Constraints-Based Local Search

Pascal Van Hentenryck
Brown University

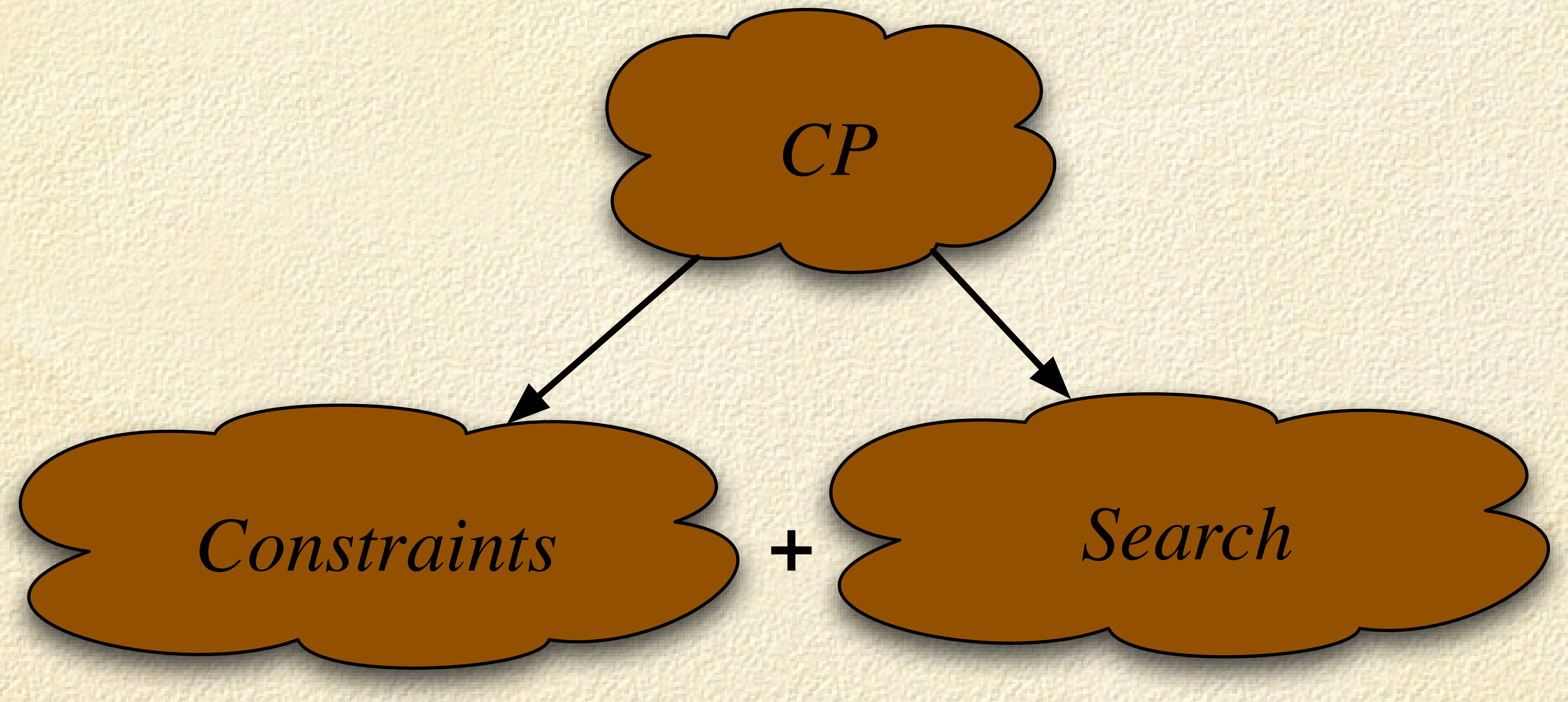
Laurent Michel
University of Connecticut

Overview

- Introduction
  - Perspectives
  - Basic example & Computation Models
- Puzzles
- Summary
- *Larger* Application
- Implementation
- Conclusions

Constraint Programming

- Central Idea



Constraint Programming

- First key idea
  - Convey the combinatorial structure
- Rich modeling language
  - Numerical Constraints
  - Combinatorial Constraints
  - Constraint Combinators
    - Logical and cardinality constraints
    - Reification: constraint → variable
  - Vertical extensions
    - Scheduling / Routing

Constraint Programming

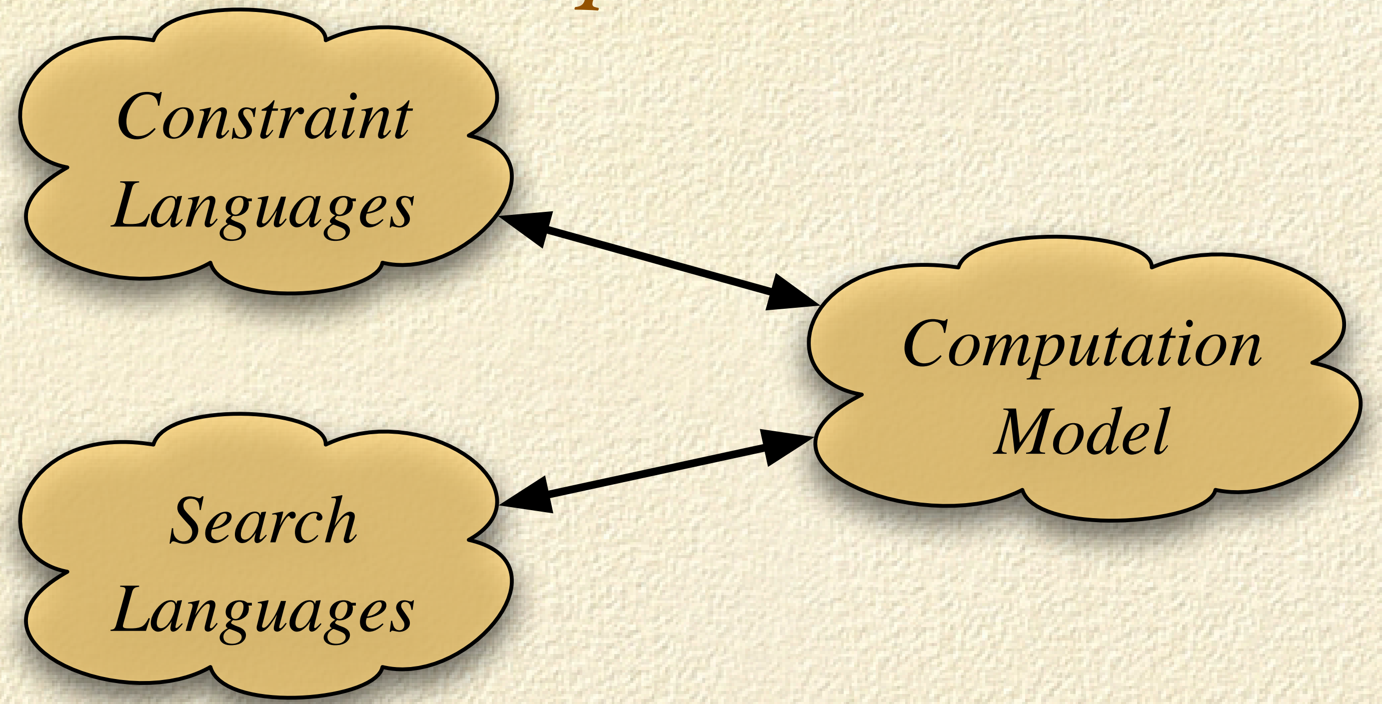
- Why such a rich modeling language?
  - Expressiveness
  - Efficiency
- Expressiveness
  - Easily express complex/idiosyncratic constraints
  - More natural and easier to read
- Efficiency
  - Exploit special structure in filtering algorithms

Constraint Programming

- Second key idea
  - Specify the search procedure
- Rich language for specifying search algorithms
  - Nondeterministic control structures
    - Specifying the search tree
  - Search Strategies
    - How to explore the tree

Constraint Programming

- Key observations: *Independence*



- Can it be useful for another technology?
  - Local search?
  - Integer programming?

Local Search

- No communication at the model level
  - Rare to see the word constraint in papers
- No modeling language
  - No coding at model level
  - No compositionality, reuse, modularity
- Efficiency is an issue
  - Imperative in nature
  - Incrementality

Local Search

- Large scale optimization
  - thousands of variables
- Optimization under time constraints
  - online optimization
- Various classes optimization problems
  - complex scheduling
  - vehicle routing
  - frequency allocation

Comet (2001-)

- Constraint language for local search
  - Rich language for expressing problems
  - Rich language for expressing search
- Problem modeling
  - Declarative specification of the solutions
- Search
  - High-level control structures
  - Modularity and genericity

Goals of the Talk

- Constraint-based language
  - For Local Search
- Computation model
  - For Constraint-based Local Search
- Applications

Central Message

CP = Constraints + Search

LS = Constraints + Search

- Constraints
  - Express structure
- Search
  - Exploit structure

Overview

- Introduction
  - Perspective
  - Basic example & Computation Models
- Puzzles
- Summary
- *Larger* Application
- Implementation
- Conclusions

Getting Started

- Problem
  - 8-Queens....
- Model
  - Decision variables
    - A row assignment for each column
  - Constraints
    - Properties of the solution
  - Search
- Goals
  - Illustrate modeling
  - Illustrate search

First in CP....
... Then in CB-LS

Queens Model in CP

Decision Variables

```
range R = 1..8;
var R queen[R];
int pos[k in R] = k;
int neg[k in R] = -k;
solve {
  allDifferent(queen);
  allDifferent(queen,neg);
  allDifferent(queen,pos);
};
```

Combinatorial Constraints

Searching with CP

```
range R = 1..8;
var R queen[R];
solve {
   ...
};
search {
 forall(i in R ordered by increasing dSize(queen[i]))
    tryall(v in R)
       queen[i] = v;
};
```

Non deterministic choice

Variable selection heuristic

Queens Model in Comet

```
range R = 1..8;
UniformDistribution d(R);
LocalSolver m();
var{int} queen[i in R](m) := d.get();
ConstraintSystem S(m);
  S.post(alldifferent(queen));
  S.post(alldifferent(all(k in R) queen[k]+k));
  S.post(alldifferent(all(k in R) queen[k]-k));
m.close();
```

Initial value

Combinatorial constraints

Searching in LS

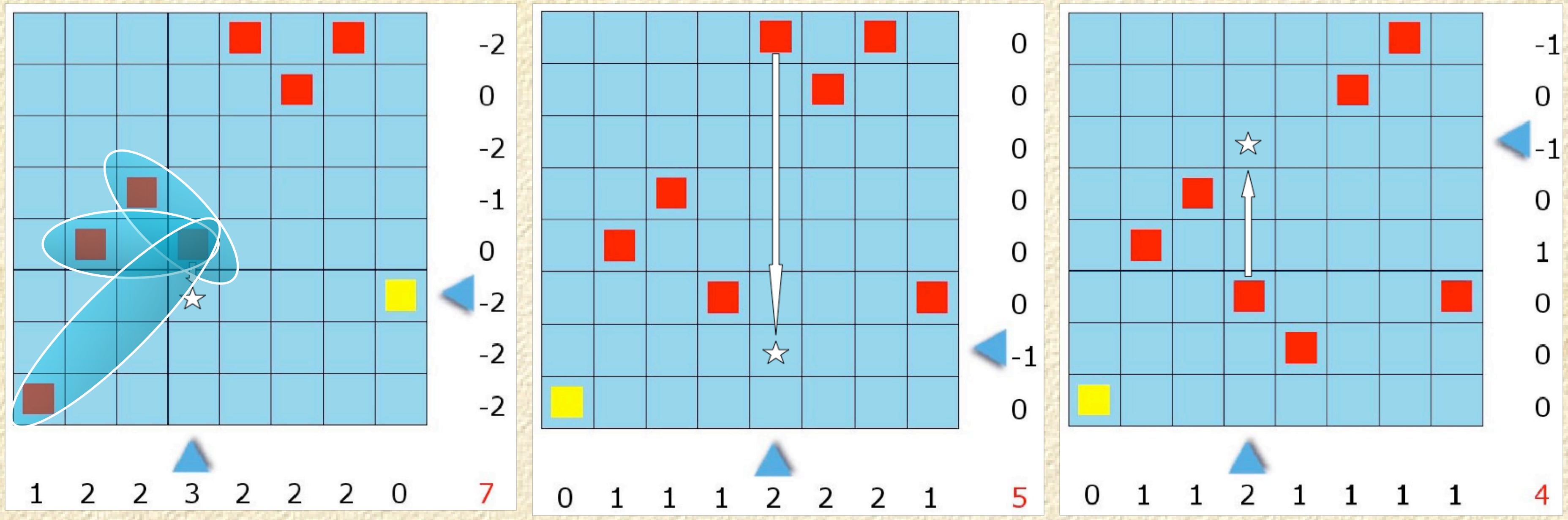
```
while (S.violationDegree())
  selectMax(k in R)(S.getViolations(queen[k]))
   selectMin(v in R)(S.getAssignDelta(queen[k],v))
         queen[k] = v;
```

As long as system is violated

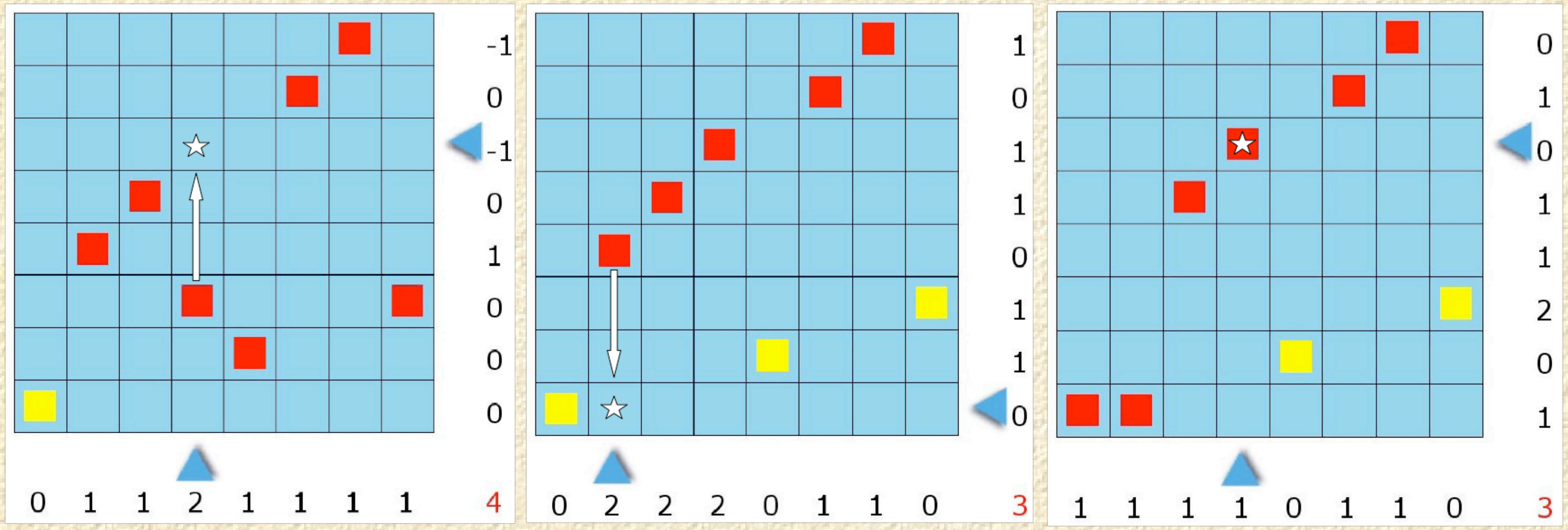
Choose most conflicting queen

Choose a row leading to largest violation decrease

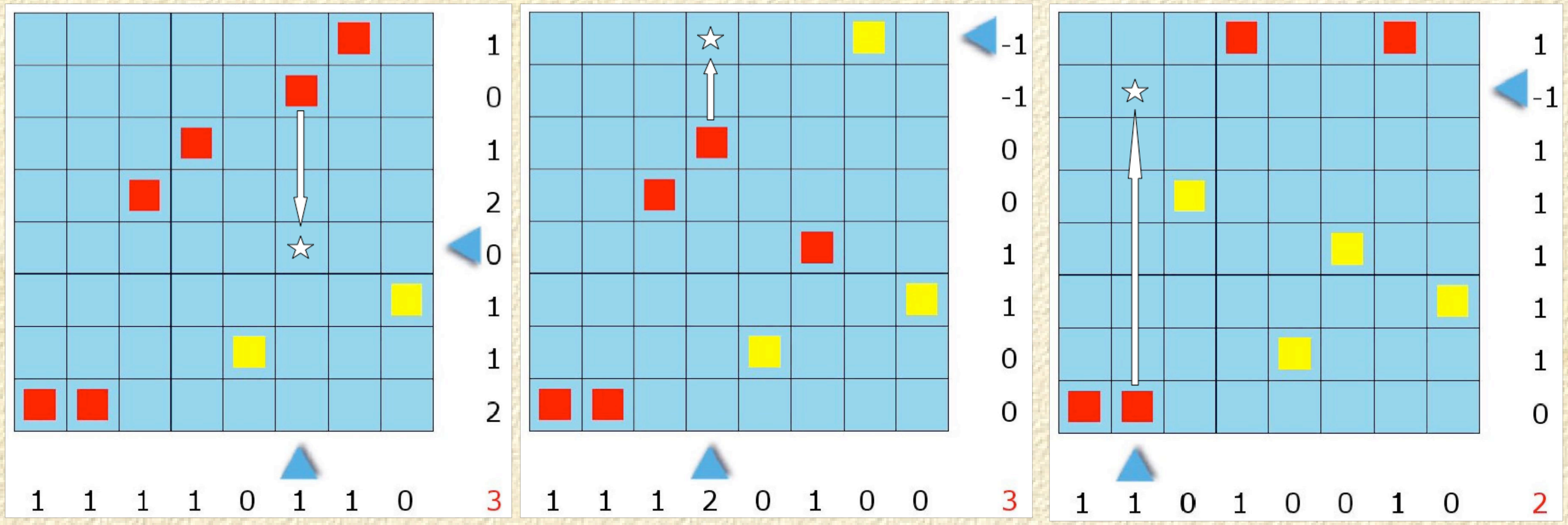
Queens Propagation in LS



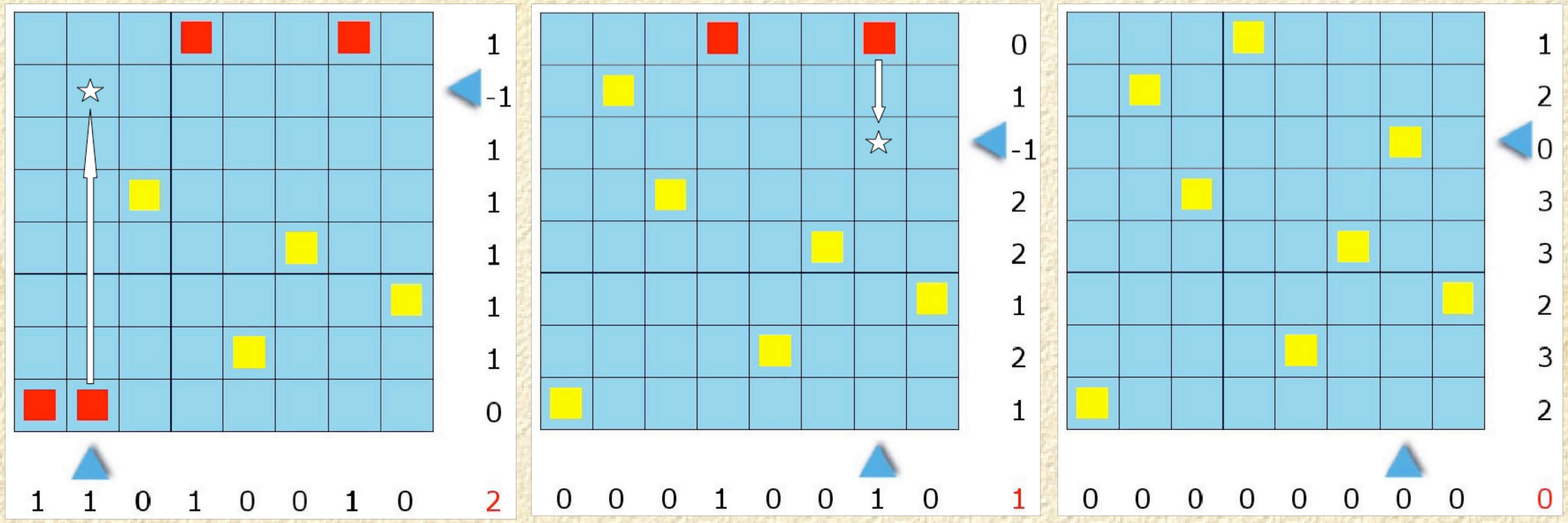
Queens Propagation in LS



Queens Propagation in LS



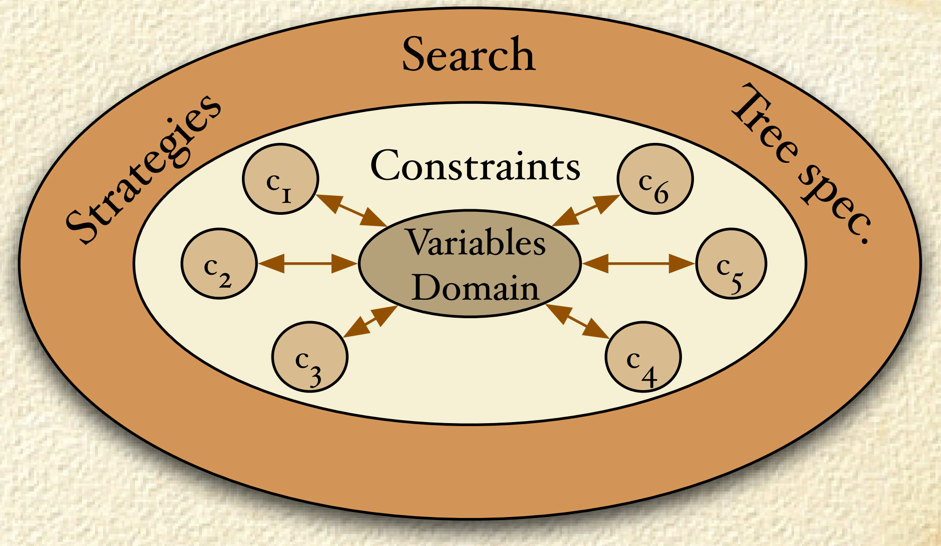
Queens Propagation in LS



Summary

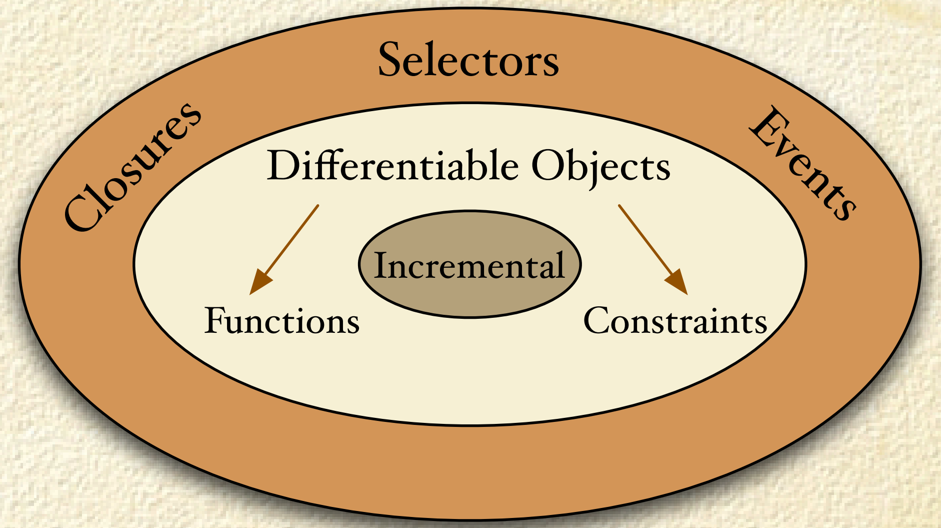
- Modeling
  - Identical for CP and LS
- Search
  - Influenced by computational model
    - CP
      - Exploit pruning
    - LS
      - Exploit violations and differentiability

The CP Architecture



- Three Layers
  - Domain variables
  - Constraints
    - Logical / Numerical
    - Combinatorial
  - Search
    - Tree
    - Strategy
- Computational model
  - Constraints ⇒ pruning
  - Search = Tree specification + Tree exploration

The LS Architecture



- Three Layers
  - Incremental variables
  - Constraints
    - Logical / Numerical
    - Combinatorial
  - Search
    - Graph exploration: Heuristics
    - Meta-Heuristics
- Computational model
  - Constraints $\Rightarrow$ violations + differentiation
  - Search = Neighborhood + Heuristic + Meta

Overview

- Introduction
  - Perspective
  - Basic example & Computation Models
- ☑ Puzzles
- Summary
- *Larger* Application
- Implementation
- Conclusions

Purpose of the section

- Modeling
  - Illustrate
    - Numerical constraints
    - Logical constraints
    - Combinatorial constraints
    - Redundant constraints
- Search
  - Illustrate
    - Typical search procedures

The Puzzles

- Send More Money!
- Magic Series
- The Zebra

Send More Money!

- Problem statement
  - Assign
    - Digit
    - To Letters
    - Satisfy crypto-puzzle
  - Approaches ?
    - Direct

```
    S E N D
+   M O R E
  M O N E Y
```

```
               S*1000+E*100+N*10+D
+              M*1000+O*100+R*10+E
= M*10000+O*1000+N*100+E*10+Y
```

Send More Money!

- Problem statement
  - Assign
    - Digit
    - To Letters
    - Satisfy crypto-puzzle
  - Approaches ?
    - Direct
    - Carry

	C_4	C_3	C_2	C_1	
		S	E	N	D
+		M	O	R	E
	M	O	N	E	Y

$$C_1 + N + R = E + 10 * C_2$$

CP Model [Carry]

```
enum Letters = {S,E,N,D,M,O,R,Y};
range Digits = 0..9;
Range Bin    = 0..1;
var Digits value[Letters];
var Bin   r[1..4];
solve {
  alldifferent(value);
  value[S] <> 0;
  value[M] <> 0;
  r[4]                                 == value[M];
  r[3] + value[S] + value[M] == value[O] + 10 * r[4];
  r[2] + value[E] + value[O]  == value[N] + 10 * r[3];
  r[1] + value[N] + value[R]  == value[E] + 10 * r[2];
         value[D] + value[E]  == value[Y] + 10 * r[1];
};
```

LS Model [Carry]

```
LocalSolver m();
enum Letters = {S,E,N,D,M,O,R,Y};
range Digits = 0..9;
UniformDistribution distr(Digits);
var{int} value[Letters](m,Digits) := distr.get();
var{int} r[1..4](m,0..1) := 1;
ConstraintSystem Sys(m);
Sys.post(alldifferent(value));
    Sys.post(            value[S]  != 0);
    Sys.post(            value[M] != 0);
    Sys.post(r[4]                           == value[M]);
    Sys.post(r[3] + value[S] + value[M]   == value[O] + 10 * r[4]);
    Sys.post(r[2] + value[E] + value[O]   == value[N] + 10 * r[3]);
    Sys.post(r[1] + value[N] + value[R]   == value[E] + 10 * r[2]);
    Sys.post(         value[D] + value[E]   == value[Y] + 10 * r[1]);
```

Magic Series

- Objective
  - Modeling
    - Meta constraints
    - Numerical constraints
    - Redundant constraints
- Approaches
  - CP
  - LS
    - Impact of redundancies

The Problem

- Find a sequence of length n such that
  - Sk = Number of occurences of k in S
- Example
  - N = 10
  - S = [6,2,1,0,0,0,1,0,0,0]
    - 6 occurences of 0
    - 2 occurences of 1
    - 1 occurrence of 0
    - ...

CP Model

```
int n = 50;
range Size = 0..n-1;
var Size magic[Size];
forall(k in Size)
   exactly(magic[k],all(j in Size) magic[j] = k);

sum(k in Size) k * magic[k] = n;
```

magic[k] is #occurrence of k in magic

LS Model

```
int n = 50;
range Size = 0..n-1;
LocalSolver m();

var{int} magic[Size](m,Size) := 0;
ConstraintSystem S(m);
forall(v in Size)
  S.post(exactly(magic[v],all(i in Size) magic[i] == v));


m.close();
```

true expressions is magic[v]

Performance

- Observation
  - The model works
  - But it takes a long time
- What is going on ?
  - The exactly constraints provide little guidance for value selection
- Solution
  - Add a redundant constraint
  - Redundant captures the importance of values

Redundant Model

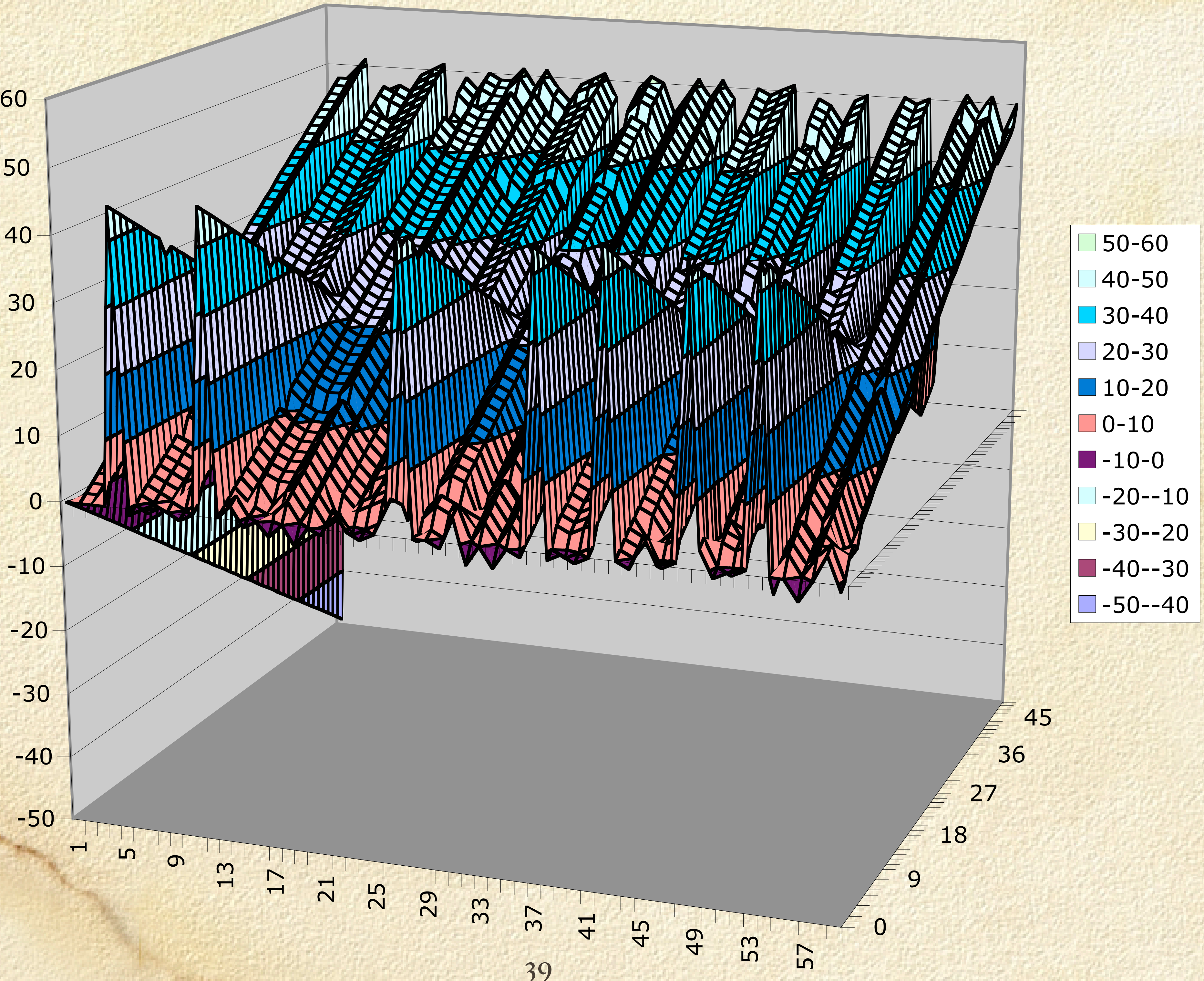
```
int n = 50;
range Size = 0..n-1;
LocalSolver m();

var{int} magic[Size](m,Size) := 0;
ConstraintSystem S(m);
forall(v in Size)
  S.post(exactly(magic[v],all(i in Size) magic[i] == v));
S.post(sum(k in Size) k * magic[k] == n);


m.close();
```

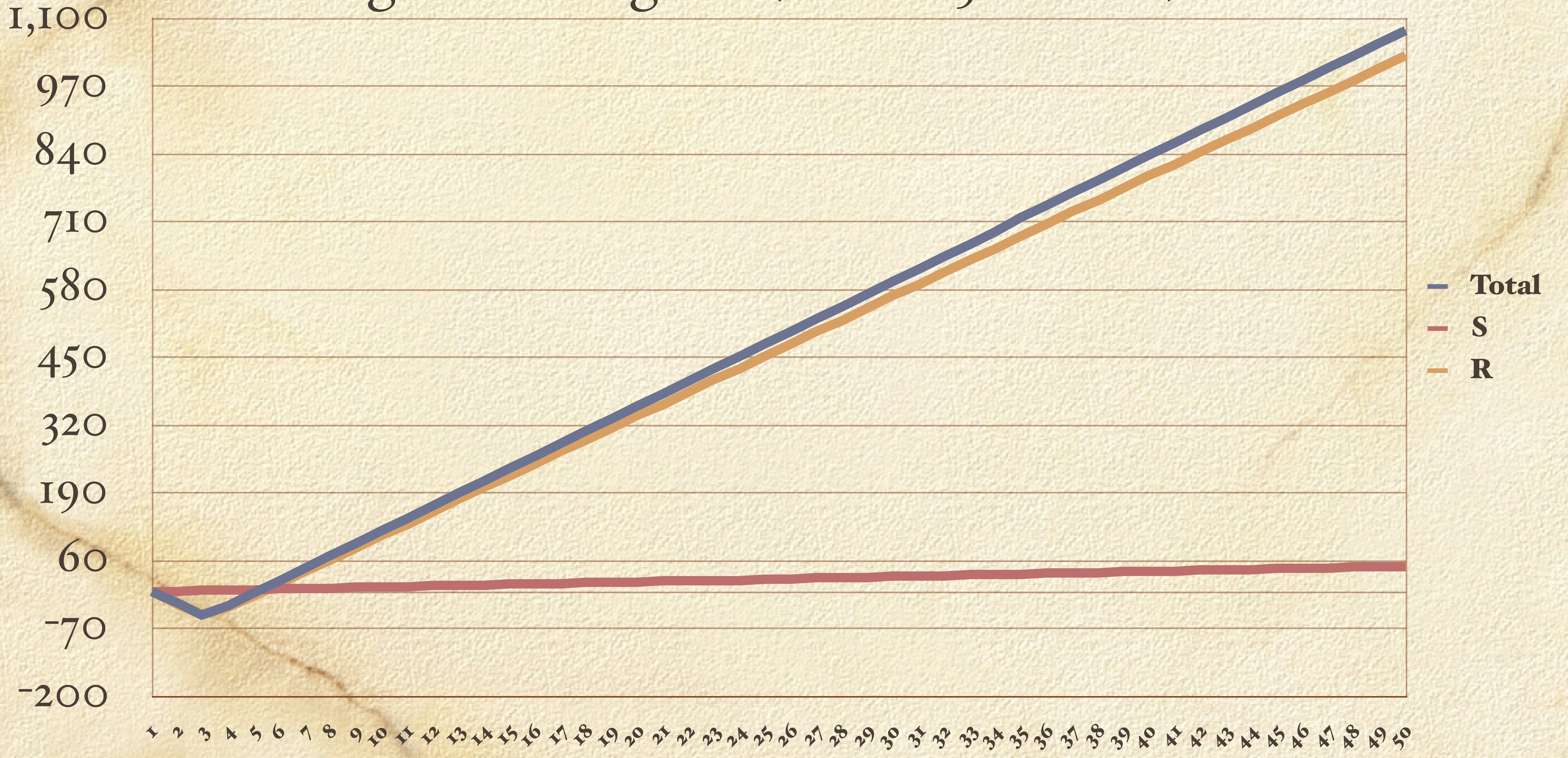
Same redundant as in CP Model!

Plots. No Redundant



Plots. With The Redundant.

- Assign Delta Signals (for the 50 values)



Who Owns The Zebra ?

- Objective
  - Modeling
    - Logical constraint
  - Search
    - Show an non-trivial procedure

Problem Statement

- Assign
  - People/Animals/Drinks/Color/Jobs
  - To Houses
  - Satisfy given constraints
    - E.g. “The Englishman in the red house”
- Question
  - Who owns the Zebra ?

LS Model. The Variables

```
enum N = { England, Spain, Japan, Italy, Norway};
enum C = { green, red, yellow, blue, white};
enum P = { painter, diplomat, violinist, doctor, sculptor};
enum A = { dog, zebra, fox, snails, horse };
enum D = { juice, water, tea, coffee, milk };
range R = 1..5;

LocalSolver m();
UniformDistribution distr(R);
var{int} n[N](m,R) := distr.get();
var{int} c[C](m,R) := distr.get();
var{int} p[P](m,R) := distr.get();
var{int} a[A](m,R) := distr.get();
var{int} d[D](m,R) := distr.get();
```

LS Model. The Constraints

```
ConstraintSystem S(m);
  S.satisfy(n[England] == c[red]);
  S.satisfy(n[Spain] == a[dog]);
  S.satisfy(n[Japan] == p[painter]);
  S.satisfy(n[Italy] == d[tea]);
  S.satisfy(n[Norway] == 1);
  S.satisfy(d[milk] == 3);
  S.satisfy(p[violonist] == d[juice]);
  S.satisfy(c[green] == d[coffee]);
  S.satisfy(p[sculptor] == a[snails]);
  S.satisfy(p[diplomat] == c[yellow]);
  S.satisfy(c[green] == c[white] + 1);
  S.satisfy(abs(a[fox] - p[doctor]) == 1);
  S.satisfy(abs(a[horse] - p[diplomat]) == 1);
  S.satisfy(abs(n[Norway] - c[blue]) == 1);
  S.post(alldifferent(n));S.post(alldifferent(c));
  S.post(alldifferent(p));S.post(alldifferent(a));
  S.post(alldifferent(d));
```

0 or 1

Satisfaction Constraints

LS Search Ingredients

- Objective
  - Minimize violations of relaxed constraints
- Constraint selection
  - Most violated constraint
- Variable selection
  - Most violating variable
- Value selection
  - Value leading to largest decrease in violations
- Meta-heuristics
  - Guide the heuristic to avoid local optima

Constrained Directed Search

Select a Violated Constraint

```
void function cdsSearch(ConstraintSystem S) {
  int it = 0;
  int tabu[S.getIdRange()] = -1;
  var{int}[] violation = S.getCstrViolations();
  while (Sys.violations() > 0) {
    select(c in violation.rng(): violation[c] > 0) {
      Constraint cstr = S.getConstraint(c);
      var{int}[] x = cstr.getVariables();
      selectMin(v in x.rng(),id=x[v].getId(),d in x[v].domain():
                      tabu[id]<=it && cstr.getAssignDelta(x[v],d) < 0)
                     (S.getAssignDelta(x[v],d)) {
        x[v] := d;
        tabu[id] = it + 4;
      }
    it++;
  }
}
```

Select a variable next

Constrained Directed Search

Select a Violated Constraint

```
void function cdsSearch(ConstraintSystem S) {
  int it = 0;
  int tabu[S.getIdRange()] = -1;
  var{int}[] violation = S.getCstrViolations();
  while (Sys.violations() > 0) {
    select(c in violation.rng(): violation[c] > 0) {
      Constraint cstr = S.getConstraint(c);
      var{int}[] x = cstr.getVariables();
      selectMin(v in x.rng(),id=x[v].getId(),d in x[v].domain():
                    tabu[id]<=it && cstr.getAssignDelta(x[v],d) < 0)
                   (S.getAssignDelta(x[v],d)) {
        x[v] := d;
        tabu[id] = it + 4;
      }
    it++;
  }
}
```

Select a value in v's Domain

Select a variable next

Constrained Directed Search

```
void function cdsSearch(ConstraintSystem S) {
  int it = 0;
  int tabu[S.getIdRange()] = -1;
  var{int}[] violation = S.getCstrViolations();
  while (Sys.violations() > 0) {
    select(c in violation.rng(): violation[c] > 0) {
      Constraint cstr = S.getConstraint(c);
      var{int}[] x = cstr.getVariables();
      selectMin(v in x.rng(),id=x[v].getId(),d in x[v].domain():
                    tabu[id]<=it && cstr.getAssignDelta(x[v],d) < 0)
                   (S.getAssignDelta(x[v],d)) {
        x[v] := d;
        tabu[id] = it + 4;
      }
    it++;
  }
}
```

The expression
to minimize

Summary

- Modeling = Constraint + Search
  - ☑ CP
  - ☑ LS
- Computational Model
  - CP
    Exploit pruning to reduce space
  - LSExploit violations to guide search

Overview

- Introduction
  - Perspective
  - Basic example & Computation Models
- Puzzles
- Summary
- *Larger* Application
- Implementation
- Conclusions

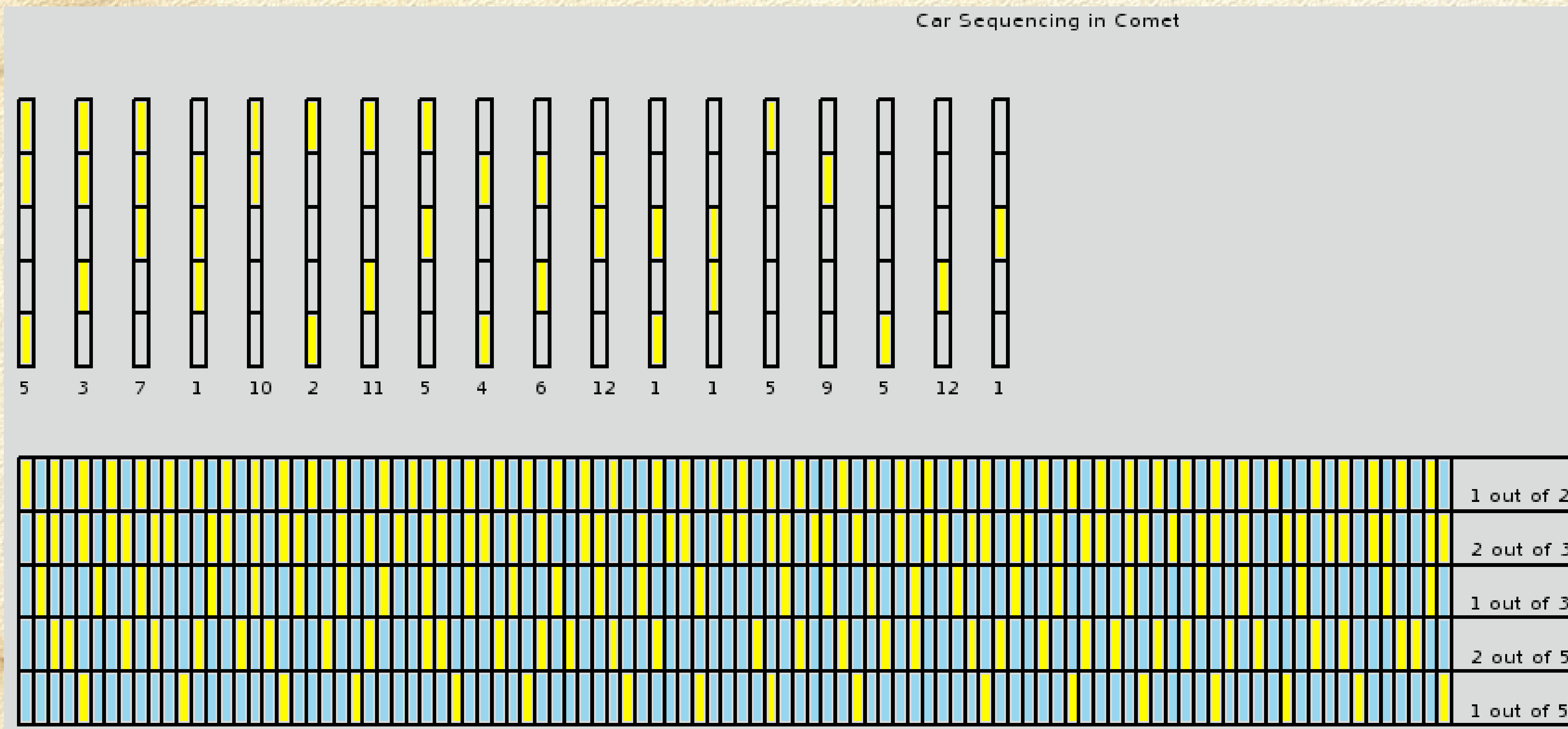
Car Sequencing

- Objective
  - Modeling
    - Higher-order constraints
    - Redundant constraints
- Approaches
  - CP
  - LS

Car Sequencing

- The Problem
  - Place cars on an assembly line subject to
    - Satisfy customers demand (orders)
    - Respect workshop constraints
      - K out of N cars can be processed for option z

Car Sequencing Solution



A Small Instance

Options	1	2	3	4	5	Demand
Class 1	✓		✓	✓		1
Class 2				✓		1
Class 3		✓			✓	2
Class 4		✓		✓		2
Class 5	✓		✓			2
Class 6	✓	✓				2
Capacity	1/2	2/3	1/3	2/5	1/5	

Globalizing

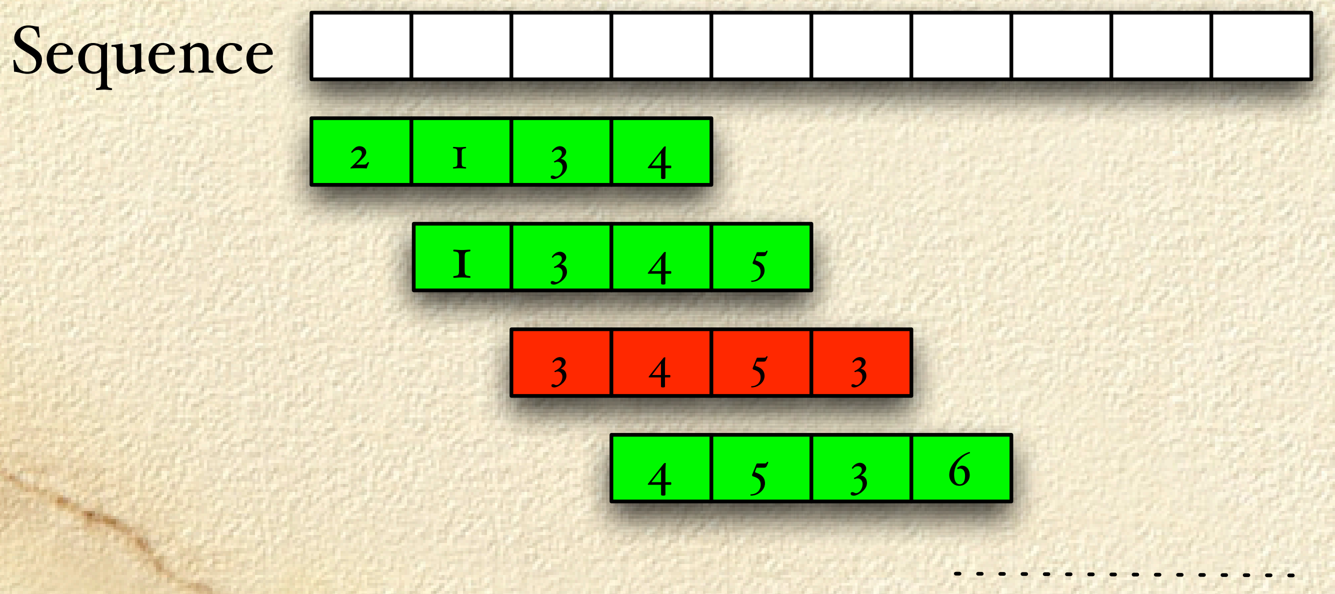
- Motivation
  - There is an underlying modeling concept
  - It arises in many applications
    - Time tabling
    - Sports scheduling
- Implication
  - Express it directly
- Solution
  - A Global constraint
    - Sequence
    - Combines several elementary constraints

Sequence Semantics

- A constraint on a sequence of values
  - Example

```
sequence(var{int}[] S,
         set{int} values,
         int     atMost,
         int     length);
```

Length = 4
Atmost = 2
Values = {3,5}



CP Constraint

```
solve {
  forall(o in Options)
      sequence(slot,options[o],capacity[o].l,capacity[o].u);
}
```

LS Model. The Constraints

```
int cars[nbCars] = ...;
RandomPermutation p(Slots);
forall(s in Slots) slot[s] := cars[p.get()];

ConstraintSystem S(m);
forall(o in Options)
  S.post(sequence(slot,options[o],cap[o].lb,cap[o].ub));
var{int} violations = S.violations();
m.close();
```

LS Model. The Search.

```
int itLimit = 2000000;
Counter it(m,0);
UniformDistribution d(1..10);
int tabu[Slots,Slots] = -1;
int best             = violations;

while (violations > 0 && it < itLimit) {
 selectMax(s in Slots)(S.getViolations(slot[s])) {
  selectMin(v in Slots,nv = S.getSwapDelta(slot[s],slot[v]):
          slot[s] != slot[v] &&
          (tabu[s,v] <= it || violations + nv < best))(nv) {
     slot[s] :=: slot[v];
     tabu[s,v] = it + violations + d.get();
     tabu[v,s] = tabu[s,v];
   }
  }
 it++;
}
```

Select *Most Violating* Slot

Select slot to swap with that
- *Yields largest* violation *decrease*
- Is *non tabu* or *outstanding*

Swap *them*!

LS Model. Meta-Heuristic

- How to introduce...
  - Diversification
    - Purpose: *When no improvement for a while*, perturb the assignment
  - Restarts
    - Purpose: *Starts from scratch every x iterations*.
- Bottom line
  - Track the best solution at all time.
  - Code it independently from the heuristic
  - Use 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

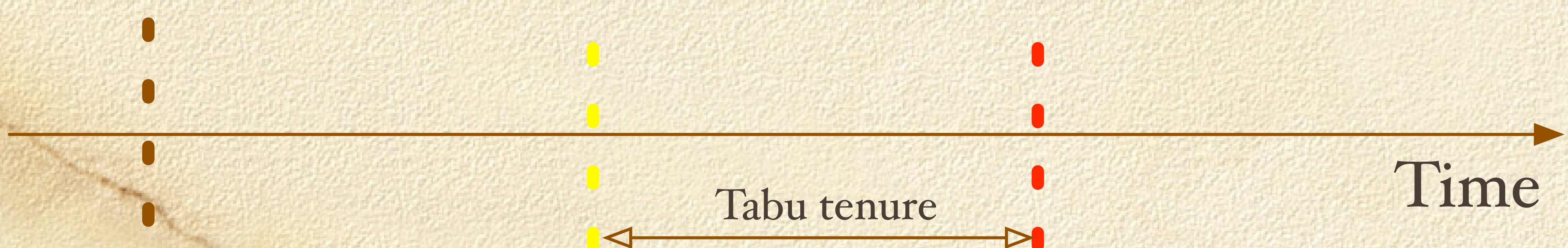
Events Anatomy

- 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

Events and Closures

```
forall(q in R) {
  whenever queen[q]@changes(int o,int n) {
        tabu.insert(q);
        when it@reaches[it+tlen]() tabu.remove(q);
  }
}
```

This yellow code is executed *much later*



LS Model. Meta-Heuristic

- First step
  - Track the best solution
  - Use an Event!

```
Solution solution = new Solution(m);

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

LS Model. Meta-Heuristic

- Second step
  - Track the stability
  - Diversification when stable too long

```
whenever it@changes(int o,int n) {
   stable++;
   if (stable == stableLimit) {
      solution.restore();
      forall(i in 1..3)
         select(c in Slots,v in Slots: slots[c] != slots[v])
            slots[c] :=: slots[v];
      best = violations;
      stable = 0;
   }
}
```

LS Model. Meta-Heuristic

- Third step
  - Restart every $2^k * 10000$ iterations.

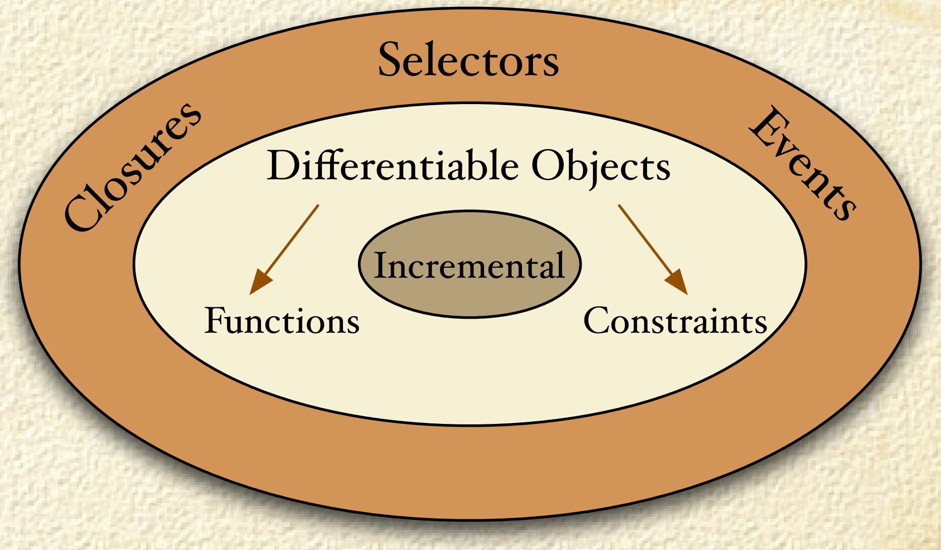
```
restartLimit = 10000;
whenever it@changes(int o,int n) {
    if (n % restartLimit == 0) {
      RandomPermutation p(Slots);
      forall(c in Cars)
         slots[c] := cars[p.get()];
      restartLimit = restartLimit * 2;
      best = violations;
      stable = 0;
    }
}
```

Overview

- Introduction
  - Perspective
  - Basic example & Computation Models
- Puzzles
- Summary
- *Larger* Application
- ☑ Implementation
- Conclusions

Implementation

- Three layers architecture
  - Invariants
  - Differentiable objects
  - Control



Invariants

- Purpose
  - Specify
    - What must be maintained incrementally
  - Automate
    - How to maintain it
  - Compose
    - Multiple invariants and their incremental code
- Example

```
LocalSolver m();
inc{int} x[i in 1..10](m) := i;

inc{int} y <- sum(i in 1..10) x[i];
```

Invariants Vocabulary

- Rich modeling
  - Numerical invariants

```
inc{int} loss[w in W] <- - sum(s in  S[w]) (d[s] - b[s]) ;
```

  - Set-based invariants

```
inc{set{int}} S[w in W] <- setof(s in S) (cost[w,s] = b[s]) ;
```

  - Combinatorial invariants

```
inc{set{int}} S[] = count(x);
```

$$S_j = |\{k \in D(x) | x[k] = j\}|$$

Differentiable Objects

- Kinds
  - Constraints
  - Objective functions
- Purpose
  - Capture properties of the solution
  - Answer differential queries
    - *E.g....*

"What is the impact of assigning variable x to value k ?"

Which Properties?

- Properties of interest
  - Truth value
  - Violation degree
  - Contributions of a variable to overall violation...
- Differential Queries
  - Variation of violation degree (or objective value) as a result of...
    - Single assignment
    - Multiple assignments
    - Swaps
    - ....

Constraint/Objective API

```
interface Constraint {
  inc{int}[] getVariables();
  inc{int} true();
  inc{int} violationDegree();
  inc{int} violations(inc{int} var);
   ...
  int getAssignDelta(inc{int} x,int v);
  int getSwapDelta(inc{int} x,inc{int} y);
  int getAssignDelta(inc{int}[] x,int[] v);
}
```

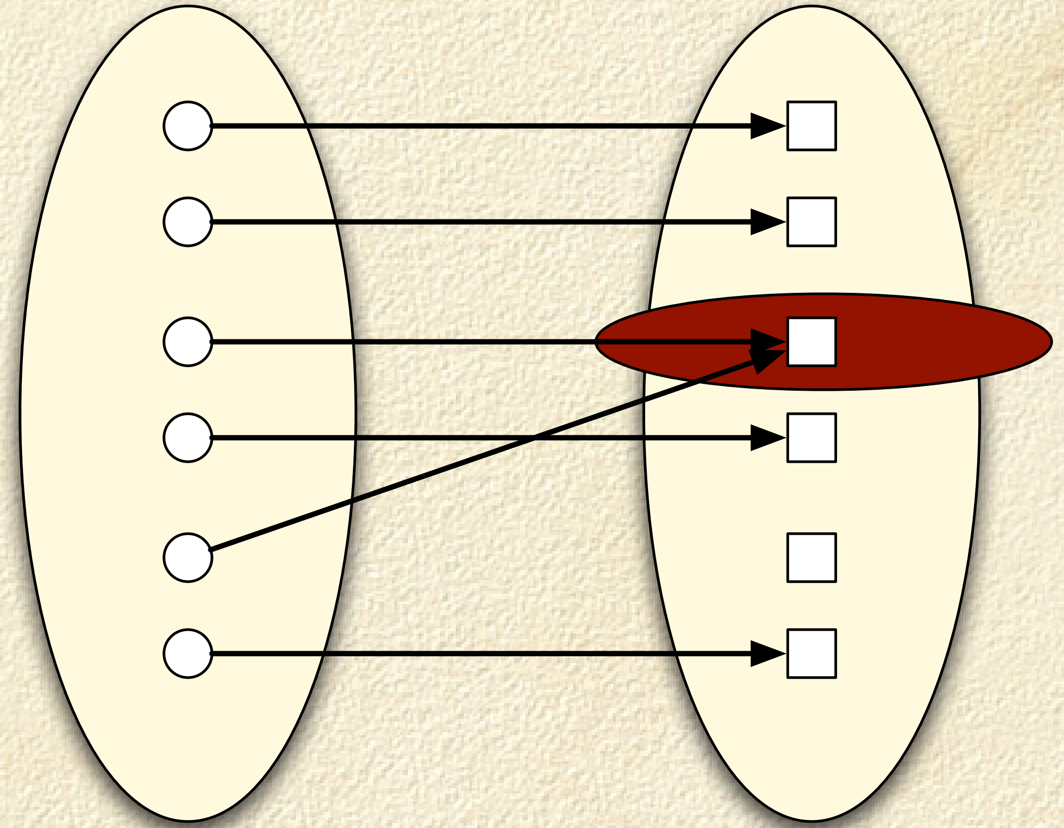
```
interface Objective {
  inc{int}[] getVariables();
  inc{int} value();
  inc{int} cost();
  inc{int} getCost(inc{int} var);
   ...
  int getAssignDelta(inc{int} x,int v);
  int getSwapDelta(inc{int} x,inc{int} y);
  int getAssignDelta(inc{int}[] x,int[] v);
}
```

API Completely open.

Constraints/Objective can be implemented in C++ / Comet

Example

- Implementing an alldifferent!
- Properties
  - Value cardinality - Value violation
  - Variable violation
  - Violation Degree
  - Truth



Alldifferent Properties

- Cardinality [⇢value violation]

$$c_\alpha[j] = |\{k \in D(x) | \alpha(x[k]) = j\}| \Rightarrow c = count(x)$$

- Variable violation

$$v_\alpha(\texttt{allDiff}, x) = \max(c_\alpha[\alpha(x)] - 1, 0)$$

- Total violation degree

$$v_\alpha(\texttt{allDiff}) = \sum_{i \in D} \max(c_\alpha[i] - 1, 0)$$

- Truth

$$v_\alpha(\texttt{allDiff}) == 0$$

All maintained
with
Invariants

Alldifferent Differential API

- Focus on
  - c.getAssignDelta(x,v)

$$\Delta(x := v) = \begin{cases} 0 & \text{if } \alpha(x) = v \\ \underbrace{(c_\alpha[v] \geq 1)}_{\text{New violations introduced on } v} - \underbrace{(c_\alpha[\alpha(x)] \geq 2)}_{\text{Old violations caused by } x} & otherwise \end{cases}$$

Overview

- Introduction
  - Perspective
  - Basic example & Computation Models
- Puzzles
- Summary
- *Larger* Application
- Implementation
- ☑ Conclusions

Conclusions

- Key ideas in constraint languages
  - Applications = Constraints + Search
- Constraints
  - Make structure explicit
- Search
  - Exploit structure
- Technology independent
  - Constraint programming and local search

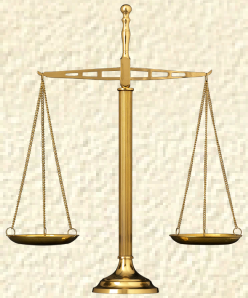
Conclusions

- Constraints
  - Numerical
  - Combinatorial
  - Constraint combinators: Logical, cardinality
- Different uses
  - Pruning in constraint programming
  - Violations and differentiation in local search
- Modeling techniques
  - Redundancy: useful in both for different reasons
  - Symmetries: useful in CP, detrimental in LS?

Conclusions

- Search
  - Independent from model
  - Genericity
- Different computation models
  - Branching in constraint programming
  - Neighborhood exploration/selection in LS
- Commonalities
  - Exploit the model properties (generically)
  - High-level abstractions
  - Significant reduction in programming effort

CP and LS contrasted



Issue	CP	LS
Variables Constraints	Logical / Domain Logical Numeric Combinatorial	Incremental Logical Numeric Combinatorial
Search	Tree Non-deterministic Strategies	Graph Randomized Meta-Heuristics
Architecture Constraints Search	3 Layers Pruning Choices & Backtrack	3 Layers Differentiability Closure & Inverse

Questions…

EXTRA MATERIAL

Scheduling Vertical Extension

- Take Home Message
  - Very successful in CP
  - Very natural and effective in LS too
  - Similar declarative models
  - The core differences
    - The search
    - The scope
      - CP

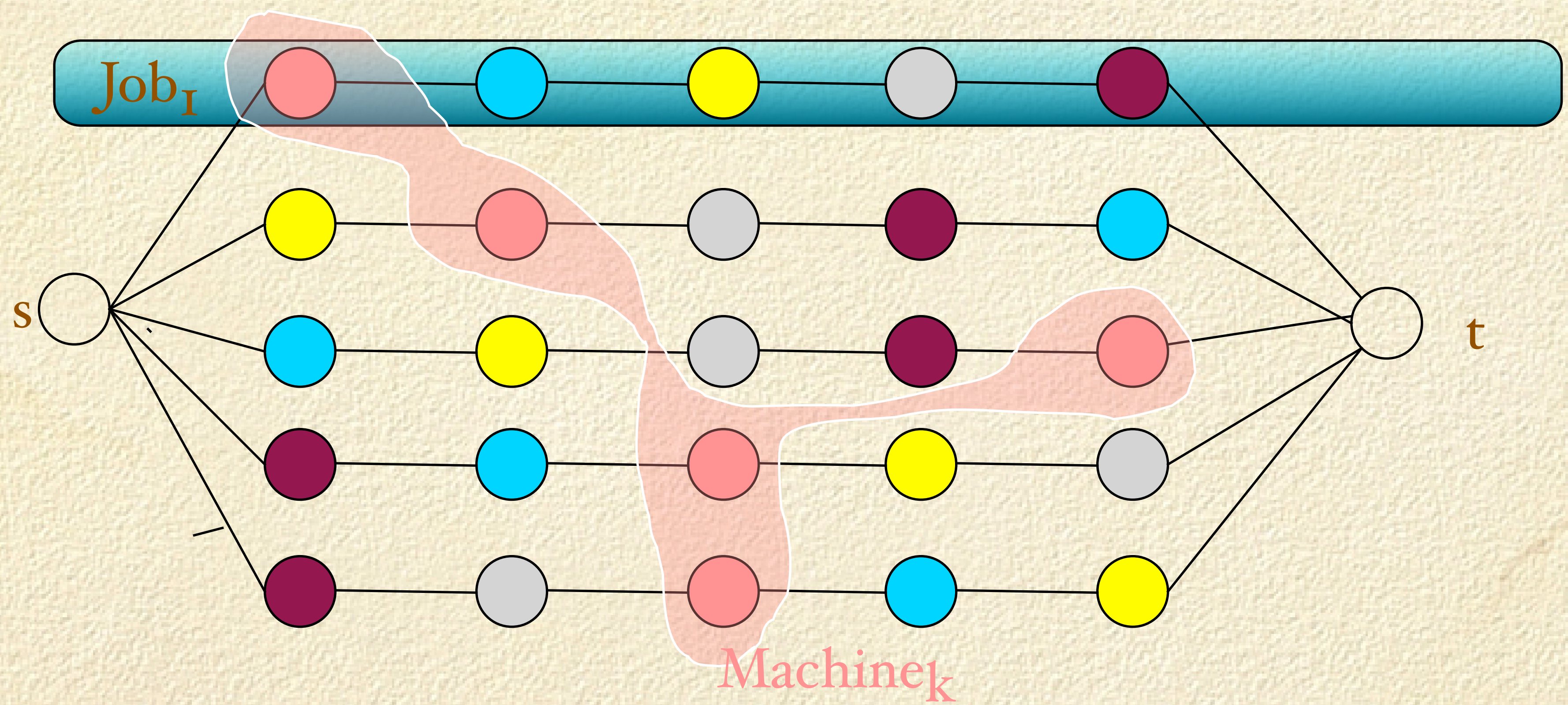
        optimality proof. "Small" instances
      - LS

        no optimality proof. "Large" instances

CP-based modeling

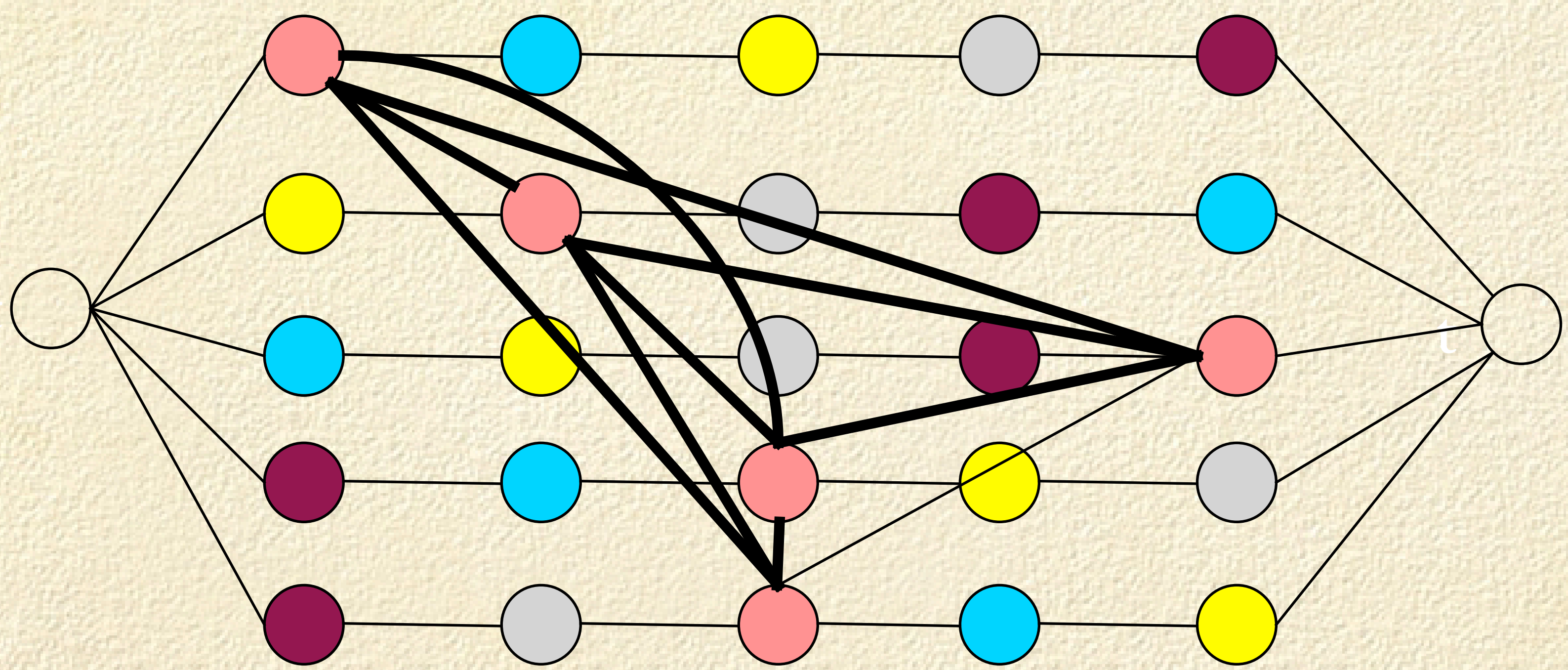
- Activities
- Resources
  - Unary
  - Cumulative
- Precedence

Jobshop Example



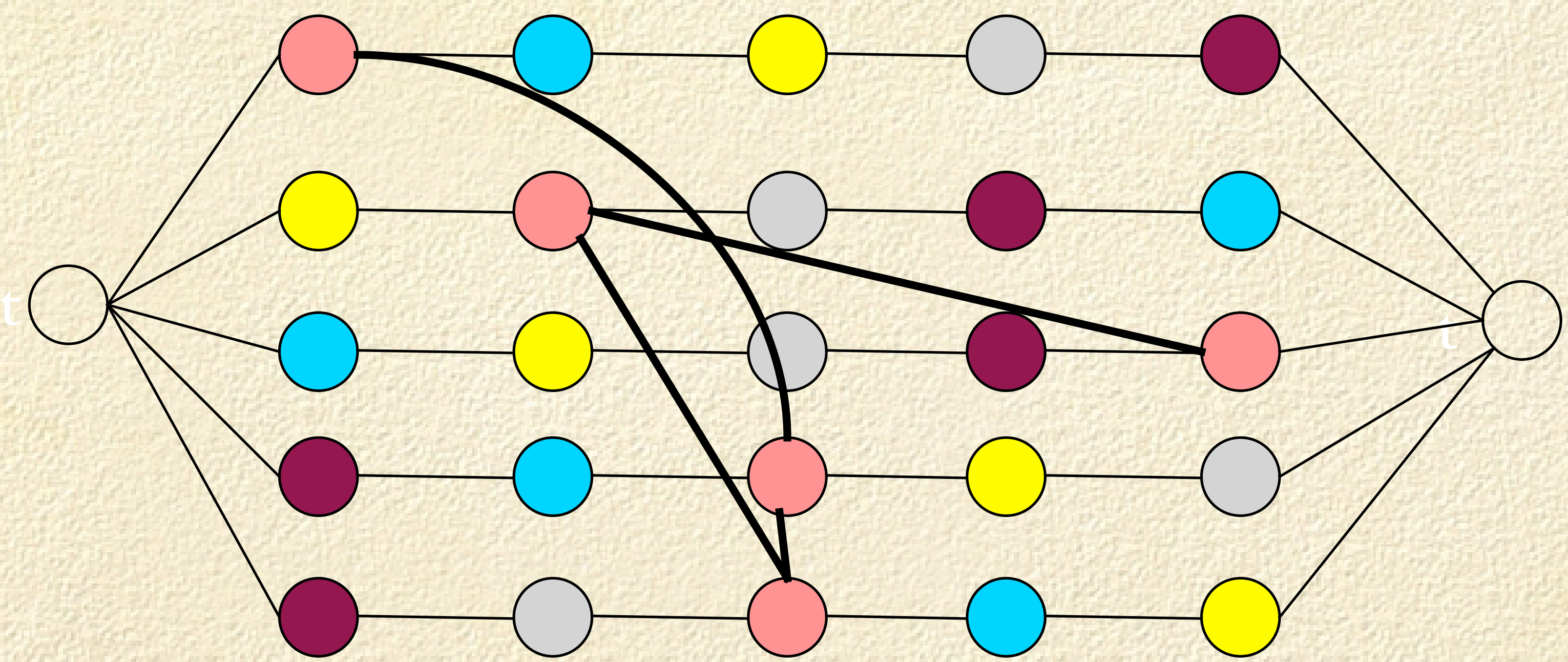
Jobshop Example

- A machine handle activities in sequence
  - Find a activity ordering on each machine



Jobshop Example

- Solution =
  - A Directed acyclic precedence graph



Jobshop Example

- Possible Objectives
  - Minimize makespan
    - Length of longest path s ⇢ t
  - Minimize weighted tardiness
    - Lateness of tasks
  - Minimize total tardiness
    - Sum of all tasks lateness

Cumulative

```
int demand[Tasks] = ...;
ScheduleHorizon = totalDuration;
Activity task[j in Jobs,t in Tasks](duration[j,t]);
Activity makespan(0);
DiscreteResource tool(cap);

minimize makespan.end
subject to {
  forall(j in Jobs)
    task[j,nbTasks] precedes makespan;

  forall(j in Jobs,t in 1..nbTasks-1)
    task[j,t] precedes task[j,t+1];

  forall(j in Jobs, t in Tasks)
    task[j,t] requires(demand[t]) tool;
}
```

Objectives

- Implement a common interface

```
interface Objective {
  inc{int}[] getVariables();
  inc{int} value();
  inc{int} cost();
  inc{int} getCost(inc{int} var);
   ...
  int getAssignDelta(inc{int} x,int v);
  int getSwapDelta(inc{int} x,inc{int} y);
  int getAssignDelta(inc{int}[] x,int[] v);
}
```

Scheduling Objective

- Purpose
  - Provide additional services
  - Provide domain specific services
  - Provide services hard to encode in low level terms

```
interface ScheduleObjective {
  ...
  int evalMoveBackwardDelta(...);
  int evalMoveForwardDelta(...);
  int evalInsert(Activity,DisjunctiveResource);

  int estimateMoveBackwardDelta(...);
}
```

Using Objectives

- Idea
  - Exploit differential API of objective
  - Exploit objective compositionality

```
tardiness.evalMoveBackwardDelta(a);
```

Scheduling Objectives

- Scheduling supports several objectives
  - Makespan
  - Tardiness

```
Makespan      mks(sched);            // A makespan objective
Tardiness     tard(sched,a,dueDate); // a tardiness objective
```

- Objectives compose!

```
Tardiness tard[j in Job](sched,job[j].getLast(),dueDate[j]);
ScheduleObjectiveSum totalTard(sched);
forall(k in Jobs)
  totalTard.add(tard[k]);
```

LS Model. [Jobshop]

```
LocalSolver m();
Schedule sched = new DisjunctiveSchedule(m);

Job job[Jobs];
Activity act[j in Jobs,t in Tasks](sched,duration[j,t]);
DisjunctiveResource tool(sched);
Objective obj = new Makespan(sched);


forall(j in Jobs,t in Tasks)
  act[j,t].requires(tool[res[j,t]]);


forall(j in Jobs,t in 1..nbTasks-1)
  act[j,t].precedes(act[j,t+1]);

m.close();
```

Create Precedence Graph

Create Job Sequences

Create the Activities

Declare objective function

LS Search

- Overall strategy
  - Create an initial solution
    - Use a constructive heuristic
    - Little efforts towards optimization
    - Build a satisfiable solution
  - Conduct an iterative improvement
    - Perform a local change
    - Gear towards better value of Objective
- What is *Difficult* ?

LS Search

- Difficulty
  - Iterative improvement scheme
  - In practice
    - Union of several neighborhood functions
    - Temporal separation of
      - Neighborhood *exploration*
      - From neighbor *selection*
      - From actual *transition*

Tool-less solution

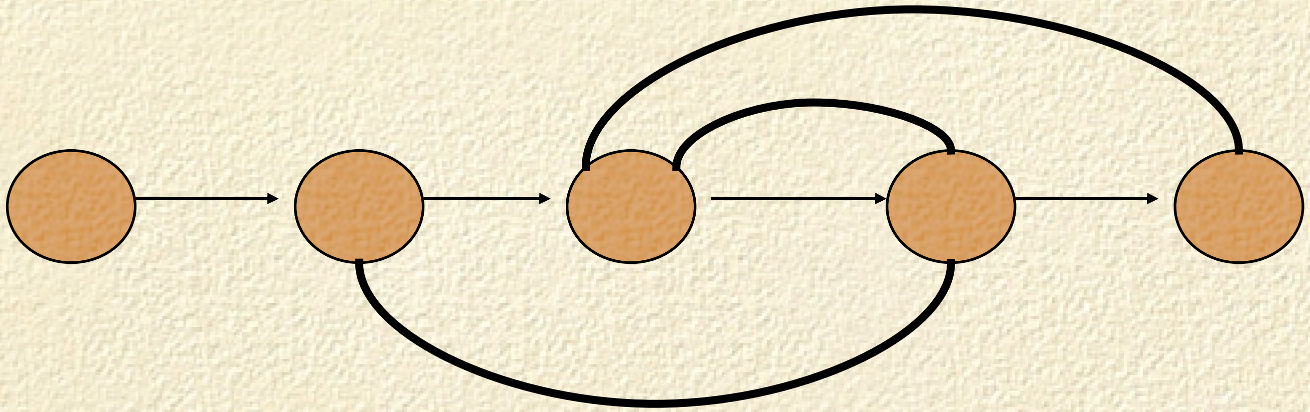
- Typical solution
  - Create classes for each move
    - (with a common interface)
  - Create instances during the scanning phase to select
  - Extract the selection and execute it
- Drawbacks
  - Heavy machinery (hence not generally done)
  - Code fragmentation between
    - Evaluation
    - Execution

LS Search Example

- 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++
  - About 6 months to reproduce the results

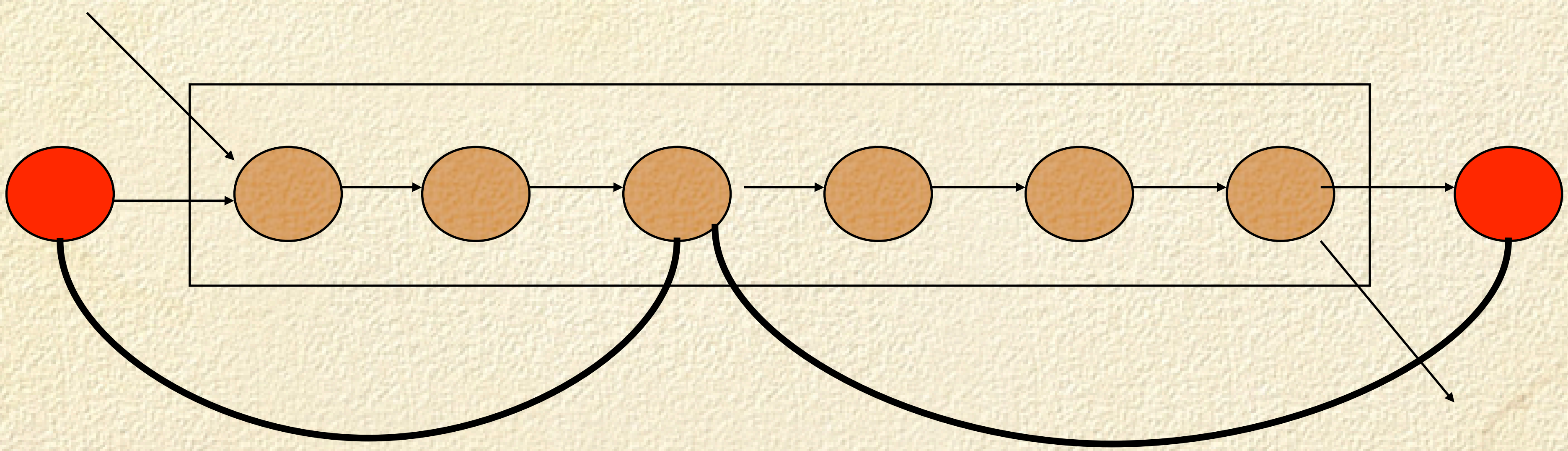
Neighborhood NA

- Swapping vertices on a critical path



Neighborhood NB

- Moving tasks in before or after a critical block



Neighborhood Exploration

```
void exploreNeighborhood(NeighborSelector N){
   exploreNA(N);
   exploreNB(N);
}
void exploreNA(NeighborSelector N) {
  forall(v in Critical) {
         int delta = obj.moveBackwardDelta(v);
         if (acceptNA(v,delta))
           neighbor(delta,N)
                  sched.moveBackward(v);
   }
}
```

Neighborhood Exploration

```
void exploreNB(NeighborSelector N) {
  forall(v in Critical) {
        int lm = sched.getLeftMostFeasible(v);
        while (lm > 1) {
          int delta = obj.moveBackwardDelta(v,lm);
          if (acceptNB(v,lm,delta)) {
                  neighbor(delta,N)
                    sched.moveBackward(v,m);
                  break;
          }
          lm--;
        }
   }
```

the yellow code is a closure created on demand

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

Experimental Results

	abz5	abz6	abz7	abz8	abz9	mt10
DT*	6.2	3.8	14.2	15.1	14.2	6.9
KS*	4.6	4.8	12.2	13.6	11.9	5.1
CO	5.9	5.7	11.7	9.9	9	6.7

Cumulative Scheduling in LS

- Of course!
  - Modeling front
    - New object:
      - CumulativeResource
  - Search front
    - Different procedure
      - iFlat-iRelax [ICAPS'04]
- Strength
  - Best algorithm for large cumulative problems.

Experimental Results

	Set A		Set B		Set MT	
Relax	Best	Avg	Best	Avg	Best	Avg
1	2.2	7.86	2.7	7.47	7.15	13.03
2	0.21	2	-0.33	1.86	2.01	6.07
4	-0.01	1.07	-1.17	0.47	0.37	3.41
6	-0.13	0.78	-1.23	-0.04	0.84	2.88