PURDUE CS47100

INTRODUCTION TO AI

RECAP: ADVERSARIAL SEARCH & CSP

- Adversarial search
  - Dealing with resource limit: Depth limited search + Evaluation function
  - Dealing with randomness: Introduce chance node in the game tree and compute the expected minimax value

- Constraint satisfaction problems (CSPs)
  - State is defined by **variables X_i** with values from a **domain D**
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables
  - Allows useful general-purpose algorithms with more power than standard search algorithms

EXAMPLE: MAP COLORING

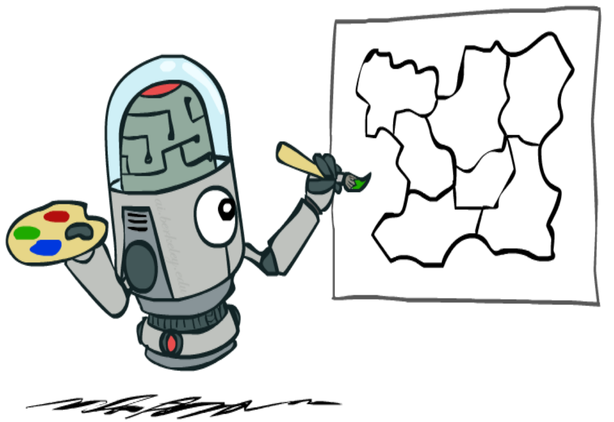
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{red, green, blue\}$
- Constraints: adjacent regions must have different colors

  Implicit: $WA \neq NT$

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

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

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



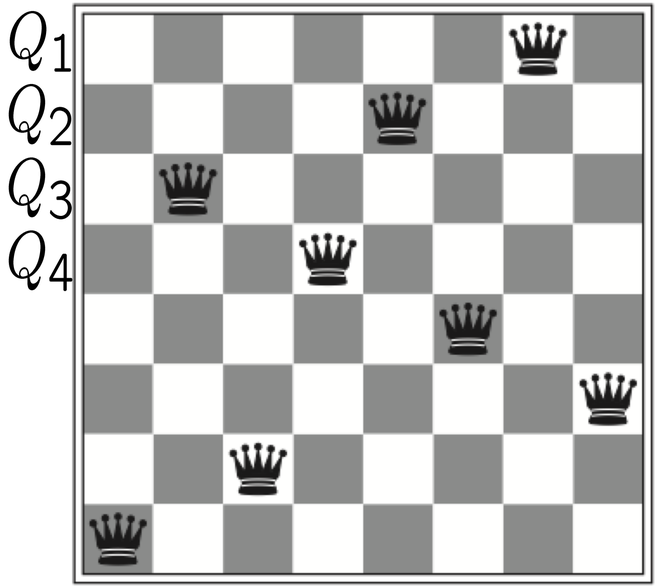
EXAMPLE: N-QUEENS

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

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

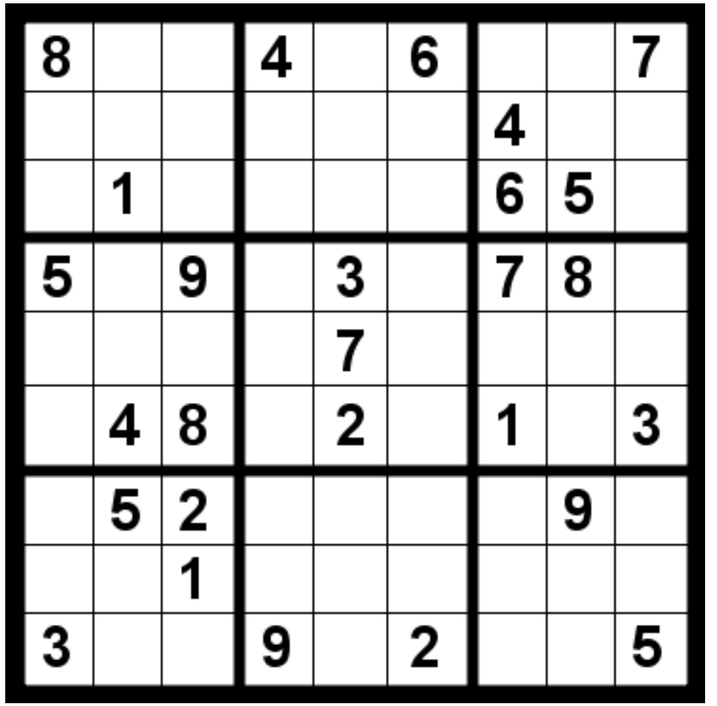
  Explicit: $(Q_1, Q_2) \in \{(1,3), (1,4), \ldots\}$

  $\ldots$



EXAMPLE: SUDOKU

- Objective
  - Fill the empty cells with numbers between 1 and 9
- Rules
  - Numbers can appear only once on each row
  - Numbers can appear only once on each column
  - Numbers can appear only once on each region
- Variables? Domain?
- Constraints?



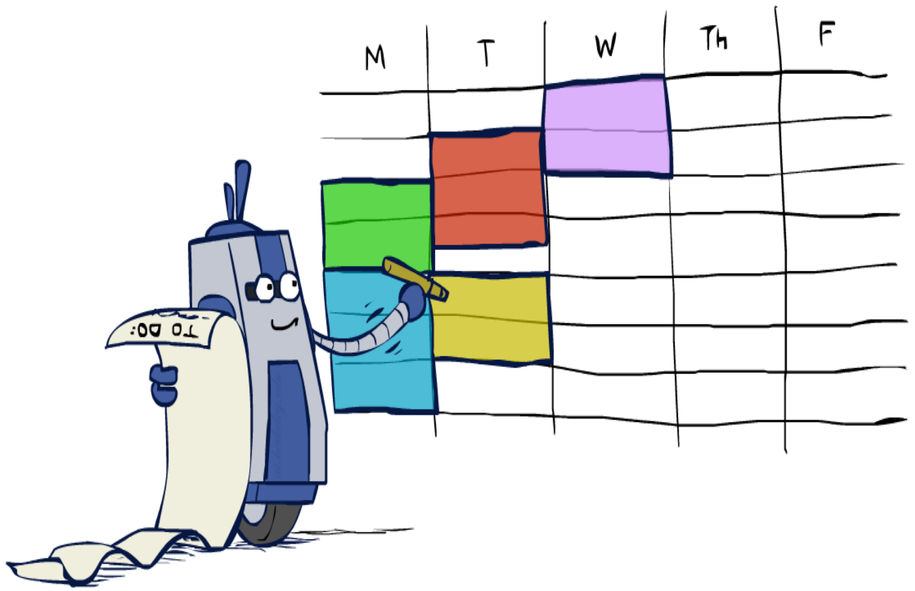
SUDOKU CSP FORMULATION

- Variables
  - v_{ij} *is the value in the jth cell of the ith row*
  - D_{ij}=*{1,2,3,4,5,6,7,8,9}*
- Constraints
  - $C^R : \forall i, \cup_j v_{ij} = D$
  - $C^C : \forall j, \cup_i v_{ij} = D$
  - $C^B : \forall k, \cup_{i,j} (v_{ij} \mid ij \in B_k) = D$

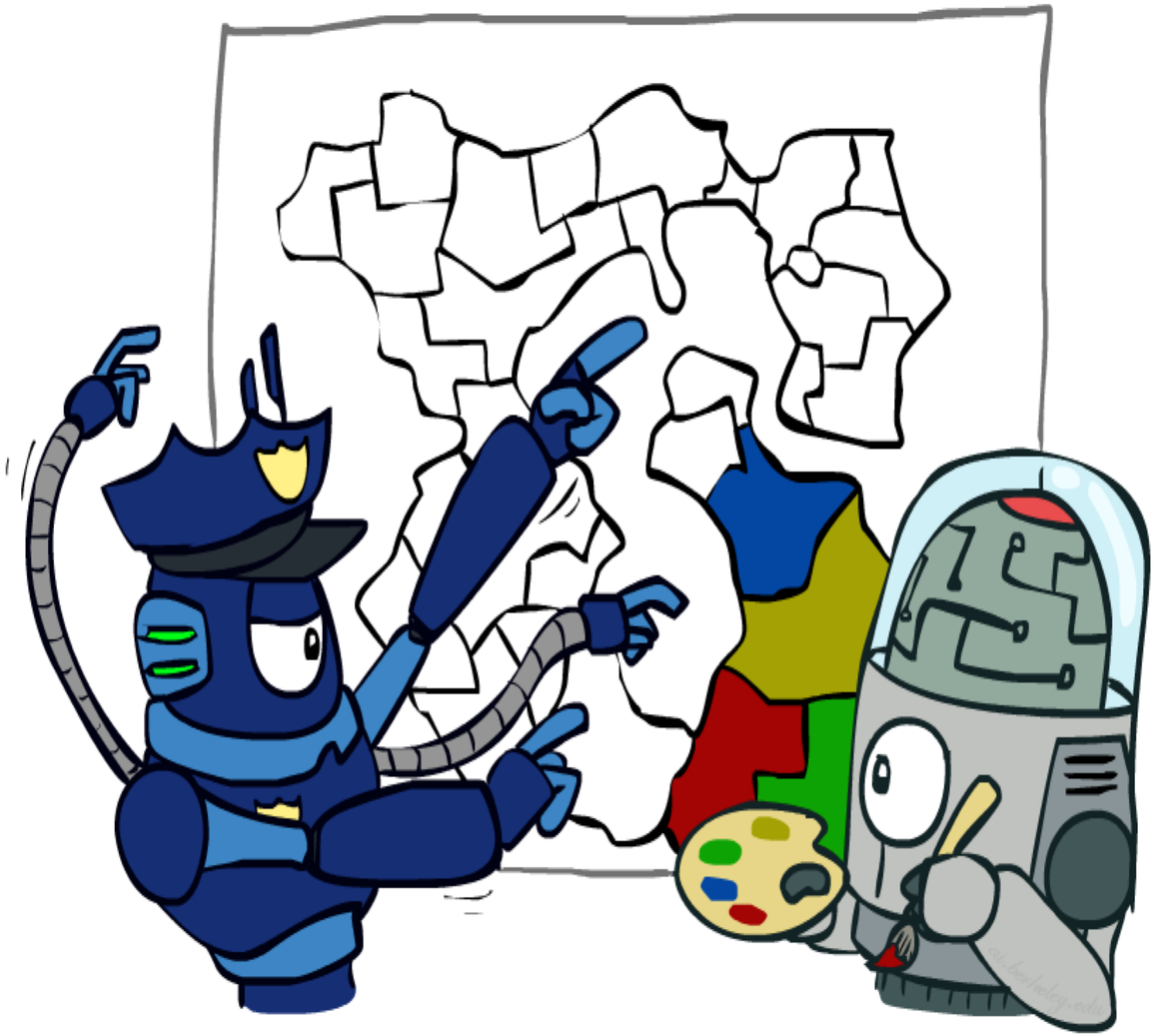
8	v_{12}	v_{13}	4	v_{15}	6	v_{17}	v_{18}	7
v_{21}	v_{22}	v_{23}	v_{24}	v_{25}	v_{26}	4	v_{28}	v_{29}
v_{31}	1	v_{33}	v_{34}	v_{35}	v_{36}	6	5	v_{39}
5	v_{42}	9	v_{44}	3	v_{46}	7	8	v_{49}
v_{51}	v_{52}	v_{53}	v_{54}	7	v_{56}	v_{57}	v_{58}	v_{59}
v_{61}	4	8	v_{64}	2	v_{66}	1	v_{68}	3
v_{71}	5	2	v_{74}	v_{75}	v_{76}	v_{77}	9	v_{79}
v_{81}	v_{82}	1	v_{84}	v_{85}	v_{86}	v_{87}	v_{88}	v_{89}
3	v_{92}	v_{93}	9	v_{95}	2	v_{97}	v_{98}	5

REAL-WORLD CSPS

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



VARIETIES OF CSPS AND CONSTRAINTS



VARIETIES OF CSP VARIABLES

- Discrete Variables
  - Finite domains (e.g., Boolean CSPs, Boolean satisfiability)
    - Size d means $O(d^n)$ complete assignments
  - Infinite domains (e.g., job scheduling, variables are start hours for each job)
    - Linear constraints are solvable, nonlinear are undecidable
- Continuous variables (e.g., start precise times for Hubble Telescope observations)
  - Linear constraints solvable in polynomial time by LP methods

VARIETIES OF CSP CONSTRAINTS

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

$$SA \neq green$$

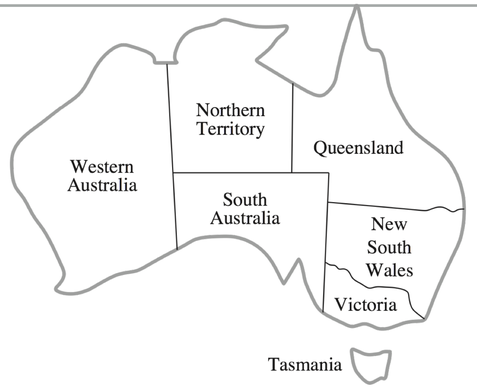
- Binary constraints involve pairs of variables, e.g.:

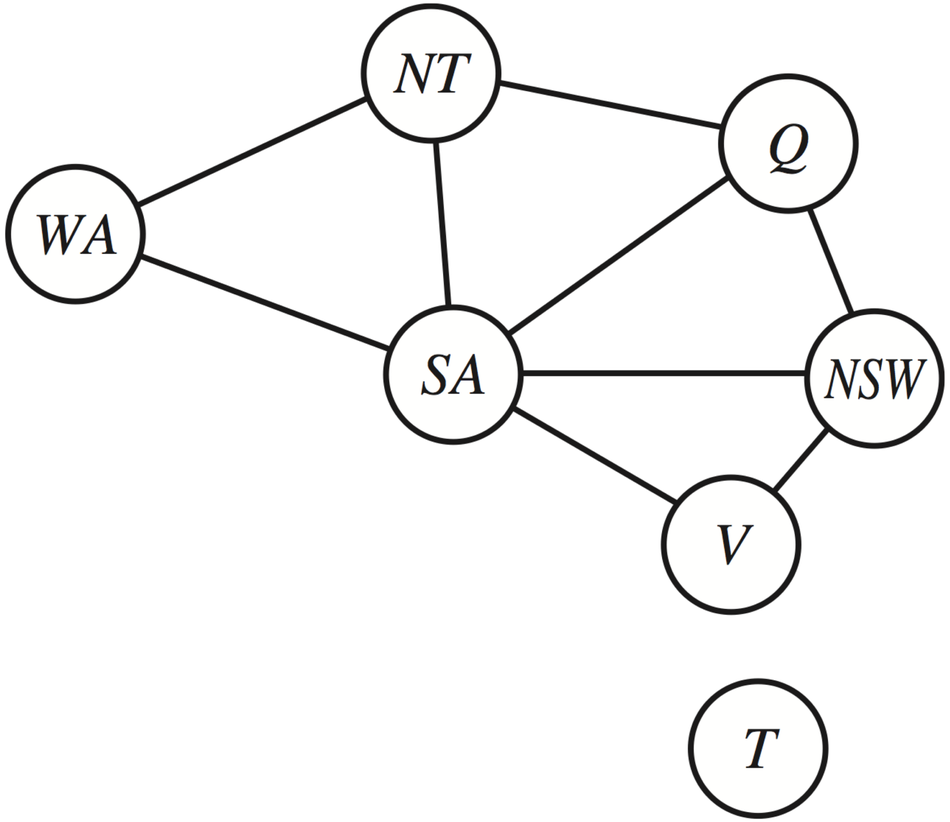
$$SA \neq WA$$

- Higher-order constraints involve 3 or more variables: e.g., Sudoku row constraints
- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems

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

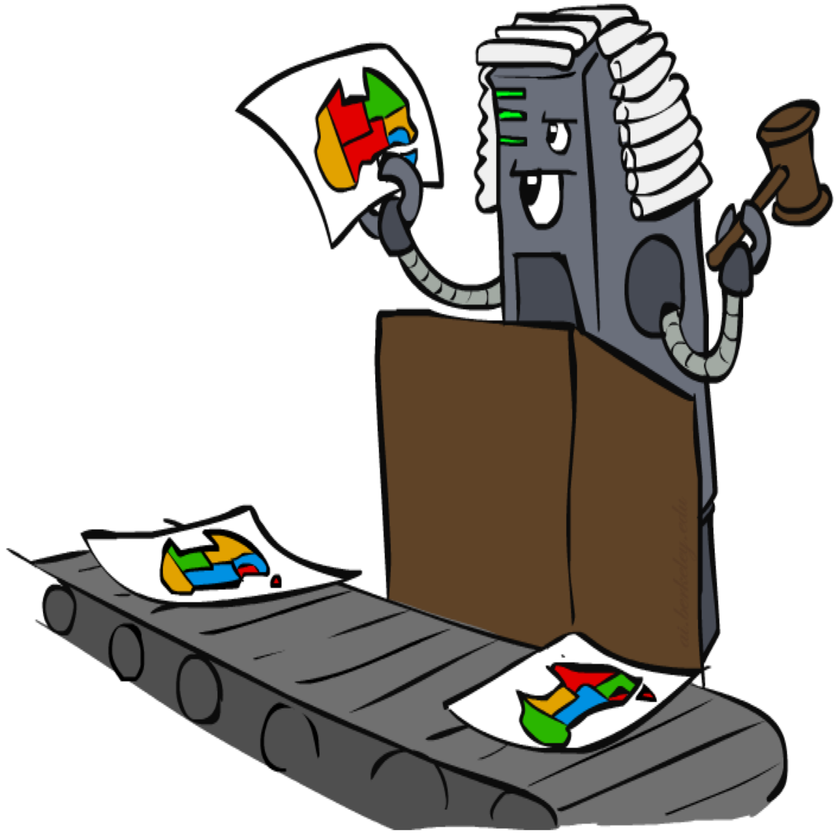




SOLVING CSPS

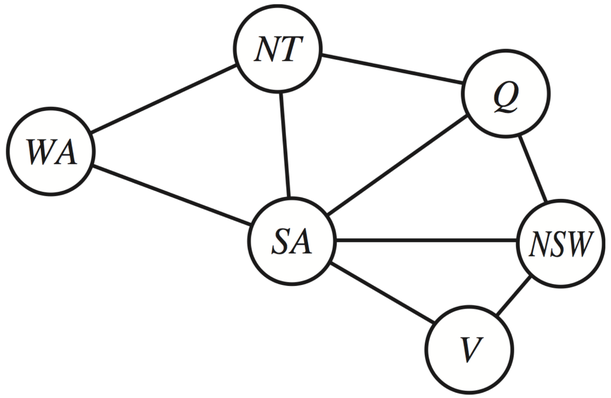
STANDARD SEARCH FORMULATION

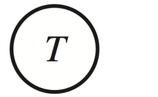
- Standard search formulation of CSPs
- States defined by the values assigned so far (i.e., partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

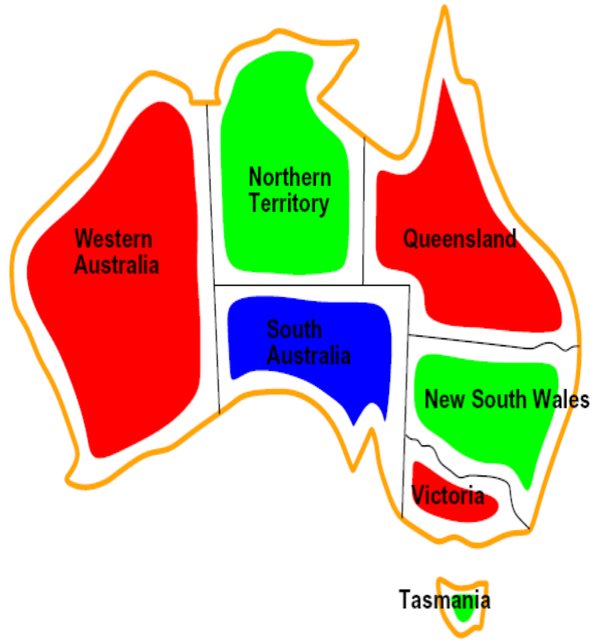


SEARCH METHODS

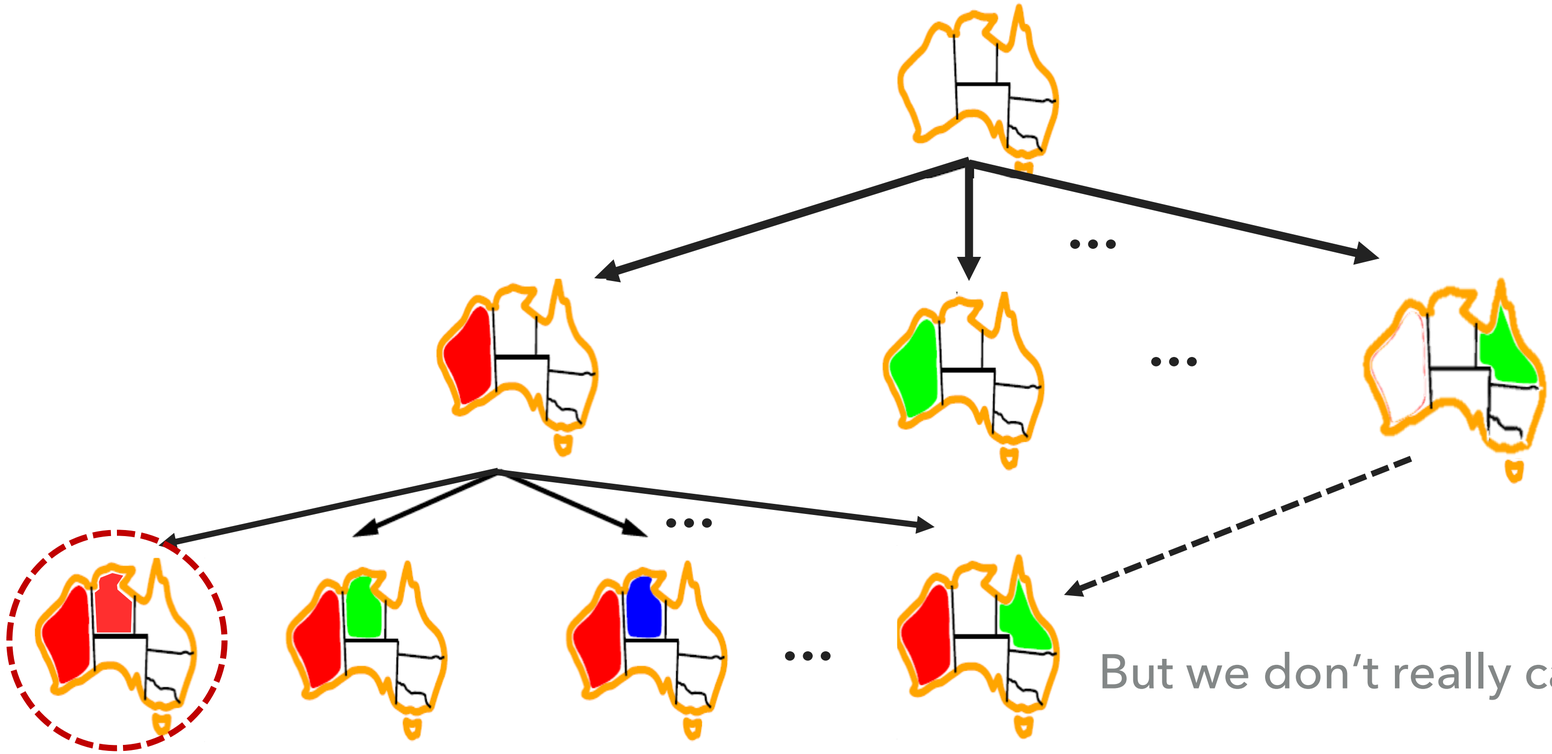
- What would BFS do?
- What would DFS do?
- What problems does naive state space search have in this setting?







SOLVING CSP AS A STANDARD SEARCH PROBLEM: SEARCH TREE



Branching factor = nd!

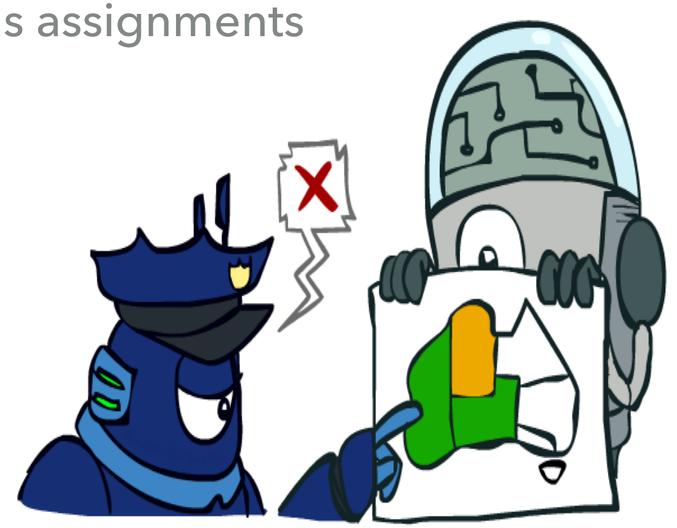
But we don’t really care the order of coloring…

This won’t lead to any consistent solution!

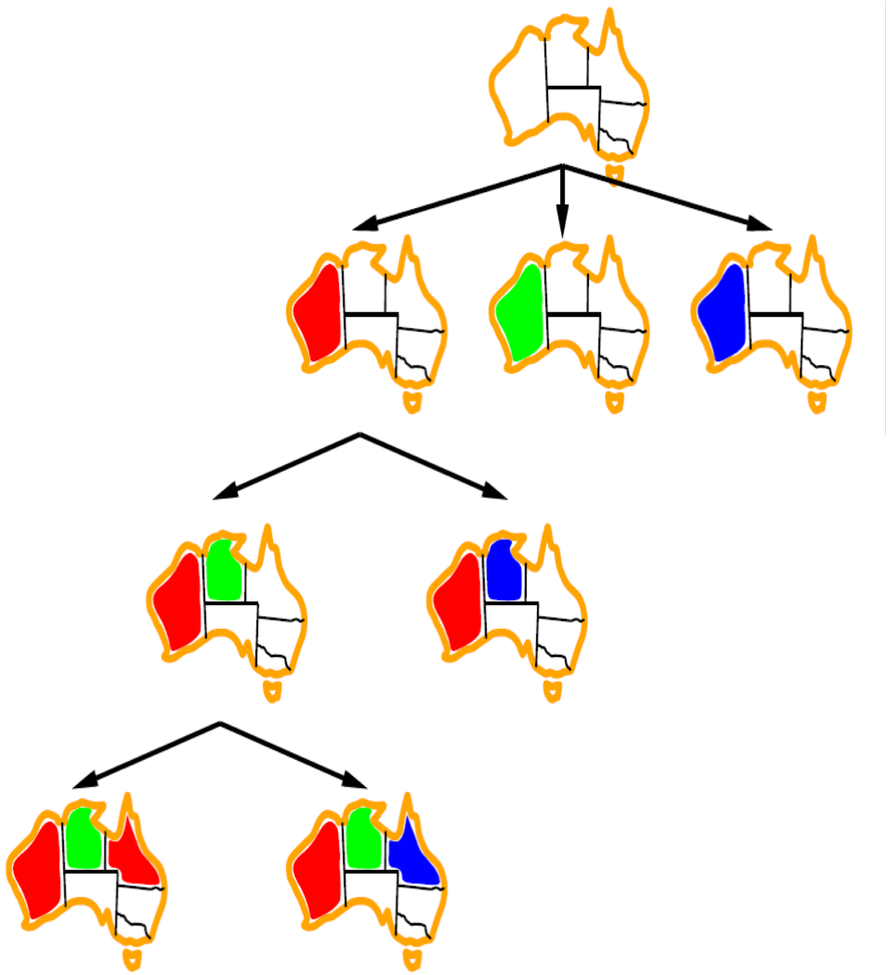
BACKTRACKING SEARCH

BACKTRACKING SEARCH

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are **commutative**, i.e., [WA = red then NT = green] is the same as [NT = green then WA = red]
  - Fix ordering of variables and only consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - "Incremental goal test" i.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
- 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

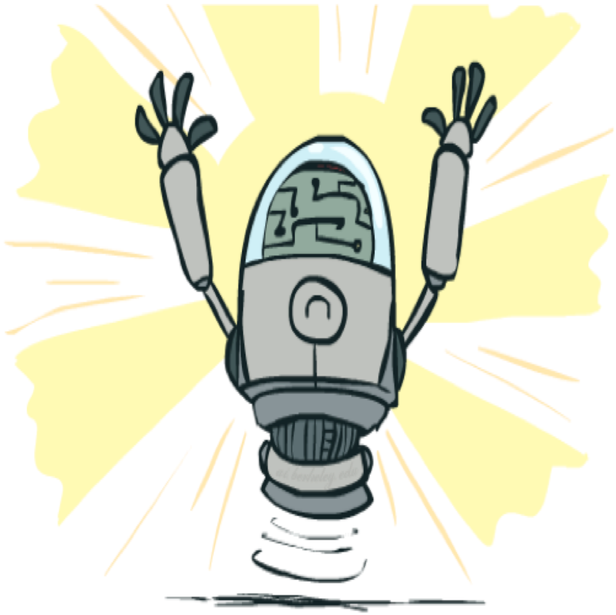
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({}, csp)

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
```

- Backtracking = DFS + variable-ordering + fail-on-violation

IMPROVING BACKTRACKING

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



FILTERING

FROM BACKWARD CHECKING TO FORWARD CHECKING

```
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
```

Detect failure early by checking backwards, i.e., complete partial goal test!

These failures occur because some previous value assignment, so why not do the check even earlier when the values just get assigned?

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 a value assignment is added to the existing assignment