Outline

- The consistency of a constraint (tradeoff filtering vs time)
- The sum constraint
  - NP-hardness
  - Bound consistency
  - Idempotence
  - Variable views
- The element constraint
  - 1D and 2D, on array of constants and array of variables
  - Hybrid consistency and domain consistency

Constraints

Consistency notions

Remember, a constraint has two responsibilities

- Check if it is feasible: yes / no / possibly
- Remove infeasible values by an algorithm,
  known as a *propagator*, *filtering algorithm* (or *constraint*)

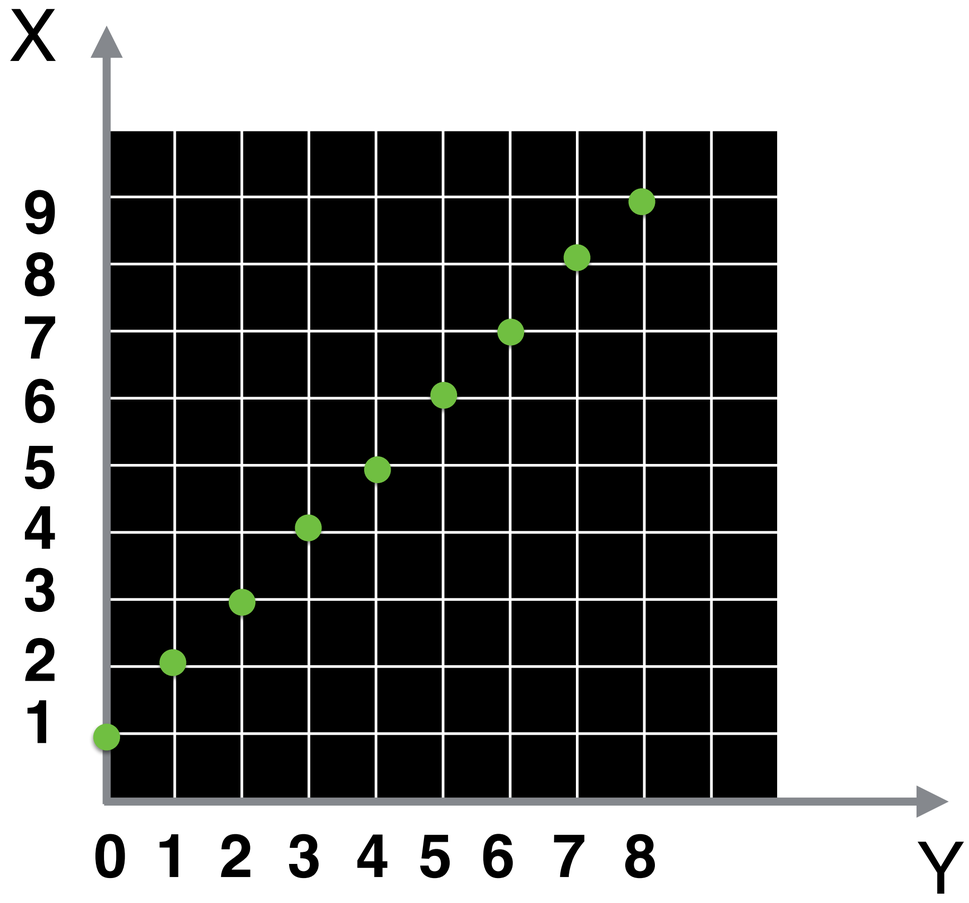
- But how much can a filtering algorithm remove 🤔?

Constraints

Domain Consistency

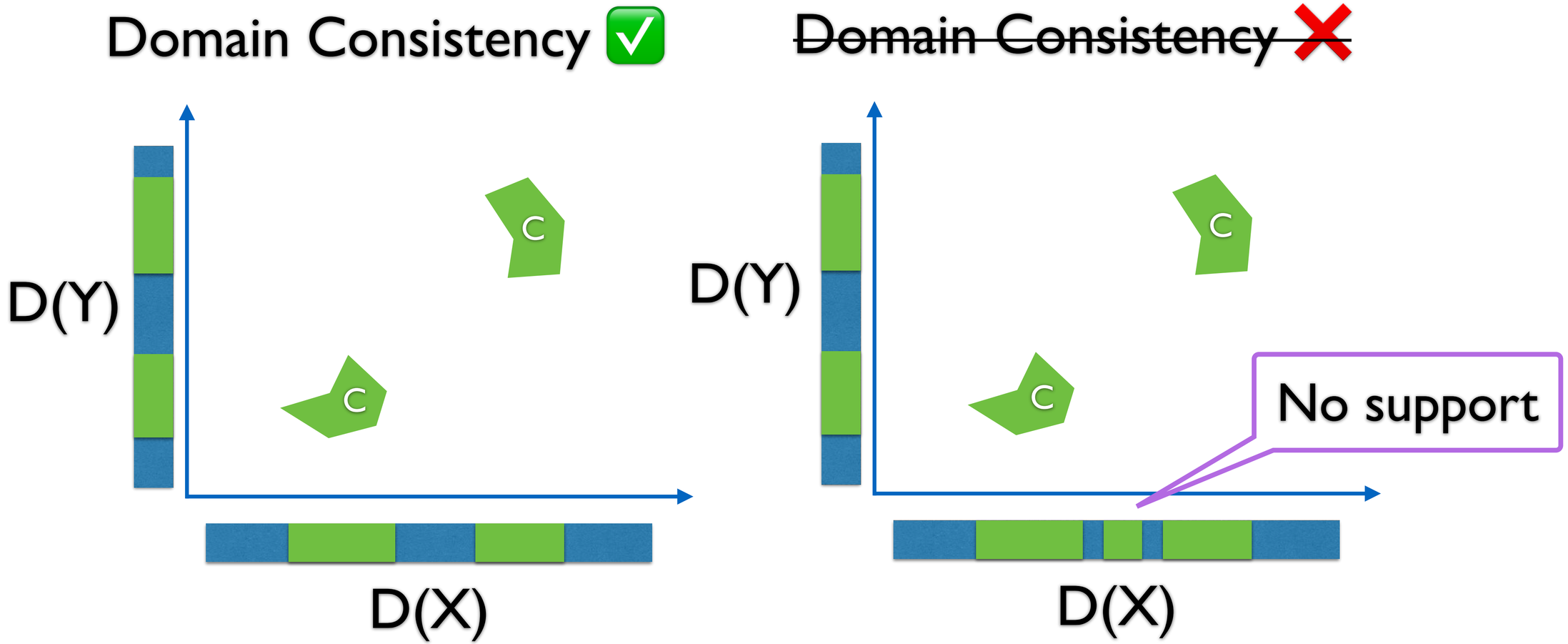
Constraint X = Y+1

Set of solutions for D(X) = {1..9} and D(Y) = {0..8}:



Domain Consistency (DC): Definition

A constraint *C* is *domain consistent* iff each value of the domain of each of its variables participates in at least one solution to *C*.

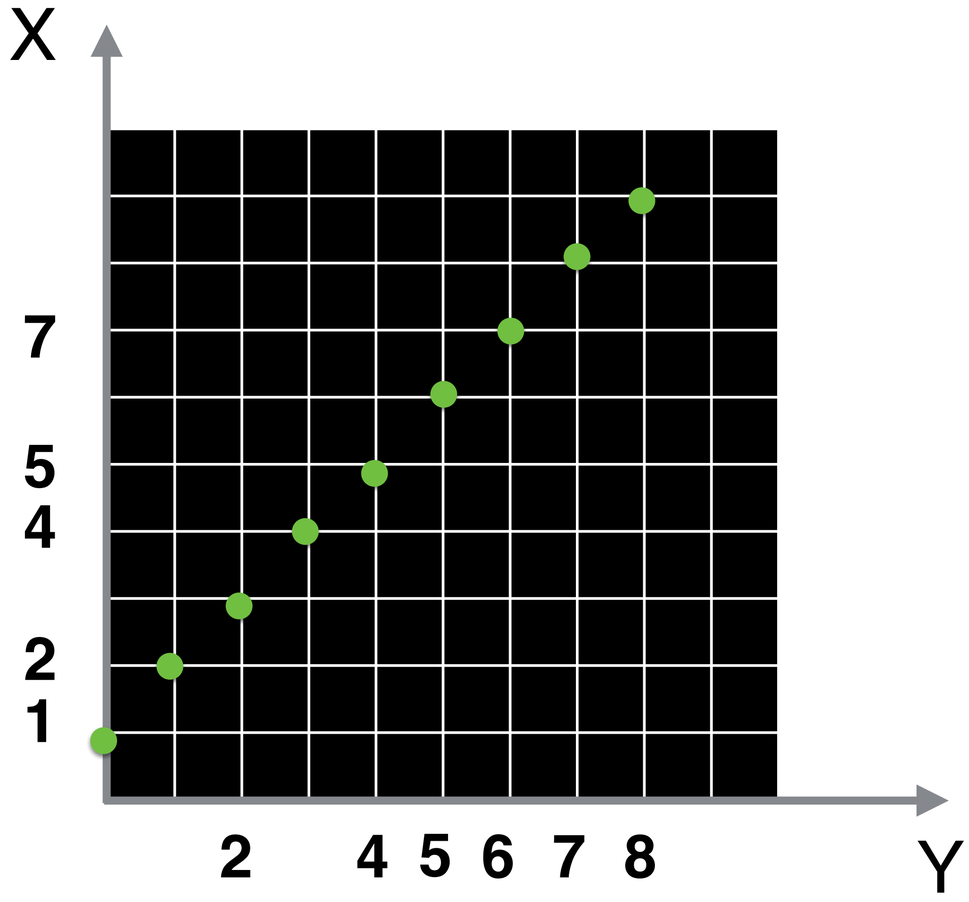


DC Filtering for X = Y+1

$$\mathcal{F}_c(D)(x) = \{v \in D(x) \mid v - 1 \in D(y)\}$$

$$\mathcal{F}_c(D)(y) = \{v \in D(y) \mid v + 1 \in D(x)\}$$

Remove all the points that do not participate in a solution to the constraint, given the domains.

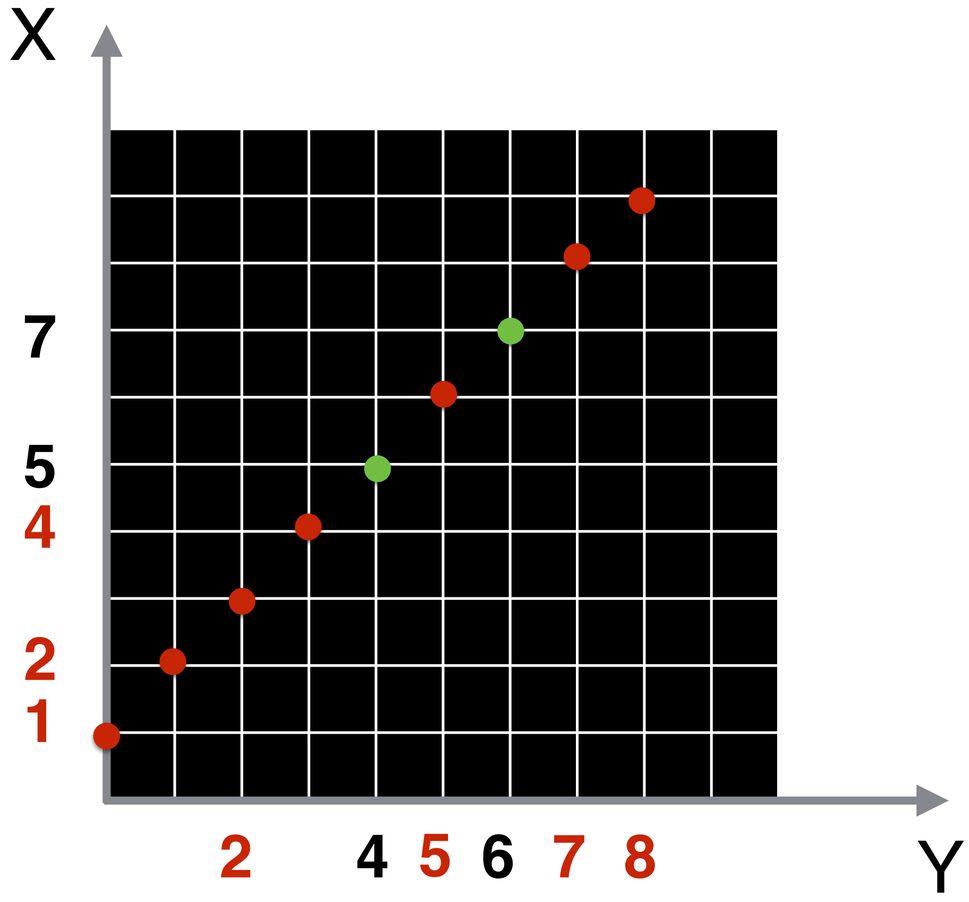


DC Filtering for X = Y+1

$$\mathcal{F}_c(D)(x) = \{v \in D(x) \mid v - 1 \in D(y)\}$$

$$\mathcal{F}_c(D)(y) = \{v \in D(y) \mid v + 1 \in D(x)\}$$

Remove all the points that do not participate in a solution to the constraint, given the domains.



DC Filtering for X = Y+1 (Domain Consistent)

```
public class EqualPlusOneDC extends AbstractConstraint {

    IntVar x, y;

    @Override
    public void post() {
        x.propagateOnDomainChange(this);
        y.propagateOnDomainChange(this);
        propagate();
    }

    @Override
    public void propagate() {
        // propagate from x to y
        for (int v = x.min(); v <= x.max(); v++) {
            if (x.contains(v) && !y.contains(v-1)) {
                x.remove(v);
            }
        }
        // propagate from y to x
        for (int v = y.min(); v <= y.max(); v++) {
            if (y.contains(v) && !x.contains(v+1)) {
                y.remove(v);
            }
        }
    }
}
```

Time complexity: $O(|D(x)| + |D(y)|)$

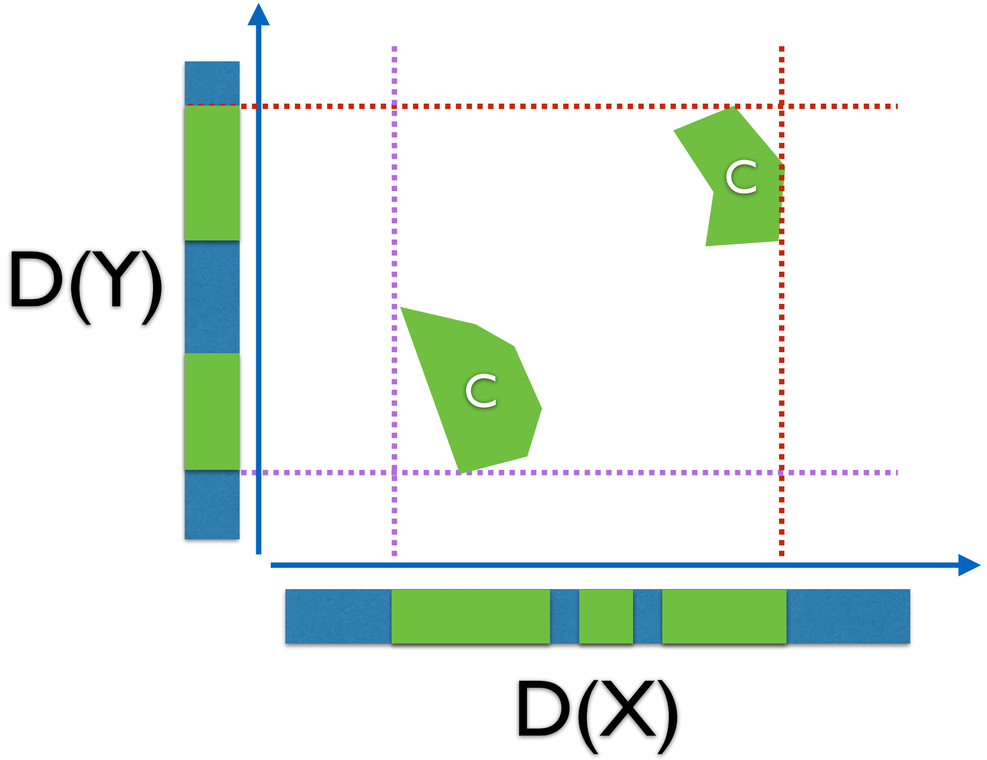
Constraints

Bound Consistency

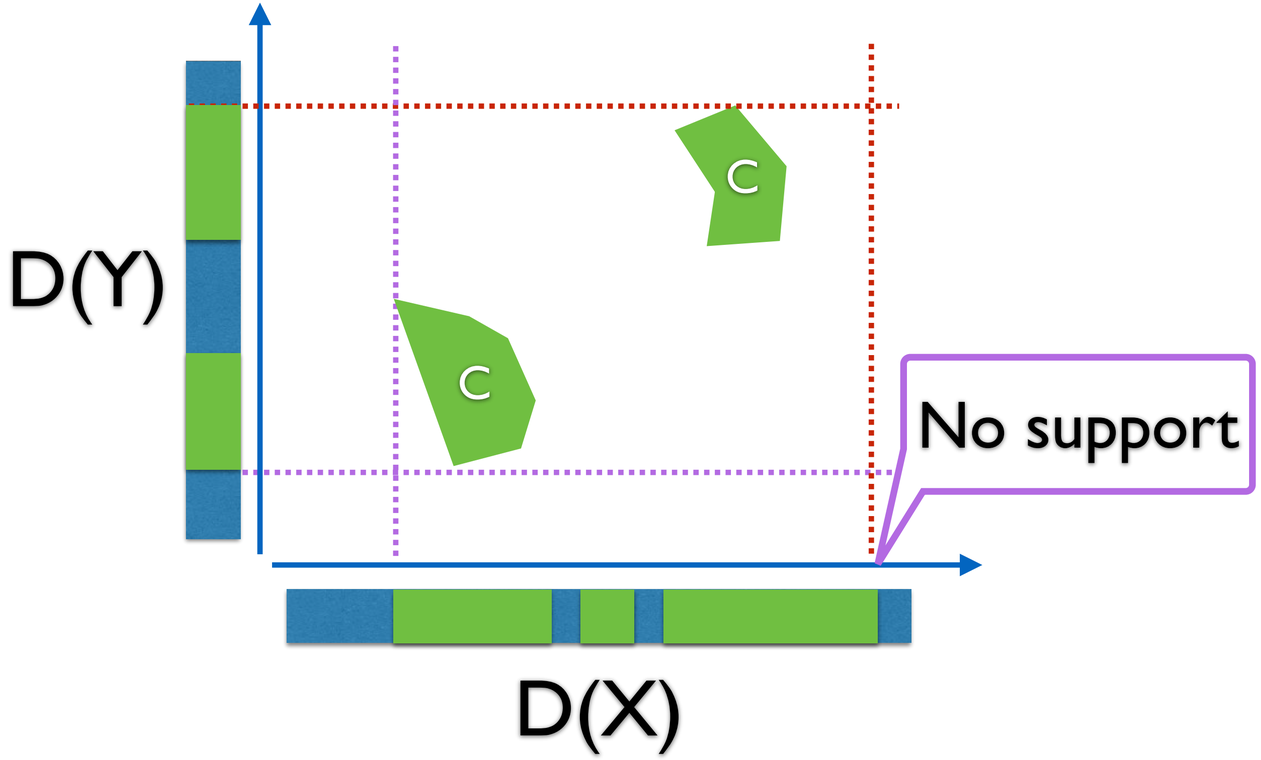
Bound Consistency (BC): Definition

A constraint *C* is *bound consistent* iff the minimum and maximum of the domain of each of its variables participate in at least one solution to *C*, assuming interval domains.

Bound Consistency ✅



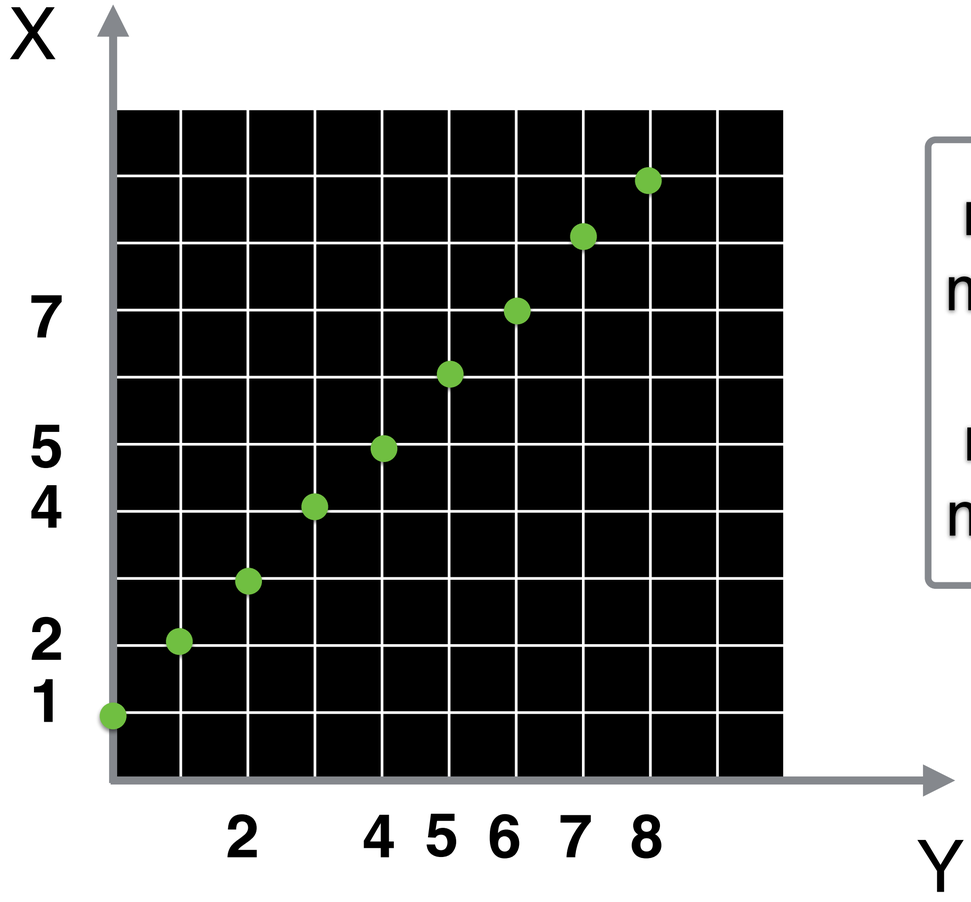
~~Bound Consistency~~ ❌



BC Filtering for X = Y+1: Reasoning on the Bounds

$$\mathcal{F}_c(D)(x) = \{v \in D(x) \mid \min(D(y)) + 1 \leq v \leq \max(D(y)) + 1\}$$

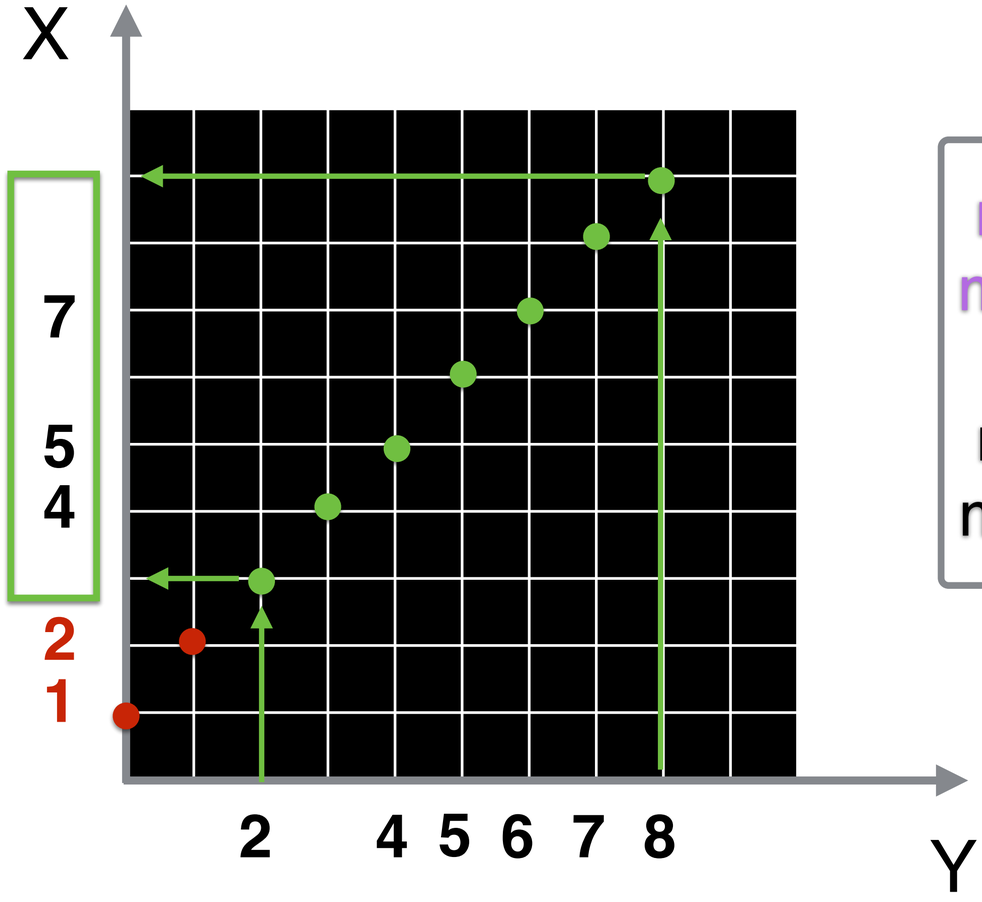
$$\mathcal{F}_c(D)(y) = \{v \in D(y) \mid \min(D(x)) - 1 \leq v \leq \max(D(x)) - 1\}$$



min(X) >= max(min(Y)+1,min(X))
max(X) <= min(max(Y)+1,max(X))

min(Y) >= max(min(X)–1,min(Y))
max(Y) <= min(max(X)–1,max(Y))

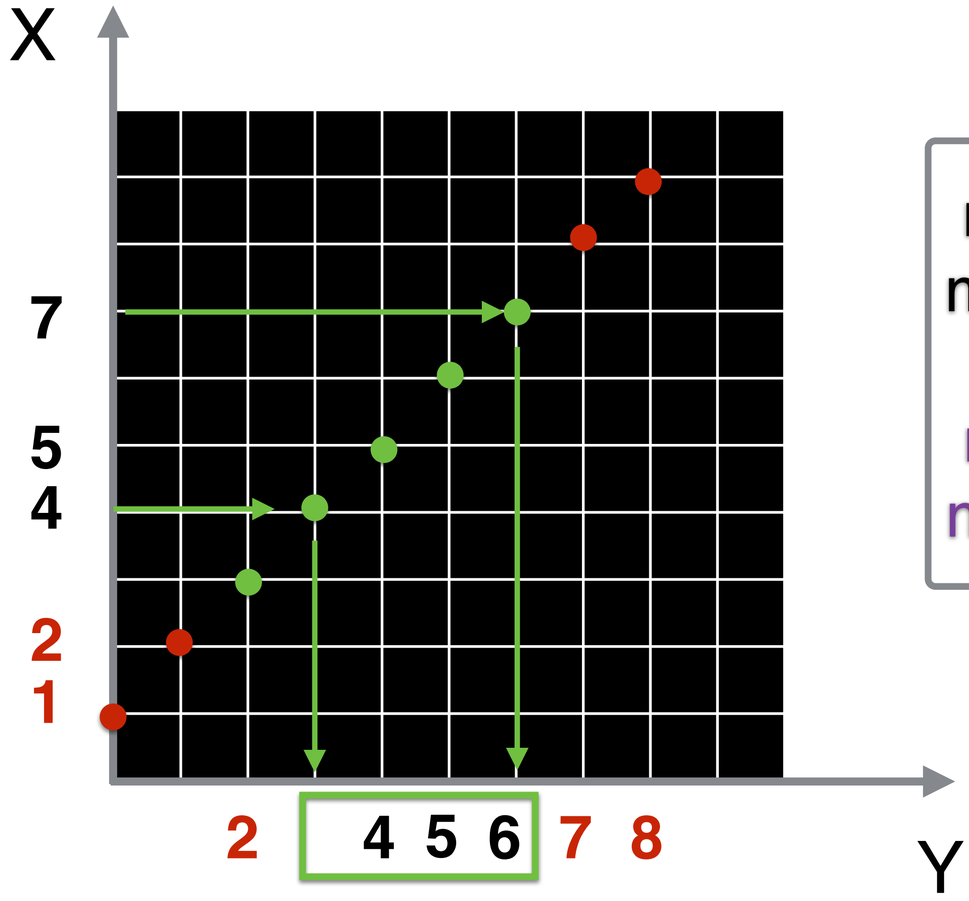
BC Filtering for X = Y+1: Reasoning on the Bounds



min(X) >= max(min(Y)+1,min(X))
max(X) <= min(max(Y)+1,max(X))

min(Y) >= max(min(X)–1,min(Y))
max(Y) <= min(max(X)–1,max(Y))

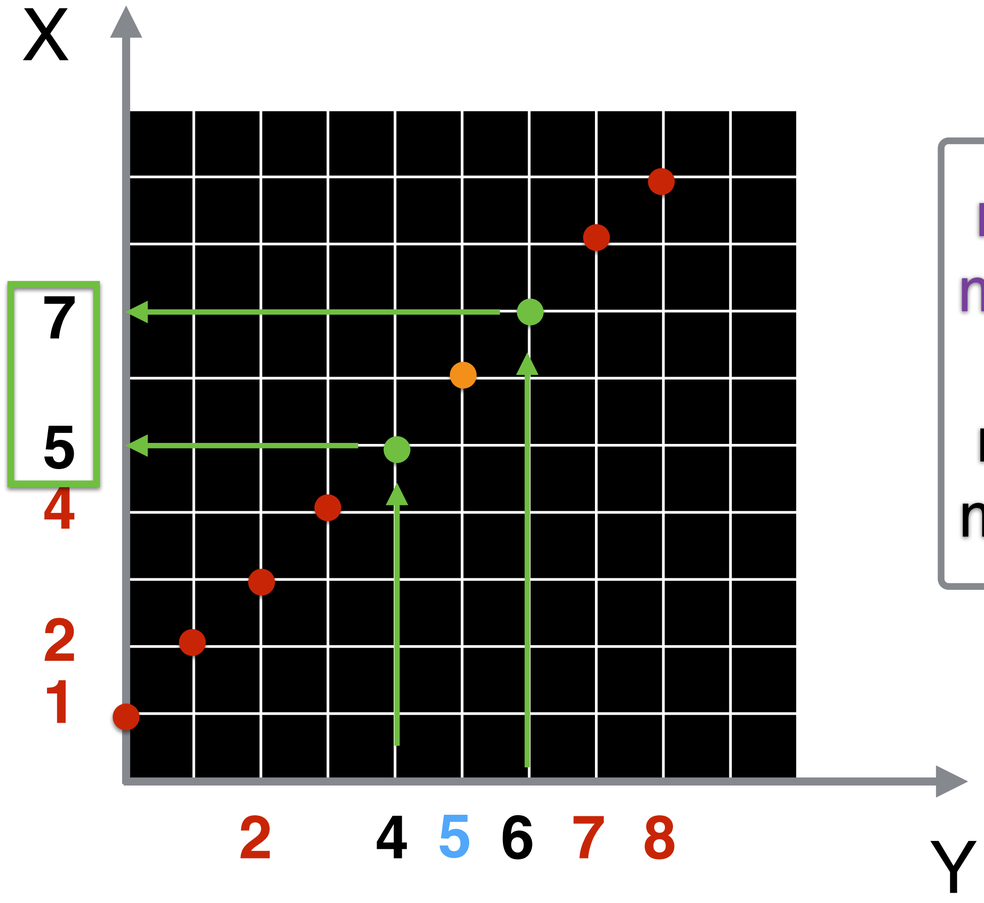
BC Filtering for X = Y+1: Reasoning on the Bounds



min(X) >= max(min(Y)+1,min(X))
max(X) <= min(max(Y)+1,max(X))

min(Y) >= max(min(X)–1,min(Y))
max(Y) <= min(max(X)–1,max(Y))

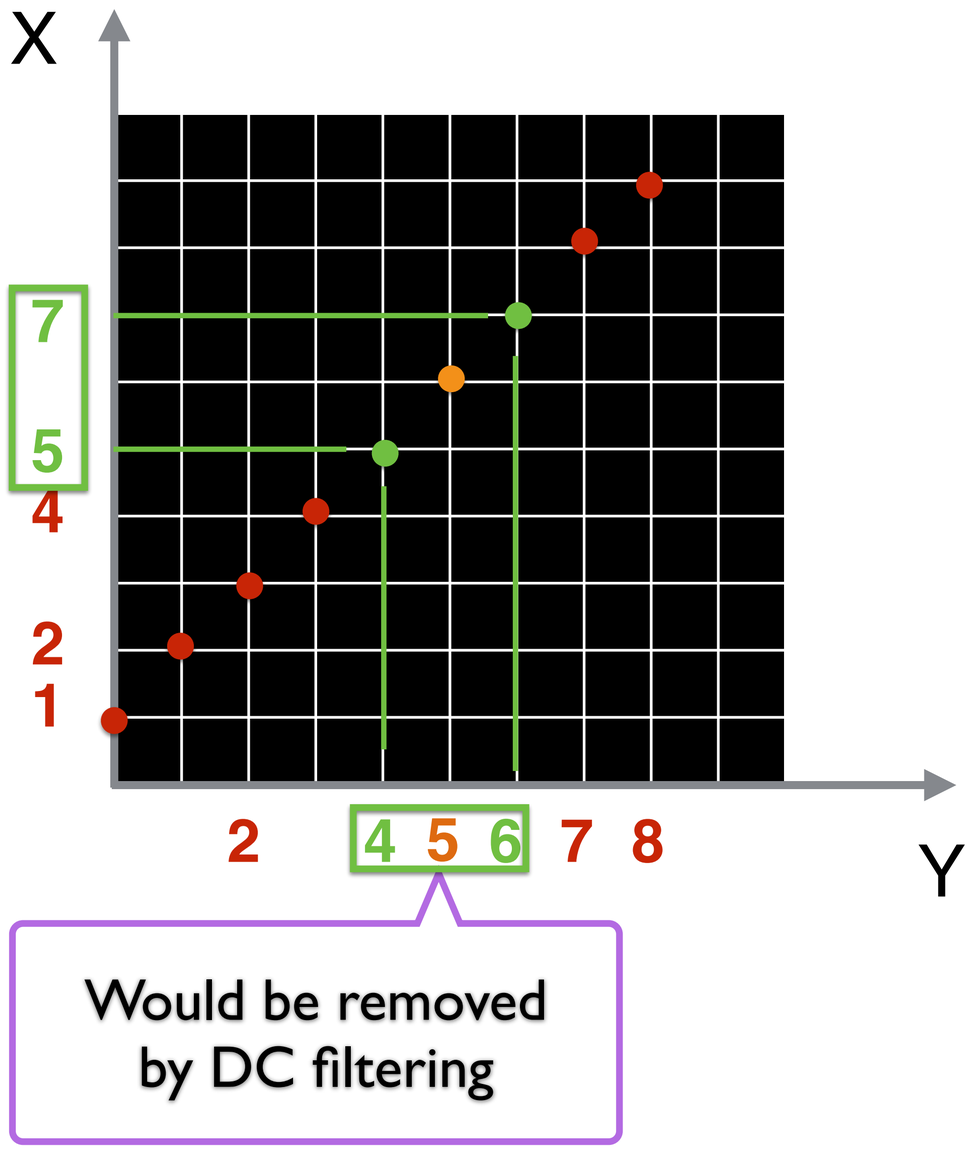
BC Filtering for X = Y+1: Reasoning on the Bounds



$$\min(X) \geq \max(\min(Y)+1, \min(X))$$
$$\max(X) \leq \min(\max(Y)+1, \max(X))$$

$$\min(Y) \geq \max(\min(X)-1, \min(Y))$$
$$\max(Y) \leq \min(\max(X)-1, \max(Y))$$

BC Filtering for X = Y+1: Reasoning on the Bounds



BC Filtering for X = Y+1 (Bound Consistent)

```
public class EqualPlusOneBC extends AbstractConstraint {

    IntVar x, y;

    @Override
    public void post() {
        x.propagateOnBoundChange(this);
        y.propagateOnBoundChange(this);
        propagate();
    }

    @Override
    public void propagate() {
        // propagate from x to y
        y.removeAbove(x.max()-1);
        y.removeBelow(x.min()-1);
        // propagate from y to x
        x.removeAbove(y.max()+1);
        x.removeBelow(y.min()+1);
    }
}
```

Time complexity: $O(1)$
if the domain representation enables $O(1)$
for `removeBelow` and `removeAbove`.

O(#removed values) for a sparse set.

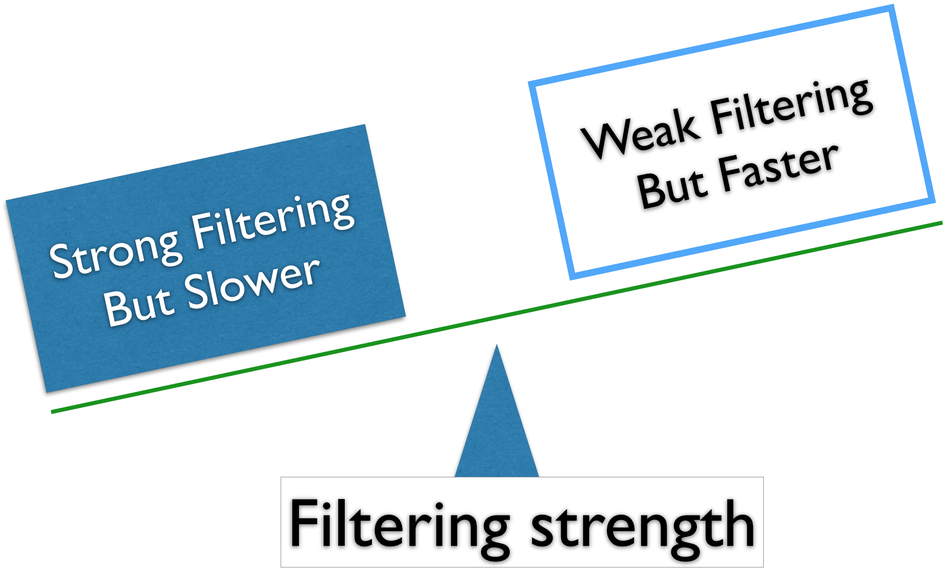
Filtering Powers

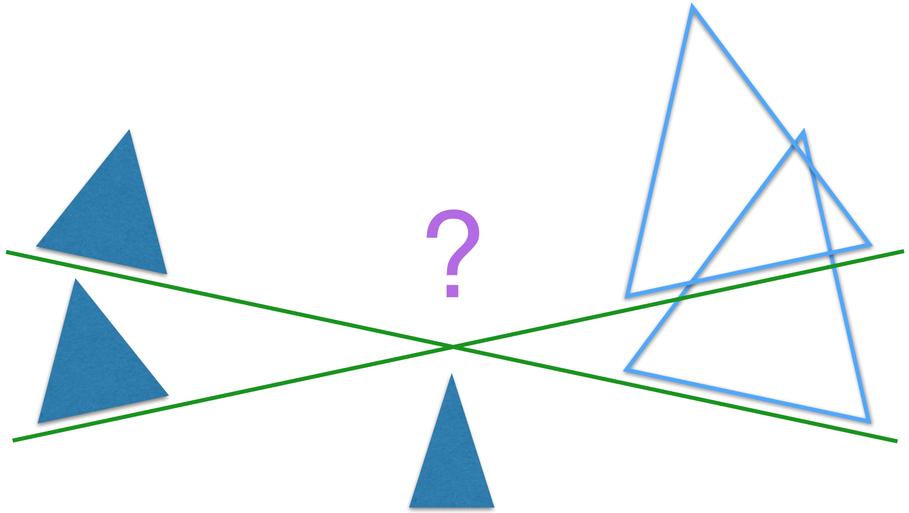
- Bound consistency: reason with the domain bounds only.
- Domain consistency: reason with all the domain values.

▸ Which one to prefer? It depends on the application!

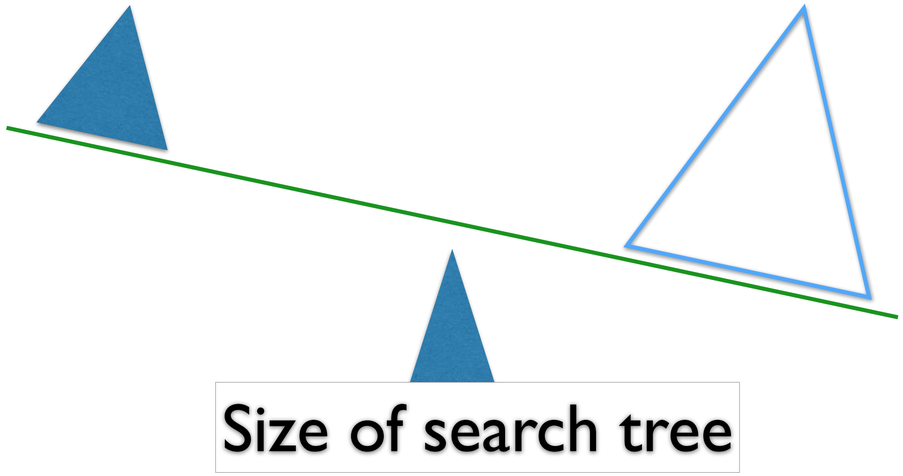
- Bound consistency:
  - Faster per execution: complexity depends on the number of variables.
  - Prunes less: larger search tree.
- Domain consistency:
  - Slower per execution: complexity depends on the sizes of the domains.
  - Prunes more: smaller search tree.

Filtering Strength: Discussion





Time to explore the tree?



Hard to predict!

Constraint Programming

Sum Constraint and Domain Consistency

Sum Constraint

- Definition:

$$\sum_i x_i = y \equiv \text{Sum(IntVar[] x, IntVar y)}$$

- Assume x = [{1,3}, {0,2,3}, {2,4,5,6}] and y = {6}:
  - Which values can be removed?

- How difficult is it to remove *all* the impossible values?

Complexity?

- Claim:
  - Removing all the impossible values for Sum(x,y) is **NP-hard**.
- Proof:
  - By **reduction** from subset-sum to removing all the impossible values.
  - Subset-sum is an NP-hard problem [https://en.wikipedia.org/wiki/Subset_sum_problem]:
    - Consider an integer set S of non-zero numbers.
    - Does any non-empty subset of S sum to 0?
    - Example: S = {–3 , 8 , –1 , –13 , 5 , 7} ✅ S = {–3 , 8 , –31 , 20 , 5} ❌

Proof

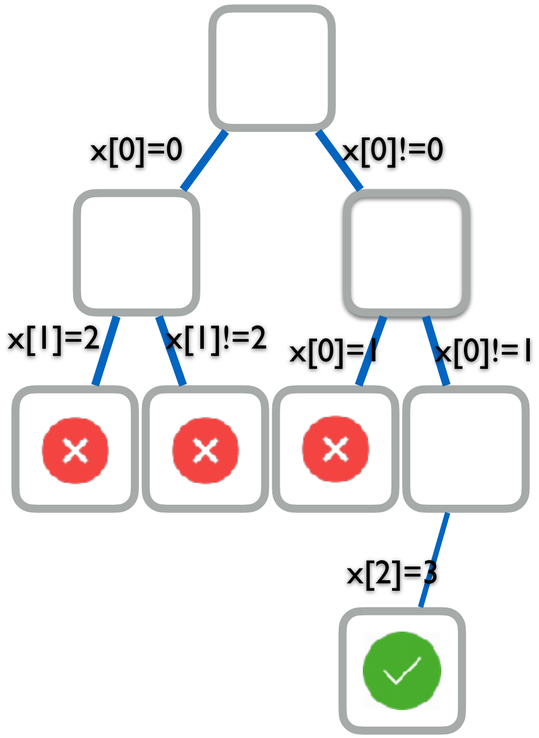
- Encoding:
  - Given a set $S = \{ a_1, \ldots, a_r \}$ of non-zero numbers.
  - Encode each a_i by a variable x_i with domain $D(x_i) = \{ 0, a_i \}$.
  - Create a variable y with domain $D(y) = \{ 0 \}$.
  - State the constraint $Sum(x,y)$.
  - **Filter** all the impossible values for $Sum(x,y)$.
- Outcomes:
  - $\forall\, i \in \{1..r\} : D(x_i) = \{0\}$ ➔ There are **NO** non-empty subsets that sum to 0.
  - $\exists\, i \in \{1..r\} : D(x_i) \neq \{0\}$ ➔ **YES**, at least one non-empty subset sums to 0.

So…

What does that mean?

Domain Consistency for Sum

- Achieving domain consistency is *hard!*
- Because:
  - We need to do so at *each* node of the search tree.
  - Each filtering can be very expensive: time exponential in the domain size. We do not want to do that (as it rarely pays off).



Constraint Programming

Sum Constraint and Bound Consistency

Bound Consistency: Example

- Consider:
  - $D(x_1) = [-100..10]$, $D(x_2) = [4..6]$, $D(x_3) = [20..60]$
  - Constraint: $x_1 + x_2 + x_3 = 0$
- Alternatively:
  - Constraint: $x_2 + x_3 = \underbrace{-x_1}_{z}$
  - Namely: $[4..6] + [20..60] = -[-100..10]$
    $[4..6] + [20..60] = [-10..100]$
- Apply filtering:
  - So: $[24..66] = [-10..100]$
    $\Rightarrow D(-x_1) = [-10..100] \cap [24..66] = [24..66]$
    $\Rightarrow D(x_1) = [-66..-24]$

BC Filtering for Sum: $x_1 + \ldots + x_n = 0$

- Feasibility check: Θ(n) time:
  - Feasibility implies $\sum_{j=1}^{n} x_j^{\min} \leq 0 \leq \sum_{j=1}^{n} x_j^{\max}$
- Filtering: Θ(n) time for each variable:

$$x_i^{\min} \leftarrow \max\left(x_i^{\min}, \sum_{j=1:j\neq i}^{n} - x_j^{\max}\right) \qquad x_i^{\max} \leftarrow \min\left(x_i^{\max}, \sum_{j=1:j\neq i}^{n} - x_j^{\min}\right)$$

- Example:

$[-100..10] + [4..6] + [20..60] = 0$

$-6 + -60 = -66 \Rightarrow x_1 \geq -66$

$-4 + -20 = -24 \Rightarrow x_1 \leq -24$

$-10 + -60 = -70 \Rightarrow x_2 \geq -70$

$-(-100) + -20 = 80 \Rightarrow x_2 \leq 80$

$-10 + -6 = -16 \Rightarrow x_3 \geq -16$

$-(-100) + -4 = 96 \Rightarrow x_3 \leq 96$

Bottom line

Can we do better?

Improving Runtime for Sum: $x_1 + \ldots + x_n = 0$

- Idea:
  - We keep recomputing the sums!
  - Make it incremental while scanning the array of variables.
- Steps:
  - Precompute in O(n) time: $s^{\min} \leftarrow \sum_{j=1}^{n} x_j^{\min}$ $s^{\max} \leftarrow \sum_{j=1}^{n} x_j^{\max}$
  - Check feasibility in O(1) time: $s^{\min} \leq 0 \leq s^{\max}$
  - Filter in O(1) time per variable: $x_i^{\min} \leftarrow \max(x_i^{\min}, -s^{\max} + x_i^{\max})$

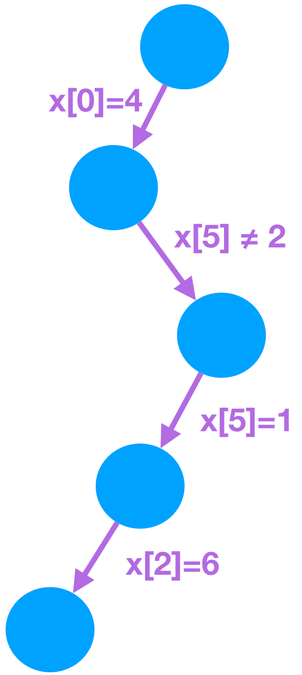
$$x_i^{\max} \leftarrow \min(x_i^{\max}, -s^{\min} + x_i^{\min})$$

Further Improvements?

- Make it incremental with respect to the search-tree traversal.
- Why?
  - As we descend in the search tree, variables become fixed.
  - Therefore, we should only loop over the variables that are *un*fixed.
- What is needed?
  - Track (during the search) the sum over the fixed variables.
  - Track (during the search) which variables are unfixed.

State

  - Use two StateInt for the sum and number of the fixed variables.
  - Use a stateful sparse set for the indices to the unfixed variables.
  - The state gets restored on backtrack.



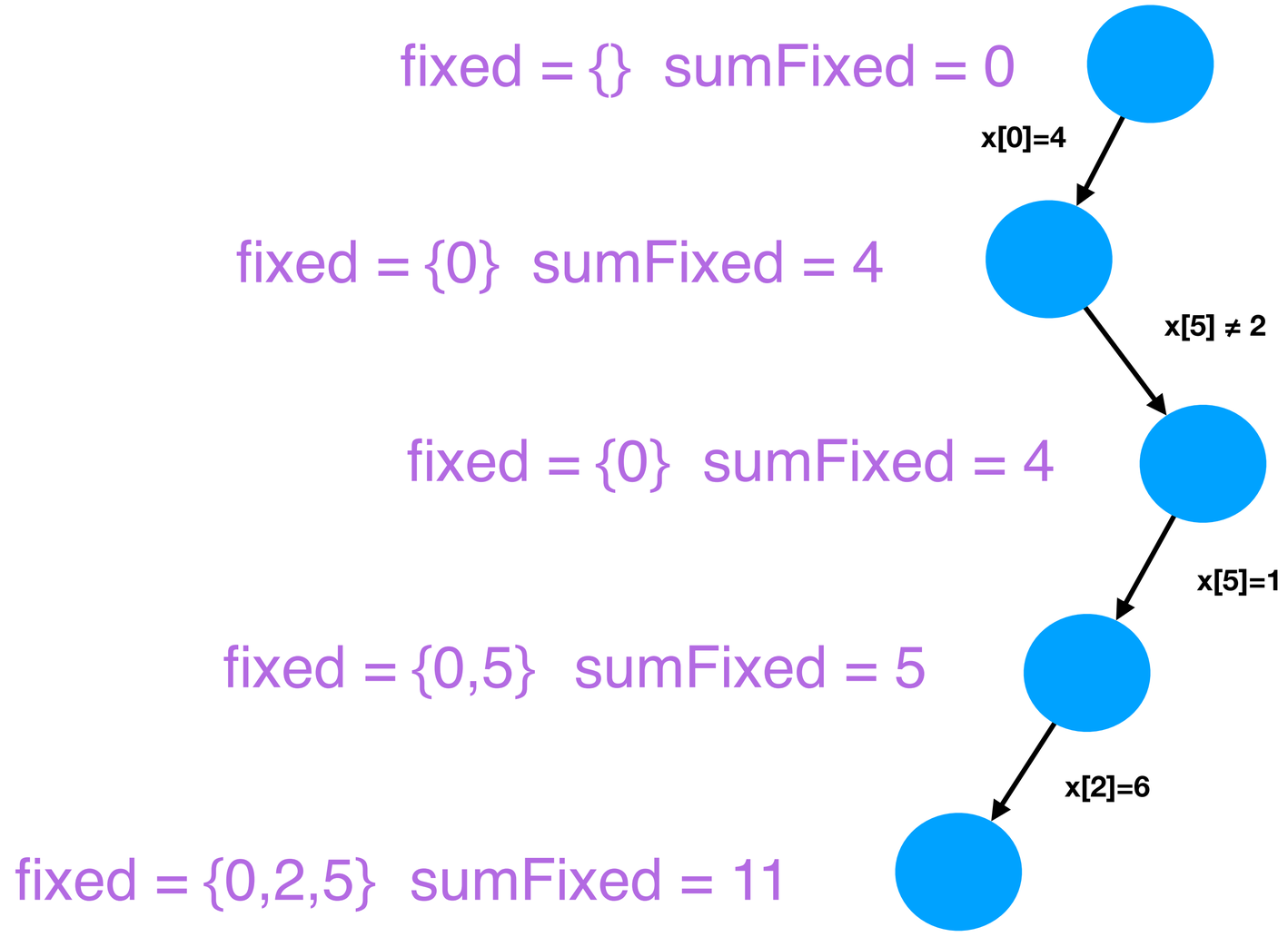
Process

- Check the tree:
  - The fixed set grows.

$$sumFixed \leftarrow \sum_{j \in fixed} x_j^{\min}$$

$$s^{\min} \leftarrow sumFixed + \sum_{j \notin fixed} x_j^{\min}$$

$$s^{\max} \leftarrow sumFixed + \sum_{j \notin fixed} x_j^{\max}$$



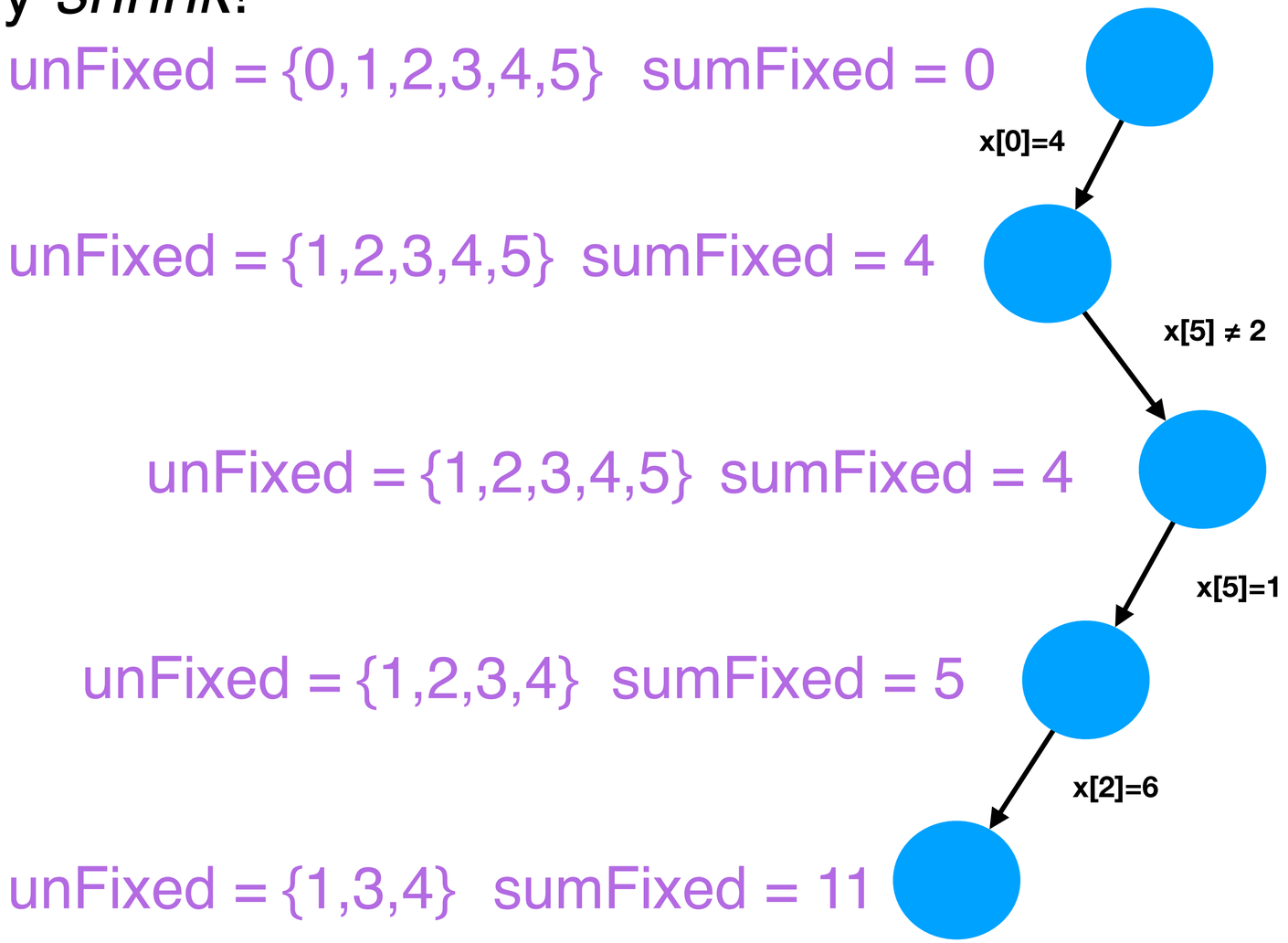
Revised Process

- But a stateful sparse set can only *shrink*!
  - Work with the complement!

$$sumFixed \leftarrow \sum_{j \in fixed} x_j^{\min}$$

$$s^{\min} \leftarrow sumFixed + \sum_{j \notin fixed} x_j^{\min}$$

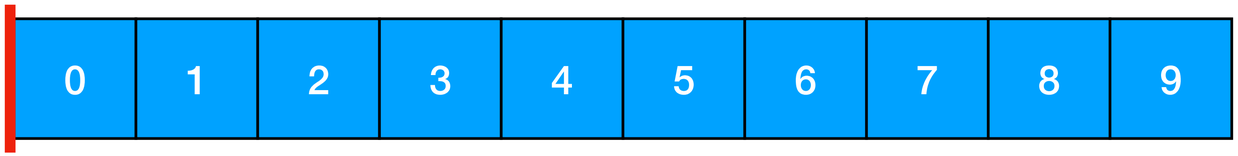
$$s^{\max} \leftarrow sumFixed + \sum_{j \notin fixed} x_j^{\max}$$

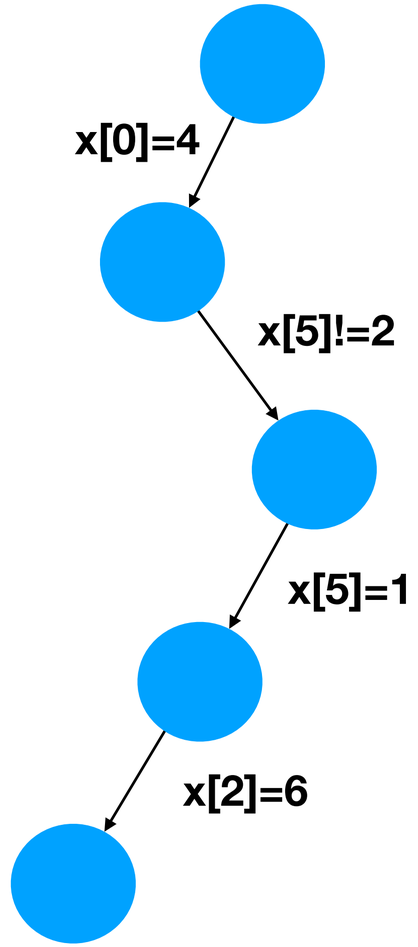


Incremental State Update for Sum

- StateInt: nFixed = 0
- StateInt: sumFixed = 0

$\sum_{j \in \text{fixed}} x_j^{min}$

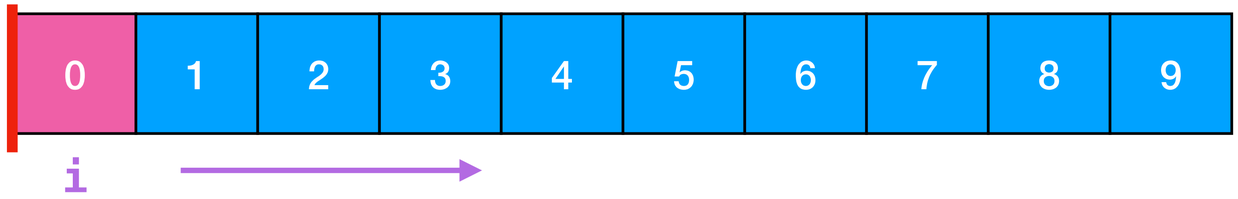


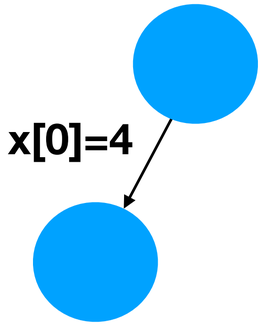



```
int nF = nFixed.value();
long sumMin = sumFixed.value(), sumMax = sumFixed.value();
for (int i = nF; i < x.length; i++) {
    int idx = fixed[i];
    min[idx] = x[idx].min(); sumMin += min[idx];
    max[idx] = x[idx].max(); sumMax += max[idx];
    if (x[idx].isFixed()) {
        sumFixed.setValue(sumFixed.value() + x[idx].min());
        fixed[i] = fixed[nF]; // Swap the variables
        fixed[nF] = idx;
        nF++;
    }
}
nFixed.setValue(nF);
```

Incremental State Update for Sum

- StateInt: nFixed = 1
- StateInt: sumFixed = 4

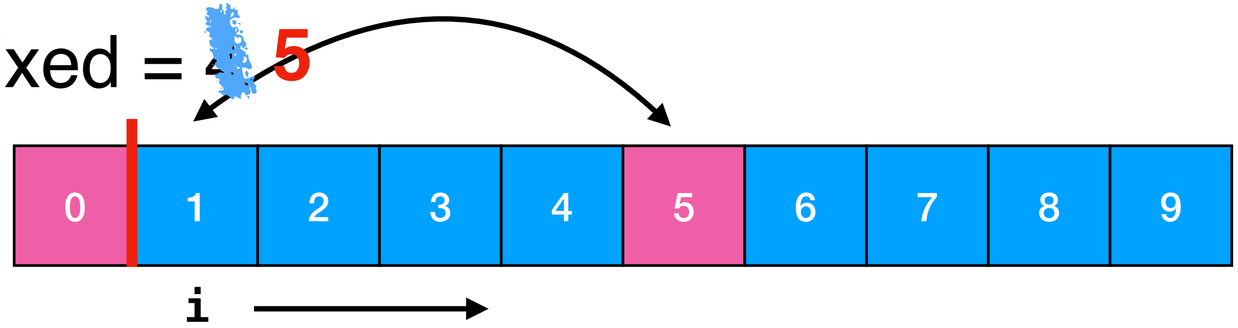


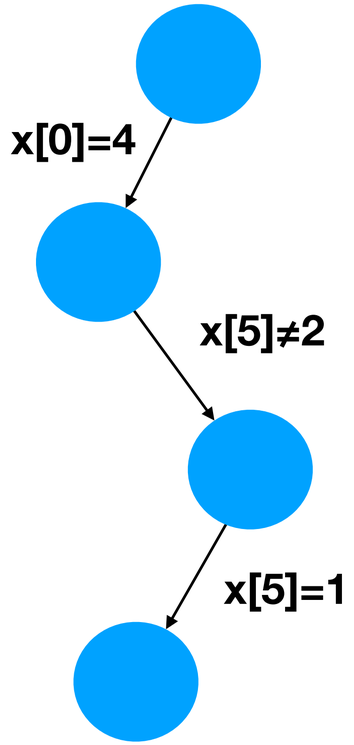



```
int nF = nFixed.value();
long sumMin = sumFixed.value(), sumMax = sumFixed.value();
for (int i = nF; i < x.length; i++) {
    int idx = fixed[i];
    min[idx] = x[idx].min(); sumMin += min[idx];
    max[idx] = x[idx].max(); sumMax += max[idx];
    if (x[idx].isFixed()) {
        sumFixed.setValue(sumFixed.value() + x[idx].min());
        fixed[i] = fixed[nF]; // Swap the variables
        fixed[nF] = idx;
        nF++;
    }
}
nFixed.setValue(nF);
```

Incremental State Update for Sum

- StateInt: nFixed = 2
- StateInt: sumFixed = 5

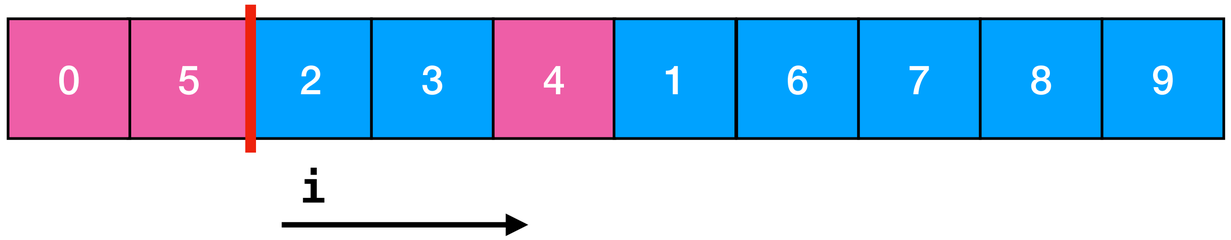


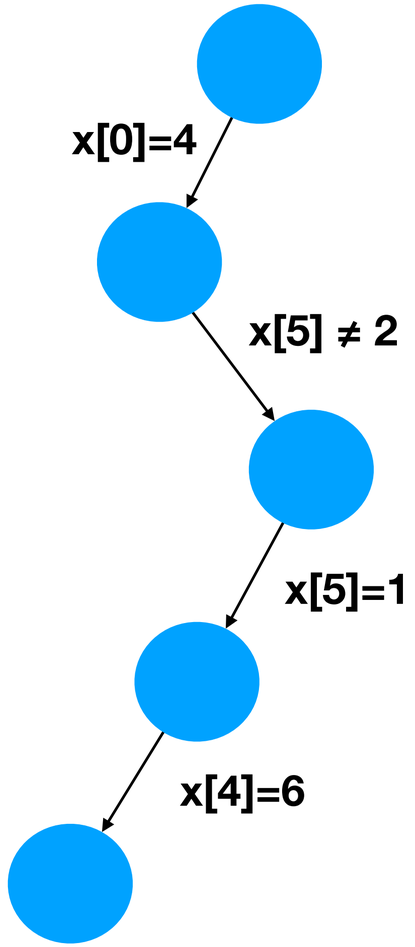



```
int nF = nFixed.value();
long sumMin = sumFixed.value(), sumMax = sumFixed.value();
for (int i = nF; i < x.length; i++) {
    int idx = fixed[i];
    min[idx] = x[idx].min(); sumMin += min[idx];
    max[idx] = x[idx].max(); sumMax += max[idx];
    if (x[idx].isFixed()) {
        sumFixed.setValue(sumFixed.value() + x[idx].min());
        fixed[i] = fixed[nF]; // Swap the variables
        fixed[nF] = idx;
        nF++;
    }
}
nFixed.setValue(nF);
```

Incremental State Update for Sum

- StateInt: nFixed = 2 3
- StateInt: sumFixed = 3 11

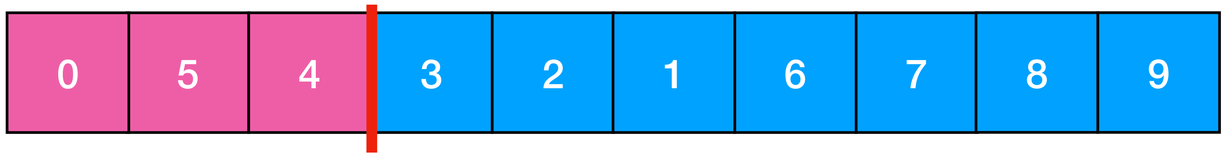


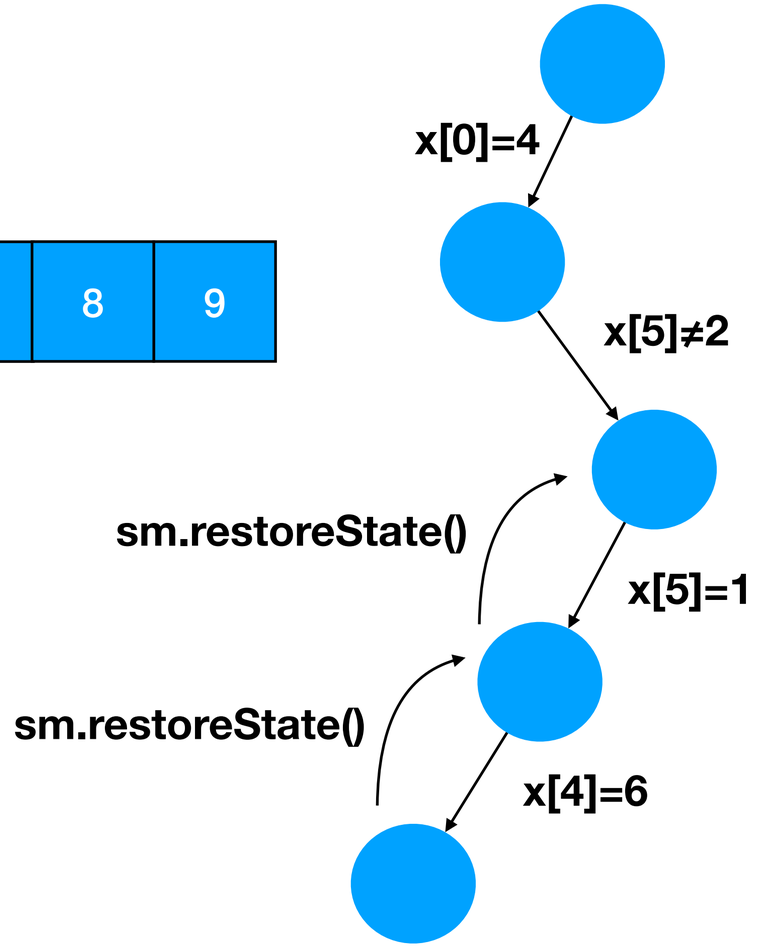



```
int nF = nFixed.value();
long sumMin = sumFixed.value(), sumMax = sumFixed.value();
for (int i = nF; i < x.length; i++) {
    int idx = fixed[i];
    min[idx] = x[idx].min(); sumMin += min[idx];
    max[idx] = x[idx].max(); sumMax += max[idx];
    if (x[idx].isFixed()) {
        sumFixed.setValue(sumFixed.value() + x[idx].min());
        fixed[i] = fixed[nF]; // Swap the variables
        fixed[nF] = idx;
        nF++;
    }
}
nFixed.setValue(nF);
```

And on Backtrack?

- StateInt: nFixed = 1
- StateInt: sumFixed = 4





Filtering Property

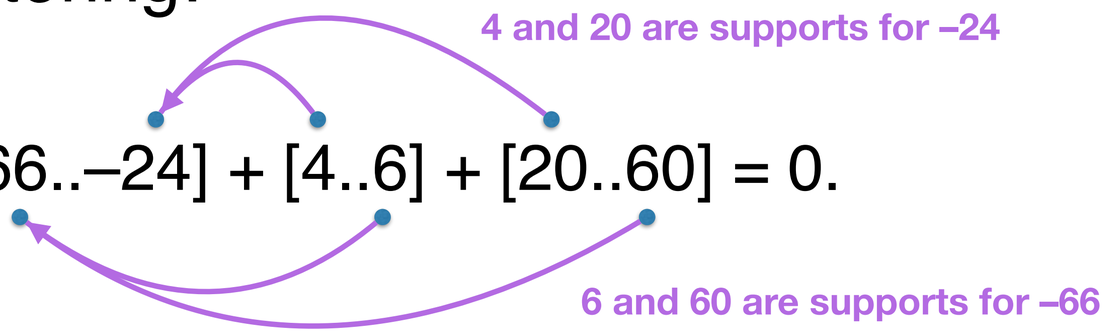
Idempotence

Idempotence

- A filtering algorithm is *idempotent* if executing it twice in a row always leads to exactly the same domains as just one execution: it reaches its own fixpoint in one execution.
- Consider the constraint $x_1 + \ldots + x_n = 0$:
  - Denote $D_1 = D(x_1)$, …, $D_n = D(x_n)$.
  - Let $[D'_1,\ldots,D'_n] = \text{Sum}^{BC}(D_1,\ldots,D_n)$ be the new domains after BC filtering.
- Question:
  - $\text{Sum}^{BC}(D_1,\ldots,D_n)$ **=?=** $\text{Sum}^{BC}(D'_1,\ldots,D'_n)$
  - If true, then sum^{BC} *is* idempotent.
- Relevance?
  - We can avoid extraneous scheduling!

Is Sum^{BC} Idempotent?

- Recall the previous example:
  - We had the constraint [–100..10] + [4..6] + [20..60] = 0.
- After BC filtering:



  - We got [–66..–24] + [4..6] + [20..60] = 0.
- Definition:
  - A *support* is a value that participates in a solution.
- Insight:
  - If one tightens an upper (resp. lower) bound in an interval domain, then all the other lower (resp. upper) bounds are supports and need thus not be reconsidered: conditional idempotence.

Is Sum^{BC} Idempotent?

- Let us modify the previous example:
  - Consider now the constraint $\{-100,-25,-24,10\} + [4..6] + [20..60] = 0$.

Is Sum^{BC} Idempotent?

- Let us modify the previous example:
  - Consider now the constraint $\{-100,-25,-24,10\} + [4..6] + [20..60] = 0$.
- After BC filtering once:
  - We get $-66 \leq \{-100,-25,-24,10\} \leq -24$
  - So $D(x_1) = \{-$[illegible]$,-25,-24,$[illegible]$\}$

Is Sum^{BC} Idempotent?

- Let us modify the previous example:
  - Consider now the constraint $\{-100,-25,-24,10\} + [4..6] + [20..60] = 0$.
- After BC filtering once:
  - We get $-66 \leq \{-100,-25,-24,10\} \leq -24$
  - So $D(x_1) = \{-25,-24,10\}$
- After BC filtering a second time:
  - We start from $\{-25,-24\} + [4..6] + [20..60] = 0$.
  - We have $-(-25) + (-20) = 5 \quad \Rightarrow x_2 \leq 5$

    and $-(-25) + (-4) = 21 \quad \Rightarrow x_3 \leq 21$
  - Hence $\{-25,-24\} + [4..5] + [20..21] = 0$.
  - So Sum^{BC} is in general *not* idempotent, because of holes in the domains.

Domain Views

Some (binary) constraints are easy

- Offset: X = Y + o,
- Opposite: X = –Y,
- Scale: X = a * Y (with a > 0)

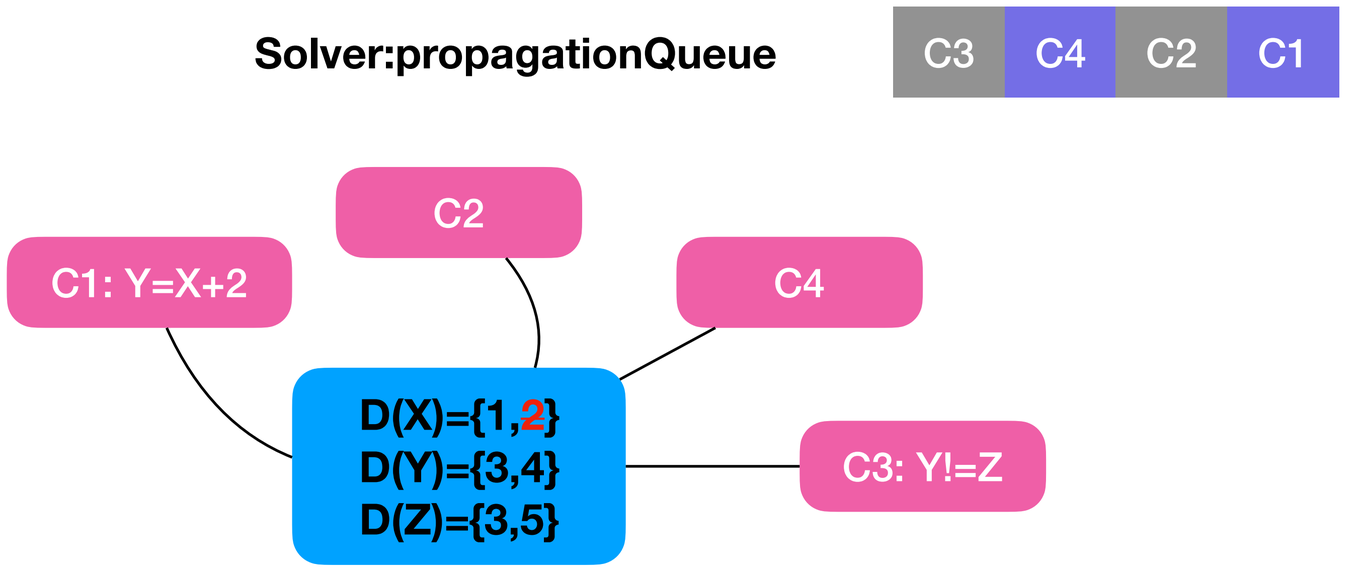
- We can create for each such small constraint an Object that extends Constraint in order to implement its filtering.
  Quite easy but there are big disadvantages with this architecture.

Disadvantages

Those constraints are functional: removing something from the domain of variable X or Y can directly be reflected on the domain of the other variable.

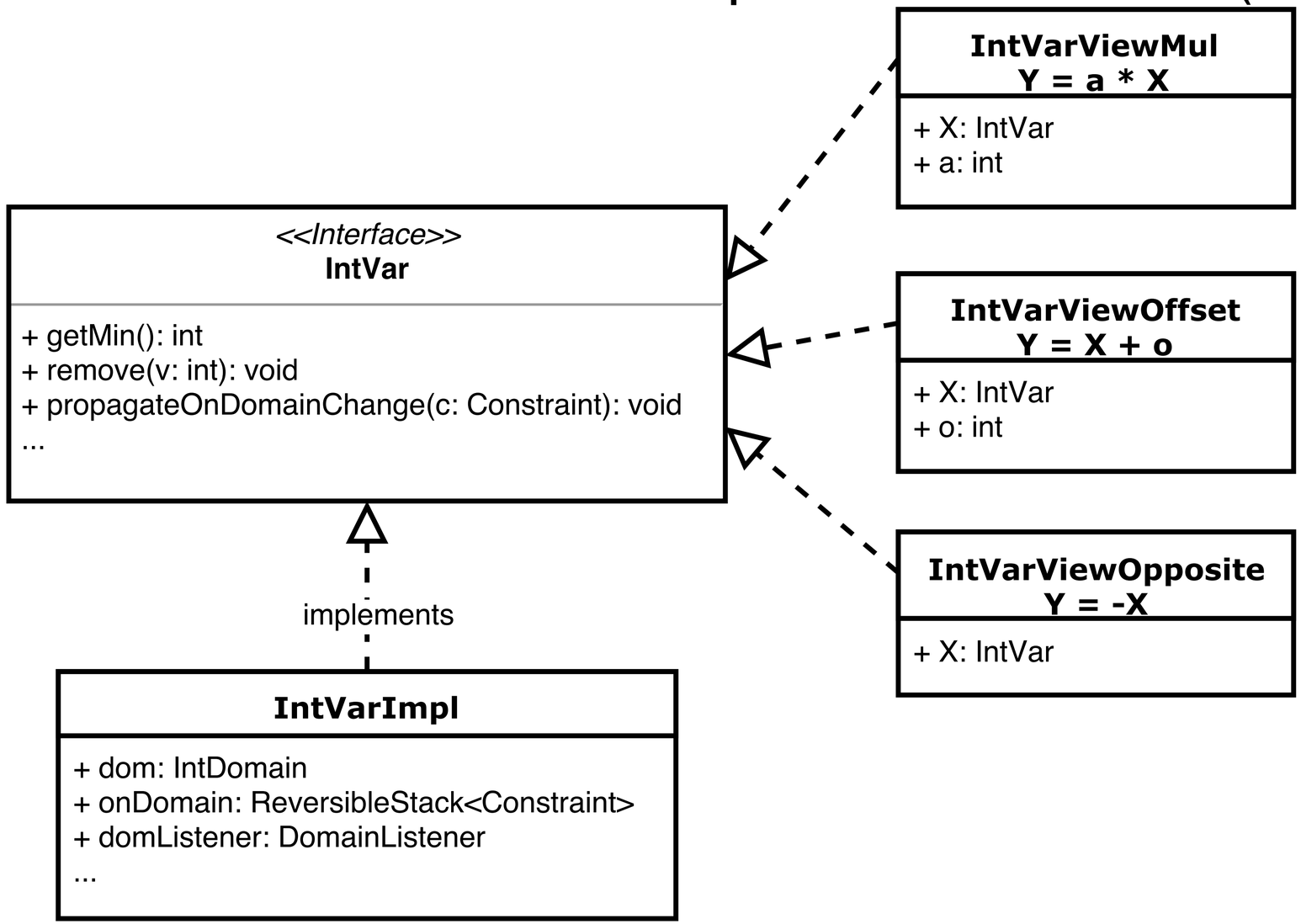
The fixpoint can take longer to compute because of them.

Example: Assume 2 is deleted from D(X): C3 is first propagated but has nothing to do, while first propagating C1 would have shortened the fixpoint computation:



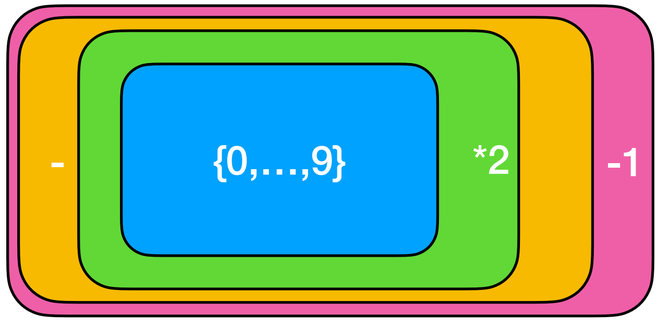
The idea

- Create a view on the Variables that wrap another variable (decorator pattern)



Usage

```
// D(x) = [0..9]
// y = -2*x - 1
// D(y)= {-19,…,-1}
IntVar y = minus(opposite( mul(makeIntVar(cp,10),2) ),1);
```




```
assertEquals(y.getMin(),-19);
assertEquals(y.getMax(),-1);
assertFalse(y.contains(10));
y.remove(-19);
assertFalse(y.contains(-19));
```

View Implementation (some methods)

```
public class IntVarViewOffset implements IntVar {

    private final IntVar x;
    private final int o;

   @Override
   public int getMin() {
       return x.getMin() + o;
   }

   @Override
   public void propagateOnDomainChange(Constraint c) {
       x.propagateOnDomainChange(c);
   }

   @Override
   public boolean isFixed() {
       return x.isFixed();
   }

   @Override
   public boolean contains(int v) {
       return x.contains(v - o);
   }

   @Override
   public void remove(int v) throws InconsistencyException {
       x.remove(v - o);
   }
}
```

Caveat Emptor: The Problem of Views

- Almost no filtering can be idempotent when views are allowed in a solver.
- Why:
  - Because views possibly have «holes» in their domains.
  - Because there directly is a side effect when updating a variable domain.
- Consider D(x) = [1..4], D(y) = view(2*x) = {2,4,6,8}.
- x+y = 5 actually means 3x = 5 (infeasible, but it takes 3 calls to detect it):
  - Propagation 1: $x \leq 3$, $y \leq 4$, but this will remove 3 from D(x), so we end up with D(x) = [1..2], D(y) = view(2*x) = {2,4}.
  - Propagation 2: $3 \leq y$, but this will remove 1 from D(x), so we end up with D(x) = {2}, D(y) = view(2*x) = {4}.
  - Propagation 3: this is infeasible, as $6 \leq 5 \leq 6$ does not hold.

Boolean variables

Boolean Variables

- Boolean variables are important to express logical constraints, such as

$$(X \mid Y) \,\&\, Z$$

- Ideally they should be considered as 0/1 variables.
- Let X[] be an array of IntVar, let Y and Z be two IntVar, and we want Z to be the number of entries in X[] that are larger than Y:

$$Z = \sum_i (Y < X[i])$$

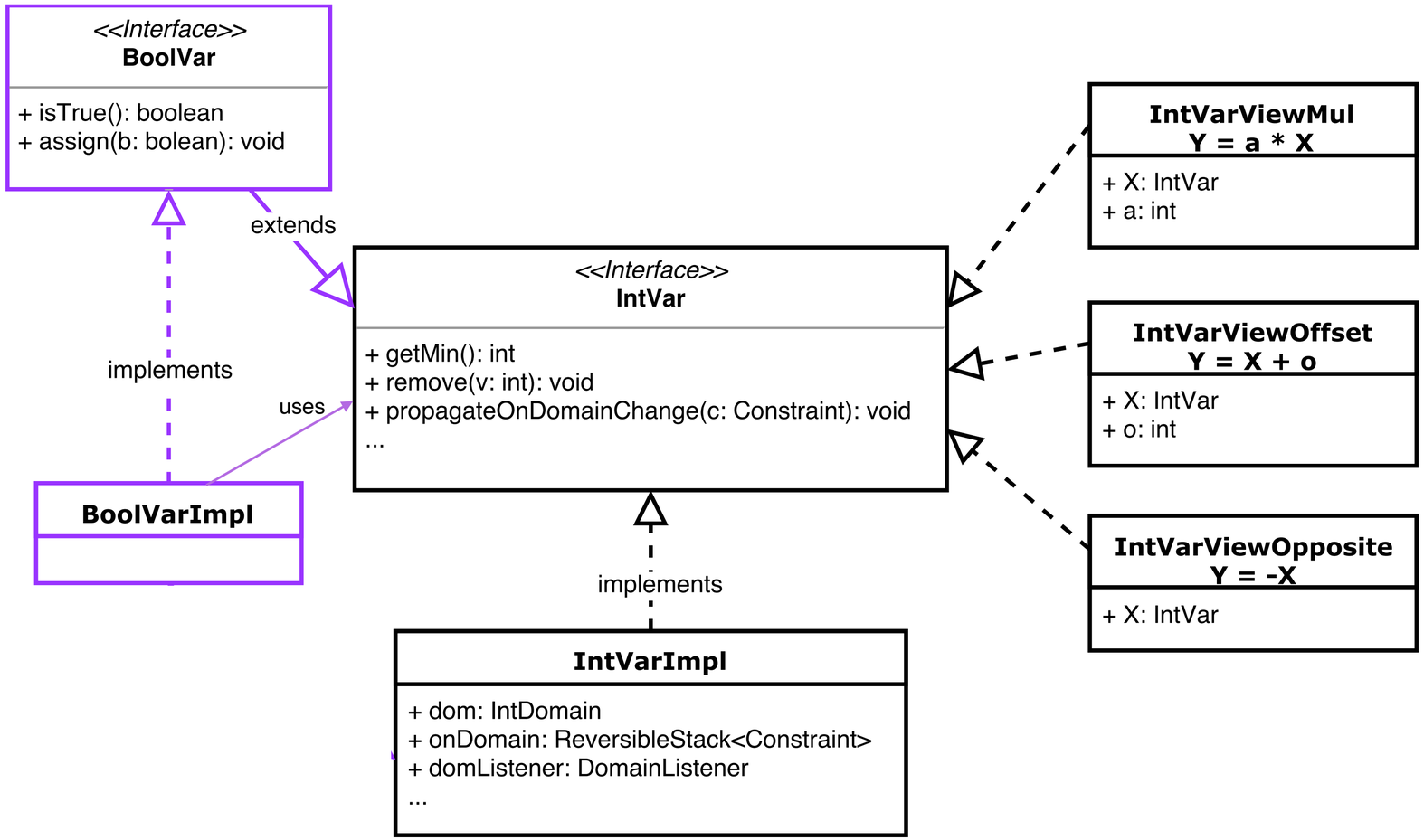
This is a Boolean variable but since it is appearing in a Sum constraint it should act as a 0/1 variable

Solution 1: Create a constraint

- BoolToInt(B: BoolVar, Y: IntVar)
  Y = 1 if and only if B = true, and Y = 0 otherwise
- If you have many Boolean variables involved in some arithmetic constraints this will slow down considerably the computation of the fixpoint.

Solution 2: Extend IntVar

- A BoolVar is thus a 0/1 IntVar.
  It can be used in any constraint expecting an IntVar (sums, allDiff, etc).



Solution 2

```
public interface BoolVar extends IntVar {

    /**
     * Tests if the variable is fixed to true
     * @return true if the variable is fixed to true (value 1)
     */
    boolean isTrue();

    /**
     * Tests if the variable is fixed to false
     * @return true if the variable is fixed to false (value 0)
     */
    boolean isFalse();

    /**
     * Assigns the variable
     * @param b the value to assign to this boolean variable
     * @exception InconsistencyException
     *            is thrown if the value is not in the domain
     */
    void fix(boolean b);


}
```

Implementation

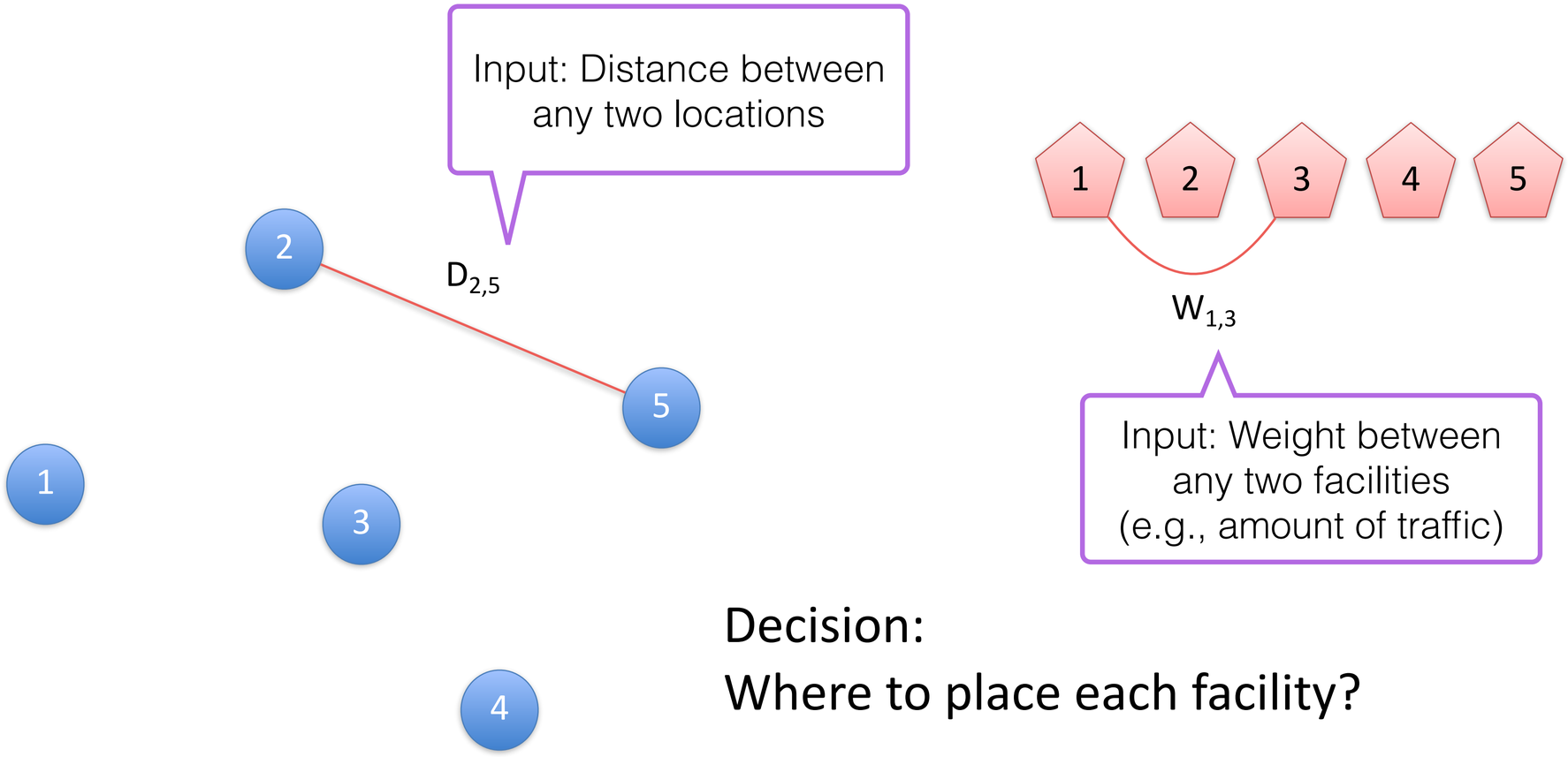
```java
public interface BoolVar extends IntVar {

    /**
     * Tests if the variable is fixed to true
     * @return true if the variable is fixed to true (value 1)
     */
    boolean isTrue();

    /**
     * Tests if the variable is fixed to false
     * @return true if the variable is fixed to false (value 0)
     */
    boolean isFalse();

    /**
     * Assigns the variable
     * @param b the value to assign to this boolean variable
     * @exception InconsistencyException
     *            is thrown if the value is not in the domain
     */
    void fix(boolean b);


}
```

```java
public class BoolVarImpl implements BoolVar {

    private IntVar binaryVar;


    public BoolVarImpl(IntVar binaryVar) {
        if (binaryVar.max() > 1 || binaryVar.min() < 0) {
            throw new IllegalArgumentException("must be a binary {0,1} variable");
        }
        this.binaryVar = binaryVar;
    }
    @Override
    public boolean isTrue() {
        return min() == 1;
    }

    @Override
    public boolean isFalse() {
        return max() == 0;
    }

    @Override
    public int min() {
        return binaryVar.min();
    }

    @Override
    public int max() {
        return binaryVar.max();
    }

    @Override
    public void fix(int v) {
        binaryVar.fix(v);
    }

    @Override
    public void fix(boolean b) {
        fix(b ? 1 : 0);
    }
}
```

Quadratic Assignment

Element Constraints

Quadratic Assignment Problem (QAP)

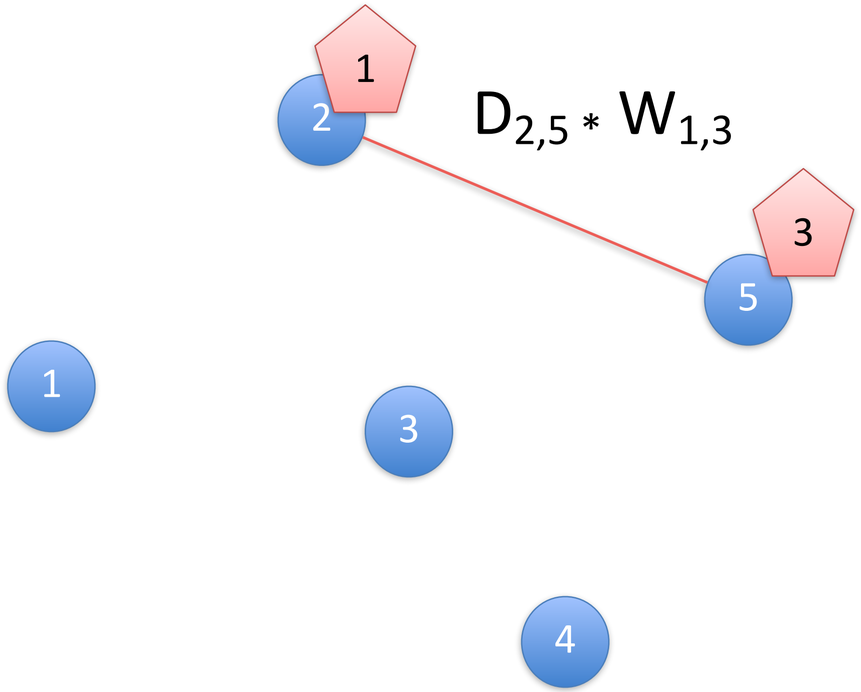


Decision:

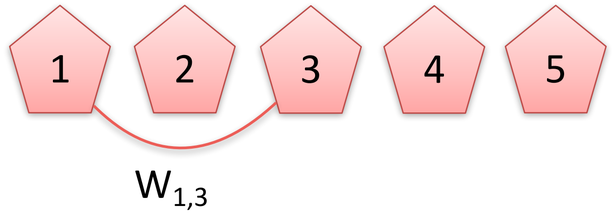
Where to place each facility?

QAP

Locations:



Facilities:



Problem:

Assign all facilities to different locations

(let x_i denote the location of facility i), minimizing

$$\sum_{i,j} D_{x_i,x_j} \cdot W_{i,j}$$

2D element constraint:
2D array indexed by two variables

Quadratic Assignment Model

```
Solver cp = makeSolver();
IntVar[] x = makeIntVarArray(cp, n, n);

cp.post(allDifferent(x));


// build the objective function
IntVar[] weightedDist = new IntVar[n * n];
for (int k = 0, i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        weightedDist[k] = mul(element(d, x[i], x[j]), w[i][j]);
        k++;
    }
}
IntVar totCost = sum(weightedDist);
Objective obj = cp.minimize(totCost);
```

$$D_{x_i,x_j} \cdot W_{i,j}$$

Element2D(int[][] T, IntVar x, IntVar y, IntVar z)

- T[x][y] = z

y

x	0	1	2	3
0	1	8	9	6
1	1	9	2	4
2	9	8	9	8
3	1	9	2	5

- How to create an efficient propagator for Element2D?
- We don’t want to create holes in D(z), but holes are fine in D(x) and D(y).

2D Element Constraint

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	4
3	1	9	2	5	4
cSup	4	4	4	4	

sorted

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..9] (interval domain)

T[x][y] = z

y

x	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	4
3	1	9	2	5	4
cSup	4	4	4	4	

low →

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

up →

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- Assume D(z) = [1..7] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	4
3	1	9	2	5	3
cSup	4	3	4	4	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}

D(z) = [1..**7**] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	3
3	1	9	2	5	3
cSup	4	3	3	4	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	2
3	1	9	2	5	3
cSup	3	3	3	4	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}

D(z) = [1..7] (interval domain)

T[x][y] = z

y

x	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	3
2	9	8	9	8	2
3	1	9	2	5	3
cSup	3	2	3	4	

low →

up →

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	3
1	1	9	2	4	3
2	9	8	9	8	2
3	1	9	2	5	3
cSup	3	2	2	4	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

T[x][y] = z

y

x	0	1	2	3	rSup
0	1	8	9	6	3
1	1	9	2	4	3
2	9	8	9	8	1
3	1	9	2	5	3
cSup	3	2	2	3	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

T[x][y] = z

y

x	0	1	2	3	rSup
0	1	8	9	6	3
1	1	9	2	4	3
2	9	8	9	8	0
3	1	9	2	5	3
cSup	3	1	2	3	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,~~2~~,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

T[x][y] = z

y

x	0	1	2	3	rSup
0	1	8	9	6	2
1	1	9	2	4	3
2	9	8	9	8	0
3	1	9	2	5	3
cSup	3	0	2	3	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,~~2~~,3}
- D(y) = {0,~~1~~,2,3}
- D(z) = [1..6,~~7~~] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	2
1	1	9	2	4	3
2	9	8	9	8	0
3	1	9	2	5	3
cSup	3	0	2	3	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,3}
- D(y) = {0,2,3}
- Assume D(z) = [2..6] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	2
2	9	8	9	8	0
3	1	9	2	5	2
cSup	0	0	2	3	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,3}
- D(y) = {~~0~~,2,3}
- D(z) = [2..6] (interval domain)

T[x][y] = z

y

x	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	2
2	9	8	9	8	0
3	1	9	2	5	2
cSup	0	0	2	3	

low →

up →

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,3}
- Assume D(y) = {~~2~~,3}
- D(z) = [2..6] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	2
2	9	8	9	8	0
3	1	9	2	5	2
cSup	0	0	2	3	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,3}
- D(y) = {~~2~~,3}
- D(z) = [2..6] (interval domain)

T[x][y] = z

y

x	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	1
2	9	8	9	8	0
3	1	9	2	5	1
cSup	0	0	0	3	

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,3}
- D(y) = {~~2~~,3}
- D(z) = [~~2,3~~,4..6] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	1
2	9	8	9	8	0
3	1	9	2	5	1
cSup	0	0	0	3	

low

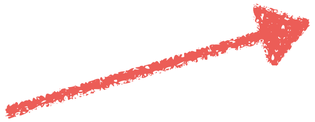
up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

Why do we only need to restore these values on backtrack?

Implementation 1/2

```
public class Element2D extends AbstractConstraint {

        private final int[][] T;
        private final IntVar x, y, z;
        private int n, m;
        private final StateInt[] rSup;
        private final StateInt[] cSup;

        private final StateInt low;
        private final StateInt up;
        private final ArrayList<Triple> zxy;

        @Override
        public void post() {
            … // some init
            x.propagateOnDomainChange(this);
            y.propagateOnDomainChange(this);
            z.propagateOnBoundChange(this);
            propagate();
        }
}
```



z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

```
private void updateSupports(int lostPos) {
    if (rSup[zxy.get(lostPos).x].decrement() == 0)
        x.remove(zxy.get(lostPos).x);
    if (cSup[zxy.get(lostPos).y].decrement() == 0)
        y.remove(zxy.get(lostPos).y);
}
public void propagate() {
    int l = low.value(), u = up.value();
    int zMin = z.min(), zMax = z.max();

    while (zxy.get(l).z < zMin ||
           !x.contains(zxy.get(l).x) ||
           !y.contains(zxy.get(l).y)) {
        updateSupports(l++);
        if (l > u) throw new InconsistencyException();
    }
    z.removeBelow(zxy.get(l).z);
    low.setValue(l);
    // similarly for up
}
```

We decrement the support counters as we only remove values. The state manager will take care to restore everything.

Set the low value to the first consistent entry in the table. The state manager will restore it on backtrack.

1D Element Constraint

Element1D Domain Consistency

T[y]=z, D(y)={0,1,2,3,4,5}, D(z)={3,4,5}

D(y)	0	1	2	3	4	5
T	3	4	5	5	4	3

zSup	2	2	2
D(z)	3	4	5

Element1D Domain Consistency

T[y]=z, D(y)={0,1,2,3,4,5}, assume D(z)={~~3~~,4,5}

D(y)	0	1	2	3	4	5
T	3	4	5	5	4	3

Filtering: T[i] ∉ D(z) ⇒ D(y) ← D(y) \ { i }

zSup	2	2	2
D(z)	3	4	5

Element1D Domain Consistency

T[y]=z, D(y)={~~0~~,1,2,3,4,~~5~~}, D(z)={~~3~~,4,5}

D(y)	0	1	2	3	4	5
T	3	4	5	5	4	3

Filtering: T[i] ∉ D(z) ⇒ D(y) ← D(y) \ { i }

zSup	2	2	2
D(z)	3	4	5

Element1D Domain Consistency

- T[y]=z, assume D(y)={~~1~~,2,3,~~4~~} ,D(z)={4,5},

	0	1	2	3	4	5
D(y)						
T	3	4	5	5	4	3

	2	2	2
zSup			
D(z)	3	4	5

For each value v in D(z), set
zSup(v) ← | { i in D(y) : T[i] = v } |
Filtering: zSup(v) = 0 ⇒ D(z) ← D(z) \ { v}

Element1D Domain Consistency

- T[y]=z, assume D(y)={~~1~~,2,3,~~4~~} ,D(z)={~~4~~,5},

D(y)	0	1	2	3	4	5
T	3	4	5	5	4	3

Filtering: T[i] ∉ D(z) ⇒ D(y) ← D(y) \ { i }

zSup	2	0	2
D(z)	3	4	5

For each value v in D(z), set
zSup(v) ← | { i in D(y) : T[i] = v } |
Filtering: zSup(v) = 0 ⇒ D(z) ← D(z) \ { v}

Implementation and Time Complexity T[y]=z

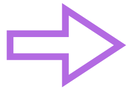
For each value i in D(y):

$T[i] \notin D(z) \Rightarrow D(y) \leftarrow D(y) \setminus \{ i \}$

For each value v in D(z):

$zSup(v) \leftarrow |\{ i \text{ in } D(y) : T[i] = v \}|$

$zSup(v) = 0 \Rightarrow D(z) \leftarrow D(z) \setminus \{ v\}$



supports = {}

For each value i in D(y):

    supports = supports ∪ {T[i]}

For each value v in D(z):

    If v not in supports:

        $D(z) \leftarrow D(z) \setminus \{ v\}$

Element Application

Stable Matching Problem

Stable Matching

Students

Google

IBM

NASA

SAP

Companies

Inputs, say for internships:

- Every company provides a ranking of all the students.
- Every student provides a ranking of all the companies.

Stable Matching

- A matching of student Halle with IBM is *stable* if:
  - If IBM prefers another student, say George, over Halle, then George must prefer his matched company over IBM.
  - If student Halle prefers another company, say NASA, over IBM, then NASA must prefer their matched student over Halle.

- These stability rules make a matching stable!

More precisely

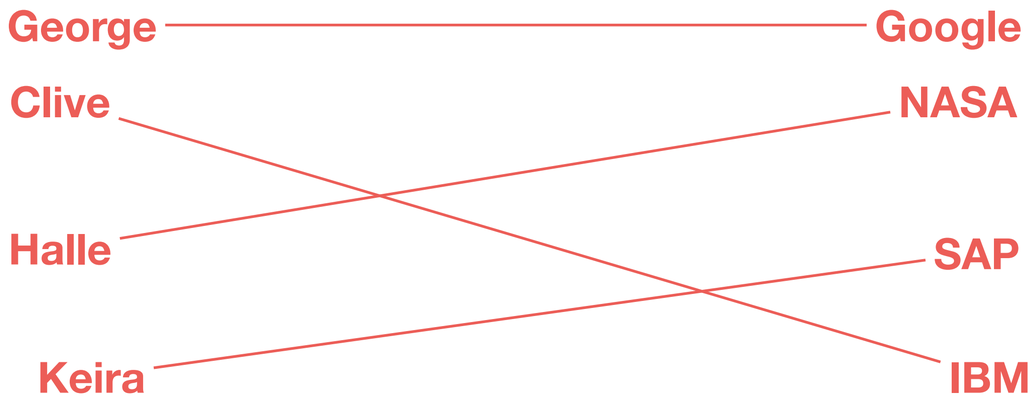
- Input:

  - Given are n students and n companies, where each student (resp. company) has ranked each company (resp. student) with a unique number between 1 and n in order of preference (the lower the number, the higher the preference), say for summer internships.

- Problem:

  - Match the students and companies such that there is no pair of a student and a company who would both prefer to be matched with each other than with their actually matched ones.

Input/Output



IBM prefers Keira over Clive (1 vs 2) (smaller number is higher preference).
But Keira prefers her matched company (SAP) over IBM (2 vs 4).

rankStudents[c,s]

	George	Halle	Keira	Clive
Google	1	2	3	4
IBM	4	3	1	2
NASA	1	2	3	4
SAP	3	4	1	2

rankCompanies[s,c]

	Google	IBM	NASA	SAP
George	1	2	3	4
Halle	2	1	3	4
Keira	1	4	3	2
Clive	4	3	1	2

Difficult Problem?

Not really:

```
function stableMatching {
    Initialize all m ∈ M and w ∈ W to free
    while ∃ free man m who still has a woman w to propose to {
       w = first woman on m's list to whom m has not yet proposed
       if w is free
         (m, w) become engaged
       else some pair (m', w) already exists
         if w prefers m to m'
            m' becomes free
           (m, w) become engaged
         else
           (m', w) remain engaged
    }
}
```

In 1962, David Gale and Lloyd Shapley proved that, for any equal number of men and women, it is always possible to solve the stable matching problem and make all marriages stable.

https://en.wikipedia.org/wiki/Gale-Shapley_algorithm#Algorithm

Stable Matching

Data and variables

`rankCompanies[Halle,Google]` is
the ranking of Google in Halle's preferences

```
enum Students = {George,Halle,Keira,Clive};
enum Companies = {Google,IBM,NASA,SAP};

int rankCompanies[Students,Companies];
int rankStudents[Companies,Students];
...

var{Companies} company[Students];
var{Students} student[Companies];
```

`rankStudents[Google,Halle]` is
the ranking of Halle in the preferences of Google

Stable Matching

```
solve {
   forall(s in Students)
      student[company[s]] = s;
   forall(c in Companies)
      company[student[c]] = c;

   ...
}
```

Stable Matching

```
solve {
   forall(s in Students)
      student[company[s]] = s;
   forall(c in Companies)
      company[student[c]] = c;

   forall(s in Students, c in Companies)
     rankCompanies[s,c] < rankCompanies[s,company[s]]
      => rankStudents[c,student[c]] < rankStudents[c,s];
   forall(c in Companies, s in Students)
      rankStudents[c,s] < rankStudents[c,student[c]]
      => rankCompanies[s,company[s]] < rankStudents[s,c];
}
```

s prefers c over their company

Stable Matching

```
solve {
   forall(s in Students)
      student[company[s]] = s;
   forall(c in Companies)
      company[student[c]] = c;

   forall(s in Students, c in Companies)
      rankCompanies[s,c] < rankCompanies[s,company[s]]
      => rankStudents[c,student[c]] < rankStudents[c,s];
   forall(c in Companies, s in Students)
      rankStudents[c,s] < rankStudents[c,student[c]]
      => rankCompanies[s,company[s]] < rankStudents[s,c];
}
```

c prefers their student over s

Stable Matching

```
enum Students = {George,Halle,Keira,Clive};
enum Companies = {Google,IBM,NASA,SAP};
int rankCompanies[Students,Companies];
int rankStudents[Companies,Students];
...
var{Companies} company[Students];
var{Students} student[Companies];
solve {
   forall(s in Students)
      student[company[s]] = s;
   forall(c in Companies)
      company[student[c]] = c;

   forall(s in Students, c in Companies)
      rankCompanies[s,c] < rankCompanies[s,company[s]]
      => rankStudents[c,student[c]] < rankStudents[c,s];
   forall(c in Companies, s in Students)
      rankStudents[c,s] < rankStudents[c,student[c]]
      => rankCompanies[s,company[s]] < rankStudents[s,c];
}
```

non-standard Element constraints T[y]=z because T is an array of variables

logical constraints

Stable Matching

- Two interesting features:
  - Element constraint over an array of variables:
    useful in many applications.
  - Logical combination of constraints.
- The Element constraint:
  - Ability to index an array/matrix with a variable or an expression containing variables.
- Logical combination of constraints:
  - Can be handled by reification, for instance.

Element Constraint

Over array of variables: relaxed domain consistency

Element1DVar Constraint

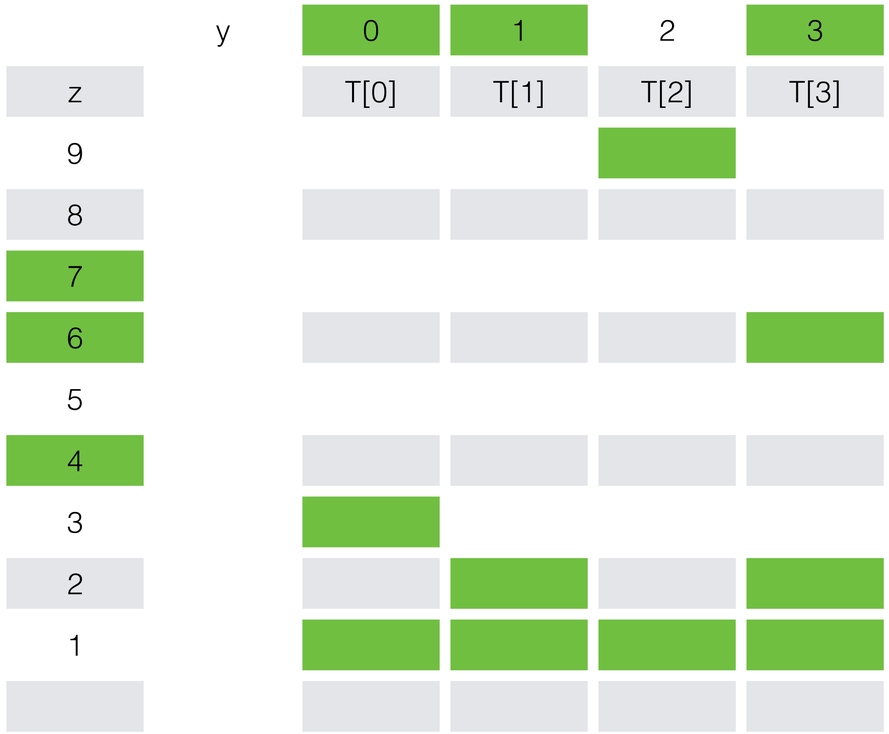
- T[y]=z

```
public Element1DVar(IntVar[] T, IntVar y, IntVar z)
```

- How to propagate efficiently?
- Two possibilities:
  - Domain consistency
  - Relaxed (aka hybrid) domain consistency
    - assume interval domains for z and all T[i], and enforce bound consistency for them
    - enforce domain consistency for y

Element1DVar: Relaxed Domain Consistency

- T[y]=z
  - T = [{1,3},[1..2],{1,9},{1,2,6}]
  - y = {0,1,3}
  - z = {4,6,7}
  - What can we remove and how?



Element1DVar: Relaxed Domain Consistency

- T[y]=z
  - T = [{1,3},[1..2],{1,9},{1,2,6}]
  - y = {0,1,3}
  - z = {4,6,7}
- Step 0: Relax T and z: interval domains
  - T = [[1..3],[1..2],[1..9],[1..6]]
  - y = {0,1,3}
  - z = [4..7]

Element1DVar: Relaxed Domain Consistency

- T[y]=z
  - T = [{1,3},[1..2],{1,9},{1,2,6}]
  - y = {0,1,3}
  - z = {4,6,7}
- Step 1: Filter (from T and z to) y
  - T = [[1..3],[1..2],[1..9],[1..6]]
  - y = {0,1,3}
  - z = [4..7]

$\forall i \in D(y)$:
$D(T[i]) \cap D(z) = \varnothing \Rightarrow D(y) \leftarrow D(y) \setminus \{i\}$

Element1DVar: Relaxed Domain Consistency

- T[y]=z
  - T = [{1,3},[1..2],{1,9},{1,2,6}]
  - y = {0,1,3}
  - z = {4,6,7}
- Step 2: Filter (from T and y to) z
  - T = [[1..3],[1..2],[1..9],[1..6]]
  - y = {3}
  - z = [4..6]

$\min(D(z)) \leftarrow \max(\min(D(z)), \min_{i \in D(y)} \min(T[i]))$

$\max(D(z)) \leftarrow \min(\max(D(z)), \max_{i \in D(y)} \max(T[i]))$

Element1DVar: Relaxed Domain Consistency

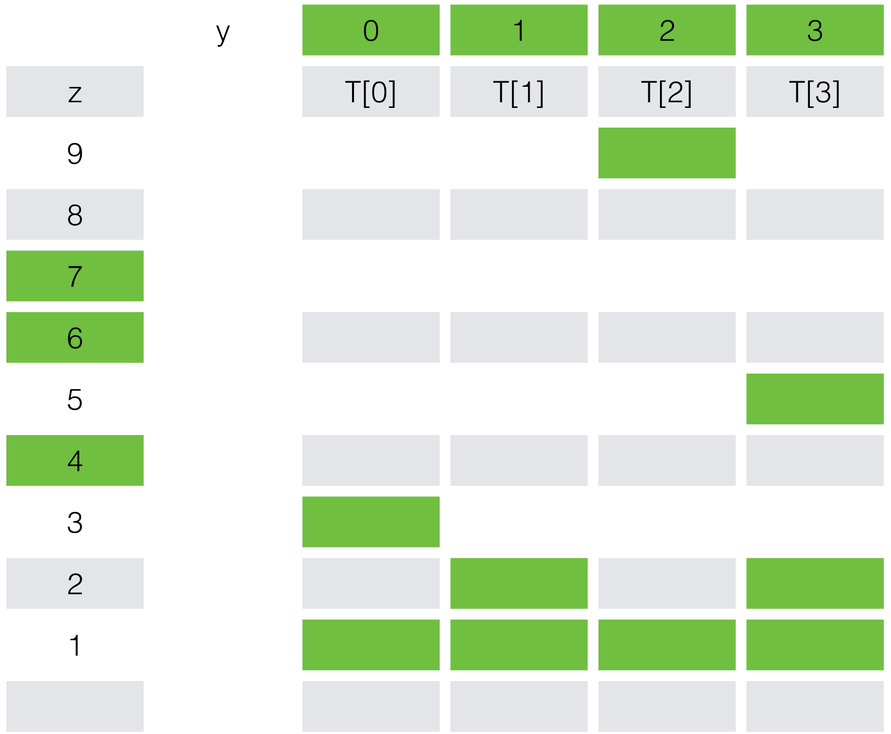
- T[y]=z
  - T = [{1,3},[1..2],{1,9},{1,2,6}]
  - y = {0,1,3}
  - z = {4,6,7}
- Step 3: Filter (from z and y to) T
  - T = [[1..3],[1..2],[1..9],[4..6]]
  - y = {3}
  - z = [4..6]

|D(y)|=1 ⇒ T[y]=z (equality constraint)

The domain of a variable T[i] can only be filtered under that condition

Element1DVar: Relaxed Domain Consistency

- T[y]=z
  - Now assume T = [{1,3},[1..2],{1,9},{1,2,5}]
  - y = {0,1,3}
  - z = {4,6,7}
  - Notice that this constraint is infeasible
  - But we do not detect it



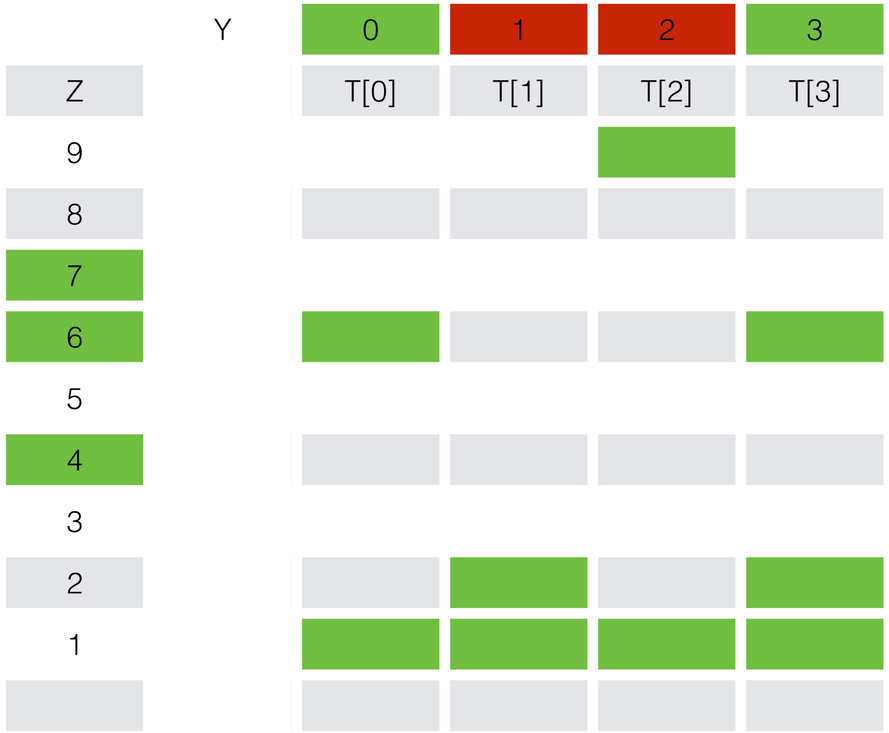
Not detected

Element Constraint

Over array of variables: domain consistency

Element1DVar: Domain Consistency

- T[y]=z
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,1,2,3}
  - z = {4,6,7}
- Step 1: Filter (from T and z) to y
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,1,2,3}
  - z = {4,6,7}



$$\forall i \in D(y): \quad D(T[i]) \cap D(z) = \varnothing \Rightarrow D(y) \leftarrow D(y) \setminus \{i\}$$

Can be quite slow to compute:
Efficient intersection with sparse-set domains?

DC Filtering: residue = support caching

- $\forall i \in D(y)$:
  $D(T[i]) \cap D(z) = \varnothing \Rightarrow D(y) \leftarrow D(y) \setminus \{i\}$
- If we find for a value $i \in D(y)$ some value v such that $v \in D(T[i])$ and $v \in D(z)$ then remember it (caching). Let us call it supportT[i].
- There is a high chance that, on the next propagation, this value is still preventing the removal $D(y) \leftarrow D(y) \setminus \{i\}$.

  But if supportT[i] $\notin D(T[i])$, then one needs to look for a new support, if not possibly perform $D(y) \leftarrow D(y) \setminus \{i\}$.
- O(1) check if the support is still valid, else O(|D(T[i])|).

Element1DVar: Domain Consistency

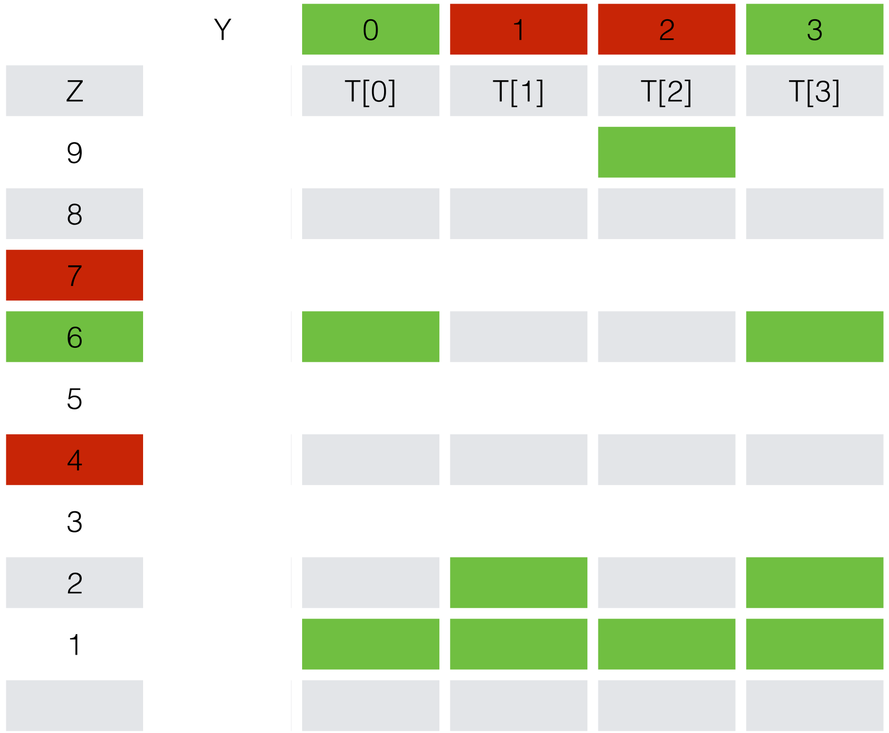
- T[y]=z
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,1,2,3}
  - z = {4,6,7}
- Step 1: Filter (from T and z) to y
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,1,2,3}
  - z = {4,6,7}

$$\forall i \in D(y): \quad D(T[i]) \cap D(z) = \varnothing \Rightarrow D(y) \leftarrow D(y) \setminus \{i\}$$

Can be quite slow to compute:
efficient intersection with sparse-set domains?

Element1DVar: Domain Consistency

- T[y]=z
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,1,2,3}
  - z = {4,6,7}
- Step 2: Filter (from T and y) to z
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,3}
  - z = {4,6,7}



$$\forall v \in D(z): \nexists i \in D(y): v \in D(T[i]) \Rightarrow D(z) \leftarrow D(z) \setminus \{v\}$$

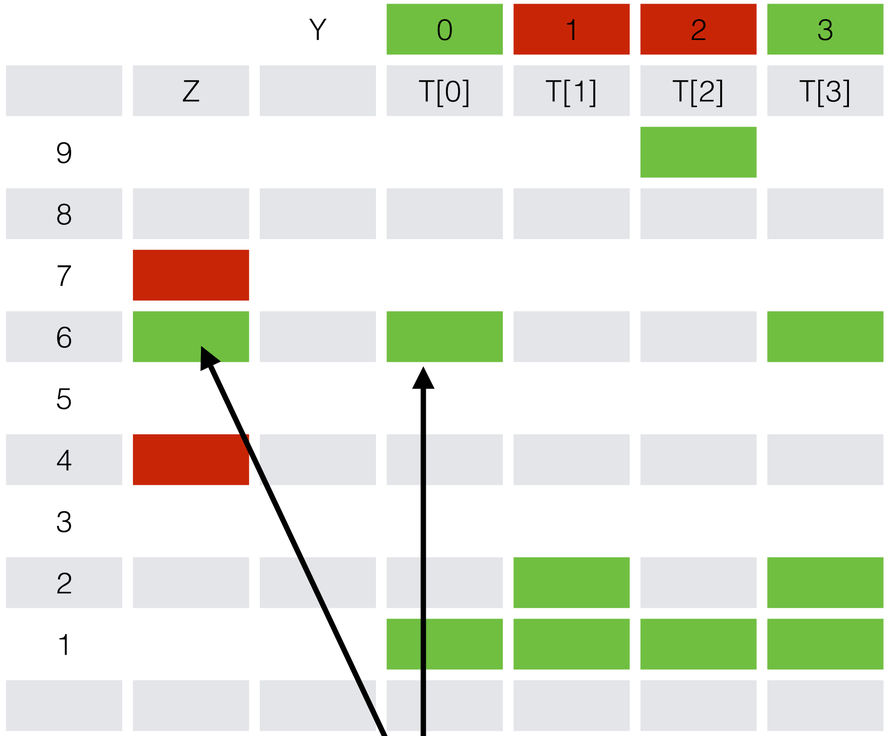
Can be quite slow to compute:
need to scan possibly the whole domain of y for each value v

DC Filtering: Support caching

- $\forall v \in D(z)$:
  $\nexists i \in D(y): v \in D(T[i]) \Rightarrow D(z) \leftarrow D(z) \setminus \{v\}$
- Again, the same *caching* idea: assume v cannot be removed.
  Then we can store an index i such that $v \in D(T[i])$: call it support_z[v].
- There is a high chance that, on the next propagate, $v \in D(T[support\_z[v]])$, and in this case we cannot remove v from D(z).
- Otherwise, look for a new support.
- O(1) check if the support is still valid, else O(|D(y)|).

Element1DVar: Domain Consistency

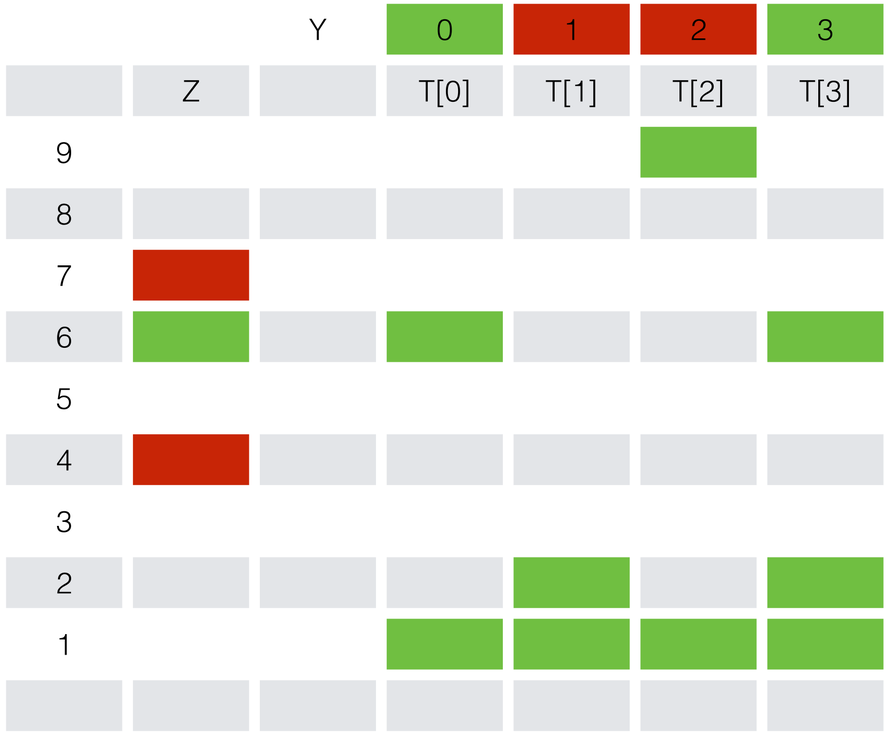
- T[y]=z
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,1,2,3}
  - z = {4,6,7}
- Step 2: Filter (from T and y) to z
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,3}
  - z = {4,6,7}



$\forall v \in D(z)$:
$\nexists i \in D(y): v \in D(T[i]) \Rightarrow D(z) \leftarrow D(z) \setminus \{v\}$

Element1DVar: Domain Consistency

- T[y]=z
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,1,2,3}
  - z = {4,6,7}
- Step 3: Filter (from z and y) to T
  - T = [{1,6},[1..2],{1,9},{1,2,6}]
  - y = {0,3}
  - z = {6}

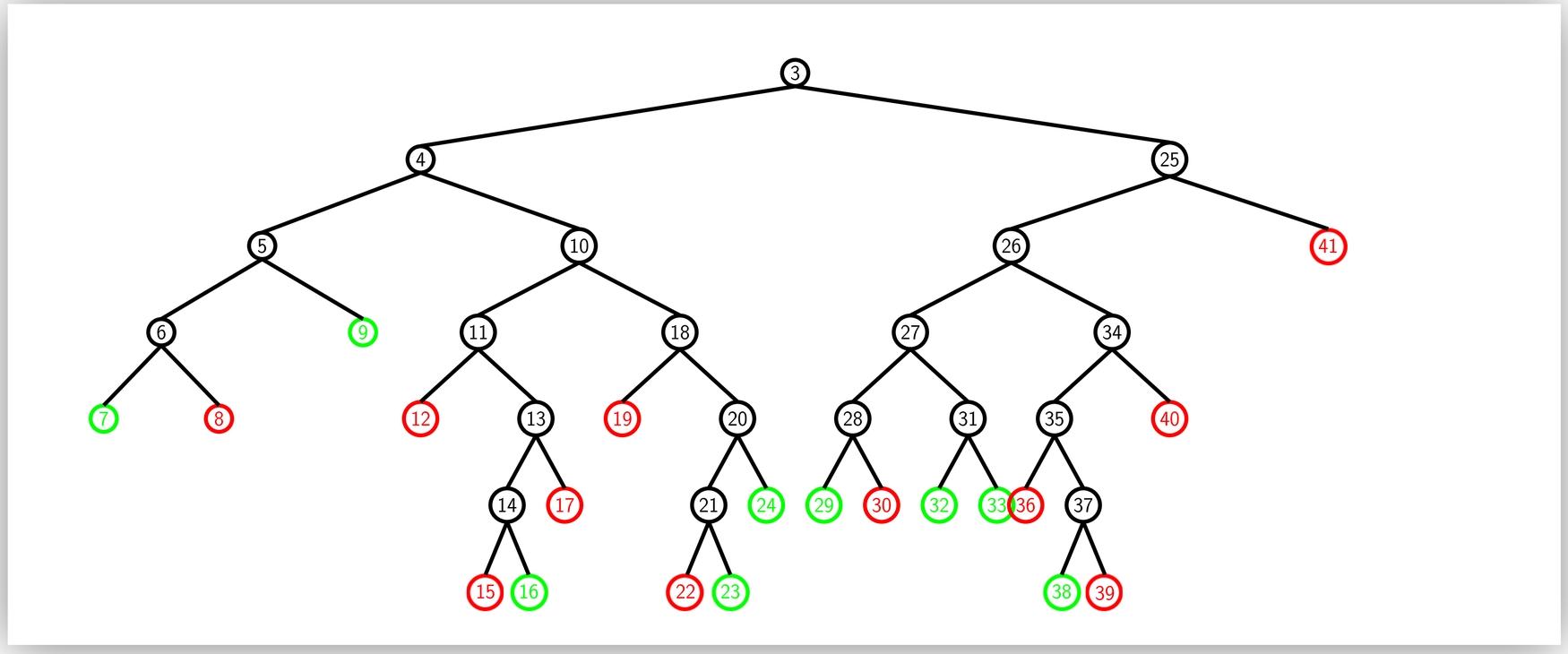


|D(y)|=1 ⇒ T[y]=z (equality constraint)

The domain of a variable T[i] can only be filtered under that condition

Remark about Caching

- Those cached supports remain valid on backtrack (no need for reversibles) because a support for the constraint in a node of the search tree is also a support for the ancestors of that node: do you see why?



Logical Constraints

and reified constraints

IsLessOrEqual

```
public class IsLessOrEqual extends AbstractConstraint { // b <=> x <= v

    private final BoolVar b;
    private final IntVar x;
    private final int v;

    @Override
    public void post() {
        if (b.isTrue()) {
            x.removeAbove(v);
        } else if (b.isFalse()) {
            x.removeBelow(v + 1);
        } else if (x.max() <= v) {
            b.fix(1);
        } else if (x.min() > v) {
            b.fix(0);
        } else {
            b.whenFixed(() -> {
                // should deactivate the constraint as it is entailed
                if (b.isTrue()) {
                    x.removeAbove(v);

                } else {
                    x.removeBelow(v + 1);
                }
            });
            x.whenBoundChange(() -> {
                if (x.max() <= v) {
                    // should deactivate the constraint as it is entailed
                    b.fix(1);
                } else if (x.min() > v) {
                    // should deactivate the constraint as it is entailed
                    b.fix(0);
                }
            });
        }
    }
}
```

```
enum Students = {George,Halle,Keira,Clive};
enum Companies = {Google,IBM,NASA,SAP};
int rankCompanies[Students,Companies];
int rankStudents[Companies,Students];
...
var{Companies} company[Students];
var{Students} student[Companies];
solve {
   forall(s in Students)
      student[company[s]] = s;
   forall(c in Companies)
      company[student[c]] = c;

   forall(s in Students, c in Companies)
      rankCompanies[s,c] < rankCompanies[s,company[s]]
      => rankStudents[c,student[c]] < rankStudents[c,s];
   forall(c in Companies, s in Students)
      rankStudents[c,s] < rankStudents[c,student[c]]
      => rankCompanies[s,company[s]] < rankStudents[s,c];
}
```

Logical Constraints

- How to implement cp.post(x > y ⇒ w < z) ?

- Easy: (x > y) can be reified b1: BoolVar ≡ x > y

  Same for: (w < z) can be reified b2: BoolVar ≡ w < z

  b1 ⇒ b2 ≡ (!b1 or b2) ≡ (1–b1)+b2 ≥ 1 (views, sum, etc) ≡ b1 ≤ b2

- In order to make it even more general, we have

```
public void post(BoolVar b) throws InconsistencyException {
    b.fix(true);
    fixPoint();
}
```

- This way you can post arbitrary logical expressions in MiniCP