Weakly Monotonic Propagators

Christian Schulte
KTH - Royal Institute of Technology, Sweden

Guido Tack
PS Lab, Saarland University, Germany

Int'l Conference on Principles and Practice of Constraint Programming

September 20th, 2009, Lisbon, Portugal

Reality check

Reality check

Beautiful model:

Propagators are monotonic!

closure operators, unique fixpoints, predictable behavior,
more constraints means less search

[Saraswat et al, Benhamou et al, Van Hentenryck et al, Apt, ...]

Reality check

Beautiful model:

Propagators are monotonic!

closure operators, unique fixpoints, predictable behavior, more constraints means less search

[Saraswat et al, Benhamou et al, Van Hentenryck et al, Apt, ...]

Ugly reality:

Propagators are not monotonic!

approximate NP-hard constraints, use heuristics or randomization

[Baptiste et al, Menana/Demassey, Katriel, Stergiou, Mehta/Van Dongen...]

Reality check

Beautiful model:

Propagators are monotonic!

closure operators, unique fixpoints, predictable behavior, more constraints means less search

[Saraswat et al, Benhamou et al, Van Hentenryck et al, Apt, ...]

Ugly reality:

Propagators are not monotonic!

approximate NP-hard constraints, use heuristics or randomization

[Baptiste et al, Menana/Demassey, Katriel, Stergiou, Mehta/Van Dongen...]

Reality check

Propagators are monotonic! **Propagators are not monotonic!**

- Realistic formal model
  - establish properties of propagators
  - minimally restrictive (maximally realistic)
- Consequences
  - what does propagation compute?
  - does recomputation work?

Reality check

Propagators are weakly monotonic!

- Realistic formal model
  - establish properties of propagators
  - minimally restrictive (maximally realistic)
- Consequences
  - what does propagation compute?
  - does recomputation work?

A Model

Propagation

Variables $x, y, z \in X$

Values V

Domains $d(x) \subseteq V$

Propagation

Variables $x, y, z \in X$

Domains $d(x) \subseteq V$

Values V

all finite

Propagation

Variables $x, y, z \in X$

Domains $d(x) \subseteq V$

Values V

all finite

Propagators

$p(d)(x) \subseteq d(x)$

(contracting)

Propagation

Variables $x, y, z \in X$

Domains $d(x) \subseteq V$

Values V

all finite

Propagators

$p(d) \quad \subseteq d$

(contracting)

Propagation

Variables $x, y, z \in X$

Domains $d(x) \subseteq V$

Values V

all finite

Propagators

$p(d) \subseteq d$ (contracting)

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

Propagation

Variables $x, y, z \in X$

Domains $d(x) \subseteq V$

Values V

all finite

Propagators

$p(d) \subseteq d$ (contracting)

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

$p(p(d)) = p(d)$ (idempotent)

Propagation

Variables $x, y, z \in X$

Domains $d(x) \subseteq V$

Values V

all finite

Propagators

$p(d) \subseteq d$ (contracting)

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

$p(p(d)) = p(d)$ (idempotent)

closure operators

Propagation

Variables $x, y, z \in X$

Domains $d(x) \subseteq V$

Values V

all finite

Propagators

$p(d) \subseteq d$ (contracting)

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

$p(p(d)) = p(d)$ (idempotent)

closure operators

Knaster-Tarski ⇒ weakest mutual fixpoint exists

Fixpoints

Fixpoints

- given P, d: unique weakest mutual fixpoint of all $p \in P$ stronger than d exists

Fixpoints

- given P, d: unique weakest mutual fixpoint of all p∈P stronger than d exists
- it can be computed in a finite number of steps:

```
while (∃ p∈P such that p(d)≠d)
    d ← p(d)
```

Fixpoints

- given P, d: unique weakest mutual fixpoint of all p∈P stronger than d exists
- it can be computed in a finite number of steps:

```
while (∃ p∈P such that p(d)≠d)
    d ← p(d)
```

- it does not depend on order of propagation

Fixpoints

- given P, d: unique weakest mutual fixpoint of all p∈P stronger than d exists
- it can be computed in a finite number of steps:

```
while (∃ p∈P such that p(d)≠d)
    d ← p(d)
```

- it does not depend on order of propagation
- **idempotency is irrelevant**

Fixpoints

- given P, d: unique weakest mutual fixpoint of all p∈P stronger than d exists
- it can be computed in a finite number of steps:

```
while (∃ p∈P such that p(d)≠d)
    d ← p(d)
```

- it does not depend on order of propagation
- **idempotency is irrelevant**
- **monotonicity is crucial**

Propagation

Variables $x, y, z \in X$ Values V

Domains $d(x) \subseteq V$

Propagators

$p(d)(x) \subseteq d(x)$ (contracting)

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

$p(p(d)) = p(d)$ (idempotent)

closure operators

Knaster-Tarski $\Rightarrow$ weakst mutual fixpoint exists

Propagation

Variables $x, y, z \in X$ Values V

Domains $d(x) \subseteq V$

Propagators

$p(d)(x) \subseteq d(x)$ (contracting) ✔

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

$p(p(d)) = p(d)$ (idempotent)

closure operators

Knaster-Tarski ⇒ weakst mutual fixpoint exists

Propagation

Variables $x, y, z \in X$ Values V

Domains $d(x) \subseteq V$

Propagators

$p(d)(x) \subseteq d(x)$ (contracting) ✔

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

~~$p(p(d)) = p(d)$~~ (idempotent)

closure operators

Knaster-Tarski ⇒ weakst mutual fixpoint exists

Propagation

Variables $x, y, z \in X$

Values V

Domains $d(x) \subseteq V$

Propagators

$p(d)(x) \subseteq d(x)$ (contracting) ✔

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

$p(p(d)) = p(d)$ (idempotent)

?

closure operators

Knaster-Tarski ⇒ weakst mutual fixpoint exists

Non-monotonic propagators

- Some constraints are NP-hard to propagate
  - Hamiltonian circuit, knapsack, scheduling
- Approximative propagation algorithms are often non-monotonic, e.g.
  - consider some instead of all subsets
  - consider only small domains
  - randomize
  - start graph algorithm at one instead of all nodes

Example: Hamiltonian circuit

`circuit(`x_1`,...,`x_n`)`

graph with edges `i→`x_i

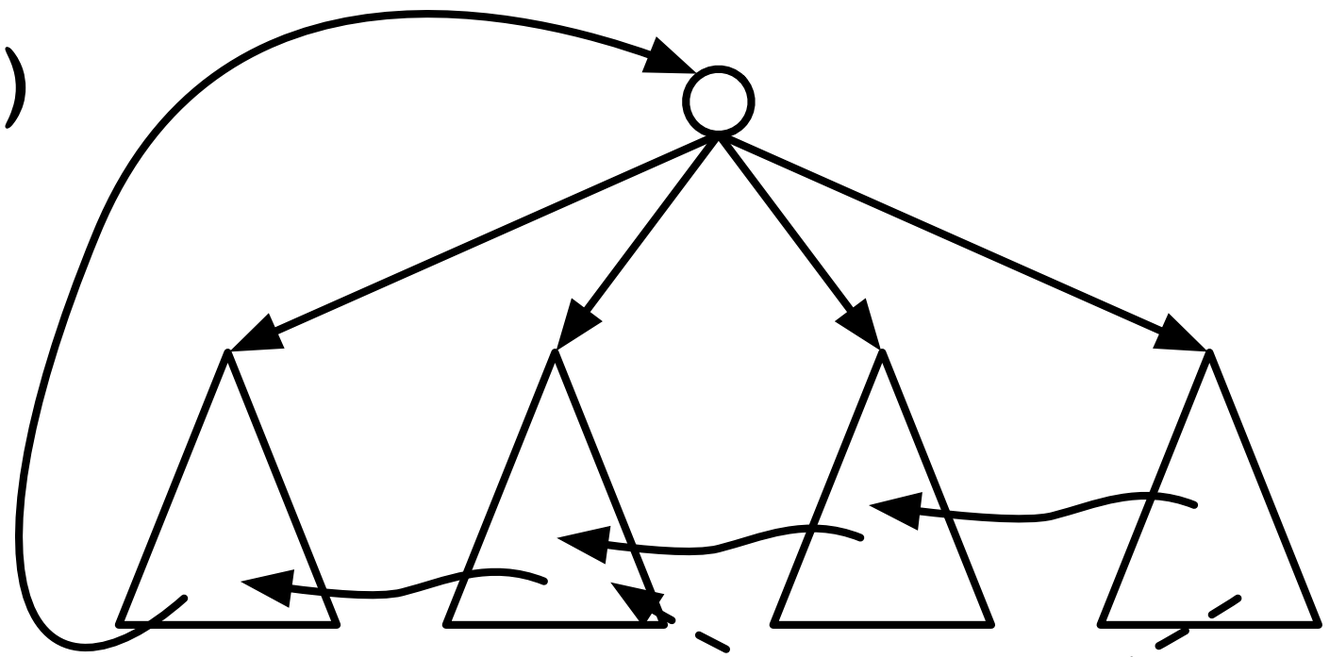
has single cycle

covering all nodes

Example: Hamiltonian circuit

`circuit(`x_1`,...,`x_n`)`

graph with edges `i`→x_i
has single cycle
covering all nodes

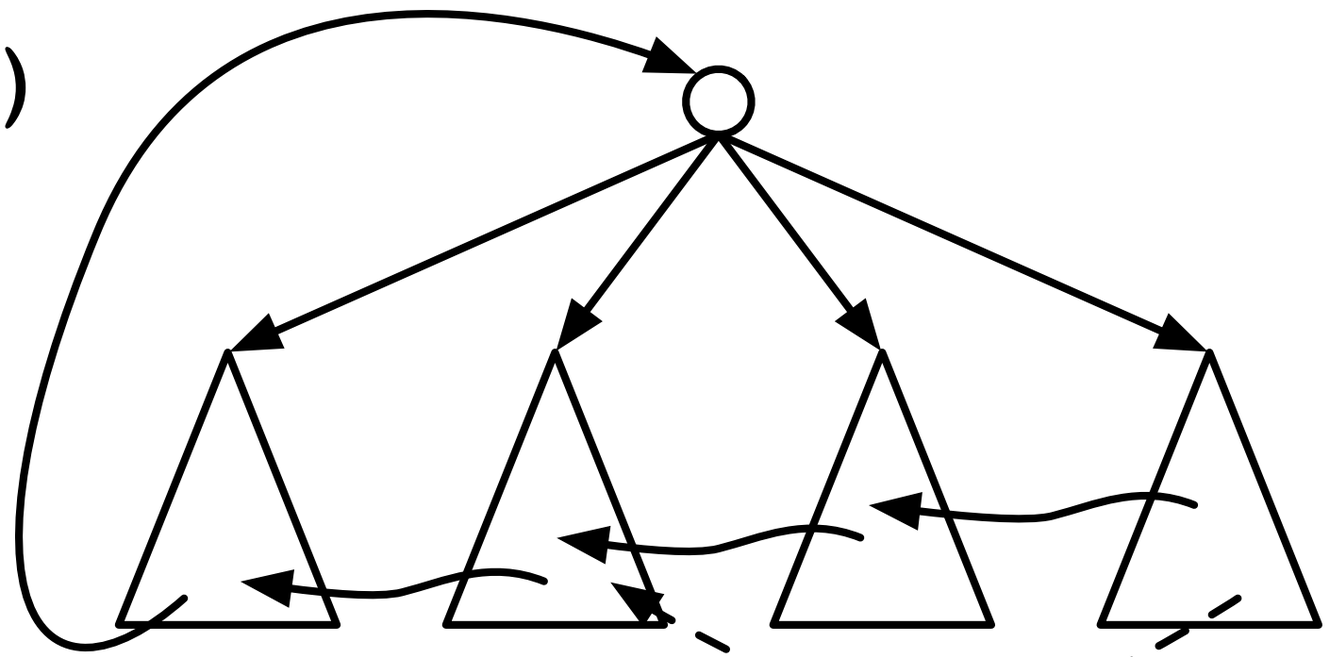


- check with DFS that graph has a single SCC

Example: Hamiltonian circuit

`circuit(`x_1`,...,`x_n`)`

graph with edges `i`→x_i
has single cycle
covering all nodes

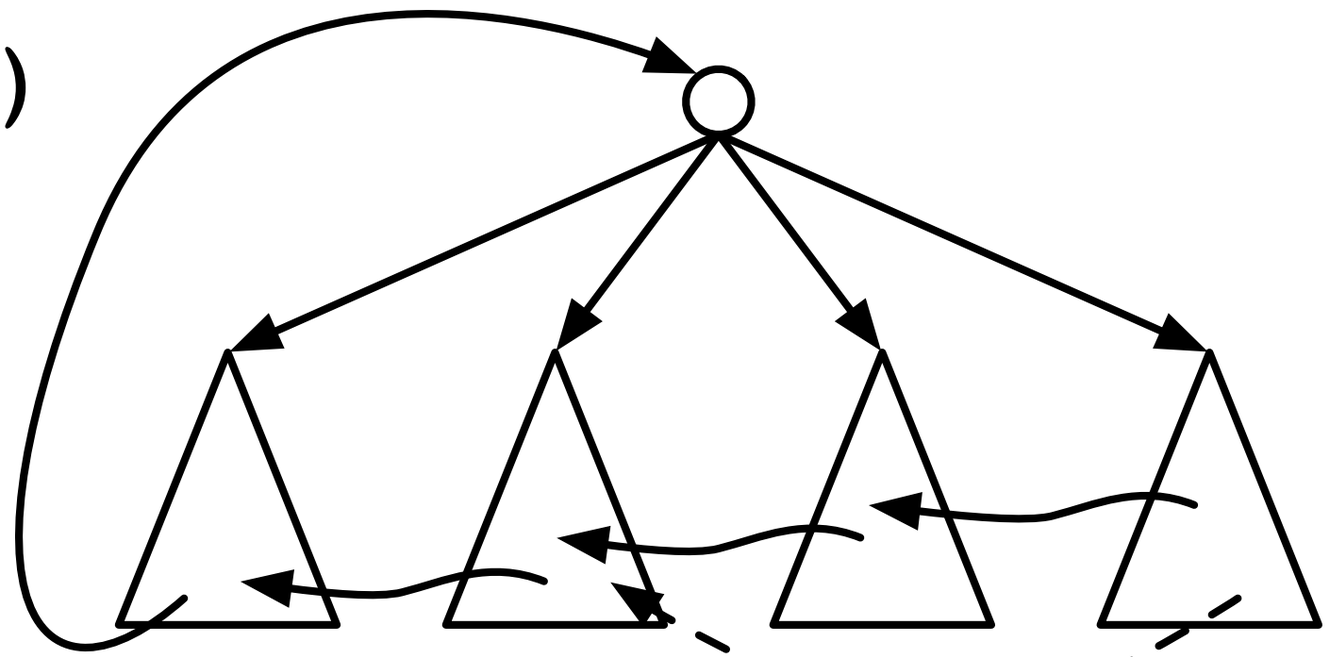


- check with DFS that graph has a single SCC
- prune edges between non-neighbor subtrees

Example: Hamiltonian circuit

`circuit(`x_1`,...,`x_n`)`

graph with edges `i`→x_i
has single cycle
covering all nodes

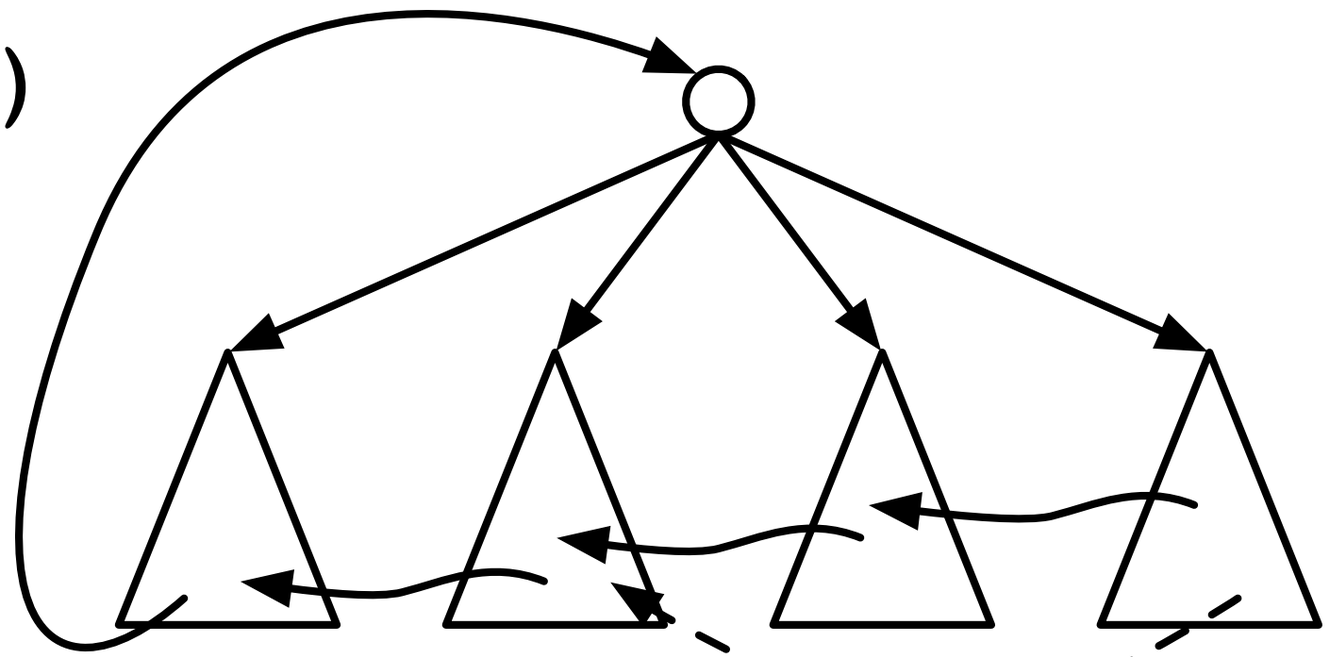


- check with DFS that graph has a single SCC
- prune edges between non-neighbor subtrees
- fix edges if single neighbor-edge is left

Example: Hamiltonian circuit

`circuit(`x_1`,...,`x_n`)`

graph with edges `i`→x_i
has single cycle
covering all nodes



- check with DFS that graph has a single SCC
- prune edges between non-neighbor subtrees
- fix edges if single neighbor-edge is left
- DFS start node arbitrary, thus **non-monotonic**

Example: Hamiltonian circuit

circu

graph v

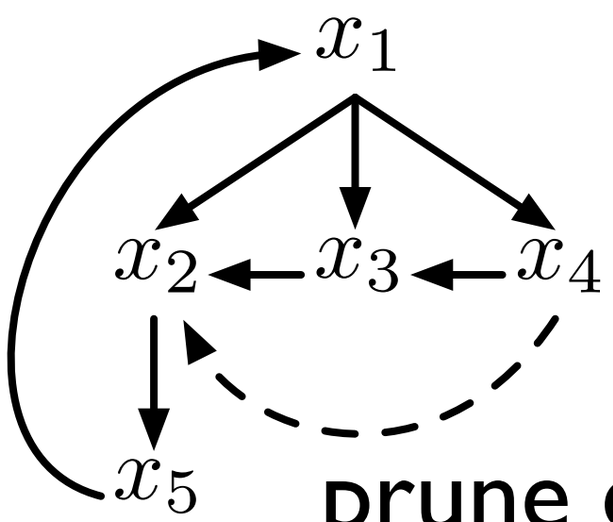
has sing

coverin

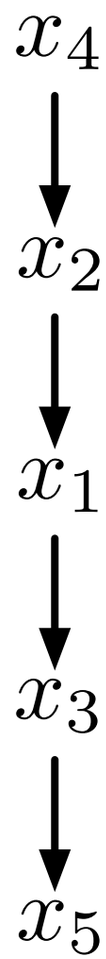
- chec
- prun
- fix e
- DFS

$$x_1 \in \{2,3,4\} \quad x_2 \in \{1,5\} \quad x_3 \in \{1,2,5\}$$

$$x_4 \in \{2,3\} \quad x_5 \in \{1,2\}$$



prune edge

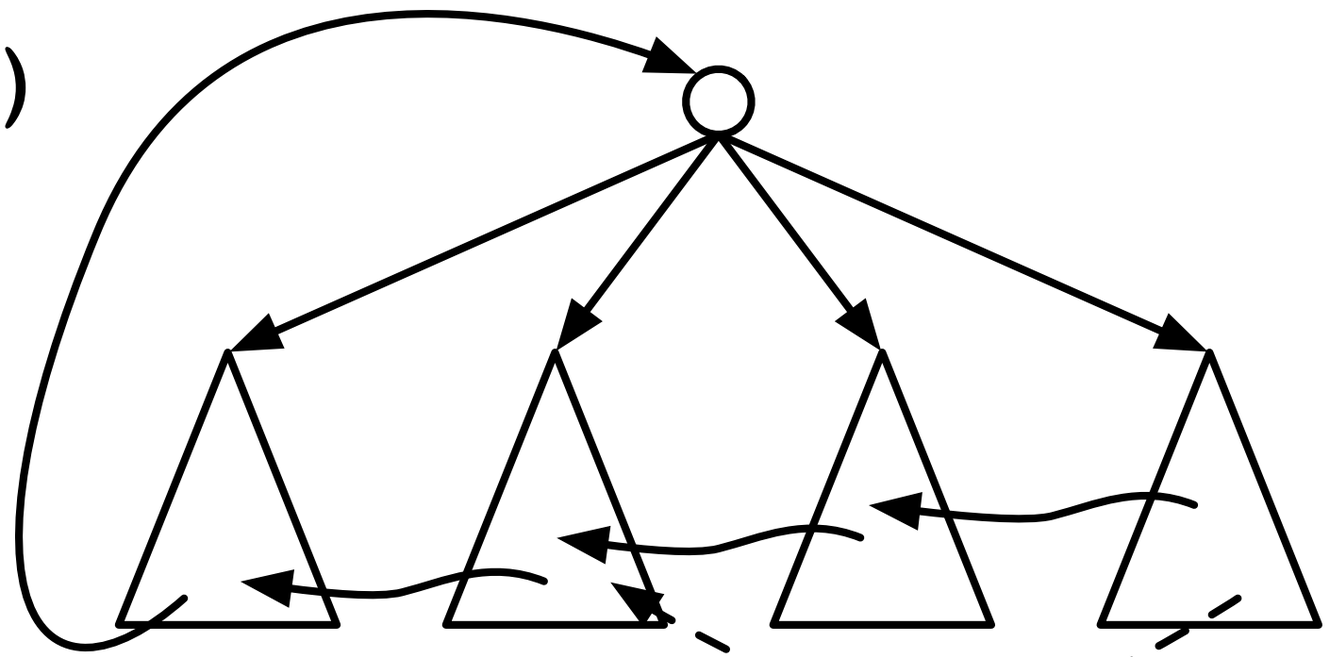


no pruning

Example: Hamiltonian circuit

`circuit(`x_1`,...,`x_n`)`

graph with edges `i`→x_i
has single cycle
covering all nodes



- check with DFS that graph has a single SCC
- prune edges between non-neighbor subtrees
- fix edges if single neighbor-edge is left
- DFS start node arbitrary, thus **non-monotonic**

A Realistic Model

Constraints

Assignments $a \in \mathrm{Asn} = X \to V$ $a(x) \in V$

Constraints $c \subseteq \mathrm{Asn}$

Constraints

Assignments $a \in \mathrm{Asn} = X \to V$ $a(x) \in V$

Constraints $c \subseteq \mathrm{Asn}$

$$
\begin{array}{rcll}
X & = & \{x, y, z\} & \\
V & = & \{1, 2, 3, 4\} & \\
d & = & \{x \mapsto V, y \mapsto V, z \mapsto V\} & \\
c_1 & = & \{a \in \mathrm{Asn} \mid a(x) \neq a(y) \wedge & \mathrm{distinct}(x, y, z) \\
 & & \quad a(x) \neq a(z) \wedge & \\
 & & \quad a(y) \neq a(z)\} & \\
c_2 & = & \{a \in \mathrm{Asn} \mid a(x) + a(y) = a(z)\} & x + y = z
\end{array}
$$

Propagators

- remove values that are inconsistent:

$$p(d) \subseteq d \quad \text{(contracting)}$$

Propagators

- remove values that are inconsistent:

$$p(d) \subseteq d \quad \text{(contracting)}$$

- decide on assignment if constraint is satisfied:

$$p(\{a\}) = \{a\} \quad \text{(accept)} \qquad p(\{a\}) = \emptyset \quad \text{(reject)}$$

Propagators

- remove values that are inconsistent:

$$p(d) \subseteq d \quad \text{(contracting)}$$

- decide on assignment if constraint is satisfied:

$$p(\{a\}) = \{a\} \ \text{(accept)} \qquad p(\{a\}) = \emptyset \ \text{(reject)}$$

- induce a constraint (accepted assignments):

$$c_p = \{a \in \mathrm{Asn} \mid p(\{a\}) = \{a\}\}$$

Propagators

- remove values that are inconsistent:

$$p(d) \subseteq d \quad \textbf{(contracting)}$$

- decide on assignment if constraint is satisfied:

$$p(\{a\}) = \{a\} \;\; \textbf{(accept)} \qquad p(\{a\}) = \emptyset \;\; \textbf{(reject)}$$

- induce a constraint (accepted assignments):

$$c_p = \{a \in \mathrm{Asn} \mid p(\{a\}) = \{a\}\}$$

- **must prune *consistently*:**

$$\forall a \in d : \;\; a \notin p(d) \Rightarrow a \notin c_p$$

Consistent Pruning

$$c_p = \{a \in \mathrm{Asn} \mid p(\{a\}) = \{a\}\}$$

$$\forall a \in d : \;\; a \notin p(d) \Rightarrow a \notin c_p$$

Consistent Pruning

$$c_p = \{a \in \mathrm{Asn} \mid p(\{a\}) = \{a\}\}$$

$$\forall a \in d : \;\; a \notin p(d) \Rightarrow a \notin c_p$$

- if p ever prunes an assignment from a domain, it must not belong to c_p

Consistent Pruning

$$c_p = \{a \in \mathrm{Asn} \mid p(\{a\}) = \{a\}\}$$

$$\forall a \in d : \;\; a \notin p(d) \Rightarrow a \notin c_p$$

- if p ever prunes an assignment from a domain, it must not belong to c_p

- monotonicity guarantees consistent pruning:

$$\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d)$$

$$\iff \quad a \in d \wedge p(\{a\}) = \{a\} \Rightarrow a \in p(d)$$

Consistent Pruning

$$c_p = \{a \in \mathrm{Asn} \mid p(\{a\}) = \{a\}\}$$

$$\forall a \in d : \;\; a \notin p(d) \Rightarrow a \notin c_p$$

- if p ever prunes an assignment from a domain, it must not belong to c_p

- monotonicity guarantees consistent pruning:

$$\boxed{\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d)}$$

$$\iff \quad a \in d \wedge p(\{a\}) = \{a\} \Rightarrow a \in p(d)$$

- this is **weak** monotonicity

Propagation

Variables $x, y, z \in X$ Values V

Domains $d(x) \subseteq V$

Propagators $\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d)$ (weakly m.)

$p(d)(x) \subseteq d(x)$ (contracting) ✔

$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$ (monotonic)

$p(p(d)) = p(d)$ (idempotent)

closure operators

Knaster-Tarski ⇒ weakest mutual fixpoint exists

$p(d) \subseteq dom(c_p \cap d)$ (complete)

Propagation

Variables $x, y, z \in X$ Values V

Domains $d(x) \subseteq V$

Propagators $\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d)$ **(weakly m.)**

$p(d)(x) \subseteq d(x)$ **(contracting)** ✔

~~$d' \subseteq d \Rightarrow p(d') \subseteq p(d)$~~ **(monotonic)**

~~$p(p(d)) = p(d)$~~ **(idempotent)**

closure operators

Knaster-Tarski ⇒ weakest mutual fixpoint exists

~~$p(d) \subseteq dom(c_p \cap d)$~~ **(complete)**

Propagation (revisited)

Variables $x, y, z \in X$ Values V

Domains $d(x) \subseteq V$

Propagators

$p(d)(x) \subseteq d(x)$ (contracting)

$\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d)$ (weakly m.)

Knaster-Tarski $\Rightarrow$ weakest mutual fixpoint exists

Propagation (revisited)

Variables $x, y, z \in X$ Values V

Domains $d(x) \subseteq V$

Propagators

$p(d)(x) \subseteq d(x)$ (contracting)

$\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d)$ (weakly monotonic)

Knaster-Tarski $\Rightarrow$ weakest mutual fixpoint exists

Propagation (revisited)

Variables $x, y, z \in X$ Values V

Domains $d(x) \subseteq V$

Propagators

$p(d)(x) \subseteq d(x)$ (contracting)

$\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d)$ (weakly monotonic)

Knaster-Tarski **does not apply**

Fixpoints (revisited)

- what does this compute now?

```
while (∃ p∈P such that p(d)≠d)
    d ← p(d)
```

Fixpoints (revisited)

- what does this compute now?

```
while (∃ p∈P such that p(d)≠d)
    d ← p(d)
```

- it still terminates with a mutual fixpoint

Fixpoints (revisited)

- what does this compute now?

```
while (∃ p∈P such that p(d)≠d)
    d ← p(d)
```

- it still terminates with a mutual fixpoint
- "weaker than GAC but not weaker than FC"

Fixpoints (revisited)

- what does this compute now?

```
while (∃ p∈P such that p(d)≠d)
    d ← p(d)
```

- it still terminates with a mutual fixpoint
- "weaker than GAC but not weaker than FC"
- **solution space is invariant**

Fixpoint Issues

- All depends on propagator order:
  - individual fixpoints (may not even be comparable!)
  - the shape of the search tree
  - the order of solutions

Fixpoint Issues

- All depends on propagator order:
  - individual fixpoints (may not even be comparable!)
  - the shape of the search tree
  - the order of solutions
- Adding constraints can yield bigger search trees!

Fixpoint Issues

- All depends on propagator order:
  - individual fixpoints (may not even be comparable!)
  - the shape of the search tree
  - the order of solutions
- Adding constraints can yield bigger search trees!
- **C'est la vie**

  (also happens with randomized heuristics, restarts, or parallel search)

Minimal model

$$p(d)(x) \subseteq d(x)$$ (contracting)

$$\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d)$$ (weakly monotonic)

Any contracting function can be made weakly monotonic by composition with a **checker!**

Minimal model

$$p(d)(x) \subseteq d(x) \qquad \text{(contracting)}$$

$$\{a\} \subseteq d \Rightarrow p(\{a\}) \subseteq p(d) \qquad \text{(weakly monotonic)}$$

Any contracting function can be made weakly monotonic by composition with a **checker!**

Model is minimally restrictive

Randomized Propagation

- What if a propagator prunes randomly?

Randomized Propagation

- What if a propagator prunes randomly?
- Propagators become relations

Randomized Propagation

- What if a propagator prunes randomly?
- Propagators become relations
- Iteration possibly does not even compute fixpoints

Randomized Propagation

- What if a propagator prunes randomly?
- Propagators become relations
- Iteration possibly does not even compute fixpoints
- But: propagator still checks its constraint

Randomized Propagation

- What if a propagator prunes randomly?
- Propagators become relations
- Iteration possibly does not even compute fixpoints
- But: propagator still checks its constraint
- **WMP guarantees correctness**

Randomized Propagation

- W
- Pr
- Ite
  fix
- Bu
- **W**

Circuit: random vs. fixed heuristic

runtime in % (smaller=better)

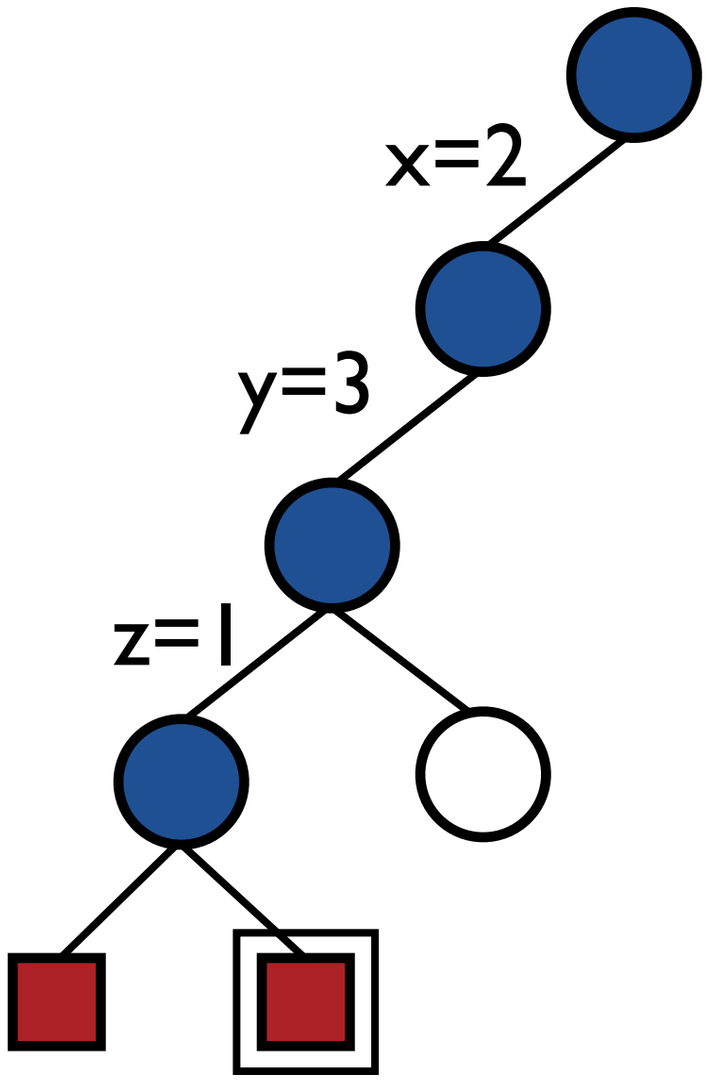
Knights 18	**17.0%**
Knights 20	98.1%
Knights 22	**1.1%**
Knights 24	**7.2%**
TSP br17	102.3%
TSP ftv33	97.1%

Randomized Propagation

- What if a propagator prunes randomly?
- Propagators become relations
- Iteration possibly does not even compute fixpoints
- But: propagator still checks its constraint
- **WMP guarantees correctness**

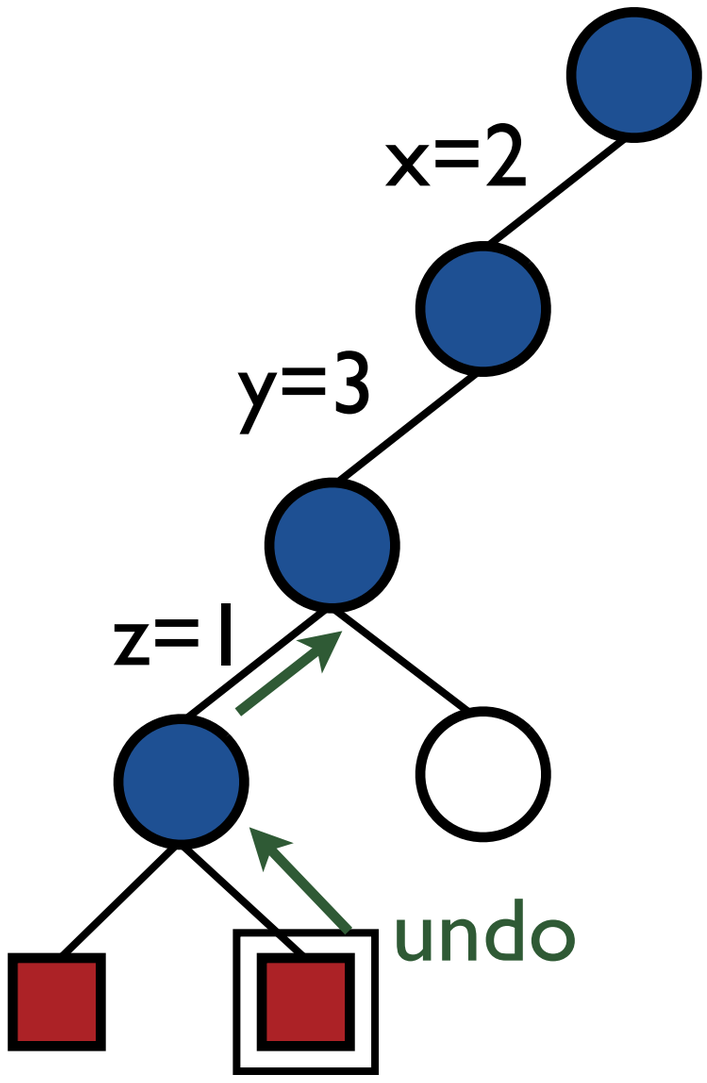
Non-monotonic search

Trailing



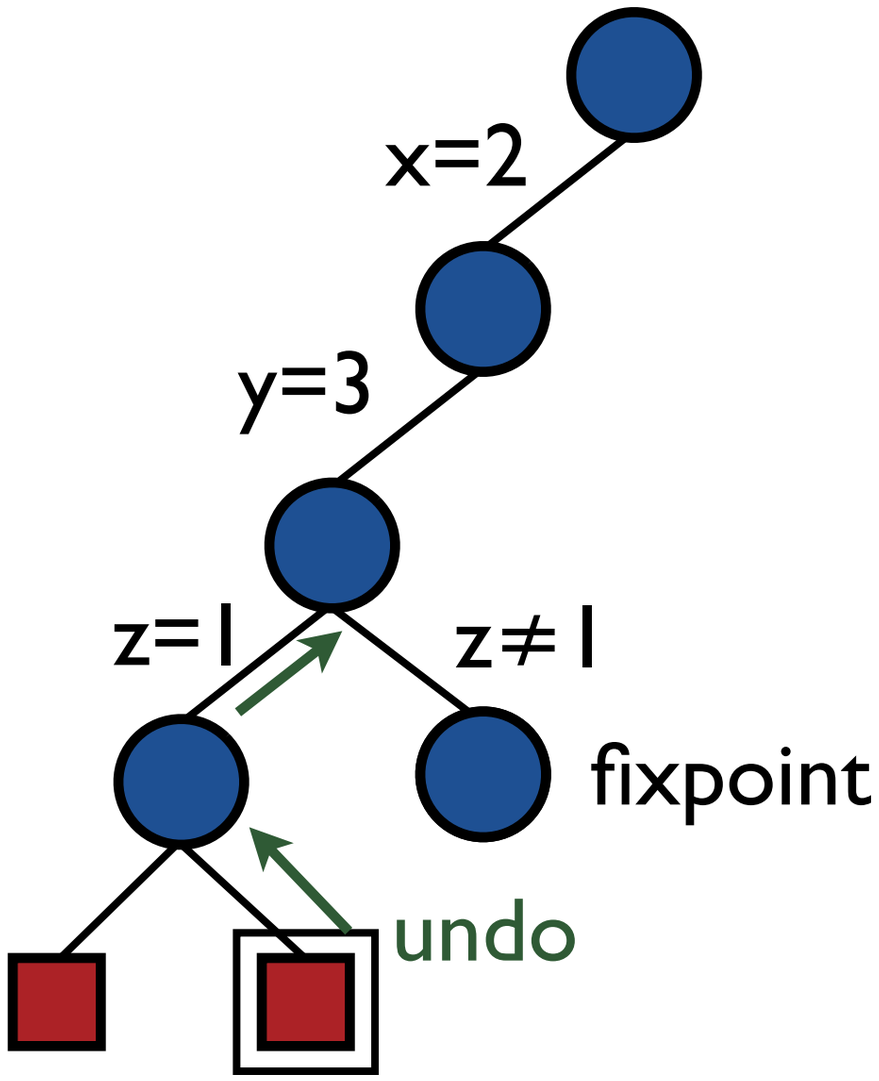
- record undo information on a *trail*
- on backtrack, restore ancestor state using undos
- for every node in the tree, compute exactly one fixpoint
- **supports non-monotonic propagation without change**

Trailing



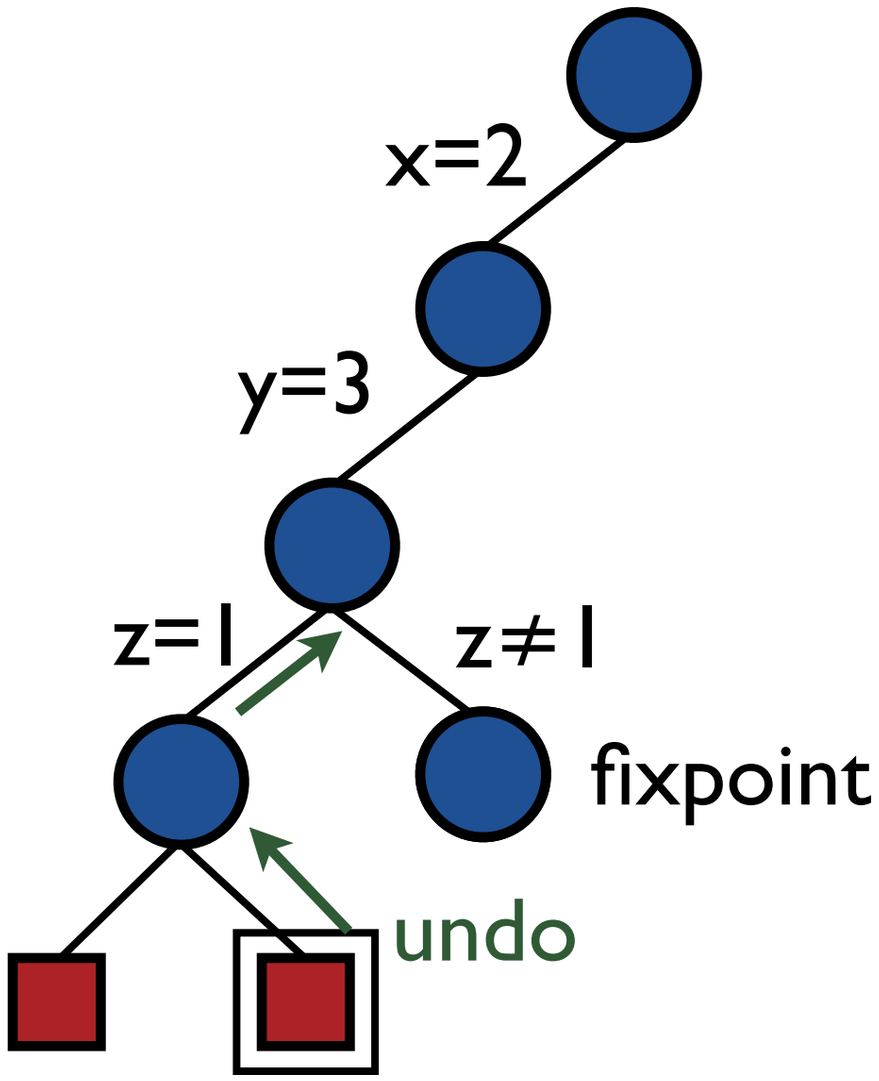
- record undo information on a *trail*
- on backtrack, restore ancestor state using undos
- for every node in the tree, compute exactly one fixpoint
- **supports non-monotonic propagation without change**

Trailing



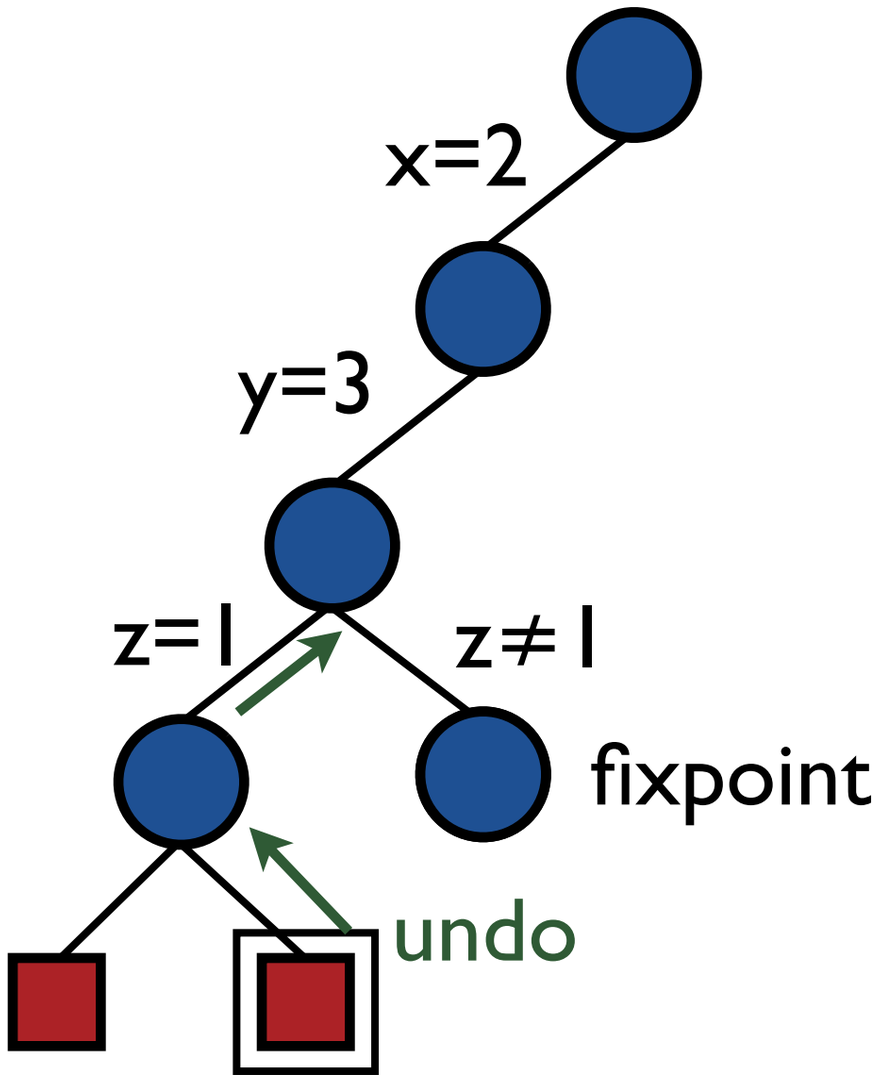
- record undo information on a *trail*
- on backtrack, restore ancestor state using undos
- for every node in the tree, compute exactly one fixpoint
- **supports non-monotonic propagation without change**

Trailing



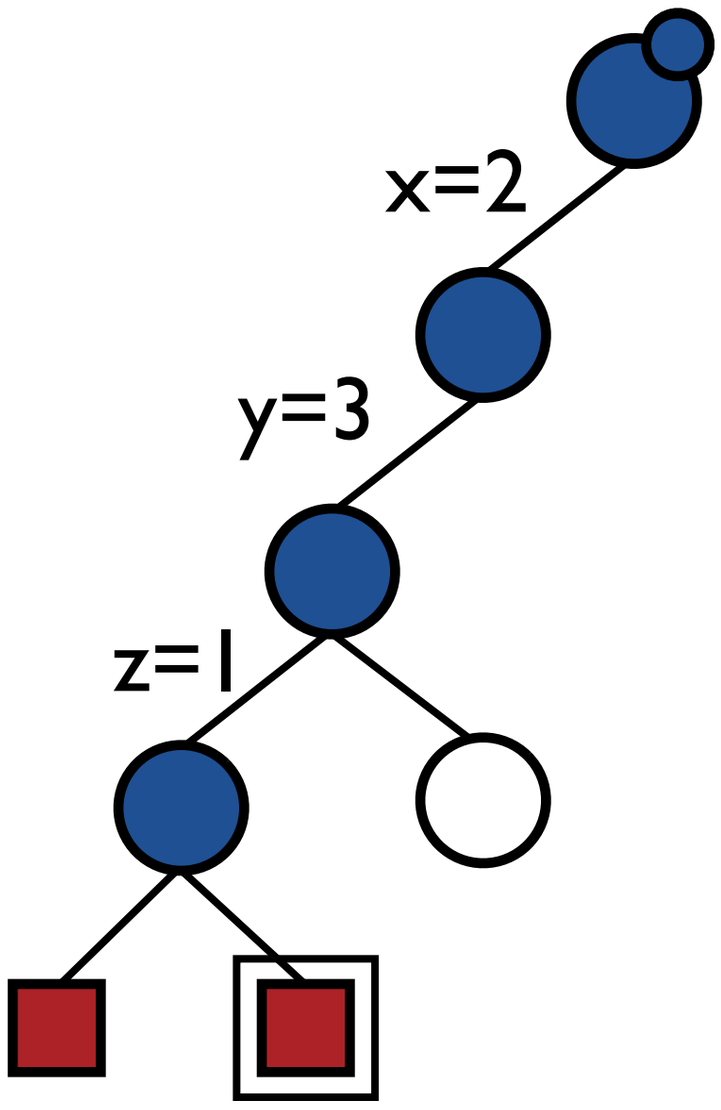
- record undo information on a *trail*
- on backtrack, restore ancestor state using undos
- for every node in the tree, compute exactly one fixpoint

Trailing



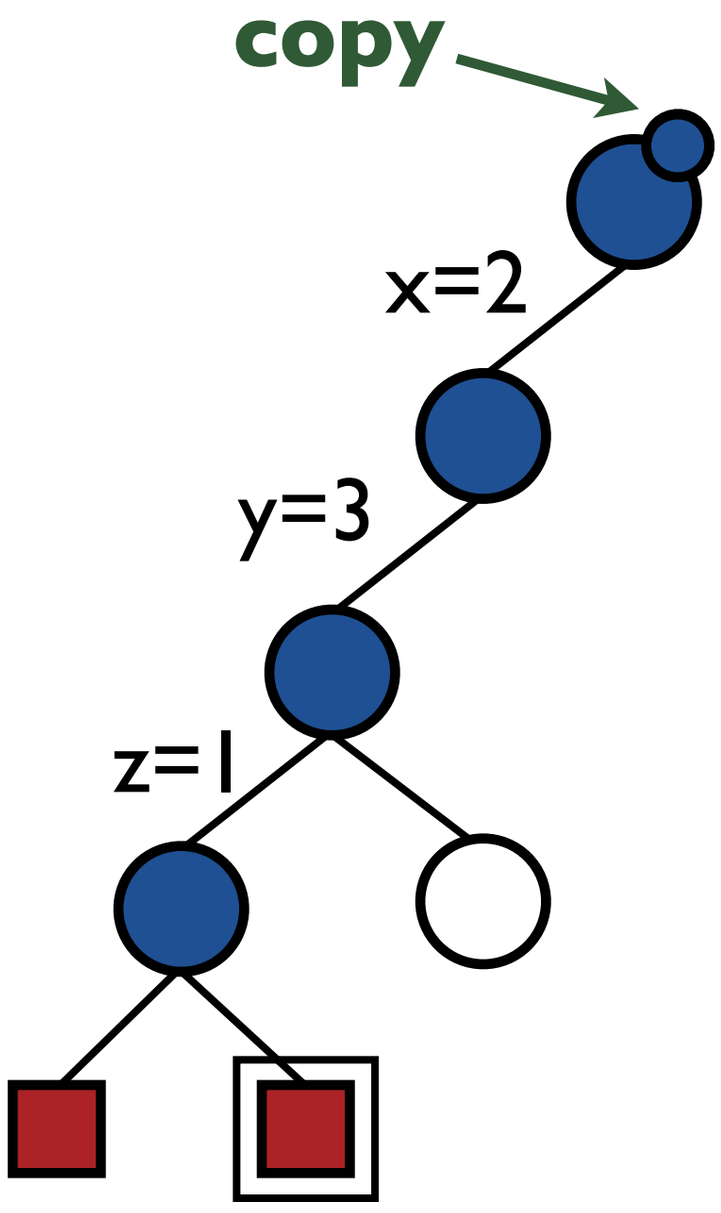
- record undo information on a *trail*
- on backtrack, restore ancestor state using undos
- for every node in the tree, compute exactly one fixpoint
- **supports non-monotonic propagation without change**

Fixpoint Recomputation (naive)



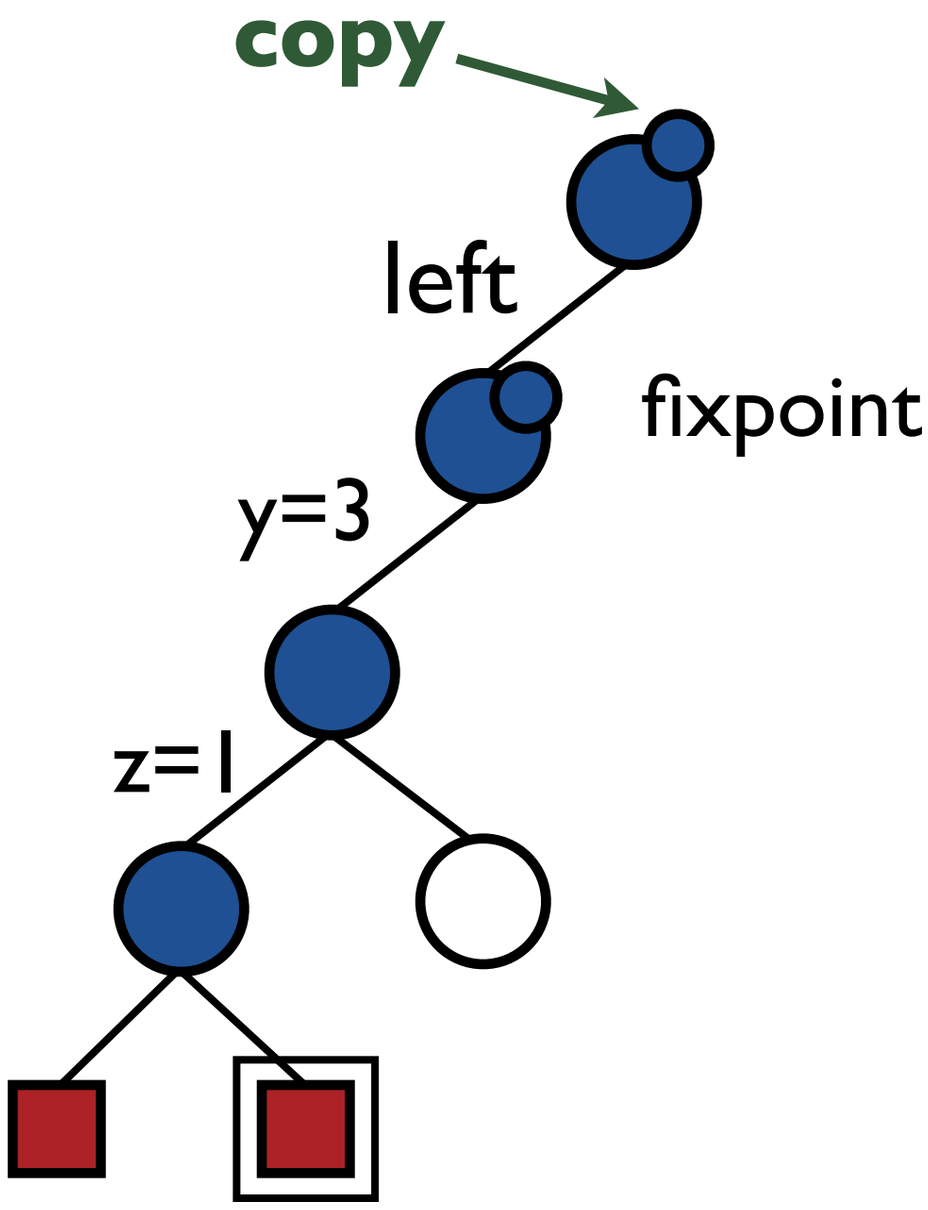
- store copies of some nodes
- upon backtrack, redo steps using path information (left, left, right)
- compute one fixpoint per step
- **does not work with non-monotonic propagators!**

Fixpoint Recomputation (naive)



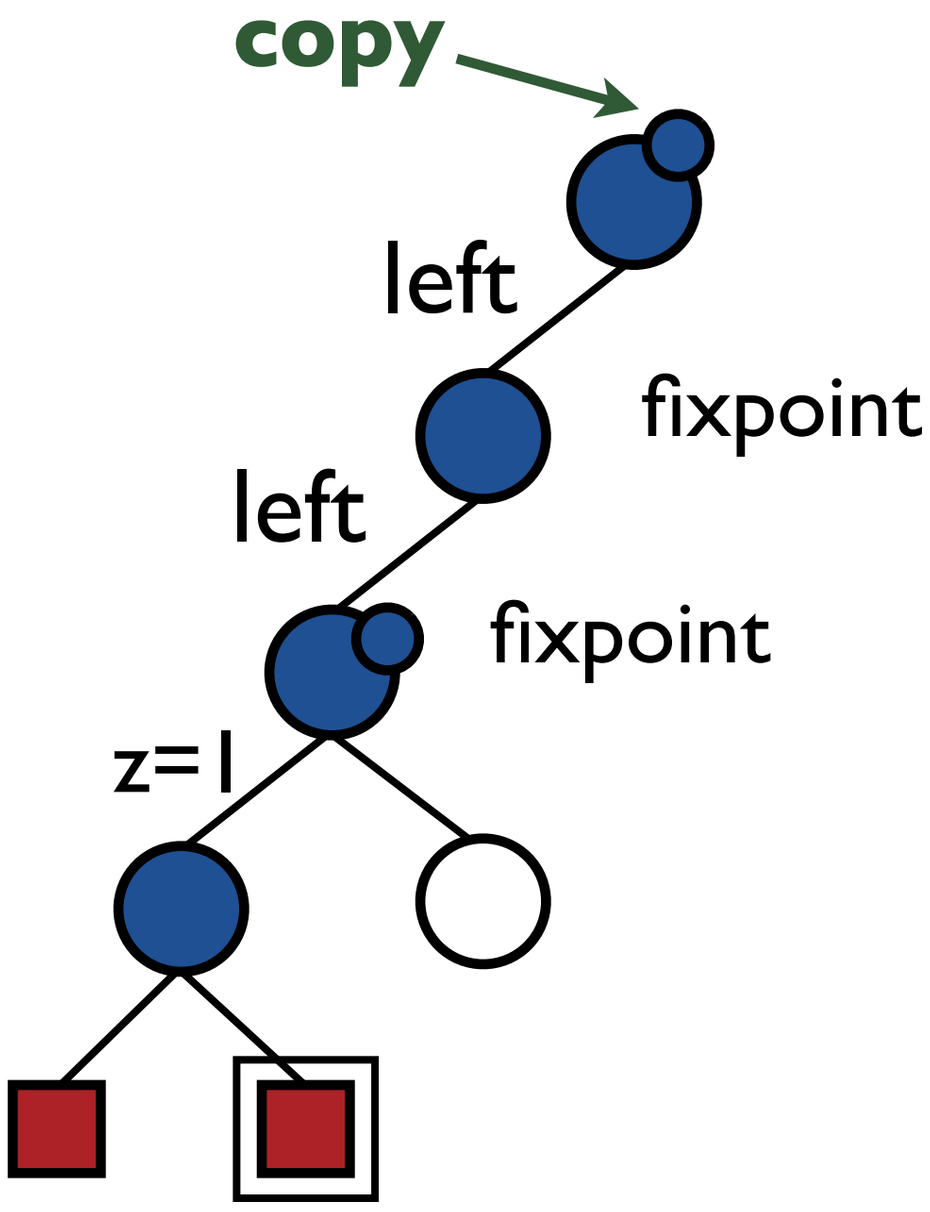
- store copies of some nodes
- upon backtrack, redo steps using path information (left, left, right)
- compute one fixpoint per step
- **does not work with non-monotonic propagators!**

Fixpoint Recomputation (naive)



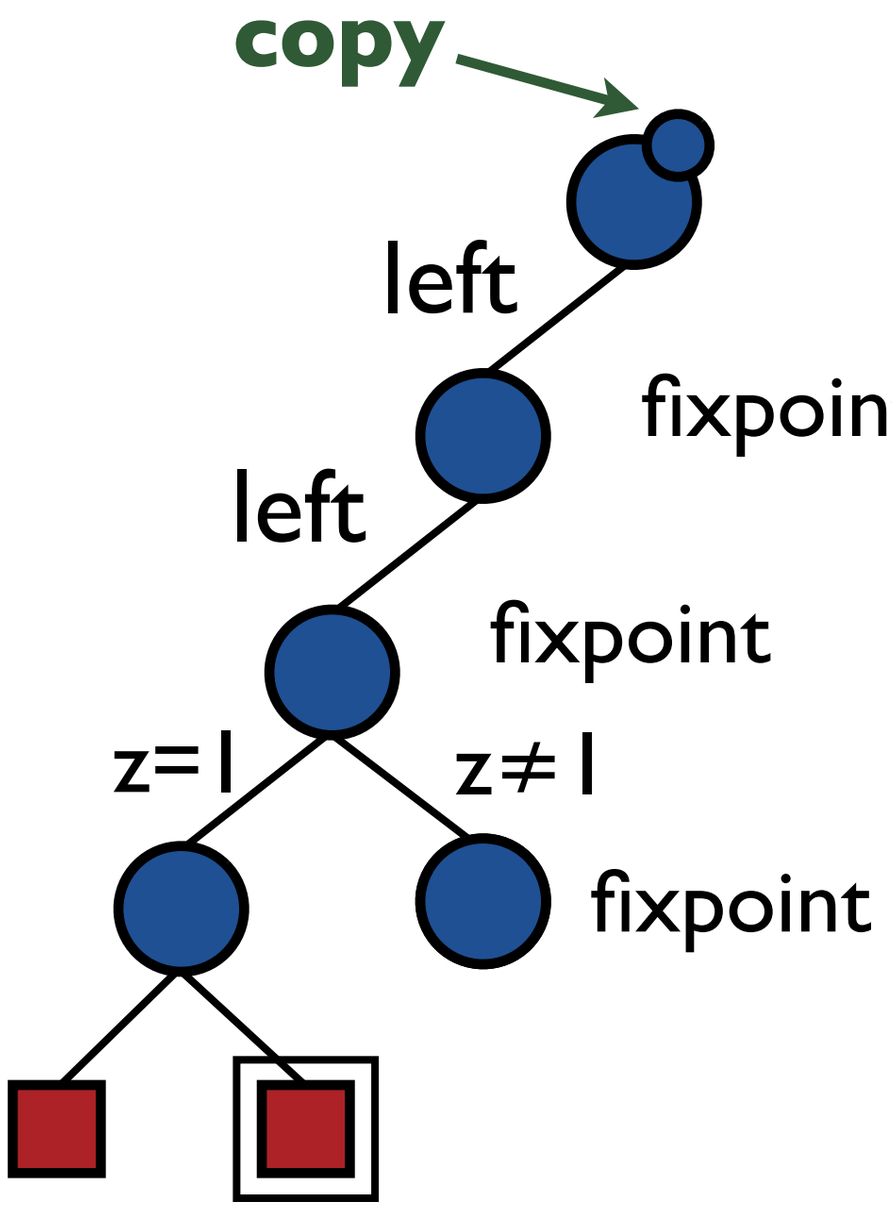
- store copies of some nodes
- upon backtrack, redo steps using path information (left, left, right)
- compute one fixpoint per step
- **does not work with non-monotonic propagators!**

Fixpoint Recomputation (naive)



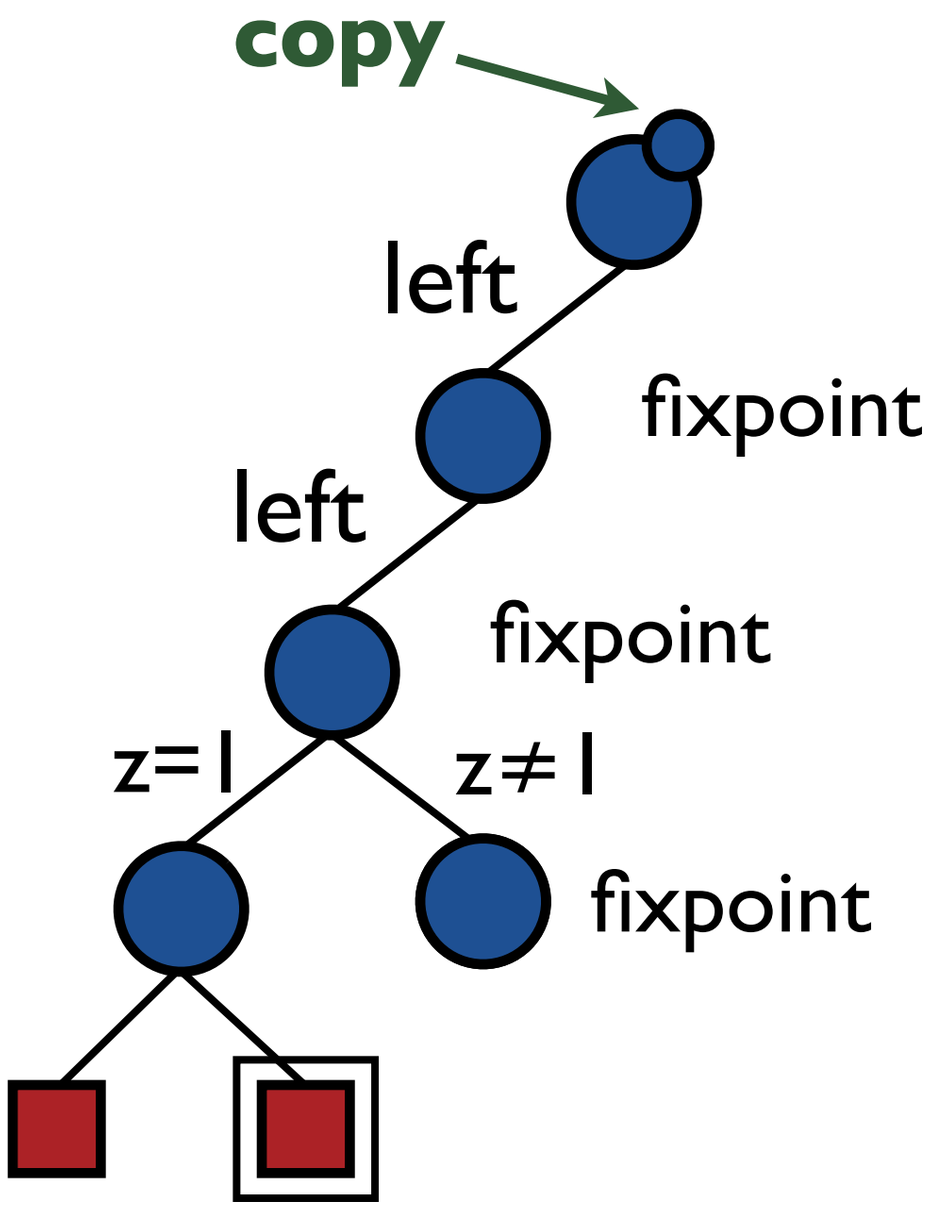
- store copies of some nodes
- upon backtrack, redo steps using path information (left, left, right)
- compute one fixpoint per step
- **does not work with non-monotonic propagators!**

Fixpoint Recomputation (naive)



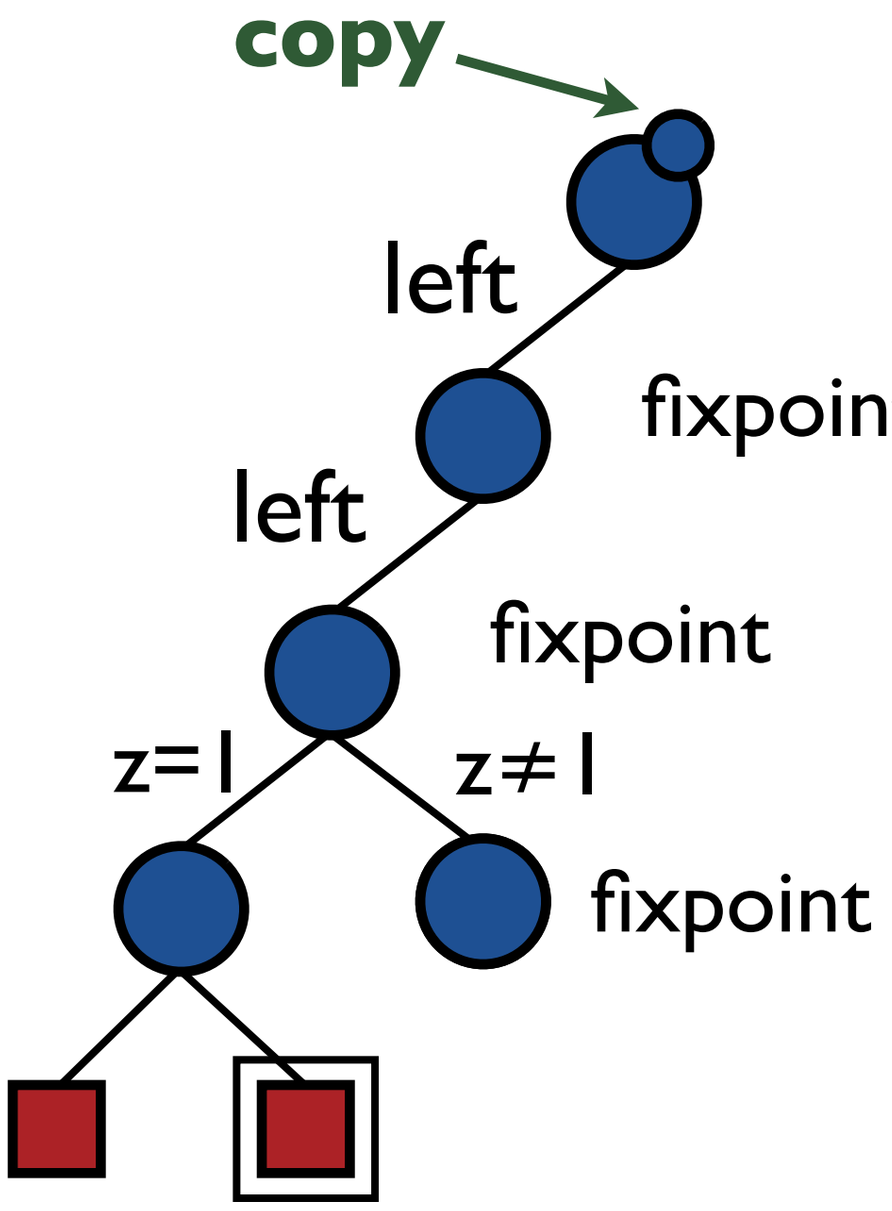
- store copies of some nodes
- upon backtrack, redo steps using path information (left, left, right)
- compute one fixpoint per step
- **does not work with non-monotonic propagators!**

Fixpoint Recomputation (naive)



- store copies of some nodes
- upon backtrack, redo steps using path information (left, left, right)
- compute one fixpoint per step

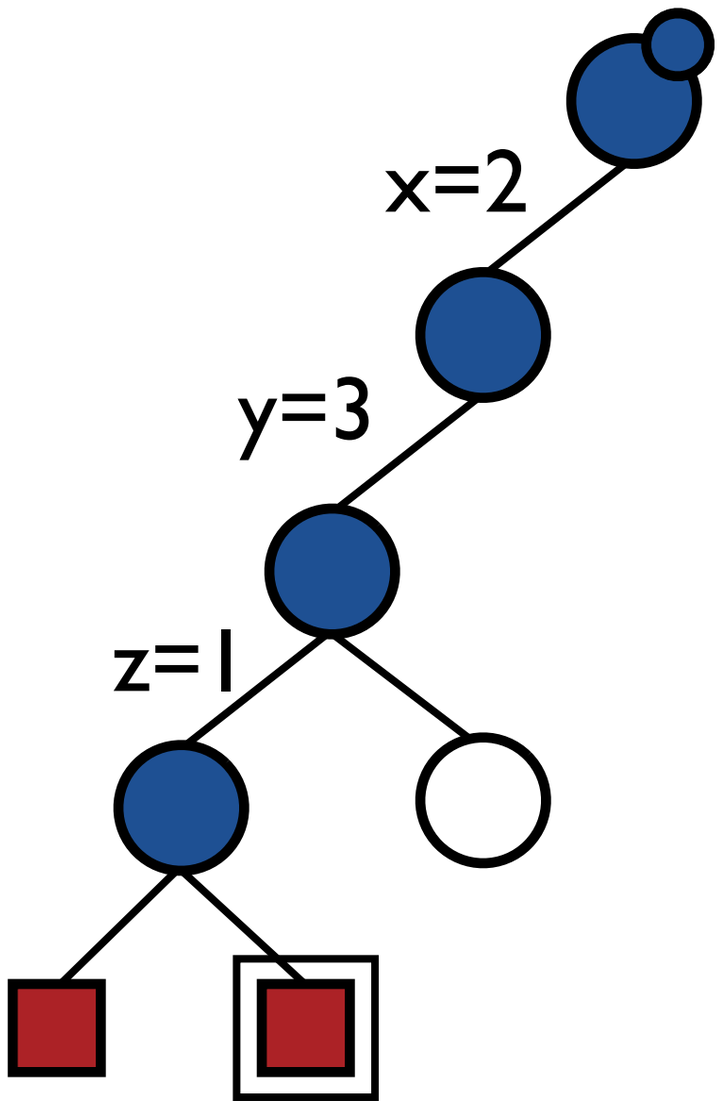
Fixpoint Recomputation (naive)



- store copies of some nodes
- upon backtrack, redo steps using path information (left, left, right)
- compute one fixpoint per step
- **does not work with non-monotonic propagators!**

[Choi et al 2001, Michel/Van Hentenryck 2004]

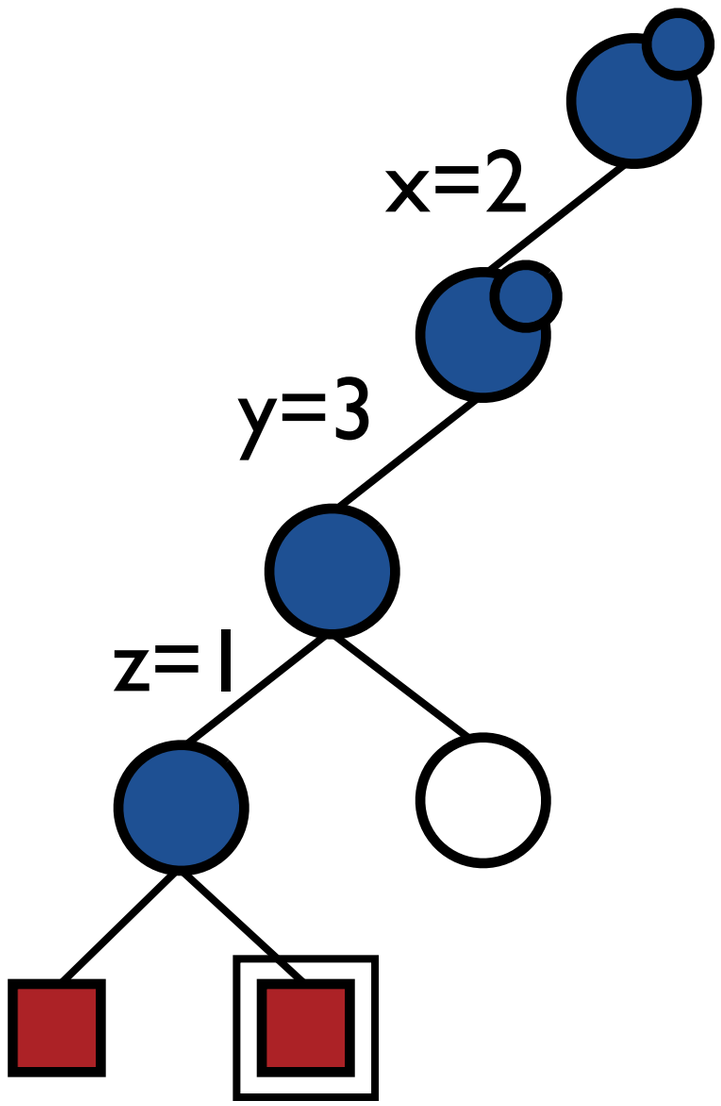
Path Recomputation



- upon backtrack, redo steps using *constraint* information (x=2,y=3,z≠1)
- only one fixpoint per node
- fixpoint may differ from exploration
- But: same set of solutions!
- **This works!**

[Choi et al 2001, Michel/Van Hentenryck 2004]

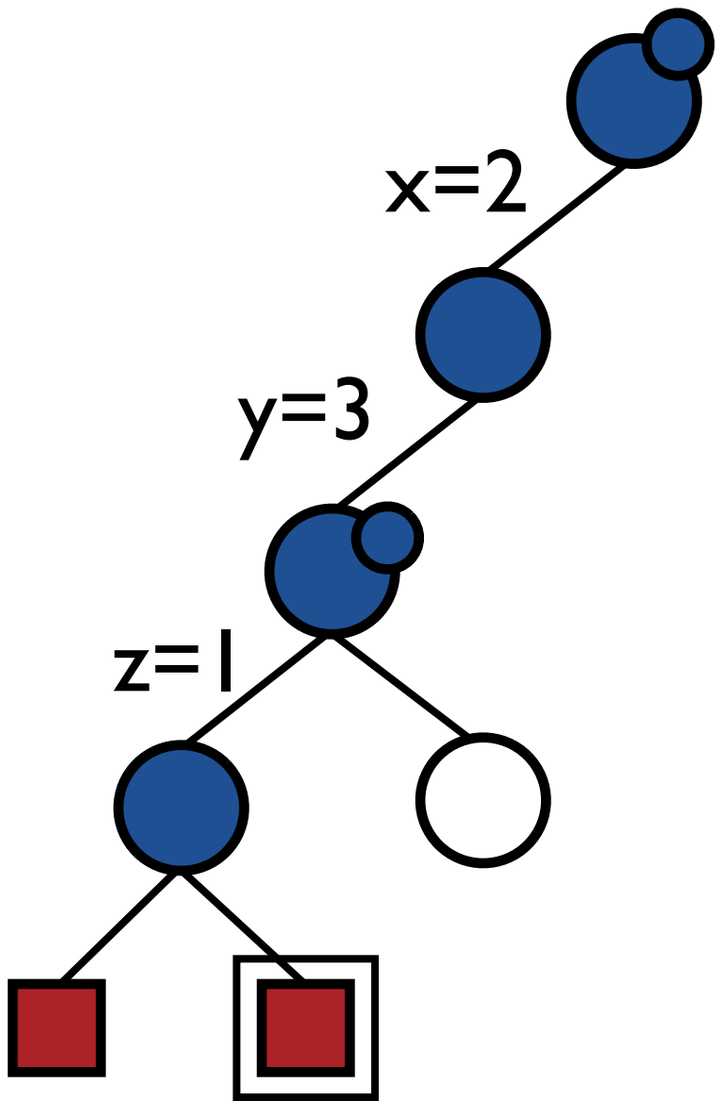
Path Recomputation



- upon backtrack, redo steps using *constraint* information (x=2,y=3,z≠1)
- only one fixpoint per node
- fixpoint may differ from exploration
- But: same set of solutions!
- **This works!**

[Choi et al 2001, Michel/Van Hentenryck 2004]

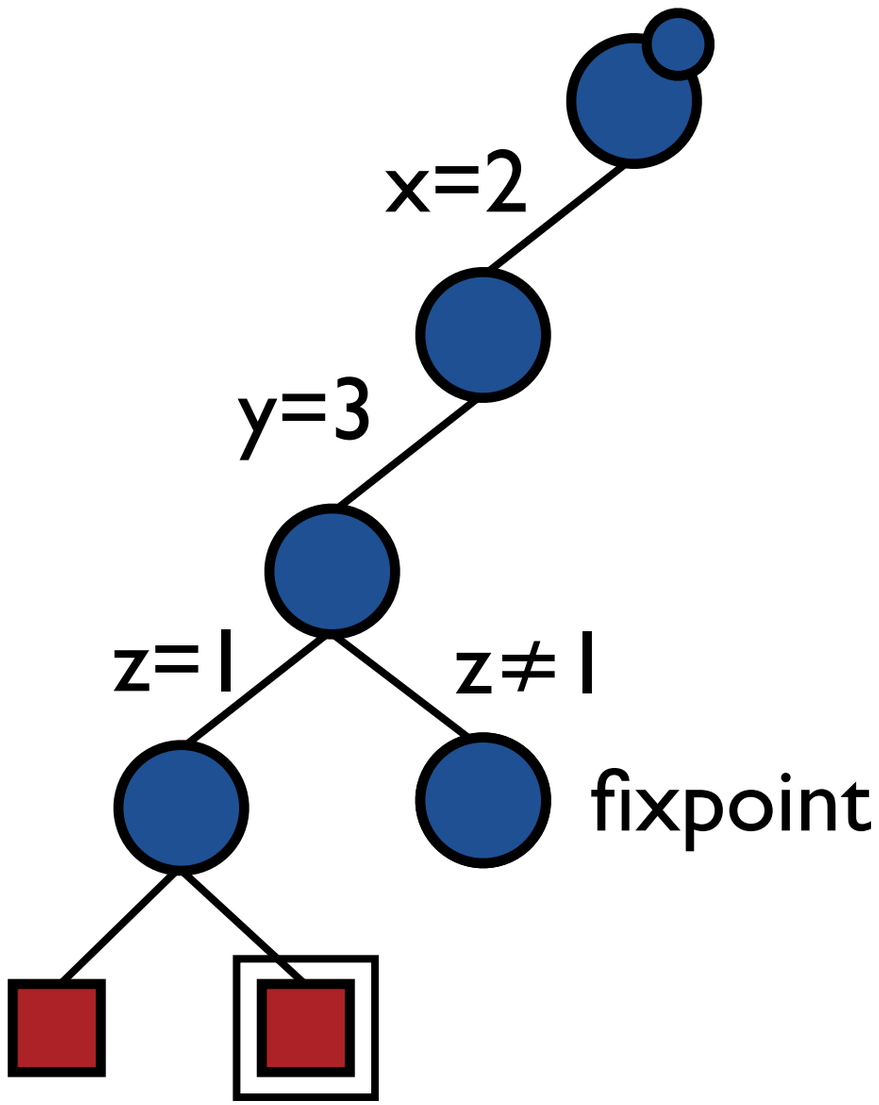
Path Recomputation



- upon backtrack, redo steps using *constraint* information (x=2,y=3,z≠1)
- only one fixpoint per node
- fixpoint may differ from exploration
- But: same set of solutions!
- **This works!**

[Choi et al 2001, Michel/Van Hentenryck 2004]

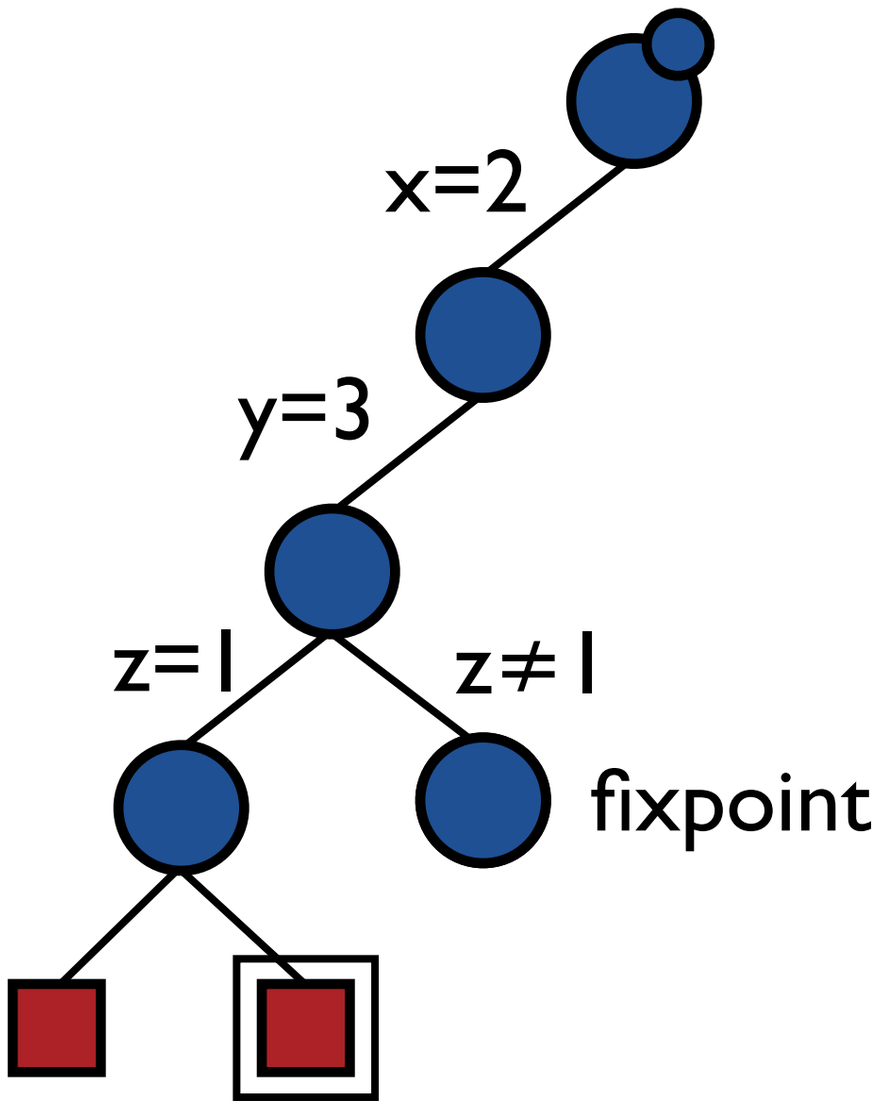
Path Recomputation



- upon backtrack, redo steps using *constraint* information (x=2,y=3,z≠1)
- only one fixpoint per node
- fixpoint may differ from exploration
- But: same set of solutions!
- **This works!**

[Choi et al 2001, Michel/Van Hentenryck 2004]

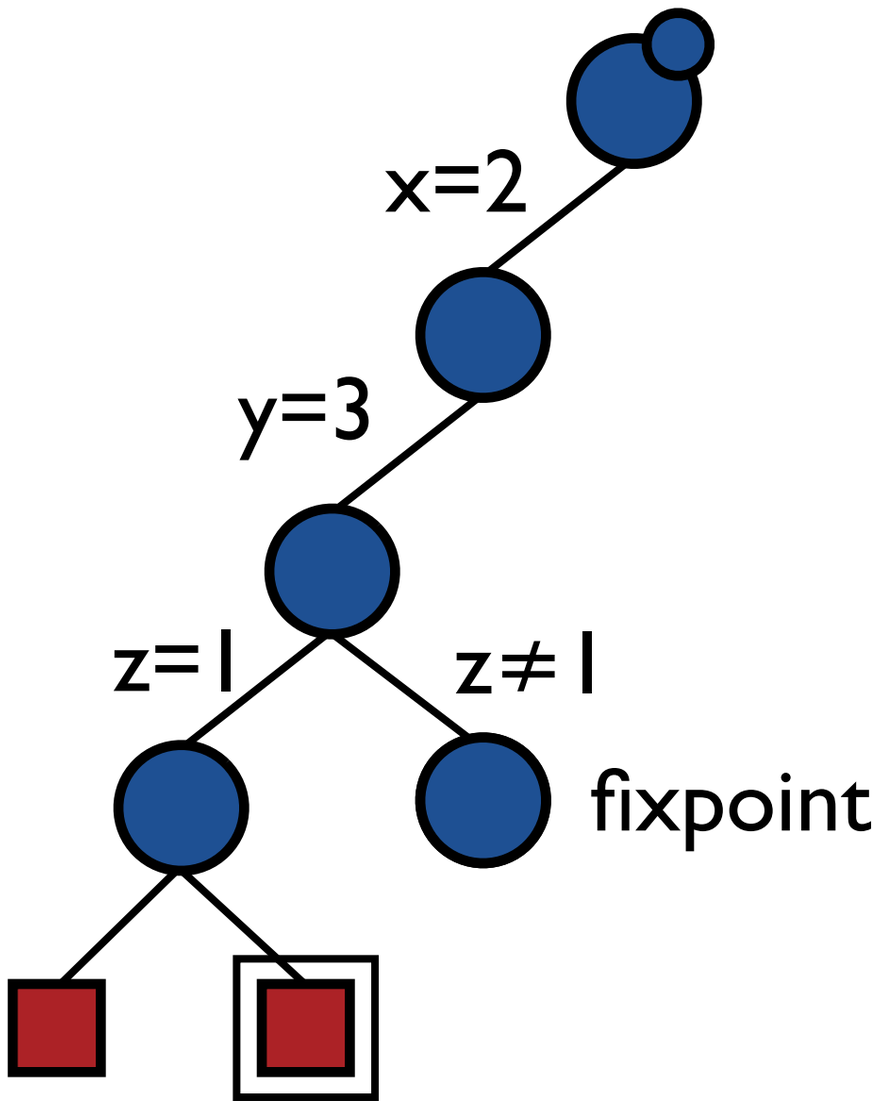
Path Recomputation



- upon backtrack, redo steps using *constraint* information (x=2,y=3,z≠1)
- only one fixpoint per node
- fixpoint may differ from exploration

[Choi et al 2001, Michel/Van Hentenryck 2004]

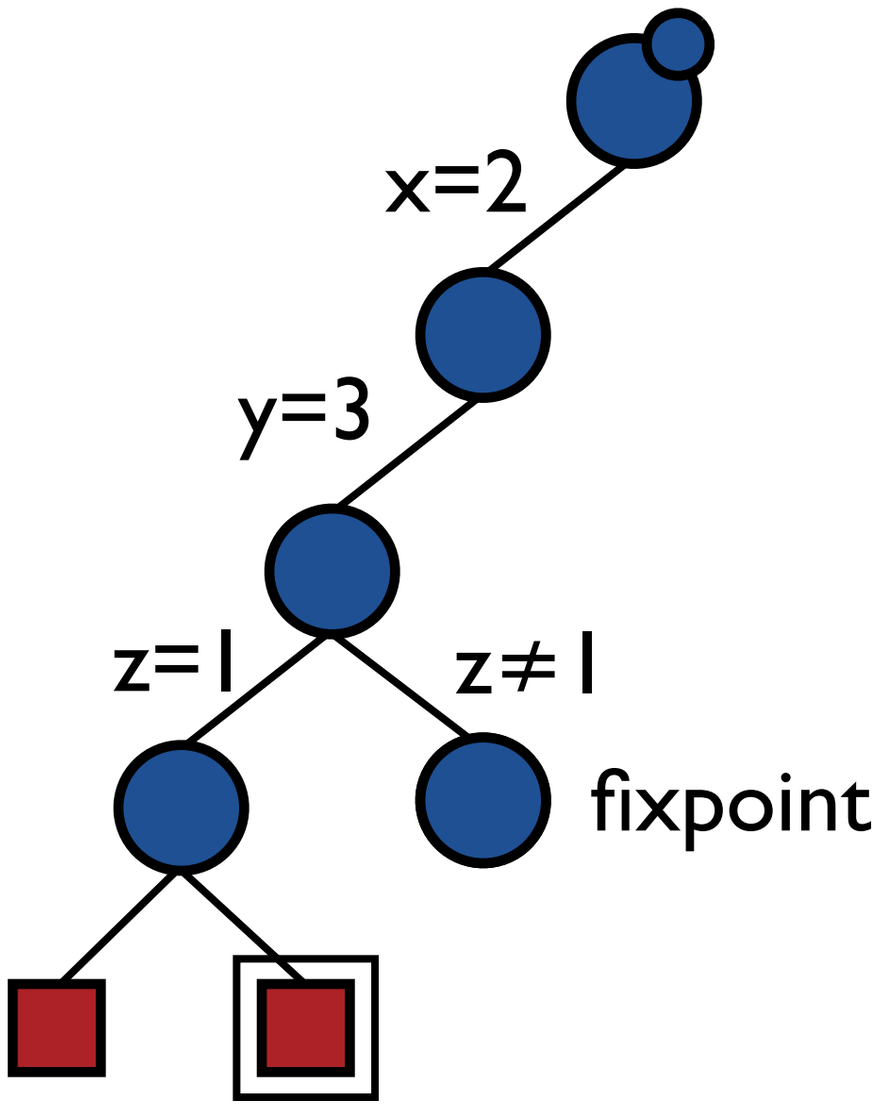
Path Recomputation



- upon backtrack, redo steps using *constraint* information (x=2,y=3,z≠1)
- only one fixpoint per node
- fixpoint may differ from exploration
- But: same set of solutions!

[Choi et al 2001, Michel/Van Hentenryck 2004]

Path Recomputation



- upon backtrack, redo steps using *constraint* information (x=2,y=3,z≠1)
- only one fixpoint per node
- fixpoint may differ from exploration
- But: same set of solutions!
- **This works!**

Why recomputation matters

- Essential for systems based on copying
- Control memory consumption
- Implement involved search strategies (e.g. A^*)

Why recomputation matters

- Essential for systems based on copying
- Control memory consumption
- Implement involved search strategies (e.g. A^*)
- Greatly simplifies parallel search
  - pass copies to different workers
  - work stealing through recomputation

Why recomputation matters

- Essential for systems based on copying
- Control memory consumption
- Implement involved search strategies (e.g. A^*)
- Greatly simplifies parallel search
  - pass copies to different workers
  - work stealing through recomputation
- **More later in this session**

Summary

- Monotonicity is unrealistic: prohibits approximative, heuristic, randomized algorithms
- Realistic model:

  contraction + weak monotonicity

- Minimal model that makes solver sound and complete
- Works with recomputation

Summary

- Monotonicity is unrealistic: prohibits approximative, heuristic, randomized algorithms
- Realistic model:

contraction + weak monotonicity

- Minimal model that makes solver sound and complete
- Works with recomputation

Thanks!