report of AI_HW3　109550027 紀竺均

a. implementation and explanation

Part 1 : Adversarial search

1-1: Minimax Search

```
def getAction(self, gameState):
    # Begin your code
    #util.raiseNotDefined()
    ind = 0
    actions = gameState.getLegalActions(ind)
    values = []
    for action in actions: #for every possible action
        nextstate = gameState.getNextState(ind, action) #compute nextstate of current action
        values.append(self.minimax(nextstate,0,1)) #initial with depth=0 and index=1
        #go to minimax function
    maxvalue = max(values) #return max value for pacman (index=0)
    return actions[values.index(maxvalue)]

def minimax(self, gameState, depth, ind):
    actions = gameState.getLegalActions(ind)
    values = []
    if (depth==self.depth or len(actions)==0 or gameState.isWin() or gameState.isLose()):
        return scoreEvaluationFunction(gameState) #return the value if 1)arrive the last depth
                                                  # 2)no legal actions
                                                  # 3)game end (win/lose)
    if ind == 0: # compute agent pacman => return max value
        for action in actions:
            nextstate = gameState.getNextState(ind, action)
            values.append(self.minimax(nextstate,depth,1)) #next agent = first ghost (index=1)
        return max(values)
    else: # compute ghost agents => return min value
        for action in actions:
            nextstate = gameState.getNextState(ind, action)
            if ind == gameState.getNumAgents()-1: #the last ghost agent
                values.append(self.minimax(nextstate,depth+1,0)) #compute pacman's next move
            else: #other ghost agents
                values.append(self.minimax(nextstate,depth,ind+1)) #compute next ghost
        return min(values)
    # End your code
```

1-2: Expectimax Search

Almost same as minimax search, but instead of return min value for ghost agents, return the average values. This is because we assume the ghosts will choose their action randomly, every action will have the same probability.

```
    else:# compute ghost agents => return avg value
        for action in actions:
            nextstate = gameState.getNextState(ind, action)
            if ind == gameState.getNumAgents()-1:#the last ghost agent
                values.append(self.expectimax(nextstate,depth+1,0))#compute pacman's next move
            else: #other ghost agents
                values.append(self.expectimax(nextstate,depth,ind+1)) #compute next ghost
        total=0.0
        for value in values:
            total+=value
        return total/len(values)
    # End your code
```

1-3 better evaluation function

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

    DESCRIPTION: <write something here so we know what you did>
    1) if this game win, I'll give a high score to reward.
    2) if lose, I'll give a low score to avoid.
    3) consider the distance to foods and ghosts, return corresponding score
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    score = currentGameState.getScore()
    if(currentGameState.isWin()):
        return 55555
    elif(currentGameState.isLose()):
        return -55555
    else:
        pacman = currentGameState.getPacmanPosition()
        ghosts = currentGameState.getGhostPositions()

        foods = currentGameState.getFood().asList()
        ghost_dist=[]
        food_dist=[]

        for ghost in ghosts:
            ghost_dist.append(util.manhattanDistance(pacman, ghost)) #distance with ghosts
        for food in foods:
            food_dist.append(util.manhattanDistance(pacman, food)) #distance with foods
        closest_ghost = min(ghost_dist)
        closest_food = min(food_dist)
        if(closest_ghost<=2): #if the ghost is very close
            if(len(currentGameState.getCapsules())>0): #but we still get capsules
                score-=50 # a little penalty, wish it seek for the capsule
            else: #ghost is close and no capsules left
                score-=1000
        if(len(foods)<=3 and closest_food<=2): #few last food
            closest_food*=0.01 # rush to it
        return score+10.0/closest_food
```

Part 2 : Q-learning

2-1: Value Iteration

The value iteration agent construct the value dictionary of every state base on Markov Decision Processes(MDP).

The formula can be written as:

$$V^*(s) = \max_a \sum_{s'} P(s'|s;a)[R(s,a,s') + \gamma V^*(s')]$$

where P(s’|s;a) is the probs, R(s,a,s’) is the rewards, and γ stands for discount.

```
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
    #util.raiseNotDefined()
    total=0.0
    statesandprobs = self.mdp.getTransitionStatesAndProbs(state,action) #get nextstates and probabilities
    for stateandprob in statesandprobs:
        nextstate = stateandprob[0]
        prob = stateandprob[1]
        # the formula (written in report)
        total+=prob*(self.mdp.getReward(state,action,nextstate)+self.discount*self.values[nextstate])
    return total
    # End your code
```

Then, value iteration function. For #iterations, we should sum up all the states' values and return self.values for every state.

```
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    states = self.mdp.getStates()
    for i in range(self.iterations):
        vals = util.Counter() #dictionary with all zeros #vals: temp dict
        for state in states:
            if(self.mdp.isTerminal(state)): #no legal action
                vals[state]=0
                continue
            maxvalue = -float('inf')
            actions = self.mdp.getPossibleActions(state) #get actions of the state
            for action in actions:
                total = self.computeQValueFromValues(state,action) #get Q value!!
                if(total>maxvalue):
                    maxvalue = total #keep track of the max value
                    vals[state]=maxvalue #put in the dict
        #after go thru every state, put them in self.value.
        for state in states:
            self.values[state]=vals[state]
    # End your code
```

Finally, compute_action_from_values function will compute actions corresponding to the max Q-value.

```
def computeActionFromValues(self, state):
    "*** YOUR CODE HERE ***"
    # Begin your code
    actions = self.mdp.getPossibleActions(state)
    values = []
    if(len(actions)==0):
        return None
    for action in actions:
        values.append(self.computeQValueFromValues(state,action))
    maxvalue = max(values)
    return actions[values.index(maxvalue)]
```

Part 2-2: Q-learning (10%)

Q-learning updates Q-value base on Q-real and Q-estimate.

```
def computeActionFromQValues(self, state):
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    actions = self.getLegalActions(state)
    if(len(actions)==0): #no legal actions, return action = None
        return None
    qvalues = []
    for action in actions:
        qvalues.append(self.getQValue(state,action)) # get Qvalue of action
    maxvalue = max(qvalues)
    #put all the actions with maxvalue into a list 'maxaction'
    maxaction = [actioni for actioni in actions if qvalues[actions.index(actioni)] == maxvalue]
    return random.choice(maxaction) #random choose an action from maxaction
    # End your code
```

```python
def computeValueFromQValues(self, state):
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    actions = self.getLegalActions(state)
    if(len(actions)==0): #no legal actions
        return 0.0
    qvalues = []
    for action in actions:
        qvalues.append(self.getQValue(state,action))
    return max(qvalues)
    # End your code
```

update Q-value's formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

```python
def update(self, state, action, nextState, reward):
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    qestimate = self.getQValue(state,action)
    qreal = self.computeValueFromQValues(nextState)
    a = self.alpha
    r = self.discount
    self.qvalue[(state,action)] = qestimate+a*(reward+r*qreal-qestimate)
    # End your code
```

Part 2-3: epsilon-greedy action selection (5%)

```python
def getAction(self, state):
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    if(len(legalActions)==0): #terminal state
        return None
    if(util.flipCoin(self.epsilon)): #return true with probability epsilon
        return random.choice(legalActions) #randomly pick up an action
    else: #return true with probability 1-epsilon
        return self.computeActionFromQValues(state) # follow current best action
    # End your code
```

Part 2-4: Approximate Q-learning (10%)

Q-value formula:

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

```python
def getQValue(self, state, action):
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get weights and feature
    #util.raiseNotDefined()
    ext = self.featExtractor
    features = ext.getFeatures(state,action)
    qvalue=0
    for feature,value in features.items():
        qvalue += self.weights[feature]*value
    return qvalue
    # End your code
```

update formula:

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

```
def update(self, state, action, nextState, reward):
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    ext = self.featExtractor
    features = ext.getFeatures(state,action)
    qestimate = self.getQValue(state,action)
    qreal = self.computeValueFromQValues(nextState)
    for feature,value in features.items():
        correction = reward+self.discount*qreal-qestimate
        self.weights[feature] = self.weights[feature]+self.alpha*correction*value
    # End your code
```

b. Observations

Part1: Adversarial Search

1-1: Minimax Search

python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3 -n 10

result:

```
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

Pacman think that it’s impossible to win this game (to escape), so it rushes to the closest ghost because the longer it stays alive, the lower score it will get.

python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 –n 100

result:

```
Average Score: 172.63
Scores:        516.0, -495.0, 516.0, -492.0, 516.0, -494.0, 516.0, -495.0, 516.0, 516.0, 516.0, -492.0, 516.0, 516.0, 51
6.0, 516.0, 516.0, 516.0, 516.0, 516.0, 514.0, 513.0, 516.0, 516.0, 516.0, -495.0, 516.0, 516.0, 516.0, -492.0, 516.0, 5
13.0, 516.0, -492.0, -492.0, -492.0, 516.0, 513.0, -495.0, 514.0, 516.0, 514.0, 516.0, -492.0, 516.0, -492.0, 516.0, 516
.0, -495.0, 516.0, 516.0, -495.0, -492.0, 516.0, 516.0, -492.0, -492.0, 516.0, -492.0, -495.0, -492.0, -492.0, 516.0, 51
6.0, 516.0, -492.0, 516.0, 516.0, 513.0, 516.0, 516.0, 516.0, -495.0, 516.0, -497.0, 514.0, 516.0, 516.0, 516.0, -492.0,
 -492.0, 516.0, -495.0, 516.0, 516.0, 516.0, -492.0, 513.0, 516.0, -492.0, 516.0, -492.0, 516.0, 514.0, -495.0, -495.0,
516.0, 516.0, -492.0, 516.0
Win Rate:      66/100 (0.66)
Record:        Win, Loss, Win, Loss, Win, Loss, Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win,
Win, Win, Win, Win, Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Loss, Loss, Loss, Win, Win, Loss, Win, Win, Win, Win,
 Loss, Win, Loss, Win, Win, Loss, Win, Win, Loss, Loss, Win, Win, Loss, Loss, Win, Loss, Loss, Loss, Loss, Win, Win, Win
, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win, Win, Loss, Loss, Win, Loss, Win, Win, Win, Loss, W
in, Win, Loss, Win, Loss, Win, Win, Loss, Loss, Win, Win, Loss, Win
```

1-2: Expectimax Search

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

result:

```
Average Score: -88.4
Scores:        -502.0, -502.0, -502.0, 532.0, 532.0, 532.0, -502.0, 532.0, -502.0, -502.0
Win Rate:      4/10 (0.40)
Record:        Loss, Loss, Loss, Win, Win, Win, Loss, Win, Loss, Loss
```

Here, unlike minimax, pacman won’t suicide in the beginning because it thinks there might be some chances to escape. Thus get a better result.

python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=4 –n 100

result:

```
Average Score: 333.91
Scores:        -493.0, 516.0, -497.0, 516.0, 516.0, 516.0, 516.0, 516.0, -497.0, 516.0, 516.0, 516.0, -493.0, 516.0, 516
.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, -493.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516
.0, 516.0, 516.0, 516.0, 511.0, 516.0, 516.0, 516.0, -493.0, -493.0, 516.0, 513.0, 516.0, 516.0, 516.0, 516.0, 513.0, 51
6.0, 516.0, 516.0, 516.0, 516.0, -493.0, 516.0, 516.0, -493.0, 511.0, 516.0, 512.0, 516.0, 516.0, 512.0, 516.0, -493.0,
516.0, 516.0, 516.0, -493.0, 516.0, 511.0, -496.0, 516.0, 516.0, 516.0, 516.0, 516.0, -493.0, 516.0, 516.0, 516.0
, 516.0, 516.0, 516.0, -497.0, 516.0, 516.0, -493.0, 516.0, -493.0, -493.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 51
3.0, -493.0
Win Rate:      82/100 (0.82)
Record:        Loss, Win, Loss, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, W
in, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Loss, Win, Win, Win
, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win,
Win, Loss, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Loss, Win,
Loss, Loss, Win, Win, Win, Win, Win, Win, Win, Loss
```

1-3 better evaluation function:

```
Average Score: 577.76
Scores:        851.0, 858.0, 204.0, 833.0, 780.0, 678.0, 922.0, 839.0, 47.0, 940.0, -239.0, 1022.0, 834.0, -87.0, 1334.0, 734.0
, 1047.0, 447.0, 510.0, 221.0, -1542.0, 674.0, 732.0, 469.0, 929.0, 795.0, 924.0, 874.0, 453.0, 749.0, 887.0, -116.0, 825.0, 77
3.0, 502.0, -2530.0, 164.0, 838.0, 1049.0, -6472.0, 905.0, 1275.0, 811.0, 974.0, 185.0, 139.0, 898.0, 994.0, 932.0, -182.0, 816
.0, 832.0, 734.0, 961.0, 819.0, 563.0, 946.0, 799.0, 733.0, 1029.0, 1080.0, 842.0, 799.0, 910.0, 764.0, 1060.0, 899.0, -262.0,
876.0, 723.0, 1128.0, 1087.0, 794.0, 558.0, -1096.0, 1028.0, 946.0, 1036.0, 1009.0, -50.0, -109.0, 850.0, 1038.0, -628.0, 842.0
, 808.0, 851.0, 940.0, 465.0, 969.0, 921.0, 829.0, 719.0, 492.0, 877.0, 1005.0, 861.0, 938.0, 943.0, 1119.0
Win Rate:      94/100 (0.94)
```

Conclusion for part1:

layout = minimax classic, n=100	**Avg score**	**Win rate**
Minimax	172.63	0.66
Expectimax	333.91	0.82
Expectimax using betterEvalFn	577.76	0.94

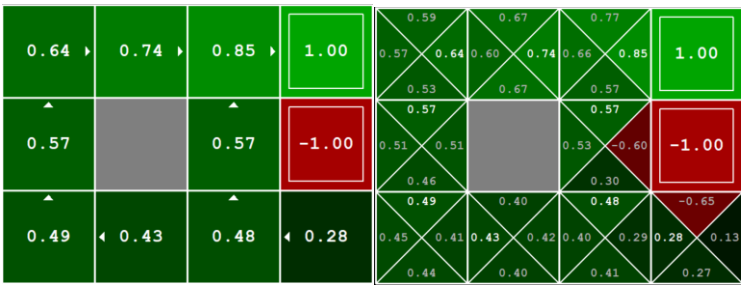
1. Minimax is a more pessimistic algorithm compared to Expectimax. It goes extreme sometimes. Because the ghost agents in this game isn't as clever as minimax thought it would be, so minimax might not be a great algorithm in this game. On the other hand, expectimax assume the ghosts act randomly. As the result, Expectimax performs better in this case.
2. Better evaluation function helps expectimax to have a higher score and win rate.

Part2:

2-1: Value Iteration

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

result: V(start) is close to avg reward!

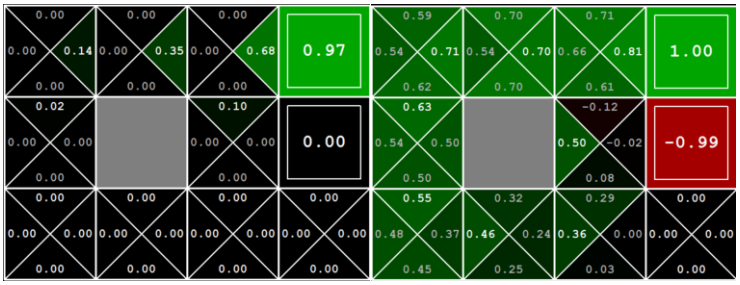



```
EPISODE 10 COMPLETE: RETURN WAS 0.43046721000000016

AVERAGE RETURNS FROM START STATE: 0.48744949500000007
```

2-2: Q-learning、2-3: epsilon-greedy action selection

python gridworld.py -a q -k 5 –m / python gridworld.py -a q -k 10




```
EPISODE 100 COMPLETE: RETURN WAS 0.0984770902183612

AVERAGE RETURNS FROM START STATE: 0.3162088296609486
```

➔ average return is lower than the Q-values predicted due to random actions and initial learning phase.

test q-learning agent on pacman

python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid –q

```
Average Score: 497.1
Scores:        499.0, 495.0, 502.0, 495.0, 495.0, 495.0, 501.0, 495.0, 499.0, 495.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

python pacman.py -p PacmanQAgent -x 50 -n 60 -l mediumGrid

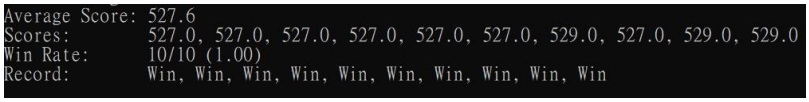
```
Average Score: -503.0
Scores:        -498.0, -500.0, -516.0, -504.0, -487.0, -490.0, -507.0, -505.0, -515.0, -508.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

2-4 Aproximate Q-learning

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

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

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



conclusion for part 2:

1.

layout = smallGrid, train = 2000, test=10	**Avg score**	**Win rate**
Q-learning	497.1	1
Approximate Q-learning	499.8	1

2.

layout = mediumGrid, train = 60, test=10	**Avg score**	**Win rate**
Q-learning	-503.0	0
Approximate Q-learning	527.6	1

➔ In my opinion, Q-learning need a lot amount of training (ex: 2000) to have a good behavior, while Approximate Q-learning can learn very fast and smart given custom feature extractor.

c. comparison of different methods

#test episode = 100	**layout**	**Avg score**	**Win rate**	**#train ep**
Minimax	MinimaxClassic	172.63	0.66	0
Expectimax	MinimaxClassic	333.91	0.82	0
Minimax	smallClassic	-122.88	0.1	0
Expectimax	smallClassic	115.23	0.19	0
Q-learning	SmallClassic	-378.61	0	2000
Approximate Q-learning	SmallClassic	823.48	0.86	2000
DQN	SmallClassic	1260.23	0.79	10000

We can conclude several things from the result:

1. Adversarial search agent performs the worst of all three. They seems to have a good behavior in simple layout like minimaxClassic, but on smallClassic layout, the win rate is quite low.
2. Q-learning's win rate is 0 in smallClassic. I guess this is because smallClassic is too complicated for Q-learning, so I test it on smallGrid:

#test episode = 100	**layout**	**Avg score**	**Win rate**	**#train ep**
Q-learning	SmallGrid	499.8	1	2000
Q-learning	SmallClassic	-378.61	0	2000

   ➔ We can observe from the result that my guess is correct! Q-learning only have a good behavior in simple layout like smallGrid, it is totally defeated in smallClassic.
3. Normally, DQN should have a better performance than normal Q-learnings, but my result doesn't fit (on aspect of win rate). I guess there are 2 possible reasons to explain this:
   (1) My computer resources. My friend has a win rate of 0.87 with the same parameters.
   (2) 10000 times of training is not enough for this neural network. A higher number of training episodes should gave a higher win rate.
4. Approximate Q-learning have a high win rate, I guess this is due to the well written of features.

d. problems encounter

However I write the better evaluation function, autograder for part1-3 return the same scores as normal evaluation function. Finally, I found out that I return scoreEvaluationFunction instead of self.evaluationFunction in expectimax.

Then, it turns out I only get 5 points in autograder part1-3. Here's why:

At first, I let the score-=500 in this condition. And I get the following result.

```
if(closest_ghost<=2): #if the ghost is very close
    if(len(currentGameState.getCapsules())>0): #but we still get capsules
```

```
Average Score: 222.34
Scores:        851.0, 143.0, 556.0, -127.0, -19033.0, 1103.0, 1077.0, 1096.0
0, 1037.0, 901.0, 951.0, -8031.0, 824.0, 1036.0, 619.0, 684.0, 830.0, 461.0,
98.0, 871.0, 858.0, 1055.0, 808.0, 671.0, 762.0, 939.0, 818.0, 1020.0, 975.0
 916.0, 821.0, 853.0, 610.0, 830.0, 903.0, 726.0, 877.0, 945.0, 983.0, 759.0
Win Rate:      93/100 (0.93)
```

The average score is lower than expectimax using normal evaluation function. I guess this is because there are some extreme low scores in some cases (-19033 for example).

Then, I consider a scenario that pacman want to eat the capsule to kill the ghosts, I realize I'm the one who stand in its way!! So, I turn the value to -50, thus get a way better result in scores.

```
if(closest_ghost<=2): #if the ghost is very close
    if(len(currentGameState.getCapsules())>0): #but we still get capsules
        score-=50 # a little penalty, wish it seek for the capsule
```

```
Average Score: 577.76
Scores:        851.0, 858.0, 204.0, 833.0, 780.0, 678.0, 922.0, 839.0, 47.0, 940.0, -239.0, 1022.0, 834.0, -87.0, 1334.0, 734.0
, 1047.0, 447.0, 510.0, 221.0, -1542.0, 674.0, 732.0, 469.0, 929.0, 795.0, 924.0, 874.0, 453.0, 749.0, 887.0, -116.0, 825.0, 77
3.0, 502.0, -2530.0, 164.0, 838.0, 1049.0, -6472.0, 905.0, 1275.0, 811.0, 974.0, 185.0, 139.0, 898.0, 994.0, 932.0, -182.0, 816
.0, 832.0, 734.0, 961.0, 819.0, 563.0, 946.0, 799.0, 733.0, 1029.0, 1080.0, 842.0, 799.0, 910.0, 764.0, 1060.0, 899.0, -262.0,
876.0, 723.0, 1128.0, 1087.0, 794.0, 558.0, -1096.0, 1028.0, 946.0, 1036.0, 1009.0, -50.0, -109.0, 850.0, 1038.0, -628.0, 842.0
, 808.0, 851.0, 940.0, 465.0, 969.0, 921.0, 829.0, 719.0, 492.0, 877.0, 1005.0, 861.0, 938.0, 943.0, 1119.0
Win Rate:      94/100 (0.94)
```

And I get 7 points in autograder part1-3. (Not the best evaluation function but a better evaluation function ☺)