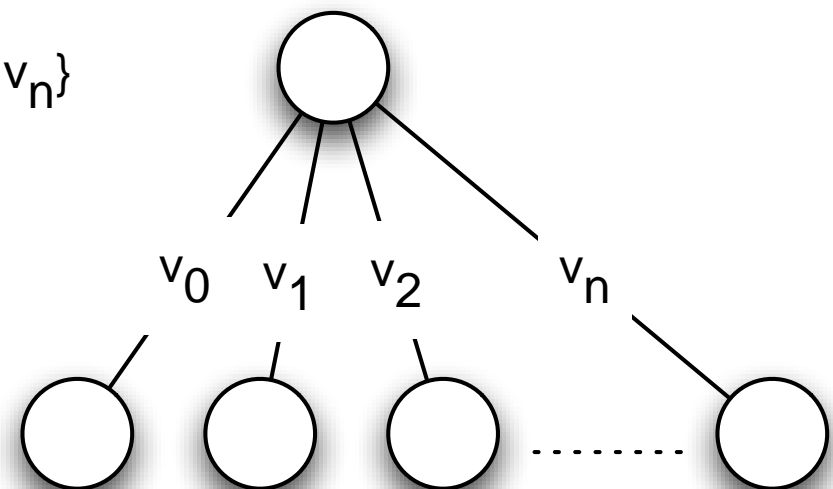
# Extra Control: Non-determinism

- Let us express choices

- N-ary
- Branches given by set  $S$

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

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





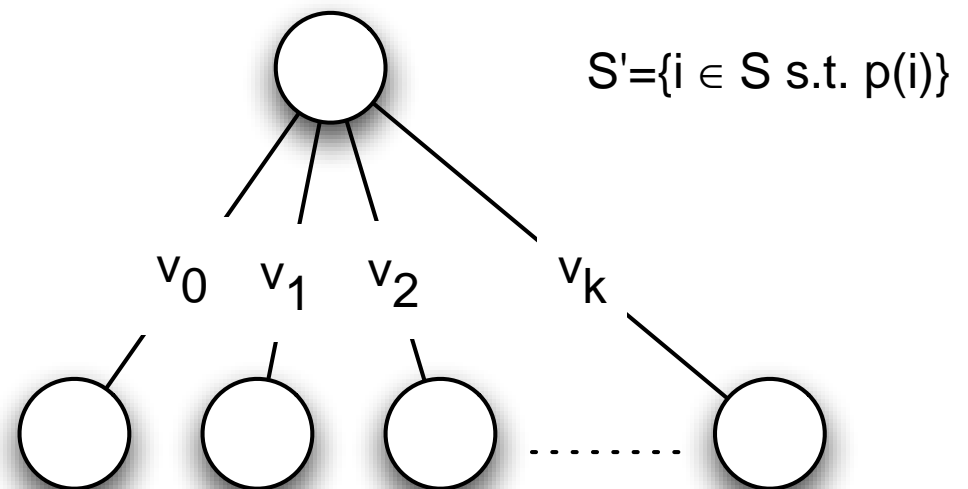
# Extra Control: Non-determinism

- Let us express choices

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

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

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





# Extra Control: Non-determinism

- Let us express choices

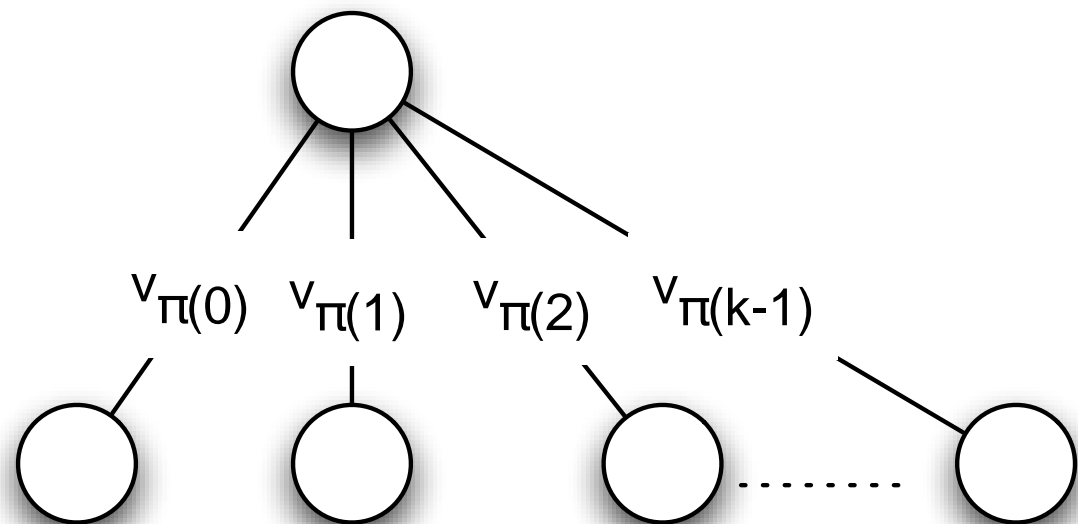
- N-ary
- Consider choices in order of increasing  $f(i)$

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

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

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

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



# COMET Introduction

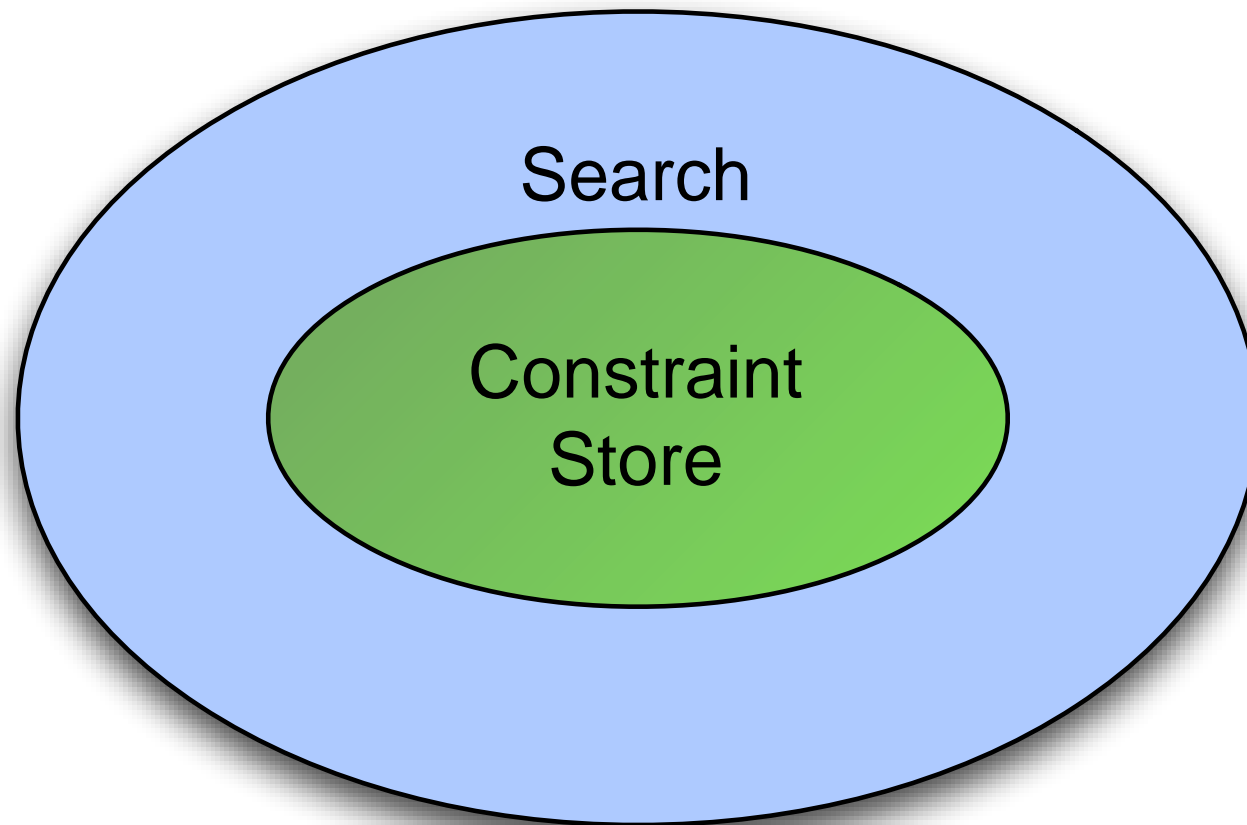
---

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

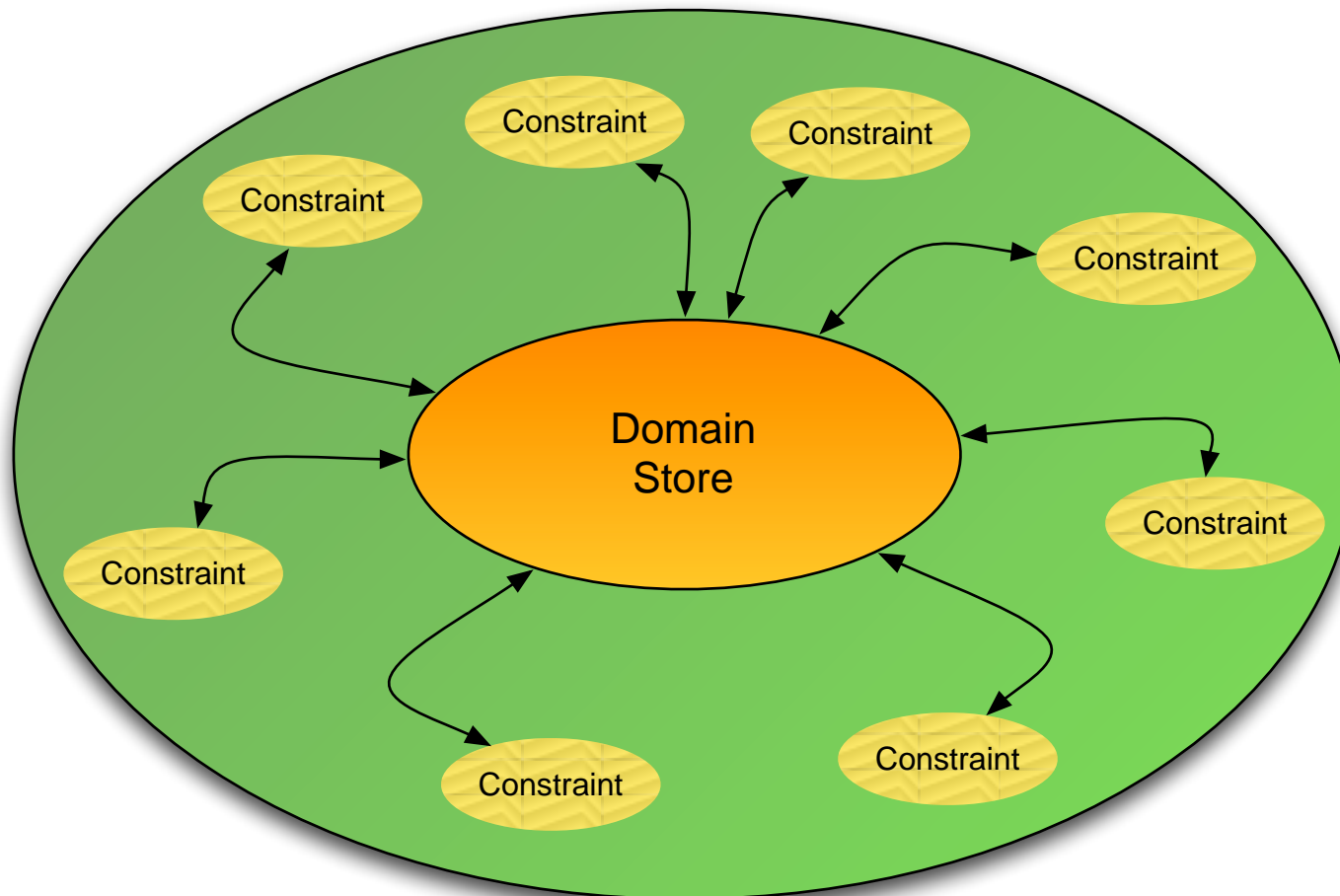


# CP Computational Model

---



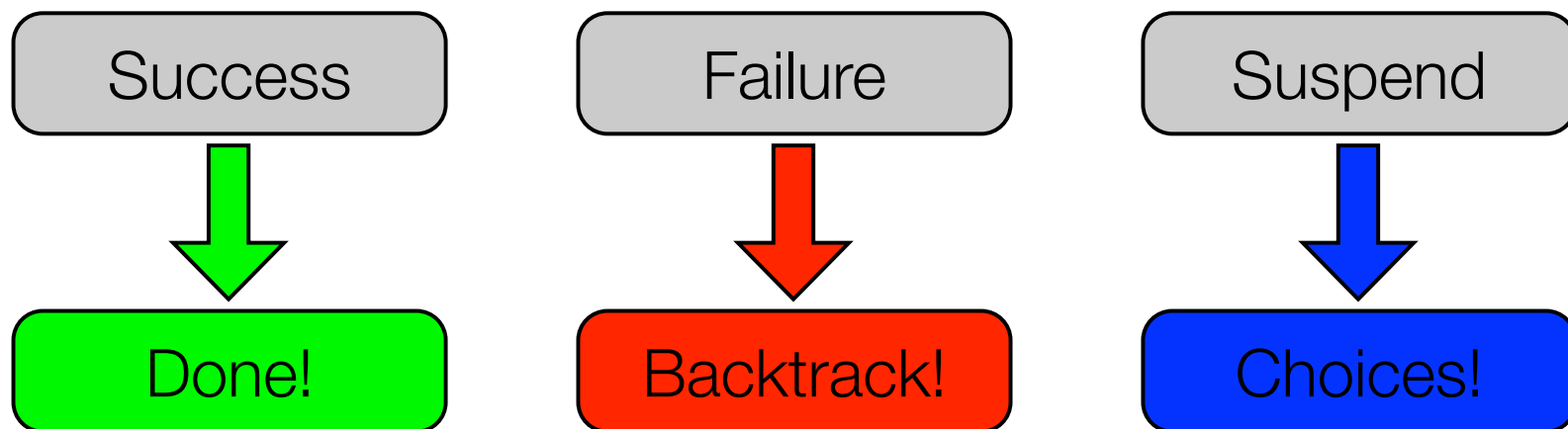
# Solver Architecture



# 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



# 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);
```



# 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,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: `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];
```

### 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

• A serie of length 5      **0 1 2 3 4**  
                                 **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
  - Domain splitting
  - ....

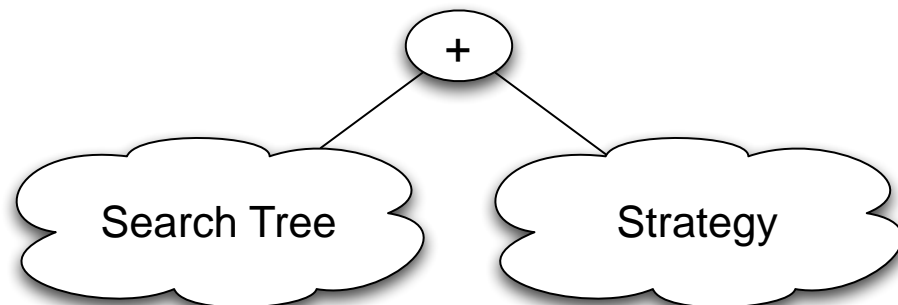




# Search anatomy

---

- **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;
```



# 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...

```
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)

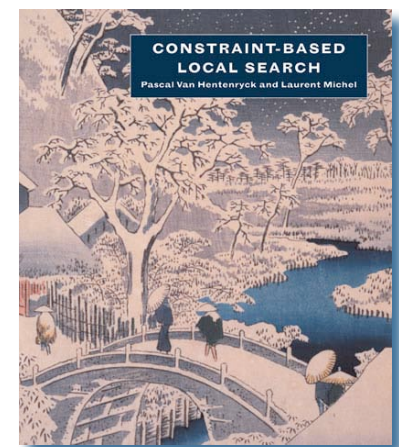
```
import cotfd;

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;
```

# COMET Introduction

---

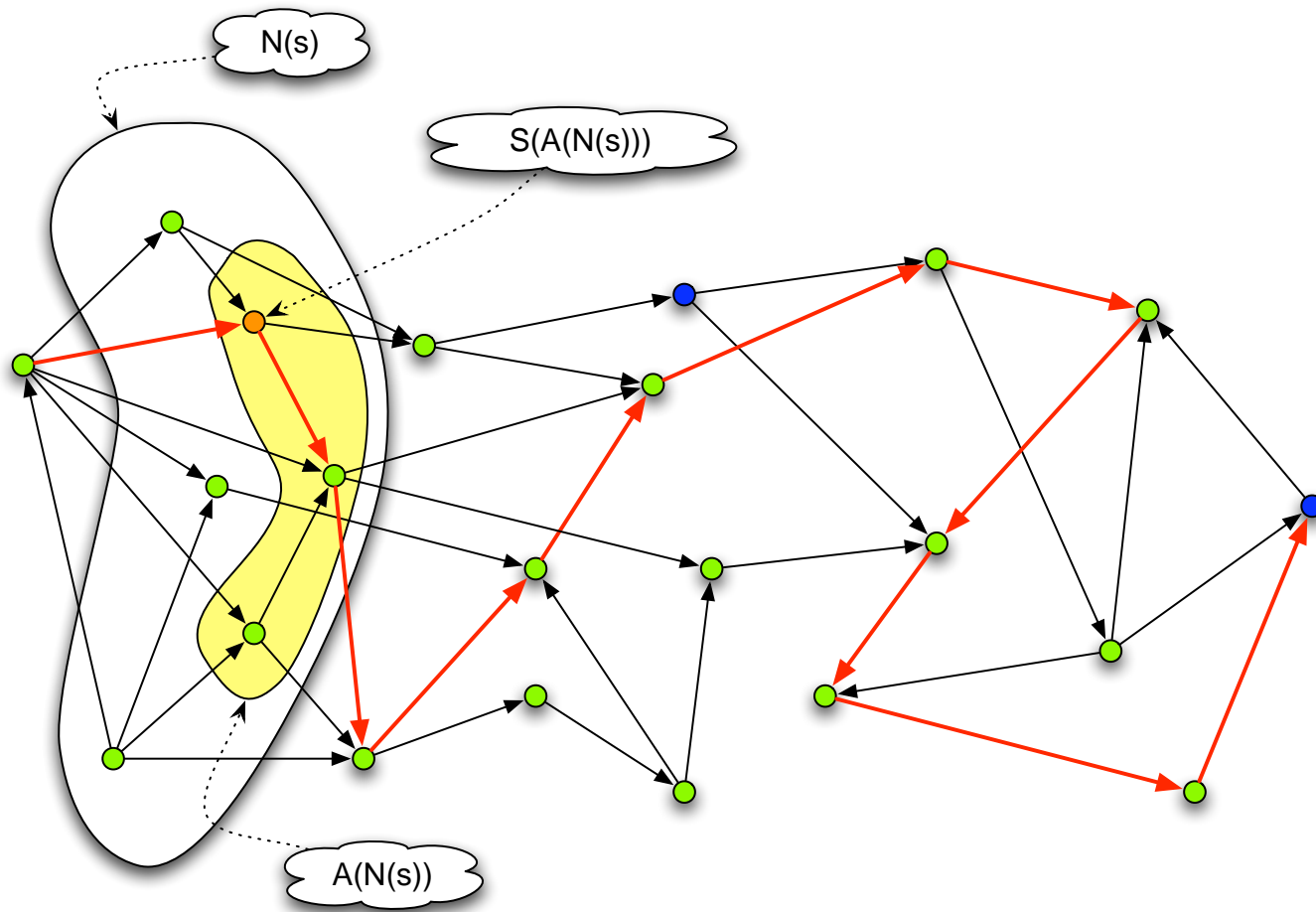
- The COMET Platform
  - Core Language
  - The CP Solver
    - Declarative Model
    - Search Procedures
- The LS Solver
  - Declarative Model
  - Search Procedures





# CBLS Computational Model

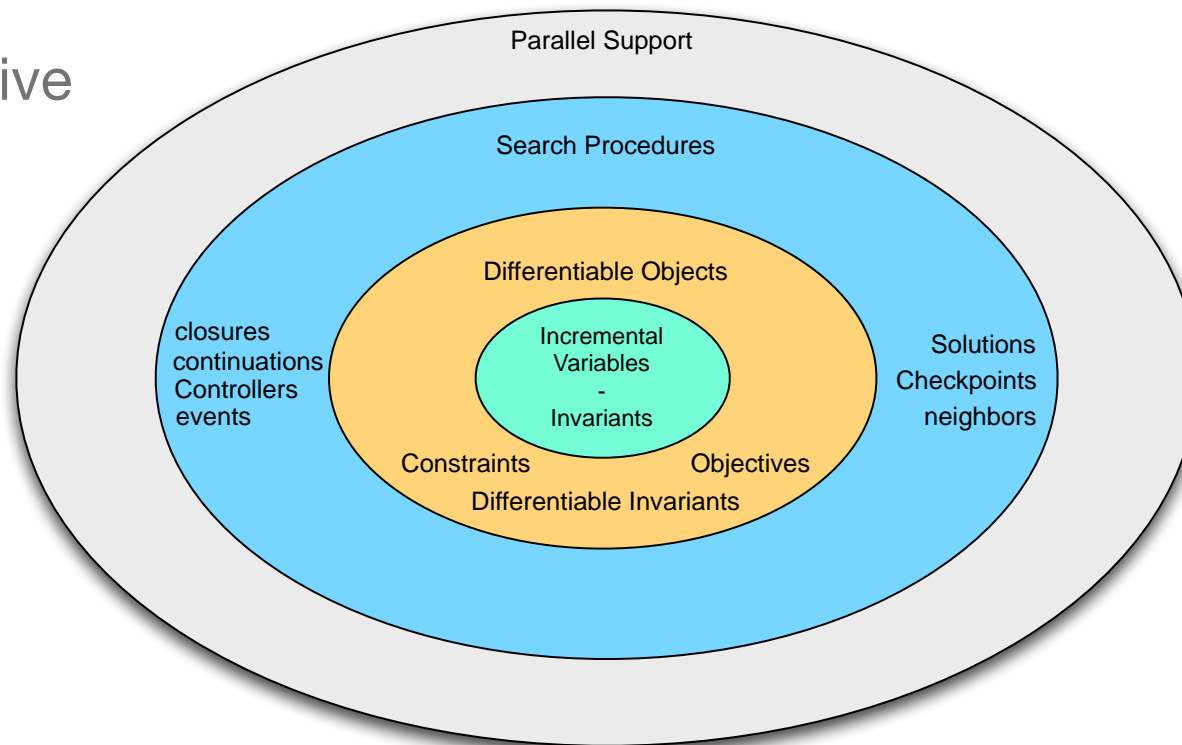
- Graph Exploration



# Solver Architecture

## •Four Levels

- Incremental
- Declarative
- Control
- Meta





# 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 cotls;  
Solver<LS> ls();  
  
var{int}          x(cp,D);  
var{bool}         y(cp);  
  
var{set{int}}     z(cp);
```



# Declarative Model

---

- **Model states**

- The nature of the problem
  - Constraint Satisfaction Problem
    - Find **one** solution
    - **CANNOT** find all solutions
  - Constraint Optimization Problem
    - Look for the highest quality solution.
    - **CANNOT** prove optimality
- the constraints
  - Arithmetic / Logical / Combinatorial



# Modeling Constraint Satisfaction

---

- Recall the Computational Model
  - Iteratively “repair” a candidate solution.
- Model
  - Set of constraints
- Search
  - Relax some of the constraints
  - Introduce an objective function
    - To measure the violation of the relaxed part
  - Search = minimize the violation
  - Solution when objective is 0



# Modeling Constraint Optimization

---

- **Model**

- Set of Constraints
- Objective Function

- **Search**

- Relax some of the constraints
- Combine with the existing objective function (e.g., weighted sum)
- Solve as a CSP



# Outcomes

---

- **When softened constraints are satisfied**
  - Feasible solution, further improvements are on the objective
- **Design choices**
  - Could limit moves to preserve feasibility
    - Smaller neighborhood
    - “Phased” approach
  - Could authorized “degrading” moves (w.r.t. feasibility)
    - Larger neighborhoods (more options to improve)
    - Need to alternate/oscillate between optimizing / satisfying





# Modeling language

---

- Once again
  - Algebraic
  - Logical
  - Global



# Arithmetic Constraints

---

- Use all the traditional arithmetic operators

- Binary operators: `+` `-` `*` `/` `^` `min` `max`

- absolute value: `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);
```



# Combinational Objects

---

- **Global Constraints**

- alldifferent
- knapsack
- multiKnapsack
- cardinality (allPresent, atmost, exactly, atleast)
- sequence

- **Functions**

- nbDistinct, minNbDistinct, maxNbDistinct
- sumMinCost
- Constraint reification operator



# Search Procedure

```
0 function void conflictSearch(ConstraintSystem S) {  
1   var{int}[] x = S.getVariables();  
2   while (S.violations() > 0) {  
3     selectMax(i in x.getRange())(S.violations(x[i]))  
4     selectMin(v in x[i].getDom())(S.getAssignDelta(x[i],v))  
5     x[i] := v;  
6   }
```

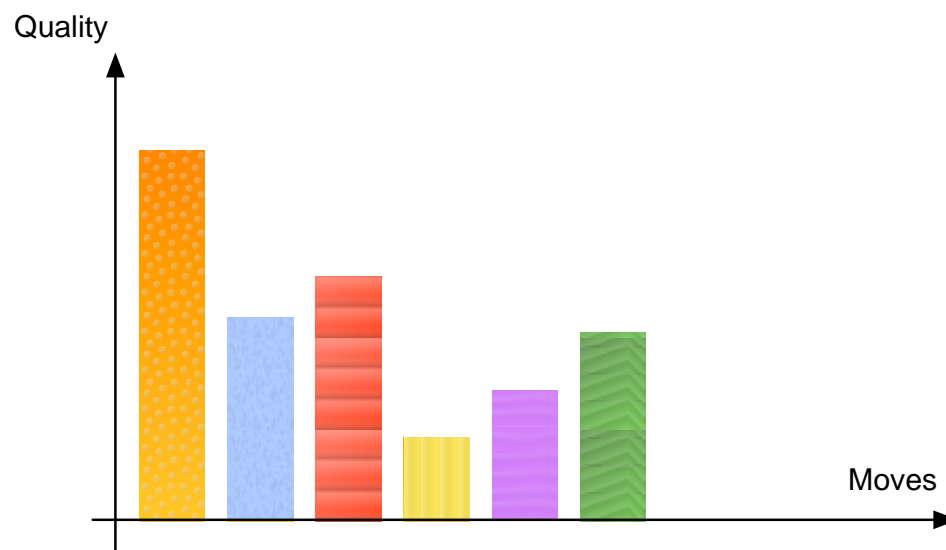
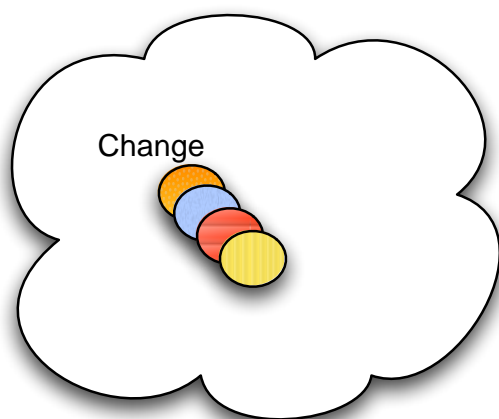
Query the Constraint System

# Motivation

- **Key principles**

- Evaluate *lots* of moves w.r.t. quality measure [violations]
- Moves are *small* changes to state

Evaluate the moves fast to  
evaluate as many moves as possible





# Incrementality

---

- Necessary to meet the objective
- Possible given the locality of the move

Questions

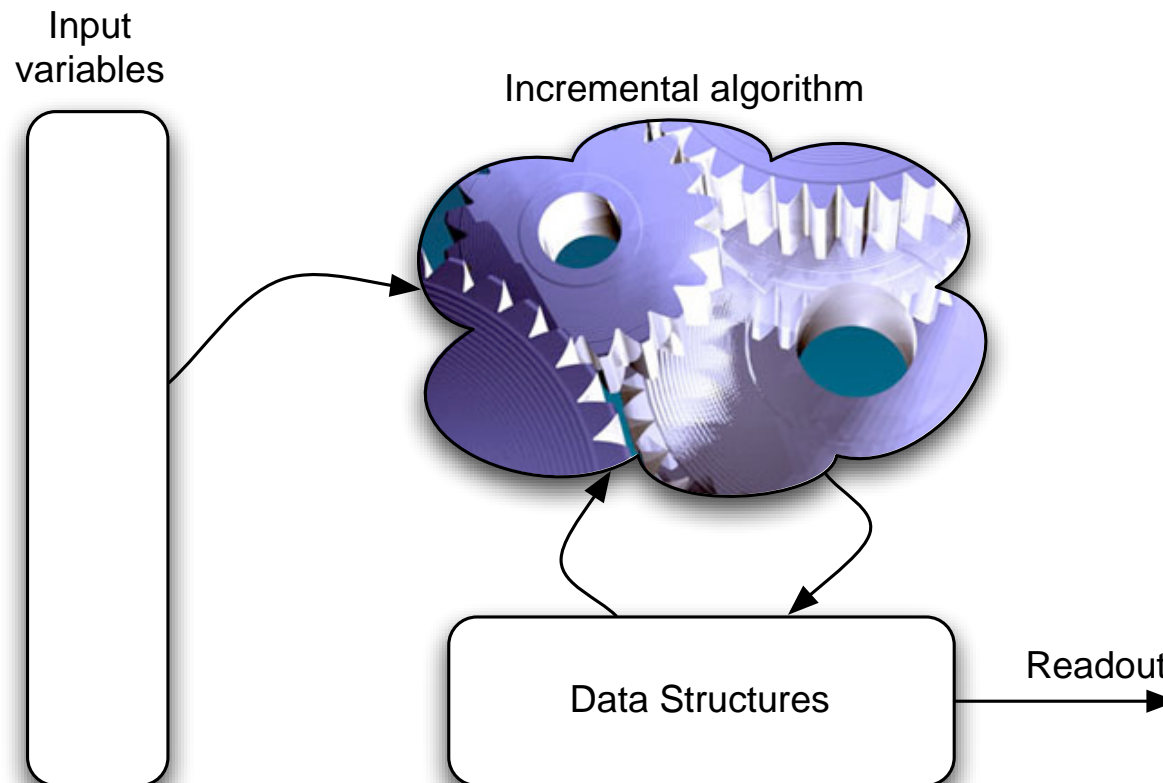
How to do it?

How to do it easily?



# Doing it manually

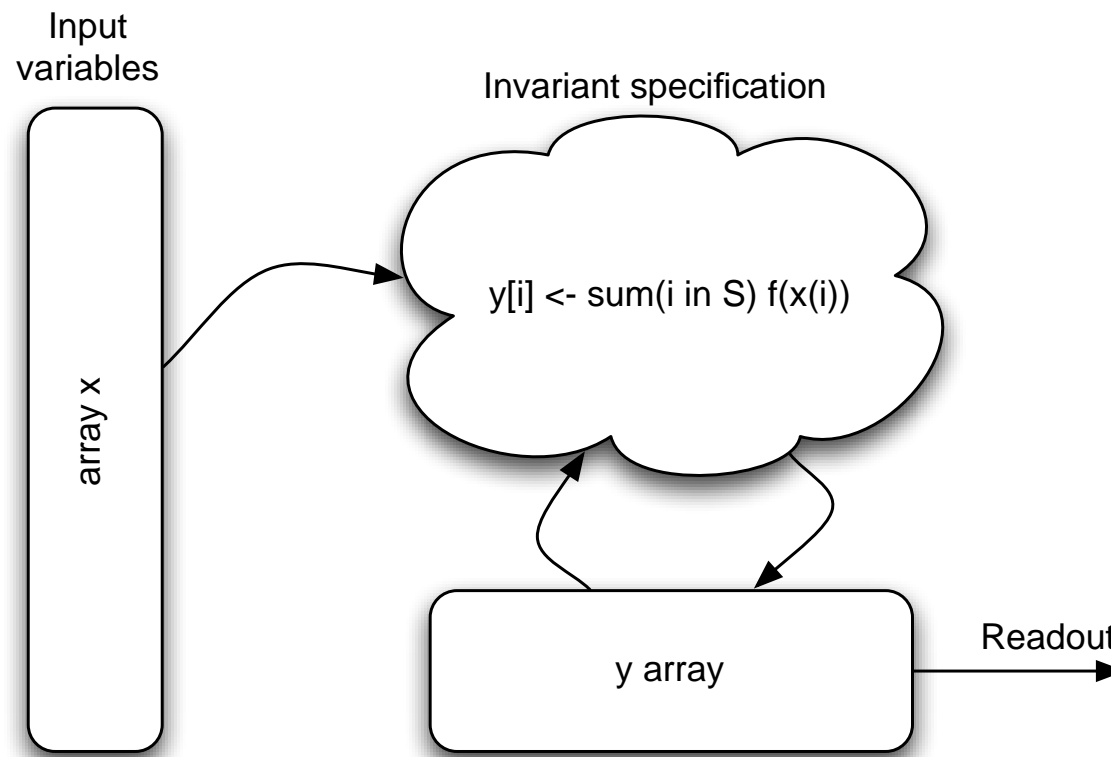
- Design a data structure to hold *derived data*
- Write dedicated algorithms to update the *derived data*



# Doing it with *invariants* & *Accumulators*

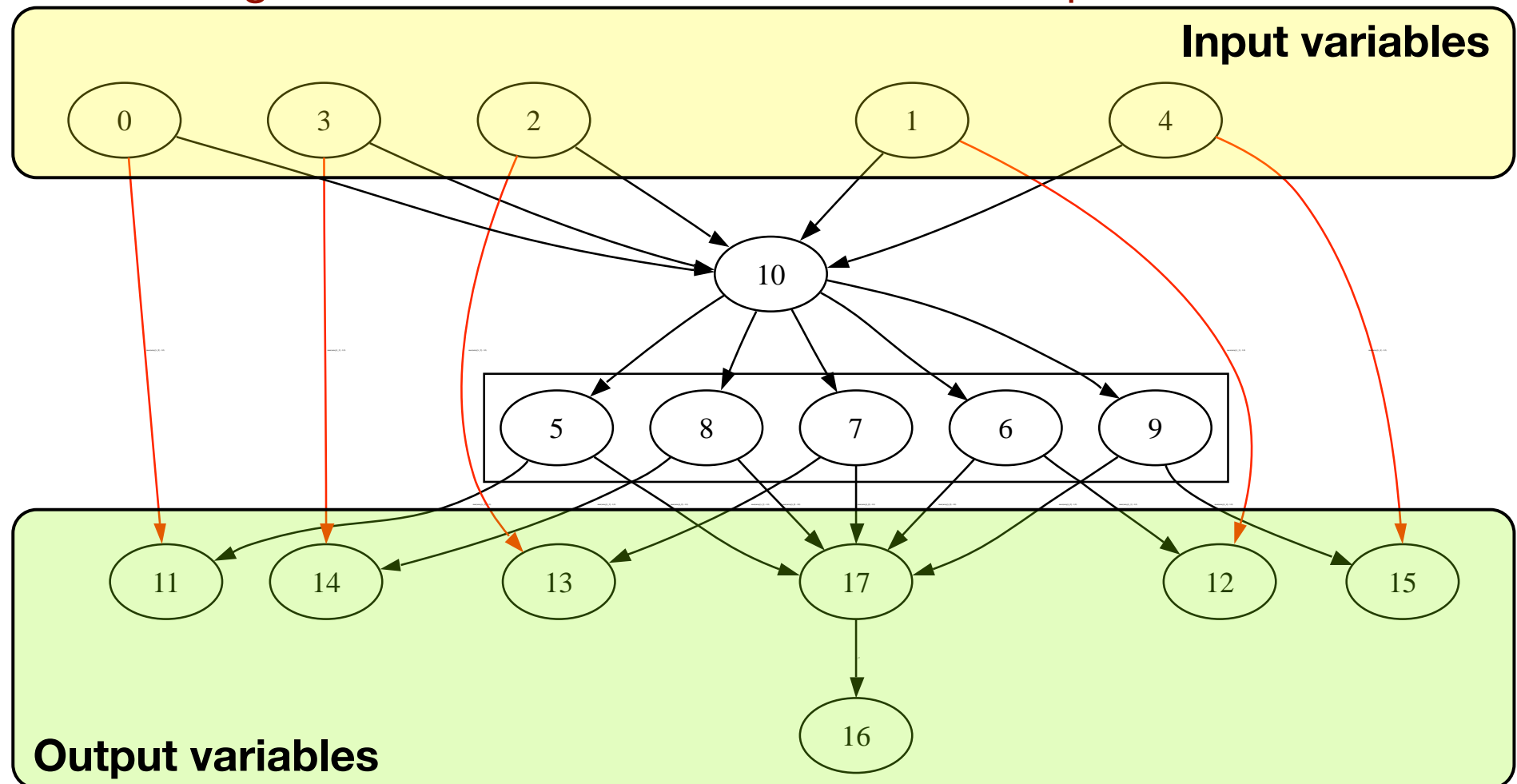
- **Basic idea**

- Replace the dedicated algorithm with declarative specification.



# Bottom-Up Propagation of Updates

- Invariants give rise to a schedule of incremental updates





# Challenge of Invariants

---

- **Getting the right complexity**
  - From any arbitrary algebraic expression
  - Form any arbitrary set-based expression
- **Accumulators**
  - Generalize the invariants
  - Can deliver better complexity
  - Capture multi-dependencies

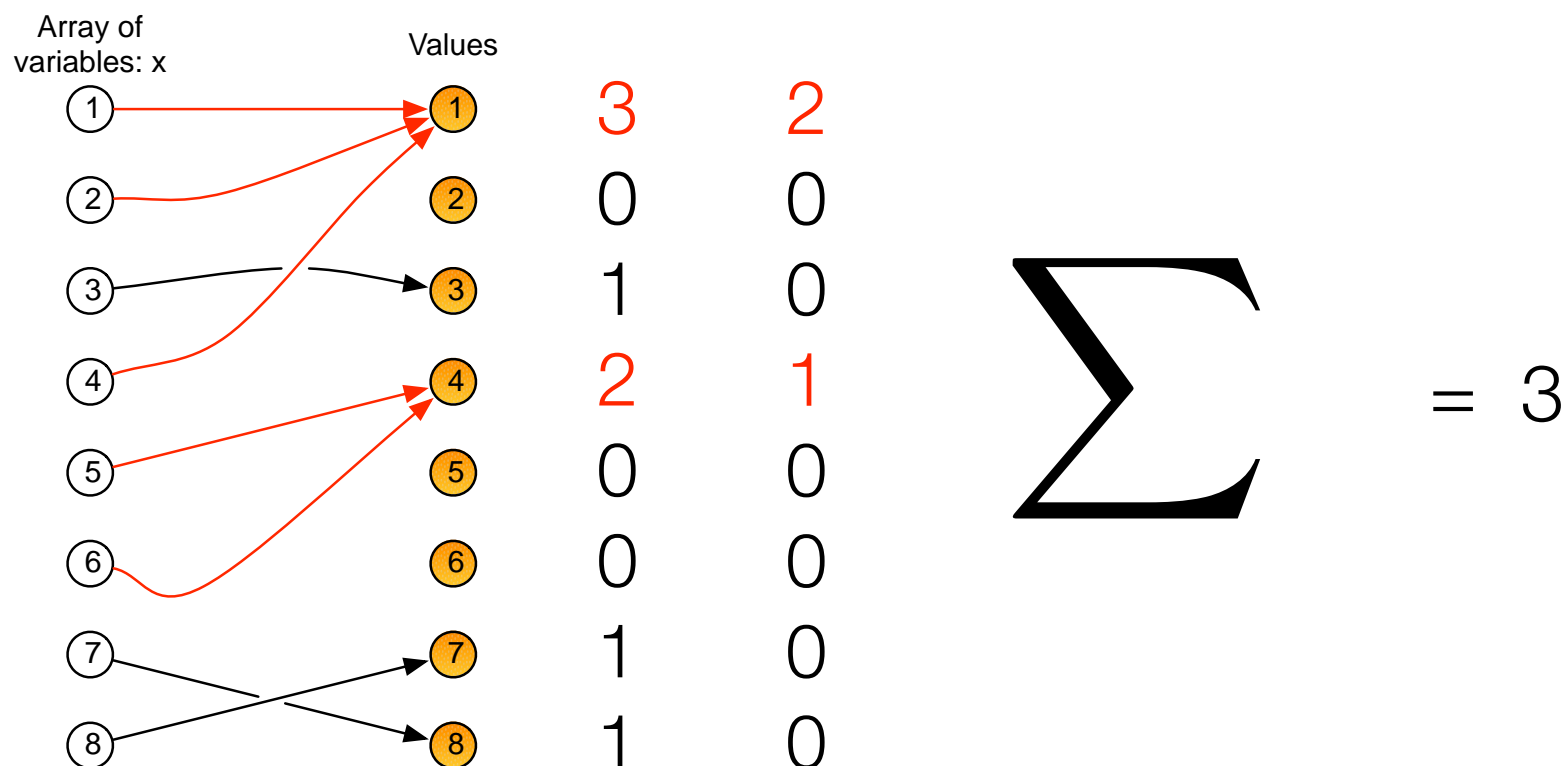


# Example: Counting Accumulators

- **Purpose**

- Count the number of variables that take specific values

- **Example: AllDifferent constraint violations**





# Counting Accumulators

---

- **Purpose**

- Count the number of variables that take specific values

- **Example: AllDifferent constraint done “batch style”**

```
1 int AllDifferent::violations() {  
2     int occ[D] = 0;  
3     forall(i in R) occ[x[i]] += 1;  
4     int violations = sum(d in D) max(0, occ[d] - 1);  
5     return violations;  
6 }
```



# Counting Accumulators

- **Central Idea**

- Lift the batch code into incremental code.

```
1  var{int} AllDifferent::stateViolations() {  
2      var{int} occ[D]() := 0;  
3      forall(i in R) occ[x[i]] <-+ 1;  
4      var{int} violations() <- sum(d in D) max(0, occ[d]-1);  
5      return violations;  
6  }  
7  var{int} AllDifferent::violations() {  
8      if (_violations == null) _violations = stateViolations();  
9      return _violations;  
10 }
```



# Stitching it together

```
0 function void conflictSearch(ConstraintSystem S) {  
1   var{int}[] x = S.getVariables();  
2   while (S.violations() > 0) {  
3     selectMax(i in x.getRange())(S.violations(x[i])  
4       selectMin(v in x[i].getDom())(S.getAssignDelta(x[i],v)  
5         x[i] := v;  
6   }
```

```
1 var{int} AllDifferent::stateViolations() {  
2   var{int} occ[D]() := 0;  
3   forall(i in R) occ[x[i]] <-+ 1;  
4   var{int} violations() <- sum(d in D) max(0,occ[d]-1);  
5   return violations;  
6 }  
7 var{int} AllDifferent::violations() {  
8   if (_violations == null) _violations = stateViolations();  
9   return _violations;  
10 }
```





# Summary

---

## •CP

- Monotonic reasoning
- Tree search
- Propagation = Filtering
- Machinery = Propagation + Backtracking

## •CBLS

- Non-monotonic reasoning (full assignment repair)
- Graph search
- Propagation = Incremental update + Incremental evaluation
- Machinery = Incremental + fast “sequential” + backtracking

# Comparison

---



	+	-
CP	complete proofs rich models	"small"
CBLS	"large" quick to high quality useful when over-constrained	incomplete



# Why Should we Hybridize?

---

- **Get the best of both worlds!**
  - Larger instances
  - Possibly recover proofs
  - Fast “diving” on high-quality solutions
  - Provide better heuristic guidance
  - Provide stronger bounding
    - It can enable stronger back-propagation!



# Hybrid Challenge

---

- Hybrids need

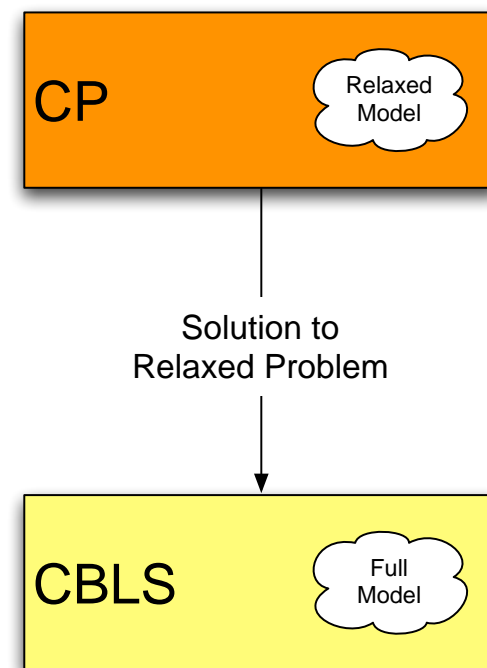
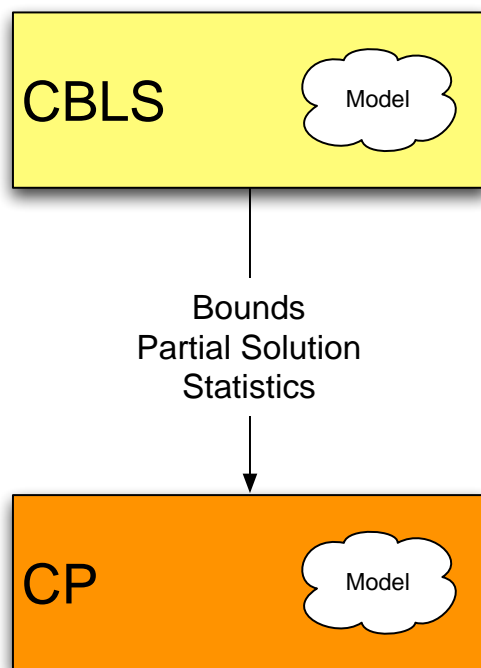
- CP technology
- CBLS technology

- But...

- Different models [e.g., symmetries help CBLS, hurt CP!]
- Different underlying implementation
- Different representations (domains vs. assignments)
- Different requirement
- Different computation models

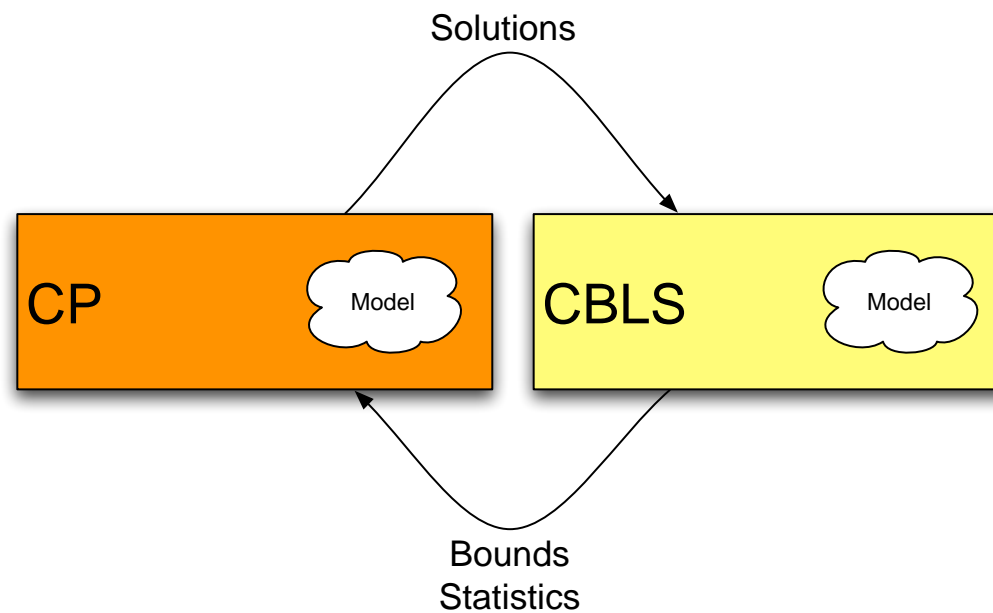
# Classes of Hybrid

- Sequential composition



# Classes of Hybrid

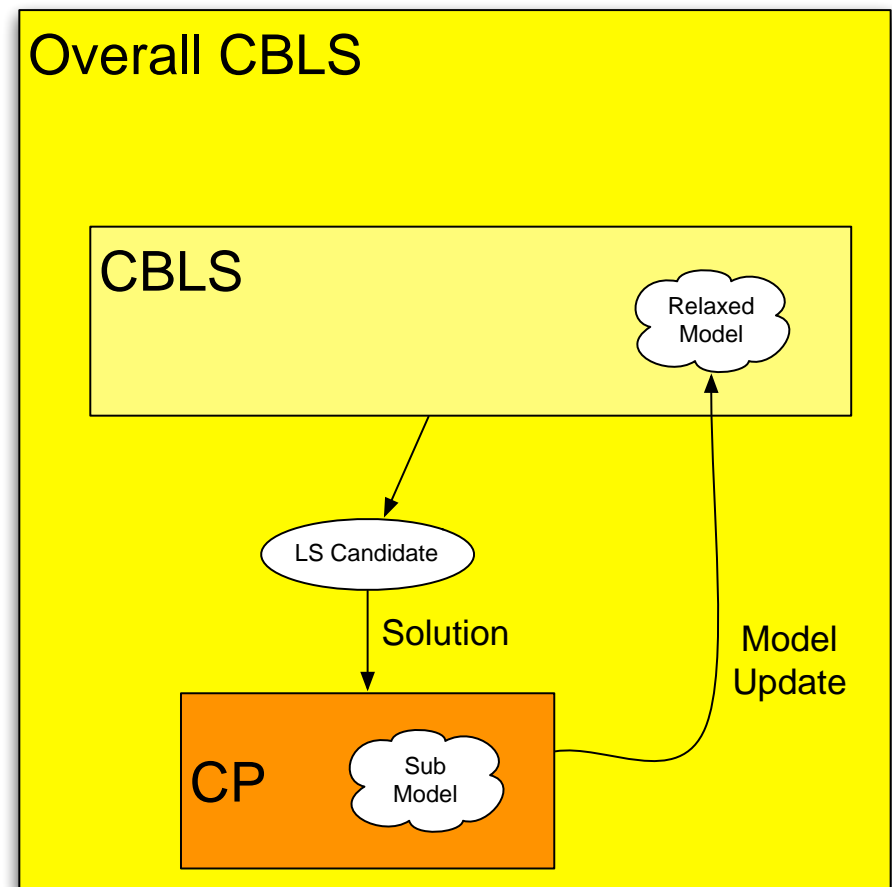
- Parallel composition



# Classes of Hybrids

- **Bender's style**

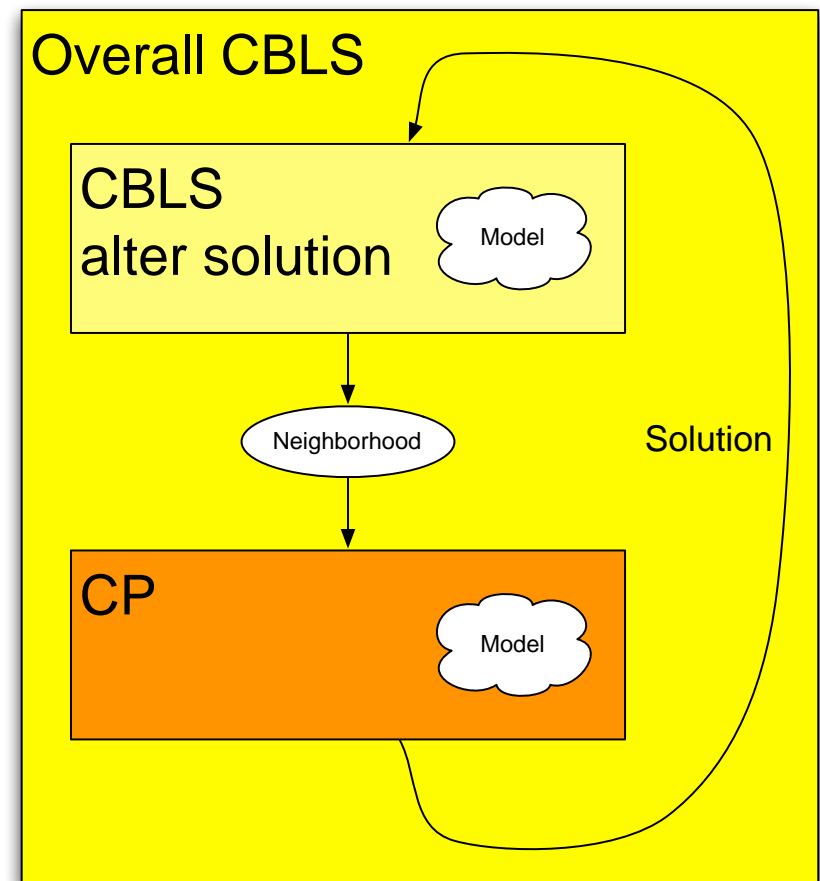
- Master
  - LS + weighted violation
- Slave
  - CP for block sub-problem



# Classes of Hybrid

- **Large Neighborhood Search**

- Overall a CBLS
- nested CBLS to “perturb”
- nested CP to explore neighborhood





# Part II

---

## Sequential & Parallel Hybrids





# Sequential Composition

---

- **Simple idea really**

- Run LS first to get a good upper-bound
- Run CP to improve upon that upper-bound

- **Benefit**

- LS can get a high-quality fast
- Better starting bound for CP means stronger back-propagation

- **Requirement**

- Write an LS model (takes advantage of CBLS support)
- Write an CP model (takes advantage of CP support)
- “String them together”



# Simple Example

---

- Consider the uncapacitated warehouse location problem
  - Hard for CP/MIP
  - LS is very good at this (high quality solution quite fast)



# Problem Statement

---

- **Given**

- A number of sites where to place warehouses
- Transportation costs for goods
  - from warehouse locations
  - to stores
- An opening cost for warehouses

- **Find**

- Where to open the warehouse to minimize
  - The fixed opening costs
  - The variable transportation costs



# CP Model

```
Solver<CP> cpm();
var<CP>{int} open[Warehouses](cpm, 0..1);
int tcostMax = max(i in Warehouses, j in Stores) tcost[i, j];
var<CP>{int} supp_tcost[Stores](cpm, 0..tcostMax);
var<CP>{int} supp_tcost_k[Stores](cpm, Warehouses);
var<CP>{int} obj(cpm, 0..100000000);

cout << "SEARCHING... " << endl;

minimize<cpm> obj
subject to {
    cpm.post(obj == sum(w in Warehouses) fcost[w] * open[w] +
              sum(i in Stores) supp_tcost[i]);
    forall(i in Stores)
        cpm.post(supp_tcost[i] == tcost[supp_tcost_k[i], i]);
    forall(i in Stores)
        cpm.post(open[supp_tcost_k[i]] == 1);
} using {
    labelFF(open);
    labelFF(cpm);
}
```

Which  
warehouse  
supports a store

Support cost  
for a store



# CBLS Model

```
int bestLS = System.getMAXINT();
Model<LS> lsm();
var{bool} open[Warehouses](lsm.getLocalSolver(), 0..1);
lsm.minimizeObj(sum(w in Warehouses) fcost[w] * open[w] +
    sumMinCost(open, tcost));
lsm.close();
TabuSearch search(lsm);
whenever search@localBest(int localBestEval, int obj, Solution s) {
    if (obj < search.getBestCost())
        cout << "LS tightening: " << obj << endl;
}
search.setMaxDiversification(10);
search.apply();
bestLS = search.getBestCost();
```

Minimal  
transportation cost

use Tabu Search

What to do each  
time we get an  
improvement

# Sequential Composition

```
int bestLS = Sys.getBestCost();
{
  Model<LS> ls = new LSModel();
  var{bool} open[Warehouses](ls, 0..1);
  ls.minimize sum(w in Warehouses) fcost[w] * open[w] +
    sum(i in Stores) supp_tcost[i];

  lsm.close();
  TabuSearch search = new TabuSearch(lsm);
  whenever search.finished() {
    if (obj < bestLS) {
      bestLS = obj;
      cout << "Found a better solution: " << obj << endl;
    }
  }
  search.setMaxDiversification(10);
  search.setMaxRestarts(2);
  search.apply();
  bestLS = search.getBestCost();
}
```

Note:

If bestLS is globally optimal...  
Then the CP model proves optimality  
So preserve the LS solution!

```
{
  Solver<CP> cpm();
  var<CP>{int} open[Warehouses](cpm, 0..1);
  int tcostMax = max(i in Warehouses, j in Stores) tcost[i, j];
  var<CP>{int} supp_tcost[Stores](cpm, 0..tcostMax);
  var<CP>{int} supp_tcost_k[Stores](cpm, Warehouses);
  var<CP>{int} obj(cpm, 0..bestLS-1);
  cout << "SEARCHING... " << bestLS-1 << endl;
  minimize<cpm> obj
  subject to {
    cpm.post(obj == sum(w in Warehouses) fcost[w] * open[w] +
      sum(i in Stores) supp_tcost[i]);
    forall(i in Stores)
      cpm.post(supp_tcost[i] == tcost[supp_tcost_k[i], i]);
    forall(i in Stores)
      cpm.post(open[supp_tcost_k[i]] == 1);
  } using {
    labelFF(open);
    labelFF(cpm);
  }
}
```



# Performance?

---

- **Benchmark**


- 1331GapBS.txt

- **CP alone**

- 342 minutes

- **CP/CBLS**

- 15 minutes

A yellow sticky note with a close button in the top right corner and a small icon in the bottom right corner. It contains two lines of text.

real 342m13.079s

real 15m4.880s





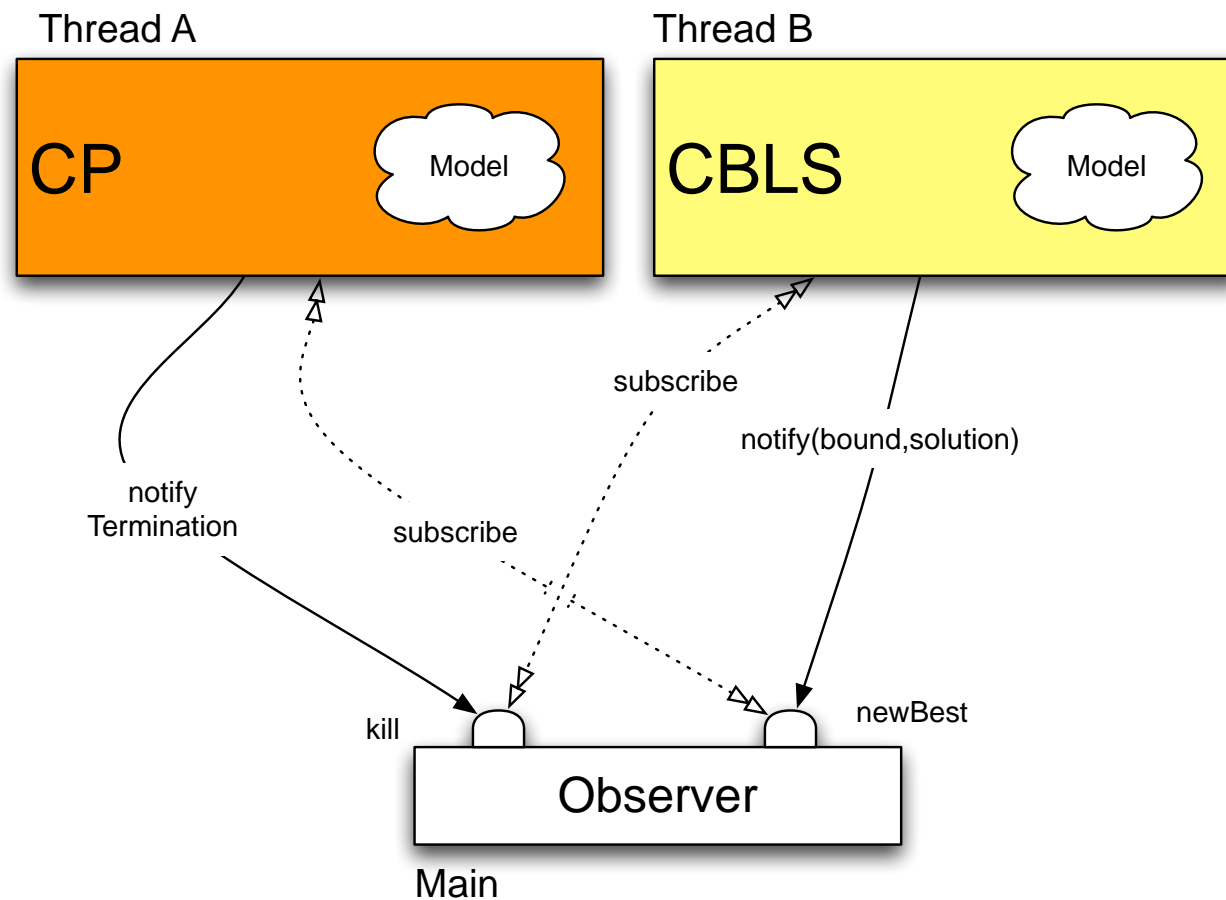
# Parallel Composition

---

- Same idea but....
  - Assume LS will run much longer
  - Assume LS produces a stream of improving bounds/solution
- Run both side-by-side
- Pass the bounds as they become available

# Communication

- Easy with the right abstraction
  - Use COMET events





# Events

---

- **Benefits**

- Separation of concerns, reuse, modularity

- **Separate**

- Animations from the constraints and search
- Constraints/Heuristics/Meta-heuristics
- GUI from algorithms

- **Why?**

- These components are independent
- They are often presented separately



# Events Anatomy

---

- **Publish**

- Event declarations inside a class

- **Subscribe**

- Many “users” can subscribe to the same event
- “when”/“whenever” construct on objects

- **Notify**

- The object implicitly notifies subscribers
- It sends information along with the notification
- Subscribers are executed upon notification



# Notifications as Messaging

---

- **Inter-thread communication**
  - Listener in one thread
  - Notifier in another
  - Notification delivered asynchronously to listener thread



# Example for Visualizations

```
LocalSolver m(); range R = 1..65000;  
var{int} queen[q in R](m,R) := q;  
ConstraintSystem S(m);  
  S.post(alldifferent(queen));  
  S.post(alldifferent(all(i in R) queen[i] + i));  
  S.post(alldifferent(all(i in R) queen[i] - i));  
  
forall(q in R)  
  whenever queen[q]@changes(int o,int n)  
    animation.updateQueens(q,o,n);
```

Code to execute  
in response to  
event

Variable notifies  
changes



# Events for Meta-Heuristics

---

- Useful for

- Tracking solutions
- Restarting logic
- Diversification/Intensification

```
Integer best();
ConstraintSystem S();
...
var{int} violations = S.violations();
Solution solution = null;

whenever violations@changes(int o,int n) {
    if (n < best) {
        solution = new Solution(m);
        best = violations;
    }
}
```



# Events for Communication

---

- Local search

- Sends notification of new solutions
- Listens for requests to end.

- CP

- Listens for new incumbent (and tighten the upper bound)
- Sends notification of proof completion





# Schema

---

```
class Observer {
  Event kill;
  Event newBest(int obj);
  Observer() {}
}
class Stop { Stop() {} }
Observer observer();
thread TLS {
  <LSMODEL>
  whenever search@localBest(int localBestEval, int obj, Solution s)
    if (obj < search.getBestCost())
      notify observer@newBest(obj);
  try {
    whenever observer@kill()
      throw new Stop();
  in <LSSEARCH>
  } catch(Stop s) {}
}
```



# Schema

```
class Observer {
    Event kill;
    Event newBest(int obj);
    Observer() {}
}
class Stop { Stop() {} }
Observer observer();
thread TLS {
    <LSMODEL>
    whenever search@localBest(int localBestEval, int obj, Solution s)
        if (obj < search.getBest())
            notify observer@newBest(obj);
    try {
        whenever observer@kill()
            throw new Stop();
        in <LSSEARCH>
    } catch(Stop s) {}
}

thread TCP {
    <CPMODEL>
    whenever observer@newBest(int best) {
        cout << "Lowering CP primal to: " << best << endl;
        <CPMODEL>.setPrimalBound(new MinimizeIntValue(best));
    }
    <CPSEARCH>
    notify observer.kill();
}
```



# Actual Code

```
class SearchObserver {
    Event kill();
    SearchObserver() {}
    Event newBest(int best);
}

class Stop { Stop() {} }

SearchObserver observer();

thread LST {
    Model<LS> lsm();
    TabuSearch search(lsm);
    var{bool} open[Warehouses](lsm.getLocalSolver(), 0..1);
    lsm.minimizeObj(sum(w in Warehouses) fcost[w] * open[w]
                    sumMinCost(open, tcost));
    lsm.close();
    whenever search@localBest(int localBestEval, int obj, So
        if (obj < search.getBestCost()) {
            cout << "New LS bound: " << obj << endl;
            notify observer@newBest(obj);
        }
    }
    search.setMaxDiversification(10);
    whenever observer@kill()
        throw new Stop();
    in
        search.apply();
}
```

```
thread CPT {
    Solver<CP> cpm();
    var<CP>{int} open[Warehouses](cpm, 0..1);
    int tcostMax = max(i in Warehouses, j in Stores) tcost[i, j];
    var<CP>{int} supp_tcost[Stores](cpm, 0..tcostMax);
    var<CP>{int} supp_tcost_k[Stores](cpm, Warehouses);
    var<CP>{int} obj(cpm, 0..1000000);

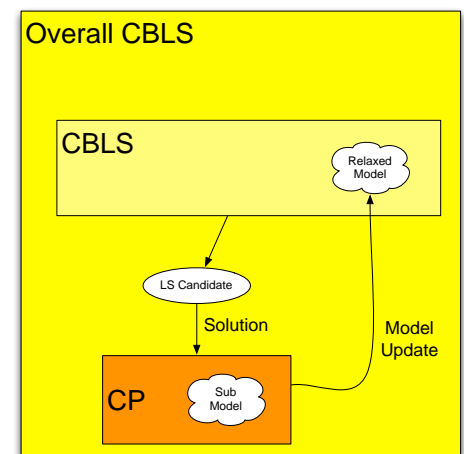
    minimize<cpm> obj
    subject to {
        cpm.post(obj == sum(w in Warehouses) fcost[w] * open[w] +
                  sum(i in Stores) supp_tcost[i]);
        forall(i in Stores)
            cpm.post(supp_tcost[i] == tcost[supp_tcost_k[i], i]);
        forall(i in Stores)
            cpm.post(open[supp_tcost_k[i]] == 1);

        whenever observer@newBest(int best) {
            cout << "Lowering CP primal to: " << best << endl;
            cpm.setPrimalBound(new MinimizeIntValue(best));
        }
    }
    using {
        labelFF(open);
        labelFF(cpm);
    }
    notify observer@kill();
}
```

# Application

---

RAMBO





# Overview

---

- RAMBO in a Nutshell
- Quorum Systems
- The Quorum Configuration Problem
- Modeling RAMBO
  - Mathematical Model
  - Constraint Programming Model
  - Hybrid CBLS/CP Model
  - Parallel Composition Model
- Experimental Results

# RAMBO in a Nutshell

---

## **R**econfigurable **A**tomistic **M**emory for **B**asic **O**bjects

[Lynch & Shvartsman, 02]

- Replicated data objects in a dynamic network
- Algorithm tolerates:
  - Processor joins, leaves, and stopping failures
  - Message loss
  - Arbitrary patterns of asynchrony
- **Example:** mobile networks
- Specified with a formal language:  $[T]IOA$  [Lynch, 89; Kaynar, 06]





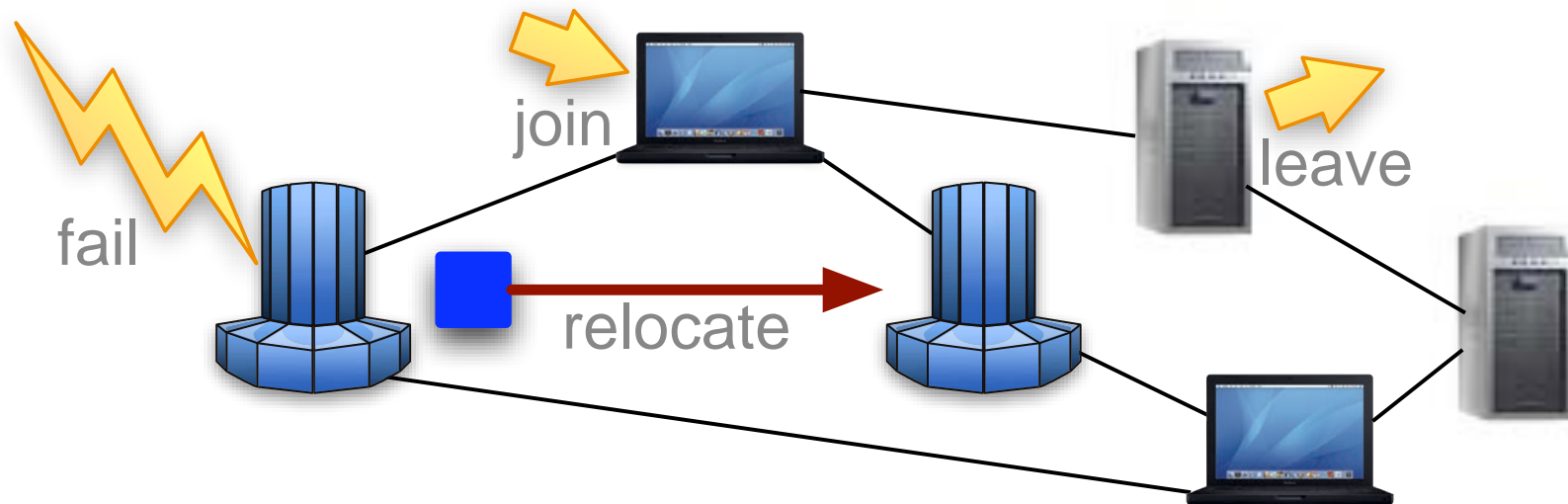
# RAMBO Features

---

- Participants communicate using background [gossiping](#)
  - Every participant sends status to every other participant
  - No separate messages to read or write data
- To guarantee atomicity, operations performed using [quorums](#)
  - Fault tolerance
  - Single-copy centralized view of data presented to users
- To survive network changes, quorums are [reconfigurable](#)

# What is Reconfiguration?

- Relocation of quorum members as participants join, leave, & fail



- Goal: long, reliable life for RAMBO service





# RAMBO Reconfiguration

---

- **Algorithm specifies:**

- How new quorum configurations are installed
- How old quorum configurations are “garbage collected”



- **Algorithm does NOT specify:**

- When reconfiguration should take place
- What the new configuration should be

Focus of paper



# Overview

---

- RAMBO in a Nutshell
- Quorum Systems
- The Quorum Configuration Problem
- Modeling RAMBO
  - Mathematical Model
  - Constraint Programming Model
  - Hybrid CBLS/CP Model
  - Parallel Composition Model
- Experimental Results



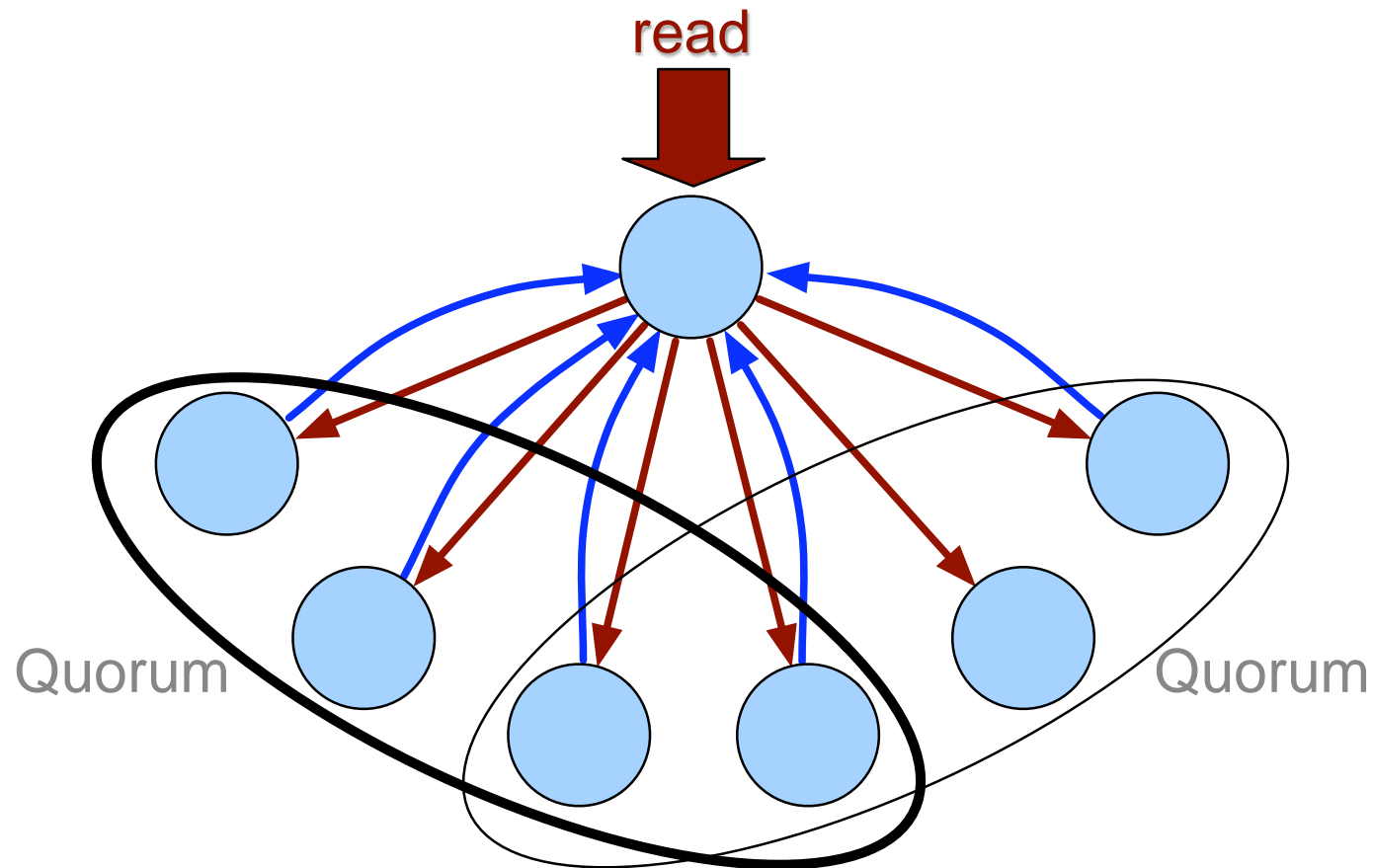
# Quorum System

---

- Set of quorum members:  $\mathcal{M}$
- Set of quorums:  $\mathcal{Q} \subseteq \mathcal{P}(\mathcal{M})$
- Every quorum intersects with every other quorum
  - $\forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \cap Q_2 \neq \emptyset$
- Data values time-stamped or tagged to mark order
  - Every operation contacts a quorum
  - Overlap of quorums guarantees previous operation seen
- Example: Majorities [Thomas, 79]
  - Every majority intersects with every other majority



# Contacting a Quorum

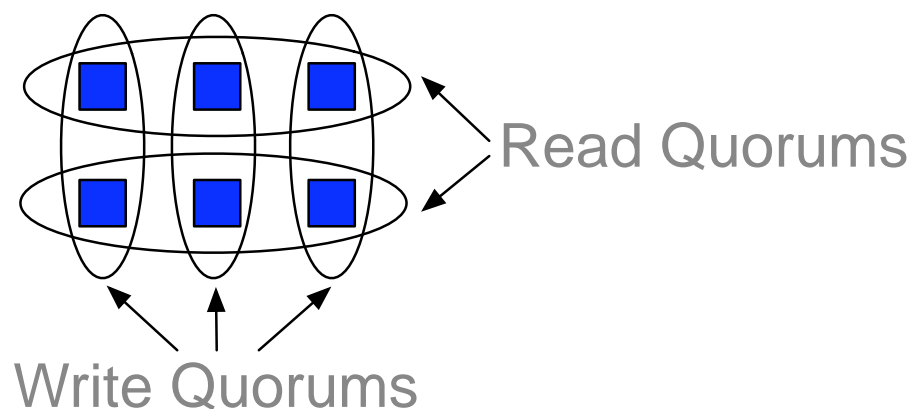


Operation completes when response received from slowest member of fastest quorum



# Read and Write Quorum System

- Set of quorum members:  $\mathcal{M}$
- Sets of read and write quorums:  $\mathcal{R}, \mathcal{W} \subseteq \mathcal{P}(\mathcal{M})$  ← Used by RAMBO
- Every read quorum intersects every write quorum and vice versa
  - $\forall R \in \mathcal{R}, \forall W \in \mathcal{W} : R \cap W \neq \emptyset$
- Typically smaller than general quorums
- Example:





# RAMBO Read and Write Operations

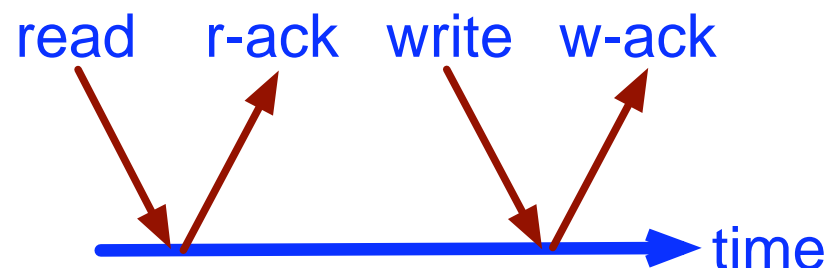
---

- To read data object:

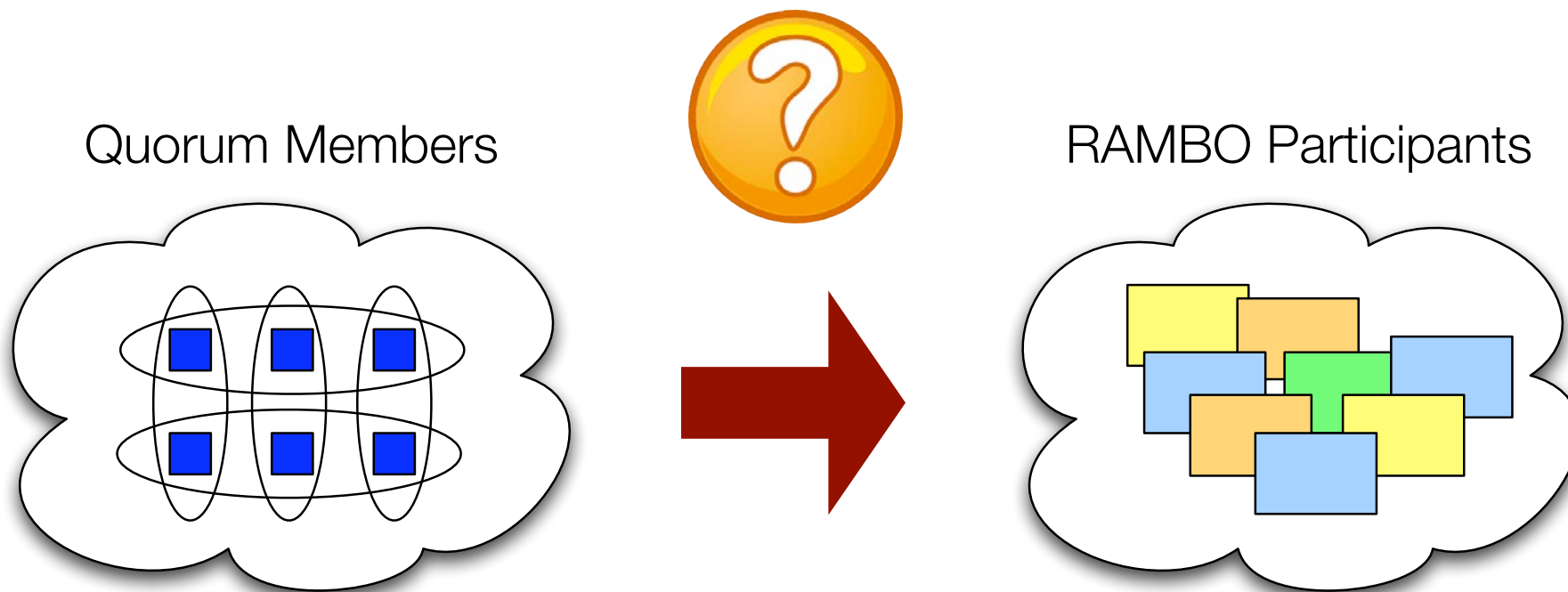
- Gather value from a read quorum (of every active configuration)
- Propagate most recent value to a write quorum

- To write data object:

- Gather value from a read quorum
- Update value and tag
- Propagate new value to a write quorum



# Quorum Configuration Problem



- maximize fault tolerance
- minimize communication costs
- balance loads on quorum members



# Overview

---

- RAMBO in a Nutshell
- Quorum Systems
- The Quorum Configuration Problem
- Modeling RAMBO
  - Mathematical Model
  - Constraint Programming Model
  - Hybrid CBLS/CP Model
  - Parallel Composition Model
- Experimental Results





# RAMBO Communication Costs

---

- Participants have no knowledge of underlying network
- **Assumption:** Current behaviors are best predictors for future behaviors
- Participants collect behavior observations
  - Average message round-trip delay with every participant
  - Average frequency of read and write operations
- Behavior observations added to gossip messages



# Formal Quorum Configuration Problem

---

- **Input data**

- $H$ : set of host participants
- $f_h$ : average frequency of read and write requests for host  $h$
- $d_{h_1, h_2}$ : average round-trip delay from host  $h_1$  to host  $h_2$
- $c$ : abstract configuration to be deployed on  $H$ , consisting of:
  - ▶  $\mathcal{M}$ : set of members
  - ▶  $\mathcal{R}$ : set of read quorums with  $\mathcal{R} \subseteq \mathcal{P}(\mathcal{M})$
  - ▶  $\mathcal{W}$ : set of write quorums with  $\mathcal{W} \subseteq \mathcal{P}(\mathcal{M})$
- $\alpha$ : load balancing factor



# Formal Quorum Configuration Variables

---

- **Decision variables**

- $x_m$  : host for quorum member  $m$
- $readQ_h$  : best (minimum delay) read quorum for host  $h$
- $writeQ_h$  : best (minimum delay) write quorum for host  $h$



only used for load balancing



# Formal Objective and Constraints

## •Objective

$$\min \left( \sum_{h \in H} f_h \times \left( \min_{q \in \mathcal{R}} \left( \max_{m \in q} d_{h, x_m} \right) + \min_{q \in \mathcal{W}} \left( \max_{m \in q} d_{h, x_m} \right) \right) \right)$$

 min/max                       min/max

## •Separation Constraint

$$\forall m, m' \in \mathcal{M} : m \neq m' \Rightarrow x_m \neq x_{m'}$$



# Formal Constraints (continued)

---

- Load Balancing Constraints

$$readQ_h = r \Rightarrow \max_{m \in r} d_{h,x_m} = \min_{q \in \mathcal{R}} \left( \max_{m \in q} d_{h,x_m} \right)$$

$$writeQ_h = w \Rightarrow \max_{m \in w} d_{h,x_m} = \min_{q \in \mathcal{W}} \left( \max_{m \in q} d_{h,x_m} \right)$$

$$load_m = \sum_{h \in H} \left( \sum_{m \in readQ_h} f_h + \sum_{m \in writeQ_h} f_h \right)$$

$$\max_{m \in \mathcal{M}} load_m \leq \alpha \times \left( \min_{m \in \mathcal{M}} load_m \right)$$



# Overview

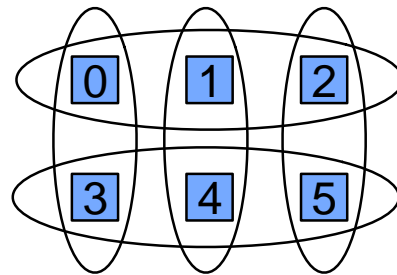
---

- RAMBO in a Nutshell
- Quorum Systems
- The Quorum Configuration Problem
- Modeling RAMBO
  - Mathematical Model
  - Constraint Programming Model
  - Hybrid CBLS/CP Model
  - Parallel Composition Model
- Experimental Results



# Variable Symmetries

- Symmetry breaking among quorum members [Smith, 05]
  - Added input data: *Order* : set of member pairs  $\langle m, m' \rangle$
  - Added constraint:  $\langle m, m' \rangle \in \textit{Order} \Rightarrow x_m < x_{m'}$



$$\textit{Order} = \{ \langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 0, 3 \rangle, \langle 0, 4 \rangle, \langle 0, 5 \rangle, \langle 1, 2 \rangle \}$$



# CP Model for Reconfiguration

---

- Added calculated parameters

- $nbrQ_m$ : number of quorums to which member  $m$  belongs
- $degree_h$ : approximate number of neighbors of host  $h$
- $readQC_{m,r}$ : true if  $m$  is a member of read quorum  $r$
- $writeQC_{m,w}$ : true if  $m$  is a member of write quorum  $w$
- $RQ$ ,  $WQ$  ranges of  $\mathcal{R}$  and  $\mathcal{W}$

- Added auxiliary variables

- $readD_{h,r}$ : delay if host  $h$  uses read quorum  $r$
- $writeD_{h,w}$ : delay if host  $h$  uses write quorum  $w$





# Declarations in CP Model

- Simple, direct encoding in COMET

```
Solver<CP> cp();
range H = ...;           // The Hosts
int[] f = ...;           // The frequency matrix
int[,] d = ...;          // The delays matrix
range M = ...;           // The Members of quorums
set{int} R = ...;         // The Read quorums
set{int} W = ...;         // The Write quorums
int alpha = ...;         // The load factor
set{tuple{;}} Order = ...; // The Order of quorum members

int[] nbrQ = ...;         // The number of quorums per member
int[] degree = ...;       // The approx. degree of each host
boolean[,] readQC = ...;  // True if member in read quorum
boolean[,] writeQC = ...; // True if member in write quorum
range RQ = R.getRange();  // The range of read quorums
range WQ = W.getRange();  // The range of write quorums
```



# Decision Variables & Objective in CP Model

---

```
var<CP>{int} x[m in M] (cp,H);
var<CP>{int} readQ[h in H] (cp,RQ);
var<CP>{int} writeQ[h in H] (cp,WQ);

var<CP>{int} load[m in M] (cp,0..10000);
var<CP>{int} readD[h in H,r in RQ] (cp,0..10000);
var<CP>{int} writeD[h in H,w in WQ] (cp,0..10000);

minimize<cp> sum(h in H) f[h] *
    (min(r in RQ) readD[h,r] + min(w in WQ) writeD[h,w])
```



# Constraints in CP Model

```
subject to {  
    cp.post(alldifferent(x), onDomains);                                separation  
  
    forall (o in Order) cp.post(x[o.low] < x[o.high]);                symmetry  
  
    forall (h in H, r in RQ)  
        cp.post(readD[h,r] == max(m in R[r]) d[h,x[m]]);            load  
                                                                           balancing  
  
    forall (h in H, w in WQ)  
        cp.post(writeD[h,w] == max(m in W[w]) d[h,x[m]]);  
  
    forall (h in H) {  
        cp.post(readD[h,readQ[h]] == min(r in RQ) readD[h,r]);  
        cp.post(writeD[h,writeQ[h]] == min(w in WQ) writeD[h,w]);  
    }  
  
    forall (m in M) cp.post(load[m] == sum(h in H) f[h] *  
        (readQC[m,readQ[h]] + writeQC[m,writeQ[h]]));  
  
    cp.post(max(m in M) load[m] <= alpha * min(m in M) load[m]);  
}
```



# Search Procedure in CP Model

- Two phases: minimize cost and balance load

using {

```
while (sum(k in M) x[k].bound() < M.getSize())  
  selectMax(m in M: !x[m].bound()) (nbrQ[m])  
    tryall<cp>(h in H: x[m].memberOf (h)) by (-degree[h])  
      cp.label(x[m], h);  
      onFailure cp.diff(x[m], h);
```

deploy members

➡ once<cp>

```
  forall (h in H: !readQ[h].bound() || !writeQ[h].bound()) by (-f[h]) {  
    label(readQ[h]);  
    label(writeQ[h]);  
  }  
}
```

assign quorums



# Overview

---

- RAMBO in a Nutshell
- Quorum Systems
- The Quorum Configuration Problem
- Modeling RAMBO
  - Mathematical Model
  - Constraint Programming Model
  - Hybrid CBLS/CP Model
  - Parallel Composition Model
- Experimental Results



# Hybrid CBLS/CP Model

---

- RAMBO reconfiguration must occur before old configuration fails
- Local search used to find a high-quality solution quickly
- Master/slave approach [Benders, 62]
  - Master: local search
    - ▶ Finds a deployment that minimizes communication costs
    - ▶ Uses simulated annealing [Kirkpatrick et. al., 83; Johnson et. al., 89]
  - Slave: finite-domain
    - ▶ Finds one feasible quorum assignment that balances load



# Master for CBLS/CP Model

```
// Same declarations as in pure CP plus the following:
FunctionSum<LS> 0 (ls) = ...; // Same objective as in pure CP
var{int} v (ls,0..1) := 1;    // boolean to indicate success of slave
var{float} w (ls) := 0.5;    // weight
var{float} obj (ls) <- sqrt(0.value()^2 + (w*v)^2); // combined obj.

while (it < maxIt) {
  select (m in M, n in H: m != n) {
    float delta = (lookahead (ls, obj) makeMove (m, n);) - obj;
    if (distr.accept (-delta/t)) {
      v := makeMove (m, n);
      if (v == 0) bf = min (obj, bestFeasible);
      else bi = min (obj, bestInfeasible);
    }
  }
  it++; stableIt++; t = t * 0.9995; w := w + 0.01;
  if (bf < bestFeasible || bi < bestInfeasible) updateBest();
  if (stableIt >= 1000) reheat();
  if (rounds >= 10) diversify();
}
```



# Master for CBLS/CP Model: makeMove

```
var<CP>{int} loadMin[m in M] (ls) <- ...;    \\ minimum possible load
var<CP>{int} loadMax[m in M] (ls) <- ...;    \\ maximum possible load

function int makeMove (int m, int n) {
  x[m] :=: x[n];
  if ((O.value() < bestF) && (max (m in M) loadMin[m] <= alpha *
    min (m in M) loadMax[m])) {
    return isFeasible() ? 0 : 1;
  }
  else return 1;
}
```





# Slave for CBLS/CP Model

```
function boolean isFeasible () {
  Solver<CP> cp();
  cp.limitFailures(1000);
  var<CP>{int} readQ[h in H](cp, RQ);
  var<CP>{int} writeQ[h in H](cp, WQ);
  var<CP>{int} load[M](cp, 0..100000);
  solve<cp> {
    forall (h in H) {
      cp.post(readD[h, readQ[h]] == min(r in RQ) readD[h,r]);
      cp.post(writeD[h, writeQ[h]] == min(w in WQ) writeD[h,w]);
    }
    forall (m in M) cp.post(load[m] == sum(h in H) f[h] *
      (readQC[readQ[h],m] + writeQC[writeQ[h],m]));
    cp.post(max (m in M) load[m] <= alpha * min (m in M) load[m]);
  } using {
    forall(h in H: !readQ[h].bound() || !writeQ[h].bound()) by (-f[h]) {
      label (readQ[h]); label (writeQ[h]); }
    return true;
  }
  return false;
}
```



# Parallel Composition Model

---

- **CP model and Hybrid CBLS/CP model run in separate threads**
  - Hybrid model notifies CP model with each new solution
  - CP model notifies Hybrid model on termination with optimum
- **Expected improvements**
  - Optimal solution found more quickly
  - Proof of optimality completed more quickly



# Overview

---

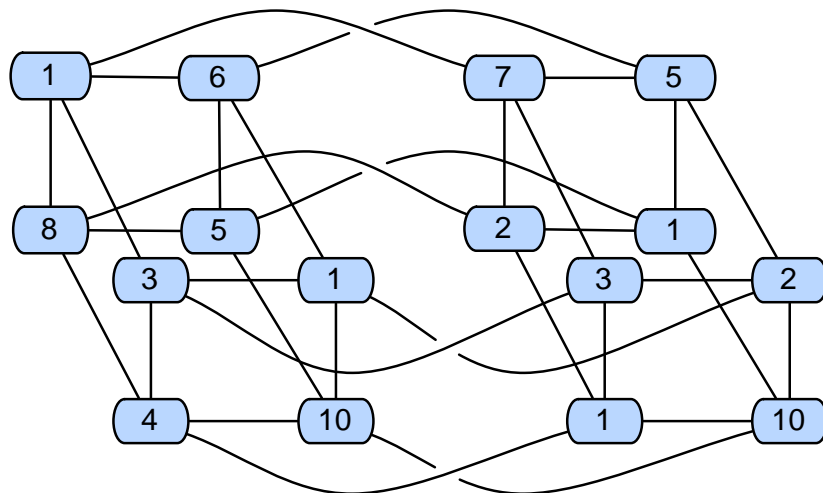
- RAMBO in a Nutshell
- Quorum Systems
- The Quorum Configuration Problem
- Modeling RAMBO
  - Mathematical Model
  - Constraint Programming Model
  - Hybrid CBLS/CP Model
  - Parallel Composition Model
- Experimental Results

- frequency  
of operations

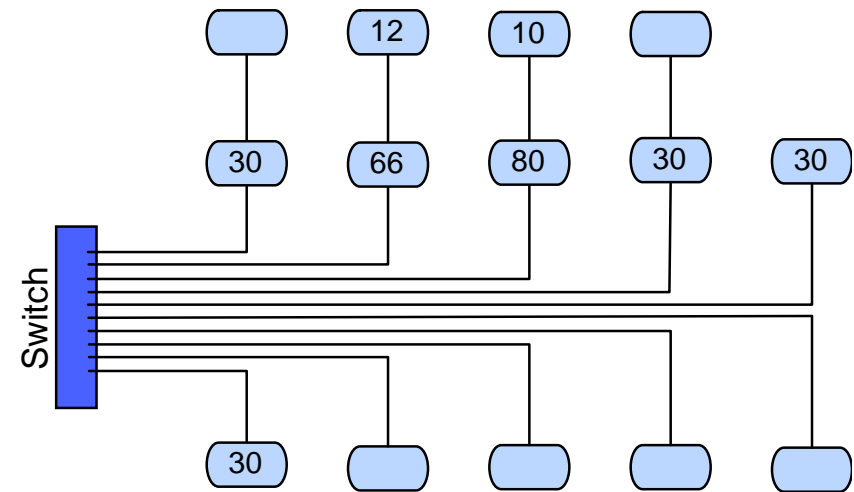


# Benchmark Networks

- 16 components in a hypercube and 14 components with a switch



Hyper16

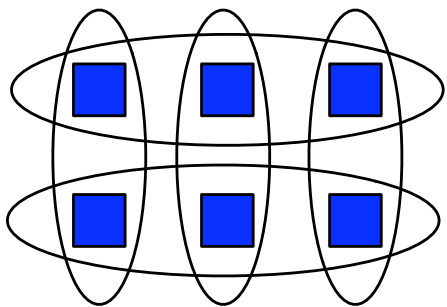


Switch

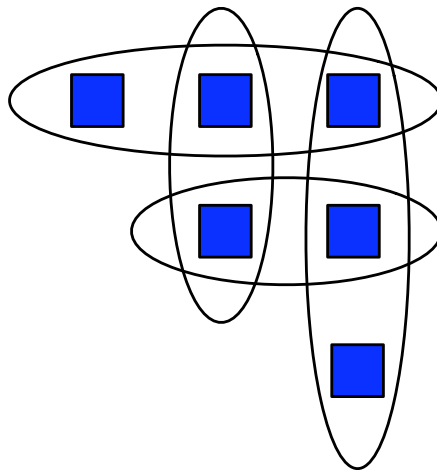


# Benchmark Quorum Systems

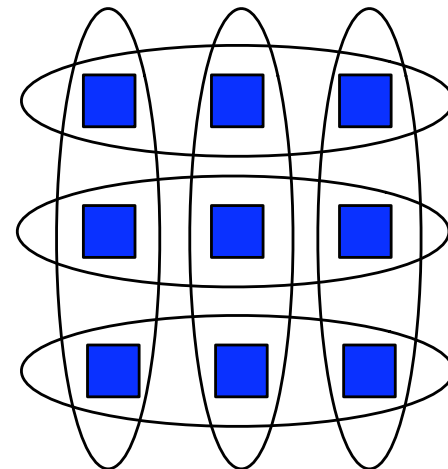
Majority:  $\{\{0, 1, 2, 3\}, \{1, 2, 3, 4\}, \{2, 3, 4, 5\}, \{0, 1, 4, 5\},$   $\leftarrow \mathcal{R}$   
 $\{0, 1, 3, 4\}, \{0, 1, 2, 5\}, \{1, 2, 4, 5\}, \{0, 3, 4, 5\}\}$   $\leftarrow \mathcal{W}$



3x2



3Step



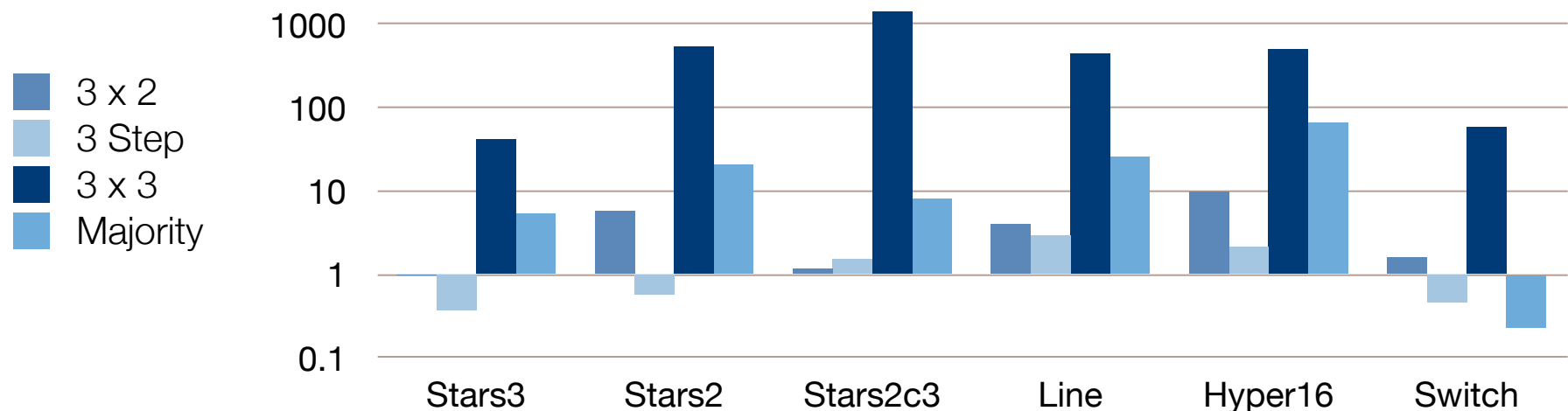
3x3

3x3 quorum system is  
hardest for all benchmarks;  
3Step is easiest for all

Some optimal solutions  
found early in search;  
others found late in search



$\alpha = 2$	3 x 2		3 Step		3 x 3		Majority	
Benchmark	T_end	T_opt	T_end	T_opt	T_end	T_opt	T_end	T_opt
Stars3	0.98	0.41	0.37	0.09	42.12	10.95	5.30	0.03
Stars2	5.70	2.76	0.57	0.30	536.50	92.69	20.76	1.35
Stars2c3	1.16	0.21	1.52	1.37	1414.09	914.20	8.07	0.03
Line	4.02	2.76	2.92	2.59	445.28	319.96	26.04	6.91
Hyper16	9.87	5.04	2.13	0.85	508.28	226.04	66.94	3.37
Switch	1.59	0.71	0.46	0.30	59.03	45.06	0.23	0.08



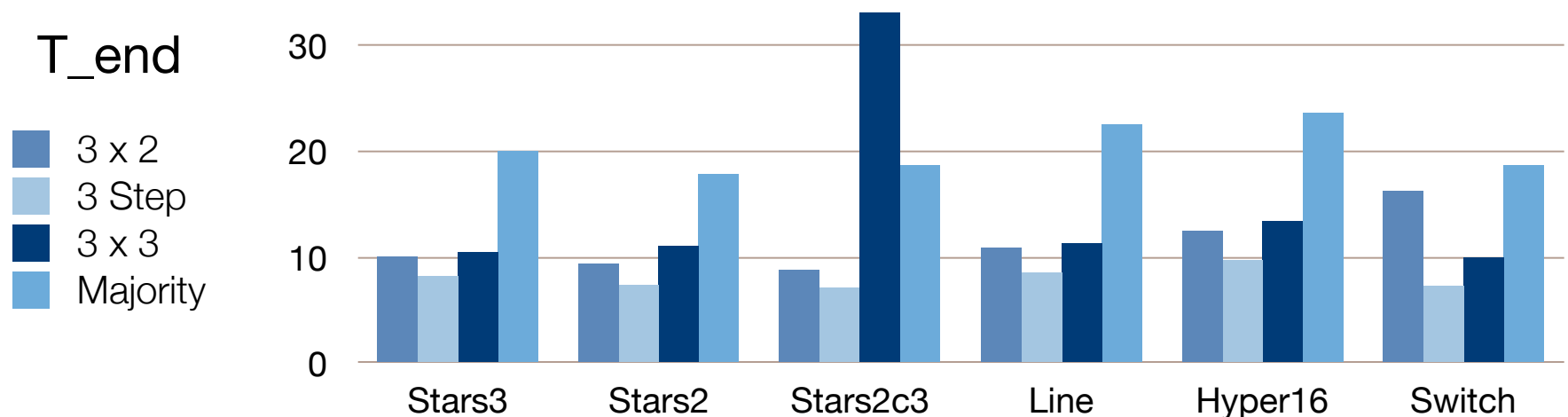
Optimal solution found very reliably (#Opt is number of times out of 50)

Even in this case, the average best solution is only 1% greater than optimum



All benchmarks complete in less than 34 seconds

All benchmarks complete in less than 34 seconds	3 Step			3 x 3			Majority					
	End	T_best	#Opt	T_end	T_best	#Opt	T_end	T_best	#Opt			
	1.43	50	10.48	1.61	50	19.99	0.36	50				
	0.38	50	11.03	1.79	50	17.85	0.29	50				
	0.32	50	33.12	17.10	13	18.71	0.25	50				
Line	10.83	1.24	50	8.52	2.23	48	11.25	5.57	16	22.53	0.70	50
Hyper16	12.46	2.97	48	9.69	2.80	46	13.35	2.98	3	23.61	8.37	48
Switch	16.26	0.10	50	7.26	0.03	50	9.95	0.09	50	18.66	0.12	50



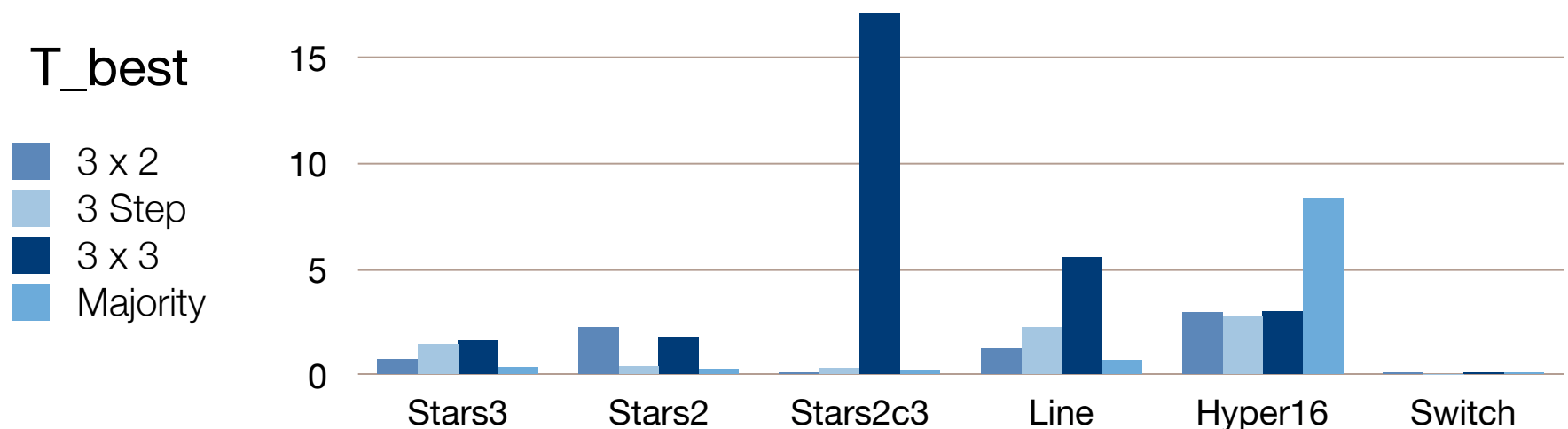


Optimum found faster than CP model for all but the easiest benchmarks

Large number of invocations of slave search in this case



$\alpha = 2$	3 x 2			3 Step			3 x 3			Majority		
Benchmark	T_end	T_best	#Opt	T_end	T_best	#Opt	T_end	T_best	#Opt	T_end	T_best	#Opt
Stars3	10.02	0.74	50	8.18	1.43	50	10.48	1.61	50	19.99	0.36	50
Stars2	9.37	2.24	50	7.37	0.38	50	11.03	1.79	50	17.85	0.29	50
Stars2c3	8.74	0.11	50	7.07	0.32	50	33.12	17.10	13	18.71	0.25	50
Line	10.83	1.24	50	8.52	2.23	48	11.25	5.57	16	22.53	0.70	50
Hyper16	12.46	2.97	48	9.69	2.80	46	13.35	2.98	3	23.61	8.37	48
Switch	16.26	0.10	50	7.26	0.03	50	9.95	0.09	50	18.66	0.12	50

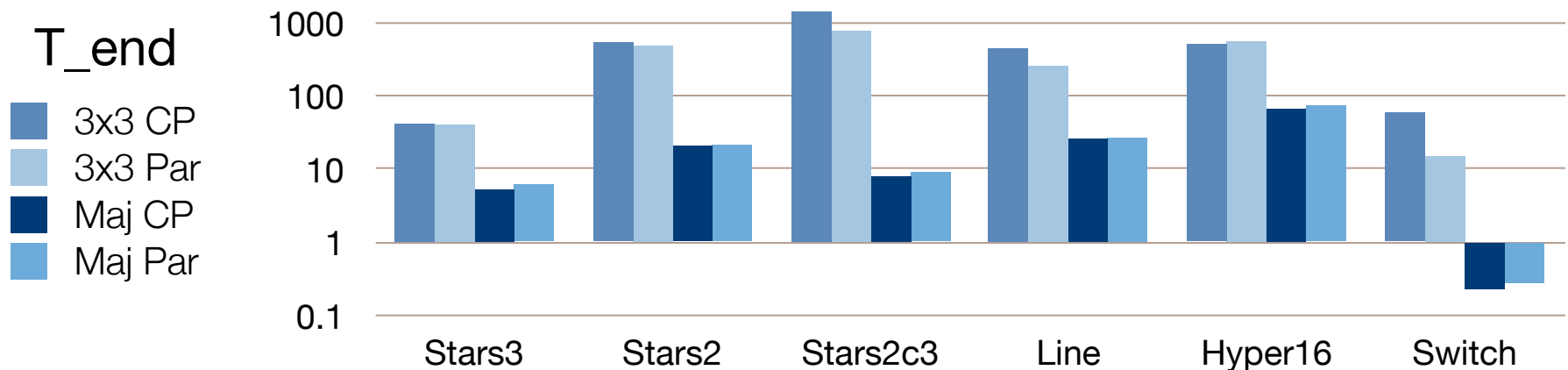


Most improvement with  
parallel composition model  
in these cases



## Parallel Results

$\alpha = 2$ Benchmark	3 x 3				Majority			
	CP		Parallel		CP		Parallel	
	T_end	T_opt	T_end	T_opt	T_end	T_opt	T_end	T_opt
Stars3	42.12	10.95	40.88	3.16	5.30	0.03	6.16	0.41
Stars2	536.50	92.69	490.79	3.14	20.76	1.35	21.30	0.58
Stars2c3	1414.09	914.20	779.81	175.85	8.07	0.03	9.07	0.27
Line	445.28	319.96	257.86	75.15	26.04	6.91	26.95	0.78
Hyper16	508.28	226.04	553.73	262.76	66.94	3.37	73.62	6.83
Switch	59.03	45.06	15.14	0.10	0.23	0.08	0.28	0.13

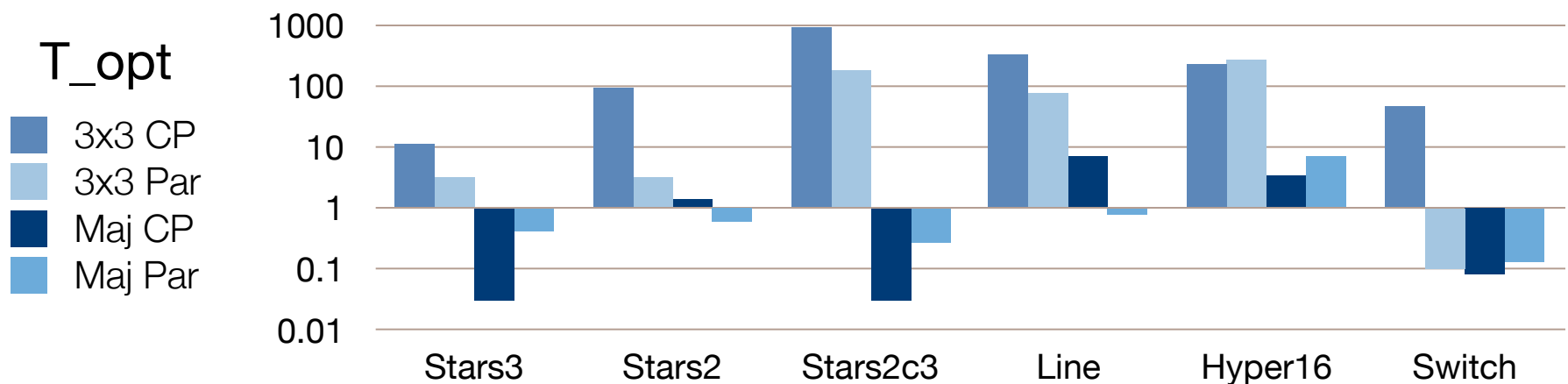


Parallel composition model  
delivers optimum more  
quickly in almost all cases

## Parallel Results



$\alpha = 2$ Benchmark	3 x 3				Majority			
	CP		Parallel		CP		Parallel	
	T_end	T_opt	T_end	T_opt	T_end	T_opt	T_end	T_opt
Stars3	42.12	10.95	40.88	3.16	5.30	0.03	6.16	0.41
Stars2	536.50	92.69	490.79	3.14	20.76	1.35	21.30	0.58
Stars2c3	1414.09	914.20	779.81	175.85	8.07	0.03	9.07	0.27
Line	445.28	319.96	257.86	75.15	26.04	6.91	26.95	0.78
Hyper16	508.28	226.04	553.73	262.76	66.94	3.37	73.62	6.83
Switch	59.03	45.06	15.14	0.10	0.23	0.08	0.28	0.13

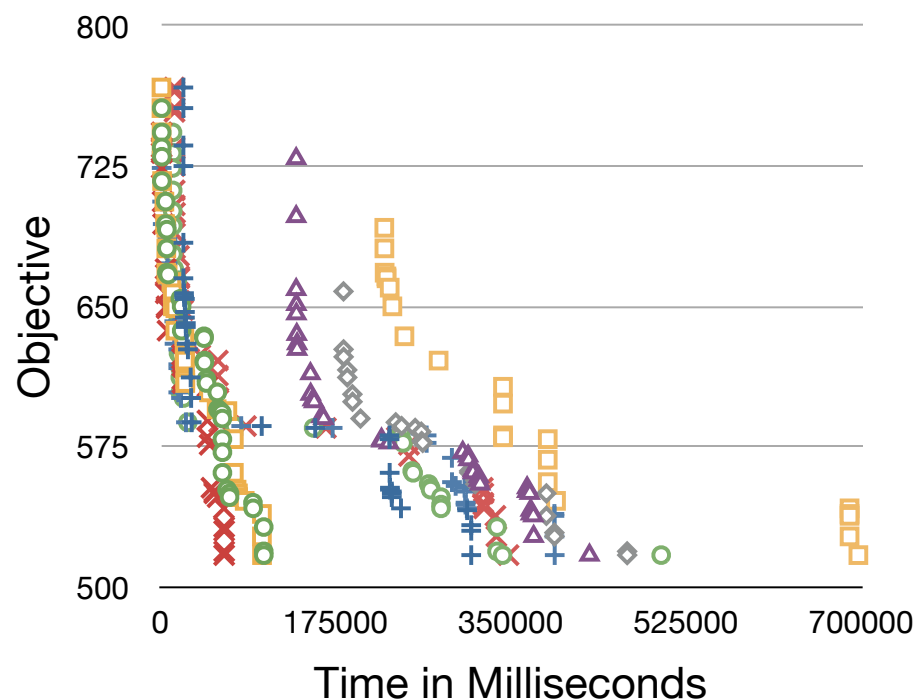




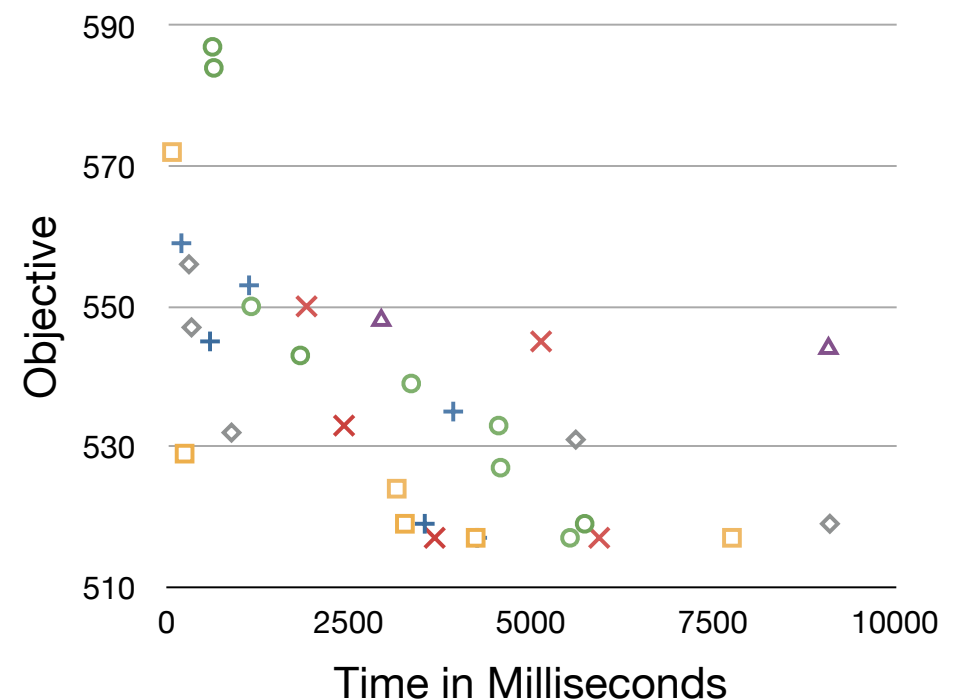
# Comparison of CP and Parallel Results

- Parallel composition model delivers good values very quickly

## Line 3x3 CP



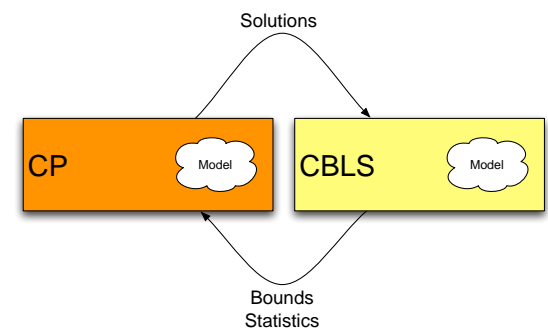
## Line 3x3 Parallel



# Application

---

A Hybrid LS/CP Approach to Solve the Weekly Log-Truck Scheduling Problem  
[Gendreau et al., 2009]





# Application Overview

---

- **Objective**

- Schedule the provisioning of wood mills from forests
- Provision the right type of wood [hardwood, pine, ....]
- Satisfy forest inventory
- Satisfy mill storage (buffer) constraints
- Minimize transportation cost and forest access cost



# Challenge

---

- **Multi-scale!**

- Weekly horizon [“mid-term” scheduling]
  - Coarse modeling
  - Focus on stocks
  - Min-cost flow model
- Daily horizon [“short-term” scheduling]
  - Given the weekly target and daily decision on sites....
  - Schedule the actual trips for the trucks
  - Taking into account the resources (Loaders/Unloaders)

- **Decompose!**

- Solve hierarchically as well



# Weekly Problem

---

- “Easy” to describe as an IP!
- Reasonably solved with a good IP solver.
- Relax the low-level scheduling constraints

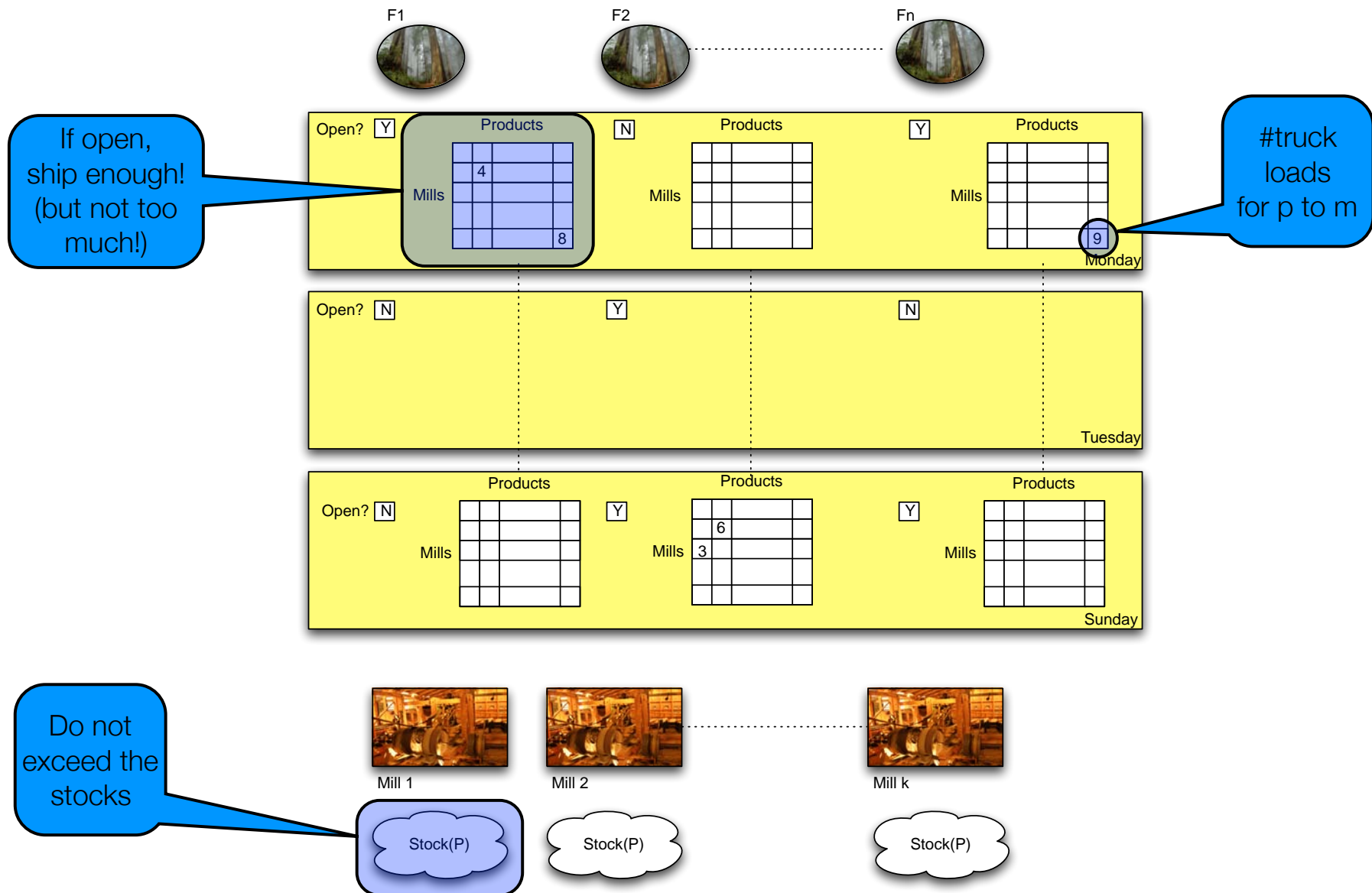
$$\min \sum_{j \in J} \sum_{f \in F} C^* \text{open}_f^j + \sum_{j \in J} \sum_{f \in F} \sum_{w \in W} \sum_{p \in P} c_{fw} \text{trip}_{fwp}^j$$

subject to

$$\begin{aligned} \text{stock}_{wp}^j &\leq \text{MaxStock} && \forall w \in W, p \in P, j \in J \\ \text{stock}_{wp}^{j-1} + \sum_{f \in F} \text{trip}_{fwp}^j &= D_{wp}^j + \text{stock}_{wp}^j && \forall w \in W, p \in P, j \in J \\ \sum_{w \in W} \sum_{p \in P} \text{trip}_{fwp}^j &\leq U \cdot \text{open}_f^j && \forall f \in F, j \in J \\ \sum_{w \in W} \sum_{p \in P} \text{trip}_{fwp}^j &\geq L \cdot \text{open}_f^j && \forall f \in F, j \in J \end{aligned}$$



# Weekly Scheduling ... in Picture!





## Second Phase

---

- Once a weekly schedule is established....
- Must determine the actual *daily* scheduling of trucks.
  - Enforce the low-level constraints
  - But all the days are now decoupled
- Solving the problem ?
  - Portfolio with
    - CBLS [scheduling]
    - CP [scheduling]
  - Exchange bounds & solutions to seed the rounds.



# Model Skeleton

```
minimize<m>
  sum(r in R)  costEmpty    * D[M[prec[r],F[r]]] +
  sum(r in R)  costWait     * (WForest[r] + WMill[r]) +
  sum(f in F)  costOpen     * (Open[f].end - Open[f].start)

subject to {
  forall(r in R) pickup[r].requires(loader[F[r]]);
  forall(r in R) delivery[r].requires(unloader[Mill[r]]);
  forall(f in F) m.post(Open[f].start == min(r in R : F[r]==f) pickup[r].start);
  forall(f in F) m.post(Open[f].end   == max(r in R : F[r]==f) pickup[r].end);
  forall(a in I,b in O)
    m.post(prec[b]==a => delivery[a].end + D[Mill[a],Forest[b] <= pickup[b].start);
  forall(a in R,b in R)
    m.post(prec[b]==a => WForest[b] == pickup[b].start - (delivery[a].end + D[Mill[a],Forest[b]]));

  forall(r in R) m.post(pickup[r].end + D[Mill[r],Forest[r]] <= delivery[r].start);
  forall(r in R) m.post(WMill[r] == delivery[r].start - (pickup[r].s + D[Forest[r],Mill[r]]));

  m.post(alldifferent(prec));
  forall(r in R) m.post(prec[r] != r);
  forall(r in R) delivery[r].precedes(Horizon);
}
```



# Model

---

- **Scheduling based**

- One “group” of activities per truck trip
  - Transition Times for traveling
  - Loading activity for pickup of wood at forest
  - Unloading activity for delivery of wood at mill
  - Loading / Unloading use unary resources (the loaders)
- Derive
  - waiting time for trucks at sites
  - how long each forest is “open”
  - cost of empty truck trips (from mills back to forest)
- Minimize
  - Sum of empty trips, waiting cost and forest open cost.



# Express this model twice.

---

- **With CP**

- Search based on classic scheduling searches (e.g., setTimes)

- **With CBLS**

- Search based on multiple TSP-like neighborhoods (e.g., 2-opt, cross-exchange, insertions, deletions)

- **Hybrid is a portfolio**

- Incumbent bound of CBLS used by CP model
- Solution of CP kept to diversify CBLS
- Stop CP based on a limit (e.g., a time Limit)
- Switch to CP when CBLS improves the solution again.
- Completely stop when time budget exhausted.



# Application

---

Jobshop Scheduling



# Hybrid Scheduling [Nowicki & Smutnicki, 96]

---

- Algorithm for Jobshop

- Core

- Purpose

- Show *non-standard* search strategy

- Example Tabu search

- Intensification
  - Elite solutions
  - Limited backtracking

Partial Search  
Non-chronological  
Node Discarding  
*Hard to implement!*



# Jobshop Scheduling

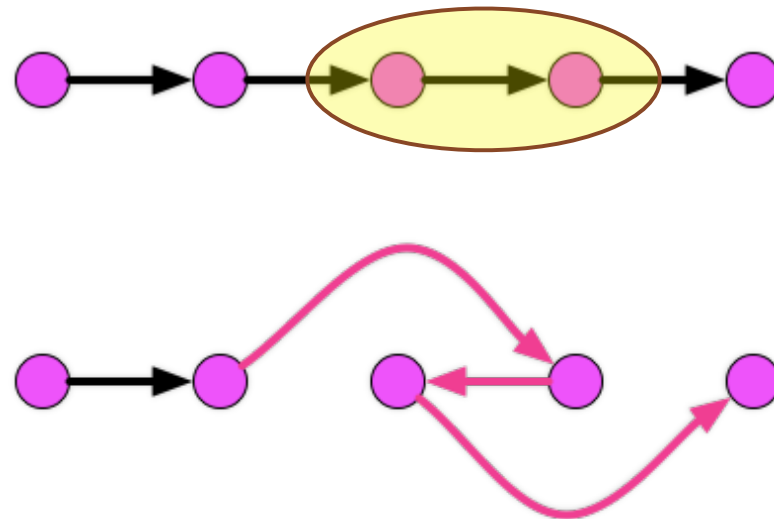
---

- Local Search for jobshop scheduling
  - high-quality solutions quickly
  - choosing machine sequences
- Dell'Amico & Trubian, 1993
  - fast
  - complex neighborhood (RNA + NB)
  - 5,000 lines of C++



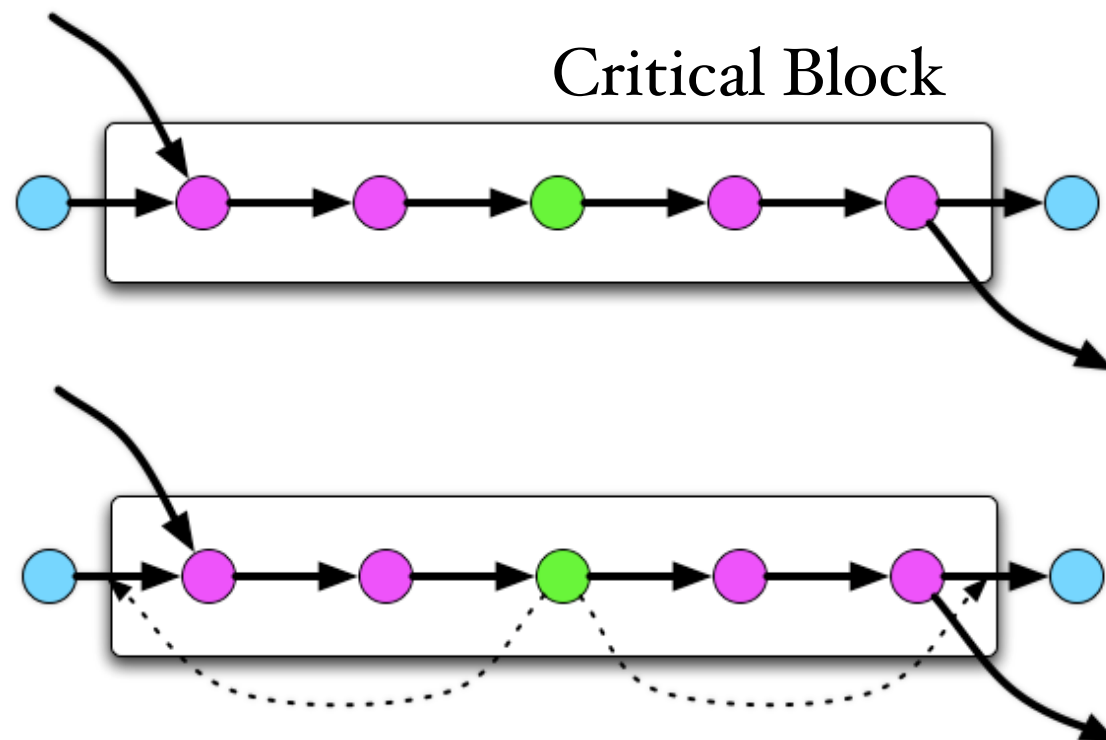
# Neighborhood NA

- Swapping vertices on a critical path



# Neighborhood NB

- Moving tasks in before or after a critical block





# Neighborhood Exploration

```
void exploreNeighborhood(NeighborSelector N){  
    set{int} Criticals = setof(v in Activities)  
                           obj.isStronglyCritical(act[v]);  
  
    exploreNA(N,Criticals);  
    exploreNB(N,Criticals);  
}  
  
void exploreNA(NeighborSelector N,set{Activity} critical) {  
    forall(v in critical) {  
        int delta = obj.moveBackwardDelta(v);  
        if (acceptNA(v,delta))  
            neighbor(delta,N)  
                v.moveBackward();  
    }  
}
```

Property

Differential

Local Move



# Neighborhood Exploration

```
void exploreNB(NeighborSelector N, set{Activity} critical) {  
    forall(v in critical) {  
        int lm = obj.getLeftMostCriticalShift(v);  
        while (lm > 1) {  
            int delta = obj.moveBackwardDelta(v, lm);  
            if (acceptNB(v, lm, delta)) {  
                neighbor(delta, N)  
                v.moveBackward(m);  
                break;  
            }  
            lm--;  
        }  
        ... // same on right  
    }  
}
```

Local Move



# Neighbor Selection

---

- **Neighborhood exploration**

- Define what to explore
- Not how to use to the neighborhood

- **Neighbor selection**

- Specify how to use the neighborhood
- Select the best neighbor
- Select a k-best neighbor (semi-greedy algorithm)
- Select all the neighbors (Nowicki & al)



# Neighbor Transition

---

- **Neighborhood exploration**

- What to consider

- **Neighbor selection**

- How to use

- **Neighbor transition**

- How to move

```
void executeMove() {  
    MinNeighborSelector sel();  
    exploreNeighborhood(sel);  
    if (sel.hasMove())  
        call(sel.getMove());  
}
```

neighbor(delta, N)  
sched.moveBackward(v, m);



# Jobshop Scheduling with LS ?

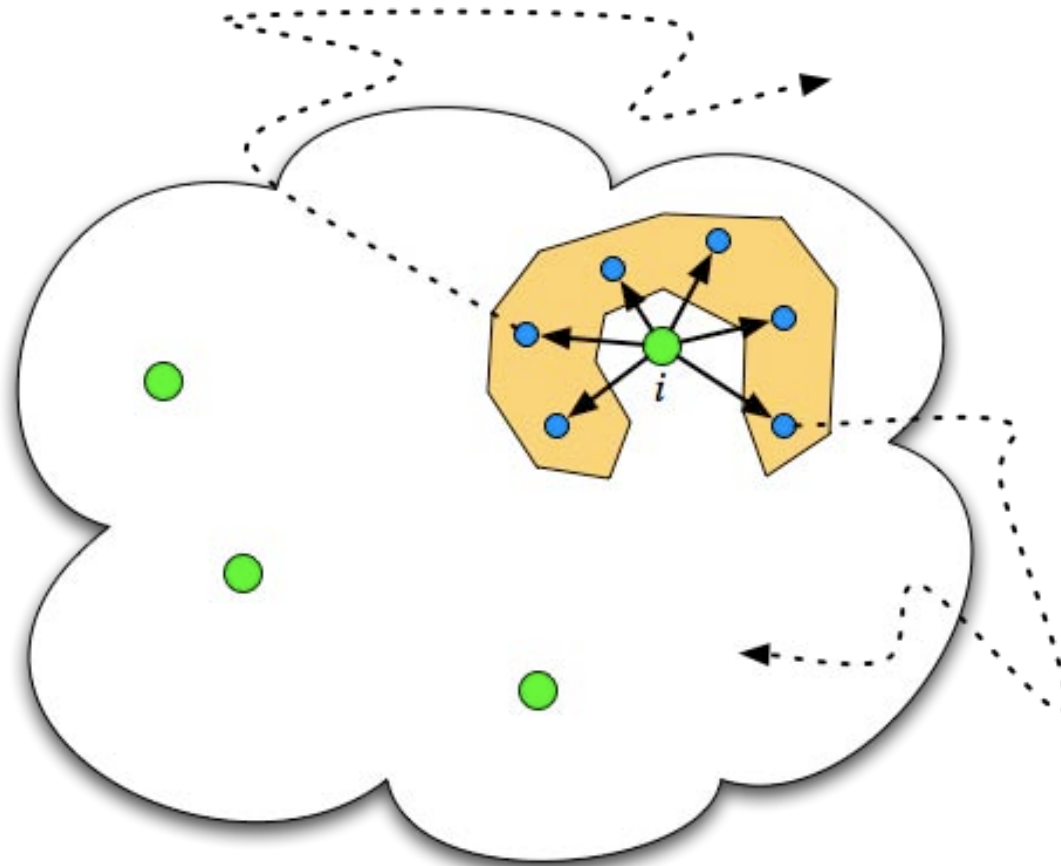
---

- **Ease of use**
  - Avoid the heavy class machinery (200 lines)
- **Readability and separation of concern**
  - Allow to keep the code in one place
  - separate the neighborhood from its use
- **Extensibility**
  - Smooth integration of other neighborhoods
- **Efficiency?**
  - comparable to specific implementations

# Pictorially

- Elite Solutions.

- $|\text{Elite}| \leq k$

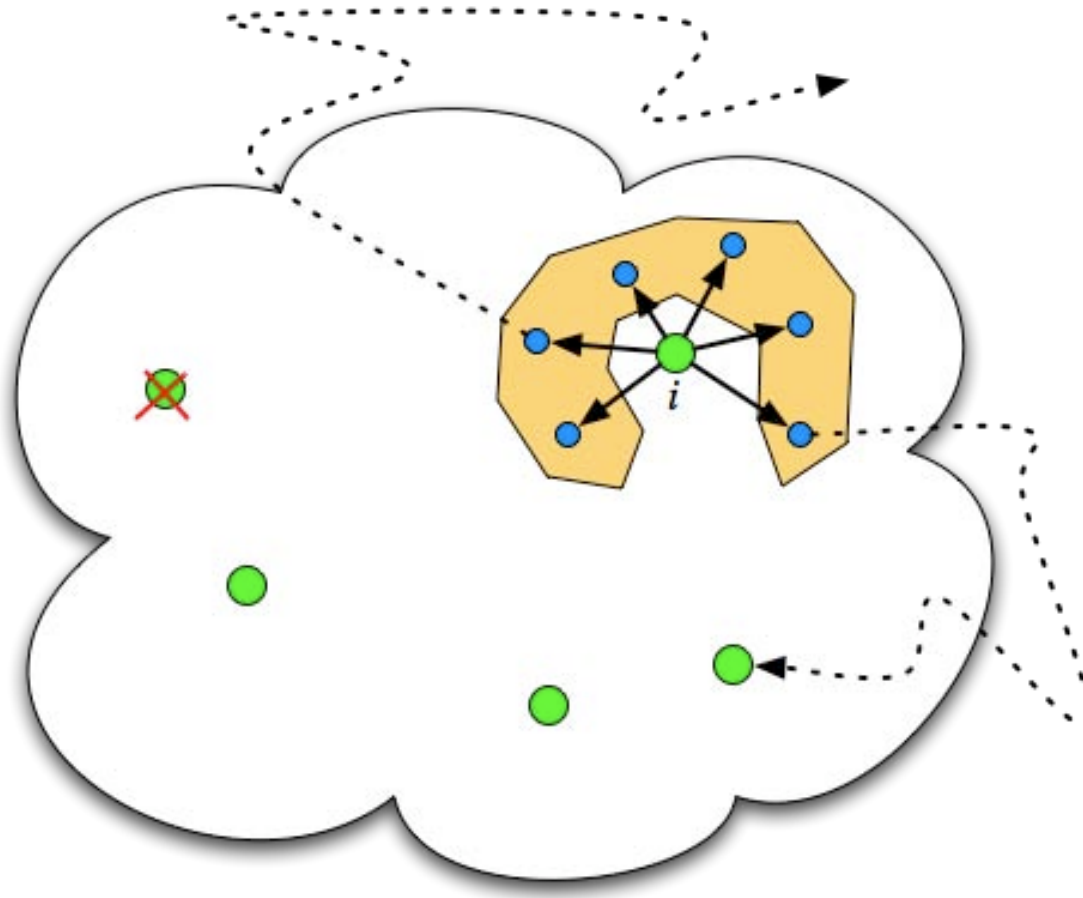




# Pictorially

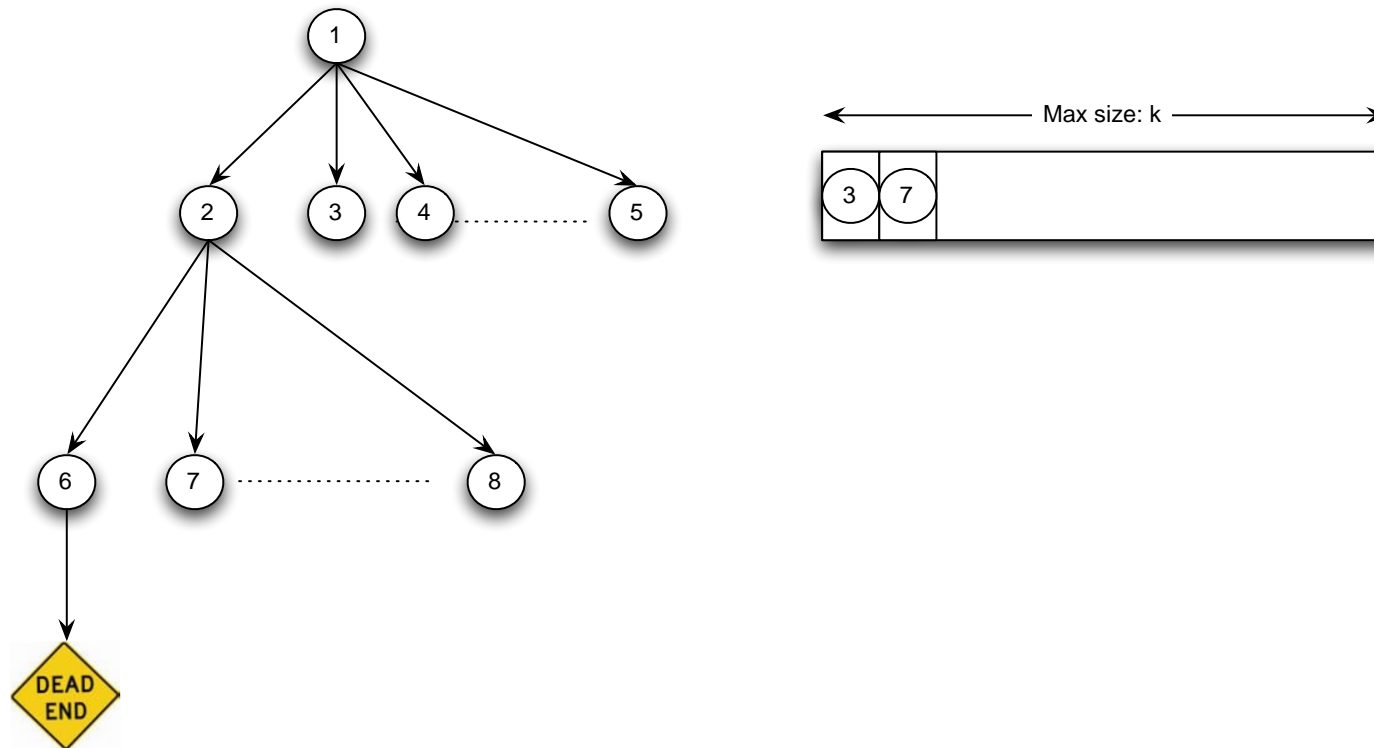
- Elite Solutions.

- $|\text{Elite}| \leq k$

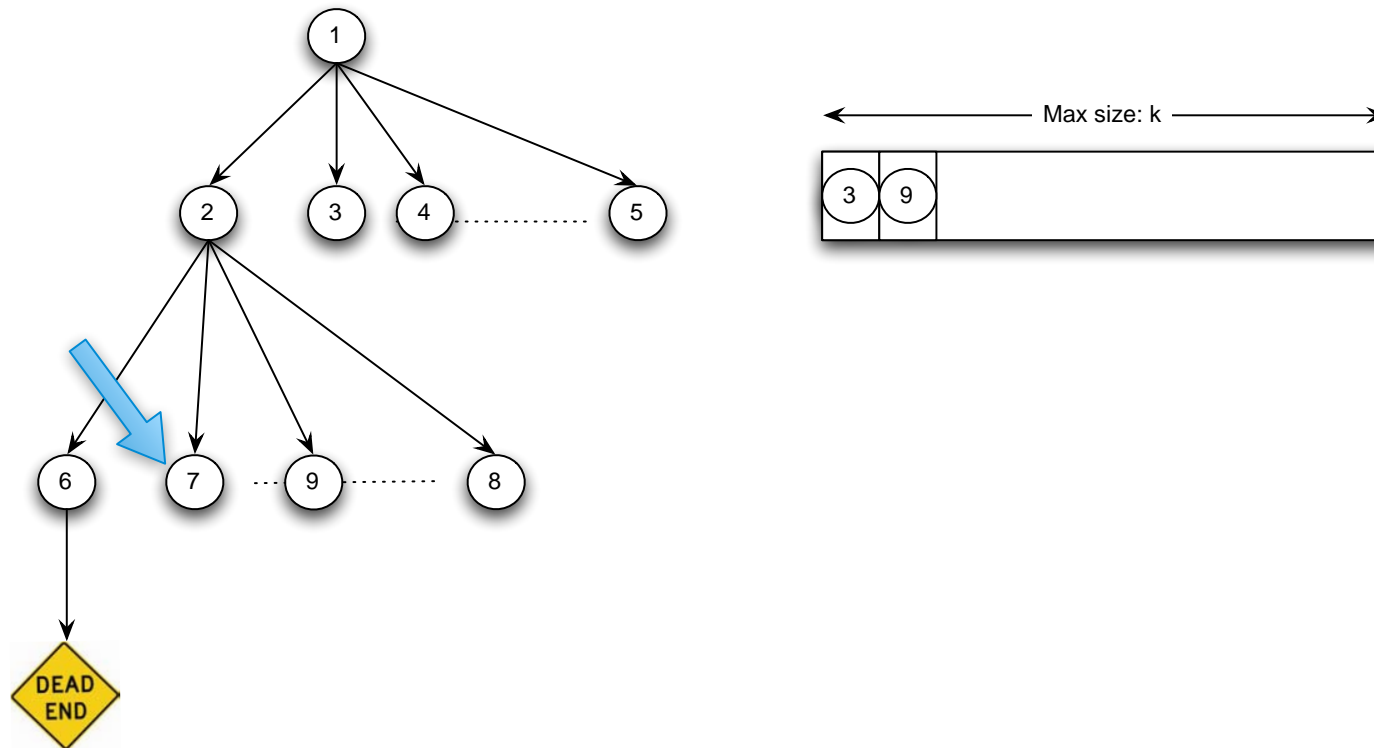




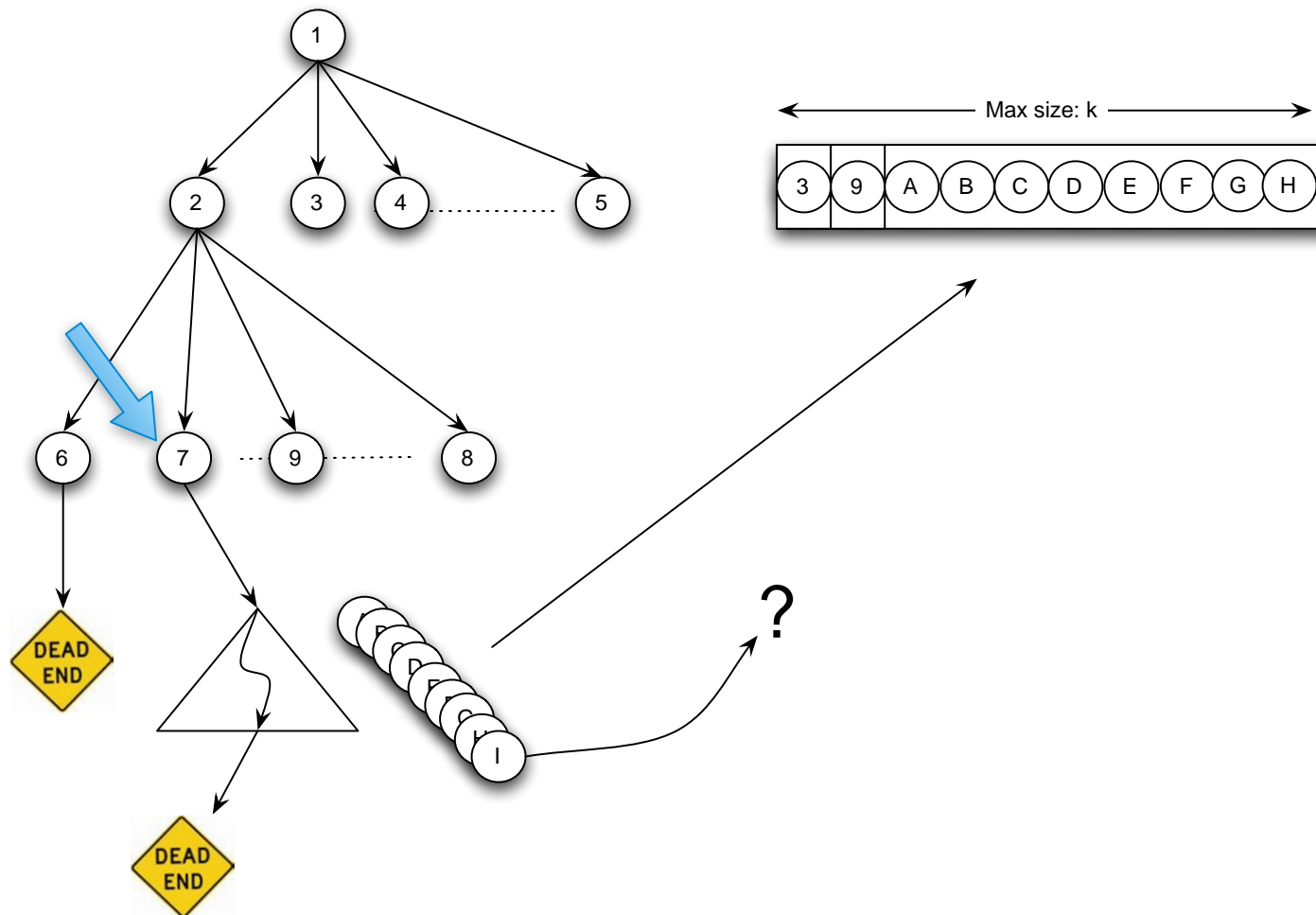
# Illustration



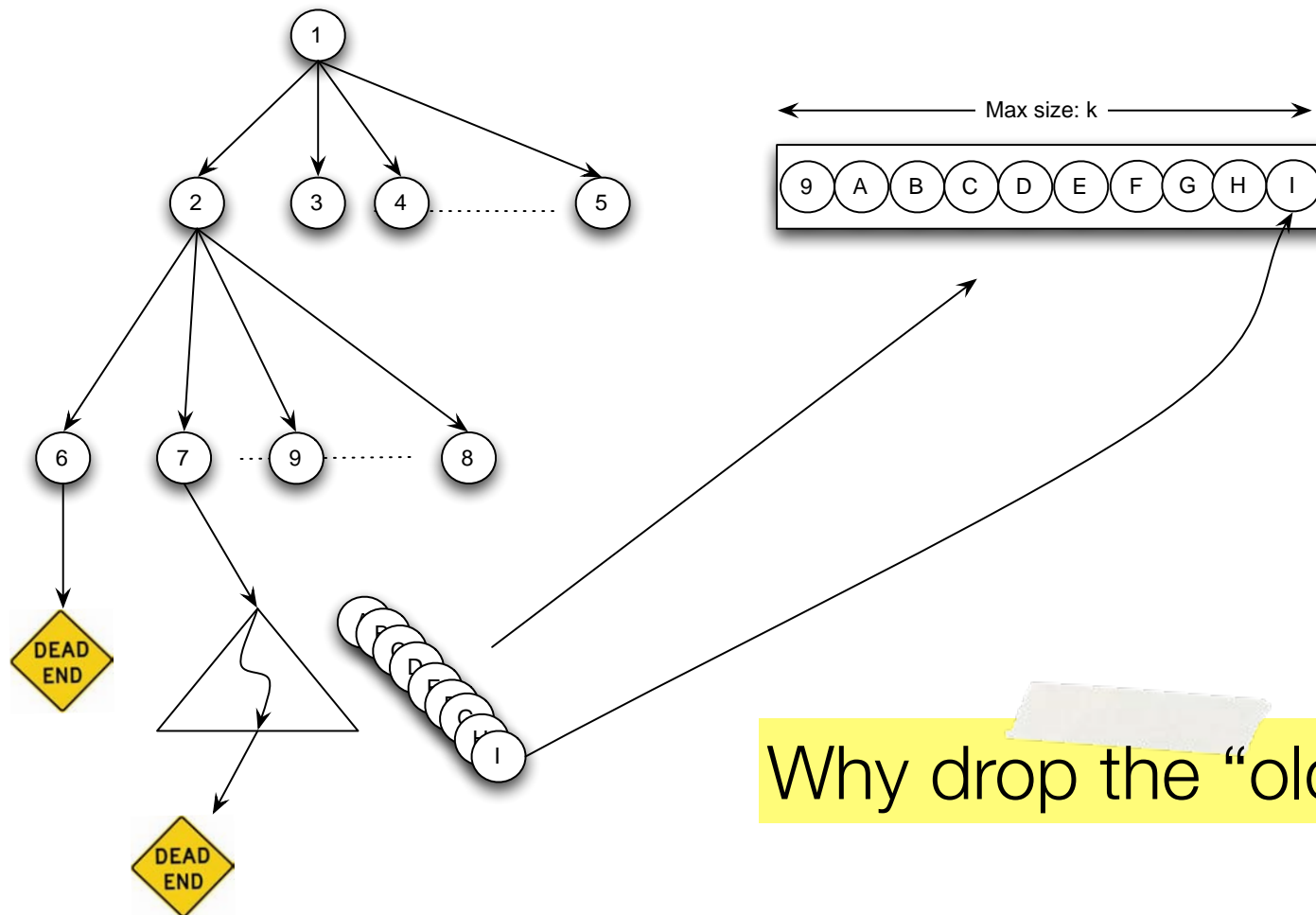
# Illustration



# Illustration

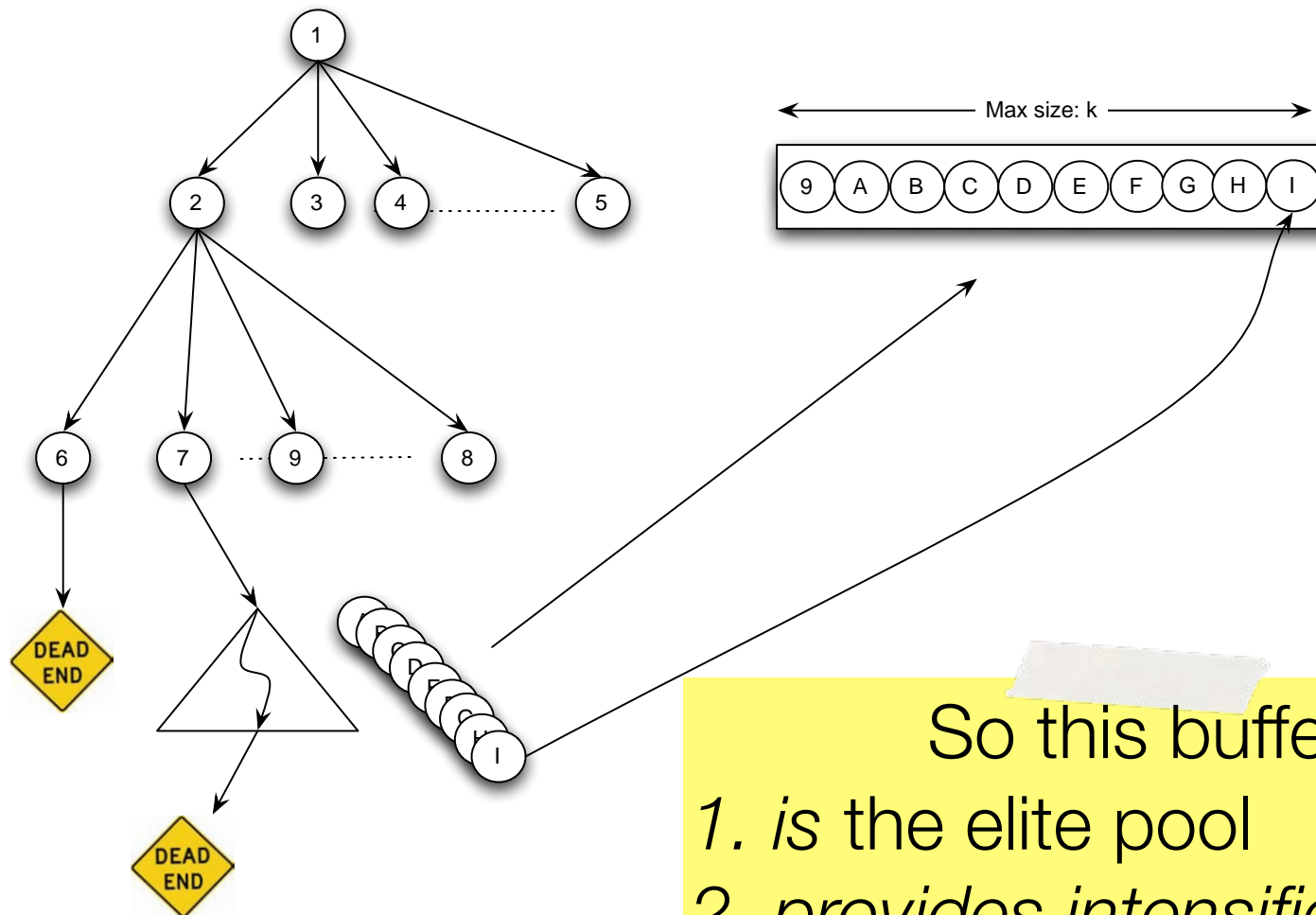


# Illustration



Why drop the “oldest”?

# Illustration





# The Search

```
void Jobshop::search() {
    memento = new Memento(mgr,k);
    MinNeighborSelector N();
    int li = 0;
    exploreall<memento> {
        do {
            int oldBest = bestSoFar;
            if (exploreNeighborhood(N)) {
                call(N.getMove());
                if (oldBest > makespan.value()) {
                    li = 0;bestSoFar = makespan.value();curIter = maxIter;
                    visitAllNeighbors();
                }
            } else memento.exit();
        } while (li++ <= curIter);
    }
}
```

```
void Jobshop::visitAllNeighbors() {
    AllNeighborSelector allN();
    exploreNeighborhood(allN);
    tryall<memento>(i in 1..allN.getSize()-1)
        call(allN.getMove());
}
```



# An Elite Controller

```
class Memento implements SearchController {
    Stack{Continuation}    _c;
    Stack{Solution}        _s;
    int                    _maxSize;
    ...
    void addChoice(Continuation c) {
        if (_c.getSize() == _maxSize) {
            _c.drop();
            _s.drop();
        }
        _c.push(c);
        _s.push(new Solution(_m));
    }
    void fail() {
        if (_c.empty()) exit();
        else {
            _s.pop().restore();
            call(_c.pop());
        }
    }
}
```





## Part III

---

Large Neighborhood Search [Shaw98]





# What is LNS?

---

- LNS was introduced first for Vehicle Routing [Shaw98]
- Premise
  - A wish to leverage LS-style capabilities in CP-style framework
- Rationale?
  - LS is quite effective to get good solutions
  - But LS is hampered by *side-constraints*
  - CP can easily handle side constraints....
  - But tree-search can be misguided by weak heuristics



# LNS Key Idea

---

- Use An Iterative Process
  - Relax
  - Optimize
- Use a CP solver
  - Handles the “side” constraints easily



# Relaxation

---

- Given a (sub-optimal) solution
  - Identify a subset of *related* variables to relax
  - Unbind the identified variables (reset their domain)
- Relatedness is critical
  - Independent variables are likely to be reassigned the same value!
- Relatedness is problem dependent



# Optimization

---

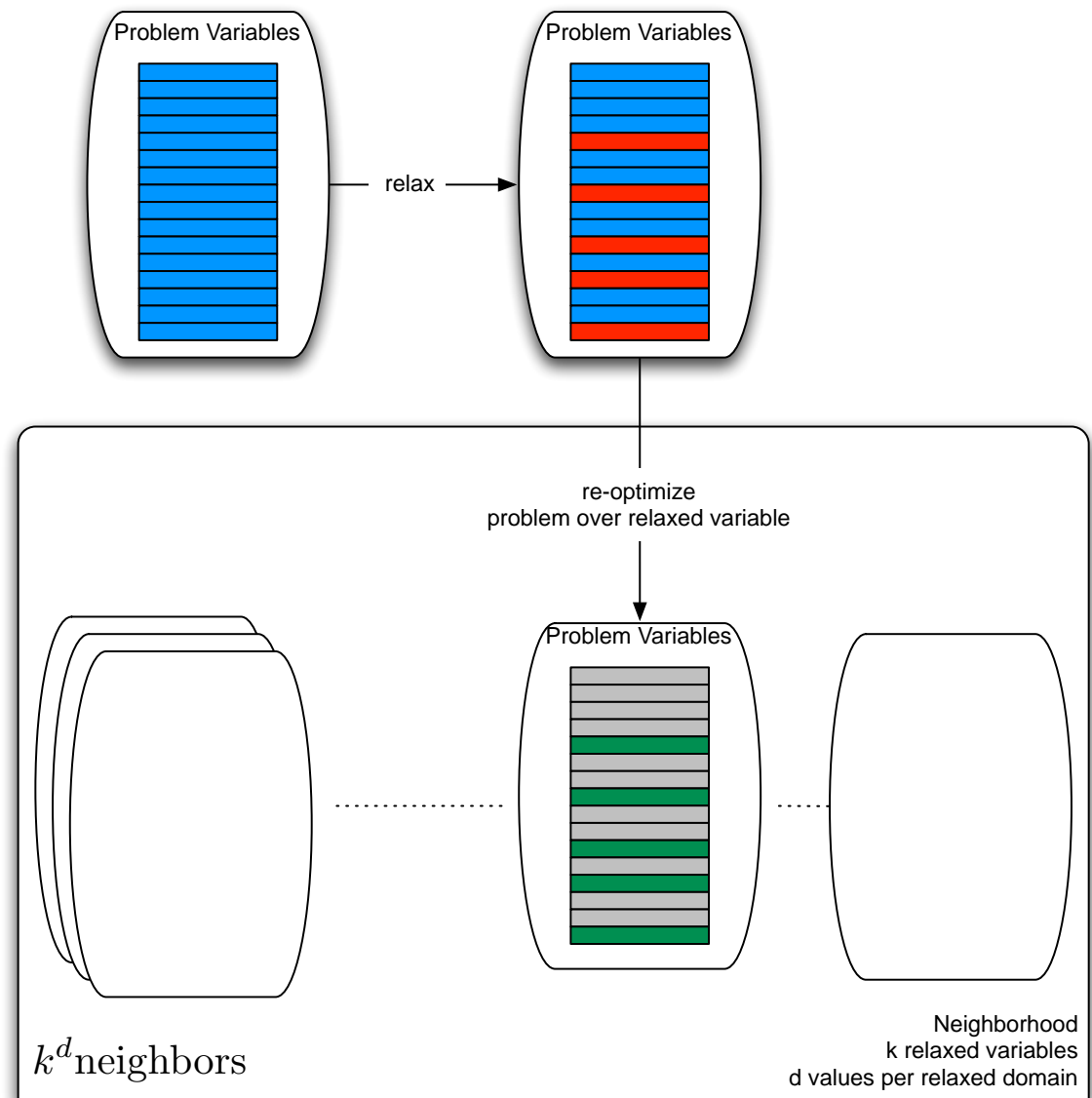
- With the relaxed *partial* solution
  - Re-optimize it
    - Either fully
    - Or partially (e.g., with a bounded strategy like LDS)
- Repeat from the top
  - Stop with an LS-style criterion
    - (time limit, lack of improvement, ....)





# Local Search Semantics

- **LNS is a move operator**
  - (a multi-assignment)
- **Relaxed variables**
  - define the neighborhood
- **CP is used to explore the neighborhood**
  - It *implicitly* examines the neighbors.
  - It *automatically* handles “nasty” side constraints
- **The moves are “big”**





# First Example : Steel Mill Slab Design

---

- **Problem statement**

- Given a set of slab capacities  $\{u_1, u_2, \dots, u_k\}$
- Given a set of  $k$  orders with colors and weight  $\langle c_i, w_i \rangle$

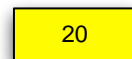
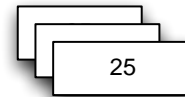
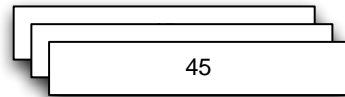
- **Find**

- An assignment of  $n$  orders to  $m$  slabs

- **Which satisfies**

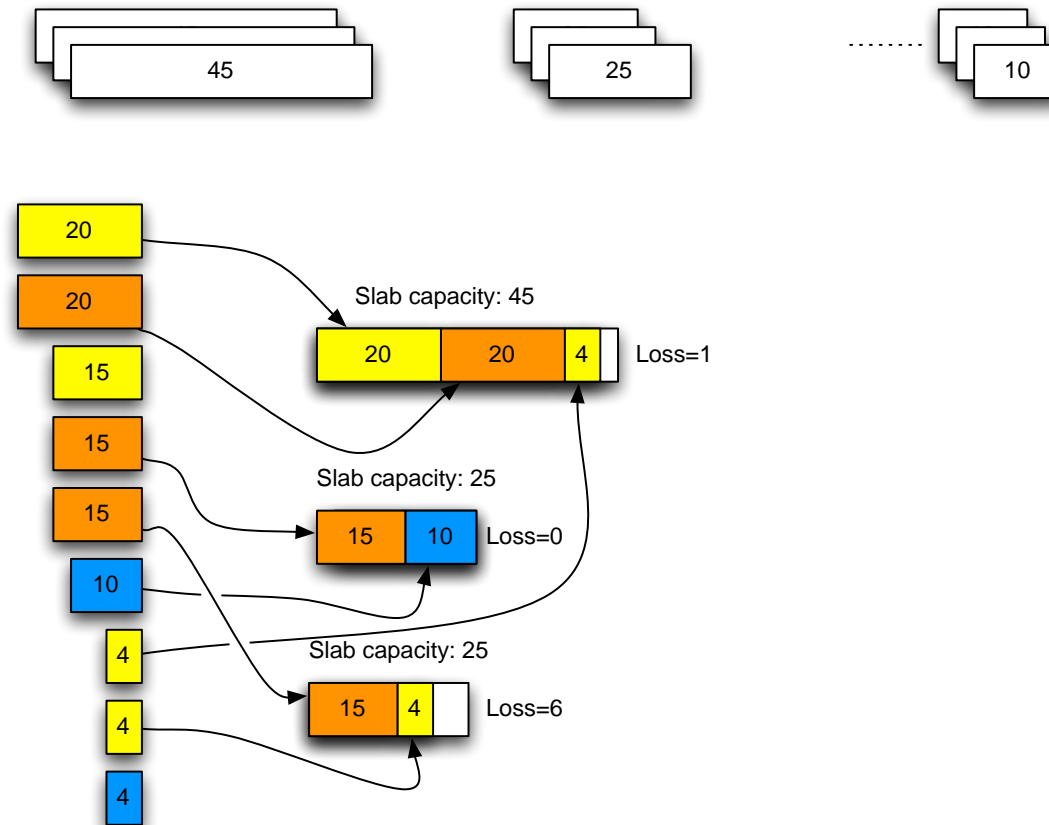
- The total weight on a slab cannot exceed its capacity
- No more than two colors per slab

# Pictorially





# Pictorially





# Pure CP Model

---

- **Decision Variables**

- For every order  $\Rightarrow$  assign it to a slab
- For every slab its length

- **Constraints**

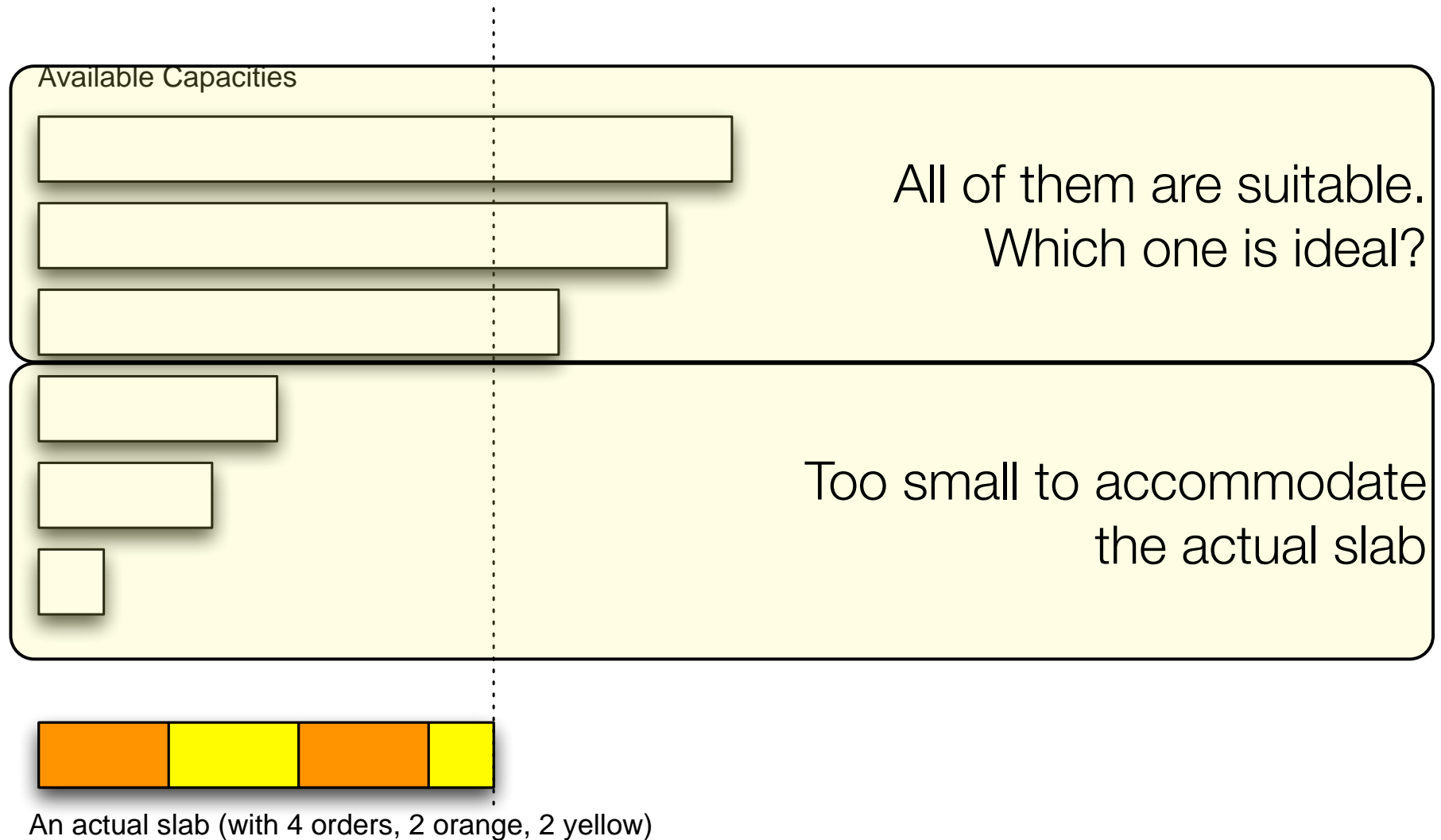
- Compute the needed length of each slab
  - From the orders assigned to it
- Ensure that no more than 2 colors on each slab
- Figure out the loss on a slab (given the available capacities)

- **Minimize**

- The total loss across all slabs



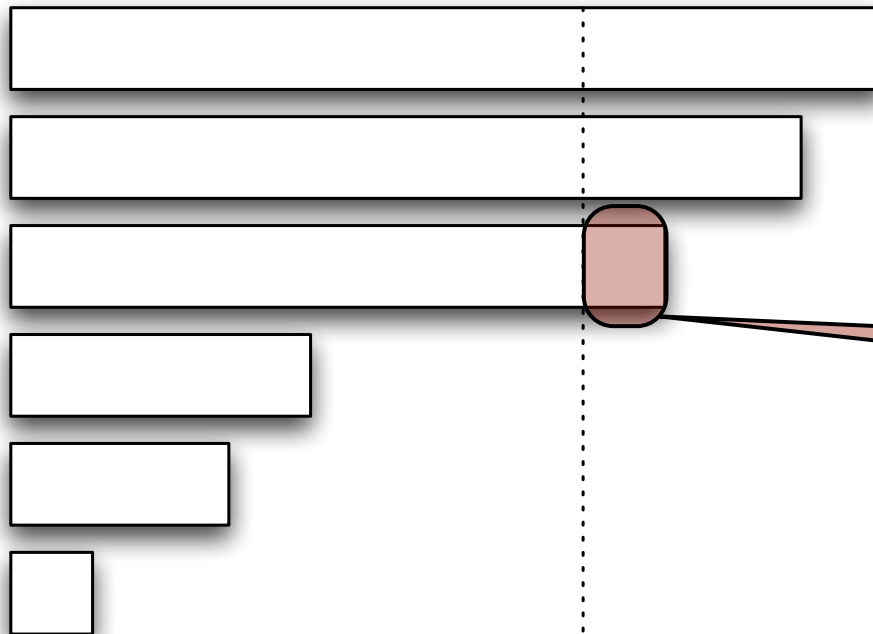
# Determining the losses





# Determining the losses

Available Capacities



This is the actual loss for that target length



An actual slab (with 4 orders, 2 orange, 2 yellow)



# Basic CP Model

```
int loss[c in 0..maxCap] = min(i in Caps: capacities[i] >= c) capacities[i] - c;

Solver<CP> CP();
var<CP>{int} x[Orders](CP,Slabs);
var<CP>{int} l[Slabs](CP,0..maxCap);
var<CP>{int} obj(CP,0..nbSlabs*maxCap);

UniformDistribution dist(1..100);

minimize<CP> obj
subject to {
    CP.post(obj == sum(s in Slabs) loss[l[s]]);
    CP.post(multiknapsack(x,weight,l));
    forall(s in Slabs)
        CP.post(sum(c in Colors) (or(o in colorOrders[c]) (x[o] == s)) <= 2);
} using {
    forall(o in Orders: !x[o].bound()) by (x[o].getSize(),-weight[o]) {
        tryall<CP>(s in Slabs)
            CP.label(x[o],s);
        onFailure
            CP.diff(x[o],s);
    }
    cout << x << endl;
    cout << "Solution value= " << obj.getMin() << endl;
    cout << "#choices = " << CP.getNChoice() << endl;
}
```

# Adding LNS

---

- First get a solution (somehow)
  - For instance, do a limited branch & bound tree search
- Second, restart the search
  - When restarting....
    - Partition the variables into “relax” vs. “freeze”
    - Fix the frozen variables to their value in the solution
    - Leave the “relaxed” variables alone
    - Carry on with the normal search!





# Adding LNS II

---

- How to partition into “relax” vs. “freeze” ?
- That is problem dependent....
- However, there are 2 effective generic ideas
  - relax “related” variables together
  - relax randomly
- **Example**
  - Consider the Steel Mill problem
  - What does related mean?

## Adding LNS III

---

- How do we get the initial solution?
- We already have everything!
  - Simply apply the CP search with *everyone relaxing ;-)*
  - But...
  - Limit the search (with time/failures/....)







# Adding LNS IV (Random Style)

```
int loss[c in 0..maxCap] = min(i in Caps: capacities[i] >= c) capacities[i] - c;

Solver<CP> CP();
var<CP>{int} x[Orders](CP,Slabs);
var<CP>{int} l[Slabs](CP,0..maxCap);
var<CP>{int} obj(CP,0..nbSlabs*maxCap);
UniformDistribution dist(1..100);

CP.lnsOnFailure(3000); // Budget of 3000 backtracks for neighborhood exploration
minimize<CP> obj
subject to {
    ... // as before
} using {
    ... // as before
}
onRestart {
    Solution s = CP.getSolution();
    if (s != null) {
        tryall<CP>(i in 1..10) { // try up to 10 different relaxations
            forall(o in Orders) // scan the orders
                if (dist.get() <= 90-10*i) // flip a biased coin to pick "relax/frozen"
                    CP.post(x[o] == x[o].getSnapshot(s)); // freeze
        }
        CP.setPrimalBound(s.getObjectiveValue()); // impose the known upper bound
    }
}
```



## Adding LNS IV (“Related” Style)

```
int loss[c in 0..maxCap] = min(i in Caps: capacities[i] >= c) capacities[i] - c;

Solver<CP> CP();
var<CP>{int} x[Orders](CP,Slabs);
var<CP>{int} l[Slabs](CP,0..maxCap);
var<CP>{int} obj(CP,0..nbSlabs*maxCap);
UniformDistribution dist(1..100);

CP.lnsOnFailure(3000);    // Budget of 3000 backtracks for neighborhood exploration
minimize<CP> obj
subject to {
    ... // as before
} using {
    ... // as before
}
onRestart {
    Solution s = CP.getSolution();
    if (s != null) {
        set{int} onSlab[i in Slabs] = setof(o in Orders) (x[o].getSnapshot(s)==i);
        tryall<CP>(i in 1..10) {
            forall(j in Slabs)
                if (dist.get() <= 100 - 10*i)
                    forall(o in onSlab[j])
                        CP.post(x[o] == x[o].getSnapshot(s));
        }
        CP.setPrimalBound(s.getObjectiveValue());    // impose the known upper bound
    }
}
```



# Application

---

Asymmetric TSP with Time Windows



# Overview

---

- **Motivation**

- “Simple” TSP
  - Before looking at a VRP
- Look at it thoroughly
- LNS is competitive with other state-of-the-art hybrids (CP/IP)  
[Focacci/Lodi/Milano,2002]



# Problem

---

- Typical TSP....

- Given  $n$  cities with asymmetric travel distances
- Given that each city has a time window  $[s,e]$  for its visit
- Given that each visit takes a certain amount of *service time*

- Find...

- A tour that
  - visits each city exactly once
  - satisfies the time windows and service time
  - minimizes the distance travelled.



# First Simple CP Model

---

$$\min \sum_{i \in V} Cost_i$$

subject to

$$\left\{ \begin{array}{l} \text{alldifferent}([next_0, \dots, next_{n+1}]) \\ \text{nocycle}([next_0, \dots, next_{n+1}]) \\ next_i = j \Leftrightarrow prev_j = i \quad \forall i, j \in V \\ next_i = j \Rightarrow Cost_j = t_{i,j} \quad \forall i, j \in V \\ next_i = j \Rightarrow start_i + d_i + Cost_i \leq start_j \quad \forall i, j \in V \\ a_i \leq start_i \leq b_i \quad \forall i \in V \setminus \{0, n+1\} \end{array} \right.$$



# Implementation can be improved!

---

- **Map the model to a Scheduling formulation**
  - The visit of cities are activities
  - These activities have duration (the service time)
  - The activities all require a unary resource (the vehicle!)
  - The travel times are *transition Costs*
  - The time windows remain “plain” inequalities on the starting times



# COMET Implementation

```
range Activities = 1..nbCities-2;
int horizon = 2*we[destination];
range Horizon = 0..horizon;
int transitionTimes[i in Activities,j in Activities] = cost[i,j];

Scheduler<CP> cp(0,horizon);
Activity<CP> act[i in Activities](cp,service[i],i);
UnaryResource<CP> vehicle(cp,transitionTimes);
var<CP>{int} tt(cp,0..horizon);
forall(i in Activities)
    act[i].requires(vehicle);
vehicle.close();

minimize<cp> tt
subject to {
    forall(i in Activities) {
        cp.post(act[i].start() >= a[i]);
        cp.post(act[i].start() <= b[i]);
    }
    vehicle.postSumTransitionTimes(tt);
} using {
    vehicle.labelSucc();
    forall(i in Activities) label(act[i].start());
    printThePath(...);
}
```

Setup Activities  
& service time

Setup unary  
resource

Setup Time  
windows

Branch on  
precedences





# Introducing LNS

---

- Questions

- What to relax ?
- How to relax?



# Relaxation Step

---

- **Two Options**

- Take 1:

- Flip a biased coin for each visit to decide whether to relax it

- Take 2:

- Pick a location in the tour (uniformly at random)
    - Relax a segment of  $k$  visits starting from that location



# Relaxation Code

```
range Activities = 1..nbCities-2;
int horizon = 2*we[destination];
range Horizon = 0..horizon;
int transitionTimes[i in Activities,j in Activities] = cost
[i,j];
```

```
Scheduler<CP> cp(0,horizon);
Activity<CP> act[i in Activities](cp,service[i],i);
UnaryResource<CP> vehicle(cp,transitionTimes);
var<CP>{int} tt(cp,0..horizon);
forall(i in Activities)
    act[i].requires(vehicle);
vehicle.close();
```

```
int prInit = 100 - 2500/nbCities;
UniformDistribution LNSPr(0..1);
UniformDistribution Pr(prInit-5..prInit+5);
UniformDistribution d(1..100);
cp.timeLimit(60);
cp.lnsOnFailure(max(500,5*nbCities));
```

```
minimize<cp> tt
subject to {
    forall(i in Activities) {
        cp.post(act[i].start() >= a[i]);
        cp.post(act[i].start() <= b[i]);
    }
    vehicle.postSumTransitionTimes(tt);
} using {
    vehicle.labelSucc();
    forall(i in Activities) label(act[i].start());
    printThePath(...);
}
```

Flip a coin

```
onRestart {
    var<CP>{int} succ[i in Activities] = vehicle.getSucc(act[i]);
    Solution s = cp.getSolution();
    range Origins = vehicle.getOrigins();
    origin = vehicle.getOrigin();
    int destination = vehicle.getDestination();
    Activity<CP> sink = vehicle.getSink();
    if (s != null) {
        with atomic(cp) {
```

```
            if (LNSPr.get() == 0) {
                dict{Activity<CP> -> bool} fixed();
                select(i in Activities) {
```

```
                    Activity<CP> curr = act[i];
                    int nb = 0;
                    while (curr != sink && nb++ < 20) {
                        fixed{curr} = true;
                        curr = vehicle.getSuccessor(curr,s);
                    }
                    set{Activity<CP>} f = fixed.getKeys();
                    forall(a in f)
                        cp.post(succ[a] == succ[a].getSnapshot(s));
                }
            } else {
```

```
                int pr = Pr.get();
                forall(i in Activities)
                    if (d.get() < pr)
                        cp.post(succ[i] == succ[i].getSnapshot(s));
            }
        }
    }
}
```

Relax a Segment

Relax at random

# Demo

---





# Application

---

## VRPTW

A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows [Bent,2004]



# Outline

[Slides from R. Bent & P. Van Hentenryck]

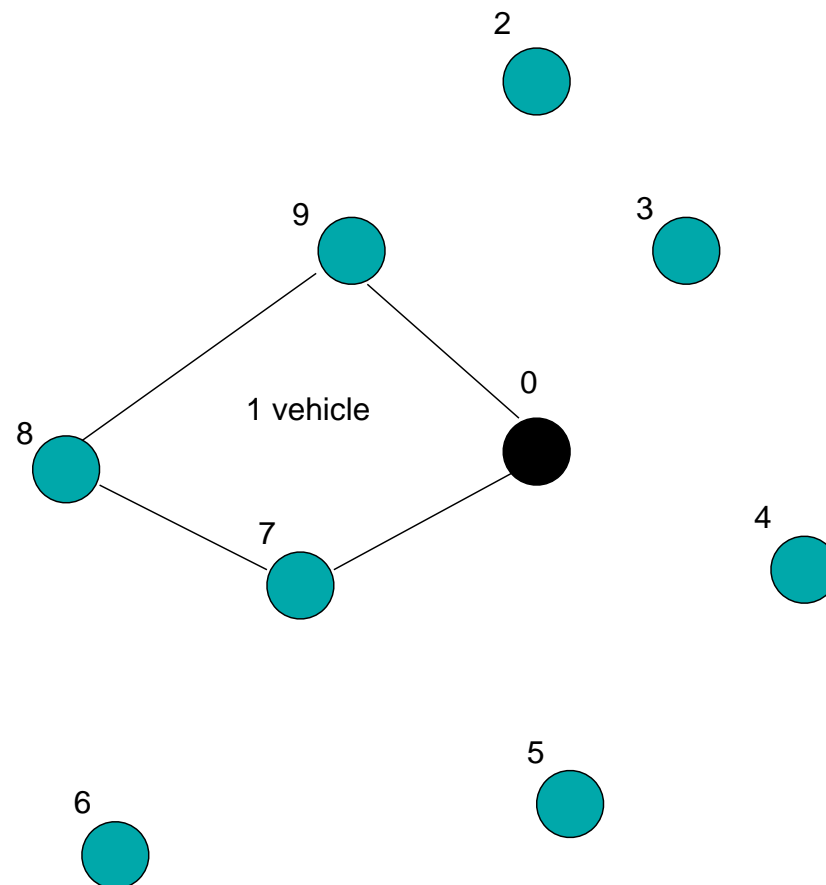
---

- The Vehicle Routing Problem (review)
- Local Search Issues
- Limited Discrepancy Search
- Large Neighborhood Search
- Two Stage Search



# Vehicle Routing

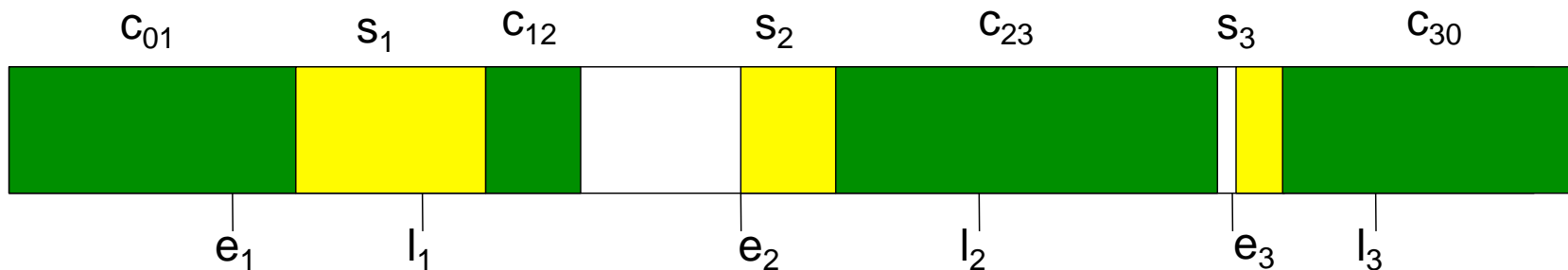
- $n$  customers
- 1 Depot
- $c_{ij}$ : Euclidean travel distance between  $i$  and  $j$
- $m$  identical vehicles
  - Each customer visited exactly once by a vehicle





# Time Window Constraints

- $[e_i, l_i]$  earliest and latest start of service time of customer  $i$
- $s_i$  service time of customer  $i$
- $d_i$  departure time of customer  $i$ 
  - $d_0 = 0$
  - $d_i = \max(d_{i-1} + c_{i-1}, e_i) + s_i$
- $a_i$  actual service time start
- $a_i = \max(d_{i-1} + c_{i-1}, e_i)$
- Constraint satisfied if  $a_i \leq l_i$  ( $0 \leq i \leq n$ )







# Capacity Constraints

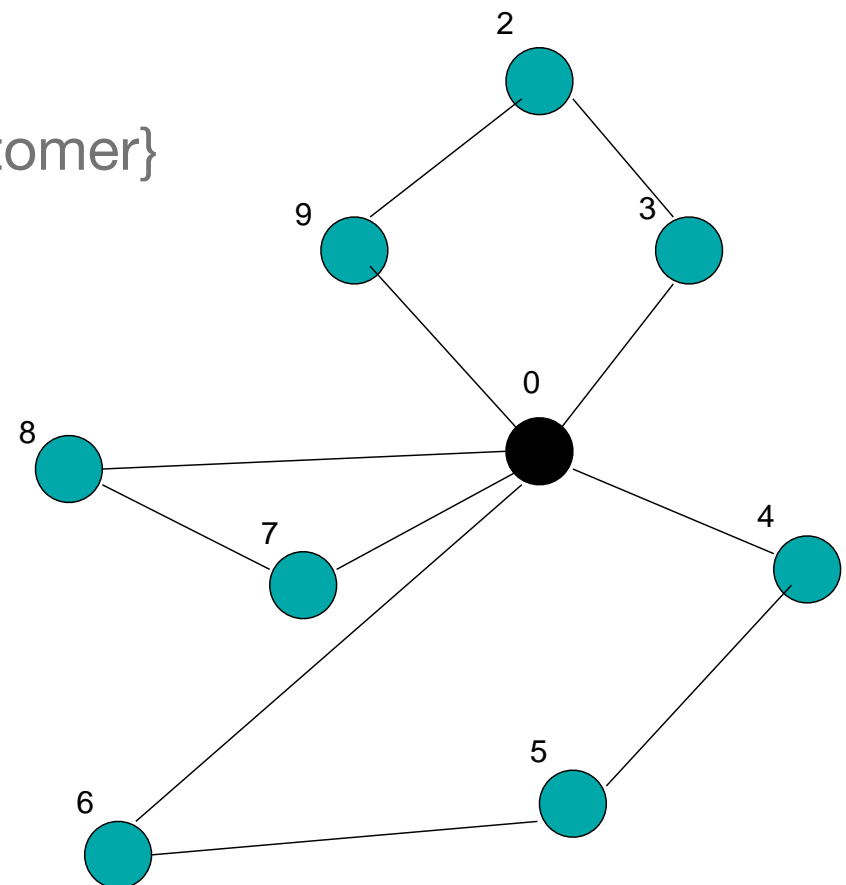
---

- Customer  $i$  has demand  $q_i$
- $q(v)$  denotes demand utilized by vehicle  $v$
- $q(v) = \sum \{ q_i \mid i \text{ served by } v \}$
- Constraint satisfied when  $q(v) \leq Q$



# Objective Function

- Lexicographic
- (1) Minimize vehicles used
  - $|V| = \{v \mid v \text{ serves at least one customer}\}$
- (2) Minimize total travel distance





# Local Search

---

- How good is local search?
- 56 Solomon Benchmarks (100 customers – proposed in 1986)
  - 1995 – 19 of the best known solutions discovered
  - 1997 – 1 “
  - 1998 – 1 “
  - 1999 – 3 “
  - 2000 – 5 “
  - 2001 – 9 “
  - 2002 – 14 “
  - 2003 – 3 “
- These problems are still quite open, new and improved local searches are coming out all the time



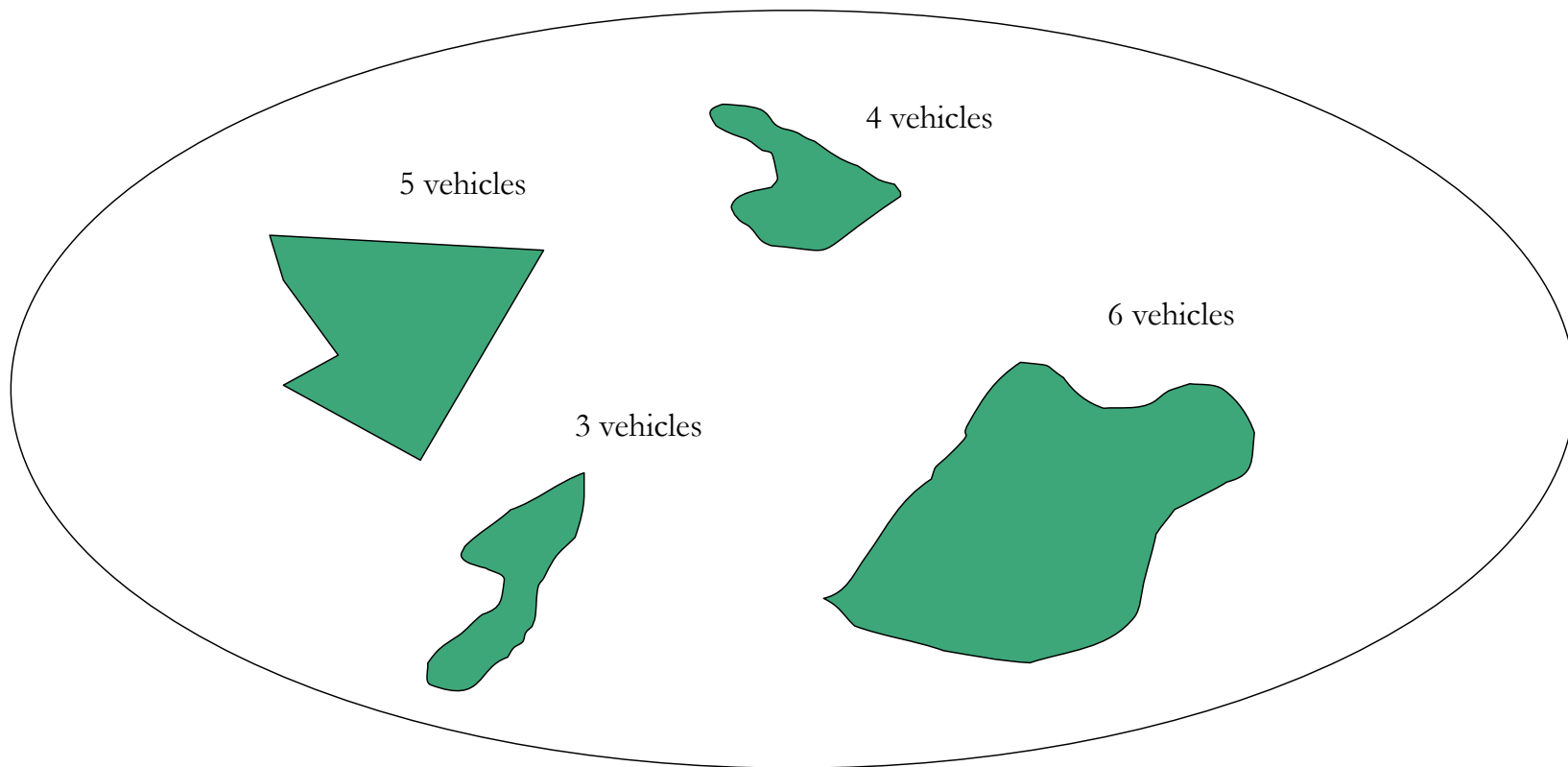
# Local Search

---

- This problem is complicated...
- What aspects might cause problems for local search?

# Local Search

- **Constraints! Did I say constraints?**
  - The feasible space of a neighborhood may be disjoint



Example: exchange neighborhood 213



# How can we cope?

---

- Allow the search to consider infeasible solutions
  - Graph Coloring
  - Generally does not work well on VRP
- Consider larger, more complex neighborhoods
- Combine techniques from Global Search with Local Search



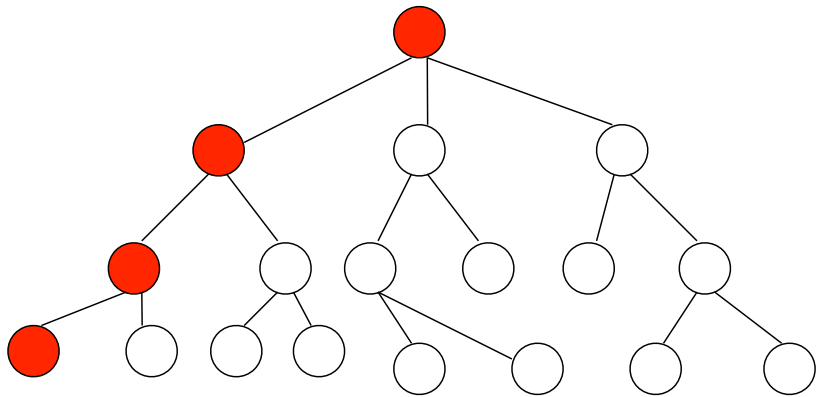
# Limited Discrepancy Search

---

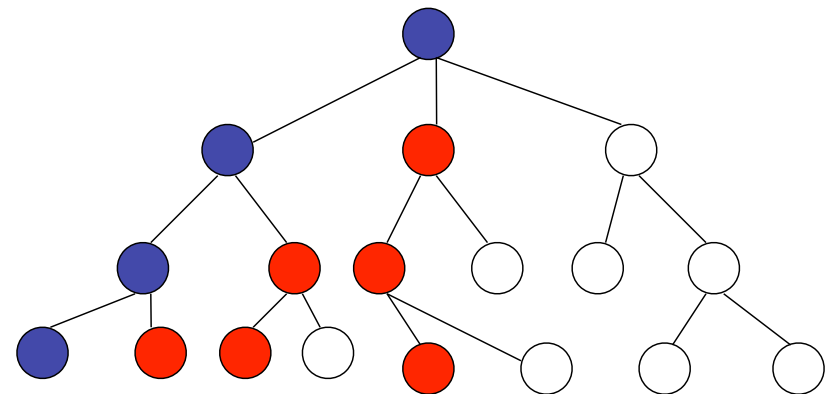
- **Branch and Bound, Constraint Programming**

- Motivation: You have a really good branching heuristic
- Allow up to  $d$  *deviations* of your branching heuristic
- Example: TSP
  - Branch from nearest to furthest neighbor
  - $d = 0$ , Only branch on the nearest neighbor
  - $d = 1$ , Allowed to branch once on the second nearest neighbor on any path in the search tree.
  - etc.

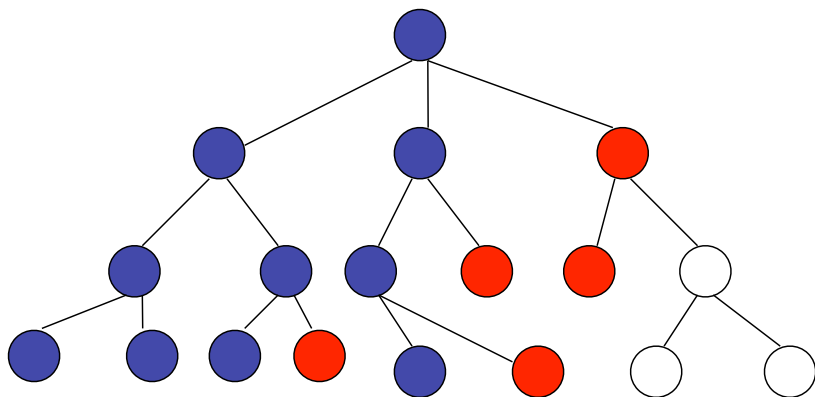
# LDS



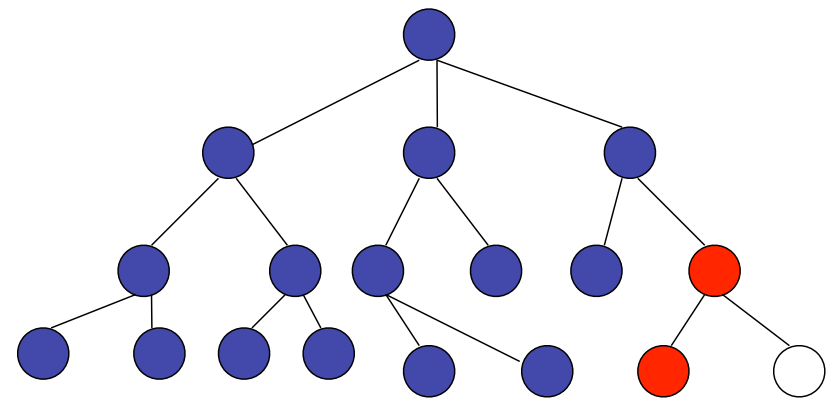
$d = 0$



$d = 1$



$d = 2$



$d = 3$





# Limited Discrepancy Search

---

- $d = 0$ , yields a greedy heuristic (like nearest neighbor)
- $d = n * |D|$ , yields optimal solution
  - $|D|$  = size of largest domain
- Progressively increase  $d$  to get closer and closer to optimal solution
- Complexity for  $d$  discrepancies?
  - Roughly  $O(n^{d+1})$



# LNS: Key ideas

---

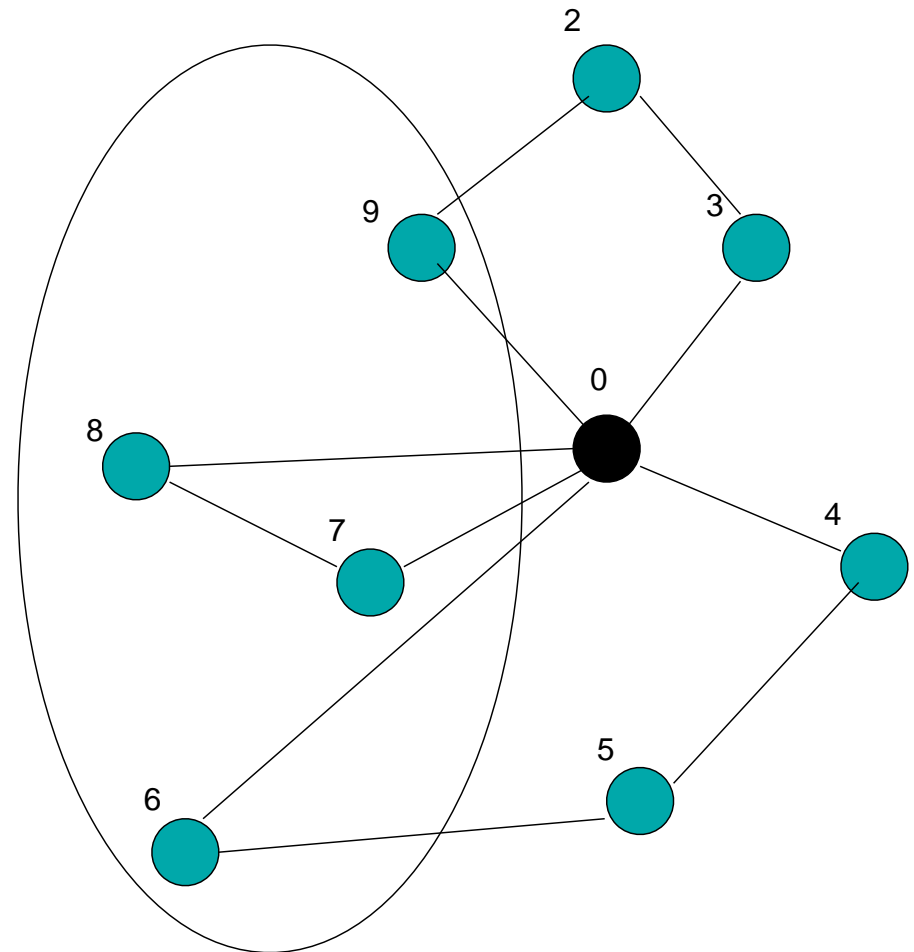
- **Combine Global Search and Local Search**
  - Branch and Bound
  - Constraint Propagation
  - Local Search
- **Generate Large Neighborhoods**
  - Remove subsets of customers
  - Explore reinsertion points

# LNS

- Remove a Set of Customers

- What is the neighborhood?

- How do you explore it?



# LNS

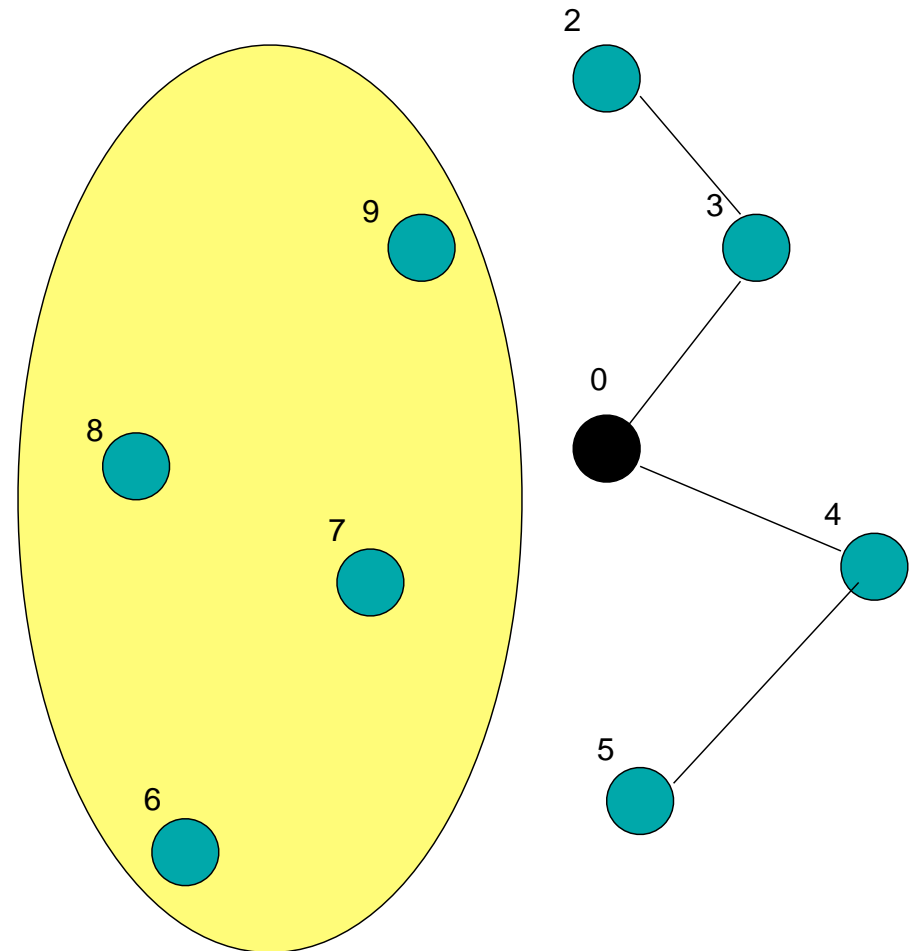
- Remove a Set of Customers

- What is the neighborhood?

Feasible solutions that can be reached by inserting the removed customers

- How do you explore it?

Constraint Programming  
Branch and Bound





# LNS

---

- Variables
  - The removed customers
- Domains
  - Feasible insertion points
- Use CP!
  - Prune the domains based on
    - Precedence
    - Time windows
    - Optimality!



# Branching Heuristics

---

- Choose a customer to re-insert
- Branch from
  - best insertion
  - to worse insertion
- How to select the customer?
  - First Fail
  - Greedy (best insertion)



# Customer Removal

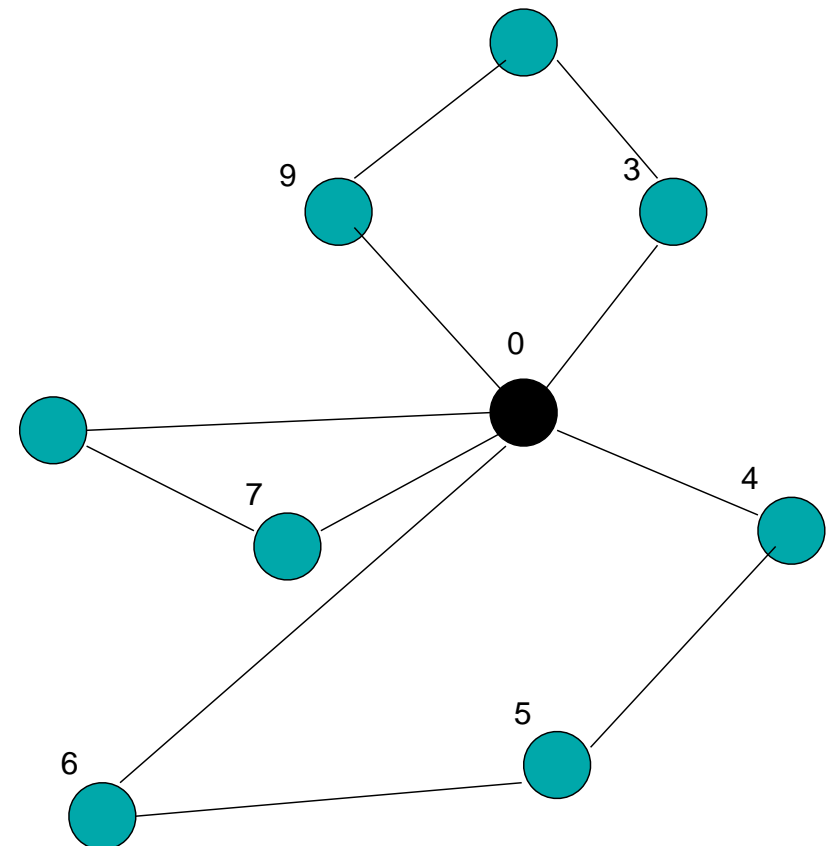
---

- How many should be removed?
- Progressively try removing more and more customers!

```
LNS() {  
  for i = 1 ... MAX_REMOVAL  
    for j = 1 ... MAX_TRIES  
      S = Choose(i);  
      Remove(S);  
      ExploreNeighborhood(S);  
      if (improve)  
        j = 1;  
}
```

# Relatedness

- What is a good set to remove
  - “Related customers”
  - Customers close in proximity
  - Customers on the same vehicle
- Basic idea:
  - choose a random set, weighted to favor sets with the above properties







# Exploration Cost

---

- Q: What is the problem with LNS stated as of now?
  - Say you are removing 30 customers at a time.  
What is the size of this neighborhood?
- A: Exponential neighborhood size. Many solutions in the neighborhood not very good.
  - We can use Limited Discrepancy Search to explore a subset of the neighborhood
  - A tradeoff in time utilization between exploring larger neighborhoods vs. exploring more neighborhoods



# LNS for VRPTW Revised (again!)

---

```
LNS(int d) {  
  for i = 1 ... MAX_REMOVAL  
    for j = 1 ... MAX_TRIES  
      S = Choose(i);  
      Remove(S);  
      ExploreNeighborhoodLDS(S, d);  
      if (improve)  
        j = 1;  
}
```



## More LS Trouble?

---

- What other aspects of this problem might cause problems for local search?

How might the  
lexicographic objective  
function cause problems?



# Lexicographic Function

---

- **Example: Solomon Problem R104**
  - Best solution with 10 vehicles: 981.23
  - Best solution with 9 vehicles: 1007.24
- **The two components of the objective function do not mesh**
  - Reducing travel cost of a solution makes it more difficult to find a solution with fewer vehicles

## Solution:

Two stage local search. Decompose the problem into two parts, one that minimizes vehicles and the other that minimizes travel distance.



# Vehicle Reduction

---

- **Objective: Minimize the number of vehicles used**
  - Remind you of anything?
- **Objective: Minimize the number of colors used**
  - Like graph coloring, the objective function is not a very good guide for local search. It cannot distinguish between two solutions with the same number of vehicles

$$\sum_{v \in m} |v|^2$$

Generally a very good function  
when the objective is to  
minimize the number of sets



# Minimal Delay

---

- This is typically not sufficient in the case of ties.
  - Key idea: We'd like a measure of how difficult it is to reduce the vehicles (color classes) by 1
- Step 1: Find the vehicle,  $v$ , with the fewest customers
- Step 2: For each customer on  $v$ , calculate the minimal delay for placing that customer on another vehicle
- Minimal delay = 0,
  - if the customer can be feasibly placed on another vehicle
- Otherwise...
  - find the insertion point that causes the smallest time window violation. This violation is the minimal delay



# Minimal Delay

---

- This is typically not sufficient in the case of ties.
  - Key idea: We'd like a measure of how difficult it is to reduce the vehicles (color classes) by 1
- Step 1: Find the vehicle,  $v$ , with the fewest customers
- Step 2: For each customer on  $v$ , calculate the minimal delay for placing that customer on another vehicle
- Minimal delay = 0,
  - if the customer can be feasibly placed on another vehicle
- Otherwise...
  - find the insertion point that causes a violation. This violation is the minimal delay

## Drawbacks

- Very inefficient to calculate
- problem specific



# Vehicle Reduction

---

- **Surrogate Objective Function: Lexicographic objective**

- Number of vehicles
- Graph coloring style objective function
- Minimal delay objective function

- **Search**

- Step 1: Choose a move (relocation, exchange, 2-opt, or-opt, cross) and a customer at random.
- Step 2: Generate the neighborhood
- Step 3: Choose a move at random from this neighborhood (weighted towards the good ones)
- Step 4: Accept the moving using simulated annealing probabilities

That's a Winner!





# Two Stages Search

---

- Key Idea: Split the search into two components to focus on each part of the local search.
- Avoids getting stuck in sub-optimal solutions

1. Simulated Annealing

2. Large Neighborhood Search



## Summing up...

---

- This is the tip of the iceberg on local search techniques used here
  - Genetic Algorithms,
  - Ejection Chains,
  - Variable Neighborhood Search,
  - Squeaky Wheels....



# Going Further?

---

- What is really difficult ?
  - Decide what to relax
  - Do it *automatically*



# What we should relax

---

- Common wisdom
  - relax related things.

- But, why?

- Consider a trivial example

$$x \in \{1..10\}$$

$$y \in \{2..100\}$$

$$x == y$$

- When  $x=1 \Rightarrow y=1$
- Does it make sense to relax  $x$  and *not*  $y$ ?



# How to find that out?

---

- **Observations**

- Related variables are in the same constraints
- If x is related to y ...
  - Then when a “trigger” occurs on x
  - y changes as a result of the trigger

- **So....**

- Propagation really exploits the relationships
- Propagation “traces” those dependencies

- **Morale**

- Use propagation to identify cluster of related variables. [Perron,2004]



## First a detour...

---

- Impact Based Search [Refalo,2004]
- Purpose
  - Build “generic” variable and value ordering heuristics
- How ?
  - Learn which branching decisions (assignments) are most effective
- Mean
  - Instrument the propagation to collect some key statistics



# Search Space

---

- **First**

- Estimate the size of the search space

$$\mathcal{S}(P) = \prod_{x \in X} |D(x)|$$



# Branching Effect

---

- Question

- What happens when you branch with  $x == a$  ?

- Answer

- The propagation eliminates inconsistent values
  - The search space therefore *shrinks*

- Extreme cases

- If  $x == a$  has no effects
    - ➔ search space after  $\sim$  search space before
  - If  $x == a$  has the maximal effect
    - ➔ we get a solution! search space after is a singleton.





# Branching Effect

---

- In a nutshell

- Size before decision (at node  $k$  of search tree)  $\mathcal{S}(P_{k-1})$

- Size after decision (at node  $k$  of search tree)  $\mathcal{S}(P_k)$

- Impact of decision  $x=a$ : 
$$I(x = a) = 1 - \frac{\mathcal{S}(P_k)}{\mathcal{S}(P_{k-1})}$$



# But over the entire tree?

---

- **Refalo's observation**

- Impacts do not change a lot from nodes to nodes

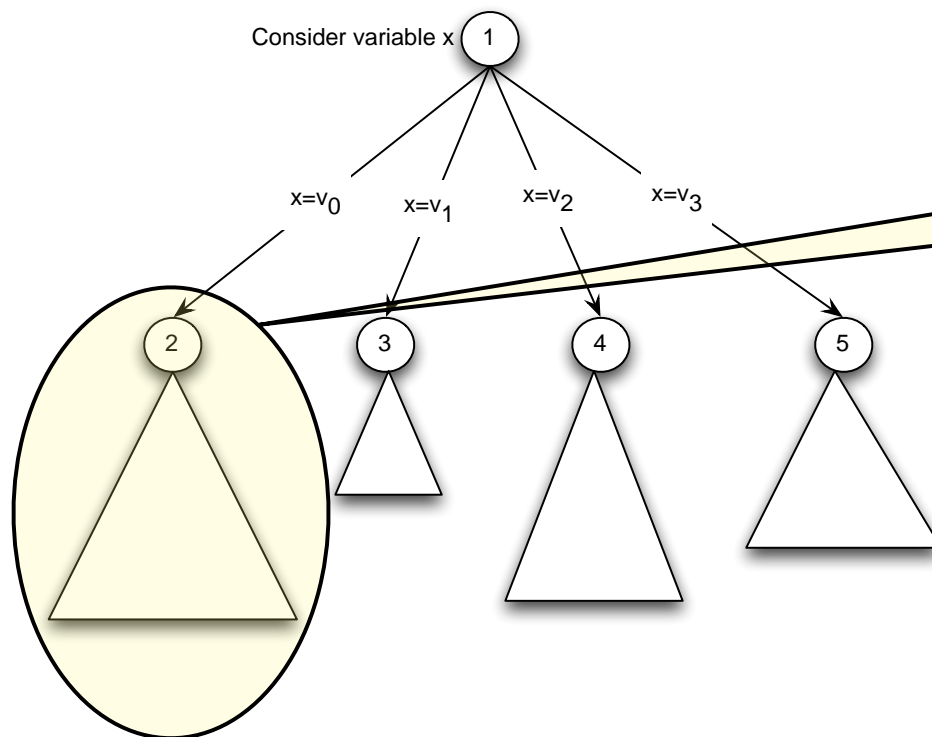
- **Therefore**

- Simply average over all the nodes

$$\bar{I}(x = a) = \frac{\sum_{k \in \mathcal{K}} 1 - \frac{\mathcal{S}(P_k)}{\mathcal{S}(P_{k-1})}}{|\mathcal{K}|}$$

# Variable Impact

- If we have the average impact for each assignment
- We can aggregate the impact over all values



How big is that tree?

$$P_k \cdot (1 - \bar{I}(x == v_0))$$

$$\sum_{v \in D_k(x)} P_k \cdot (1 - \bar{I}(x == v))$$

$$\mathcal{I}(x) = \sum_{a \in D_k(x)} 1 - \bar{I}(x = a)$$



# Variable/Value Ordering from Impacts

---

- Choose the variable that has the highest impact

$$\arg\text{Min}_{x \in X} \mathcal{I}(x) = \arg\text{Max}_{x \in X} - \mathcal{I}(x)$$

- Choose the value most likely to leave us with the most solution

$$\arg\text{Min}_{v \in D_k(x)} \bar{I}(x == v)$$



# Relation to LNS Relatedness of variables

---

- When you try  $x==a$

- There is some propagation

- The domain of some variables shrink

- The  $m$  variables with the smallest ratio  $\frac{|D_k(y)|}{|D_{k-1}(y)|}$  are the most related to  $x==a$

- So if we relax  $x$ , we ought to relax those too!

- Alternatively....

- If we freeze  $x$ , we ought to freeze those too.



# Freeze Set Size

---

- How many should we freeze?

- PGLNS point of view:

- Meet a target size for the neighborhood of LNS

- Size of neighborhood is  $\mathcal{S}(N) = \prod_{x \in relax(X)} |D(x)|$

- Target value:  $\log \left( \prod_{x \in relax(X)} |D(x)| \right) \leq \alpha$

$$\sum_{x \in relax(X)} |D(x)| \leq \alpha$$



# PGLNS in a Nutshell

---

```
PGLNS(<X,D,C>) {  
  while (relaxSize >  $\alpha$ ) {  
    if (relatedList is empty)  
      choose an unbound variable  
    else choose variable in relatedList  
    freeze chosen variable  
    propagate  
    collect relatedList  
  }  
}
```

Any downsides?



# What is Really Happening

---

- The algorithm *freezes related* variables
- So the relaxed neighborhood contains....
  - The leftovers!
- We might want the neighborhood to contain the related variables
  - And the frozen part should have the leftovers....
- Second variant
  - Reversed Propagation Guided LNS





# Reversing Scheme

---

- **Idea**

- Use “normal” PGLNS to measure relatedness of variables

- **Output**

- For each variable, a sorted list of related variables.

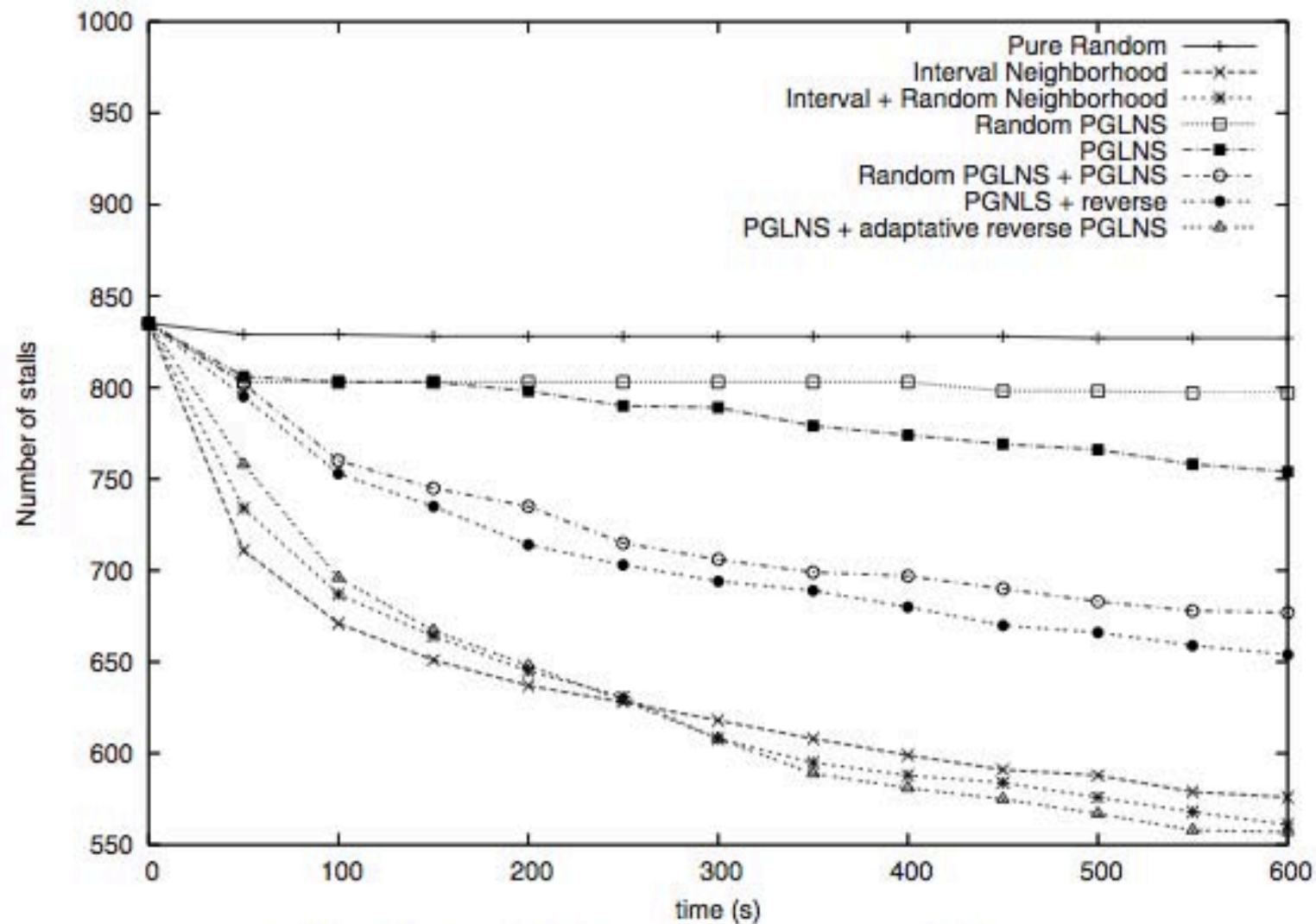
- **Then apply a reverse algorithm that**

- Iteratively grows the relaxed set
  - Add to the relaxed set the most related variable first
  - Merge candidate list with most related to newly added
  - When done: freeze the rest

- **Target-size for relaxed set**

- As before + modulate with context-sensitive widening constant

# A look at results on car sequencing [Perron,04]



**Fig. 3.** Results for problems with size 500.



# Lessons....

---

- CP/LS Hybrids use a few simple ideas
  - Sequential composition
  - Parallel composition
  - Iterative model refinements (of relaxed models)
  - LNS
- CP/LS Hybrids can be extremely effective combinations
  - Better solutions quickly
  - Reduce proof time
  - Ideal for optimization



# Lessons...

---

- Modeling is still a craft...
  - Decide what to relax
  - Decide how to chain
  - Try different combinations!
- Key to ease of use
  - “Compatible” solvers
  - Ease of communication



# A Few Bibliographic links for LNS

---

- Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems
  - [P. Shaw, 1998]
- VNS/LDS+CP : A Hybrid Method for Constraint Optimization in Anytime Contexts
  - [Loudni et al.,2001]
- Propagation Guided Large Neighborhood Search
  - [Perron et al. 2004]
- An Efficient Model and Strategy for the Steel Mill Slab Design Problem
  - [Gargani, 2007]
- Protein Structure Prediction with Large Neighborhood Constraint Programming Search
  - [Dotu et al, 2008]
- A Hybrid LS-CP Solver for the Shifts and Breaks Design Problem
  - [Gaspero et al.,2010]
- Solving Steel Mill Slab Problems with constraint-based techniques: CP, LNS, and CBLs
  - [Schaus et al., 2011]