Advanced AI – Multi-Agent Search Assignment

Pac-Man: Minimax, Alpha-Beta, Expectimax, and (Mis)Alignment

Course: Advanced Artificial Intelligence

January 2026

General Instructions

This assignment contains **both written and programming components**. You will implement multi-agent search agents for classic Pac-Man and reflect on how objective design can lead to unintended behavior.

You should clearly distinguish between:

- **Search problem setup** (states, legal actions, terminal conditions), and
- **Search algorithms** (minimax, alpha-beta pruning, expectimax) and their assumptions.

Unless otherwise stated, you will modify only `submission.py`.

Background: Multi-Agent Pac-Man

Pac-Man moves in a maze and tries to eat all food pellets while avoiding ghosts. In this assignment, Pac-Man is the **maximizing** agent, and ghosts are either **minimizers** (adversarial) or **random** (stochastic), depending on the problem.

There are $n + 1$ agents $a_0, \ldots, a_n$, where a_0 is Pac-Man and the rest are ghosts. A single search **depth** consists of all $n + 1$ agents moving once (in order), so depth 2 search involves Pac-Man and each ghost moving two times.

Warmup (Not for Credit)

Run a game of classic Pac-Man:

- `uv run pacman.py`

Then run the provided reflex agent baseline:

- `uv run pacman.py -p ReflexAgent`

Inspect `ReflexAgent` in `submission.py` and make sure you understand how it uses the `GameState` API.

Problem 1: Minimax

Written

1.a [**Written**] Write the recurrence for $V_{\text{minmax}}(s, d)$ as a **piecewise** mathematical definition for depth-limited minimax with multiple ghosts. Your recurrence should be written in terms of the following functions:

- IsEnd(s): whether s is terminal,
- Utility(s): terminal utility,
- Eval(s): depth-limited evaluation function,
- Player(s): which agent moves in state s,
- Actions(s): legal actions from s,
- Succ(s, a): successor state.

You may use n in your solution, but do *not* use $d_{\max}$.

Programming

1.b [**Programming**] Implement `MinimaxAgent` in `submission.py` for an arbitrary number of ghosts. Your implementation should:

- Expand the game tree to depth `self.depth`.
- Use `self.evaluation_function` at depth-limited leaves.
- Use terminal utility equal to `GameState.get_score()`.
- Break ties between best actions arbitrarily.

Problem 2: Alpha-Beta Pruning

Programming

2.a [**Programming**] Implement `AlphaBetaAgent` in `submission.py` using alpha-beta pruning. Your pruning logic must correctly handle **multiple** minimizing ghost layers between Pac-Man layers.

You should observe a meaningful speed-up relative to minimax at the same depth. For example:

- `uv run pacman.py -p AlphaBetaAgent -a depth=3`

Problem 3: Expectimax

Written

3.a [**Written**] Assume each ghost follows a uniform random policy over its legal actions. Write the recurrence for $V_{\text{exptmax}}(s, d)$, the **maximum expected utility** for Pac-Man. Your recurrence should resemble your answer to Problem 1(a), and should be written in terms of the same functions: IsEnd, Utility, Eval, Player, Actions, Succ.

Programming

3.b [**Programming**] Implement `ExpectimaxAgent` in `submission.py`. Pac-Man should maximize expected utility and assume each ghost selects uniformly at random among its legal moves.

You may validate behavior on:

- `uv run pacman.py -p ExpectimaxAgent -l trapped_classic -a depth=3`

Problem 4 (Extra Credit): Evaluation Function

Notes:

- If you attempt extra credit, submit the same `submission.py` to both the standard and extra-credit submissions.

Programming

4.a [**Programming**] Implement a stronger state evaluation function `better_evaluation_function`. The evaluation function should evaluate *states* (not actions). You may use any features available from the `GameState` and any helper utilities provided.

Target behavior: with depth 2 search, your agent should clear `small_classic` with two random ghosts more than half the time while running at a reasonable rate, e.g.:

- `uv run pacman.py -l small_classic -p ExpectimaxAgent -a eval_fn=better -q -n 20`

Written

4.b [**Written**] Clearly describe your evaluation function. Discuss:

- The high-level motivation and the main features you used,
- What else you tried, what worked, and what did not,
- If you place in the top 3 on the leaderboard, whether you consent to having your approach shared after grades are released.

Problem 5: AI (Mis)Alignment and Reward Hacking

Run (several times) the following two agents on `trapped_classic`:

- `uv run pacman.py -p MinimaxAgent -l trapped_classic -a depth=3`
- `uv run pacman.py -p ExpectimaxAgent -l trapped_classic -a depth=3`

You should observe that minimax tends to rush toward the closest ghost, while expectimax can sometimes collect all pellets and win.

Written

5.a [**Written**] Describe why the behavior of minimax and expectimax agents differs in this environment. In particular:

- One sentence on why minimax rushes the closest ghost (instead of the further ghost),
- One sentence on why expectimax does not (sometimes),
- Explicitly state the key assumptions made by minimax that lead to this phenomenon.

5.b [**Written**] Suggest one potential change to the default state evaluation function Eval(s) (i.e., `score_evaluation_function`) and/or the default utility function Utility(s) (i.e., final game score) that would prevent the minimax agent from dying instantly on `trapped_classic` and behave more like the expectimax agent. Provide 1–2 sentences explaining why your change would work. No code is required.

5.c [**Written**] Give another realistic example *outside of Pac-Man* where a designer specifies an objective, but:

- the objective is susceptible to **reward hacking**, and/or
- the resulting optimized behavior causes **negative side effects**.

In 2–5 sentences, describe the scenario and clearly state whether it is reward hacking, negative side effects, or both, with a brief justification.

Submission Guidelines

You may choose **one** of the following submission formats:

Option 1 (Recommended)

- A **PDF report** generated from LaTeX containing all written answers and figures.
- Your **Python source file** `submission.py` containing your implementations.

Option 2

- A single **Jupyter Notebook (.ipynb)** containing implementations, written explanations, and results.

We recommend Option 1 for readability and cleaner separation of code and write-up.

Go Pac-Man Go!