Project 2: Adversarial Search

Deadlines

- Parts 1-2 (individual) due [one week after release]
- Parts 1-4 (team) due [two weeks after release]

1. Expectiminimax Search (10 pts)

(Individual and team scores will be averaged to find your final score)

In this part you will implement and experiment with expectiminimax search. Ultimately you will use it to create an optimal Tic-Tac-Toe player.

Minimax search

In *expectiminimax.py* there is a function stub for `minimax`. Ultimately you will implement the full algorithm, with alpha-beta pruning. For now, start with the basics: implement minimax search with no pruning.

You may assume that `problem` has the following useful methods:

- `getState()` – gets the current state (the initial state, at first)
- `setState()` – sets the current state (for instance, maybe set it to the one you are evaluating)
- `getTurn()` – returns a number representing whose turn it is: -1 for min, 1 for max, and 2 if the state is terminal
- `getSuccessors(state)` – returns a list of successors to the given state as 4-tuples in the following form: (next state, action to reach that state, whose turn it is in that state, the final score if the state is terminal). The last item will be `None` if the successor state is not terminal.

The function should fill in a dictionary that will represent the *strategy* in the given game. It should map states to pairs. The first item should be an optimal move in the state (there may be more than one – you should return the first optimal move in the order that the successors are given). The second item in the pair should be the minimax score of the state.
Note: when the function returns, the state of `problem` *should be the same as it was when it was passed in.*

You should *not* assume that turns always alternate between the min and max players – your minimax algorithm should check in each node whose turn it is and be general enough to handle any order of turns.

You may test your minimax algorithm on a simple example using *gametree.py*.

Specifically, if you run

```
python3 gametree.py
```
the program will use your function to calculate a strategy on a randomly generated game tree. In this tree there are two actions. The nodes marked with ^ are maximizing nodes and the nodes marked with V are minimizing nodes. The utilities of the terminal states are on the bottom.

When you run the program it will display your strategy on the tree. It will display the minimax value under each node and also mark the optimal action your strategy assigns to each node by marking its branch with #. You can control the depth of the tree with the -d option.

You can also, if you wish, specify your own tree. This program only supports game trees where each level of the tree is all the same player's turn (which need not be true in general). To specify a tree, you need to give a turn order with the `–t` option and list of payoffs at the leaves with the `–y` option. For instance, to specify the tree below, you would use the command:

```
python3 gametree.py –t 1 –1 1 –1 –y 1 3 1 1 –3 –1 –5 –3 –4 –1 –6 0 –7 0 1 –8
```

1 3 1 1 -3 -1 -5 -3 -4 -1 -6 0 -7 0 1 -8

Go ahead and use this program to thoroughly test your minimax algorithm by generating trees (randomly or manually) and checking the minimax values by hand.

Once you are confident in your minimax implementation, you can use it to play a perfect game of Tic-Tac-Toe! Run
```
python3 tictactoe.py
```
to pit your minimax agent against itself. Note that to keep things interesting, the order of the successor states in the search is randomized, so the strategy will not always be the same. You can also use `–o human` to play against your agent yourself!

Alpha-beta pruning

Now add alpha-beta pruning to your algorithm. There is an optional parameter to the `minimax` function called `prune`. If it is true, your algorithm should apply alpha-beta pruning (and not if it is false!). For correct output, states that are pruned (i.e.

not expanded at all) should not be added into the `strategy` dictionary. You can solve game trees with your alpha-beta search by running
`python3 gametree.py -p`
The output will not display branches to nodes that were not included in the strategy (i.e. not expanded), nor will it display minimax values for them. Once again, I recommend that you perform the algorithm by hand as well to make sure that your output is correct.

If you run
`python3 tictactoe.py -p`
you should notice a substantial speed-up in calculating the strategy. For instance, my implementation goes from 30 seconds to 1.5 seconds (your mileage may vary).

Expectiminimax

Now fill in the function stub named expectiminimax. It should allow for chance nodes in the game tree as well as max and min nodes. A few notes:

- When it is chance's turn, `getTurn` returns 0 (same in the output of `getSuccessors`).
- You may assume that in chance nodes the actions are equally likely. In other words, the score at a chance node is the average of the scores of its children.
- In the strategy dictionary, it doesn't matter what optimal move you give to a chance node, but you should provide the correct expectiminimax score.
- When there are chance nodes in the tree, pruning is…more complicated. Don't use any pruning in this algorithm (notice there is no `prune` argument).

You can test your algorithm by running
`python3 gametree.py -c`
That will generate a random game tree that includes chance nodes and use expectiminimax to solve it (chance nodes are represented with `O`). In this program chance nodes are assumed to give either outcome with equal probability. Again, I highly recommend that you do some examples by hand and verify the output.

If you run
`python3 tictactoe.py -o random`
you will see a Tic-Tac-Toe game between your minimax agent and an agent that takes random actions.

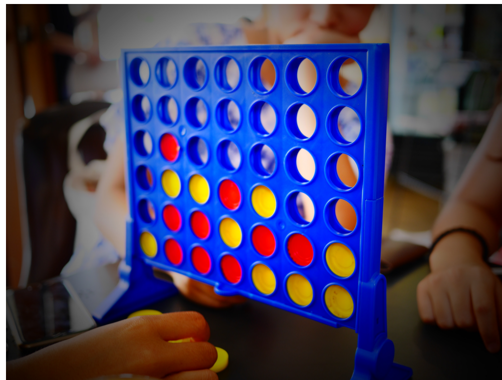
You can also use alpha-beta pruning against the random opponent, as long as you run the search at every step of the game. To do that, run
`python3 tictactoe.py -o random -p -e`

Finally, if you run
`python3 tictactoe.py -o random -s expectimax`
the agent will use expectimax, modeling the opponent with chance nodes rather than minimizing nodes (in this case that is an accurate assumption!).

2. Connect Four Agent (10 pts)

(Individual and team scores will be averaged to find your final score)

Now you'll try your hand at a bigger game that would be far harder to solve outright: Connect Four (pictured below). In the game, players take turns dropping checkers into the 7x6 grid. Whoever gets four checkers in a row first wins the game. You can play a game of Connect Four against an agent that takes random actions:
`python3 connectfour.py -p1 human`



In Tic-Tac-Toe the maximum depth of the tree is 9, and the branching factor shrinks as the tree gets deeper. In Connect Four, games can take 42 moves to finish, and in many branches all 7 actions (one for each column) remain available until nearly the end. As such, there are approximately $7^{42} = 3.1 \times 10^{35}$ nodes in the game tree! I've gone ahead and implemented depth-limited heuristic minimax for you in *heuristicminimax.pyc*[1]. Your job is to implement the heuristic evaluation function!

Heuristic Evaluation Function

The file *c4agent.py* will get you started. You should fill in the `C4HeuristicEval` class, particularly the `eval` method, which should take a state and return a heuristic value representing an estimate of who will win once the game is in this state (more negative means the minimizing player is more likely to win, more positive means the maximizing player is more likely to win).

When it is ready to test you can pit your agent against the random agent like this:
`python3 connectfour.py -p1 c4agent.py`
This will use your heuristic in depth 4 minimax search. Using the `-t` option you can have the program play multiple games in a row. For instance, if you use `-t 10`, 10

---

[1] Since this is a .pyc file, you will not be able to see the source code. You are not permitted to decompile or otherwise extract the source code from this file. If you get an "incorrect magic number" error, you need to update Python to the latest version. This file is provided for your convenience, but in a pinch you can always test your agent with the autograder to see the official results.

games will be played with your agent as player 1, and then 10 games will be played with your agent as player 2.

Move Order Heuristic

Recall from class that considering better successors first leads to more pruning, which saves time. You can't know for sure which successor is best (otherwise, why are we doing this??), but some heuristic orderings are likely to be better than others. In *c4agent.py* fill in the eval method of the `C4OrderHeuristic` class. It takes a state and should return a list of the successors of that state in whatever order you choose. Right now, it just returns the successors in the default order (each action is a column, ordered left to right). A good move order heuristic can reduce the number of nodes expanded during search and can also have a (modest) effect on the quality of the strategy as well.

Benchmark

To help measure your progress, I have also supplied a basic agent of my own in *c4benchmarkagent.pyc* (another .pyc file, see footnote 1). This agent also uses minimax search with a depth of 4. It explores successor states in a random order. The heuristic evaluation function counts the number of columns with a black tile on top (call that number b) and the number of columns with a red tile on top (call that number r). Then the value it gives a state is $(b - r)/7$. This essentially encodes that if black is on top of more columns then black is more likely to win and if red is on top of more columns then red is more likely to win. The difference is divided by 7 in order to keep the heuristic value between -1 and 1. This ensures that a winning configuration is always more desirable than a non-winning configuration with a high heuristic value. You may pit your agent against mine like this:

```
python3 connectfour.py -p1 c4agent.py -p2 c4benchmarkagent.pyc
```

Some of your score for this part depends on how often your agent wins against mine. I will run the above with `-t 25` and assign points based on the number of times your agent wins or draws.

Wins + draws (out of 50)	Max points:
20 – 24	2
25 – 29	4
30 – 34	6
35 – 39	8
40 – 50	8+1 bonus

Some of your score depends on how much your move order heuristic improves on the number of nodes expanded. If you use the `-d1` option the program will perform the search for player 1 at each step with both your heuristic and the default order and report the average number of nodes expanded for both (the actual action is

taken from the search using your heuristic). You can do the same for player 2 with `–d2`. You will receive points according to how much your move order improves the average number of nodes expanded during the above experiment.

% change with heuristic	Max points:
5% – 10%	1
> 10%	2

You may receive fewer than the maximum number of points if:

- your heuristics are nonsensical (they should represent reasonable attempts to estimating the winner from a given state or to put good actions first), or
- your code is notably buggy or poorly written.

Notes and hints

- The program will enforce a 2-second time limit on each turn. If your agent times out, then a random action will be taken for it. The 2-second limit also applies to your agent's constructor, so don't go trying to sneak in a bunch of expensive pre-computation! I don't think you are likely to hit this limit, but the autograder script is the final arbiter of the time limit so if you think you are close to the edge, test it there to be sure!
- As we discussed in class, a common strategy for creating heuristic evaluation functions is to make a weighted combination of several *features* of the game state. Think about what patterns might be important for telling whether a board is good or bad for the agent, assign them weight (importance), and add them together.
- The most helpful methods in `ConnectFour` for your heuristics are probably `getTile(row, column)` and `getHeights()`. Note that in this game row 0 is the *bottom* row (rather than the top, as in Tic-Tac-Toe).
- Make sure that your heuristic values are between -1 and 1, otherwise search might prefer non-winning nodes over winning nodes! Other than that, the magnitude of the values doesn't really matter – just order of preference.
- If you want to import a non-standard module, make sure you include that file with your submission. Don't use anyone else's code related to this specific problem, but if you use someone else's implementation of a well-known algorithm, please cite your source.
- If you gain inspiration from the existing literature on Connect Four, make sure you cite it!

3. Optional Bonus: Connect Four Tournament

(To be completed as a team only)

You are invited to submit an agent to a Connect Four tournament, with bonus points as the prize! In the above assignments, I restricted your agent in several ways. Most notably, I limited you to fixed-depth search at depth 4. For the tournament you will have significantly more freedom.

To prepare an agent for the competition, fill in the file *teamname_tournament.py.* **You should rename it with your actual team name!** That file contains a class called `C4TournamentAgent`, which currently only has a constructor and one method. The method `getMove` returns an action for the agent to perform in the game (assume that the current state of the game is the one returned by `self.__problem.getState()`. The details are up to you!

The rules are:

- Your agent must be single-threaded.
- At the end of your agent's `getMove` method `self.__problem`'s state must be the same as when `getMove` was called.
- To qualify for the tournament, your agent must defeat an agent of mine that would get full credit in Part 2 as both Player 1 and Player 2.

I will conduct a round-robin tournament amongst the qualifying agents. The two agents with the most points after the tournament will compete in a showcase match in class. The runner-up team will get 1 bonus point. The developers of the winning agent will get 2 bonus points. To have a shot at the gold, be creative and keep improving your agent!

To make it easier for you to test your agent out you can run connectfour.py in tournament mode with the `-r` option. That causes it to load a tournament agent and, when it's time for a player to move, it just calls `getMove` rather than doing its own minimax search. To get you started with a first opponent, you can use *c4benchmarkagent.pyc* in this mode as well. To see if your agent qualifies for the tournament, submit it to the autograder!

4. Report (20 pts)

(To be completed as a team only)

In *games.pdf* include the following:

Tic-Tac-Toe Analysis (10 pts)

Run *tictactoe.py* with the following configurations:

- `-O -s minimax -p -o minimax -t 100`
    - Alpha-beta pruning against itself
- `-O -s minimax -p -e -o random -t 100`
    - Alpha-beta pruning (searching in every step) against random play
- `-O -s expectimax -o minimax -t 100`
    - Expectimax against minimax
- `-O -s expectimax -o random -t 100`
    - Expectimax against random play

The `-O` option makes the agent play O instead of X, `-s` chooses the agent's strategy, `-o` chooses the opponent's strategy, and `-t` sets the number of games to play. Make a clear, well-labeled table of the results. Then, referring to these results, answer the following email from a fellow student.

To: you@acollege.edu
From: astudent@acollege.edu
Subject: Tic-tac-toe

Hey again,
Okay, so now I'm trying to make a tic-tac-toe program. I've been testing it out on my 4-year-old niece, since tic-tac-toe is still fun for her. The weird thing is that expectimax is doing *better* against her than minimax! At least it's winning a lot more against her. I thought minimax was supposed to give an optimal strategy?!?! Minimax can't even beat expectimax when you make them play each other. I don't get it. How can something that is not optimal do better than something that is optimal? What am I missing?
-A

A little while later you get another email:

To: you@acollege.edu
From: astudent@acollege.edu
Subject: Alpha-beta pruning?!?

All right. Now things are getting weird. I got tired of waiting for minimax to compute the optimal strategy so I implemented alpha-beta pruning. I know some people make it search every time but that takes forever. To save time I just use alpha-beta search to pre-compute the strategy – that's way faster. But now the agent doesn't always do the

right thing. Sometimes it could win the game, but makes a different move instead and ends up in a draw. I've even seen my niece win a couple times! Shouldn't that be impossible?! I must have a bug, right? But I tested it really well and it was working fine before. I'm so confused.
-A

Answer this email too. You should confirm the student's experience using your own agent by running *tictactoe.py* with `-O -s minimax -p -o random -t 100` (that is, a strategy pre-computed with alpha-beta search against random play) and then refer to those results in your response. You might also want to consider illustrating your explanation by performing alpha-beta search on a simple game tree and considering what would happen if that strategy were employed against an opponent who plays randomly.

Connect Four Agent Description/Analysis (10 pts)

Write a brief, clear description of your Connect Four agent. You may assume that your reader is familiar with minimax search, but you should describe anything you did beyond the "vanilla" algorithm. Ideally, after reading your description, a classmate should be able to re-implement your agent's algorithm and try it out for themselves. You don't need to describe every algorithmic step (in fact, for clarity and brevity's sake, you really shouldn't), but it should be clear what you compute and how you use the results.

Also, investigate what decisions are key to your agent's success. Consider two design decisions in your heuristics that you made that you think made a big difference. In the files *c4comparisonagent1.py* and *c4comparisonagent2.py* create agents that are just like the one in *c4agent.py* but that are each missing a key element or idea (perhaps an aspect of the evaluation function or the move order). Empirically measure the impact of these design decisions by comparing these agents to your final agent. This is sometimes called an *ablation study* because you are metaphorically removing part of your agent's "brain" to study what happens in its absence. Pit them all against the benchmark agent to see if your final agent wins more consistently. Make sure you run multiple trials (games) to be sure that your results are not just luck of the draw.

Write a clear description of the comparison agents and precisely how you gathered your results (again, ideally your description should be clear and thorough enough to allow a classmate to recreate your experiment). Present the results and clearly state your conclusions about the significance of these design decisions.