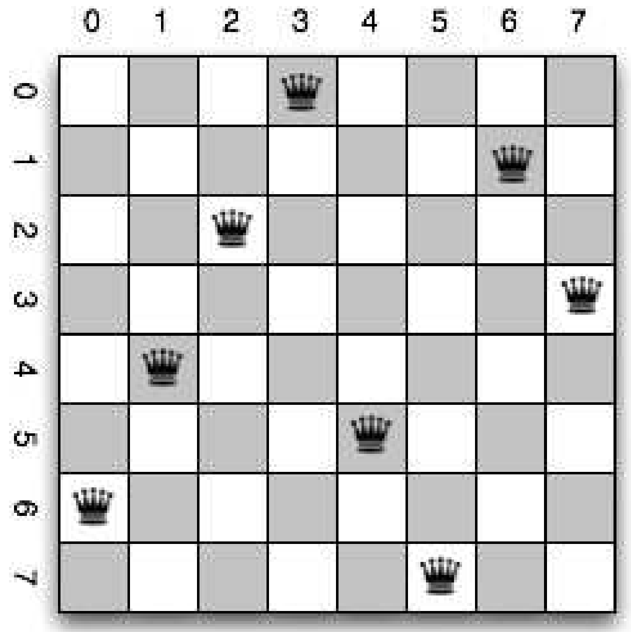
Knowledge Representation and Reasoning: Solving Constraint Satisfaction Problems

Chapter 7

Constraint Satisfaction Problems



◇ Binary constraint network $\gamma = \langle V, D, C \rangle$
— V a finite set of variables $v_1, \ldots, v_n$
— D a set of [finite] sets $D_{v_1}, \ldots, D_{v_n}$
— C a set of binary relations $\{C_{u,v} \mid u, v \in V, u \neq v\}$
$C_{u,v} \subseteq D_u \times D_v$

Outline of the lecture

- ♢ Introduction
- ♢ Variable and value ordering
- ♢ Inference
- ♢ Forward checking
- ♢ Arc consistency

Recall Backtracking

function BACKTRACK(γ, a) **returns** solution, or "inconsistent"

  if a is inconsistent with γ **then return** "inconsistent"
  if a is total **then return** a
  select variable v for which a is not defined
  for each d in D_v **do**
    $a' \leftarrow a \cup \{(v, d)\}$
    $a'' \leftarrow$ BACKTRACK(γ, a')
    if $a'' \neq$ "inconsistent" **then return** a''
  end
  return "inconsistent"

call: BACKTRACK(γ, $\{\}$)

// Tree constructed

Backtracking: the Good and the Bad

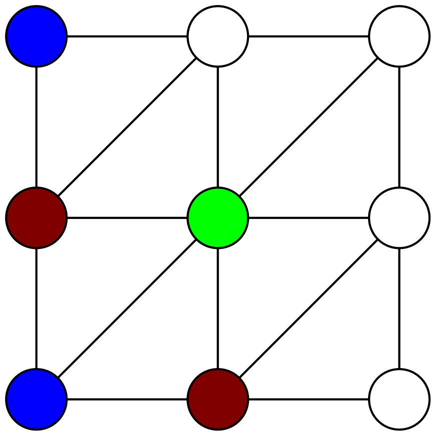
- Better that exhaustive search: avoids enumerating many inconsistent (partial) assignments by detecting them as soon as they happen

- Once an inconsistent partial assignment is reached, all of its extensions are pruned

- **Advantages:**
  Very simple to implement
  Very fast (per node of the search tree)
  Complete (always gives a decision)

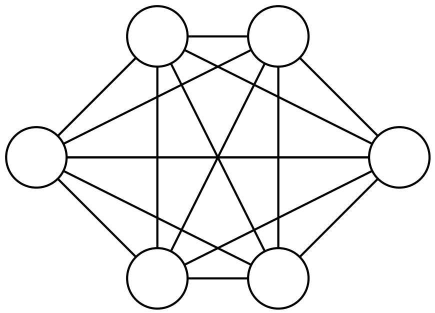
- **Disadvantages:**
  Does no reasoning except detecting actual inconsistency
  Cannot look further ahead than the current state

Simple example: Graph colouring



- Given an undirected graph with n nodes, given k colours, assign a colour to each node so that no two adjacent nodes (with an edge between them) are the same colour.
- Representation using binary constraints is easy.
- Problem is NP-complete, so difficult in the worst case.

Special case: a clique with k = n



◇ In an optimal search tree, generating all $n!$ solutions: the number of partial assignments is

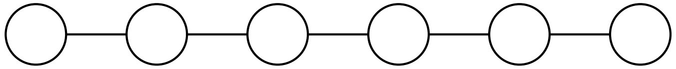
$$\sum_{i=0}^{n} \frac{n!}{i!} \quad \longrightarrow \quad n!e$$

$\Uparrow$ constant

◇ In pure backtracking search, every consistent partial assignment is extended to a solution

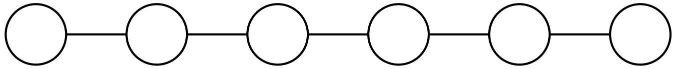
◇ Hence in this case simple backtracking is essentially optimal: the number of consistent nodes expanded is as small as it could be.

Another special case: a chain



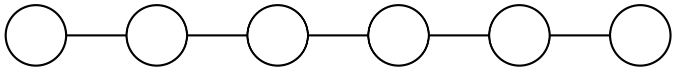
- Suppose we always choose the leftmost uncoloured node to colour next
- k consistent choices for the first node
- $k - 1$ consistent choices for the second node
- $k - 1$ consistent choices for the third node

Another special case: a chain



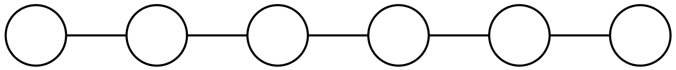
- Suppose we always choose the leftmost uncoloured node to colour next
- k consistent choices for the first node
- $k - 1$ consistent choices for the second node
- $k - 1$ consistent choices for the third node

Another special case: a chain



- $\diamondsuit$ Suppose we always choose the leftmost uncoloured node to colour next
- $\diamondsuit$ k consistent choices for the first node
- $\diamondsuit$ $k-1$ consistent choices for every other node

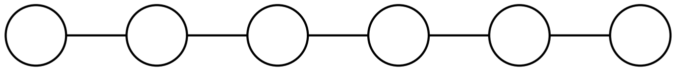
Another special case: a chain



- Suppose we always choose the leftmost uncoloured node to colour next
- k consistent choices for the first node
- $k-1$ consistent choices for every other node
- $k(k-1)^{i-1}$ consistent partial assignments of length i where $1 \leq i \leq n$
- Total number of consistent partial assignments is

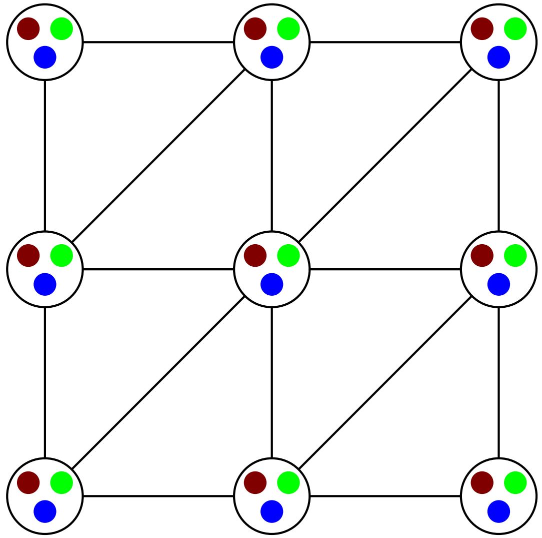
$$1 + k \sum_{i=0}^{n-1} (k-1)^i$$

Another special case: a chain



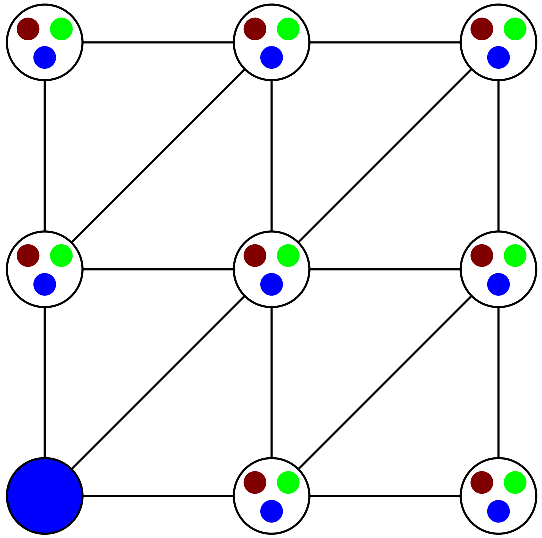
- ♢ Again pure backtracking is optimal in consistent partial assignments
- ♢ Other variable orderings give different numbers of partial assignments
- ♢ But in all cases pure backtracking is as good as anything
- ♢ That means no inference is needed: every consistent partial assignment can be extended to a complete assignment
- ♢ This sort of case is rare

Inference



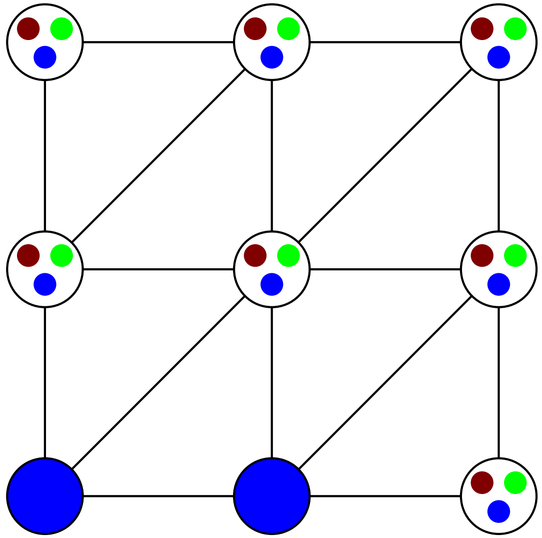
◊ Assign values from the bottom left corner, going across the rows

Inference



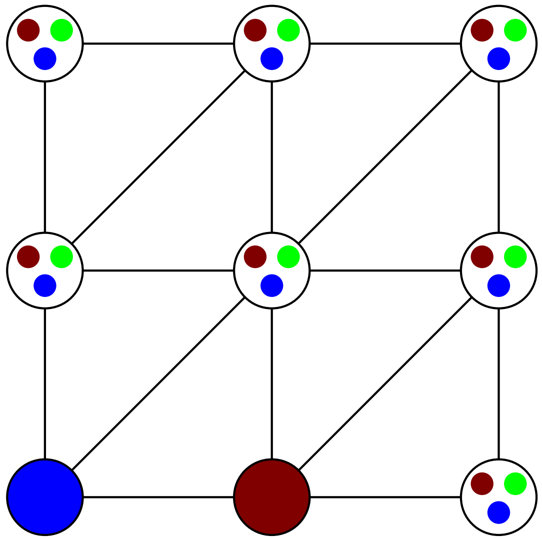
- Assign values from the bottom left corner, going across the rows
- Choose blue first

Inference



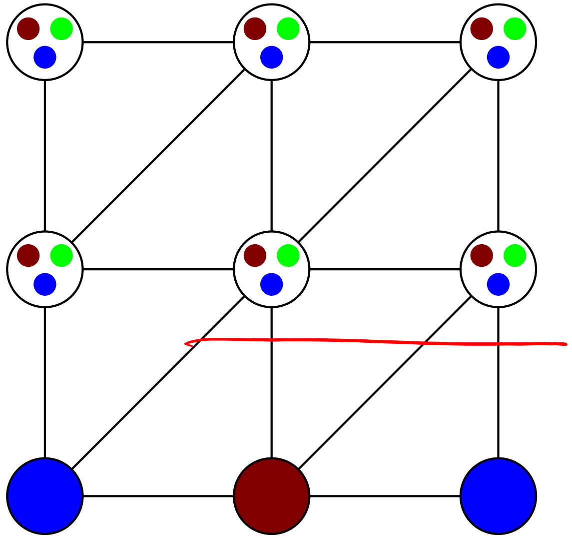
- ♢ Assign values from the bottom left corner, going across the rows
- ♢ Choose blue first
- ♢ Inconsistent

Inference



- ♢ Assign values from the bottom left corner, going across the rows
- ♢ Choose blue first
- ♢ Choose red next

Inference

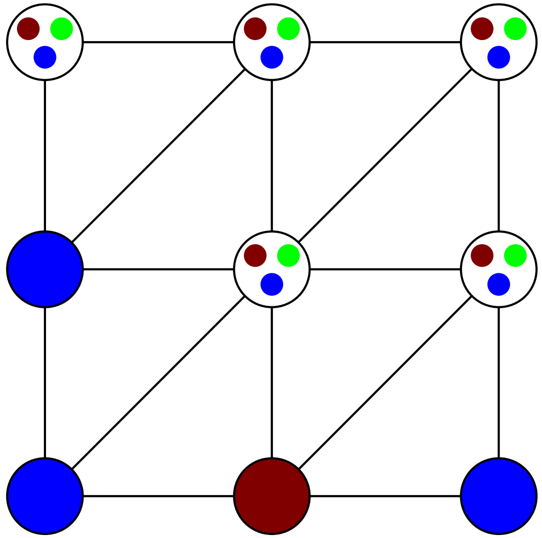


forced to be green

but backtracking does not look ahead.

- ♢ Assign values from the bottom left corner, going across the rows
- ♢ Choose blue first
- ♢ Now nodes 5 and 6 must both be green

Inference

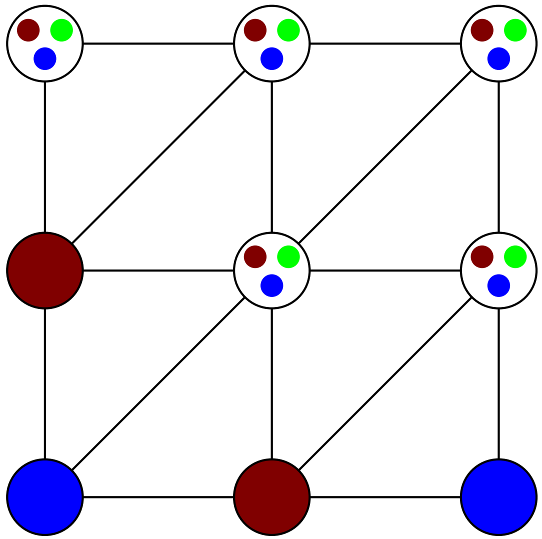


◊ Assign values from the bottom left corner, going across the rows

◊ Choose blue first

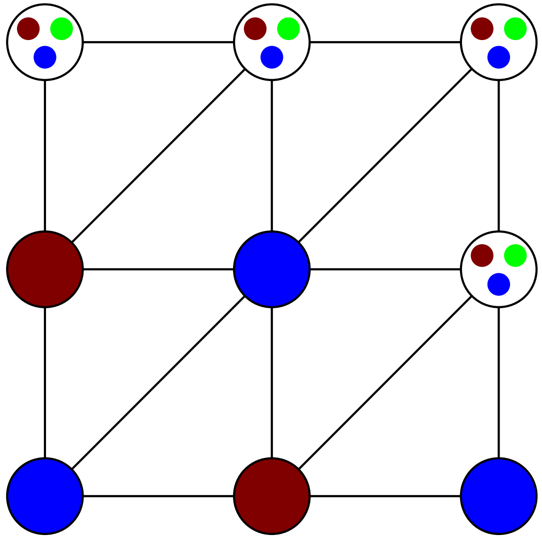
◊ **Nodes 5 and 6 must both be green**

Inference



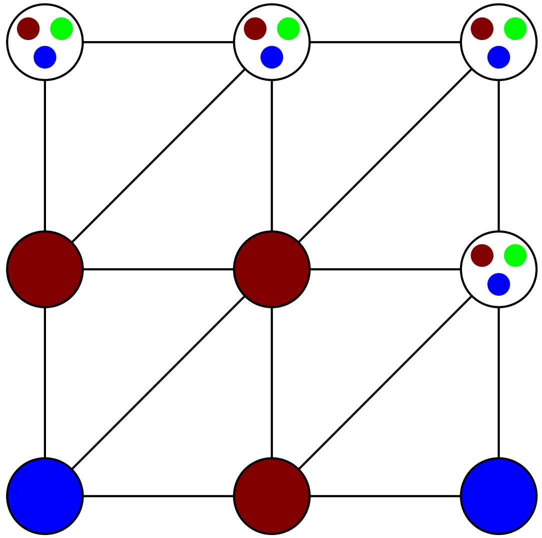
- Assign values from the bottom left corner, going across the rows
- Choose blue first
- **Nodes 5 and 6 must both be green**

Inference



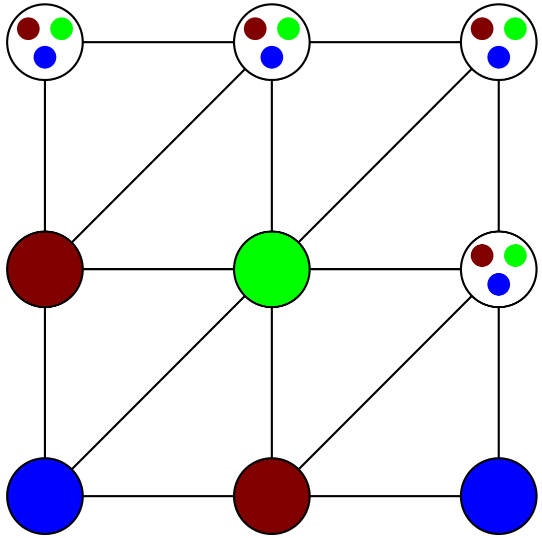
- ♢ Assign values from the bottom left corner, going across the rows
- ♢ Choose blue first
- ♢ **You are wasting your time!**

Inference



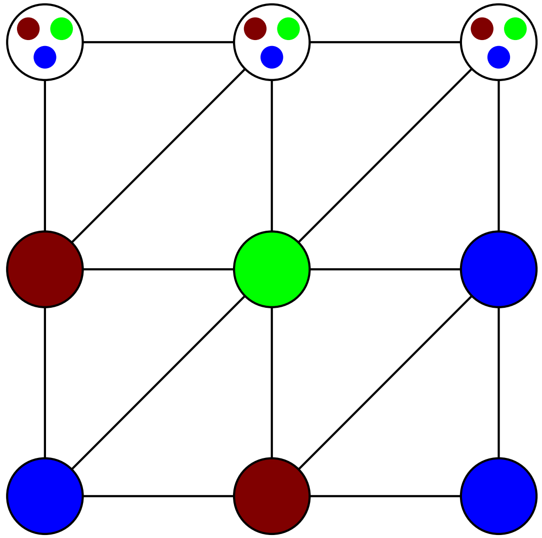
- Assign values from the bottom left corner, going across the rows
- Choose blue first
- **You are wasting your time!**

Inference



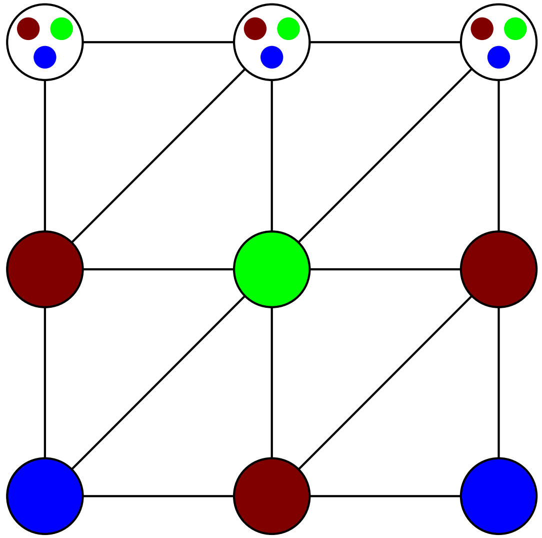
- Assign values from the bottom left corner, going across the rows
- Choose blue first
- **It won't work!!**

Inference



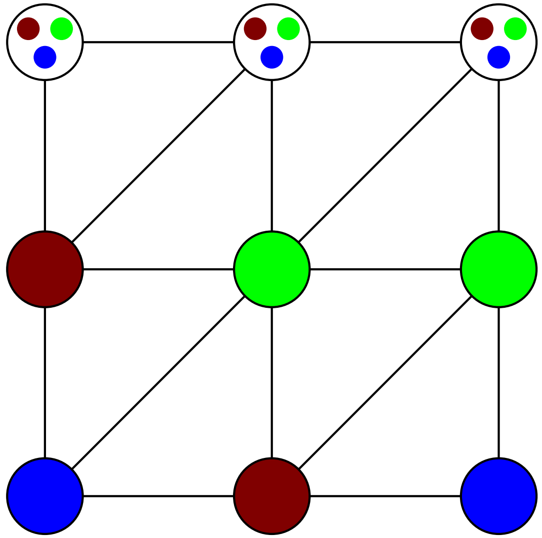
- Assign values from the bottom left corner, going across the rows
- Choose blue first

Inference



- Assign values from the bottom left corner, going across the rows
- Choose blue first

Inference



- Assign values from the bottom left corner, going across the rows
- Choose blue first
- So a way of detecting the problem early could save work.
  So could assigning the green ones before the red one to their left

[intelligent decisions]

Making choices

function BACKTRACK(γ, a) **returns** solution, or "inconsistent"

    if a is inconsistent with γ **then return** "inconsistent"
    if a is total **then return** a
    select some variable v for which a is not defined
    for each d in D_v in some order **do**
        $a' \leftarrow a \cup \{(v, d)\}$
        $a'' \leftarrow$ BACKTRACK(γ, a')
        if $a'' \neq$ "inconsistent" **then return** a''
    end
    return "inconsistent"

call: BACKTRACK(γ, $\{\,\}$)

The size of the search space depends on the order in which we choose variables and values.

Variable ordering

◇ Common strategy: most constrained variable (aka "first-fail")
Choose a variable with the smallest (consistent) domain
Minimise $|\{d \in D_v : a \cup \{(v, d)\} \text{ consistent}\}|$

◇ Minimises branching factor (at the current node) // Work well in most cases

◇ Extreme case: select variables with unique possible values first — of course not all of them.
— Value is forced by the existing assignment
— Obviously should be done in all cases
— Compare unit propagation in SAT solving

Other variable ordering strategies

diff?

◇ Most constraining variable
Involved in as many constraints as possible
Maximise $|\{u \in V : a(u) \text{ undefined}, C_{u,v} \in C\}|$

[do most to restrict domains of other variables]

◇ Seek biggest effect on domains of unassigned variables
Detect inconsistencies earlier, shortening search tree branches

◇ Others include history-dependent strategies
— e.g. involved in a lot of (recent) conflicts
— or selected many/few times before

◇ Random selection can also help, especially for tie-breaking

recall your assignment 1 heuristics.

Value ordering

- Common strategy: least constraining value
  Choose a value that won't conflict much with others
  Minimise $|\{\{d' \in D_u : a(u)$ undefined$, C_{u,v} \in C, (d,d') \notin C_{u,v}\}|$

- Minimise useless backtracking below current node

- If no solutions, or if we want all solutions, value ordering doesn't matter: we have to go over the whole sub-tree anyway.

- If there is a solution, we may be lucky and find it without backtracking on this value choice

cf.

More about inference

usually do not record them in graphs ↓

rather we do them in a way that we can easily backtrack with them

◇ Inference in CSP solving: deducing additional constraints that follow from the already known constraints.

◇ Hence a matter of replacing γ by an equivalent and strictly tighter constraint network γ'.

"logically"

◇ γ and γ' with the same variables are equivalent iff they have the same solutions.

◇ $\gamma' = (V, D', C')$ is tighter than $\gamma = (V, D, C)$ iff:

(i) For all $v \in V$, $D'_v \subseteq D_v$

(ii) For all $C_{u,v} \in C$, $C'_{u,v} \subseteq C_{u,v}$

γ' is strictly tighter than γ if it is tighter and $\gamma \neq \gamma'$

◇ Inference reduces the number of consistent partial assignments

[without ↓ the set of solutions]

How to use inference

Inference as **offline pre-processing**

◇ Just once before search starts

◇ Little runtime overhead, modest pruning power. Not considered here.
— but important in SAT solving, for instance

Inference **during search**

◇ At every recursive call of backtracking

◇ When backing up out of a search branch, retract any inferred constraints that were local to that branch because they depend on a

◇ Strong pruning power. May have large runtime overhead / drawback.

Trade: Time Length vs. Depth of Pruning.

Backtracking with inference

```
function BACKTRACK(γ, a) returns solution, or "inconsistent"
    if a is inconsistent with γ then return "inconsistent"
    if a is total then return a
    γ' ← a copy of γ
    γ' ← Inference(γ, a)
    if exists v with D'_v = { } then return "inconsistent"
    select variable v for which a is not defined
    for each d in D_v do
        a' ← a ∪ {(v, d)}
        a'' ← BACKTRACK(γ, a')
        if a'' ≠ "inconsistent" then return a''
    end
    return "inconsistent"
```

copy constraint network

infer on copy [keep original]

search with copy [below this node]

back & discard this copy

◇ Inference sets $D_v = \{d\}$ for each $(v, d) \in a$ and then delivers a tighter equivalent network.

Forward Checking

◇ Inference: for all variables v and u where $a(v) = d$ is defined and $a(u)$ is undefined, set D_u to $\{d' : d' \in D_u, (d', d) \in C_{u,v}\}$.

◇ That is, remove from domains any value not consistent with those that have been assigned.

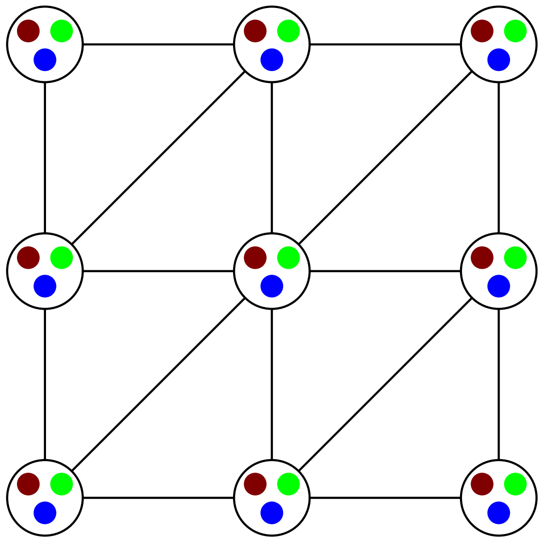
◇ Obviously sound: it does not rule out any solutions [if the value we assign is in the solution]

◇ Can be implemented incrementally for efficiency: only necessary to consider v to be the variable which has just been assigned.

◇ Simple to implement and low computational cost

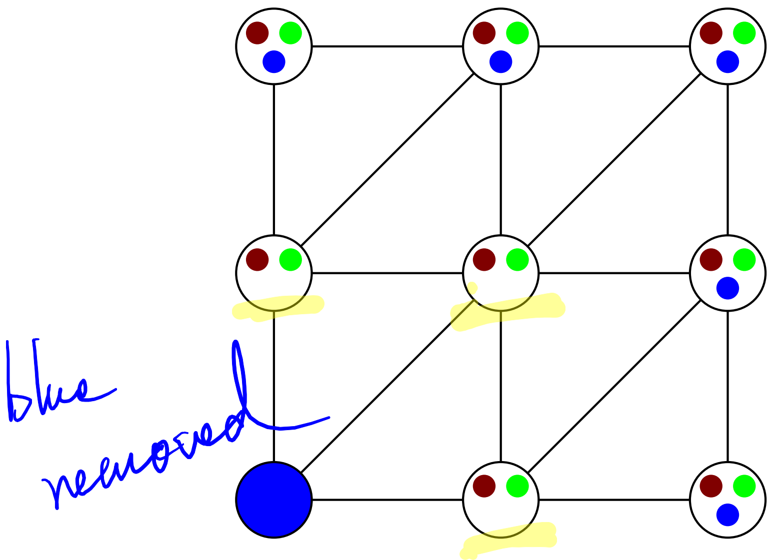
◇ Almost always pays off (unless subsumed by stronger inferences)

Forward Checking example



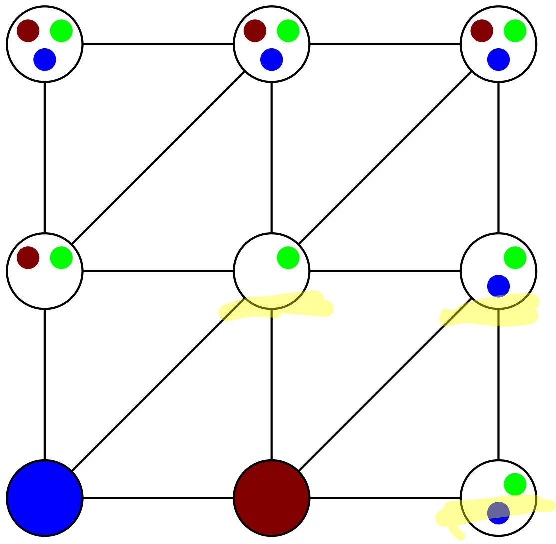
♢ As before, start in the bottom left corner and go across the rows

Forward Checking example



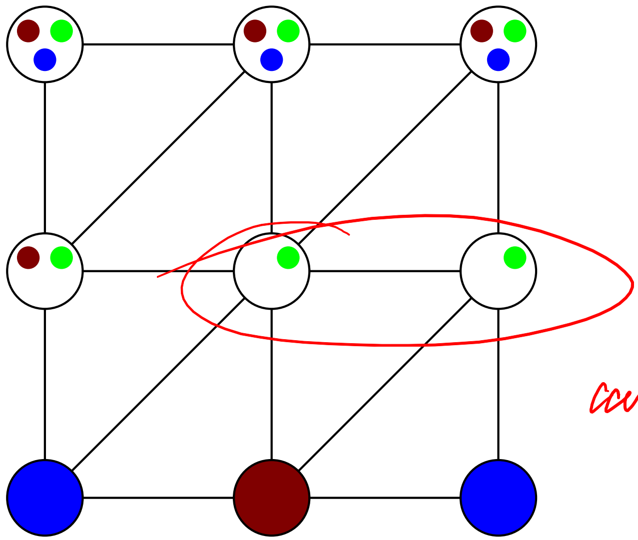
◊ Impossible values get removed from related domains

Forward Checking example



◇ So no inconsistent assignment actually gets reached

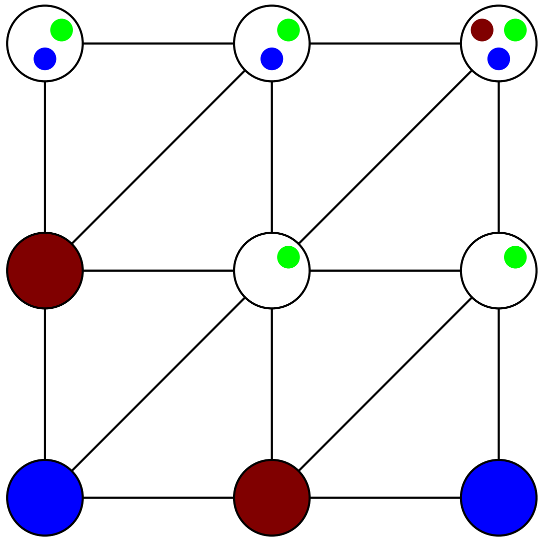
Forward Checking example



can't see here [2 steps far]

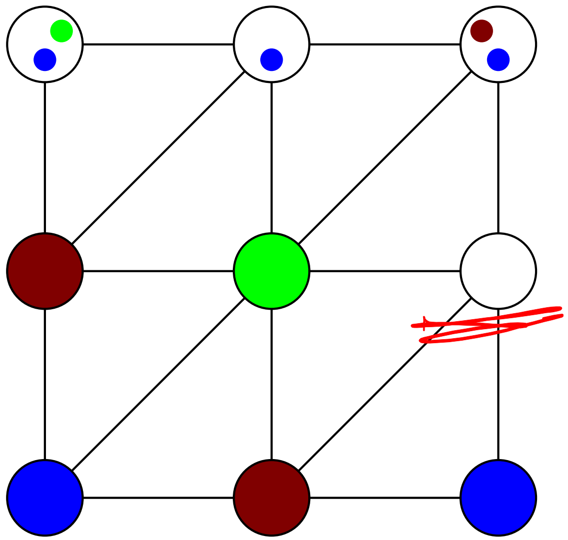
◇ We still don't make two-step inferences

Forward Checking example



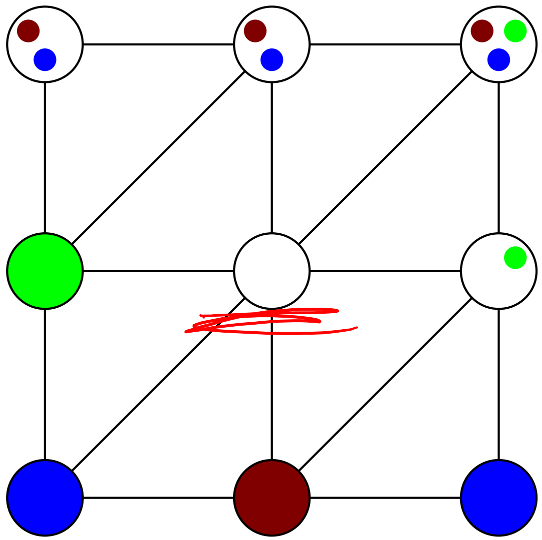
♢ We still don't make two-step inferences

Forward Checking example



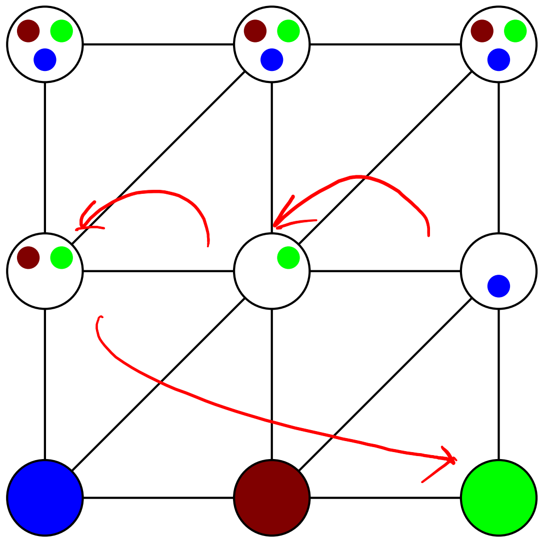
♢ Now there is a wipeout: a variable with an empty domain

Forward Checking example



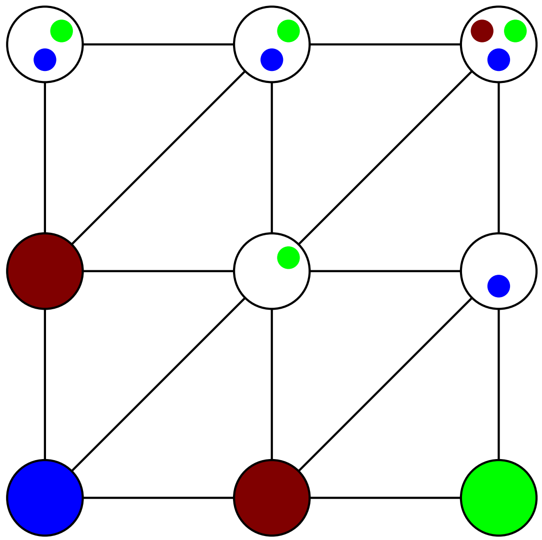
♢ Backtrack and change – but now there is another wipeout

Forward Checking example



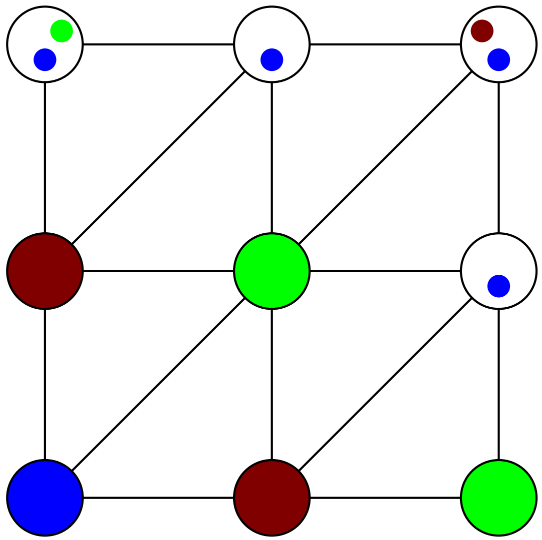
◇ So backtrack some more

Forward Checking example



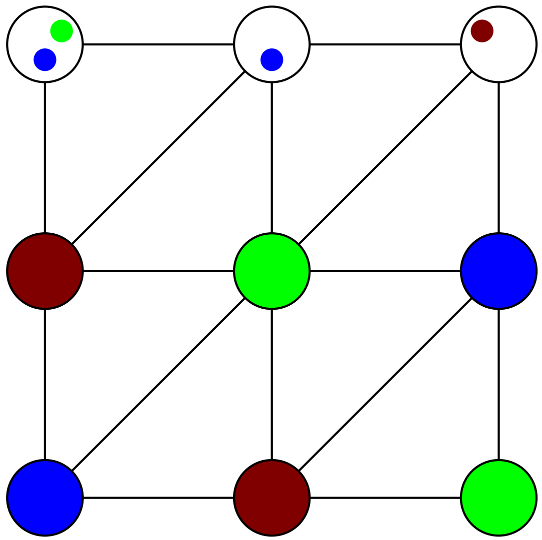
◊ Continue to explore the branch

Forward Checking example



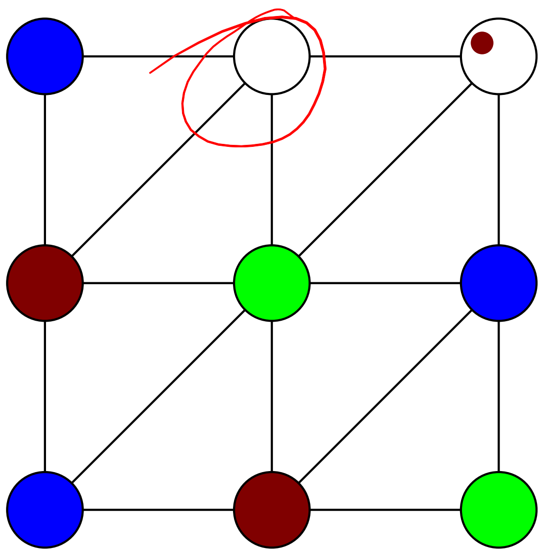
◊ Now some moves are forced

Forward Checking example



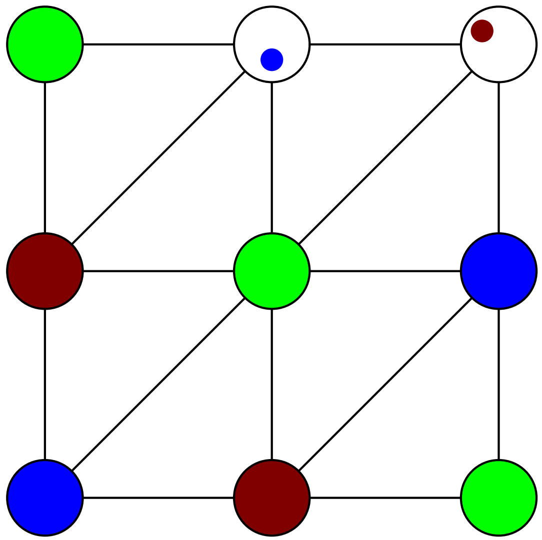
◊ Now some moves are forced, and still consistent

Forward Checking example



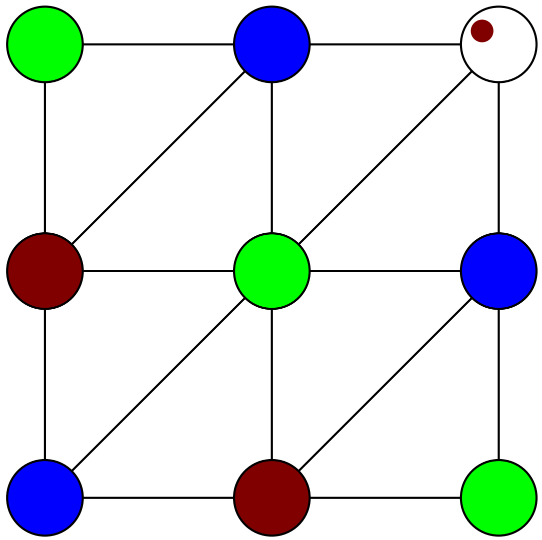
◊ Blue is the bad choice

Forward Checking example



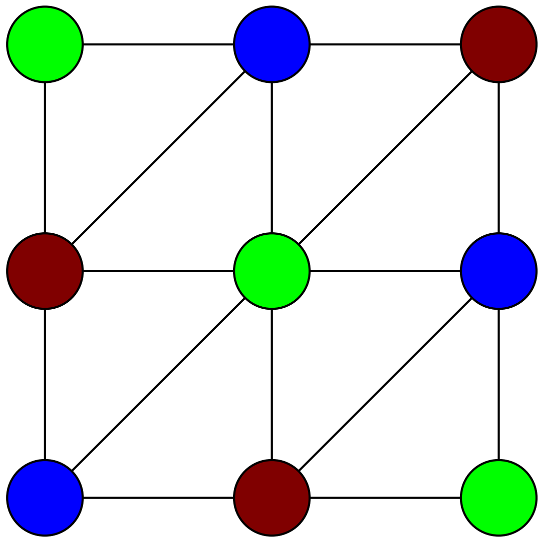
◊ And ...

Forward Checking example



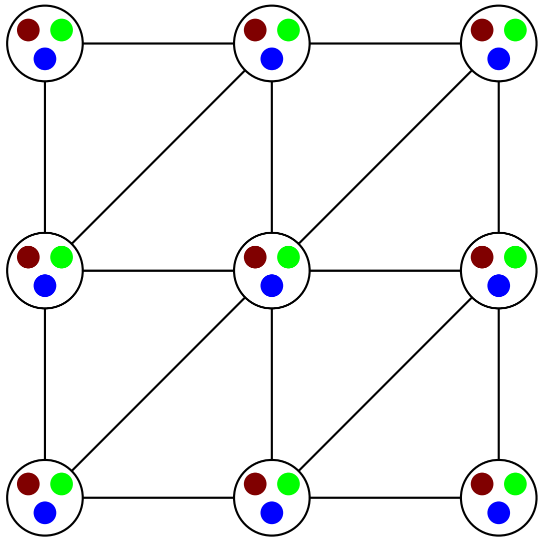
♢ And we're . . .

Forward Checking example



♢ And we're done!

Forward Checking plus First-Fail



variable to be assigned next should be the one which is "most likely" to lead to a dead-end.

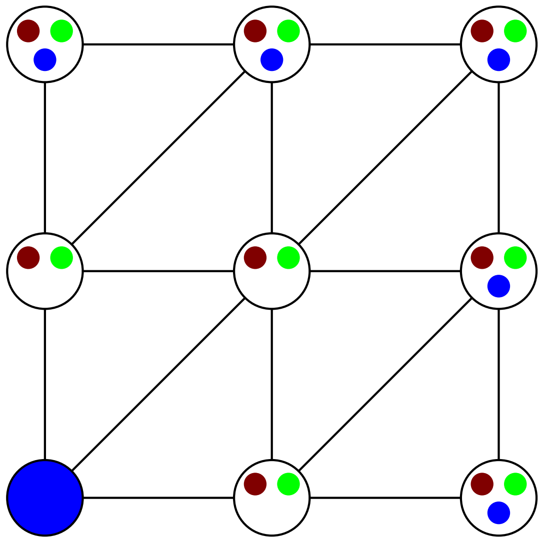
- Forward checking is rather weak on its own, but it combines well with the first-fail heuristic for variable ordering, to make a powerful technique.

- Unit propagation (selecting variables with singleton domains) is particularly important when forward checking is used.

this is one way to operationalize it.

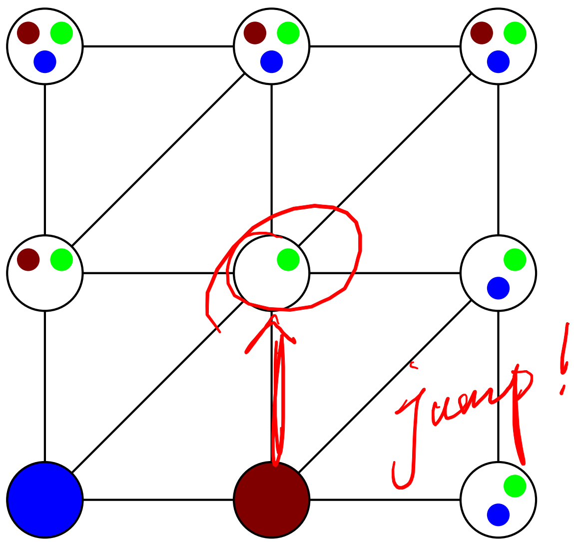
if so, assign them next

Forward Checking plus First-Fail



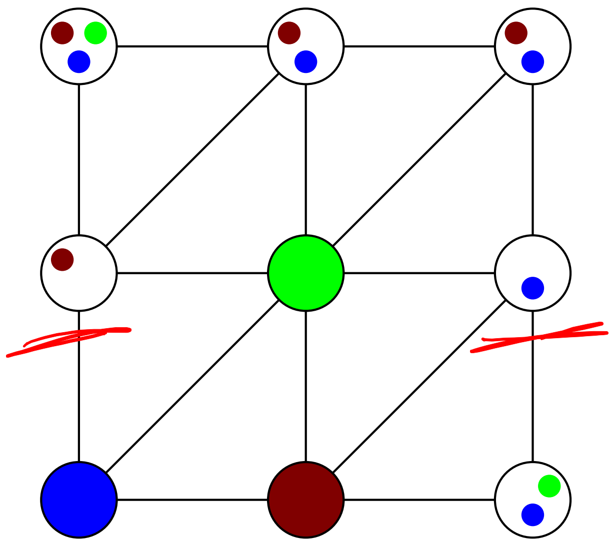
◊ Impossible values get removed from related domains

Forward Checking plus First-Fail



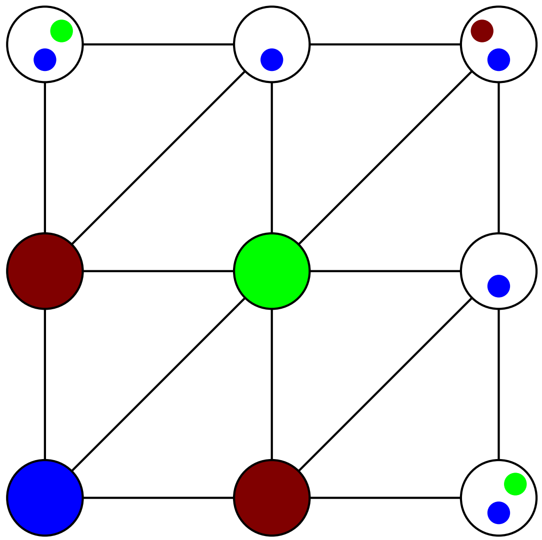
◇ Note that there is only one value in the middle

Forward Checking plus First-Fail



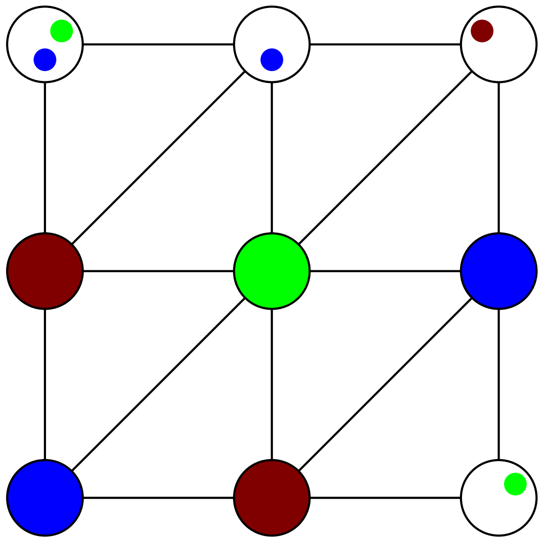
- Colour that one green, as it has the smallest domain
- More domains are reduced to singletons

Forward Checking plus First-Fail



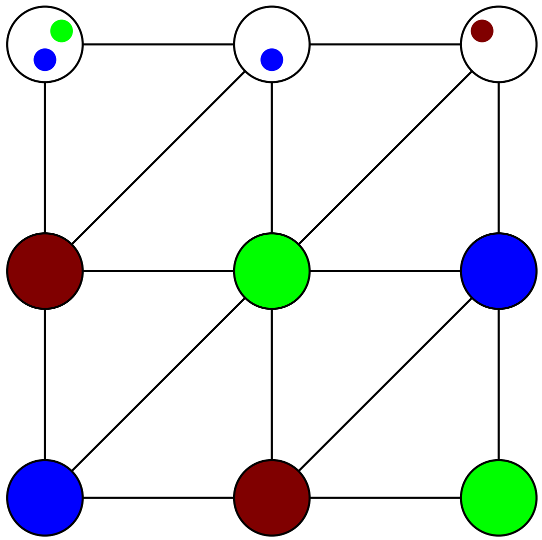
◊ Forced move

Forward Checking plus First-Fail



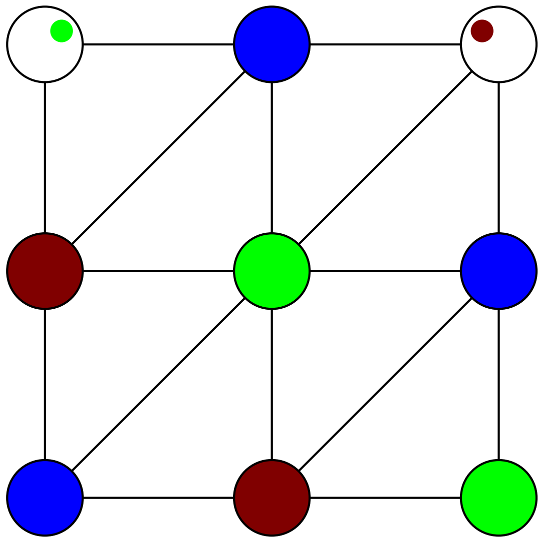
◇ Forced move

Forward Checking plus First-Fail



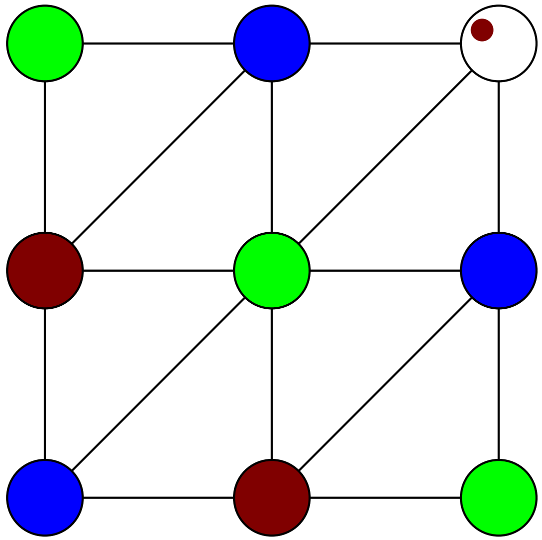
◊ Forced move

Forward Checking plus First-Fail



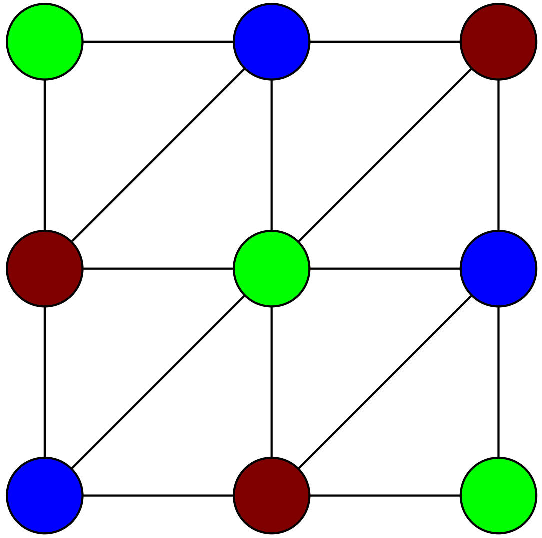
♢ Forced move

Forward Checking plus First-Fail



◇ Forced move

Forward Checking plus First-Fail



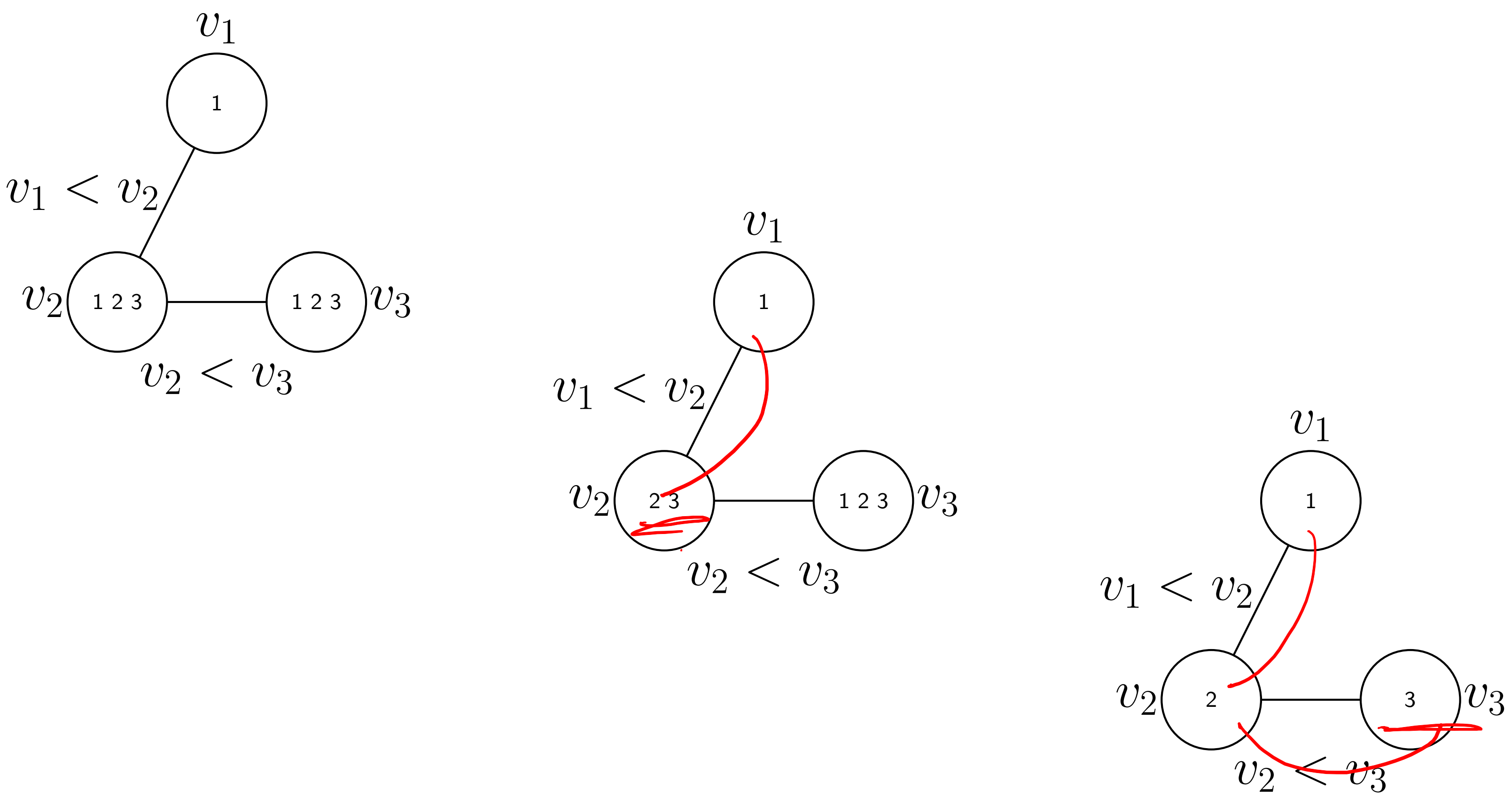
♢ Search was backtrack-free!

Arc Consistency

FC is AC only within variables that with values assigned

- A stronger inference rule: make all variables arc consistent
- Variable v is arc consistent with respect to another variable u iff for every $d \in D_v$ there is at least one $d' \in D_u$ such that $(d, d') \in C_{v,u}$. A CSP $\gamma = (V, D, C)$ is said to be arc consistent (AC) iff every variable in V is arc consistent with every other. / "as if they are already assigned"
- Any $d \in D_v$ which has no support in D_u is incapable of being assigned to v in any solution, so it can be removed from D_v.
- Removing all unsupported variables makes γ AC. This is clearly a valid constraint inference, as no solutions are lost.
- Enforcing AC subsumes both forward checking and unit propagation.

Arc Consistency: example



Arc consistency: AC-3

function REVISE(γ, u, v) **returns** modified γ

  for each $d \in D_u$ **do**
    if there is no $d' \in D_v$ with $(d, d') \in D_v$ **then**
      $D_u \leftarrow D_u \setminus \{d\}$
  end
  return γ

function AC-3(γ) **returns** modified γ

  $M \leftarrow \{(u,v),(v,u) : C_{u,v} \in C\}$
  while $M \neq \{\}$ **do**
    remove some element (u, v) from M
    REVISE(γ, u, v)
    if D_u has changed **then**
      $M \leftarrow M \cup \{(w,u) : C_{w,u} \in C, w \neq v\}$
  end
  return γ

Arc consistency: notes

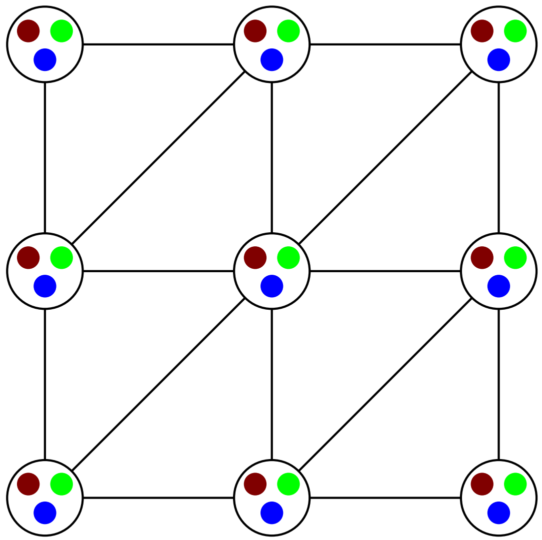
◇ At every iteration, all arcs not in the queue M are consistent

◇ On termination, the network is AC / always

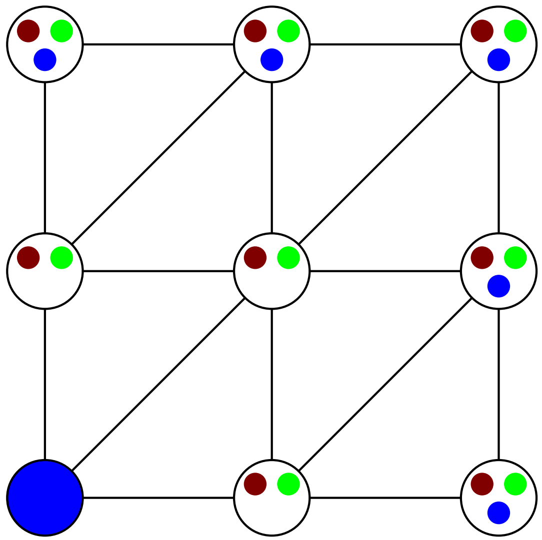
◇ Unlike forward checking, makes inferences from unassigned variables

◇ Arc consistency is widely used in modern CSP solvers

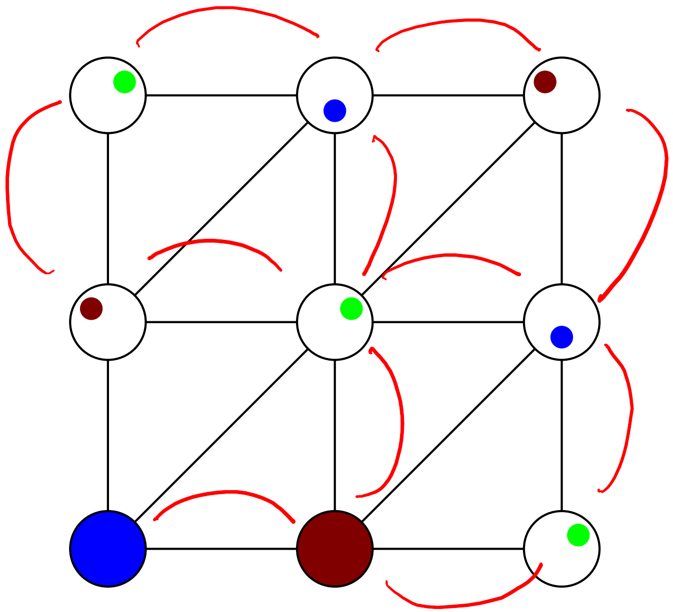
Arc Consistency



Arc Consistency



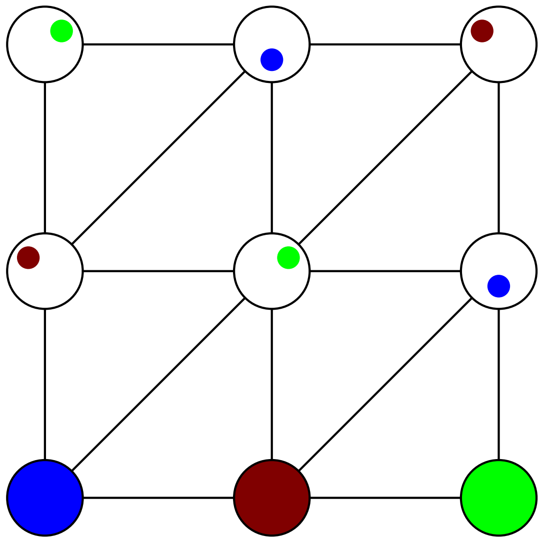
Arc Consistency



◊ Already done: since this is AC, the only possible assignment must be a solution.

◊ Now it's just a matter of filling in the values

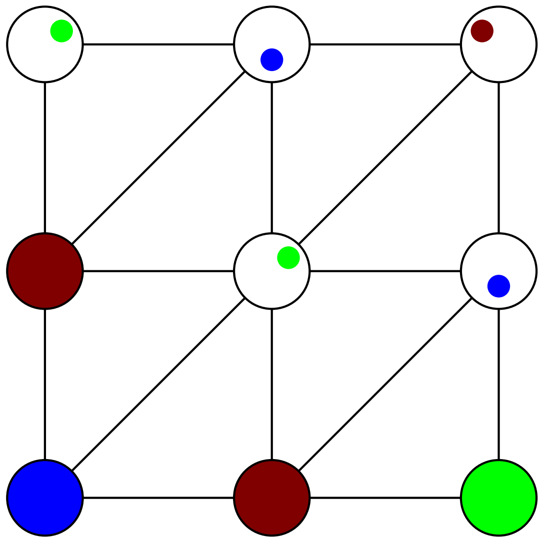
Arc Consistency



♢ Already done: since this is AC, the only possible assignment must be a solution.

♢ Now it's just a matter of filling in the values

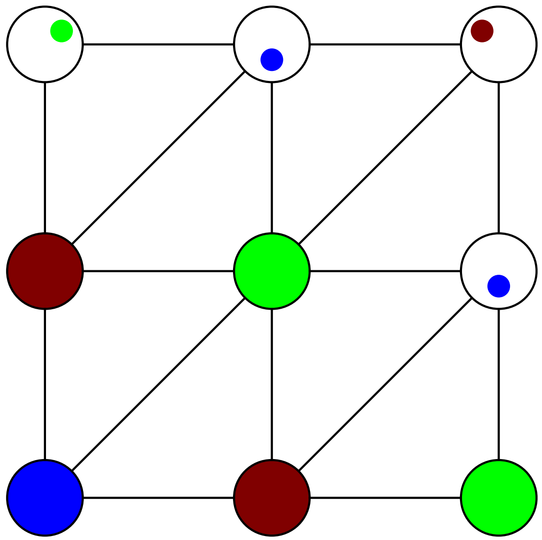
Arc Consistency



◇ Already done: since this is AC, the only possible assignment must be a solution.

◇ Now it's just a matter of filling in the values

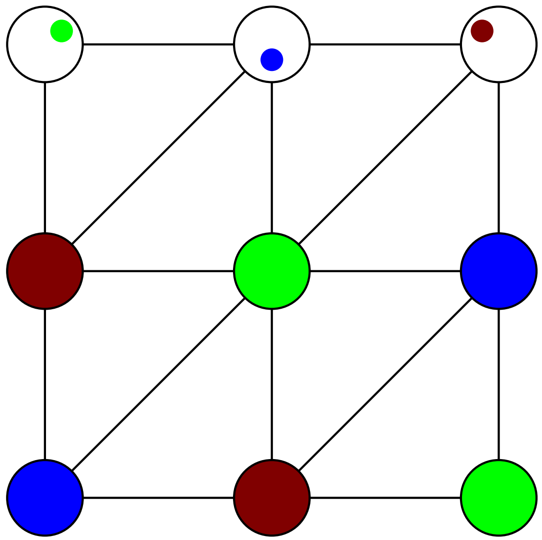
Arc Consistency



◇ Already done: since this is AC, the only possible assignment must be a solution.

◇ Now it's just a matter of filling in the values

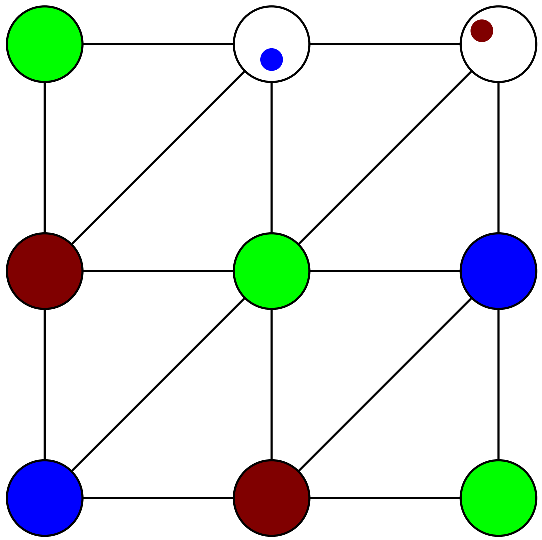
Arc Consistency



♢ Already done: since this is AC, the only possible assignment must be a solution.

♢ Now it's just a matter of filling in the values

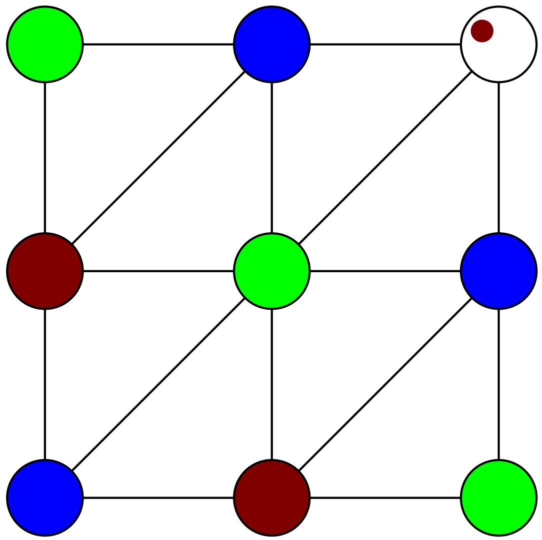
Arc Consistency



◇ Already done: since this is AC, the only possible assignment must be a solution.

◇ Now it's just a matter of filling in the values

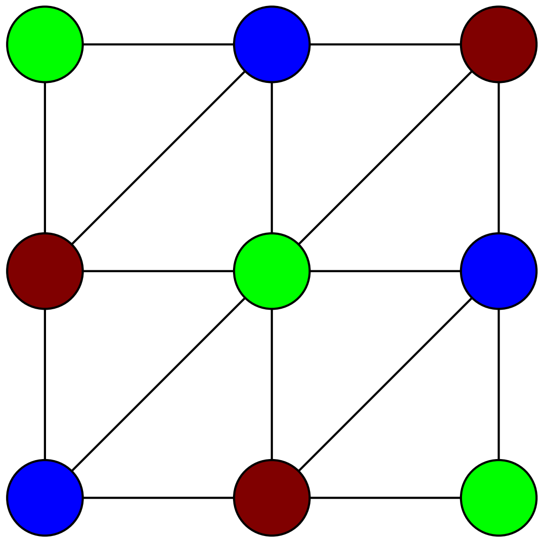
Arc Consistency



◇ Already done: since this is AC, the only possible assignment must be a solution.

◇ Now it's just a matter of filling in the values

Arc Consistency



◇ Search was backtrack-free—and all over at step 2

Summary

◊ **Variable orderings** in backtracking can dramatically reduce the size of the search tree. **Value orderings** don't, but they may lead to solutions earlier.

/ depend on your heuristics

◊ **Inference** tightens γ without losing equivalence, during backtracking. This reduces the amount of search needed. The benefit in reduced tree size must be traded off against the time cost of the reasoning.

space ↓

◊ **Forward checking** removes values conflicting with an assignment already made

⇑ stronger.

◊ **Arc consistency** extends this to all variables, whether assigned or not. It is stronger than forward checking and unit propagation, but costs more to compute.