

[CSE537] The Multi-Agent Pacman

# **REPORT for TEAM PROJECT #2**

**by Oleksii Starov, Hyungjoon Koo**

**Oct. 10, 2013**

## A. General Information

- (1) Team-up (Group 41): Oleksii Starov ([ostarov@cs.stonybrook.edu](mailto:ostarov@cs.stonybrook.edu)), Hyungjoon Koo ([hykoo@cs.stonybrook.edu](mailto:hykoo@cs.stonybrook.edu))
- (2) Project information: Pacman Search (PJT#1), <http://www.cs.sunysb.edu/~ram/cse537/project01-2013.html> (Due date: Oct. 10, 2013)
- (3) Implementation: ReflexAgent Evaluation, MiniMaxAgent, AlphaBetaAgent

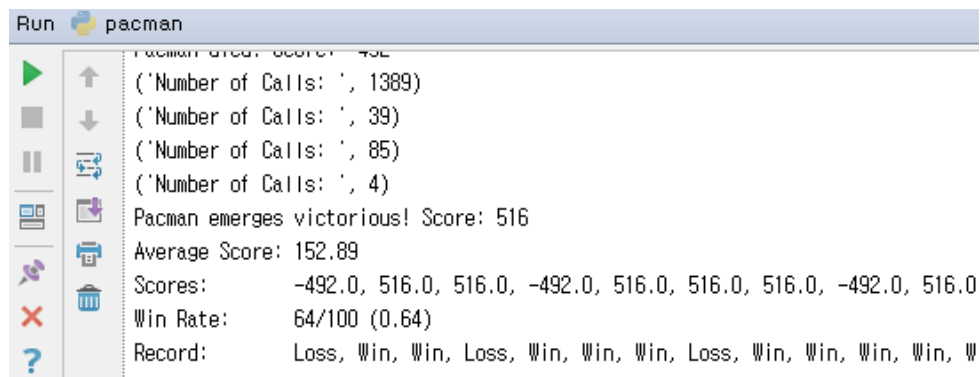
## B. Implementation Note

### (1) Reflex Agent Evaluation (Q1):

We provided weighted formula for the evaluation function which considers three parameters: number of food left, closeness of the food and minimal distance to a ghost. The priorities to get the best empirical results are 40%, 30% and 30% respectively, so that the Pacman is able to aware what direction the food is, not to be afraid of getting foods although they are not on the adjacent cells, by taking away from ghosts. This approach is common and similar to Chess evaluation, when you count the number of black and white figures left, and each type of figure has its own weight. The win rate was 64%~65%. We ran this code twice with the following command:

```
# python pacman -p MinimaxAgent -l minimaxClassic -a depth=4 -n 100 -q
```

```
Scores:      516.0, -495.0, 516.0, -492.0, 516.0, 516.0, 516.0, 516.0, -492.0, -495.0, 516.0, 516.0, -495.0, 516.0, 516.0,
516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, -492.0, 516.0, -492.0, -492.0, -492.0, 516.0, 516.0, 514.0, 516.0, -
492.0, -492.0, -492.0, 516.0, 514.0, -494.0, 516.0, 516.0, 516.0, -492.0, 516.0, -494.0, -494.0, -492.0, -492.0, -492.0, -492.0,
516.0, -494.0, 516.0, -492.0, -492.0, 516.0, -492.0, -492.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, -495.0, 516.0,
516.0, 516.0, 516.0, -492.0, -492.0, -492.0, 516.0, 516.0, -494.0, 516.0, 516.0, 516.0, 516.0, 516.0, -494.0, 516.0, -
495.0, 516.0, 516.0, 516.0, -492.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, -495.0, 516.0, 516.0, 516.0, -495.0
Win Rate:      65/100 (0.65)
Record:      Win, Loss, Win, Loss, Win, Win, Win, Win, Loss, Loss, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win,
Win, Loss, Win, Loss, Loss, Loss, Win, Win, Win, Win, Loss, Loss, Loss, Win, Win, Loss, Win, Win, Win, Loss, Win, Loss, Loss,
Loss, Loss, Loss, Loss, Win, Loss, Win, Loss, Loss, Win, Loss, Loss, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win,
Loss, Loss, Loss, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win,
Win, Loss, Win, Win, Win, Win, Loss
```



```
Run pacman
('Number of Calls: ', 1389)
('Number of Calls: ', 39)
('Number of Calls: ', 85)
('Number of Calls: ', 4)
Pacman emerges victorious! Score: 516
Average Score: 152.89
Scores:      -492.0, 516.0, 516.0, -492.0, 516.0, 516.0, 516.0, -492.0, 516.0
Win Rate:      64/100 (0.64)
Record:      Loss, Win, Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Win, W
```

```

def evaluationFunction(self, currentGameState, action):

    # Useful information you can extract from a GameState (pacman.py)

    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

    """ YOUR CODE HERE """

    # Initialize minimum distance to Ghosts
    # Find the minimum distance of Ghost which threats Pacman the most.
    minDistToGhost = 99999999
    for ghostState in newGhostStates:
        distGhost = manhattanDistance(newPos, ghostState.getPosition())
        if distGhost < minDistToGhost:
            minDistToGhost = distGhost

    # Initialize minimum distance to Foods
    # Find the minimum distance of Food which Pacman eat nearby.
    curFood = currentGameState.getFood()
    minDistToFood = 99999999
    for foodPosition in curFood.asList():
        distFood = manhattanDistance(newPos, foodPosition)
        if distFood < minDistToFood:
            minDistToFood = distFood

    # Consider three parameters to get the adjusted evaluation
    # (1) number of food left, (2) closeness of the food and (3) minimal distance to a ghost.
    # with each priorities are 40%, 30%, and 30% respectively

    adjustedEvaluation = - (len(newFood.asList()) * 0.4 + minDistToFood * 0.3 +
                           (max(successorGameState.getWalls().width, successorGameState.getWalls().height) - minDistToGhost) * 0.3)
    return adjustedEvaluation

```

## (2) MiniMax Agent (Q2):

We considered the "level" of depth to be a Max and Min recursive calls. In this problem, although many Min calls – multiple ghosts – occurred at a single game, but it counts as only one level, which we were confused at first. Here we provided two separate methods for Min and Max that invoke back and forth: one is for Max or Pacman and the other is for Min or ghosts. In this way, the code looks more readable. We defined the global variable, called *callsCount* so that we can compare the number of calls of *MiniMax* Agent with that of *AlphaBeta* Agent. (Commented below) To improve our speed of algorithm, we got rid of *STOP* state as instructed, by implementing *removeStop()* method. The method *maxValue()* takes five arguments – *gameState*, *depth*, *numOfAgents*, and *agentIndex*. Here we initialized the depth as a real depth - `self.depth * gameState.getNumAgents()`. And *agentIndex* 0 means the index of PacMan. Similarly, *minValue()* takes five parameters. As you might expect, *maxValue()* always returns the maximum value, and *minValue()* does in an opposite way.

```

class MinimaxAgent(MultiAgentSearchAgent):

    # We used this global variable to compare with the number of recursive calls of AlphaBetaAgent
    # callsCount = 0

    def getAction(self, gameState):

        """ YOUR CODE HERE """

        MinimaxAgent.callsCount = 0

        # If the game state is lost or win then just return the evaluation value
        if gameState.isLose() or gameState.isWin():
            return self.evaluationFunction(gameState)

        # Calling maxVal with game state, depth, number of agents and the agent index
        # This leads recursion calls in MiniMax tree to make a decision
        finalDirection = self.maxValue(gameState, self.depth * gameState.getNumAgents(), gameState.getNumAgents(), 0)
        # print("Number of Calls: ", MinimaxAgent.callsCount)
        return finalDirection[1]

    # The helper function to remove STOP action to improve efficiency
    def removeStop(self, legalMoves):
        stopDirection = Directions.STOP
        if stopDirection in legalMoves:
            legalMoves.remove(stopDirection)

    # The maxVal method for PacMan whose agent index is zero
    # Note that the initial depth is self.depth * gameState.getNumAgents()
    # because we considered the "level" of depth to be a Max and Min recursion calls

    def maxValue(self, gameState, depth, numOfAgents, agentIndex):
        # MinimaxAgent.callsCount += 1

        # Get the legal moves list and remove stops to improve the efficiency
        legalMoves = gameState.getLegalActions(agentIndex)
        self.removeStop(legalMoves)

        # The next states, either passed in to getAction or generated via GameState.generateSuccessor
        nextStates = [gameState.generateSuccessor(agentIndex, action) for action in legalMoves]
        if depth == 0 or len(nextStates) == 0:
            return (self.evaluationFunction(gameState), Directions.STOP)

        maxVal = -9999999
        action = legalMoves[0]
        moveIndex = 0
        chooseMax = []

        # There are only one case to call minVal from Pacman perspective
        for nextState in nextStates:
            res = self.minValue(nextState, depth-1, numOfAgents-2, numOfAgents, (agentIndex+1)%numOfAgents)
            chooseMax.append(res)

```

```

        if res > maxValue:
            maxValue = res
            action = legalMoves[moveIndex]
            moveIndex = moveIndex + 1

    return (maxValue, action)

# The minValue method for each ghost whose index is agentIndex

def minValue(self, gameState, depth, numOfMins, numOfAgents, agentIndex):
    # MinimaxAgent.callsCount += 1

    # Get the legal moves list and remove stops to improve the efficiency
    legalMoves = gameState.getLegalActions(agentIndex)
    self.removeStop(legalMoves)

    # The next states, either passed in to getAction or generated via GameState.generateSuccessor
    nextStates = [gameState.generateSuccessor(agentIndex, action) for action in legalMoves]
    chooseMin = []

    # Considering three cases:
    # 1. If there is no more depth or next states then return evaluation value
    if depth == 0 or len(nextStates) == 0:
        return self.evaluationFunction(gameState)

    # 2. If all ghosts are evaluated then call maxValue for Pacman agent
    if numOfMins == 0:
        for nextState in nextStates:
            res = self.maxValue(nextState, depth-1, numOfAgents, (agentIndex+1)%numOfAgents)
            chooseMin.append(res[0])

    # 3. If there are more ghosts left then call minValue for Ghost agent
    else:
        for nextState in nextStates:
            res = self.minValue(nextState, depth-1, numOfMins-1, numOfAgents, (agentIndex+1)%numOfAgents)
            chooseMin.append(res)

    # Return the minimum value each time
    return min(chooseMin)

```

### (3) AlphaBetaAgent (Q3):

We implemented Alpha-Beta pruning algorithm based on *MiniMax* agent by adding two additional parameters – alpha and beta - to our methods. They will be updated on each level, when new candidate is found at the result set of that level, and at the same time "cut-off" condition will be checked. As stated above, we defined the global variable, called *callsCount* so that we can compare the number of calls of *MiniMax* Agent with that of *AlphaBeta* Agent. (Commented below) In *maxValueAlphaBeta()* and *minValueAlphaBeta()*, we keep cutting off unnecessary branches to explore new nodes with alpha and beta values. It turns out there were almost 50% improvement compared to *MiniMax* agent. According to Wikipedia, "when nodes are ordered at random, the average number of nodes evaluated is roughly  $O(b^{3d/4})$ ".

```

class AlphaBetaAgent(MultiAgentSearchAgent):

    # We used this global variable to compare with the number of recurssive calls of MinimaxAgent
    # callsCount = 0

    def getAction(self, gameState):

        """ YOUR CODE HERE """
        AlphaBetaAgent.callsCount = 0
        # If the game state is lost or win then just return the evaluation value
        if gameState.isLose() or gameState.isWin():
            return self.evaluationFunction(gameState)

        # This part is almost the same with MinimaxAgent other than two additional parameters of alpha and beta values
        # by recursion calls in MiniMax tree to make a decision
        # alpha = the best choice of the highest values we have found so far along the path for MAX.
        # (initializes -9999999)
        # beta = the best choice of the lowest values we have found so far along the path for MIN.
        # (initializes +9999999)

        finalDirection = self.maxValueAlphaBeta(gameState, self.depth * gameState.getNumAgents(), gameState.getNumAgents(),
            0, -9999999, 9999999)
        print("Number of Calls: ", AlphaBetaAgent.callsCount)
        return finalDirection[1]

    # The helper function to remove STOP action to improve efficiency
    def removeStop(self, legalMoves):
        stopDirection = Directions.STOP
        if stopDirection in legalMoves:
            legalMoves.remove(stopDirection)

    # The maxValue method for PacMan whose agent index is zero
    def maxValueAlphaBeta(self, gameState, depth, numOfAgents, agentIndex, alpha, beta):

        # AlphaBetaAgent.callsCount += 1
        legalMoves = gameState.getLegalActions(agentIndex)
        self.removeStop(legalMoves)

        # The next states, either passed in to getAction or generated via GameState.generateSuccessor
        nextStates = [gameState.generateSuccessor(agentIndex, action) for action in legalMoves]
        if depth == 0 or len(nextStates) == 0:
            return (self.evaluationFunction(gameState), Directions.STOP)

        maxValue = -9999999
        action = legalMoves[0]
        moveIndex = 0
        chooseMax = []

        # There are only one case to call minValue from Pacman perspective
        for nextState in nextStates:
            res = self.minValueAlphaBeta(nextState, depth-1, numOfAgents-2, numOfAgents, (agentIndex+1)%numOfAgents, alpha, beta)
            chooseMax.append(res)

```

```

        if res > maxValue:
            maxValue = res
            action = legalMoves[moveIndex]
            # Check "cut-off" condition
            if res > alpha: # If alpha is smaller than the current Max value
                alpha = res # Then replace alpha with the current Max value
            if alpha >= beta: # When alpha value is larger than beta value
                return (alpha, action) # beta cut-off
        moveIndex = moveIndex + 1

    return (maxValue, action)

# The minValue method for each ghost whose index is agentIndex
def minValueAlphaBeta(self, gameState, depth, numOfMins, numOfAgents, agentIndex, alpha, beta):

    # AlphaBetaAgent.callsCount += 1

    legalMoves = gameState.getLegalActions(agentIndex)
    self.removeStop(legalMoves)

    # The next states, either passed in to getAction or generated via GameState.generateSuccessor
    nextStates = [gameState.generateSuccessor(agentIndex, action) for action in legalMoves]
    chooseMin = []

    # Considering three cases:
    # Case 1. If there is no more depth or next states then return evaluation value
    if depth == 0 or len(nextStates) == 0:
        return self.evaluationFunction(gameState)

    # Case 2. If all ghosts are evaluated then call maxValue for Pacman agent
    if numOfMins == 0:
        for nextState in nextStates:
            res = self.maxValueAlphaBeta(nextState, depth-1, numOfAgents, (agentIndex+1)%numOfAgents, alpha, beta)
            chooseMin.append(res[0])
            # Check "cut-off" condition
            if min(chooseMin) < beta: # If beta is larger than the current Min value
                beta = min(chooseMin) # Then replace beta with the current Min value
            if alpha >= beta: # When alpha value is larger than beta value
                return alpha #alpha cut-off

    # Case 3. If there are more ghosts left then call minValue for Ghost agent
    else:
        for nextState in nextStates:
            res = self.minValueAlphaBeta(nextState, depth-1, numOfMins-1, numOfAgents, (agentIndex+1)%numOfAgents, alpha, beta)
            chooseMin.append(res)
            # Check "cut-off" condition
            if min(chooseMin) < beta: # If beta is larger than the current Min value
                beta = min(chooseMin) # Then replace beta with the current Min value
            if alpha >= beta: # When alpha value is larger than beta value
                return alpha #alpha cut-off

    return min(chooseMin)

```

### C. The comparison between *MinimaxAgent* and *AlphaBetaAgent* in terms of number of recursive calls

The table shows the number of calls as steps go between pruning and naïve *Minimax*. Well, we can just estimate the reduction rate of calls since each step might have different states. The below is the result of depth 4 or real depth 16.

Step	After Pruning	Naïve Minimax	Rate
1	147	259	-43%
2	240	964	-75%
3	233	905	-74%
4	241	1203	-80%
5	230	621	-63%
6	210	147	43%
7	245	232	6%
8	364	532	-32%
9	153	595	-74%
10	234	481	-51%
Avg			-44%

The graph shows clearly that alpha beta pruning method reduces the complexity as much as half times on average. (-40~50% in depth 3)

```
# python pacman -p AlphaBetaAgent -a depth=3 -l smallClassic --frameTime 0
```

