CIS 471/571 (Winter 2020): Introduction to Artificial Intelligence

Lecture 5: Constraint Satisfaction Problems (Part 2)

Thanh H. Nguyen

Source: http://ai.berkeley.edu/home.html
Announcements

- Project 2:
  - Deadline: Feb 02, 2020

- Homework 2:
  - Deadline: Feb 03, 2020
  - Will be posted on Jan 22th, 2020
Reminder: CSPs

- CSPs:
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a list of the legal tuples)
    - Unary / Binary / N-ary

- Goals:
  - Here: find any solution
  - Also: find all, find best, etc.
Backtracking Search

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
Improving Backtracking

- General-purpose ideas give huge gains in speed

- Filtering: Can we detect inevitable failure early?
  - Arc consistency
  - Forward checking
  - Constraint propagation

- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

- Structure: Can we exploit the problem structure?
Example: Map Coloring

Western Australia
Northern Territory
Queensland
South Australia
New South Wales
Victoria
Tasmania
Example: Map Coloring

- 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.
- Enforcing consistency of $X \rightarrow Y$: filter values of the tail $X$ to make $X \rightarrow Y$ **consistent**.
- Forward checking: Enforcing consistency of arcs pointing to each new assignment.
Example: Map Coloring

- **Constraint propagation**: enforce arc consistency of entire CSP
  - Maintain a queue of arcs to enforce consistency
- **Important**: If X loses a value, neighbors of X need to be rechecked!
  - After enforcing consistency on $X \rightarrow Y$, if X loses a value, all arcs pointing to X need to be added back to the queue
Ordering
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: 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
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 $O(d^n)$
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: $O(n \cdot d^2)$ (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
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

1. Choose a cutset
2. Instantiate the cutset (all possible ways)
3. Compute residual CSP for each assignment
4. Solve the residual CSPs (tree structured)
Cutset Quiz

- Find the smallest cutset for the graph below.
Tree Decomposition*

- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions

\[
\begin{align*}
\{(W_A=r, S_A=g, N_T=b), & \quad (W_A=b, S_A=r, N_T=g), \\
\{(N_T=r, S_A=g, Q=b), & \quad (N_T=b, S_A=g, Q=r),
\end{align*}
\]

Agree on shared vars

\[
\{(W_A=g, S_A=g, N_T=g), \quad (N_T=g, S_A=g, Q=g)\}
\]

Agree: \((M_1, M_2) \in \{(W_A=g, S_A=g, N_T=g), (N_T=g, S_A=g, Q=g)\}, \ldots\)
Iterative Improvement
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: $c(n) =$ 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

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

- Iterative min-conflicts is often effective in practice
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

objective function

global maximum

shoulder

local maximum

"flat" local maximum

current state

state space
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?
Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

function SIMULATED-ANNEALING( problem, schedule) returns a solution state

inputs: problem, a problem
        schedule, a mapping from time to “temperature”

local variables: current, a node
                 next, a node
                 T, a “temperature” controlling prob. of downward steps

current ← Make-Node(Initial-State[problem])
for t← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE[next] - VALUE[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability $e^{ΔE/T}$