Part I. Implementation

- **Part 1: Minimax Search**

```
def getAction(self, gameState):
    """...
    # Begin your code (Part 1)

    def MiniMax(gameState, agentIndex, depth=0):
        # Get all legal actions
        legalActionList = gameState.getLegalActions(agentIndex)
        numIndex = gameState.getNumAgents() - 1
        bestAction = None

        # If node is terminal, then return score
        if (gameState.isLose() or gameState.isWin() or (depth == self.depth)):
            return [self.evaluationFunction(gameState)]
        elif agentIndex == numIndex: # Pacman and Ghost's turns are over -> next depth
            depth += 1
            childAgentIndex = self.index
        else: # Increase agent_index by 1 as it will be next player's turn now
            childAgentIndex = agentIndex + 1
        # Ghost
        if agentIndex != 0:
            v = float("inf") # Initialize
            for legalAction in legalActionList: # For each legal action of ghost agent
                # Generate successor
                successorGameState = gameState.getNextState(agentIndex, legalAction)
                # Get the minimax score of successor
                min = MiniMax(successorGameState, childAgentIndex, depth)[0]
                if min == v:
                    if bool(random.getrandbits(1)):
                        bestAction = legalAction # choose bestaction randomly
                elif min < v:
                    v = min
                    bestAction = legalAction
```

```
                if min == v:
                    if bool(random.getrandbits(1)):
                        bestAction = legalAction # choose bestaction randomly
                elif min < v:
                    v = min
                    bestAction = legalAction
            return v, bestAction
        # Pacman
        else:
            v = -float("inf") # Initialize
            for legalAction in legalActionList: # For each legal action of Pacman
                # Generate successor
                successorGameState = gameState.getNextState(agentIndex, legalAction)
                # Get the minimax score of successor
                max = MiniMax(successorGameState, childAgentIndex, depth)[0]
                if max == v:
                    if bool(random.getrandbits(1)):
                        bestAction = legalAction # choose bestaction randomly
                elif max > v:
                    v = max
                    bestAction = legalAction
            return v, bestAction

    bestScoreActionPair = MiniMax(gameState, self.index)
    bestScore = bestScoreActionPair[0]
    bestMove =  bestScoreActionPair[1]

    return bestMove

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

- **Part 2: Alpha-Beta Pruning**

```python
def getAction(self, gameState):
    """..."""
    # Begin your code (Part 2)

    def max_agent(state, depth, alpha, beta):
        # If node is terminal, then return score
        if state.isWin() or state.isLose():
            return state.getScore()
        # Get all legal actions of Pacman
        legalActionList = state.getLegalActions(0)
        # Initalize
        bestScore = float("-inf")
        score = bestScore
        bestAction = Directions.STOP

        for legalAction in legalActionList: # For each legal action of Pacman
            # For each successor of state: score = max(score, value(successor, α, β))
            score = min_agent(state.getNextState(0, legalAction), depth, 1, alpha, beta)
            if score > bestScore:
                bestScore = score
                bestAction = legalAction
            # Updata the value of alpha
            alpha = max(alpha, bestScore)
            # Prune
            if bestScore > beta:
                return bestScore
        if depth == 0:
            return bestAction
        else:
            return bestScore

    def min_agent(state, depth, ghost, alpha, beta):
```

```python
    def min_agent(state, depth, ghost, alpha, beta):
        # If node is terminal, then return score
        if state.isLose() or state.isWin():
            return state.getScore()
        next_player = ghost + 1
        if ghost == state.getNumAgents() - 1: # All ghost are over
            next_player = 0
        # Get all legal actions of Ghost
        legalActionList = state.getLegalActions(ghost)
        # Initalize
        bestScore = float("inf")
        score = bestScore
        for legalAction in legalActionList: # For each legal action of Ghost
            if next_player == 0: # On the last ghost and it will be Pacman's turn next.
                if depth == self.depth - 1: # If node is terminal, return the evaluation of score
                    score = self.evaluationFunction(state.getNextState(ghost, legalAction))
                else: # If not, call max_agent
                    score = max_agent(state.getNextState(ghost, legalAction), depth + 1, alpha, beta)
            else: # For Ghost
                # For each successor of state: score = min(score, value(successor, α, β))
                score = min_agent(state.getNextState(ghost, legalAction), depth, next_player, alpha, beta)
            if score < bestScore:
                bestScore = score
            # Update the value of beta
            beta = min(beta, bestScore)
            # Prune
            if bestScore < alpha:
                return bestScore
        return bestScore
    return max_agent(gameState, 0, float("-inf"), float("inf"))
    raise NotImplementedError("To be implemented")
    # End your code (Part 2)
```

- **Part 3: Expectimax Search**

```
def getAction(self, gameState):
    """...
    # Begin your code (Part 3)
    def expectimax(gameState, agentIndex, depth=0):
        # Get all legal actions
        legalActionList = gameState.getLegalActions(agentIndex)
        numIndex = gameState.getNumAgents() - 1
        bestAction = None
        # If node is terminal, then return score
        if (gameState.isLose() or gameState.                    th): # Pacman and Ghost's turns are over -> next depth
            return [self.evaluationFunction(gameState)]
        elif agentIndex == numIndex:
            depth += 1
            childAgentIndex = self.index
        else:
            childAgentIndex = agentIndex + 1

        numAction = len(legalActionList)
        # If player(pos) == MAX: value = -infinity
        if agentIndex == self.index:
            value = -float("inf")
        # If player(pos) == CHANCE: value = 0
        else:
            value = 0

        for legalAction in legalActionList: # For each legal action of player
            # Generate successor
            successorGameState = gameState.getNextState(agentIndex, legalAction)
            expectedMax = expectimax(successorGameState, childAgentIndex, depth)[0]
            if agentIndex == self.index:
                if expectedMax > value:
                    # value, best_move = nxt_val, move
```

(parameter) gameState: Any

```
            for legalAction in legalActionList: # For each legal action of player
                # Generate successor
                successorGameState = gameState.getNextState(agentIndex, legalAction)
                expectedMax = expectimax(successorGameState, childAgentIndex, depth)[0]
                if agentIndex == self.index:
                    if expectedMax > value:
                        # value, best_move = nxt_val, move
                        value = expectedMax
                        bestAction = legalAction
                else:
                    # value = value + prob(move) * nxt_val
                    value = value + ((1.0/numAction) * expectedMax)
            return value, bestAction

        bestScoreActionPair = expectimax(gameState, self.index)
        bestScore = bestScoreActionPair[0]
        bestMove =  bestScoreActionPair[1]
        return bestMove
        raise NotImplementedError("To be implemented")
        # End your code (Part 3)
```

- **Part 4: Evaluation Function**

```
def betterEvaluationFunction(currentGameState):
    """...
    # Begin your code (Part 4)
    """
    States are evaluated into sections that linearly correspond to the total
    of the evaluation: win, lose, score, foodScore, and ghost.
    Best case scenario is win, so win contributes the most to an evaluationFunction.
    Score is second important, so it scales up by 10K.
    In a state, we want to get rid of food, capsules, and ghosts (when they
    are to be eaten) so as Pacman gets closer, their respective evaluation
    scales down.
    Avoidance of ghosts is important, but not that important. Ghosts do not
    run faster than Pacman, and are only a real threat if they are 1 step away.
    A light heuristic to get far away from the ghosts as necessary (without
    being too scared from getting food) is all that is needed.

    """
    pacmanPosition = currentGameState.getPacmanPosition()
    ghostPositions = currentGameState.getGhostPositions()
    ghostStates = currentGameState.getGhostStates()
    scaredTimes = [ghostState.scaredTimer for ghostState in ghostStates]
    numCapsules = len(currentGameState.getCapsules())
    foodList = currentGameState.getFood().asList()
    numFood = currentGameState.getNumFood()
    # Initialize
    badGhost = []
    yummyGhost = []
    total = 0
    win = 0
    lose = 0
    score = 0
    foodScore = 0
```

```
    foodList = currentGameState.getFood().asList()
    numFood = currentGameState.getNumFood()
    # Initialize
    badGhost = []
    yummyGhost = []
    total = 0
    win = 0
    lose = 0
    score = 0
    foodScore = 0
    ghost = 0
    if currentGameState.isWin():
        win = 10000000000000000000000000000
    elif currentGameState.isLose():
        lose = -10000000000000000000000000000
    score = 10000 * currentGameState.getScore()
    capsules = 10000000000/(numCapsules+1)
    for food in foodList:
        foodScore += 50/(manhattanDistance(pacmanPosition, food)) * numFood
    for index in range(len(scaredTimes)):
        if scaredTimes[index] == 0:
            badGhost.append(ghostPositions[index])
        else:
            yummyGhost.append(ghostPositions[index])
    for index in range(len(yummyGhost)):
        ghost += 1/(((manhattanDistance(pacmanPosition, yummyGhost[index])) * scaredTimes[index])+1)
    for death in badGhost:
        ghost +=  manhattanDistance(pacmanPosition, death)
    total = win + lose + score + capsules + foodScore + ghost
    return total
    raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

Part II. Results & Analysis