Artificial Intelligence

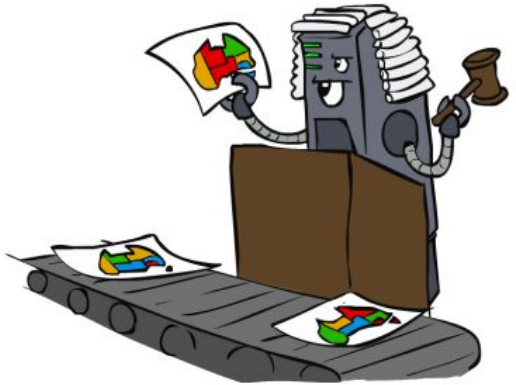
Lec 13: Constraint Satisfaction Problems

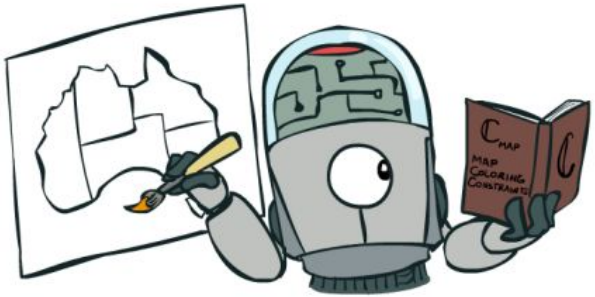
Pratik Mazumder

Content taken from various sources including
CS188 Intro to AI at UC Berkeley

Constraint Satisfaction Problems

- Standard search problems:
    - state is a “black box”—any data structure that supports goal test, eval, successor
    - Goal test can be any function over states.
- 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.
- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms.

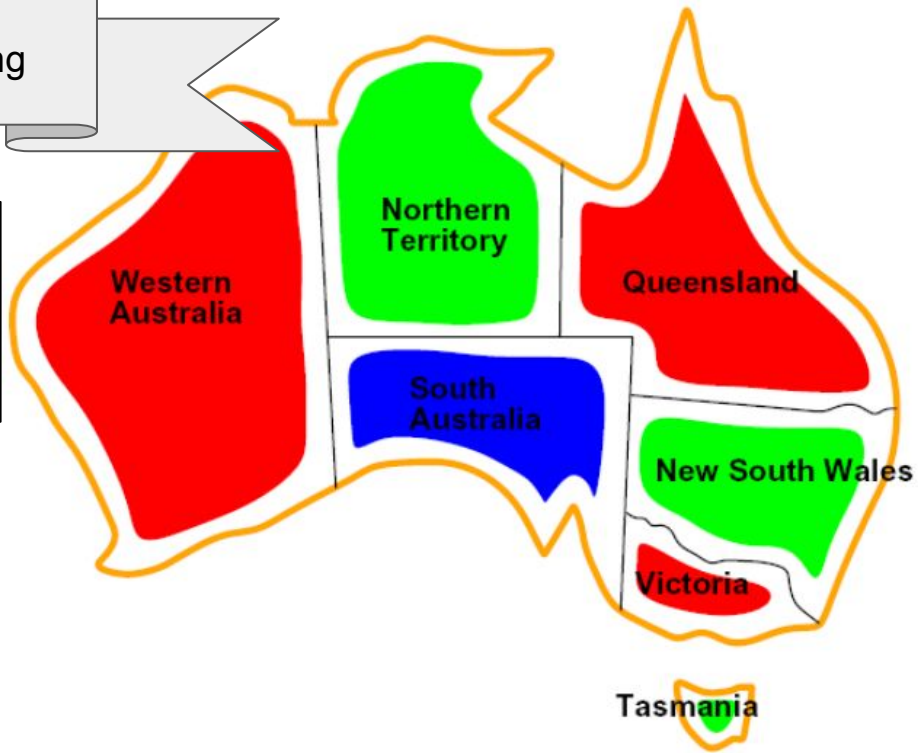




CSP Example

Map Coloring

Assign a color to each region such that no two adjacent regions have the same color.

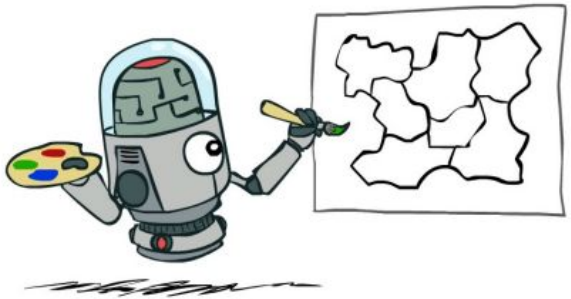


CSP Example: Map Coloring

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

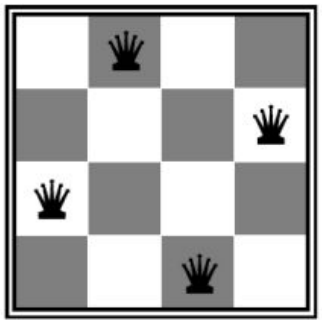
  Implicit: WA ≠ SA
  Explicit: (WA,SA) ∈ {(red,blue), {red,green}, …}
- Solutions are assignments satisfying all constraints, e.g.:

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



CSP Example: N-Queens

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



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

Is this needed?

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

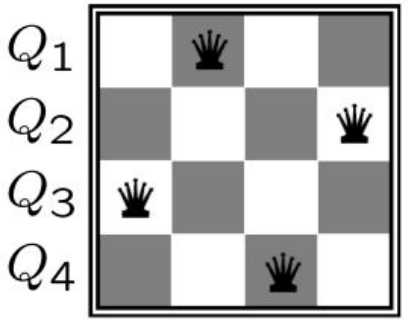
CSP Example: N-Queens

- **Formulation 2:**
  - Variables: Q_k
  - Domains: $\{1, 2, 3, \ldots N\}$
  - Constraints:

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

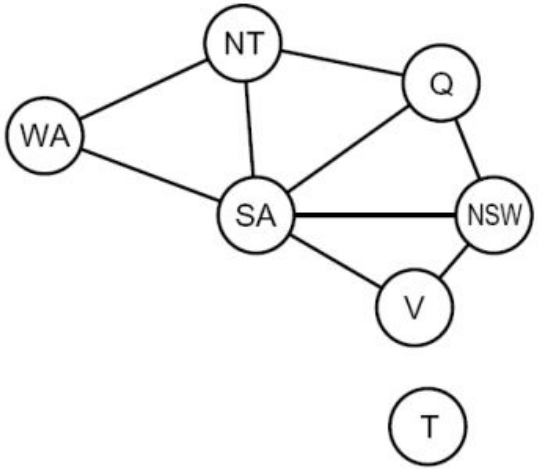
Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$

$\ldots$

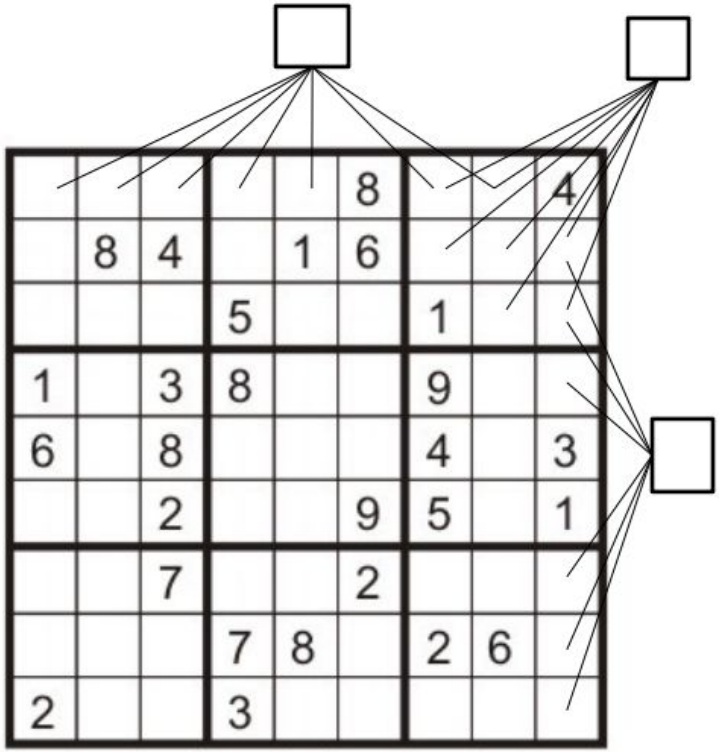


Constraint Graphs

- 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.



CSP Example: Sudoku



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

  9-way alldiff for each column

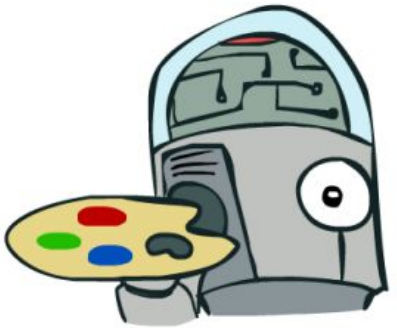
  9-way alldiff for each row

  9-way alldiff for each region

  (or can have a bunch of pairwise inequality constraints)

Varieties of CSPs

- Discrete Variables:
  - Domain Size d means d^n possible assignments
  - E.g., Boolean CSPs, including Boolean satisfiability
- Continuous Variables



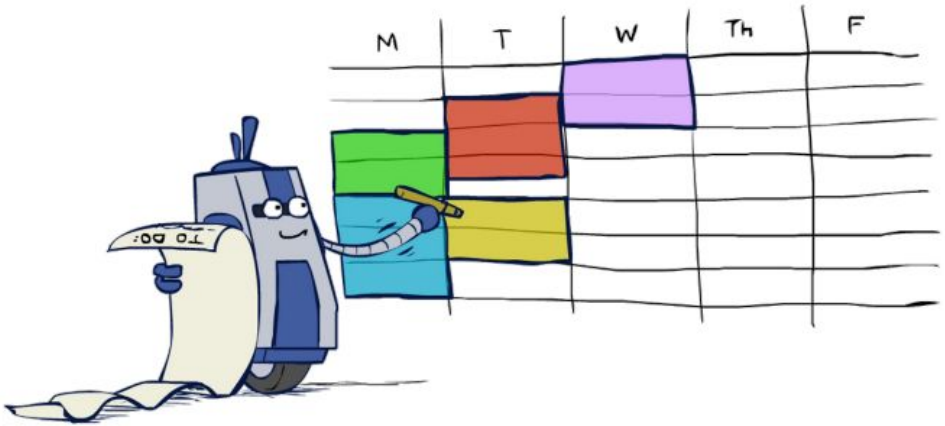
Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:
    SA ≠ green
  - Binary constraints involve pairs of variables, e.g.:
    SA ≠ 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.

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!

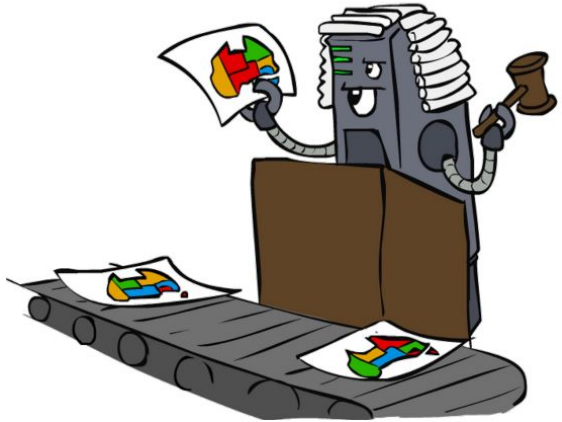


- Many real-world problems involve real-valued variables…

Solving CSPs

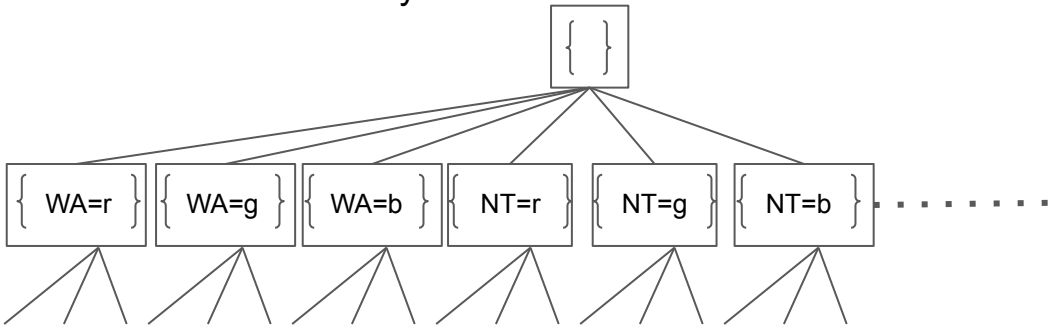
Standard Search Formulation

- Standard search formulation of CSPs: Let's start with the normal search setting.
- 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.
- We'll start with the straightforward, naive approach, then improve it.



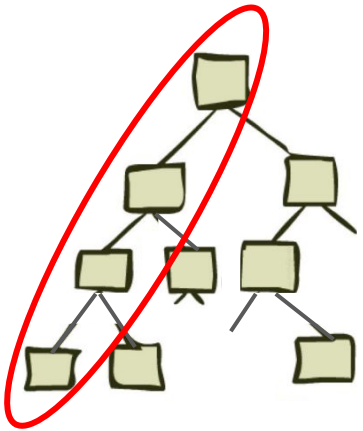
Search Methods

- What would BFS do?
  - Will expand in all directions and will traverse the entire tree till it reaches the last layer of the tree.



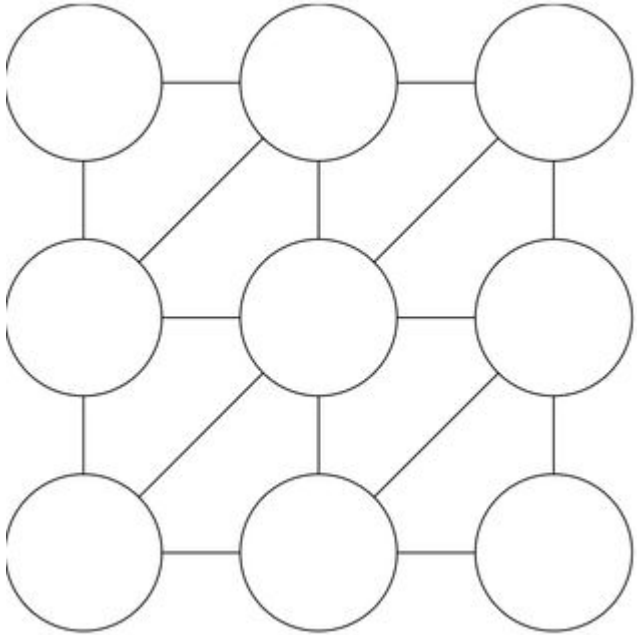
Search Methods

- What would BFS do?
  - Will expand in all directions and will traverse the entire tree till it reaches the last layer of the tree.
- What would DFS do?
  - Will expand along a branch till it reaches the leaf node at the last layer of the tree.
- Complete assignments will occur only in the last layer.
  - So DFS is more reasonable in this case.



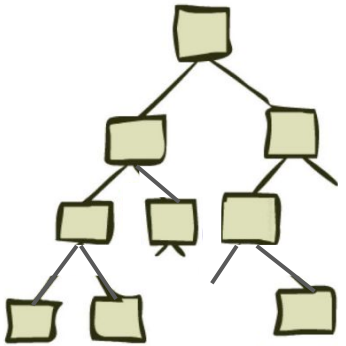
Search Methods

- DFS

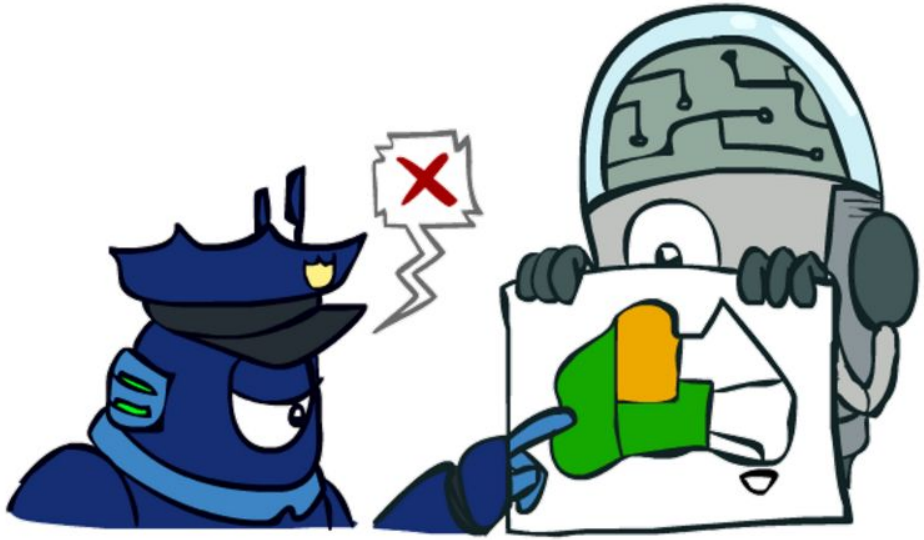


Search Methods

- What problems does naive search have?
  - You have to go all the way to the bottom of the search space where all the assignments have been done to check whether the current assignment is correct.
  - CSPs have better structure, and we don’t need to wait till the last variable assignment to check for correctness.

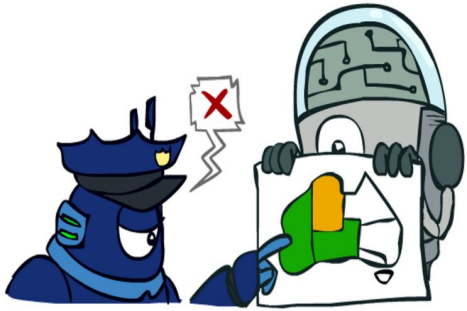


Backtracking Search



Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Only need to consider assignments to a single variable at a time.
- Idea 1: One variable at a time:
  - Variable orderings are commutative, so fix the ordering
  - i.e., [WA = red then NT=green] is the same as [NT=green then WA = red]
  - Only need to consider assignments to a single variable at each step
    - Reduces the branching factor
- Idea 2: Check constraints as you go:
  - i.e., consider only values that 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.

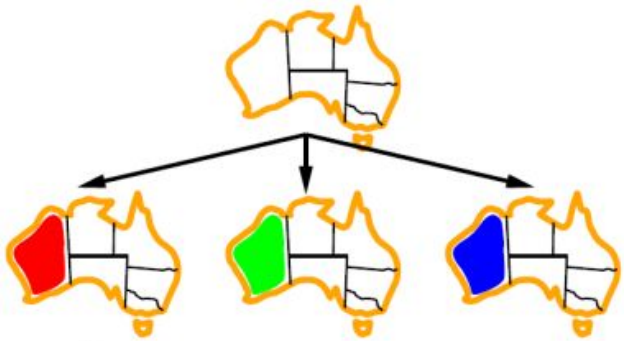


Backtracking Search

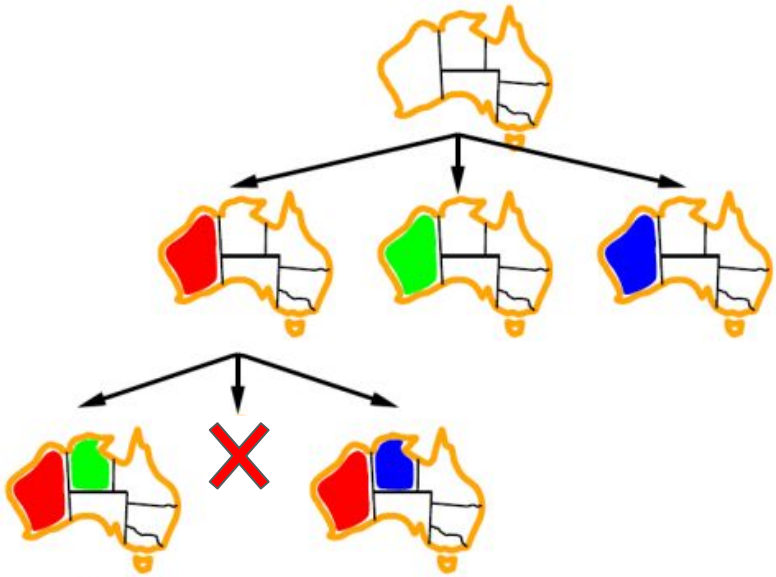
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

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
```

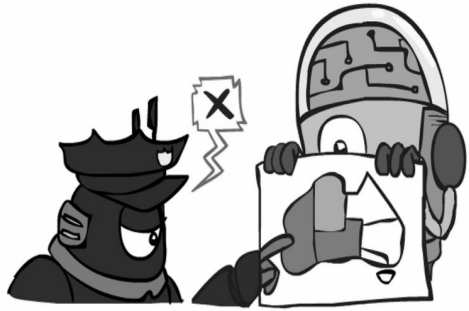
Backtracking Example

Backtracking Example

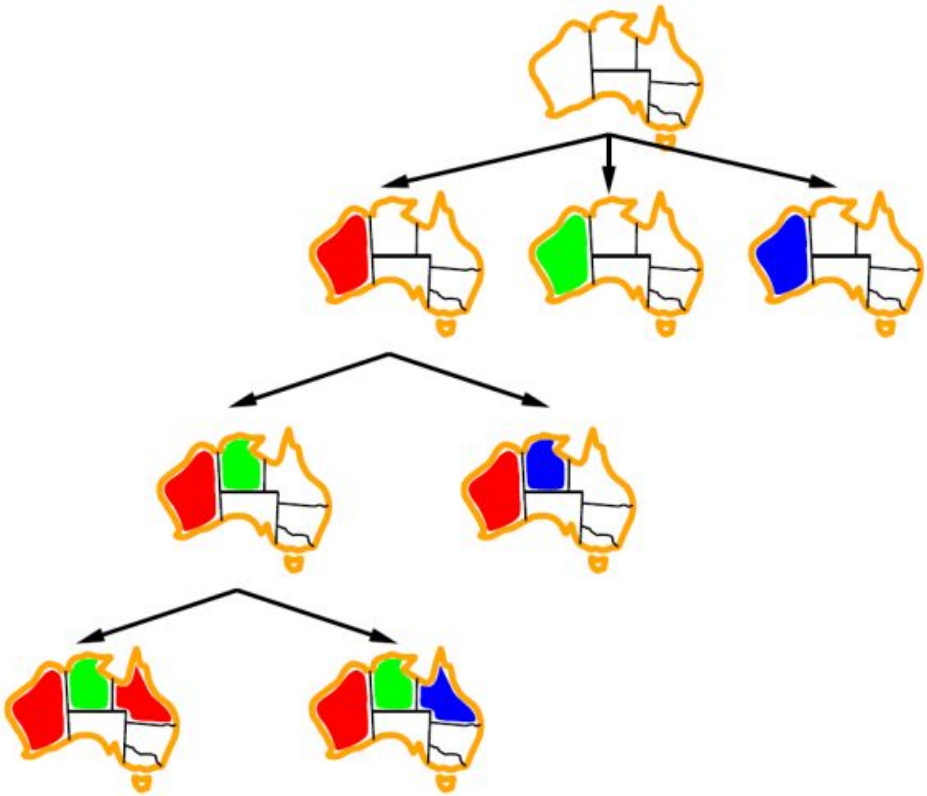


Backtracking Example

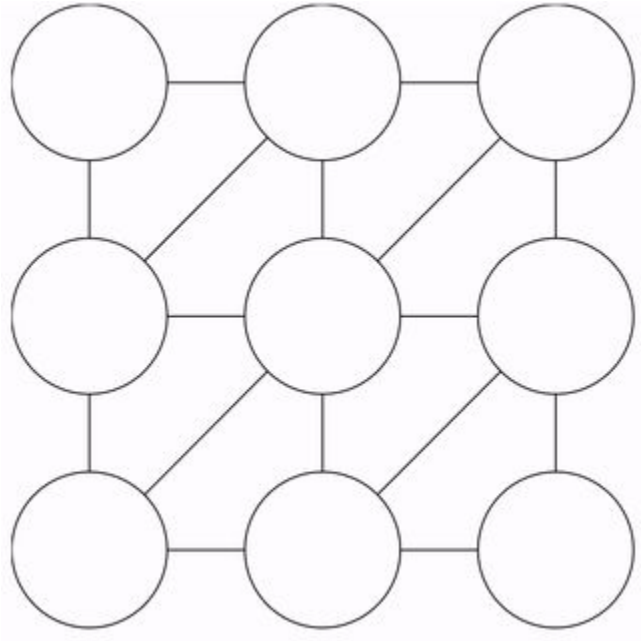




Backtracking Example



Backtracking Example



Improving Backtracking

- Filtering: Can we detect inevitable failure early?
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

Filtering

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.

Filtering: Forward Checking

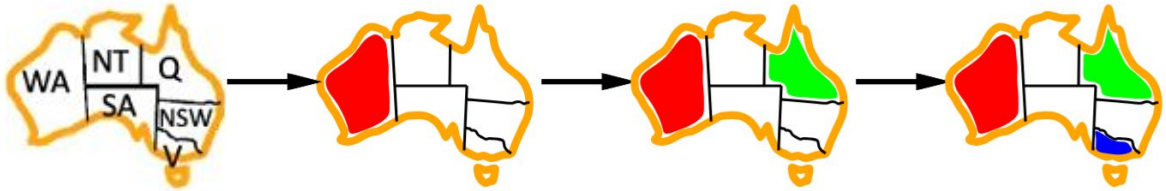
- Filtering: Keep track of domains for unassigned variables and cross off bad options.
- Forward checking: Cross off values that violate a constraint when added to the existing assignment.

Filtering: Forward Checking

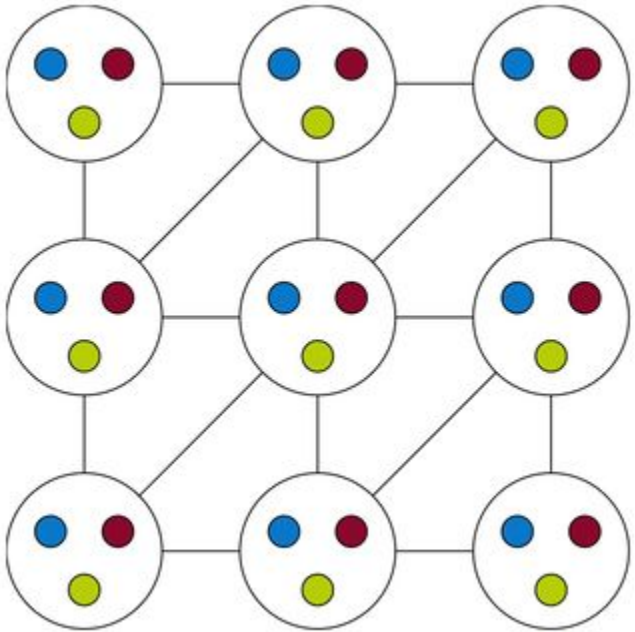
- Filtering: Keep track of domains for unassigned variables and cross off bad options.
- Forward checking: Cross off values that violate a constraint when added to the existing assignment.

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.



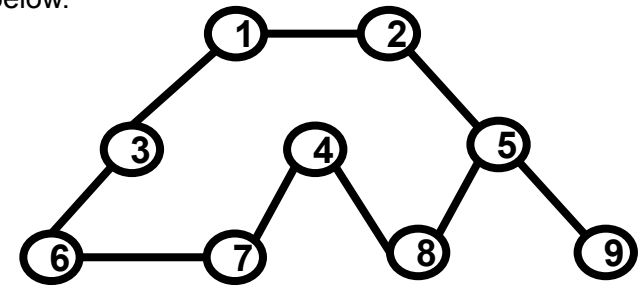
Filtering: Forward Checking



Practice Question

Consider the following problem: 8 cricketers arrive at a hotel after their match and have to be allotted rooms for their stay. The cricketers include 3 batsmen: B1, B2, B3, 3 bowlers: W1, W2, W3 and 2 all rounders: A1, A2. There are 9 rooms **(variables)** in the hotel that are arranged as shown in the graph below.

Sol: Link



	B1	B2	B3	W1	W2	W3	A1	A2	Order
1	B1	X	X	X	X	X	X	X	1
2	X								
3	X								
4	X								
5	X								
6	X								
7	X								
8	X								
9	X								

Solve the room allotment problem with the following constraints:

C1: The path between the room/nodes of the same types of cricketers should contain atleast 1 node, excluding the start and end nodes, e.g., if W2 gets room 2, then no other bowler can be allotted rooms 1, 5

C2: Room number of B3 > Room number of W2

Solve using CSP with backtracking and forward checking. Fill the Table. If more than one room is equally eligible to be allotted next, follow the increasing order of the room number to allot first. Stop at the first Failure and mention FAILURE. Failure can be caused by not having any rooms for a specific cricketer or having more than one room that cannot be allotted to anyone. Room 1 is allotted to B1 by default. A cricketer can be allotted only one room and one room can have only one cricketer. Order column should mention the order in which you tried to allot the room, even if you were not able to assign any cricketer to that room.