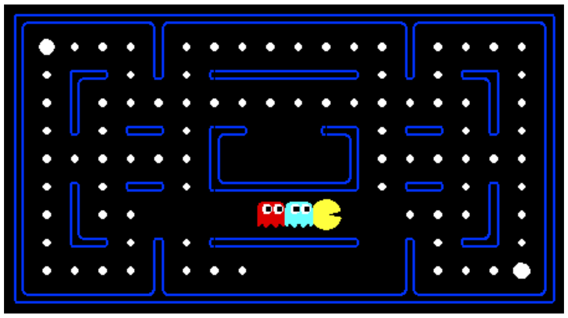
Project – Part I: Multi-Agent Search*

Deadline: Sunday 16/11/2025



Pacman chased by ghosts: Minimax, Expectimax, Evaluation

Introduction

You will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement the minimax search. Each group can have **at most four** students.

The code for this project contains the following files, available as a zip archive.

Files you'll edit:

multiAgents.py	Where all of your multi-agent search agents will reside.

Files you might want to look at:

pacman.py	The main file that runs Pacman games. This file also describes a Pacman GameState type, which you will use extensively in this project.
game.py	The logic behind how the Pacman 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 project, but may find other functions defined here to be useful.

Files to Edit and Submit: You will fill in portions of multiAgents.py during the assignment. Once you have completed the assignment. Please do not change the other files in this distribution. You will submit this part along with all the parts for the project. Each group will have one submission at the end of the semester.

**This mini project is based on CS188 projects Berkeley*

Welcome to Multi-Agent Pacman

First, play a game of classic Pacman by running the following command:

python pacman.py

and using the arrow keys to move. Now, run the provided ReflexAgent in multiAgents.py

python pacman.py -p ReflexAgent

Note that it plays quite poorly even on simple layouts:

python pacman.py -p ReflexAgent -l testClassic

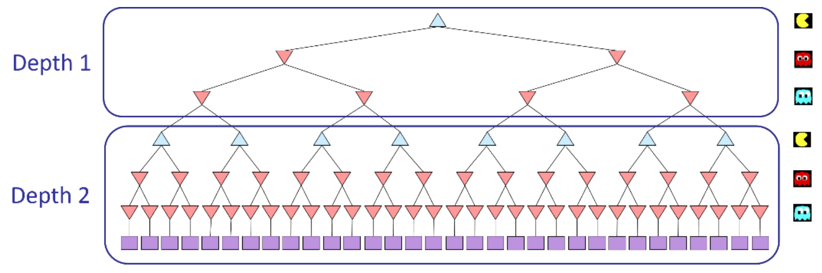
Inspect its code (in multiAgents.py) and make sure you understand what it's doing.

Q1: Minimax

Now you will write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in lecture. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Your code should also expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied self.evaluationFunction, which defaults to scoreEvaluationFunction. MinimaxAgent extends MultiAgentSearchAgent, which gives access to self.depth and self.evaluationFunction. Make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options.

Important: A single search layer is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times (see diagram below).



Minimax tree with depth 2

Grading: We will be checking your code to determine whether it explores the correct number of game states. This is the only reliable way to detect some very subtle bugs in implementations of minimax.

Hints and Observations

- Implement the algorithm recursively using helper function(s).
- The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behaviour.
- The evaluation function for the Pacman test in this part is already written (self.evaluationFunction). You shouldn't change this function, but recognize that now we're evaluating states rather than actions, as we were for the reflex agent.
- Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.

The minimax values of the initial state in the minimaxClassic layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively.

Initially, you can start testing your code for depth 1 and two agents (the player and the adversarial player).

```
python pacman.py -p MinimaxAgent -l smallClassic -a depth=1   (two agents, depth =1)
```

```
 python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 (multiple agents, and depth =4)
```

Pacman is always agent 0, and the agents move in order of increasing agent index.

All states in minimax should be GameStates, either passed in to getAction or generated via GameState.generateSuccessor.

On larger boards such as openClassic and mediumClassic (the default), you'll find Pacman to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. He might even thrash around right next to a dot without eating it because he doesn't know where he'd go after eating that dot.

When Pacman believes that his death is unavoidable, he will try to end the game as soon as possible because of the constant penalty for living. Sometimes, this is the wrong thing to do with random ghosts, but minimax agents always assume the worst:

```
  python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

Q2: Expectimax

Minimax is great, but it assumes 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 question you will implement the ExpectimaxAgent, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.

To expedite your own development, we've supplied some test cases based on generic trees. You can debug your implementation on small the game trees using the command:

Debugging on small and manageable test cases is recommended and will help you to find bugs quickly.

```
python pacman.py -p ExpectimaxAgent -l smallClassic -a depth= 1
```

```
 python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth= 2
```

Once your algorithm is working on small trees, you can observe its success in Pacman. Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. ExpectimaxAgent will no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act. To simplify your code, assume you will only be running against an adversary which chooses amongst their getLegalActions uniformly at random.

To see how the ExpectimaxAgent behaves in Pacman, run:

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

You should now observe a more cavalier approach in close quarters with ghosts. In particular, if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Investigate the results of these two scenarios:

```
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
```

You should find that your ExpectimaxAgent wins sometimes, while
your MinmaxAgent always loses.

The correct implementation of expectimax will lead to Pacman losing some of the tests.