Report

109550005 張可晴

Explanation of code

Part1-1

```
def MinMaxLevel(gameState, depth, agentIndex):
    if agentIndex == 0:
        if gameState.isWin() or gameState.isLose() or depth + 1 == self.depth:
            return self.evaluationFunction(gameState)
        val = []
        actions = gameState.getLegalActions(0)
        val = [MinMaxLevel(gameState.getNextState(0,action), depth + 1, 1) for action in actions]
        return max(val)
    else:
        if gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)
        val = []
        actions = gameState.getLegalActions(agentIndex)
        if agentIndex < gameState.getNumAgents() - 1:
            val = [MinMaxLevel(gameState.getNextState(agentIndex, action), depth, agentIndex+1) for action in actions]
        else:
            val = [MinMaxLevel(gameState.getNextState(agentIndex,action), depth, 0) for action in actions]
        return min(val)

score = []
actions = gameState.getLegalActions(0)
score = [MinMaxLevel(gameState.getNextState(0, action), 0, 1) for action in actions]
return actions[score.index(max(score))]
```

To finish this part, I additionally built a subfunction called “MinMaxLevel”. If the agentIndex = 0, meaning the agent is pacman. Firstly, we need to check whether this run is the base case by checking if we are win or lose this game or the next depth is as deep as the self.depth. If one of the condition mentioned above is right, return the score evaluated by “self.evaluationFunction()”. Otherwise, we go to the next depth which will return minimal value to us for all legal action generated by “getLegalAction()”. Finally, return the maximal value among all of the next depth’s returning values. If agentIndex > 0, representing the agent is ghost. What we need to do firstly is similar to what the pacman does, except for that the condition of base case is merely whether we are win or lose. And we need to go through all this depth’s ghosts to get its value from the recursive function. Moreover, if the ghost is the last one in that depth we change to call the pacman at the same depth to get the maximal value.

After defining the “MinMaxLevel” subfunction, I put all the 0-depth pacman’s possible actions to the “MinMaxLevel” and keep the score in a list. Last but not least, return the action having greatest score.

Part1-2

```
def ExpectMaxLevel(gameState, depth, agentIndex):
    if agentIndex == 0:
        if gameState.isWin() or gameState.isLose() or depth + 1 == self.depth:
            return self.evaluationFunction(gameState)

        val = []
        actions = gameState.getLegalActions(0)
        val = [ExpectMaxLevel(gameState.getNextState(0, action), depth + 1, 1) for action in actions]
        return max(val)
    else:
        if gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)
        actions = gameState.getLegalActions(agentIndex)
        if len(actions) == 0:
            return 0
        ExpectiVal = []
        if agentIndex < gameState.getNumAgents() - 1:
            ExpectiVal = [ExpectMaxLevel(gameState.getNextState(agentIndex, action), depth, agentIndex + 1) for action in actions]
        else:
            ExpectiVal = [ExpectMaxLevel(gameState.getNextState(agentIndex, action), depth, 0) for action in actions]
        return float(sum(ExpectiVal) / len(actions))

actions = gameState.getLegalActions(0)
score = []
score = [ExpectMaxLevel(gameState.getNextState(0, action), 0, 1) for action in actions]
return actions[score.index(max(score))]
```

What we need to do in this part is similar to the last part. the only difference is that in the subfunction “ExpectMaxLevel” when the agent is ghost, instead of finding the minimal value over all the ghost’s possible actions, I return the value that the sum of all the ghosts’ value divided by the number of all the ghosts’ possible actions to realize what the spec said “choose among its legal actions uniformly at random”.

Part1-3

```
def betterEvaluationFunction(currentGameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (part1-3).

    DESCRIPTION:
        1. calculate the manhattan diastance between pacman and foods
        3. the closer the food not eaten is, the greater the score is
        4. general score = currentScore + eaten food + food not eaten
        5. calculate the manhattan diastance between pacman and ghosts
        6. if normal ghost, the closer the ghost is, the lower the score is
        7. Otherwise, the closer the ghost is, the greater the score is. and the scaredTime will also considered

    """
    "*** YOUR CODE HERE ***"
    pacPos = currentGameState.getPacmanPosition()
    foodPos = currentGameState.getFood().asList()
    food2pacPos =[manhattanDistance(pacPos, food) for food in foodPos if manhattanDistance(pacPos, food) > 0]
    foodEaten = len(currentGameState.getFood().asList(False))

    if len(food2pacPos) != 0:
        score = currentGameState.getScore() + (1.0 /min(food2pacPos)) + foodEaten
    else:
        score = currentGameState.getScore() + foodEaten

    ghosts = currentGameState.getGhostStates()
    ghost2pacPos =[manhattanDistance(pacPos, ghost.getPosition()) for ghost in ghosts]
    scaredTimes = [ghostState.scaredTimer for ghostState in currentGameState.getGhostStates()]


    for scaredTime, ghost in zip(scaredTimes, ghost2pacPos):
        if scaredTime == 0 :
            if ghost == 0:
                return 0
            else:
                score += ghost
        else:
            score += scaredTime + (-1 * ghost)

    return score
```

As the description I wrote as the comment. For starter, I get the pacman’s position by calling “currentGameState.getPacmanPosition()”, getting the positions of foods by “currentGameState.getFood().asList()”, and calculate the Manhattan distance

between pacman and foods by calling the function “manhattanDistance()”. After that, if the size of the list which store the Manhattan distance is zero, all the food were eaten by the pacman, so we just plus the current score got by “getScore()” and the number of the foods as the score. Otherwise, we need to add the reciprocal of the closest food that hasn’t been eaten by the pacman as well. Next, let’s calculate the Manhattan distance between the pacman and ghosts and keep them in a list. In addition, we also keep the scared time of each ghost in a list. Finally, we run a for loop to go through both lists mentioned above. If the scared time is zero and the distance between pacman and ghost is zero, I will return zero since the pacman is captured by the ghost. If the scared time equals to zero and the distance between pacman and ghost is not zero, we add up the score with the distance because the greater the distance is when the ghost is normal, the higher the score is. if the scared time is not zero, which means the ghost is scared. Hence, we can add up the score with scared time and the reciprocal of the distance between the pacman and ghost, as when the ghost is scared, the earlier the time we capture it, the higher the score is. Meanwhile, the lower the distance is, the higher the score is, so we subtract the distance from the score.

Part2-1

```
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    nextStatePr = self.mdp.getTransitionStatesAndProbs(state, action)
    val = [pr*(self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState]) for nextState, pr in nextStatePr]
    return sum(val)
    # End your code
```

$\sum_{s'} \mathrm{P}(s'|s;a)[R(s,a,s') + \gamma V^*(s')]$ I just realize this formula in this part. At the beginning, I keep all the probability of the next state given on the current state and action in a list. Secondly, run a for loop depending on the probability, multiply the probability and the sum of the reward plus the discount value multiply by the value of the next state, and store the value in a list. Finally, return the summation of the list.

```
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    maxVal = float('-inf')
    returnAction = None
    for action in self.mdp.getPossibleActions(state):
        val = self.computeQValueFromValues(state, action)
        if maxVal < val:
            maxVal = val
            returnAction = action

    return returnAction
    # End your code
```

$\pi^*(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s;a)[R(s,a,s') + \gamma V^*(s')];$ In this part, I return the action with the greatest value by going through all the possible actions, using the "computeQValueFromValues" to get the corresponding value, setting a variable maxVal to keep the maximal value for comparison, and keep the action with maximal value in returnAction. Moreover, if it's terminal state, we just return None.

```
def runValueIteration(self):
    # Write value iteration code here
    """
      calculate the best state total reward given the best action reward
      for the iteration time accoreding to formula
    """
    # Begin your code
    for iteration in range(self.iterations):
        tmp = util.Counter()
        for state in self.mdp.getStates():
            maxVal = float('-inf')
            for action in self.mdp.getPossibleActions(state):
                val = self.computeQValueFromValues(state, action)
                maxVal = max(maxVal, val)
                tmp[state] = maxVal
        for state in self.mdp.getStates():
            self.values[state] = tmp[state]

    # End your code
```

for $h \leftarrow 0$ to $H-1$ do
  foreach s do
    $V^*(s) \leftarrow \max_a \sum_{s'} P(s'|s;a)[R(s,a,s') + \gamma V^*(s')];$
  end
end

For this function, we need to realize value iteration algorithm. the number of step we need to go through to get the total reward is self.iteration. Firstly, I construct a dictionary to store the maximal value and its corresponding state temporarily. After that, we go through all the states and their actions. Use "computeQValueFromValues()" to get the value and keep the maximal

values in the dictionary. Last but not least, keep the value of temporary dictionary into self.values.

Part2-2

```
def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we have never seen a state
      or the Q node value otherwise
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    return self.qvalue[state, action]
    # End your code
```

Get the Q-value from the qvalue dictionary according to state and action.

```
def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    actions = self.getLegalActions(state)
    val = []

    if len(actions) == 0: #terminal state
        return 0.0
    else:
        val = [self.getQValue(state, action) for action in actions]

    return max(val)

    # End your code
```

Depend on the given state, we go through all the actions, getting its Q-value by “getQValue()”, and keep them in a list. Return the maximal value in the list eventually. what’ more, if the state is the terminal one, return zero.

```
def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    actions = self.getLegalActions(state)
    maxValAction = []
    maxVal = float('-inf')

    if len(actions) == 0:
        return None
    else:
        for action in actions:
            if self.getQValue(state, action) > maxVal:
                maxVal = self.getQValue(state, action)
                maxValAction = [action]
            elif self.getQValue(state, action) == maxVal:
                maxValAction.append(action)
        return random.choice(maxValAction)
    # End your code
```

This part is similar to what the last part does, except for that if it's not the terminal state, we keep all the actions with maximal Q-value in a list in advance. Finally, return one of them randomly by random.choice()

```
def update(self, state, action, nextState, reward):
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here

      NOTE: You should never call this function,
      it will be called on your behalf
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    oldQ = self.getQValue(state, action)
    maxNewQ = self.computeValueFromQValues(nextState)

    self.qvalue[state, action] = oldQ + self.alpha * (reward + self.discount * maxNewQ - oldQ)
    # End your code
```

$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$ I use "getQValue()" to get the original Q-value and "computeValueFromQValues" to get the maximal Q-value from the next state. After that, update the Q-value calculated as the formula shows.

Part2-3

```
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.

      HINT: You might want to use util.flipCoin(p)
          returns True with probability p and False with probability 1-p
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    # Begin your code
    if len(legalActions) == 0:
        return action
    else:
        if util.flipCoin(1 - self.epsilon):
            return self.computeActionFromQValues(state)
        else:
            return random.choice(legalActions)
    # End your code
```

$$\pi_{\text{act}}(s) = \begin{cases} \arg\max_{a \in \text{Actions}} \hat{Q}_{\text{opt}}(s,a) & \text{probability } 1-\epsilon, \\ \text{random from Actions}(s) & \text{probability } \epsilon. \end{cases}$$

In this part, we realize the epsilon-greedy policy. If the probability of (1- epsilon) is true, return the action with maximal Q-value by calling "computeActionFromQValues()". Otherwise, return any of the action randomly.

Part2-4

```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator

      getFeatures(self, state, action) : Returns a dict from features to counts
      Usually, the count will just be 1.0 for indicator functions.

    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get weights and feature
    features = self.featExtractor.getFeatures(state, action)
    qValue = []
    qValue = [self.weights[feature] * val for feature, val in features.items()]
    return sum(qValue)
    # End your code
```

$$Q(s,a) = \sum_{i}^{n} f_i(s,a) w_i$$

As the formula shows, we get all the features corresponding to the state and action. Then, make them do dot product with the weights

given on the feature.

```
def update(self, state, action, nextState, reward):
    """
       1. Should update your weights based on transition
       2. update qvalue
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    nextActions = self.getLegalActions(nextState)
    valQw = []
    maxValQw = 0

    if len(nextActions) != 0:
        valQw = [self.getQValue(nextState, nextAction) for nextAction in nextActions]
        maxValQw = max(valQw)

    correction = (reward + self.discount * maxValQw) -  self.getQValue(state, action)
    features = self.featExtractor.getFeatures(state, action)
    for feature, val in features.items():
        self.weights[feature] += self.alpha * correction * val
    # End your code
```

$$w_i \leftarrow w_i + \alpha[correction] f_i(s,a)$$
$$correction = (R(s,a) + \gamma V(s')) - Q(s,a)$$

To begin with, if the current state is not the terminal one, we get the maximal Q-value by "getQValue" defined in this part and pick up the maximal one. Otherwise, the maximal Q-value will be zero. After that calculate the correction as the formula shows. Eventually, update the new weight.

Discussion over observation

Part1-1

Make sure you understand why pacman rushes to the closest ghost in this case:

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

In this condition, as the minimax algorithm, there is no opportunity for pacman to win, so it will rush to the ghost to prevent from reduction of the score as the game continues.

- The minimax values of the initial state in the minimaxClassic layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win, despite the dire prediction of depth 4 minimax, whose command is shown below. Our agent wins 50-70% of the time: Be sure to test on a large number of games using the -n and -q flags.

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

n	Win rate
100	61%
200	54%
300	56%

From this table, we can observe that its win rate falls in 50 – 70 %

Part1-2

To see how the ExpectimaxAgent behaves in Pac-Man, run the following command:

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
```

You should observe a more cavalier approach in close quarters with ghosts.

I find that even if the pacman seems to be trapped it will still try hard to escape from being captured and eat some foods by the GUI.

In particular, if pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Investigate the results of the cases:

```
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

You should find that your ExpectimaxAgent wins about half the time. Make sure you understand why the behavior here differs from the minimax case.

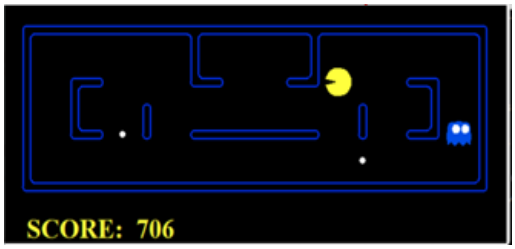
```
$ python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Average Score: 428.6
Scores:        532.0, 532.0, 532.0, -502.0, 532.0, 532.0, 532.0, 532.0, 532.0, 532.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win
```

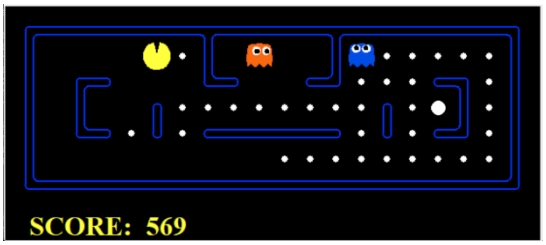
the win rate is 90%, which is over half the time as the spec mentioned.

By contrast with the minimax algorithm, the expectimax will take the condition that maybe the ghost merely hunts around into consideration. As a result, pacman won't rush to the ghost as the minimax does.

Part1-3

With depth 2 search, your evaluation function should clear the smallClassic layout with one random ghost more than half the time and still run at a reasonable rate.






```
$ python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -a depth=2 -k 1
Pacman emerges victorious! Score: -331
Average Score: -331.0
Scores:        -331.0
Win Rate:      1/1 (1.00)
Record:        Win
```

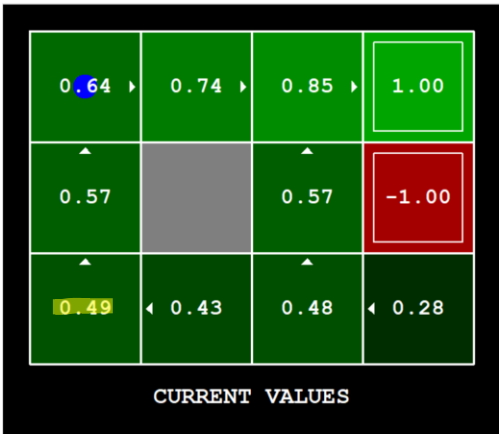
```
$ python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -a depth=2
Pacman emerges victorious! Score: 1266
Average Score: 1266.0
Scores:        1266.0
Win Rate:      1/1 (1.00)
Record:        Win
```

I observe two cases under the condition that layout = smallClassic, depth = 2, and there is only on ghost or two ghosts. While under the one-ghost condition, the rate to win the game is much lower that two-ghosts one. As a result, the former's score is much lower that the latter's one.

Part2-1

The following command loads your ValueIterationAgent, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a value -i 100 -k 10
```




```
AVERAGE RETURNS FROM START STATE: 0.4949546229000001
```

We can observe that V(start) = 0.49 is quite close to the empirical resulting average reward = 0.4949546229000001

Part2-3

After implementing the getAction method, observe the following behavior of the agent in gridworld.

```
python gridworld.py -a q -k 100
```

Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predicted because of the random actions and the initial learning phase.

	Return
Episode = 1	-0.03815204244769462

```
EPISODE 1 COMPLETE: RETURN WAS -0.03815204244769462
```

```
EPISODE 60 COMPLETE: RETURN WAS 0.5904900000000002
```

```
EPISODE 100 COMPLETE: RETURN WAS 0.47829690000000014

AVERAGE RETURNS FROM START STATE: 0.1920880002409935
```

Episode = 60	0.5904900000000002
Episode = 100	0.47829690000000014
Average	0.1920880002409935

Due to the epsilon-greedy policy, sometimes it will return low value owning to random selection, and thus the average return will lower than the Q-values predicted.

You can also observe the following simulations for different epsilon values. Does that behavior of the agent match what you expect? (discuss it in your report)

	Average returns
Epsilon = 0.9	0.03912515573327896
Epsilon = 0.5	0.291344706309012
Epsilon = 0.1	0.47788947809197824

This result meets my expectation. Because the higher the epsilon is, the lower the probability that we can get the action with maximal Q-value is. Therefore, the higher the epsilon is, the lower the average return is.

Part2-4

By default, ApproximateQAgent uses the `IdentityExtractor`, which assigns a single feature to every (state, action) pair. With this feature extractor, your approximate Q-learning agent should work identically to PacmanQAgent. You can test this with the following command:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

ApproximateQAgent

```
Average Score: 499.8
Scores:        503.0, 503.0, 503.0, 495.0, 495.0, 499.0, 503.0, 499.0, 503.0, 49
5.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

PacmanQAgent

```
Average Score: 500.1
Scores:        503.0, 503.0, 499.0, 499.0, 502.0, 499.0, 503.0, 499.0, 499.0, 49
5.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Once you're confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid
```

```
$ python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60
 -l mediumGrid
Beginning 50 episodes of Training
Training Done (turning off epsilon and alpha)
-----------------------------------------
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 523
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 529
Average Score: 527.6
Scores:        527.0, 529.0, 523.0, 529.0, 529.0, 527.0, 529.0, 527.0, 527.0, 52
9.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

As the spec predicts, the win rate is 100%.

```
$ python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic
Beginning 50 episodes of Training
Training Done (turning off epsilon and alpha)
-----------------------------------------
Pacman emerges victorious! Score: 1338
Pacman emerges victorious! Score: 1339
Pacman emerges victorious! Score: 1340
Pacman emerges victorious! Score: 1343
Pacman emerges victorious! Score: 1344
Pacman emerges victorious! Score: 1331
Pacman emerges victorious! Score: 1330
Pacman emerges victorious! Score: 1341
Pacman emerges victorious! Score: 1331
Pacman emerges victorious! Score: 1311
Average Score: 1334.8
Scores:        1338.0, 1339.0, 1340.0, 1343.0, 1344.0, 1331.0, 1330.0, 1341.0, 1331.0, 1311.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

I choose larger layout size by replacing medium grid with medium classic, it still wins easily.

Comparison with the performance of every method and discussion

	Average Score	Win Rate (%)
Minimax	-59.375	15
Expectimax	103.475	21
Q-learning (epsilon = 0.05, alpha = 0.2)	-416.4	0
Approximate Q-learning	748.73	80
DQN	1233.67	77

All the statics shown above is under the condition that layout = smallClassic, n = 200 (x = 100).

In the light of Minimax search and Expectimax search, we can observe that both the average score and the win rate of the latter outperform the former.

As for Q-learning and approximate Q-learning, we can find that it's weird that the former's win rate is zero. In my humble opinion, it's all because that the layout is too large to pick up optimal policy since there are too many choices. By contrast, for approximate Q-learning, we can get high average score and win rate simultaneously. I

believe that the reason is because it is provided with a set of features, it can choose the optimal state as the next one depending on the updated weight.

Moreover, the search method will perform worse than Q-learning. I think it's because that Q-learning is a kind of reinforcement learning that it will maximize the reward based on the environment, while the search method won't.

To solve the problem like we met in smallClassic for Q-learning, which the information being too many to store and calculate leads to the failure occurs. Deep Q-learning uses the deep neural network to get the Q-value rather then construct a table to keep the value.

It is surprise that DQN doesn't perform better than approximate Q-learning. Nonetheless, I guess that it may because the lack of my computer's resource since my roommate's win rate is 88% indicating that DQN can outperform Q-learning.

Describe problems you meet and how you solve them

1. In part 1-1, I first thought that I needed to perform minimax search with alpha-beta pruning. Therefore, I couldn't get the full score until I used the vanilla minimax algorithm.
2. Part 1-3 cost me lots of time since I didn't take the scared time into consideration initially. After that, I still couldn't get the score as I plused the score with the reciprocal of the distance between the pacman and the ghost. Eventually, I turned to subtract the distance to get the full score.
3. In part 2-1, I continuously got error in "runValueIteration" function. Not until I added up a temporary dictionary to store the value did I get a right result.
4. In part 2-4, I had no idea why I couldn't get the full score even if I tried many different methods to implement it.
5. When I tried to do comparison in the report, I often run into a problem that I gave the incorrect command to the bash since I wanted to make all method return the result under the same condition.