Richard Robinson

Pacman Project Part 2

Professor: Ilmi Yoon
Artificial Intelligence
Due: Sunday, April 8, 2018

Files Included: MultiAgents.py
Concepts and implementation details are discussed in the problems

Problems
1) Reflex Agent
2) Minimax
3) Alpha-Beta Pruning
4) Expectimax
5) Evaluation Function

1) Reflex Agent

A reflex agent only looks at the current game state and potentially prior game states to plan its next action—it does not predict future behaviors.

It uses an evaluation function for the current choices. My evaluation function implements Manhattan distance to find the shortest distance to the next item using tail recursion (takes a list of x,y pairs).

```
def recurseManhattan(position, list):
    if len(list) == 0: return 1

    sums = []
    min = 99999
    minIndex = 0
    for xy in list:
        sums.append( (manhattanDistance(position, xy)) )

    for x in range(0, len(sums), 1):
        if sums[x] < min:
            min = sums[x]
            minIndex = x

    return min + recurseManhattan(list.pop(minIndex), list )
```

This is called on food and capsules then combined with the next option's score with some weighing to decide the next action. If a ghost

is within a range of 2 there is a huge negative score returned so the Pacman agent avoids the ghost.

```
if ( (manhattanDistance(newPos, newGhostStates[0].getPosition())) < 2 ):
    return -100000
score = .3*successorGameState.getScore() + 5.0/( recurseManhattan(newPos, newFood.asList() ))
        + 30/recurseManhattan(newPos, successorGameState.getCapsules())
return score
```

2) Minimax

Minimax searches to a certain depth, calling an evaluation function once that depth is reached or if there are no more successors. The agents can either be minimizers or maximizers. The goal of the minimizers is to follow the path that returns the lowest score. The goal of the optimizers is to follow the path that returns the highest score.

If the agent is a minimizer it calls minimize, if it is a maximizer it calls maximize. Each time all agents are parsed through it will increment the current depth as that is one turn sequence.

```
def minimax(self, gameState, passedIndex, passedDepth ):
    currentIndex = passedIndex
    currentDepth = passedDepth

    if (currentIndex == gameState.getNumAgents()):
        currentDepth += 1
        currentIndex = 0

    if ( len(gameState.getLegalActions(currentIndex)) == 0 or (currentDepth == self.depth)):
        return '', self.evaluationFunction(gameState)

    if (currentIndex == 0): return self.maximize( gameState, currentIndex, currentDepth )
    else: return self.minimize( gameState, currentIndex, currentDepth )
```

The minimize component starts with a value of infinity and an action of no action (empty string). For all possible successors it parses through them, calls minimax on them, then chooses the lowest value and the action that leads to that path.

```
def minimize(self, gameState, passedIndex, passedDepth):
    currentIndex = passedIndex
    currentDepth = passedDepth

    value = float('inf')
    action = ''

    for tempAction in gameState.getLegalActions(currentIndex):
        tempValue = self.minimax(gameState.generateSuccessor(currentIndex , tempAction), currentIndex + 1, currentDepth)[1]

        if (tempValue < value ):
            action = tempAction
            value = tempValue
    return action, value
```

The maximize component starts with a value of negative infinity and an action of no action (empty string). For all possible successors it parses through them, calls minimax on them, then chooses the highest value and the action that leads to that path

```
def maximize(self, gameState, passedIndex, passedDepth):
    currentIndex = passedIndex
    currentDepth = passedDepth

    value = -float('inf')
    action = ''

    for tempAction in gameState.getLegalActions(currentIndex):
        tempValue = self.minimax(gameState.generateSuccessor(currentIndex , tempAction), currentIndex + 1, currentDepth)[1]

        if (tempValue > value):
            action = tempAction
            value = tempValue
    return action, value
```

3) Alpha-Beta Pruning

Alpha beta pruning follows the same logic as minimax, except it prunes paths (doesn't expand upon them) if they will never return a path that will not be pursued. This is the same thing as minimax

Simply added an alpha and beta to the function calls :

```
def alphaBeta(self, gameState, passedIndex, passedDepth, alpha = - float('inf'), beta = float('inf')):
def minimize(self, gameState, passedIndex, passedDepth, alpha = - float('inf'), beta = float('inf')):
def maximize(self, gameState, passedIndex, passedDepth, alpha = - float('inf'), beta = float('inf')):
```

Added a check against the alpha value in minimize and set the beta value:

```
    if (tempValue < value ):
        action = tempAction
        value = tempValue
    if (tempValue < alpha):
        return action, value
    beta = min(beta, value)
return action, value
```

Added a check against the beta value in maximize and set the alpha value:

```
    if (tempValue > value):
        action = tempAction
        value = tempValue
    if (tempValue > beta):
        return action, value
    alpha = max(alpha, value)
return action, value
```

4) Expectimax

This is used because oftentimes your opponent is not going to play optimally, can possibly win when minimax will suicide because it assumes the worse possible outcome.

This is the same as minimax except you now average on minimize. This is all that was changed.

```
for tempAction in gameState.getLegalActions(currentIndex):
    tempValue = self.expectimax(gameState.generateSuccessor(currentIndex, tempAction), currentIndex + 1, currentDepth)[1]

    value += tempValue
    count += 1

value = float(value) / float(count)
return action, value
```

5) Evaluation Function

I did nothing. I used the same evaluation function and it worked, see above for details.

Conclusions:

This project showed to us various types of decision making for the purpose of artificial intelligence: reflex, minimax, and expectimax. Alpha beta pruning was also displayed for the minimax, allowing for faster computations. It was displayed how these data structures can be useful, but how they also have drawbacks. As one searches deeper and deeper to make a well-informed decision the time to make a decision increases dramatically. With a depth of 5-6 the problem stops progressing. This limits the scope that can be viewed for the decision. Elements outside of the given range are not known and have to be estimated or assumed, causing a blindness or fog around the agent.

This project did a great introduction to the structures and showed how easy it is to expand upon existing components.