Introduction to AI - A.Y. 2024/2025

Project 2: Multi-Agent Search

Fall '24

Abstract

This project belongs to the projects series from the Introduction to AI Course by Berkeley University. Further information can be found at the link `http://ai.berkeley.edu`.
Modules and data are stored in the `Project2AGameTheory.zip` folder from the course page.
At the end of this project, students complete the course topic on Adversarial (Multi-Agent) Search. They should be able to implement, analyze, and compare adversarial search algorithms in simple-to-complex problems.

1 Introduction

1.1 Problem & Purpose

Students implement **adversarial agents**, **minimax**, and **expectimax** in this project. The major difference between Project 2A and Project 1 is the coexistence of adversarial agents in the same environment. The project is subdivided into *Tasks* that students have to refer to. To submit their work, students should refer to Section 7.

1.2 Project Folder & Modules

The code base has not changed much from the previous project, but starting with a fresh installation is strongly suggested. All the modules belong to the `Project2AGameTheory.zip` folder from the course webpage.

Files to be edited: [1]

- `multiAgents.py`: multi-agent search agents will reside here.
- `pacman.py`: this is the main file that runs Pacman games. This file describes a Pacman *GameState* type, that is used in this project.

[1] *DO NOT* change the other files

- `game.py`: this module contains the logic behind how the Pacman world works. Furthermore, it describes several supporting types like *AgentState*, *Agent*, *Direction*, and *Grid*.
- `util.py`: here, useful data structures for implementing search algorithms are given.

Other project files are briefly described in Appendix A.

Running the project: After downloading the code, unzipping it, and changing the directory, one can play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

This is the basic command students could refer to to understand whether they correctly set the working folder.
Once this one doesn't generate any issues, two other commands have to be used to ensure the right functioning of the starting code:

`python pacman.py -p ReflexAgent.py`: Pacman game launch with a specific built-in Pacman agent, *ReflexAgent*.

`python pacman.py -p ReflexAgent.py -l testClassic`: the same command as the previous one, but now with a different map layout. This example shows this *ReflexAgent* performs quite poorly even on simple layouts.

The module `pacman.py` supports other commands too.

2 Task 1 (4 pts): Reflex Agent

The *ReflexAgent* code provides some useful methods to retrieve information about the *GameState*.
Students have to improve the *ReflexAgent* in `multiAgents.py` to consider both food and ghost locations.

Note 1: As features, you could try the reciprocal of important values (such as distance to food) rather than just the values themselves.

Note 2: The evaluation function you are writing is evaluating state-action pairs; in later parts of the project, you will be evaluating states.

Note 3: Default ghosts are random; you can also play with slightly smarter directional ghosts using `-g DirectionalGhost`. If the randomness is preventing you from telling whether your agent is improving, you can use `-f` to run with a fixed random seed (i.e., to have the same random choices every game). You can also play multiple games in a row with `-n`. Finally, turn off graphics with `-q` to run lots of games quickly.

Code check: The built agent should easily and reliably clear the *testClassic* layout:

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

Students should then try their reflex agent on the default *mediumClassic* layout with one ghost or two (and animation off to speed up the display):

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1
```

```
python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

The agent will often die with 2 ghosts on the default board, unless the built evaluation function is quite good.

3 Task 2 (5 pts): Minimax

Students have to implement an adversarial search agent in the *MinimaxAgent* class in `multiAgents.py`.

Note 1: This minimax agent should work with any number of ghosts, so students will have to write an algorithm such that the minimax tree will have multiple min layers (one for each ghost) for every max layer.

Note 2: The code should also expand the game tree to an arbitrary depth. Students should assign scores to the leaves of the tree with *self.evaluationFunction*, which defaults to *scoreEvaluationFunction*. *MinimaxAgent* extends *MultiAgentSearchAgent*, which gives access to *self.depth* and *self.evaluationFunction*. Students should make sure their minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options.

Note 3: 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 are evaluating states rather than actions: look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.

Note 4: Pacman is always agent 0, and the agents move in order of increasing agent index. A single search is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving 2 times.

Note 5: All states in minimax should be *GameStates*, either passed in to *getAction* or generated via *GameState.generateSuccessor*. In this project, you will not be abstracting to simplified states.

Note 6: The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not an error.

Note 7: On large boards such as *openClassic* and *mediumClassic*, students will find Pacman to be good at not dying, but quite bad at winning. He will often thrash around without making progress. This problem will be tackled in Task 5.

Code check: The only reliable way to detect some bugs in the implementation of minimax is to look at the number of explored game states. The autograder will be very sever about how many times you call *GameState.generateSuccessor*. If you call it any more or less than necessary, the autograder will complain.

```
python autograder.py -q q2
```

This will show how the algorithm works on a number of small trees, as well as on a pacman game. To run it without graphics, use

```
python autograder.py -q q2 --no-graphics
```

In the *minimaxClassic* layout, the minimax values of the initial state are $9, 8, 7, -492$ for depth $1, 2, 3, 4$ respectively. The minimax agent will often win (approx. 665/1000 games).

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

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

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

4 Task 3 (5 pts): Alpha-Beta Pruning

Students have to implement a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in *AlphaBetaAgent*. Part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

Note 1: The only reliable way to detect some bugs in the implementation is to look at the number of explored game states. Because the autograder makes this check, it is important to perform alpha-beta pruning without reordering children. In other words, successor states should always be processed in the *GameState.getLegalActions*. Again, do not call *Gamestate.generateSuccessor* more than necessary.

Note 2: You must not prune on equality in order to match the set of states explored by our autograder (Anyway, notice that an alternative, incompatible with our autograder, would be to also allow for pruning on equality and invoke alpha-beta once on each child of the root node).

Note 3: The correct implementation of alpha-beta pruning will lead to Pacman losing the game in some tests. This is not an error.

Code check: Students should see a speed-up (depth 3 alpha-beta should run approximately as fast as depth 2 minimax). Ideally, depth 3 on *smallClassic* should run in just a few seconds per move or faster.

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

The *AlphaBetaAgent* minimax values should be identical to the *MinimaxAgent* minimax values, although the actions it selects can vary because of different tie-breaking behavior. Again, the minimax values of the initial state in the *minimaxClassic* layout are $9, 8, 7, -492$ for depths $1, 2, 3, 4$ respectively.

Students should further test and debug their code by running

```
python autograder.py -q q3
```

```
python autograder.py -q q3 --no-graphics
```

5 Task 4 (5 pts): Expectimax

Both Minimax and alpha-beta assume Pacman is playing against an adversary who makes optimal decisions. This is not always the case. In this part, **students will implement the *ExpectimaxAgent***, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.

Code check: Students can debug their implementation on small game trees using the command
`python autograder.py -q q4`
Students have to make sure to use floats when they compute their averages (as integer division in Python truncates, so that $1/2 = 0$).
Once the algorithms is working on small trees, students should move to Pacman problems. Ghosts shouldn't be modeled with minimax search: *Expectimax* will no longer take the minimum over all ghost actions, but the expectation according to the agent's model of how the ghosts act. To simplify the code, students should assume an adversary which chooses among their *getLegalActions* uniformly at random.

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

From this, one should find that if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Students have to investigate the results of the following 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
```

The *ExpectimaxAgent* should win about half of the time (the correct implementation will lead to Pacman losing the game in some tests, but this is not an error), while *AlphaBetaAgent* always loses.

6 Task 5 (6 pts): Evaluation Function

Students have to implement a better evaluation function for pacman in the provided function *betterEvaluationFunction*.

Note 1: The evaluation function should evaluate states, rather than actions like the reflex agent evaluation function did.

Note 2: As for the reflex agent evaluation function, you may want to use the reciprocal of important values (such as distance to food) rather than the values themselves.

Note 3: One way to write the evaluation function is to use a linear combination of features. That is, compute values for features about the state that you think are important, and then combine those features by multiplying them by different values (proportionally to their importance) and adding the results together.

Check code: With depth 2 search, the evaluation function should clear the *smallClassic* layout with one random ghost more than half of the time asn still run at a reasonable rate:

```
python autograder.py -q q5
```

7 Submission

To submit their work, students must upload their folder on the course page on Microsoft Teams.

The purpose of module `autograder.py` is to briefly evaluate students' work. Using the autograder is helpful for both students and instructors. In particular, there is no limit to the number of times it can be used.

A Further Files: Info

- `graphicsDisplay.py`: graphics for Pacman.
- `graphicsUtils.py`: support for Pacman graphics.
- `textDisplay.py`: ASCII graphics for Pacman.
- `ghostAgents.py`: agents to control ghosts.
- `keyboardAgents.py`: keyboard interfaces to control Pacman.
- `layout.py`: code for reading layout files and storing their contests.
- `autograder.py`: project autograder.
- `testParser.py`: parses autograder test and solution files.
- `testClasses.py`: general autograding test classes.
- `multiagentTestClasses.py`: project 2 specific autograding test classes.
- `test_cases/`: directory containing the test cases for each question.