Assignment 1

Problem 1

A simple algorithm to control a robot to the corners is the following: move left until the we find a wall and then move downwards until reach the bottom.
The following production system is a implementation of this algorithm.

If $\bar{s}_8$ then WEST
If $s_8\bar{s}_6$ then SOUTH

Problem 2

Knowing that the threshold is 1 and the weights vector is the following:

$$x_1 = 1.1 \quad x_2 = 3.1 \quad x_3 = -1 \quad x_4 = -2 \quad x_5 = 0.5$$

We can easily deduct that the x_5 wont have any effect on the Boolean function because when added x_5 it will never surpass the threshold. We still can detect if both x_1 and x_2 are on we exceed the threshold everytime.

If x_1 is on the way to surpass the threshold is if neither x_3 or x_4 are on, so we can deduct the following expression. We don't care if x_2 is on or off therefore we don't need to include on the expression

$$x_1\bar{x}_3\bar{x}_4$$

When the x_2 is on we never can have x_3 and x_4 on at the same time.

$$x_2\bar{x}_3x_4$$
$$x_2x_3\bar{x}_4$$

The final function is the following:

$$x_1x_2 + x_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_2x_3\bar{x}_4$$

Problem 3

1. The fitness function created simply read the train data and with their inputs computed a prediction using the weights and threshold that we are evolving. If the computed result of the perceptron matched the target we increase our score. Our goal is to get a score of 49 (number of inputs).

2. The crossover operator randomly pic a midpoint in the weights array and fill the first part (until the midpoint) with the weights of the first parent and the rest with weights of the second parent.

3. We create a new population with the crossovers from the parents.

4. We created a mutation rate variable that allow us to chose the mutation frequency. Using this varible we create a random number and if it is less than the mutation rate then we replace some weight.

5. To the first population are created 200 programs. We create each component using a randomly uniform distribution between -5 and 5.

6. The result is the following:

$$[-0.3129, -1.4176, 2.3665, -0.4281, 4.4479, 1.2509, -4.9492, -1.8977, 1.4514, -0.2548]$$

Problem 4

```
# reactiveAgents.py
from game import Directions
from game import Agent
from game import Actions
import util
import time
import search

class NaiveAgent(Agent):
    "An agent that goes West until it can't."
    def __init__(self):
        self.lap = 0
        self.orientation = 2

    def getAction(self, state):
        "The agent receives a GameState (defined in pacman.py)."
        sense = state.getPacmanSensor()

        #State variables
        i1 = sense[0]
        i2 = sense[1]
        i3 = sense[2]
        i4 = sense[3]
        i5 = sense[4]
        i6 = sense[5]
        i7 = sense[6]
        i8 = sense[7]
```

```
x1 = i2 or i3
x2 = i4 or i5
x3 = i6 or i7
x4 = i8 or i1

#Resolves the one line problem
if i1 and i2 and i3 and i5 and i6 and i7 and not i8:
    return Directions.WEST
elif i1 and i2 and i3 and i5 and i6 and i7 and i8:
    return Directions.STOP


#Resolves if the map looks like a funnel
if i7 and i5 and not i6 and self.lap != 2:
    print "down"
    self.orientation = 1  # Orientation down
    self.lap += 1
if i7 and i5 and not i6 and self.lap == 2 and self.orientation == 1 or
x4 and x2 and not i6 and self.orientation == 1:
    return Directions.SOUTH

if i1 and i3 and not i2 and self.lap != 2:
    print "up"
    self.orientation = 0  # Orientation Up
    self.lap += 1
if i1 and i3 and not i2 and self.lap == 2 and self.orientation == 0 or
x4 and x2 and not i2 and self.orientation == 0:
    return Directions.NORTH

#Follow every boundary
if x4 and not x1:
    return Directions.NORTH
elif x3 and not x4:
    return Directions.WEST
elif x2 and not x3:
    return Directions.SOUTH
elif x1 and not x2:
    return Directions.EAST
else:
    return Directions.NORTH

if x1 and x2 and x3 and x4 and not i8:
    return Directions.WEST
```

Problem 5

```
class ECAgent(Agent):
    "An agent that follows the boundary using error-correction."

    def __init__(self):
        self.north = Perceptron("../north.csv", 0.1, 2)
        self.north.train()

        self.west = Perceptron("../west.csv", 0.1, 2)
        self.west.train()

        self.south = Perceptron("../south.csv", 0.1, 2)
        self.south.train()

        self.east = Perceptron("../east.csv", 0.1, 2)
        self.east.train()

        self.cycle = 0


    def getAction(self, state):

        inputs = state.getPacmanSensor()

        isN = self.north.predict(inputs)
        isW = self.west.predict(inputs)
        isS = self.south.predict(inputs)
        isE = self.east.predict(inputs)

        if isN and not isW and not isS and not isE:
            return Directions.NORTH
        elif not isN and isW and not isS and not isE:
            return Directions.WEST
        elif not isN and not isW and isS and not isE:
            return Directions.SOUTH
        elif not isN and not isW and not isS and isE:
            return Directions.EAST

        moreThanOne = 1 if isN + isW + isS + isE > 1 else 0

        if moreThanOne:
            if self.cycle == 0:
                self.cycle = 1
                return Directions.NORTH
            elif self.cycle == 1:
                self.cycle = 2
                return Directions.EAST
            elif self.cycle == 2:
                self.cycle = 3
                return Directions.SOUTH
            elif self.cycle == 3:
```

```
                self.cycle = 0
                return Directions.WEST


        return Directions.NORTH



class Perceptron:

    def __init__(self, file, learningRate, learningEpochs):
        self.learningEpochs = learningEpochs
        self.file = file
        self.learningRate = learningRate
        self.threshold = 1
        self.weights = np.random.uniform(low=-1, high=1, size=8)
        self.train()

def train(self):
        my_data = genfromtxt(self.file, delimiter=',')
        for i in range(self.learningEpochs):
            for e in my_data:
                    inputs = e[:-1]
                    d = e[-1]
                    f = self.predict(inputs)
                    old = self.weights
                    self.weights = self.weights + \
                        self.learningRate * (d - f) * inputs
                    error = np.mean(old != self.weights)
            print "Generation -->" + str(i)

    def predict(self, inputs):
        activation = 0.0
        for i in range(len(inputs)):
            activation += inputs[i] * self.weights[i]

        return 1.0 if activation >= self.threshold else 0.0
```