CS 343: Artificial Intelligence

CSPs II + Local Search

Prof. Yuke Zhu

The University of Texas at Austin

[These slides based on those of Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]
Announcements

- **Homework 1 & Reading: Adversarial Search, Utilities**
  - Due Yesterday 2/7

- **Project 1: Search**
  - Reminder: Due Wednesday 2/9 at 11:59 pm

- **Homework 2: CSPs**
  - Due Monday 2/23, at 11:59 pm.
  - Multiple resubmissions allowed.
Last time: 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.
Last time: Backtracking
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.

Delete from the tail!
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!
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?
Improving Backtracking

- General-purpose ideas give huge gains in speed
  - ... but it’s all still NP-hard

- Filtering: Can we detect inevitable failure early?

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

- Structure: Can we exploit the problem 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., \( n = 80, d = 2, c = 20 \)
  - \( 2^{80} = 4 \text{ billion years at 10 million nodes/sec} \)
  - \( (4)(2^{20}) = 0.4 \text{ seconds at 10 million nodes/sec} \)
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($\text{Parent}(X_i), X_i$)
  - 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→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
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), removing any inconsistent domain values w.r.t. cutset assignment

\[ d^c \]

\[ (n-c)d^2 \]
Exercise: Cutset Exercise
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 $h(n) = \text{total number of violated constraints}$

- Can get stuck in local minima (we’ll come back to this idea in a few slides)
Example: 4-Queens

- States: 4 queens in 4 columns \( (4^4 = 256 \text{ states}) \)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: \( c(n) = \text{number of attacks} \)
Video of Demo Iterative Improvement – n Queens
Performance of Min-Conflicts

- Runtime of min-conflicts is on n-queens is *roughly independent of problem size*!
  - Why?? Solutions are densely distributed in state space

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000) in ~50 steps!

- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

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

- 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 Exercise
Simulated Annealing

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

```haskell
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 $e^{ΔE/T}$
```

Shake!
Beam Search

- Like greedy hillclimbing search, but keep K states at all times:

- Variables: beam size, encourage diversity?
- The best choice in MANY practical settings
- Complete? Optimal?
- Why do we still need optimal methods?
Gradient Methods

- Continuous state spaces
  - Problem! Cannot select optimal successor

- Discretization or random sampling
  - Choose from a finite number of choices

- Continuous optimization: Gradient ascent
  - Take a step along the gradient (vector of partial derivatives)

- What if you can’t compute gradient?
  - i.e. maybe you can only sample the function
  - Estimate gradient from samples!
  - “Stochastic gradient descent”
  - We will return to this in neural networks / deep learning

\[
\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)
\]

\[
x \leftarrow x + \alpha \nabla f(x)
\]
Genetic algorithms use a natural selection metaphor

- Keep best N hypotheses at each step (selection) based on a fitness function
- Also have pairwise crossover operators, with optional mutation to give variety

Possibly the most misunderstood, misapplied (and even maligned) technique around
Example: N-Queens

- Why does crossover make sense here?
- When wouldn’t it make sense?
- What would mutation be?
- What would a good fitness function be?
Next Time: Adversarial Search!