Artificial Intelligence

Lecture 04: Constraint Satisfaction Problems

(AIMA C6)

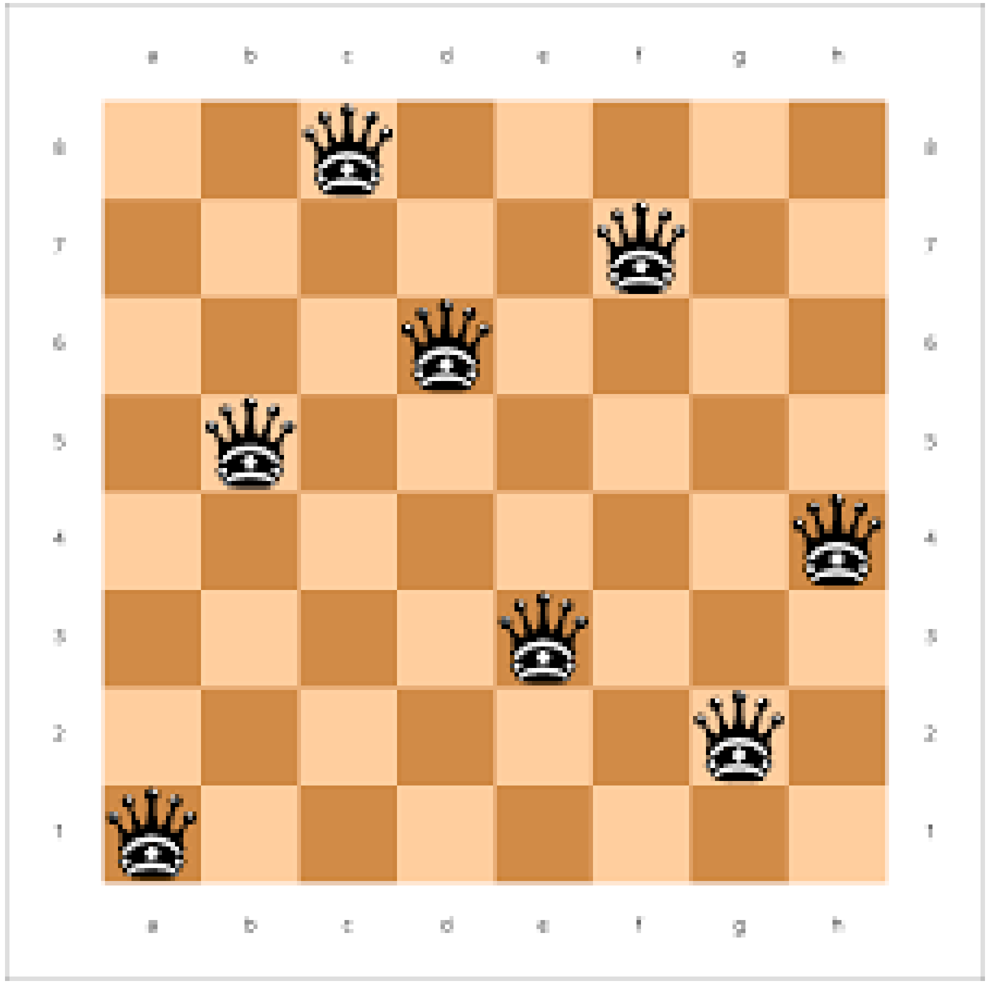
Lecture Summary

- Constraint Satisfaction Problem (covered in previous lecture)
- Backtracking Search

Constraints

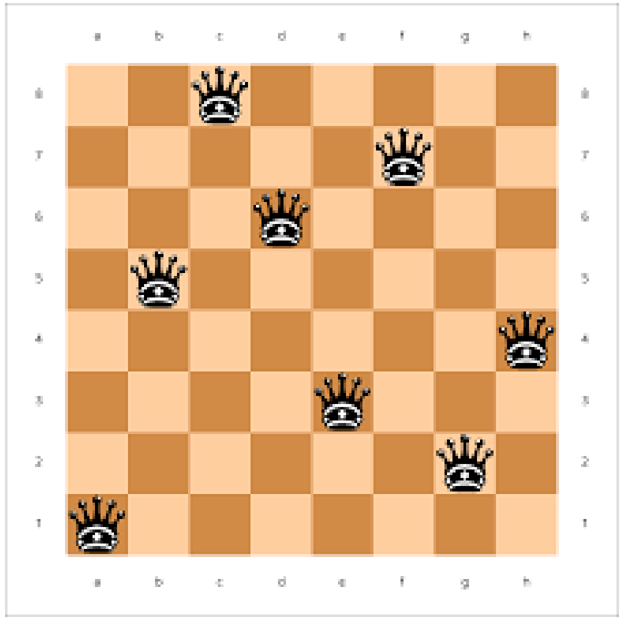
8 Queens

- Queens may not occupy the same row, column or diagonal
- How to place eight queens?



Constraint Problems

- If all constraints are satisfied, that is considered an optimal solution
- If multiple solutions exist, all correct solutions are considered equally valid



Map Colouring Problem

The Problem:

- There are regions present on a 2-dimensional surface.
- Each region may not share an adjacent edge with another region of the same colour.

Send More Money

C_4 C_3 C_2 C_1

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

Hidden 'rule'
S, M $\neq$ 0

$M + C_4 = 0 + 0$

While solving, elaborate the 'rules' you are making?

Task: Make this 'sum' make sense
Each letter represents a unique number.

Send More Money

	C_4	C_3	C_2	C_1
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

Task: Make this ‘sum’ make sense
Each letter represents a unique number.

$$
\begin{aligned}
& 1000S + 100E + 10N + 1D \\
& + 1000M + 100O + 10R + 1E \\
& = 10000M + 1000O + 100N + 10E + 1Y
\end{aligned}
$$

variables domain

$\forall \boxed{S, E, N, D, M, O, R, Y} \in \boxed{\{0, 1, \ldots, 9\}}$

constraints

$$
\boxed{
\begin{aligned}
1000S + 91E - 90N + 1D - 9000M - 900O + 10R - 1Y = 0 \\
S \mathrel{!}= 0 \\
M \mathrel{!}= 0 \\
M = 1
\end{aligned}
}
$$

S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2

9000 + 91*5 – 90*6 + 7 – 9000 – 900*0 + 10*8 – 2 = 0!

Size of the naive search space: $O(d^n)$

Domain size: d = 10
Number of variables: n = 8

Varieties of Constraints

Unary:

- Involves a single variable
- WA ≠ blue

Binary Constraints:

- Involves pairs of variables
- WA ≠ QLD

Higher Order Constraints

- Involve 3 or more variables

Preferences:

- Sometimes a constraint isn't a hard constraint, rather an optimisation
- E.g., maybe **red** is better than **blue**?
- Can be represented as a cost

Some types of CSPs

Discrete variables

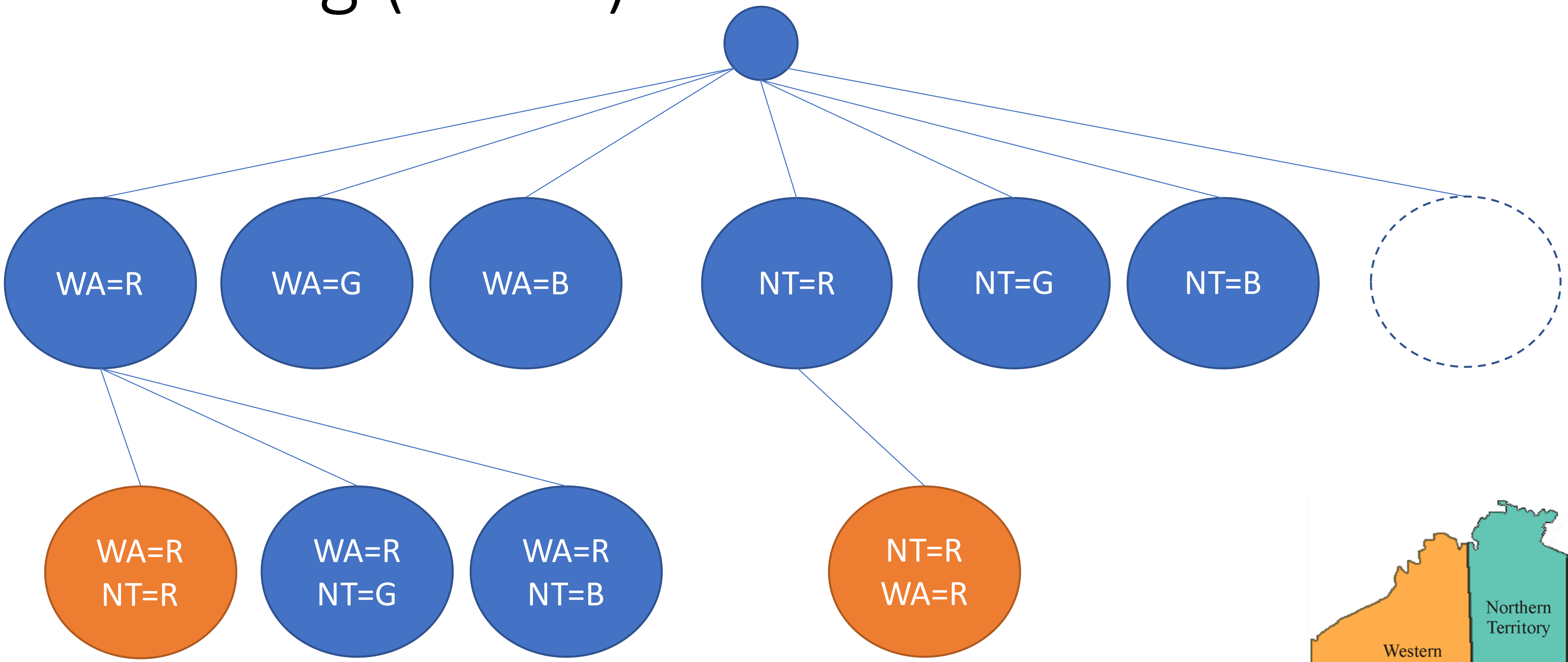
- Finite domains;
  - Size of the naive search space: $O(d^n)$
  - Boolean CSPs, Boolean satisfiability (NP-Complete problem)
- Infinite domains; (strings, integers)
  - Job scheduling (variables are the start/end days for each job)
  - Need a **constraint language** ($startJob_1 + 5 < startJob_3$ etc)

Continuous variables

- Start end times for Hubble Telescope observations
- Linear constraints are solvable in polynomial time via Linear Programming methods

Find a vector	$\mathbf{x}$
that maximizes	$\mathbf{c}^\mathsf{T}\mathbf{x}$
subject to	$A\mathbf{x} \le \mathbf{b}$
and	$\mathbf{x} \ge \mathbf{0}.$

Searching (Naïve)



Searching (Naïve)

The logic:

Initial: {} (empty)

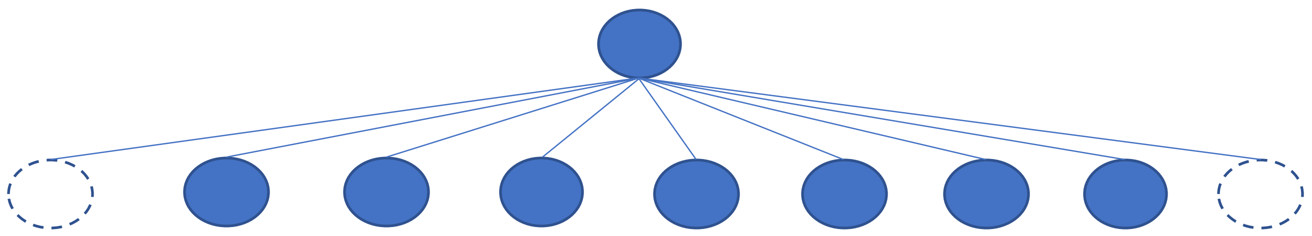
Successor: Add an unassigned value, check it does not conflict

Goal: Are we done?

All solutions appear at depth **n** with **n** variables

Some issues:

- This approach suffers from repetition
- The branching factor is large (7 * 3)
- In general, it is: (n – l) * d
    - (depth = l)
- Leaves = $n!d^n$ (oh dear)



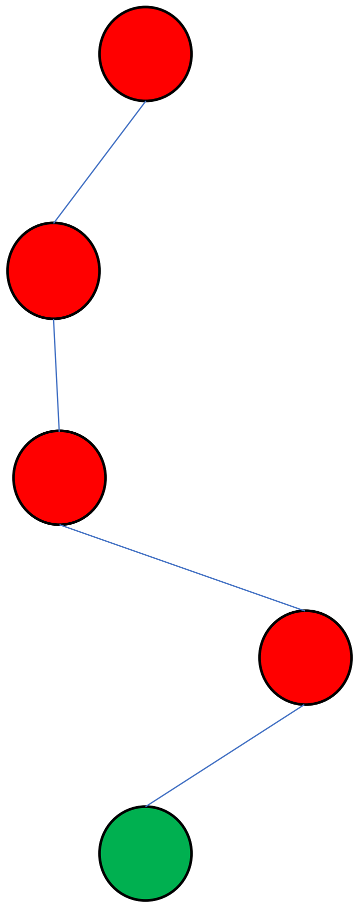
Which search would we start with?

BFS or DFS?

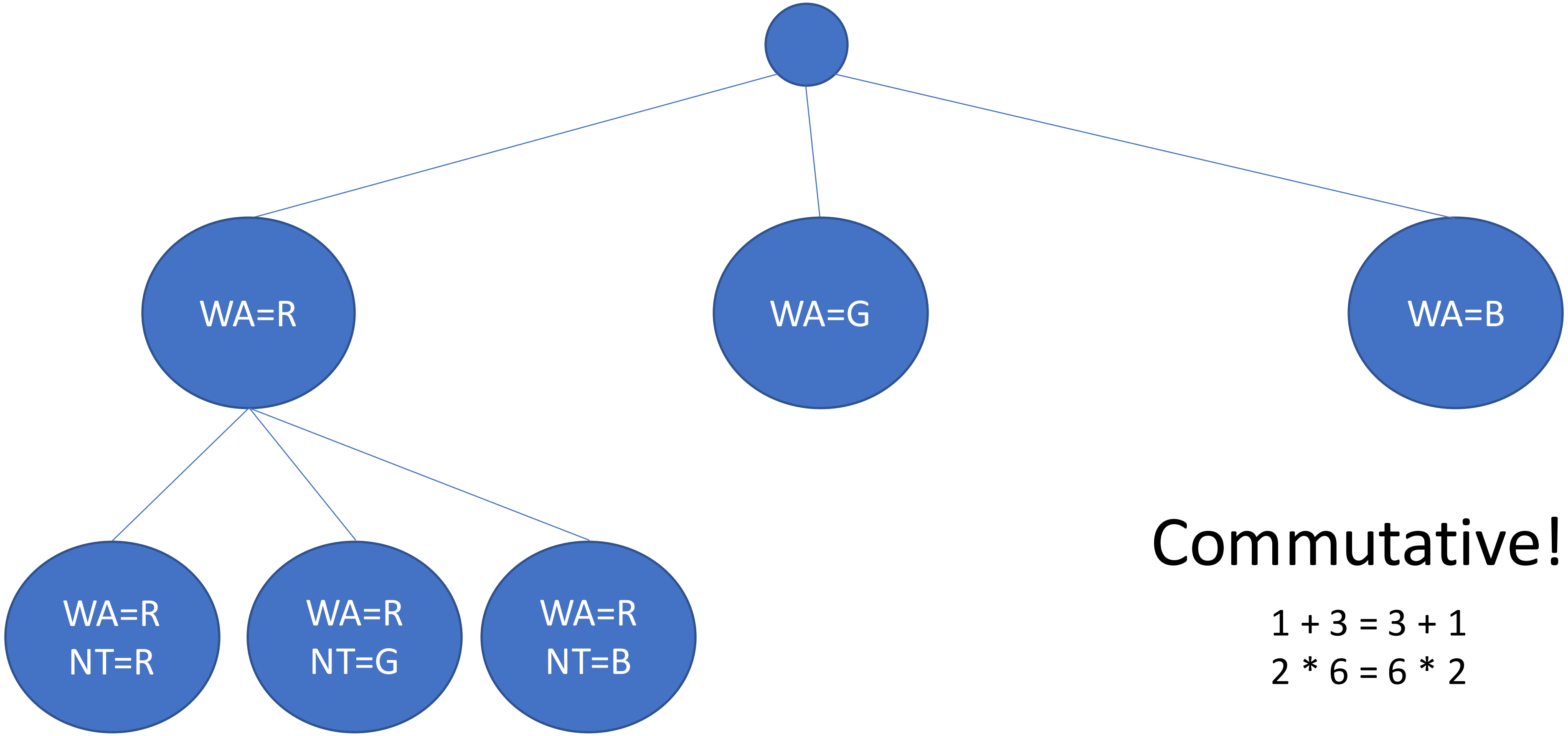
- DFS

Why?

- None of the interim states are solutions



Searching (Optimising)



WA = R, NT = B is the same as **NT = B, WA = R**

Commutative!!

1 + 3 = 3 + 1
2 * 6 = 6 * 2

2 ^ 6 ≠ 6 ^ 2

Searching (Optimising)

Approach #1

Pick a random state and expand from there

- BF is 7x3 = 21.

Approach #2

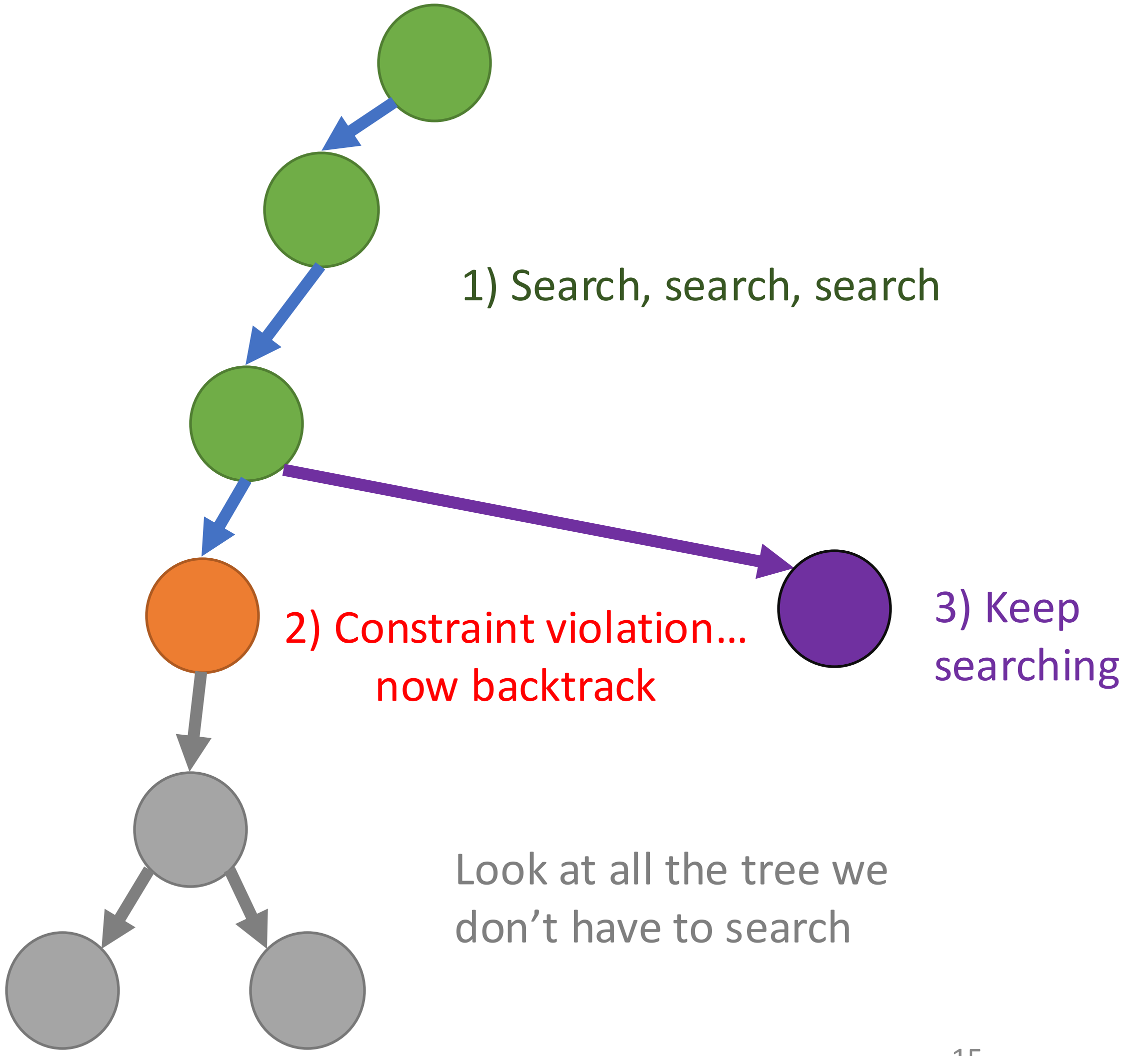
Pick a specific state, knowing that all states will eventually be picked.

- BF goes from 21 to 3.

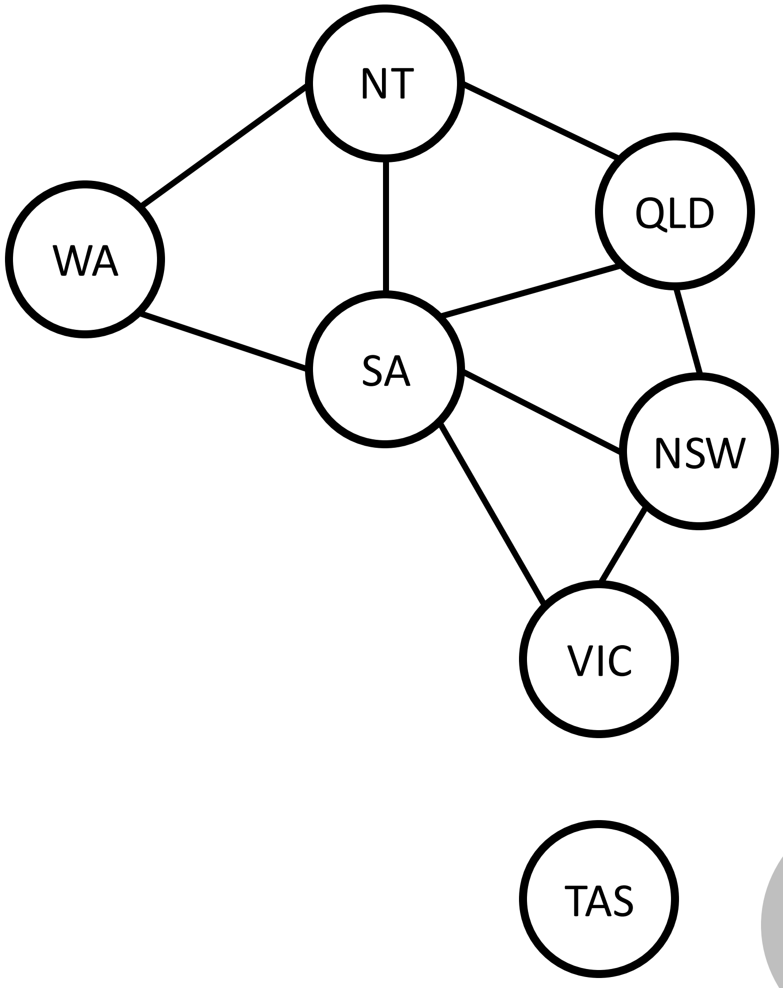
Searching with Backtracking

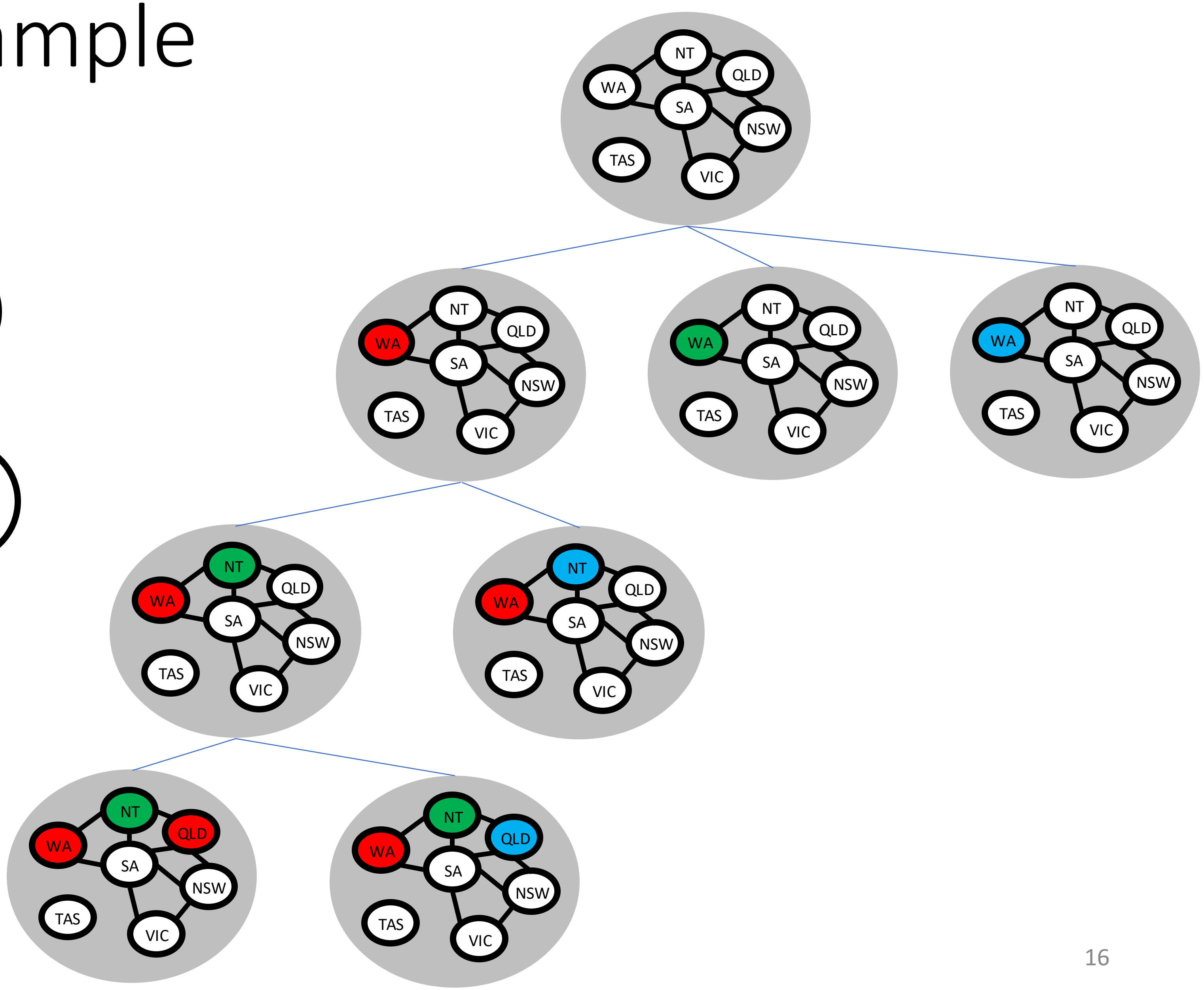
Backtracking

- In these examples we go forward… until a constraint is violated
- Then we back track to a new branch and then continue…



Backtracking example



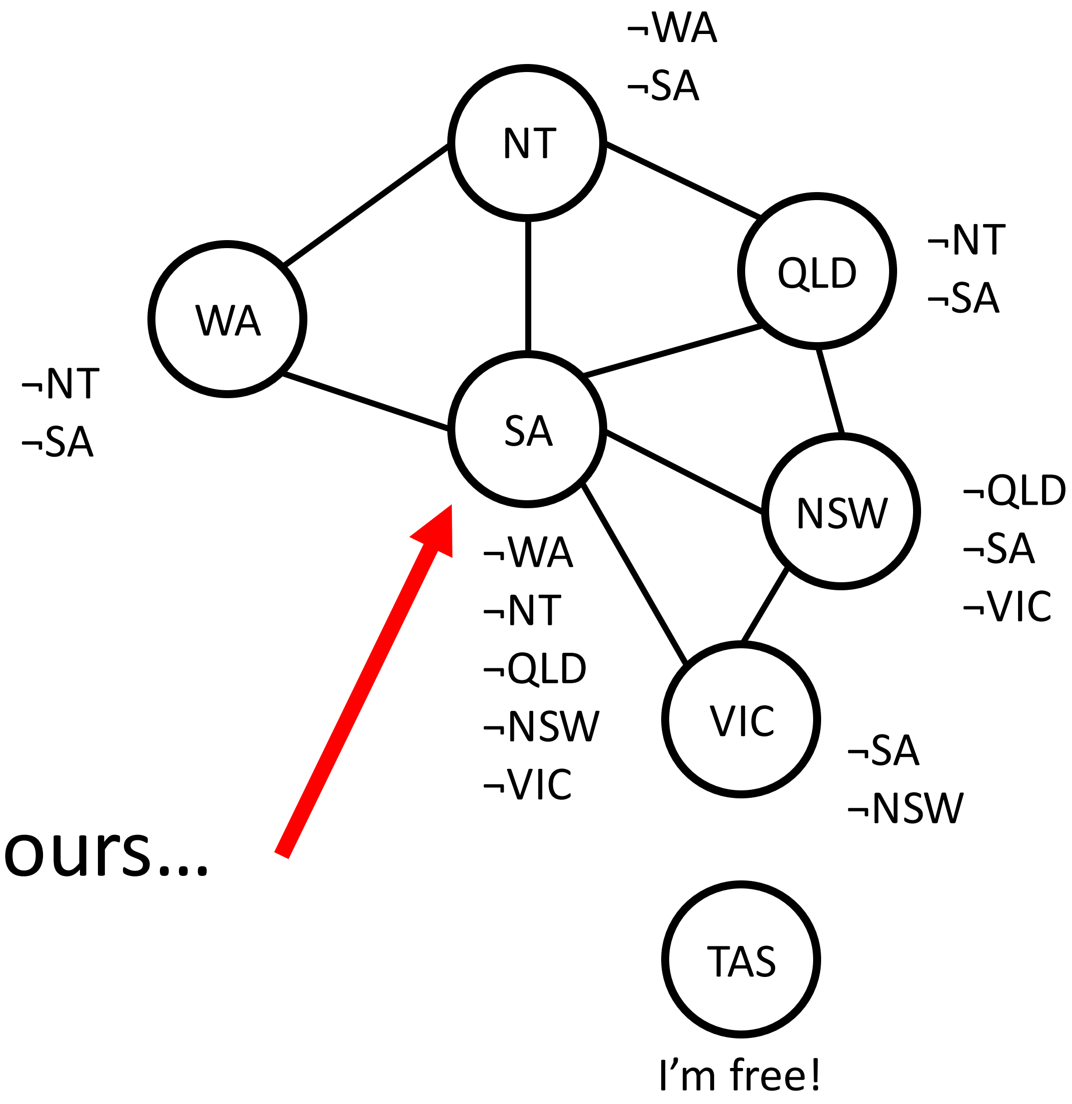


Backtracking is pretty garbage… (naively)

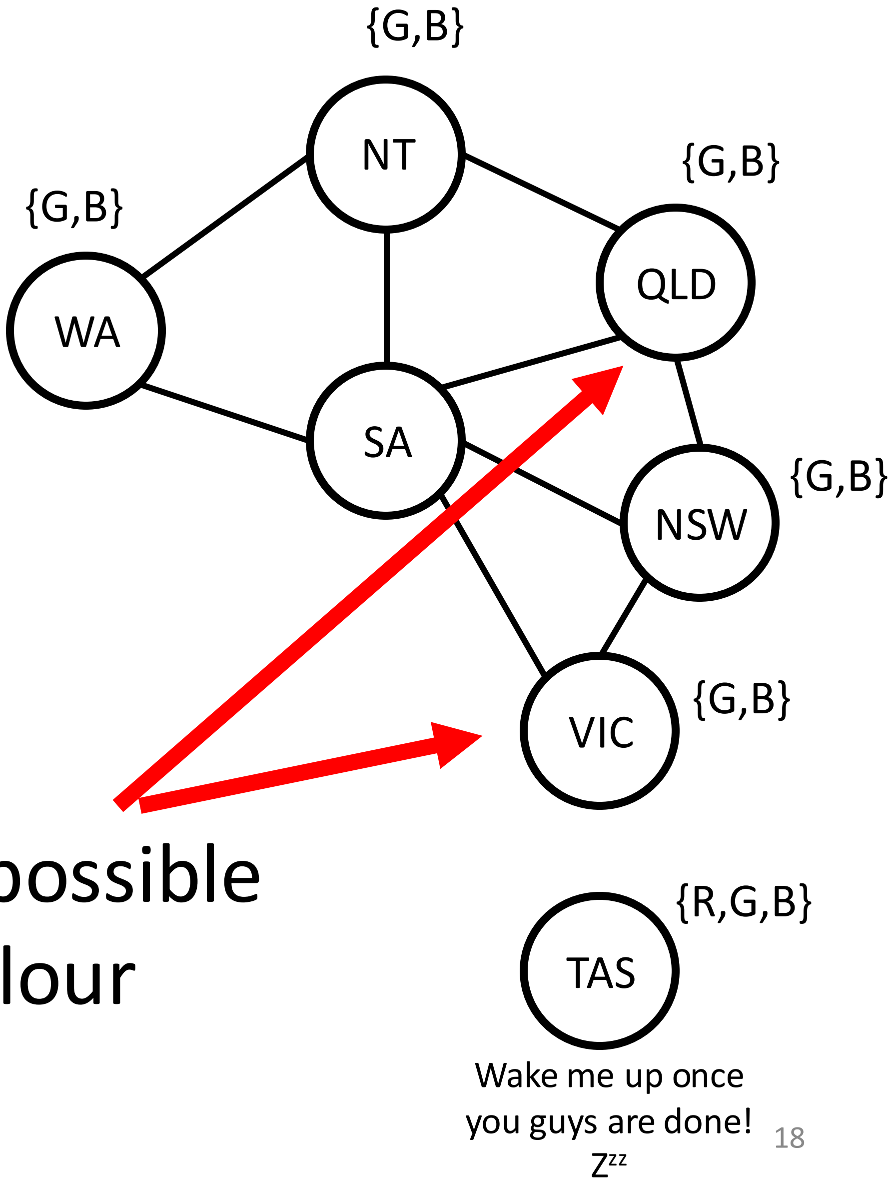
What are some ways to improve backtracking search?

- Which variable should be assigned next?
  - Variables with many constraints are good candidates
- In what order should its values be tried?
  - Start with a restrictive (or least restrictive) one?
- Can we detect inevitable failure early?
  - Meta-rules (high level strategies/heuristics)
- Can we take advantage of the problem structure?

Where to start?



5 neighbours…

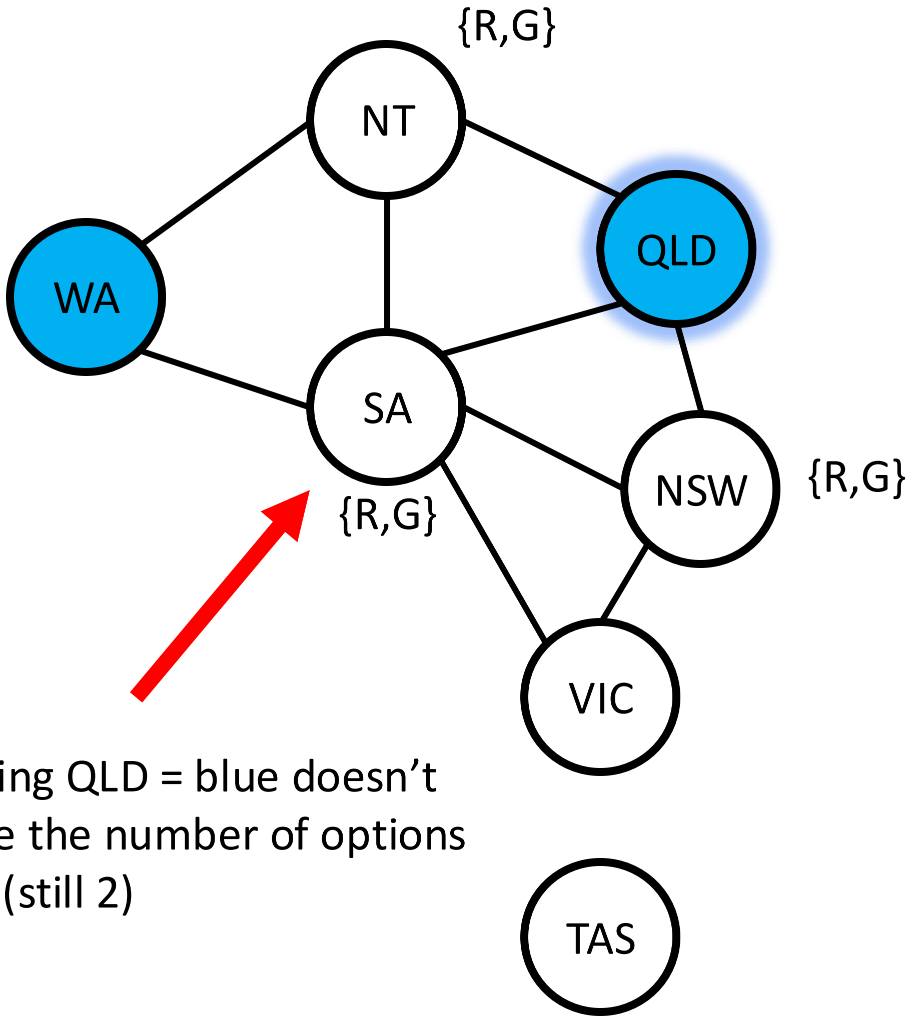


1 possible colour

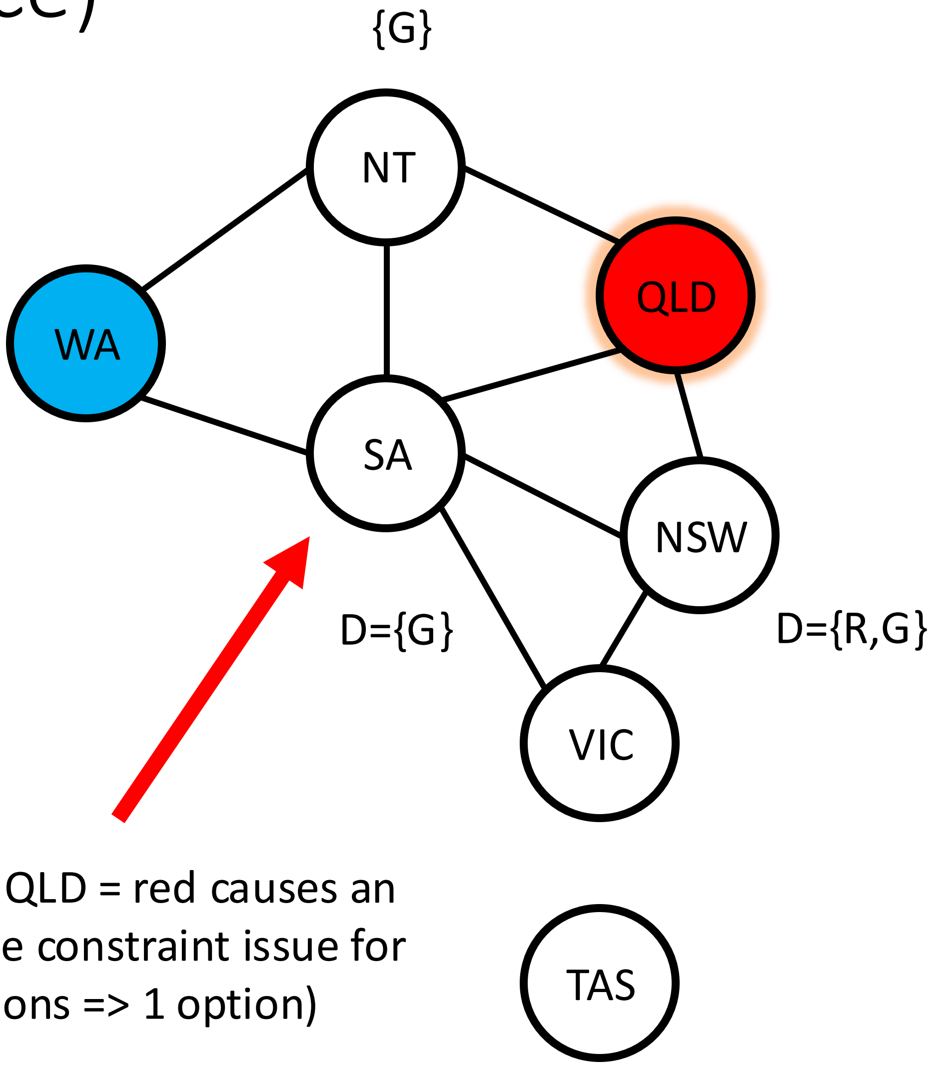
Minimal Remaining Values

- If we search through the remaining choices, and then go for the one with the fewest possible options
  - In Sudoku, this is often a cell with only one possible value
- Searching through all of the nodes will only be O(n) rather than O(n *??) so finding the 'most constrained' node is cheap.

Constraint (Least Constraining Choice)



Choosing QLD = blue doesn’t change the number of options for SA (still 2)



Choosing QLD = red causes an immediate constraint issue for SA (2 options => 1 option)

Arc Consistency

A variable in a CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints.

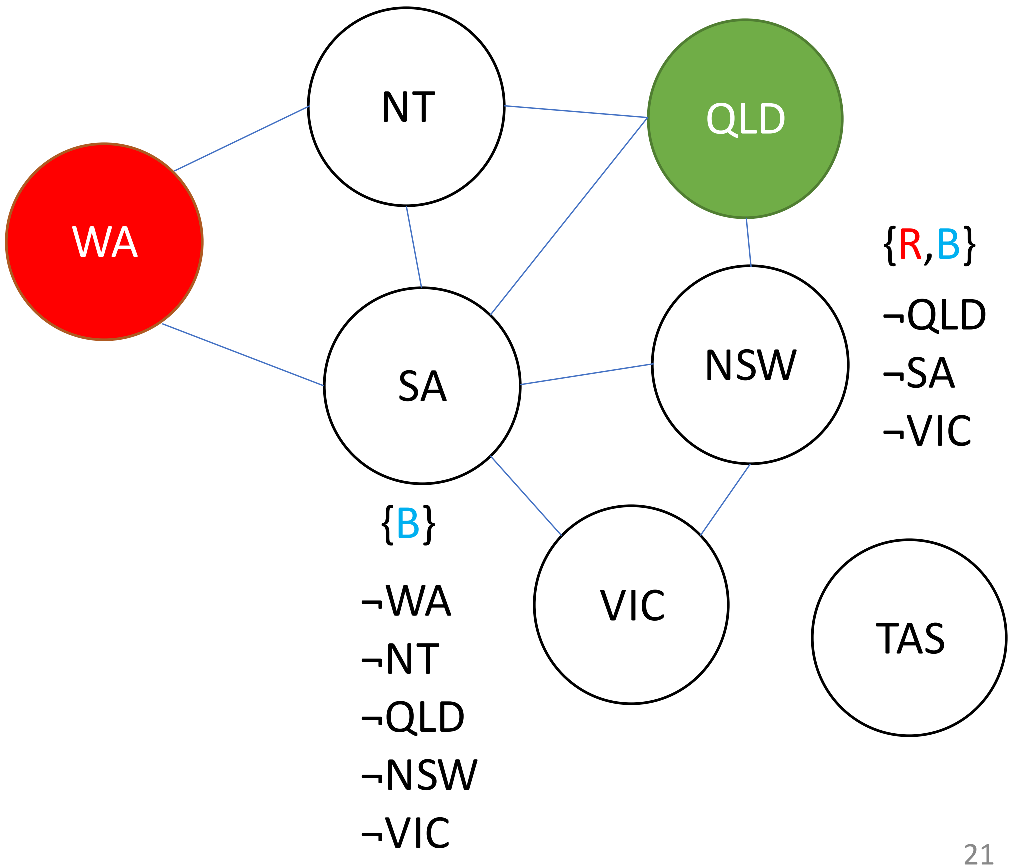
Example

For each possible value of **SA** {B}, is there a *possible* value for **NSW**?

- Yes, **NSW** -> {**R**}
- SA is arc-consistent w.r.t. NSW.

For each possible value of **NSW** {**R**, **B**}, is there a *possible* value for **SA**?

- No, if **NSW** is B, **SA** has no legal values, **SA** -> {}.
- NSW is NOT arc-consistent w.r.t. SA



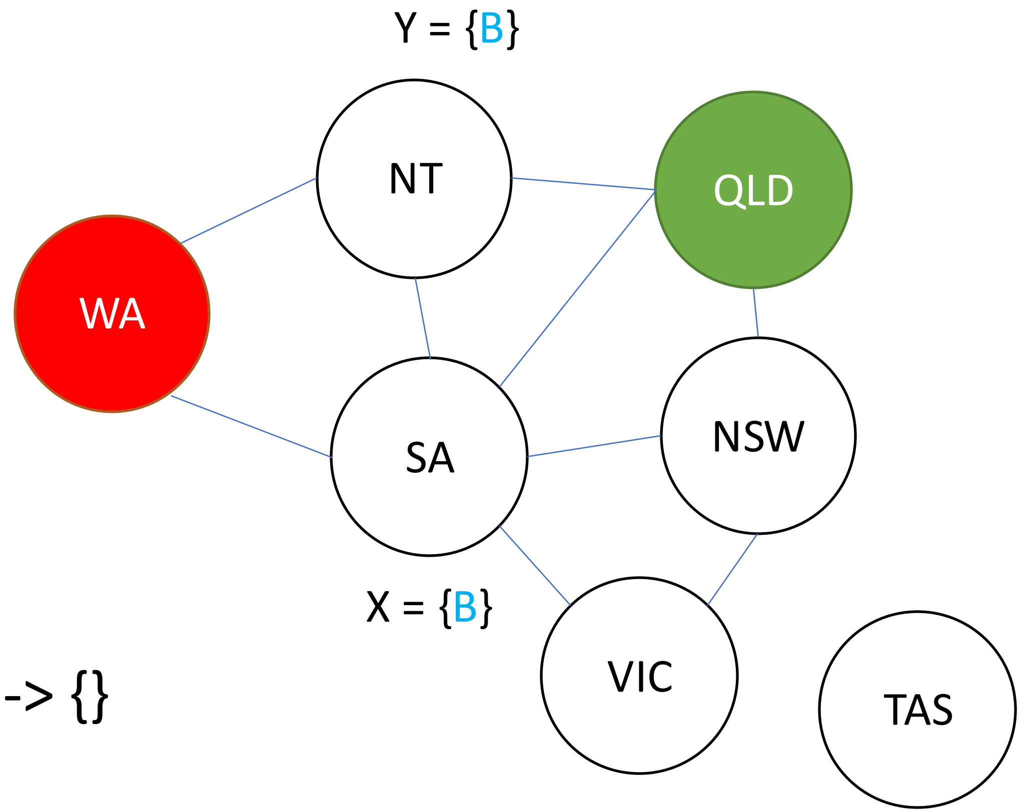
Arc Consistency

Simplest form of propagation makes each **arc consistent**

X -> Y is consistent iff

For **every** value of x of X there is **some** allowed y

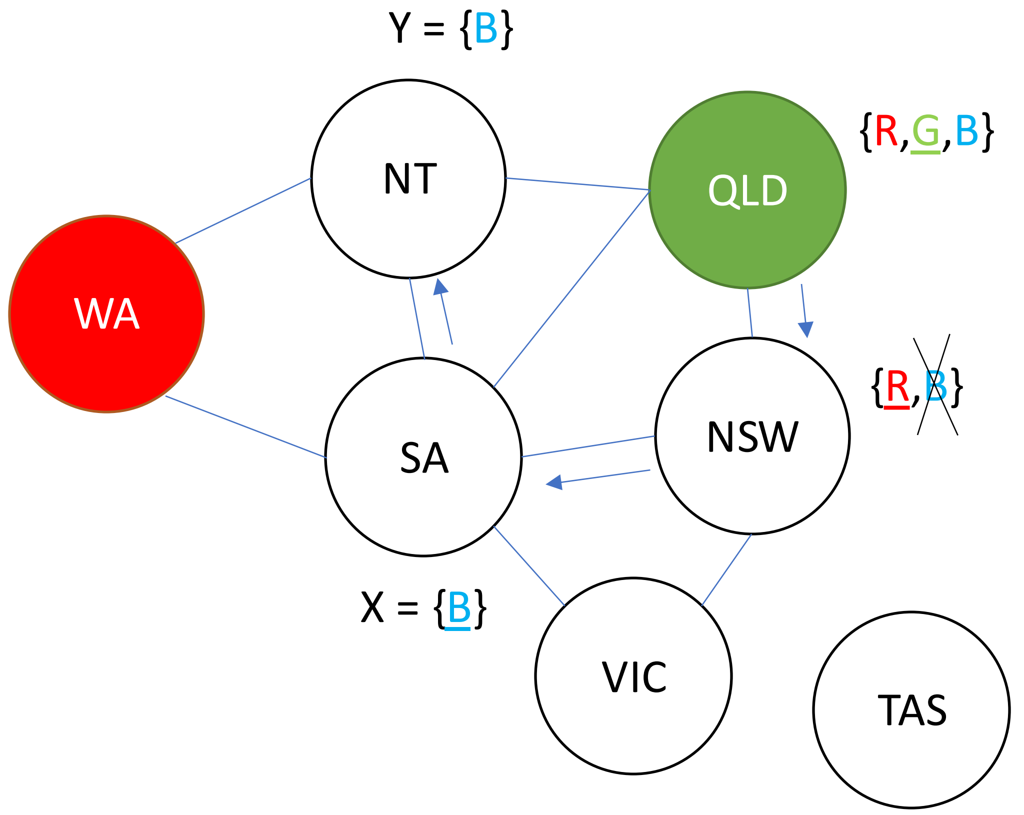
If SA is **B**, there is no allowed value for NT -> {}



Arc Consistency

Arc Consistencies can propagate

- If State X is value x_i… then State Y can't be value y_j
- If some other state denies X one of its potential values… say x_k, other nodes relying on x_k (i.e., adjacent nodes need to be rechecked)



Order: WA->QLD->NSW->SA->NT

AC-3

function AC-3 **returns** the CSP

inputs: csp, a binary CSP with variables {X_1, X_2, …}

local: queue (list of arcs, initially all in the csp)

while queue is not empty **do**

(X_i, X_j) <= remove first

if removeInconsistentValues(X_i, X_j) **then**

for each X_k in Neighbours[X_i] **do**

add(X_k, X_i) to queue

function removeInconsistentValues(X_i, X_j) **returns** true/false

removed <- false

for each x **in** DOMAIN[X_i] **do**

if no value y in DOMAIN[X_j] allows (x, y) to satisfy $X_i \leftrightarrow X_j$

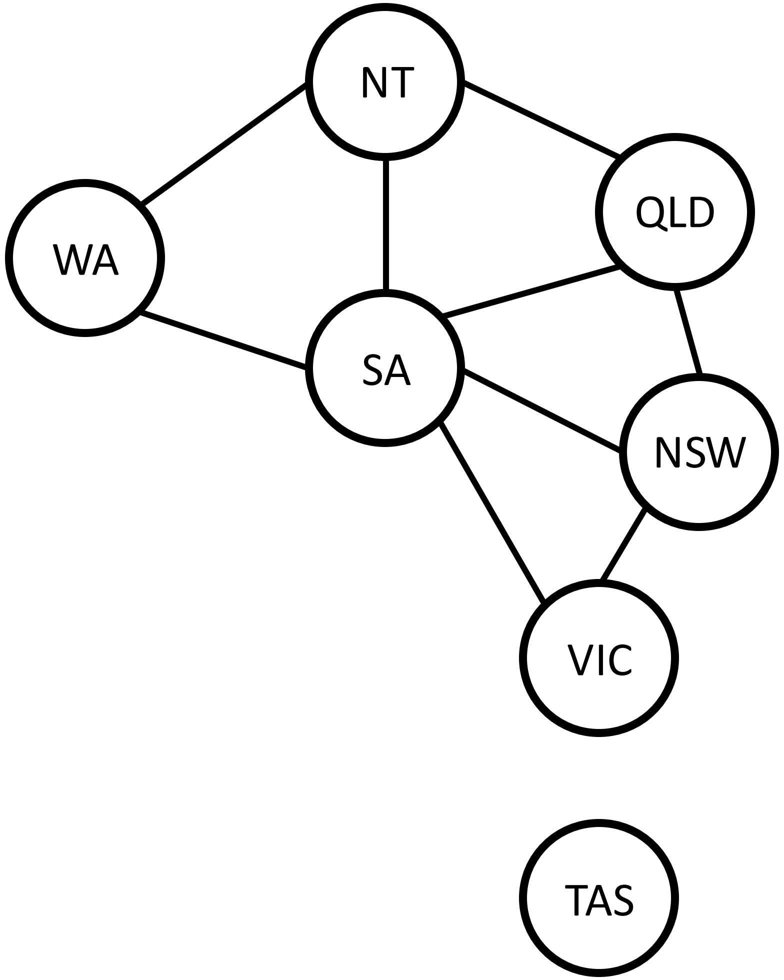
then delete x from DOMAIN[X_i]; removed <- true

return removed

In Human-speak:

1. Make a queue of all edges
2. Go through one by one, looking for inconsistent values
3. Check each edge, checking each domain in turn
4. When an inconsistency arises, remove the domain (colour etc.) and then re-add any edges adjacent to the affected node (to propagate any inconsistencies)

Forward Checking



Forward Check: **FAIL**

For each unassigned variable Y that is connected to X by a constraint, delete from Y ’s domain any value that is inconsistent with the value chosen for X.

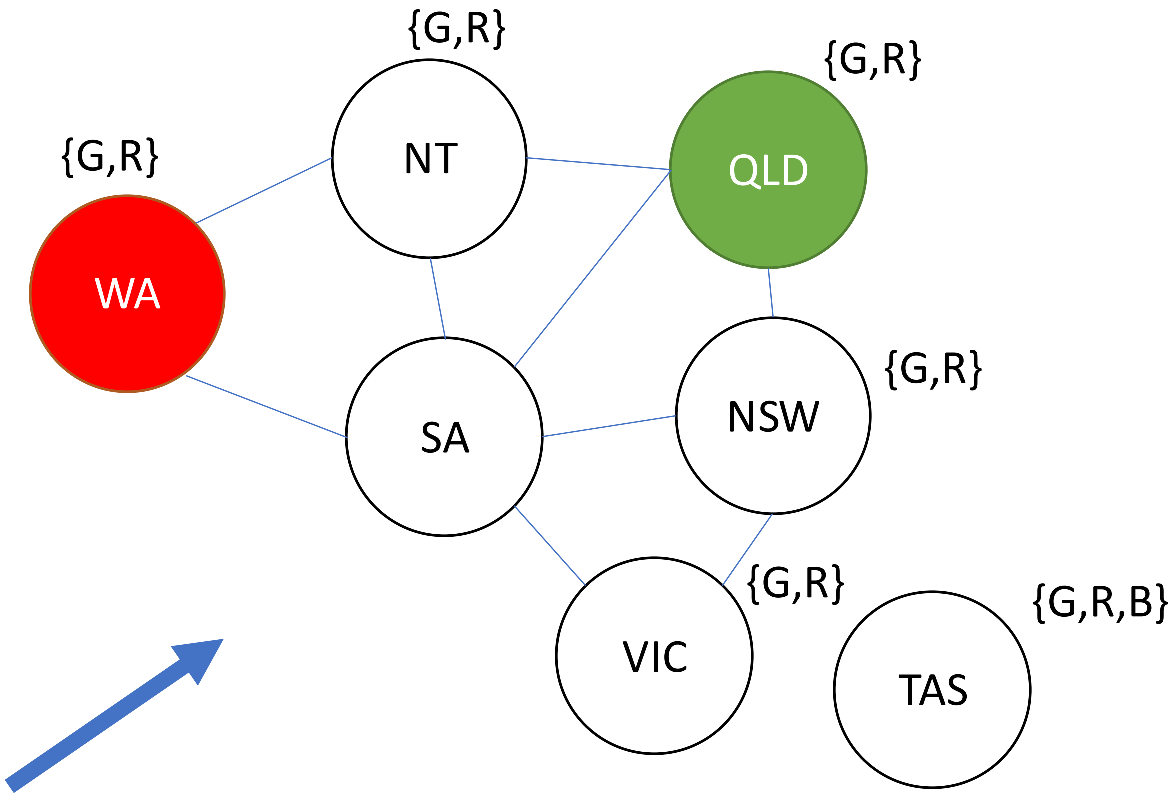
Tree Structured CSPs

Problems with no loops have more efficient solutions:

- $O(n) x O(d^2) = (nd^2)$

This compares to normally

- $O(d^n)$ - size of the naive search space



Domain size: d
Number of variables: n
Number of arcs: n - 1

Can you turn a problem into a tree?

Real World CSPs

- Timetabling (which class in which room?)
  - Constraint: You can’t have two classes in the same room
- Transportation & Logistics
- Factory scheduling
- Floorplanning
- Assignment problems (who has what task?)
  - Constraint: One person cannot do two tasks simultaneously

Questions

What are your questions?