Homework 3: Adversarial Search

In homework 3, you will implement an “AI” to play two-player Sweep Fives. The instructions for this game were sent out separately.

You should not discuss this homework with anyone other than the TA and instructor. Make sure to include this statement in your submission:

On my honor, as a UNT student, I have neither given nor received unauthorized assistance on this work.

(Later problem instructions will be elaborated over the course of the next week.)

1. **Legal Actions Generation (5 points):** A key part of implementing an adversarial search is to generate the legal actions (or plays). In a simple depth first search, these actions could potentially be generated one at a time and a given action could be immediately explored before the next action is generated. However, in class we saw that the total search space is enormous, so we will eventually want to guide the search in a way that optimizes our results for a given amount of search time. In order to accomplish that, we first want to generate all of the legal actions, and then we can use a heuristic to order their exploration advantageously.

   **Update** the `actions()` function in the code provided to you for this homework, such that it returns a *list of lists*, where each embedded list represents a *legal action* (a play the agent can make) and the set of all of those lists is complete (it includes every possible action that is legal according to the rules of the game; the list should not just be the single best play – that will be selected as part of the implementation of the Minimax algorithm in a subsequent part of this homework).

   The key data structures required for this task are the agent’s (player’s) hand and the table (the dominos face-up on the table). Each of these is represented as a *list of tuples* (specifically, pairs), where a *pair* represents a single domino (or card) by indicating the two pip counts of the domino, ranging from zero to six. See the code for details.

   Each action list should start with the pair representing the domino from the player’s hand that is to be played. When the action is a discard, this will be the only pair in the list. When the domino is being used to take other dominos from the table, the card from the hand should be followed in the list by *n* pairs representing the *n* dominos to be taken from the table.

   Test your revised `GenerateActions.py` file using the test harness and cases provided.

   **Submit** your revised `GenerateActions.py` file.
2. **Minimax Algorithm (20 points):** Normally Sweep Fives is not a fully observable game, but in this question, we will treat it as such. Follow all of the normal rules of sweep fives except, assume there are only two players and that the game consists of a single hand (three dominos), which are all dealt face up (you can observe your opponent’s hand). Implement the Minimax algorithm to play an optimal game strategy for the single hand.

Update the `minimax(self, agent)` function in HW3.2_Minimax.py, such that it returns a list of tuples representing the optimal action to be taken according to the Minimax algorithm. This function should call `min_play(self, agent)` and `max_play(self, agent)`, as shown in AIMA: Russell and Norvig.

The key data structures required for this task are the agent’s (player’s) hand, the computer’s (house’s) hand, and the table (the dominos face-up on the table). See (1) above and the code for details.

As in (1) above, the action list should start with the pair representing the domino from the player’s hand that is to be played. When the action is a discard, this will be the only pair in the list. When the domino is being used to take other dominos from the table, the card from the hand should be followed in the list by \( n \) pairs representing the \( n \) dominos to be taken from the table.

Estimate how many cards you think will be on the table after each play (play 1 through play 6).

Test your implementation: `python HW3.2_Minimax.py -t`

Make sure the test cases file is in the program directory or on your path...

Compute and submit the empirical average and standard deviation of how many cards were face-up at the end of each ply (1–6) over 1000 random deals. How does this compare to your estimate above?

Guestimate the branching factor for each play (1–6). *(In class, we estimated this under the assumption there were 12 cards in the players’ hands, but here, there are 3 to 1 cards.)*

Compute and submit the empirical average and standard deviation of how many legal actions there are at the beginning of each ply (1–6) over 1000 random deals. How does this compare to your guestimate?

Submit your revised HW3.2_Minimax.py file.

Bonus (5% of problem #2): Submit the four estimates requested in problem #3 along with this response to problem #2.
3. **Alpha-beta Search (20 points):** Alpha-beta pruning avoids exploring branches of the search tree that cannot possibly result in a better expectation than the best found so far. This reduces the computational complexity of complete searches and can allow you to explore to a deeper depth in the search when a complete search is not possible.

**Implement 3.1** the `alpha_beta_search(..)` function from Figure 5.7, p170, which returns the optimal action to be taken (in a fully-observable complete search, or the optimal expectation otherwise). This function should use all of the same data structures as in `minimax(..)` above. Similarly, it also assumes just one fully-observable 3-card hand per game.

**Estimate 3.1** the percentage of total actions that you think will be pruned in this game scenario by the basic `alpha_beta_search()` algorithm relative to `minimax`. **Estimate 3.2** the percentage of the branching factor that will be pruned.

**Test 3.1** your implementation:
```
python adversarial_search.py -s alphabeta -t abs_test.txt
```
Make sure the test cases file is in the program directory or on your path.

**Implement 3.2** a `prioritize_moves(..)` function (see AIMA p169) that re-sequences the actions in an attempt to maximize the number of legal (but irrelevant) actions that can be pruned. Call this function only if the variable `alpha_beta_search(..., prioritize)` is True.

**Run 3.2** your implementation (Note: the optional ",p" sets prioritize to True, otherwise it is False):
```
python adversarial_search.py -s alphabeta[ ,p]
```

**Estimate 3.3** the percentage of total actions that you think will be pruned in this game scenario by `alpha_beta_search` with `prioritize_moves(prioritized alpha_beta_search)` relative to basic `alpha_beta_search`. **Estimate 3.4** the percentage of the branching factor that will be pruned.

**Test 3.2** your implementation:
```
python adversarial_search.py -s compare -d 1000
```
This will iterate over $d=1000$ random deals and, for each deal, will call `minimax`, basic `alpha_beta_search`, and prioritized `alpha_beta_search` (each function starting from the deal's initial state). It will report the empirical average and standard deviation of:

a. the number of calls to `actions(.)`,
b. the number of actions generated,
c. the combined number of calls to `min_f(.)` plus `max_f(.)`,
d. the effective branching factor $(c)/(a)$, and
e. the amount of time expended in the search.

It will also report the complexity reduction proportion in terms of nodes explored (c), branching factor (d), and time expended (e) for:

f. (babs) basic `alpha_beta_search` relative to (mm) `minimax = (mm-babs)/mm` and
g. (pabs) prioritized `alpha_beta_search` relative to (babs) = (babs-pabs)/babs.

**Explain** why you think your original estimates of the savings from pruning (above) are different from the values calculated here.

**Submit** your implemented `alpha_beta_search.py` file.

**Bonus** (5% of Pr. #3): Submit any estimates requested in Pr. #4 below with this response to Pr. #3.
4. **Imperfect Real-Time Decisions (20 points):** As we saw in class, the computational complexity involved in executing a complete minimax search in Sweep Fives is beyond the capabilities of normal computational power. Hence, we must cut the search off early and return a heuristically computed utility value. For this question, a deal will include all 28 dominoes – four hands of three dominoes per player (the first four dominos are placed face up on the table, then dominos are distributed one at a time to players clockwise starting to the left of the dealer until each player has three dominoes. After every player has played all of the dominos in their 3-domino hand, another hand of three dominoes is allocated from the stack (the cards that have not yet been distributed). Note: only after the completion of the last hand in a deal does the player who collected the last meld from the table pick up all of the dominos remaining on the table and then points are tallied for the deal – this does not happen after each hand of three dominoes). While the rest of the rules follow Sweep Fives, we will maintain the fully observable scenario, so we will know at the beginning of play (after each deal) exactly what each player’s current and future hands will be.

**Calculate and submit** (showing your equations) the expected depth you can search while keeping the time per domino played (time per action) under 1 second. Compute this using the empirical branching factors reported in (3.d), and the time (3.e) per node (3.c). Calculate expected depth for:

- a. minimax,
- b. basic alpha_beta_search, and
- c. prioritized alpha_beta_search.

**Implement 4.1** a heuristic eval(state) function that estimates the value of state (see AIMA 5.4.1, p171, for detailed descriptions of the search, state, heuristic value of a state, etc.)

**Implement 4.2** a modified cutoff_test(...) function that limits the search to no more than m milliseconds per action (see AIMA 5.4.2, p173, for details). *(The penalty for exceeding the specified m milliseconds will be severe.)*

**Estimate 4.1-4** do your estimates 3.1-4 change, given this four-hand (24 plies) scenario?

**Describe** three general ways in which your agent can be made to improve its win ratio.

**Test** your implementation against various levels of AI with time limit m=1000 milliseconds:

```
python adversarial_search.py -s compare -o fully -m 1000 -d 1000 -h * -a *
```

This will iterate over d=1000 random (-o fully observable) deals and, for each deal (of h hands, *=complete deal – four 3-card hands for 2 players), will execute the cross-product of search algorithm (minimax, basic alpha_beta_search, and prioritized alpha_beta_search) by opponent AI level (beginner through expert), (starting the play for each combination from the deal’s initial state). The test harness will report the win, loss and tie ratio vs. each AI and the empirical average and standard deviation over the d random deals of:

- d. the difference in points between your agent and each AI,
- e. the number of calls to actions(.),
- f. the number of actions generated,
- g. the combined number of calls to min_f(.) plus max_f(.),
- h. the effective branching factor (c)/(a), and
- i. the amount of time expended in the search.

It will also report the complexity reduction proportion in terms of nodes explored, branching factor, and time expended for:

- j. (babs) basic alpha_beta_search relative to (mm) minimax = (mm-babs)/mm and
- k. (pabs) prioritized alpha_beta_search relative to (babs) = (babs-pabs)/babs.

**Explain** why you think your original estimates of the savings from pruning (above) are different from the values calculated here.

**Submit** any revised adversarial_search.py-related files.

**Bonus** (5% of Pr. #4): Submit any estimates requested in Pr. #5 below with this response to Pr. #4.
5. **Stochastic Games (20 points)**: For this question, we will maintain the fully observable scenario for the current hand, but we no longer are provided with the foresight that indicates what either players’ future hands will be. This changes the branching factor for the game, since we could consider all the possible stochastic scenarios. That is, after reaching the depth in the search that exhausts the first hand, we could generate all of the possible deals and compositions for the second hand along with their probabilities, and after the search simulates the play of those cards, we could similarly generate all of the possible deals for the third hand with their probabilities, etc. However, we will instead use a Monte Carlo (MC) simulation to estimate the value of various plays.

**Calculate and submit 5.1 (showing your equations):**
   a. the number of possible distinct second hands for the pair of players given the knowledge available to the agent in the above scenario at the beginning of the first hand’s play,
   b. the probability of the hands in (a),
   c. the number of distinct possibilities for both the second and third hands, and
   d. the probability of the hands in (c).

**Calculate and submit 5.2 (showing your equations) the expected depth you can search to keep the time per domino played (time per action) under 1 second if you were considering all of the possible hands discussed in the preceding calculation when:**
   e. using **expectiminimax**,
   f. using basic **alpha_beta_search** with expectiminimax-style probabilistic play, and
   g. using prioritized **alpha_beta_search** with expectiminimax-style play.

Instead of generating and exploring every possible future state, we will use Monte Carlo (MC) simulation. We could wait until we reach a later child state in our search tree where we need the probabilistic information about future hands in order to take the next step. Instead of waiting, we will generate the information for our simulations at the top level of the search. Just generate more complete future card distributions (in other words, overestimate the depth you will achieve).

**Calculate and submit 5.3 (showing equations) the expected depth you can search to keep the time per action under 1 second given 100 MC simulations at each required point when:**
   h. using logic similar to expectiminimax,
   i. using basic **alpha_beta_search** with expectiminimax-style probabilistic play, and
   j. using prioritized **alpha_beta_search** under those conditions.

**Implement 5.1** `sweepfives_montecarlo_deal(state, n)`, which should generate `n` random possible future pseudo-states (distributions of cards to future hands). Specifically, depending on the depth to which you can search, you should generate as many **plausible** (valid) random hands as you will be exploring in the given search – so if you can search more than 12 plies (all the plays associated with the first hand plus the second hand), then you should not just generate a possible next hand per player, but the hand after that as well, and so on.

**Implement 5.2** `mc_minimax` and `mc_alpha_beta_search` to integrate the Monte Carlo simulation from the above discussion.

**Estimate 5.1-4** do your estimates in 4.1-4 change, for the Monte Carlo simulation scenario?

**Test** your implementation vs. various AI levels with `mc=100` MC simulations:

```
python adversarial_search.py -s compare -o hand --mc 100 --m 1000 --d 1000 -h * -a *
```

The test harness will perform as in problem #4, expect using the Monte Carlo version of functions.

**Explain** why you think your original estimates of the savings from pruning (above) are different from the values calculated here.

**Submit** any revised `adversarial_search.py`-related files.

**Bonus** (5% of Pr. #5): Submit any estimates requested in Pr. #6 below with this response to Pr. #5.
6. **Partially Observable Games (20 points):** For this question, we will move to normal Sweep Fives play, where we no longer can observe what the other player has in their hand. This again expands the branching factor for the game, since we should consider the possible cards they might hold. We will again use Monte Carlo simulation to explore several possible worlds (states).

**Update** the appropriate functions to use Monte Carlo simulation to handle the lack of observability of the opponent AI’s hand.

**Estimate 5.1-4** do your estimates in 5.1-4 change, for this partially observable scenario?

**Test** your implementation vs. various AI levels with $mc=100$ MC simulations:

```
python adversarial_search.py -s compare --mc 100 -m 1000 -d 1000 -h * -a *
```

The test harness will perform as in problem #5, expect handling non-observable opponent hands.

**Explain** why you think your original estimates of the savings from pruning (above) are different from the values calculated here.

**Submit** any revised `adversarial_search.py`-related files.
7. **Bonus Problems (80 bonus points as indicated below):** Update or implement the appropriate code to accomplish the following tasks, submit the source code, sign off on my using the code for future classes and any and all other purposes, release all rights to the code.

**Implement 7.1 (5 bonus points)** the appropriate code to play multiple deals for a full game to 10 points (or $k$ points per a parameter). Switch dealers for each deal, collect all 28 dominoes, reshuffle, and redeal. Play continues until one player has 10 or more points and has more points than their opponent (there is not a tie).

**Implement 7.2 (25 bonus points)** a server-side system that can appropriately interact with a student’s client-side code, such that they can test their system against other AIs, including against other students’ AIs.

**Implement 7.3 (25 bonus points)** a version of the game that allows a human to play interactively against an AI.

**Implement 7.4 (25 bonus points)** a graphical user interface based on existing open source dominoes software that has a license that allows the code to be used for any and all purposes without being forced to contribute the revised code back to the project.