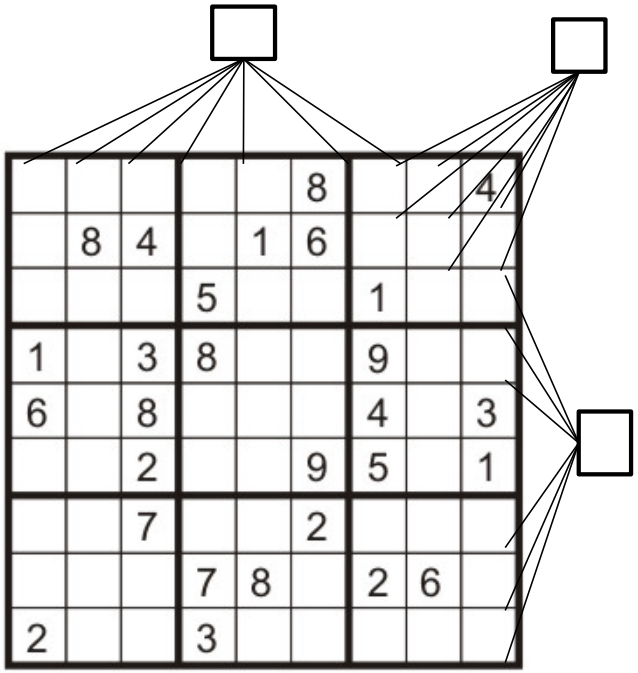
Artificial Intelligence

Constraint Satisfaction Problems



Jay Urbain, PhD

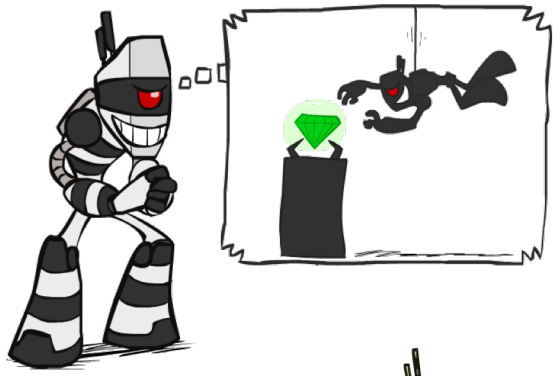
Credits: Stuart Russel, Peter Norvig, AIMA

Dan Klein, Pieter Abbeel, University of California, Berkeley

What is Search For?

- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

- Planning problems: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

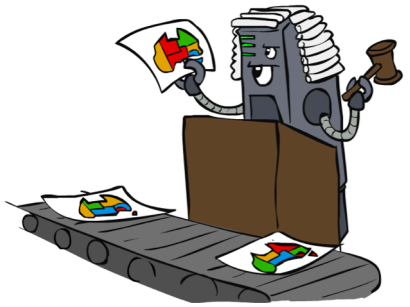


- Identification problems: assignments to variables
  - The goal itself is important, not the path
  - All paths are at the same depth (for some formulations)
  - CSPs are a specialized class of identification problems

Constraint Satisfaction Problems

- Standard search problems:
  - State is a "black box": arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

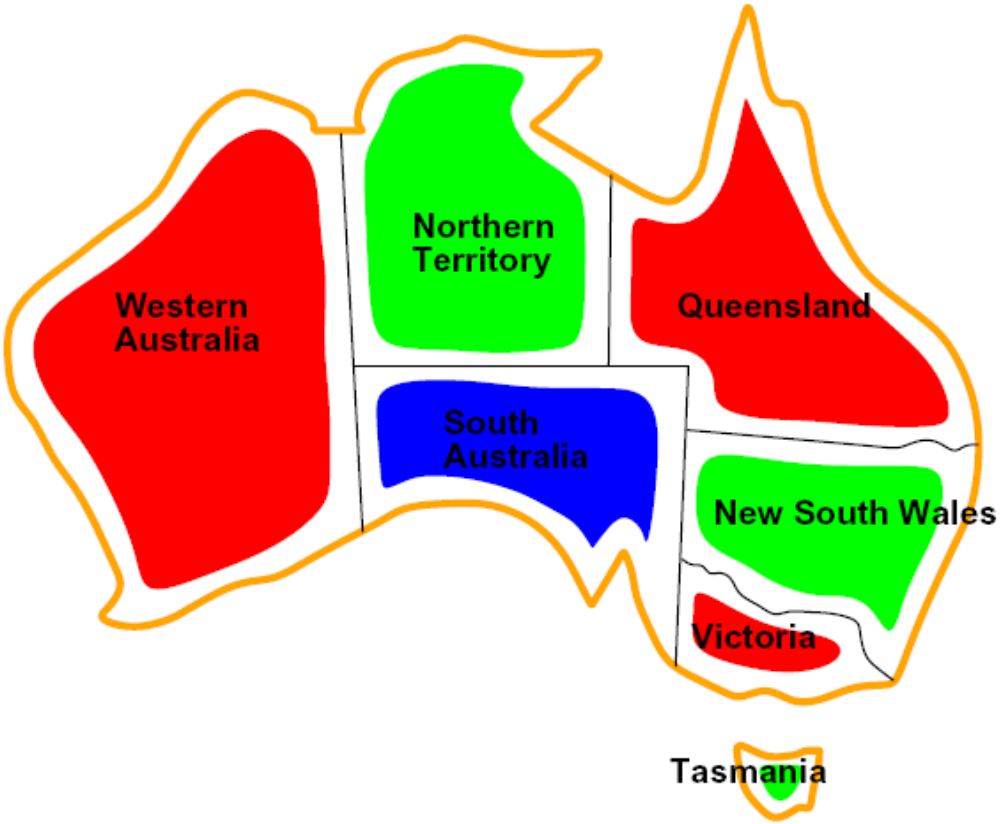
- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables (judge)



- Simple example of a *formal representation language*

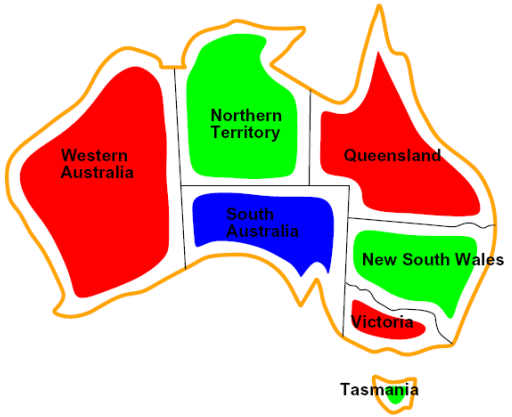
- Allows useful general-purpose algorithms with more power than standard search algorithms

CSP Examples



Example: Map Coloring

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{red, green, blue\}$
- Constraints: adjacent regions must have different colors

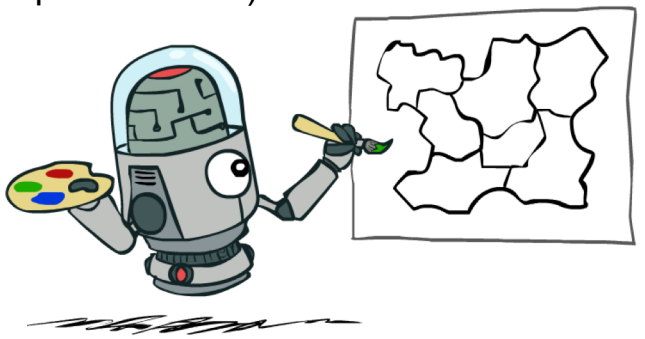


Implicit: $WA \neq NT$ (Snippet of code you execute)

Explicit: $(WA, NT) \in \{(red, green), (red, blue), \ldots\}$ (logic representation)

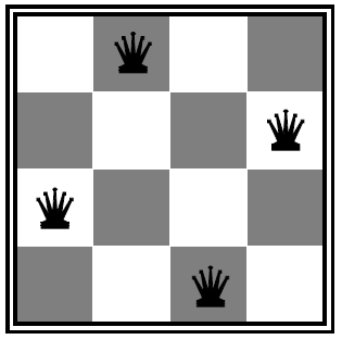
- Solutions are assignments satisfying all constraints, e.g.:

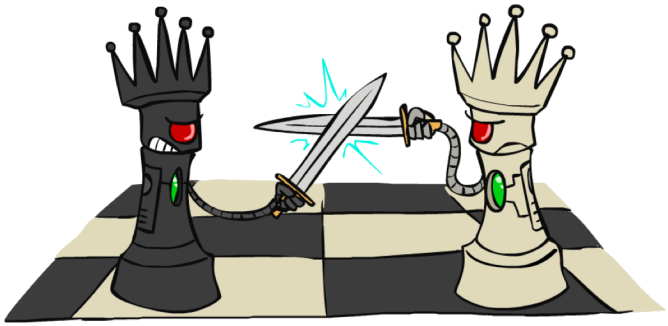
{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}



Example: N-Queens

- Formulation 1:
  - Variables: X_{ij}
  - Domains: $\{0, 1\}$
  - Constraints





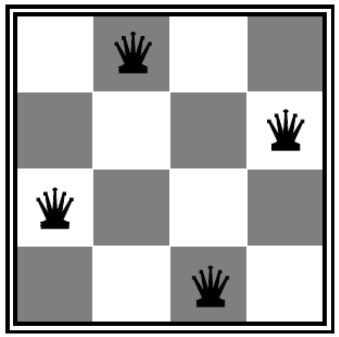
Can't threaten vertically, horizontally, or diagonally.

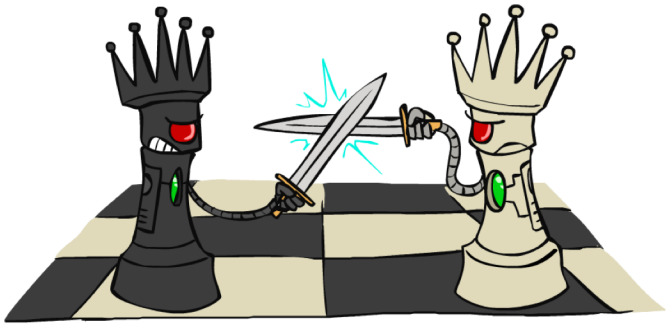
$$\forall i, j, k \;\; (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

Constraints say what configurations are allowed.
Can't threaten vertically, horizontally, or diagonally.
What's the easy solution?

Example: N-Queens

- Formulation 1:
  - Variables: X_{ij}
  - Domains: $\{0, 1\}$
  - Constraints





Can't threaten vertically, horizontally, or diagonally.

$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

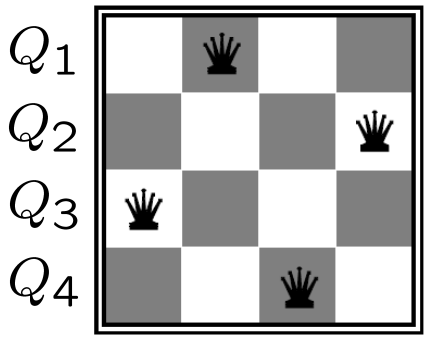
$$\sum_{i,j} X_{ij} = N$$

All queens are placed

Constraints say what configurations are allowed.
Can't threaten vertically, horizontally, or diagonally.
What's the easy solution?

Example: N-Queens

- Formulation 2: What column is Queen in?
  - Variables: Q_k
  - Domains: $\{1, 2, 3, \ldots N\}$
  - Constraints:

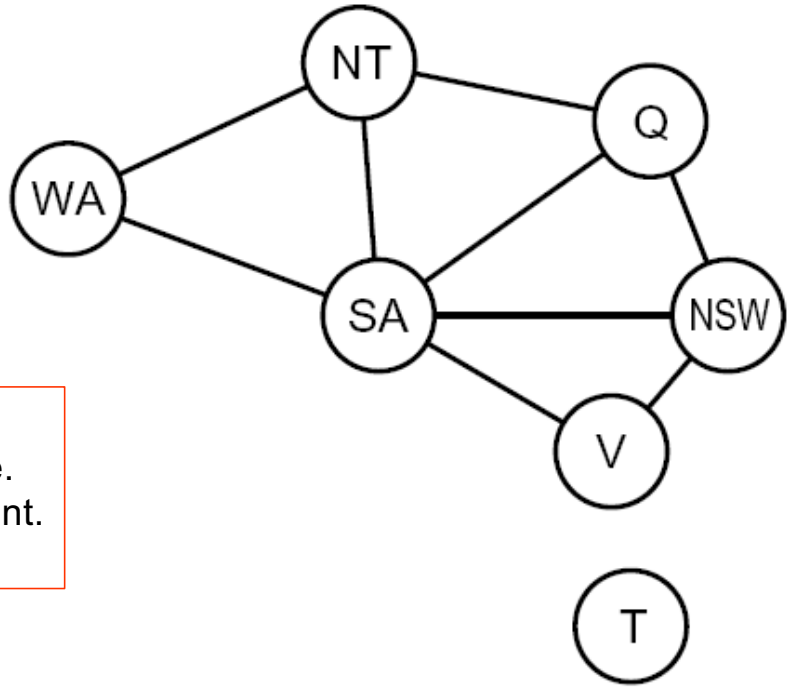


Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$

$\ldots$

Constraint Graphs

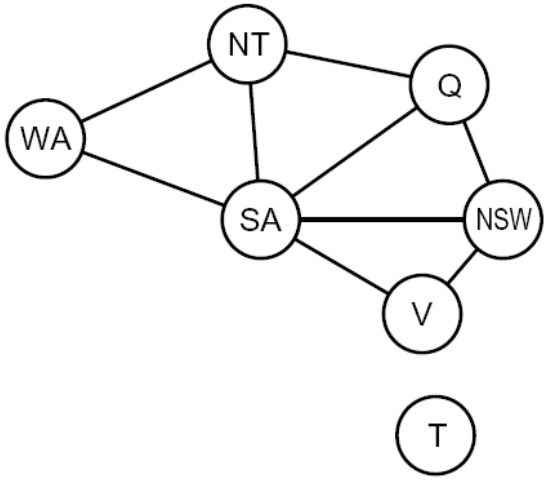


Australia:

- Node for each variable.
- Edge for each constraint.

Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



http://aimacode.github.io/aima-javascript/6-Constraint-Satisfaction-Problems/#csp-with-map-coloring

Example: Cryptarithmetic

- Variables:

  $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$

- Domains:

  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

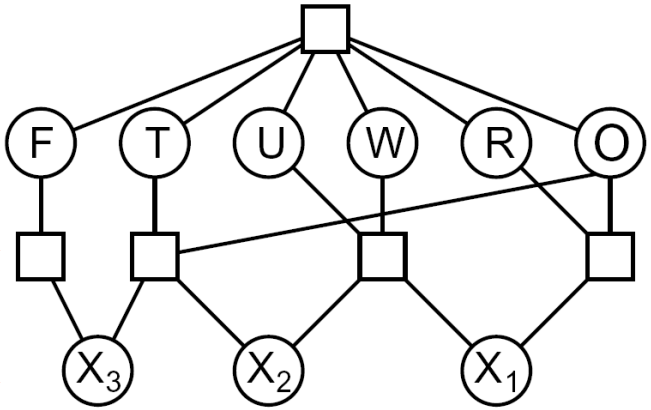
- Constraints:

  alldiff(F, T, U, W, R, O)

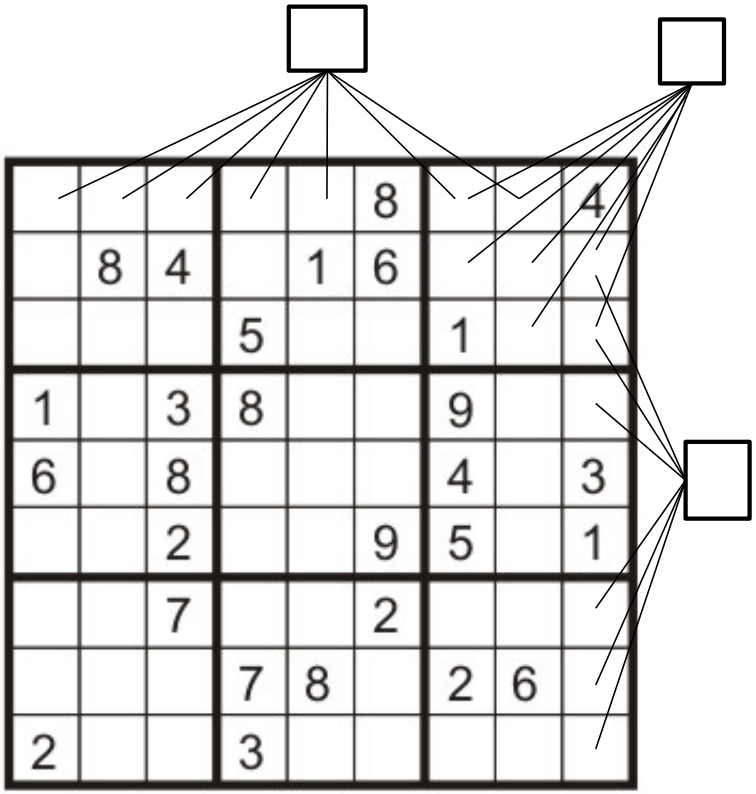
  O + O = R + 10 · X1
  X1 + W + W = U + 10 · X2
  X2 + T + T = O + 10 · X3
  X3 = F

```
  T W O
+ T W O
-------
F O U R
```

A number puzzle in which a group of arithmetical operations has some or all of its digits replaced by letters or symbols, and where the original digits must be found. In such a puzzle, each letter represents a unique digit.



Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

  9-way alldiff for each column

  9-way alldiff for each row

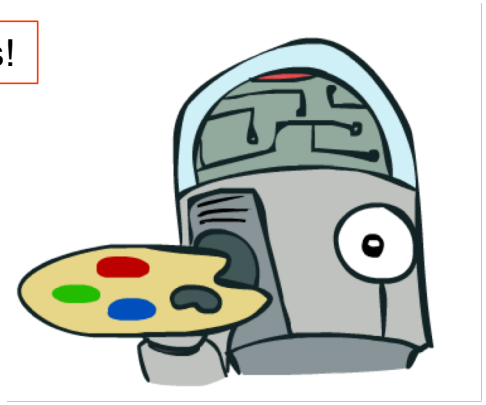
  9-way alldiff for each region

  (or can have a bunch of pairwise inequality constraints)

Varieties of CSPs

- Discrete Variables (focus)
  - Finite domains
    - Size d means $O(d^n)$ complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable*

Exponential in the number of variables!



- Continuous variables
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by LP methods

**an undecidable problem is a decision problem for which it is proved to be impossible to construct an algorithm that always leads to a correct yes-or-no answer*

Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

    $$SA \neq green$$

  - Binary constraints involve pairs of variables, e.g.:

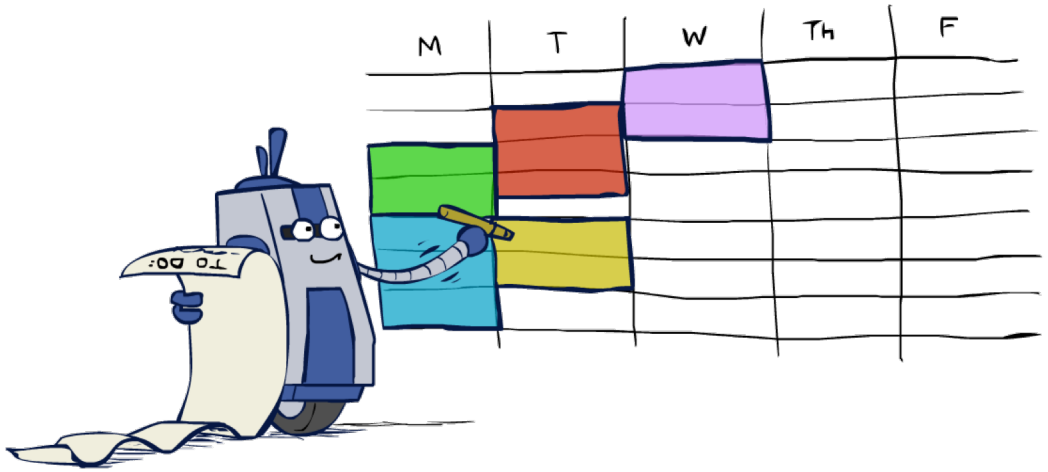
    $$SA \neq WA$$

  - Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints

- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (Use Bayes' nets)

Real-World CSPs

- Scheduling problems: e.g., when can we all meet?
- Timetabling problems: e.g., which class is offered when and where?
- Assignment problems: e.g., who teaches what class
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!



- Many real-world problems involve real-valued variables…

Solving CSP Problems: Standard Search Formulation

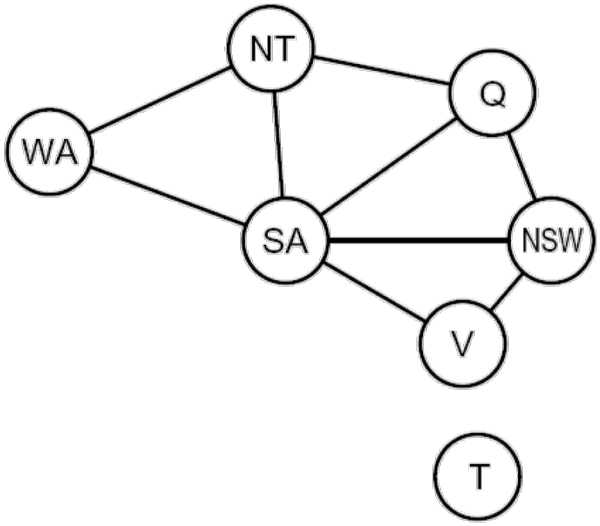
- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

- Start with the straightforward, naïve approach, then improve it

Search Methods

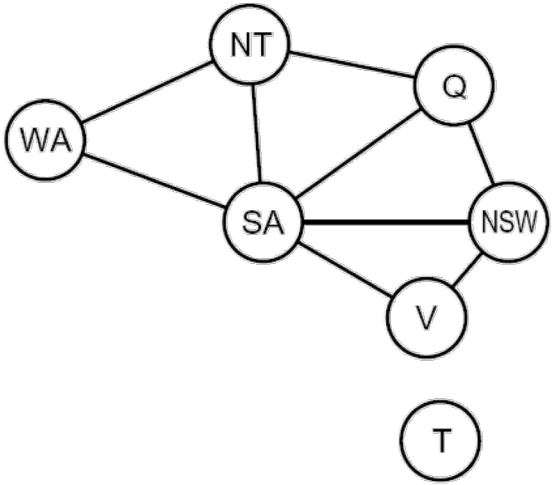
- What would BFS do?
- What would DFS do?
- What problems does naïve search have?



Settings:
Graph = Simple
Algorithm = Naïve Search
Ordering = None
Filtering = None

Naïve Search Methods

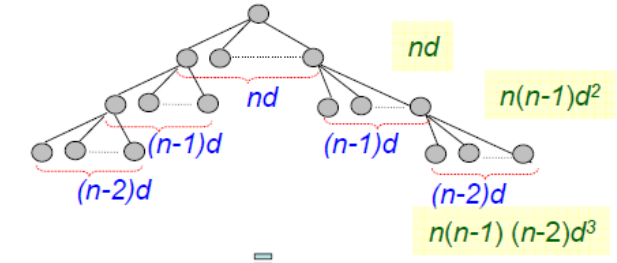
- What would BFS do?
  - Open all nodes to test for goal
  - All solutions at bottom with all nodes open
- What would DFS do?
  - Recursively visit each terminal state
- What problems does naïve search have?
  - Clearly need to test constraints as we go



Settings:
Graph = Simple
Algorithm = Naïve Search
Ordering = None
Filtering = None

Standard Search Approach

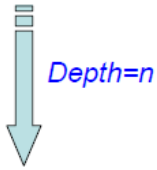
- Can formulate as standard search problem – but order not important!
- If **incremental formulation** is used
- Breadth-first search with search tree with depth limit *n*



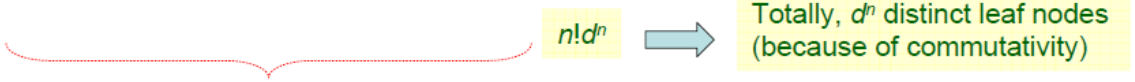
Initial state: empty assignment {}

Successor function: a value can be assigned to any unassigned variables, provided that no conflict occurs

Goal test: the assignment is complete



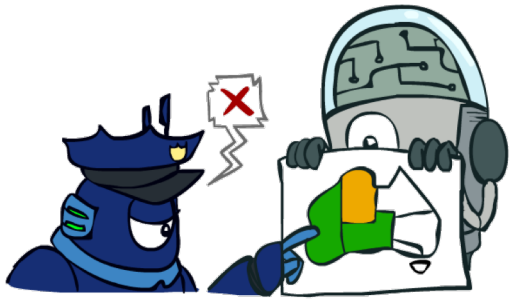
Variables (n) and Values (d)



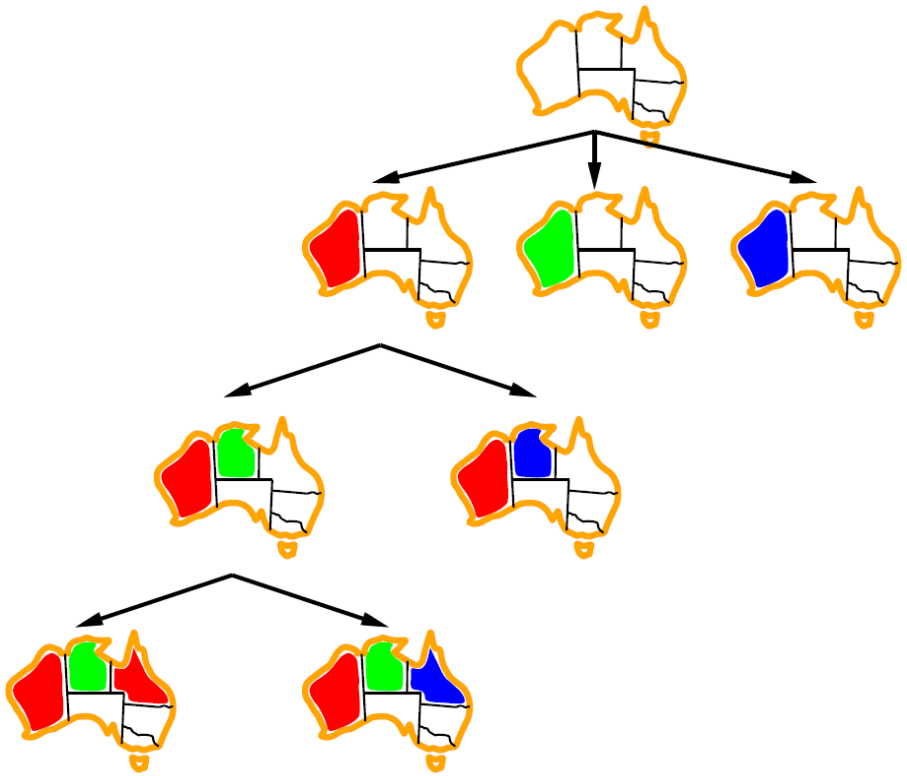
  - Every solution appears at depth n with n variable assigned
  - DFS (or depth-limited search) also can be applied (smaller space requirement)

Backtracking Search – check constraints as we go!

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict with previous assignments
  - Might have to do some computation to check the constraints
  - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for n ≈ 25



Backtracking Example



Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Settings:
Graph = Simple
Algorithm = Backtracking
Ordering = None
Filtering = None

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

[http://localhost:8888/notebooks/aima-python/csp.ipynb]

Improving Backtracking

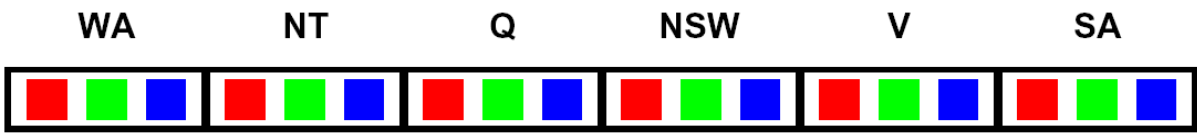
- General-purpose ideas give huge gains in speed
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?

Filtering

- Filtering is about ruling out variables

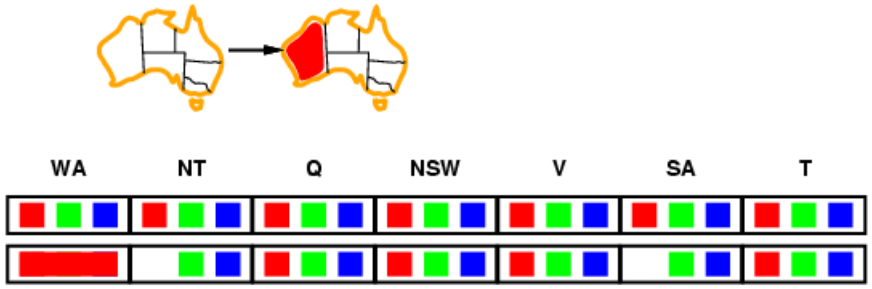
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



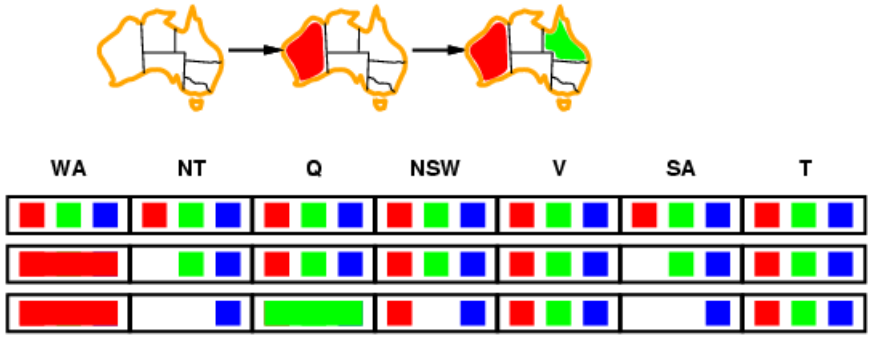
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



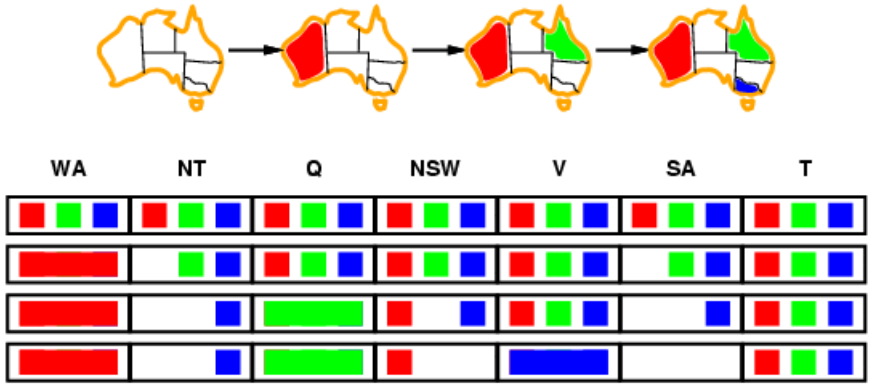
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



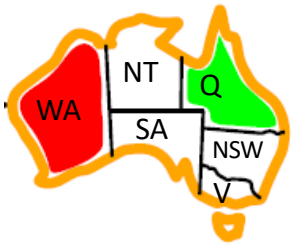
Forward checking

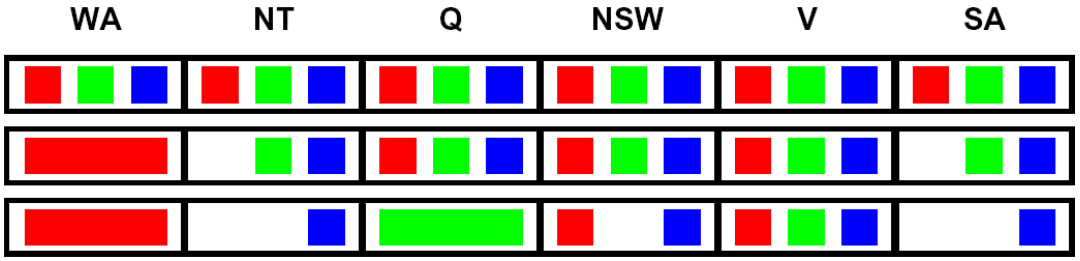
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



Filtering: Need Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



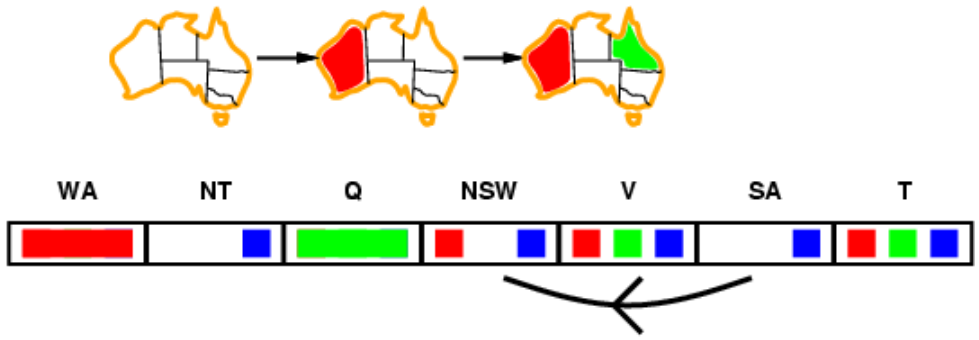


- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Need Constraint propagation:* reason from constraint to constraint

Arc consistency

- A variable is arc-consistent if every value in its domain satisfies the variable's *binary* constraints.
- Simplest form of propagation makes each arc *consistent*
- *X* → *Y* is consistent iff

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

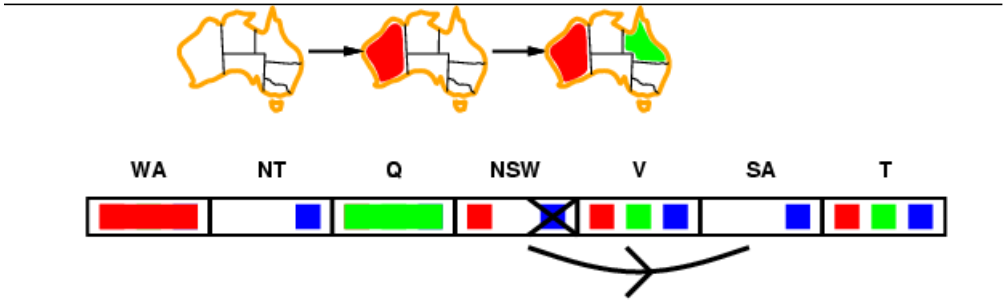


Remember: Delete from the tail!

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

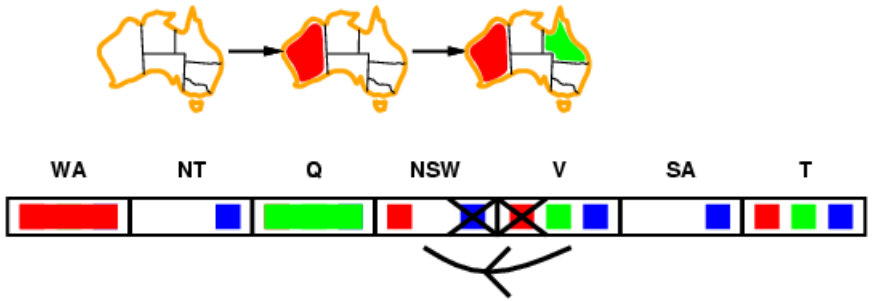
  for every value x of X there is some allowed y



Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y

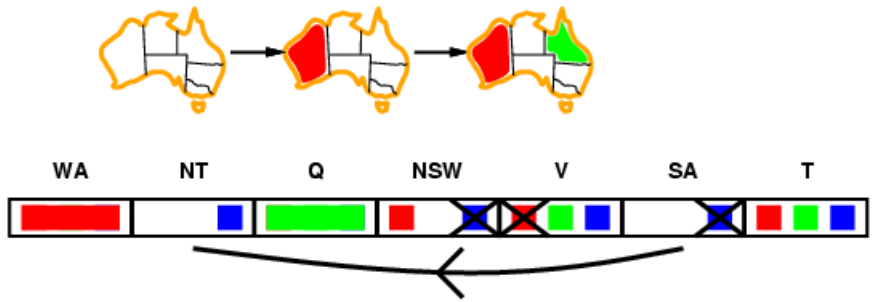


- If X loses a value, neighbors of X need to be rechecked

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed y



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Enforcing Arc Consistency in a CSP

Settings:
Graph = Complex
Algorithm = Backtracking
Ordering = None
Filtering = Forward Checking (first) / Arc Consistency (second)

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  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 REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue

---

function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

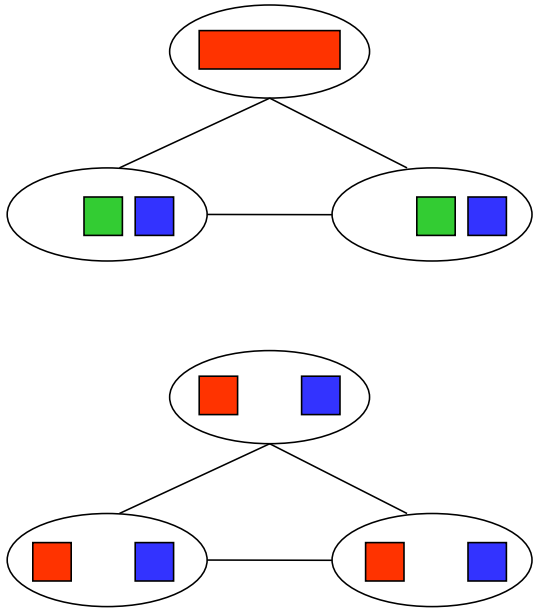
- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard

[Demo: n-queens]

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

- Arc consistency still runs inside a backtracking search!

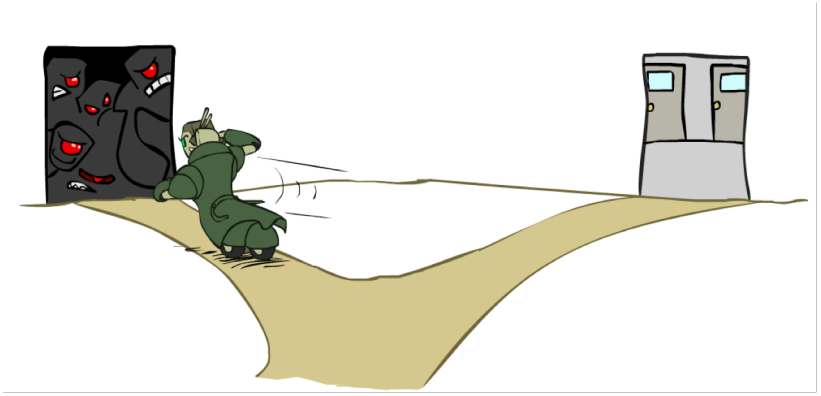


What went wrong here?

Ordering: Minimum Remaining Values

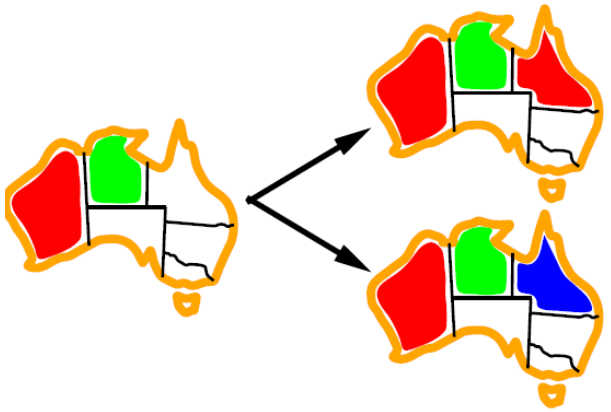
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering



Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



- Why least rather than most?

- Combining these ordering ideas makes 1000 queens feasible

Settings:
Graph = Complex
Algorithm = Backtracking
Ordering = MRV
Filtering = Forward Checking