1 Constraint Satisfaction Problems (CSPs)

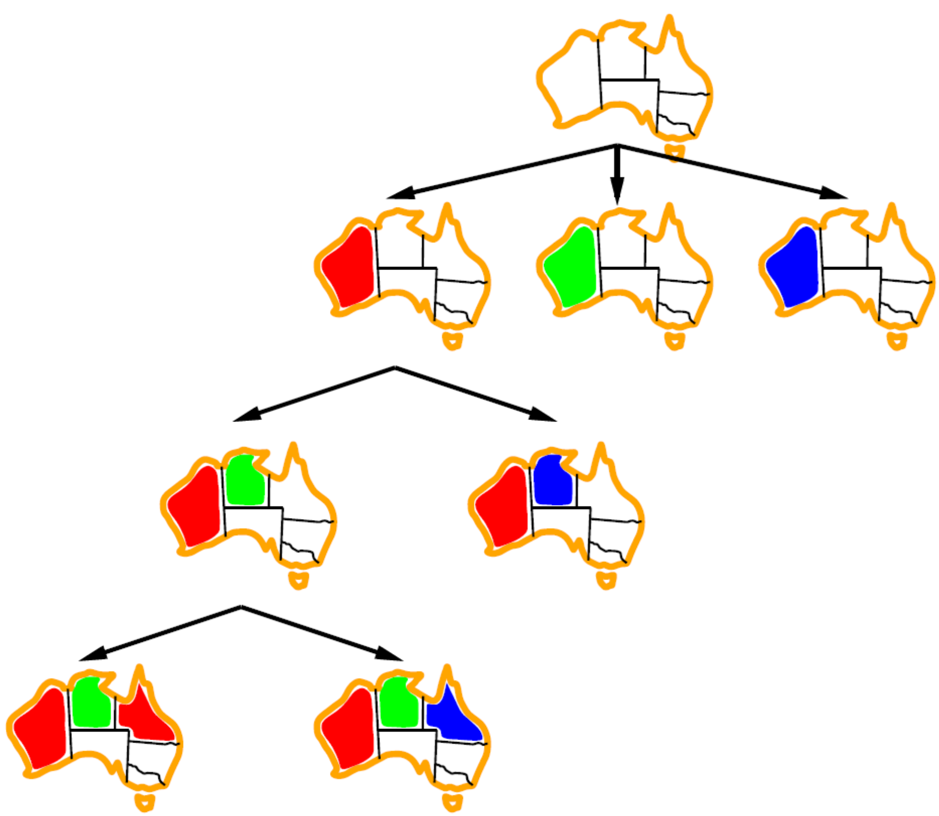
Standard search algorithms (like A* or BFS) treat states as atomic black boxes—they search for a *sequence* of actions (a path) to a goal. In **Constraint Satisfaction Problems (CSPs)**, the path is irrelevant. We treat states as **factored representations** (sets of variables and values) and simply search for a **goal state** that satisfies all requirements.

Definition: CSP

A CSP is defined by a triplet (X, D, C):
- **Variables (X):** A finite set of variables $\{X_1, \ldots, X_n\}$.
- **Domains (D):** A set of domains $\{D_1, \ldots, D_n\}$, where each variable X_i must take a value from the discrete set D_i.
- **Constraints (C):** A set of constraints specifying allowable combinations of values for subsets of variables.

1.1 Types of Assignments

- **Partial Assignment:** A state where values are assigned to only a subset of variables.
- **Consistent (Legal) Assignment:** An assignment that does not violate any constraints among the assigned variables.
- **Complete Assignment:** Every variable in X is assigned a value.
- **Solution:** A **complete** and **consistent** assignment.



1.2 Constraint Graphs

To visualize the structure of a CSP, we use a **Constraint Graph**. This abstraction is crucial because the topology of the graph (e.g., whether it contains loops or is a tree) dictates the complexity of solving it.

- **Nodes:** Represent the variables X_i.
- **Edges:** Connect any two variables that participate in the same constraint.

1.3 Types of Constraints

1. **Unary Constraint:** Restricts the value of a single variable (e.g., $SA \neq$ green). These can often be processed simply by filtering the domain D_i before search begins.
2. **Binary Constraint:** Relates two variables (e.g., $SA \neq WA$). These form the edges of the Constraint Graph.
3. **Higher-order Constraint:** Involves 3 or more variables (e.g., Cryptarithmetic puzzles like $TWO + TWO = FOUR$, where columns depend on carry bits).
4. **Soft Constraints (Preferences):** Constraints that are not mandatory but preferred (e.g., "Red is better than Green"). This shifts the problem from standard CSP to **Constrained Optimization**, often solved via cost functions.

1.4 Solving CSPs: Search Strategies

We can model a CSP as a standard search problem:

- **Initial State:** Empty assignment $\{\}$.
- **Successor Function:** Assign a value to an unassigned variable.
- **Goal Test:** Current assignment is complete and consistent.

1.4.1 Commutativity and Search Space

A naïve Breadth-First or Depth-First search would branch on every variable and every value in every order, creating a factorial search space ($n! \cdot d^n$).

However, CSPs are **commutative**: The order in which we assign variables does not change the final state (assigning $WA =$ red then $NT =$ green leads to the same state as $NT =$ green then $WA =$ red).

- **Implication:** We fix the order of variables or choose only *one* variable to branch on at each depth.
- **Benefit:** The search tree depth is fixed at n (number of variables). The search space reduces to d^n.

1.4.2 Backtracking Search

Backtracking Search is the fundamental algorithm for solving CSPs. It is essentially a Depth-First Search (DFS) that operates on partial assignments. Unlike standard search where actions lead to new states, here an action is the assignment of a value to a variable.

Backtracking Algorithm Logic

The algorithm relies on the **commutativity** of assignments (the order in which we assign variables doesn't matter for the final solution). This allows us to consider only a single variable at each node of the search tree, drastically reducing the branching factor from $n! \cdot d^n$ to d^n.

Algorithm Steps:

1. **Base Case:** If the assignment is complete (all variables have values), return the assignment as the solution.
2. **Variable Selection:** Select an unassigned variable using a specific strategy (see Heuristics).
3. **Value Iteration:** Iterate through the values in the domain of the selected variable.

4. **Consistency Check:** For each value, check if assigning it violates any constraints with currently assigned variables.
5. **Recursive Step:**
   - If consistent: Add $\{var = value\}$ to the assignment.
   - Call RECURSIVE-BACKTRACKING again.
   - If the recursive call returns a success, return that result.
   - **Backtrack:** If the recursive call fails, remove $\{var = value\}$ from the assignment and try the next value in the domain.
6. **Failure:** If all values have been tried and none work, return failure.

1.5 Heuristics: Improving Backtracking

Pure backtracking is slow. We use heuristics to decide *which* variable to pick and *which* value to try, effectively pruning the tree.

1. Which variable to assign next? (*Fail-First*)
2. In what order to try values? (*Fail-Last*)
3. Can we detect inevitable failure early? (*Inference*)

1.5.1 Variable Selection (Which variable next?)

1. **Minimum Remaining Values (MRV):**
   - **Strategy:** Choose the variable with the *fewest* legal values remaining.
   - **Intuition ("Fail-First"):** If a variable has only 1 legal value left, we must assign it now. If we wait, it might become 0, causing a failure deeper in the tree. We want to force failures as high up in the tree as possible to prune large branches.
2. **Degree Heuristic:**
   - **Strategy:** Choose the variable involved in the largest number of constraints with *other unassigned* variables.
   - **Usage:** Often used as a tie-breaker for MRV.
   - **Intuition:** Assigning the most connected variable exerts the maximum "pressure" on the rest of the graph, reducing the branching factor for future steps.

1.5.2 Value Selection (Which value first?)

1. **Least Constraining Value (LCV):**
   - **Strategy:** Given a variable, try the value that rules out the *fewest* values in the domains of neighboring variables.
   - **Intuition ("Fail-Last"):** We want to find *a* solution, not all solutions. Therefore, we should pick the path most likely to succeed by leaving the maximum flexibility for the remaining variables.

1.6 Constraint Propagation (Inference)

Search implies "trying" values and undoing them if they fail. **Propagation** implies logically deducing which values are impossible and removing them *before* we try them.

1.6.1 Levels of Consistency

- **Node Consistency:** Every variable satisfies its unary constraints.
- **Arc Consistency:** A variable X is arc-consistent with respect to Y if, for every value $x \in D_X$, there is some value $y \in D_Y$ satisfying the binary constraint (X, Y).
- **Path Consistency:** Ensures consistency for triples of variables.

Constraint propagation is the process of using constraints to reduce the legal domain of a variable, which in turn reduces the domains of its neighbors, and so on. This happens *before* or *during* search to reduce the search space.

1.6.2 Algorithms for Propagation

Forward Checking Forward checking is a simple form of propagation performed during backtracking search.

Algorithm Steps:

1. When variable X is assigned value v:
2. Look at all unassigned variables Y that are connected to X by a constraint.
3. Remove any value from D_Y that conflicts with $X = v$.
4. **Early Termination:** If any domain D_Y becomes empty, stop this branch immediately (backtrack).

Limitation: It only checks direct neighbors. It does not detect if the reduction in Y's domain makes Y incompatible with a third variable Z.

Arc Consistency (AC-3 Algorithm) AC-3 propagates constraints globally. It ensures that every arc in the graph is consistent.

AC-3 Algorithm

AC-3 is a more powerful general-purpose algorithm that propagates constraints globally until the network is **Arc Consistent**. A variable X is arc-consistent with respect to Y if for every value $x \in D_X$, there is some allowed value $y \in D_Y$.

1. **Queue Initialization:** Create a queue containing all arcs (binary constraints) in the CSP: $\{(X_i, X_j), (X_j, X_i), \dots\}$.
2. **While Queue is not empty:**
   - Pop an arc (X_i, X_j) from the queue.
   - Call REMOVE-INCONSISTENT-VALUES(X_i, X_j).
   - **If values were removed** from D_i:
     - The domain of X_i has become smaller. This might ruin consistency for its neighbors.
     - Add all arcs (X_k, X_i) (where X_k is a neighbor of X_i) back into the queue.

Function Remove-Inconsistent-Values(X_i, X_j):

- Iterate through every value x in D_i.
- Check if there exists a value y in D_j that satisfies the constraint between X_i and X_j.
- If no such y exists, delete x from D_i. Return *true* (indicating change occurred).

1.7 Local Search for CSPs

Local Search algorithms (like Hill Climbing) operate on **complete states**, meaning all variables are assigned a value at all times, even if the assignment is inconsistent (constraints are violated). The goal is to iteratively repair the assignment.

Algorithm Steps:

1. **Initialization:** Start with a complete assignment for all variables (usually generated randomly).

2. **Loop** (until a solution is found or max steps reached):
   - Check if the current assignment satisfies all constraints. If yes, return it.
   - **Variable Selection:** Randomly select a variable that is currently involved in a conflict (violating a constraint).
   - **Value Selection (Minimization):** Choose a new value for this variable that minimizes the number of conflicts with other variables.
   - Update the variable to this new value.

Note: Like other local search methods, this can get stuck in local optima (plateaus), specifically when the ratio of constraints to variables is critical.

1.8 Problem Structure and Decomposition

1.8.1 Independent Subproblems

If the constraint graph consists of connected components that are disjoint, we can solve each component independently.

- **Impact:** Reduces complexity from exponential in total variables $O(d^n)$ to exponential in the size of the largest component $O(d^c)$.

1.8.2 Tree-Structured CSPs

If the constraint graph forms a tree (no loops), we can solve the CSP in linear time $O(n \cdot d^2)$ instead of exponential time.

Algorithm Steps:

1. **Topological Sort:** Linearlize the variables (order them $X_1, \ldots, X_n$) such that every variable appears after its parent.
2. **Backward Pass (Consistency):**
   - Iterate from $j = n$ down to 2.
   - Apply arc consistency to the arc $(Parent(X_j), X_j)$.
   - This ensures that for every value in the parent's domain, there is a valid value in the child's domain.
3. **Forward Pass (Assignment):**
   - Iterate from $i = 1$ to n.
   - Assign any value to X_i that is consistent with the assignment of its parent.
   - Because of the backward pass, a valid assignment is guaranteed to exist. NO backtracking is required.

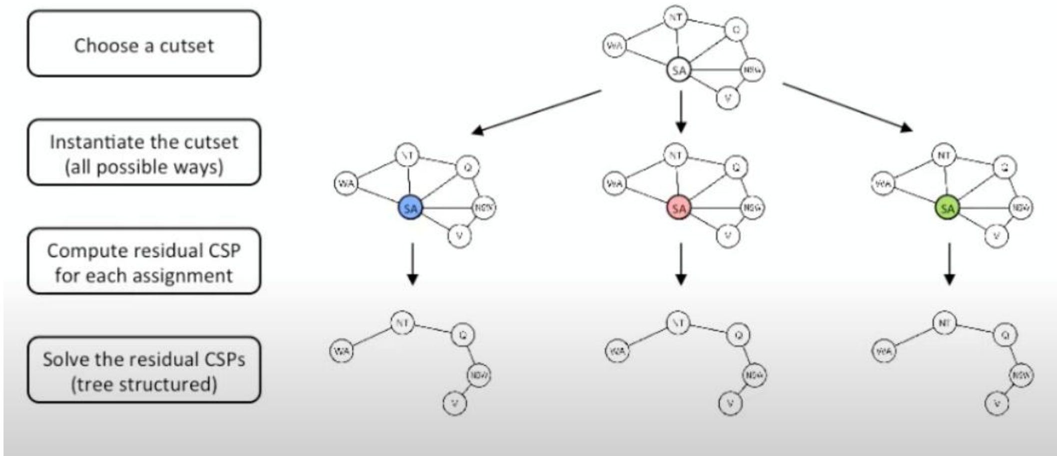
1.8.3 Nearly Tree-Structured Problems (Cutset Conditioning)

Most real-world problems are not trees, but often "close" to trees. We can exploit this via **Cutset Conditioning**. This technique is used for constraint graphs that are *nearly* trees. It turns a cyclic graph into a tree by removing specific nodes.

- **Cycle Cutset:** A subset of variables S such that removing them renders the remaining graph a tree.

Algorithm:

1. Identify the Cutset S.
2. Iterate through all possible consistent assignments for variables in S.
3. For each assignment of S, the values are fixed. This simplifies the constraints on the remaining variables.
4. Solve the remaining variables (which now form a tree) using the efficient Tree CSP algorithm.
5. **Complexity:** $O(d^{|S|} \cdot (n - |S|)d^2)$. Efficient if the cutset $|S|$ is small.


Choose a cutset
Instantiate the cutset (all possible ways)
Compute residual CSP for each assignment
Solve the residual CSPs (tree structured)