CSPs
CSP Formulation
- State representation
  - Variables: `X = {x_1, ... , x_n}`
  - Domains: `D = {d_1, ... , d_k}`
    - Such that `x_i` has domain `d_i`
  - Initial state: all variables unassigned
  - Intermediate state: partial assignment
- Actions, costs, and transition
  - Assignment of values (within domain) to variables
  - Costs are unnecessary
- Goal test
  - Constraints: `C = {c_1, ... , c_m}`
    - Defined via a constraint langauge (algebra, logic, sets)
    - Each c_i corresponds to a requirement on some subset of `X`
    - Can be unary ($|scope| = 1$), binary ($|scope| = 2$), or global ($|scope| > 2$)
  - Objective is a complete and consistent assignment
    - Find a legal assignment $(y_1, ..., y_n)$ s.t. $y_i \in D_i \ \forall i \in [1, n)$
    - Complete: all variables assigned values
    - Consistent: all constraints in `C` satisfied

Algo
```
Function CSPSolver(variables, domains, constraints):
    assignments = initial_state       # no assignments made
    while assignments incomplete:
        if no possible assignments left: return failure
        current = assign a value to non-assigned variable
        if current is consistent then: assignments.store(current)
    return assignments
```
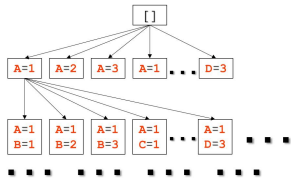
Search tree size
At depth l: $(|X| - l) \cdot |d|$ states
Total number of leaf states:
$$nm \times (n-1)m \times (n-2)m \times \ldots \times 2m \times m = n!m^n$$
where $n = |x|$ and $m = |d|$
Order of variable assignments is not important, just consider assignments to one variable per level (m^n leaves)

Backtracking algorithm
```
def backtrack(csp, assgt) -> bool:
    if assgt is complete:
        return assgt
    var = select_unassigned_variable(csp, assgt)
    for val in order_dom_values(csp, var, assgt):
        if val is consistent with assgt:
            add {var = val} to assgt
            inferences = inference(csp, var, assgt)
            if inferences != failure:
                add inferences to csp
                result = backtrack(csp, assgt)
                if result != failure:
                    return result
                remove inferences from csp
            remove {var = value} from assgt
    return failure
```

Variable-order heuristics
MRV (minimum remaining values)
- Choose variable with fewest legal values
- Most constrained variable
- Sort by domain size, choose variable with smallest domain size
- Places larger subtrees closer to the root so that any invalid state found prunes a larger subtree $\Rightarrow$ eliminates larger subtrees earlier

Degree Heuristic
- To break ties in MRV heuristics
- Pick unassigned variable with most constraints (highest degree in constraint graph)
- This reduces the branching factor b

Recommended variable selection: MRV, then degree, then random

Value-order Heuristics
LCV (least constraining value)
- Choose the value that rules out the fewest choices from remaining domain values
  - Given assignment of value v to variable x', determine set of unassigned variables U that share a constraint with x' and pick v that maximises sum of consistent domain sizes of variables in U
- Avoids failure by avoiding empty domains

With variables, fail first. With values, fail last. Cuz need to look at all variables but dn to look at all values.
BUT if all solutions required, then value-ordering irrelevant

Forward Checking
Track remaining legal values for unassigned variables, terminate search when any variable has no legal values (based on constraints with recently assigned variable)

Constraint propagation
Inference step to ensure local consistency of all variables
- Traverse constraint graph to ensure variable at each node is consistent, eliminate all values in variable's domain that are not consistent with linked constraints

Node-consistent (vertex-consistent)
- Domain of a variable is consistent with its unary constraints
- Achieved through pre-processing step before backtracking

Arc-consistent (edge-consistent)
- Domain of a variable is consistent with its binary constraints
- The variable's domain value must have a partnering domain value in other variable that will satisfy the binary constraint

X_i is arc-consistent w.r.t $X_j \iff \forall x \in D_i \ \exists y \in D_j$ s.t. binary constraint on arc (X_i, X_j) is satisfied
When checking arc (X_a, X_b), remove values from D_a

AC-3 Algo
```
function ac3(csp) -> bool:
    queue = queue of all arcs in csp (both dirs)
    while queue:
        (X_i, X_j) = queue.pop()
        if revise(csp, X_i, X_j):
            if len(D_i) == 0:
                return False
            for X_k in X_i.neighbors - {X_j}:
                queue.append((X_k, X_i))
    return True

function revise(csp, X_i, X_j):
    revised = False
    for x in D_i:
        if no value in D_j allows (x, y) to satisfy
        the constraint between X_i and X_j:
            delete x from D_i
            revised = true
    return revised
```

Time complexity of AC-3
With n variables, there are at most $2 \times \binom{n}{2} \in O(n^2)$ directed arcs
Each arc can be reinserted at most d times because X_i has hat most d values to delete where d is domain size
Checking consistency of arc (`revise()`) takes $O(d^2)$ time
Overall time complexity: $O(n^2 \times d \times d^2) = O(n^2 \times d^3)$

Advesarial search
Assume 2 players, zero-sum game. MAX player wants to maximise value (our agent), MIN player wants to minimise value (opponent)

Formulating games
- State representation: as per general search formulation
- `TO-MOVE(s)`: returns p, the player to move in state s
- `ACTIONS(s)`: legal moves in state s
- `RESULT(s, a)`: the transition model, returns resultant immediate state when taking action a at state s
- `IS-TERMINAL(s)`: returns `True` when game is over, `False` otherwise
- `UTILITY(s, p)`: defines a numeric value (score) for player p when the game ends at terminal state s
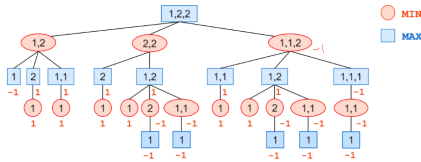
For zero-sum games, at terminal state s,
`utility(MAX, s) + utility(MIN, s) = 0`.
Just use `utility(MAX) = -utility(MIN)`

Strategies: optimal decisions via minimax

$$Minimax(s) = \begin{cases} \text{Utility}(s, \text{MAX}) & \text{if Is\-Terminal}(s) \\ \max_{a \in Actions(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if To\-Move}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if To\-Move}(s) = \text{MIN} \end{cases}$$

Remember that `Result(s,a)` outputs the successor state (given a taken at s) (i.e., MIN wants state with lowest utility, while MAX wants state with highest)

Backwards induction used
Properties
- Complete assuming finite game tree
- Optimal assuming optimal gameplay (opponent plays optimally)
- Time complexity $O(b^m)$
- Space complexity $O(bm)$
- Time polynomial to tree size

Backwards Induction Issue
Game trees are massive, impossible to expand entire tree, must find ways to shrink search tree

α - β pruning
General idea: don't explore moves that would never be considered.
α bounds MAX's values, β bounds MIN's values.
Prune subtrees that will never affect Minimax decision