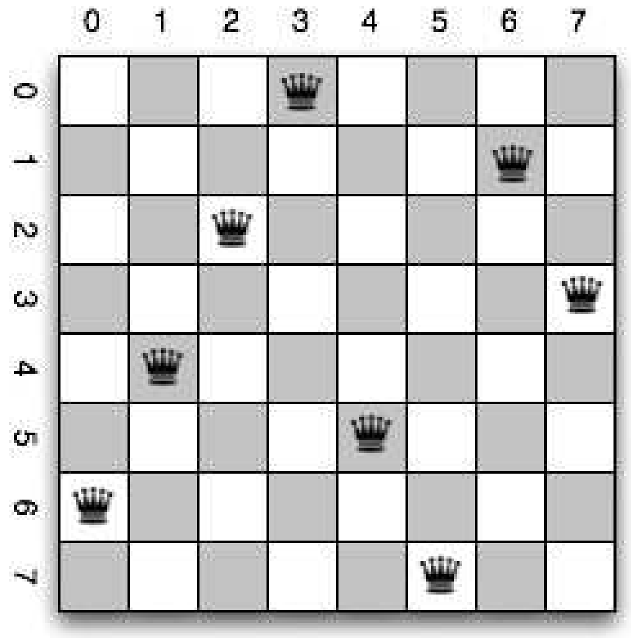
Knowledge Representation and Reasoning: Solving Constraint Satisfaction Problems

Chapter 6.2, 6.3

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

◇ Constraint modelling

◇ Inference

◇ Forward checking

◇ Variable and value ordering

◇ Arc consistency

Constraint Modelling

- Before any constraint solving can happen, the CSP must be defined
- Model must define V, D and C
- Explicit definition of C is painful, so use high-level description
- Hence we want to write a logical models
  — Write constraints as formulae of first order logic
  — Describe what would count as a solution to the problem
  — Compiler will turn this into a low-level constraint network
  — **Logical model is purely declarative: no algorithm!**
- Old style constraint programming: logic is implicit in the program

MiniZinc

We shall use the constraint modelling language MiniZinc

```
-----------------------------------------------
% N Queens Problem: place N queens on an NxN
% chessboard so that no queen attacks another

int: N;
array[1..N] of var 1..N: q;
constraint forall (x,y in 1..N where x < y) (
           q[x] != q[y] /\
           abs(q[x]-q[y]) != y-x);
solve satisfy;
-----------------------------------------------
```

MiniZinc is essentially first order logic with some syntactic sugar and basic support for arithmetic etc.

Minizinc

◇ MiniZinc model is completely solver-independent

◇ Also good to separate the "conceptual model" of the problem from data defining a specific problem instance
— e.g. for the N queens, the data file could specify `N = 8;`

◇ MiniZinc gets transformed into a simple fragment "Flat Zinc"

◇ Flat Zinc can be turned into input code for many solvers
— Finite domain (FD) solvers
— Mixed integer programming (MIP) solvers
— SAT solvers
— Local search solvers

◇ Mapping into low-level data structures is internal to the solvers

◇ Logical model + default mapping: the Holy Grail

Recall Backtracking

function $\textsc{Backtrack}(\gamma, a)$ **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**
$a' \leftarrow a \ \cup \ \{(v, d)\}$
$a'' \leftarrow \textsc{Backtrack}(\gamma,\ a')$
if $a'' \ \neq$ "inconsistent" **then return** a''
end
return "inconsistent"

call: $\textsc{Backtrack}(\gamma,\ \{\,\})$

Backtracking: the Good and the Bad

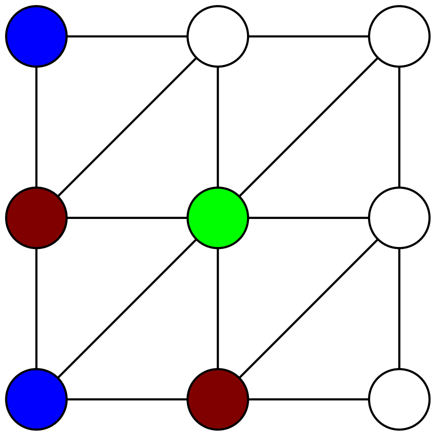
◇ Better that exhaustive search: avoids enumerating many inconsistent (partial) assignments by detecting them as soon as they happen

◇ 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)

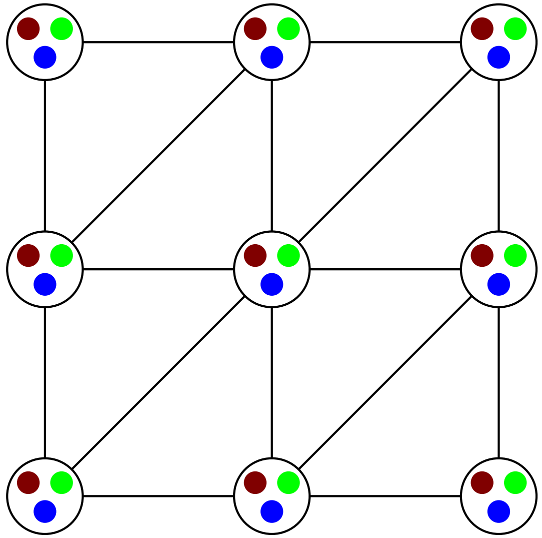
◇ **Disadvantages:**
  Does no reasoning except detecting actual inconsistency
  Cannot look further ahead than the current state

Simple example: Graph colouring



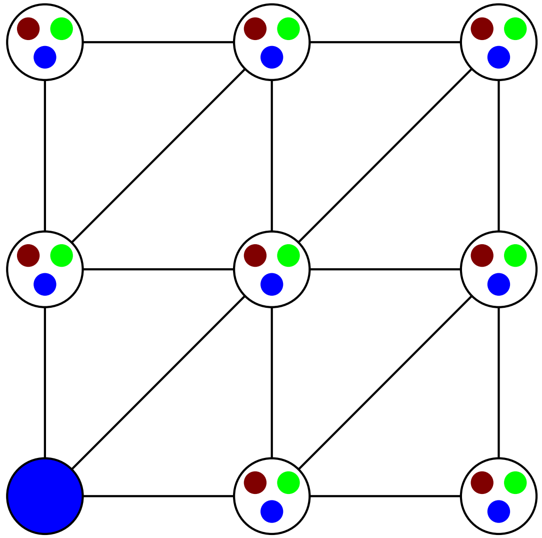
- Given an undirected graph with n nodes, given k colours, assign a colour to each node so that no two adjacent nodes (with an edge between them) are the same colour.
- Representation using binary constraints is easy.
- Problem is NP-complete, so difficult in the worst case.

Inference



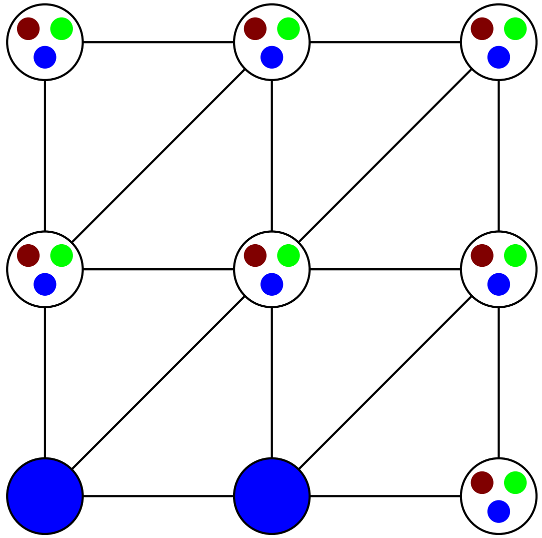
◇ Assign values from the bottom left corner, going across the rows

Inference



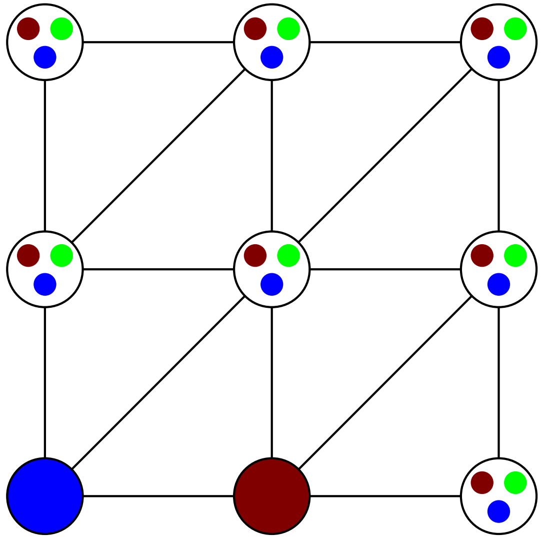
- Assign values from the bottom left corner, going across the rows
- Choose blue first

Inference



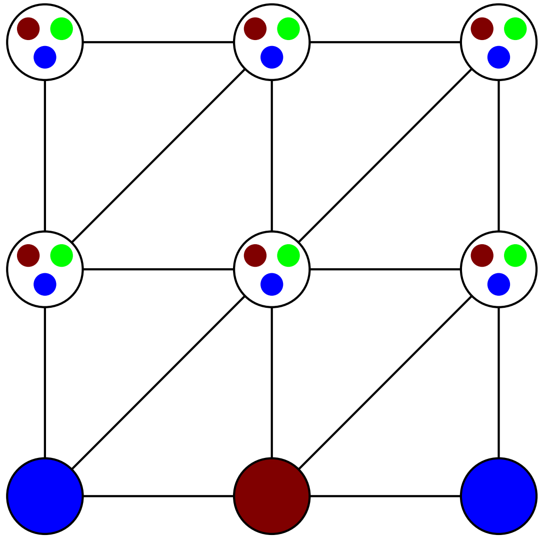
- Assign values from the bottom left corner, going across the rows
- Choose blue first
- Inconsistent

Inference



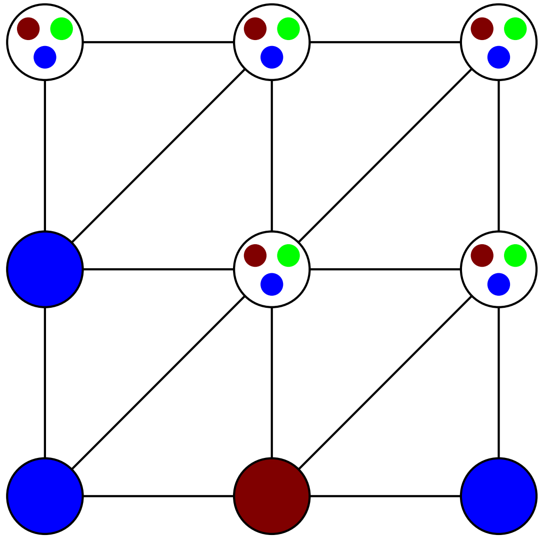
- Assign values from the bottom left corner, going across the rows
- Choose blue first
- Choose red next

Inference



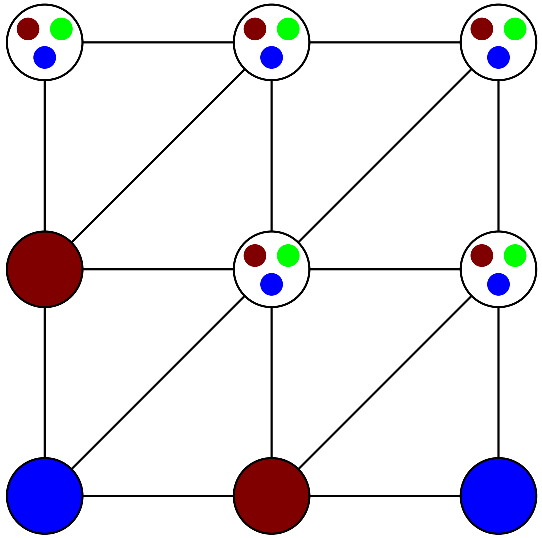
- ♢ Assign values from the bottom left corner, going across the rows
- ♢ Choose blue first, then red
- ♢ Now nodes 5 and 6 must both be green

Inference



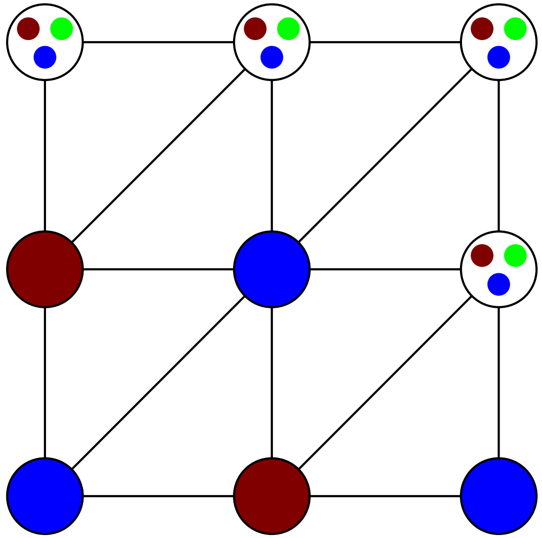
- ♢ Assign values from the bottom left corner, going across the rows
- ♢ Choose blue first, then red
- ♢ **Nodes 5 and 6 must both be green**

Inference



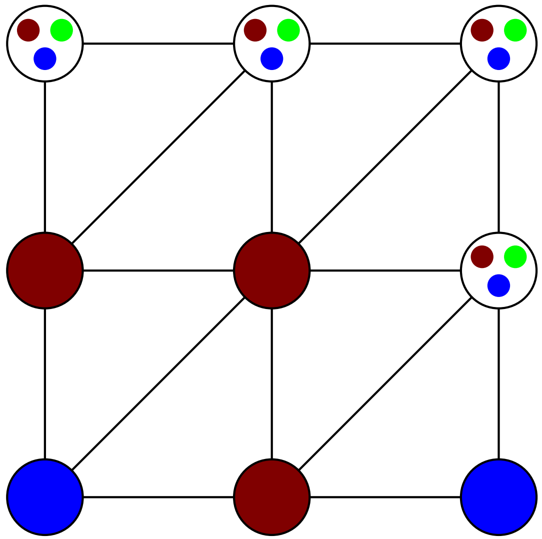
- Assign values from the bottom left corner, going across the rows
- Choose blue first, then red
- **Nodes 5 and 6 must both be green**

Inference



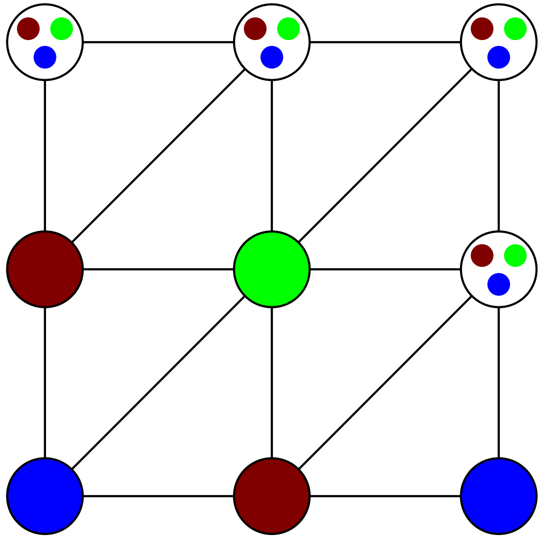
- Assign values from the bottom left corner, going across the rows
- Choose blue first, trhen red
- **You are wasting your time!**

Inference



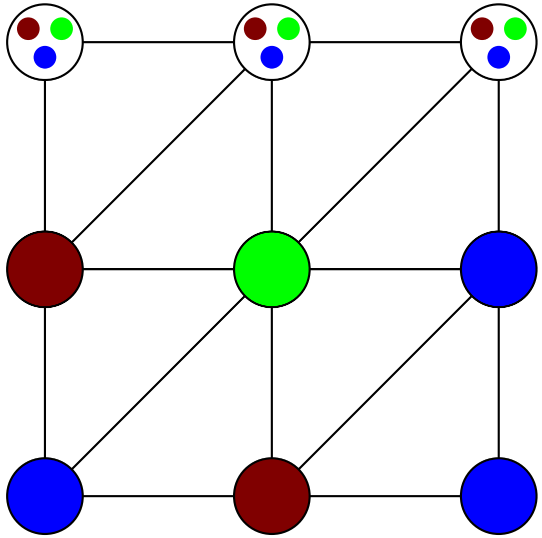
- Assign values from the bottom left corner, going across the rows
- Choose blue first, then red
- **You are wasting your time!**

Inference



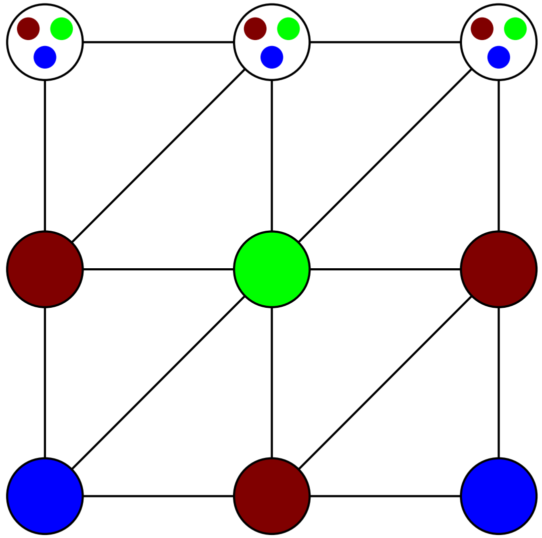
- Assign values from the bottom left corner, going across the rows
- Choose blue first, then red, then green
- **It won't work!!**

Inference



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

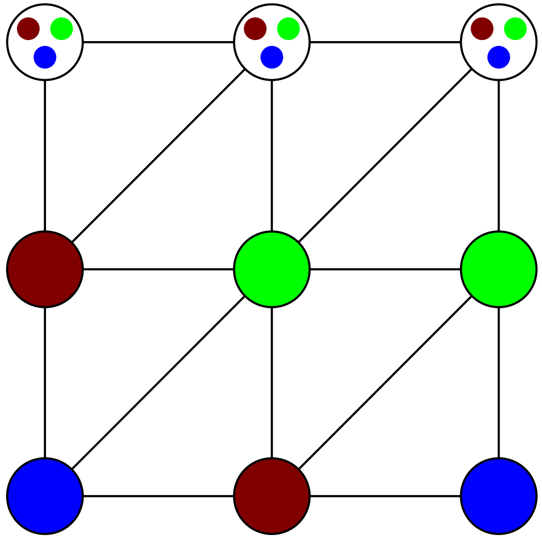
Inference



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

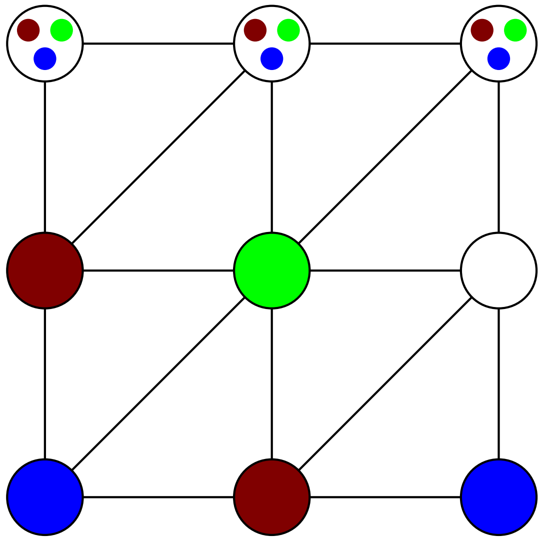
◇ Choose blue first, then red, then green

Inference



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

Inference



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

More about inference

♢ 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'.

♢ 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

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

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"
```

♢ 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.

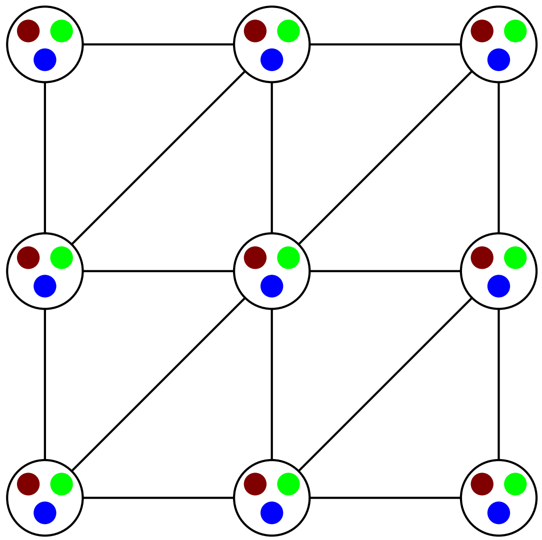
♢ Obviously sound: it does not rule out any solutions

♢ 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

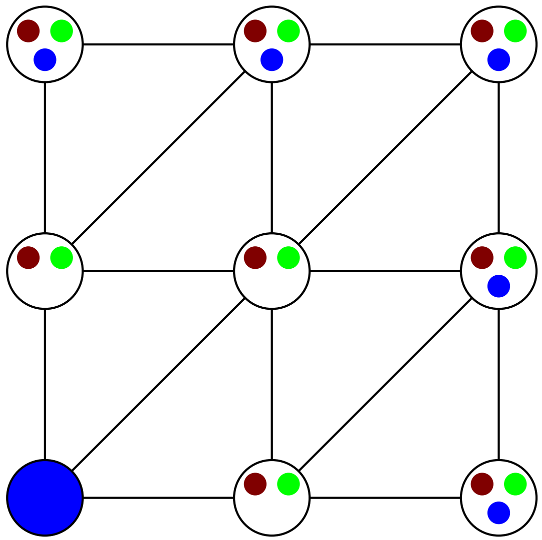
♢ Almost always pays off (unless subsumed by stronger inferences)

Forward Checking example



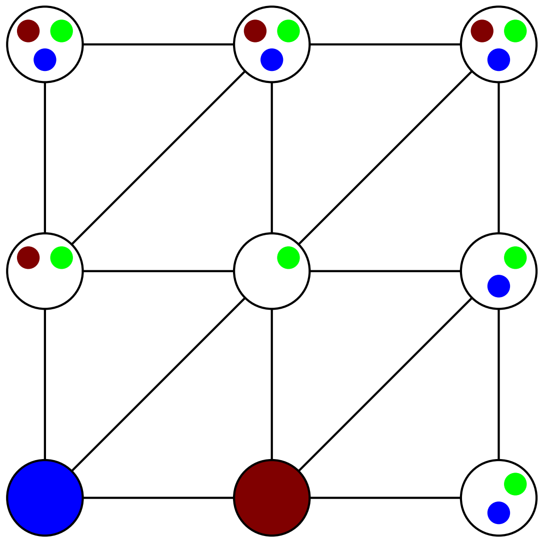
◇ As before, start in the bottom left corner and go across the rows

Forward Checking example



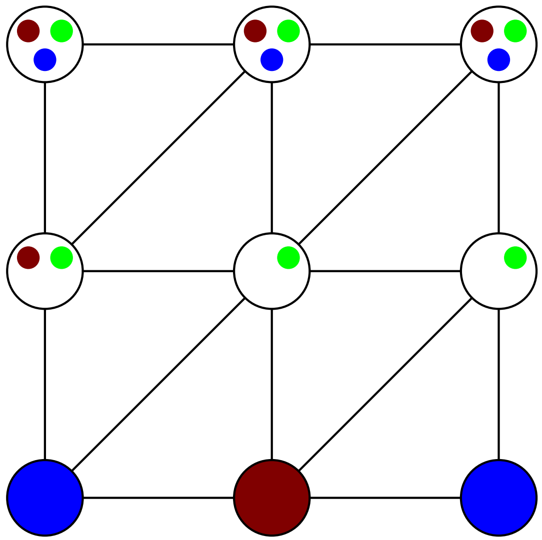
♢ Impossible values get removed from related domains

Forward Checking example



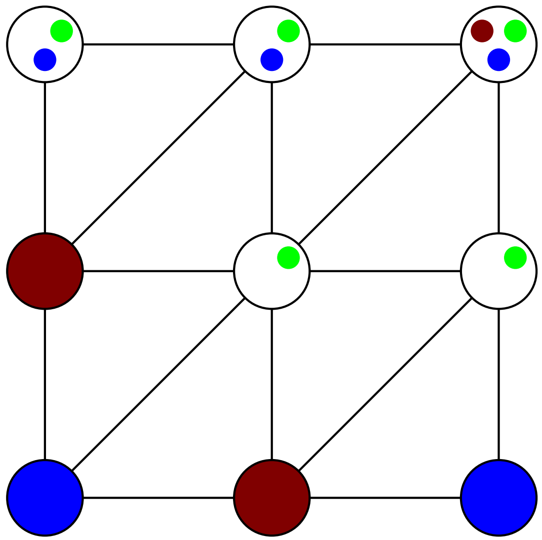
♢ So no inconsistent assignment actually gets reached

Forward Checking example



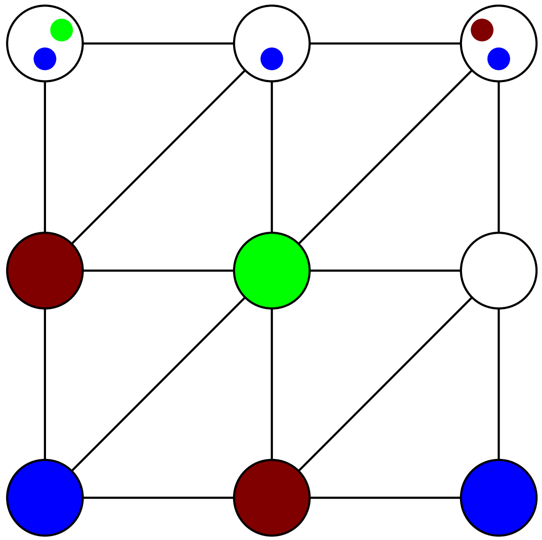
♢ We still don't make two-step inferences

Forward Checking example



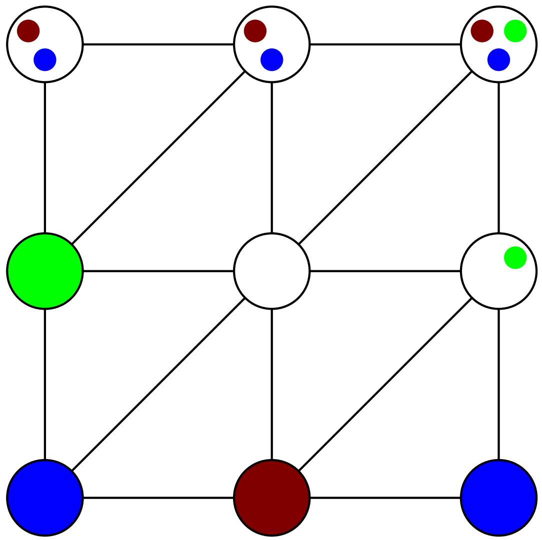
♢ We still don't make two-step inferences

Forward Checking example



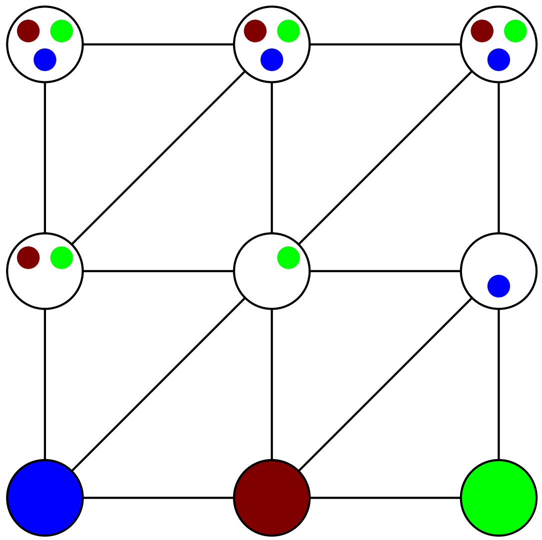
◊ Now there is a wipeout: a variable with an empty domain

Forward Checking example



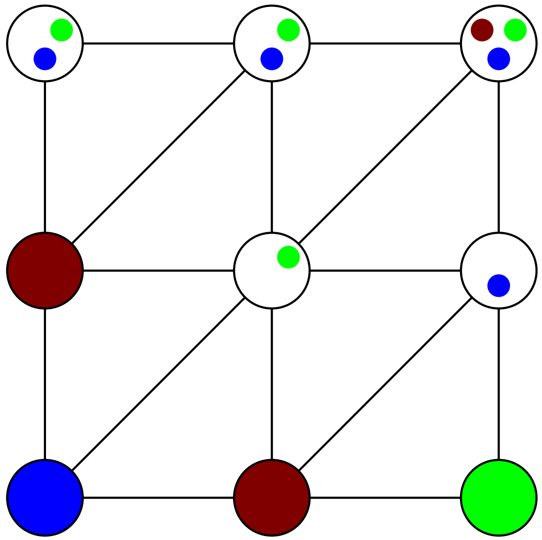
♢ Backtrack and change – but now there is another wipeout

Forward Checking example



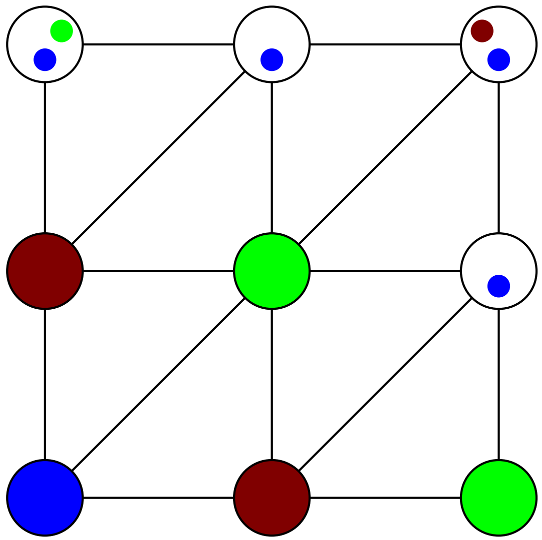
- ♢ So backtrack some more

Forward Checking example



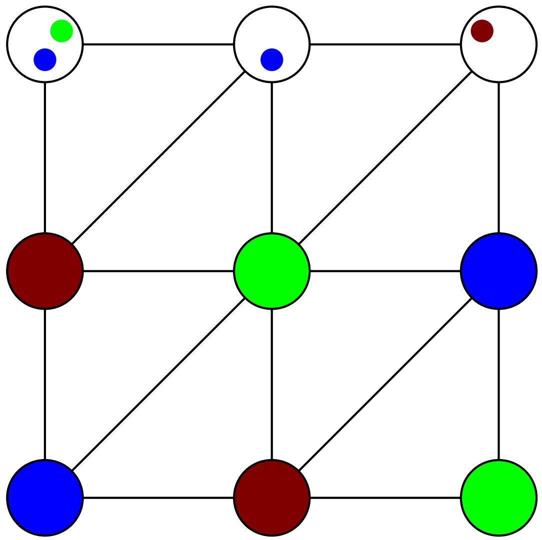
◇ Continue to explore the branch

Forward Checking example



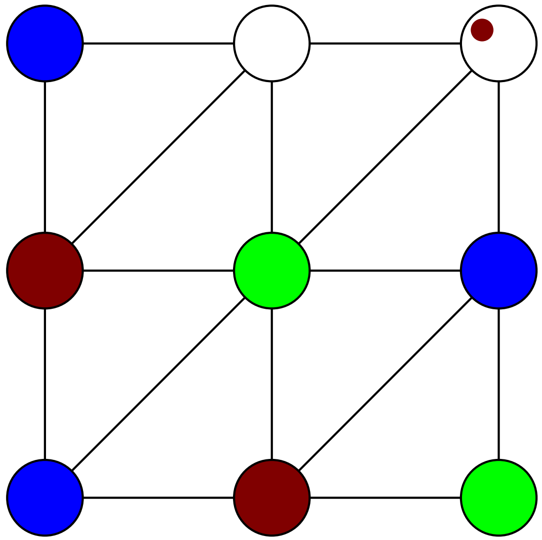
◇ Now some moves are forced

Forward Checking example



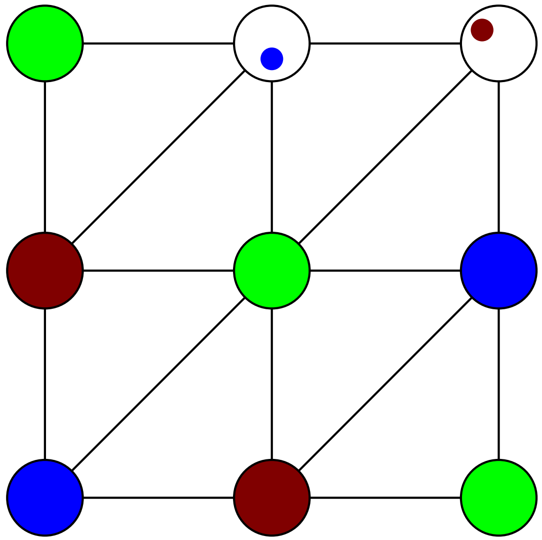
♢ Now some moves are forced, and still consistent

Forward Checking example



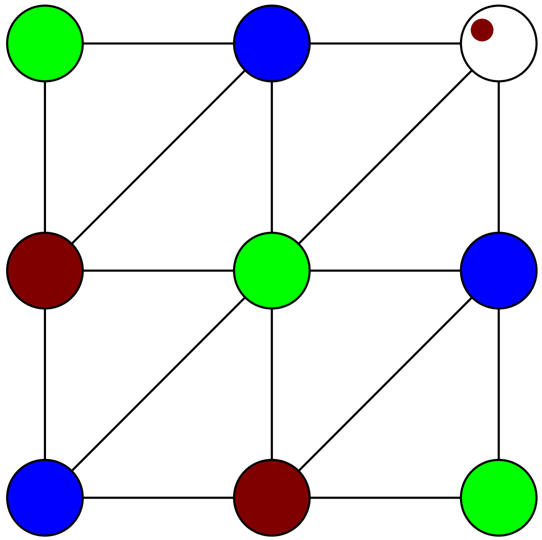
◇ Blue is the bad choice

Forward Checking example



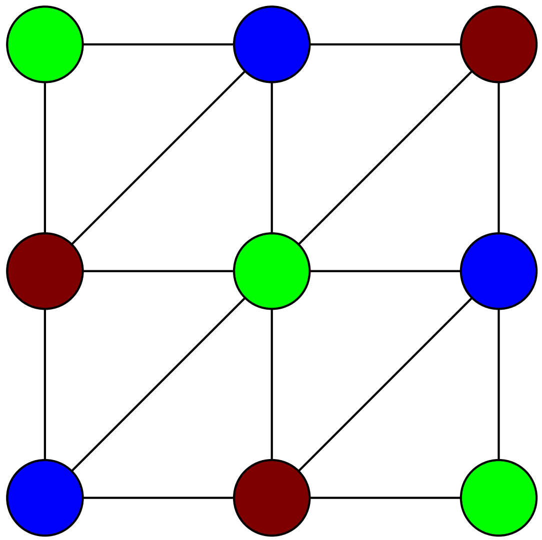
◇ And . . .

Forward Checking example



◇ And we're . . .

Forward Checking example



◇ And we're done!

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' ← a ∪ {(v, d)}
        a'' ← BACKTRACK(γ, a')
        if a'' ≠ "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 $|\{d \in D_v : a \cup \{(v, d)\}$ consistent$\}|$

◇ Minimises branching factor (at the current node)

◇ Extreme case: select variables with unique possible values first
— Value is forced by the existing assignment
— Obviously should be done in all cases
— Compare unit propagation in SAT solving

Other variable ordering strategies

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

- 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

Value ordering

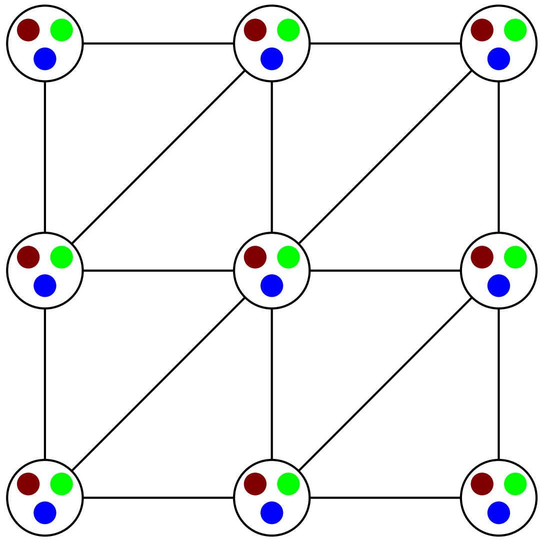
◊ Common strategy: least constraining value
Choose a value that won't conflict much with others
Minimise $|\{\{d' \in D_u : a(u)$ undefined$, C_{u,v} \in C, (d, d') \notin C_{u,v}\}|$

◊ 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

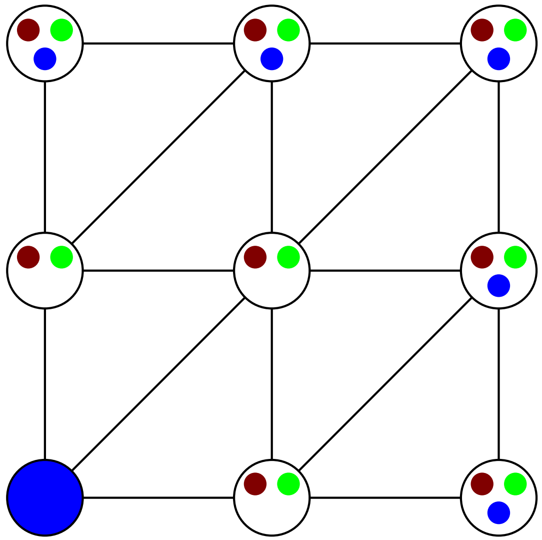
Forward Checking plus First-Fail



◇ 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.

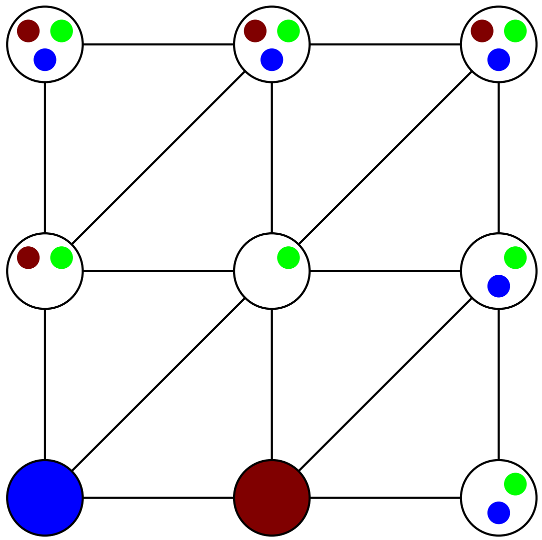
◇ Unit propagation (selecting variables with singleton domains) is particularly important when forward checking is used.

Forward Checking plus First-Fail



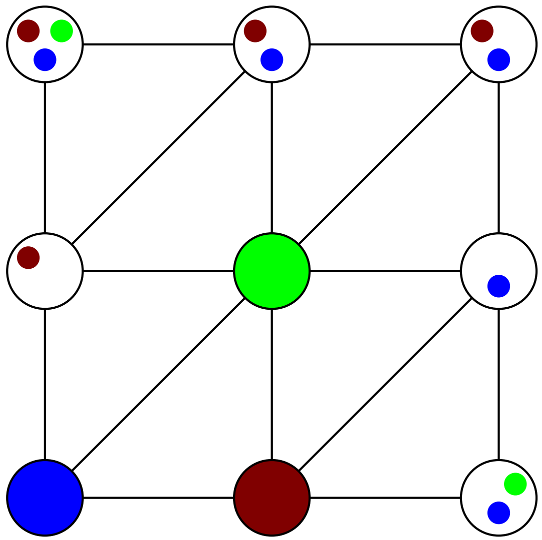
◇ Impossible values get removed from related domains

Forward Checking plus First-Fail



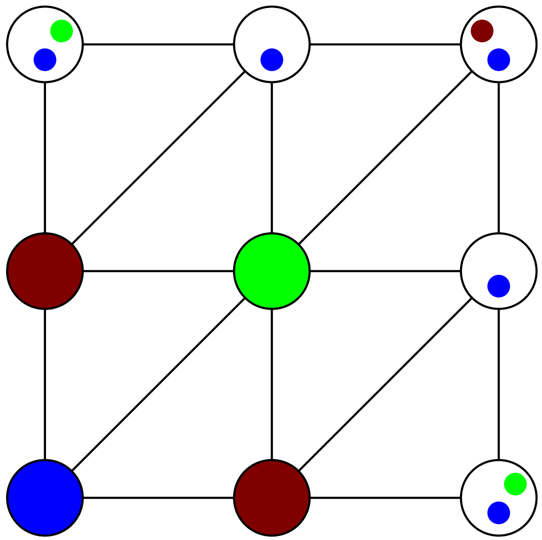
◇ Note that there is only one value in the middle

Forward Checking plus First-Fail



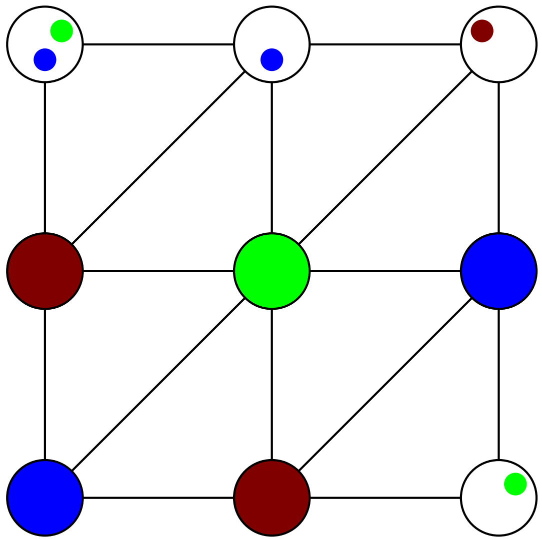
- ♢ Colour that one green, as it has the smallest domain
- ♢ More domains are reduced to singletons

Forward Checking plus First-Fail



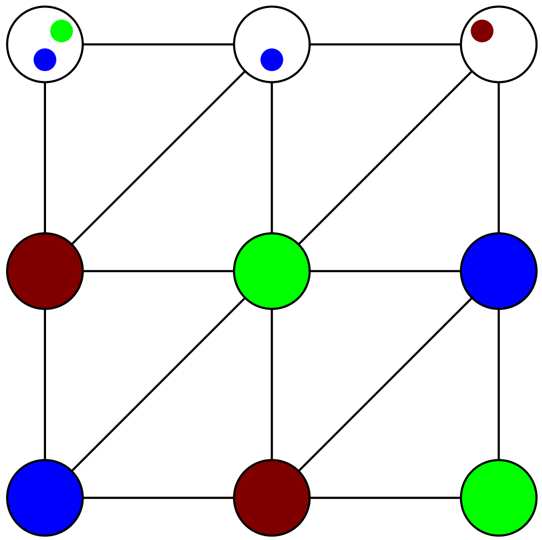
◇ Forced move

Forward Checking plus First-Fail



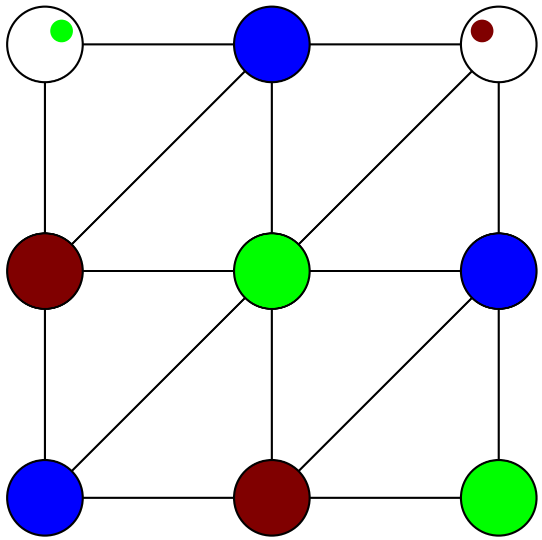
◇ Forced move

Forward Checking plus First-Fail



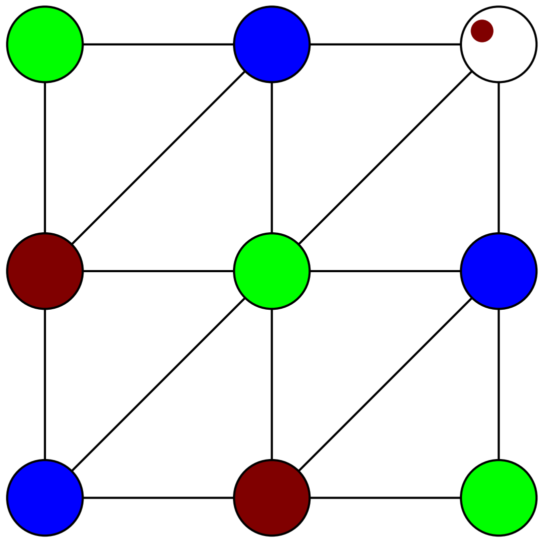
◊ Forced move

Forward Checking plus First-Fail



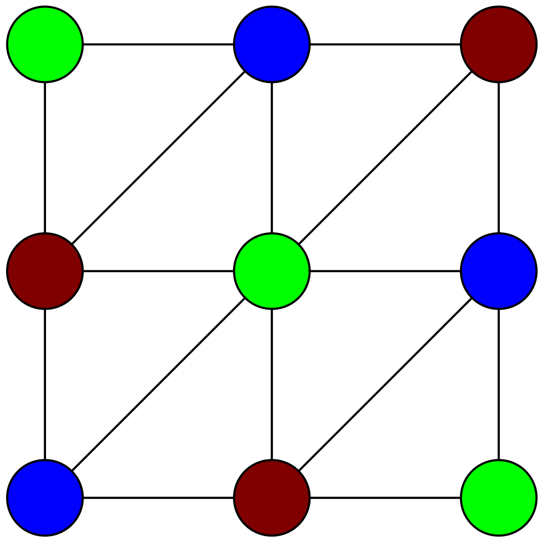
♢ Forced move

Forward Checking plus First-Fail



◇ Forced move

Forward Checking plus First-Fail



◇ Search was backtrack-free!

Arc Consistency

- ◇ A stronger inference rule: make all variables arc consistent

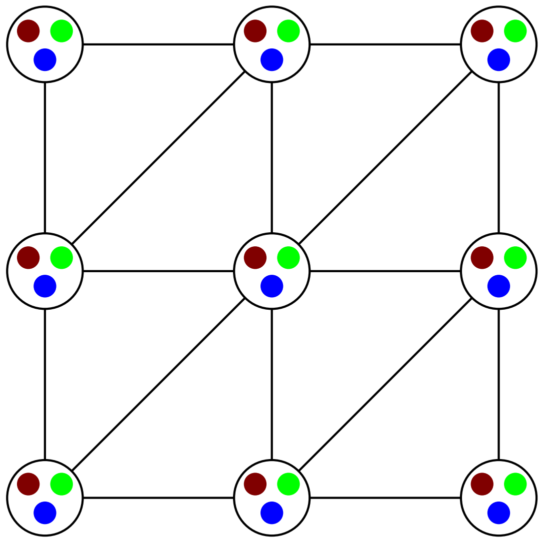
- ◇ Variable v is arc consistent with respect to another variable u iff for every $d \in D_v$ there is at least one $d' \in D_u$ such that $(d, d') \in C_{v,u}$. A CSP $\gamma = (V, D, C)$ is said to be arc consistent (AC) iff every variable in V is arc consistent with every other.

- ◇ 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.

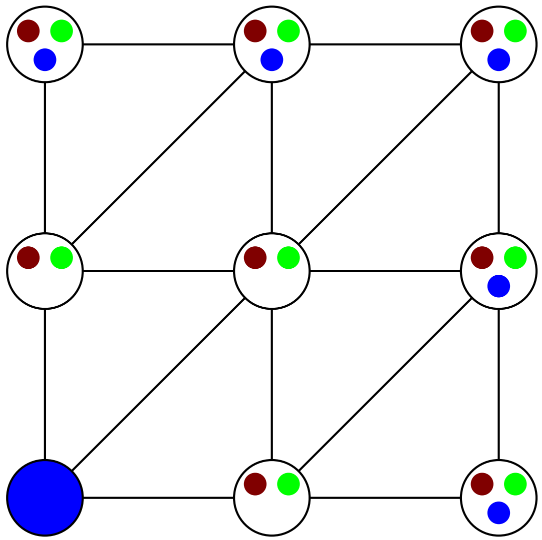
- ◇ Removing all unsupported values makes γ AC. This is clearly a valid constraint inference, as no solutions are lost.

- ◇ Enforcing AC subsumes both forward checking and unit propagation.

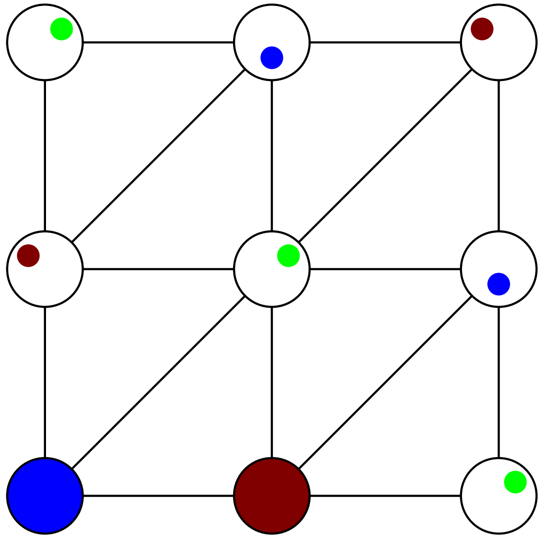
Arc Consistency



Arc Consistency



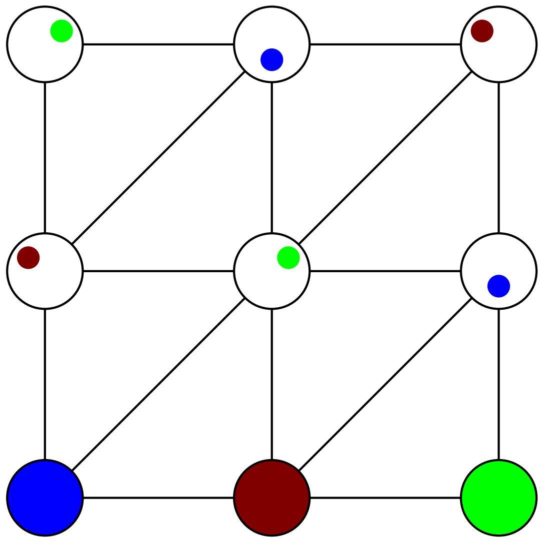
Arc Consistency



◇ 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

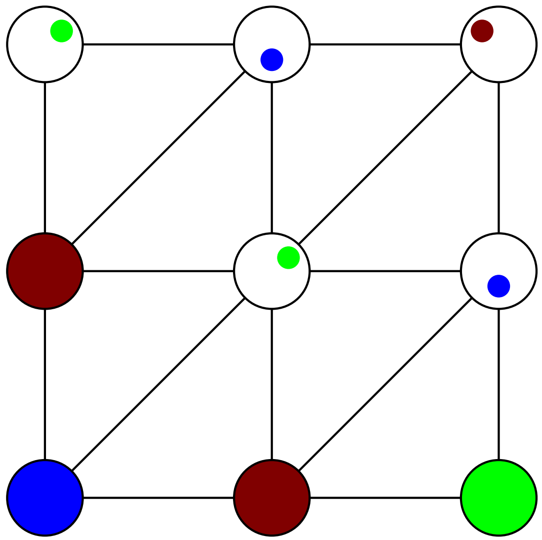
Arc Consistency



◇ 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

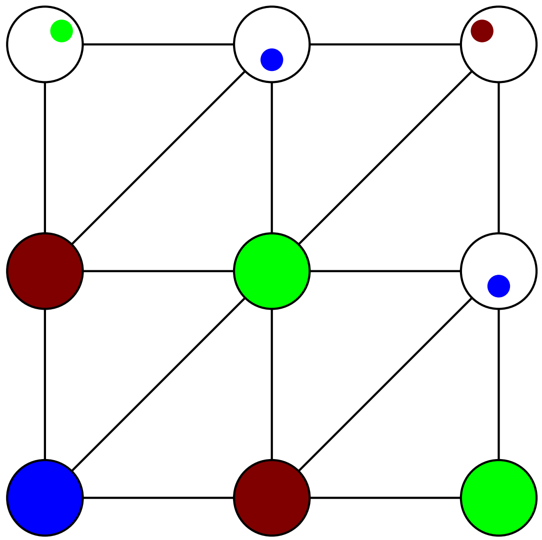
Arc Consistency



◇ 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

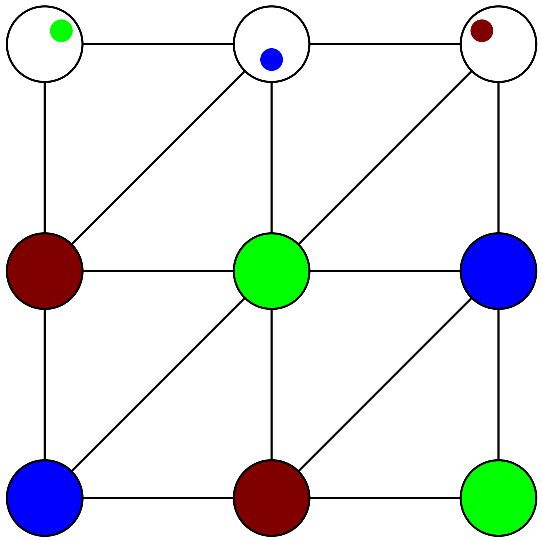
Arc Consistency



◇ 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

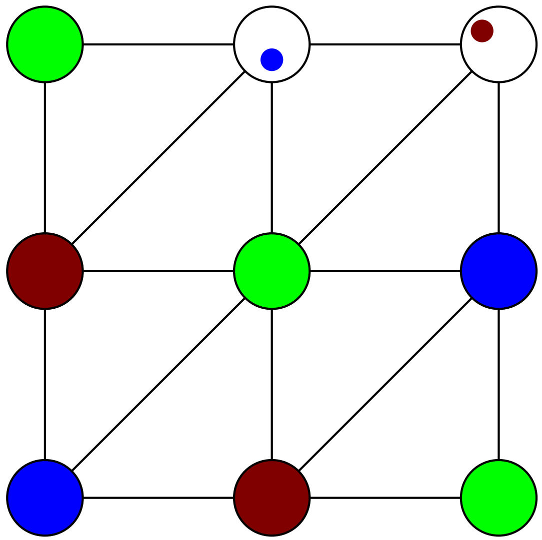
Arc Consistency



◇ 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

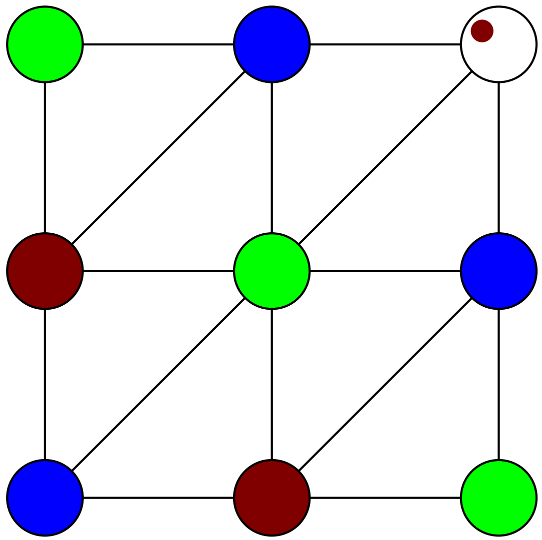
Arc Consistency



◇ 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

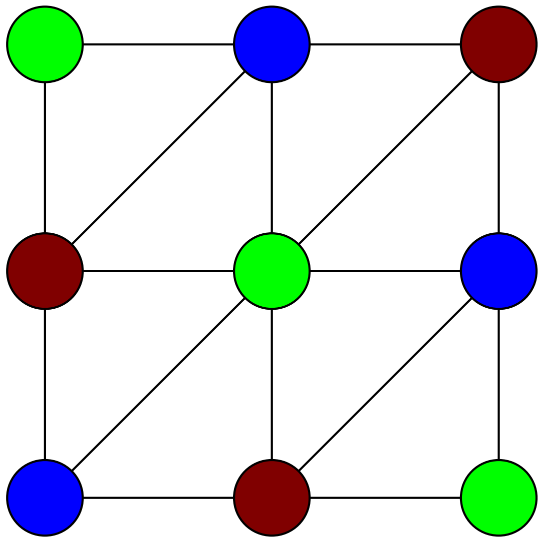
Arc Consistency



♢ 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

Arc consistency: AC-3

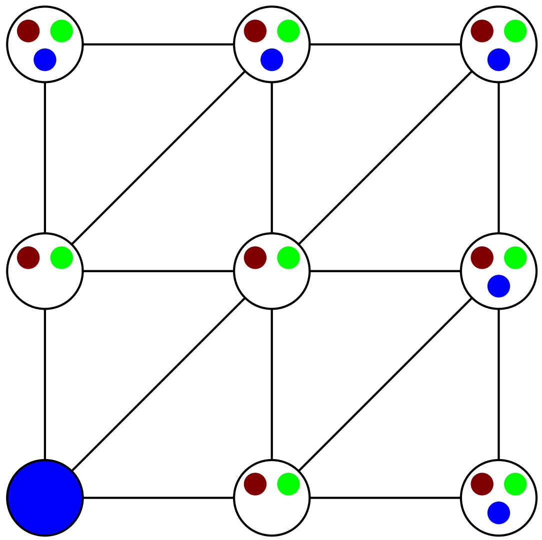
function Revise(γ, u, v) **returns** modified γ

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

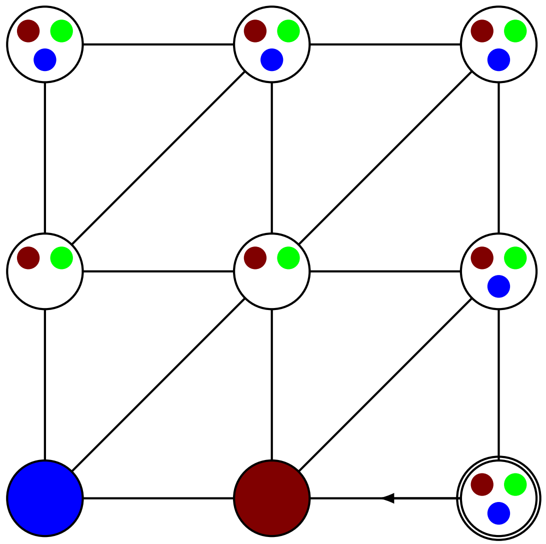
function AC-3(γ) **returns** modified γ

  $M \leftarrow \{(u, v), (v, u) : C_{u,v} \in C\}$
  while $M \neq \{\}$ **do**
    remove some element (u, v) from M
    $\gamma \leftarrow$ 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

AC-3 Example

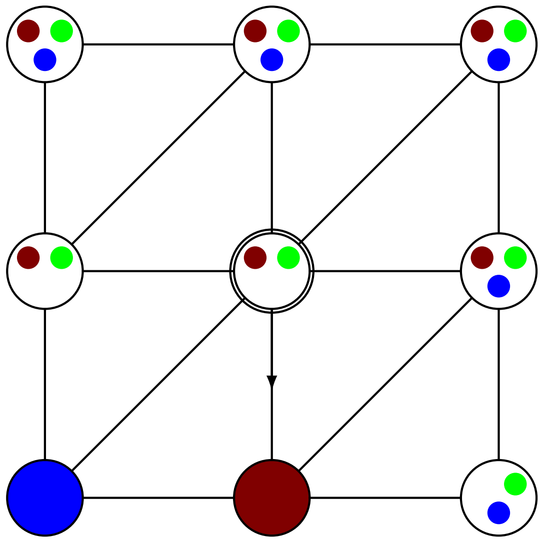


AC-3 Example



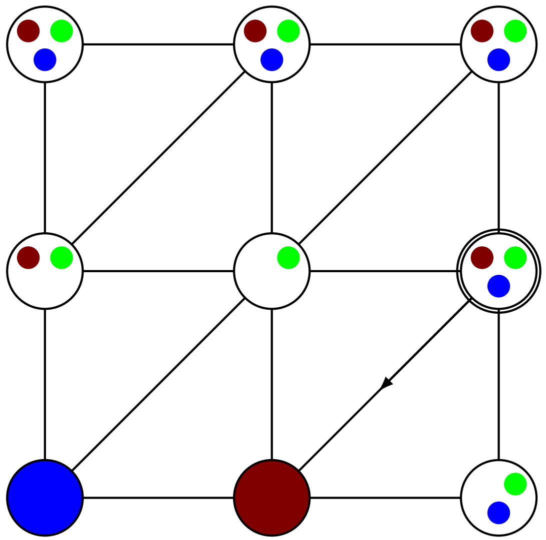
$M = \{(3,2),(5,2),(6,2)\}$

AC-3 Example



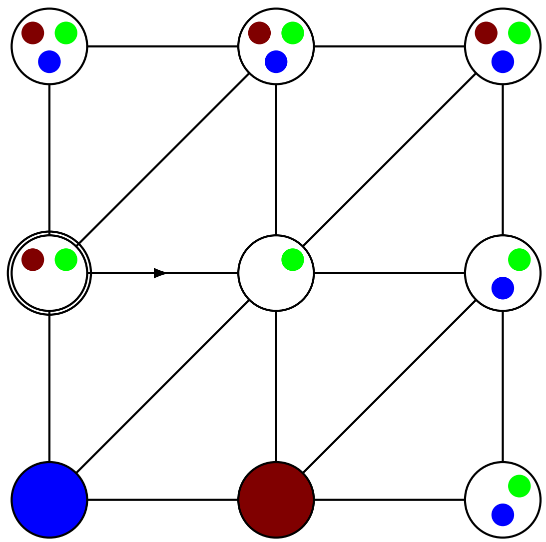
$M = \{(5, 2), (6, 2), (6, 3)\}$

AC-3 Example



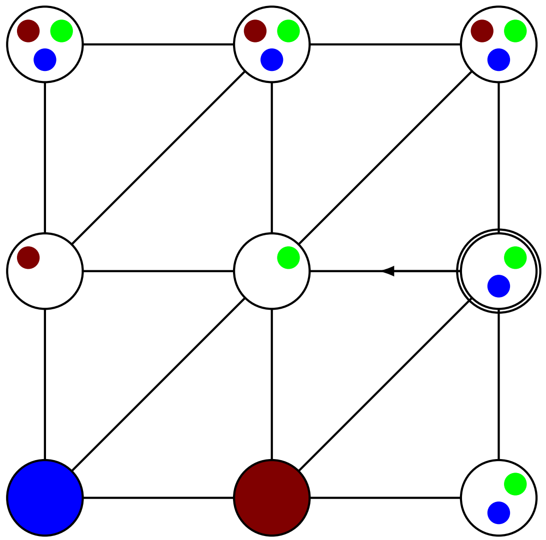
$$M = \{(6,2),(6,3),(4,5),(6,5),(8,5),(9,5)\}$$

AC-3 Example



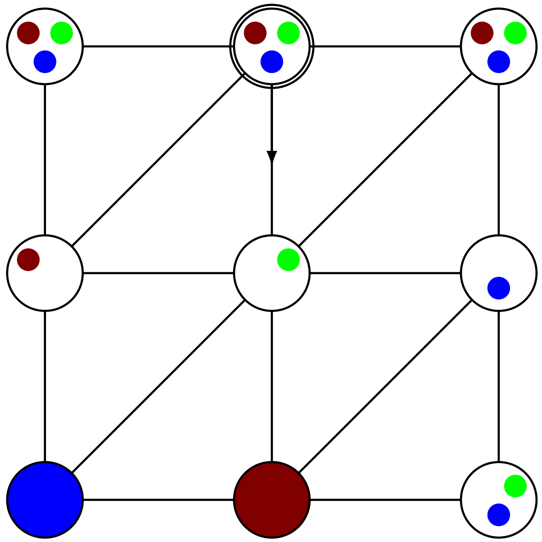
$$M = \{(6,3), (4,5), (6,5), (8,5), (9,5), (3,6), (5,6), (9,6)\}$$

AC-3 Example



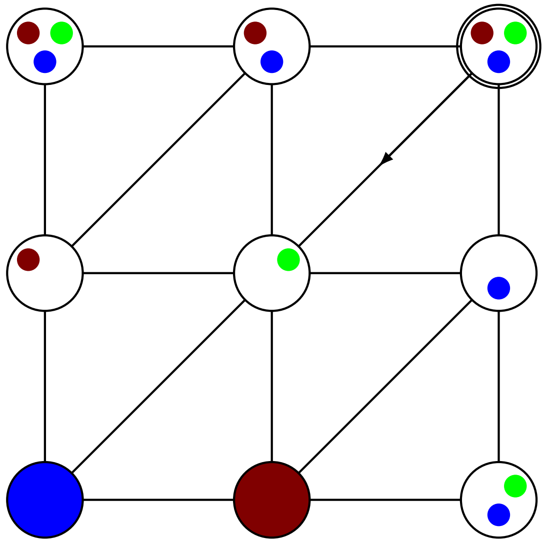
$$M = \{(6,5),(8,5),(9,5),(3,6),(5,6),(9,6),(7,4),(8,4)\}$$

AC-3 Example



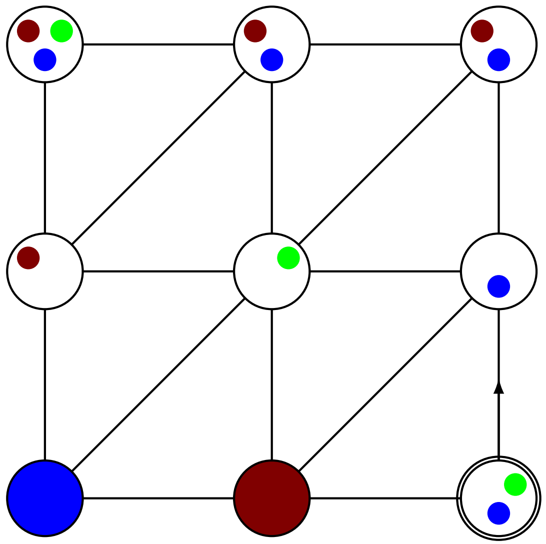
$$M = \{(8,5),(9,5),(3,6),(5,6),(9,6),(7,4),(8,4),(2,6)\}$$

AC-3 Example



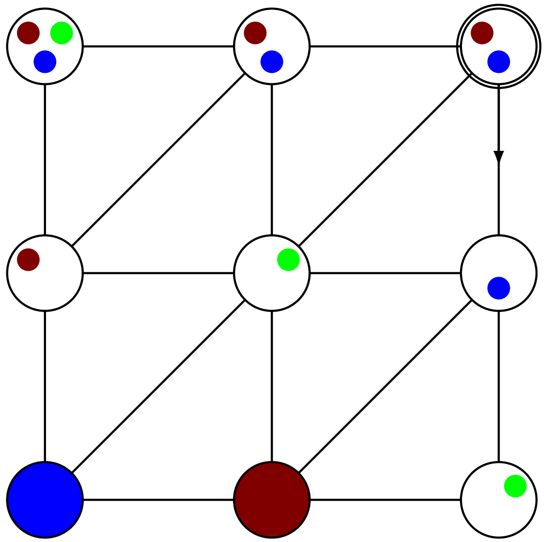
$$M = \{(9,5),(3,6),(5,6),(9,6),(7,4),(8,4),(2,6),(4,8),(7,8),(9,8)\}$$

AC-3 Example



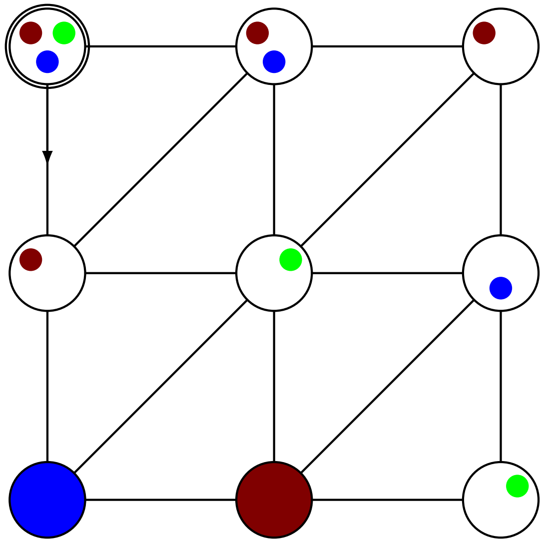
$$M = \{(3,6),(5,6),(9,6),(7,4),(8,4),(2,6),(4,8),(7,8),(9,8),(6,9),(8,9)\}$$

AC-3 Example



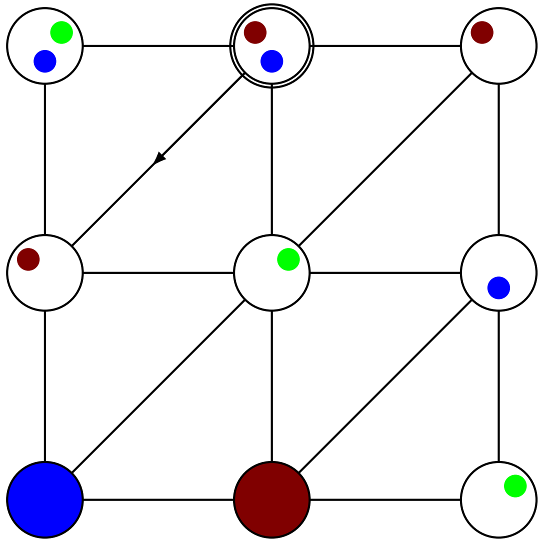
$$M = \{(5,6),(9,6),(7,4),(8,4),(2,6),(4,8),(7,8),(9,8),(6,9),(8,9),(2,3)\}$$

AC-3 Example



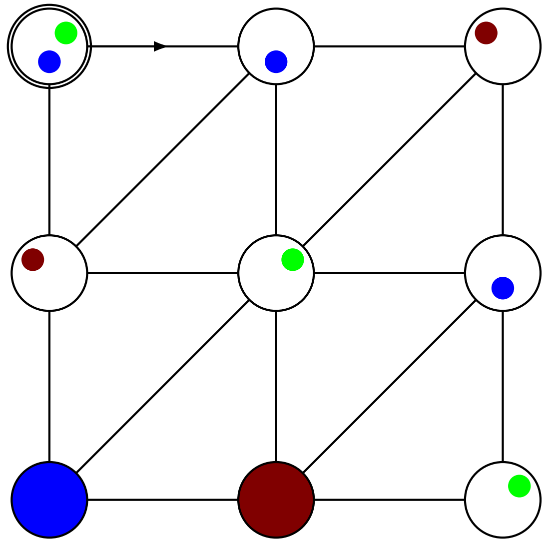
$$M = \{(7,4),(8,4),(2,6),(4,8),(7,8),(9,8),(6,9),(8,9),(2,3),(5,9)\}$$

AC-3 Example



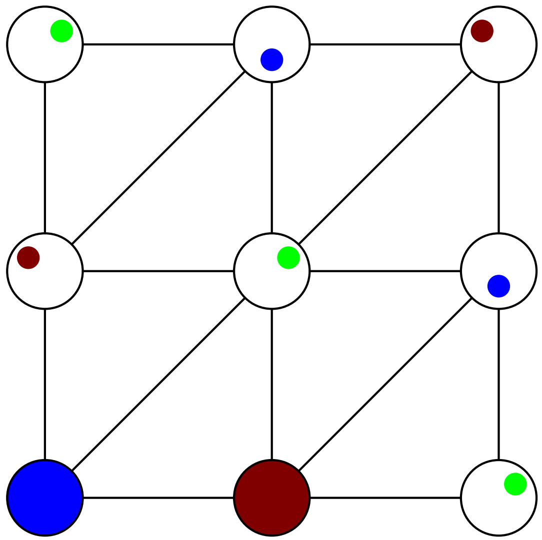
$$M = \{(8,4),(2,6),(4,8),(7,8),(9,8),(6,9),(8,9),(2,3),(5,9,(8,7))\}$$

AC-3 Example



$$M = \{(2,6),(4,8),(7,8),(9,8),(6,9),(8,9),(2,3),(5,9),(8,7),(5,8)\}$$

AC-3 Example



$$M = \{(9,8),(6,9),(8,9),(2,3),(5,9),(8,7),(5,8),(4,7)\}$$

Arc consistency: notes

◊ At every iteration, all arcs not in the set M are consistent
— M contains the arcs that still need to be checked
— M often implemented as a queue, but it doesn't have to be

◊ On termination, the network is AC

◊ Unlike forward checking, makes inferences from unassigned variables

◊ Arc consistency is widely used in modern CSP solvers

◊ Slower (per node) than forward checking, but prunes more

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.

◊ **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.

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

◊ **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.