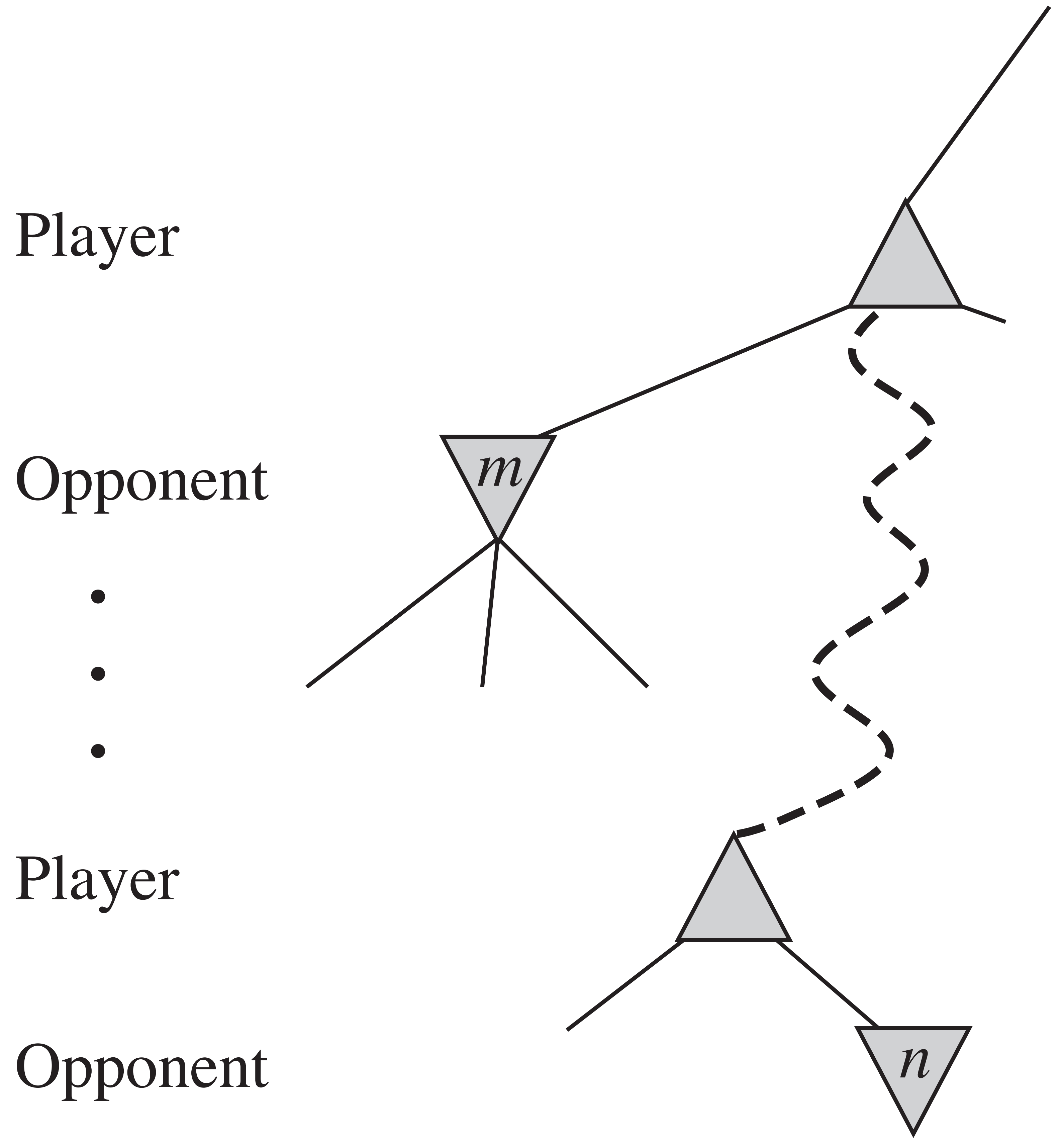
PURDUE CS47100
SEPT 11, 2019
PROF. JENNIFER NEVILLE

# INTRODUCTION TO AI

# RECAP: ADVERSARIAL SEARCH

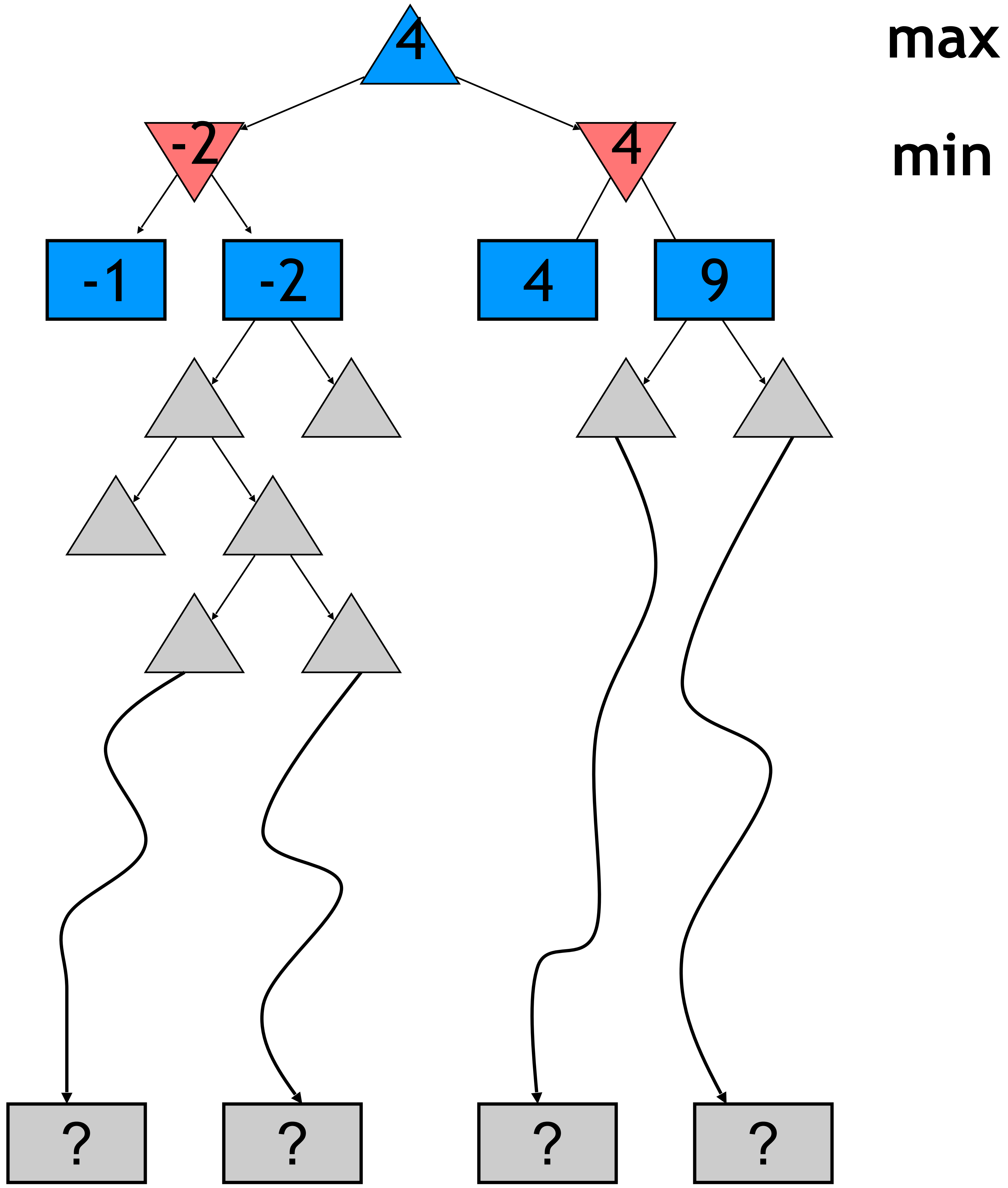
- Minimax search is a way of finding an optimal move in a zero-sum two player game
- Alpha-beta pruning is a way of finding the optimal minimax solution while avoiding searching subtrees of moves which won't be selected
  - Some branches will never be played by rational players since they include sub-optimal decisions (for either player)
  - Pruning produces results that are exactly equivalent to complete (unpruned) search
  - Node *ordering* can improve effectiveness; Perfect ordering gives time complexity $O(b^{m/2})$, thus, can search twice as far as ordinary minimax in equal time



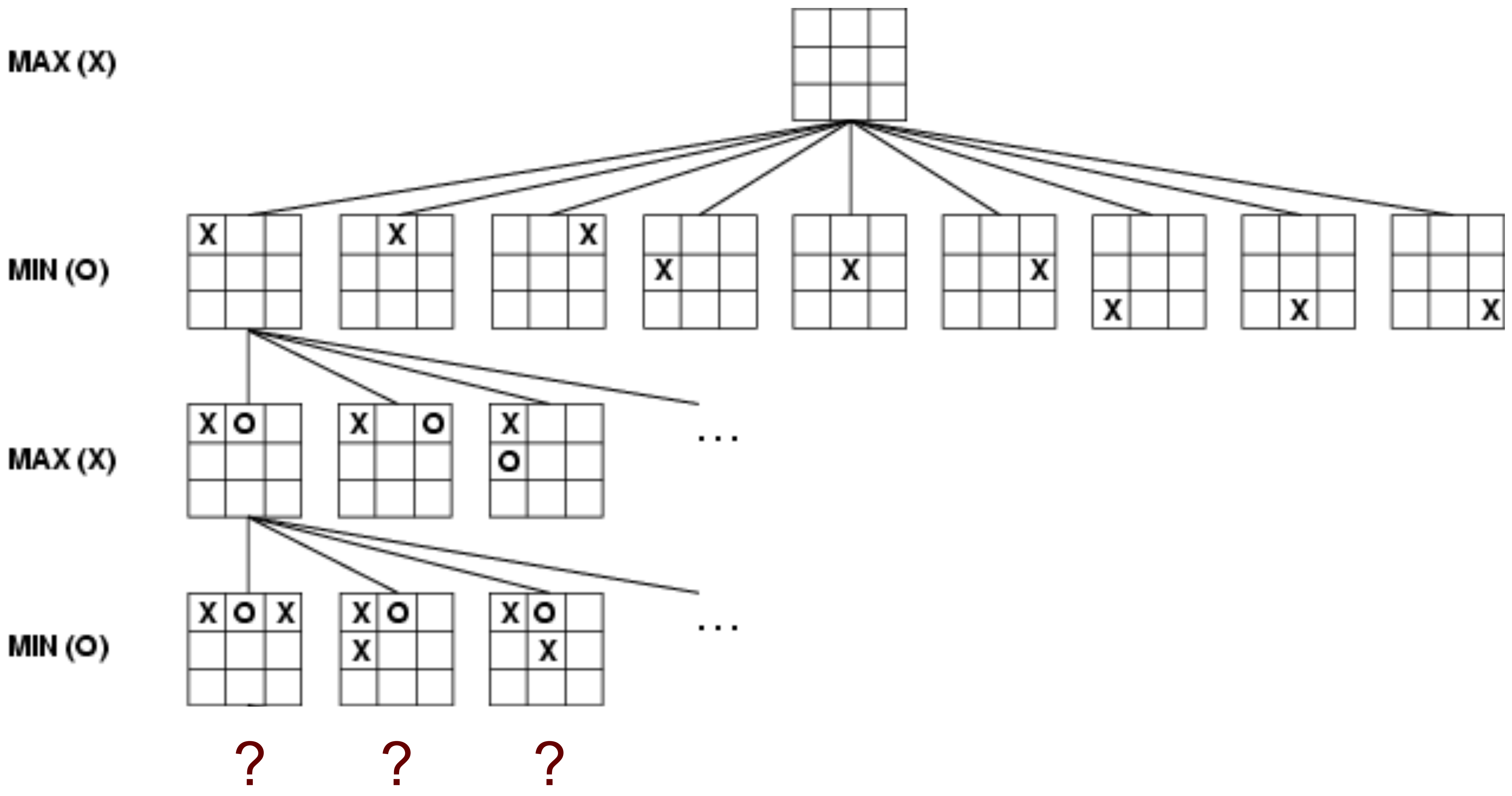
# RESOURCE LIMITS

# RESOURCE LIMITS

- **Problem**: In realistic games, cannot search to leaves
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - $\alpha$-$\beta$ reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone; More plies makes a BIG difference
- **Solution**: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an **evaluation function** for non-terminal positions

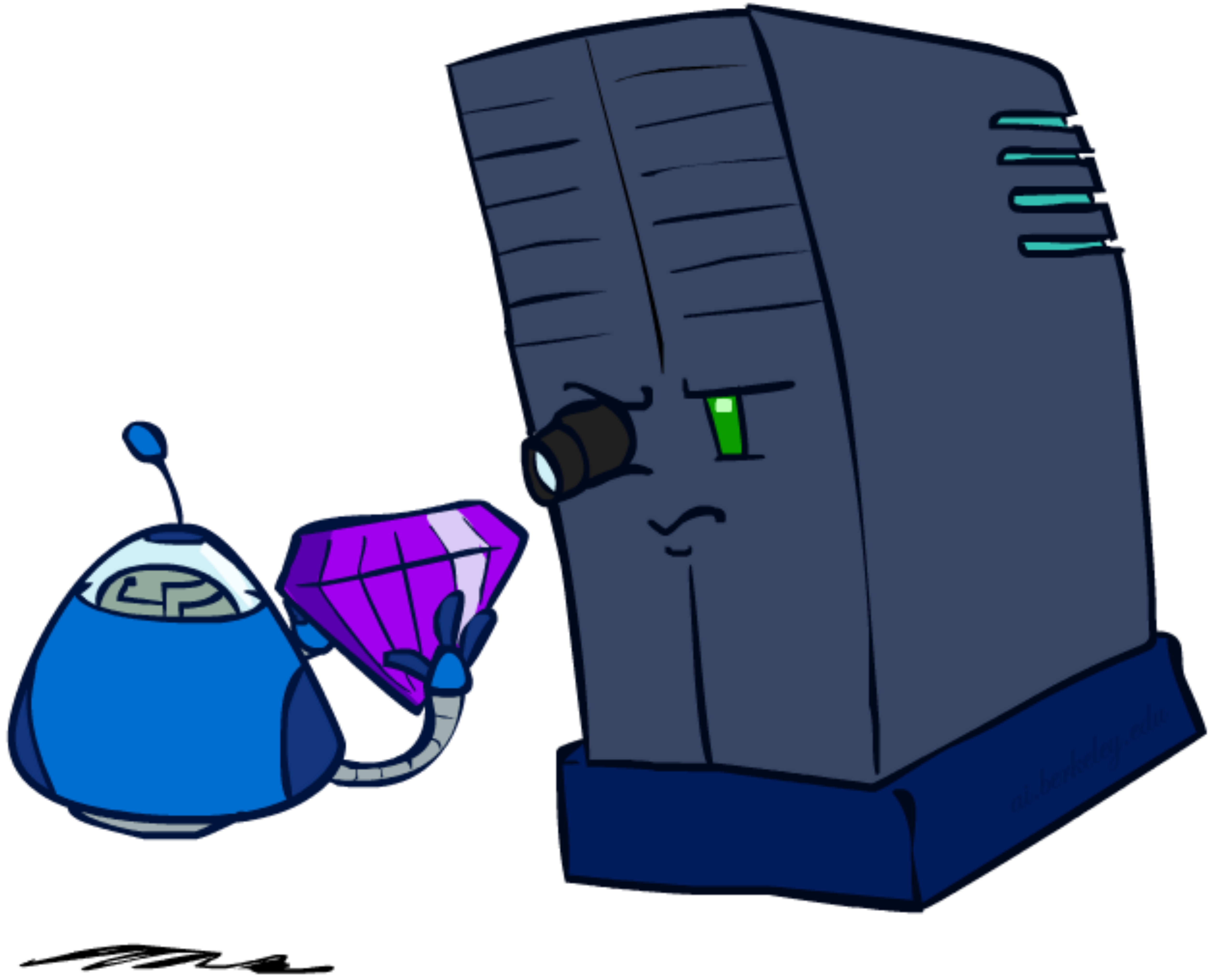


# WHAT TO DO WHEN SEARCH IS INTRACTABLE




- Stop the search before you reach terminal states (using depth cutoff)
- Evaluate nodes using an evaluation function – What properties should the evaluation function have?

# EVALUATION FUNCTIONS



# EVALUATION FUNCTIONS

- Desirable properties
  - Order terminal states in same way as true utility function
  - Strongly correlated with the actual minimax value of the states
  - Efficient to calculate
- Typically use **features** – simple characteristics of the game state that are correlated with the *probability of winning*
- The evaluation function combines feature values to produce a score:

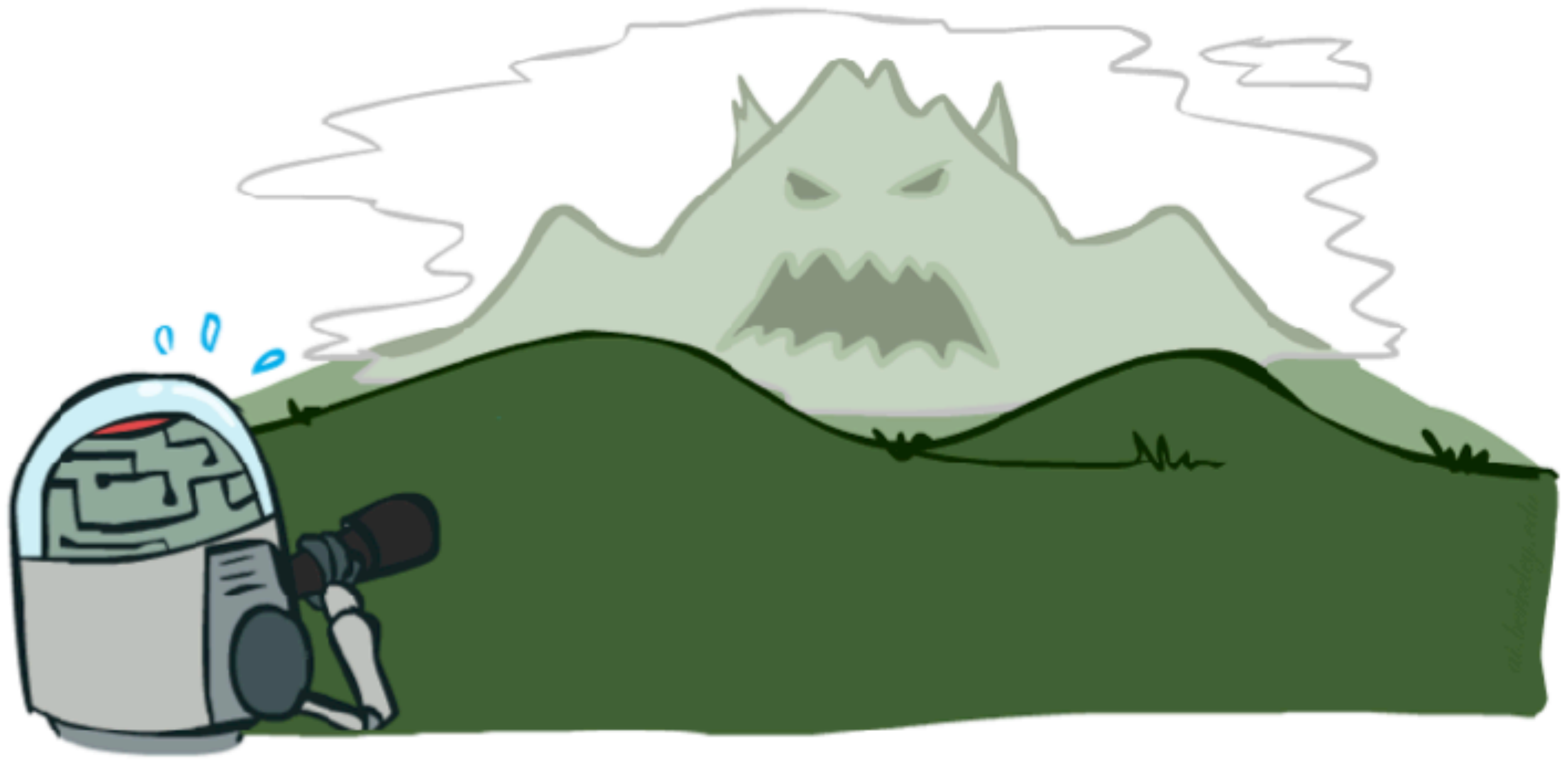
$$Eval(x) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

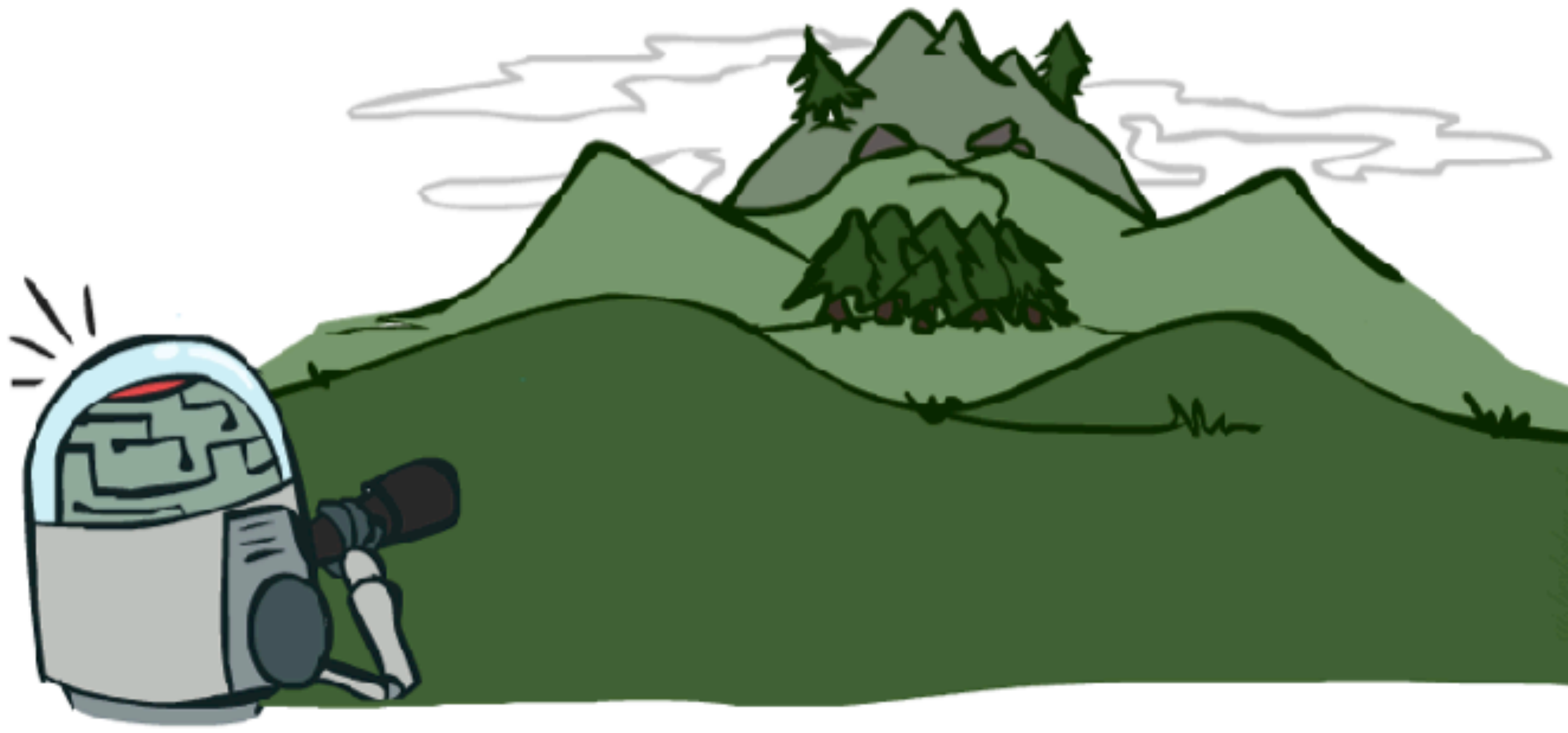
# EXAMPLE FEATURES

- What would be some useful features for chess?
  - Relative number of Bishops; Knights; Rooks; Pawns
  - Total number of pieces
  - Has queen?
  - Castled?
  - In check?
  - Distance of furthest pawn from start
  - Relative freedom (relative total number of possible moves)
  - etc.

# DEPTH MATTERS

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

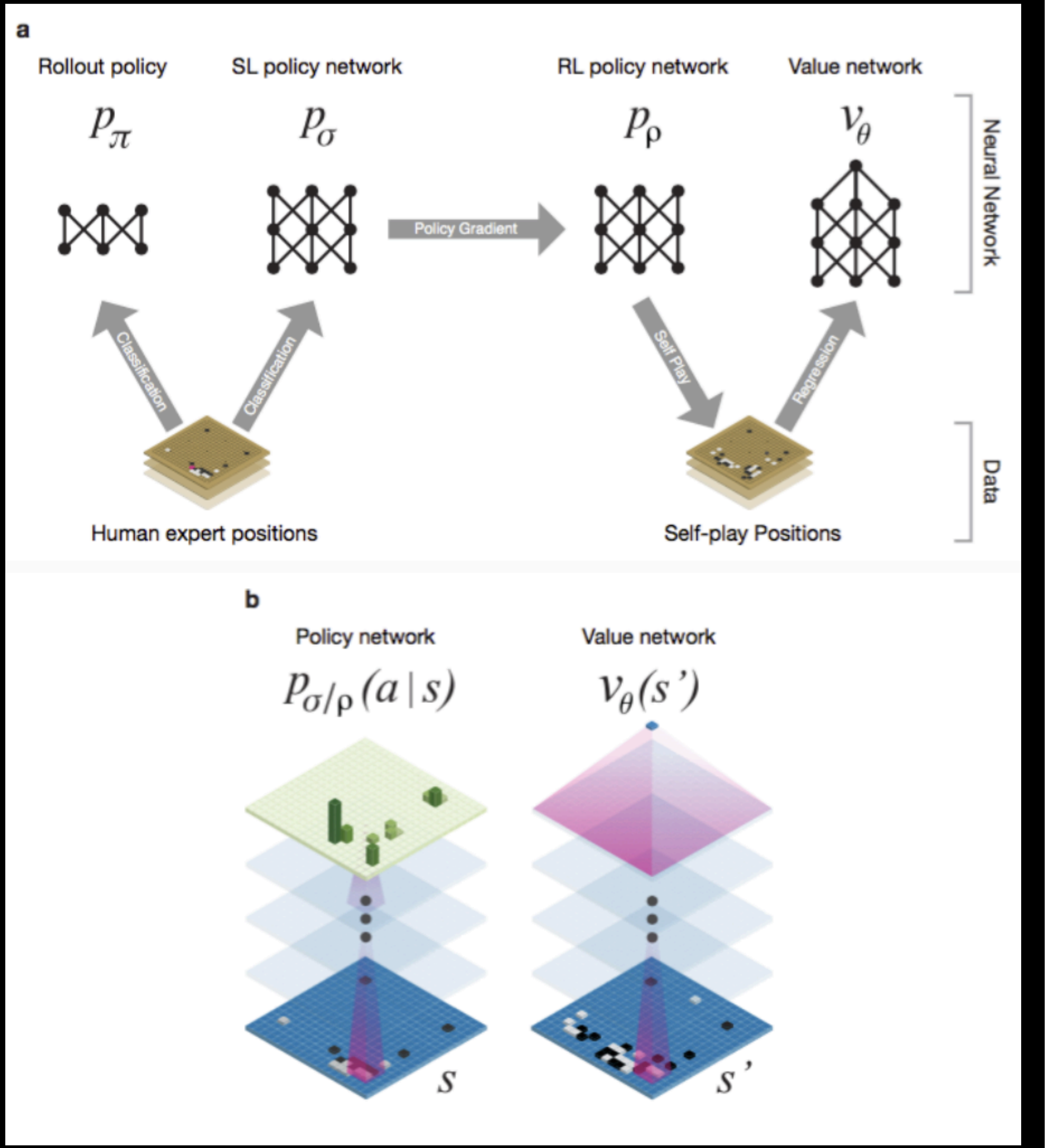




# HOW COULD YOU LEARN A GOOD EVALUATION FUNCTION?

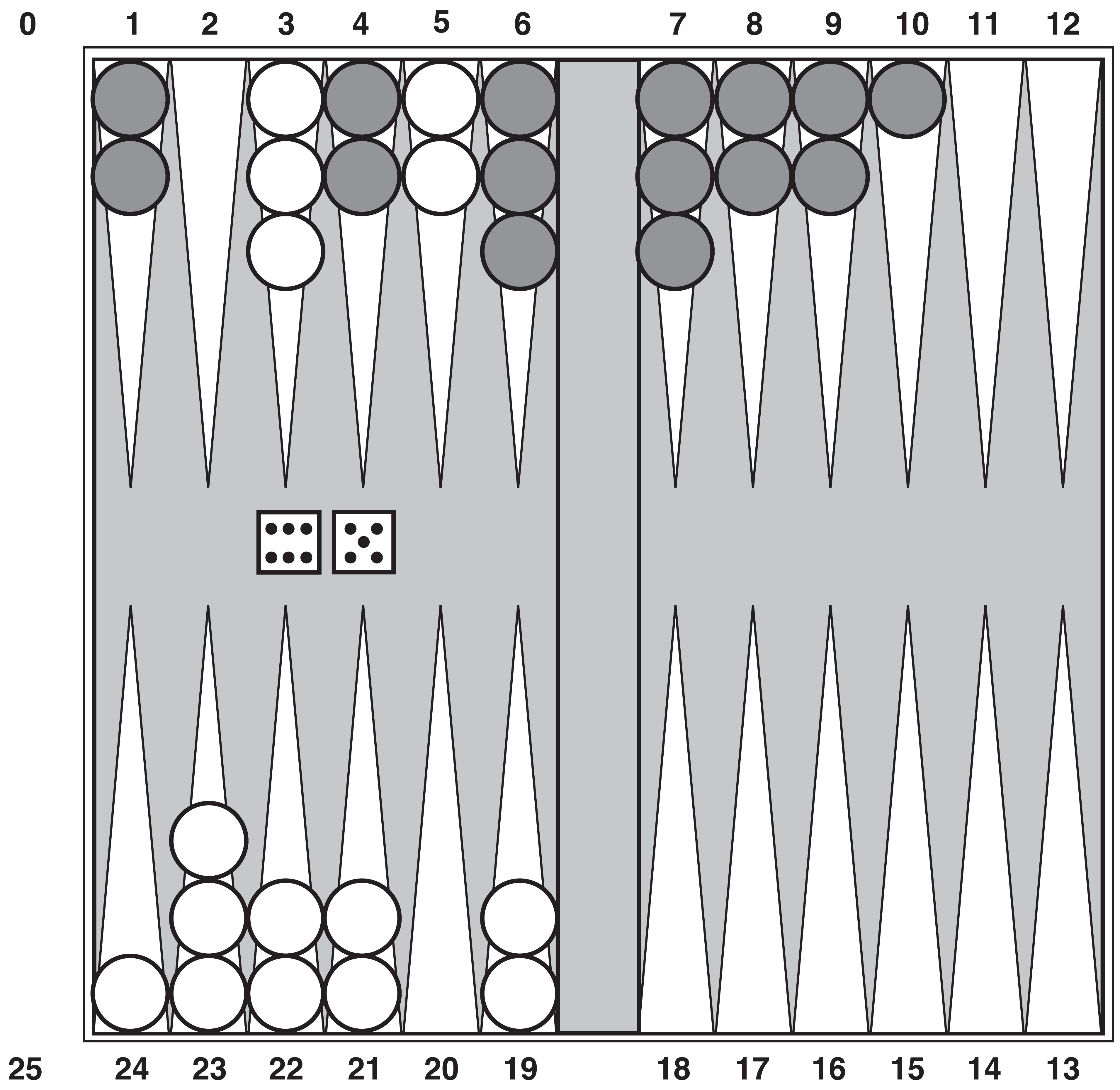
# ALPHAGO SYSTEM

- **Deep learning + Reinforcement learning**
- One model predicts next move, given current state of board, *trained on **30 million positions** from human games*
- Another model predicts likelihood of winning given current state, *trained on **30 million positions** from self-play*
- **System** combines two models using Monte Carlo search

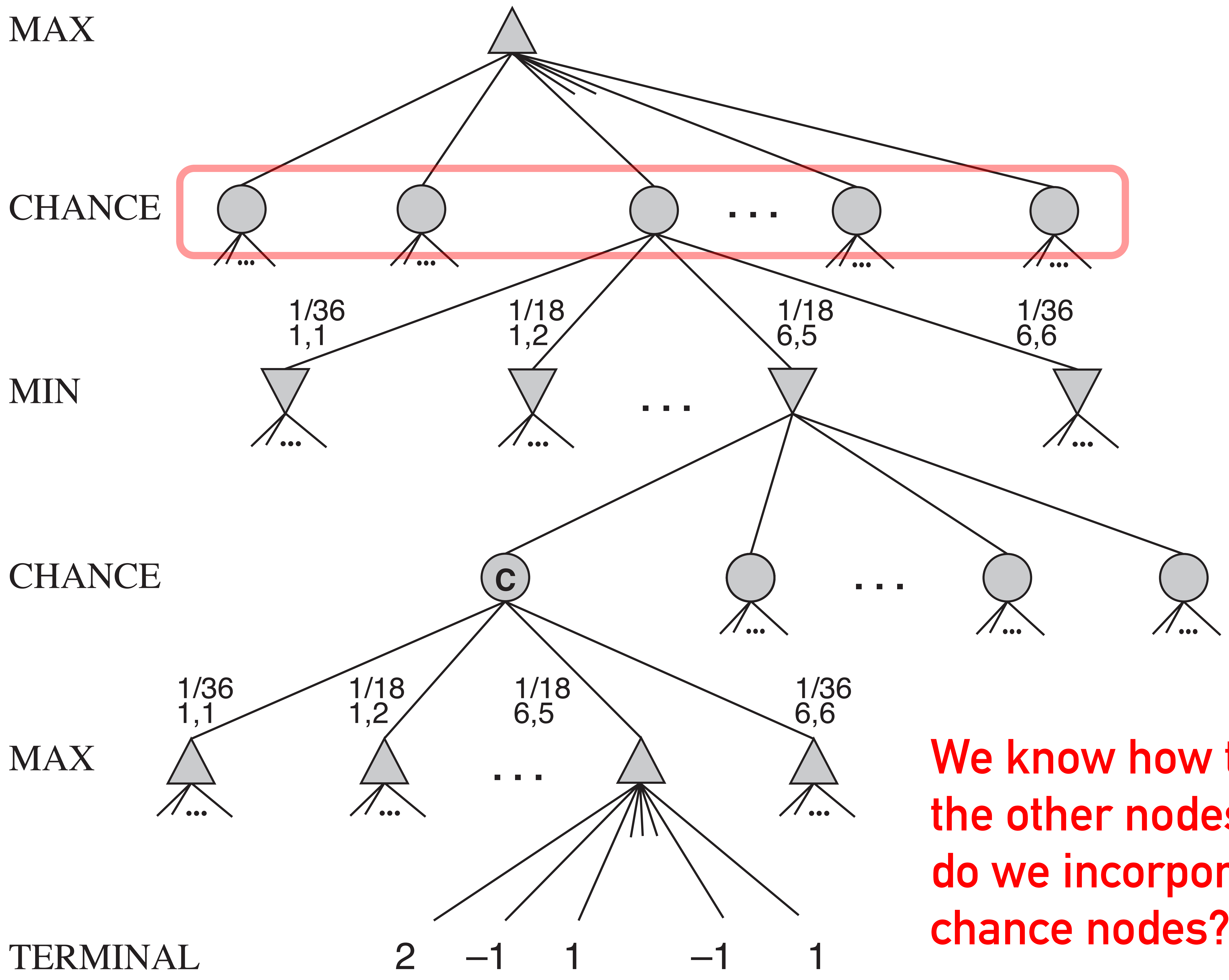



Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529.7587 (2016): 484-489.

# WHAT IF A GAME HAS A "CHANCE ELEMENT"?



# GAME TREE WITH CHANCE ELEMENT




We know how to value the other nodes. How do we incorporate chance nodes?

# EXPECTED VALUE

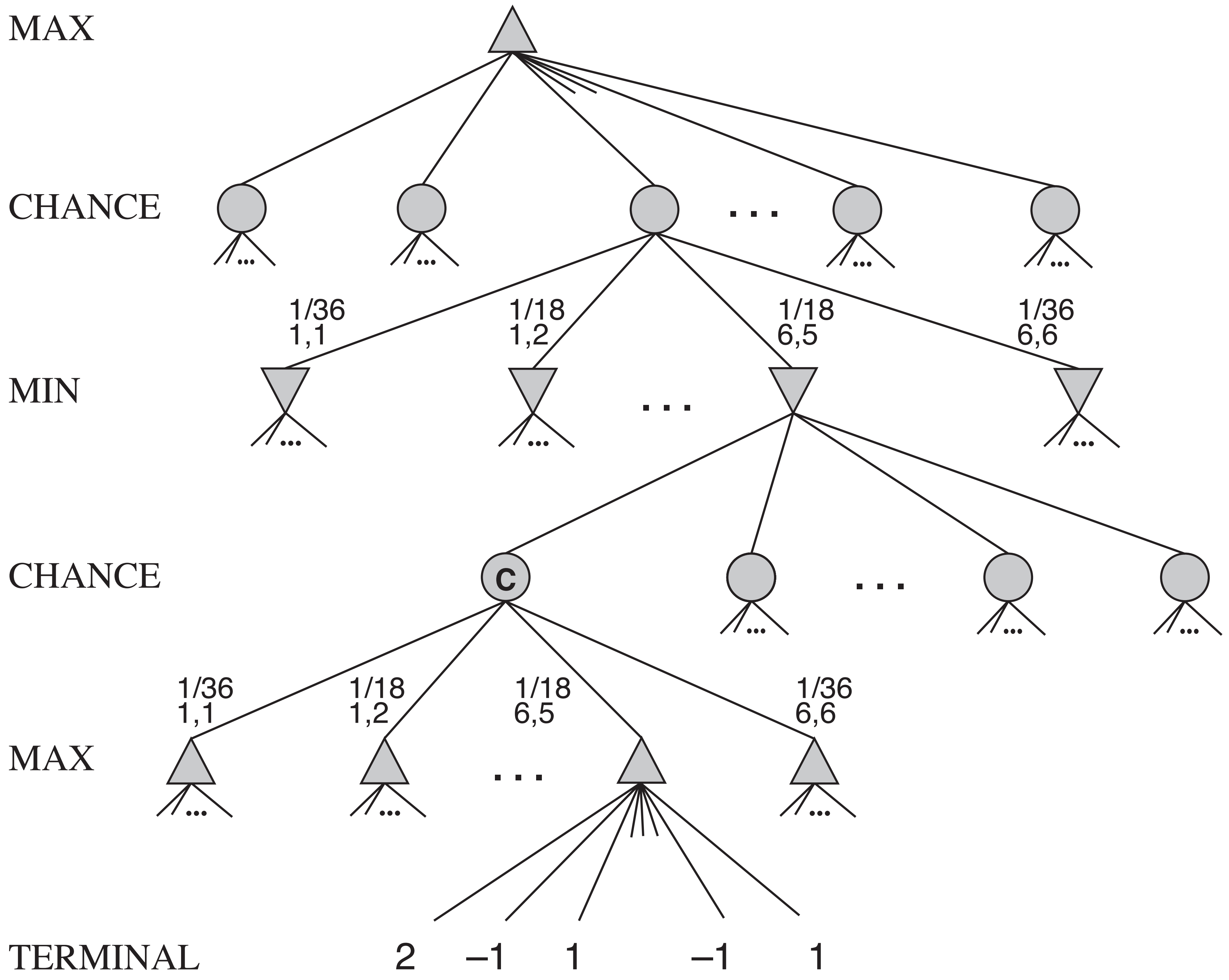
- The sum of the probability of each possible outcome multiplied by its value:

$$E(X) = \sum_i p_i x_i$$

- Are there pathological cases where this statistic could do something strange
  - Extreme values ("outliers")
  - Functions that are a non-linear transformation of the probability of winning

# EXPECTED MINIMAX VALUE

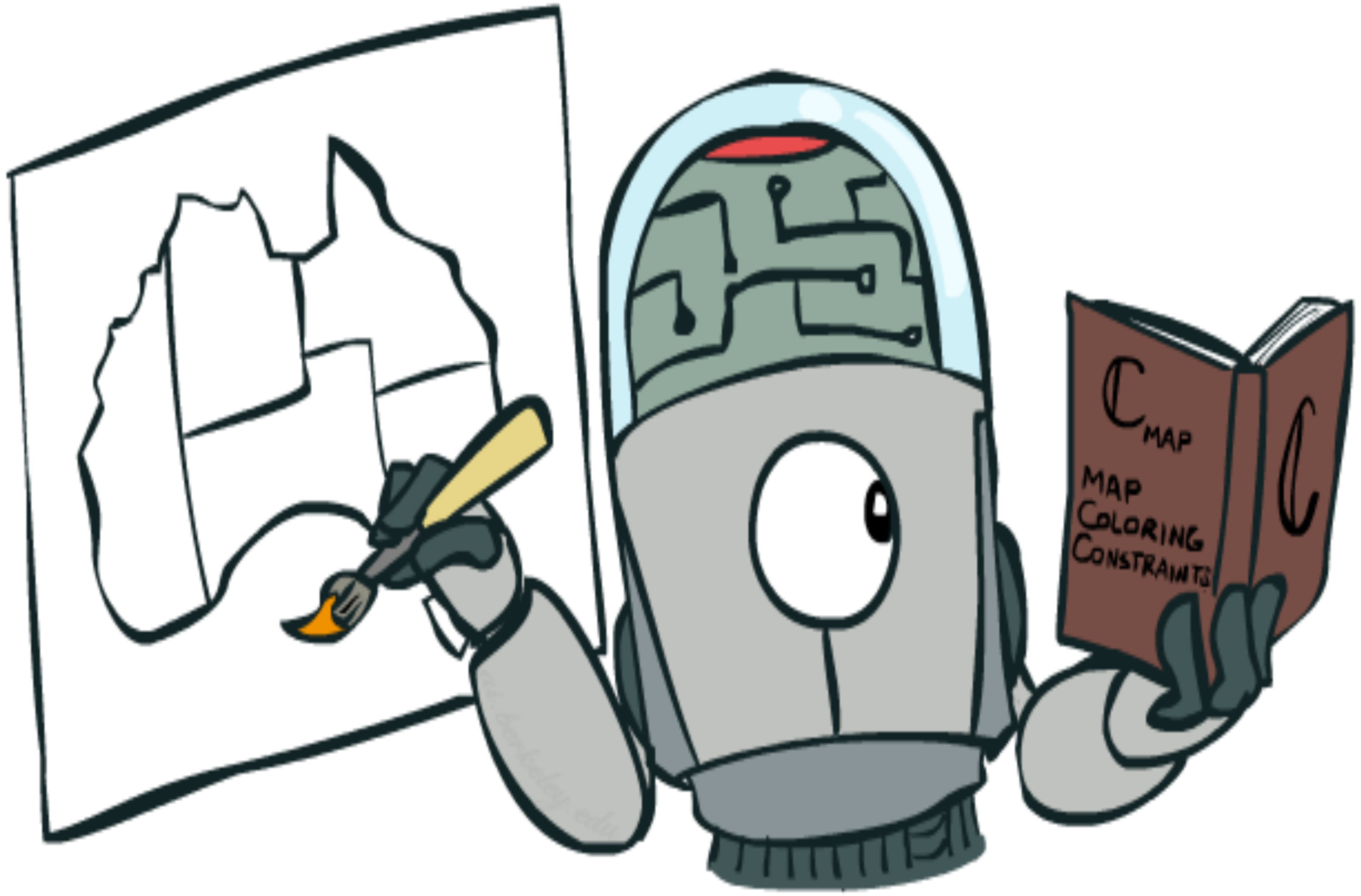
- Now three different cases to evaluate, rather than just two.
  - MAX, MIN, CHANCE




- EXPECTED-MINIMAX-VALUE(n) =

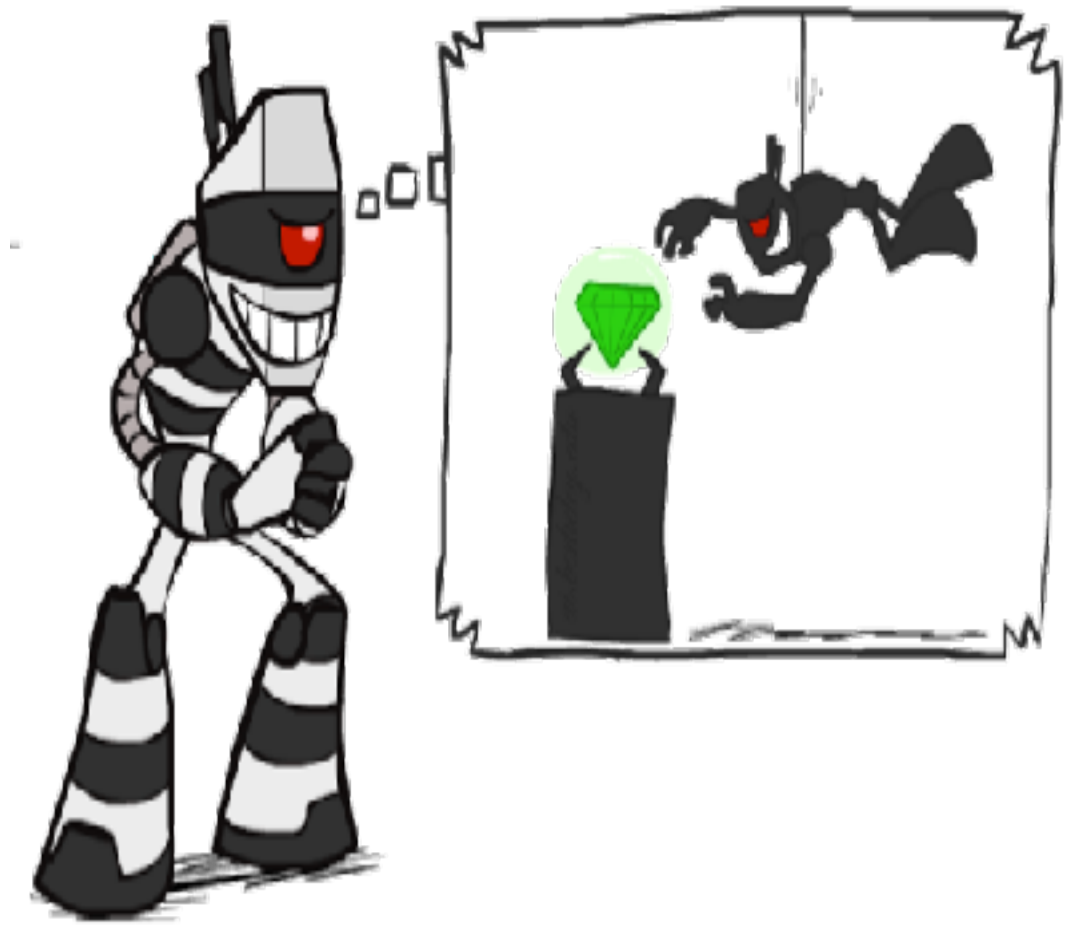
$$
\begin{cases}
\text{UTILITY}(n) & \text{if terminal node} \\
\max_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s) & \text{if MAX node} \\
\min_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s) & \text{if MIN node} \\
\sum_{s \in \text{successors}(n)} P(s) \times \text{EXPECTEDMINIMAX}(s) & \text{if CHANCE node}
\end{cases}
$$

# CONSTRAINT SATISFACTION PROBLEMS



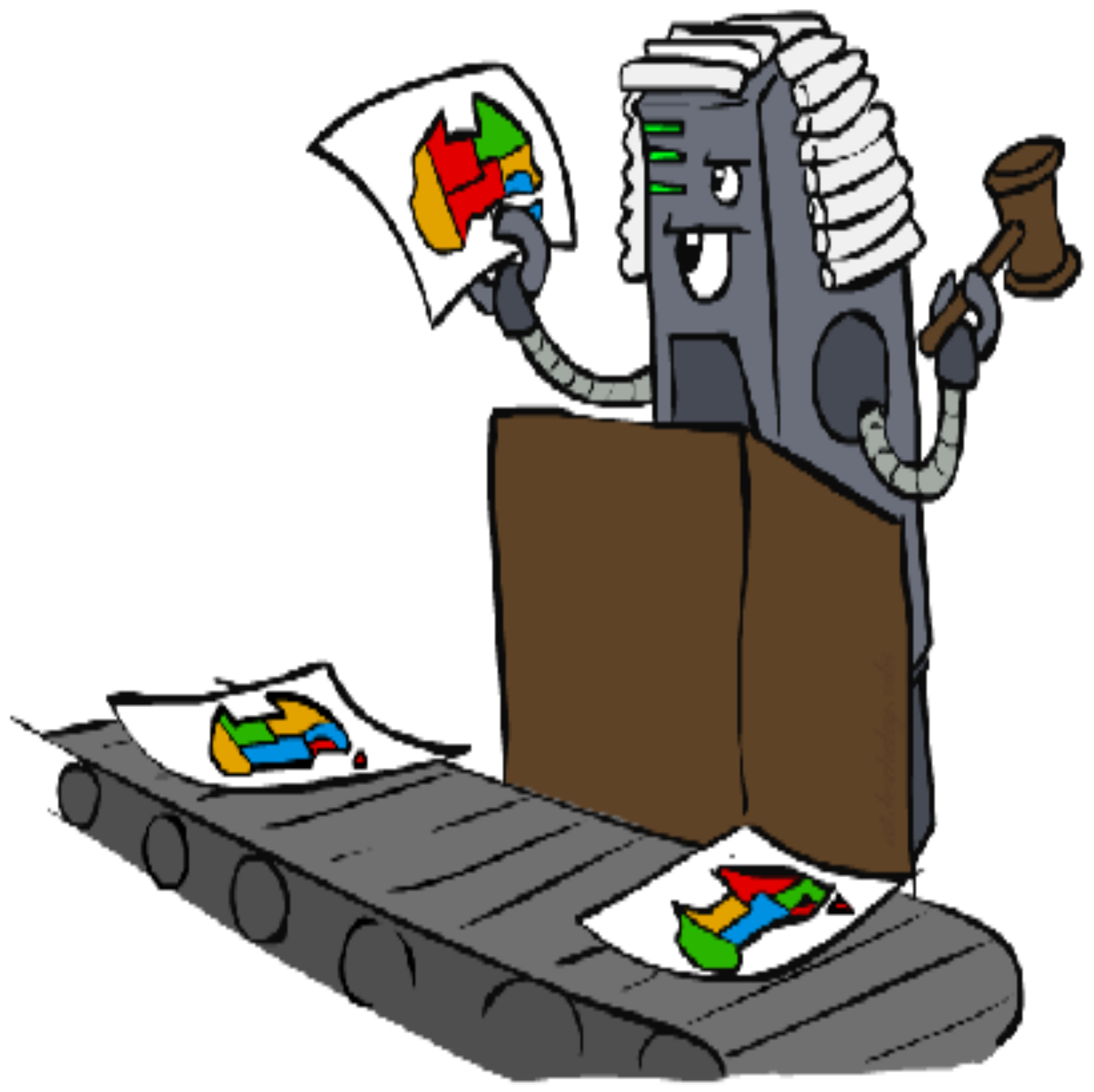
# WHAT IS SEARCH FOR?

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

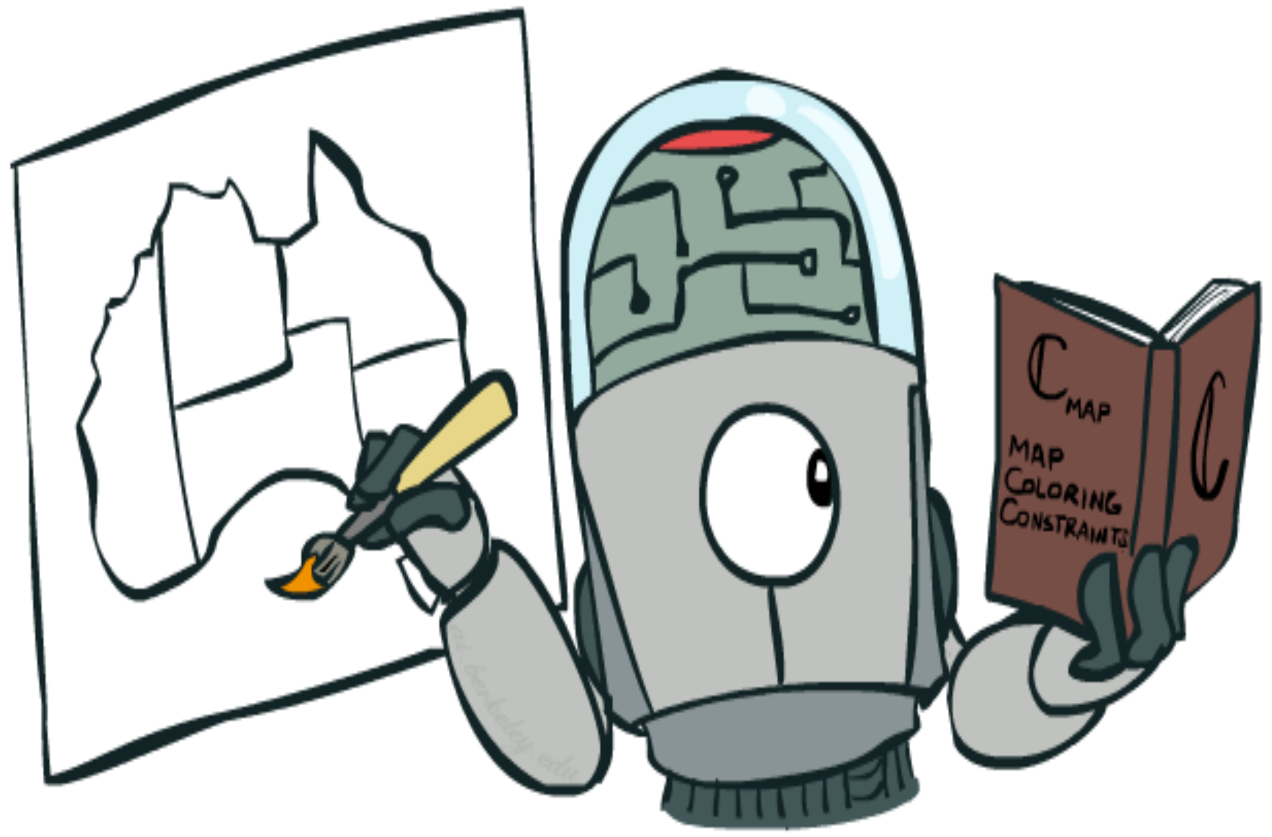


- **Identification**: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)

# CONSTRAINT SATISFACTION PROBLEMS

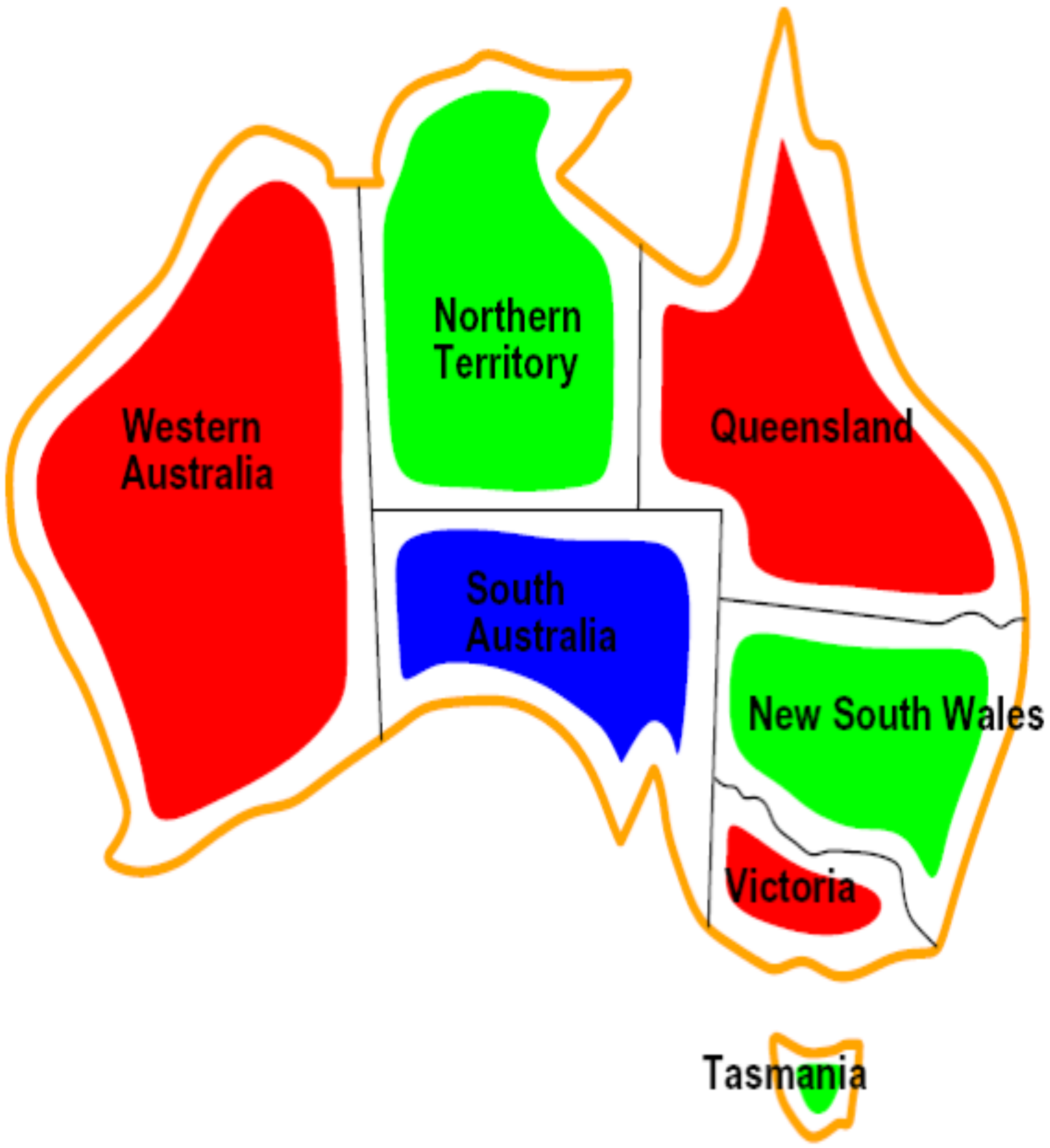


- Standard search problems:
    - State is a "black box": arbitrary data structure
    - Goal test can be any function over states
- Constraint satisfaction problems (CSPs) – a special subset of search problems
    - State is defined by **variables $X_i$** with values from a **domain D**
    - Goal test 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 EXAMPLES



# EXAMPLE: MAP COLORING

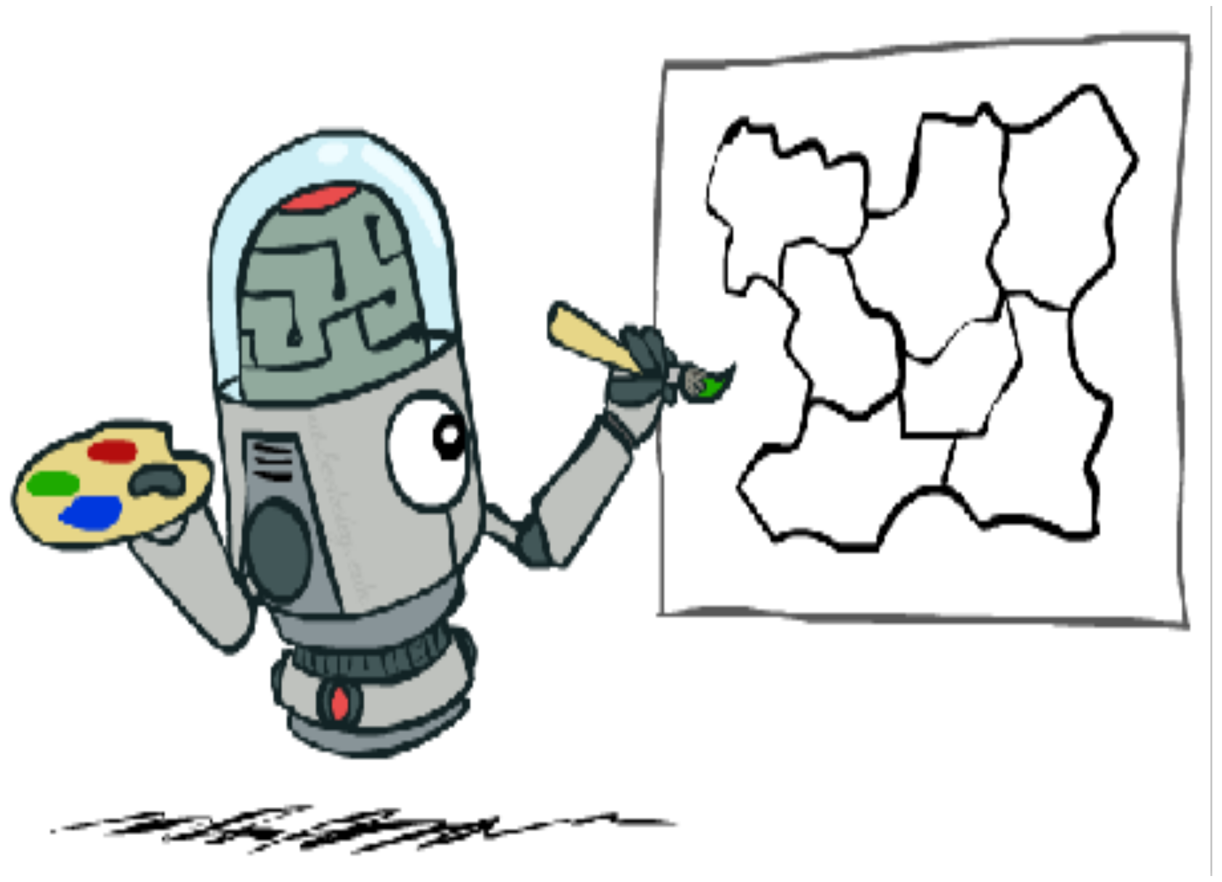
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red}, \text{green}, \text{blue}\}$
- Constraints: adjacent regions must have different colors

Implicit: $WA \neq NT$

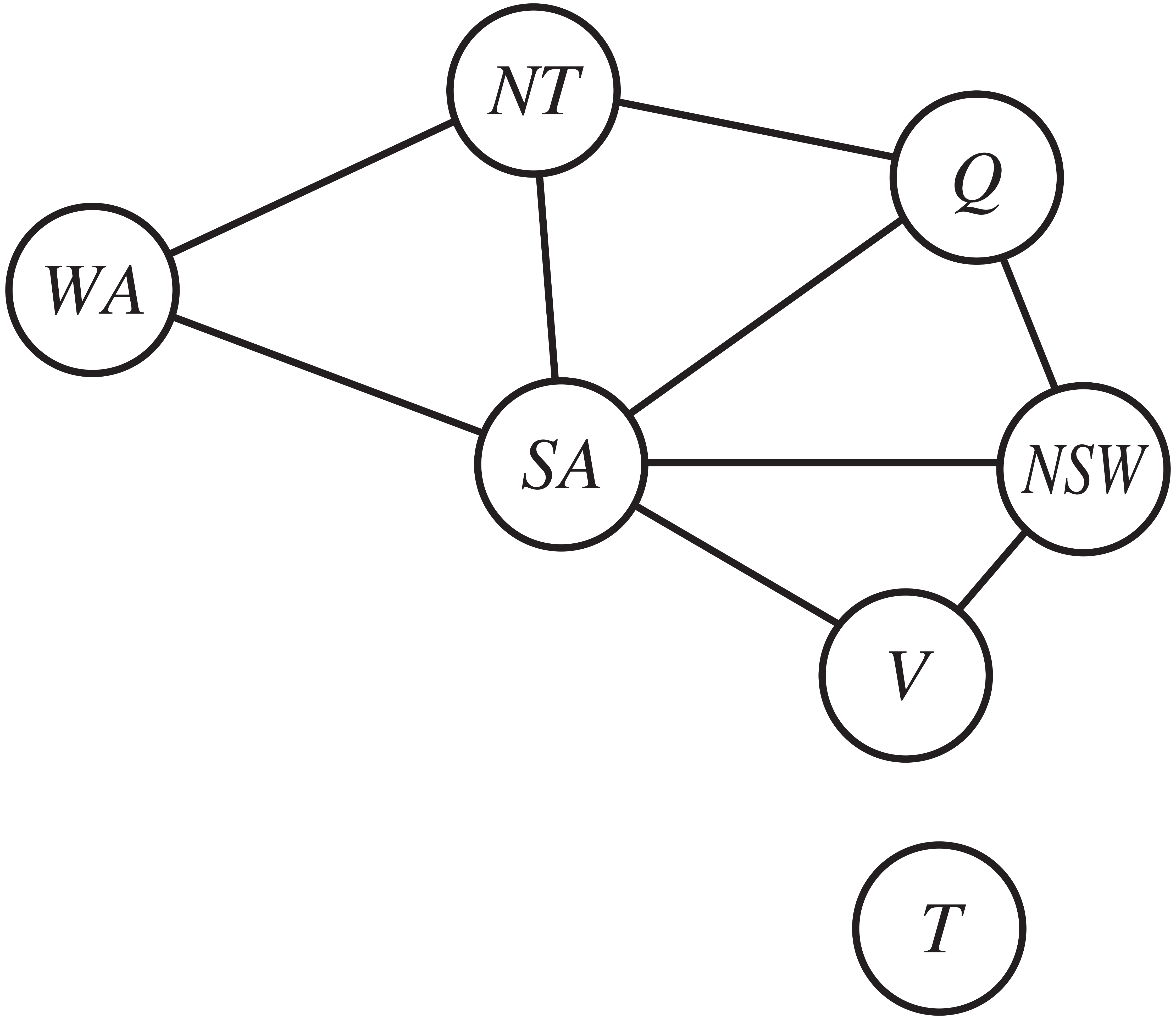
Explicit: $(WA, NT) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), \ldots\}$

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

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

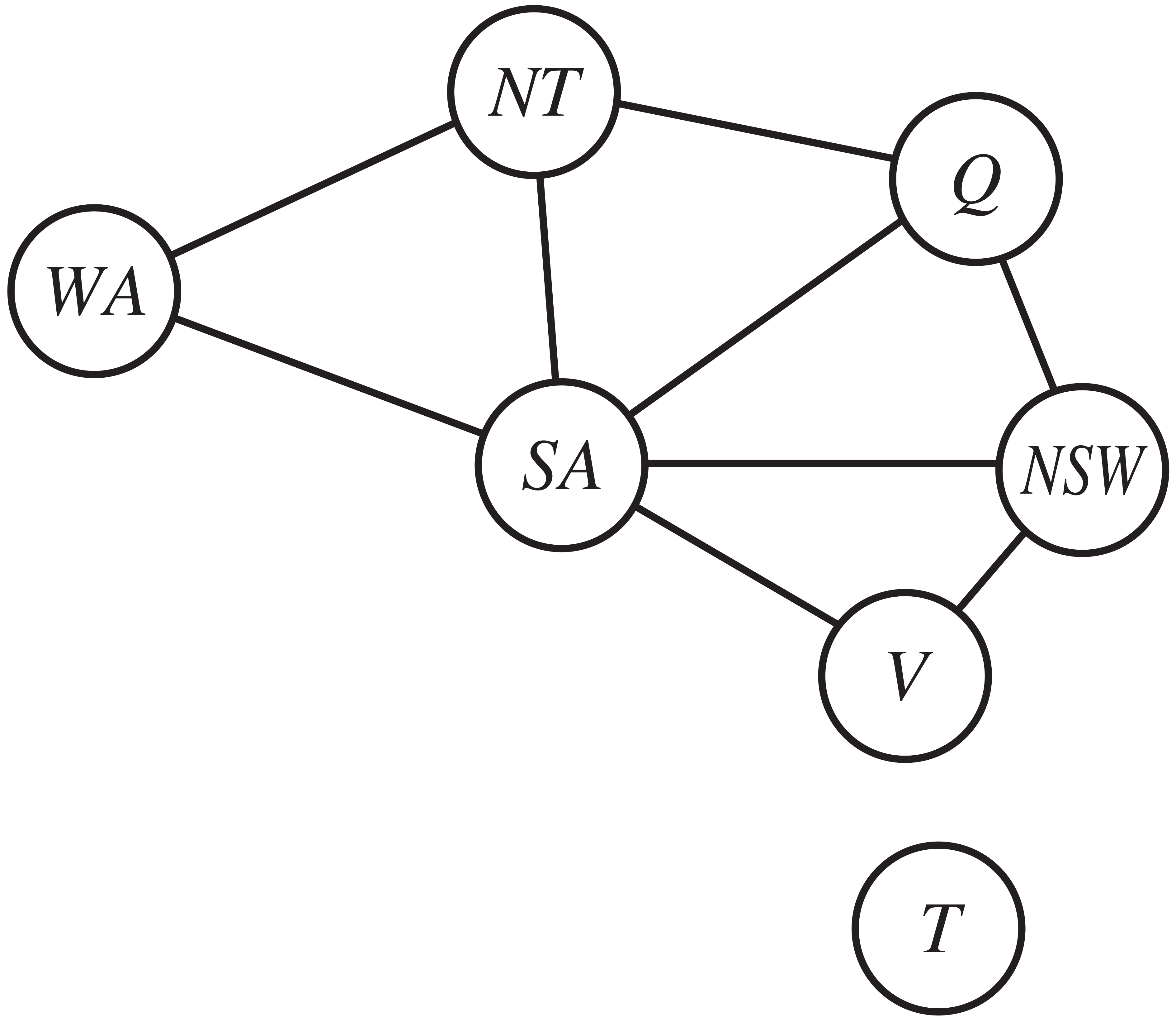



# CONSTRAINT GRAPHS




## 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



# EXAMPLE: N-QUEENS

- Variables: $Q_k$

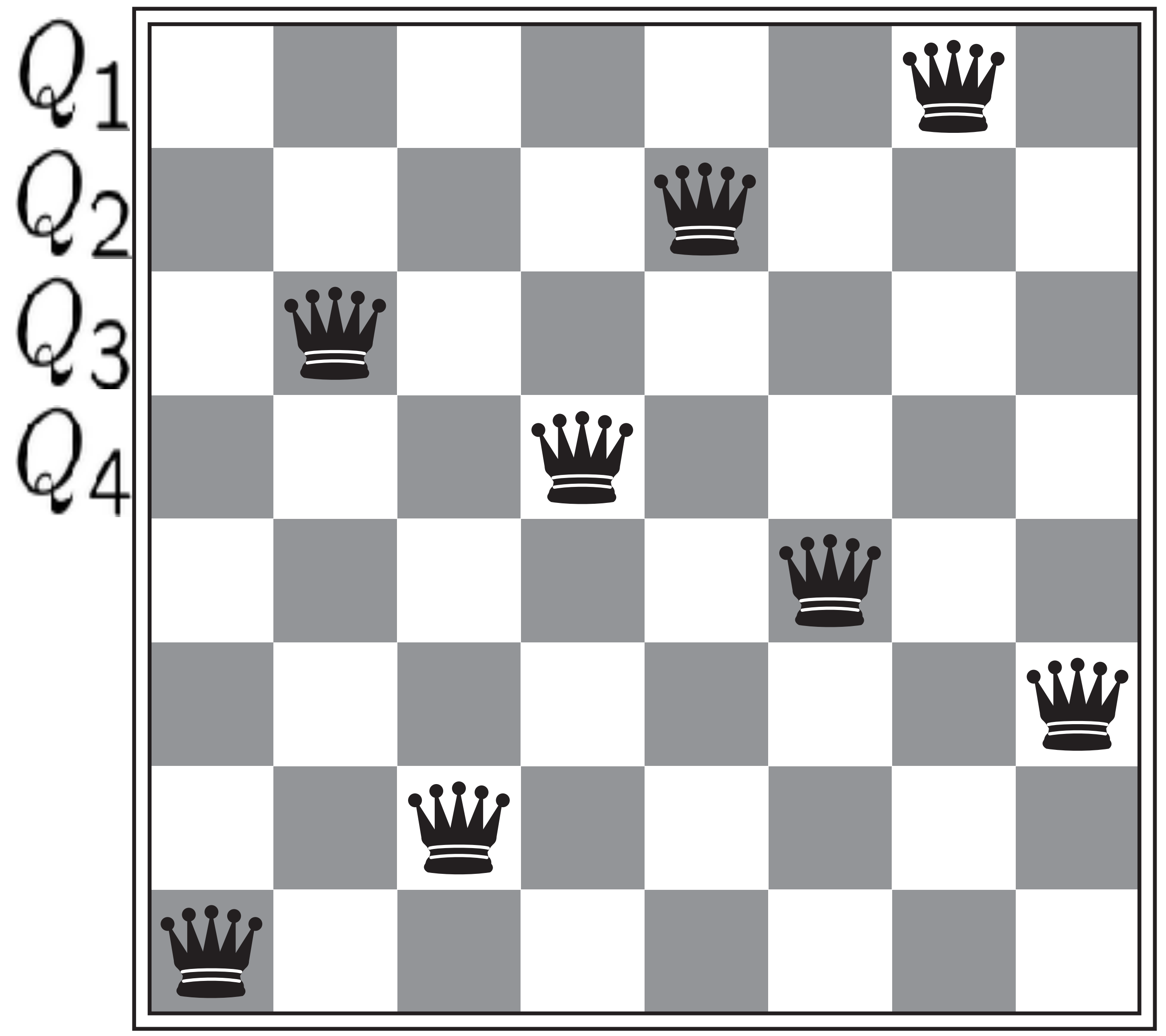
- Domains: $\{1, 2, 3, \ldots N\}$

- Constraints:

Implicit: $\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$

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

$\ldots$



# EXAMPLE: SUDOKU

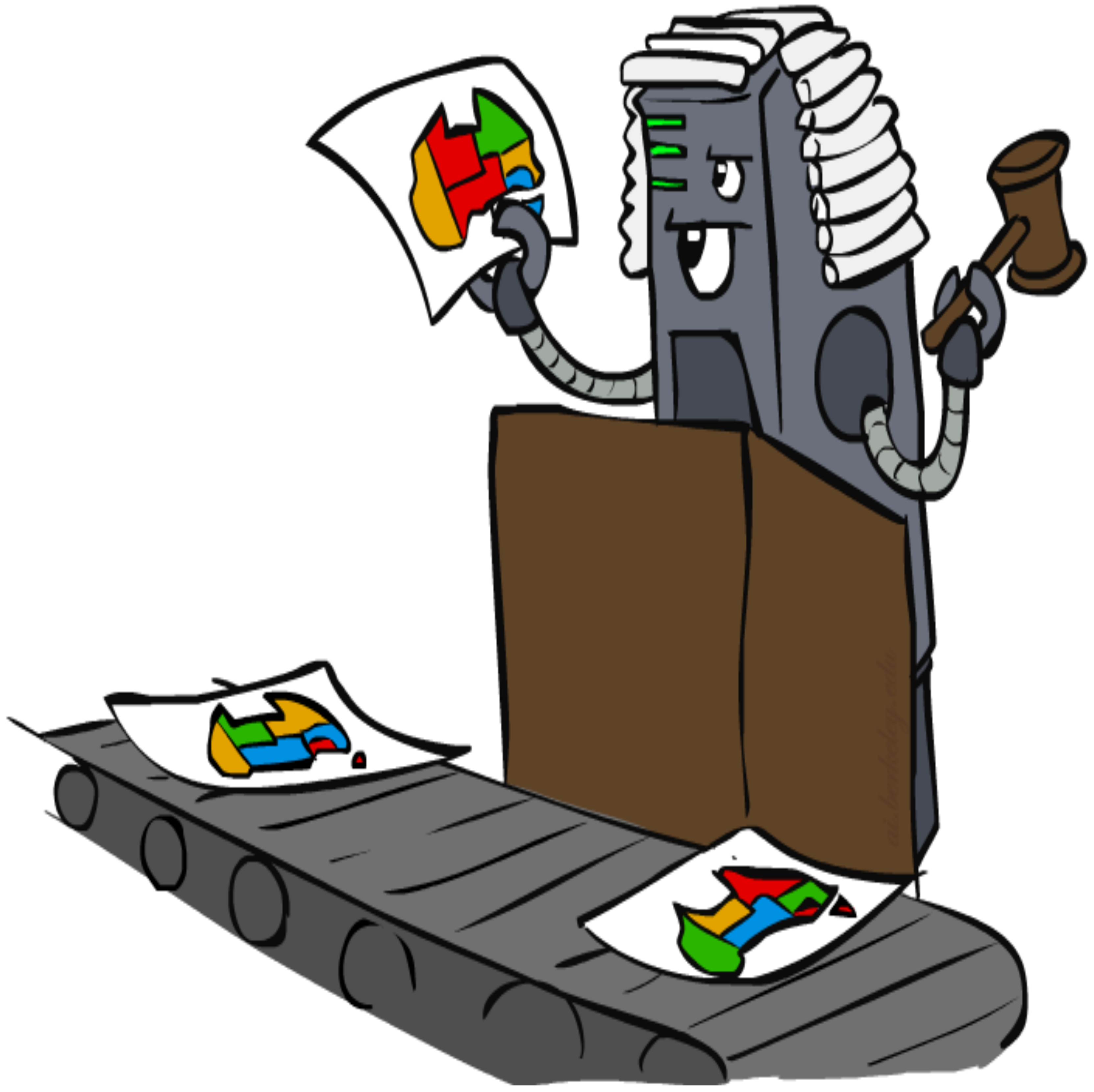
- Objective
  - Fill the empty cells with numbers between 1 and 9
- Rules
  - Numbers can appear only once on each row
  - Numbers can appear only once on each column
  - Numbers can appear only once on each region
- Variables? Domain?
- Constraints?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | | | 4 | | 6 | | | 7 |
| | | | | | | 4 | | |
| | 1 | | | | | 6 | 5 | |
| 5 | | 9 | | 3 | | 7 | 8 | |
| | | | | 7 | | | | |
| | 4 | 8 | | 2 | | 1 | | 3 |
| | 5 | 2 | | | | | 9 | |
| | | 1 | | | | | | |
| 3 | | | 9 | | 2 | | | 5 |

# SOLVING CSPS

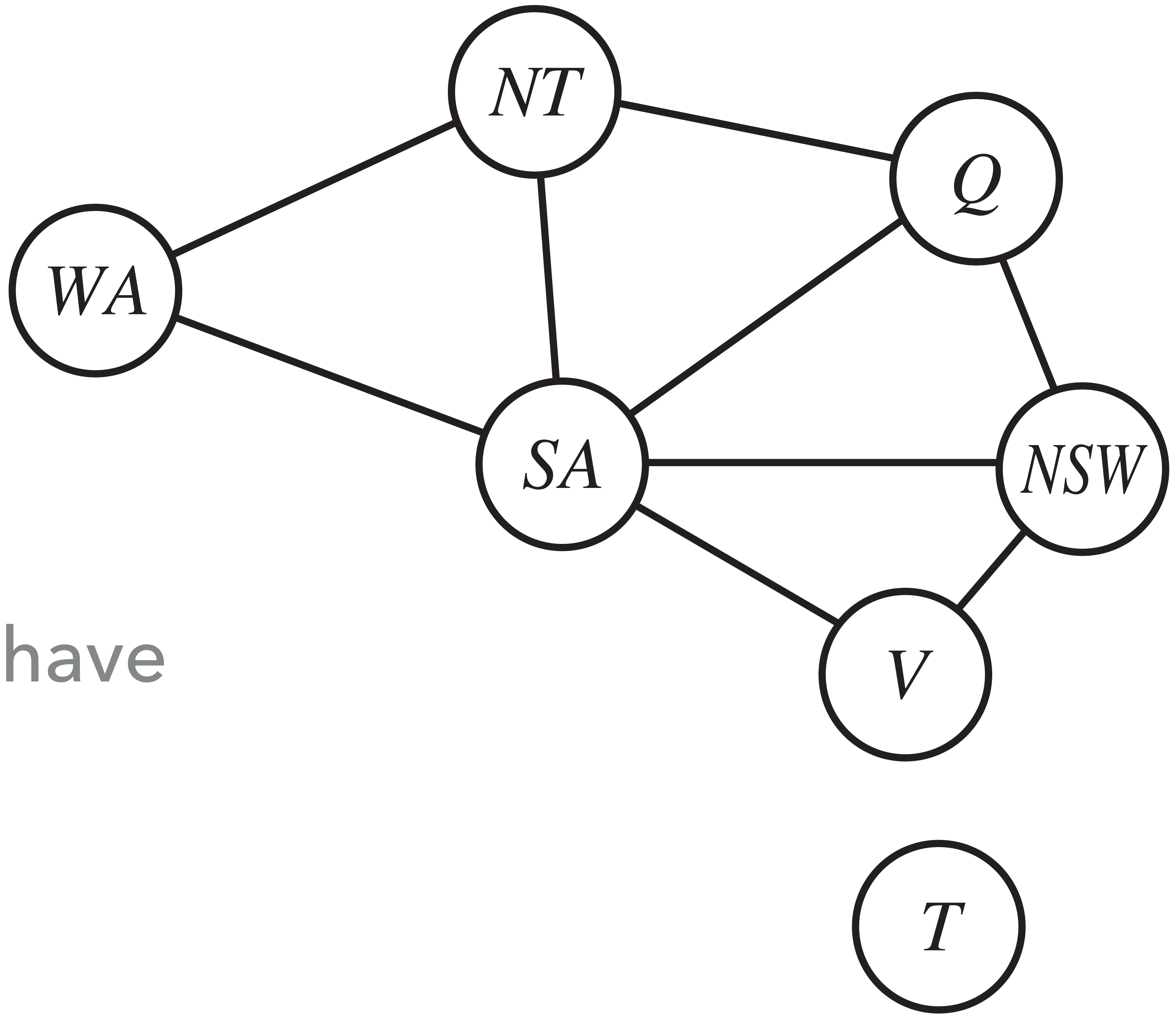
# STANDARD SEARCH FORMULATION

- Standard search formulation of CSPs
- States defined by the values assigned so far (ie. 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

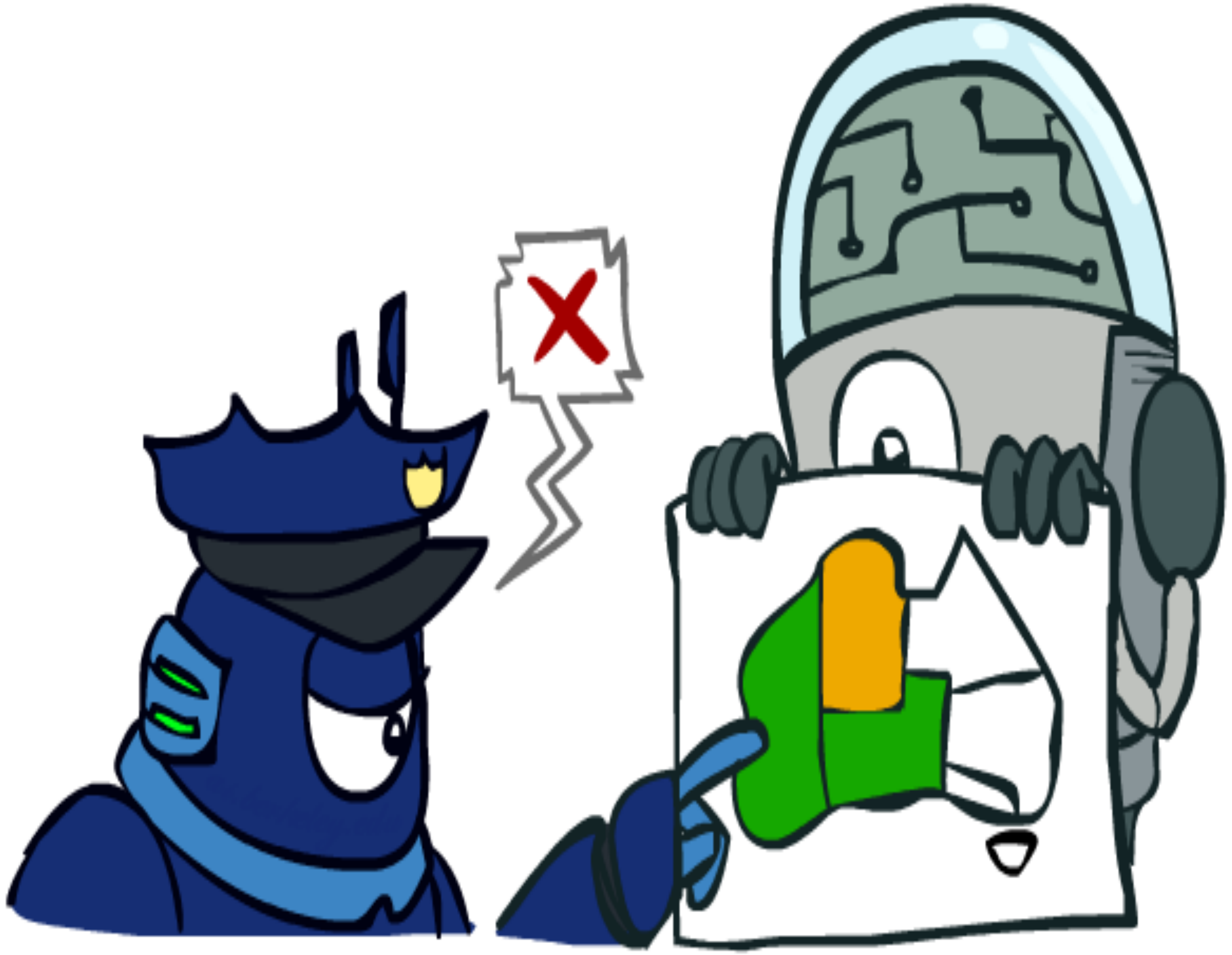


# SEARCH METHODS

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

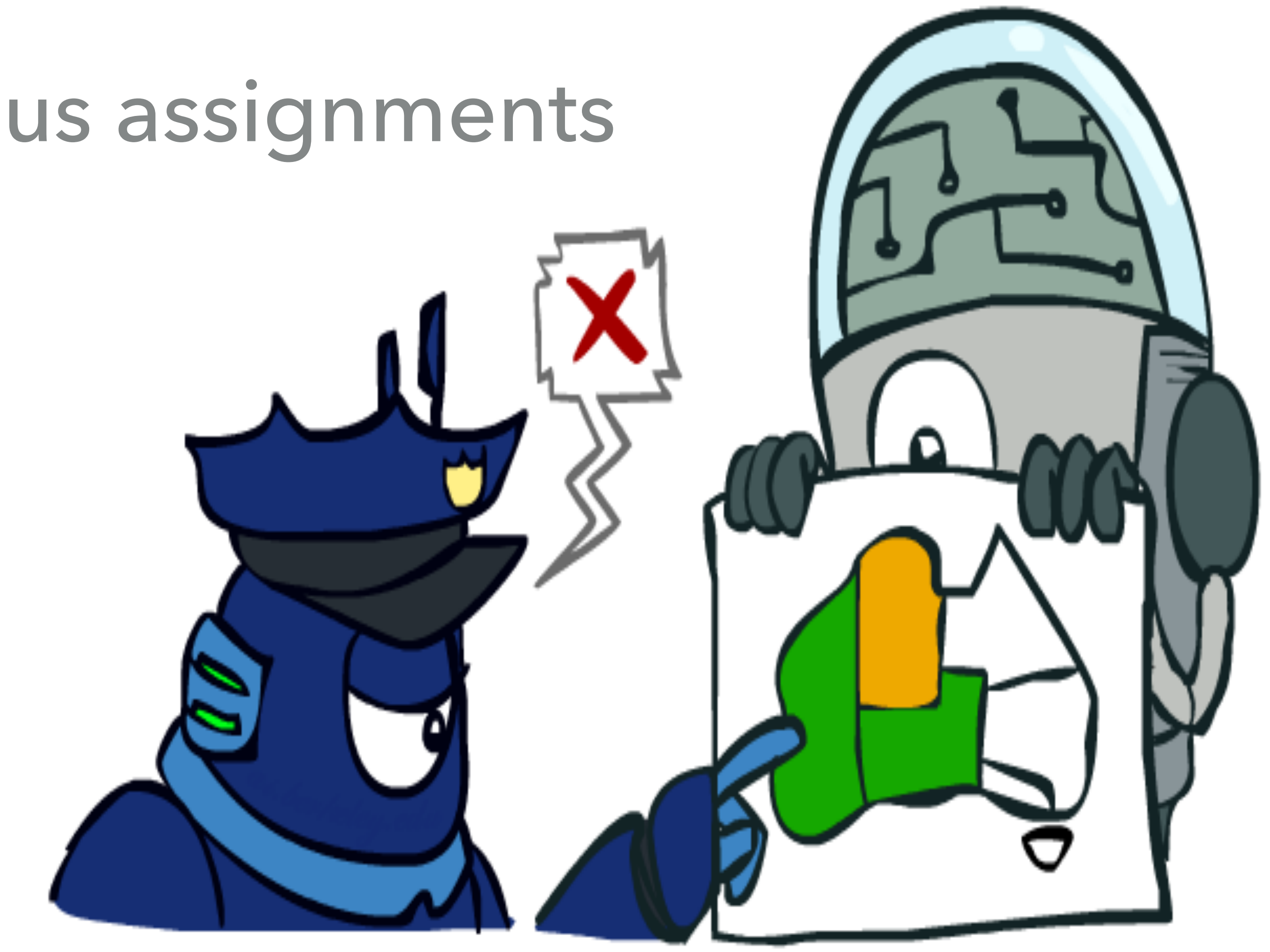


# BACKTRACKING SEARCH



# BACKTRACKING SEARCH

- Backtracking search is the basic uninformed algorithm for  solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering and only consider assignments to a single variable at each step
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
- 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