Homework 3: Multi-Agent Search

110550080 何曉嫻

Part I. Implementation (5%):

Part 1

```python
        # Begin your code (Part 1)
        """
        1. call minimax function
        2. depth: the current depth of the tree
           agentIndex: index of the current agent
        3. Roll over agent index and increase current depth if all agents have finished
        4. Return the value of evaluationFunction if max depth is reached
        5. If it is max (pacman) turn:
            Get the maximun score of successor
            Increase agent_index by 1 as it will be next player's (ghost) turn
            Pass the new game state generated by pacman's action
            Update the best score and action
        6. If it is min (ghost) turn:
            Get the minimun score of successor
            Increase agent_index by 1 as it will be next player's (ghost or pacman) turn
            Pass the new game state generated by ghost's action
            Update the best score and action
        7. If it is a leaf state with no successor states, return the value of evaluationFunction
        8. Return the best_action and best_score
        9. getAction function retuen the action
        """
        action, score = self.minimax(0, 0, gameState)
        return action
    def minimax(self, depth, agentIndex, gameState):
        if agentIndex>= gameState.getNumAgents():
            agentIndex=0
            depth+=1
        if(depth==self.depth):
            return None,self.evaluationFunction(gameState)

        best_action, best_score = None, None
        actions= gameState.getLegalActions(agentIndex)
        if agentIndex==0:
            for action in actions:
                next= gameState.getNextState(agentIndex, action)
                _,score=self.minimax(depth, agentIndex+1, next)
                if best_score==None or score>best_score :
                    best_score=score
                    best_action=action
        else:
            for action in actions:
                next= gameState.getNextState(agentIndex, action)
                _,score=self.minimax(depth, agentIndex+1, next)
                if best_score==None or score<best_score :
                    best_score=score
                    best_action=action
        if best_score is None:
            return  None,self.evaluationFunction(gameState)
        return best_action, best_score

        raise NotImplementedError("To be implemented")
        # End your code (Part 1)
```

Part 2

```python
        # Begin your code (Part 2)
        """
        1. Same as minimax
        2. Add alpha and beta
            alpha: The best choice (highest-value) we have found so far along the path of Maximizer.
                   The initial value of alpha is -inf.
            beta: The best choice (lowest-value) we have found along the path of Minimizer.
                  The initial value of beta is inf.
        3. Prune the tree if alpha is greater than beta
        """
        action, score = self.AlphaBeta(0, 0, gameState, -float('inf'), float('inf'))
        return action
    def AlphaBeta(self, depth, agentIndex, gameState, alpha, beta):
        if agentIndex>= gameState.getNumAgents():
            agentIndex=0
            depth+=1
        if(depth==self.depth):
            return None,self.evaluationFunction(gameState)

        best_action, best_score = None, None
        actions= gameState.getLegalActions(agentIndex)
        if agentIndex==0:
            for action in actions:
                next= gameState.getNextState(agentIndex, action)
                _,score=self.AlphaBeta(depth, agentIndex+1, next, alpha, beta)
                if best_score==None or score>best_score :
                    best_score=score
                    best_action=action
                if score>alpha:
                    alpha=score
                if alpha>beta:
                    break
        else:
            for action in actions:
                next= gameState.getNextState(agentIndex, action)
                _,score=self.AlphaBeta(depth, agentIndex+1, next, alpha, beta)
                if best_score==None or score<best_score :
                    best_score=score
                    best_action=action
                if score<beta:
                    beta=score
                if alpha>beta:
                    break
        if best_score is None:
            return  None,self.evaluationFunction(gameState)
        return best_action, best_score
        raise NotImplementedError("To be implemented")
```

Part 3

```python
        # Begin your code (Part 3)
        """
        1. Same as minimax, but changes the "mini" part to "expect"
        2. If it is chance (ghost) turn:
           The score is expected value of its children
           The best action is chooseed uniformly at random
        """
        action, score = self.Expectimax(0, 0, gameState)
        return action
    def Expectimax(self, depth, agentIndex, gameState):
        if agentIndex>= gameState.getNumAgents():
            agentIndex=0
            depth+=1
        if(depth==self.depth):
            return None,self.evaluationFunction(gameState)

        best_action, best_score = None, None
        actions= gameState.getLegalActions(agentIndex)
        if agentIndex==0:
            for action in actions:
                next= gameState.getNextState(agentIndex, action)
                _,score=self.Expectimax(depth, agentIndex+1, next)
                if best_score==None or score>best_score :
                    best_score=score
                    best_action=action

        else:
            n=len(actions)
            for action in actions:
                next= gameState.getNextState(agentIndex, action)
                _,score=self.Expectimax(depth, agentIndex+1, next)
                if best_score==None:
                    best_score=0
                best_score+=score*1/n
                best_action=action

        if best_score is None:
            return  None,self.evaluationFunction(gameState)
        return best_action, best_score
        raise NotImplementedError("To be implemented")
        # End your code (Part 3)
```

Part 4

```python
    # Begin your code (Part 4)
    """
    1. Calculating distance to the closest food pellet
    2. Calculating the distances from pacman to the ghosts.
       Also, checking for the ghosts around pacman (at distance of 1).
    3. Obtaining the number of capsules available
    4. Combination of the above calculated metrics.
    """
    position= currentGameState.getPacmanPosition()
    foods = currentGameState.getFood().asList()
    ghostState=currentGameState.getGhostPositions()
    capsules = currentGameState.getCapsules()
    min_food= -1
    for food in foods:
        distance = manhattanDistance(position, food)
        if  distance < min_food  or min_food == -1:
            min_food = distance
    ghost_dis = 1
    scared_ghost= 0
    for ghost in ghostState:
        distance = manhattanDistance(position, ghost)
        ghost_dis += distance
        if distance <= 1:
            scared_ghost += 1
    capsule = len(capsules)
    return currentGameState.getScore() + (1 / float(min_food)) - (1 / float(ghost_dis)) - scared_ghost- capsule
    raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

Part II. Results & Analysis (5%):

Using manhattanDistance to calculating distance is best, and the scores are close. Tuning these parameters and coming up with these calculation functions are kind of arbitrary to me because the only way to tell if the formula works is to test it on the map. It is empirically proved to be fine, but it may not represent the real optimal evaluation strategy of the game. It is slightly biased by the map.