CS 188: Artificial Intelligence

Constraint Satisfaction Problems

Instructor: Nicholas Tomlin
University of California, Berkeley

[These slides adapted from Dan Klein, Pieter Abbeel, and Anca Dragan]
Announcements

- Waitlist has been emptied
- Lecture notes should be posted tonight
- Project 0 and Homework 1 due tomorrow, 11PM
Last time…
Last time…
Last time...
Q: Where do heuristics come from?
A: We have to create them!

Not the best heuristic…
Q: Where do heuristics come from?
A: We have to create them!

Not the best heuristic…
Q: Where do heuristics come from?
A: We have to create them!

What’s a better heuristic?
Q: Where do heuristics come from?
A: We have to create them!

What’s a better heuristic?
Last time...

- Failure to detect repeated states can cause exponentially more work.
Last time…

- Idea: never **expand** a state twice

- How to implement:
  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
Last time...
Last time...
Last time...

\[ f = g + h \]
\[ = 1 + 1 \]
\[ = 2 \]

\[ f = g + h \]
\[ = 1 + 4 \]
\[ = 5 \]

\[ f = g + h \]
\[ = 3 + 1 \]
\[ = 4 \]
Last time...

This heuristic isn’t consistent

“Triangle inequality”

\[ h(u) \leq d(u,v) + h(v) \]
This heuristic isn’t consistent

“Triangle inequality”
\[ h(u) \leq d(u,v) + h(v) \]

Q: Is \( h(A) \leq d(A,C) + h(C) \)?
This heuristic isn’t consistent

“Triangle inequality”
\( h(u) \leq d(u,v) + h(v) \)

Q: Is \( h(A) \leq d(A,C) + h(C) \)?
A: No: \( 4 \not\leq 1 + 1 \)
Summary of A*

- **Tree search:**
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)

- **Graph search:**
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)

- Consistency implies admissibility

- In general, most natural admissible heuristics tend to be consistent, especially if it comes from a relaxed problem
Bonus: Optimality of A* Graph Search

- Consider what A* does:
  - Expands nodes in increasing total f value (f-contours)
    Reminder: $f(n) = g(n) + h(n) = \text{cost to } n + \text{heuristic}$
  - Proof idea: the optimal goal(s) have the lowest f value, so it must get expanded first
Bonus: Optimality of A* Graph Search

Proof by contradiction:
- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
- Let $p$ be the ancestor of $n$ that was on the queue when $n'$ was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because $n'$ is suboptimal
- $p$ would have been expanded before $n'$
- Contradiction!
Beyond Pathfinding

A* can be used in a variety of domains besides path planning.

Even has applications to LLMs!
Constraint Satisfaction Problems

N variables
domain D
constraints

states
goal test
successor function

partial assignment
complete; satisfies constraints
assign an unassigned variable
What is Search For?

- Assumptions about the world: a 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 give problem-specific guidance

- 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 can be any function over states
  - Successor function can also 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

- Allows useful general-purpose algorithms with more power than standard search algorithms
CSP Examples
Example: Map Coloring

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

- **Formulation 1:**
  - **Variables:** $X_{ij}$
  - **Domains:** \{0, 1\}
  - **Constraints**

\[
\forall i, j, k \ (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \ (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \ (X_{ij}, X_{i+k,j+k}) \in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \ (X_{ij}, X_{i+k,j-k}) \in \{(0, 0), (0, 1), (1, 0)\} \\
\sum_{i,j} X_{ij} = N
\]
Example: N-Queens

○ Formulation 2:
  ○ Variables: $Q_k$
  ○ Domains: $\{1, 2, 3, \ldots N\}$

○ Constraints:
  Implicit: $\forall i, j \text{ non-threatening}(Q_i, Q_j)$
  Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$
    \[\ldots\]
Example: Cryptarithmetic

- **Variables:**
  \[ F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3 \]

- **Domains:**
  \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

- **Constraints:**
  - `alldiff(F, T, U, W, R, O)`
  - `O + O = R + 10 \cdot X_1`
  - \ldots
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
  (or can have a bunch of pairwise inequality constraints)
Varieties of Constraints

○ Varieties of Constraints
  ○ Unary constraints involve a single variable (equivalent to reducing domains), e.g.:
    \[ SA \neq \text{green} \]
  ○ Binary constraints involve pairs of variables, e.g.:
    \[ SA \neq WA \]
  ○ 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
- Circuit layout
- Fault diagnosis
- … lots more!

- Many real-world problems involve real-valued variables…
Solving CSPs
Standard Search Formulation

- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - 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

- We’ll start with the straightforward, naïve approach, then improve it
Search Methods

- What would BFS do?

{WA=g} {WA=r} … {NT=g} …

[Demo: coloring -- dfs]
Search Methods

- What would BFS do?
- What would DFS do?
  - let’s see!
- What problems does naïve search have?

[Demo: coloring -- dfs]
Video of Demo Coloring -- DFS
Backtracking Search
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs

- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - 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"

- Depth-first search with these two improvements is called backtracking search (not the best name)

- Can solve n-queens for n ≈ 25
Backtracking Example
Backtracking Search

```plaintext
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
```

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?
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?
Filtering

Keep track of domains for unassigned variables and cross off bad options
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]
Video of Demo Coloring – Backtracking with Forward Checking
Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

- 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!
  - 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!
Enforcing Arc Consistency in a CSP

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
  \((X_i, X_j) \leftarrow \text{REMOVE-FIRST}(queue)\)
  if \text{REMOVE-INCONSISTENT-VALUES}(X_i, X_j) then
    for each \(X_k\) in \text{NEIGHBORS}[X_i] do
      add \((X_k, X_j)\) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds
removed \leftarrow false
for each \(x\) in \text{DOMAIN}[X_i] do
  if no value \(y\) in \text{DOMAIN}[X_j] allows \((x, y)\) to satisfy the constraint \(X_i \rightarrow X_j\)
    then delete \(x\) from \text{DOMAIN}[X_i]; removed \leftarrow 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?
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!

[Demo: coloring -- arc consistency]
[Demo: coloring -- forward checking]
Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph
Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph
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 kth node.

- Higher k more expensive to compute

- (You need to know the 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)
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
Demo: Coloring -- Backtracking + Forward Checking + Ordering
Summary

- Work with your rubber duck to write down:
  - How we order variables and why
  - How we order values and why
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 \( h(x) = \text{total number of violated constraints} \)
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
Iterative Improvement – n Queens
Iterative Improvement – Coloring
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

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

![Graph showing CPU time vs. critical ratio](image)
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 – turns out trees are easy!

- 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?

- What’s good about it?
Hill Climbing Diagram

- **Objective function**
- **Global maximum**
- **Shoulder**
- **Current state**
- **Local maximum**
- **"Flat" local maximum**
- **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

```plaintext
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
```
Simulated Annealing

- Theoretical guarantee:
  - Stationary distribution: \( p(x) \propto e^{\frac{E(x)}{kT}} \)
  - 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 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?
Bonus (time permitting): 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., $n = 80$, $d = 2$, $c = 20$
  - $2^{80} = 4$ 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(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 $O\left( (d^c)(n-c)\ d^2 \right)$, 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.
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

\[ \{(WA=r, SA=g, NT=b), (WA=b, SA=r, NT=g), \ldots\} \]

\[ \{(NT=r, SA=g, Q=b), (NT=b, SA=g, Q=r), \ldots\} \]

Agree: \((M1, M2) \in \{(WA=g, SA=g, NT=g), (NT=g, SA=g, Q=g)\}, \ldots\)