Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Start with Depth First Search
  - “backtracking search” IS a Kind of depth first search with these 2 details:
    - Idea 1: One variable at a time
      - Variable assignments are commutative, so fix ordering
      - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
      - Only need to consider assignments to a single variable at each step
    - Idea 2: Check constraints as you go
      - I.e. consider only values which do not conflict previous assignments
      - Might have to do some computation to check the constraints
      - “Incremental goal test”
- Can solve n-queens for $n \approx 25$
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: 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]
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!
- 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?
### AC-3 algorithm for Arc Consistency

**Algorithm**

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables \{X_1, X_2, \ldots, X_n\}
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
  (X_i, X_j) ← REMOVE-FIRST(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 ← false
for each \(x\) in Domain[X_i] do
  if no value \(y\) in Domain[X_j] allows \((x, y)\) to satisfy the constraint \(X_i \rightarrow X_j\)
  then delete \(x\) from Domain[X_i]; 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 aiaspace.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)

- **Even with Arc Consistency you still need backtracking search:**
  - Could run at every step of that search
  - Usually better to run it *once, before search*
<table>
<thead>
<tr>
<th>Video of Demo Arc Consistency – CSP Applet – n Queens</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph</th>
</tr>
</thead>
</table>
Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph

K-Consistency
K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single variable’s domain has a value which meets that variables unary constraints
  - 2-Consistency (Arc Consistency): For each pair of variables, any consistent assignment to one can be extended to the other
  - 3-Consistency (Path Consistency): For every set of 3 vars, any consistent assignment to 2 of the variables can be extended to the third var
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the kth 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
  - ...
### Ordering

### Backtracking Search

```python
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return Recursive-Backtracking(\{ \}, csp)

function Recursive-Backtracking(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given Constraints[csp] then
            add \{ var = value \} to assignment
            result ← Recursive-Backtracking(assignment, csp)
            if result ≠ failure then return result
            remove \{ var = value \} from assignment
        return failure
```

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

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
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 $O((n/c)(d^3))$, 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($\text{Parent}(X_i), X_i$)
- Assign forward: For $i = 1 : n$, assign $X_i$ consistently with $\text{Parent}(X_i)$

- Runtime: $O(n d^2)$ (why?)
Tree-Structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
  - Proof: Each $X \rightarrow Y$ 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.

![Graph Image]

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 \textit{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

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]
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

\[
R = \frac{\text{number of constraints}}{\text{number of variables}}
\]

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 (cutset conditioning)

- Iterative min-conflicts is often effective in practice