WEEK-5 NOTE

CSP constraint satifaction problems

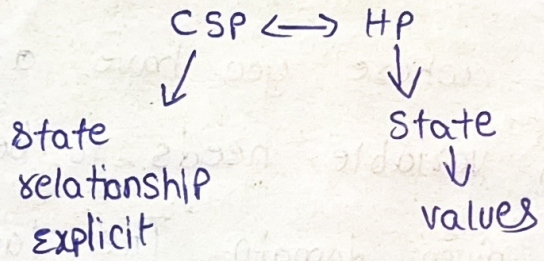
It differs from heuristic search in the following ways

HS

In heuristic search A state was collections of values in that particular state

CSP :-

In CSP A state will have relationships with other states made explicit



Outline

CSP Example

Here, Standard search problem which means our heuristics Searchs

The state is a black box Here, any old data structure can be used

All that we need to do is support the gold test, that is test, Evaluation, Successor & find what is next comming to be done

1) SSP

The State is treated as black box that means it can represent any data structure

2) Process :- uses three keys those are

Gold Test :- check if a state meet desired objective

Eval :- Evaluate the state to guide the search

Success()

Generates possible next states

* CSP

CSP is a puzzle where you have a set of variables & each variable needs to be assigned a value from a given domain. The catch is that there are constraints that limit which combination of values are allowed for certain subsets of (valu>) variables

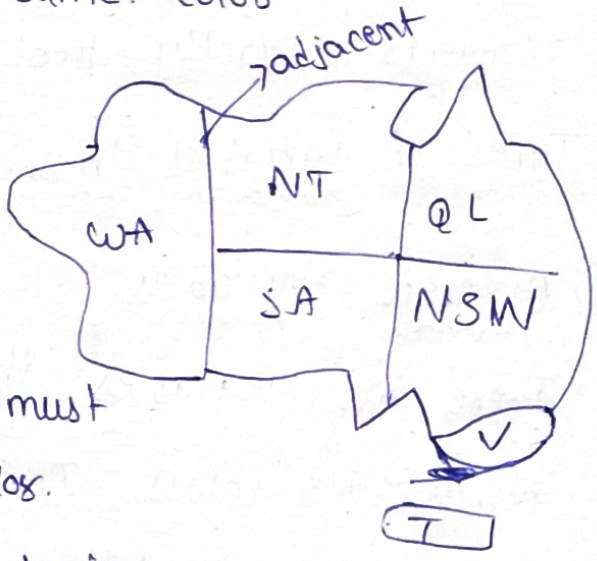
In contrast a CSP has more structured ~~str~~ representation.

State :- Defined by variable (X) with values from their respective Domains (D_i)

Goal Test : A set of constraints that specify the allowable combination of values for subsets of variables.

It simple Example of formal representation language

Ex : Imagine a simple map coloring problem you have 7 regions (variables) & Each region needs to be colored with one of three colors (red, blue & green). The constraint is that no two adjacent regions can have the same color.



variables : WA, NT, QL, SA NSW, V, T

Domains : Red, green, blue.

Constraints : adjacend-gregions must have different color.

Here, adjacents means that besides the country to next to another country.

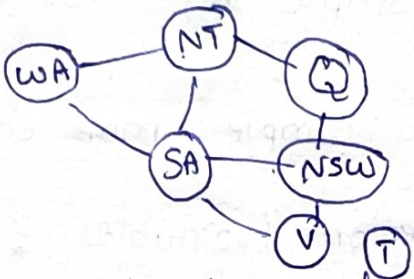
* Constraints graph

A C.G is visual representation of a csp. It helps in understanding the relationship blw variable & their Constraints

Key Component

Nodes : Each nodes represent a variable in the csp.

Archs : An arcs (d) edge blw two nodes indicates there is a constraint blw the corresponding variables



Digit

Binary CSP

A binary CSP is a special types of CSP where each constraints involves only two variables In the constraints of binary csp each arcs connects exactly two nodes.

Types of variables in CSP

1) Discrete variables

These are variables that have a finet set of possible values. The seze of the domain is denoted by 'd'.

Ex: Boolean CSP (variable can be either true or false) & other problems where variables have a limited no of choices (eg: assiging colors.

complete assignment

you must assign values to all variables while satisfying all constraints. The number of possible assign grows, $O(d^n)$

where d is domain size & n is no of variables

Infinite Domains

These variables can take any values from an. infinte set (like integers or strings)

* continuous variables

Ex:- problems where variables represents continous values such as start/end times for HT obeserva tions.

Constraint lang :- constraint are typically expressed using linear eqn or inequalities

Solvability :- linear constraints involing continous variables can be solved in polynomial time using L.P methods. This making solving CSPs, with continous variables generally more efficients that those. discrete variables & complexes constraints

varities of constraints

1) onary :- These involves a single variables

Ex:- SA ≠ green means that variables SA can't take green value

2) Binary :- These involves paris of variables

Ex:- SA = WA means the variables SA & WA must

have the same values

3) Higher order

These involves three or more variables

Ex: cryptarithmetic.

4) preferences or soft constraints

These are not strict req but rather priorities or preferences

Eg: red is better than green

often representable by a cost of each variable assignment.

Ex: cryptarithmetic problem

Use youtube channel

Easy Engineering class

L47, 46 class to understand

Constraint Satisfaction Problems

Chapter 5

Outline

- CSP examples
- Backtracking search for CSPs
- Problem structure and problem decomposition
- Local search for CSPs

Real-world CSPs

Assignment problems

Who teaches what class? - We need to assign teachers to classes, making sure each class has a teacher and that teachers don't have classes at the same time.

* When and where is each class? - We need to schedule classes in rooms, making sure there are no clashes and that rooms are available.
Hardware configuration

* Building a computer - We need to choose parts that work together (like a CPU and a motherboard) and fit within our budget.
* Spreadsheets - We use formulas to calculate things, and we need to make

sure the numbers in different cells are correct.
* Delivering goods - We need to plan routes for trucks, making sure they reach their destinations on time and don't go over their weight limits.
* Making things in a factory - We need to schedule when machines are used and when workers work, making sure everything is done on time and efficiently.
* Designing a building - We need to decide where each room goes, making sure they are the right size and have the right connections (like hallways)Notice that many real-world problems involve real-valued variables

Standard search formulation (incremental)

1) * States:
   * A state represents the current partial assignment of values to variables.
   * The initial state is the empty assignment, meaning no variables have been assigned values yet.

   Successor Function:
   * This function determines the next possible states.
   * It assigns a value to an unassigned variable.
   * Importantly, it checks for conflicts: if the new assignment violates any constraints, the state is considered "not fixable" and discarded.

   * Goal Test:
   * The goal is to reach a complete assignment, meaning all variables have been assigned values without any constraint violations.

2) The slide suggests using depth-first search to explore the possible states.

* Depth-first search goes deep into one branch of the search tree before exploring other branches.

4. Complexity

* The slide points out that the number of leaves in the search tree can grow very quickly as the number of variables (n) and the size of their domains (d) increase.

* This is because there are n!d^n possible leaves in the worst case.

Backtracking search

Variable assignments are commutative, i.e.,

[WA=red then NT =$green$] same as [NT =$green$ then WA=red]

Only need to consider assignments to a single variable at each node

⇒ b=d and there are d^n leaves

Depth-first search for CSPs with single-variable assignments is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

Can solve n-queens for $n \approx 25$

Explanation of backtracking search

• Variable assignments are commutative, meaning the order in which variables are assigned does not matter. For example, assigning WA = red and NT = green is the same as NT = green and WA = red.
• This reduces the number of combinations to consider, making the process more efficient

It is a **depth-first search** algorithm for CSPs where variables are assigned values one by one, ensuring constraints are satisfied.
• At each node, the algorithm only assigns a single variable and proceeds.
3. **Efficiency**:
• The algorithm focuses on only valid assignments and prunes the search space, solving complex problems like the n-queens problem for .

Backtracking search

```
function Backtracking-Search(csp) returns solution/failure
    return Recursive-Backtracking({},csp)

function Recursive-Backtracking(assignment,csp) returns
    soln/failure if assignment is complete then return
    assignment var←Select-Unassigned-
    Variable(Variables[csp],assignment,csp) for each value in
    Order-Domain-Values(var,assignment,csp) do if value is
    consistent with assignment given Constraints[csp] then ac
    {var = value} to assignment
            result←Recursive-
            Backtracking(assignment,csp) if result 6=
            failure then return result remove {var =
            value} from assignment
    return failure
```

1. **Function**:

• The main function Backtracking-Search initiates the search process by calling Recursive-Backtracking.Explanation of backtracking search

2. **Steps in Recursive-Backtracking**:

• **Base Case**: If the assignment is complete (all variables are assigned), return the assignment as a solution.
• **Select Variable**: Choose an unassigned variable from the CSP.
• **Iterate over Possible Values**: For each value in the variable's domain:
• Check if it is consistent with the current assignment.
• If consistent, temporarily add it to the assignment.
• **Recursive Call**: Recursively call the function to assign the next variable.
• **Backtrack**: If the recursive call fails, undo the current variable assignment and try the next value.

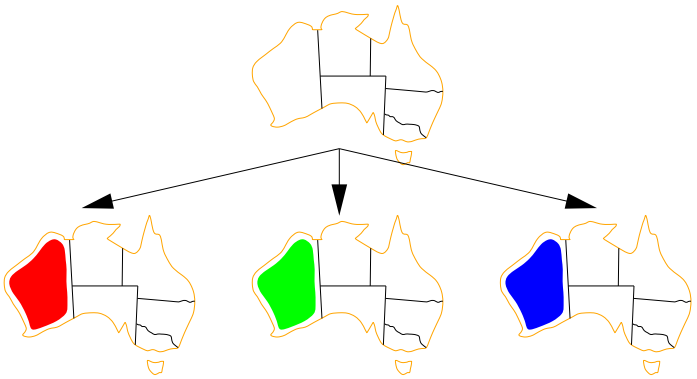
3. **Output**:
• The algorithm either finds a solution or returns "failure" if no solution exists.

How it works:

* Start at the root node: The algorithm begins at the top (root) of the tree.
* Explore branches: It moves down the tree, making choices at each level. In the image, each level represents a different decision or choice.
* Backtrack when necessary: If a choice leads to a dead end or a solution that doesn't satisfy the problem's constraints, the algorithm backtracks to the previous level and explores other choices. This is represented by the arrows going back up the tree.

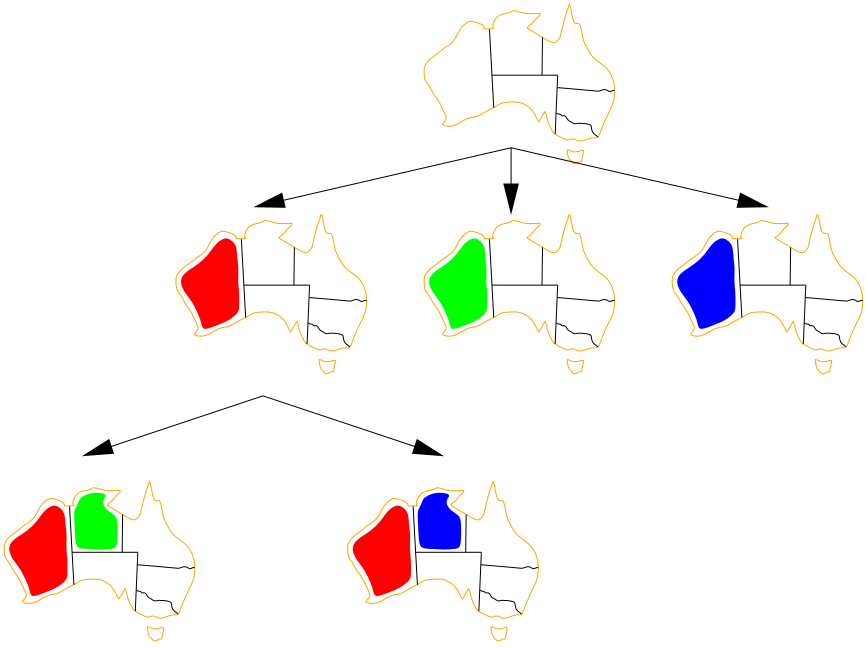
Why use backtracking:
* Systematic exploration: It ensures that all possible combinations are explored in a structured way.
* Efficiency: It can be more efficient than brute-force search, as it avoids exploring paths that are guaranteed to lead to invalid solutions.

Backtracking example



Step 1

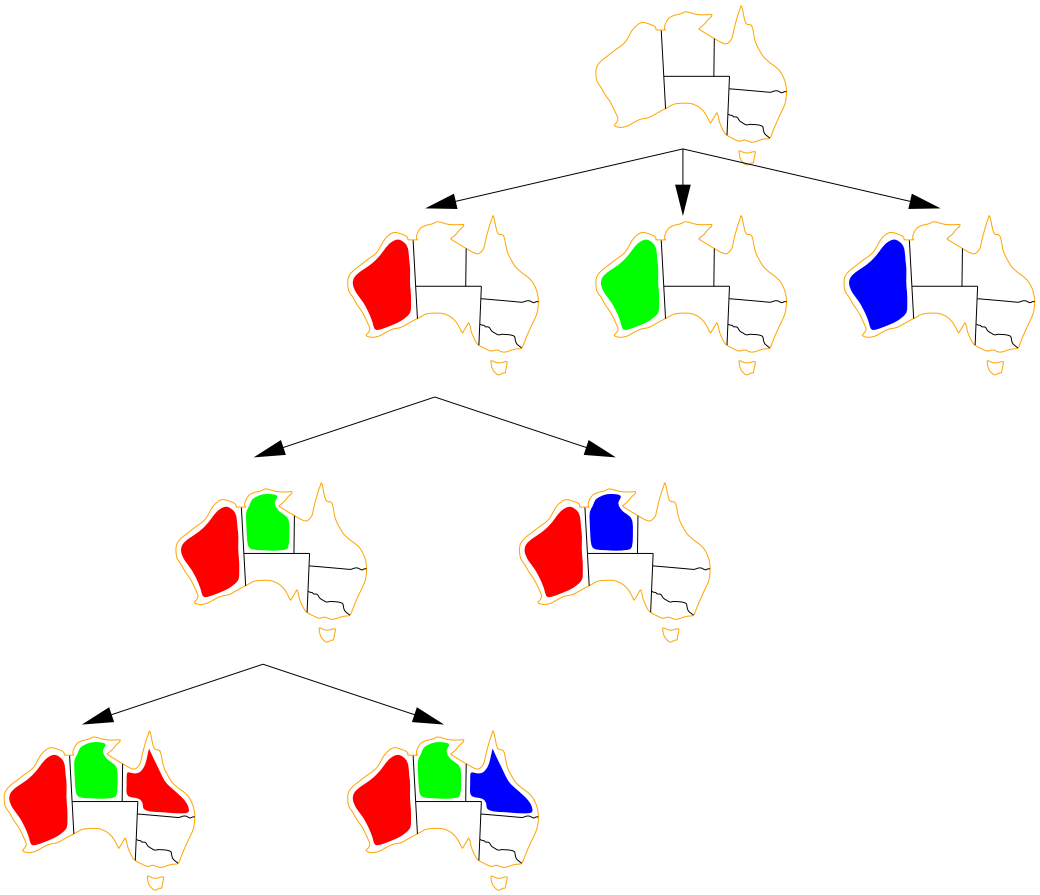
Start at the root node: The algorithm starts at the top node.



Step 2

Explore the second branch: If the red node

doesn't lead to a solution, the algorithm backtracks to the root node and explores the second child node (green).



Step 3

Explore the third branch: If the green node doesn't lead to a solution, the algorithm backtracks to the root node and explores the third child node (yellow).

* Continue exploring: The algorithm continues exploring the tree in this manner, making choices

and backtracking as needed.
* Reach a solution: When the algorithm reaches a node that represents a valid solution to the problem, it stops and returns the solution.
Note: This is a simplified explanation. The actual backtracking algorithm can be more complex depending on the specific problem being solved.

Minimum remaining values

Minimum remaining values (MRV):

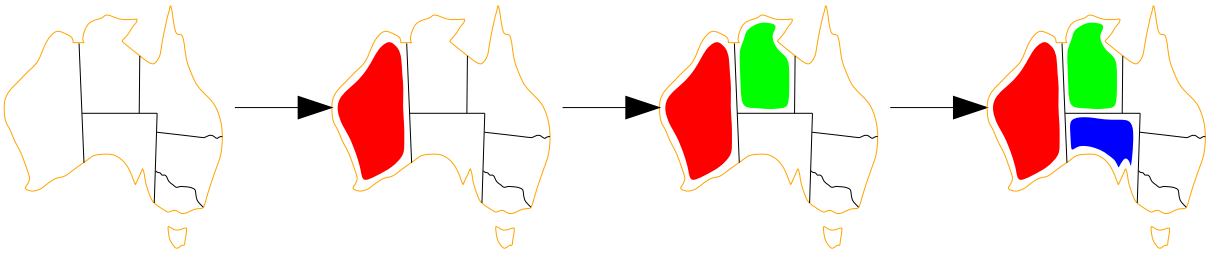
choose the variable with the fewest legal values

Explanation

It means:

- Look at all empty spots in a puzzle.
- Start with the spot that has the fewest possible pieces to fit.

This helps focus on the hardest parts first, reducing wasted time.

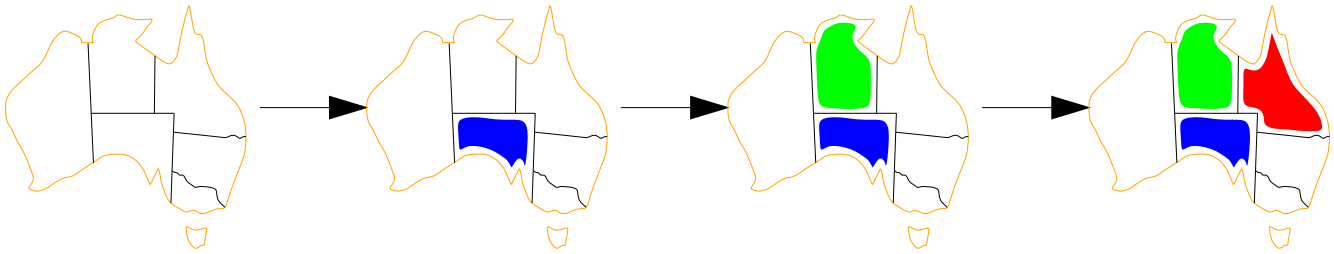


Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:

choose the variable with the most constraints on remaining variables



How it works:

• If multiple choices have the same possibilities (MRV), pick the one that affects the most other parts of the problem.

Example:

In a tower puzzle:

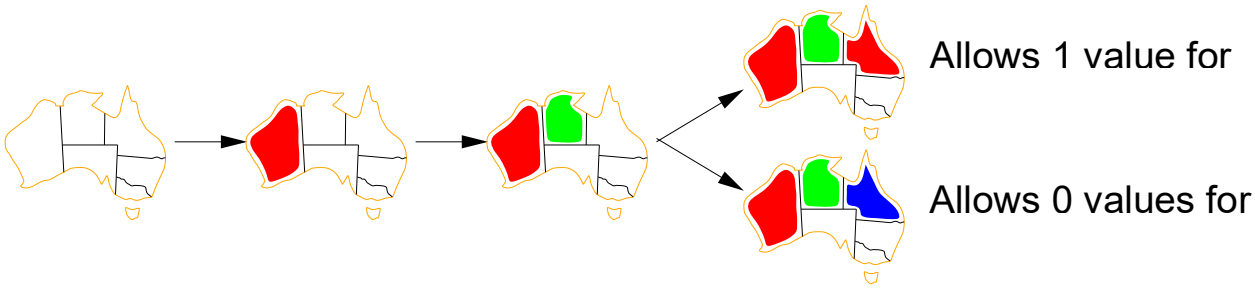
• MRV: Focus on levels with the fewest block choices.

• Degree Heuristic: If levels are tied, choose the one supporting the most other levels.

This helps solve problems faster by prioritizing high-impact choices.

Least constraining value

Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



How it works:

• After using MRV and Degree Heuristic, if there are still multiple choices, look at how each choice affects the remaining problem.

• Choose the option that leaves the most flexibility for future decisions.

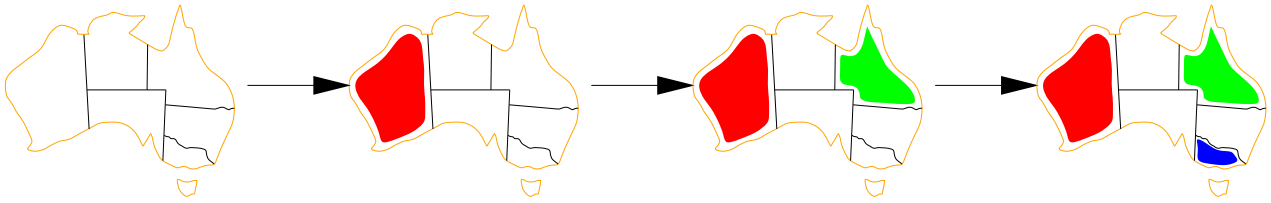
Example:

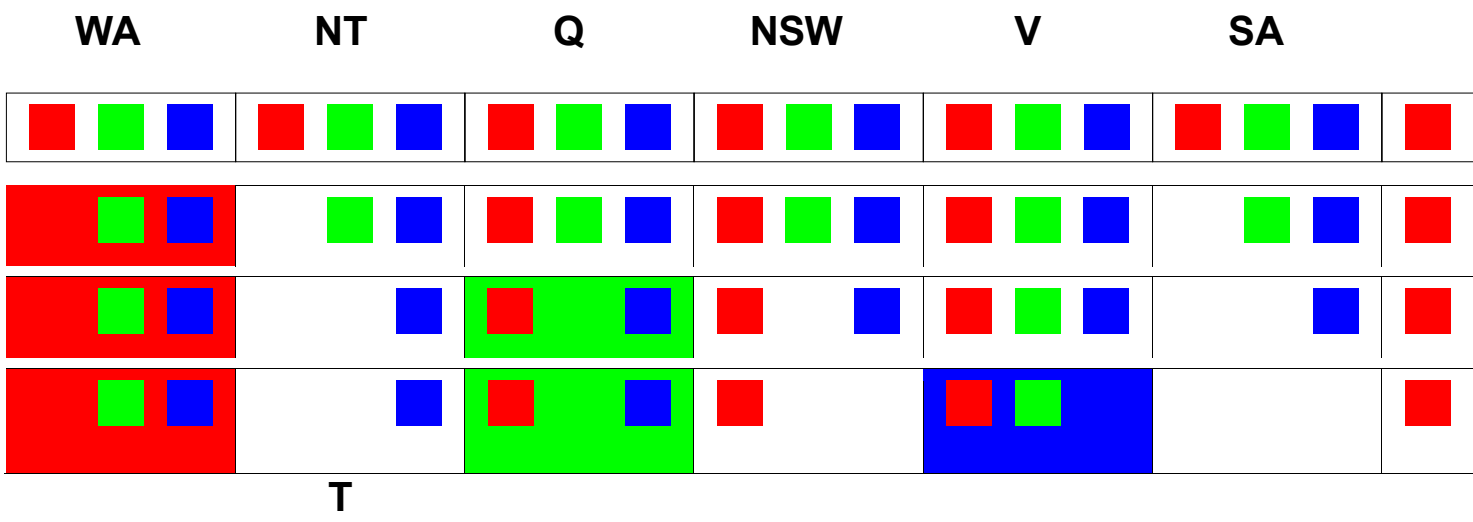
In a tower puzzle:

• Pick the block that limits the fewest choices for the levels above.

Forward checking

Terminate search when any variable has no legal values





What is Forward Checking?

Forward Checking is a method to avoid wasting time on choices that lead to dead ends by looking ahead at the impact of each decision.

How It Works (Step-by-Step):

1. Start the problem

- Begin with an empty assignment (nothing is

decided yet).
• Each variable has a list of possible values (its domain).
2. Pick a variable
• Choose one variable to assign a value to. You can use a strategy like picking the one with the fewest options first (MRV).
3. Assign a value
• Pick a value from the variable's domain and assign it to the variable.
4. Look ahead (Forward Check)
• For each unassigned variable connected to the one you just assigned:
• Remove values from their domains that would create a conflict with your choice.
5. Check for problems
• If any unassigned variable now has no valid values left (its domain is empty):
• Stop and backtrack.
• Try a different value for the previous variable.
6. Repeat until done
• If no conflicts are found, repeat steps 2–5 for the next variable until all are assigned.

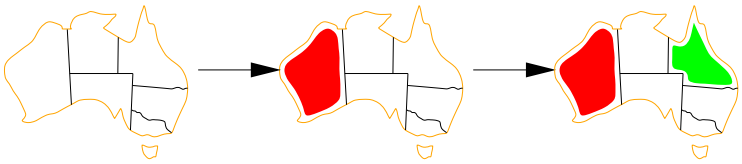
Example:

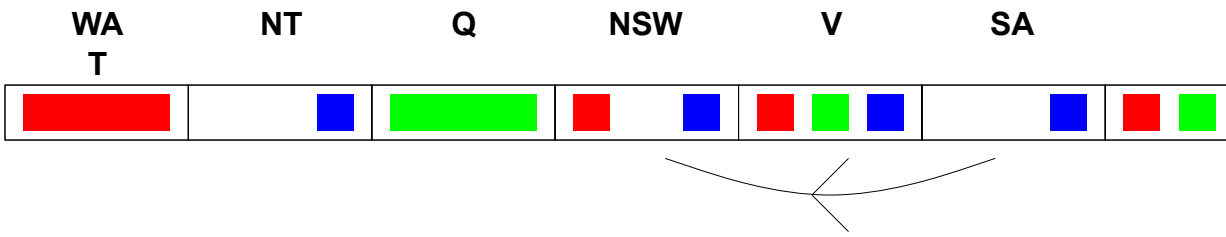
Let’s say you’re coloring a map, and regions (WA, NT, Q) must each get a different color: Red, Green, or Blue.

1. Start:
• WA, NT, and Q can all have [Red, Green, Blue].
2. Pick WA:
• Assign Red to WA.
3. Forward Check:
• NT and Q cannot also be Red. Update their domains:
• NT: [Green, Blue]
• Q: [Green, Blue]
4. Pick NT:
• Assign Green to NT.
5. Forward Check:
• Q cannot be Red or Green. Update its domain:
• Q: [Blue]
6. Pick Q:
• Assign Blue to Q.
7. Done:

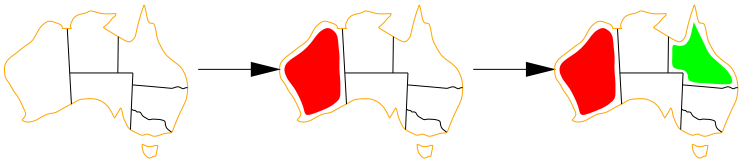
- All regions are assigned a color with no conflictshas context menu

$X \rightarrow Y$ is consistent iff for **every**
value x of X there is **some**
allowed y





$X \rightarrow Y$ is consistent iff for **every**
value x of X there is **some**
allowed y

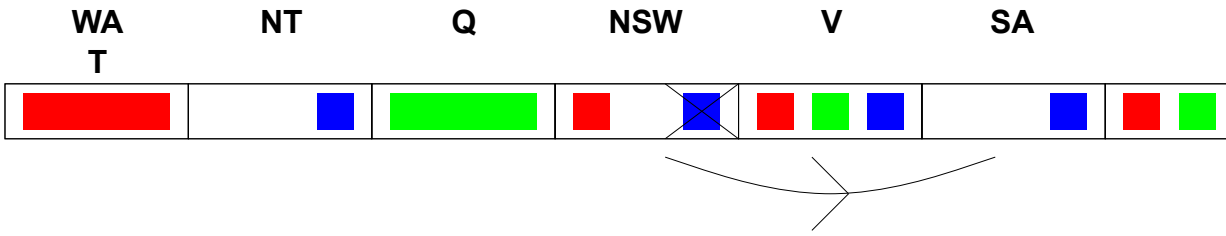


How It Works:

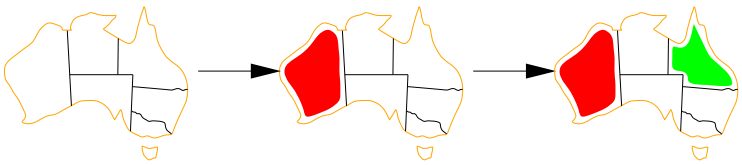
1. Look at two connected variables (X, Y).
2. For each value in X, make sure there's a matching value in Y.
3. If not, remove the unmatched value from X.
4. Repeat this for all pairs until no more changes are needed.

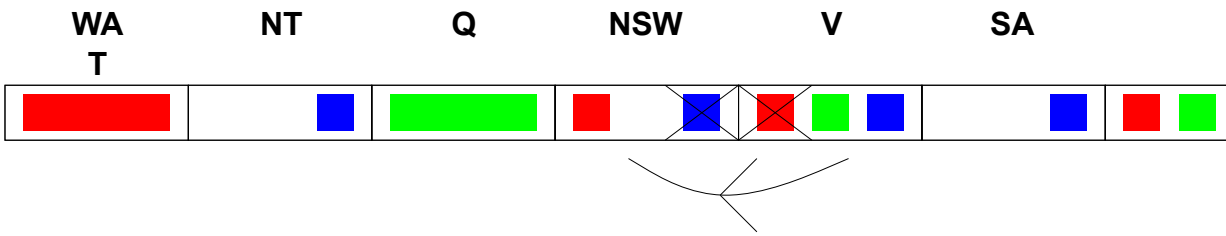
Example (Map Coloring):

- Regions can't have the same color.
- If WA = [Red, Blue] and NT = [Blue], remove Red from WA because NT can't match it



$X \rightarrow Y$ is consistent iff for **every** value x of X there is **some** allowed y





If X loses a value, neighbors of X need to be rechecked

Arc Consistency removes impossible options in a problem.

1. How it works:

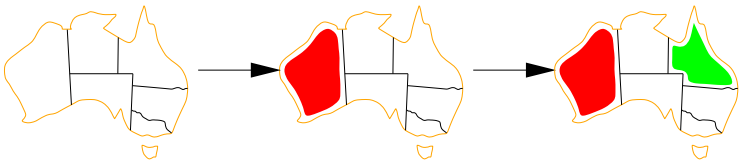
• For each pair of connected variables, make sure each value in one has a matching value in the other.

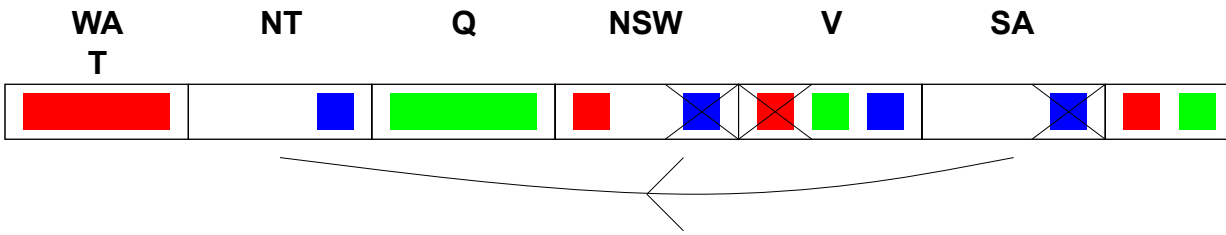
• If no match, remove that value.

2. Example:

• If two regions can't have the same color, remove conflicting colors.

$X \rightarrow Y$ is consistent iff for **every**
value x of X there is **some**
allowed y





If X loses a value, neighbors of X need to be rechecked

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

How It Works:

1. Arc: A constraint between two variables (X, Y).
2. Consistency Check: For each value in X, ensure there's a matching value in Y.
3. Arc Revision: If no match exists for a value in X, remove it from X's domain.
4. Repeat: Continue checking and revising until all arcs are consistent.

Example:

• In map coloring, if WA = [Red, Blue] and NT = [Blue], remove Red from WA because NT cannot match Red.

Benefits:

• Reduces search space and speeds up problem-solving by eliminating invalid options early.

```
function AC-3(csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables {X1, X2, ..., Xn} local variables:
    queue, a queue of arcs, initially all the arcs in csp while queue is not
    empty do
        (Xi, Xj)←Remove-First(queue)
        if Remove-Inconsistent-Values(Xi, Xj) then for each Xk
            in Neighbors[Xi] do add (Xk, Xi) to queue
```

```
function Remove-Inconsistent-Values(Xi, Xj) returns true iff succeeds remo
    for each x in Domain[Xi] do if no value y in Domain[Xj] allows (x,y) to sa
        constraint Xi ↔ Xj then delete x from Domain[Xi]; removed←true
    return removed
```

1. Initialization:

- Start with a CSP and a queue of arcs (constraints).

2. Arc Consistency Check:

- While the queue isn't empty:
- Remove an arc (X_i, X_j) from the queue.
- Check for inconsistency using REMOVE-

INCONSISTENT-VALUES(X_i, X_j).

• If values are removed from X_i’s domain, add affected arcs to the queue.

3. REMOVE-INCONSISTENT-VALUES Function:

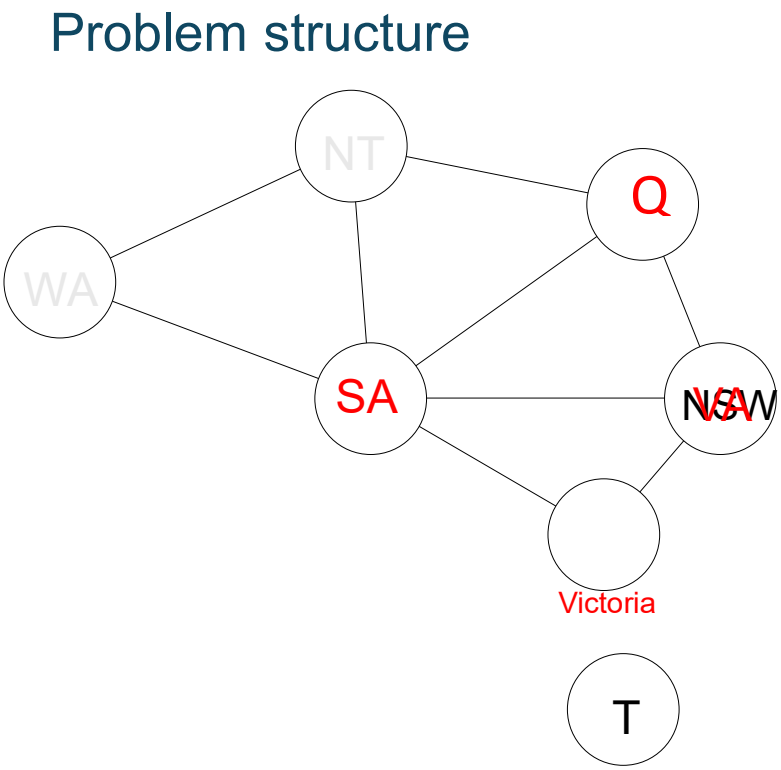
• For each value in X_i’s domain, check if there’s a matching value in X_j’s domain.

• If no match, remove the value from X_i’s domain.

4. Termination:

• The algorithm ends when the queue is empty, meaning all arcs are consistent.

$O(n^2d^3)$, can be reduced to $O(n^2d^2)$ (but detecting **all** is NP-hard)



Tasmania and mainland are independent subproblems

Subproblems and Independence: Breaking a CSP into smaller, independent parts makes it easier to solve.

• Connected Components: Some parts of the problem (like Tasmania and the mainland) are not connected by constraints, making them independent subproblems.

Impact of Subproblem Size:

- Solution Cost: The bigger the subproblem, the harder and longer it takes to solve.

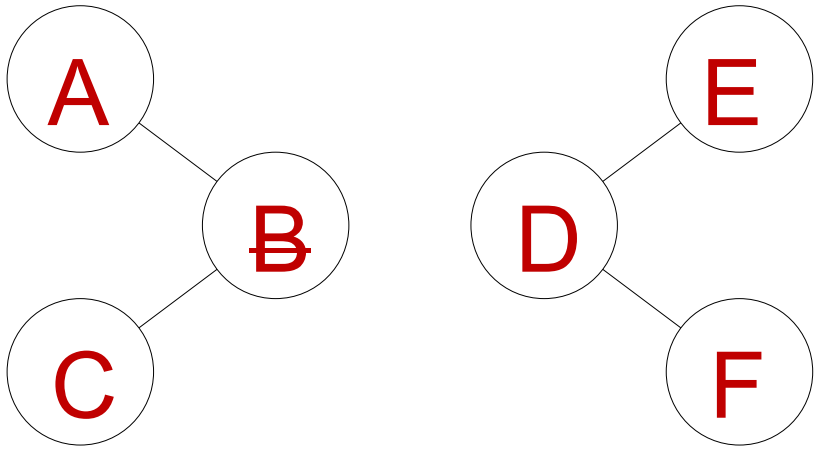
Example:

- A large subproblem might take billions of years to solve, but a small one only seconds.

Key Takeaways:

- Dividing the problem into smaller parts speeds up the solution.
- Smaller subproblems are much faster to solve.

Tree-structured CSPs



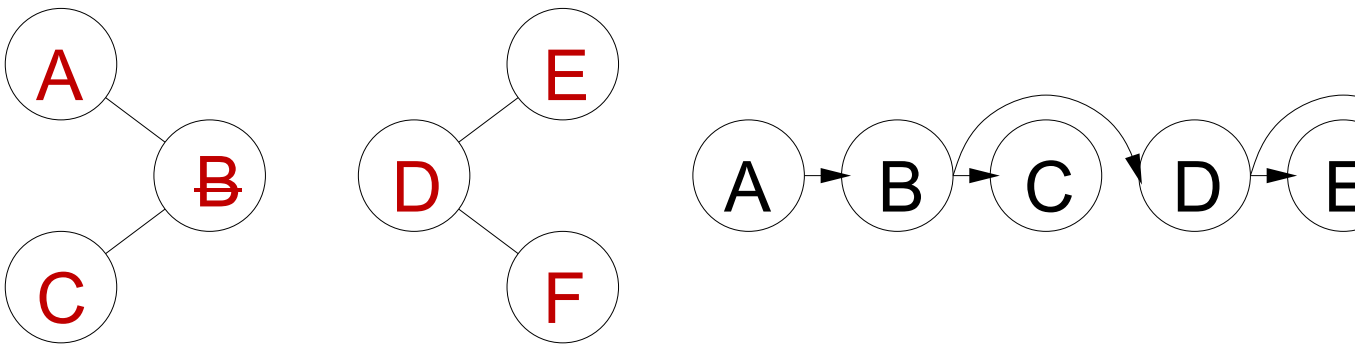
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time

Compare to general CSPs, where worst-case time is $O(d^n)$

Algorithm for tree-structured CSPs

---

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



2. For j from n down to 2, apply RemoveInconsistent$(Parent(X_j), X_j)$

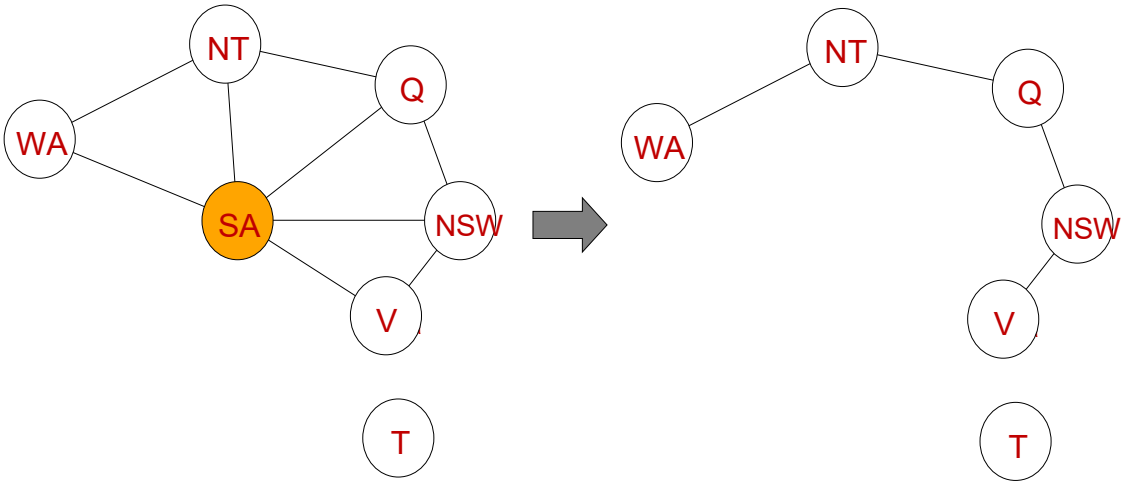
3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

1. Order the Variables: Arrange the variables in a way where the root comes first, followed by its children, ensuring that each parent comes before its child.
2. Remove Conflicting Values: Starting from the leaves and moving toward the root, remove any values from a variable that don't satisfy the constraints with its parent.
3. Assign Values: Start from the root and assign values to each variable that don't conflict with the value of its parent.

This approach works efficiently because it removes inconsistent values early, making the assignment of values straightforward and avoiding conflicts.

Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size c ⇒ runtime $O(d^c \cdot (n-c)d^2)$, very fast for small c

Explanation

Nearly Tree-Structured CSPs:

- Challenge: Some CSPs have extra edges that create cycles.
- Solution: Use "cutset conditioning."
- How it works: Identify a small set of variables

(cutset) whose removal makes the CSP tree-structured.

• Process: Try all possible combinations for the cutset variables, then solve the remaining tree-structured CSP.

• Complexity: O(d^c * n * d^2), where c is the cutset size, n is the number of variables, and d is the domain size.

Iterative Algorithms for CSPs:

• Backtracking: Explore the search space by trying different assignments and backtrack when a contradiction occurs.

• Forward Checking: Prune the search space by removing invalid values from unassigned variables.

• Arc Consistency: Enforce that for every constraint, there is a compatible value in both variable domains.

SUMMARY

What is a CSP?

• CSPs involve finding values for variables that meet given rules.

Solving CSPs:

• **Backtracking**: Try values and go back if something goes wrong.

• **Heuristics**: Choose the best options to speed up the process.

• **Forward Checking**: Prevent bad choices early.

• **Constraint Propagation**: Detect mistakes quickly to avoid unnecessary checks.

Problem Structure and Efficiency:

• The structure affects how fast a CSP is solved.

• **Tree-structured CSPs**: Can be solved quickly.

• **Iterative min-conflicts**: A fast method for some CSPs.