Problem Set 2: Games

The goal of this problem set is to implement and get familiar with search algorithms.

To run the autograder, type the following command in the terminal:

```
python autograder.py
```

If you wish to run a certain problem only (e.g. problem 1), type:

```
python autograder.py -q 1
```

where 1 is the number of the problem you wish to run.

Instructions

In the attached python files, you will find locations marked with:

```
#TODO: ADD YOUR CODE HERE
NotImplemented()
```

Remove the `NotImplemented()` call and write your solution to the problem. **DO NOT MODIFY ANY OTHER CODE**; The grading of the assignment will be automated and any code written outside the assigned locations will not be included during the grading process.

IMPORTANT: Starting from this problem set, you must document your code (explain the algorithm you are implementing in your own words within the code) to get the full grade. Undocumented code will be penalized.

For this assignment, you should submit the following file only: `search.py`

Put the 2 files in a compressed zip file named `solution.zip` which you should submit to Blackboard.

Problem Definitions

There are two problems defined in this problem set:

1. **Tree Problem**: where the environment is a tree where the states are nodes and the actions are edges to the children states. The problem definition is implemented in `tree.py` and the problem instances are included in the `trees` folder.
2. **Dungeon Crawling**: where the environment is a 2D grid in which the player `'@'` has to find a key `'K'` then reach the exit door `'E'`. The dungeon contains monsters `'M'` which will kill the player and the

player can collect daggers `'~'` to kill monsters. When the player and a monster meets in the same tile, if the player has a dagger, it will kill the monster, otherwise the monster kills the player. A dagger can only be used once. The player can also collect coins `'$'` but they are not essential to win the game. Both the player and the monsters cannot stand in a wall tile `'#'` so they have a find a path that consists of empty tiles `'.'`. In addition, no more then one monster can stand on the same tile. The problem definition is implemented in `dungeon.py` and the problem instances are included in the `dungeons` folder.

You can play a graph routing or a dungeon scavenging game by running:

```
# For playing a dungeon (e.g. dungeon1.txt)
python play_dungeon.py dungeons\dungeon1.txt

# For playing a tree (e.g. tree1.json)
python play_tree.py tree\tree1.json
```

You can also let an search agent play the game in your place (e.g. a Minimax Agent) as follow:

```
python play_dungeon.py dungeons\dungeon1.txt -a minimax
```

For trees, you have to specify the 2nd player (adversary) too:

```
python play_tree.py tree\tree1.json -a minimax -adv random
```

The agent options are:

- `human` where the human play via the console
- `random` where the computer plays randomly
- `greedy` where the computer greedily selects the action that increases the heuristic value (1-step look-ahead).
- `minimax` where the computer use Minimax search to select the action.
- `alphabeta` where the computer use Alpha-beta pruning to select the action.
- `negamax` where the computer use negamax to select the action.
- `expectimax` where the computer use Expectimax search to select the action.

To get detailed help messages, run `play_dungeon.py` and `play_graph.py` with the `-h` flag.

Important Notes

The autograder will track the calls to `game.is_terminal` to check the pruning and compare with the expected output. Therefore, **ONLY CALL** `game.is_terminal` **once on each unpruned node in the game tree** in all algorithms. During expansion, make sure to loop over the actions in same order as returned by `game.get_actions` except in `alphabeta_with_move_ordering`. If two actions have the same value, pick the action that appears first in the `game.get_actions`.

Problem 1: Minimax Search

Inside `search.py`, modify the function `minimax` to implement Minimax search. The return value is a tuple containing the expected value of the game tree and the best action based on the algorithm. If the given state is terminal, the returned action should be `None`.

Problem 2: Alpha Beta Pruning

Inside `search.py`, modify the function `alphabeta` to implement Alpha Beta pruning. The return value is a tuple containing the expected value of the game tree and the best action based on the algorithm. If the given state is terminal, the returned action should be `None`.

Problem 3: Negamax

Negamax search is a variant form of minimax search that relies on the zero-sum property of a two-player game.This algorithm relies on the fact that `max(a,b)= -min(-a,-b)` to simplify the implementation of the minimax algorithm. instead of doing max and min search it does only max search. Once with normal state values (to obtain the max value) and another with negative state values (to obtain the min value)

Inside `search.py`, modify the function `negamax` to implement negamax. The return value is a tuple containing the expected value of the game tree and the best action based on the algorithm. If the given state is terminal, the returned action should be `None`.

IMPORTANT HINTS: You have to change the sign of the value retrieved from the search based on the change of the turn

Problem 4: Expectimax Search

Inside `search.py`, modify the function `expectimax` to implement Minimax search. The return value is a tuple containing the expected value of the game tree and the best action based on the algorithm. If the given state is terminal, the returned action should be `None`.

IMPORTANT NOTE: For the chance nodes, assume that all the children has the same probability. In other words, the probability of a child is 1 / (the number of children).

Time Limit

In case your computer has a different speed compared to mine (which I will use for testing the submissions), you can use the following information as a reference: `On my computer, running speed_test.py takes ~17 seconds`. You can measure the run time for this operation on computer by running:

```
python speed_test.py
```

If your computer is too slow, you can increase the time limit in the testcases files.

Delivery

The delivery deadline is `Saturday May 7th 2022 23:59`. It should be delivered on **Blackboard**. This is an individual assignment. The delivered code should be solely written by the student who delivered it. Any evidence of plagiarism will lead to receiving **zero** points.