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 (space complexity!)

---

Reflex Agent vs. Goal-based Agents

- 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
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

```
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
```

- 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
- 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 \(\rightarrow\) computation of heuristics

- 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:** \( D = \{ \text{red, green, blue} \} \)
- **Constraints:** adjacent regions must have different colors
  - Implicit: \( WA \neq NT \)
  - Explicit: \( (WA, NT) \in \{ (\text{red, green}), (\text{red, blue}), (\text{green, red}) \} \ldots \}

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

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
**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 \( O( (d^c)(n-c)d^2) \), 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?
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

![Expectimax Diagram]
Stochastic Two-Player

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

```
if state is a Max node then
  return the highest EXPECTIMINMAX-VALUE of SUCCESSORS(state)
if state is a Min node then
  return the lowest EXPECTIMINMAX-VALUE of SUCCESSORS(state)
if state is a chance node then
  return average of 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…
Recap MDPs and RL

- **Markov Decision Processes (MDPs)**
  - Formalism \((S, A, T, R, \gamma)\)
  - Solution: policy \(\pi\) which describes action for each state
  - Value Iteration (vs. Expectimax --- VI more efficient through dynamic programming)
  - Policy Evaluation and Policy Iteration

- **Reinforcement Learning (don’t know T and R)**
  - Model-based Learning: estimate \(T\) and \(R\) first
  - Model-free Learning: learn without estimating \(T\) or \(R\)
    - Direct Evaluation [performs policy evaluation]
    - Temporal Difference Learning [performs policy evaluation]
    - Q-Learning [learns optimal state-action value function \(Q^*\)]
    - Policy Search [learns optimal policy from subset of all policies]
  - Exploration
  - Function approximation --- generalization

---

Markov Decision Processes

- An MDP is defined by:
  - A set of states \(s \in S\)
  - A set of actions \(a \in A\)
  - A transition function \(T(s, a, s')\)
    - Prob that \(a\) from \(s\) leads to \(s'\)
    - \(i.e., P(s' | s, a)\)
    - Also called the model
  - A reward function \(R(s, a, s')\)
    - Sometimes just \(R(s)\) or \(R(s')\)
  - A start state (or distribution)
  - Maybe a terminal state

- MDPs are a family of non-deterministic search problems
  - Reinforcement learning: MDPs where we don’t know the transition or reward functions
What is Markov about MDPs?

- “Markov” generally means that given the present state, the future and the past are independent.

- For Markov decision processes, “Markov” means:

\[
P(S_{t+1} = s'|S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0)
\]

\[
= P(S_{t+1} = s'|S_t = s_t, A_t = a_t)
\]

- Can make this happen by proper choice of state space.

Value Iteration

- Idea:
  - \(V_i'(s)\): the expected discounted sum of rewards accumulated when starting from state \(s\) and acting optimally for a horizon of \(i\) time steps.

- Value iteration:
  - Start with \(V_0'(s) = 0\), which we know is right (why?)
  - Given \(V_i'\), calculate the values for all states for horizon \(i+1\):

\[
V_{i+1}'(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i'(s') \right]
\]

  - This is called a value update or Bellman update.
  - Repeat until convergence.

- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values.
  - Policy may converge long before values do.
  - At convergence, we have found the optimal value function \(V^*\) for the discounted infinite horizon problem, which satisfies the Bellman equations:

\[
\forall s \in S: \quad V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]
\]
Complete Procedure

- 1. Run value iteration (off-line)
  \( \rightarrow \) Returns \( V \), which (assuming sufficiently many iterations is a good approximation of \( V^* \))

- 2. Agent acts. At time \( t \) the agent is in state \( s_t \) and takes the action \( a_t \):

\[
\text{arg} \max_a \sum_{s'} T(s_t, a, s') [R(s_t, a, s') + \gamma V^*(s')] 
\]

Policy Iteration

- Policy evaluation: with fixed current policy \( \pi \), find values with simplified Bellman updates:
  - Iterate until values converge

\[
V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} T(s, \pi_k(s), s') \left[ R(s, \pi_k(s), s') + \gamma V_{i}^{\pi_k}(s') \right]
\]

- Policy improvement: with fixed utilities, find the best action according to one-step look-ahead

\[
\pi_{k+1}(s) = \text{arg} \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_{\pi_k}(s') \right]
\]

- Will converge (policy will not change) and resulting policy optimal
Sample-Based Policy Evaluation?

\[ V_{i+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_i^\pi(s')] \]

- Who needs T and R? Approximate the expectation with samples (drawn from T!)

\[ \text{sample}_1 = R(s, \pi(s), s'_1) + \gamma V^\pi_i(s'_1) \]
\[ \text{sample}_2 = R(s, \pi(s), s'_2) + \gamma V^\pi_i(s'_2) \]
\[ \vdots \]
\[ \text{sample}_k = R(s, \pi(s), s'_k) + \gamma V^\pi_i(s'_k) \]

\[ V_{i+1}^\pi(s) \leftarrow \frac{1}{k} \sum_i \text{sample}_i \]

Temporal-Difference Learning

- Big idea: learn from every experience!
  - Update \( V(s) \) each time we experience \((s, a, s', r)\)
  - Likely \( s' \) will contribute updates more often

- Temporal difference learning
  - Policy still fixed!
  - Move values toward value of whatever successor occurs: running average!

**Sample of \( V(s) \):**

\[ \text{sample} = R(s, \pi(s), s') + \gamma V^\pi(s') \]

**Update to \( V(s) \):**

\[ V^\pi(s) \leftarrow (1 - \alpha) V^\pi(s) + \alpha \text{sample} \]

**Same update:**

\[ V^\pi(s) \leftarrow V^\pi(s) + \alpha (\text{sample} - V^\pi(s)) \]
Exponential Moving Average

- Exponential moving average
  - Makes recent samples more important
  \[
  \bar{x}_n = \frac{x_n + (1 - \alpha) \cdot x_{n-1} + (1 - \alpha)^2 \cdot x_{n-2} + \ldots}{1 + (1 - \alpha) + (1 - \alpha)^2 + \ldots}
  \]
  - Forgets about the past (distant past values were wrong anyway)
  - Easy to compute from the running average
  \[
  \bar{x}_n = (1 - \alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n
  \]
  - Decreasing learning rate can give converging averages

Detour: Q-Value Iteration

- Value iteration: find successive approx optimal values
  - Start with \( V_0(s) = 0 \), which we know is right (why?)
  - Given \( V_i \), calculate the values for all states for depth \( i+1 \):
    \[
    V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i(s') \right]
    \]
  - But Q-values are more useful!
  - Start with \( Q_0(s,a) = 0 \), which we know is right (why?)
  - Given \( Q_i \), calculate the q-values for all q-states for depth \( i+1 \):
    \[
    Q_{i+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_i(s', a') \right]
    \]
Q-Learning

- **Learn Q**(s,a) values
  - Receive a sample (s,a,s',r)
  - Consider your new sample estimate:
    \[ Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \]
    \[ \text{sample} = R(s, a, s') + \gamma \max_{a'} Q(s', a') \]
  - Incorporate the new estimate into a running average:
    \[ Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha [\text{sample}] \]
- Amazing result: Q-learning converges to optimal policy
  - If you explore enough
  - If you make the learning rate small enough but not decrease it too quickly!
- **Neat property: off-policy learning**
  - learn optimal policy without following it

Exploration Functions

- **Simplest: random actions (ε greedy)**
  - Every time step, flip a coin
  - With probability ε, act randomly
  - With probability 1-ε, act according to current policy
- Problems with random actions?
  - You do explore the space, but keep thrashing around once learning is done
  - One solution: lower ε over time
- **Exploration functions**
  - Explore areas whose badness is not (yet) established
  - Take a value estimate and a count, and returns an optimistic utility, e.g. \( f(u, n) = u + k/n \) (exact form not important)
  - That becomes:
    \[ Q_{i+1}(s, a) \leftarrow (1 - \alpha) Q_i(s, a) + \alpha \left( R(s, a, s') + \gamma \max_{a'} Q_i(s', a') \right) \]
    \[ Q_{i+1}(s, a) \leftarrow (1 - \alpha) Q_i(s, a) + \alpha \left( R(s, a, s') + \gamma \max_{a'} f(Q_i(s', a'), N(s', a')) \right) \]
Feature-Based Representations

- **Solution**: describe a state using a vector of features
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - $1 / (\text{dist to dot})^2$
    - Is Pacman in a tunnel? (0/1)
    - ...... etc.
  - Can also describe a q-state $(s, a)$ with features (e.g. action moves closer to food)

Linear Feature Functions

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:

  $$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

  $$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a)$$

- Advantage: our experience is summed up in a few powerful numbers
- Disadvantage: states may share features but be very different in value!
Policy Search

- Problem: often the feature-based policies that work well aren’t the ones that approximate V/Q best
- Solution: learn the policy that maximizes rewards rather than the value that predicts rewards
- This is the idea behind policy search, such as what controlled the upside-down helicopter
- Simplest policy search:
  - Start with an initial linear value function or Q-function
  - Nudge each feature weight up and down and see if your policy is better than before
- Problems:
  - How do we tell the policy got better?
  - Need to run many sample episodes!
  - If there are a lot of features, this can be impractical

Degree 15 polynomial

Overfitting
Part II: Probabilistic Reasoning

- Probability
  - Random Variables
  - Joint and Marginal Distributions
  - Conditional Distribution
  - Inference by Enumeration
  - Product Rule, Chain Rule, Bayes’ Rule
  - Independence

- Distributions over LARGE Numbers of Random Variables
  - Representation
  - Inference [not yet covered for large numbers of random variables]

Probability recap

- Conditional probability
  \[ P(x|y) = \frac{P(x,y)}{P(y)} \]

- Product rule
  \[ P(x,y) = P(x|y)P(y) \]

- Chain rule
  \[ P(X_1,X_2,\ldots,X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1,X_2)\ldots \]

- X, Y independent iff:
  \[ \forall x, y : P(x,y) = P(x)P(y) \]
  equivalently, iff:
  \[ \forall x, y : P(x|y) = P(x) \]
  equivalently, iff:
  \[ \forall x, y : P(y|x) = P(y) \]

- X and Y are conditionally independent given Z iff:
  \[ \forall x, y, z : P(x,y|z) = P(x|z)P(y|z) \]
  equivalently, iff:
  \[ \forall x, y, z : P(x|y,z) = P(x|z) \]
  equivalently, iff:
  \[ \forall x, y, z : P(y|x,z) = P(y|z) \]
Inference by Enumeration

- $P(\text{sun})$?

- $P(\text{sun} \mid \text{winter})$?

- $P(\text{sun} \mid \text{winter}, \text{hot})$?

- $P(s, t, w)$?

<table>
<thead>
<tr>
<th>S</th>
<th>T</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>summer</td>
<td>hot</td>
<td>sun</td>
<td>0.30</td>
</tr>
<tr>
<td>summer</td>
<td>hot</td>
<td>rain</td>
<td>0.05</td>
</tr>
<tr>
<td>summer</td>
<td>cold</td>
<td>sun</td>
<td>0.10</td>
</tr>
<tr>
<td>summer</td>
<td>cold</td>
<td>rain</td>
<td>0.05</td>
</tr>
<tr>
<td>winter</td>
<td>hot</td>
<td>sun</td>
<td>0.10</td>
</tr>
<tr>
<td>winter</td>
<td>hot</td>
<td>rain</td>
<td>0.05</td>
</tr>
<tr>
<td>winter</td>
<td>cold</td>
<td>sun</td>
<td>0.15</td>
</tr>
<tr>
<td>winter</td>
<td>cold</td>
<td>rain</td>
<td>0.20</td>
</tr>
</tbody>
</table>

Chain Rule $\Rightarrow$ Bayes net

- Chain rule: can always write any joint distribution as an incremental product of conditional distributions

$$P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)$$

$$P(x_1, x_2, \ldots x_n) = \prod_i P(x_i|x_1 \ldots x_{i-1})$$

- Bayes nets: make conditional independence assumptions of the form:

$$P(x_i|x_1 \ldots x_{i-1}) = P(x_i|\text{parents}(X_i))$$

giving us:

$$P(x_1, x_2, \ldots x_n) = \prod_{i=1}^{n} P(x_i|\text{parents}(X_i))$$
Probabilities in BNs

- Bayes’ nets implicitly encode joint distributions
  - As a product of local conditional distributions
  - To see what probability a BN gives to a full assignment, multiply all the relevant conditionals together:

\[ P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|\text{parents}(X_i)) \]

- Example:
\[ P(+\text{cavity}, +\text{catch}, -\text{toothache}) \]

- This lets us reconstruct any entry of the full joint
- Not every BN can represent every joint distribution
  - The topology enforces certain conditional independencies

Example: Alarm Network

| B     | P(B)   | A | J | P(J|A) | A | M | P(M|A) | P(A|B,E) |
|-------|--------|---|---|-------|---|---|-------|---------|
| +b    | 0.001  | +a| +j | 0.9   | +a| +m| 0.7   | +b +e +a | 0.95    |
| ~b    | 0.999  | +a| ~j | 0.1   | +a| ~m| 0.3   | +b +e ~a | 0.05    |
|       |        | ~a| +j | 0.05  | ~a| +m| 0.01  | +b ~e ~a | 0.06    |
|       |        | ~a| ~j | 0.95  | ~a| ~m| 0.99  | ~b +e ~a | 0.29    |
|       |        |   |    |       |   |    |       | ~b ~e +a | 0.71    |
|       |        |   |    |       |   |    |       | ~b ~e ~a | 0.001   |
|       |        |   |    |       |   |    |       | ~b ~e ~a | 0.999   |