CIS 471/571 (Fall 2019): Introduction to Artificial Intelligence

Lecture 4: Constraint Satisfaction Problems (Part 1)

Thanh H. Nguyen

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

- Homework 1: Search
  - Deadline: Oct 16\textsuperscript{th}, 2019

- Project 1: Search
  - Deadline: Oct 18\textsuperscript{th}, 2019
Today

- Informed Search
  - Graph Search
- Constraint Satisfaction Problems
  - Backtracking Search
  - Filtering
  - Ordering
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work. Why?
In BFS, for example, we shouldn’t bother expanding some nodes (which, and why?)
Graph Search

- Very simple fix: never expand a state type twice

function $\text{GRAPH-SEARCH}(\text{problem}, \text{fringe})$ returns a solution, or failure

- $\text{closed} \leftarrow \text{an empty set}$
- $\text{fringe} \leftarrow \text{INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)}$

loop do
  
  if $\text{fringe}$ is empty then return failure
  
  $\text{node} \leftarrow \text{REMOVE-FRONT}(\text{fringe})$
  
  if $\text{GOAL-TEST(.problem, STATE[node])}$ then return $\text{node}$

  if $\text{STATE[node]}$ is not in $\text{closed}$ then
    add $\text{STATE[node]}$ to $\text{closed}$
    add $\text{STATE[node]}$ to $\text{fringe}$

end

- Can this wreck completeness? Why or why not?
- How about optimality? Why or why not?
A* Graph Search Gone Wrong

State space graph

Search tree

S (0+2) -> A (1+4) -> B (1+1) -> C (2+1) -> G (5+0)
S (0+2) -> A (1+4) -> C (3+1) -> C (3+1) -> G (6+0)
Consistency of Heuristics

- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]
- Consequences of consistency:
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost(A to C)} + h(C) \]
    \[ f(A) = g(A) + h(A) \leq g(A) + \text{cost(A to C)} + h(C) \leq f(C) \]
  - A* graph search is optimal
Optimality of A* Graph Search
Optimality of A* Graph Search

- Sketch: consider what A* does with a consistent heuristic:
  - Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
  - Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
  - Result: A* graph search is optimal
Optimality

- 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 from relaxed problems
Constraint Satisfaction 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: $\text{WA} \neq \text{NT}$
  - Explicit: $(\text{WA}, \text{NT}) \in \{(\text{red, green}), (\text{red, blue}), \ldots\}$
- Solutions are assignments satisfying all constraints, e.g.:
  $\{\text{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}\}$
Example: N-Queens

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

\[
\begin{align*}
\forall i, j, k \quad (X_{ij}, X_{ik}) &\in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \quad (X_{ij}, X_{kj}) &\in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) &\in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) &\in \{(0, 0), (0, 1), (1, 0)\}
\end{align*}
\]
Example: N-Queens

- Formulation 2:
  - Variables: $Q_k$
  - Domains: $\{1, 2, 3, \ldots N\}$
  - Constraints:
    
    Implicit: $\forall i, j$ non-threatening($Q_i, Q_j$)

    Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$
    
    \ldots
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: Sudoku

- **Variables:**
  - Each (open) square

- **Domains:**
  - \(\{1,2,\ldots,9\}\)

- **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 CSPs and Constraints
Varieties of CSPs

- **Discrete Variables**
  - Finite domains
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end days for each job

- **Continuous variables**
  - E.g., start/end times for Hubble Telescope observations
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

- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
Real-World CSPs

- Scheduling problems: e.g., when can we all meet?
- Timetabling problems: e.g., which class is offered when and where?
- Assignment problems: e.g., who teaches what class
- 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?
- What would DFS do?
- What problems does naïve search have?

[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
  - 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 with 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

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?

- Structure: Can we exploit the problem structure?
Filtering
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
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.

---

Delete from the tail!

- 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 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 \leftarrow false
 for each \(x\) in DOMAIN[X_i] do
   if no value \(y\) in DOMAIN[X_j] allows \((x, y)\) to satisfy the constraint \(X_i \leftarrow X_j\)
     then delete \(x\) from DOMAIN[X_i]; removed \leftarrow true
 return removed

- Runtime: O(n²d³)
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!
K-Consistency
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 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)