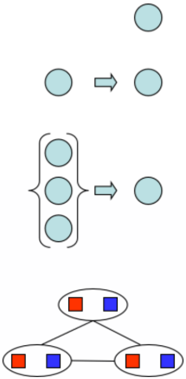
k-Consistency

k-Consistency

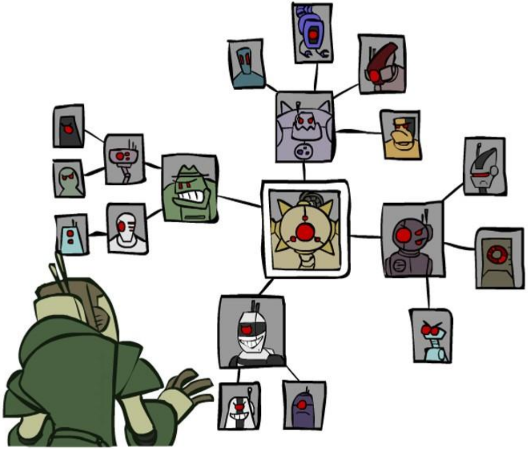
- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - k-Consistency: For each k nodes, any consistent assignment to $k-1$ can be extended to the k^{th} node.
- The higher the k, the more expensive to compute



Strong k-Consistency

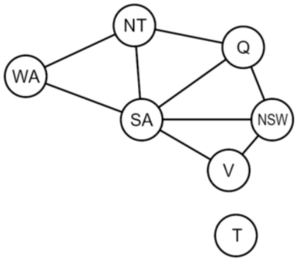
- Also $k-1, k-2, \ldots, 1$ consistent
- Claim: strong n-consistency means we can solve without backtracking!
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - $\ldots$

Problem Structure



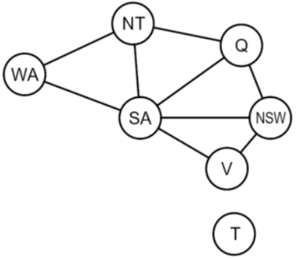
Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact



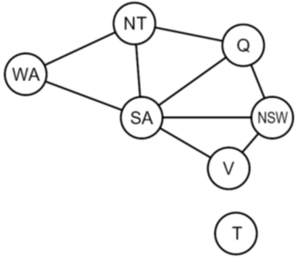
Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph

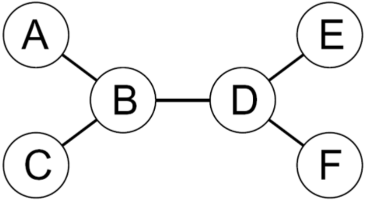


Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph
  - Use DFS!
- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - Compared to $O(d^n)$ for naïve backtracking
    - e.g., n = 80, d = 2, c = 20
    - $2^{80} = 4$ billion years at 10 million nodes/sec
    - $(4)(2^{20}) = 0.4$ seconds at 10 million nodes/sec



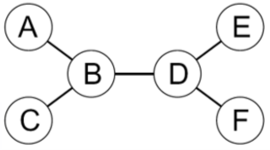
Tree-Structured CSPs



- 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 $O(d^n)$
- Only one incoming arc per node

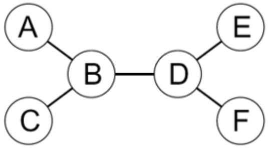
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



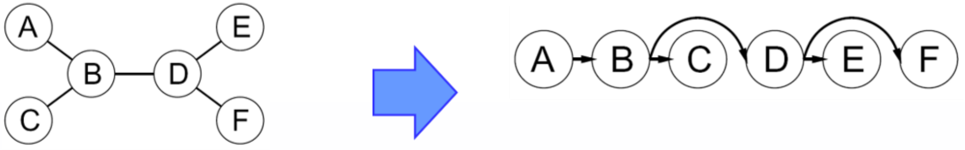
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



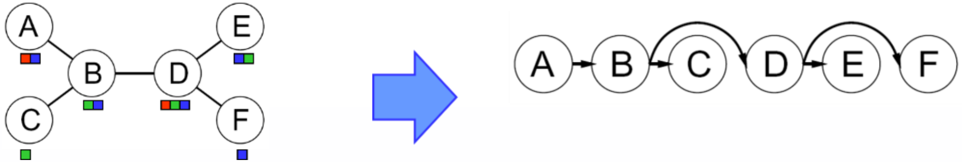
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



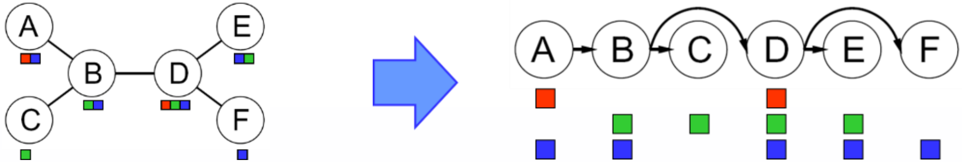
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



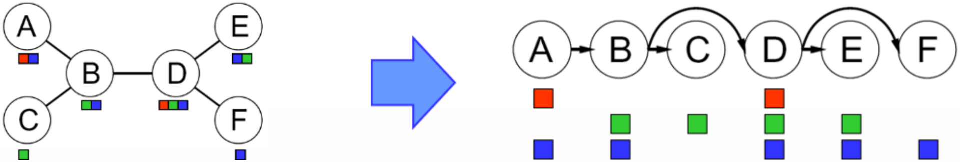
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



Tree-Structured CSPs

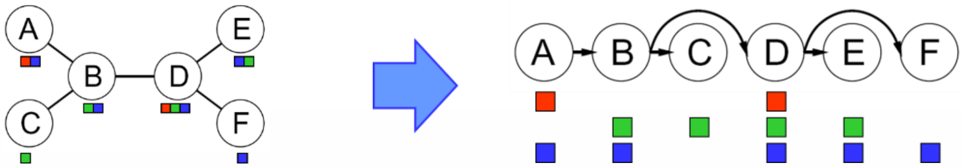
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - Remove backward: For $i = n : 2$, apply RemoveInconsistent(Parent(X_i), X_i)

Tree-Structured CSPs

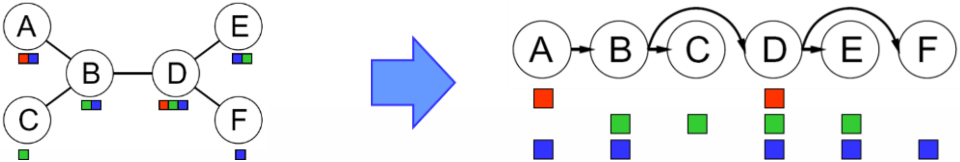
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



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

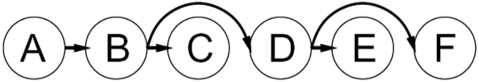
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



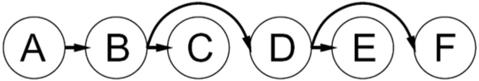
  - Remove backward: For $i = n : 2$, apply RemoveInconsistent(Parent(X_i), X_i)
  - Assign forward: For $i = 1 : n$, assign X_i consistently with Parent(X_i)
- Runtime: $O(nd^2)$
  - Go from tail to head, and then head to tail $\rightarrow O(n)$
  - Check pairs of values for consistency/assignment $\rightarrow O(d^2)$

Tree-Structured CSPs



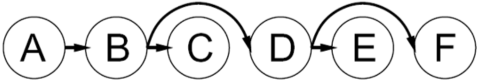
- Claim #1: After backward pass, all root-to-leaf arcs are consistent

Tree-Structured CSPs



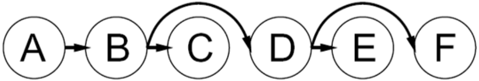
- Claim #1: After backward pass, all root-to-leaf arcs are consistent
  - Proof: Each $X \rightarrow Y$ was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)

Tree-Structured CSPs



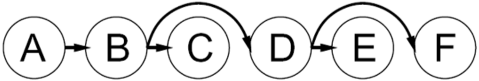
- Claim #1: After backward pass, all root-to-leaf arcs are consistent
  - Proof: Each $X \rightarrow Y$ was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)
- Claim #2: If root-to-leaf arcs are consistent, forward assignment will not backtrack

Tree-Structured CSPs



- Claim #1: After backward pass, all root-to-leaf arcs are consistent
  - Proof: Each $X \rightarrow Y$ was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)
- Claim #2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
  - Proof: Induction on position

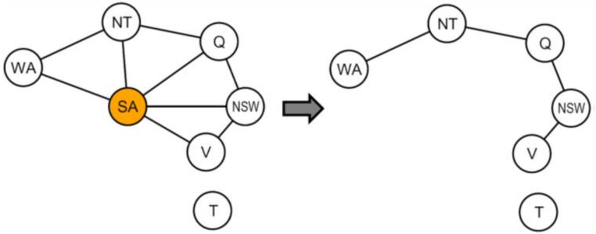
Tree-Structured CSPs



- Claim #1: After backward pass, all root-to-leaf arcs are consistent
  - Proof: Each $X \rightarrow Y$ was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)
- Claim #2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
  - Proof: Induction on position
- Why doesn't this algorithm work with cycles in the constraint graph?

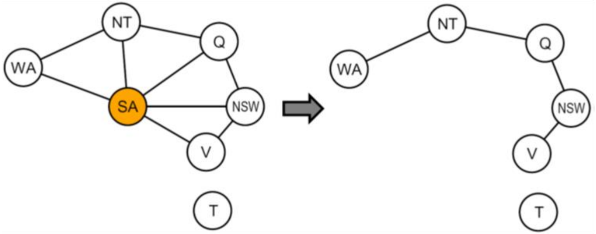
Improving Structure

Nearly Tree-Structured CSPs



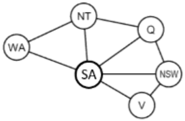
- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Nearly Tree-Structured CSPs



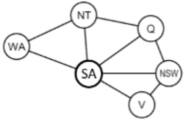
- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime $O((d^c)(n-c)d^2)$, very fast for small c
  - Total number of instantiation: $O(d^c)$
  - Total number of remaining subproblems: $(n-c)$

Cutset Conditioning



Cutset Conditioning

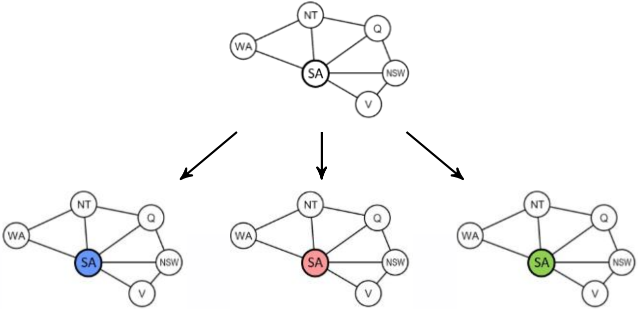
Choose a cutset



Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

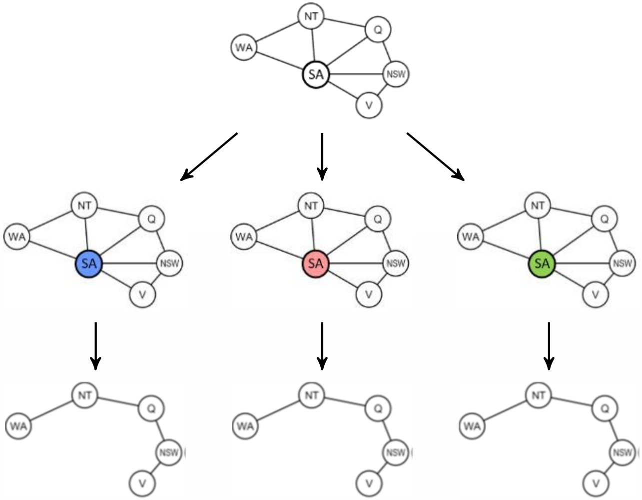


Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment



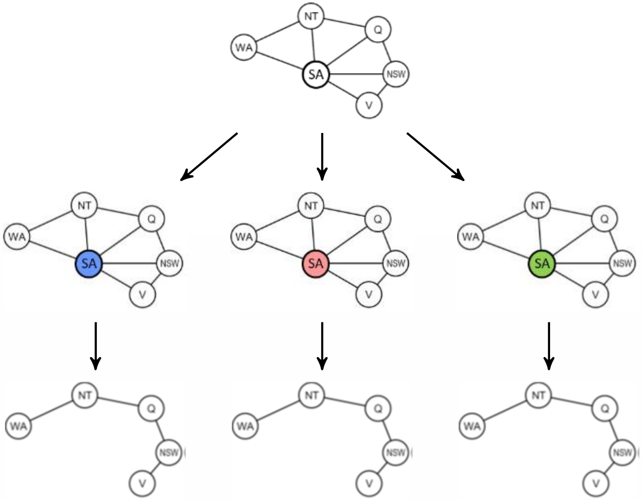
Cutset Conditioning

Choose a cutset

Instantiate the cut-set (all possible ways)

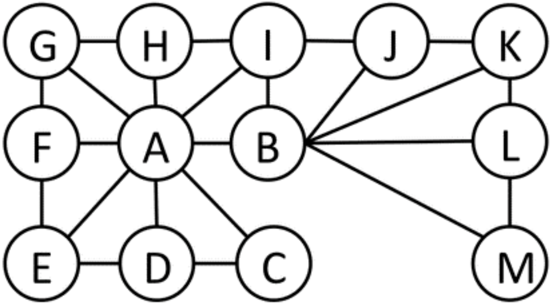
Compute residual CSP for each assign-ment

Solve the residual CSPs (tree struc-tured)

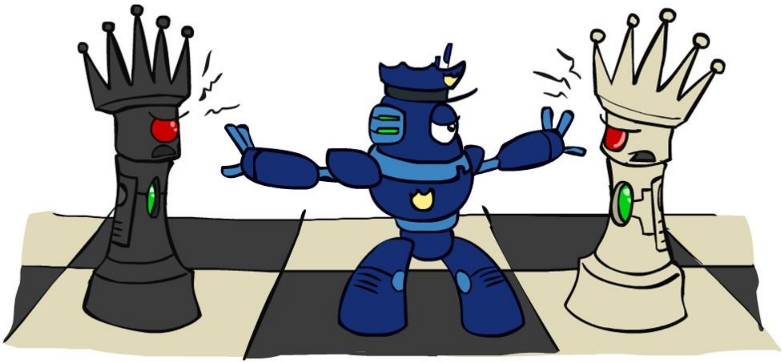


Cutset Quiz

- Find the smallest cutset for the graph below:



Iterative Improvement

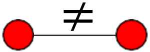


Iterative Algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned

Iterative Algorithms for CSPs

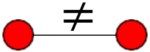
- Local search methods typically work with “complete” states, i.e., all variables assigned



Iterative Algorithms for CSPs

- Local search methods typically work with "complete" states, i.e., all variables assigned

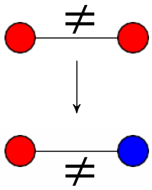
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe!



Iterative Algorithms for CSPs

- Local search methods typically work with "complete" states, i.e., all variables assigned

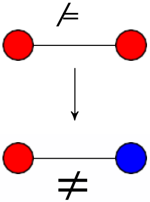
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe!



Iterative Algorithms for CSPs

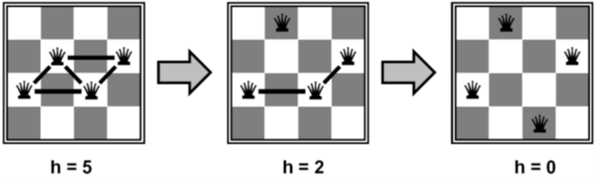
- Local search methods typically work with "complete" states, i.e., all variables assigned

- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe!



- 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

Example: 4-Queens



- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: $c(n)$ = number of attacks

Video: 5-queens-iterative-improvement
Website: [complex - iterating improvement]

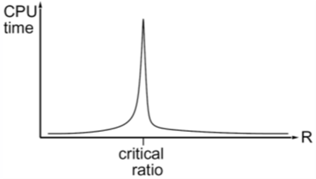
Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!

Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!
- The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

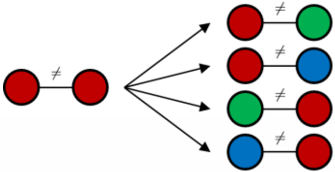
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Local Search

Local Search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)
- Local search: improve a single option until you can't make it better (no fringe!)
- New successor function: local changes

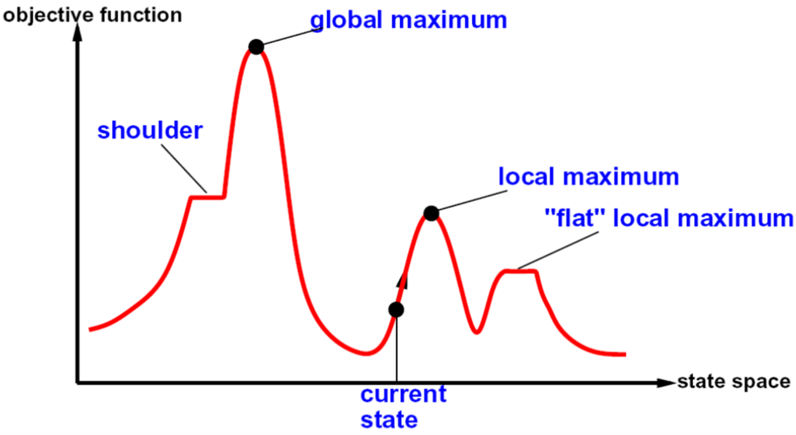


- Generally much faster and more memory efficient (but incomplete and suboptimal)

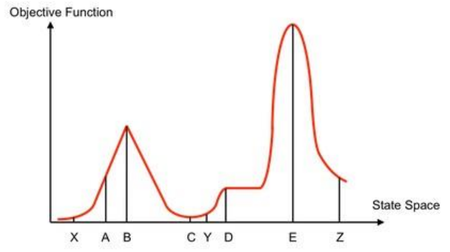
Hill Climbing

- Simple, general idea
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit
- What's bad about this approach?
  - Complete?
  - Optimal?
- What's good about it?

Hill Climbing Diagram



Hill Climbing Quiz



Starting from X, where do you end up?

Starting from Y, where do you end up?

Starting from Z, where do you end up?

Suggested Reading

- Russell & Norvig: Chapter 6.2-6.5