Improving CSPs

Filtering

Arc Consistency

- Arc $X \to Y$ 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

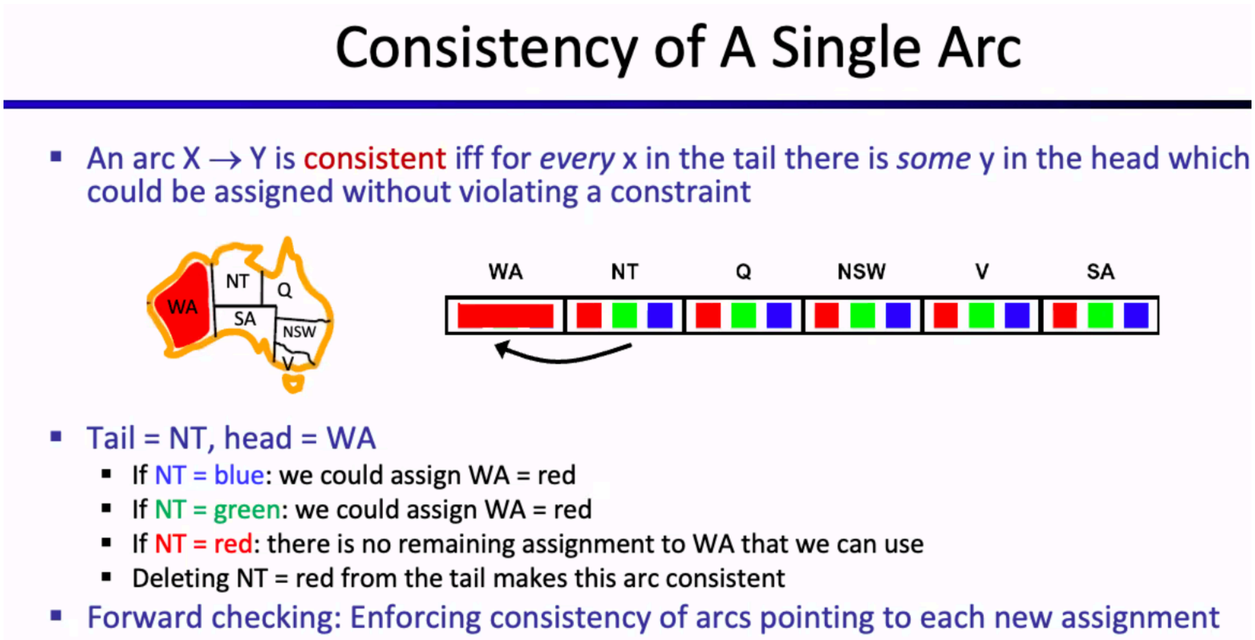

Figure 1: Screenshot_2023-09-07_at_5.24.50_PM.png

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

```
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
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi];  removed ← 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 $O(nd^2)$

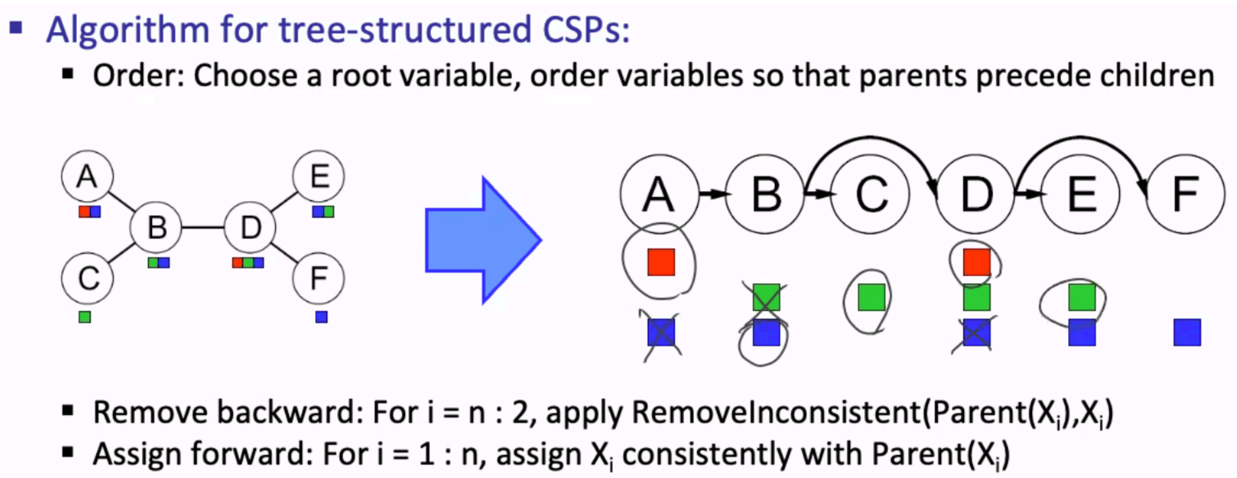

Figure 2: Screenshot_2023-09-07_at_6.10.04_PM.png

-
- 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

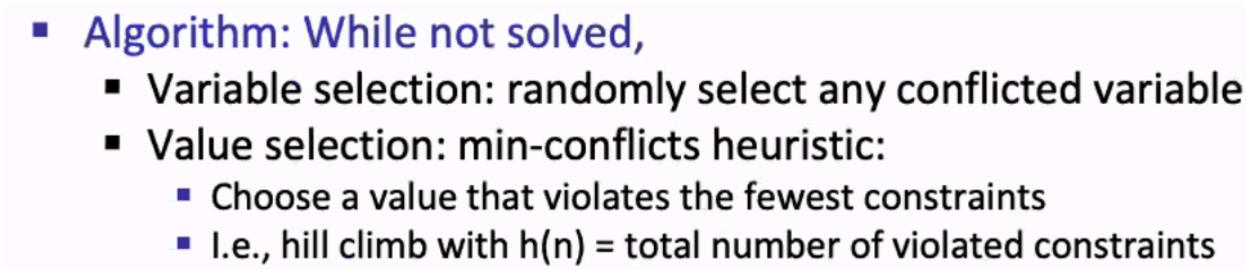

Figure 3: Screenshot_2023-09-07_at_6.21.18_PM.png

Local Search

- Improve a single option until you can't make it better (no fringe)
- Generalized iterative improvement to search problems

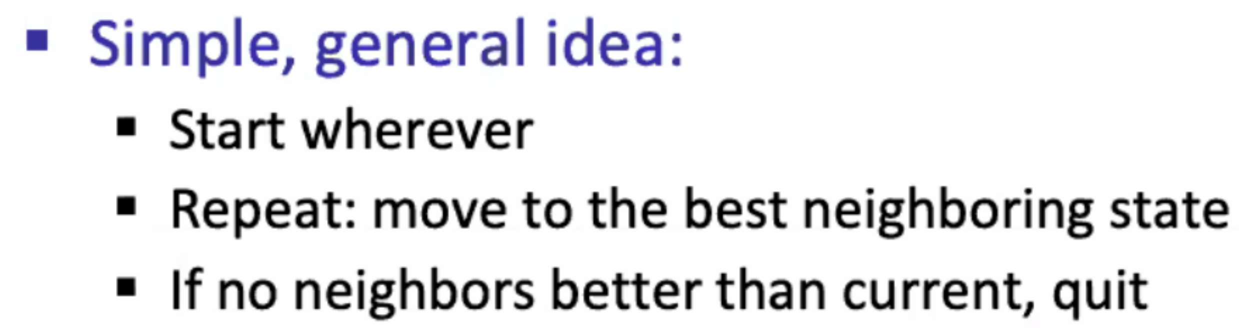

Figure 4: Screenshot_2023-09-07_at_6.28.11_PM.png

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