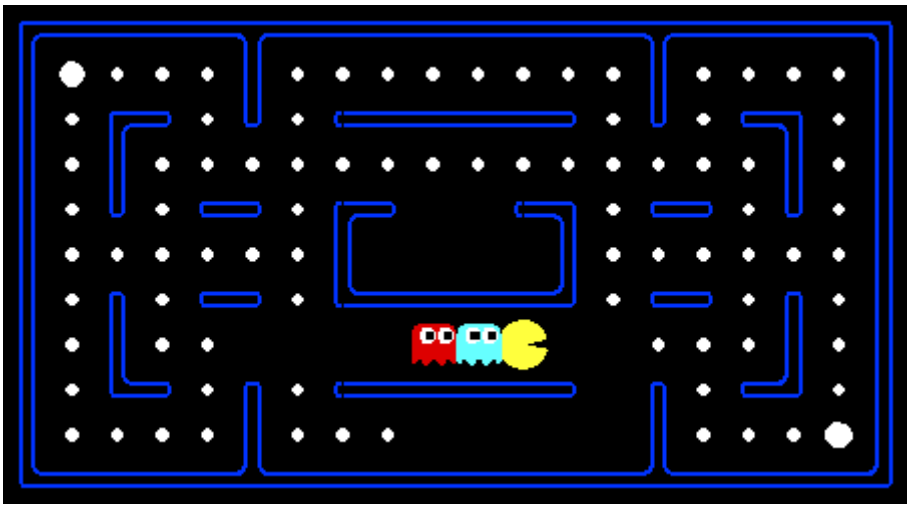
Spring 2026
Introduction to Artificial Intelligence

Homework 2: Multi-Agent Search

Due Date: 2026/4/14 (Tue.) 12:00 PM (Noon)

Last update: 4/1 9:00



Introduction

For those of you not familiar with Pac-Man, it's a game where pacman (the yellow circle with a mouth) moves around in a maze and tries to eat as many food pellets (the small white dots) as possible, while avoiding the ghosts (the other two agents with eyes). If pacman eats all the food in a maze, it wins. The big white dots are capsules, which give pacman power to eat ghosts in a limited time.

In this assignment, you will design agents for the classic version of Pac-Man by implementing three adversarial search algorithms, i.e., minimax search, alpha-beta pruning and expectimax search.

Welcome to Multi-Agent Pac-Man

The code base of Pac-Man was developed by UC Berkeley.
(https://inst.eecs.berkeley.edu/~cs188/sp21/project2/)

You can only run the code base on a local machine. Google Colab cannot execute it because it has a GUI. Please install python 3 on your own machine and be familiar with the code with CLI.
The tested environment includes:
- Python 3.10.19

First, play a game of classic Pac-Man by running the following command:

```
python pacman.py
```

and using the arrow keys to move.
Next, run the given ReflexAgent in multiAgents.py:

```
python pacman.py -p ReflexAgent
```

Note that the ReflexAgent performs poorly even on simple layouts:

```
python pacman.py -p ReflexAgent -l testClassic
```

Other Options:

1. Default ghosts are random. You can also play for fun with slightly smarter directional ghosts using -g DirectionalGhost.
2. You can play multiple games in one command with -n.
3. You can turn off graphics with -q to run games quickly.
4. Use -h to know more options.

The code base contains the following files:

Files you will edit:	
multiAgents.py	All of your multi-agent search agents will reside here.
Files/Folders you might want to look at:	
pacman.py	The main file that runs Pac-Man games. This file also describes a pacman GameState type, which you will use extensively in this assignment.
game.py	The logic behind how the Pac-Man world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
util.py	Useful data structures for implementing search algorithms. You don't need to use these for this assignment, but may find other functions defined here to be useful.
layouts/	Contains several different map layouts. You may refer to these files to see the available game boards.
Other files you might want to look at, if you are interested in the details of	

this game.

ReflexAgent

1. getAction() — Select the Best Action
    a. Get all legal actions from the current state
    `legalMoves = gameState.getLegalActions()`
    b. Compute a score for each action using the evaluation function
    `scores = [self.evaluationFunction(gameState, action) for action in legalMoves]`
    c. Find the highest score
    `bestScore = max(scores)`
    d. If multiple actions have the same best score, randomly choose one
    `chosenIndex = random.choice(bestIndices)`
    e. Return the selected action

2. evaluationFunction() — Evaluate Each Action

    a. Generate the next state after taking the action
    `childGameState = currentGameState.getPacmanNextState(action)`

    b. Extract information from the next state

        i. Pacman's new position
        newPos = childGameState.getPacmanPosition()
        ii. Remaining food locations
        newFood = childGameState.getFood()
        iii. Ghost positions
        newGhostStates = childGameState.getGhostStates()
        iv. Score difference
        scoreDiff = childGameState.getScore() - currentGameState.getScore()
        v. Distance to the nearest food
        nearestFoodDistance = min([manhattanDistance(pos, food) for food in currentGameState.getFood().asList()])

    c. Scoring rules implemented in the code

        i. Ghost distance ≤ 1 OR action == STOP → return 0
        ii. Score increases (food eaten) → return 8
        iii. Move closer to food → return 4
        iv. Continue the same direction → return 2
        v. Otherwise → return 1

Note

These are commonly used functions in the Pacman environment.
The ReflexAgent shows how to extract useful information from the game state (such as food distance, ghost distance, and score changes) to evaluate actions.

You can refer to this implementation when building more advanced agents in the later parts of the assignment.

Evaluation and Final Score

This homework has three grading components:

1. **Implementation** (40%)
2. **Report** (40%)
3. **Oral test** (20%)

Final score is computed as:

```
final score = Implementation + Report + Oral test
```

Autograding

In this assignment, TAs will use an autograder to grade your implementation. The autograder has been included in the code base. You can use the following command to test by yourself.

```
python autograder.py
```

The autograder will check your code to determine whether it explores the correct number of game states. This will show what your implementation does on some simulated trees and Pac-Man games. After that, it will show the score you get.

Using the autograder to debug is recommended and will help you to find bugs quickly. To test and debug your code for one particular part, run the following command:

```
python autograder.py -q part1
```

To run it without graphics, use the following command:

```
python autograder.py -q part1 --no-graphics
```

Please do not change the names of any provided functions or classes within the code for technical correctness, or you will wreak havoc on the autograder.

Implementation (40%)

1. Please modify the codes in multiAgents.py between # Begin your code and # End your code. In addition, do not import other packages.

2. All agents you implement should work with any number of ghosts. In particular, your search tree will have multiple min/chance layers (one for each ghost) for every max layer.

3. Your code should also expand the game tree to arbitrary depth with the supplied self.depth. A single level of the search is considered to be one pacman move and all the ghosts' responses, so depth 2 search will involve pacman and each ghost moving twice.

4. Your code should score the leaves of your search tree with the provided self.evaluationFunction, which defaults to scoreEvaluationFunction.

5. All agents you implement extend MultiAgentSearchAgent, which gives access to self.depth and self.evaluationFunction. Those two variables are populated in response to command line options.

Part 1: Minimax Search (10%)

- Write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py.
- The actual ghosts operating in the environment may act partially randomly, but the minimax algorithm assumes the worst.

Question1:

In the initial state of the `minimaxClassic` layout, the Minimax evaluation values are a stable 9, 8, and 7 at search depths 1, 2, and 3, respectively. However, why does the evaluation value suddenly plummet to a dismal -492 at depth 4? What specific event is the algorithm foreseeing at depth 4 that causes this drastic drop?
Please execute the following command.

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

Part 2: Alpha-Beta Pruning (10%)

- Implement a new agent that uses alpha-beta pruning to efficiently explore the minimax tree in AlphaBetaAgent class in multiAgents.py.
- Again, your algorithm will be slightly more general than the pseudo-code discussed in the lecture. This part of the homework is to extend the alpha-beta pruning logic appropriately to multiple MIN agents.
- You must not prune on equality in order to match the set of states explored by our autograder. The pseudo-code below represents the algorithm you should implement for this part.

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, v)
    return v
```

- The autograder will check your code to determine whether it explores the correct number of game states. It is important that you perform alpha-beta pruning without reordering children. In other words, child states should always be processed in the order returned by GameState.getLegalActions. Again, do not call GameState.getNextState more than necessary.

Question2:

Explain the differences between Alpha-Beta pruning and standard Minimax when performing a depth 3 search, and discuss why these differences occur.

Please execute the following command.

python pacman.py -p MinimaxAgent -a depth=3 -l minimaxClassic -q -n 100
python pacman.py -p AlphaBetaAgent -a depth=3 -l minimaxClassic -q -n 100

Part 3: Expectimax Search (10%)

- Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case.
- In this part, you will implement the ExpectimaxAgent class in multiAgents.py, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.
- Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. Rather than taking the MIN over all ghost actions, expectimax agent will take the expectation according to your agent's model of how the ghosts act. To simplify your code, please assume you will only be running against an adversary that chooses among its legal actions uniformly at random.

- To see how the ExpectimaxAgent behaves in Pac-Man, run the following command:

  ```
  python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
  ```

  You should observe a more cavalier approach in close quarters with ghosts.

Question3:

Why does the `ExpectimaxAgent` achieve an approximate 50% win rate on the `trappedClassic` layout, while the `AlphaBetaAgent` consistently loses 100% of the time under the exact same conditions?

Please execute the following command.

```
python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

Part 4: Evaluation Function (10%)

- Write a better evaluation function for pacman in the provided function betterEvaluationFunction in multiAgents.py. The evaluation function should evaluate only states not including actions.

Observations:

- With depth 2 search, your evaluation function should clear the smallClassic layout with one random ghost more than half the time and still run at a reasonable rate.

Question4:

What specific factors are incorporated into your custom Evaluation Function, why are they fundamentally important for guiding the agent, and how do they collectively contribute to a win rate of more than half the time?

Please execute the following command.

```
python pacman.py -p ExpectimaxAgent  -l smallClassic -q -n 10
python pacman.py -p ExpectimaxAgent -a evalFn=better -l smallClassic -q -n 10
```

Grading:

- **Test set in autograder.py** (10%):
  The autograder will run your agent on the smallClassic layout 10 times. We will assign points to your evaluation function in the following way:
  - If you win at least once without timing out the autograder, you get **1** point. Any agent not satisfying these criteria will receive 0 points.
  - **+1** for winning at least 4 times, **+2** for winning at least 7 times, **+3** for winning all 10 times.
  - **+2** for an average score of at least 600, **+4** for an average score of at least 1200(including scores on lost games)
  - **+1** for no timeout at least 5 times, **+2** for no timeout all 10 times.
- The autograder will be run on the same machine with --no-graphics.

Report (40%)

- You should write your report following the report template
- The report can be written in either Chinese or English. (Using English for technical terms is recommended.)
- Please save the report as a **.pdf** file. (font size: 12)
- For part 1 ~ 4, please take some screenshots of your code and explain how you implement the codes in detail. In addition, please answer the following questions for each corresponding part in your report.
    - Explain your implementation 1-4 (4%*4)
    - Questions 1-4 (5%*4)
- For Question 5, please refer to the provided report template for the detailed question description and answer it accordingly in your report. (4%)

QA Page

If you have any questions about this homework, please ask them on the following Notion page. We will answer them as soon as possible. Additionally, we encourage you can answer other students' questions if you can.

QA Page link

Submission

Due Date: 2026/4/14 (Tue.) 12:00 PM (Noon).

Please submit your code and report as **two separate files**.

1. Compress your `multiAgents.py` into `STUDENTID_hw2.zip`.
2. Save your report as `STUDENTID_hw2_report.pdf`.

The file structure should look like:

```
📂{student_id}_hw2.zip
  ┗📜multiAgents.py
📜{student_id}_hw2_report.pdf
```

Wrong submission format leads to 0 points.

Late Submission Policy

Late submissions receive 0 points.