HW3 – Report

I. Implementation

- Part 1. MiniMax Search

```
# Begin your code (Part 1)

def minimax(state, index, agent_count, max_index):
    if state.isWin() or state.isLose() or index == max_index:  # Terminal State or Max Depth reached
        return self.evaluationFunction(state), None  # return the heuristic value
    actions = state.getLegalActions(index % agent_count)  # all legal moves of current player
    scores = [minimax(
        state.getNextState(index % agent_count, act),
        index + 1,
        agent_count,
        max_index
    )[0] for act in actions]  # get every possible action's value
    if (index % agent_count) == 0:  # Max Player
        max_score = max(scores)  # choose the best one
        move = scores.index(max_score)  # and get its corresponding action
        return max_score, actions[move]
    else:  # Min Player
        min_score = min(scores)  # choose the worst one
        move = scores.index(min_score)  # and get its corresponding action
        return min_score, actions[move]

num = gameState.getNumAgents()
score, action = minimax(gameState, 0, num, self.depth * num)
# maxIndex (total rounds) = depth (turns) * num (players)

return action

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

- Part 2. Alpha-Beta Pruning

```
# Begin your code (Part 2)

def alpha_beta(state, index, agent_count, max_index, alpha, beta):
    if state.isWin() or state.isLose() or index == max_index:  # Terminal State or Max Depth reached
        return self.evaluationFunction(state), None  # return the heuristic value
    actions = state.getLegalActions(index % agent_count)  # all legal moves of current player
    if (index % agent_count) == 0:  # Max Player
        val = (-0xffffffff, None)  # init val = -inf
        for act in actions:
            v = alpha_beta(
                state.getNextState(index % agent_count, act),
                index + 1,
                agent_count,
                max_index,
                alpha,
                beta
            )  # v = value(successor)
            if v[0] > val[0]:  # val = max(v, val)
                val = v[0], act  # max value and its action
            if val[0] > beta:  # beta prune
                return val
            alpha = max(alpha, val[0])  # update alpha
        return val
```

```
    else:  # Min Player
        val = (0xffffffff, None)  # init val = inf
        for act in actions:
            v = alpha_beta(
                state.getNextState(index % agent_count, act),
                index + 1,
                agent_count,
                max_index,
                alpha,
                beta
            )  # v = value(successor)
            if v[0] < val[0]:  # val = min(v, val)
                val = v[0], act  # min value and its action
            if val[0] < alpha:  # alpha prune
                return val
            beta = min(beta, val[0])  # update beta
        return val

num = gameState.getNumAgents()
score, action = alpha_beta(gameState, 0, num, self.depth * num, -0xffffffff, 0xffffffff)
# Alpha-Beta(state, alpha = -inf, beta = inf)

return action

# raise NotImplementedError("To be implemented")
# End your code (Part 2)
```

- Part 3. ExpectiMax Search

```
def expectimax(state, index, agent_count, max_index):
    if state.isWin() or state.isLose() or index == max_index:  # Terminal State or Max Depth reached
        return self.evaluationFunction(state), None  # return the heuristic value
    actions = state.getLegalActions(index % agent_count)  # all legal moves of current player
    scores = [expectimax(
        state.getNextState(index % agent_count, act),
        index + 1,
        agent_count,
        max_index
    )[0] for act in actions]  # get every possible action's value
    if (index % agent_count) == 0:  # Max Player
        max_score = max(scores)  # choose the best one
        move = scores.index(max_score)  # and get its corresponding action
        return max_score, actions[move]
    else:  # Chance Player
        avg_score = sum(scores) / len(actions)  # calculate the average value of all possible moves
        return avg_score, None

num = gameState.getNumAgents()
score, action = expectimax(gameState, 0, num, self.depth * num)

return action

# raise NotImplementedError("To be implemented")
# End your code (Part 3)
```

- Part 4. Evaluation Function

```
# Begin your code (Part 4)

pos = currentGameState.getPacmanPosition()
ghostStates = currentGameState.getGhostStates()

scaredTimes = [ghostState.scaredTimer for ghostState in ghostStates if ghostState.scaredTimer > 0]
capsules = currentGameState.getCapsules()
foods = currentGameState.getFood().asList()

nearestGhostDist = min([manhattanDistance(pos, state.getPosition()) for state in ghostStates])
nearestScaredGhostDist = -1 if not scaredTimes else min(
    [manhattanDistance(pos, ghost.getPosition()) for ghost in ghostStates if ghost.scaredTimer > 0])
nearestFoodDist = 0 if not foods else min([manhattanDistance(pos, food) for food in foods])
nearestCapsuleDist = 0 if not capsules else min([manhattanDistance(pos, capsule) for capsule in capsules])

if nearestGhostDist <= 1 and nearestScaredGhostDist == -1:  # the least willing result, DIE
    return -0xffff  # huge penalty
```

```
if nearestScaredGhostDist != -1:
    """
    if there is a scared ghost,
    then eating capsules and foods are not that important.
    start chasing ghosts instead.
    scared ghost distance (as close as possible, the most important, so times -50)
    food and capsule distance (as close as possible, so times -5)
    """
    return nearestScaredGhostDist * -50 + nearestFoodDist * -5 + nearestCapsuleDist * -5
else:
    """
    normal time, game over when touching a ghost.
    so nearest ghost distant is an factor, but not that important.
    trying to get close to foods and capsules are the most valuable factor,
    also the count doesn't changes a lot, so it needs a big multiplier, say 1000.
    ghost distance (as far as possible, not that important, so / 10)
    food and capsule distance (as close as possible, so times -10)
    # of capsules and foods (as small as possible, very important, so * -1000)
    """
    return nearestGhostDist / 10 + nearestFoodDist * -10 + nearestCapsuleDist * -10 + \
        len(capsules) * -1000 + len(foods) * -1000

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

II. Result & Analysis

- Result:

```
Finished at 22:54:24

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

- Analysis:

  At the first, I considered only the distance of nearest food and capsule, but I found player often stay aside an item and doesn't go to consume it. After some consideration, I added number of items as a factor and times a large number, it finally works normally.

  Chasing scared ghosts is also a good way pursue high score and high win rate, but if a general heuristic function considers ghost distance as a factor (further -> higher value), the player cannot utilize the advantage of eating a ghost. Thus, I separate the case of chasing ghost from general case, let getting closer to scared glost obtains a higher value, and the overall score increased a lot after the modifications above.