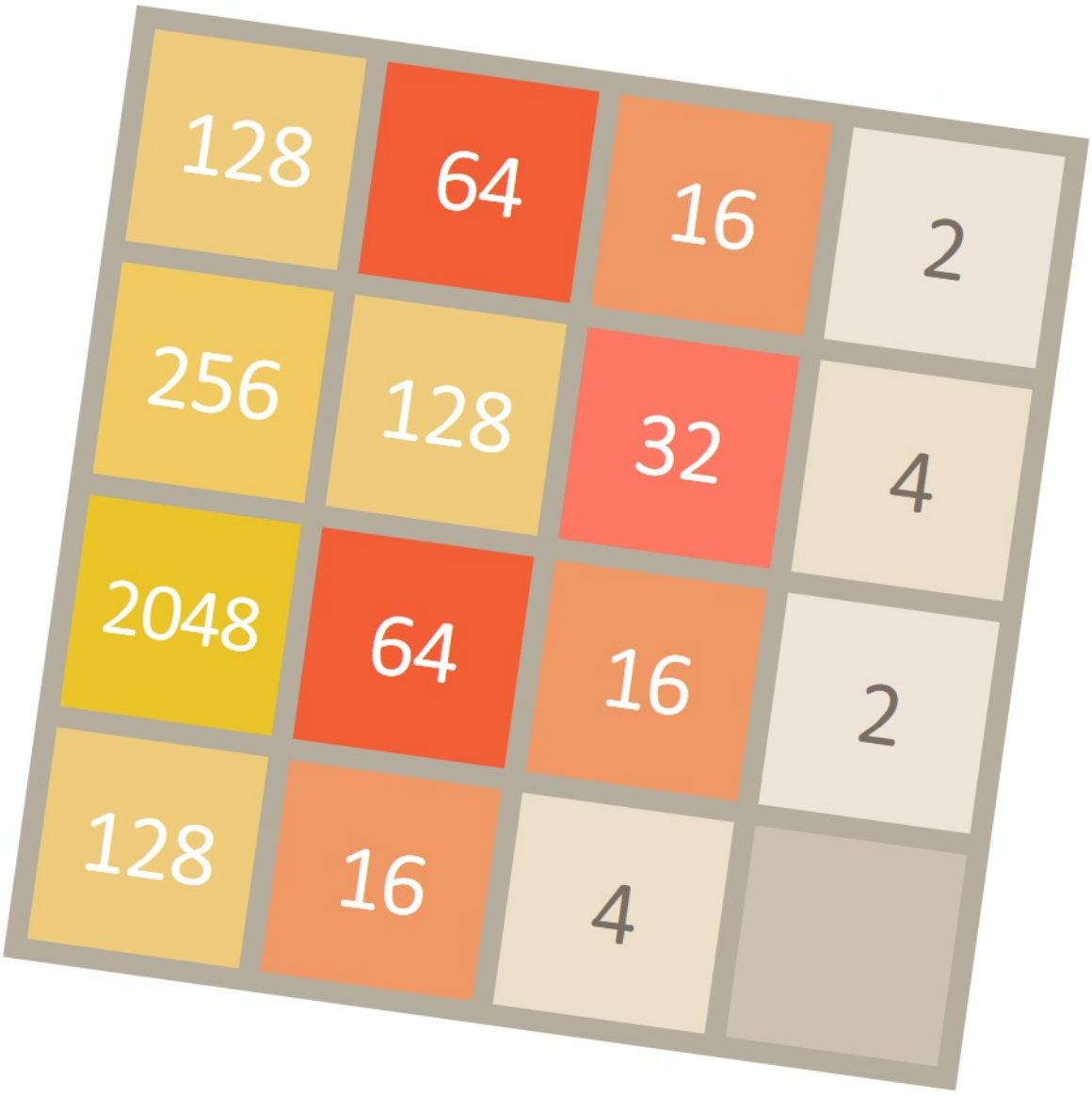
Introduction to Artificial Intelligence
Exercise 3: 2048



Deadline: June 20 2024

Introduction

In this project, you will design agents for the 2048 game. Along the way, you will implement both minimax and expectimax search and try your hand at designing an evaluation function.

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

Key files to read:

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

2048.py - The main file that runs the 2048 games.

game_state.py - This file describes a 2048 `GameState` type, which you will use extensively in this project.

game.py - The logic behind how the 2048 game works. This file describes several supporting types like `Agent`, `OpponentAction`, and `Action`.

util.py - Useful data structures for implementing search algorithms.

Supporting files you can ignore:

graphics_display.py - Graphics for 2048.

game_grid.py - Support for games graphics.

game2048_grid.py - Support for 2048 graphics.

displays.py - Summary graphics for 2048.

keyboard_agents.py - Keyboard interfaces to control 2048.

What to submit:

Code - You will fill in portions of multi_agents.py during the assignment. You should submit this file (only) and a `README.txt`.

Answers - You will answer question that will require you to think about the theory, your results and what's between them. These questions are marked by "Understanding Questiron". You should submit a pdf file with your answers.

Each team should submit exactly one tar file that contains the three files from above.

Evaluation: Your code will be autograded for technical correctness. The autograder machine is running Python 3 (don't use Python 2.7). Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. Please make sure you follow the `README` format **exactly**.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy code from someone else and submit it with minor changes, *we will know*. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: If you have any question, you are probably not alone. Please post your questions via the relevant exercise forum on the course Moodle. **Please do not write to our personal e-mail addresses!**

README format: Please submit a `README.txt` file. The `README` should include the following lines (exactly):

1. id1 --- student 1 id
2. id2 --- student 2 id
3. ***** --- 5 stars denote end of i.d information
4. comments

For an example check out the README.txt provided with your project. This `README` will be read by a script, calling the autograder. Note that if you decide to submit alone, you should remove line 2, i.e.:

1. id1 --- student 1 id
2. ***** --- 5 stars denote end of i.d information
3. comments

2048

First, sit back relax and play a nice game of 2048:

```
python3 2048.py
```

Now, run the provided `ReflexAgent` in multi_agents.py:

```
python3 2048.py --agent=ReflexAgent
```

Note that it does not play that well. Inspect its code (in multi_agents.py) and make sure you understand what it's doing.

Question 1 (3 points)

Improve the `evaluation_function` in `ReflexAgent`. The provided reflex agent code provides some helpful examples of methods that query the `GameState` for information.

```
python3 2048.py --agent=ReflexAgent --num_of_games=10 --display=SummaryDisplay
```

How does your agent fare? It will likely to achieve 512 sometime, and 256 most of the times. The autograder will check the performances of your agent performances on 20 games.

Don't spend too much time on this question, though, as the meat of the project lies ahead.

Question 2 (5 points)

Now you will write an adversarial search agent in the provided `MinimaxAgent` class in multi_agents.py.

Your code should also expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied `self.evaluation_function`, which defaults to `score_evaluation_function`. `MinimaxAgent` extends `MultiAgentAgent`, which gives access to `self.depth` and `self.evaluation_function`. 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 ply is considered to be one agent move and the board response (addition of random tile), so depth 2 search will involve an agent move two times.

Hints and Observations:

- The evaluation function in this part is already written (`self.evaluation_function`). You shouldn't change this function, 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 `test_layout` layout are 4, 12, 16 for depths 1, 2, and 3 respectively.
- We are using random seeds for reproducible results, however we may change the seeds in our tests so don't relay on this specific *Random* board.

  ```
  python3 2048.py --agent=MinmaxAgent --depth=1 --random_seed=1 --initial_board=test_layout.txt
  ```

- Depth 1 should be pretty quick, but depth 2 will be slower and depth 3 very slow. Don't worry, the next question will speed up the search somewhat.
- All states in minimax should be `GameStates`, either passed in to `get_action` or generated via `GameState.generate_success`. In this project, you will not be abstracting to simplified states.

  ```
  python3 2048.py --agent=MinmaxAgent --depth=2
  ```

Question 3 (3 points)

Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in `AlphaBetaAgent`. You should see a small speed-up (but `depth=3` should still be too much for online playing in this game).

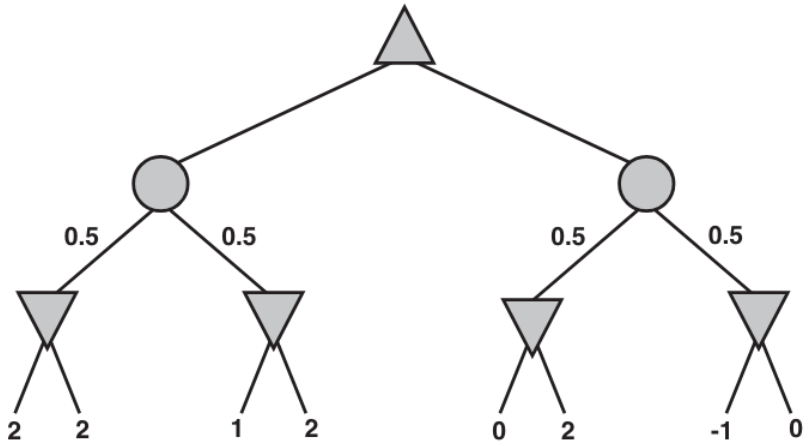
```
python3 2048.py --agent=AlphaBetaAgent --depth=2
```

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. The minimax values of the initial state in the `test_layout` layout are 4, 12, 16 for depths 1, 2, and 3 respectively.

Question 4 (2 points)

In the previous question, you implemented an alpha-beta agent that makes minimax search more efficient using pruning. In this question, we will deal with the option of pruning for an agent that also computes expectations.

This two-player game with chance is given in the following figure:



"Normal" triangles (with their base facing down) represent nodes of the MAX player, circles are CHANCE nodes (random sampling) and upside-down triangles are MIN nodes. Assume that the leaf nodes are to be evaluated in left-to-right order.

1. Copy the figure, mark the value of all the internal nodes, and indicate the best move (for the max player) at the root with an arrow. (You don't have to state your solution on the figure as long as you present the solution clearly.)
2. Given the values of the first six leaves, do we need to evaluate the seventh and eighth leaves? Given the values of the first seven leaves, do we need to evaluate the eighth leaf? Explain your answers
3. Suppose the leaf node values are known to lie between –2 and 2 inclusive. After the first two leaves are evaluated, what is the value range for the left-hand chance node?
4. Given the assumption of (3), and assuming we keep evaluating the leaves left to right until we know what is the best action for the max player, after which leave we can stop and return the answer?

Question 5 (3 points)

Random board responses is, of course, not optimal minimax agents, and so modeling them with minimax search may not be appropriate. Fill in `ExpectimaxAgent`, where your agent will no longer take the minimum over all board

possible responses, but the expectation according to your agent's model of how the board acts. To simplify your code, assume the board response uniformly at random (although in the original rules there is a higher probability for the 2 tile than others).

You should now observe a more optimistic approach that ignores possible blocking. Investigate the results of the following scenarios:

```
python3 2048.py --agent=AlphaBetaAgent --depth=2
--initial_board=risk_layout.txt --num_of_initial_tiles=0
python3 2048.py --agent=ExpectimaxAgent --depth=2
--initial_board=risk_layout.txt --num_of_initial_tiles=0
```

You should find that your `ExpectimaxAgent` achieves 1024 about half the time, while your `AlphaBetaAgent` usually achieves just 512. Make sure you understand why the behavior here differs from the minimax case.

```
python3 2048.py --agent=AlphaBetaAgent --depth=2 --num_of_games=10 --display=SummaryDisplay

python3 2048.py --agent=ExpectimaxAgent --depth=2 --num_of_games=10 --display=SummaryDisplay
```

Question 6

In the previous questions you implemented minimax (with and without pruning) and expectimax agents. In this question, you will compare these adversarial search algorithms theoretically and empirically. You will do this with respect to the game 2048.

Please note: when you present empirical results, choose a form of presentation that is easy to draw conclusions from, for example a suitable graph or table. When displaying metrics from several experiments, you better show more than just the mean - add metrics such as standard deviation or visually represent the distribution using a histogram.

success in the game (3 points)

1. Theoretically, which of the algorithms is more suitable for the game 2048? Explain what each of the algorithms assumes about the opponent and which of the assumptions better suits the game.
2. Empirically, which of the algorithms is more successful in 2048? To answer the question, run each of the algorithms for 10 games and display the scores and the largest tiles that each of the algorithms achieves.
3. Are the theoretical and empirical results consistent with each other? If so, explain. If not, explain how this is possible.
4. Compare the standard deviations in the score and the largest tile of the two algorithms. Give an intuitive explanation for the difference in results.

Alternative games (2 points)

1. Imagine an alternative game "2048 Boom" where after every move of the player there is a one in a billion chance of a "boom" where the game stops and the player finishes with a score of 0. How will the change affect the performance of the algorithms? In this section only, assume an exact expectimax algorithm that takes into account the probabilities of the moves.
2. In our implementation the expectimax player does not calculate the exact expectation but assumes that all outcomes have equal probability. Use this fact to create a new simple game where the minimax player outperforms this not-exactly-expectimax player.

Question 7 (6 points)

Write a better evaluation function for 2048 in the provided function `betterevaluation_function`. The evaluation function should evaluate states. You may use any tools at your disposal for evaluation, including your search code from the last project.

Grading: 3 point for any evaluation function that when running with `AlphaBetaAgent depth=2` most of times achieve score greater than 7000. The other 3 point will be awarded based on best score. The top 40% submissions will receive full credit; the next 35% will get 2 points and the other submissions will be awarded with one point.

```
python3 2048.py --agent=AlphaBetaAgent --depth=2 --evaluation_function=better --num_of_games=5
```

Document your evaluation function! Please describe your evaluation function at the `README` file. We're very curious about what great ideas you have, so don't be shy. We reserve the right to reward bonus points for clever solutions and show demonstrations in class.

Hints and Observations:

One way you might want to write your evaluation function is to use a linear combination of features. That is, compute values for features about the states that you think are important, and then combine those features by multiplying them by different values and adding the results together. You might decide what to multiply each feature by based on how important you think it is.

Good Luck!