Introduction to Artificial Intelligence HW3

0711239 李勝維

Part 1: Minimax search:

```python
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        return minimax(agent=0, depth=0, gameState=gameState)
```

Let’s see what minimax function does:

```python
def minimax(agent, depth, gameState):
    # leaf node
    if gameState.isWin() or gameState.isLose() or depth >= self.depth:
        return self.evaluationFunction(gameState)
```

First part of the minimax function checks whether the given game state is a leaf node or not. If true, it returns the evaluated value of the game state.

There are three situations of leaf node:

1. The game state wins
2. The game state loses
3. Exceeds depth limit

```python
    # max layer
    if agent == 0:
        maximum = -1.8446744e+19
        action = None
        for move in gameState.getLegalActions(agent):
            nextState = gameState.getNextState(agent, move)
            value = minimax(agent=1, depth=depth, gameState=nextState)
            if value > maximum:
                maximum = value
                action = move
        if depth == 0:  # initial call returns action
            return action
        else:
            return maximum
```

Second part of the minimax function checks whether the agent is pacman or ghost. If the agent is pacman, which means this is a max node.

The max node returns the maximum value across all legal actions by doing exhaustive search, if the depth equals zero, which means this is the root of the search tree, then it should return the action that leads to the maximum value instead of the maximum value.

```
    # min layer
    else:
        next_agent = agent + 1
        if next_agent == gameState.getNumAgents():  # go to next depth
            next_agent = 0
            depth += 1
        minimum = 1.8446744e+19
        for move in gameState.getLegalActions(agent):
            nextState = gameState.getNextState(agent, move)
            value = minimax(agent=next_agent, depth=depth,
                                gameState=nextState)
            minimum = min(minimum, value)
        return minimum
```

We only reach this part of minimax function if the agent is a ghost, which means this is a min node.

Since we have no idea whether the next layer of min node is a min node or a max node, we check if next_agent exceeds the number of ghosts. If true, the next_agent should be a pacman. If false, the next_agent should be a ghost

Next, the min node returns the minimum value across all legal actions by doing exhaustive search.

Part 2: Alpha-Beta Pruning:

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        alpha = -1.8446744e+19
        beta = 1.8446744e+19
        return ab_search(agent=0, depth=0, gameState=gameState, alpha=al
pha, beta=beta)
```

First, we initialize alpha with extreme small value and beta with extreme large value. Let’s see what ab_search function does:

```
def ab_search(agent, depth, gameState, alpha, beta):
    # leaf node
    if gameState.isWin() or gameState.isLose() or depth >= self.depth:
        return self.evaluationFunction(gameState)
```

The first part is the same as minimax function, it checks whether the given game state is a leaf node or not

```
    # max layer
    if agent == 0:
        maximum = -1.8446744e+19
        action = None
        for move in gameState.getLegalActions(agent):
            nextState = gameState.getNextState(agent, move)
            value = ab_search(agent=1, depth=depth,gameState=nextState,
                              alpha=alpha, beta=beta)
            if value > maximum:
                maximum = value
                action = move
```

The second part is mostly the same as minimax function, except the following part:

```
            # pruning
            if maximum > alpha:
                alpha = maximum
            if maximum > beta:
                if depth == 0:  # initial call returns action
                    return action
                else:
                    return maximum
```

This part does two things:

1. Updates the value of alpha if found any value greater than current alpha
2. Returns (prune) immediately if found any value greater than current beta.

```
        if depth == 0:
            return action
        else:
            return maximum
```
We only reach this part if pruning doesn’t happen, the max node returns the maximum value across all legal actions, if the depth equals zero, which means this is the root of the search tree, then it should return the action that leads to the maximum value instead of the maximum value.

```
    # min layer
    else:
        next_agent = agent + 1
        if next_agent == gameState.getNumAgents():  # go to next depth
            next_agent = 0
            depth += 1
        minimum = 1.8446744e+19
        for move in gameState.getLegalActions(agent):
            nextState = gameState.getNextState(agent, move)
            value = ab_search(agent=next_agent, depth=depth,
                                gameState=nextState, alpha=alpha, beta=beta)
            minimum = min(minimum, value)

            # pruning
            if minimum < beta:
                beta = minimum
            if minimum < alpha:
                return minimum

        return minimum
```
The third part of ab_search function is mostly the same as minimax function, except two things:

1. Updates the value of beta if found any value less than current beta.
2. Returns (prune) immediately if found any value less than current alpha

Part 3: Expectimax Search:

```
class ExpectimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        return expectimax(agent=0, depth=0, gameState=gameState)
```

Let’s see what expectimax function does:

```
def expectimax(agent, depth, gameState):
    # leaf node
    if gameState.isWin() or gameState.isLose() or depth >= self.depth:
        return self.evaluationFunction(gameState)

    # max layer
    if agent == 0:
        maximum = -1.8446744e+19
        action = None
        for move in gameState.getLegalActions(agent):
            nextState = gameState.getNextState(agent, move)
            value = expectimax(agent=1, depth=depth,
                                    gameState=nextState)
            if value > maximum:
                maximum = value
                action = move
        if depth == 0:  # initial call returns action
            return action
        else:
            return maximum
```

The first two parts does the exact same thing as minimax function does.
First it checks whether the given game state is a leaf node or not. Next it checks if the agent is pacman.

```
    # min layer
    else:
        next_agent = agent + 1
        if next_agent == gameState.getNumAgents():  # go to next depth
            next_agent = 0
            depth += 1
        sum = 0
        for move in gameState.getLegalActions(agent):
            nextState = gameState.getNextState(agent, move)
            sum += expectimax(agent=next_agent,
                                          depth=depth, gameState=nextState)

        # returns average instead of minimum
        return sum / len(gameState.getLegalActions(agent))
```

The third part of expectimax function is mostly the same as minimax function, except that expectimax function returns the average value of all legal actions instead of the minimum. The reason for simple averaging the sum is that we assume the ghost chooses actions uniformly at random, i.e. the weight of each action is the same.

Part 4: Better Evaluation Function:

```
def betterEvaluationFunction(currentGameState):
    gameState = currentGameState
    score = gameState.getScore()
    pac_pos = gameState.getPacmanPosition()
```

First, we get the current score and the position of pacman.

```
    K = 7
    Foods = gameState.getFood().asList()
    food_list = []
    for food_pos in Foods:
        tmp = util.manhattanDistance(pac_pos, food_pos)
        food_list.append(tmp)
    food_list.sort()

    if len(food_list) == 0:
        score_food = 1
    else:
        score_food = 0
        K = min(len(food_list), K)
        for i in range(K):
            score_food += food_list[i]
        score_food /= K
```

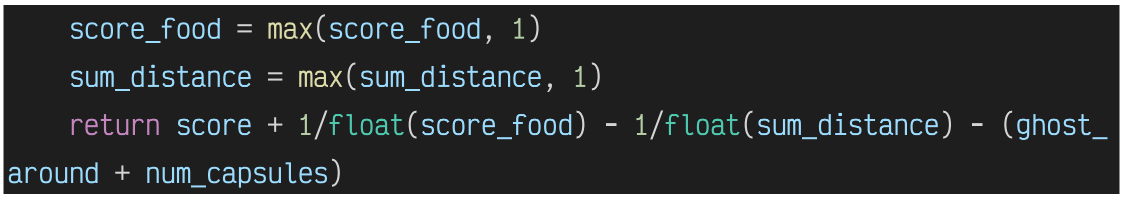
In this part, it calculates the average manhattan distance of K nearest foods, if the total amount of food is less than K, then set K to the amount of food. If there are no food left, then set score_food (score of food) to 1 to avoid divide by zero error. After some experiments, I set the default value of K to 7 for better performance.

```
    sum_distance = 0
    ghost_around = 0
    for ghost_pos in gameState.getGhostPositions():
        distance = util.manhattanDistance(pac_pos, ghost_pos)
        sum_distance += distance
        if distance == 1:
            ghost_around += 1
```

In this part, it calculates the sum of manhattan distance to all ghosts and the number of ghosts nearby (distance <= 1).

```
    num_capsules = len(gameState.getCapsules())
```

In this part, it calculates the number of remaining capsules.

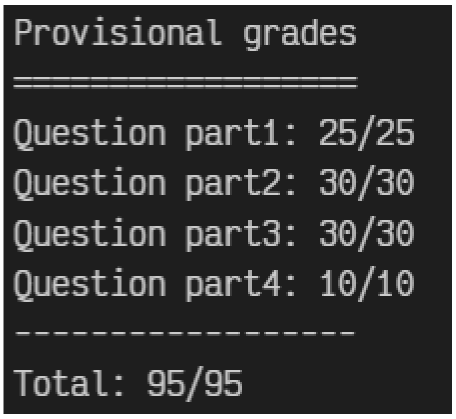


```
    score_food = max(score_food, 1)
    sum_distance = max(sum_distance, 1)
    return score + 1/float(score_food) - 1/float(sum_distance) - (ghost_
around + num_capsules)
```

Before we calculate the final score, we first set the minimum of score_food and sum_distance to 1 to avoid divide by zero error.
Variable score_food represents the distance to the food, the less value the better it is.
Variable sum_distance represents the disatance to ghosts, the greater value the better it is.
I normalize these values by doing reciprocal operation.
For ghost_around(amount of ghosts nearby) and num_capsules(amount of capsules left), since the less value the better they are, I simply deduct them from the total score.

Finally, it returns the calculated total score.

Autograder results:



```
Provisional grades
==================
Question part1: 25/25
Question part2: 30/30
Question part3: 30/30
Question part4: 10/10
------------------
Total: 95/95
```

Problems I meet and how I solve them:
At first, I failed to build the search tree bottom up, instead I use a recurrence approach to build the search tree.