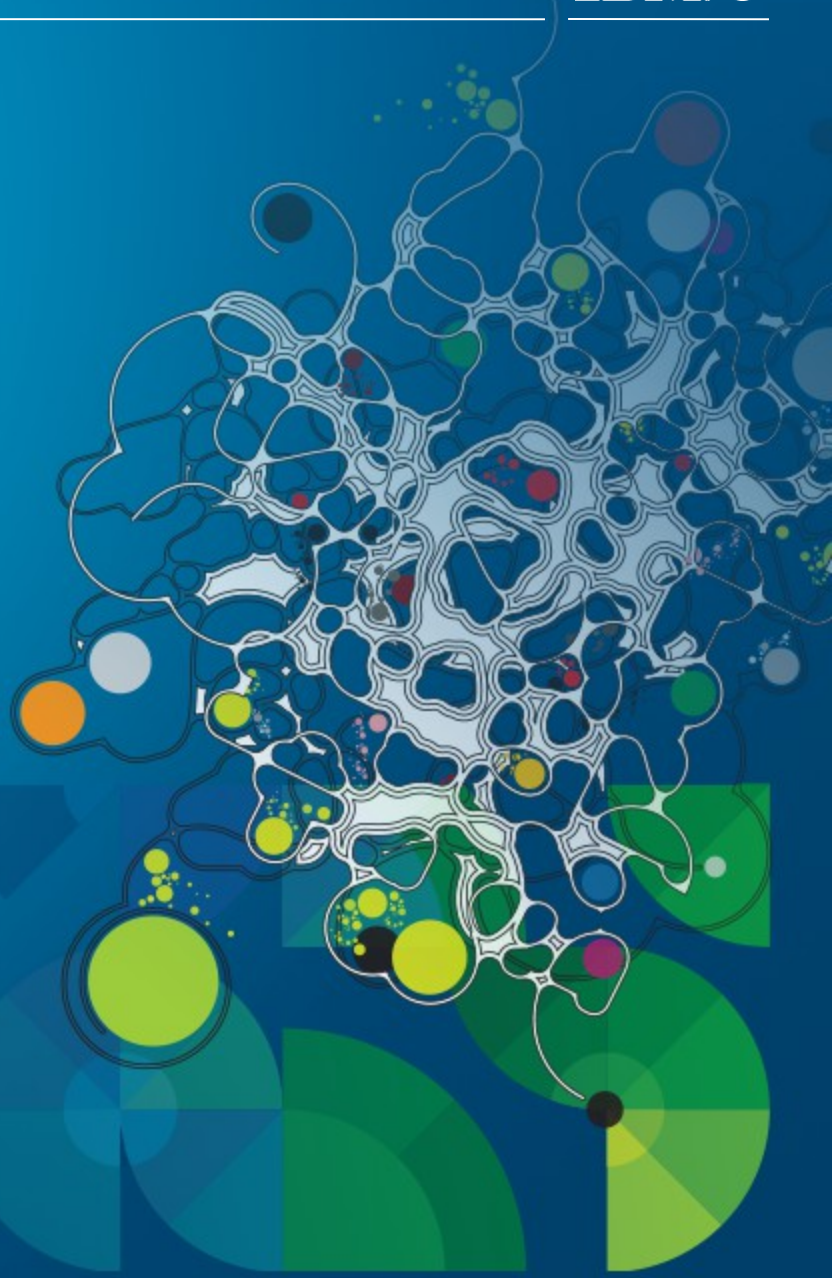
Constraint Programming in Practice

Paul Shaw



First step

- Get your hands on a good CP Solver
- I recommend **IBM CP Optimizer**
- Free for academic use
  - http://www-03.ibm.com/ibm/university/academic/pub/page/academic_initiative
- Product to download is “CPLEX Optimization Studio”
- Comes with all the CPLEX Solvers and the **OPL language** and development studio

Modelling

- As for MIP, modelling is the most important part of the process of solving a problem
- CP modelling languages and systems are generally rich compared to MIP-based ones
  - specialized constructs and constraints
- Good models will generally have stronger propagation (make stronger inferences) than poor ones
- Variables are normally finite domain
  - CP Optimizer supports floating-point *expressions*

Use the right modelling constructs

- CP solvers work best when you make use of the right modelling constructs (variables, constraints and expressions) for your problem
- You could avoid these and “roll your own” using lower-level constraints and expressions
  - But your model would be more complex and in general will be much harder to solve!
  - These specialized constructs will propagate more than a combination of simpler ones

Modelling constructs in CP Optimizer

Arithmetic

```
+   -   ×   /   div
abs   min   max
pow   log   exp
```

Logical

```
=    !=    <    <=
&&    ||    !    =>
x == v    x <= v
```

Specialized

```
all-different,
element [], count,
allowedAssignments
...
```

Scheduling

```
interval, sequence, cumul, noOverlap, alternative, ...
startAfterEnd, startAtEnd, ...
presenceOf, startOf, endOf, lengthOf, ...
```

Modelling constructs in CP Optimizer

Arithmetic	Logical	Specialized
+ - × / div	= != < <=	all-different,
abs min max	&& || ! =>	element [], count,
pow log exp	x == v x <= v	allowedAssignments
		...

Scheduling

interval, sequence, cumul, noOverlap, alternative, ...

startAfterEnd, startAtEnd, ...

presenceOf, startOf, endOf, lengthOf, ...

Modelling constructs in CP Optimizer

Arithmetic

```
+   -   ×   /   div
abs     min     max
pow     log     exp
```

Logical

```
=    !=    <    <=
&&    ||    !    =>
x == v    x <= v
```

Specialized

```
all-different,
element [], count,
allowedAssignments
...
```

Scheduling

```
interval, sequence, cumul, noOverlap, alternative, ...
startAfterEnd, startAtEnd, ...
presenceOf, startOf, endOf, lengthOf, ...
```

Modelling constructs in CP Optimizer

Arithmetic

```
+   -   ×   /   div
abs   min   max
pow   log   exp
```

Logical

```
=    !=    <    <=
&&    ||    !    =>
x == v    x <= v
```

Specialized

```
all-different,
element [], count,
allowedAssignments
...
```

Scheduling

```
interval, sequence, cumul, noOverlap, alternative, ...
startAfterEnd, startAtEnd, ...
presenceOf, startOf, endOf, lengthOf, ...
```

Modelling constructs in CP Optimizer

Arithmetic

```
+   -   ×   /   div
abs   min   max
pow   log   exp
```

Logical

```
=    !=    <    <=
&&    ||    !    =>
x == v    x <= v
```

Specialized

```
all-different,
element [], count,
allowedAssignments
...
```

Scheduling

```
interval, sequence, cumul, noOverlap, alternative, ...
startAfterEnd, startAtEnd, ...
presenceOf, startOf, endOf, lengthOf, ...
```

Use the right modelling constructs

Example: Element expression

Mini model

y should take a value equal to that of the *k*'th prime

y and *k* are integer decision variables

Right

```
dvar int k in 0..9;
dvar int y;
int a[0..9]=[2,3,5,7,11,13,17,19,23,29];
y == a[k];
```

Propagation

There is complete propagation between *y* and *k*. The initial domain of *y* will be deduced as exactly the first 10 primes.

Later, if we deduce that $k \neq 3$, then 7 will be removed from the domain of *y*. Likewise, if we deduce that $y \neq 5$, then 2 will be removed from the domain of *k*.

Use the right modelling constructs

Example: Element expression

Mini model

y should take a value equal to that of the *k*'th prime

y and *k* are integer decision variables

Wrong

```
dvar int k in 0..9;
dvar int y;
int a[0..9]=[2,3,5,7,11,13,17,19,23,29];
y == sum (j in 0..9) a[j] * (k == j);
```

Propagation

There is incomplete propagation between *y* and *k*. There are two reasons for this.

1) The solver does not see that exactly one term of the sum must be non-zero, so the initial bounds of *y* will be [0, 129] (129 is the sum of all elements of *a*)

2) Invariably, CP solvers propagate only *bounds* over sum expressions, so if we deduce, say, $y \neq 7$, this will have no effect on the domain of *k*

Use the right modelling constructs

Example: Generalized assignment

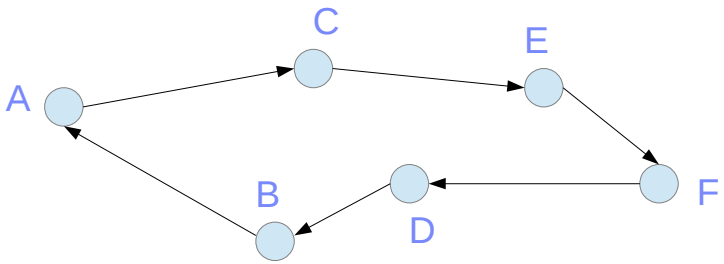
```
int N = …; // Number of objects
int M = …; // Number of agents

int ac[i][j] = …; // Assignment cost

// x[i] = j means object i is assigned to agent j
dvar int x[1..N] in 1..M;

dexpr int obj1 = sum (i in 1..N) ac[i][x[j]];

dexpr int obj2 = sum (j in 1..M)
                    sum(i in 1..N)
                       ac[i][j] * (x[i] == j);
```

Use the right modelling constructs

Example: Jobshop

```
int nbJobs = ...;
int nbMchs = ...;

range Jobs = 0..nbJobs-1;
range Mchs = 0..nbMchs-1;

tuple Operation {
  int mch; // Machine
  int pt;  // Processing time
};

Operation Ops[j in Jobs][m in Mchs] = ...;

dvar interval itvs[j in Jobs][o in Mchs] size Ops[j][o].pt;
dvar sequence mchs[m in Mchs] in all(j in Jobs, o in Mchs : Ops[j][o].mch == m) itvs[j][o];

minimize max (j in Jobs) endOf(itvs[j][nbMchs-1]);

subject to {
  forall (m in Mchs)
    noOverlap(mchs[m]);
  forall (j in Jobs, o in 0..nbMchs-2)
    endBeforeStart(itvs[j][o], itvs[j][o+1]);
}
```

Example: Travelling salesman



city

A	C	E	F	D	B
0	1	2	3	4	5

(map the position to the city)

next

C	A	E	B	F	D
A	B	C	D	E	F

(map a city to the next city)

A First Model

(This is a complete model for the TSP)

```
int N = …;
range Position = 0..N-1;
range City = 0..N-1;
int dist[City][City] = …;

dvar int city[Position] in City; // Map position to city
dvar int next[City] in City;     // Map city to next city

minimize sum (i in City) dist[i][next[i]];

constraints {
  allDifferent(city);
  forall (j in Position)
    next[city[j]] == city[(j+1) % N];
}
```

Boost Propagation

Most solvers (including CP Optimizer) allow you to control how much effort is spent finding extra inferences.

```
int N = …;
range Position = 0..N-1;
range City = 0..N-1;
int dist[City][City] = …;

dvar int city[Position] in City; // Map position to city
dvar int next[City] in City;     // Map city to next city

execute {
  cp.param.DefaultInferenceLevel = "Extended";
}

minimize sum (i in City) dist[i][next[i]];

constraints {
  allDifferent(city);
  forall (j in Position)
    next[city[j]] == city[(j+1) % N];
}
```

Find implied constraints

Make *explicit* an implicit property of solutions

Since in a TSP a solution is a loop, all the variables in the "next" array must have different values.

```
int N = …;
range Position = 0..N-1;
range City = 0..N-1;
int dist[City][City] = …;

dvar int city[Position] in City; // Map position to city
dvar int next[City] in City;     // Map city to next city

execute {
  cp.param.DefaultInferenceLevel = "Extended";
}

minimize sum (i in City) dist[i][next[i]];

constraints {
  allDifferent(city);
  allDifferent(next);
  forall (j in Position)
    next[city[j]] == city[(j+1) % N];
}
```

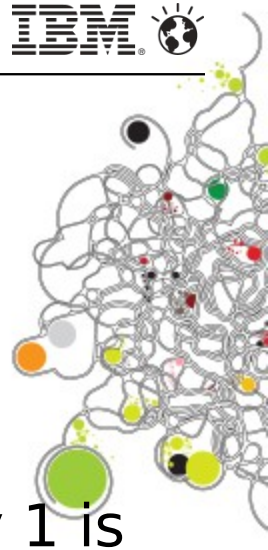
Be aware of symmetry

There is rotational symmetry on the “city” variables. Any solution can be “rotated” into an equivalent one. Equivalently, choose one city (here, city zero) as the “home city”.

```
int N = …;
range Position = 0..N-1;
range City = 0..N-1;
int dist[City][City] = …;

dvar int city[Position] in City; // Map position to city
dvar int next[City] in City;     // Map city to next city

execute {
  cp.param.DefaultInferenceLevel = “Extended”;
}

minimize sum (i in City) dist[i][next[i]];

constraints {
  allDifferent(city);
  allDifferent(next);
  forall (j in Position)
    next[city[j]] == city[(j+1) % N];
  city[0] == 0;
}
```

Extending the model

Adding precedence constraints



- Imagine we would like to be able to say that city 1 is visited before city 2
  - *i.e.* City 1 appears between the home city and city 2
- Modelling this quite cumbersome:

```
forall (j in Position)
  (city[j] == 2) => (or (k in 0..j-1) (city[k] == 1));
```

- For each position, if city 2 is in that position, then city 1 must be in an earlier position
  - We need all these constraints for each precedence!
- We would really like to know the position of a city

An alternative view (“dual model”)

Create “position” variables and connect these to the “city” variables using *channeling constraints* of the form

`position[city[j]] == j`

```
int N = …;
range Position = 0..N-1;
range City = 0..N-1;
int dist[City][City] = …;

dvar int city[Position] in City;     // Map position to city
dvar int next[City] in City;         // Map city to next city
dvar int position[City] in Position; // Map city to its position

execute {
  cp.param.DefaultInferenceLevel = “Extended”;
}

minimize sum (i in City) dist[i][next[i]];

constraints {
  allDifferent(city);
  allDifferent(next);
  forall (j in Position) {
    next[city[j]] == city[(j+1) % N];
    position[city[j]] == j;
  }
  city[0] == 0;
  position[1] < position[2];
}
```

An alternative view (“dual model”)

Create “position” variables and connect these to the “city” variables using *channeling constraints* of the form

`position[city[j]] == j`

```
int N = …;
range Position = 0..N-1;
range City = 0..N-1;
int dist[City][City] = …;

dvar int city[Position] in City;     // Map position to city
dvar int next[City] in City;         // Map city to next city
dvar int position[City] in Position; // Map city to its position

execute {
  cp.param.DefaultInferenceLevel = “Extended”;
}

minimize sum (i in City) dist[i][next[i]];

constraints {
  allDifferent(city);
  allDifferent(next);
  inverse(city, position);
  forall (j in Position)
    next[city[j]] == city[(j+1) % N];
  city[0] == 0;
  position[1] < position[2];
}
```

Modelling (some good practices)

- Good models have the solver make *strong inferences*
- Use the *right constructs.* Look to see if some specialized (global) constraints fit what you want to do
  - Better propagation
  - Compact models
- Experiment with inference strength
- Look for implied constraints
- Be aware of symmetry
- Look for alternative views to make modelling simpler

Solving

- Once a model is built, instances (instantiated models) must be solved. Unlike MIP solvers, traditionally CP solvers provide only a “search toolkit” to the user so that they can control solution search
- CP Optimizer was the first CP solver with an intelligent *automatic* search process
- Numerous techniques which have been traditionally coded by users to solve problems using classical solvers are *implemented inside* the CP Optimizer automatic search

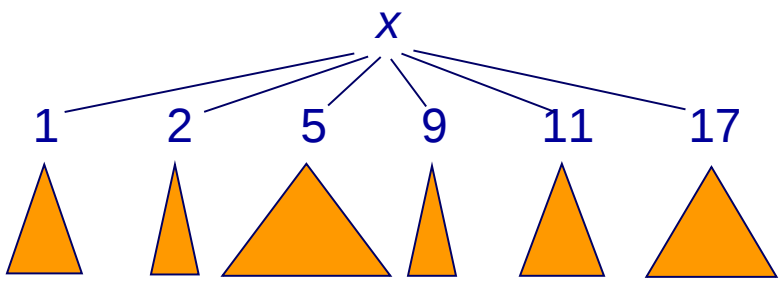
Tree Search: Branching Heuristics

- CP solvers are essentially constructed to perform *depth-first* search
- Decisions are taken at each level of the search tree:
  - Which element of the model should be branched upon (normally called a *variable selection* rule)
  - Which branch should be followed first (normally called a *value selection* rule)
- CP solvers offer users open APIs to define domain-specific heuristics to help get to good solutions
- Domain-independent heuristics also exist where no intuition or domain-specific heuristics are available

Tree Search: Generic Heuristics

- Variable Selection
  - Smallest domain (“first failure”) [Haralick & Elliot]
  - Largest degree (involved in the most constraints)
  - Combinations of the above (*e.g.* Brelaz, max. degree / domain)
  - Constrainedness (“Kappa”) [Gent *et al.*]
  - Impacts [Refalo]
  - Randomized
- Value Selection
  - “Promise” measure [Geelen]
  - Constrainedness (“Kappa”) [Gent *et al.*]
  - Impacts [Refalo]
  - Randomized

Tree Search: Impact-based heuristic



Estimation for an instantiation

Let S be the product of domain sizes. The estimation of the search space size below $x = a$ is

$$e(x = a) \simeq S \times (1 - I(x = a))$$

Estimation for a variable

We assume that all values in the domain D_x will be tried. The estimation is:

$$E(x) = \sum_{a \in D_x} e(x = a)$$

Tree Search: Specific Heuristics

- Specific heuristics often mean some programming has to be done. Implemented in CP Optimizer using programming APIs based on either:
  - Callbacks written by the user which dynamically determine the variable and value selection rules to be followed at any point during the tree search
  - Full control using a backtracking system (“goals”) in a Prolog style. Closures are the primitive objects to control the search. Roughly speaking, a closure is called which gives back a combination of the branching rule to follow, together with a (recursive) specification of what to do after that point in the form of another closure.

Tree Search: Giving Hints

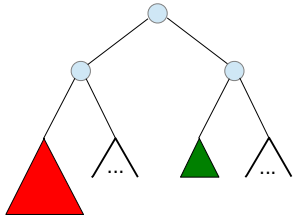
- Often, you can have some basic intuition about a problem, but perhaps not to any depth
  - Prefer the solver to handle the details
- For the TSP, perhaps you think it is best to branch on the “next” variables:

```
var f = cp.factory;
cp.setSearchPhases(f.searchPhase(next));
```

- This kind of high level information is often effective
- Finer specification of variable and value selection rules is also possible without actually programming. *e.g.*

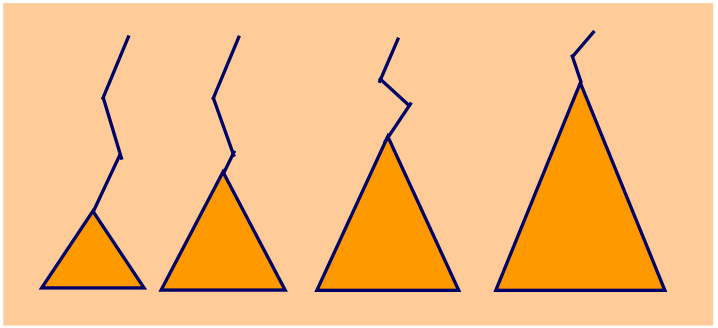
```
f.searchPhase(x, f.selectSmallest(f.domainSize()),
                 f.selectLargest(value()));
```

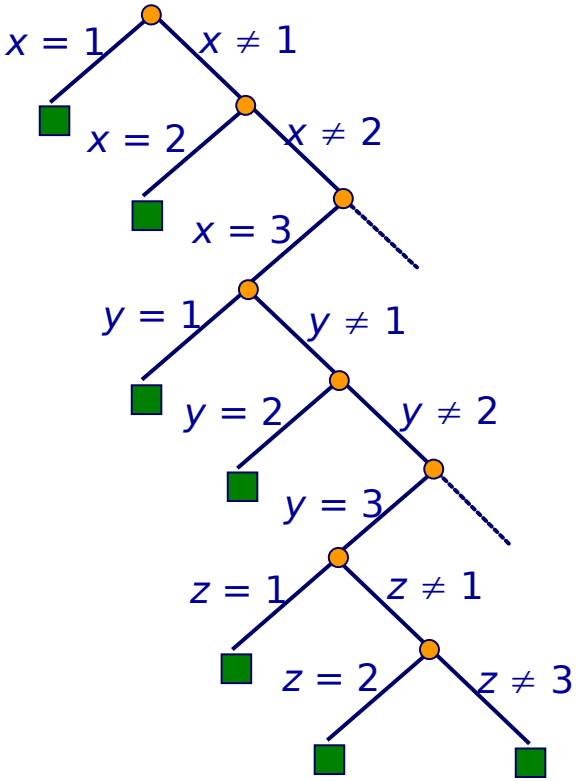
Tree Search: Strategies



- Even if heuristics help search go to good solutions, depth first search (DFS) can be problematic
  - If heuristics make “high up” mistakes, DFS takes a long time to correct these mistakes
- CP Solvers including CP Optimizer often support different strategies and you can use the general search mechanisms to build your own strategy
  - It is quite easy to build a discrepancy-based method
- CP Optimizer has built in:
  - Depth-first, restarts, multi-point (evolutionary), large neighbourhood search

Tree Search: Restarts





Clause Generation

$$\begin{cases} x \neq 1, \\ x \neq 2, \\ x \neq 3 \vee y \neq 1, \\ x \neq 3 \vee y \neq 2, \\ x \neq 3 \vee y \neq 3 \vee z \neq 1, \\ x \neq 3 \vee y \neq 3 \vee z \neq 2, \\ x \neq 3 \vee y \neq 3 \vee z \neq 3, \end{cases}$$

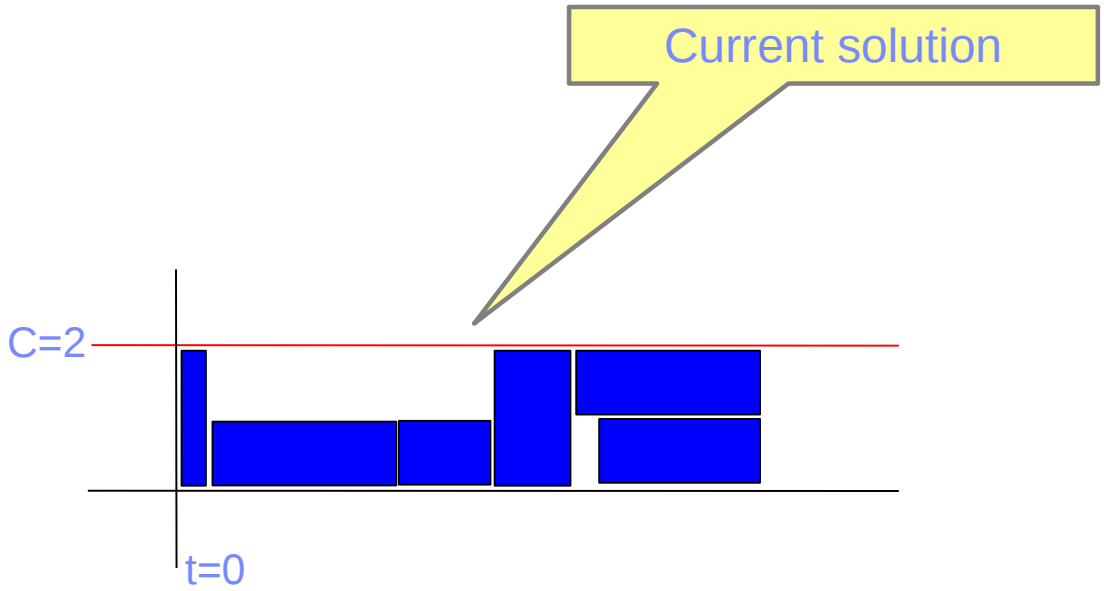
Restart

Increase limit

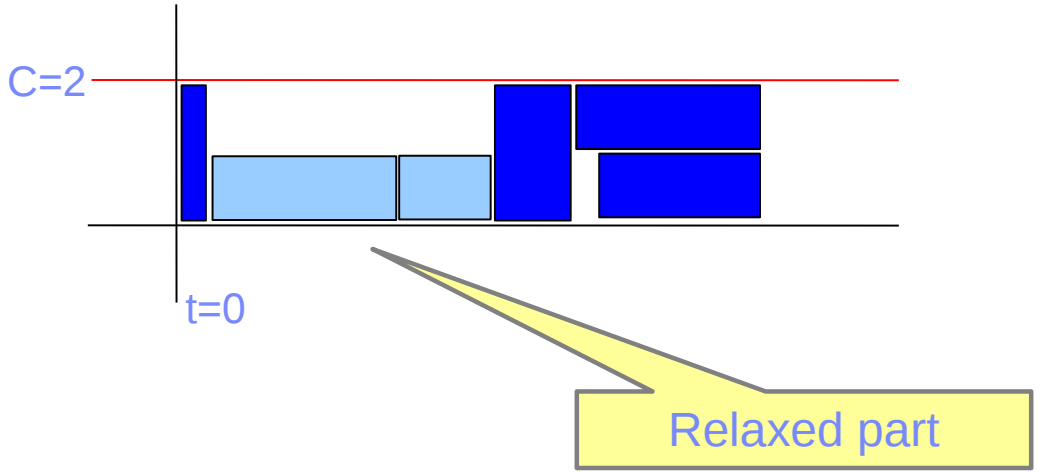
Hybrid Methods: Large Neighbourhood Search

- In scheduling, a common way of *improving* solutions is to use Large Neighborhood Search
  - Depends on the notion of an incumbent solution
- Some operations are “relaxed” - they can move freely while still obeying problem constraints
- The remaining operations stay “rigid” which means that they can shift in time but stay in much the same order as in the current solution
- The start times of the free and rigid parts are then decided by a search using heuristics. The aim is to find a solution of lower objective value

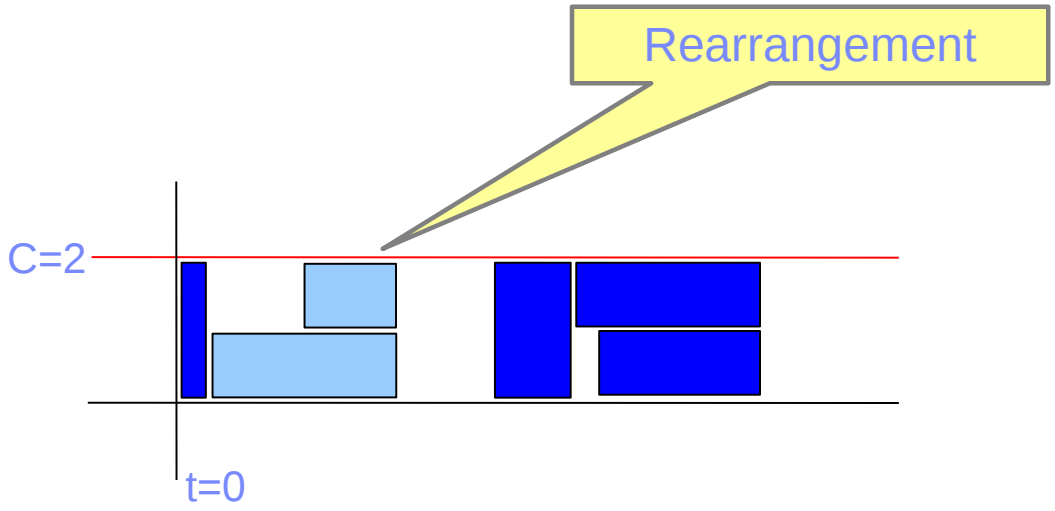
Hybrid Methods: Large Neighbourhood Search



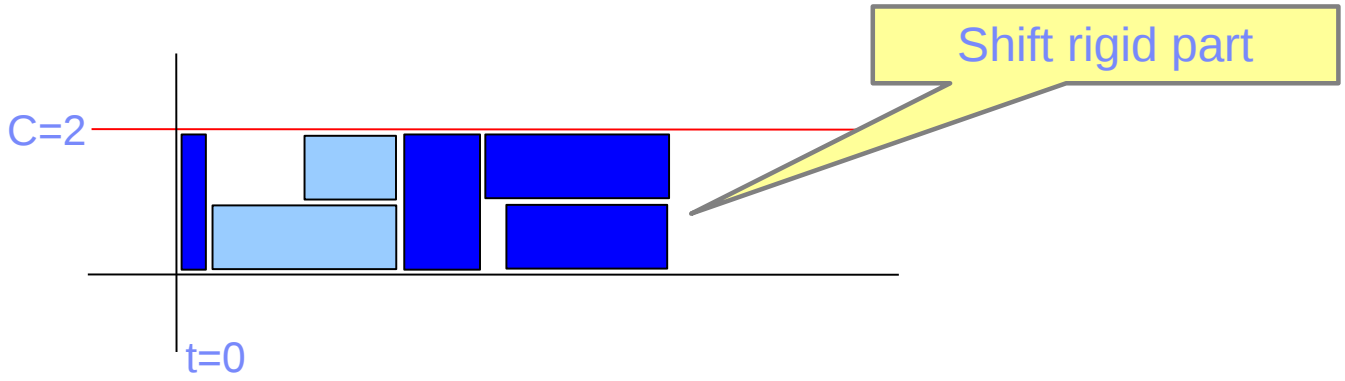
Hybrid Methods: Large Neighbourhood Search



Hybrid Methods: Large Neighbourhood Search



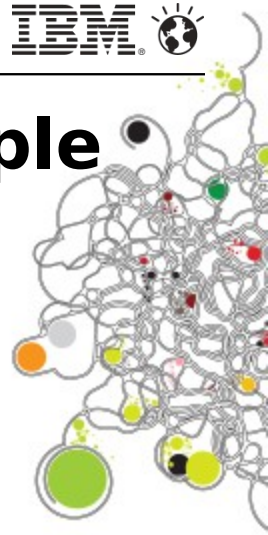
Hybrid Methods: Large Neighbourhood Search



Custom Constraints

- Most CP solvers allow the users to write *custom constraints* with whatever semantics they wish
- CP Optimizer uses an event-based mechanism
- The custom constraint is alerted when the values are removed from a variable domain
  - It then removes values from domains of other vars
- Example:
  - In the TSP example, a custom constraint could dynamically solve a MST problem based on the current domains of the “next” variables
  - Forms a lower bound on the objective function

Custom Constraints: CP Optimizer example

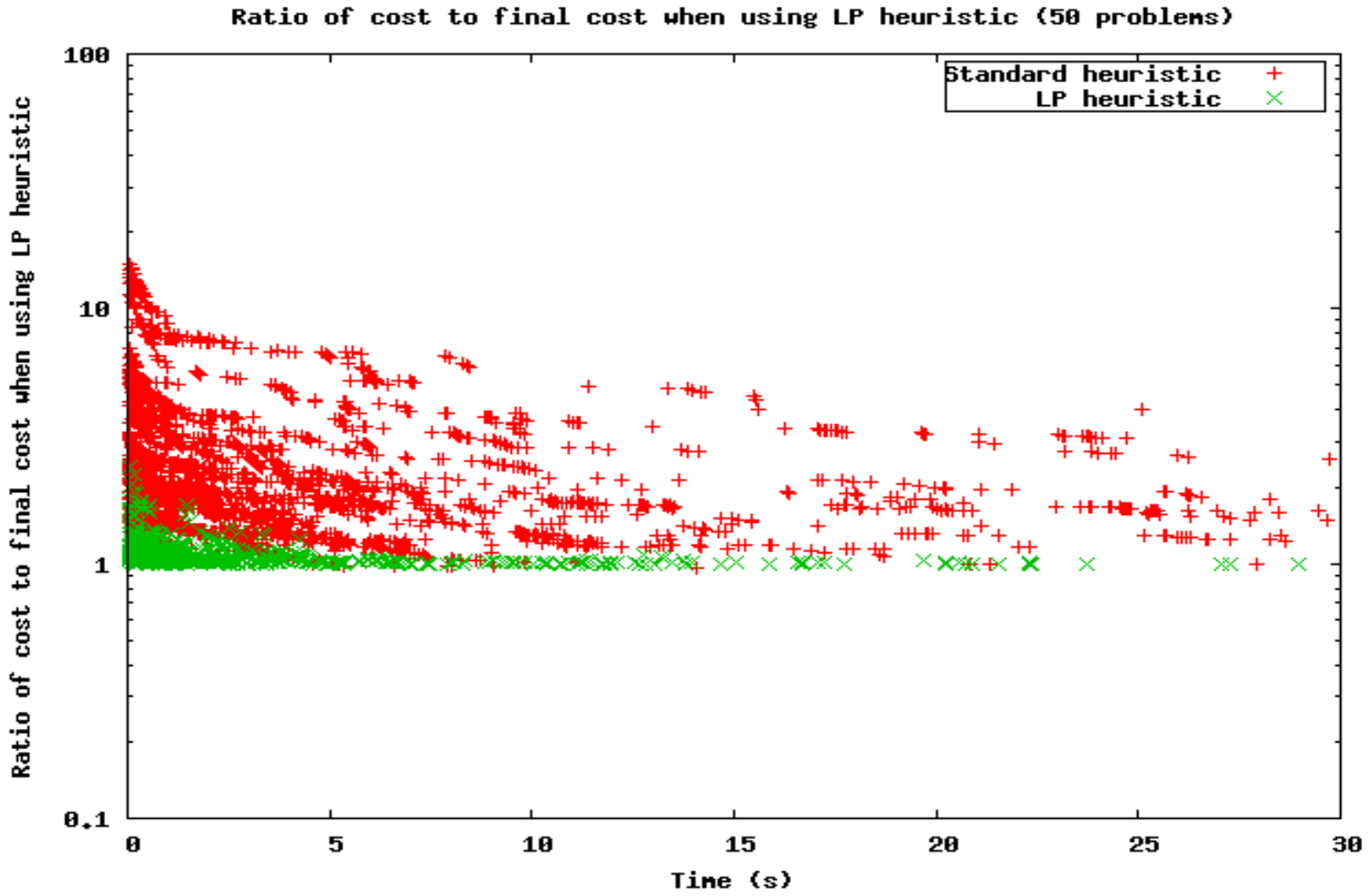


```
IloCP cp = ...;
ILCDEMON1(runMST, MSTObject *, mst) {
      // Code to calculate and impose lower bound
}
...
IlcDemon runMSTDemon = runMST(cp, myMSTObject);
for (int j = 0; j < N; j++)
      myMSTObject->next[i].whenDomain(runMST);
```

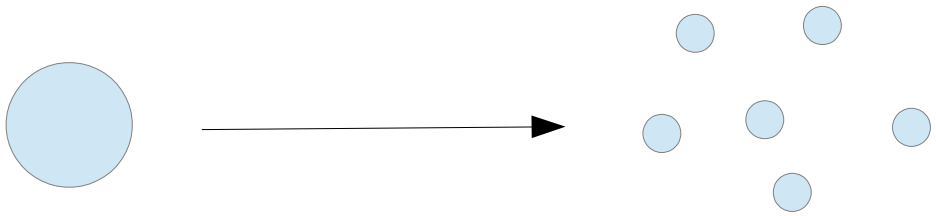
Hybrid Methods: Using an LP Solver

- It may be possible to linearize a part of a CP model and run an LP on that part to
  - provide bounds on the objective
  - provide information to the branching heuristic (choice of value)
- [Beck and Refalo] report good improvements on early/tardy cost problems
- Implemented in the CP Optimizer automatic search
  - Linearize precedences, logical constraints, execution / non-execution costs, alternatives
  - Use PWL functions to linearize complex cost functions

Hybrid Methods: Using an LP Solver



Decomposition



- Some problems are sometimes too large to get a good solution in reasonable time
- Often, a MIP can be used to split an initial problem into a number of smaller CP problems
  - In CPLEX Studio, you get the CPLEX Optimizer & CP Optimizer together
- Examples
  - Prod. planning with CPLEX, scheduling with CPO
  - Resource assignment w/ CPLEX, scheduling w/ CPO
  - Rostering using column generation: Master solved with CPLEX, subproblem with CPO

Summary

- Branching Heuristics
  - Generic
  - Dedicated
  - Hints
- Strategies
  - Restarts
  - Large Neighbourhood Search
- Other techniques
  - Decomposition
  - Custom constraints
  - Using an LP solver

- Automatically used by CP Optimizer
- Can be implemented with CP Optimizer

Some Techniques used in CP Optimizer

- Automatic search uses a set of techniques working together
  - These typically use domain filtering and tree search as a building block, but are not limited to this
- Examples of techniques used
  - Restarting techniques
  - No-good tracking
  - Impact-based branching
  - Opportunistic probing
  - Large Neighborhood Search
  - Large Neighborhood Search
  - Evolutionary algorithms
  - Constraint aggregation
  - Dominance rules
  - Machine learning
  - LP-assisted heuristics