



# Discrete Optimization for Agents

Peter J. Stuckey and countless others!



Australian Government  
Department of Broadband, Communications  
and the Digital Economy  
Australian Research Council

## NICTA Funding and Supporting Members and Partners



# Conspirators

---



- Ignasi Abio, Ralph Becket, Sebastian Brand, Geoffrey Chu, Michael Codish, Toby Davies, Greg Duck, Nick Downing, Thibaut Feydy, Kathryn Francis, Graeme Gange, Vitaly Lagoon, Nir Lipovetzky, Nick Nethercote, Olga Ohrimenko, Adrian Pearce, Andreas Schutt, Guido Tack, Pascal Van Hentenryck, Mark Wallace
- All **errors** and **outrageous lies** are **mine**

# Outline

---



- Optimization = intelligence
  - Discrete optimization has advanced rapidly
- Solver independent modelling
  - MiniZinc: a high level modelling language
- Nogood learning for discrete optimization
  - The laziness principle in action
- Resolving similar problems
  - “lifelong learning” and nested constraint programs
- Concluding remarks

# Outline

---



- Optimization = intelligence
  - Discrete optimization has advanced rapidly
- Solver independent modelling
  - MiniZinc: a high level modelling language
- Nogood learning for discrete optimization
  - The laziness principle in action
- Resolving similar problems
  - “lifelong learning” and nested constraint programs
- Concluding remarks

# Intelligent Agents

---



- An intelligent agent can be a [Russell+Norvig]
  - simple reflex agent
  - model-based reflex agent
  - goal-based agent
  - utility-based agent
  - learning agent
- Utility-based agent
  - optimizing utility
  - goal-based agent wants to optimize time to goal
- Learning agent
  - trying to optimize learning performance

# Intelligent Agents + Discrete Optimization

---



- Models the environment
- Makes a **decision** to act
- In order to move to achieve some goal
- Has **limited resources** and **limited actions**
- **Discrete Optimization**
  - choose from a **limited** set of possibilities
  - a solution which optimizes a **utility**
  - usually subject to (complex) constraints
    - like **limited resources**

# Automated Planning

---



- Planning is a technology targeted for agents
- Mature technology for finding a solution to multi-agent propositional problems
- But
  - planning with **utility** (cost-optimal planning) technology is still quite immature
  - planning is not very good at modelling **limited resources**
    - some evidence random search better than planning when resources are scarce
  - temporal planning  $\cong$  scheduling with optional tasks

# Discrete Optimization for Agents

---



- Not as rich as planning
- But very effective at
  - making a decision
  - to optimize an objective
  - subject to limitations (constraints)
- A useful tool for building intelligent agents



# Why should you listen to this talk

---



- Discrete Optimization
  - is easier to use now
    - solver independent modelling
  - is more effective than before
    - nogood learning solvers
  - is better at “resolving” a similar problem
    - assumptions and nogood learning
  - has well defined approaches to stochasticity
    - stochastic optimization
  - can express very complicated problems succinctly
    - nested constraint programs
- Useful for other parts of your CS life

# Outline

---



- Optimization = intelligence
  - Discrete optimization has advanced rapidly
- Solver independent modelling
  - MiniZinc: a high level modelling language
- Nogood learning for discrete optimization
  - The laziness principle in action
- Resolving similar problems
  - “lifelong learning” and nested constraint programs
- Concluding remarks

# Solver independent modelling

---



- There are many approaches to solving discrete optimization problems
  - mixed integer programming (MIP)
  - local search (LS)
    - simulated annealing, tabu search, CBLS
  - population-based search (PS)
    - genetic algorithms, evolutionary algorithms, beam search
  - constraint programming (CP)
  - Boolean satisfiability (SAT)
  - SAT modulo theories (SMT)
  - Answer set programming (ASP)
- Different technologies have different strengths and weaknesses

# Solver independent modelling

---



- Building a discrete optimization solution can be a large undertaking
- Early commitment to solving technology
  - may not choose correctly
  - wastes a lot of work
  - may indeed prevent other approaches being tried
- The answer
  - capture the problem independent of solving technology



- A solver independent modelling language
- Supports
  - CP solvers: almost all except commercial ones
  - MIP solvers: CBC, Cplex, Gurobi
  - SAT solvers (by translation): fztini, picat-SAT
  - SMT solvers: fzn2smt
  - ASP solvers: minisatID
  - local search (CBLS) solvers: oscar, yacs
- De facto standard for CP modelling
- Translates a high level model to
  - a form suitable for the underlying solver

# Jobshop Scheduling



```
int: n; set of int: Job=1..n; % no of jobs
int: m; set of int: Task=1..m; % task per job
int: span; % max end time
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
            where mc[i,j] = k],
            [d[i,j] | i in Job, j in Task
            where mc[i,j] = k])));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```



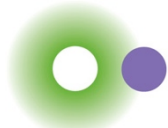
# Jobshop Scheduling



```
int: n; set of int: Job=1..n; % no of jobs
int: m; set of int: Task=1..m; % tasks
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
            where mc[i,j] = k],
            [d[i,j] | i in Job, j in Task
            where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Parameters

# Jobshop Scheduling



```
int: n; set of int: Job=1..n; % no
int: m; set of int: Task=1..m; % task
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
        where mc[i,j] = k],
        [d[i,j] | i in Job, j in Task
        where mc[i,j] = k])));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```



# Jobshop Scheduling

```
int: n; set of int: Job=1..n; % no
int: m; set of int: Task=1..m; % task
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
        where mc[i,j] = k],
        [d[i,j] | i in Job, j in Task
        where mc[i,j] = k])));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

Dependent

Parameters

Variables

# Jobshop Scheduling

```
int: n; set of int: Job=1..n;
int: m; set of int: Task=1..m;
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
            where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
            where mc[i,j] = k])));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

**Dependent**

**Parameters**

**Variables**

**Comprehensions**

# Jobshop Scheduling

```
int: n; set of int: Job=1..n; % no
int: m; set of int: Task=1..m; % task
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
            where mc[i,j] = k],
            [d[i,j] | i in Job, j in Task
            where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

**Dependent**

**Parameters**

**Variables**

**Constraints**

**Comprehensions**

# Jobshop Scheduling

```
int: n; set of int: Job=1..n;
int: m; set of int: Task=1..m;
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
            where mc[i,j] = k],
           [d[i,j] | i in Job, j in Task
            where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

**Dependent**

**Parameters**

**Variables**

**Global**

**Constraints**

**Comprehensions**

# Jobshop Scheduling

```
int: n; set of int: Job=1..n; % jobs
int: m; set of int: Task=1..m; % tasks
int: span;
array[Job,Task] of int: d;
array[Job,Task] of Task: mc;
array[Job,Task] of var 0..span: s;
constraint forall(i in Job, j in 1..m-1)
    (s[i,j] + d[i,j] <= s[i,j+1]);
constraint forall(k in Task)
    (unary([s[i,j] | i in Job, j in Task
            where mc[i,j] = k],
            [s[i,j] | i in Job, j in Task
            where mc[i,j] = k]));
var int: obj = max([s[i,m] + d[i,m] | i in Job]);
solve minimize obj;
```

**Dependent**

**Parameters**

**Variables**

**Global**

**Constraints**

**Objective**

**Comprehensions**

# An Agent Example

---



- Bulk rail scheduling
  - A number of services are demanded
  - Service: move cargo from start to end
  - Time windows for start and end
  - Payment for completion

```
int: nservice;  
set of int: SERVICE = 1..nservice;  
array[SERVICE] of NODE: start;  
array[SERVICE] of TIME: earliest_start;  
array[SERVICE] of TIME: latest_start;  
array[SERVICE] of NODE: end;  
array[SERVICE] of TIME: earliest_end;  
array[SERVICE] of TIME: latest_end;  
array[SERVICE] of int: payment;
```

# An Agent Example

---



- Rail track network
  - Nodes, and (directed) edges
  - Cost to use edge dependent on time of use
  - Travel time for edge

```
enum NODE;  
int: nedge;  
set of int: EDGE = 1..nedge;  
array[EDGE,1..2] of NODE: edge;  
int: horizon; % time horizon in minutes  
set of int: TIME = 0..horizon;  
set of int: HOUR = 0..horizon div 60;  
array[EDGE,HOUR] of int: cost;  
set of int: DUR = 0..horizon;  
array[EDGE] of DUR: travel; % travel time for edge
```

# An Agent Example

---



- Consist decisions
  - which service to undertake
  - path and travel times
  - waiting (dwelling) at nodes

```
var SERVICE: which;  
var TIME: start_time;  
array[NODE] of var TIME: arrive;  
int: maxdwell;  
set of int: DWELL = 0..maxdwell;  
array[NODE] of var DWELL: dwell;  
array[NODE] of var NODE: next;% next node to visit or self  
set of int: EDGE0 = 0..nedge;  
array[NODE] of var EDGE0: route; % edge taken or 0
```



# An Agent Example

---



- Constraints: meets time criteria

```
start_time = arrive[start[which]] /\
dwell[start[which]] = 0 /\
arrive[start[which]] >= earliest_start[which] /\
arrive[start[which]] <= latest_start[which] /\
arrive[end[which]] >= earliest_end[which] /\
arrive[end[which]] <= latest_end[which];
```

- Constraints: correct path

```
path(start[which], end[which], next);
```

- Global constraints

- are translated differently for each solver
- encapsulate common combinatorial subproblems
- ~150 supported by MiniZinc
- ~420 in global constraint catalog

# An Agent Example



- Constraints: determining route and times

```
forall(n in NODE) (  
    if next[n] = n then  
        route[n] = 0 /\ arrive[n] = horizon /\ dwell[n] = 0  
    elseif n = end[which] then  
        route[n] = 0 /\ dwell[n] = 0  
    else edge[route[n],1]=n /\ edge[route[n],2]=next[n] /\  
        arrive[next[n]] = arrive[n]+dwell[n]+travel[route[n]]  
    endif);
```

- Objective: maximize profit

```
solve maximize payment[which] -  
    sum(n in NODE)  
        (if route[n] = 0 then 0 else  
        cost[route[n], (arrive[n] + dwell[n]) div 60]  
        endif);
```

# Bulk Rail Scheduling in reality

---



- Multiple consists
  - no overlap in track usage (unary resources)
- Multiple services
  - plan route from service to next service
- Crewing + Maintenance constraints
  - visit yard, enforced dwell times
- Dependent travel times
  - loaded or unloaded
- Load and unload times

# MiniZinc

---



- Information at [www.minizinc.org](http://www.minizinc.org)
- Download system and interactive development environment
  - comes with a number of solvers
- Tutorial
  - how to use MiniZinc
- Documentation
  - ref manual, global constraints library
- Coursera: Modeling discrete optimization
  - 8 week course using MiniZinc for modelling

# Embedding a MiniZinc model

---



- Piping text files (?)
  - well it works ...
- Link directly to libminizinc (C++)
  - construct data using API
  - receive solutions using API
  - release imminent (available at [github.com/MiniZinc](https://github.com/MiniZinc))
- Use Python interface to libminizinc
  - release imminent (available at [github.com/MiniZinc](https://github.com/MiniZinc))
- Use JSON interface (for web applications)
  - pass data as JSON
  - receive solutions as JSON
  - release date not fixed yet

# Advantages of Using MiniZinc

---



- Rapid creation of high level models
- No commitment to solver technology
- Many solvers to try
  - different CP solvers with different default search
- Open source
- OPL: IBM modelling language product
  - similar to MiniZinc
- Only usable with IBMs CP and MIP solver
  - CP Optimizer
  - CPLEX

# Outline

---



- Optimization = intelligence
  - Discrete optimization has advanced rapidly
- Solver independent modelling
  - MiniZinc: a high level modelling language
- Nogood learning for discrete optimization
  - The laziness principle in action
- Resolving similar problems
  - “lifelong learning” and nested constraint programs
- Concluding remarks

# Nogood learning for Optimization

---



- **Nogood** = set of decisions leading to no solution
- Boolean satisfiability (SAT) technology for nogood learning
  - drastically increased the size of SAT models solvable
- **Steal** this technology for propagation based solving (CP)
  - discrete optimization problems are not like SAT problems!



# Nogood learning for Optimizaton

---



- Outline
  - Brief example of CP solving
  - Lazy clause generation
    - a CP and SAT hybrid
  - Boasting about effectiveness

# Propagation Solving (CP)



- Complete solver for **atomic** constraints
  - $x = d, x \neq d, x \geq d, x \leq d$
  - Domain  $D(x)$  records the result of solving (!)
- Constraints implemented by propagators  
**Propagators** infer new atomic constraints from old ones
  - $x_2 \leq x_5$  infers from  $x_2 \geq 2$  that  $x_5 \geq 2$
  - $x_1 + x_2 + x_3 + x_4 \leq 9$  from  $x_1 \geq 1 \wedge x_2 \geq 2 \wedge x_3 \geq 3$  that  $x_4 \leq 3$
- Inference is interleaved with search
  - Try adding  $c$  if that fails add *not*  $c$
- Optimization is repeated solving
  - Find solution  $obj = k$  resolve with  $obj < k$

# Finite Domain Propagation Ex.



```

array[1..5] of var 1..4: x;
constraint alldifferent(x[1],x[2],x[3],x[4]);
constraint x[2] <= x[5];
constraint x[1] + x[2] + x[3] + x[4] <= 9;
    
```

	$x_1=1$	<i>alldiff</i>	$x_2 \leq x_5$	$x_5 > 2$	$x_2 \leq x_5$	<i>alldiff</i>	$sum \leq 9$	<i>alldiff</i>
$x_1$	1	1	1	1	1	1	1	1
$x_2$	1..4	2..4	2..4	2..4	2	2	2	2
$x_3$	1..4	2..4	2..4	2..4	2..4	3..4	3	✗
$x_4$	1..4	2..4	2..4	2..4	2..4	3..4	3	✗
$x_5$	1..4	1..4	2..4	3..4	2	2	2	2

# FD propagation

---



- **Strengths**
  - High level modelling
  - Specialized global propagators capture substructure
    - and all work together
  - Programmable search
- **Weaknesses**
  - Weak autonomous search (improved recently)
  - Optimization by repeated satisfaction
  - Small models can be intractable

# Lazy Clause Generation (LCG)

---



- A hybrid SAT and CP solving approach
- Add **explanation** and **nogood learning** to a propagation based solver
- Key change
  - Modify propagators to explain their inferences as clauses
  - Propagate these clauses to build up an implication graph
  - Use SAT conflict resolution on the implication graph

# LCG in a Nutshell



- Integer variable  $x$  in  $l..u$  encoded as **Booleans**
  - $[x \leq d]$ ,  $d$  in  $l..u-1$
  - $[x = d]$ ,  $d$  in  $l..u$
- **Dual** representation of domain  $D(x)$
- Restrict to **atomic changes** in domain (literals)
  - $x \leq d$  (itself)
  - $x \geq d$  !  $[x \leq d-1]$  use  $[x \geq d]$  as shorthand
  - $x = d$  (itself)
  - $x \neq d$  !  $[x = d]$  use  $[x \neq d]$  as shorthand
- Clauses DOM to model relationship of Booleans
  - $[x \leq d] \rightarrow [x \leq d+1]$ ,  $d$  in  $l..u-2$
  - $[x = d] \Leftrightarrow [x \leq d] \wedge ! [x \leq d-1]$ ,  $d$  in  $l+1..u-1$

# LCG in a Nutshell

---



- Propagation is clause generation
  - e.g.  $[x \leq 2]$  and  $x \geq y$  means that  $[y \leq 2]$
  - clause  $[x \leq 2] \rightarrow [y \leq 2]$
- Consider
  - `alldifferent([x[1], x[2], x[3], x[4]]) ;`
- Setting  $x_1 = 1$  we generate new inferences
  - $x_2 \neq 1, x_3 \neq 1, x_4 \neq 1$
- Add clauses
  - $[x_1 = 1] \rightarrow [x_2 \neq 1], [x_1 = 1] \rightarrow [x_3 \neq 1], [x_1 = 1] \rightarrow [x_4 \neq 1]$
  - i.e.  $![x_1 = 1] \vee ![x_2 = 1], \dots$
- Propagate these new clauses

# Lazy Clause Generation Ex.

*alldiff*

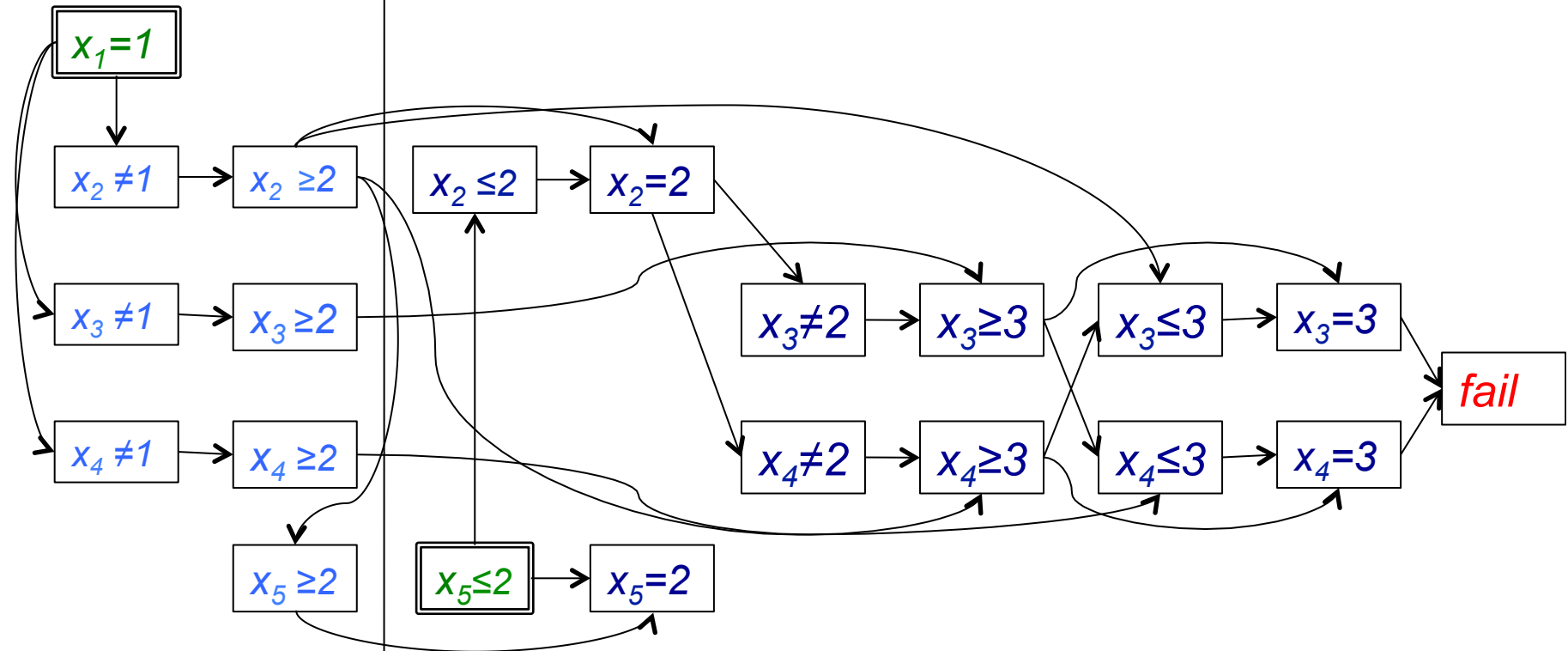
$x_2 \leq x_5$

$x_2 \leq x_5$

*alldiff*

$sum \leq 9$

*alldiff*





# 1UIP Nogood Creation

*alldiff*

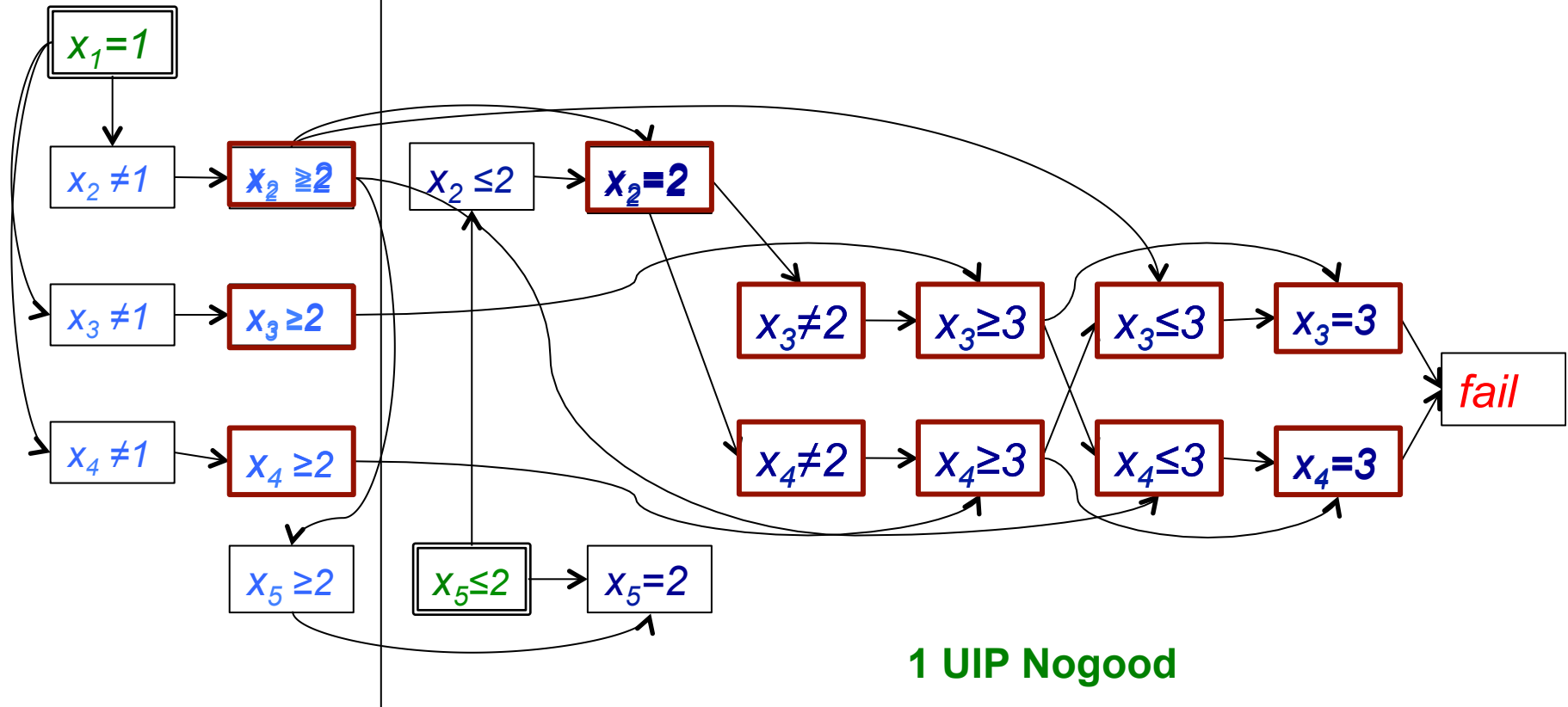
$x_2 \leq x_5$

$x_2 \leq x_5$

*alldiff*

$sum \leq 9$

*alldiff*



1 UIP Nogood

$\{[x_2 \leq 1], [x_3 \leq 1], [x_4 \leq 1], \neg[x_2 = 2]\}$

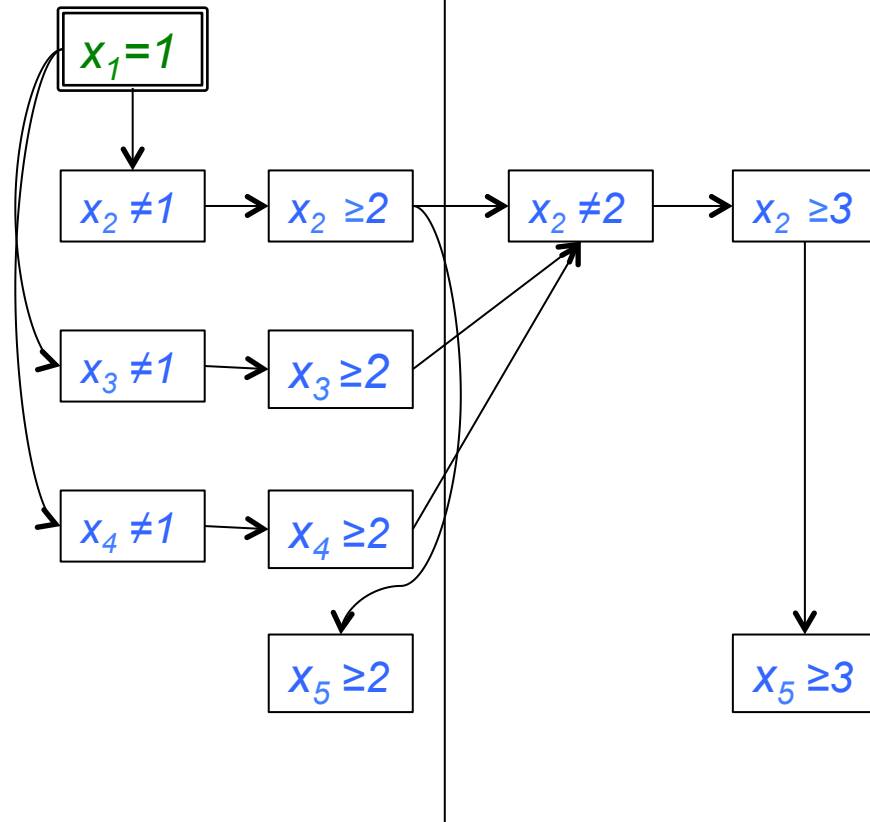
$\{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_2 = 2\} \rightarrow \text{false}$

# Backjumping



*alldiff*

$$x_2 \leq x_5$$



$$x_2 \leq x_5$$

- Backtrack to **second last** level in nogood
- Nogood will propagate
- Note **stronger** domain than usual backtracking
  - $D(x_2) = \{3..4\}$

$\{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_2 = 2\} \rightarrow \text{false}$

# What's Really Happening

---



- CP model = **high level** “Boolean” model
- Clausal representation of the Boolean model is generated “**as we go**”
- All generated clauses are **redundant** and can be removed at any time
- We can **control the size** of the active “Boolean” model

# Comparing to SAT



- For some models we can generate all possible explanation clauses before commencement
  - usually this is too big
- Open Shop Scheduling (tai benchmark suite)
  - averages

	Time	Solve only	Fails	Max Clauses
SAT	318	89	3597	13.17
LCG	62		6651	1.0

# Lazy Explanation

---

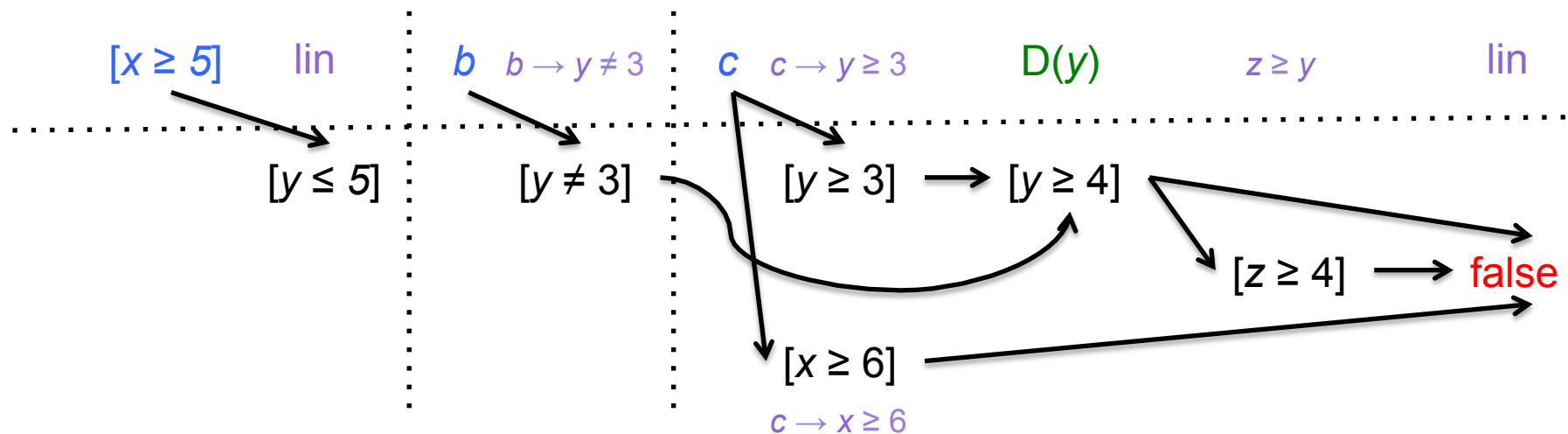


- Explanations only needed for nogood learning
  - Forward: record propagator causing atomic constraint
  - Backward: ask propagator to explain the constraint
- Only create **needed explanations**
- Scope for:
  - Explaining a **more general failure** than occurred
  - Making use of the **current nogood** in choosing an explanation

# (Original) LCG propagation example



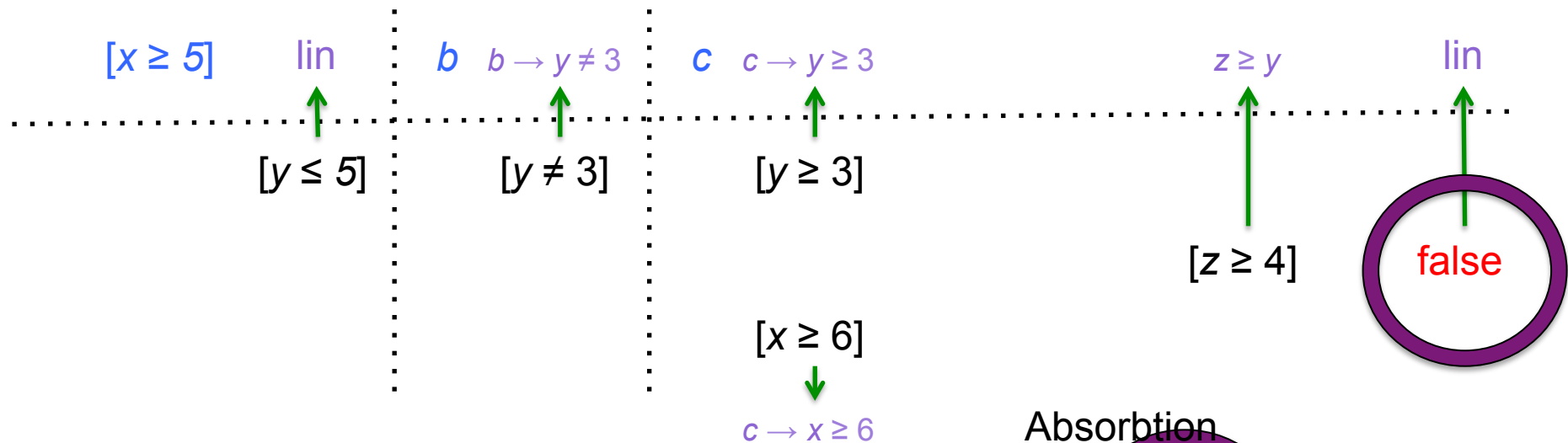
- Variables:  $\{x, y, z\}$   $D(v) = [0..6]$  Booleans  $b, c$
- Constraints:
  - $z \geq y, b \rightarrow y \neq 3, c \rightarrow y \geq 3, c \rightarrow x \geq 6,$
  - $4x + 10y + 5z \leq 71$  (lin)
- Execution



1UIP nogood:  $c \wedge [y \neq 3] \rightarrow \text{false}$  or  $[y \neq 3] \rightarrow !c$

# LCG propagation example

- Execution



**Explanation:**  $x \geq 6 \wedge \neg(y \geq 5) \wedge \neg(z \geq 4) \wedge [x \geq 5] \wedge [y \geq 4] \wedge [y \geq 3] \rightarrow \text{false}$

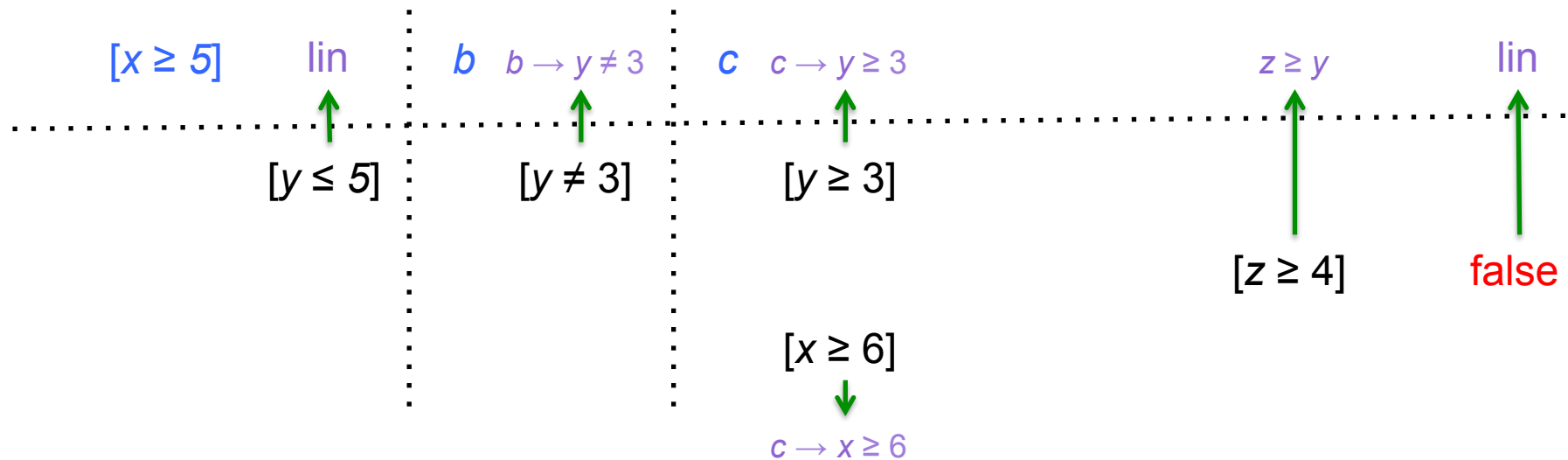
**Lifted Explanation:**  $x \geq 4 \wedge z \geq 4 \rightarrow z \geq 3 \wedge 4x + 10y + 7z \leq 7 \rightarrow \text{false}$

**Lifted Explanation:**  $y \geq 3 \wedge \neg(x \geq 5) \wedge [x \geq 5] \wedge [y \geq 4] \wedge [z \geq 3] \rightarrow \text{false}$

# LCG propagation example



- Execution



**Nogood:**  $[x \geq 5] \wedge [y \geq 4] \rightarrow \text{false}$

**1UIP Nogood:**  $[x \geq 5] \wedge [y \geq 4] \rightarrow \text{false}$

**1UIP Nogood:**  $[x \geq 5] \rightarrow [y \leq 3]$



# LCG propagation example



- Backjump



**Nogood:**  $[x \geq 5] \wedge [y \geq 4] \rightarrow \text{false}$

# Lazy Clause Generation

---



- **Strengths**
  - High level modelling
  - Learning avoids repeating the same subsearch
  - Strong autonomous search
  - Programmable search
  - Specialized global propagators (but requires work)
- **Weaknesses**
  - Optimization by repeated satisfaction search
  - Overhead compared to FD when nogoods are useless

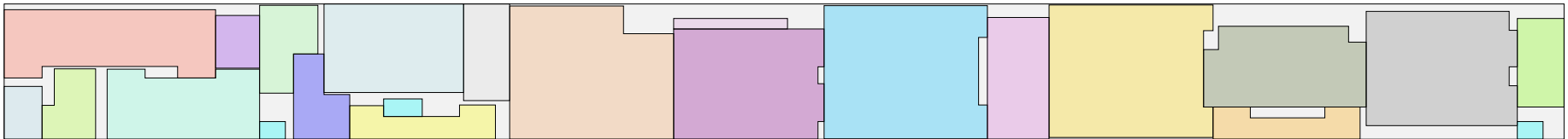
- Scheduling
  - Resource Constrained Project Scheduling Problems (RCPSP)
    - (probably) the most studied scheduling problems
    - LCG closed 71 open problems
    - Solves more problems in 18s then previous SOTA in 1800s
  - RCPSP/Max (more complex precedence constraints)
    - LCG closed 578 open instances of 631
    - LCG recreates or betters all best known solutions by any method on 2340 instances except 3
  - RCPSP/DC (discounted cashflow)
    - Always finds solution on 19440 instances, optimal in all but 152 (versus 832 in previous SOTA)
    - LCG is the SOTA complete method for this problem

# LCG Successes

---



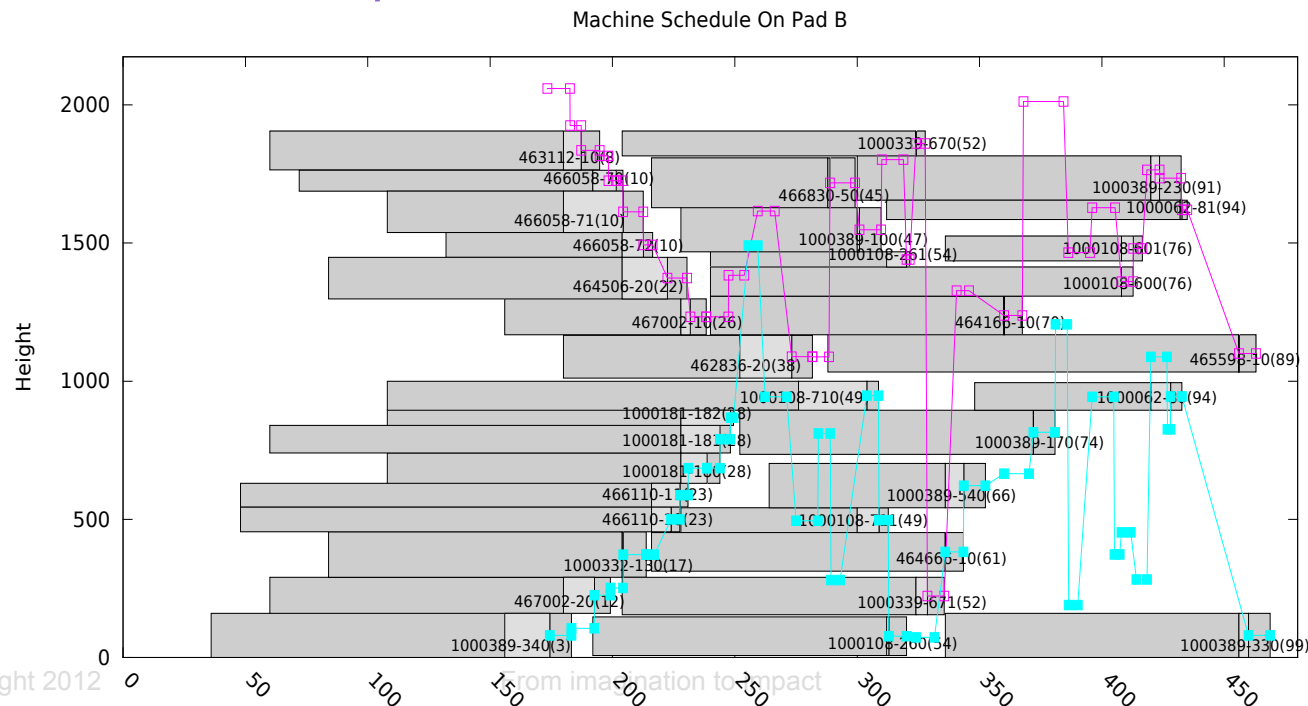
- Real World Application
  - Carpet Cutting
    - Complex packing problem
    - Cut carpet pieces from a roll to minimize length
    - Data from deployed solution



- Lazy Clause Generation Solution
  - First approach to find and prove optimal solutions
  - Faster than the current deployed solution
  - Reduces waste by 35%

# LCG Successes

- Real World Application
  - Bulk Mineral Port Scheduling
    - Combined scheduling problem and packing problem
    - Pack placement of cargos on a pad over time (2d)
    - Schedule reclaiming of cargo onto ship
    - LCG solver produces much better solutions



# Why you should use LCG

---



- State of the art constraint programming solvers
- Runs high level models directly
- Default search is very strong
- Programmed search/default search hybrid
- Basically the best of SAT and CP together

# Outline

---



- Optimization = intelligence
  - Discrete optimization has advanced rapidly
- Solver independent modelling
  - MiniZinc: a high level modelling language
- Nogood learning for discrete optimization
  - The laziness principle in action
- Resolving similar problems
  - “lifelong learning” and nested constraint programs
- Concluding remarks

# A Changing Environment

---



- Agents exist in a changing environment
- More information gathered
  - changes the problem
- An agent maximizing utility
  - reacts to the changing environment
  - by resolving the optimization problem
  - but most things are likely unchanged
- How can we take advantage of this?



# A Changing Environment Example

---



- Multi-agent bulk rail scheduling changes
  - SERVICES become unavailable
  - Track agent (environment) increases cost of edges
    - to price out overusage of edge
    - to avoid collisions
- Consist agent
  - needs to resolve if its SERVICE is taken
  - needs to replan route depending on cost changes

# Assumptions and Nogoods

---



- Assumptions:
  - Data that is “assumed true” but may later change
- Learning
  - Assumptions are set true in a special 0<sup>th</sup> decision level
  - Nogoods relying on assumptions incorporate the assumptions
- “Lifelong Learning”
  - Rerunning with changed assumptions
  - All previous nogoods are valid
    - and hence can be reused in search

# Re-Optimization



- For different changes in parameters

(a) Radiation Therapy

diff	scratch		para		reuse	speedup
1%	7.10	23000	0.08	32	97%	88.8
2%	7.25	23219	0.31	433	93%	23.8
5%	7.03	22556	1.17	2496	81%	6.02
10%	7.57	23628	2.52	6198	74%	3.00
20%	7.68	23326	4.35	11507	55%	1.77

(b) MOSP

diff	scratch		para		reuse	speedup
1%	38.67	111644	1.03	921	93%	37.5
2%	41.65	114358	4.57	6700	88%	9.11
5%	42.80	111745	31.02	75600	71%	1.38
10%	47.80	95521	40.77	88909	57%	1.17
20%	27.37	86231	33.26	88186	45%	0.82

(c) Graph Coloring

diff	scratch		para		reuse	speedup
1%	15.12	54854	7.72	17026	74%	1.96
2%	18.78	61630	16.17	35994	61%	1.16
5%	23.65	70710	24.96	52026	35%	0.94
10%	45.14	96668	45.54	79763	20%	0.99
20%	48.12	101668	45.96	87431	9%	1.04

(d) Knapsack

diff	scratch		para		reuse	speedup
1%	18.21	48714	6.74	11339	100%	2.70
2%	18.47	48640	7.32	13042	100%	2.52
5%	18.80	49154	16.91	37786	100%	1.11
10%	19.38	49298	21.12	44952	100%	0.92
20%	20.83	50007	24.15	49387	100%	0.86

- re-running from **scratch**, or with **parametric** assumptions, reuse of nogoods, speedup

# An Unknown Environment

---



- Often an agent needs to make decisions without knowing enough about the environment
  - Stochastic (discrete) optimization
- Stochastic discrete optimization
  - usually optimizes expected utility over a finite set of future/environment scenarios
  - given a distribution: finite scenarios by sampling

# Stochastic Discrete Optimization

---



- Scenario determinization
  - solve a  $n \times$  larger problem combining  $n$  scenarios
- Policy based search
  - solve by  $n \times$  larger search of “original” problem
- Progressive hedging
  - solve  $n$  separate problems
  - change objective to make answers align
    - resolve  $n$  separate problems

# Stochastic MiniZinc (Beta)

---



- stage annotations
  - mark decisions and data in stages
  - stage  $k$  decisions made before seeing stage  $k+1$  data
- scenario lists
  - multiple data files, one per scenario, and scenario weights
  - generates a combined scenario data file
- automatic transformation
  - determinization
  - policy based search (requires MiniSearch)
  - progressive hedging (requires MiniSearch) (2 stage)
- [www.minizinc.org/stochastic/](http://www.minizinc.org/stochastic/)

# Stochastic BRFS

---



- Add that, e.g.
  - travel times, edge costs are stochastic

```
array[EDGE] of DUR: travel :: stage(2);  
array[EDGE,HOUR] of int: cost :: stage(2);
```

- Route decisions are stage 1

```
var SERVICE: which :: stage(1);  
var TIME: start_time :: stage(1);  
array[NODE] of var NODE: next :: stage(1);  
array[NODE] of var EDGE0: route :: stage(1);
```

- Dwell can be adjusted

```
array[NODE] of var TIME: arrive :: stage(2);  
array[NODE] of var DWELL: dwell :: stage(2);
```

- Need to rewrite time bounds as penalties

# Stochastic BFRS



- Constraints: remove end time bound constraints

```
start_time = arrive[start[which]] /\
dwell[start[which]] = 0 /\
arrive[start[which]] >= earliest_start[which] /\
arrive[start[which]] > latest_start[which]);
```

- Modify objective

```
solve maximize payment[which] -
    sum(n in NODE)
        (if route[n] = 0 then 0 else
            cost[route[n], (arrive[n] + dwell[n]) div 60]
        endif)
- (arrive[end[which]] < earliest_end[which])*eepenalty
- (arrive[end[which]] > latest_end[which])*lepenalty;
```



# Nested Constraint Programs

---



- A powerful language for nested optimization problems
- Based on aggregator constraints
  - $y = \mathbf{agg}( [ f(x_1, \dots, x_n, z_1, \dots, z_m) \mid z_1, \dots, z_m \text{ where } C(x_1, \dots, x_n, z_1, \dots, z_m) ] )$   
where **agg** is a function on multisets
  - e.g. sum, min, max, average, median, exists, forall
- Lazy evaluation
  - wait until  $x_1, \dots, x_n$  are fixed
  - evaluate the multiset by search over  $z_1, \dots, z_m$
  - set  $y$  to the appropriate value

# Nested Constraint Programs



- Highly expressive:
  - #SAT, QBF, QCSP, Stochastic CP, ...
- Find the minimal number of clues  $x_{ik,jk} = d_k$  required to make a proper sudoku problem (exactly one solution)

```
y = min( [ sum([b_k | k in 1..n]) | b_1, ..., b_n where
          1 = sum([ 1 | x_11 in 1..9, ... x_99 in 1..9 where
                    forall([ b_k → x_ik,jk = d_k | k in 1..n]) /\
                    sudoku(x_11, ..., x_99) ] ) ] )
```

- where **sudoku** are sudoku constraints

# Nested Constraint Programs

---



- Naïve approach
  - completely solved by grounding
  - BUT completely impractical
- Actual approach
  - one copy of constraints
  - search on outer aggregator
    - wake a new copy of inner aggregator
- Improvements
  - learning (across invocations of inner aggregators)
  - short circuiting (e.g. when we find two solns we stop)
  - use grounding when known size and small

# Nested Constraint Programs



- Book production (stochastic) planning problem
  - uncertain demand 100..105 in each period
  - plan a production run so that we can cover demand 80% of the time
- Compare with stochastic CP using policy search and scenario generation (determinization)

stages	NCP		policy		scenario	
	fails	time	fails	time	fails	time
1	8	0.01	10	0.01	4	0.00
2	16	0.01	148	0.03	8	0.02
3	24	0.01	3604	0.76	24	0.16
4	32	0.01	95570	19.07	42	1.53
5	40	0.01	2616858	509.95	218	18.52
6	48	0.01	---	TO	1260	474.47

# Outline

---



- Optimization = intelligence
  - Discrete optimization has advanced rapidly
- Solver independent modelling
  - MiniZinc: a high level modelling language
- Nogood learning for discrete optimization
  - The laziness principle in action
- Resolving similar problems
  - “lifelong learning” and nested constraint programs
- Concluding remarks

# Concluding Remarks

---



- My highly biased opinion!
- Many agents could benefit from having decisions driven by discrete optimization problems
- Discrete optimization technology is now
  - easier to use than ever before
  - more powerful than ever before (MIP too)
  - more expressive than ever before
- Encourage you to dip your toe ...

# Pervasive Discrete Optimization

---



- Many problems in CS are examples of discrete optimization
  - once they get complicated enough
  - bespoke algorithms + greedy methods **fail**
- Discrete optimization is now an essential component for
  - semi-supervised/constrained machine learning
  - program analysis/concolic testing
  - combinatorics – when the maths runs out
  - termination testing, ...

# What Can You Play With

---



- MiniZinc 2.0 [www.minizinc.org](http://www.minizinc.org)
  - component based model and translation system
  - Stochastic MiniZinc beta
  - MiniSearch about to be released
- Opturion CPX [www.opturion.com](http://www.opturion.com)
  - state of the art commercial LCG solver
  - free academic license
- Chuffed [github.com/geoffchu/chuffed](https://github.com/geoffchu/chuffed)
  - state of the art experimental LCG solver
  - support for Nested CP is coming
- Coursera course
  - Modeling Discrete Optimization