Recap Search I

- Agents that plan ahead \(\rightarrow\) formalization: Search
- Search problem:
  - States (configurations of the world)
  - Successor function: a function from states to lists of (state, action, cost) triples; drawn as a graph
  - Start state and goal test

- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)

- Search Algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
Recap Search II

- Tree Search vs. Graph Search
- Priority queue to store fringe: different priority functions → different search method
  - Uninformed Search Methods
    - Depth-First Search
    - Breadth-First Search
    - Uniform-Cost Search
  - Heuristic Search Methods
    - Greedy Search
    - A* Search --- heuristic design!
      - Admissibility: \( h(n) \leq \text{cost of cheapest path to a goal state} \). Ensures when goal node is expanded, no other partial plans on fringe could be extended into a cheaper path to a goal state.
      - Consistency: \( c(n \rightarrow n') \geq h(n) - h(n') \). Ensures when any node \( n \) is expanded during graph search the partial plan that ended in \( n \) is the cheapest way to reach \( n \).
- Time and space complexity, completeness, optimality
- Iterative Deepening (great space complexity!)

### Reflex Agent

- Choose action based on current percept (and maybe memory)
- May have memory or a model of the world’s current state
- Do not consider the future consequences of their actions
- Act on how the world IS
- Can a reflex agent be rational?

### Goal-based Agents

- Plan ahead
- Ask “what if”
- Decisions based on (hypothesized) consequences of actions
- Must have a model of how the world evolves in response to actions
- Act on how the world WOULD BE
Search Problems

- A search problem consists of:
  - A state space
  - A successor function
  - A start state and a goal test
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

Example State Space Graph

Ridiculously tiny search graph for a tiny search problem
Search Trees

A search tree:
- This is a “what if” tree of plans and outcomes
- Start state at the root node
- Children correspond to successors
- Nodes contain states, correspond to PLANS to those states
- For most problems, we can never actually build the whole tree

General Tree Search

function TREE-SEARCH( problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state then return the corresponding solution
  else expand the node and add the resulting nodes to the search tree
end

Important ideas:
- Fringe
- Expansion
- Exploration strategy

Main question: which fringe nodes to explore?
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work. Why?

Graph Search

- Very simple fix: never expand a state twice

```plaintext
function Graph-Search(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test(problem, State[node]) then return node
        if State[node] is not in closed then
            add State[node] to closed
            fringe ← InsertAll(Expand(node, problem), fringe)
        end
    end
```

- Can this wreck completeness? Optimality?
Admissible Heuristics

- A heuristic $h$ is **admissible** (optimistic) if:
  \[ h(n) \leq h^*(n) \]
  where $h^*(n)$ is the true cost to a nearest goal
- Often, admissible heuristics are solutions to *relaxed problems*, with new actions (“some cheating”) available
- Examples:
  - Number of misplaced tiles
  - Sum over all misplaced tiles of Manhattan distances to goal positions

Trivial Heuristics, Dominance

- Dominance: $h_a \geq h_c$ if
  \[ \forall n : h_a(n) \geq h_c(n) \]
- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible
    \[ h(n) = \max(h_a(n), h_b(n)) \]
- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic
Consistency

- Consistency: $c(n, a, n') \geq h(n) - h(n')$

- Required for A* graph search to be optimal
  - It ensures that when a node gets expanded, that node’s final state was reached along the shortest path to reach that final state

- Consistency implies admissibility

A* heuristics --- pacman trying to eat all food pellets

- Consider an algorithm that takes the distance to the closest food pellet, say at $(x,y)$. Then it adds the distance between $(x,y)$ and the closest food pellet to $(x,y)$, and continues this process until no pellets are left, each time calculating the distance from the last pellet. Is this heuristic admissible?

- What if we used the Manhattan distance rather than distance in the maze in the above procedure?
A* heuristics

- A particular procedure to quickly find a perhaps suboptimal solution to the search problem is in general not admissible.
  - It is only admissible if it always finds the optimal solution (but then it is already solving the problem we care about, hence not that interesting as a heuristic).
- A particular procedure to quickly find a perhaps suboptimal solution to a relaxed version of the search problem need not be admissible.
  - It will be admissible if it always finds the optimal solution to the relaxed problem.

Recap CSPs

- 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 (why?, tree or graph search?) with
  - Branching on only one variable per layer in search tree
  - Incremental constraint checks ("Fail fast")
- Heuristics at our points of choice to improve running time:
  - Ordering variables: Minimum Remaining Values and Degree Heuristic
  - Ordering of values: Least Constraining Value
  - Filtering: forward checking, arc consistency
    - Computation of heuristics + pruning of domains might lead to early realization need to backtrack
- Structure: Disconnected and tree-structured CSPs are efficient
  - Non-tree-structured CSP can become tree-structured after some variables have been assigned values
- Iterative improvement: min-conflicts is usually effective in practice
Example: Map-Coloring

- **Variables:** $WA$, $NT$, $Q$, $NSW$, $V$, $SA$, $T$
- **Domain:** $D = \{red, green, blue\}$
- **Constraints:** adjacent regions must have different colors
  - **Implicit:** $WA \neq NT$
  - **Explicit:** $(WA, NT) \in \{(red, green), (red, blue), (green, red), \ldots\}$

- Solutions are assignments satisfying all constraints, e.g.:
  
  \[
  \{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}
  \]

Consistency of An 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.
  
  - If $X$ loses a value, neighbors of $X$ need to be rechecked!
  - Arc consistency detects failure earlier than forward checking, but more work!
  - Can be run as a preprocessor or after each assignment
  - Forward checking = Enforcing consistency of each arc pointing to the new assignment
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 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 $O\left((d^c)(n-c)\right) d^2$, very fast for small $c$

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit
- Why can this be a terrible idea?
  - Complete?
  - Optimal?
- What’s good about it?
Hill Climbing Diagram

- Random restarts?
- Random sideways steps?

Recap Games

- Want algorithms for calculating a strategy (policy) which recommends a move in each state
- Deterministic zero-sum games
  - Minimax
  - Alpha-Beta pruning:
    - speed-up up to: $O(b^d) \rightarrow O(b^{d/2})$
    - exact for root (lower nodes could be approximate)
  - Speed-up (suboptimal): Limited depth and evaluation functions
  - Iterative deepening (can help alpha-beta through ordering!)
- Stochastic games
  - Expectimax
- Non-zero-sum games
Minimax Properties

- Optimal against a perfect player. Otherwise?
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - $O(bm)$
- For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?

Pruning
Evaluation Functions

- With depth-limited search
  - Partial plan is returned
  - Only first move of partial plan is executed
  - When again maximizer’s turn, run a depth-limited search again and repeat

- How deep to search?

Expectimax
Stochastic Two-Player

- E.g. backgammon
- Expectiminimax (!)
  - Environment is an extra player that moves after each agent
  - Chance nodes take expectations, otherwise like minimax

\[
\text{if } \text{state is a } \text{MAX node then} \\
\quad \text{return the highest } \text{EXPECTIMINMAX-VALUE of SUCCESSORS(state)}
\]

\[
\text{if } \text{state is a } \text{MIN node then} \\
\quad \text{return the lowest } \text{EXPECTIMINMAX-VALUE of SUCCESSORS(state)}
\]

\[
\text{if } \text{state is a chance node then} \\
\quad \text{return average of } \text{EXPECTIMINMAX-VALUE of SUCCESSORS(state)}
\]

Non-Zero-Sum Utilities

- Similar to minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own utility and propagate (or back up) nodes from children
  - Can give rise to cooperation and competition dynamically…