Homework 3: Multi-Agent Search Report

Part I. Implementation (5%):

- **Screenshots and explanations of my codes:**

Part 1:

```python
        # Begin your code (Part 1)
        """
            In part 1, I use a recursion function called "value" to implement
        minimax search. When entering this function, first, we check whether it is
        the end of a depth and change the agent to the next one who will pick
        the max/min value from its leaf nodes.
            And then, check whether the currenGameState is terminal state or
        acheives self.depth. If it is, return the self.evaluationFunction value
        of the state; If not, get the max(pacman)/min(ghost) value from recursing
        this function with all the NextState which derived from the legalMoves of
        currentGameState.
            Finally, return the optimal chose value and action which will lead
        to the next state.
        """
        def value(currentGameState, now_depth, agentIndex):
            if agentIndex == currentGameState.getNumAgents() - 1:
                now_depth += 1
                agentIndex = 0
            else:
                agentIndex += 1

            if currentGameState.isWin() or currentGameState.isLose() or now_depth > self.depth:
                return self.evaluationFunction(currentGameState), 0

            legalMoves = currentGameState.getLegalActions(agentIndex)
            NextState = [currentGameState.getNextState(
                agentIndex, action) for action in legalMoves]
            stateValue = [value(nextState, now_depth, agentIndex)[0]
                          for nextState in NextState]

            if agentIndex == 0:
                v = max(stateValue)
            else:
                v = min(stateValue)

            val_indice = [index for index in range(
                len(stateValue)) if stateValue[index] == v]
            chosenIndex = random.choice(val_indice)

            return v, legalMoves[chosenIndex]

        (v, action) = value(gameState, 1, -1)
        return action
        raise NotImplementedError("To be implemented")
        # End your code (Part 1)
```

Part 2:

```python
        # Begin your code (Part 2)
        """
            In part 2, I use same recursive method which is roughly like part 1.
        The thing different is that I moved the "max_value" and "min_value" out
        of the "value" function and add alpha and beta parameters into each
        function to implement alpha-beta pruning.
            When entering "max_value" function, initialize v to negative infinite.
        Then, get the nextStateValue of one of the legal actions, if it larger than
        v, update val with it. If v larger than beta, return (v, action) immediately
        to prune the leaf nodes. Otherwise, keep getting other values and repeat
        steps above. At last, return (v, action).
            Similarly, "min_value" is alike to "max_value", the thing needed to do
        is alpha, beta exchanged and '>', '<' reversed.
        """
        def value(currentGameState, now_depth, agentIndex, alpha, beta):
            if agentIndex == currentGameState.getNumAgents() - 1:
                now_depth += 1
                agentIndex = 0
            else:
                agentIndex += 1

            if currentGameState.isWin() or currentGameState.isLose() or now_depth > self.depth:
                return self.evaluationFunction(currentGameState), 0
            if agentIndex == 0:
                return max_value(currentGameState, now_depth, agentIndex, alpha, beta)
            else:
                return min_value(currentGameState, now_depth, agentIndex, alpha, beta)

        def max_value(currentGameState, now_depth, agentIndex, alpha, beta):
            v = float('-inf')
            legalMoves = currentGameState.getLegalActions(agentIndex)

            for index in range(len(legalMoves)):
                nextStateValue = value(currentGameState.getNextState(
                    agentIndex, legalMoves[index]), now_depth, agentIndex, alpha, beta)[0]

                if v < nextStateValue:
                    chosenIndex = index
                    v = nextStateValue

                if v > beta:
                    return v, legalMoves[index]
                alpha = max(alpha, v)

            return v, legalMoves[chosenIndex]

        def min_value(currentGameState, now_depth, agentIndex, alpha, beta):
            v = float('inf')
            legalMoves = currentGameState.getLegalActions(agentIndex)

            for index in range(len(legalMoves)):
                nextStateValue = value(currentGameState.getNextState(
                    agentIndex, legalMoves[index]), now_depth, agentIndex, alpha, beta)[0]

                if v > nextStateValue:
                    chosenIndex = index
                    v = nextStateValue

                if v < alpha:
```

```
                    return v, legalMoves[index]
                beta = min(beta, v)

            return v, legalMoves[chosenIndex]

        alpha = float('-inf')
        beta = float('inf')
        (v, action) = value(gameState, 1, -1, alpha, beta)
        return action
        raise NotImplementedError("To be implemented")
        # End your code (Part 2)
```

Part 3:

```
        # Begin your code (Part 3)
        """
            In part 3, I use the same recursive method which is roughly like part 1.
        The thing different is that I moved the "max_value" and "expected_value"
        out of the "value" function to implement expectimax search.
            The "expected_value" function returns the expected value of a random-
        moving ghost. In this function, after evaluating all the values of NextState,
        we calculate its expected value as v by: sum(NextStateValue)/len(legalMoves).
        And then randomly choose a action from legalMoves. Last, return (v, action).
        """
        def value(currentGameState, now_depth, agentIndex):
            if agentIndex == currentGameState.getNumAgents() - 1:
                now_depth += 1
                agentIndex = 0
            else:
                agentIndex += 1

            if currentGameState.isWin() or currentGameState.isLose() or now_depth > self.depth:
                return self.evaluationFunction(currentGameState), 0
            if agentIndex == 0:
                return max_value(currentGameState, now_depth, agentIndex)
            else:
                return expected_value(currentGameState, now_depth, agentIndex)

        def max_value(currentGameState, now_depth, agentIndex):

            legalMoves = currentGameState.getLegalActions()
            NextState = [currentGameState.getNextState(
                agentIndex, action) for action in legalMoves]
            stateValue = [value(nextState, now_depth, agentIndex)[0]
                          for nextState in NextState]

            v = max(stateValue)
            val_indice = [index for index in range(
                len(stateValue)) if stateValue[index] == v]
            chosenIndex = random.choice(val_indice)

            return v, legalMoves[chosenIndex]

        def expected_value(currentGameState, now_depth, agentIndex):

            legalMoves = currentGameState.getLegalActions(agentIndex)
            NextState = [currentGameState.getNextState(
                agentIndex, action) for action in legalMoves]

            expect_value = 0
```

```
            for nextState in NextState:
                expect_value += (value(nextState, now_depth,
                                       agentIndex)[0] / len(NextState))
            action = random.choice(legalMoves)

            return expect_value, action

        (v, action) = value(gameState, 1, -1)
        return action
```

Part 4:

```
    # Begin your code (Part 4)
    """
        The idea my evaluation function is pretty simple, I initialize the value to
    zero and use four factors to determine it.
    1. "Win/Lose" is the end of the game, so we directly return positive/negative
        infinite.
    2. "The remaining number of food" would mainly affect the action of pacman. This
        factor makes the pacman tend to eat food.
    3. "Letting ghost be scared". As we know, eating scared ghost will get much score.
        So, this factor would make the pacman eat the capsule if it finds any capsules
        nearby.
    4. "The min distance between pacman and ghost" can affect the action of pacman
        slightly. This factor let pacman would not stay in place when there is no food
        around it, or get as close to the ghost as possible when the ghost is scared.
    """
    pos = currentGameState.getPacmanPosition()
    GhostStates = currentGameState.getGhostStates()
    minGhostDistance = min(
        [manhattanDistance(pos, state.getPosition()) for state in GhostStates])

    value = 0
    if currentGameState.isWin():
        value = float('inf')
        return value
    elif currentGameState.isLose():
        value = float('-inf')
        return value

    value -= currentGameState.getNumFood()
    value -= (minGhostDistance / 25000)

    for state in GhostStates:
        if state.scaredTimer > 0:
            value += 1

    return value

    raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

Part II. Results & Analysis (5%):

- **Screenshot the results:**

```
問題    輸出    偵錯主控台    終端機

***         >= 5:  1 points
***         >= 10:  2 points
***     10 wins (4 of 4 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 4:  2 points
***         >= 7:  3 points
***         >= 10:  4 points

### Question part4: 10/10 ###


Finished at 14:59:59

Provisional grades
==================
Question part1: 20/20
Question part2: 25/25
Question part3: 25/25
Question part4: 10/10
------------------
Total: 80/80


                     ALL HAIL GRANDPAC.
              LONG LIVE THE GHOSTBUSTING KING.

                  ---      ----      ---
                  |  \    /  + \    /  |
                  | + \--/      \--/ + |
                  |   +     +          |
                  | +     +        +   |
                @@@@@@@@@@@@@@@@@@@@@@@@@@
              @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
            @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
            @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
            \   @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
             \ /  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
              V   \   @@@@@@@@@@@@@@@@@@@@@@@@@@@@
                   \ /  @@@@@@@@@@@@@@@@@@@@@@@@@@
                    V     @@@@@@@@@@@@@@@@@@@@@@@@
```

- **Observation and analysis of my evaluation function:**

1. My evaluation function let pacman has much higher overall win rate. Only when it is surrounded by multiple ghosts or there is no way to escape (get trapped), it would have a little possibility to lose.
2. "The min distance between pacman and ghost", a factor I designed in the function, is a double-edged sword. Because this factor makes pacman and ghost closer under the condition that the former will not lose the game. Therefore, that makes pacman have more chance to eat scared ghost and gain extra score. But simultaneously, the factor also makes pacman move more slowly when eating a chain of food, and thus causing a little score loss.