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.

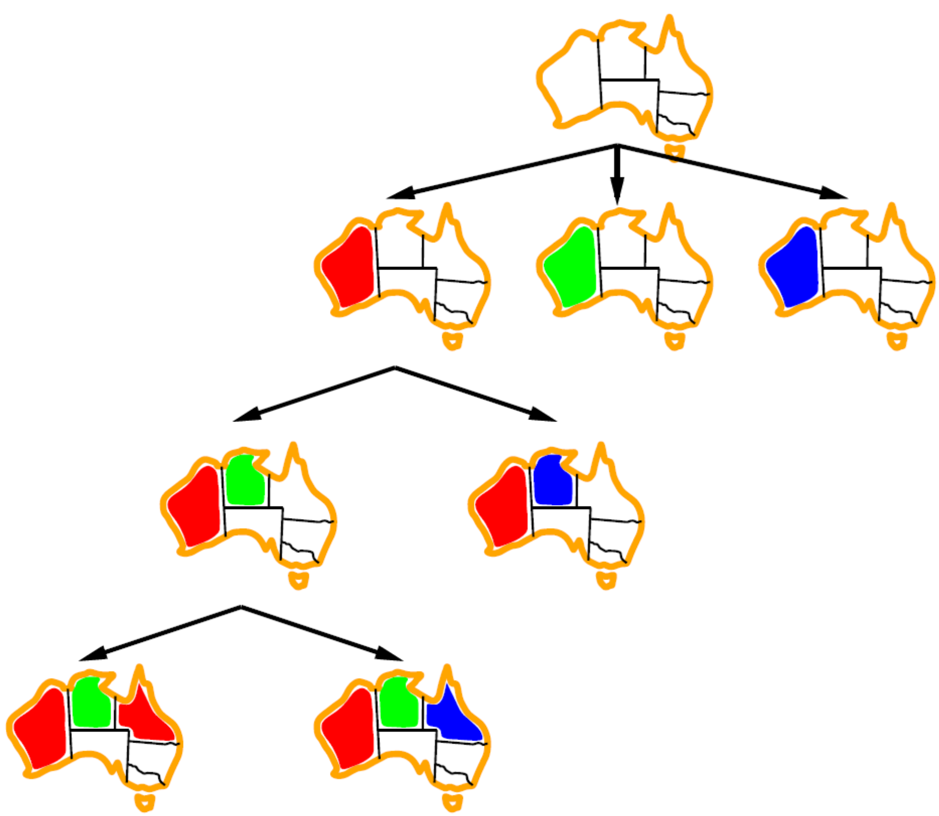
Definition: CSP

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 is the fundamental uninformed algorithm for CSPs. It is a Depth-First Search (DFS) that checks constraints *incrementally*.

Backtracking Algorithm Logic

Function BACKTRACK(assignment, csp):

1. **Base Case:** If assignment is complete, return assignment (Success).
2. **Variable Selection:** Select an unassigned variable var.
3. **Value Iteration:** For each value val in ORDER-DOMAIN-VALUES(var):
   - If val is consistent with current assignment:
     (a) Add $\{var = val\}$ to assignment.
     (b) **Inference (Optional):** Run Forward Checking or AC-3. If inference fails, skip step 3.
     (c) result $\leftarrow$ BACKTRACK(assignment, csp).
     (d) If result $\neq$ failure, return result.
     (e) Remove $\{var = val\}$ from assignment (**backtrack**).
4. Return failure.

1.5 Heuristics: Improving Backtracking

Standard backtracking is slow. To speed it up, we need "intelligence" at three key decision points:

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.

1.6.2 Algorithms for Propagation

Forward Checking Whenever a variable X is assigned a value v:

1. Identify all unassigned neighbors Y connected to X.
2. Delete any value from D_Y that conflicts with $X = v$.
3. **Fail:** If any D_Y becomes empty, backtrack immediately.

Limitation: Forward checking is short-sighted. It checks $X \to Y$, but does not check if the changes in Y cause issues for a third variable Z (e.g., Y and Z might be forced to take the same value).

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

AC-3 Algorithm

1. **Queue Initialization:** Add all arcs (X_i, X_j) in the CSP to a queue.
2. **Process Queue:** While the queue is not empty:
   - Pop an arc (X_i, X_j).
   - **Revise**(X_i, X_j)**:** Check if every value in D_i has a valid support in D_j. Remove values in D_i that do not.
   - **Cascade:** If D_i was modified (values removed):
     - If D_i is empty, return Failure (no solution possible).
     - Otherwise, add all neighbors X_k of X_i (arcs (X_k, X_i)) back to the queue.
     - *Reasoning:* Removing a value from D_i might effectively remove the "support" it provided for a value in D_k, so we must re-check X_k.

1.7 Local Search for CSPs

Constructive search (Backtracking) starts with empty states. **Local Search** starts with a *complete* but *inconsistent* assignment and tries to fix the conflicts iteratively.

- **State:** Complete assignment of all variables (some constraints violated).
- **Goal:** Minimize the total number of conflicts (violated constraints).
- **Min-Conflicts Heuristic:**
  1. Pick a variable var that is currently involved in a conflict (randomly).
  2. Compute the number of conflicts for every possible value of var.
  3. Reassign var to the value that minimizes conflicts.

This method is incredibly effective for problems like the N-Queens, capable of solving $N = 1,000,000$ in near-constant time, though it is not guaranteed to find a solution (can get stuck in local minima).

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 is a **Tree** (no loops), the CSP can be solved in linear time $O(n \cdot d^2)$ rather than exponential time.

Algorithm for Tree CSPs:

1. **Topological Sort:** Linearize the variables such that every node appears after its parent ($X_1 \to X_2 \to \cdots \to X_n$).
2. **Backward Pass (Cleanup):** Iterate from X_n down to X_2. For each node, make its parent arc-consistent with it. This removes any values from parents that have no valid child.
3. **Forward Pass (Assignment):** Iterate from X_1 to X_n. Assign any value consistent with the parent's assignment. Because of the backward pass, we are guaranteed that a valid value exists. Backtracking is never needed.

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**.

- **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.