CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Decision Analytics

Lecture 10-11: Satisfiability and Linear Constraints

Constraint Satisfaction Problem

- A **Constraint Satisfaction Problem** (CSP) is defined by
  - A tuple of n variables
    $$X =< x_1, \dots, x_n >$$
  - a corresponding tuple of domains
    $$D =< D_1, \dots, D_n >$$
    defining the potential values each variable can assume
    $$x_i \in D_i$$
  - and a tuple of t constraints
    $$C =< C_1, \dots, C_t >$$
    each being defined itself by a tuple
    $$\mathrm{C}_i =< R_{S_i}, S_i >$$
    - comprising the scope of the constraint $S_i \subset X$, being the subset of variables the constraint operates on
    - and the relation $R_{S_i} \subset D_{s_{i_1}} \times \cdots \times D_{s_{i_{|S_i|}}}$, being the set of valid variable assignments in the scope of the constraint

Constraint Satisfaction Problem

- So far we have looked at Boolean domains

$$D_i = \{0,1\}$$

```
model = cp_model.CpModel()
model.NewBoolVar("b")
```

- And Boolean constraints

$$R_{S_i} = \{< x_{s_{i_1}}, \ldots, x_{s_{i_{|s_i|}}} > \mid T[x_{s_{i_1}}, \ldots, x_{s_{i_{|s_i|}}}] = True\}$$

```
model.AddBoolAnd([x1])
model.AddBoolOr([x1.Not(),x2,x3])
model.AddBoolAnd([x1,x2,x3]).OnlyEnforceIf(x4)
```

Constraint Satisfaction Problem

- Now we are going to generalise this towards **bound integer domains**

$$D_i = \{b_l, \dots, b_u\}$$

```
model.NewIntVar(bl, bu, "x")
```

- And **linear constraints** (both positive and negative)

$$\begin{array}{lll} a^T x = b & a^T x \leq b & a^T x \geq b \\ a^T x \neq b & a^T x > b & a^T x < b \end{array}$$

```
model.Add(0.2*x1 + 3.5*x2 - 4*x3 < 3)
```

Constraint Satisfaction Problem

- Boolean variables can directly occur in the linear constraints with the convention that False=0 and True=1

- To integrate the linear constraints with the SAT problem we need to introduce **channelling constraints**

$$y \Rightarrow \; a^T x = b$$
$$\neg y \Rightarrow a^t x \neq b$$

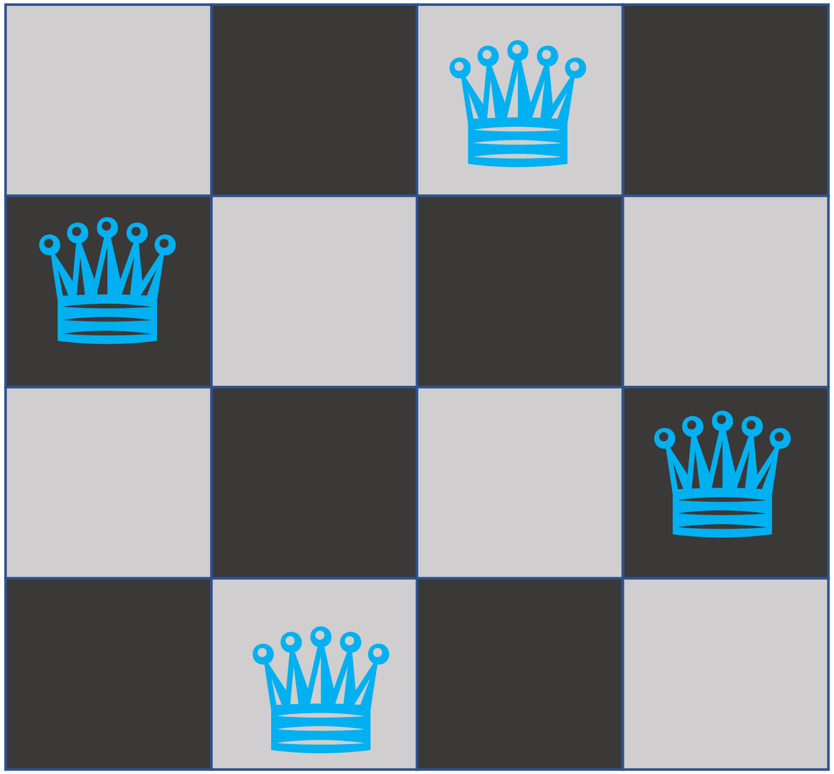
```
model.Add(a1*x1 + a2*x2 == b).OnlyEnforceIf(y)
model.Add(a1*x1 + a2*x2 != b).OnlyEnforceIf(y.Not())
```

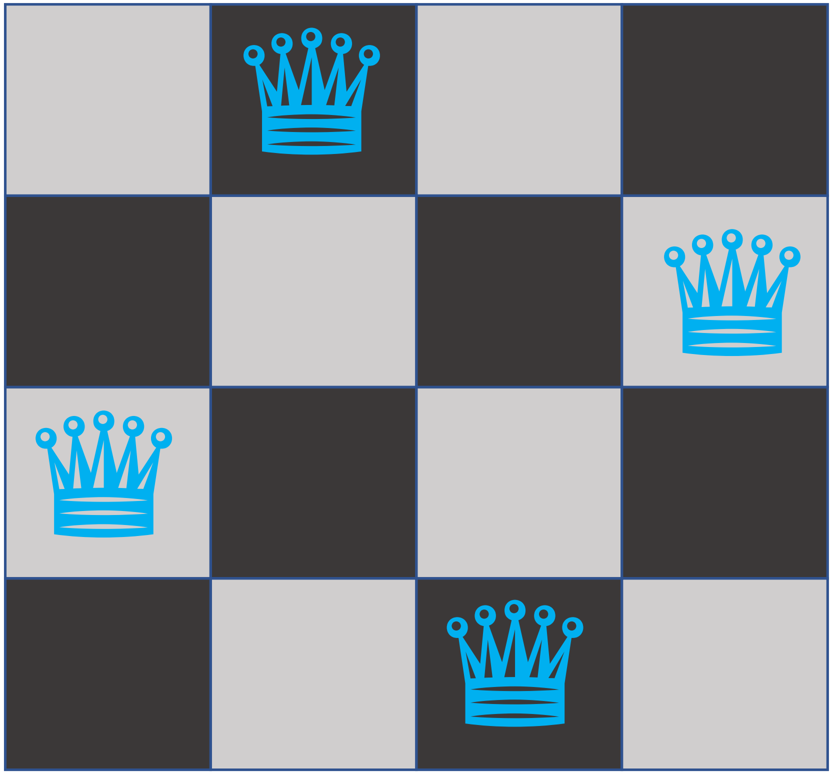
- The Boolean variable y is now equivalent to the linear constraint and we can use it in a Boolean SAT model
- If we use only one of the two constraints it is called **partial channelling**

Example: The N-Queens puzzle

Can we place 4 queens on a 4x4 chessboard so that they do not threaten each other?

What happens when we generalise this problem to N queens on a NxN board?





First attempt

- Let's start with what we know already: SAT

- Each field can be occupied by a queen or not, therefore we could define one Boolean variable per field

$$X = < x_{11}, \dots, x_{1N}, \dots, x_{N1}, \dots, x_{NN} >$$

x_{11}	x_{12}	x_{13}	x_{14}
x_{21}	x_{22}	x_{23}	x_{24}
x_{31}	x_{32}	x_{33}	x_{34}
x_{41}	x_{42}	x_{43}	x_{44}

```
field = []
for i in range(N):
    row = []
    for j in range(N):
        row.append(model.NewBoolVar(str(i)+"_"+str(j)))
    field.append(row)
```

First attempt

- If a field is occupied it implies that neither vertically nor horizontally any field is occupied

$$\forall x_{ij} \forall x_{ik} \colon x_{ij} \Rightarrow \neg x_{ik}$$
$$\forall x_{ij} \forall x_{kj} \colon x_{ij} \Rightarrow \neg x_{kj}$$

```
for i in range(N):
   for j in range(N):
       horizontal = []
       vertical = []
       for k in range(N):
           if j!=k:
               horizontal.append(field[i][k].Not())
           if i!=k:
               vertical.append(field[k][j].Not())
       model.AddBoolAnd(horizontal).OnlyEnforceIf(field[i][j])
       model.AddBoolAnd(vertical).OnlyEnforceIf(field[i][j])
```

First attempt

- Also diagonally fields cannot be occupied

$$\forall x_{ij} \forall x_{i-k,j-k} : x_{ij} \Rightarrow \neg x_{i-k,j-k}$$
$$\forall x_{ij} \forall x_{i+k,j+k} : x_{ij} \Rightarrow \neg x_{i+k,j+k}$$
$$\forall x_{ij} \forall x_{i+k,j-k} : x_{ij} \Rightarrow \neg x_{i+k,j-k}$$
$$\forall x_{ij} \forall x_{i-k,j+k} : x_{ij} \Rightarrow \neg x_{i-k,j+k}$$

```
for i in range(N):
   for j in range(N):
       diagonal = []
       for k in range(1,N):
           if ((i-k)>=0) and ((j-k)>=0):
               diagonal.append(field[i-k][j-k].Not())
           if ((i+k)<N) and ((j+k)<N):
               diagonal.append(field[i+k][j+k].Not())
           if ((i+k)<N) and ((j-k)>=0):
               diagonal.append(field[i+k][j-k].Not())
           if ((i-k)>=0) and ((j+k)<N):
               diagonal.append(field[i-k][j+k].Not())
       model.AddBoolAnd(diagonal).OnlyEnforceIf(field[i][j])
```

First attempt

- Finally, we need to ensure that exactly N queens are on the board

- To do that we introduce $\binom{N}{N^2}$ new Boolean variables for each possibility of positioning N queens on a NxN size board
- We add constraints, that each of these is only true if the exact fields corresponding to the combination is occupied
- And finally a constraint that one of these combinations must be achieved

```
coords = []
for i in range(N):
    for j in range(N):
        coords.append((i,j))
queen_positionings = []
for queen_at in combinations(coords, N):
    queen_positioning = model.NewBoolVar(str(queen_at))
    queen_positionings.append(queen_positioning)
    for i in range(N):
        for j in range(N):
            if (i,j) in queen_at:
                model.AddBoolOr([field[i][j]]).OnlyEnforceIf(queen_positioning)
            else:
                model.AddBoolOr([field[i][j].Not()]).OnlyEnforceIf(queen_positioning)
model.AddBoolOr(queen_positionings)
```

First attempt

- How does this perform to find a solution and to find all solutions?

# of solutions: 1
N = 4
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 1836
conflicts: 0
branches: 21
propagations: 2025
integer_propagations: 220
walltime: 0.134439
usertime: 0.134439
deterministic_time: 0.00017556

# of solutions: 2
N = 4
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 1836
conflicts: 0
branches: 21
propagations: 2025
integer_propagations: 220
walltime: 0.13697
usertime: 0.13697
deterministic_time: 0.000176548

This is a lot of variables

First attempt

- What happens if we increase N?

# of solutions: 1
N = 5
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 53155
conflicts: 0
branches: 78
propagations: 54057
integer_propagations: 0
walltime: 5.68753
usertime: 5.68753
deterministic_time: 0.0044876

It takes more than 5sec to find a solution

# of solutions: 10
N = 5
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 53155
conflicts: 0
branches: 107
propagations: 54322
integer_propagations: 0
walltime: 6.84708
usertime: 6.84708
deterministic_time: 0.00452449

The problem size is only N=5, so this approach is infeasible

The number of variables increases exponentially!

First attempt, refinement

- The constraint that exactly N queens are on the board forced us to introduce $\binom{N}{N^2}$ additional Boolean variables
- Can we do any better?
- We do not need to restrict our constraints to Boolean expressions
- Instead we could introduce a linear integer constraint that the variables sum up to N

$$\sum_{i,j} x_{ij} = N$$

```
all_fields = []
for i in range(N):
    for j in range(N):
        all_fields.append(field[i][j])
model.Add(sum(all_fields)==N)
```

First attempt, refinement

- So how does this perform now?

```
# of solutions:  1
N =  5
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 25
conflicts: 0
branches: 57
propagations: 338
integer_propagations: 395
walltime: 0.029582
usertime: 0.029582
deterministic_time: 0.000115955
```

That looks better

```
# of solutions:  10
N =  5
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 25
conflicts: 25
branches: 137
propagations: 729
integer_propagations: 790
walltime: 0.0118591
usertime: 0.0118591
deterministic_time: 0.000846217
```

And we achieved a significant performance boost

First attempt, refinement

- What happens if we increase N again?

# of solutions: 1
N = 10
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 100
conflicts: 1
branches: 216
propagations: 3034
integer_propagations: 3250
walltime: 0.0099333
usertime: 0.0099333
deterministic_time: 0.00110848

Finding a solution is quick

# of solutions: 724
N = 10
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 100
conflicts: 15546
branches: 45851
propagations: 345383
integer_propagations: 277883
walltime: 7.65094
usertime: 7.65094
deterministic_time: 6.39418

The search space for finding all solutions is large

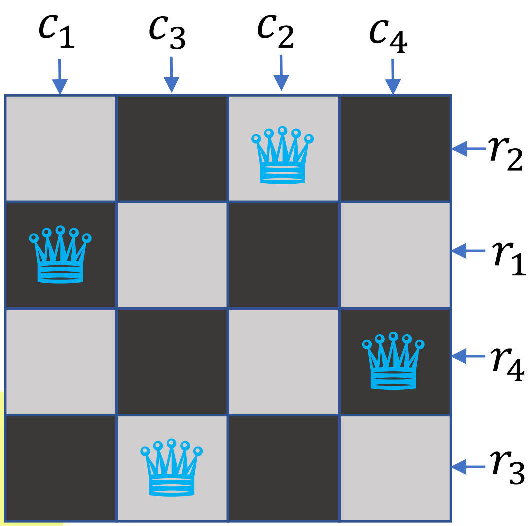
So it takes 7sec to list all solutions

Second attempt

- Can we do any better?
- We do not have to restrict out domains to Boolean
- Each of the N queens occupies one row/column on the board, so we can add a coordinate per queen as variables

$$X = < r_1, c_1, \dots, r_N, c_N >$$

$$D_{r_i} = D_{c_i} = \{0, \dots, N-1\}$$




```
row = []
col = []
for i in range(N):
    row.append(model.NewIntVar(0,N-1,"row"+str(i)))
    col.append(model.NewIntVar(0,N-1,"col"+str(i)))
```

Second attempt

- First, no pair of queens can share a row or a column

- This means that

$$\forall i, j : i \neq j \Rightarrow r_i \neq r_j \land c_i \neq c_j$$

- We can also use the fact that this is symmetric and remove redundant constaints

$$\forall i, j : i < j \Rightarrow r_i \neq r_j \land c_i \neq c_j$$

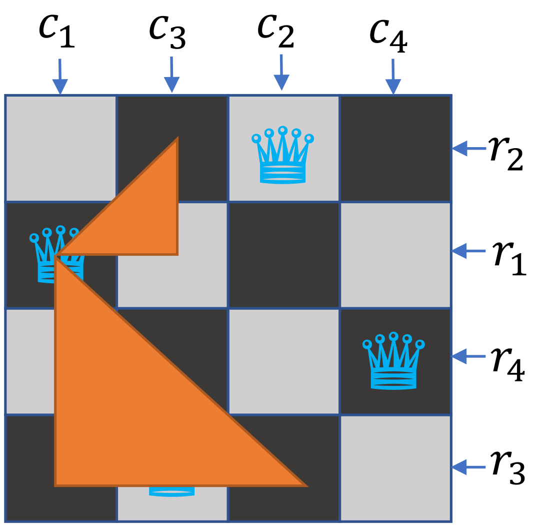
```
for i in range(N):
        for j in range(i+1,N):
            model.Add(row[i]!=row[j])
            model.Add(col[i]!=col[j])
```

Second attempt

- Next, we need to formulate a linear constraint that encodes the fact that no two queens can share a diagonal

- Using symmetry again we can formulate this as

$$\forall i,j\colon i < j \Rightarrow r_i - r_j \neq c_i - c_j$$
$$\forall i,j\colon i < j \Rightarrow r_i - r_j \neq c_j - c_i$$




```
for i in range(N):
        for j in range(i+1,N):
            model.Add(row[i]-row[j] != col[i]-col[j])
            model.Add(row[i]-row[j] != col[j]-col[i])
```

Second attempt

- So let's see how this performs

# of solutions: 1
N = 5
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 80
conflicts: 63
branches: 294
propagations: 1192
integer_propagations: 987
walltime: 0.0041959
usertime: 0.0041959
deterministic_time: 0.00015653

We end up with a large search tree

# of solutions: 1200
N = 5
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 80
conflicts: 15399
branches: 30211
propagations: 381897
integer_propagations: 178122
walltime: 0.72443
usertime: 0.72443
deterministic_time: 0.853899

Also we did not exclude symmetric solutions

Second attempt, refinement

- The solver produces a lot of symmetric solutions, increasing the size of the search tree unnecessarily
- How can we break these symmetries?

- We can impose an ordering on the queens by some criterion, for instance we can order them by row

$$\forall i, j: i < j \Rightarrow r_i < r_j$$

```
for i in range(N):
        for j in range(i+1,N):
            model.Add(row[i]<row[j])
```

Second attempt, refinement

- So let's see how breaking the symmetries performs

# of solutions: 1
N = 10
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 275
conflicts: 41
branches: 642
propagations: 4525
integer_propagations: 1502
walltime: 0.0120257
usertime: 0.0120257
deterministic_time: 0.00047294

The search space for finding a solution is slightly larger than model A

# of solutions: 724
N = 10
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 275
conflicts: 39207
branches: 73308
propagations: 2193502
integer_propagations: 662534
walltime: 1.67868
usertime: 1.67868
deterministic_time: 2.57062

Every solution is unique now

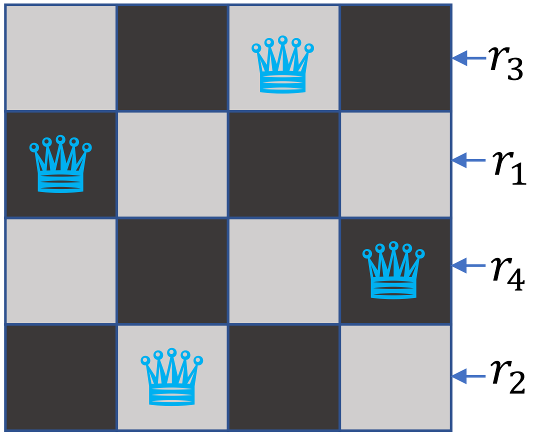
This is about 4.75x faster than model A for finding all solutions

Third attempt

- Can we do even better?
- We can try to decrease the number of parameters further and encode some of the constraints in the parameterisation itself
- Observation: there is always exactly one queen per column, so we could order the queens by column and encode only the row for each

$$X = < r_1, \ldots, r_N >$$

$$D_{r_i} = \{0, \ldots, N-1\}$$




```
row = []
for i in range(N):
    row.append(model.NewIntVar(0,N-1,"row"+str(i)))
```

Third attempt

- Now we only need to make sure that the queens don't share a row, as the columns are implicitly encoded in the parameterisation

$$\forall i, j : i < j \Rightarrow r_i \neq r_j$$

```
for i in range(N):
     for j in range(i+1,N):
          model.Add(row[i]!=row[j])
```

Third attempt

- Also we need to ensure that they do not threaten each other diagonally

- We note that now the index implicitly encodes the column, therefore we have the following constraints

$$\forall i,j: i < j \Rightarrow r_i - r_j \neq i - j$$
$$\forall i,j: i < j \Rightarrow r_i - r_j \neq j - i$$

```
for i in range(N):
    for j in range(i+1,N):
        model.Add(row[i]-row[j] != i-j)
        model.Add(row[i]-row[j] != j-i)
```

Third attempt

- So let's see how this performs

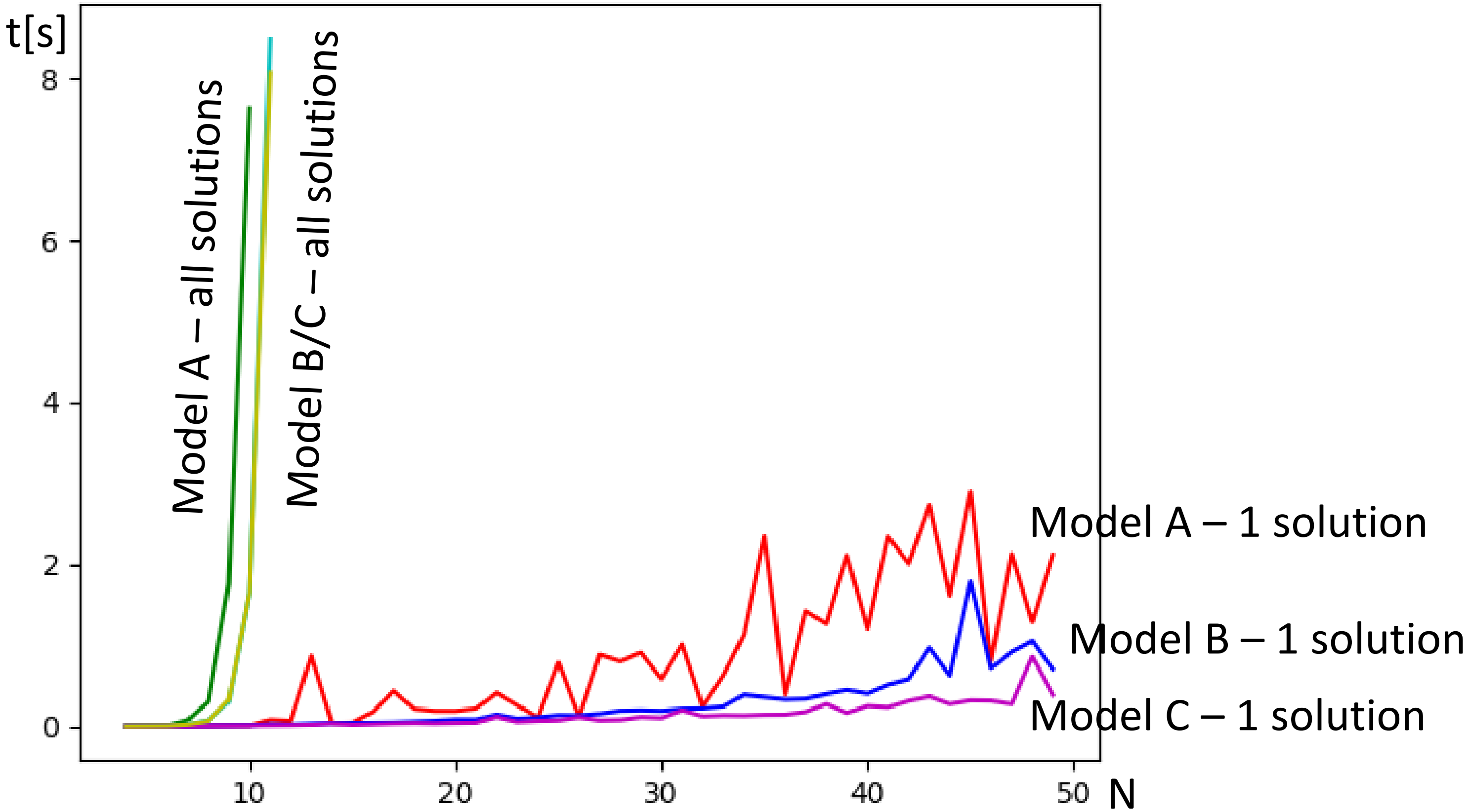
# of solutions: 1
N = 10
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 266
conflicts: 56
branches: 756
propagations: 5783
integer_propagations: 1550
walltime: 0.0097403
usertime: 0.0097403
deterministic_time: 0.00060561

The performance is similar to model B

# of solutions: 724
N = 10
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 266
conflicts: 40196
branches: 74974
propagations: 2207379
integer_propagations: 680732
walltime: 1.67594
usertime: 1.67594
deterministic_time: 2.60838

Comparison of the models

- Model C performs best on finding a solution
- Finding all solutions has exponential runtime, with models B/C performing better than model A



Specialised constraints

- Sometimes there are specialised constraints that solve particular problems very efficiently
- For example, the all-different-constraint enforces all variables to take on different values
- The two code snippets below are imposing the same constraint

```
for i in range(N):
    for j in range(i+1,N):
        model.Add(row[i]!=row[j])
```

```
model.AddAllDifferent(row)
```

Specialised constraints

- Let's compare the two formulations on finding a solution

# of solutions: 2680
N = 35
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 3566
conflicts: 399
branches: 15551
propagations: 188866
integer_propagations: 27701
walltime: 0.164131
usertime: 0.164131
deterministic_time: 0.0188187

The all-different constraint does not perform better

# of solutions: 2680
N = 35
CpSolverResponse:
status: FEASIBLE
objective: NA
best_bound: NA
booleans: 2376
conflicts: 3077
branches: 18177
propagations: 533623
integer_propagations: 152559
walltime: 0.348593
usertime: 0.348593
deterministic_time: 0.0670644

Specialised constraints

- Now let’s compare the two formulations on finding all solutions

# of solutions: 2680
N = 11
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 326
conflicts: 139954
branches: 258355
propagations: 9139830
integer_propagations: 2642628
walltime: 7.9787
usertime: 7.9787
deterministic_time: 15.3084

Now the all-different constraint does perform better

# of solutions: 2680
N = 11
CpSolverResponse:
status: OPTIMAL
objective: 0
best_bound: 0
booleans: 295
conflicts: 42358
branches: 65023
propagations: 3449460
integer_propagations: 1182979
walltime: 2.51386
usertime: 2.51386
deterministic_time: 2.50764

Summary

- Modelling makes all the difference between a feasible and infeasible solution
- Break symmetries where possible
- Avoid redundancies where possible
- Try different reifications, i.e. reformulations of constraints using different equivalent constraints, also try to see if specialised constraints can help
- Always evaluate the performance on a typical dataset for the application

Simple practical example: Staff scheduling

- Let’s assume we have a given number of staff members to fill a number of shifts per day
- Each shift needs a minimal number of staff members present
- Only one shift per day is permissible for every individual staff member
- The workload should be equally balanced between the staff members

- How can we model this scenario using CP-SAT?

Simple practical example: Staff scheduling

- Let's assume we have a given number of staff members to fill a number of shifts per day

- We create Boolean variables per shift per day and per staff member

```
shifts = {}
for staff in range(num_staff):
    for day in range(num_days):
        for shift in range(num_shifts):
            shifts[(staff,day,shift)] = model.NewBoolVar("Shift"+str(staff)+"_"+str(day)+"_"+str(shift))
```

Simple practical example: Staff scheduling

- Each shift needs a minimal number of staff members present
- A linear constraint counting the number of staff members for each day and shift can be added
  (we use equality, as we do not need more staff members present)

```
for day in range(num_days):
        for shift in range(num_shifts):
            staff_present = []
            for staff in range(num_staff):
                staff_present.append(shifts[(staff,day,shift)])
            model.Add(sum(staff_present) == staff_present_per_shift)
```

Simple practical example: Staff scheduling

- Only one shift per day is permissible for every individual staff member

- Again, a linear constraint per day per staff member can be used to limit the number of shifts permissible per day

```
for day in range(num_days):
      for staff in range(num_staff):
          shifts_worked_per_day = []
          for shift in range(num_shifts):
              shifts_worked_per_day.append(shifts[(staff,day,shift)])
          model.Add(sum(shifts_worked_per_day) <= max_shifts_per_day)
```

Simple practical example: Staff scheduling

- The workload should be equally balanced between the staff members

- We introduce two new integer variables for bounding the minimum and maximum workload
- Maximum balance is achieved by minimising the difference (i.e. every staff member works the same number of shifts)

```
max_shifts_total = model.NewIntVar(0,num_days*num_shifts, "max_shifts_total")
min_shifts_total = model.NewIntVar(0,num_days*num_shifts, "min_shifts_total")
for staff in range(num_staff):
    total_shifts_worked = []
    for day in range(num_days):
        for shift in range(num_shifts):
            total_shifts_worked.append(shifts[(staff,day,shift)])
    model.Add(sum(total_shifts_worked) <= max_shifts_total)
    model.Add(sum(total_shifts_worked) >= min_shifts_total)
model.Minimize(max_shifts_total - min_shifts_total)
```

Thank you for your attention!