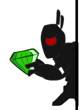
CSE 473: Artificial Intelligence
Winter 2017

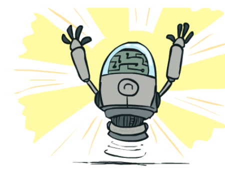
Constraint Satisfaction Problems - Part 2

Steve Tanimoto



With slides from :
Dieter Fox, Dan Weld, Dan Klein, Stuart Russell, Andrew Moore, Luke Zettlemoyer

Improving Backtracking

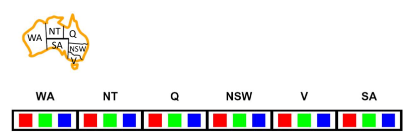


- General-purpose ideas give huge gains in speed
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?

Filtering

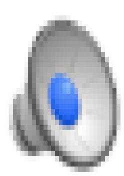
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



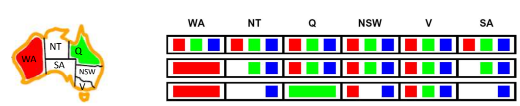
[Demo: coloring -- forward checking]

Video of Demo Coloring – Backtracking with Forward Checking



Filtering: Constraint Propagation

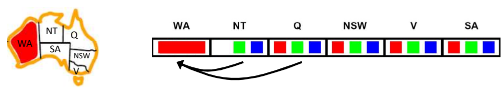
- Forward checking only propagates information from assigned to unassigned
- It doesn't catch when two unassigned variables have no consistent assignment:

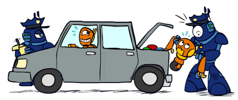


- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of a Single Arc

- An arc X → 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



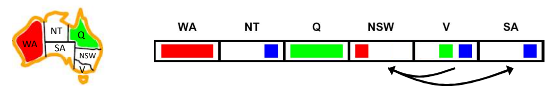


Delete from the tail!

- Forward checking: Enforcing consistency ***of arcs pointing to each new assignment***

Arc Consistency of an **Entire CSP**

- A simple form of propagation makes sure all arcs are consistent:



- Important: 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
- What's the ***downside*** of enforcing arc consistency?

Remember: Delete from the tail!

AC-3 algorithm for Arc Consistency

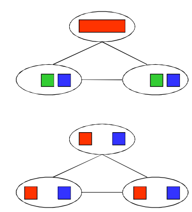
```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   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
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting ***all*** possible future problems is NP-hard – why?

[Demo: CSP applet (made available by aispace.org) -- n-queens]

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)
- Arc consistency still runs inside a backtracking search!



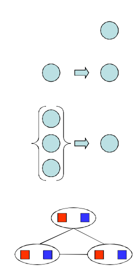
What went wrong here?

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

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.
- Higher k more expensive to compute
- (You need to know the algorithm for k=2 case: arc consistency)

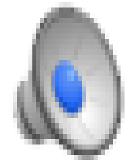


Strong K-Consistency

- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - 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
  - …
- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)

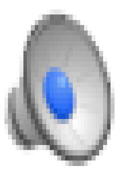
14

Video of Demo Arc Consistency – CSP Applet – n Queens



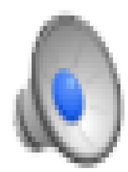
15

Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph



16

Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph



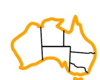
17

Ordering

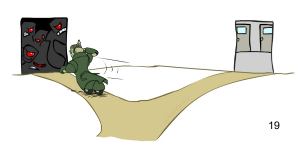
18

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain



- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering



19

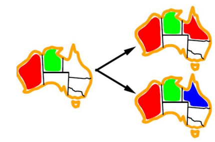
Ordering: Maximum Degree

- Tie-breaker among MRV variables
  - What is the very first state to color? (All have 3 values remaining.)
- Maximum degree heuristic:
  - Choose the variable participating in the most constraints on remaining variables

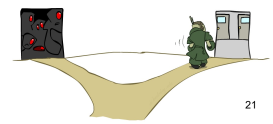
- Why most rather than fewest constraints?

Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



- Why least rather than most?

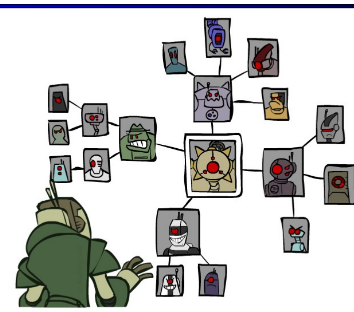


- Combining these ordering ideas makes 1000 queens feasible

Rationale for MRV, MD, LCV

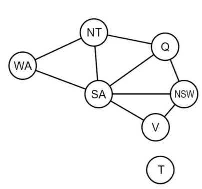
- We want to enter the most promising branch, but we also want to detect failure quickly
- MRV+MD:
  - Choose the variable that is most likely to cause failure
  - It must be assigned at some point, so if it is doomed to fail, better to find out soon
- LCV:
  - We hope our early value choices do not doom us to failure
  - Choose the value that is most likely to succeed

Structure

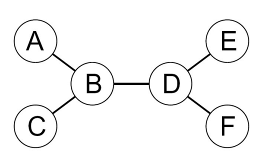


Problem Structure

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



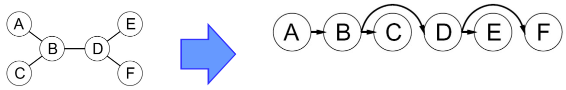
Tree-Structured CSPs



- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$
- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

Tree-Structured CSPs

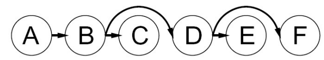
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - 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(n d^2) (why?)

Tree-Structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→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)



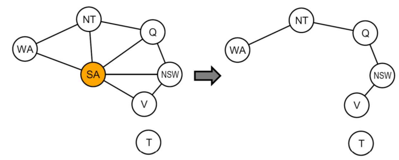
- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position
- Why doesn't this algorithm work with cycles in the constraint graph?
- Note: we'll see this basic idea again with Bayes' nets

28

Improving Structure

29

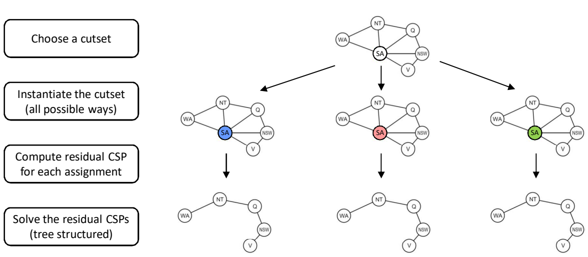
Nearly Tree-Structured CSPs



- 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

30

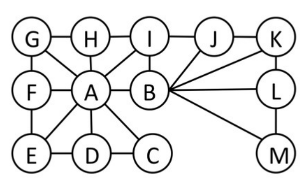
Cutset Conditioning



31

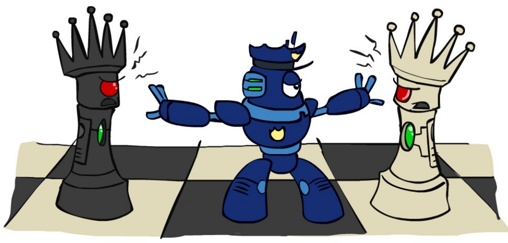
Cutset Quiz

- Find the smallest cutset for the graph below.



32

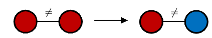
Local Search for CSPs



34

Iterative Algorithms for CSPs

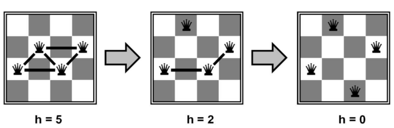
- Local search methods typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



- 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

35

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

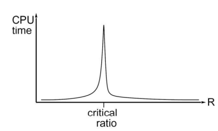
[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

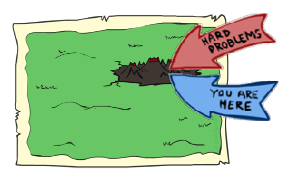
36

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

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

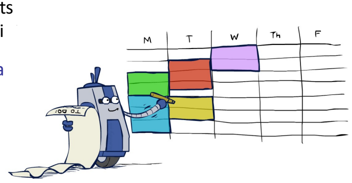




39

Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constrai
- Basic solution: backtracking sea
- Speed-ups:
  - Ordering
  - Filtering
  - Structure



- Iterative min-conflicts is often effective in practice

40