Constraint Systems

Variable Symmetries

Variable Symmetries

Symmetries may force a solver to visit equivalent solutions

- This is bad for proving optimality/infeasibility
- May led to trashing (time wasted to recover from a mistake)

Symmetries are a big topic in Constraint Programming

- A thorough discussion is too much for this course
- If you are interested: a few papers on the course web-site

Nevertheless, we will try to be more systematic

- Formal definitions (for one special case)
- One frequently employed symmetry breaking technique

Symmetries and Permutations

Symmetries are defined based on the concept of permutation

A permutation π over a discrete set $\mathbf{D}$ is a 1-1 function from $\mathbf{D}$ to $\mathbf{D}$

- Intuitively: just a re-arrangement of the elements. E.g.:

$$i : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

$$\pi(i) : \begin{pmatrix} 3 & 4 & 2 & 1 & 5 \end{pmatrix}$$

Variable Symmetries

With the permutation concept, we can say that:

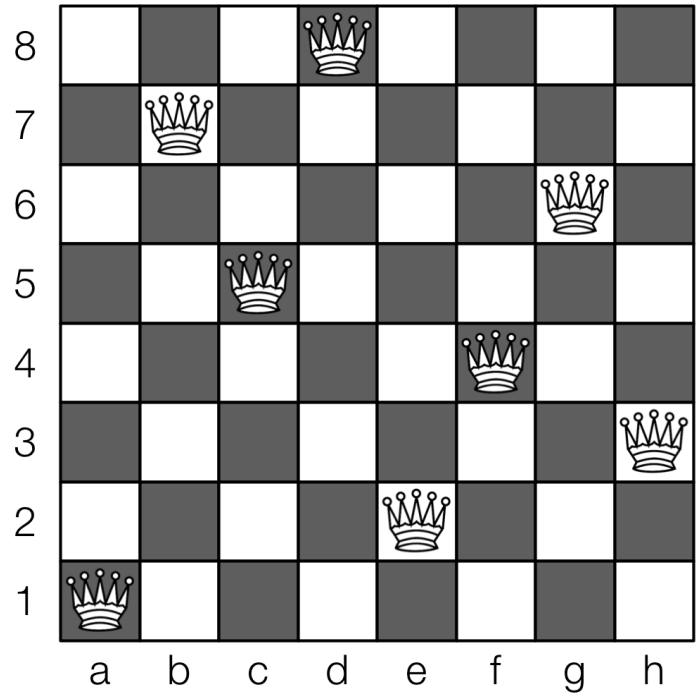
A problem has a variable symmetry iff:

- there exists a permutation π of the variable indices
- s.t. for each feasible solution…
- …we can re-arrange the variables according to π…
- and obtain another feasible solution

- Swapping variables = re-assigning the values
- The permutation π identifies a specific symmetry

Variable Symmetries - Example

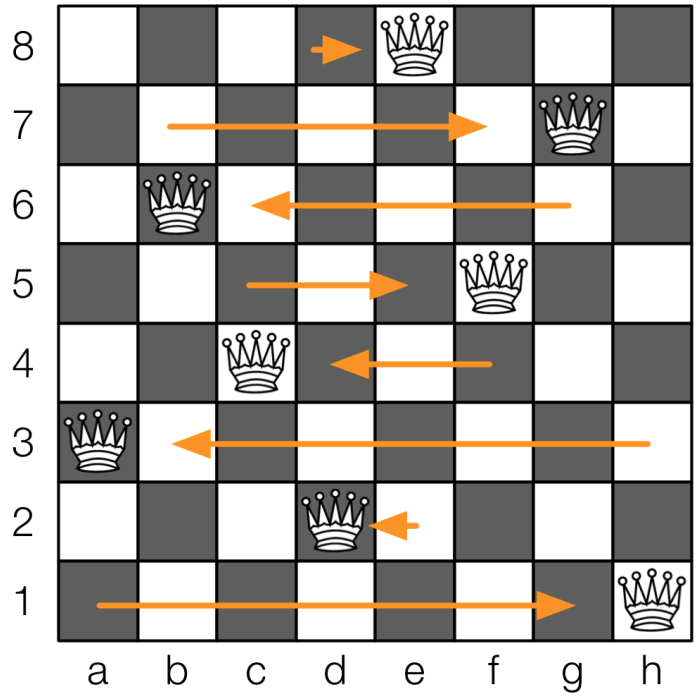
E.g. given a solution for our nqueens model:



- We can swap variable 0 with n-1, 1 with n-2...

Variable Symmetries - Example

E.g. given a solution for our nqueens model:



- We can swap variable 0 with n-1, 1 with n-2...
- ...and obtain a new feasible solution (board flipped on the y-axis)

Symmetry Breaking

There are other forms of symmetries

- We will see an example in the lecture about search...
- ...But we will not define all the possible cases

Instead, we will describe briefly some symmetry breaking techniques

Symmetry breaking = avoid visiting equivalent solutions

The are three main approaches for symmetry breaking in CP:

- Model reformulation
- Static symmetry breaking
- Dynamic symmetry breaking (in the lecture about search)

Model Reformulation for Symmetry Breaking

A first approach to remove symmetries:

- Formulate an alternative model…
- …And make sure that the symmetries are not there

An example: model 2 in Lab4!

A few caveats:

- The alternative model may have drawbacks (e.g. bad propagation)
- Some symmetries may still be there

It's all ok: the goal is finding the most effective trade-off

Static Symmetry Breaking

A second approach to remove symmetries:

- Make some of the equivalent solutions invalid…
- …By adding symmetry breaking constraints

Main idea:

- If a problem contains symmetries $\pi_0, \pi_1, \ldots$
- …Make sure that applying the permutations leads to infeasibility

Can this be done automatically? In some cases. E.g.:

- We know the permutation for each symmetry
- We are dealing with variable symmetries

Lex-Leader Method

Lex-Leader: a generic method for breaking variable symmetries:

- For each symmetry (i.e. permutation) π
- Make sure that only one of the symmetric solutions is valid
- By adding a lexicographic ordering constraint:

$$(x_0, x_1, \ldots) \preceq_{lex} (x_{\pi(0)}, x_{\pi(1)}, \ldots)$$

An example of lexicographic ordering:

$$(x_0 < x_{\pi(0)}) \vee (x_0 = x_{\pi(0)} \wedge x_1 < x_{\pi(1)}) \vee \ldots$$

- Either the fist pairs of variables in each vector are ordered...
- ...Or the following pair must be, and so on

Lex-Leader Method

In lex-leader, we break symmetries one by one:

- PRO: it is very general!
- PRO: breaks all variable symmetries
- CON: complex constraints
- CON: what if we have many symmetries?

Example: model 1 in Lab4

- If we have an order for n product units
- Every permutation corresponds to a symmetry!
- Total: $n!$ symmetries $\Rightarrow$ $n!$ constraints

Lex-Leader: a Special Case

But in model 1 we did not add $n!$ constraints

Model 1 was an instance of a special case:

- If the symmetric variables must be all different…
  - E.g. they are part of an $\textbf{ALLDIFFERENT}$ constraint
- …The lex-leader constraints become much simpler!

(Proof) Consider:

$$(x_0 < x_{\pi(0)}) \vee (x_0 = x_{\pi(0)} \wedge x_1 < x_{\pi(1)}) \vee \ldots$$

- None of the reified constraints $x_i = x_{\pi(i)}$ can be true…
- …Because the variables must be all different…
- …Therefore, the first constraint (i.e. $x_0 < x_{\pi(0)}$) must hold

Lex-Leader: a Special Case

Long story short:

- If the symmetric variables must be all different…
- …Each symmetry is broken by a single $<$ constraint…
- …And at most $n - 1$ constraints are necessary

If all $n!$ permutations are feasible:

- We just need to pick a variable ordering s (e.g. $s = [0, 1, 2, \ldots]$)
- And post the constraints:

$$x_{s(0)} < x_{s(1)} < x_{s(2)} < \ldots$$

This is what we did in model 1 in Lab4

- Actually, we also added $x_{s(0)} < x_{s(2)}, x_{s(0)} < x_{s(3)}, \ldots$
- …They were not actually necessary!

Constraint Systems

SUM Constraint and Incremental Propagation

Global SUM Constraint

Many solvers provide a global SUM constraint, e.g.:

$\text{SUM}(z, X)$, where X is a vector or variables representing the terms of the sum, and z is a variable representing the sum result

The constraint enforces Bound Consistency on the relation:

$$z = \sum_{x_i \in X} x_i$$

- This is also the notation that we will use in the slides

But why would we need it? We already have "+"...

SUM Constraint: an Example

Let's consider a practical example:

$$z = x_0 + x_1 + x_2 + x_3$$
$$x_0, x_1, x_2, x_3 \in \{0, 1\}$$

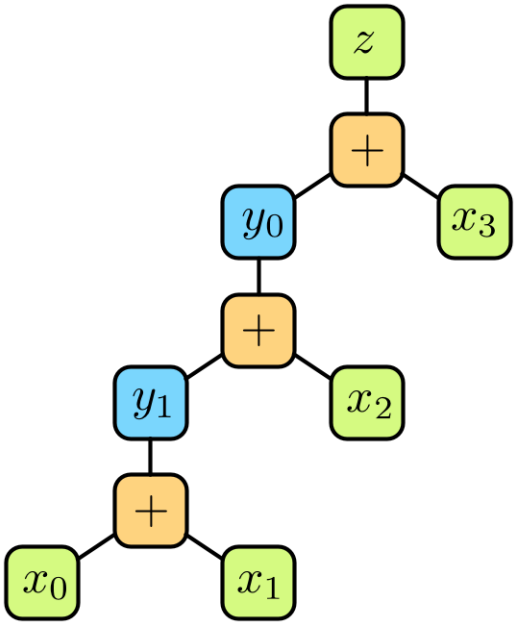
- Let us focus on computing bounds for z

With binary sums, this is equivalent to a set of constraints

$$z = y_0 + x_3$$
$$y_0 = y_1 + x_2$$
$$y_1 = x_0 + x_1$$

SUM Constraint: an Example

The network of constraints can be represented visually:



- Orange = constraints
- Green = original vars
- Blue = introduced vars/expressions

SUM Constraint: an Examples

Let us assume we want to filter the maximum of D(z)

- By reasoning on the individual constraints, we can deduce:

$$\begin{aligned} \overline{y}_1 &= \overline{x}_0 + \overline{x}_1 && \longrightarrow \overline{y}_1 = 1 + 1 = 2 \\ \overline{y}_0 &= \overline{y}_1 + \overline{x}_2 && \longrightarrow \overline{y}_0 = 2 + 1 = 3 \\ \overline{z} &= \overline{y}_0 + \overline{x}_3 && \longrightarrow \overline{z} = 3 + 1 = 4 \end{aligned}$$

- And by reasoning on the sum as a whole?

$$\overline{z} = \overline{x}_0 + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 = 4$$

We deduce exactly the same bound (of course)
Why introducing a global constraint, then?

Why a SUM Constraint?

How many operations with individual sums?

$$\bar{y}_1 = \bar{x}_0 + \bar{x}_1$$
$$\bar{y}_0 = \bar{y}_1 + \bar{x}_2$$
$$\bar{z} = \bar{y}_0 + \bar{x}_3$$

- Read access: 6
- Write access: 3
- Sum: 3

Even more if the constraints are not processed in the right order!

- E.g. if $z = y_0 + x_3$ is propagated before $y_0 = y_1 + x_2$

Why a SUM Constraint?

How many operations with a global constraint?

$$\bar{z} = \bar{x}_0 + \bar{x}_1 + \bar{x}_2 + \bar{x}_3$$

- Read access: 4 (improved)
- Write access: 1 (improved)
- Sum: 3 (same)

And the processing order does no longer matter

Sometimes, we use global constraints to perform the same propagation in a more efficient fashion

Why a SUM Constraint?

For a sum with n terms:

Individual sums

- Read: $2(n-1)$
- Write: $n-1$
- Sum: $n-1$

SUM Constraint

- Read: n
- Write: 1
- Sum: $n-1$

If n is large, the efficiency gap may grow pretty big!

But we can do even more
by exploiting incremental computation

Incremental Computation

A filtering algorithm is usually called multiple times

- In a search tree node (until the fix point it reached)
- On multiple search tree nodes

Hence, incremental computation can improve the efficiency

- The first time the algorithm is called, everything is as usual
- Except that we cache some partial results
- When the algorithm is invoked again we exploit the cached data

This requires access to more details about propagation

- Which variable has been pruned
- Which values have been pruned

Incremental Computation for SUM

In the case of the SUM constraint:

At the first invocation of the filtering algorithm, we compute:

$$ub_z = \sum_{x_i \in X} \overline{x}_i$$

- If $ub_z < \overline{z}$, then we update the maximum of z (as usual)
- But we also cache the value of ub_z
- Let the cached value be $ub_{z\$}$ ("$" stands for "cache" :-))

The algorithm will be activated again when a variable $x_i \in X$ is pruned

Incremental Computation for SUM

For the next activations, we assume that:

- The solver provides the index of the pruned variable
- The solver allows access to the pruned values

In particular, let us assume that:

- x_j is the pruned variable
- $old(\overline{x}_i)$ is the old maximum of any x_i

And we have that:

- $old(\overline{x}_i) > \overline{x}_i$ if the maximum of x_i was pruned
- $old(\overline{x}_i) = \overline{x}_i$ if some other value in $D(x_i)$ has been pruned

Incremental Computation for SUM

- When the domain of x_j is updated, we have that…
- …Our cached $ub_{z\$}$ has been computed using old domains:

$$ub_{z\$} = \sum_{x_i \in X} old(\overline{x}_i)$$

And therefore we can compute a new upper bound on z as:

$$ub_z = ub_{z\$} - old(\overline{x}_j) + \overline{x}_j$$

- Intuively, we replave the old maximum of x_j…
- With the new maximum $\overline{x}_j$

This computation is done in $O(1)$ complexity!

- Because we relied on the cached bound

Incremental Computation for SUM

For a sum with n terms, starting from the second algorithm activation:

Classical SUM

- Read: n
- Write: 1 if $ub_z < \overline{z}$
- Sum: $n - 1$

Incremental propagator

- Read: 3
- Write: 1 ($+1$ if $ub_z < \overline{z}$)
- Sum: 2

Now the efficiency gap is huge

- We have reduced the asymptotic complexity from $O(n)$ to $O(1)$

Filtering the x_i variables in SUM

What about the x_i variables?

A (non-incremental) upper bound on variable x_i is given by:

$$ub_{x_i} = \overline{z} - \sum_{x_h \in X, h \neq i} \underline{x}_h$$

Intuitively:

- We start from the largest possible value for z
- We subtract the lowest possible value of all other x_h variables

It's a generalization of our filtering rule for binary sums

Filtering the x_i variables in SUM

We can use incremental computation to speed up the process

Assuming that x_j has been pruned, our incremental bound is:

$$ub_{x_i} = \overline{z} - (lb_{z\$} - \underline{x}_i - old(\underline{x}_j) + \underline{x}_j)$$

Which is based on the fact that the cached z lower bound is given by:

$$lb_{z\$} = \sum_{x_h \in X} old(\underline{x}_h)$$

- We subtract $\underline{x}_i$ to remove the contribution of the current variable
- We subtract $old(\underline{x}_j)$ and add $\underline{x}_j$ to update the contribution of x_j

In terms of complexity:

- We can prune a single variable with $O(1)$ complexity
- And all variables with complexity $O(n)$

The complexity for a naive approach is $O(n^2)$!

Filtering the x_i variables in SUM

In practice, however:

- There are non-incremental propagators with complexity $O(n)$
- Hence incremental propagation for the x vars is less effective…

But still it helps a lot

- The best solvers implement more efficient incremental propagators
- It is usually much better to use SUM rather than binary sums
- Especially if the sum contains many terms!

Some examples:

- Our hole-counting bound in model 2 from Lab 4
- The dominance rule for model 2 in Lab 4

Constraint Systems

Solver Support
for Incremental Computation

Solver Support for Incremental Computation

From the last batch of slides:

- Incremental computation can be very powerful...
- ...but requires solver support

What do we need, exactly?

- The ability to cache values
- Knowledge of the variable that has been pruned
- The ability to access pruned values

Let's see how we can provide support for these features...

Caching Values

How to cache a value over multiple propagator executions?

A first solution: use a simple variable

Caching via a Simple Variable - Example

Let's see an example:

ops:

$v_\$ \leftarrow 4$

state:

$v_\$ = 4$

- At the root node, we write a value

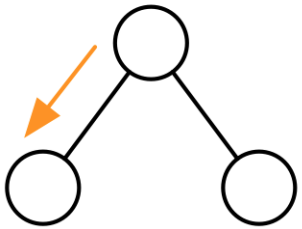
Caching via a Simple Variable - Example

Let's see an example:

ops:

state:

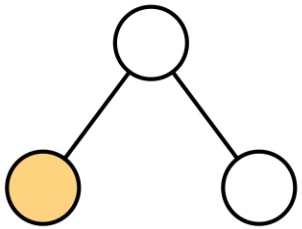
$v_\$ = 4$



- At the root node, we write a value
- We take the left branch

Caching via a Simple Variable - Example

Let's see an example:



ops:

$v_\$ \leftarrow 3$

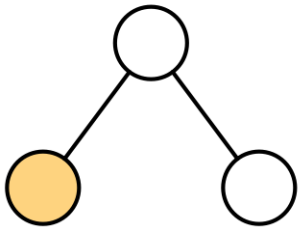
state:

$v_\$ = 3$

- At the root node, we write a value
- We take the left branch
- We update the value

Caching via a Simple Variable - Example

Let's see an example:



ops:

$v_\$ \leftarrow 2$

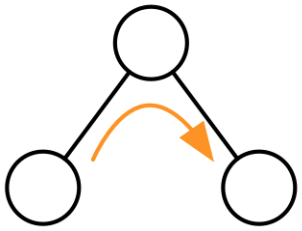
state:

$v_\$ = 2$

- At the root node, we write a value
- We take the left branch
- We update the value
- We update the value again

Caching via a Simple Variable - Example

Let's see an example:



ops: state:

$v_\$ = 2$

It should be $v_\$ = 4$!

- At the root node, we write a value
- We take the left branch
- We update the value
- We update the value again
- And then we backtrack… and we make a mistake

Caching Values

How to cache a value over multiple propagator executions?

A first solution: use a simple variable

- It works when moving from parent to child in the search tree
  - We only need access to the most recent value
- It does not work on backtrack
  - No way to access an old value once it has been overwritten

Caching Values

How to cache a value over multiple propagator executions?

A first solution: use a simple variable

- It works when moving from parent to child in the search tree
  - We only need access to the most recent value
- It does not work on backtrack
  - No way to access an old value once it has been overwritten

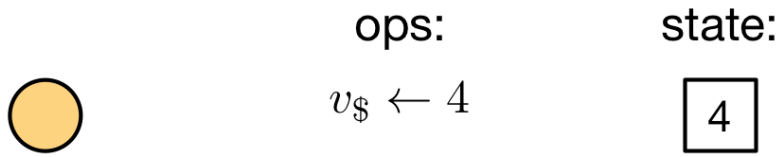
What do we need?

- We need to keep track of past values
- So that they can be restored on backtracking

This can be done by storing values in a stack

Caching via a Stack - Example

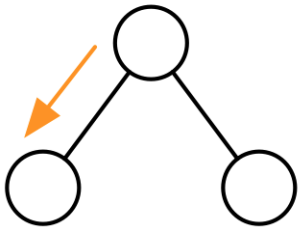
Let's see our example again:



- At the root node, we write a value and push it on the stack

Caching via a Stack - Example

Let's see our example again:



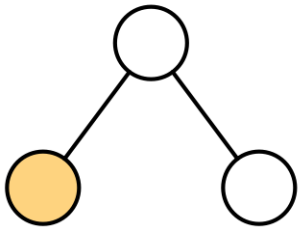
ops:

state:

- At the root node, we write a value and push it on the stack
- We take the left branch

Caching via a Stack - Example

Let's see our example again:



ops:

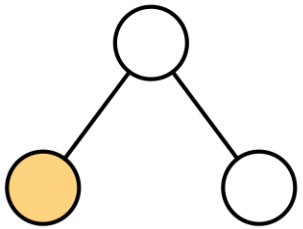
$v_\$ \leftarrow 3$

state:

- At the root node, we write a value and push it on the stack
- We take the left branch
- We update the value and push it on the stack

Caching via a Stack - Example

Let's see our example again:



ops:

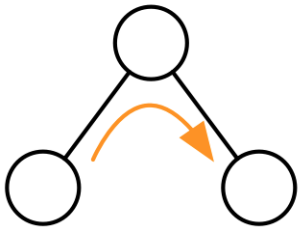
$v_\$ \leftarrow 2$

state:

- At the root node, we write a value and push it on the stack
- We take the left branch
- We update the value and push it on the stack
- We update the value again and push it again

Caching via a Stack - Example

Let's see our example again:



ops:

state:

- At the root node, we write a value and push it on the stack
- We take the left branch
- We update the value and push it on the stack
- We update the value again and push it again
- And then we backtrack... and pop some values

Caching Values

The stack idea works

Two main issues:

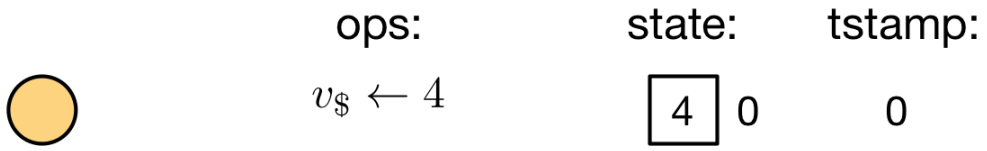
- How to keep track of the number of values to pop?
- Are all the push operations necessary?

We can solve both issues using a timestamp mechanism

- The timestamp is just an integer value
- The timestamp is incremented every time we branch/backtrack
- We push values only if the timestamp has been incremented
- On backtrack, we always pop a single value

Caching via a Stack & Timestamp - Example

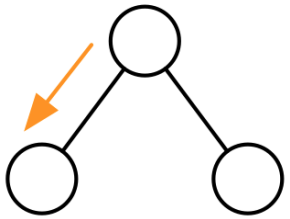
Let's see our example again:



- At the root node, we write a value and push it on the stack

Caching via a Stack & Timestamp - Example

Let's see our example again:

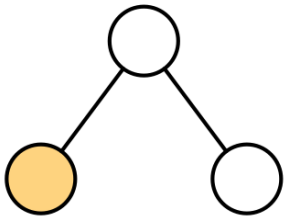


ops:	state:	tstamp:
	4 0	1

- At the root node, we write a value and push it on the stack
- We take the left branch

Caching via a Stack & Timestamp - Example

Let's see our example again:



ops:

$v_{\$} \leftarrow 3$

state:

4	0
3	1

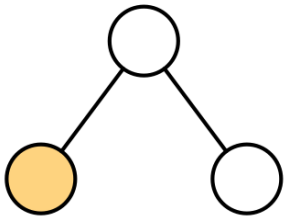
tstamp:

1

- At the root node, we write a value and push it on the stack
- We take the left branch
- We update the value and push it on the stack

Caching via a Stack & Timestamp - Example

Let's see our example again:



ops:

$v_{\$} \leftarrow 2$

state:

4	0
2	1

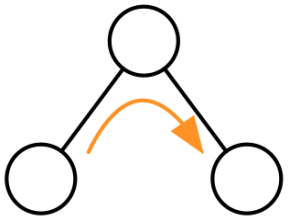
tstamp:

1

- At the root node, we write a value and push it on the stack
- We take the left branch
- We update the value and push it on the stack
- We update the value again and replace the top value

Caching via a Stack & Timestamp - Example

Let's see our example again:



ops: state: tstamp:

4 0

2 1

2

- At the root node, we write a value and push it on the stack
- We take the left branch
- We update the value and push it on the stack
- We update the value again and replace the top value
- And then we backtrack… and pop one value

Caching Values

And we got it (more or less)!

- This technique is usually called trailing
- The stack with timestamps is called trail

Trailing is also used for:

- Keeping track of the domains of the decision variables
- I.e. one of the critical operations performed by a solver

Most of the state-of-the-art CP solvers are trailing based:

- Or-tools, IBM-ILOG CP Optimizer, Choco (default behavior)…
- An important exception: Gecode

Improving our Propagation Algorithm

Second requirement: know which variable has been pruned

For this, we need to modify our basic propagation algorithm:

```
dirty = true
while dirty:
    dirty = false
    for c_j ∈ C:
        D' = filter_{c_j}(D)
        if D' ≠ D: dirty = true
```

- This was called AC1
- Is is veeery basic
- And it is definitely time to improve it

Which modifications do we need? Let's start from two of them

Improving our Propagation Algorithm

1) The constraints should know about the pruned variable

This can be done by adding a new filtering method

$$filter_{c_j}(i, D)$$

- The index of the pruned variable x_i is passed as an argument
- Such information can be employed for incremental computation

Improving our Propagation Algorithm

1) The constraints should know about the pruned variable

This can be done by adding a new filtering method

$$filter_{c_j}(i, D)$$

- The index of the pruned variable x_i is passed as an argument
- Such information can be employed for incremental computation

2) The constraints should communicate the pruned variable

We can introduce a function to prune values

$$prune(i, v)$$

- Where i is the index of the variable and v is the value to be pruned
- The method must notify the other constraints about pruning events

Pruning and Notification - Naive Approach

How should the `prune` method work?

A possible implementation:

function $prune(i, v)$
    $D(x_i) = D(x_i) \setminus \{v\}$
    for $c_j \in C$:
        $filter_{c_j}(x_i, D)$

- Calling *filter* may lead to new pruned values…
- …and therefore to recursive calls of *prune*

This is messy and inefficient: can we avoid recursive calls?

- Instead of calling *filter*…
- …we store the "pruning event" into an external data structure

Propagation Queue

For example, we can use a FIFO queue Q:

- Q keeps track of the $filter$ calls that we still need to make
- Each element in Q is a pair (c_j, x_i)

Q is often called propagation queue (or propagator queue)

We can now redefine our $\mathbf{prune}$ function:

```
function prune(i, v)
    D(x_i) = D(x_i) \ {v}
    for c_j ∈ C:
        Q.push((c_j, x_i))
```

- Instead of calling $filter_{c_j}(x_i, D)$ we add an item to Q

Propagation Queue

For example, we can use a FIFO queue $\mathbf{Q}$:

- Q keeps track of the *filter* calls that we still need to make
- Each element in Q is a pair (c_j, x_i)

Q is often called propagation queue (or propagator queue)

We can now redefine our $\mathbf{prune}$ function:

```
function prune(i, v)
    D(x_i) = D(x_i) \ {v}
    for c_j ∈ C:
        if x_i ∈ X(c_j): Q.push((c_j, x_i))
```

- Since we know that variable x_i is being pruned…
- …we can notify only the constraints that have x_i in their scope

Propagation Queue

For example, we can use a FIFO queue Q:

- Q keeps track of the *filter* calls that we still need to make
- Each element in Q is a pair (c_j, x_i)

Q is often called propagation queue (or propagator queue)

We can now redefine our `prune` function:

```
function prune(i, v)
    D(x_i) = D(x_i) \ {v}
    for c_j ∈ C:
        if x_i ∈ X(c_j) ∧ (c_j, x_i) ∉ Q: Q.push((c_j, x_i))
```

- Since we are not calling *filter* immediately…
- …we can also avoid some redundant calls (and stack overflows)

Propagation Queue

For example, we can use a FIFO queue Q:

- Q keeps track of the *filter* calls that we still need to make
- Each element in Q is a pair (c_j, x_i)

Q is often called propagation queue (or propagator queue)

We can now redefine our `prune` function:

```
function prune(i, v)
    D(x_i) = D(x_i) \ {v}
    for c_j ∈ C:
        if x_i ∈ X(c_j) ∧ (c_j, x_i) ∉ Q: Q.push((c_j, x_i))
```

- Dedicated methods are often available for pruning min/max vals
- The main idea however stays the same

A Modified Propagation Algorithm

Our goal now is just processing the queue:

$$\textbf{while } len(Q) > 0$$
$$(c_j, x_i) = Q.pop()$$
$$filter_{c_j}(i, D)$$

- New items are inserted in Q iff $filter$ prunes something
- Hence, the fix-point is reached iff Q becomes empty

At this point, only a few pieces are missing...

- What is the initial content of queue?
- What about non-incremental propagators?
- When do we perform the initial setup of incremental propagators?

A Modified Propagation Algorithm

Let's add the possibility to have $(c_j, *)$ items in Q

The wild-card $*$ means that the pruned variable is unspecified

- An item $(c_j, *)$ is processed by calling $filter_{c_j}(D)$…
- …i.e. our old filtering function…
- …which is available also for non-incremental propagators

At the root of the search tree, no variable has been pruned:

- Hence, Q should contain $(c_j, *)$ items for all constraints
- Incremental constraints may perform their setup in $filter_{c_j}(D)$

AC3 Propagation Algorithm

Overall, we get:

$Q = [(c_j, *) \text{ for } c_j \in C]$
while $len(Q) > 0$:
    $(c_j, x_i) = Q.pop()$
    if $x_i = *$:
        $filter_{c_j}(D)$
    else:
        $filter_{c_j}(i, D)$

This algorithm is often called AC3

- (Even if the original AC3 was a bit different)
- It's probably the most widespread propagation algorithm

AC3 Propagation Algorithm

Overall, we get:

$Q = [(c_j, *) \text{ for } c_j \in C]$
while $len(Q) > 0$:
    $(c_j, x_i) = Q.pop()$
    if $x_i = *$:
        $filter_{c_j}(D)$
    else:
        $filter_{c_j}(i, D)$

- There are many other propagation algorithms (AC5, AC7, AC2000...)
- We will not see them (most work for extensional constraints)
- If you are interested, there is a paper on the course web site

Instead, let's seen an example of how AC3 works...

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0 [1, 2, 3]

x_1 [1, 2, 3]

x_2 [1, 2, 3]

Q: | $c_0, *$ | $c_1, *$ | $c_2, *$ |

Let's consider this simple CSP

- For the sums, we use the global SUM constraint
- Initially, Q contains a $(c_j, *)$ item for each constraint

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, 3]
x_1	[1, 2, 3]
x_2	[1, 2, 3]

Q: $c_0, *$ | $c_1, *$ | $c_2, *$

- We pop and process the first item in Q
- No variable can be pruned

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[1, 2, 3]

Q: $c_1, *$ | $c_2, *$

- We pop and process the first item in Q
- We use the non-incremental filtering algorithm (due to the $*$)
- We initialize the values of $ub_{z\$}$ and $lb_{z\$}$
- We manage to prune x_0 and x_1

AC3 - An Example

c_0 : ALDIFFERENT($[x_0, x_1, x_2]$)

c_1 : $x_0 + x_1 = 3$

c_2 : $x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[1, 2, 3]

Q:

$c_2, *$	$c_0, *$	c_1, x_0	c_1, x_1	c_2, x_1

- We add a (c_j, x_0) item for each c_j having x_0 in the scope
- We add a (c_j, x_1) item for each c_j having x_1 in the scope

AC3 - An Example

c_0 : ALDIFFERENT($[x_0, x_1, x_2]$)

c_1 : $x_0 + x_1 = 3$

c_2 : $x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[1, 2, 3]

Q: | $c_2, *$ | $c_0, *$ | c_1, x_0 | c_1, x_1 | c_2, x_1 |

We have not seen an incremental propagator for ALLDIFFERENT

- But an incremental propagator does exist
- And it is very important for the ALLDIFFERENT performance
- However, we will not see it in this course

AC3 - An Example

c_0 : ALDIFFERENT($[x_0, x_1, x_2]$)

c_1 : $x_0 + x_1 = 3$

c_2 : $x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[1, 2, 3]

Q:

$c_2, *$	$c_0, *$	c_1, x_0	c_1, x_1	c_2, x_1

- x_0, x_1 have been pruned by c_1
- And the (c_1, x_0) and (c_1, x_1) items are inserted in Q

This may look redundant, but it is not:

- Some propagators need multiple passes to reach convergence

AC3 - An Example

$c_0 : \text{ALLDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[1, 2, 3]

Q:

$c_2, *$	$c_0, *$	c_1, x_0	c_1, x_1	c_2, x_1

- We pop and process the first queue item
- No variable can be pruned

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q:

$c_0, *$	c_1, x_0	c_1, x_1	c_2, x_1

- We pop and process the first queue item
- We can prune x_2

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q:

c_1, x_0	c_1, x_1	c_2, x_1	$c_0, *$	c_2, x_2

- We add a (c_j, x_2) event for each c_j having x_2 in the scope

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q:

c_1, x_0	c_1, x_1	c_2, x_1	$c_0, *$	c_2, x_2

- We pop and process the first queue item
- No variable can be pruned

AC3 - An Example

$c_0 : \text{ALLDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q:

c_1, x_1	c_2, x_1	$c_0, *$	c_2, x_2

- We pop and process the first queue item
- No variable can be pruned

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q: | c_2, x_1 | $c_0, *$ | c_2, x_2 |

- We pop and process the first queue item
- No variable can be pruned, even though $x_1 = 2$ is infeasible
- Because we are using the incremental propagator for SUM
- ...And we are using information about x_1 to filter x_2...
- ...And not vice-versa

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q: $c_0, *$ | c_2, x_2

- We pop and process the first queue item
- No variable can be pruned

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, ~~2~~, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q: c_2, x_2

- We pop and process the first queue item
- And we can prune x_1…
- …Because now we are using information about x_2 to filter x_1

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0\ x_1\ x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, ~~2~~, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q: $c_0, *$ | c_1, x_1 | c_2, x_1

- We insert new items in the queue, as usual

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	~~[1~~, 2, ~~3]~~
x_1	[1, ~~2~~, ~~3]~~
x_2	~~[1~~, ~~2~~, 3]

Q: | $c_0, *$ | c_1, x_1 | c_2, x_1 |

- We pop and process the first queue item...
- ...And we prune x_0

AC3 - An Example

c_0 : ALDIFFERENT($[x_0, x_1, x_2]$)

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	~~[1, 2, 3]~~
x_1	[1, ~~2~~, ~~3~~]
x_2	~~[1, 2~~, 3]

Q: | c_1, x_1 | c_2, x_1 | $c_0, *$ | c_1, x_0 |

- We insert new items as usual

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	~~[1, 2, 3]~~
x_1	[1, ~~2~~, ~~3~~]
x_2	~~[1, 2~~, 3]

Q: | c_1, x_1 | c_2, x_1 | $c_0, *$ | c_1, x_0 |

- The solution is now feasible, but the solver does not know it
- Even when all variables are bound there may be an inconsistency
- Hence, we must process all the items in Q!

AC3 - An Example

$c_0 : \text{ALLDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[~~1~~, 2, ~~3~~]
x_1	[1, ~~2~~, ~~3~~]
x_2	[~~1~~, ~~2~~, 3]

Q:

c_2, x_1	$c_0, *$	c_1, x_0

AC3 - An Example

$c_0 : \text{ALLDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	~~[1~~, 2, ~~3]~~
x_1	[1, ~~2~~, ~~3]~~
x_2	~~[1~~, ~~2~~, 3]

Q:

$c_0, *$	c_1, x_0

AC3 - An Example

$c_0 : \text{ALLDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	~~[1~~, 2, ~~3]~~
x_1	[1, ~~2~~, ~~3]~~
x_2	~~[1~~, ~~2~~, 3]

Q: c_1, x_0

AC3 - An Example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	~~[1, 2, 3]~~
x_1	[1, ~~2, 3~~]
x_2	~~[1, 2~~, 3]

Q:

- Now the queue is empty!
- Hence, we have reached the fix-point
- Incidentally, we also have a solution

Accessing Pruned Values

We are missing only one ingredient: accessing pruned values

If we want to enable this, we need to:

- Design an API for accessing the values
- Store the pruned values
- Forget the pruned values once they have been processed

The first two problems are simple:

- The API is (mostly) a matter of design choices
- Storing the values can be done in a number ways

The tricky part is choosing when to forget the stored values

Accessing Pruned Values

Consider this step of our AC3 example:

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[1, 2, 3]

Q:

$c_2, *$	$c_0, *$	c_1, x_0	c_1, x_1	c_2, x_1

- Value 3 has been pruned from x_0 (and x_1)
- And we can store it in some way

Accessing Pruned Values

Consider this step of our AC3 example

c_0 : ALDIFFERENT($[x_0, x_1, x_2]$)

c_1 : $x_0 + x_1 = 3$

c_2 : $x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[1, 2, 3]

Q:

$c_2, *$	$c_0, *$	c_1, x_0	c_1, x_1	c_2, x_1

The removal triggered the insertion of items in Q:

- Once items $(c_0, *)$ and (c_1, x_0) have been processed
- The removal of 3 from x_0 has been acknowledged

Accessing Pruned Values

Consider this step of our AC3 example

$c_0 : \text{ALDIFFERENT}([x_0, x_1, x_2])$

$c_1 : x_0 + x_1 = 3$

$c_2 : x_1 + x_2 \leq 4$

x_0	[1, 2, ~~3~~]
x_1	[1, 2, ~~3~~]
x_2	[1, 2, 3]

Q: | $c_2, *$ | $c_0, *$ | c_1, x_0 | c_1, x_1 | c_2, x_1 |

Hence, we just need to:

- Mark the last generated item for each pruned variable, i.e. (c_1, x_0)
- Once it has been processed, we can forget the stored value

Constraint Systems

ELEMENT Constraint and Support Caching

Modeling Assignment Costs

A simple problem: we need to assign a certain object

- $x \in \{0, 1, 2\}$ represent the possible assignment choices
- Assigning the object leads to a cost z
- The cost depends on the value of x:
  - If $x = 0$, the cost is 1
  - If $x = 1$, the cost is 3
  - If $x = 2$, the cost is 4

This is simple, but important:

- This kind of relation appears in many complex problems

How can we model this situation?

Sum Based Model

A first possible model:

$$z = 1\,(x = 0) + 3\,(x = 1) + 4\,(x = 2)$$

This is how we have handled costs (and capacities) so far:

- It works
- Filtering can be quite efficient thanks to SUM

However, the model has poor propagation:

- The z bounds computed by SUM are 0 and 8
- But the true bounds for z are 1 and 4!

The problem, as usual, are the reified constraints...

Sum Based Model #2

A better model:

$$z = 1 + 2\,(x = 1) + 3\,(x = 2)$$

Here's the main underlying idea:

- x must take a value
- Hence, in the best case the cost will be 1 (for value 0)
- Other values lead to higher costs

The propagation is a bit better:

- The z bounds are 1 and 6

But we are still far from the true bounds!

A Change of Perspective

Let's look at the problem from another perspective:

- We are trying to access a vector of costs (i.e. [1, 3, 4])
- And the x variable acts as an index

Formally, this can be written as:

$$z = v_x$$

where $V = [v_0, v_1, v_2, \ldots]$ is the vector of costs

- In this format, it is explicit that z must take a value from V
- But we need a custom propagator!

ELEMENT Constraint

The relation $z = v_x$ is captured by the ELEMENT constraint

ELEMENT(z, V, x), where:

- z is an "output" variable
- V is a vector of values (or variables)
- x is an "index" variable

- In most solvers the constraint is represented as an expression...
- ... and using a special syntax

We will use the syntax $z = v_x$

ELEMENT - Usage Examples

ELEMENT is a very important constraint

Some usage examples:

"*The global cost is the sum of the assignment cost of variables in* X"

$$z = \sum_{x_i \in X} c_{x_i}$$

- C is the vector of assignment costs
- The summation can be modeled (as usual) with SUM

Assignment costs are present in many practical problems

ELEMENT - Usage Examples

ELEMENT is a very important constraint

Some usage examples:

"The product in position 0 must have lower weight than position 1"

$$w_{x_0} < w_{x_1}$$

- x_i represent the product at position i
- W is a vector containing the weight of all products

ELEMENT can be used to map items to their properties

- This greatly simplifies the definition of complex constraints
- Especially with some modeling styles

A Propagator for the ELEMENT Constraint

How do we write a propagator for $z = v_x$?

A simplifying assumption: V is a vector of values

- Bound consistency for the z variable:

$$ub_z = \max_{u \in D(x)} v_u$$

$$lb_z = \min_{u \in D(x)} v_u$$

- GAC for the z variable:

$$w \in D(z) \text{ is not pruned iff } \exists u \in D(x) \ : \ v_u = w$$

BC can be enforce in $O(|D(x)|)$, GAC in $O(|D(z)|\,|D(x)|)$

A Propagator for the ELEMENT Constraint

How do we write a propagator for $z = v_x$?

A simplifying assumption: V is a vector of values

- Bound consistency for the x variable:

$$ub_x = \max\{u \in D(x) : \underline{z} \leq v_u \leq \overline{z}\}$$
$$lb_x = \min\{u \in D(x) : \underline{z} \leq v_u \leq \overline{z}\}$$

- GAC for the x variable:

$$u \in D(x) \text{ is not pruned iff } \exists w \in D(z) \ : \ v_u = w$$

- If V contains variables the propagator is a bit more complex...
- ...But the basic reasoning is the same

Incremental Propagator for ELEMENT

Can we use incremental computation for ELEMENT?

- Yes, we can, via a simple and very generalizable approach
- Main idea: store the supports

Let's see how to do it when enforcing GAC on z:

- At the first propagator execution, we do the usual stuff:

$$w \in D(z) \text{ is not pruned iff } \exists u \in D(x) \ : \ v_u = w$$

- But for each $w \in D(z)$, we store the corresponding support u
- Let us refer to it as $u(w)$

Incremental Propagator for ELEMENT

Can we use incremental computation for element?

- Yes, we can, via a simple and very generalizable approach
- Main idea: store the supports

Let's see how to do it when enforcing GAC on Z:

- Whenever x gets pruned, we check if $u(w)$ is still in the domain
- If it is, then w has still a support
- Otherwise, we search for another support
- And of course the new value becomes $u(w)$

Incremental Propagator for ELEMENT

This idea is at the basis of an algorithm called GAC-Schema

- It is often sub-optimal
- But it is also simple and very general

For those interested in the actual GAC-Schema algorithm:

- There is a paper on the course web site
- It's the survey about global constraints

Incremental Propagator for ELEMENT

In many cases (including ours), this approach has a nice property

There is no need to restore the u(w) values on backtrack

When moving to child to parent node, the domains can only grow larger

- Hence, if $u(w)$ is a support for w in a child node...
- ...it will still be a support in the parent node

We can use normal variables to store the u(w) values!

- There is not need of trailing for $u(w)$
- Hence, no stack to store and to manage the variable
- This is much more efficient

Constraint Systems

A Few More Global Constraints

MIN/MAX Constraint

- We have introduced a global constraint for sums with many terms
- We can to the same for minimum/maximum operations

> $\textsc{min}(z, X)$, where z is an output variable and X is a vector of input variables

The constraint enforces BC on the relation:

$$z = \min(X)$$

- which is often used as a special syntax for its definition

The constraint has an efficient propagator

- Incremental propagation is performed via support cachink

TABLE Constraint

- Remember our "theoretical" definition of constraint?
- It turns out it is available as a global constraint, too!

$\textsc{table}(X, T)$, where:

- X is a vector of variables
- T is a vector of tuples, corresponding to the valid assignments

- The variables in X can take a certain combination of values…
- …iff such combination appears as a tuple in T

The constraint can also be specified using invalid assignments

TABLE Constraint

TABLE allows to model arbitrary constraints, including:

- Vector-to-scalar functions:

x_0	x_1	x_2	y
1	2	0	2
...	...	...	...

- Scalar-to-vector functions:

x	y_0	y_1	y_2
0	2	0	1
...	...	...	...

TABLE Constraint

TABLE allows to model arbitrary constraints, including:

- Vector-to-vector functions:

x_0	x_1	y_1	y_2
0	1	1	0
...	...	...	...

Every constraint over discrete variables can be modeled with TABLE

- Of course, the table size may grow very large
- But TABLE often very well-optimized
- In or-tools: tables with 1M rows can be handled very efficiently