(Finite Domain)
Constraint Programming

Guido Tack

From SAT to CP

What you saw yesterday:

SAT = Boolean **variables**
+ clause **constraints**

MaxSAT = SAT **+ objective function**
(satisfy as many clauses as possible)

Solved using

unit propagation
DPLL (**branch and search**)
CDCL (**learn from failures**)

From SAT to CP

What we will look at today:

CP = all kinds of **variables** (bool, int, float, set)
+ all kinds of **constraints**

+ objective function
(arbitrary function to minimise/maximise)

Solved using

constraint propagation
branch and search
LCG (**learn from failures**)



← backtrack to previous state

Inconsistency found!

Solution found!

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

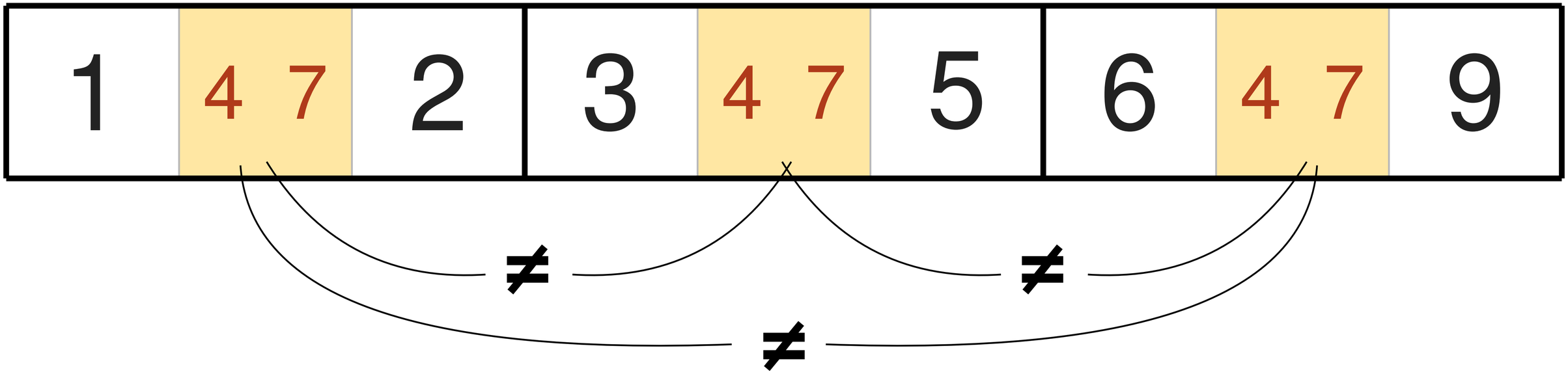
Constraint Propagation

Basic principle:

- Each search node stores **variable domains**
  (set of values still possible for each variable)
- Consider **each constraint in isolation**
- If certain values are **inconsistent, remove** them from the domain
- Only search on values that have not been found inconsistent

Constraint Propagation

Let’s look at one row of a Sudoku:



Does this have a solution?

Is each **individual** constraint still satisfiable?

Integer Constraint Propagation

- Same idea as SAT unit propagation:
  remove inconsistent values from domain
- Examples:

  - $x \in 1..4$, $y \in 1..4$, constraint $x < y$ $\Rightarrow$

  - $x,y \in 3..10$, $z \in 5..10$, constraint $x+y=z$ $\Rightarrow$

  - $a,b,c \in 3..5$, $d,e \in 2..6$, $f \in 4..7$, $g \in 2..8$, constraint alldifferent([a,b,c,d,e,f,g]) $\Rightarrow$

Example: *n queens* propagation

Integer Constraint Propagation

- Dedicated **propagation algorithm** per constraint
- Usually: polynomial complexity, often at most $O(n^2)$
- Range from very simple (x<y) to complex algorithms based on flows (*alldifferent*, *global cardinality*), automata (*regular*), or temporal reasoning (*cumulative*)

Some notation

- A **domain** *D* maps variables to sets of values, e.g.
  D(x) = {1,3,4,5}
- An **assignment** *a* maps variables to values, e.g.
  a(x) = 3
- We will talk about the **lower and upper bounds** of *x* in *D:*
  min(D,x)=min({i | i in D(x)})
  max(D,x)=max({i | i in D(x)})
- We use update operations to modify a domain:
  D(x) := {1,3,5} *(set domain of x to {1,3,5})*
  D(x) := D(x) \ {3} *(remove 3 from domain)*
  D(x) :≤ 4 *(remove all values > 4)*

More notation

A **propagator** is a function that takes a domain as input and returns an updated domain

- we will use := to simplify notation
- returning an empty domain signals failure (*domain wipeout*)

Propagators must be

- **correct** (only remove values that are inconsistent)
- **checking** (signal failure when only one value per variable is left)
- **monotonic** (do at least the same amount of pruning on smaller domains)

Propagators can be

- **idempotent** (p(p(D)) = p(D))

Activity: propagate manually

- Form **groups of four**
- Each one of you **plays a propagator**
- Only apply the rules on **your card**
- Write one new row on the sheet with **only the domains changed by your rules**
- Pass the sheet **to the person to your left**
- Stop when **nothing is changing after everyone has propagated**
- **Ask for help if you're unsure!**

Activity: propagate manually

- Debrief

Propagation rules for x<y

Simple rule-based propagation:

D(x) :< max(D,y)
D(y) :> min(D,x)

- **Q**: is this propagator *idempotent*?
  (i.e., after applying both rules, can we prove that they never "fire" immediately again?)

- **A**: yes, because we only read the max (min) of y (x) but update the min (max)

Propagation rules for x≠y

Again simple rules:

if D(x) = {d} then D(y) := D(y) \ {d}
if D(y) = {d} then D(x) := D(x) \ {d}

- **Q**: is this propagator idempotent?
- **A**: yes

Propagation rules for x≤y+z

- Isolate each variable in turn:
  - D(x) :≤ max(D,y)+max(D,z)

    D(y) :≥ min(D,x)-max(D,z)

    D(z) :≥ min(D,x)-max(D,y)
- Observation 1: these rules are **complete**, they remove all possible inconsistent values
- Observation 2: all complete propagators are idempotent

Propagation rules for x=y+z

- Isolate each variable in turn:
  - $D(x) :\leq \max(D,y)+\max(D,z)$
    $D(x) :\geq \min(D,y)+\min(D,z)$
    $D(y) :\geq \min(D,x)-\max(D,z)$
    $D(y) :\leq \max(D,x)-\min(D,z)$
    $D(z) :\geq \min(D,x)-\max(D,y)$
    $D(z) :\leq \max(D,x)-\min(D,y)$
- **Q**: are these rules complete?
- **A**: no, counterexample: $x,y\in\{1,3\}$, $z\in 1..8$
- This is an example of **bounds propagation**

Bounds vs Domain

- Strongest possible propagation:
  a propagator is **complete** for a constraint *c* if after propagation, no further value could be removed from any variable domain because of constraint *c*
- Complete propagators are also called **domain consistent**
- Often algorithms for complete propagators are computationally expensive or even NP hard!

  - example: `y = sum (i in 1..n) (c[i] * x[i])`
- **Bounds propagation** ignores "inner" domain values and only looks at min/max

Bounds Propagation

- Let's design propagation rules for $x = a \times y + b \times z$

  - $D(x) :\leq a \times \max(D,y) + b \times \max(D,z)$
    $D(x) :\geq a \times \min(D,y) + b \times \min(D,z)$
    $D(y) :\geq (\min(D,x) - b \times \max(D,z)) / a$
    $D(y) :\leq (\max(D,x) - b \times \min(D,z)) / a$
    $D(z) :\geq (\min(D,x) - a \times \max(D,y)) / b$
    $D(z) :\leq (\max(D,x) - a \times \min(D,y)) / b$

  $y = (x - b \times z)/a$

  $z = (x - a \times y)/b$

- **Q**: are these rules **bounds complete**? (i.e. do they prune the bounds as much as possible)
- **A**: no, counterexample $x=3y+5z$, $x \in 2..7$, $y \in 0..2$, $z \in 0..1$
  - reason: the rules above compute with *real numbers*

Global Constraints

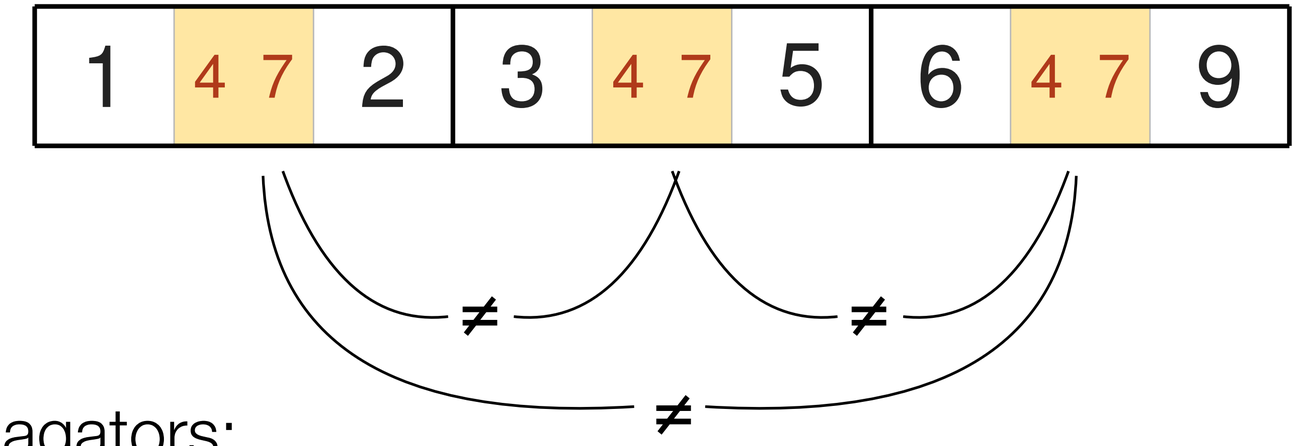
- We have seen a number of global constraints in MiniZinc: alldifferent, inverse, regular, table, …
- In constraint solvers (e.g Chuffed, Gecode, CP-SAT) many of these have **dedicated algorithms**
- Alternatively, they are **decomposed**
- Dedicated algorithms can
  - provide stronger propagation
  - have lower complexity than decomposition

Propagating alldifferent

- Decomposition: *O(n^2)* binary disequality constraints:

```
forall (i,j in 1..n where i<j) (x[i] != x[j])
```

- Not very strong, remember:



- Let's look at two dedicated propagators:
  - a **weak** propagator, but with **linear** runtime
  - a **strong** propagator with **quadratic** runtime

Weak but fast alldifferent

- Can be made linear time
- Propagates just slightly more than decomposition (can you see where?)

```
p(D) =
   let W = {x1,…,xn}

   while ∃ x∈W where D(x) = {d}
      W := W \ {x}
      for each x' ∈ W
         D(x') := D(x') \ {d}
   DV := union {D(x) | x ∈ W}
   if |DV| < |W| then
      return false
   else
      return D
```

Activity: Stronger propagation

Slower but strong alldifferent

- Domain propagation requires finding **hall sets**:
  - Set of variables W such that |W| = |union {D(x) | x∈W}|
  - Example: x,y,z ∈ {3,4,5}, a,b ∈ 2..6
  - x,y,z are a hall set, there are only 3 values for 3 variables so no other variable can take any of 3,4,5
  - consequently, a,b ∈ {2,6} is now also a hall set
- Propagation algorithm: **find all hall sets** and remove their values from all other variables

Finding all hall sets

Naive approach:

check, for each subset W of the variables X whether |W| = |union {D(x) | x∈W}|

Q: What is the problem with this approach?

A: There are **exponentially** many subsets to check.

Finding all hall sets

- Insight: can be computed **efficiently** in $O(n^{2.5})$
- Breakthrough paper:
  Régin, A filtering algorithm for constraints of difference in CSPs, AAAI 1994
- Uses **graph algorithms**
  - insight into problem structure
  - relation between solutions of the constraint and properties of the graph

Running Example

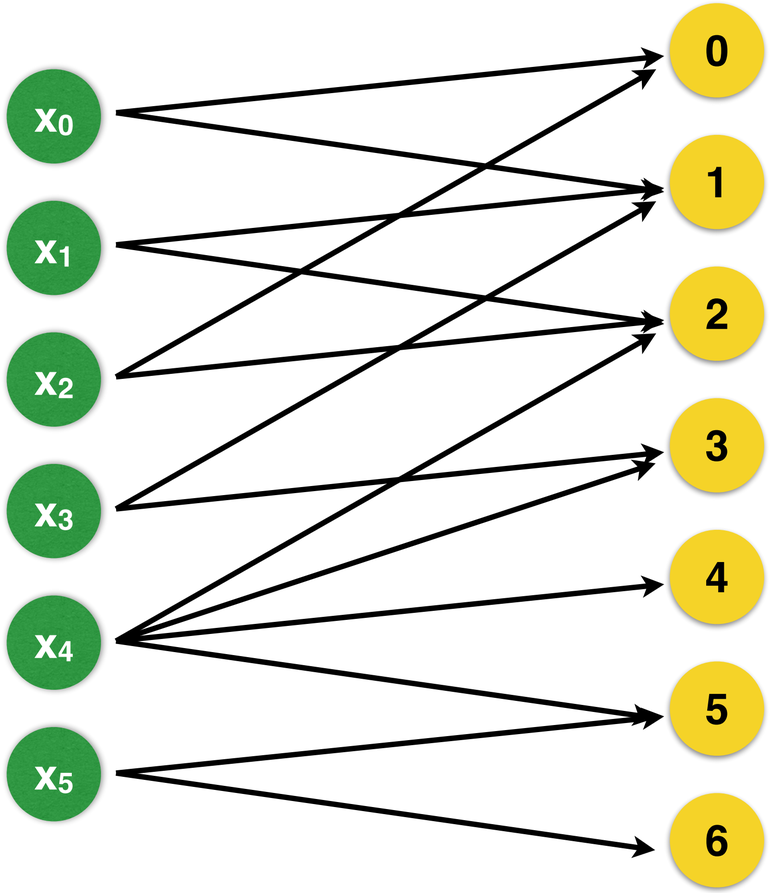
- Example:
  $x_0 \in \{0,1\}$ $x_1 \in \{1,2\}$ $x_2 \in \{0,2\}$
  $x_3 \in \{1,3\}$ $x_4 \in \{2,3,4,5\}$ $x_5 \in \{5,6\}$

- After propagation:
  $x_0 \in \{0,1\}$ $x_1 \in \{1,2\}$ $x_2 \in \{0,2\}$
  $x_3 \in \{3\}$ $x_4 \in \{4,5\}$ $x_5 \in \{5,6\}$

Approach

- Construct a **variable-value graph**
  - bipartite graph
  - variable nodes → value nodes
- Characterise solutions in the graph
  - maximal matchings
- Use matching theory
  - one matching can describe all matchings
- Remove edges not representing solutions

Variable-Value Graph



Matching

- A **matching** is a subset of edges in a graph such that **no two edges share a common vertex**
- A matching is **maximal** if the cardinality is maximal
- **Q:** When is a matching maximal in a bipartite graph?
- **A**: In a bipartite graph, a matching is maximal if it maximally covers one of the two sets of nodes
  - in our case: the set of variable nodes needs to be **completely** covered, since we need at least one value for each variable!

Matchings are Solutions

- An assignment *a* is a solution to the alldifferent constraint

  if and only if

  $x_i \rightarrow a(x_i)$

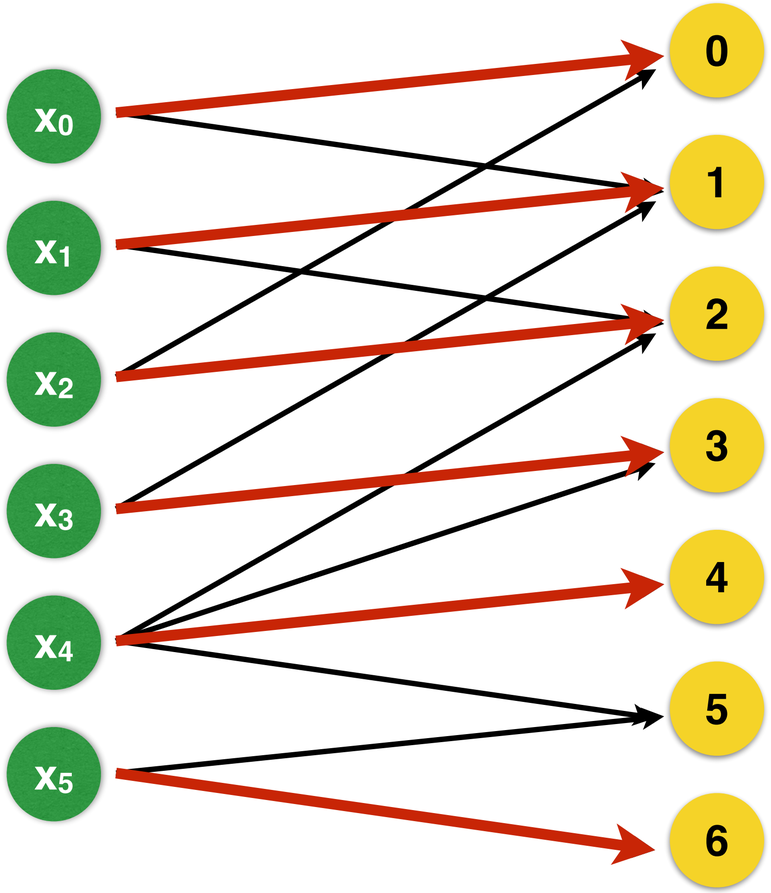
  is a maximal matching

  - No value has two incoming edges
    (no two variables have the same value)

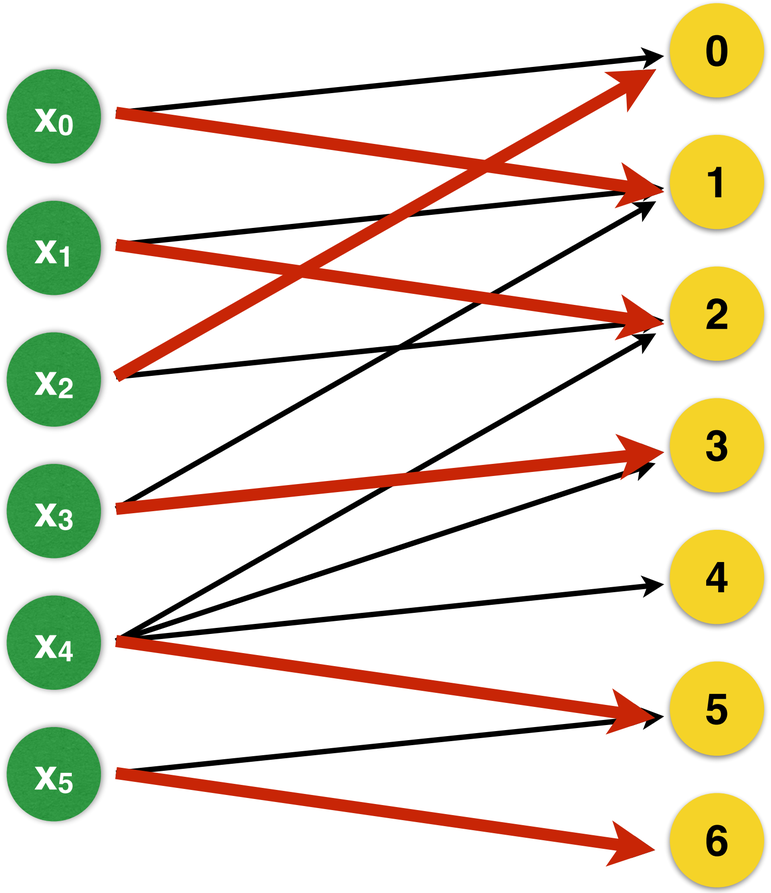
  - All variable nodes are covered
    (we have a value for each variable)

- If an alldifferent constraint has a solution at all, then **the solutions correspond to the set of maximal matchings**

Maximal Matching A



Maximal Matching B



First Idea

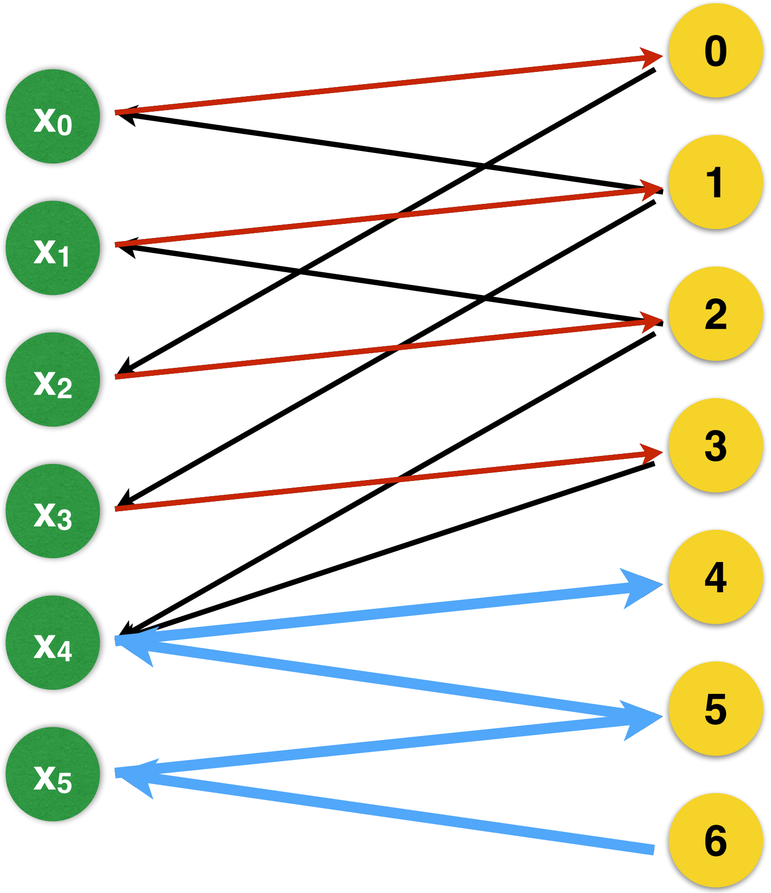
- Compute all maximal matchings
  - mark each edge if it is part of any matching
- If no maximal matching exists: failure
- Remove unmarked edges
  - edge occurs in no maximal matching
- Compact representation of all solutions of alldifferent
- But still requires enumerating all maximal matchings = all solutions to the constraint

Computing all Matchings

- Compute one initial maximal matching
- From matching theory:
  - Edge *e* belongs to some maximal matching
    if and only if
    either: *e* belongs to an even alternating path starting at a free node
    or: *e* belongs to an even alternating cycle
- Implementation: orient edges
  - from variable to value if in initial matching
  - from value to variable if free (not in initial matching)

Oriented Graph

- Alternating path
- Only free node: 6
- Mark all edges on simple path (**alternation built-in**)
- Intuition: these edges can be **permuted** and we would still get a matching

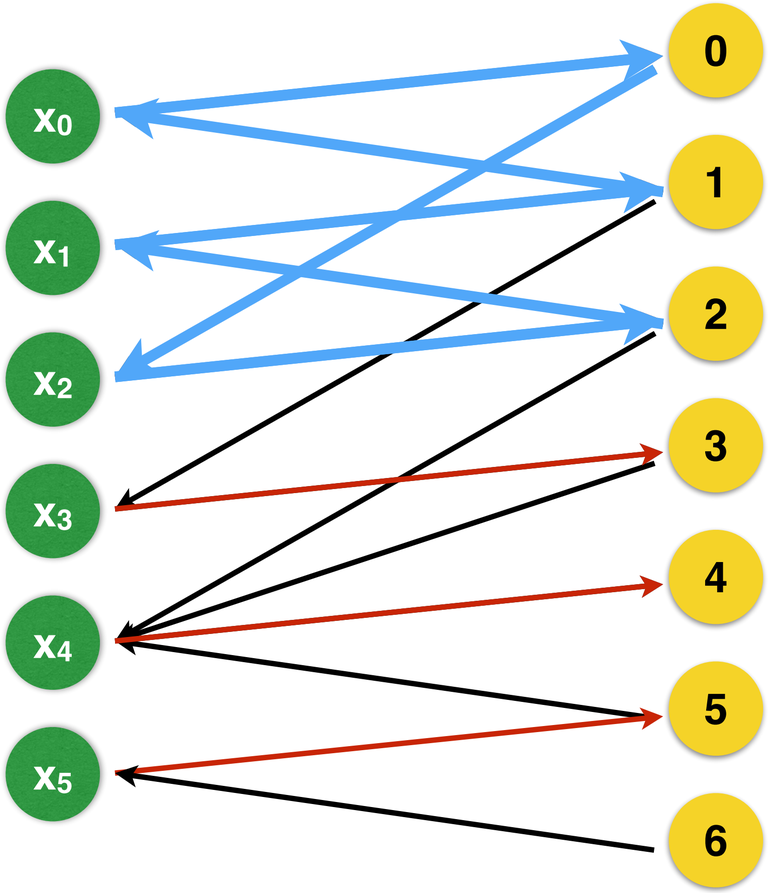


Alternating Cycles

- Compute strongly connected components (**SCCs**)
  - two nodes are strongly connected if there is a path from *a* to *b* and from *b* to *a*
  - SCC: any pair of nodes is strongly connected
- Mark all edges joining two nodes in the same SCC

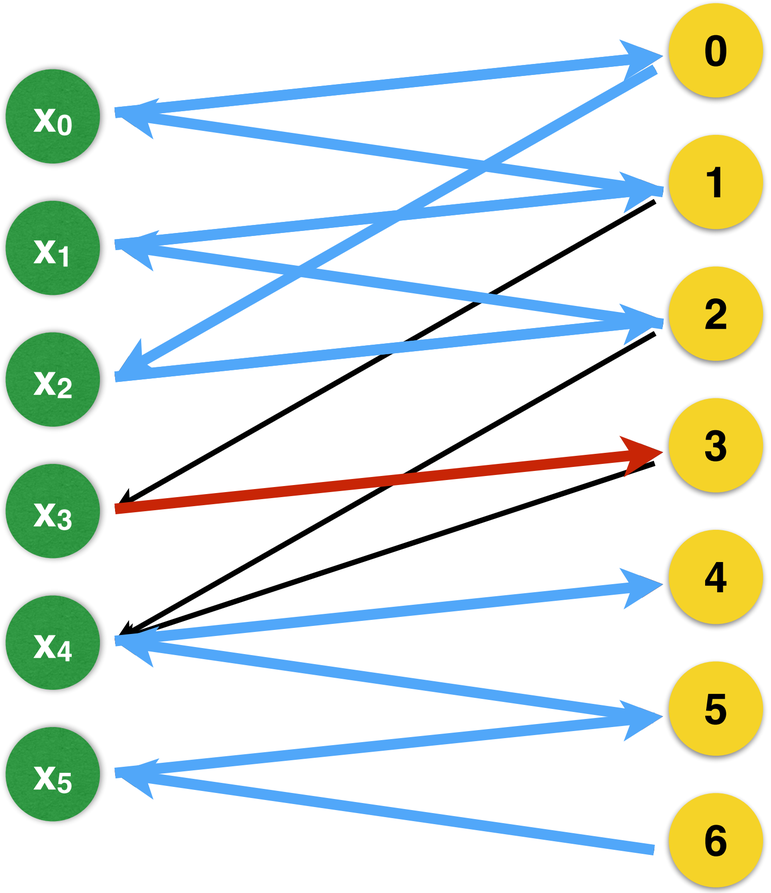
Alternating Cycles

- Nodes in SCC:
  x_0, x_1, x_2
  0, 1, 2
- Mark joining edges
- Intuition: variables take all values from SCC (we've discovered a Hall set!)



All Marked Edges

- Marked + matched edges (blue + red) represent solutions
- Remove all non-solution edges:
  $1 \rightarrow x_3$
  $2 \rightarrow x_4$
  $3 \rightarrow x_4$
- Extract new domains from graph:
  $x_0 \in \{0,1\}$, $x_1 \in \{1,2\}$, $x_2 \in \{0,2\}$
  $x_3 \in \{3\}$, $x_4 \in \{4,5\}$, $x_5 \in \{5,6\}$



Activity: manual all-different

- Work in **groups of two**
- Follow the steps of the all-different algorithm
- **Ask for help if you're unsure!**

Complexity

- Linear in number of edges for
  - construction
  - marking, SCCs
- Matching: many algorithms
  - $O(n^{2.5})$ where n is the maximum number of variables and values
- Can be improved:
  - keep variable-value graph between invocations, just update to new domains

Bounds vs domain

- We call this the **propagation strength** of a propagator
- In MiniZinc: control using `::bounds` and `::domain` *annotations*
- Stronger propagation means
  - smaller search tree
  - more time spent per node
- Sometimes domain propagation is absolutely vital to solving a problem, sometimes it is far too expensive
  - trade off between propagation and search
  - homework: try Sudoku!
  - do experiments!

Experimenting in MiniZinc

- Method 1:
  run the solver in *statistics mode* (tick the *Solving Statistics* box)
- Method 2:
  use Gecode's Gist tool for visualising the search tree

Fixed points

- Putting it all together:
  - one or more propagators per (high-level) constraint
  - at each search node, run all of them until none can *contribute* any more: this is a **mutual fixed point** of the propagators
  - we can prove: because propagators are monotonic, the **order of propagation does not matter**, we always get the same fixed point!
  - But: order may matter for efficiency

Efficient Fixed Points

Two typical techniques for making fixed point computation efficient:

- keep a **queue** of propagators that may still be able to contribute
  - initially all propagators
  - remove after propagation if no variable was changed
  - advanced: priority queue (cheap propagators first)
- only run propagators **whose variables have changed**
  - don't run x<y if only z has changed
  - don't run x<y if only the lower bound of y has changed!
  - this is called *event-based propagation*

More propagators

- CP solvers typically implement many different propagation algorithms.
- Let’s go through a few interesting ones (only at a high level).

Base constraints

- Linear constraints
- Arithmetic +/×/÷/mod/pow, abs, min/max
- Boolean constraints (and, or, xor, clause)

Scheduling

- Constraints: disjunctive, cumulative
- Algorithmic ideas:

  disjunctive: For every set of tasks on the machine, compare the total processing time with the span from its earliest start to its latest end; if it doesn't fit, deduce that some task must come before or after the whole set (edge finding, all sets in O(n log n) via a Θ-tree).

  cumulative: Build the compulsory-part profile (the time each task must be running whatever its start) and push tasks out of over-capacity regions; strengthen with energetic reasoning — compare the energy available in a window, capacity × length, with the energy the tasks must spend inside it.

Counting

- Global cardinality constraint
- Algorithmic idea: basically an extension of all-different. Each value gets a capacity interval, so domain consistency is a feasible flow in the value graph, and any arc in no feasible flow is pruned (residual-graph SCCs, exactly as in Régin's alldifferent).
- Note that alldifferent(x) is the same global_cardinality(x, [0,…,0], [1,…,1]): each value may appear between 0 and 1 times!

Table constraints

- State that an array of variables *x* must take the values of some tuple *t* from a list of given tuples
- Can express arbitrary constraints (although the list of tuples can get exponentially long)
- Algorithmic idea: Keep, for every value, whether any allowed tuple still supports it; the modern version stores the tuple set as a bitset and updates it by word-parallel AND (compact-table), which is why table constraints are fast in practice.

Grammar constraints

- E.g. regular(x, r) states that the x variables must be a word in the regular language r
- Algorithmic idea: Unfold the automaton into a layered graph, one layer per variable, and keep only the arcs that lie on some accepting path — a reachability sweep forwards and backwards.

Element constraint

- The simple constraint y = a[x], where x, y and a can all be variables
- Algorithmic idea: drop any x whose entry is no longer in D(y), and any y value no surviving index produces. Plus extensions when the a are variables.

Packing constraints

- Geometric packing (diffn)
- E.g., packing rectangles into a limited area
- Algorithmic ideas:
  - has some similarities with scheduling
  - sweep the placement space for one rectangle at a time: other rectangles' compulsory parts define forbidden regions, and the sweep finds the first feasible position, moving the origin's bounds inwards.
- More complex constraints exist (for more complex shapes)

Graph constraints

- E.g. circuit(x) states that the x variables must form a Hamiltonian circuit
- Algorithmic idea: forbid the arc that would close a subtour before all nodes are in, and use structural relaxations (connectivity, mandatory bridges/articulation points, 1-tree bounds) because checking Hamiltonicity itself is NP-hard.

Summary

- Propagation:
  Prune values from domains that are inconsistent with a constraint
- Dedicated propagation algorithms for each constraint
  - simple constraints: rule-based
  - alldifferent: find all hall sets
- Propagation strength: how aggressively a propagator removes inconsistent values
  - strong propagation: small trees but more time spent per node
- Constraint solvers are based on a fixed point loop
  - fixed point is independent of propagation order
  - efficient fixed points require events and priorities
- Branch-and-bound:
  - post constraints after each solution to find better solutions