CSE 473: Artificial Intelligence

Constraint Satisfaction

Dieter Fox

Based on slides adapted Luke Zettlemoyer, Dan Klein, Stuart Russell or Andrew Moore

What is Search For?

- Models of the world: single agent, deterministic actions, fully observed state, discrete state space

- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics to guide, fringe to keep backups

- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems

Constraint Satisfaction Problems

- Standard search problems:
  - State is a "black box": arbitrary data structure
  - Goal test: any function over states
  - Successor function can be anything

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables \( X_i \) with values from a domain \( D \) (sometimes \( D \) depends on \( i \))
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Simple example of a formal representation language

- Allows useful general-purpose algorithms with more power than standard search algorithms

Example: N-Queens

- Formulation 1:
  - Variables: \( X_{ij} \)
  - Domains: \{0, 1\}
  - Constraints:
    - \( \forall i, j, k \ (X_{ij}, X_{jk}) \in \{(0,0),(0,1),(1,0)\} \)
    - \( \forall i, j, k \ (X_{ij}, X_{kj}) \in \{(0,0),(0,1),(1,0)\} \)
    - \( \forall i, j, k \ (X_{ij}, X_{i+k,j+k}) \in \{(0,0),(0,1),(1,0)\} \)
    - \( \sum_{i,j} X_{ij} = N \)

- Note: need to make sure that constraints refer to different squares

Example: Map-Coloring

- Variables: \( WA, NT, Q, NSW, V, SA, T \)
- Domain: \( D = \{red, green, blue\} \)
- Constraints: adjacent regions must have different colors
  - \( WA \neq NT \)
  - \( (WA, NT) \in \{(red,green), (red,blue), (green,red),\ldots\} \)
- Solutions are assignments satisfying all constraints, e.g.:
  - \( WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green \)
Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Example: Cryptarithmetic

- Variables (circles):
  \[ F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3 \]
- Domains:
  \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
- Constraints (boxes):
  \[ \text{alldiff}(F, T, U, W, R, O) \]
  \[ O + O = R + 10 \cdot X_1 \]
  ... 

Example: Sudoku

- Variables:
  - Each (open) square
- Domains:
  \{1,2,...,9\}
- Constraints:
  - 9-way alldiff for each column
  - 9-way alldiff for each row
  - 9-way alldiff for each region

Varieties of CSPs

- Discrete Variables
  - Finite domains
    - Size \( n \) means \( O(n^d) \) complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable

- Continuous variables
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by LP methods

Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equiv. to shrinking domains):
    \[ SA \neq \text{green} \]
  - Binary constraints involve pairs of variables:
    \[ SA \neq WA \]
  - Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints
  - Preferences (soft constraints):
    - E.g., red is better than green
    - Often representable by a cost for each variable assignment
    - Gives constrained optimization problems
    - (We’ll ignore these until we get to Bayes’ nets)

Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Floorplanning
- Fault diagnosis
- ... lots more!
- Many real-world problems involve real-valued variables...
Standard Search Formulation

- Standard search formulation of CSPs (incremental)
- Let's start with a straightforward, dumb approach, then fix it
- States are defined by the values assigned so far
  - Initial state: the empty assignment, {}.
  - Successor function: assign a value to an unassigned variable.
  - Goal test: the current assignment is complete and satisfies all constraints.

Search Methods

- What does BFS do?
- What does DFS do?

Backtracking Search

- Idea 1: Only consider a single variable at each point
  - 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.
  - How many leaves are there?
- Idea 2: Only allow legal assignments at each point
  - I.e., consider only values which do not conflict previous assignments.
  - Might have to do some computation to figure out whether a value is ok.
  - 'Incremental goal test'.
- Depth-first search for CSPs with these two improvements is called backtracking search.
- Backtracking search is the basic uninformed algorithm for CSPs.
- Can solve n-queens for n \approx 25.

Backtracking Example
### 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?

### Forward Checking

- **Idea:** Keep track of remaining legal values for unassigned variables (using immediate constraints)
- **Idea:** Terminate when any variable has no legal values

---

**Backtracking**

**Are we done?**

**Forward Checking**

**Are We Done?**
Constraint Propagation

- Forward checking propagates information from assigned to adjacent unassigned variables, but doesn’t detect more distant failures:
  - NT and SA cannot both be blue!
  - Why didn’t we detect this yet?
  - Constraint propagation repeatedly enforces constraints (locally)

Arc consistency

- Simplest form of propagation makes each pair of variables consistent:
  - \( X \rightarrow Y \) is consistent if for every value of \( X \) there is some allowed value of \( Y \)

- If \( X \) loses a value, all pairs \( Z \rightarrow X \) need to be rechecked
Arc consistency

- Simplest form of propagation makes each pair of variables consistent:
  - $X \rightarrow Y$ is consistent iff for every value of $X$ there is some allowed value of $Y$
  - When checking $X \rightarrow Y$, throw out any values of $X$ for which there isn’t an allowed value of $Y$

- Arc consistency detects failure earlier than forward checking
- Can be run before or after each assignment

Arc Consistency

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

Limitations of Arc Consistency

- After running arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)
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\textsuperscript{th} node.

- Higher k more expensive to compute
- (You need to know the k=2 algorithm)

Ordering: Minimum Remaining Values

- Minimum remaining values (MRV):
  - Choose the variable with the fewest legal values

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

Ordering: Degree Heuristic

- Tie-breaker among MRV variables
- Degree heuristic:
  - Choose the variable participating in the most constraints on remaining variables

- Why most rather than fewest constraints?

Ordering: Least Constraining Value

- Given a choice of variable:
  - Choose the least constraining value
  - The one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this!

- Why least rather than most?
- Combining these heuristics makes 1000 queens feasible

Problem Structure

- Tasmania and mainland are independent subproblems
- Identifiable as connected components of constraint graph
- Suppose each subproblem has c variables out of n total
- Worst-case solution cost is $O(n/c)(d^c)$, linear in n
  - E.g., n = 80, d = 2, c = 20
  - $2^{20} = 4$ billion years at 10 million nodes/sec
  - $(4)(2^{20}) = 0.4$ seconds at 10 million nodes/sec

Tree-Structured CSPs

- Choose a variable as root, order variables from root to leaves such that every node’s parent precedes it in the ordering
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?)

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)

Iterative Algorithms for CSPs

- Greedy and local methods typically work with “complete” states, i.e., all variables assigned
- To apply to CSPs:
  - Allow states with unsatisfied constraints
  - Operators reassign variable values
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic:
  - Choose 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: $h(n) =$ 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 are a special kind of search problem:
  - States defined by values of a fixed set of variables
  - Goal test defined by constraints on variable values
- Backtracking = depth-first search with one legal variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- The constraint graph representation allows analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually 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

- 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

```plaintext
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  input: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   T, a "temperature" controlling prob. of downhill steps
  current ← MAKE-NODE(INITIAL-SOLUTION)  // solve!
  for t ← 1 to MAX-TEMP do
     if T = 0 then return current  // if current is the optimal solution
     next ← a randomly selected successor of current
     ΔE ← VALUE(next) - VALUE(current)
     if ΔE ≥ 0 then current ← next  // if the new solution is better
     else current ← next with probability e^(-ΔE/T)
```

- **Theoretical guarantee**: 
  - Stationary distribution: \( p(x) \propto e^{-E(x)/T} \)
  - If \( T \) decreased slowly enough, will converge to optimal state!

- Is this an interesting guarantee?
  - Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - People think hard about **ridge operators** which let you jump around the space in better ways

Genetic Algorithms

- 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?

GA’s for Locomotion

Ever wonder what it would be like to see evolution happening right before your eyes?