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: 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

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

Video of Demo Coloring – Backtracking with Forward Checking

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

- 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!
  - Can be run as a preprocessor or after each assignment.
  - What's the downside of enforcing arc consistency?

AC-3 algorithm for Arc Consistency

```plaintext
AC3 algorithm:

1. Function AC3(X):
   - Input: X
   - Output: AC3(X)
   - AC3(X): if X is not arc consistent
     - return false

2. For each pair (X, Y) in arcs:
   - if X and Y are consistent
     - continue
   - if X and Y are not consistent
     - delete (X, Y) from arcs

3. While there are removed arcs:
   - continue

AC3(X) is consistent iff there are no removed arcs.
```

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!

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)

Video of Demo Arc Consistency – CSP Applet – n Queens

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

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

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
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?

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

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

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 $O(n/c^d)$, linear in n
- E.g., n = 80, d = 2, c > 20
- $2^{40} = 4 \text{ billion years at 10 million nodes/sec}$
- $4 \times (2^{40}) = 4.4 \text{ seconds at 10 million nodes/sec}$

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?)

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?

Note: we’ll see this basic idea again with Bayes’ nets

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
- Cutset size c gives runtime O((d^c)(n-c)d^2), very fast for small c

Cutset Conditioning

- Choose a cutset
- Instantiate the cutset (all possible ways)
- Compute residual CSP for each assignment
- Solve the residual CSPs (tree structured)

Cutset Quiz

- Find the smallest cutset for the graph below.
Local Search for CSPs

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 \( h(n) = \text{total number of violated constraints} \)

Example: 4-Queens

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

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

Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints
- Basic solution: backtracking search
- Speed-ups:
  - Ordering
  - Filtering
  - Structure
- Iterative min-conflicts is often effective in practice