

Improving CSPs

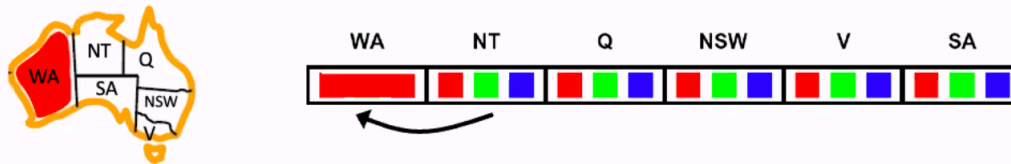
Filtering

Arc Consistency

- Arc $X \rightarrow Y$ is consistent \implies for every x in the tail, there is some y in the head which could be assigned without violating a constraint
- Arcs go in each direction

Consistency of A Single Arc

- An arc $X \rightarrow Y$ is **consistent** iff for every x in the tail there is some y in the head which could be assigned without violating a constraint



- Tail = NT, head = WA
 - If NT = blue: we could assign WA = red
 - If NT = green: we could assign WA = red
 - If NT = red: there is no remaining assignment to WA that we can use
 - Deleting NT = red from the tail makes this arc consistent
- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Figure 1: Screenshot_2023-09-07_at_5.24.50_PM.png

-
- If we change something to enforce consistency, we have to recheck all arcs to and from that updated value
- $O(n^2d^3)$ but can be reduced to $O(n^2d^2)$
 - $n :=$ number of arcs
 - $d := |D|$ $D :=$ domain
 - Poly time instead of exponential time

```

function AC-3(cs) returns the CSP, possibly with reduced domains
inputs: cs, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in cs ←
while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
    if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
        for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
            add  $(X_k, X_i)$  to queue



---


function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
    removed  $\leftarrow$  false
    for each  $x$  in DOMAIN[ $X_i$ ] do
        if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$ 
            then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
    return removed

```

- AC-3
 - Can be run inside of backtracking to enforce consistency of an entire CSP
- May result in an undefined state (no explicit solutions or none at all)

K-Consistency (Out of Scope)

- Increasing degrees of consistency
 - 1-consistency (node consistency) := each single node's domain D has a value which meets that node's unary constraints
 - 2-consistency (arc consistency) := for each pair of nodes, any consistent assignment to one can be extended to the other
 - K-consistency := for each k nodes, any consistent assignment to $k-1$ can be extended to the k^{th} node.
- Strong consistency := n -consistent $\forall n \in k$

Ordering

- Probably have to run some filtering after making a choice to see if the assignment is easy or hard

Minimum Remaining Values (MRV)

- Used for variable ordering
- Choose the variable with the fewest legal left values in its domain D
- AKA 'most constrained variable' or 'fast-fail ordering'

Least Constrained Value (LCV)

- Not used for value ordering
- Choose the option that rules out the fewest option in the remaining values
- Means you don't have to try the other values for that node

Structuring

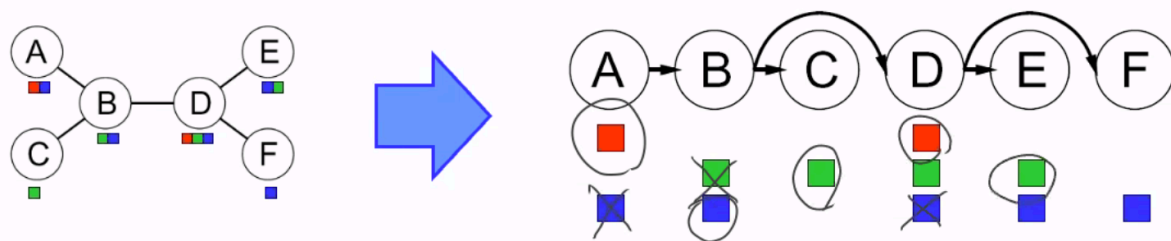
- Independent subproblems are identifiable as connected components of a constraint graph
 - Solve independent problems
- Generally there aren't independent subproblems in a CSP
 - We choose CSPs based off of conflicting variables
- Can have incredibly big affect on runtime

Tree-Structured CSPs

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

Algorithm for tree-structured CSPs:

- Order: Choose a root variable, order variables so that parents precede children



- Remove backward: For $i = n : 2$, apply $\text{RemoveInconsistent}(\text{Parent}(X_i), X_i)$
- Assign forward: For $i = 1 : n$, assign X_i consistently with $\text{Parent}(X_i)$

Figure 2: Screenshot_2023-09-07_at_6.10.04_PM.png

-
- Always works for all trees but not for SCCs because backtracking
- $O(nd^2)$

Cutset Conditioning

- Remove a cutset, instantiate the cutset, and compute residual CSP for each assignment
- $O((d^c)(n - c)(d^2))$
 - Very fast for small c
- Finding the cutset can be hard

Iterative Improvement

- Local search methods typically work with 'complete' states (all variables assigned)
- Idea: take a random assignment with unsatisfied constraints and attempt to fix the problems
- No fringe
-
- No guarantee of correctness
- Tends to be close to $O(1)$ for CSPs except in a narrow 'critical range' R
 - $R := \frac{\text{number of constraints}}{\text{number of variables}}$
 - Most CSPs are in that critical range

- **Algorithm: While not solved,**
 - **Variable selection: randomly select any conflicted variable**
 - **Value selection: min-conflicts heuristic:**
 - Choose a value that violates the fewest constraints
 - I.e., hill climb with $h(n)$ = total number of violated constraints

Figure 3: Screenshot_2023-09-07_at_6.21.18_PM.png

Local Search

- Improve a single option until you can't make it better (no fringe)
 - Generalized iterative improvement to search problems
- **Simple, general idea:**
 - **Start wherever**
 - **Repeat: move to the best neighboring state**
 - **If no neighbors better than current, quit**

Figure 4: Screenshot_2023-09-07_at_6.28.11_PM.png

-
- Not necessarily complete or optimal
- Finds a local max instead of a global max
- Can be optimized with simulated annealing or genetic search