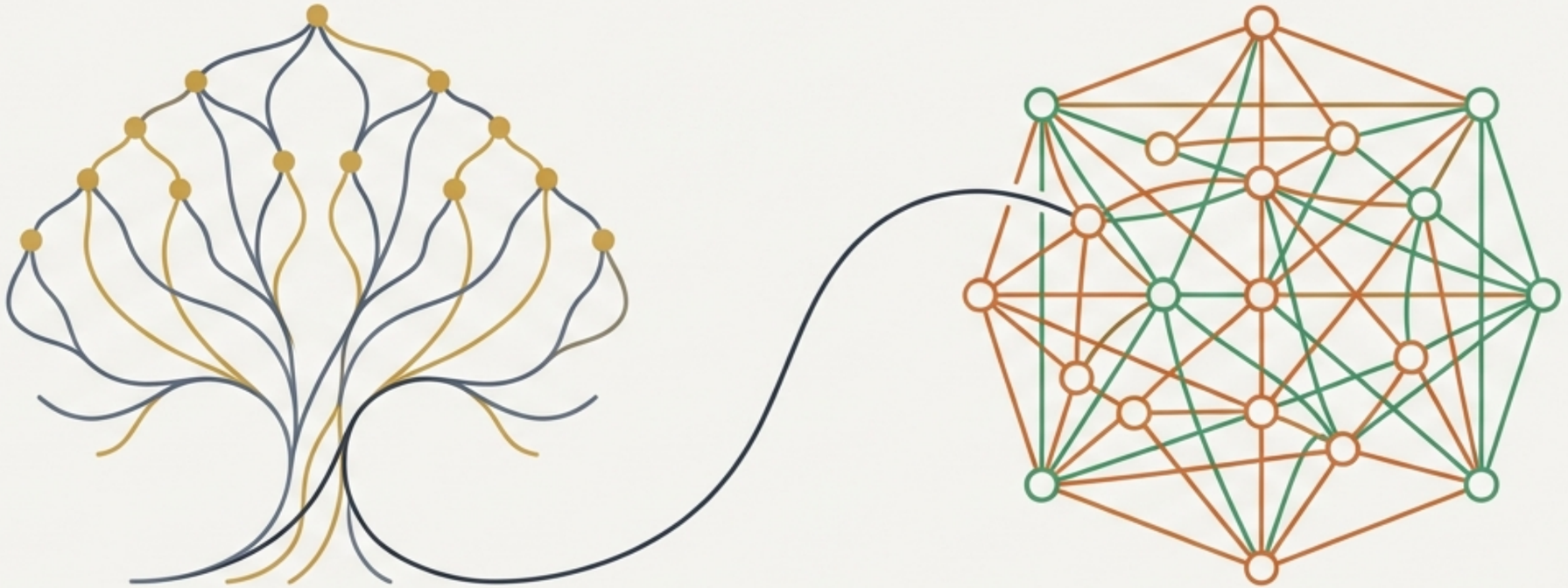
The Intelligent Opponent: A Journey Through Adversarial Search & Problem Solving



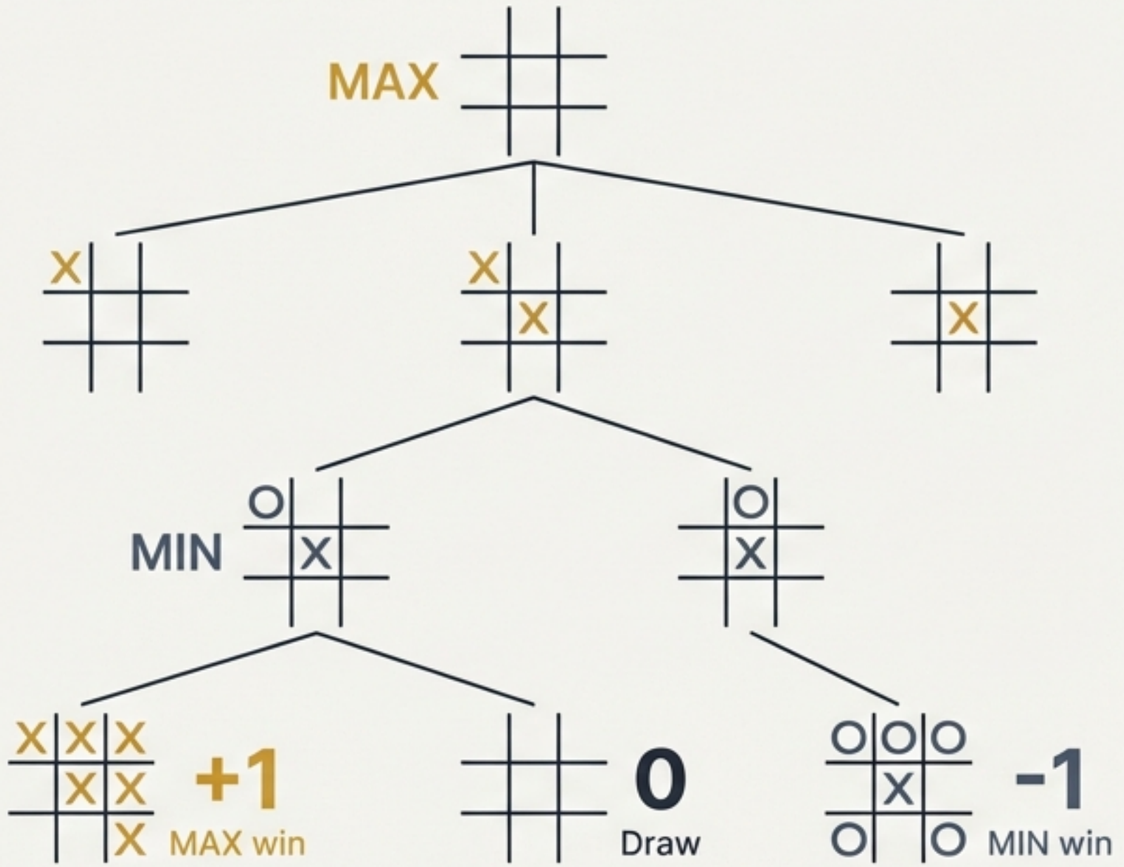
This presentation explores two foundational pillars of Artificial Intelligence for solving complex problems. We begin with the strategic challenge of outmanoeuvring an opponent in competitive games, a journey from perfect but impractical logic to clever, efficient search. We then shift perspective to the logic of constraint satisfaction, where the structure of the problem itself becomes the key to its solution.

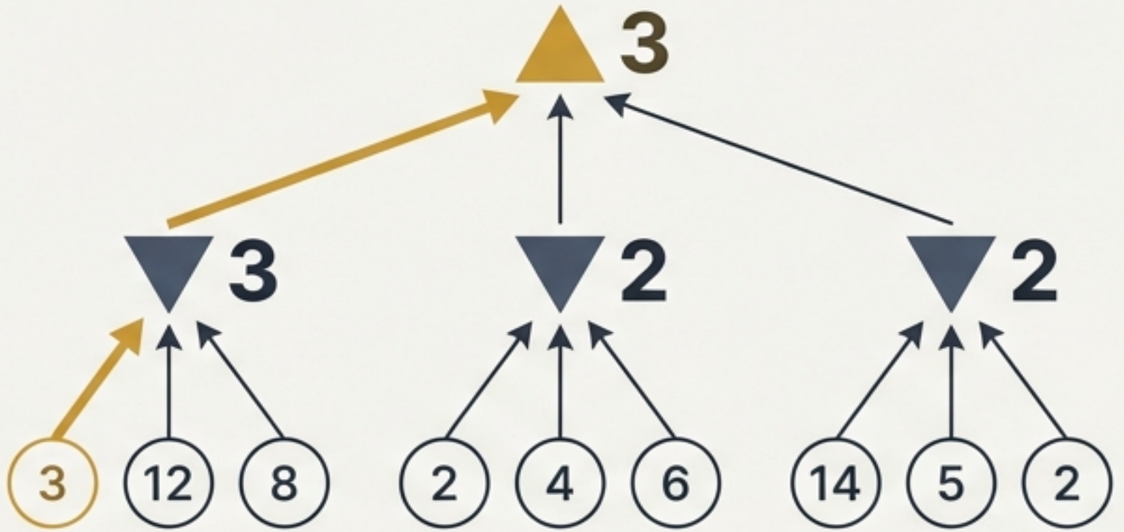
Part I: The Art of Adversarial Search

Part II: The Logic of Constraint Satisfaction

Playing the Perfect Game: The Minimax Principle

In a deterministic, two-player, zero-sum game, there is a provably optimal move. The Minimax algorithm finds it by assuming your opponent will also play perfectly to thwart you.





The algorithm explores the game tree to the very end. At each level, it calculates a node's "minimax value":

- MAX nodes (▲) choose the move leading to the highest value.
- MIN nodes (▼) choose the move leading to the lowest value.

The optimal move for MAX at the root is the one leading to the successor with the highest minimax value.

The Problem of Scale: When Perfect is Impractical

The Minimax algorithm guarantees the optimal move, but at an impossible cost. It must perform a **complete depth-first** exploration of the game tree.

Game Tree Size:
< 362,880 terminal nodes.
Feasible to explore.

$> 10^{123}$

Game Tree Size:
$> 10^{123}$ nodes.
Impossible to explore.

For context, there are an estimated 10^{80} atoms in the observable universe.

Technical Breakdown

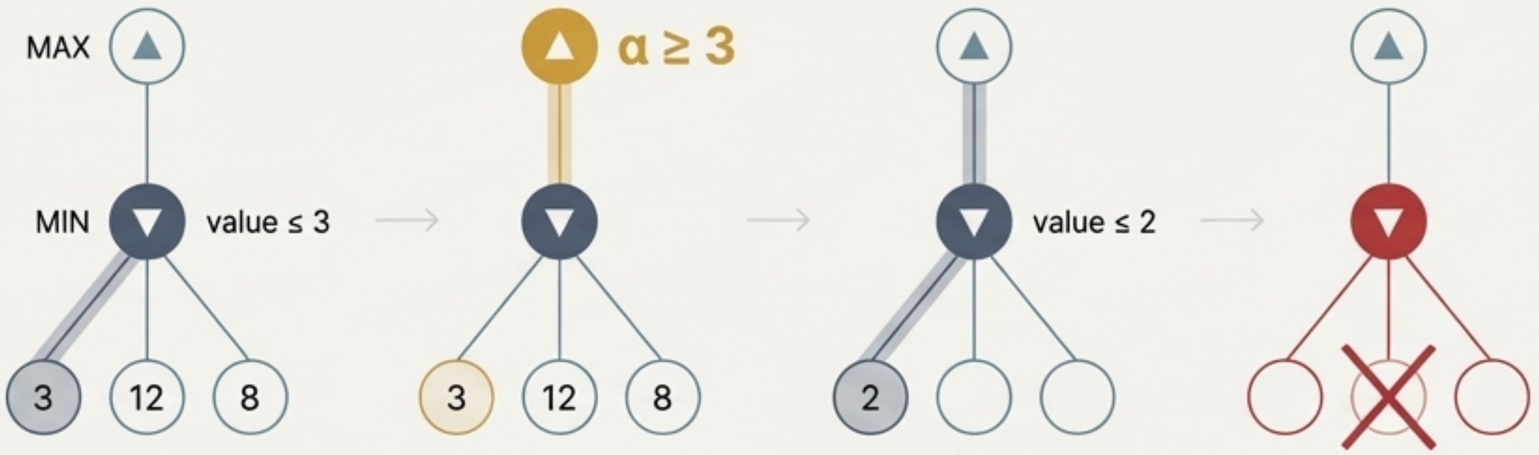
Time Complexity: $O(b^m)$

* b: branching factor (legal moves per turn)
* m: maximum depth of the game (plies)

Example: Chess. With a branching factor **(b)** of ~35 and game depth (m) of ~80 ply, a full search is not feasible within the lifetime of the universe. We need a way to find the optimal move without looking at every single possibility.

A Smarter Search: Pruning the Impossible with Alpha-Beta

We can compute the correct minimax decision without exploring the the entire tree. The key is to ignore branches that we know, logically, cannot influence the final outcome.



MAX now has a path worth at least 3.

This path guarantees a score of ≤ 2 for MAX. Since MAX already has a path worth ≥ 3, this entire branch is irrelevant. **PRUNED.**

Key Terminology & Payoff

α (alpha): The best value (highest score) found so far for MAX along the path. Think "at least".

β (beta): The best value (lowest score) found so far for MIN along the path. Think "at most".

The Rule: Pruning occurs when v ≥ β in a MAX node or v ≤ α in a MIN node.

The Payoff

With perfect move ordering, Alpha-Beta reduces the effective branching factor from b to $\sqrt{b}$. For chess, that's a reduction from ~35 to ~6, allowing programs to search roughly twice as deep in the same amount of time.

Beyond the Horizon: Using Heuristics When You Can't See the End

Even with pruning, searching to a terminal state is often impossible. We must cut the search off at a fixed depth and *estimate* the value of the resulting positions.

The Solution:

- Introduce the concept of a **Heuristic Evaluation Function (EVAL)**.
- Instead of a `UTILITY` function that only works on terminal nodes, we use an `EVAL` function that estimates a non-terminal state's utility.
- Formula: $\texttt{H-MINIMAX}(s,d) = \texttt{EVAL}(s, \texttt{MAX})$ if $\texttt{IS-CUTOFF}(s,d)$ is true.

Mid-game Position

Weighted Linear Function:

$\text{EVAL}(s) = \sum w_i * f_i(s)$

f_1: Pawn count (weight: 1.0)
f_2: Knight count (weight: 3.0)
f_3: Rook count (weight: 5.0)
f_4: Queen count (weight: 9.0)
f_5: King safety (weight: 0.5)

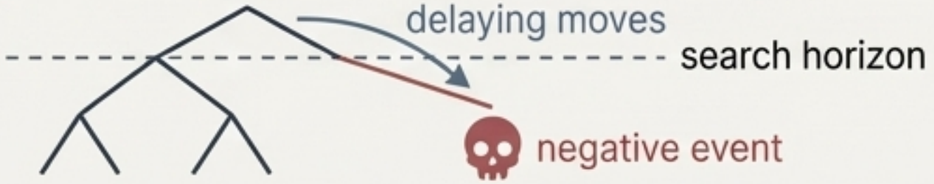
This function quickly calculates a score for a board state, providing an educated guess about who is winning.

Critical Caveats

Quiescence: The evaluation must only be applied to "quiet" positions. A pending capture could dramatically swing the evaluation on the next move.

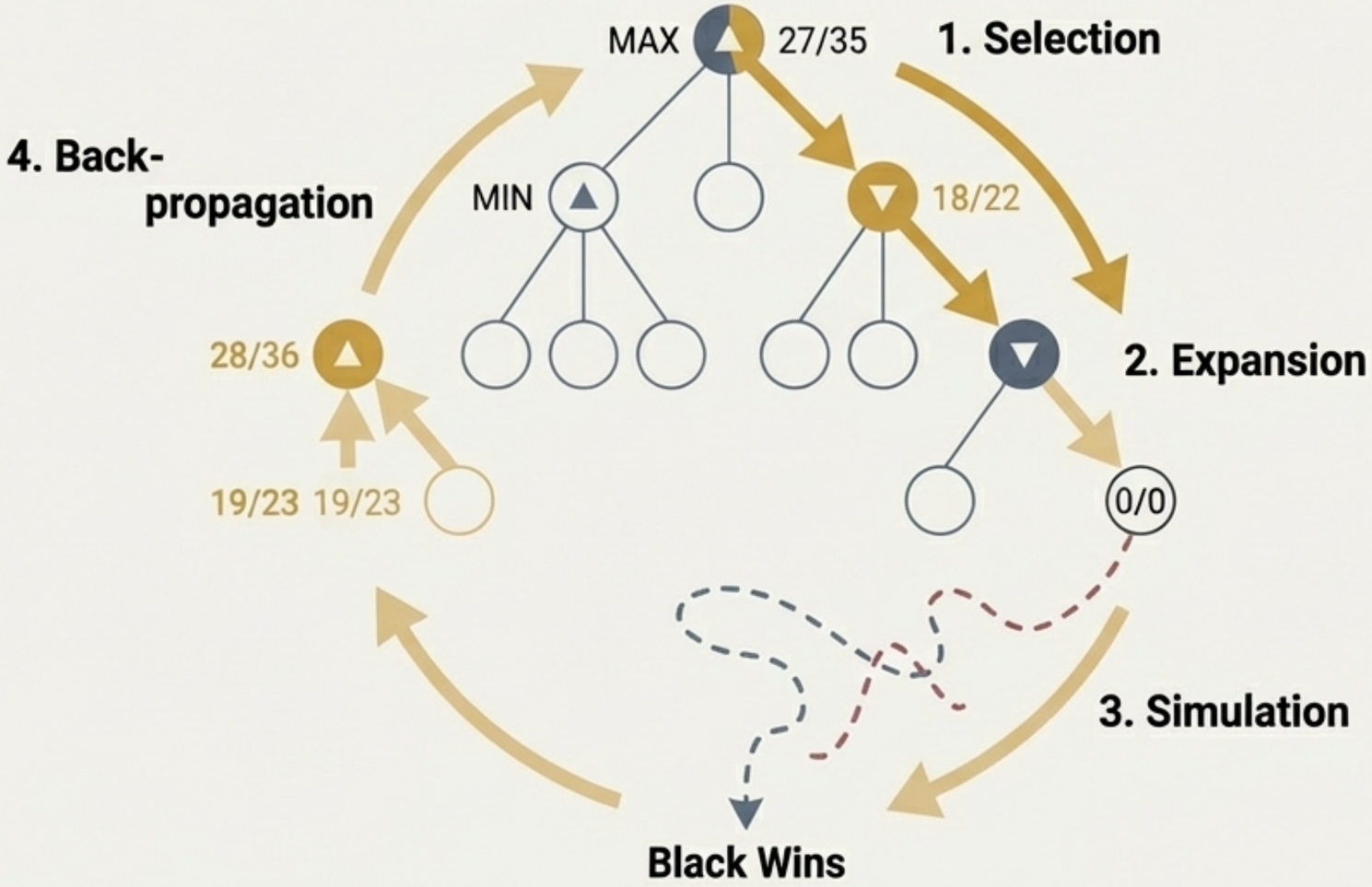
Black appears ahead on material, **but this is not a "quiet" position.**

Horizon Effect: A program might make delaying moves to push an unavoiroidable negative event beyond its search depth "horizon", making a bad situation look artificially good.



Embracing Uncertainty: Monte Carlo Tree Search (MCTS)

A New Approach: For games with huge branching factors (like Go, b≈361) or hard-to-evaluate positions, a different strategy is needed. MCTS estimates a move's value not with a **heuristic** function, but by simulating thousands of random games to see which move leads to victory most often.



The Balancing Act: MCTS elegantly resolves the tension between:

- **Exploitation:** Favouring moves that have performed well in the past.
- **Exploration:** Trying out less-visited moves to see if they might be hidden gems.

The UCT formula balances these two needs, ensuring the search eventually converges on the best moves.

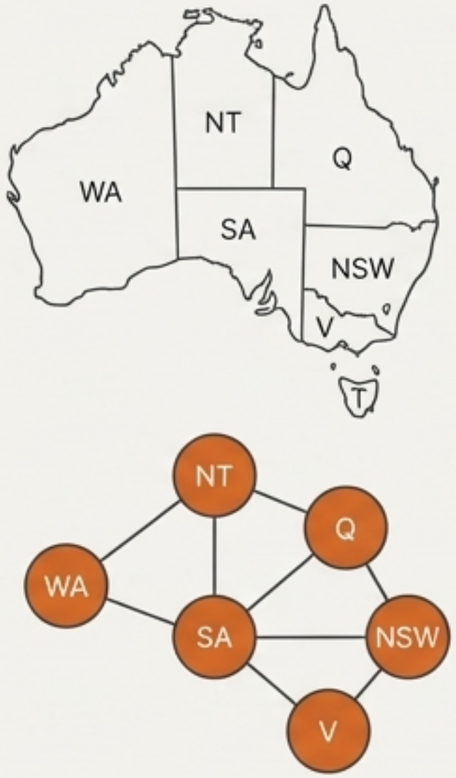
Key Advantage: MCTS is less vulnerable to a single evaluation error than Alpha-Beta and can work for any game where you know the rules, even without expert-designed heuristics.

A New Perspective: Solving Problems by Satisfying Constraints

So far, we have treated states as atomic black boxes. We now break them open. Instead of searching for a path to a goal, we define a problem by its variables and the constraints they must obey. This is a **Constraint Satisfaction Problem (CSP)**.

Formal Definition

- A CSP is defined by three components:
- **X:** A set of variables. $\{X_1, \ldots, X_n\}$
- **D:** A set of **domains** (allowable values for each variable). $\{D_1, \ldots, D_n\}$
- **C:** A set of **constraints** that specify allowable combinations of values.



CSP Formulation of the Example

- **Australian Map Colouring**
- **Variables X:** {WA, NT, Q, NSW, V, SA, T}
- **Domain D_i:** {red, green, blue} for each variable.
- **Constraints C:** {SA ≠ WA, SA ≠ NT, …}. No two neighbouring regions can have the same colour.

The Power of CSPs

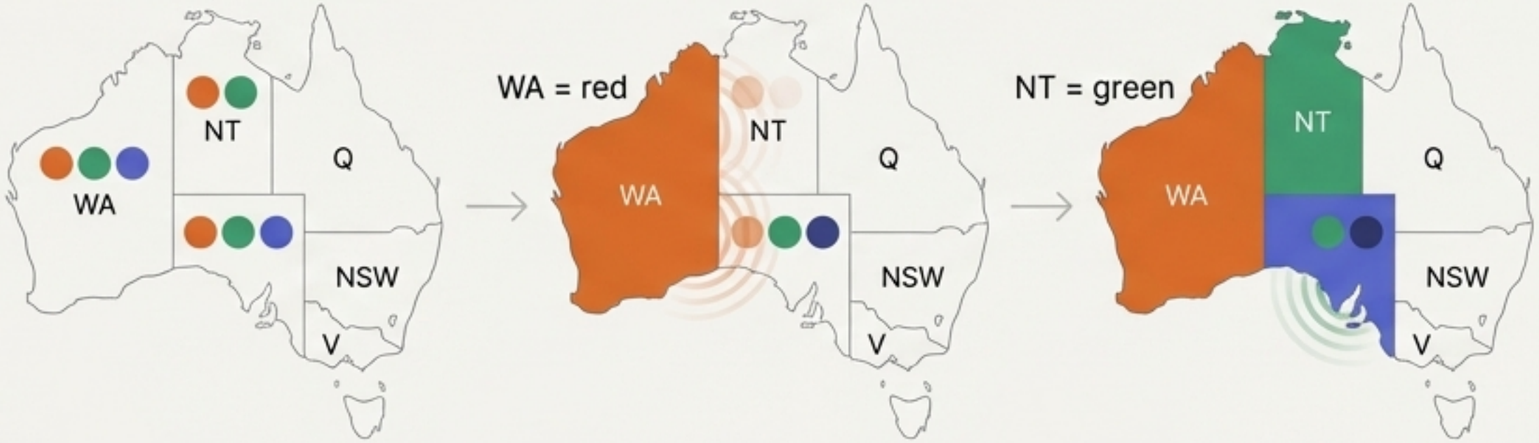
This factored representation allows us to prune huge swathes of the search space. For example, assigning SA = blue immediately tells us that five other variables cannot be blue, reducing the search space for those variables by 87%.

The Ripple Effect: How Constraint Propagation Solves Problems

Before we search, we can infer. Constraint Propagation uses the constraints to reduce the number of legal values for a variable. This reduction can then trigger further reductions for other variables, cascading through the problem.

Key Technique: Arc Consistency

- **Definition:** A variable X_i is arc-consistent with X_j if for every value in X_i's domain, there is a compatible value in X_j's domain.
- **In Plain English:** Can this value I'm considering for X_i possibly work with any of the available values for its neighbour X_j? If not, get rid of it.



Initial domains: {red, green, blue}

Propagating the constraint removes 'red' from neighbours' domains.

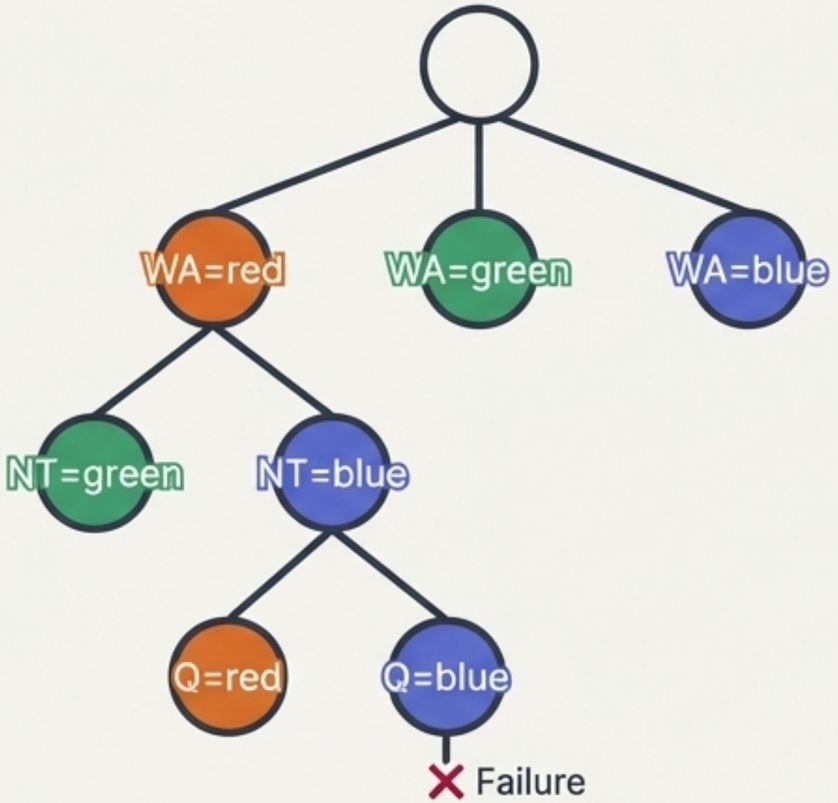
SA's value is now determined by inference, no search required.

The Engine: The AC-3 algorithm is a popular method for enforcing arc consistency across the entire problem, potentially solving it completely before search even begins.

The Art of Intelligent Search: Backtracking with Heuristics

The Need for Search

When constraint propagation stalls, we must search. Backtracking Search is a depth-first search that assigns values to one variable at a time, backing up when a variable has no legal values left.



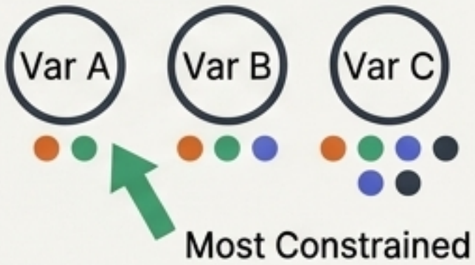
Making Search Smart

The naive order of variables and values is inefficient. We can dramatically improve performance with two key heuristics:

1. Variable Ordering: Minimum Remaining Values (MRV)

The Heuristic: Choose the variable with the fewest legal values remaining.

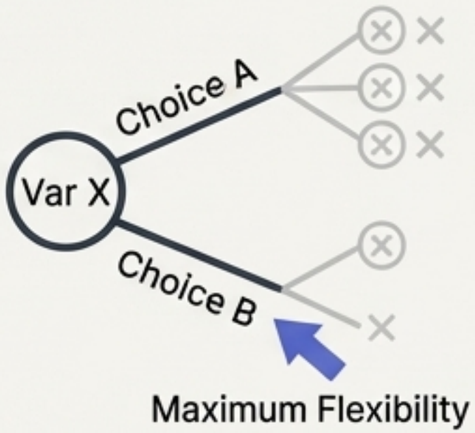
The Motto: Fail-First. By picking the most constrained variable, we are likely to prune the search tree sooner if we've made a bad choice.



2. Value Ordering: Least Constraining Value (LCV)

The Heuristic: Prefer the value that rules out the fewest choices for neighbouring variables.

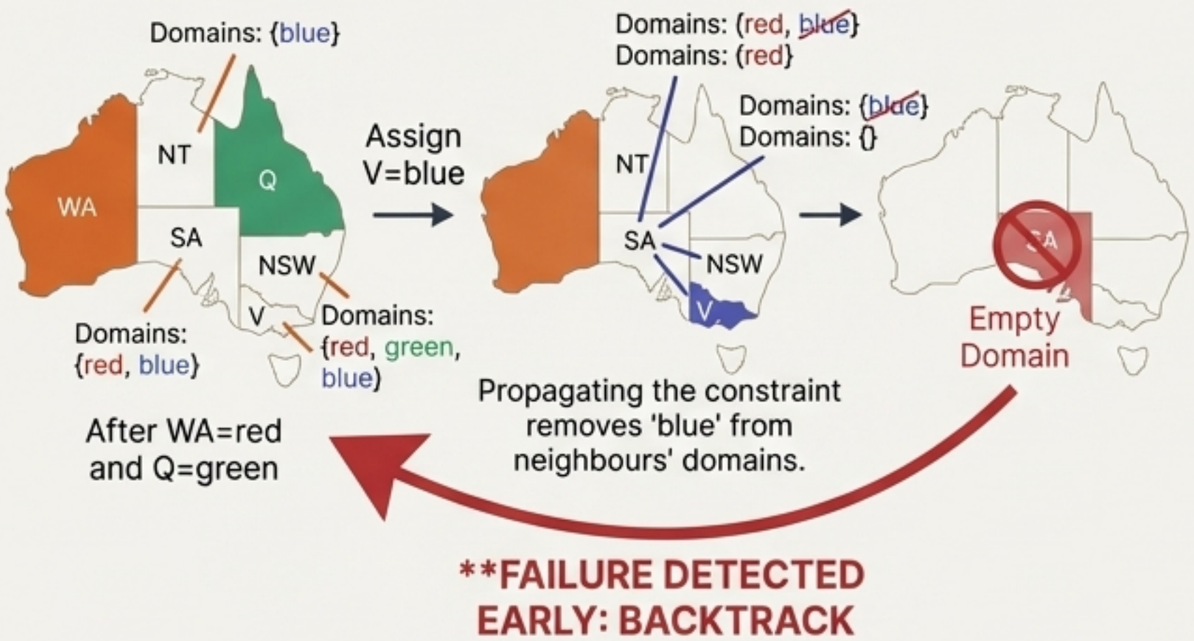
The Motto: Fail-Last. We only need one solution, so try the value most likely to be part of one, leaving maximum flexibility for the future.



Combining Powers: Inference-Driven Search and Intelligent Retreats

Interleaving Inference with Search

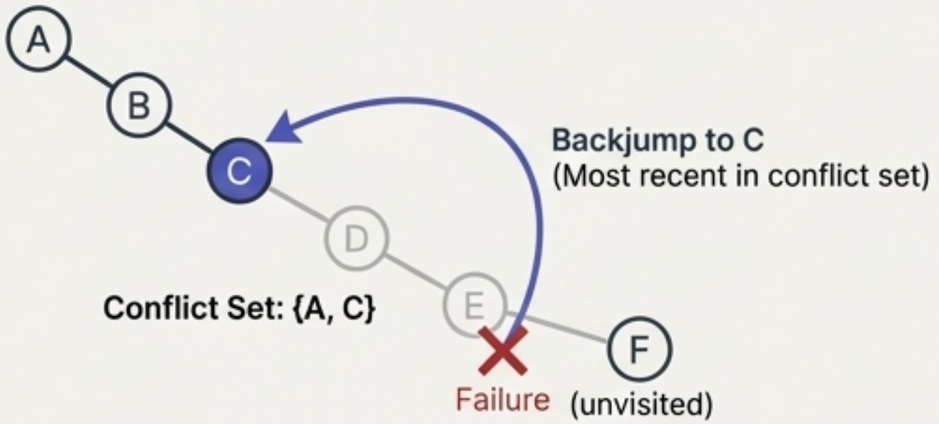
Forward Checking: A simple yet powerful technique. Whenever we assign a value to a variable X, we immediately check all its unassigned neighbours and delete any values from their domains that are now inconsistent.



Intelligent Backtracking

The Problem with Chronological Backtracking: Simple backtracking goes back to the most recent decision. But what if that decision had nothing to do with the failure?

Conflict-Directed Backjumping: A smarter approach. When a failure occurs, identify the set of prior assignments that caused it (the **conflict set**). Then, jump directly back to the most recent assignment in that set, ignoring any irrelevant decisions made in between.



Example: If colouring South Australia fails because of choices made for WA, NT, and Q, there's no point in trying a different colour for Tasmania. Backjumping goes straight to the source of the conflict.

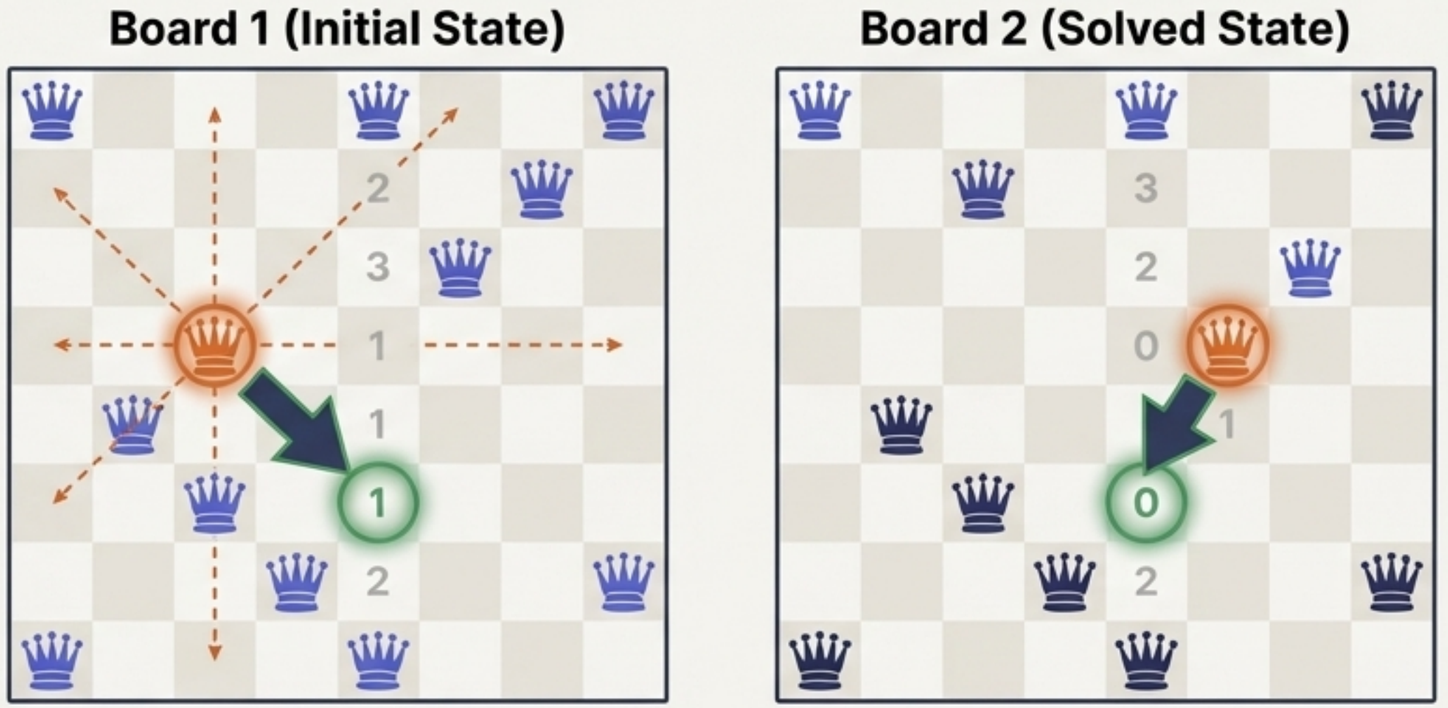
Repairing the World: Local Search and the Min-Conflicts Heuristic

A Different Philosophy

Instead of building a partial solution, local search starts with a complete, randomly assigned state and iteratively improves it. The state may initially violate many constraints.

The Min-Conflicts Heuristic

1. Start with a complete assignment (e.g., one queen per column).
2. Randomly select a **conflicted variable** (a variable involved in a violated constraint).
3. Reassign its value to the one that results in the **minimum number of conflicts** with other variables.
4. Repeat until no conflicts remain.



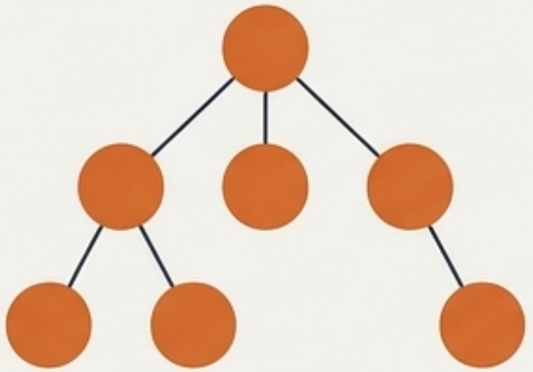
Astonishing Performance

The Min-Conflicts heuristic is incredibly effective. It can solve the *million-queens problem* in an average of 50 50 steps. It's also used for real-world problems like scheduling for the Hubble Space Telescope.

The Blueprint for a Solution: Exploiting Problem Structure

The complexity of solving a CSP is strongly related to the structure of its constraint graph. If we can simplify that structure, we can solve the problem exponentially faster.

Case 1: Tree-Structured Problems



Any CSP whose constraint graph has no loops (i.e., is a tree) can be solved in time linear in the number of variables. There is no need for backtracking.

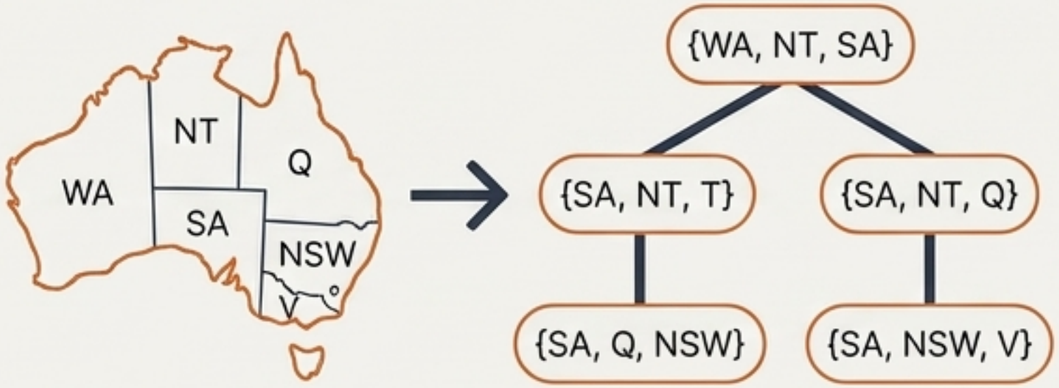
Reducing Complex Graphs to Trees

Method A: Cutset Conditioning

We assign every possible value to the cutset variables and solve the remaining tree problem for each assignment.

Method B: Tree Decomposition

The complexity depends on the size of the largest subproblem (the "tree width"). Problems with low tree width can be solved efficiently.



From Brute Force to Structural Insight

The progression from adversarial search to constraint satisfaction reveals a fundamental lesson in Artificial Intelligence. The most significant gains come not just from faster computation, but from deeper understanding. By moving from treating problems as monolithic challenges to understanding their internal structure, we unlock powerful, efficient, and elegant solutions.