EECS 391
Intro to AI

CSPs - Inference and Local Search

L9 Thu Sep 28, 2017

Recap

- Constraint Satisfactions Problems (CSPs)
- Backtracking search
  - expand one variable at a time
  - backtrack when no legal values are left
- Key functions of algorithm:

  var ← selectUnassignedVar()

  foreach val in OrderDomainVals()

- Heuristics for selecting variables:
  - ***Minimum Remaining Values*** (MRV):

    Choose variable with fewest legal values
  - ***Degree Heuristic***:

    Choose variable that most constrains others

    ⇒ maximum reduction in tree size

Types of constraints

- **unary** constraint:
  restricts the value of a variable, e.g. Q= {R,G,B}
- **binary** constraint:
  relates two variables, e.g. WA≠NT
  can be represented by a graph
- **global** constraint:
  constraints involving an arbitrary number of variables (need not be all),
  e.g. *Alldiff:* all variables have different values.

- It is always possible to transform n-ary constraints into binary-constraints.

Today

- More examples of CSPs
- Constraint Propagation: Inference in CSPs
  - node consistency
  - arc consistency
  - path consistency
  - k-consistency
  - global constraints
- Back-tracking search
  - intelligent back-tracking
- Local search for CSPs

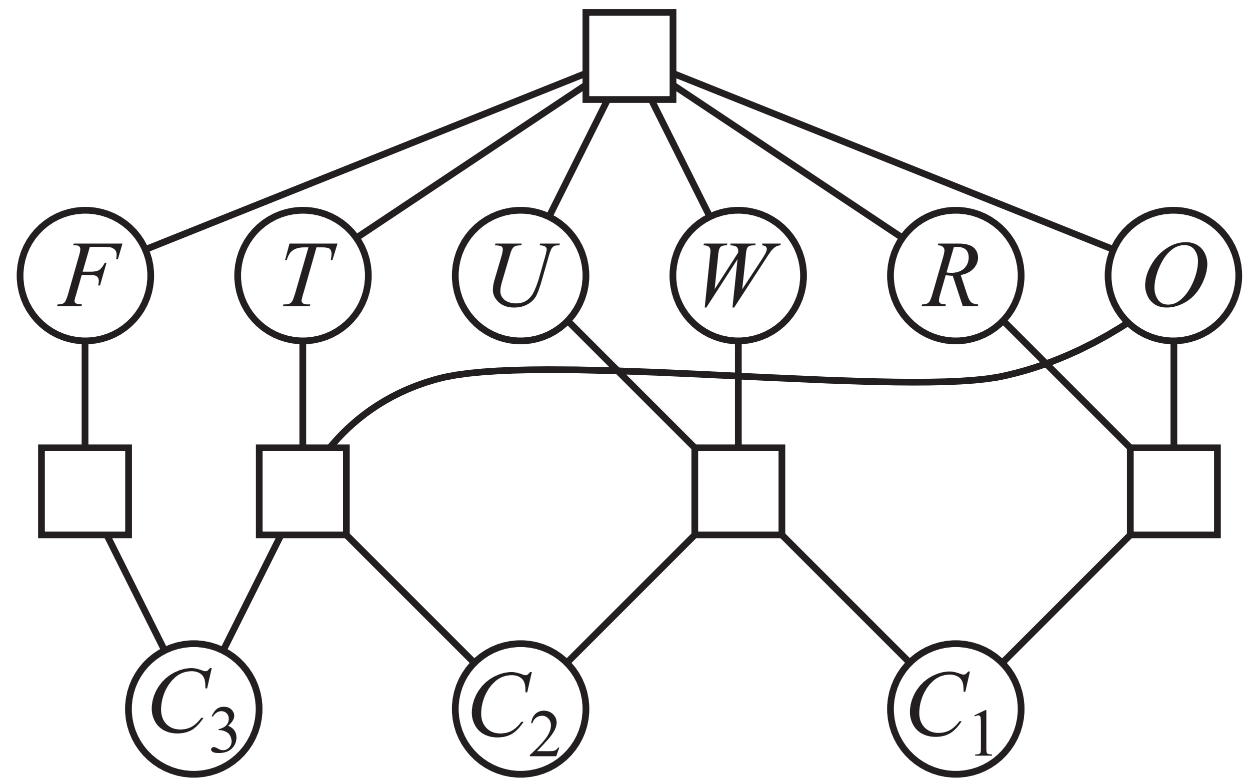
Cryptarithmatic Problem

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

- Each letter stands for a unique digit.
- Objective is to find digits for letters such that sum is arithmetically correct.
- The constraints make it interesting:
  - all digits are different
  - must follow the rules of arithmetic
- Need to specific constraint equations
- Then make a constraint graph

Cryptarithmatic Problem

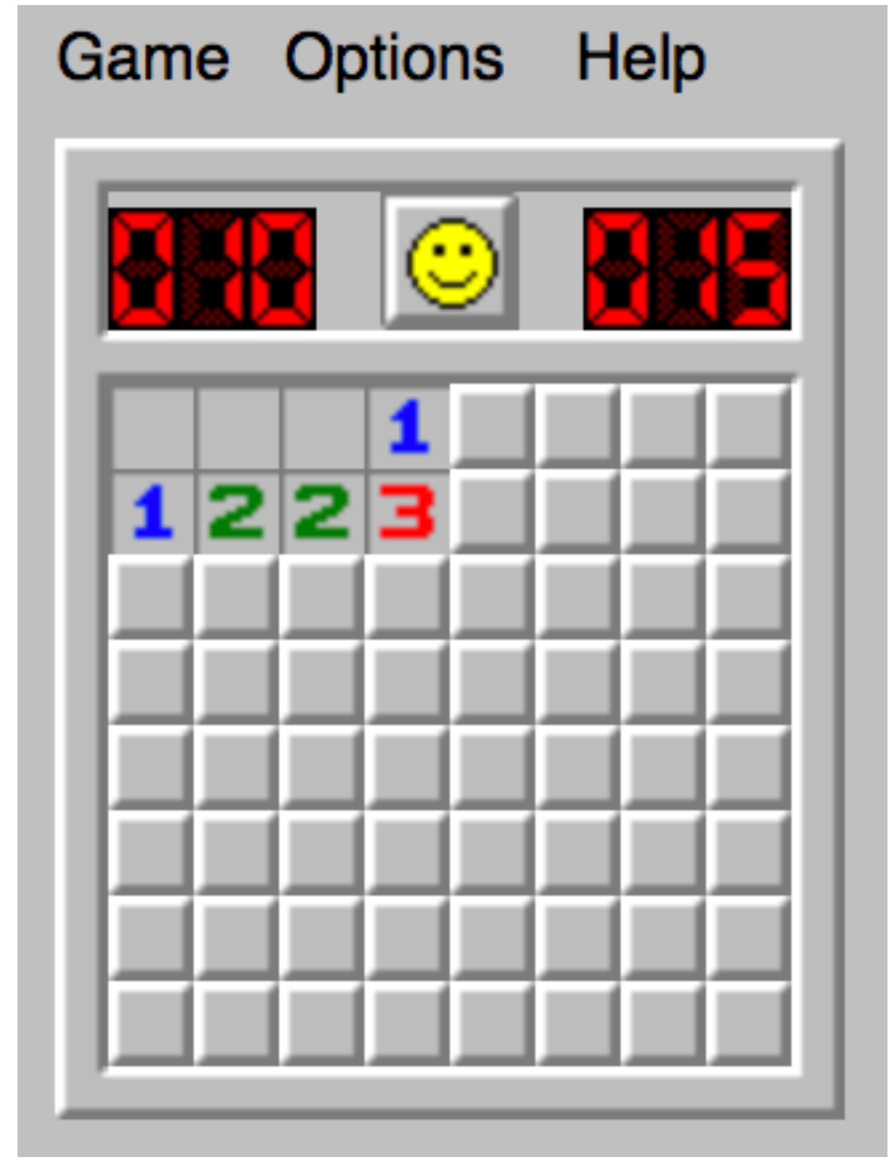
$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



- Each letter stands for a unique digit.
- Objective is to find digits for letters such that sum is arithmetically correct.
- The constraints make it interesting:
  - all digits are different
  - must follow the rules of arithmetic
- Need to specific constraint equations
- Then make a constraint graph

Minesweeper as a CSP

- An unknown number of mines are hidden beneath blank squares.
- Objective is to clear the board.
- Clicks on mine squares: player looses
- Clicks on non-mine squares: show the # of adjacent mines.
- If no mines are adjacent, the square becomes blank, and all adjacent squares are recursively revealed.
- Player can use numbers to deduce the location of mines.
- Locations can be marked to aid game play.



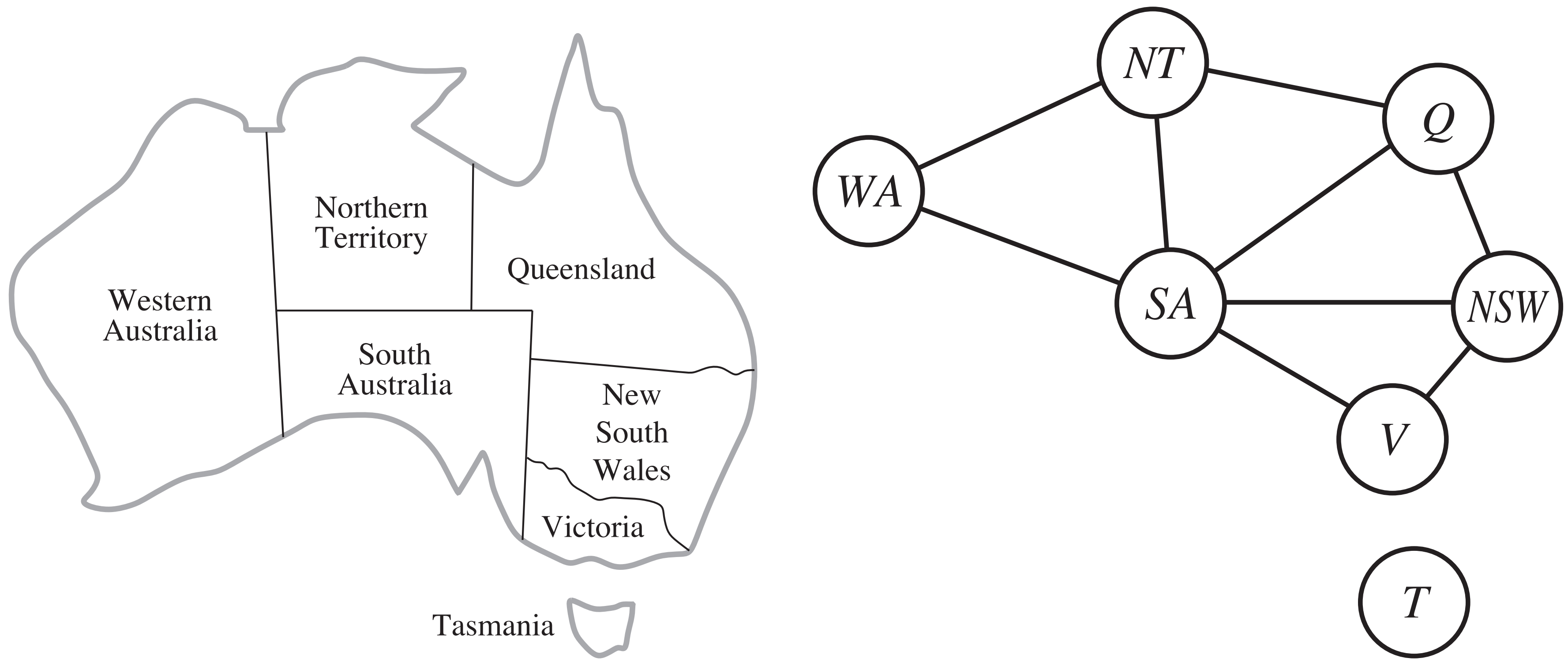
Can we do better?

- Goal is to reduce the search space as much as possible
- Can we do more intelligent checking?
- In state-space search, we only expanded nodes based on the successors.
- In CSPs:
  1. algorithm can "*search*", i.e. assign new values to variables, or
  2. do a specific type of *inference* called ***constraint propagation***

- Constraint Propagation
  - use constraints to reduce the number of legal values of one var, which in turn (i.e. propagate to) reduce the legal values of other vars.
  - could be used during search
  - or done as a pre-processing step (which might solve the problem)

Node consistency

- single vars correspond to nodes
- a var is ***node consistent*** if all the values in the var's domain satisfy the unary constraint

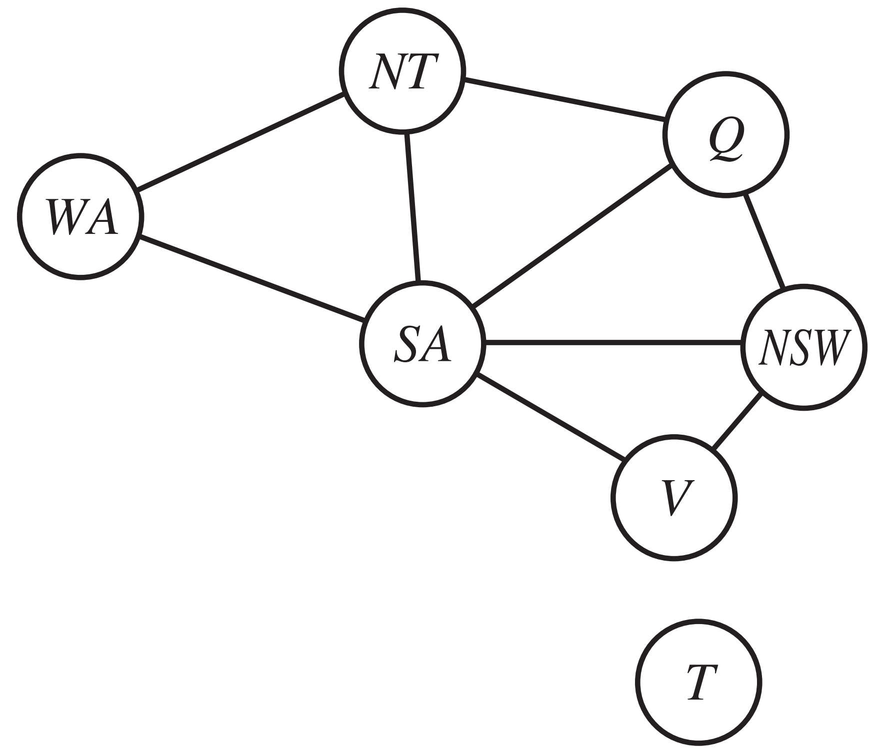


	WA	NT	Q	NSW	V	SA	T
Initial domain	RGB	RGB	RGB	RGB	RGB	RGB	RGB

Original coloring problem was unrestricted, but we could add local constraints.

Node consistency

- single vars correspond to nodes
- a var is ***node consistent*** if all the values in the var's domain satisfy the unary constraint

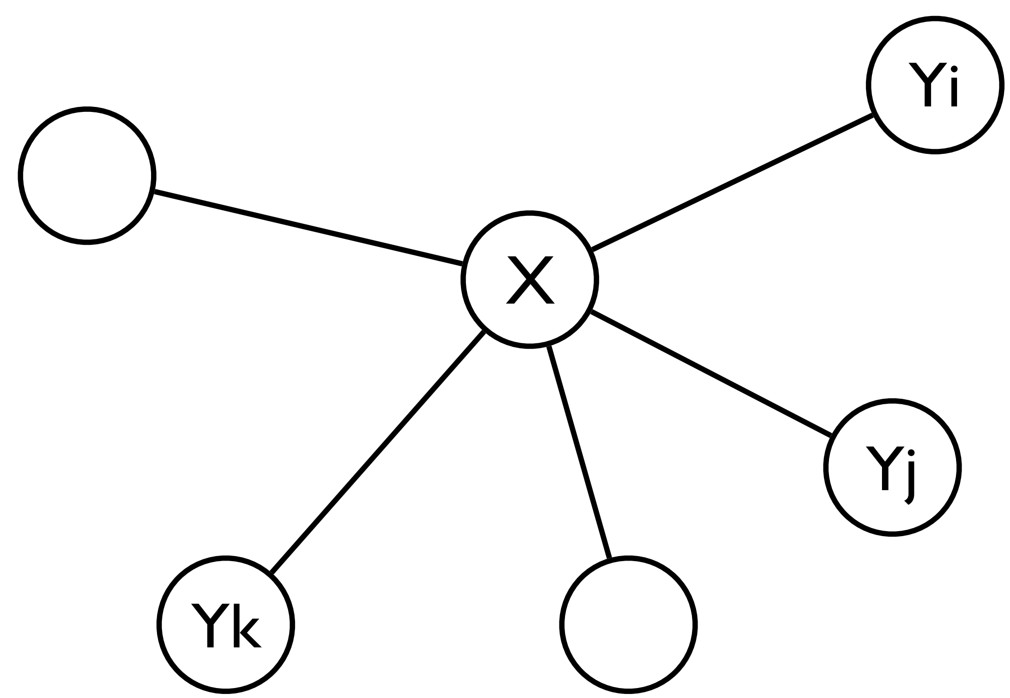


	WA	NT	Q	NSW	V	SA	T
Initial domain	xGB	RGB	B(gr)	RGB	RGB	RGB	RGx

For example, WA doesn't like red. Q prefers blue. etc.

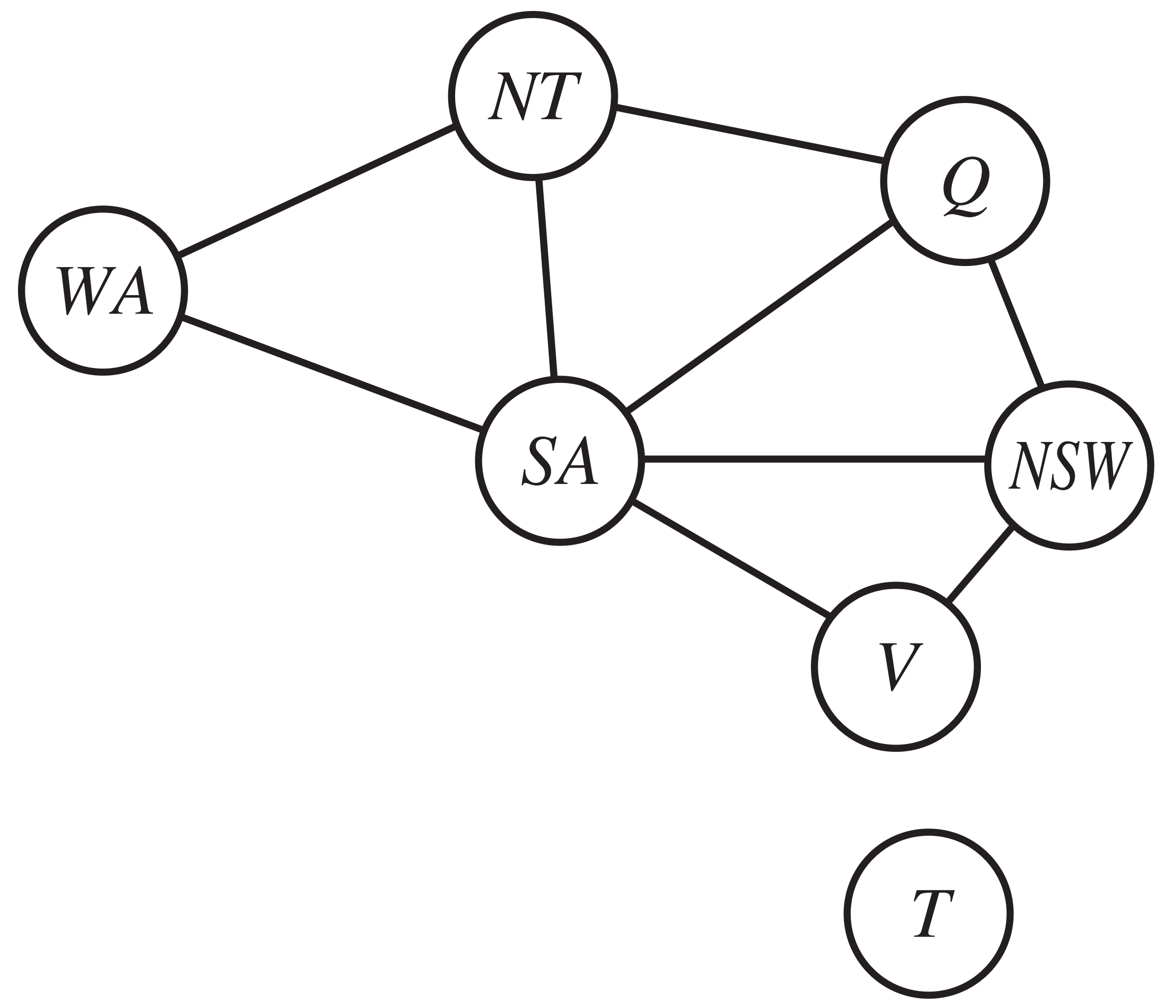
Forward Checking

- Idea: Whenever a variable X is assigned, do ***forward checking*** from X:
  - look at each unassigned variable Y that is connected to X
  - delete values from Y's domain that are inconsistent with X



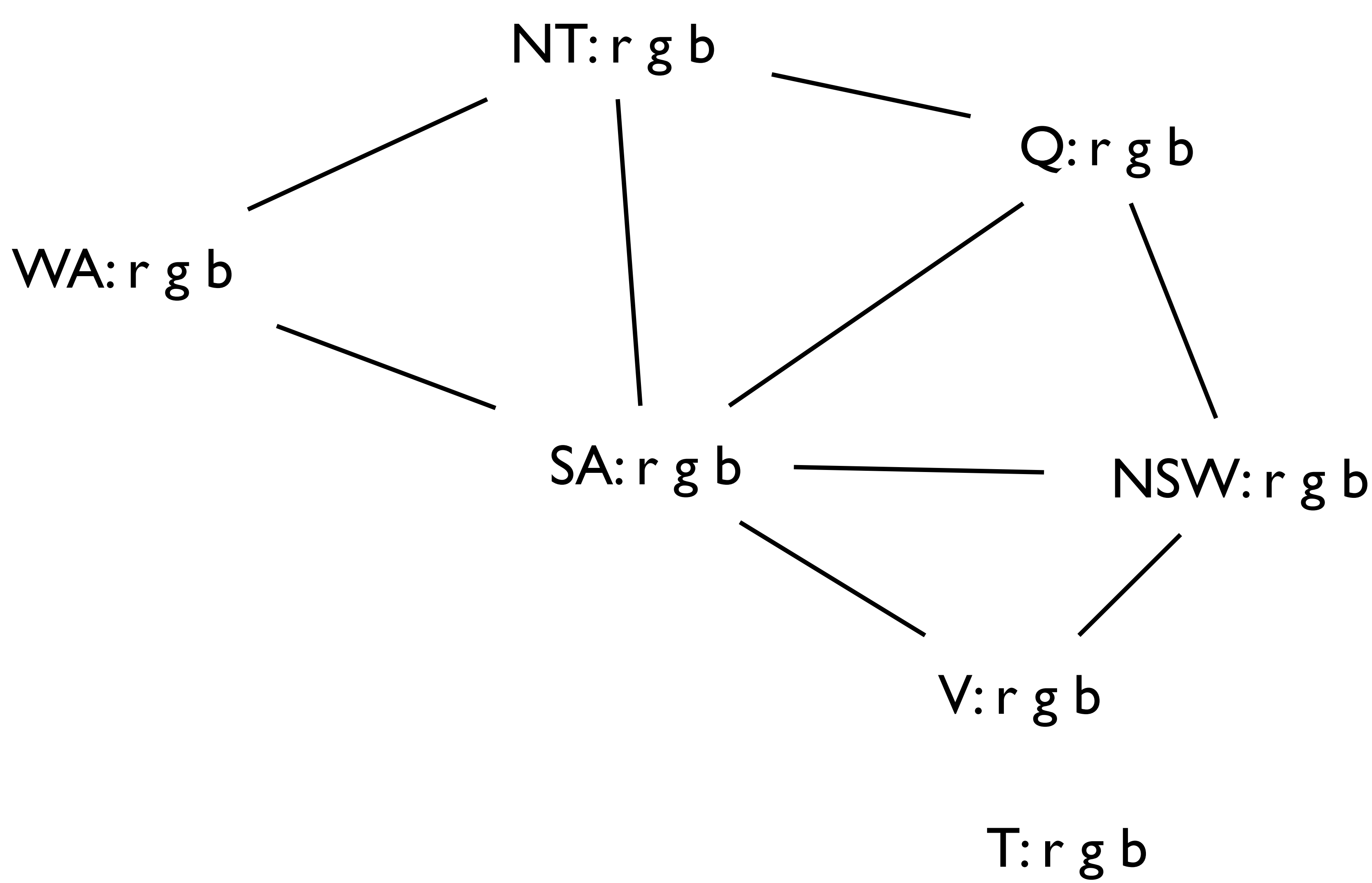
Forward Checking

- Whenever X is assigned
  - Foreach unassigned neighbor Y of X
    - delete values from Y's domain that are inconsistent with X



Forward Checking

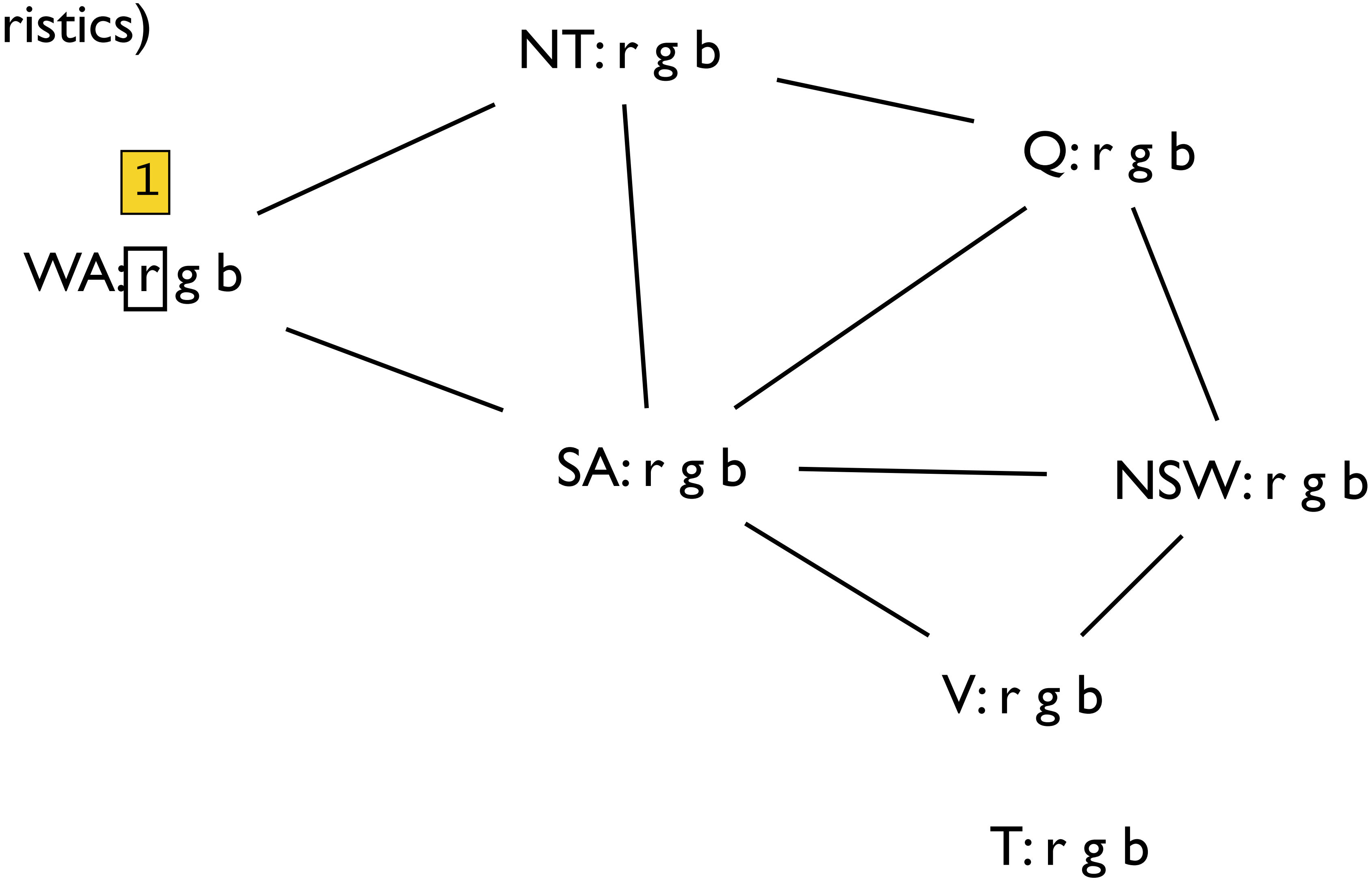
- Whenever X is assigned
  - Foreach unassigned neighbor Y of X
    - delete values from Y's domain that are inconsistent with X



Forward Checking

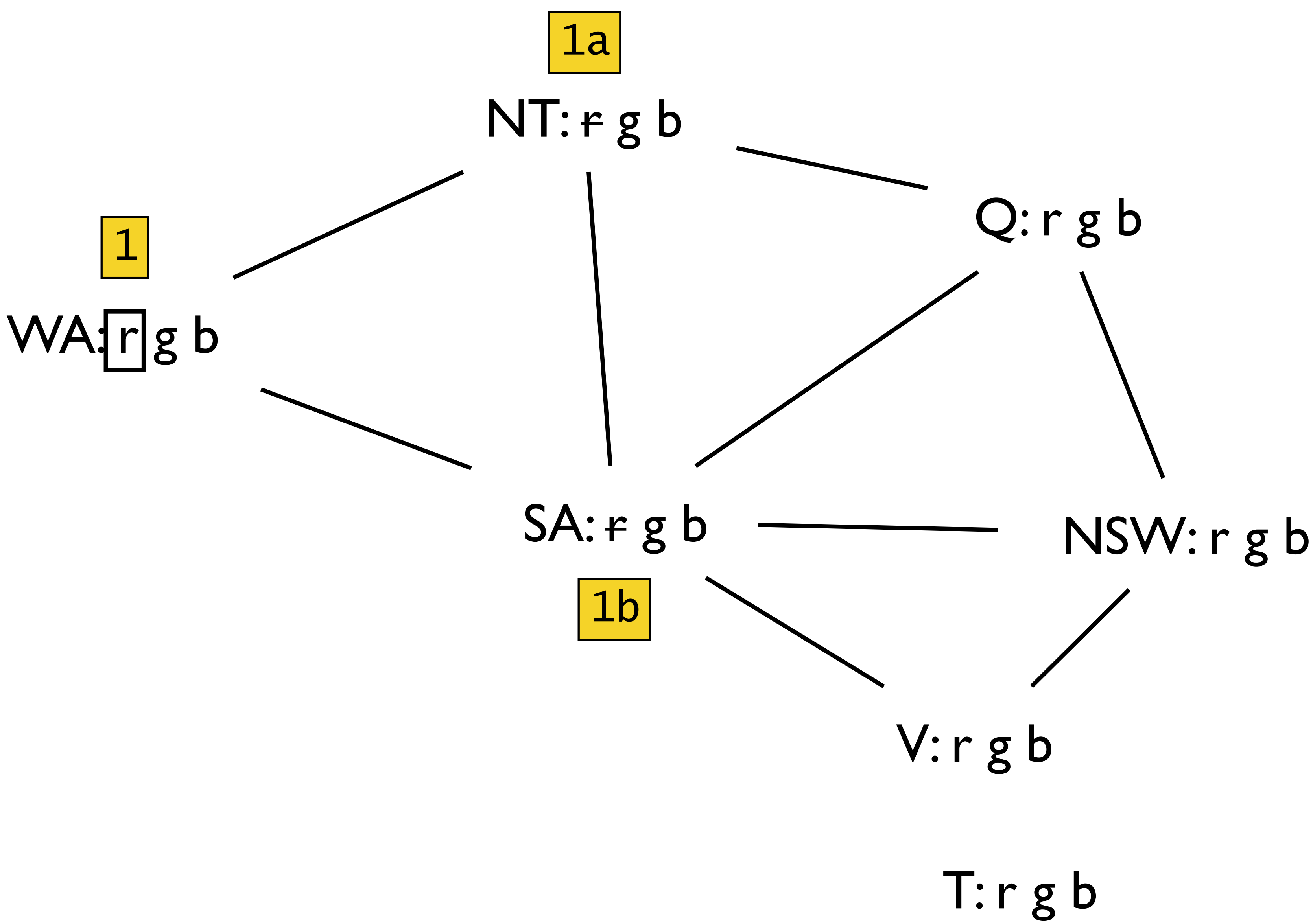
- Whenever X is assigned
  - Foreach unassigned neighbor Y of X
    - delete values from Y's domain that are inconsistent with X

(Assume we're not using any heuristics)



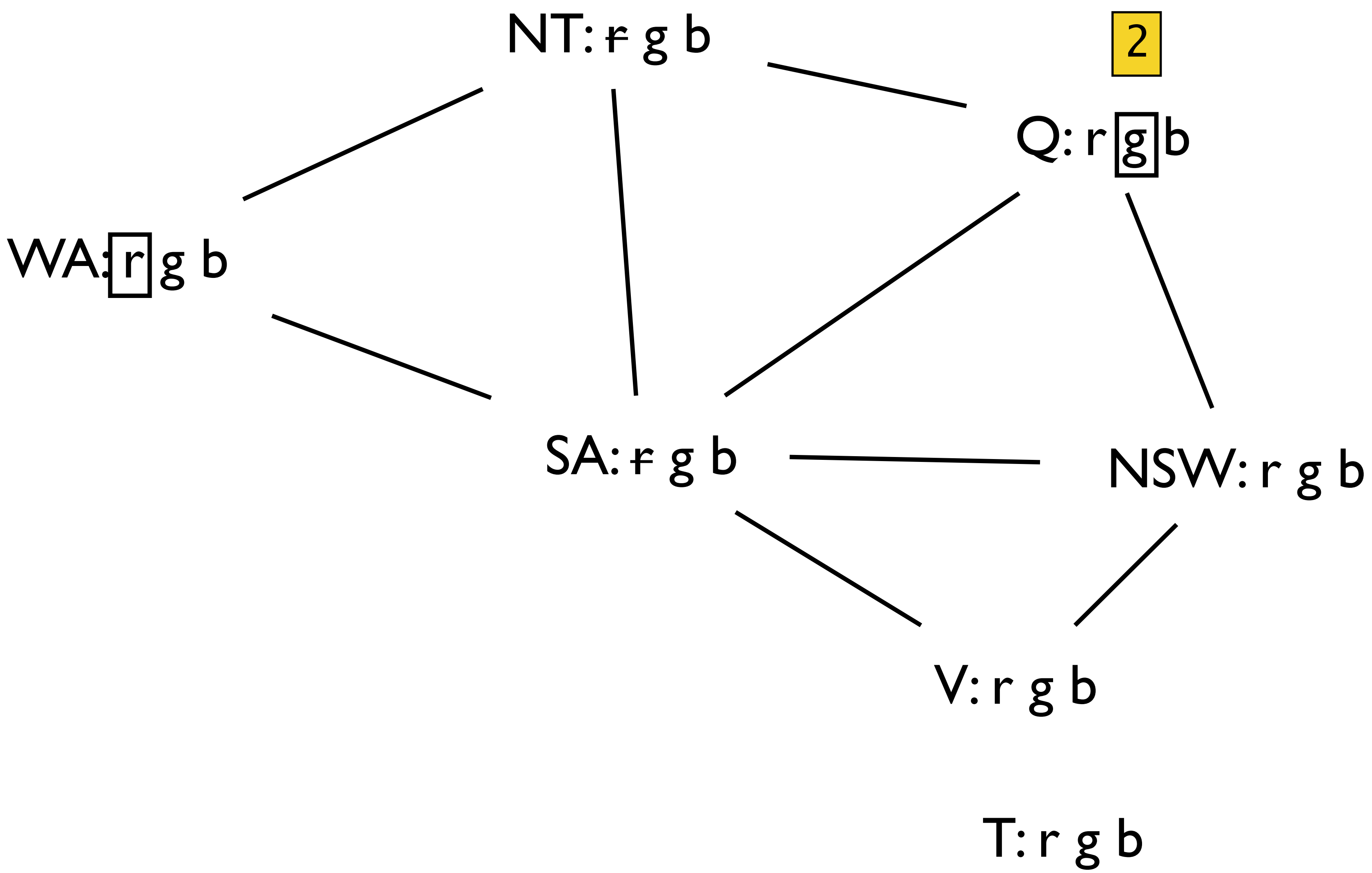
Forward Checking

- Whenever X is assigned
  - Foreach unassigned neighbor Y of X
    - delete values from Y's domain that are inconsistent with X



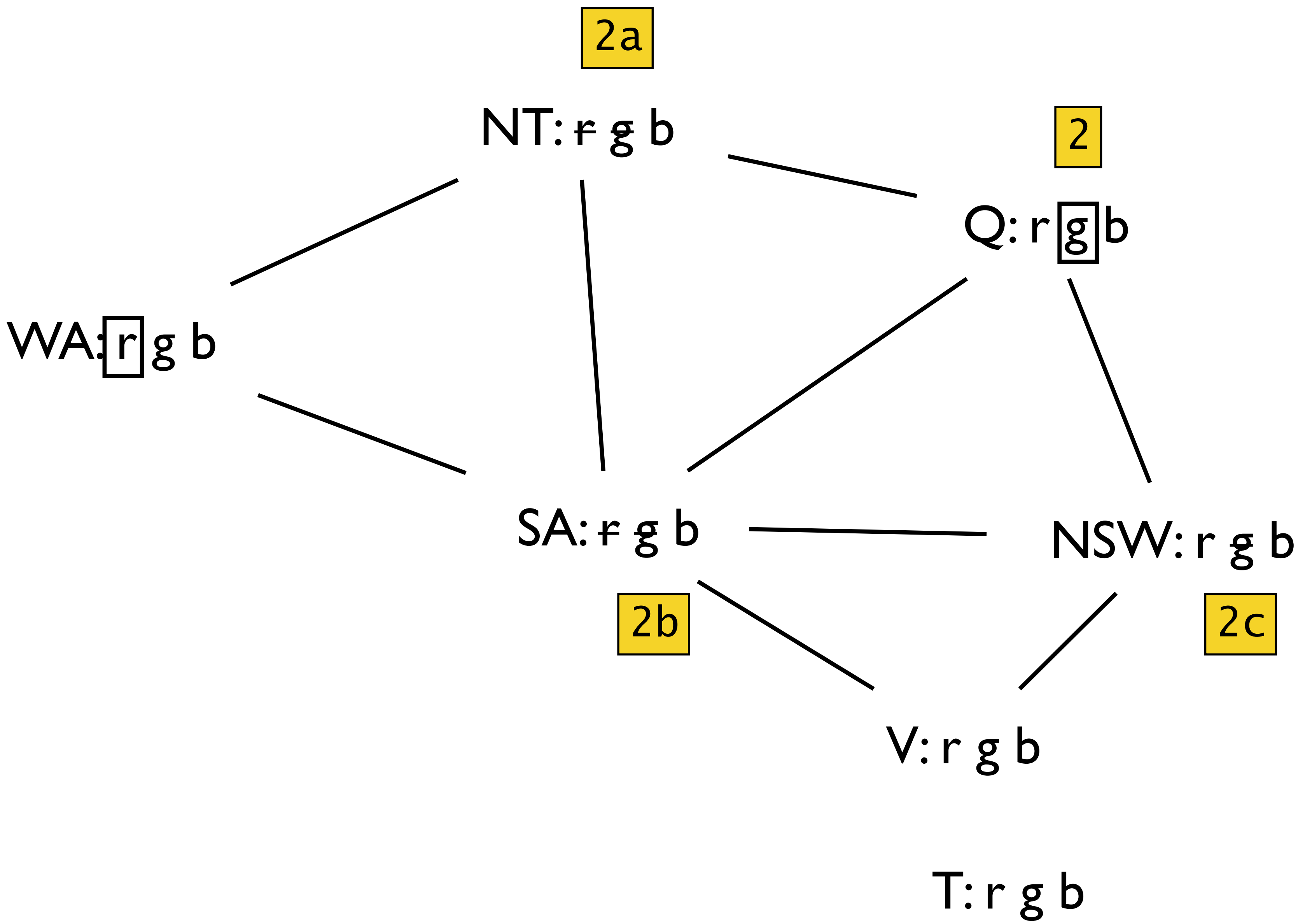
Forward Checking

- Whenever X is assigned
  - Foreach unassigned neighbor Y of X
    - delete values from Y's domain that are inconsistent with X



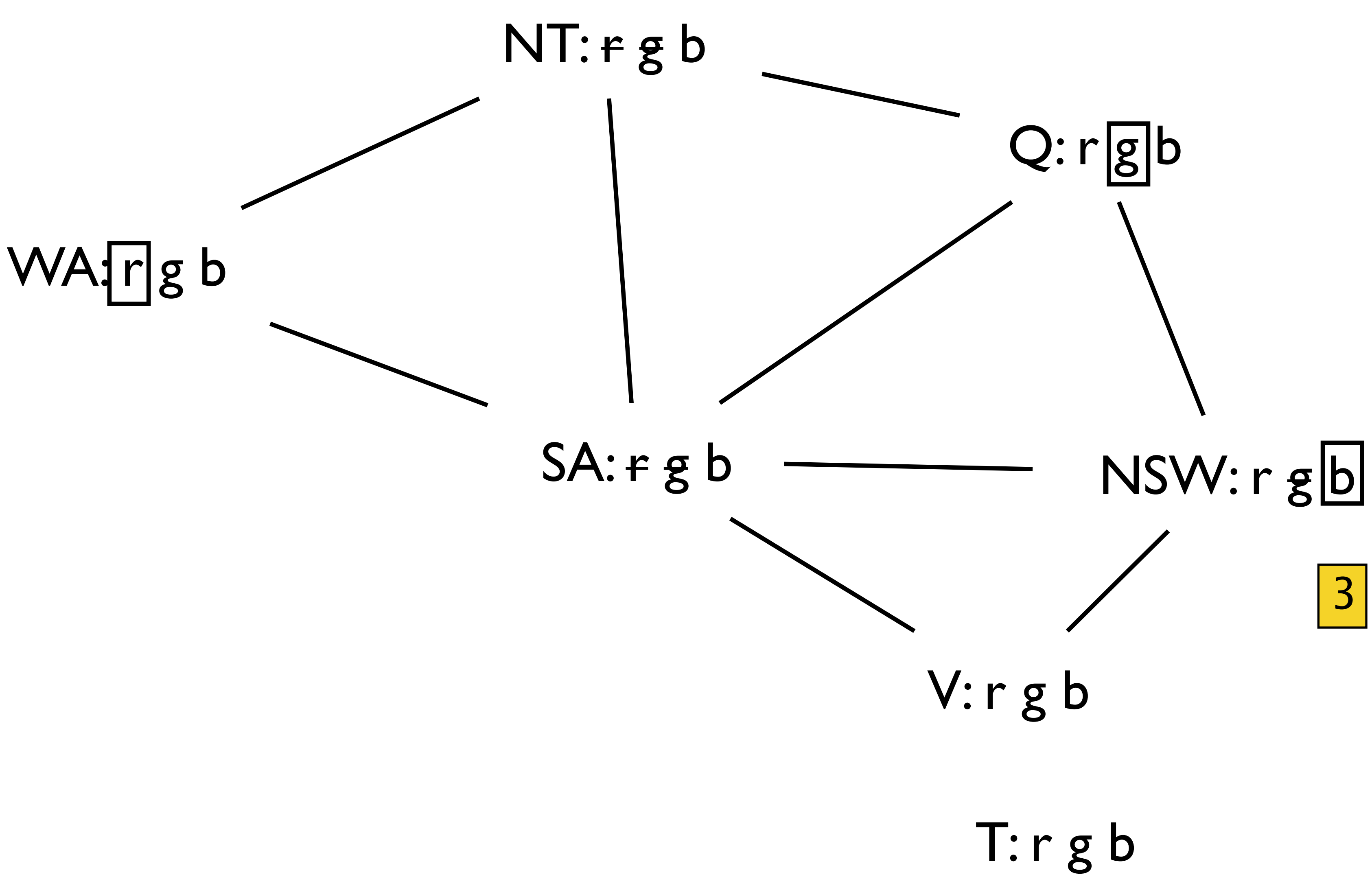
Forward Checking

- Whenever X is assigned
  - Foreach unassigned neighbor Y of X
    - delete values from Y's domain that are inconsistent with X



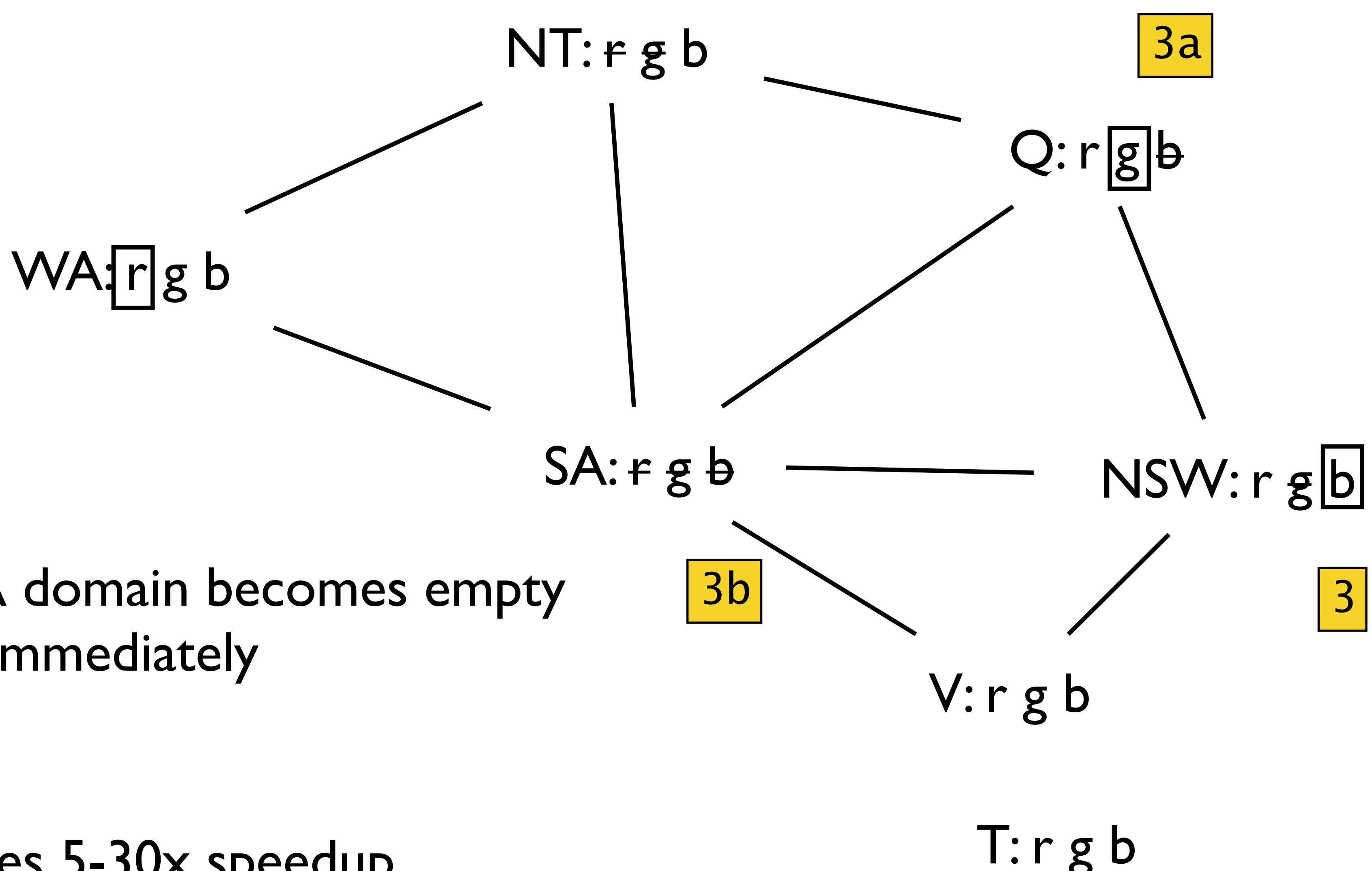
Forward Checking

- Whenever X is assigned
  - Foreach unassigned neighbor Y of X
    - delete values from Y's domain that are inconsistent with X



Forward Checking

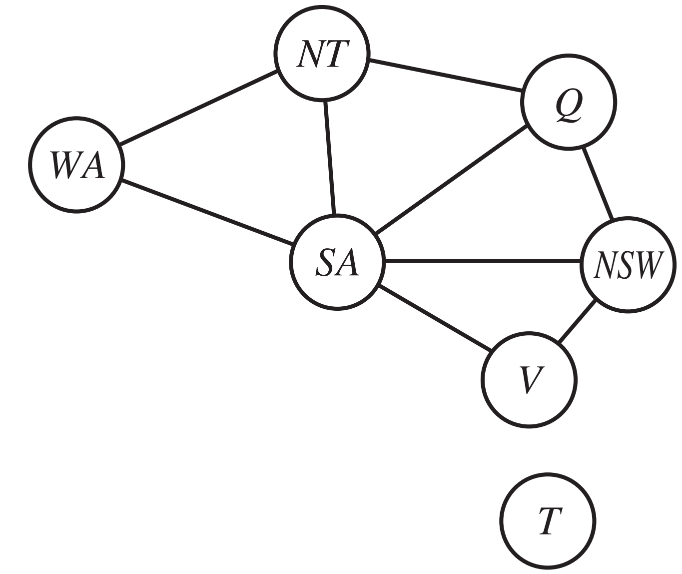
- Whenever X is assigned
  - Foreach unassigned neighbor Y of X
    - delete values from Y's domain that are inconsistent with X



At step 3b, SA domain becomes empty
⇒ backtrack immediately

MRV + FC gives 5-30x speedup.

Same example of forward checking using a table



	WA	NT	Q	NSW	V	SA	T
Initial domain	RGB	RGB	RGB	RGB	RGB	RGB	RGB
WA = R	**R**	xGB	~	~	~	xGB	~
Q = G	~	xxB	**G**	RxB	~	xxB	~
NSW = B	~	~	~	**B**	RGx	xxx	~

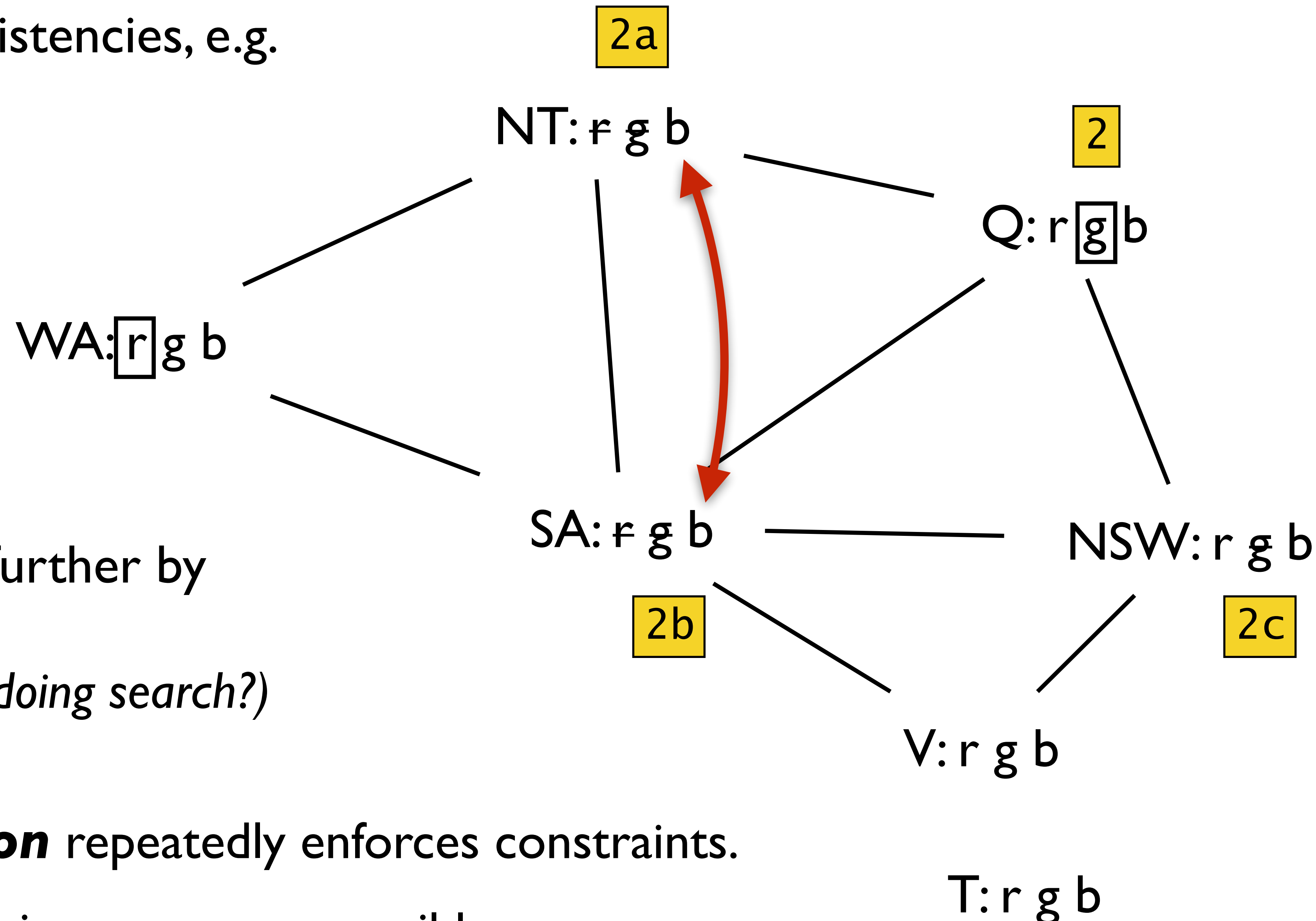
~ indicates no change
x means we've eliminated a domain value

SA domain becomes empty ⇒ backtrack.

Constraint Propagation

- Forward consistency computes the domain of each var independently at start
- Doesn't check all inconsistencies, e.g.

Last example after step 2: SA and NT can't both be blue.



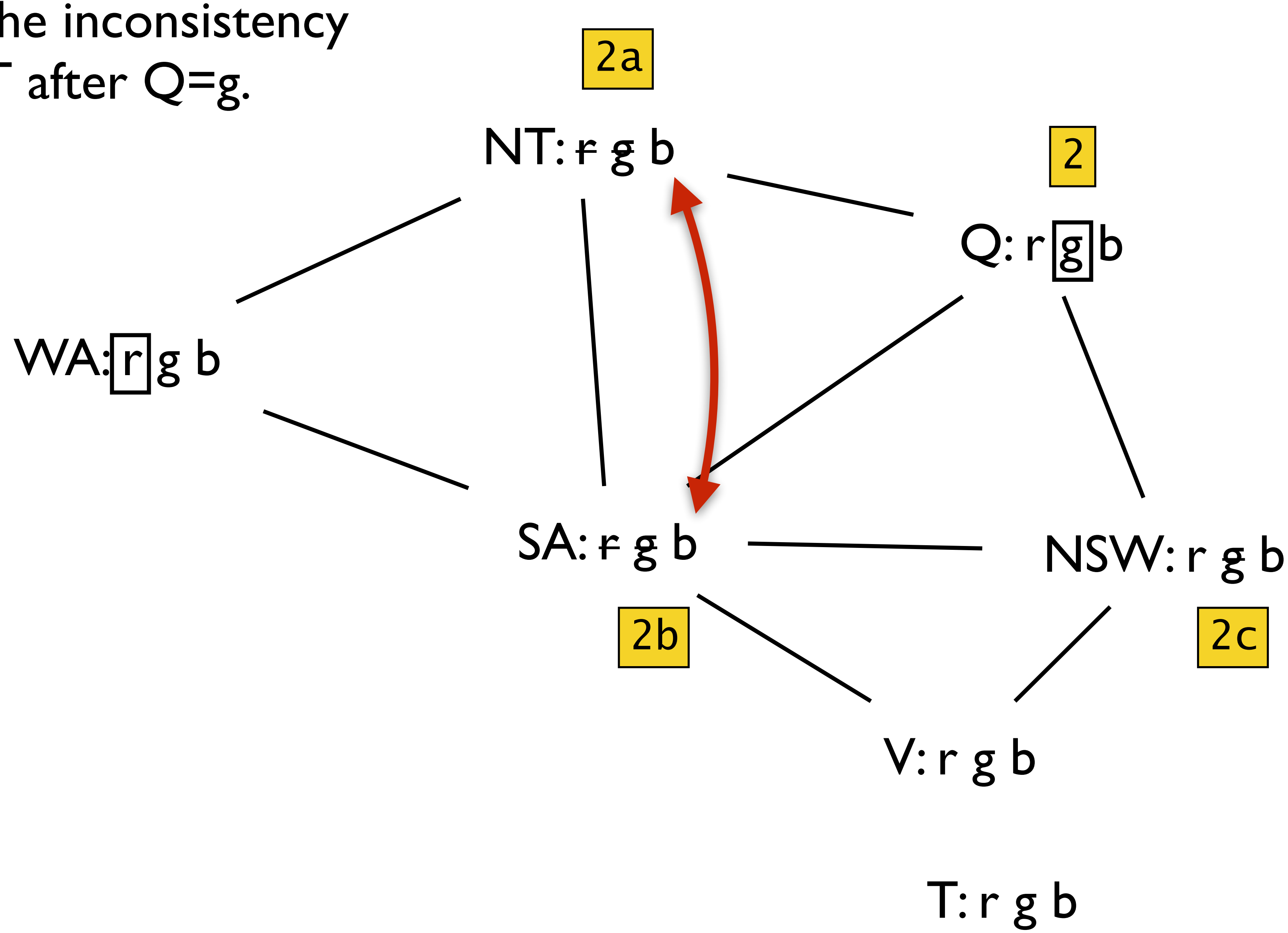
- What if we propagated further by looking more ahead?
  (*Would that be as slow as doing search?*)
- ***Constraint propagation*** repeatedly enforces constraints.
- Idea is detect inconsistencies as soon as possible.
- Simplest is ***arc-consistency***, which is a generalization of forward checking.

Arc consistency

- X_i is **arc consistent** w.r.t X_j if

$$\forall x \in D_i \;\; \exists y \in D_j \text{ satisfying constraint } (X_i, X_j)$$

- Fast method of constraint propagation. Equivalent to original problem
- Much stronger than forward checking, because it propagates constraints.
- This would detect the inconsistency between SA and NT after Q=g.



```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
   inputs: csp, a binary CSP with components (X, D, C)
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if REVISE(csp, X_i, X_j) then
         if size of D_i = 0 then return false
         for each X_k in X_i.NEIGHBORS - {X_j} do
            add (X_k, X_i) to queue
   return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
   revised ← false
   for each x in D_i do
      if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
         delete x from D_i
         revised ← true
   return revised
```

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

Need to check all arcs.

  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, $X_i,\ X_j$) **then**
      if size of $D_i\ =\ 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to $queue$
  return $true$

---

function REVISE(csp, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

Really a set, since there's no priority.

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

If domain of Xi was revised.

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then                    If domain of Xi was revised.
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

This is implementing the following:

X_i is **arc consistent** w.r.t X_j if

$$\forall x \in D_i \;\; \exists\, y \in D_j \text{ satisfying constraint } (X_i, X_j)$$

All x’s that are not arc-consistent, are deleted from X’s domain.

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

This means we can backtrack.

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
inputs: csp, a binary CSP with components $(X,\ D,\ C)$
local variables: $queue$, a queue of arcs, initially all the arcs in csp

while $queue$ is not empty **do**
$(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
if REVISE(csp, $X_i,\ X_j$) **then**
if size of $D_i\ =\ 0$ **then return** $false$
for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
add $(X_k,\ X_i)$ to $queue$
return $true$

Otherwise, need to re-check arcs to Xi: Even if these were already checked, the change in Di might further reduce Dk.

---

function REVISE(csp, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
$revised \leftarrow false$
for each x **in** D_i **do**
if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
delete x from D_i
$revised \leftarrow true$
return $revised$

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, $X_i,\ X_j$) **then**
      if size of $D_i\ =\ 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to $queue$
  return $true$

csp is arc-consistent, so we can keep searching for a solution.

---

function REVISE(csp, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

Complexity of arc-consistency

- AC-3 seems like a lot of work.
  Is *it worth it? Why not just use normal search?*
- Exponential expansion and following dead-ends is also a lot of work.
- What the complexity of AC-3?
  - **n** variables, each with domain of size **d** and **c** binary constraints (arcs)

n variables, each with **d** domain values, and **c** constraints (arcs)

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do** — **c** arcs (constraints)
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, X_i, X_j) **then** — What's the cost of an arc check?
      if size of $D_i = 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do** — Each arc can be inserted at most d times, because there are only **d** domain values.
        add $(X_k,\ X_i)$ to $queue$
  return $true$

⇒ at most **c·d** arc checks

---

function REVISE(csp, X_i, X_j) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do** — **d** domain values for x
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

d domain values for y

⇒ Checking the consistency of an arc is O(**d²**)

⇒ Worst case time is O(**c·d·d²**) = O(**cd³**)

Backtracking search with inference

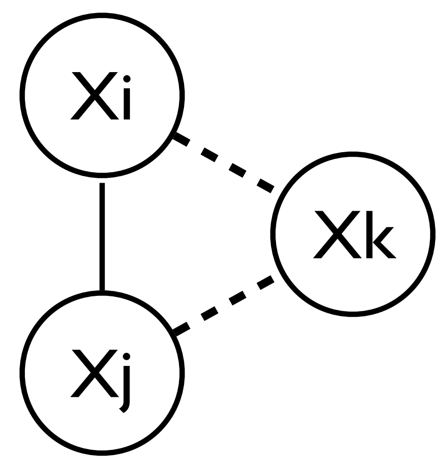
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
           add inferences to assignment
           result ← BACKTRACK(assignment, csp)
           if result ≠ failure then
              return result
     remove {var = value} and inferences from assignment
  return failure
```

Inference = AC-3 only returns T of F, but could add var vals that were inferred.

More general (and stronger) forms of constraint propagation

- k-consistency
- A CSP is k-consistent if
  - for any set of k-1 variables
    and for any consistent assignment of those variables
    - a consistent value can be assigned to the kth variable
  - k=1 1-consistency node consistency
  - k=2 2-consistency same as arc consistency
  - k=3 3-consistency path consistency

  Any consistent pair can be extended to a third variable:



- Generally: There is a trade-off between extra cost of stronger checks and reduction in branching factor.

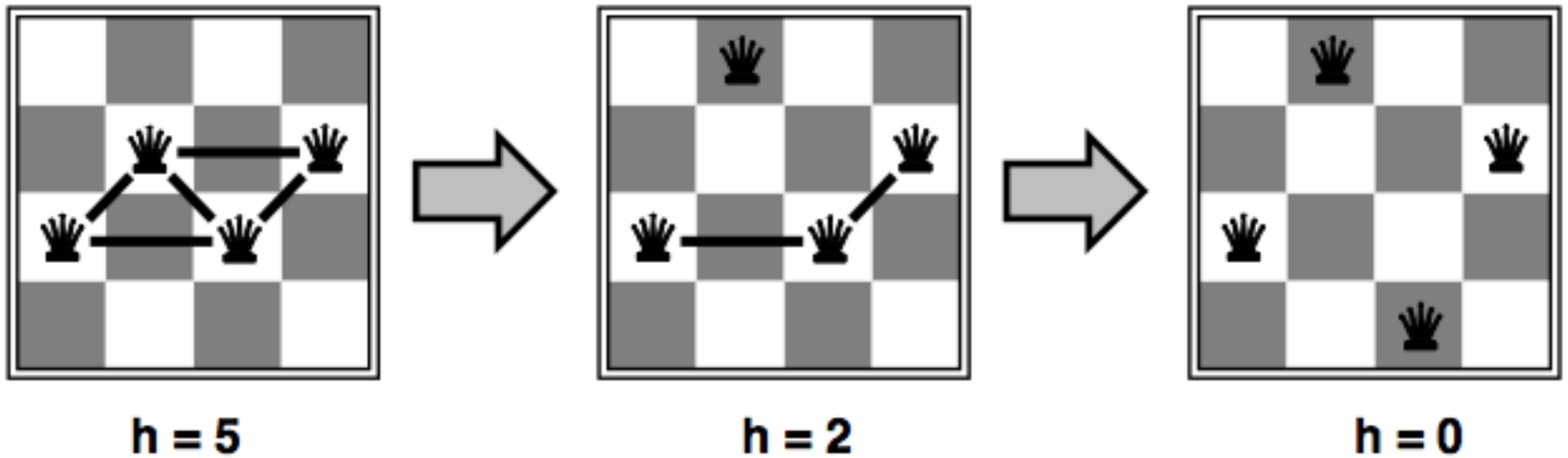
Using local search for CSPs

- Techniques like hill climbing and simulated annealing assume all variables are assigned.
- To apply CSPs:
  - allow states with unsatisfied constraints
  - operators ***reassign*** variable values to improve constraint satisfaction

- Variable section:
  - randomly select any variable inconsistent with constraints

- Value selection:
  - choose value that violates fewest constraints
  - hill climb with heuristic h(n) = number of violated constraints

4-Queens as a CSP



- **states**: 4 queens in 4 columns (4^4 = 256 states)
- **operators**: move queen in column
- **goal test**: no attacks
- **evaluation**: h(n) = number of attacks

Performance of min-conflicts

Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)

The same appears to be true for any randomly-generated CSP **except** in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$