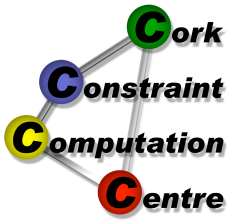
Constraint Programming and Combinatorial Optimisation in Numberjack

Emmanuel Hebrard, Eoin O'Mahony, Barry O'Sullivan

Cork Constraint Computation Centre, University College Cork
This work was supported by Science Foundation Ireland (Grant No. 05/IN/I886).



July 15, 2010

About this Tutorial

Motivation

Combinatorial optimization provides powerful support for decision-making; many interesting real-world problems are combinatorial and can be approached using similar techniques.

The Promise

This tutorial will teach attendees how to develop interesting models of combinatorial problems and solve them using constraint programming, satisfiability and mixed integer programming techniques.

Numberjack

We have developed Numberjack and released it under the LGPL license. It's an open-source project. Please consider getting involved for the benefit of academia, research and industry.

Combinatorial Optimisation

What is Combinatorial Optimisation?

Optimization problems divide neatly into two categories. Those involving:

- continuous variables
- discrete variables – these are combinatorial.

Usually we're concerned with finding a least-cost solution to a set of constraints.

Our focus in this tutorial

- Constraint Programming
- Satisfiability
- Mixed Integer Programming

What is a Combinatorial Problem?

Variables, Domains and Constraints

Given a set of variables, each taking a value from a domain of possible values, find an assignment to all variables that satisfy the constraints.

- Variables:

- Domain:

Find a solution!

- Constraints: Adjacent states must be colored differently

Combinatorial Optimization is Everywhere

7	5		9		3			6
			4	5				3
6	2			9		8		
	1	5				2	3	
		9		1			7	5
3				8	4			
9			6		1		5	7

(a) Sudoku Puzzle

7	5	8	9	2	3	1	4	6
2	4	3	1	6	7	5	9	8
1	9	6	4	5	8	7	2	3
6	2	7	3	9	5	8	1	4
8	1	5	7	4	6	2	3	9
4	3	9	8	1	2	6	7	5
3	7	1	5	8	4	9	6	2
5	6	4	2	7	9	3	8	1
9	8	2	6	3	1	4	5	7

(b) The Solution

Constraint Modelling Languages

Features

Declarative specification of the problem, separating (in so far as possible) the formulation and the search strategy.

A Constraint Model of the Sudoku Puzzle

```
matrix = Matrix(N*N,N*N,1,N*N)

sudoku
    = Model( [AllDiff(row) for row in matrix.row],

             [AllDiff(col) for col in matrix.col],

             [AllDiff(matrix[x:x+N, y:y+N].flat)
                for x in range(0,N*N,N)
                for y in range(0,N*N,N)] )
```

How do we solve a combinatorial problem?

- Polynomial-time Inference, e.g. arc consistency
- Systematic Search, e.g. backtrack search + inference
- Hybrid methods, e.g. operations research with CP
- Satisfiability – CSPs can be translated into CNF
- Local Search – heuristic guess with heuristic repair
- Large Neighbourhood Search – systematic and local search

We will focus on some of those solution methods

- **Polynomial-time Inference**, e.g. arc consistency
- **Systematic Search**, e.g. backtrack search + inference
- **Hybrid methods**, e.g. operations research with CP
- **Satisfiability** – CSPs can be translated into CNF
- ~~Local Search~~ – heuristic guess with heuristic repair
- ~~Large Neighbourhood Search~~ – systematic and local search

Boolean Satisfiability

What is SAT?

It can be regarded as a special case of CP

- Variables have Boolean domains $a \in \{0, 1\}$.
- Constraint are clauses
  - Disjunction of (negations of) atoms $(a \vee \neg b \vee \neg c)$

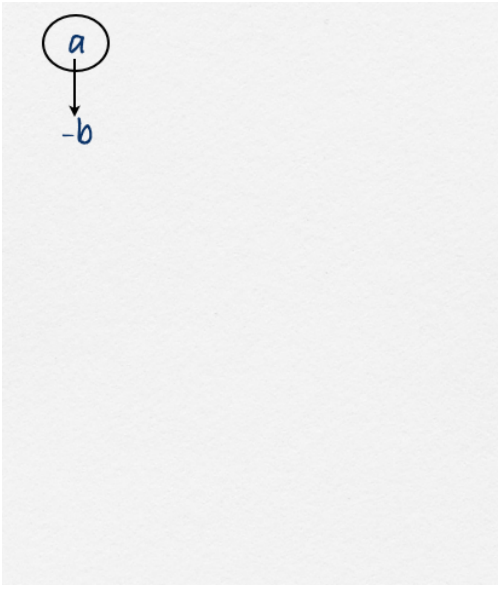
Example

- A formula:
  $(a \vee \neg d)$
  $(\neg b \vee \neg c \vee \neg d)$
  $(a \vee b \vee d)$
  $(\neg a \vee \neg b)$
  $(\neg b \vee \neg c \vee d)$
- A solution: $\neg a, b, \neg c, \neg d$
  $(a \vee \neg d)$
  $(\neg b \vee \neg c \vee \neg d)$
  $(a \vee b \vee d)$
  $(\neg a \vee \neg b)$
  $(\neg b \vee \neg c \vee d)$

Boolean Satisfiability

SAT Solvers

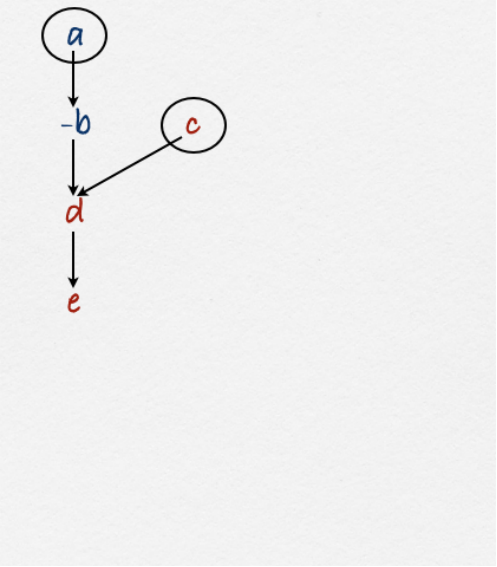
- 1^{st} decision: a
  - $(\neg a \vee \underline{\neg b})$



Boolean Satisfiability

SAT Solvers

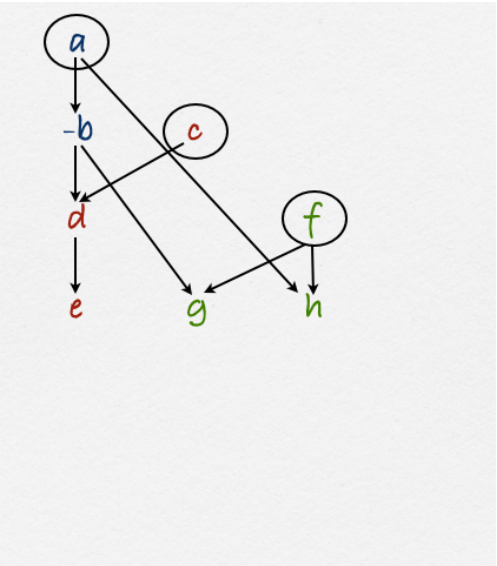
- 1^{st} decision: a
  - $(\neg a \vee \neg b)$
- 2^{nd} decision: c
  - $(b \vee \neg c \vee \underline{d})$
  - $(\neg d \vee \underline{e})$



Boolean Satisfiability

SAT Solvers

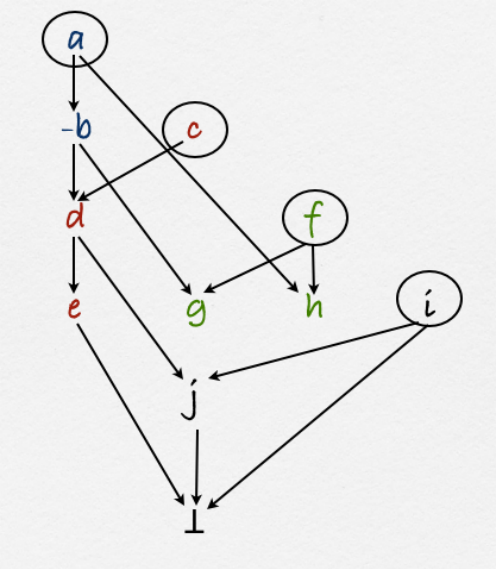
- 1^{st} decision: a
  - $(\neg a \vee \neg b)$
- 2^{nd} decision: c
  - $(b \vee \neg c \vee d)$
  - $(\neg d \vee e)$
- 3^{rd} decision: f
  - $(b \vee \neg f \vee \underline{g})$
  - $(\neg a \vee \neg f \vee \underline{h})$



Boolean Satisfiability

SAT Solvers

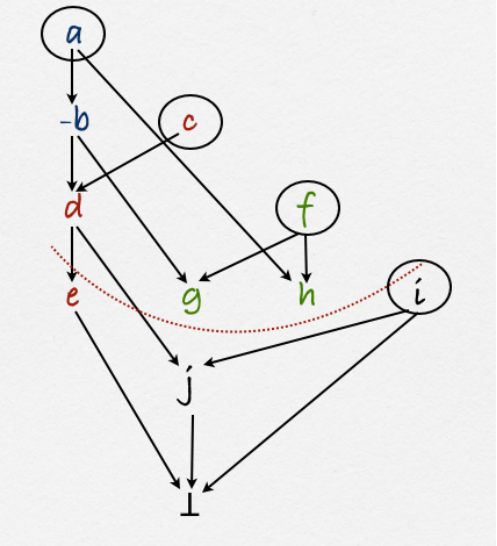
- 1^{st} decision: a
  - $(\neg a \vee \neg b)$
- 2^{nd} decision: c
  - $(b \vee \neg c \vee d)$
  - $(\neg d \vee e)$
- 3^{rd} decision: f
  - $(b \vee \neg f \vee g)$
  - $(\neg a \vee \neg f \vee h)$
- 4^{th} decision: i
  - $(\neg d \vee \neg i \vee j)$
  - $(\neg e \vee \neg i \vee \neg j)$



Boolean Satisfiability

SAT Solvers

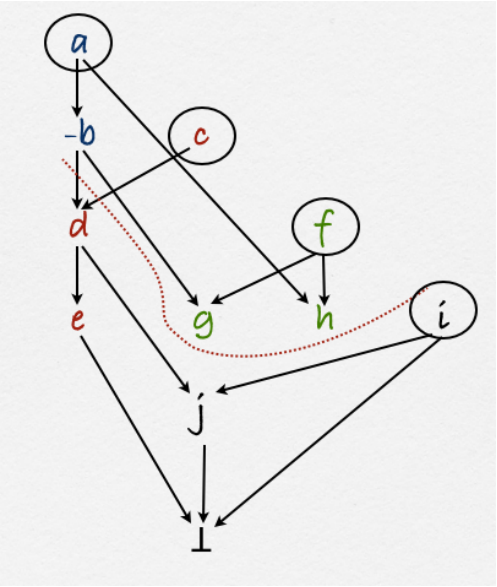
- Possible explanation
  - Any cut decisions/failure
  - $(\neg e \vee \neg i \vee \neg j)$



Boolean Satisfiability

SAT Solvers

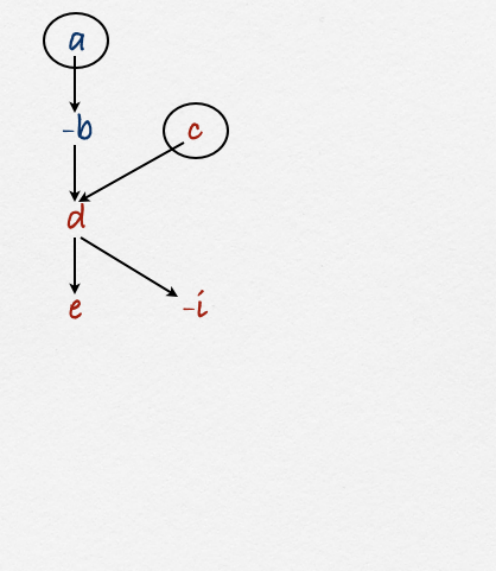
- Possible explanation
  - Any cut decisions/failure
  - $(\neg e \vee \neg i \vee \neg j)$
  - $(\neg d \vee \neg i)$



Boolean Satisfiability

SAT Solvers

- Jump over the 3^{rd} decision
- Learn the clause $(\neg d \vee \neg i)$



Mixed Integer Programming

Standard MIP Model has the following form

$$
\begin{aligned}
min \quad & cx + dy && (1)\\
s.t. \quad & Ax + By \geq 0 && (2)\\
& x, y \geq 0 && (3)\\
& y \text{ integer} && (4)
\end{aligned}
$$

Informally

The objective function is linear, e.g. a weighted sum, expression over the variables. Each constraint is linear. Some variables take integer values, other can take real values.

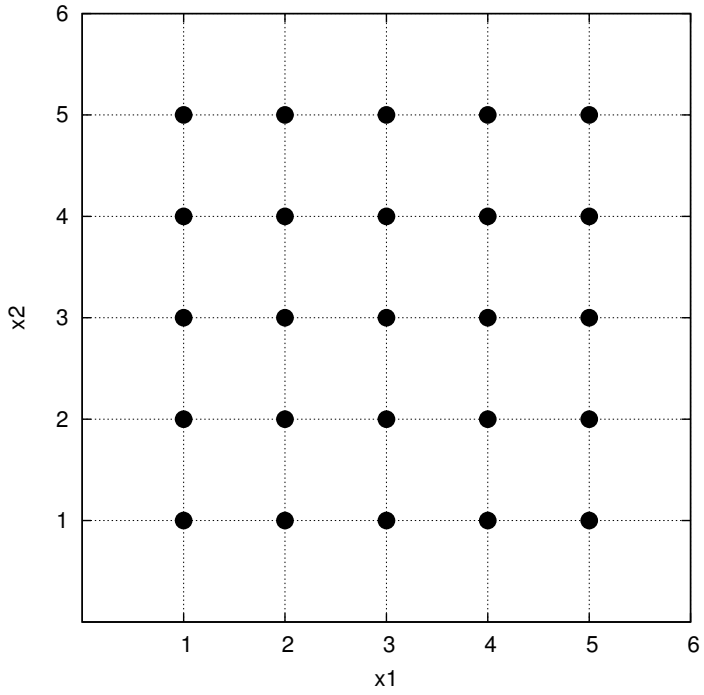
MIP Example: Combinatorial Auction for items A, B, C, D

We wish to sell 4 items (maximise revenue), given these 4 bids.

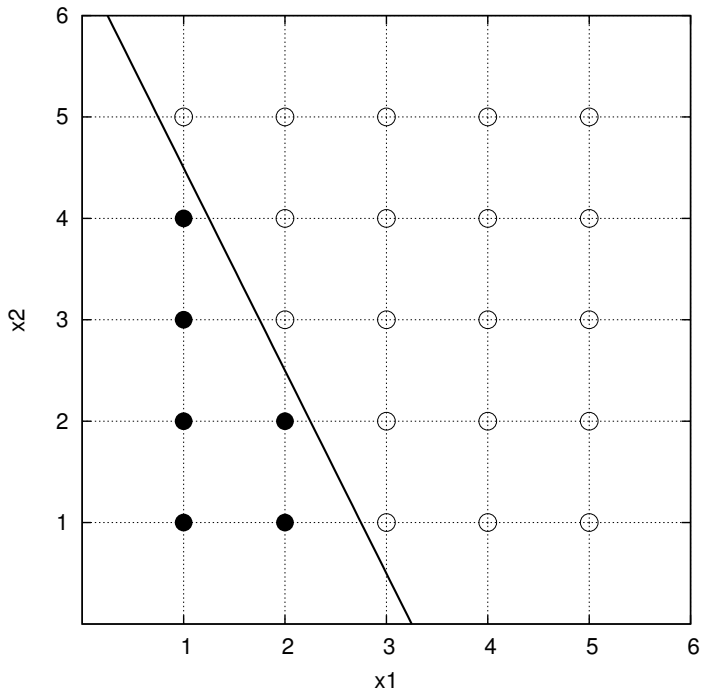
Items	Bid Amount	MIP Variable
A, B	10	x_1
A, C	20	x_2
B, D	30	x_3
B, C, D	40	x_4
A	14	x_5

$$
\begin{aligned}
max \quad & 10 \times x_1 + 20 \times x_2 + 30 \times x_3 + 40 \times x_4 + 14 \times x_5\\
s.t. \quad & x_1 + x_2 + x_5 \leq 1\\
& x_1 + x_3 + x_4 \leq 1\\
& x_2 + x_4 \leq 1\\
& x_3 + x_4 \leq 1\\
& x_1, x_2, x_3, x_4, x_5 \text{ binary}
\end{aligned}
$$

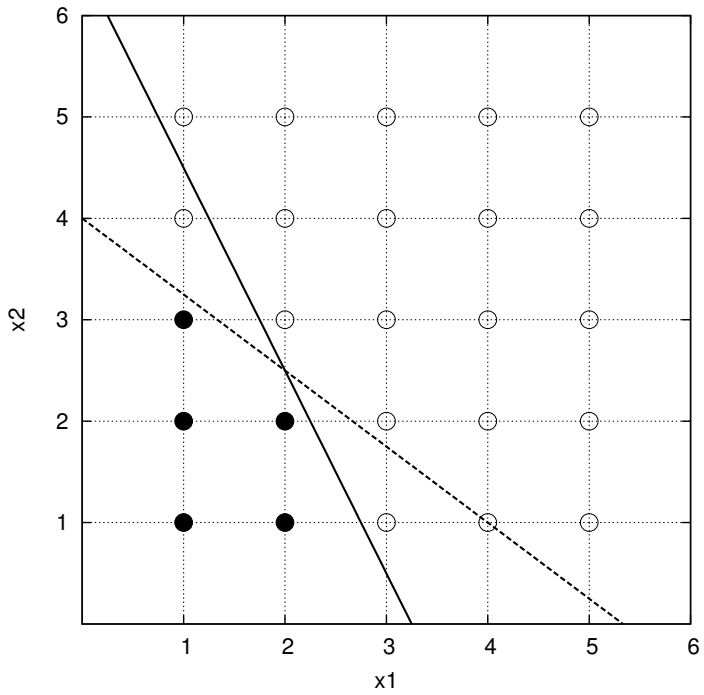
MIP Solving



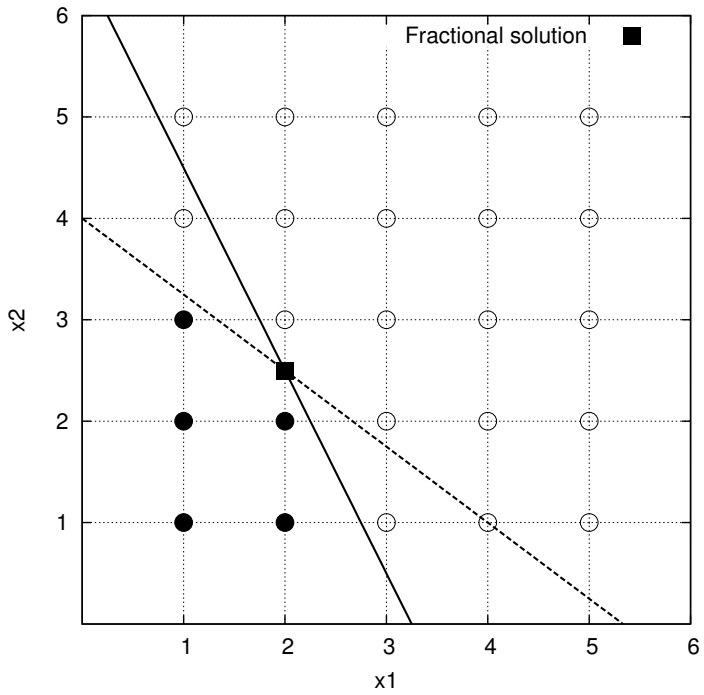
MIP Solving



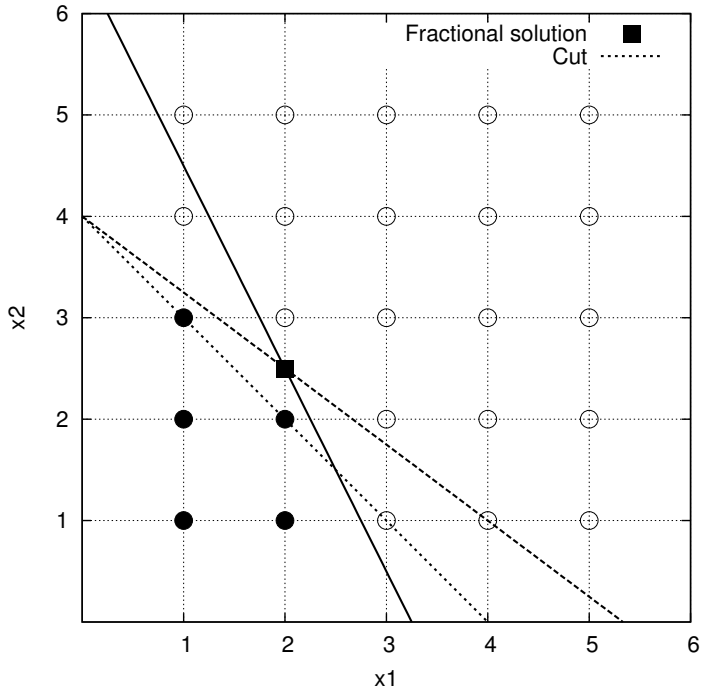
MIP Solving



MIP Solving



MIP Solving



Constraint Programming

Data Structures

- Domain types (`Integer`, `Real`, `Set`,...)
- Domain implementation

Constraint Propagation

Example

AllDifferent Variables must be assigned distinct values

$$
\begin{array}{lcl}
x_1 & \in & \{1,2,\quad 5\} \\
x_2 & \in & \{1,2,\quad 5\} \\
x_3 & \in & \{1,2,\quad 5\} \\
x_4 & \in & \{\quad 2,3,4\quad\} \\
x_5 & \in & \{1,2,3,4,5\}
\end{array}
\qquad
\begin{array}{lcl}
x_1 & \in & \{1,2,\quad 5\} \\
x_2 & \in & \{1,2,\quad 5\} \\
x_3 & \in & \{1,2,\quad 5\} \\
x_4 & \in & \{\quad 3,4\quad\} \\
x_5 & \in & \{\quad 3,4\quad\}
\end{array}
$$

CP, SAT, MIP, choice is good!

Logically equivalent

- They are all NP-complete: there always exists a reduction

Operationally different

- Algorithms are different
  - CP: Constraint propagation + Search
  - SAT: Unit propagation + Clause Learning + Search
  - MIP: Linear relaxation + Cutting planes + Branch & Bound
- Encodings matter

Numberjack

- Language: union of SAT, MIP and CP
  - That is, CP!
- The instances are encoded to best suit the back-end solver

CP: Languages and APIs

Modelling Languages

Minizinc, OPL, Numberjack, AMPL, Essence

APIs/Languages

CP Optimizer, Choco, CPLEX API, SCIP, Sugar, ECLiPse, Comet.

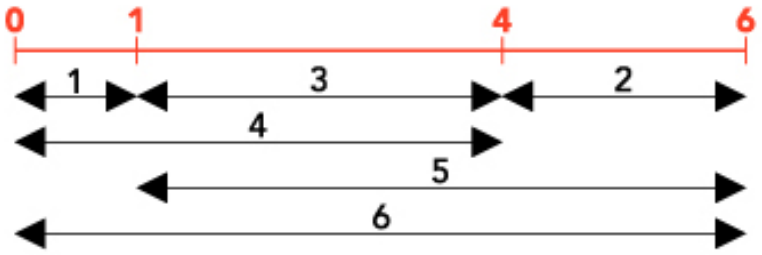
Formats

Dimacs, XCSP, Flatzinc, Minion, MPS, LP

An Example: Golomb Ruler

Problem definition

- Place N marks on a ruler
- Distance between each pair of marks is different
- Goal is to minimise the placement of the last mark



Golomb Ruler in ECLiPse – Prolog [ECLiPSe web-site]

```
golomb(M, Marks) :-
    length(Marks, M),                                    % model
    NN is 2^(M-1)-1,
    Marks :: [0..NN],
    append([0|_], [Xn], Marks),
    ordered(<, Marks),
    distances(Marks, Diffs),
    Diffs::[1..NN],
    alldifferent(Diffs),
    append([D1|_], [Dn], Diffs),
    D1 #< Dn,
    bb_min(labeling(Marks), Xn, bb_options{strategy:step}). % search

distances([], []).
distances([X|Ys], D0) :- distances(X, Ys, D0, D1), distances(Ys, D1).

distances(_, [], D, D).
distances(X, [Y|Ys], [Diff|D1], D0) :- Diff #= Y-X, distances(X, Ys, D1, D0).
```

Golomb Ruler in Choco – Java [by Hadrien Cambazard]

```
public void solve() {
    int m = 8;
    int n = (int) Math.pow(2,m);

    model = new CPModel();
    marks = makeIntVarArray("a", m, 0, n);
    distances = makeIntVarArray("d", ((m)*(m-1))/2, 0, n);
    int cpt = 0;
    for (int i = 0; i < m; i++) {
        for (int j = i+1; j < m; j++) {
            model.addConstraint(eq(distances[cpt++],
                                   minus(marks[j],marks[i])));
        }
    }
    model.addConstraint(allDifferent("cp:bc",distances));
    model.addConstraint(eq(marks[0],0));

    solver = new CPSolver();
    solver.read(model);
    solver.setObjective(solver.getVar(marks[m-1]));
    solver.minimize(false);
}
```


Golomb Ruler in Mistral – C++ [Mistral distribution]

```
int main(int argc, char *argv[])
{
  int m = 8;
  int n = 2 << m;

  CSP model;
  VarArray marks(m, 0, n-1);
  VarArray distance(m*(m-1)/2, 1, n-1);

  int i, j, k=0;
  for(i=1; i<m; ++i) {
    model.add( marks[i-1] < marks[i] );
    for(j=0; j<i; ++j)
      model.add( marks[i] == (marks[j] + distance[k++]) );
  }

  model.add( AllDifferent(distance) );
  model.add( marks[0] == 0 );
  model.add( Minimise( marks[m-1] ) );

  Solver s( model, marks );
  s.solve();
}
```

Golomb Ruler in Minizinc [Minizinc distribution]

```
int: m = 8;
int: n = 2**m;

array[1..m] of var 0..n: marks;

constraint forall ( i in 1..m-1 ) ( marks[i] < marks[i+1] );

constraint
    all_different( [ marks[j] - marks[i] | i in 1..m, j in i+1..m] );

constraint marks[1] = 0;

solve :: int_search(marks, input_order, indomain )
         minimize marks[m];
```


Golomb Ruler in Numberjack

```
m = 8
n = 2**m

marks = [Variable(n) for i in range(m)]

model = Model(

    Minimise( marks[-1] ),

    [marks[i-1] < marks[i] for i in range(1,m)],

    AllDiff( [ (first - second) for first, second in pair_of(marks) ] ),

    marks[0] == 0
)

solver = Mistral.Solver( model, marks )
solver.solve()
```

So again...

What is Numberjack?

- A platform for combinatorial optimization
  - Open and collaborative – open source project
  - Common language for diverse paradigms (CP, SAT, MIP)
- Work in progress...
  - We need you to get involved.

Yet another platform?

- Yes, but not a new language
  - API, hence deeper control the back-end solvers
  - Python is an established programming language
  - It is easy to plug into other applications

Introduction to Python

This section will cover

- Very quick introduction to Python
- Python needed to understand out examples

Everything you need to know about Python (for this tutorial)

Python

- Scripting language
- Duck typed
  - If it looks like a duck and quacks like a duck its a duck!
- Large community
- Worth learning!

The basics

This is an assignment

```
X = 5
```

- No type
- No previous definition

This is a function

```
def f():
    return 5
```

- Whitespace indentation
- Functions are not typed

The basics

Lists

```
primes = [2, 3, 5, 7, 11]
cities = ["Cork", "Toulouse", "Ithaca"]
```

- Mutable

Tuples

```
places = (1, 2, 3)
conference = ("AAAI", 2010)
```

- Immutable

The basics

Control flow

```
if boolean_expression:
   do_stuff()
```

What about loops?

```
while boolean_expression:
   do_stuff()
```

The basics

For loops

C++/Java/C:

```
for(int i = 0; i < n; ++i){
   do_stuff(i);
}
```

Python:

```
for i in range(N):
   do_stuff(i)
```


The Basics

Looping over lists

```
for item in list:
   do_stuff_with(item)
```

- Can iterate over more than initial depth of list

```
scores = [("Villa", 5),  ("Sneijder", 5), ("Henry", 0)]
for player, goals in scores:
   print player, " scored ", goals, " goals"
```

The less basics

More on playing with lists

```
numbers = [1, 2, 3]
letters = ["a", "b", "c"]
for letter, number in zip(numbers, letters):
   print number, letter
```

- Function zip returns list of tuples of elements in its arguments that have the same index

```
>>>zip(numbers, letters)
[(1, "a"), (2, "b"), (3, "c")]
```


The less basics

List comprehensions

One of Python's most useful feature for concise code is list comprehensions. The implementation is similar to that of Haskell.

```
>>> range(4)
[0, 1, 2, 3]
>>> [x*2 for x in range(4)]
[0, 2, 4, 6]
```

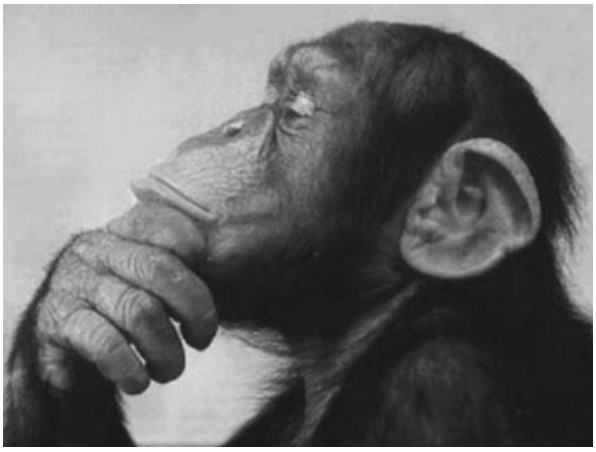
List comprehensions generally take the following form.

```
[function(x) for x in <Iterable> (if <condition>)]
```

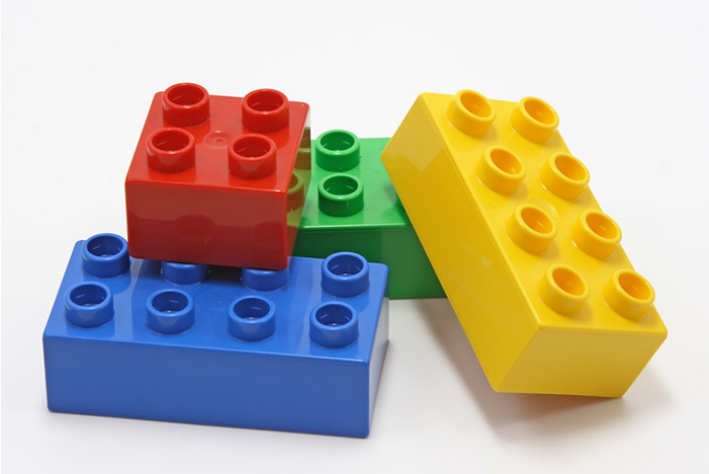
The less basics

Everything is an object!

- You can print everything
- You can pass everything
- You can return everything
- You can play with everything



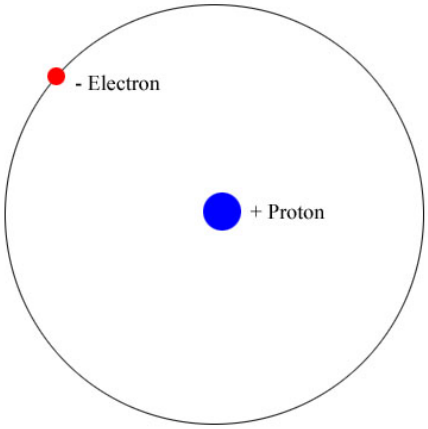
Basic Modeling and Solving



This section will cover

- Basics of CSP modeling
- Basics of modeling in Numberjack
- Solving your (numberjack) problems

Variables in Numberjack



Basic building block of modeling

- Can be specified in many different ways
- Can be contained in larger constructs

Variables in Numberjack

Different Variable constructors

Constructor	Description
`Variable()`	Binary variable
`Variable(N)`	Variable in the domain of 0, N-1
`Variable('x')`	Binary variable called 'x'
`Variable(N, 'x')`	Variable in the domain of 0, N-1 called 'x'
`Variable(l,u)`	Variable in the domain of l, u
`Variable(l,u, 'x')`	Variable in the domain of l, u called 'x'
`Variable(list)`	Variable with domain specified as a list
`Variable(list, 'x')`	Variable with domain as a list called 'x'

Constraints in Numberjack

Constraints can be specified in a number of ways

- Constraints can be specified by arithmetic operators of variables
  - `x > y`
    `x + 4 > z*3`
- Global constraints can be expressed by calling their constructor
  - `AllDiff([x,y,z])`
    `Sum([a,b,c,d]) >= e`

Combinatorial Auction Example

We wish to sell 4 items (maximise revenue), given these 4 bids.

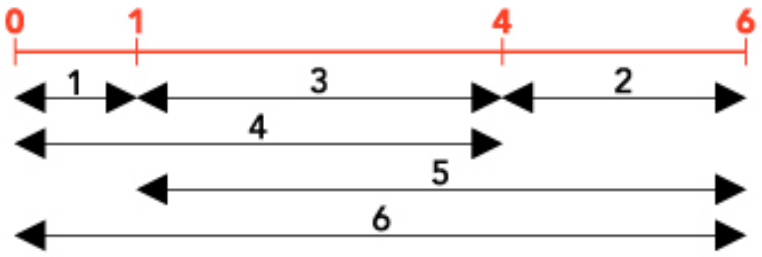
Items	Bid Amount	MIP Variable
A, B	10	x_1
A, C	20	x_2
B, D	30	x_3
B, C, D	40	x_4
A	14	x_5

$$
\begin{aligned}
max \quad & 10 \times x_1 + 20 \times x_2 + 30 \times x_3 + 40 \times x_4 + 14 \times x_5 \\
s.t. \quad & x_1 + x_2 + x_5 \leq 1 \\
& x_1 + x_3 + x_4 \leq 1 \\
& x_2 + x_4 \leq 1 \\
& x_3 + x_4 \leq 1 \\
& x_1, x_2, x_3, x_4, x_5 \text{ binary}
\end{aligned}
$$

Golomb Ruler

Problem definition

- Place N marks on a ruler
- Distance between each pair of marks is different
- Goal is to minimise the placement of the last mark
- Proposed by Sidon [1932] then independently by Golomb and Babcock



Golomb Ruler

Create the Variables

```
marks = VarArray(nbMarks,rulerSize)
```

- Each variable represents the position of a mark

Now the model

- Modeling choices are important
- We will illustrate this with a series of models

First Model

Naive Model

```
model = Model(
    Minimise( Max(marks) ),
    AllDiff(marks),
    AllDiff([first - second for first, second
       in pair_of(marks)]),
)
```

- Each mark must be at a different position
- Each pair of distances must be different
- Minimise the position of the last mark

First Model Analysis

This is not a very good model!

- The marks must all be different
- But they can also be totally ordered
- Instead of marks[0] being any mark on the ruler we can constrain it to be the first mark
- Then we just have to minimise the position of the last mark in the marks array

```
model = Model(
    Minimise( marks[-1] ),
    [marks[i] < marks[i+1] for i in range(nbMarks-1)],
    AllDiff([first - second for first, second
          in pair_of(marks)]),
)
```

Why does this work?

These extra constraints improve the speed

- These constraints reduce both the time taken and nodes searched

Adding in constraints can help

- Stronger constraints reduce the search space
- Still maintain (a set of) solutions

More modeling advances

What can we reason about the first mark?

- It must always be at zero
- Proof: Any solution with the first mark at position n can be be shifted n positions to the left while maintaining all the constraints and reducing the objective function
- Obvious to us but not to the solver

```
model = Model(
    Minimise( marks[-1] ),
    [marks[i] < marks[i+1] for i in range(nbMarks-1)],
    AllDiff([first - second for first, second
          in pair_of(marks)]),
    marks[0] == 0
)
```

Where are we searching?

What are the decision variables?

- By default the solvers will search on all variables in a problem
- In the Golomb ruler most of these will be auxiliary variables representing the distance between the marks
- These are functionally dependent on the positions of the marks
- So why bother searching on these!?

```
s = Mistral.Solver( model, marks )
```

- Tell the solver what variables to search on

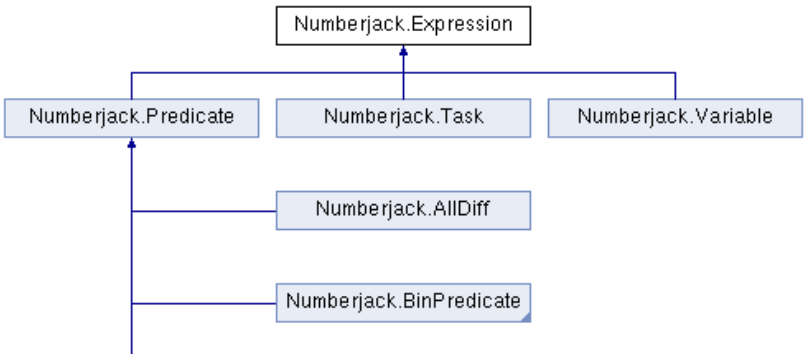
Using previous solutions

Previous solutions provide good bounds

- If you take a golomb ruler of size n, if you take $k < n$ sub-ruler this sub-ruler must be an order k golomb ruler

```
[distance[i*(i-1)/2+j] >= ruler[i-j]
   for i in range(1,nbMarks)
   for j in range(0,i-1)
      if (i-j < nbMarks-1)]
```

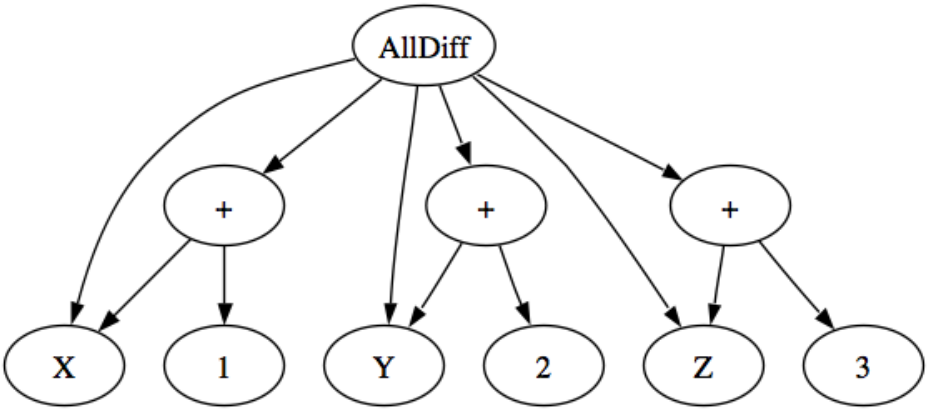
The Structure of things



Everything is an Expression

- All variables and constraints are Expressions
- Meaning depends on where they are expressed

The Structure of things

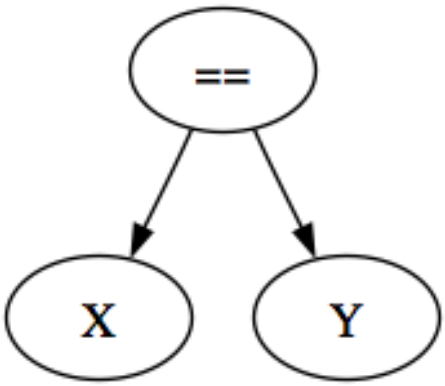



```
model.add( AllDiff([X, X + 1, Y, Y + 2, Z, Z + 3 ] )
```

Expressions are Trees

- The root node is added to the model
- Not all Expressions can be the root node of a tree
- Not all Expressions can be nodes within the tree
- Every class that extends Predicate has an attribute children

The Structure of things

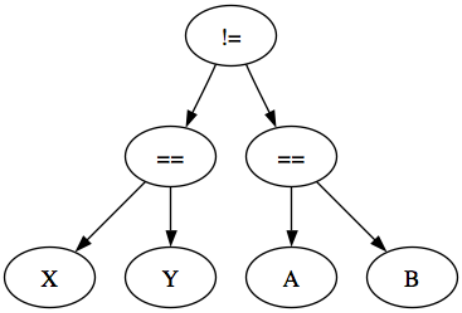


Expressions at different levels

```
model.add(X == Y)
```

- This Expression is taken to mean that the value of Expression X must be equal to the value of Expression Y
- Predicates (Constraints) specified at the root node must be satisfied

The Structure of things

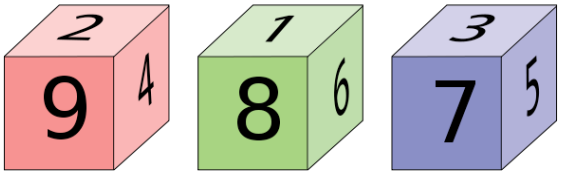


Expressions at different levels

```
model.add( (X == Y) != (A == B))
```

- The Expressions $(X == Y)$ and $(A == B)$ are taken to be the truth value of the equality relations between the Expressions X,Y,A and B respectivly
- The inequality Expression states that these two relations must not take the same value

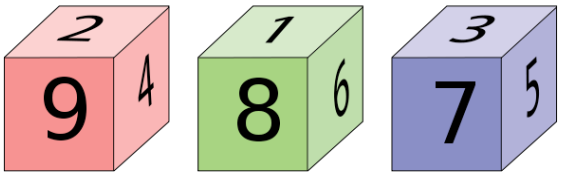
Time to get some practice!



Non Transitive dice problem

- Set of dice in which the property beats is non transitive

Non Transitive Dice



Problem Definition

- 3 dice
- Place digits on faces
- A die beats another if it rolls higher more often
- Need to have three dice such that if a beats b and b beats c then a does not beat c

Non Transitive Dice - Example

- A solution:

Die A:	1	2	3	4	5	5
Die B:	3	3	3	3	3	3
Die C:	2	2	2	3	6	6

Non Transitive Dice - Example

- A solution:

Die A:	1	2	3	**4**	**5**	**5**
Die B:	**3**	**3**	**3**	**3**	**3**	**3**
Die C:	2	2	2	3	6	6

- A beats B with probability $\frac{1}{2}$ (18 times out of 36).

Non Transitive Dice - Example

- A solution:

Die A:	**1**	**2**	3	4	5	5
Die B:	**3**	**3**	**3**	**3**	**3**	**3**
Die C:	2	2	2	3	6	6

- A beats B with probability $\frac{1}{2}$ (18 times out of 36).
- B beats A with probability $\frac{1}{3}$ (12 times out of 36).

- A **beats** B

Non Transitive Dice - Example

- A solution:

Die A:	1	2	3	4	5	5
Die B:	**3**	**3**	**3**	**3**	**3**	**3**
Die C:	**2**	**2**	**2**	3	6	6

- B beats C with probability $\frac{1}{2}$ (18 times out of 36).

- A **beats** B

Non Transitive Dice - Example

- A solution:

Die A:	1	2	3	4	5	5
Die B:	**3**	**3**	**3**	**3**	**3**	**3**
Die C:	2	2	2	3	**6**	**6**

- B beats C with probability $\frac{1}{2}$ (18 times out of 36).
- C beats B with probability $\frac{1}{3}$ (12 times out of 36).

- A **beats** B, B **beats** C

Non Transitive Dice - Example

- A solution:

Die A:	1	2	**3**	4	5	5
Die C:	**2**	**2**	**2**	3	6	6
Die A:	1	2	3	**4**	**5**	**5**
Die C:	**2**	**2**	**2**	**3**	6	6

- A beats C with probability $\frac{5}{12}$ (15 times out of 36).

- A **beats** B, B **beats** C

Non Transitive Dice - Example

- A solution:

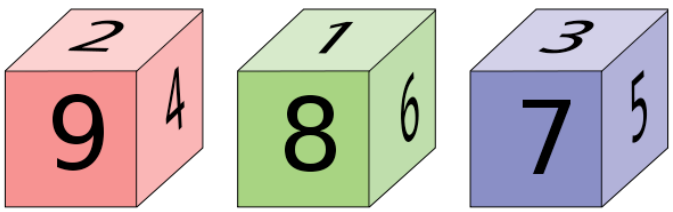
Die A:	**1**	2	3	4	5	5
Die C:	**2**	**2**	**2**	3	6	6
Die A:	**1**	**2**	3	4	5	5
Die C:	2	2	2	**3**	6	6
Die A:	**1**	**2**	**3**	**4**	**5**	**5**
Die C:	2	2	2	3	**6**	**6**

- A beats C with probability $\frac{5}{12}$ (15 times out of 36).
- C beats A with probability $\frac{17}{36}$ (17 times out of 36).

- **A beats B, B beats C, C beats A!**

Non Transitive Dice - Exercise

- Find a (different) solution
- Extend the model for any number of dice (D_1 beats D_2, D_2 beats D_3, ..., D_n beats D_1)
- Maximize the sum of the probability gaps ($\frac{1}{2} - \frac{1}{3} + \frac{1}{2} - \frac{1}{3} + \frac{17}{36} - \frac{5}{12}$)
- Maximize the number of 6-faced dice for which there is a cycle



Prizes

Winners

Best three solutions each get a small bottle of Jameson Irish Whiskey.

Losers

Worst three solutions each get a Leprechaun Of Shame.

"I do not like the cone of shame"

– Doug the Dog (Up)

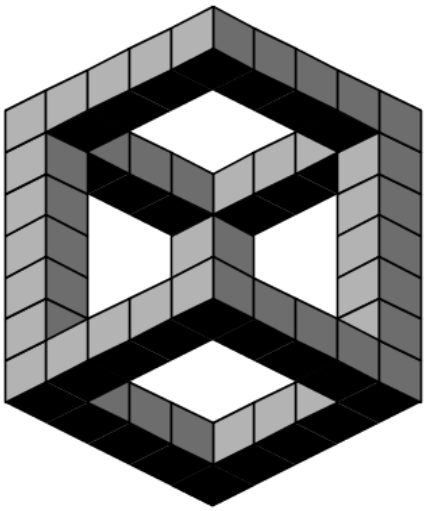
Off you go!

Advanced Modeling

This section will cover

- Advanced Numberjack constructs
- Adding custom constraints and constraint decompositions

Numberjack Constructs



Numberjack provides constructs to allow elegant modeling

- Matrix and VarArray provide an elegant solution to many modeling challenges
- Following examples showcase the ease of use of these features

Sudoku

Problem Definition

- 9 x 9 grid
- Place 1 to 9 in each column, row and square.
- Pain to model; array indexing is awful!

Sudoku

Create the variables

```
grid = Matrix(N*N,N*N,1,N*N,'cell_')
```

Matrix is a special Numberjack construct

- Allows for easy modeling in terms of matrices of variables
- Access rows
- Access columns
- Access sub matrices
- Post two dimensional element constraints
- Elegant printing

Sudoku

The Model

```
sudoku = Model( [AllDiff(row) for row in grid.row],
                [AllDiff(col) for col in grid.col],
                [AllDiff(grid[x:x+N, y:y+N].flat)
                   for x in range(0,N*N,N)
                   for y in range(0,N*N,N)],
              )
```

- Every row must have different digits
- Every column must have different digits
- Every sub matrix must have different digits

Sudoku

Accessing the sub matrices

```
[AllDiff(grid[x:x+N, y:y+N].flat)
                   for x in range(0,N*N,N)
                   for y in range(0,N*N,N)],
```

- grid[a:b, c:d] returns a Matrix construct that contains rows a to b inclusive and columns c to d inclusive.
- The flat attribute is a flattened version of this matrix

Sudoku

Adding in a partially completed grid

```
[(x == int(v)) for (x,v) in zip(grid.flat,
      "".join(open(clues)).split() ) if v != '*']
```

Isn't Python useful?!

- clues is the path to a problem file
- Entry that is not * is a constraint
- Difficult in a modeling language

```
1 * 3
4 5 *
* * 9
```


Magic Square

- N x N grid
- Contains numbers 1 to N^2
- Sum of rows, columns and diagonals equal.
- Known for at least 3000 years!
- Albrecht Dürer in 1514

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Magic Square

The Model

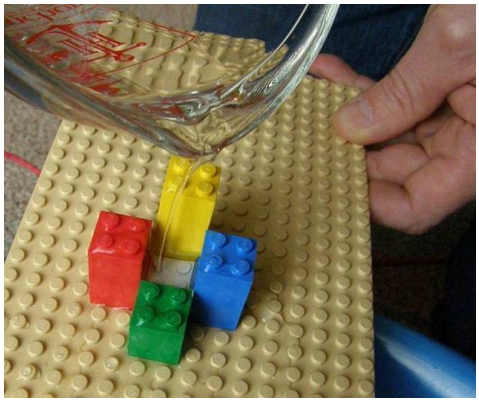
```
square = Matrix(N,N,1,N*N)
```

- N by N Matrix of variables from one to N squared

```
model = Model(
   AllDiff( square.flat ),
   [Sum(row) == sum_val for row in square.row],
   [Sum(col) == sum_val for col in square.col],
   Sum([square[a][a] for a in range(N)]) == sum_val,
   Sum([square[a][N-a-1] for a in range(N)]) == sum_val)
```


Water Retention Magic Squares

- Water retention (Craig Knecht 07)
- Magic square = 3D histogram
- Pour water on the histogram
- Maximize the amount of water



Water Retention Magic Squares

Dürer's Square

- One "lake"
- Water level = 9
- Water Depth: 3 and 2
- Water Retention = 5

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

16	3	2	13
5	10	11	8
9	6+3	7+2	12
4	15	14	1

Water Retention Magic Square

How much water will it hold?

- For a cell to hold water it must be lower than the surrounding blocks
- The water a cell will hold is the maximum of the height of the square and the minimum amount of water held by the squares surrounding cells

Water Retention Magic Square

Modeling the water

```
water = Matrix(N,N,1,N*N)
```

- Variables to represent the amount of water stored in each cell

```
water.row[0] == square.row[0]
water.row[N-1] == square.row[N-1]
water.col[0] == square.col[0]
water.col[N-1] == square.col[N-1]
```

- The water level at each of the outside squares is equal to the value at that square
- Matrices allow expression of equality among variable vectors

Water Retention Magic Square

Water in the middle

```
[water[a][b] == Max((square[a][b],
   Min((water[a-1][b], water[a][b-1],
        water[a+1][b], water[a][b+1])))
            for a in range(1,N-1)
            for b in range(1,N-1)],
```

- Level at a cell is the max of itself or the minimum water level surrounding it

```
Maximise( Sum( water.flat ) )
```

- Goal is to maximise the water collected

Water Retention Model

```
sum_val = N*(N*N+1)/2 # The magic number

square = Matrix(N,N,1,N*N)
water = Matrix(N,N,1,N*N) # Represent the water level in each cell

model = Model(
    ... # magic square constraints

    # First, no water can hold on the rim
    water.row[0] == square.row[0],
    water.row[N-1] == square.row[N-1],
    water.col[0] == square.col[0],
    water.col[N-1] == square.col[N-1],

    # Then, the level of an inner cell is the max between
    # its own height and of the water level around
    [water[a][b] == Max((square[a][b],
                         Min((water[a-1][b], water[a][b-1],
                              water[a+1][b], water[a][b+1])))
                        ) for a in range(1,N-1) for b in range(1,N-1)],

    # The objective function
    Maximise( Sum(water.flat) )
    )
```


Decomposing Constraints

Not all solvers support all constraints

- Most solvers do not implement all possible constraints

Constraints can be decomposed

- Take a complex constraint and express it with a series of simpler constraints

Decomposing Constraints

Why decompose constraints

- Allows solvers to solve models they would be otherwise unable to solve
- Allows expression of complex constraints in SAT and MIP without having to work out encodings of the constraints.

What is the difference to encoding?

- Encoding consists of creating a dedicated model for a given constraint
- Encodings tend to keep more of the structure
- For example the AllDiff constraint can be encoded as a flow problem for a MIP solver while the AllDiff constraint can be decomposed into a clique of not equal constraints.

Decomposition Example

Decomposing the AllDifferent constraint

```
def decompose_AllDiff(self):
    return [var1 != var2 for var1, var2 in
                pair_of(self.children)]
```

- Suppose solver does not have an All Different constraint
- Constraint can be decomposed into a clique of Not Equal constraints

When loading the model:

- Looks into the Solver and tries to find AllDiff constraint
- If this fails it checks to see if constraint can be decomposed
- If it can, decompose the constraint

Scheduling Decomposition Example

Constraint Programming is useful for scheduling

- Problems can be modeled in high level tasks and resource objects
- High level constraints such as tasks require resources or tasks may not overlap can be modeled easily

Scheduling Decomposition Example

Unary Resource

- Represent some resource tasks require (e.g. a meeting room)
- If tasks require the same resource they must not overlap

NoOverlap

- Constraint that ensures that two tasks do not overlap
- $t1_{start} + t1_{duration} <= t2_{start} \vee t2_{start} + t2_{duration} <= t1_{start}$

Tasks

- Tasks have a start time and a duration

Decomposing Scheduling Constraints

Neither Mistral or MiniSat have UnaryResource Constraints

- Unary Resource constraint is decomposed into NoOverlap constraints between all pairs of associated tasks

Decompositions are recursive

- MiniSat does not have a NoOverlap constraint
- Constraint is decomposed into expression
- $t1_{start} + t1_{duration} <= t2_{start} \vee t2_{start} + t2_{duration} <= t1_{start}$

Scheduling Decomposition in Numberjack

UnaryResource

```
class UnaryResource(Predicate):
    def decompose(self):
        return [NoOverlap(task1, task2)
             for task1, task2 in pair_of(self.children)]
```

NoOverlap

```
class NoOverlap(Predicate):
    def decompose(self):
       t1 = self.children[0]
       t2 = self.children[1]
       return [ t1 + t1.duration <= t2 |
                t2 + t2.duration <= t1 ]
```

Adding your own decompositions and constraints

You too can decompose constraints!

- Numberjack provides a file "decomp.py" where you can add in custom constraints or custom decompositions.

Defining your own constraints

- In this file you can place definitions of constraints such as the HammingDistance constraint already mentioned

Adding your own decompositions and constraints

```
def decompose_AllDiff(self):
    return [var1 != var2 for var1, var2 in
                pair_of(self.children)]
```

Defining your own decompositions

- You can also change the default decompositions of constraints
- This is done by adding a method of the form

```
def decompose_ConstraintName(self):
    body
```

- This custom decomposition method is bound to the constraint when Numberjack is loaded.

Controling Search

This section will cover

- Prototyping, search/heuristics/solvers
- Programming your own search method

Problem Solving

Language & problem solving

- Getting a solution quickly (Development time + Solving time)
  - Not all problems are hard
  - High level language does not mean inefficient solving
- It is very important to play with a problem
- Understanding how a problem can be best attacked takes time
  - Try different models/viewpoints
  - Try to add implied constraint, break symmetries, etc
  - Try different strategies/solvers
- Ease of use can make a great difference
  - Do not pass on a good idea because it is too difficult to implement
  - Do not spend too much time on a bad idea

Prototyping: search

Parameter tuning

- Selecting
  - Variable Ordering
  - Branching
  - Restarts
  - Randomization

Parameter setting methods are wrapped in Python

- `solver.setHeuristic('MinDomain', 'Lex', 3)`
- `solver.guide(solution)`
- `solver.setTimeLimit(cutoff)`
- `solver.solveAndRestart(LUBY, 300)`

Prototyping: search

The parameter tuning methods are solver dependent

- Some methods are standardized (`solver.setTimeLimit(cutoff)`)
- Most are not

Example (Mistral)

- `solver.setHeuristic('MinDomain', 'RandomSplit')` Set the variable ordering to 'minimum current domain' first, and branch by splitting the domain around a random pivot
- `solver.guide(solution)` Branches by first trying values of an earlier (good) solution
- `solver.solveAndRestart(LUBY, 300)` Set the restart policy to the Luby sequence with a base of 300

Prototyping: solvers

Solvers behave very differently

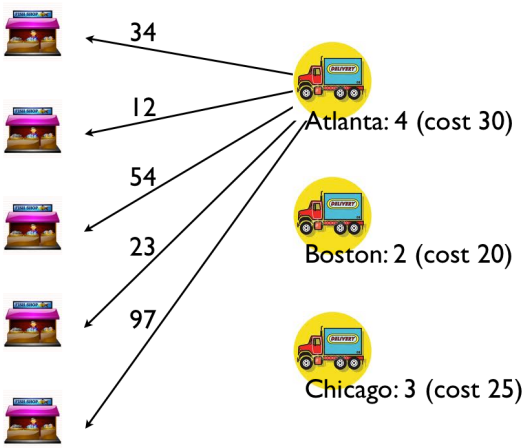
- Big difference between CP, SAT and MIP solvers in Numberjack
- Difficult to always know which paradigm is best
- Numberjack allows rapid prototyping between solvers

One model to rule them all

- The same model can be used in all solvers
- Quickly giving feedback as to which approach seems best
- Example: Warehouse allocation problem

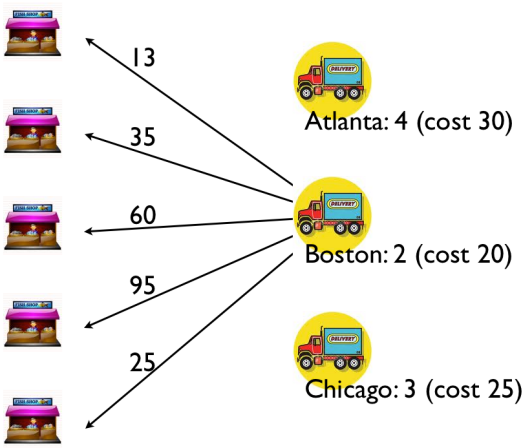
Warehouse Location Problem

- Warehouses
- Shops
- Transport cost
- Cost per warehouse
- Capacity per warehouse
- Shops must be supplied



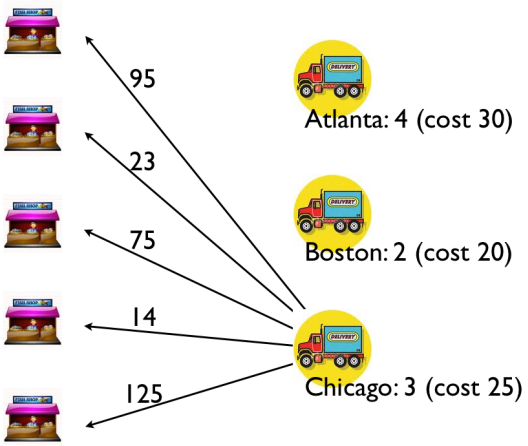
Warehouse Location Problem

- Warehouses
- Shops
- Transport cost
- Cost per warehouse
- Capacity per warehouse
- Shops must be supplied



Warehouse Location Problem

- Warehouses
- Shops
- Transport cost
- Cost per warehouse
- Capacity per warehouse
- Shops must be supplied



Warehouse Allocation Problem

The Variables

- Binary variables representing whether a warehouse is open
- Binary variables for each shop, warehouse pair representing if the shop is supplied by that warehouse

The Constraints

- Maintain warehouse capacity
- Ensure each shop is supplied
- A warehouse must be open in order to supply a shop
- Minimise the Cost

Results on warehouse allocation

SCIP is far better!

Table: SCIP vs Mistral (Warehouse Allocation).

Instance	SCIP			Mistral		
	Objective	Nodes	Time (s)	Objective	Nodes	Time (s)
cap44.dat	1184690	1	0.84	1468957	10008044	>3600
cap63.dat	1087190	14	1.82	1388391	10683754	>3600
cap71.dat	957125	1	0.69	1297505	11029722	>3600
cap81.dat	811324	1	0.65	1409091	3497095	>3600
cap131.dat	954894	5	5.30	1457632	1281009	>3600

Prototyping is useful!

One model many solvers

- Being able to run the same model on solvers quickly and easily give fast feedback as to what seems to be the most promising approach
- Quickly see that the Warehouse problem is solved quickly with SCIP, whereas Mistral runs into a wall...

Results on various puzzles

SCIP is worse!

Table: Solver Time vs Python Time (Arithmetic puzzles).

Instance	Mistral Time (s)		MiniSat Time (s)		SCIP Time (s)	
	Solver	Python	Solver	Python	Solver	Python
Costas (6)	0.0004	0.0046	0.0030	0.0043	4.0855	0.0084
Costas (7)	0.0006	0.0056	0.0054	0.0055	8.2916	0.0096
Costas (8)	0.0008	0.0071	0.0101	0.0070	11.4558	0.0111
Costas (9)	0.0020	0.0094	0.0229	0.0097	25.9925	0.0136
Costas (10)	0.0058	0.0140	0.0368	0.0113	49.1780	0.0152
Costas (11)	0.0064	0.0127	0.0863	0.0129	249.7409	0.0167
Costas (12)	0.0581	0.0155	0.3022	0.0160	199.0003	0.0196
Golomb (3)	0.0002	0.0025	0.0003	0.0020	0.0141	0.0067
Golomb (4)	0.0003	0.0032	0.0016	0.0028	0.0216	0.0076
Golomb (5)	0.0006	0.0038	0.0093	0.0038	0.9819	0.0081
Golomb (6)	0.0028	0.0048	0.0848	0.0051	2.9578	0.0089
Golomb (7)	0.0354	0.0072	1.0669	0.0064	18.5908	0.0105
Golomb (8)	0.2491	0.0078	15.7713	0.0081	804.8193	0.0119
Golomb (9)	3.4026	0.0105	375.0718	0.0106	-	-

Procedural vs. Declarative search programming

- There is no "search language" in Numberjack
  - Such language is difficult to implement
  - Moreover, it is difficult to use
- The back-end solvers can be called from Python
  - Calling `solve()`
  - Calling `reset()`
  - Setting up parameters
  - Getting statistics
- Finer grained control of the solver is possible

Programming Search

Search Primitives

- `solver.propagate()`: Reach a fixed point for some inference method.
- `solver.save()`: Save the current state.
- `solver.post( decision )`: Where `decision` is a unary constraint:
  - `x == v`, `x > v`, ...
- `solver.deduce()`: Post the complement of the decision taken at this level:
  - resp. `x != v`, `x <= v`, ... ("right" branch)
- `solver.undo()`: Restore the last saved state.

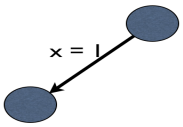
Programming Search

Full Binary Depth First Search in Numberjack

```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility: solver.deduce()
```


Programming Search

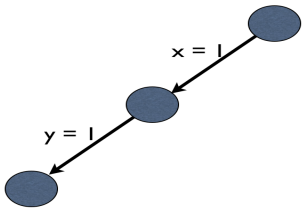
Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility: solver.deduce()
```

Programming Search

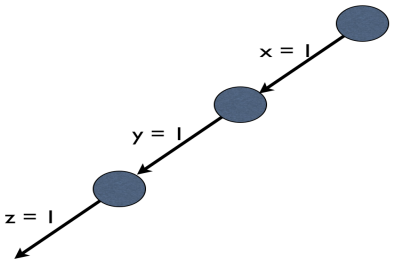
Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility: solver.deduce()
```


Programming Search

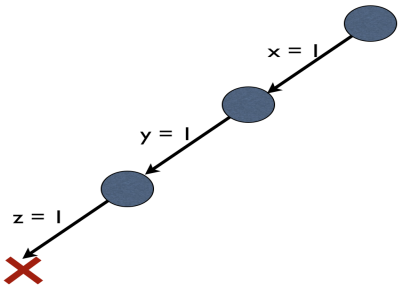
Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility: solver.deduce()
```

Programming Search

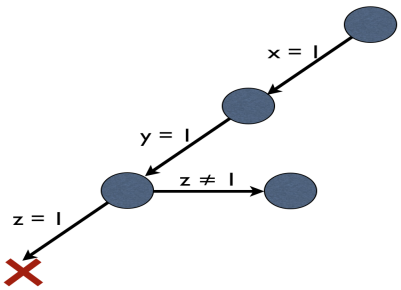
Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility: solver.deduce()
```


Programming Search

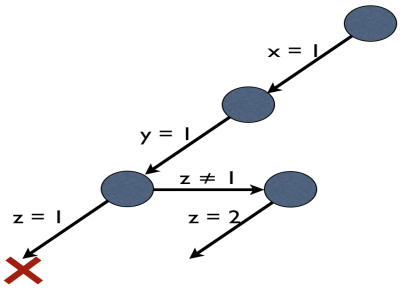
Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility: solver.deduce()
```

Programming Search

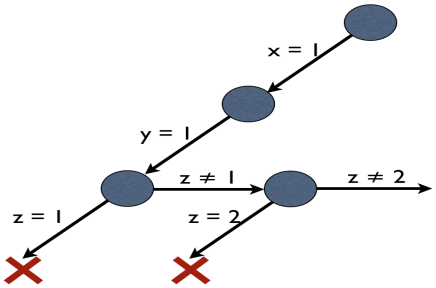
Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility: solver.deduce()
```


Programming Search

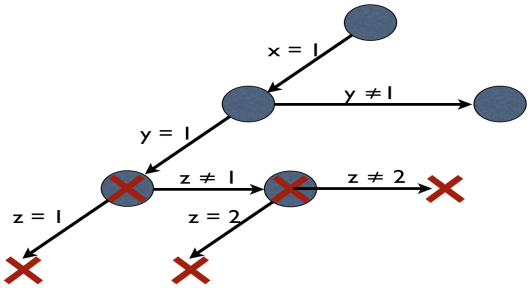
Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility:solver.deduce()
```

Programming Search

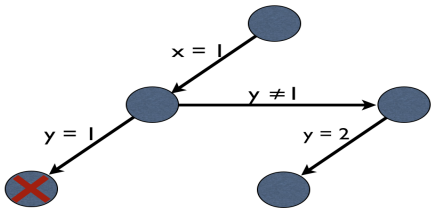
Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility:solver.deduce()
```


Programming Search

Full Binary Depth First Search in Numberjack




```
while solution is None and not proven_infeasibility:
    if solver.propagate(): ## left branch
        x = select(Xs, var_ordering)
        if x is None: solution = solver.get_solution()
        else:
            solver.save()
            solver.post( x == x.get_min() )
    else: ## right branch
        proven_infeasibility = not solver.undo()
        if not proven_infeasibility: solver.deduce()
```

K Branching

Mistral and MiniSat do not provide a K-branching DFS

```
def depth_first_k_branching(solver, Xs, var_ordering=MinDomain):
    if solver.propagate():
        x = select(Xs, var_ordering)
        if x is None: return solver.get_solution()
        else:
            for v in x:
                solver.save()
                solver.decide( x == v )
                outcome = depth_first_k_branching(solver,
                                                  filter(undecided,Xs),
                                                  var_ordering)
                if outcome == False: solver.undo()
                else: return outcome
    return False
```

- Variable and branching heuristics are parameters

Limited Discrepancy Search

[Harvey & Ginsberg 95]

```
def lds_probe(solver, Xs, var_ordering, good_choice, max_discrepancy):
    solution = None
    proven_infeasibility = False
    exhausted_discrepancies = False
    discrepancy = [0]
    while solution is None and not proven_infeasibility and not exhausted_discrepancies:
        if solver.propagate():
            x = select(Xs, var_ordering)
            if x is None: solution = solver.get_solution()
            else:
                discrepancy.append(current(discrepancy))
                solver.branch_left( good_choice(x) )
        else:
            proven_infeasibility = not solver.undo()
            if not proven_infeasibility:
                discrepancy.pop()
                while not exhausted_discrepancies and current(discrepancy) >= max_discrepancy:
                    if not solver.undo(): exhausted_discrepancies = True
                    else: discrepancy.pop()
                if not exhausted_discrepancies:
                    increment(discrepancy)
                    solver.deduce()
    return (solution, proven_infeasibility)
```

Numberjack controlled search, why?

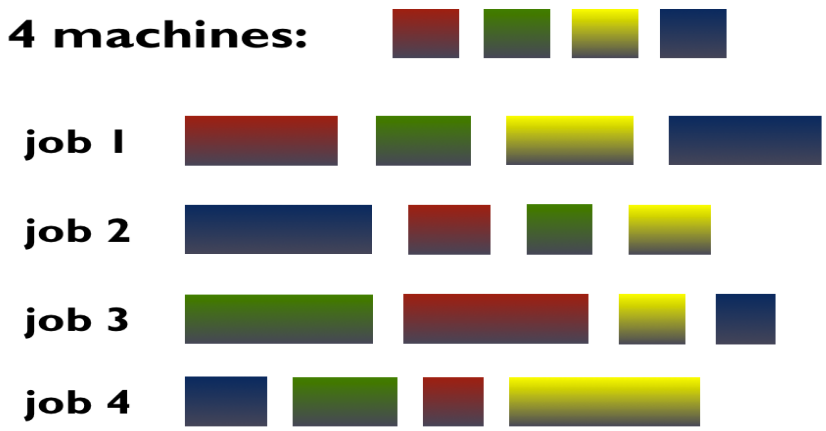
Proof of concept for your ideas

- Quick proptotyping of user defined strategies
  - Variable ordering and branching heuristics
  - Optimistic bounds
- First step toward hybridization (way of communication between solvers)
- Search visualisation and analysis
- Useful as a teaching tool

Jobshop Scheduling

Notations

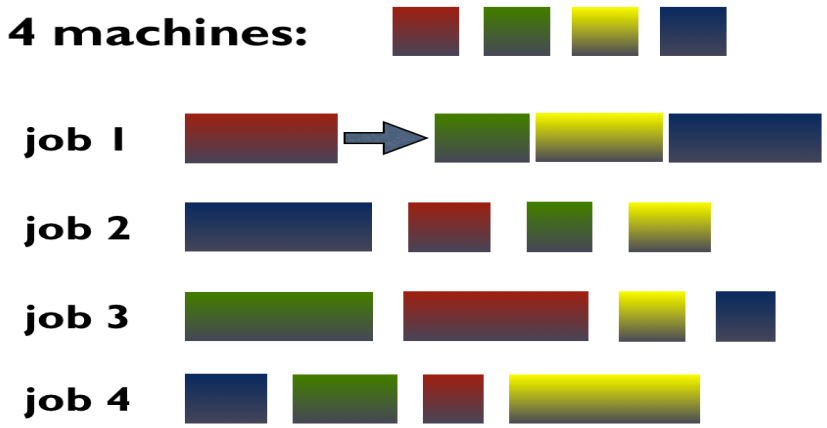
- n Jobs (sequence of tasks), m Machines
- m Tasks in each job
  - Associated to a machine (color)
  - And a processing time (length)



Jobshop Scheduling

Precedence constraints

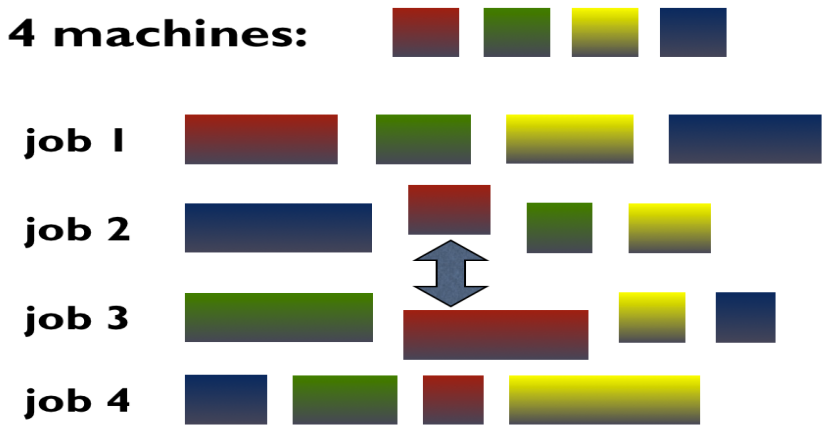
- The tasks must run in sequence for each job
- $t_{ij} + p_{ij} \leq t_{ij+1}$



Jobshop Scheduling

Resource constraints

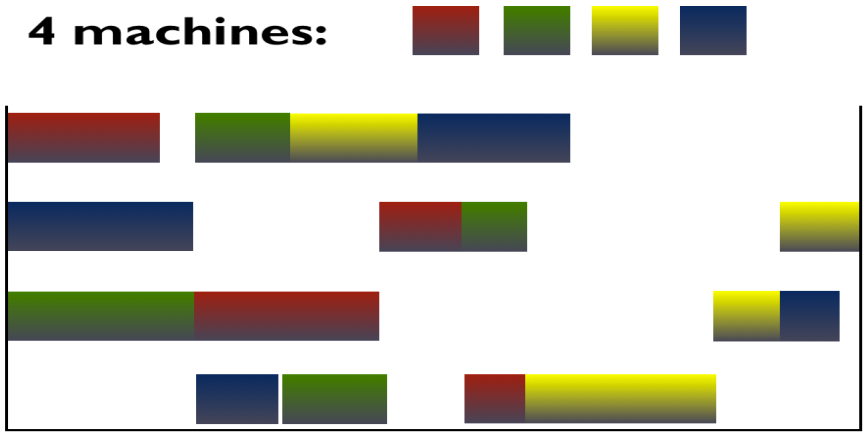
- Two tasks sharing the same resource cannot overlap
- $t_{ab} + p_{ab} \leq t_{xy}$ *or* $t_{xy} + p_{xy} \leq t_{ab}$



Jobshop Scheduling

Solution

- Objective: minimize the makespan (finishing time of the last job)



Jobshop Scheduling

Numberjack Model

- Setting up the variables

```
jsp = ...

lb = jsp.lower_bound()
ub = jsp.upper_bound()

C_max = Variable(lb, ub)

Jobs = Matrix([[Task(ub, p) for p in job] for job in jsp.job])
```

Jobshop Scheduling

Precedence constraints

- The tasks must run in sequence for each job

```
[[job[i] < job[i+1] for i in range(jsp.nMachines-1)] for job in Jobs],
```

Resource constraints

- Two tasks sharing the same resource cannot overlap

```
[UnaryResource([Jobs[m] for m in machine]) for machine in jsp.machine],
```

Objective function

- Minimize the makespan (the maximum end time of any job)

```
[job[-1] < C_max for job in Jobs],

Minimise( C_max )
```


Jobshop Scheduling

Numberjack Model

```
jsp = ...

lb = jsp.lower_bound()
ub = jsp.upper_bound()

C_max = Variable(lb, ub)

Jobs = Matrix([[Task(ub, p) for p in job] for job in jsp.job])

model = Model(
    [UnaryResource([Jobs[m] for m in machine]) for machine in jsp.machine],
    [[job[i] < job[i+1] for i in range(jsp.nMachines-1)] for job in Jobs],

    [job[-1] < C_max for job in Jobs],

    Minimise( C_max )
    )
```

Jobshop Scheduling

Solving: the basic way

```
import Mistral
solver = Mistral.Solver(model)
solver.solve()
```

Solving: more advanced

```
import Mistral
solver = Mistral.Solver(model)

solver.setHeuristic('Scheduling','Promise',2)
solver.solveAndRestart(GEOMETRIC, 256, 1.3)
```


Jobshop Scheduling

This is still not enough

- The initial upper bound is very poor
- Trashing on sub-optimal makespan

Dichotomic search

- Use dichotomic search to obtain better bounds quicker
  - As long as $lb(C_{max}) < ub(C_{max})$, we solve the decision problem where $C_{max} = \frac{lb+ub}{2}$
  - When finding a solution $ub(C_{max})$ is updated
  - When reaching the limit $lb(C_{max})$ is updated
- Use the best solution found so far to guide the branching choices

Jobshop Scheduling

Dichotomy

```
def dichotomic_search(model, solver, max_infeasible, min_feasible, cutoff):
    lb = max_infeasible
    ub = min_feasible
    best_solution = None

    while lb+1 < ub:
        C_max = int((lb + ub) / 2)
        (feasible, solution, C_max) = solve(model, solver, C_max,
                                            best_solution, cutoff)

        if feasible:
            ub = C_max
            best_solution = solution
        else:
            lb = C_max
            if feasible is not None:
                max_infeasible = C_max

    min_feasible = ub
    return (max_infeasible, min_feasible, best_solution)
```

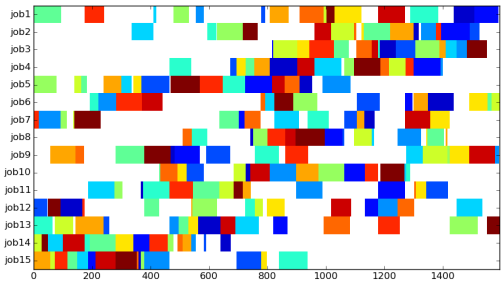

Jobshop Scheduling

Dichotomy cont'd

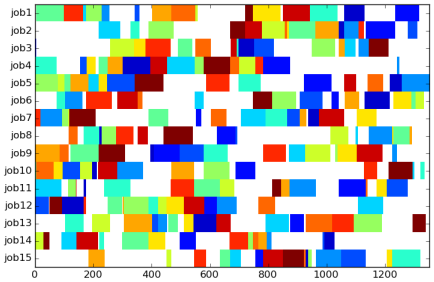
```
def solve(model, solver, C_max, best_solution, cutoff):
    solver.save() ## save since the bounds are not "safe"
    for task in model.tasks: solver.post(task < C_max)

    ## solution guiding
    if best_solution != None: solver.guide(best_solution)
    solver.setTimeLimit(cutoff)
    solver.solveAndRestart(GEOMETRIC, 256, 1.3)

    ## analyze the outcome
    outcome = (None, None, C_max)
    if solver.is_sat():
        new_C_max = max([task.get_min() + task.duration for task in model.tasks])
        outcome = (True, solver.get_solution(), new_C_max)
    elif solver.is_unsat(): outcome = (False, None, C_max)

    ## reset the solver to its previous state
    solver.reset()
    solver.undo()

    return outcome
```

Jobshop Scheduling

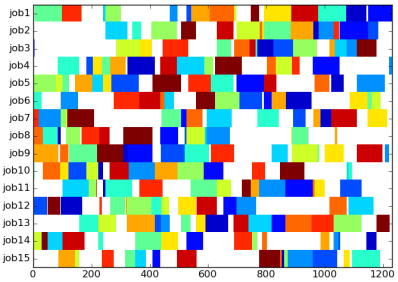
declare & solve



scheduling heuristics & restarts



dichotomy & solution guiding



Water Retention Magic Squares

- Water retention (Craig Knecht 07)
- Magic square = 3D histogram
- Pour water on the histogram
- Maximize the amount of water

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

16	3	2	13
5	10	11	8
9	6+3	7+2	12
4	15	14	1

Solving the Water Retention Magic Square Problem

Problems

- Modelling was relatively easy
- Hard to solve
  - Magic Squares are not easy to find for CP (worse for SAT and MIP)
  - Getting good upper bounds for the objective function is hard
    - Any breach, and all the water is gone!

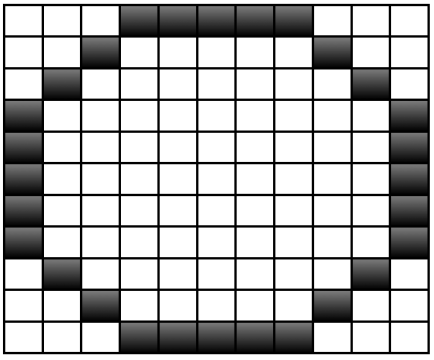
Solution

- Randomization helps a lot with finding Magic Square (Heavy tail - Gomes & Sellman)
- There is an "obvious" strategy to find Magic Square that retains a large amount of water
  - Building a wall, i.e., high values toward the edges

Solving the Water Retention Magic Square Problem

Building a wall

- The wall should not span over a whole row or column
- We want it as high as possible
  - It might makes it difficult to find a Magic Square

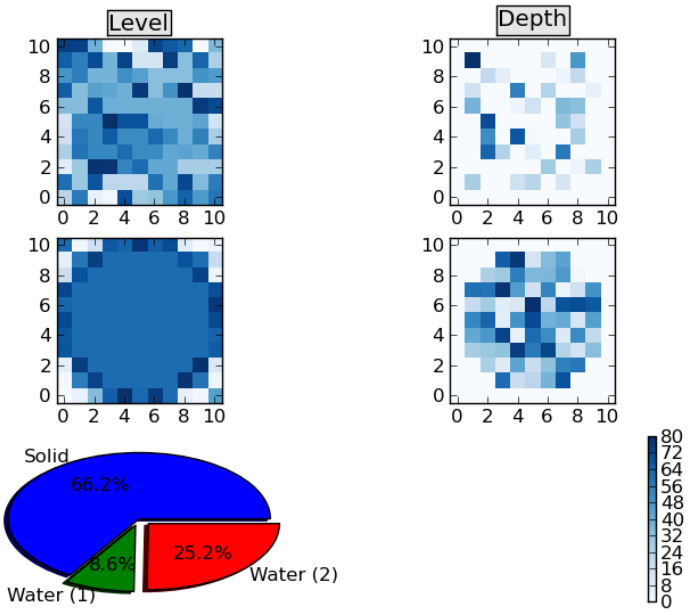


Strategy

- `solver.save()`
- `for bric in wall:  solver.post(bric > K)`
- `solver.solve()`
- `solver.reset()`
- `solver.undo()`
- `K -= 1`

Water Retention Test

- Default search (1):
- Wall search (2):
- Water Retention:



Applications of Numberjack

This section will cover

- Taking advantage of Python
- Example applications of Numberjack

Taking advantage of Python

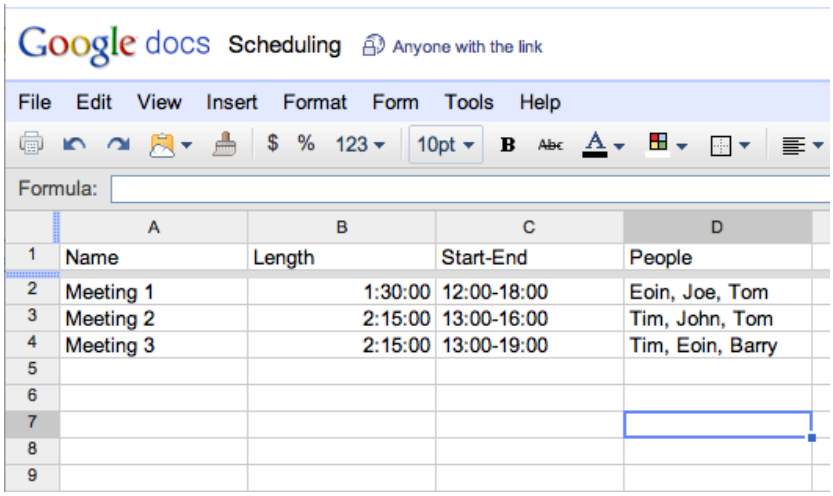
Python is widely supported

- There are many APIs available for Python
- Lots of online sites provide Python
- Fast to develop and easy to use

Case studies

- Scheduling for Google Calendar
- Product configuration on Google AppSpot
- Geo-location with Basemap

Scheduling with Google Calendar



	A	B	C	D
1	Name	Length	Start-End	People
2	Meeting 1	1:30:00	12:00-18:00	Eoin, Joe, Tom
3	Meeting 2	2:15:00	13:00-16:00	Tim, John, Tom
4	Meeting 3	2:15:00	13:00-19:00	Tim, Eoin, Barry
5				
6				
7				
8				
9				

Problem

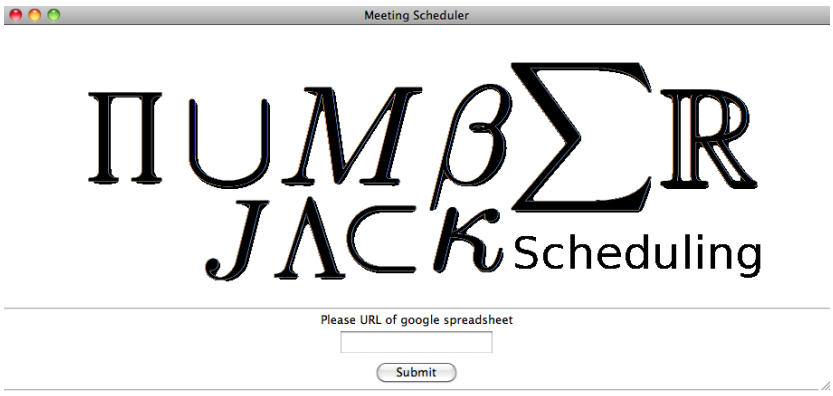
- Multiple people in an office need to schedule meeting
- The required meetings are specified in a google spreadsheet

Scheduling with Google Calendar

Build scheduling application with CP brain

- Python APIs to integrate with spreadsheet and calendar
- Use Python Qt for GUI
- Use Numberjack for decision logic
- One afternoon work

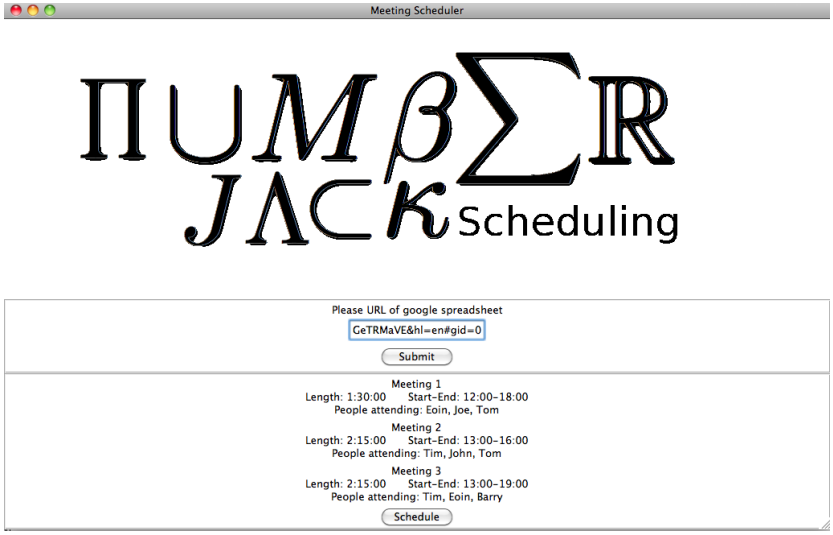
Scheduling with Google Calendar



First Step

- User enters URL of spreadsheet used to specify meetings

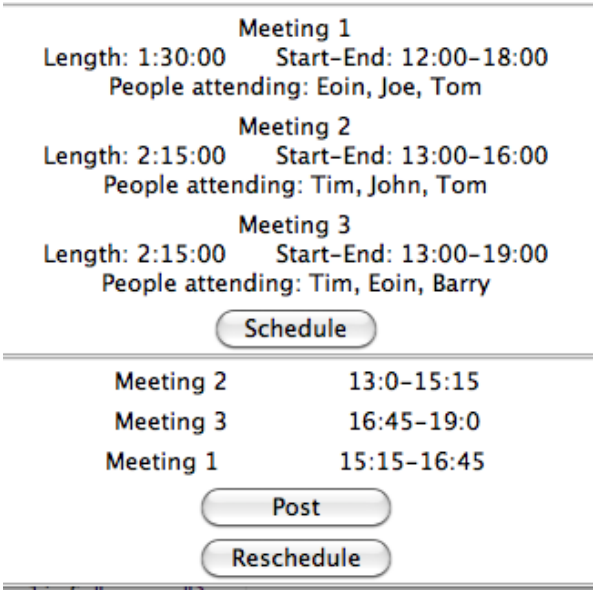
Scheduling with Google Calendar



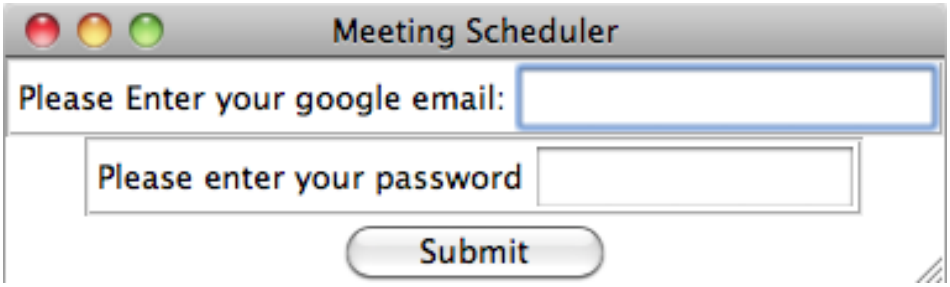
- Application retrieves and parses the meeting information
- Provides the user with a list of meetings to schedule

Scheduling with Google Calendar

- Feasible schedule is computed
- The user is presented with this schedule
- Another solution can be computed
- Schedule can be posted to Google Calendar



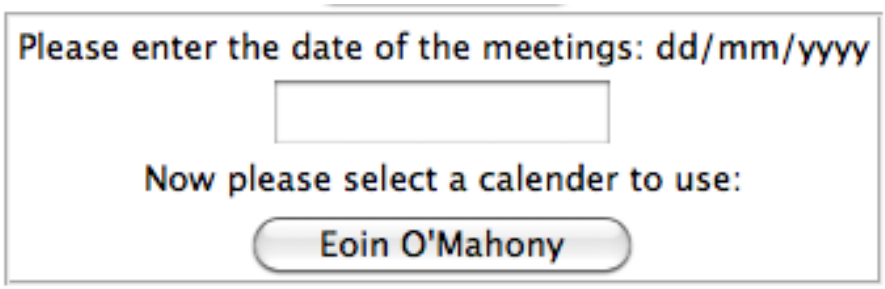
Scheduling with Google Calendar



Submit your details

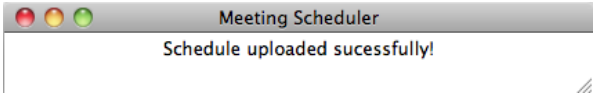
- Enter your google account information to upload chosen schedule

Scheduling with Google Calendar



Now select date

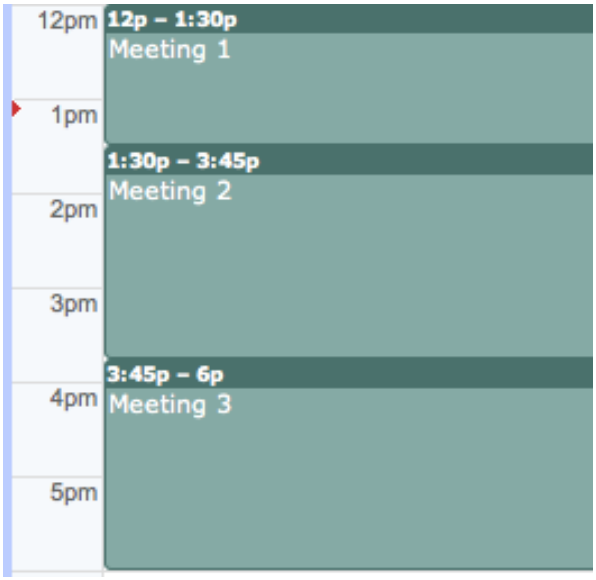
- Enter the date the meetings are to occur
- Select the Google calendar to be used



Scheduling with Google Calendar

Solution posted online

- Constraint technology embedded into online application
- Application developed quickly and easily



Product Configuration

Problem Definition

- Product has many options
- Some are not compatible with others
- Aid user to configure their product within the constraints

Product Configuration

Numberjack Application

- Web service hosted on Google app engine
- Uses Python based constraint solver
- Google spreadsheet to specify problem
- Web front end to configure products
- Computation handled in the cloud

Product Configuration

Specifying the configuration problem

- Specified online in a Google spreadsheet
- Categories have choices which have multiple options
- Constraints are specified in terms of allowed tuple constraints between configuration choices

Google docs TestConfig Anyone with the link

File Edit View Insert Format Form Tools Help

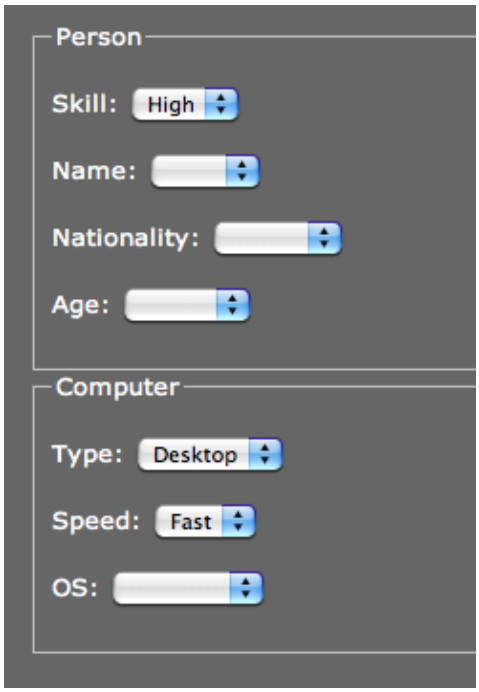
Formula: Slow

	A	B	C
1	I		
2	Variables		
3	Person		
4	Skill	High	Low
5	Name	John	Mary
6	Nationality	Italian	French
7	Age	Young	Old
8	Computer		
9	Type	Desktop	Laptop
10	Speed	Fast	Slow
11	OS	Mac	Windows
12	End		
13	Constraints		
14	Allowed	Skill	Speed
15		High	Fast
16		Low	Slow
17	Allowed	Speed	Type
18		Slow	Laptop

Product Configuration

Using the online configurator

- Given the URL of the spreadsheet the online app produces a web site for the configuration problem
- As the user makes choices, these decisions are propagated in the cloud
- The site updates with each choice made and the user is informed of options no longer available



Geolocation with Basemap

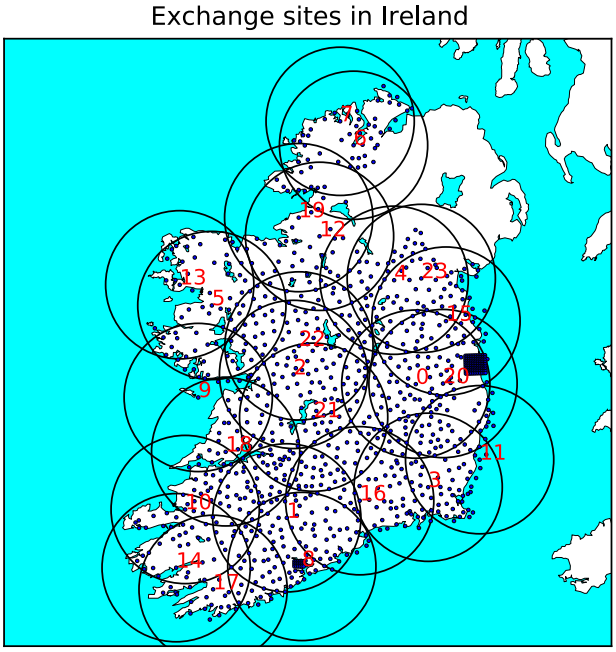
Basemap

- Python module that allows for plotting in 2D maps
- Uses geo-location information such as GPS co-ordinates
- Can be used to visualise geographical data

Geolocation with Basemap

Super node placement problem

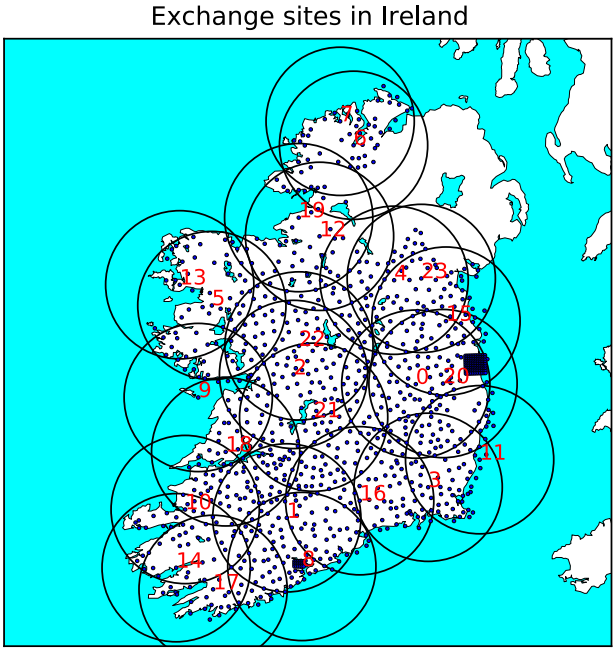
- Each blue dot represents an existing node
- Place super nodes such that each existing node is covered by at least two super nodes
- Minimise number of super nodes used



Geolocation with Basemap

Why is this useful?

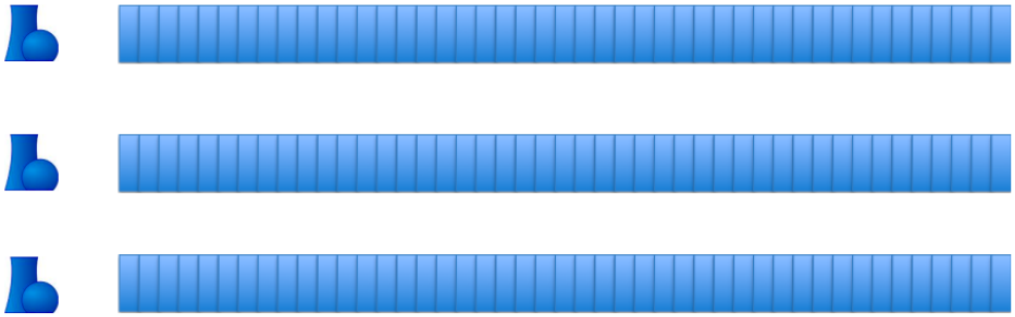
- Allows easy visualisation of search solutions
- Often observing solution gives intuition as to what should be done
- Anyone notice anything about the placement of the super nodes?



Roadef Challenge 2010

The challenge

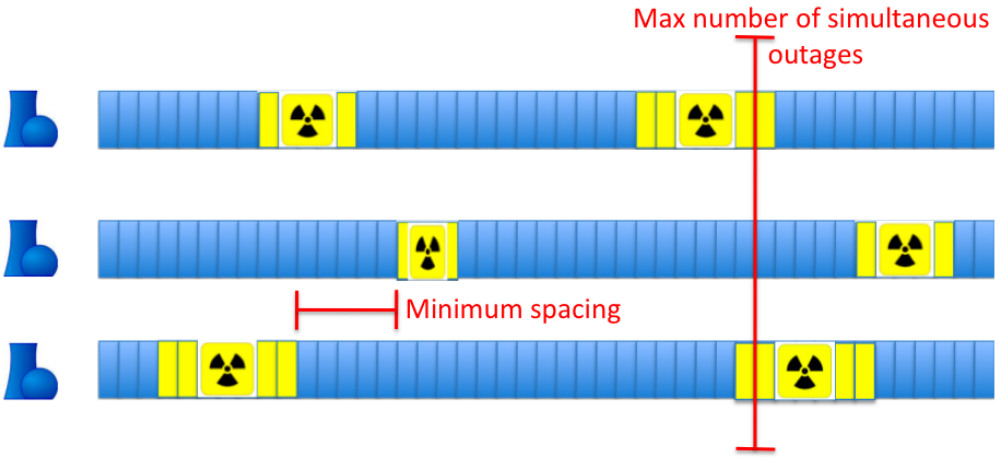
- A Large-scale Energy Management Problem
  - Organized by the French Operation Research Society (C. Artigues, E. Bourreau, M. Asfar, E. Ozcan)
  - Proposed by EDF, 44 teams participated worldwide
- Planning the production, refueling and maintenance of thermal power plants (nuclear or otherwise)



Roadef Challenge 2010

Outages scheduling

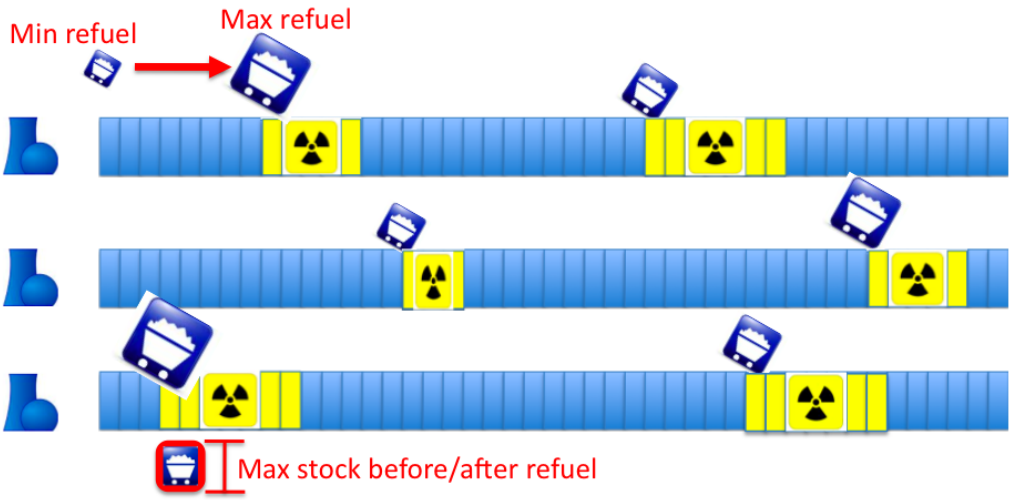
- Nuclear power plants outages
  - Subject to various resource constraints



Roadef Challenge 2010

Refueling

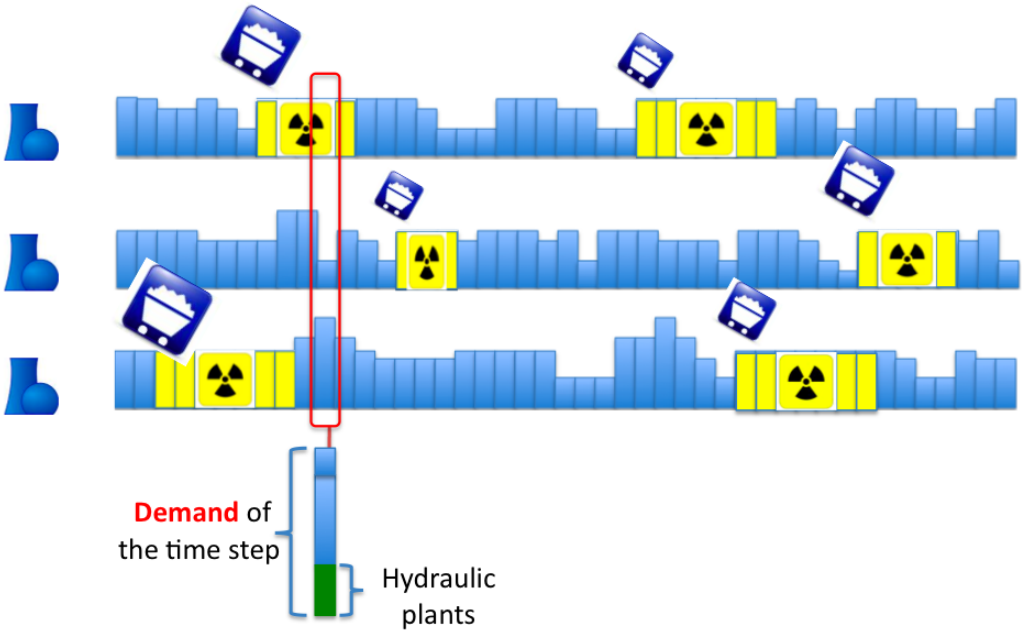
- Channelling stocks, refueling quantity, production output



Roadef Challenge 2010

Demand

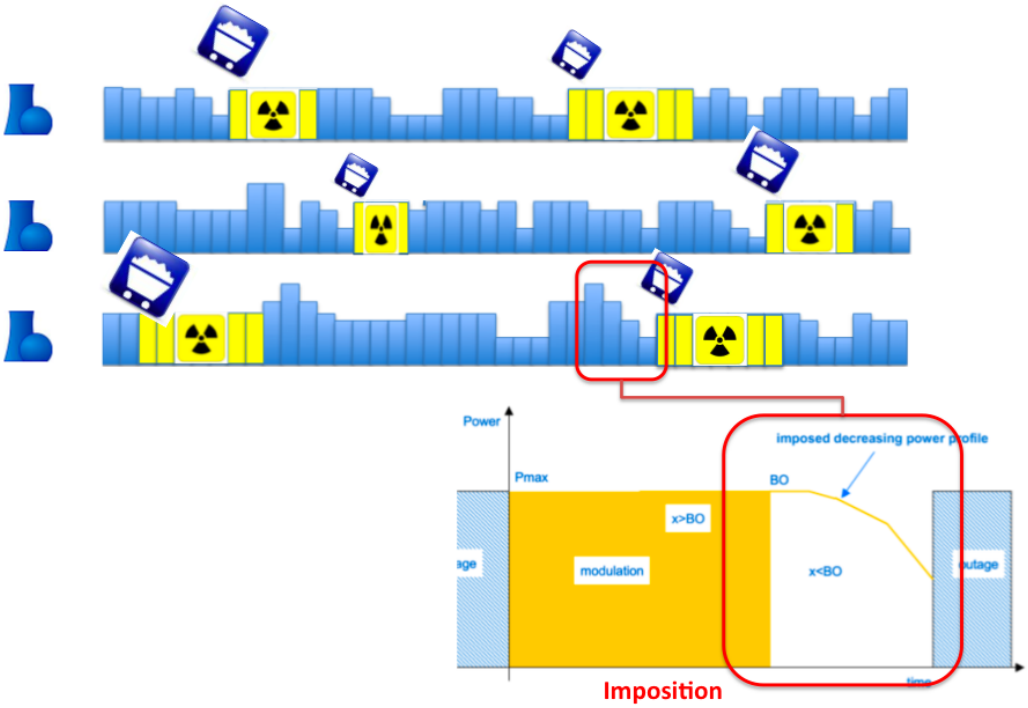
- Plan the production in order to meet the demand



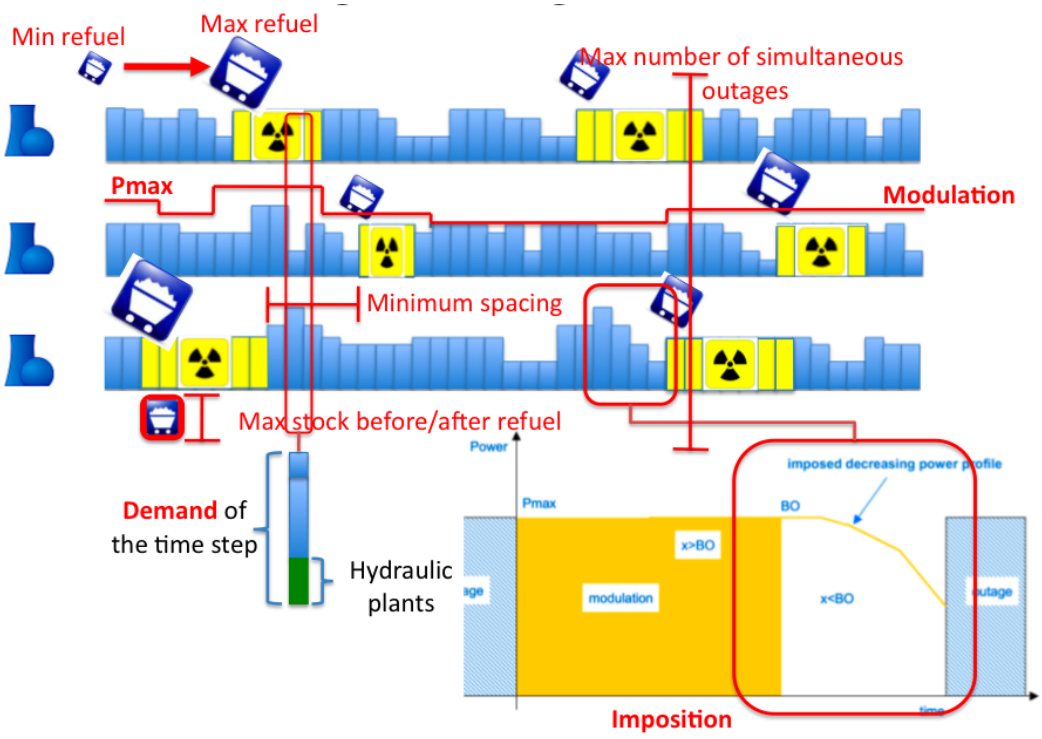
Roadef Challenge 2010

Production profile

- When the stock is too low, the production must decrease within certain bounds



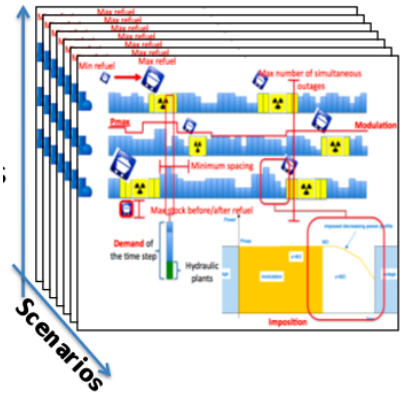
Roadef Challenge 2010



Roadef Challenge 2010

Data

- 56 Nuclear plants
- 20 Hydraulic plants
- 5800 timesteps
- 121 stochastic scenarios



Problem size

- More than 50,000,000 decision variables, with either continuous or large domains
- Each solution needs about 1 GigaByte of memory!

Roadef Challenge 2010

Our approach

```
initial_solutions = self.algorithm.get_initial_solutions(strategy,K)
if len(initial_solutions):
    good_solutions = []
    best_cost = self.algorithm.best_solution.get_cost()
    for sol in initial_solutions:
        if sol.get_cost() < best_cost*tolerance: good_solutions.append(sol)
    self.algorithm.improve_solutions(good_solutions, strategy)
```

Roadef Challenge 2010

Our approach

```
def improve_solutions(self,good_solutions,strategy):
    ## some initialisation
    while not stop:
        # if there is a best solution, guide the next run with it
        ideal = self.pool_full.select(self.last_solution)
        self.last_solution.guide(self.solver,1-heat)
        select_cost = self.last_solution.get_cost()

        # and cool the alloy
        heat *= strategy.cooling_factor

        self.solver.solveAndRestart(GEOMETRIC, 64, 1.1, 0.0)

        a_schedule_has_been_found = self.solver.is_sat()
        if a_schedule_has_been_found:

            self.last_solution.update()
            self.solve_scenarios()

            if self.last_solution.isActive():
                self.pool_full.insert(self.last_solution,(select_cost > self.last_solution.get_cost()))
                if  (self.last_solution.get_cost() < self.best_solution.get_cost()):
                    self.last_solution.deepCopy(self.best_solution)
            else:
                if (self.last_solution.isSound()): self.computeCut(-1)
        self.solver.reset()
```


Contributing

Extending Numberjack

- Write models
- Write Decompositions
- Add in Solvers

Plugging in a new solver

- The solver has to be in C/C++
- Basic template of wrapper file
  - Callbacks for adding constraints/variables
  - Callbacks for solver accessors
  - 2500 lines of codes, and only a fraction is solver dependent
- Swig does the rest!

What we set out to achieve

Motivation

Combinatorial optimization provides powerful support for decision-making; many interesting real-world problems are combinatorial and can be approached using similar techniques.

The Promise

This tutorial will teach attendees how to develop interesting models of combinatorial problems and solve them using constraint programming, satisfiability and mixed integer programming techniques.

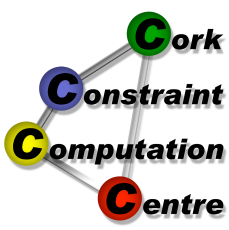
Numberjack

We have developed Numberjack and released it under the LGPL license. It's an open-source project. Please consider getting involved for the benefit of academia, research and industry.

Constraint Programming and Combinatorial Optimisation in Numberjack

Emmanuel Hebrard, Eoin O'Mahony, Barry O'Sullivan

Cork Constraint Computation Centre, University College Cork
This work was supported by Science Foundation Ireland (Grant No. 05/IN/I886).



July 15, 2010