Today

• Search Conclusion
• Constraint Satisfaction Problems

A* Review

• A* uses both backward costs $g$ and forward estimate $h$: $f(n) = g(n) + h(n)$
• A* tree search is optimal with admissible heuristics (optimistic future cost estimates)
• Heuristic design is key: relaxed problems can help

A* Graph Search Gone Wrong

Consistency

The story on Consistency:
• Definition: $cost(A \text{ to } C) + h(C) \geq h(A)$
• Consequence in search tree:
  Two nodes along a path: $N_A, N_C$
  $g(N_C) = g(N_A) + cost(A \text{ to } C)$
  $g(N_C) + h(C) \geq g(N_A) + h(A)$
• The $f$ value along a path never decreases
• Non-decreasing $f$ means you’re optimal to every state (not just goals)
Optimality Summary

- Tree search:
  - A* optimal if heuristic is admissible (and non-negative)
  - Uniform Cost Search is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- In general, natural admissible heuristics tend to be consistent
- Remember, costs are always positive in search!

What is Search For?

- Models of the world: single agents, 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:
    \[
    \forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}
    \]
    \[
    \forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}
    \]
    \[
    \forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}
    \]
    \[
    \sum_{i,j} X_{ij} = N
    \]

- Formulation 1.5:
  - Variables: \( Q_k \)
  - Domains: \{11, 12, 13, \ldots, 21, \ldots, NN\}
  - Constraints:
    \[
    \forall i, j \quad \text{non-threatening}(Q_i, Q_j)
    \]
    \[
    \forall i, j \quad (Q_i, Q_j) \in \{(11,23), (11,24), \ldots\}
    \]

- Formulation 2:
  - Variables: \( Q_k \)
  - Domains: \{1, 2, 3, \ldots, N\}
  - Constraints:
    Implicit: \( \forall i, j \quad \text{non-threatening}(Q_i, Q_j) \)
    Explicit: \( (Q_1, Q_2) \in \{(1,3), (1,4), \ldots\} \)

...there's an even better way! What is it?
Example: Map-Coloring

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

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):
  
  \(\text{alldiff}(F, T, U, W, R, O)\)
  
  \(O + O = R + 10 \cdot X_1\)
  
  \(\ldots\)

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

Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra
- An early example of a computation posed as a CSP
- Look at all intersections
- Adjacent intersections impose constraints on each other

Waltz on Simple Scenes

- Assume all objects:
  - Have no shadows or cracks
  - Three-faced vertices
  - "General position": no junctions change with small movements of the eye.
- Then each line on image is one of the following:
  
  - Boundary line (edge of an object) (\(\rightarrow\)) with right hand of arrow denoting "solid" and left hand denoting "space"
  - Interior convex edge (\(\+)
  - Interior concave edge (\(-\))
Legal Junctions
- Only certain junctions are physically possible
- How can we formulate a CSP to label an image?
- Variables: vertices
- Domains: junction labels
- Constraints: both ends of a line should have the same label

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 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 (see cs170 for a bit of this theory)

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

Standard Search Formulation
- Standard search formulation of CSPs (incremental)
- Let's start with the 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
- Simplest CSP ever: two bits, constrained to be equal

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…

Search Methods
- What does BFS do?
- What does DFS do?
- What's the obvious problem?
- What's the slightly-less-obvious problem?
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 \( n = 25 \)

Backtracking Example

Improving Backtracking

- General-purpose ideas can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
  - Can we take advantage of problem structure?

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

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?
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

Forward Checking

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

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 arc consistent:
  - $X \rightarrow Y$ is consistent if for every value $x$ there is some allowed $y$

  - If $X$ loses a value, neighbors of $X$ need to be rechecked!
  - Arc consistency detects failure earlier than forward checking
  - What’s the downside of arc consistency?
  - Can be run as a preprocessor or after each assignment

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)
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., \( n = 80, d = 2, c = 20 \)
  \[ 2^{80} = 4 \text{ billion years at 10 million nodes/sec} \]
  \[ (4)(2^{20}) = 0.4 \text{ seconds at 10 million nodes/sec} \]

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
- For \( i = n : 2 \), apply RemoveInconsistent(Parent\( (X_i) \), \( X_i \))
- For \( i = 1 : n \), assign \( X_i \) consistently with Parent\( (X_i) \)
- Runtime: \( O(n \ d^2) \)

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 logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.

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 \( c \)

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 \( h(n) = \text{total number of violated constraints} \)

Example: 4-Queens

- States: 4 queens in 4 columns (\( 4^4 = 256 \) states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: \( h(n) = \text{number of attacks} \)
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 strucutre
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice

Node Class

class Node:
def __init__(self, state, parent, action, path_cost):
    "Create a search tree Node, derived from a parent by an action."
    "YOUR CODE HERE: Set state, parent, action, path_cost"
    self.state = state
    etc......
    if parent:
        "YOUR CODE HERE: If valid parent, increment depth, path_cost"
def path(self):
    "Create a list of nodes from the root to this node."
    "i.e. Follow parent pointers back up to root"
def expand(self, problem):
    "Return a list of nodes reachable from this node"
    "i.e. Follow parent pointers back up to root"
def graph_search(problem, fringe):
    "Search through the successors of a problem to find a goal."
    return None
def depthFirstSearch(problem):
    "Search the deepest nodes in the search tree first"
    return graph_search(problem, util.Stack())
def breadthFirstSearch(problem):
    "Search the shallowest nodes in the search tree first"
    return graph_search(problem, util.Queue())