Review

Homework 1

PEAS: Performance Measure, Environment, Actuators, Sensors

- **Performance Measure** – Criteria for evaluating the agent's success.
- **Environment** – The surroundings in which the agent operates.
- **Actuators** – Components that allow the agent to take actions.
- **Sensors** – Components that enable the agent to perceive its environment.

Formulating Search Problems

1. **Problem**: Slide tiles on a 3x3 grid to match a target configuration.
2. **State Space**: All possible arrangements of 8 numbers and a blank tile (e.g., "1 2 3, 4 5 6, 7 8 _").
3. **Initial State**: A specific scrambled board (e.g., "1 2 3, 4 8 5, 7 6 _").
4. **Goal State**: The solved board (e.g., "1 2 3, 4 5 6, 7 8 _").
5. **Actions**: Move the blank tile up, down, left, or right (if legal).
6. **Transition Model**: If blank is at (2,2), "move up" → blank swaps with (1,2).
7. **Cost Function**: Each move costs 1 (or skip for BFS/DFS, since cost isn't considered).

Search Trees

Generation

- Avoid duplicate choices at the same level.
- Search trees can't have cycles but may be infinite if the state-space graph has cycles.
- A finite, acyclic state-space graph ensures a finite search tree.

GRAPH Search

Each of these algorithms create a search tree from the given state-space graph.

Tie-breakers needed to avoid multiple solutions in terms of state-expansion order.

BFS

- **Queue-based** traversal.
- **No heuristics** unless explicitly stated.
- **Stops when the goal is expanded**.

DFS

- **Stack-based** traversal.
- **No heuristics** unless explicitly stated.
- **Stops when the goal is expanded**.

UCS (Uniform Cost Search)

- **No heuristics** unless explicitly stated.
- **No revisits**: Use a **visited list** to track explored nodes.
- **Track total cost**: Sum the path costs from the start state to each node.
  - Example: If S→A = 5 and A→B = 6, then S→A→B = 11. Write 5 at A and 11 at B.
- **Expand the cheapest node** on the **frontier** (unexpanded nodes).
- **Stops when the goal state is expanded**.
- **Guaranteed minimum-cost path** if costs are **non-negative**.

A*

- **Uses heuristics**.
- **Admissibility** ensures the shortest path in graphs with **non-negative edge costs**.
- **Consistency** guarantees **admissibility** and prevents **revisiting/re-expanding**, ensuring **optimality**.
- **Extends UCS** by considering both real cost and heuristic cost:

$$f(n) = g(n) + h(n)$$

- **Tracking visited nodes**: Store the A* score $f(n)$.
  - In UCS, visited nodes could be ignored.
  - In A*, however, revisit a node **only if** a lower $f(n)$ score is found.

Definitions

- **Admissibility**: A heuristic is **admissible** if it **never overestimates** the cost to the goal. That is, the true cost is always **at least** the heuristic:

$$h(n) \leq h^*(n)$$

where $h^*(n)$ is the actual cheapest cost from n to G.
- **Consistency**: A heuristic is **consistent** if it satisfies the **triangle inequality**:

$$h(n) \leq h(m) + c(n, m)$$

where $c(n, m)$ is the cost of moving from n to m.
- A **consistent** heuristic is always **admissible**, but an **admissible** heuristic is not necessarily **consistent**.
- **State space** is the set of all possible states the agent can reach in the grid. Its size depends on what information is included in *each* state.
  - For a grid with $M \times N$ cells, if the state is defined by the agent's **position** (x, y) and **facing direction** (left, right, up, or down), the total number of states is:

$$M \times N \times 4$$

    - There are $M \times N$ states for each direction (left, right, up, and down).
    - Since there are 4 possible directions, the total state space is:

$$(M \times N) + (M \times N) + (M \times N) + (M \times N) = M \times N \times 4$$

Homework 2

Minimax with Alpha-Beta Pruning

- **Alpha**: Best explored value for the maximizer.
- **Beta**: Best explored value for the minimizer.
- Maximizer nodes update **alpha**.
- Minimizer nodes update **beta**.

Initial Values

Minimizer and maximizer start with worst-case values:

- **Maximizer**: $v = -\infty$, so **alpha** = $-\infty$.
- **Minimizer**: $v = \infty$, so **beta** = $+\infty$.
- If **alpha** is unknown, assume $-\infty$.
- If **beta** is unknown, assume $+\infty$.

Recursive Formula

- Set v to the parent node's value (or its initial default).
- **Minimizer**: If $v' < v$, update v with v'.
- **Maximizer**: If $v' > v$, update v with v'.
- **Pruning:**
  - If the node is a **maximizer** and $v' >$ beta → **prune**.
  - If the node is a **minimizer** and $v' <$ alpha → **prune**.

Difference Between Expectiminimax and Minimax

Feature	Minimax	Expectiminimax
Used In	**Deterministic** games (e.g., Chess, Tic-Tac-Toe)	**Stochastic (randomized) games** (e.g., Backgammon, Monopoly)
Types of Nodes	**Max** (agent's turn) and **Min** (opponent's turn)	**Max**, **Min**, and **Chance** (random events)
Opponent Modeling	Assumes opponent plays optimally	Accounts for randomness using **probability**
Decision Process	Chooses the move that maximizes the worst-case outcome	Computes an **expected value** for chance nodes
Tree Structure	Alternates between **Max and Min levels**	Alternates between **Max, Min, and Chance levels**
Evaluation	Uses a utility function directly	Uses a weighted average of possible outcomes at chance nodes

For expectiminimax:

$$\text{Weighted Probability} = \frac{\sum_{i=1}^{n} P(n_i) \cdot V(n_i)}{\text{Number of Chance Nodes}}$$

Homework 3

Formulas

$$V_k^{\pi_i}(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_{k-1}^{\pi_i}(s') \right]$$

Here are the fundamental equations for **Q-learning**:

Q-Value Update Equation

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Where:

- $Q(s,a)$ is the current Q-value for state s and action a.
- α is the learning rate ($0 < \alpha \leq 1$).
- r is the reward received after taking action a in state s.
- γ is the discount factor ($0 \leq \gamma \leq 1$).
- s' is the next state after taking action a.
- $\max_{a'} Q(s',a')$ is the highest Q-value of the next state.

Greedy Action Selection (Exploitation)

$$a^* = \arg\max_a Q(s,a)$$

This selects the action a^* with the highest Q-value for state s.

Epsilon-Greedy Policy (Exploration vs. Exploitation)

With probability ϵ, choose a random action; otherwise, choose the best action:

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q(s,a), & \text{with probability } 1-\epsilon \end{cases}$$

Feature-based Representation

Q-learning updates the weights of features rather than maintaining a table of $Q(s,a)$ values for each state-action pair. This is useful when the state space is large or continuous.

Q-Value Approximation

Instead of using a table, the Q-value is represented as a **linear function** of features:

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \cdots + w_n f_n(s,a) = \sum_i w_i f_i(s,a)$$

where:

- $f_i(s, a)$ are the features describing the state-action pair.
- w_i are the corresponding weights.

Weight Update Rule

The weights are updated using a gradient descent step:

$$w_i \leftarrow w_i + \alpha \delta f_i(s, a)$$

where:

- α is the learning rate.
- δ is the **TD error**, given by:

$$\delta = \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$