Artificial intelligence - Project 1
- Search problems -

Mihali Vlad

27/10/2020

1 Uninformed search

1.1 Question 1 - Depth-first search

In this section the solution for the following problem will be presented:

"In search.py, implement ***Depth-First search(DFS) algorithm*** *in function depthFirstSearch. Don't forget that DFS graph search is graph-search with the frontier as a LIFO queue(Stack)."*.

1.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  def depthFirstSearch(problem):
2      u = problem.getStartState()
3      stack = util.Stack()
4      mySet = []
5      result = []
6      stack.push((u, result))
7      while not stack.isEmpty() and not problem.isGoalState(u):
8          u, result = stack.pop()
9          mySet.append(u)
10         successors = problem.getSuccessors(u)
11         for nextSuc in successors:
12             if nextSuc[0] not in mySet:
13                 u = nextSuc[0]
14                 last = nextSuc[1]
15                 stack.push((nextSuc[0], result + [nextSuc[1]]))
16     return result + [last]
```

Explanation:

- Line 3: stack keep track of leaf nodes and resulting path.
- Line 4: mySet is a set that is used to keep track of visited nodes.
- Line 6: push the current state
- Line 8-9: pop from stack and put the element in mySet
- Line 11-15: for each successor, if it is in visited nodes, push on stack the succesor and the new path
- Line 16: return the resulting path of the goal position

Commands:

- -l tinyMaze -p SearchAgent
- -l mediumMaze -p SearchAgent
- -l bigMaze -z .5 -p SearchAgent

1.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Is the found solution optimal? Explain your answer.
A1: No. DFS is not optimal because if it finds the optimal solution does not stop, it will explore the other leafs

Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: 3/3

1.1.3 Personal observations and notes

1.2 Question 2 - Breadth-first search

In this section the solution for the following problem will be presented:

*"In **search.py**, implement the **Breadth-First search** algorithm in function breadthFirstSearch."*.

1.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```python
def breadthFirstSearch(problem):
    u = problem.getStartState()
    queue = util.Queue()
    mySet = []
    result = []
    queue.push((u, result))
    mySet.append(u)
    while not queue.isEmpty():
        u, result = queue.pop()
        if problem.isGoalState(u):
            return result
        successors = problem.getSuccessors(u)
        for nextSuc in successors:
            if nextSuc[0] not in mySet:
                mySet.append(nextSuc[0])
                queue.push((nextSuc[0], result + [nextSuc[1]]))
    return result
```

Explanation:
This implementation is similar to the implementation of DFS, but differs from it in two ways.

- it uses a queue instead of a stack;
- it checks whether a vertex has been discovered before enqueueing the vertex.

Commands:

- -l mediumMaze -p SearchAgent -a fn=bfs
- -l bigMaze -p SearchAgent -a fn=bfs -z .5

1.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Is the found solution optimal? Explain your answer.
A1: Yes and No. BFS stops at the optimal solution and it is optimal when actions are unweighted. If the graph is weighted the solution is not optimal.

Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: 3/3

1.2.3 Personal observations and notes

1.3 Question 3 - Uniform-cost search

In this section the solution for the following problem will be presented:

"In search.py, implement ***Uniform-cost graph search*** *algorithm in uniformCostSearchfunction"*

1.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def uniformCostSearch(problem):
    u = problem.getStartState()
    queue = util.PriorityQueue()
    mySet = []
    result = []
    queue.push((u, result, 0), 0)
    while not queue.isEmpty():
        u, result, cost = queue.pop()
        if u not in mySet:
            mySet.append(u)
            if problem.isGoalState(u):
                return result
            successors = problem.getSuccessors(u)
            for nextSuc in successors:
                queue.push((nextSuc[0], result + [nextSuc[1]], cost + nextSuc[2]), cost + nextSuc[2])
    return []
```

Explanation:

- The implementation is the same as BFS but instead of queue we use priority queue.
- We consider the cost on line 15

Commands:

- -l tinyMaze -p SearchAgent fn=ucs
- -l mediumMaze -p SearchAgent fn=ucs
- -l bigMaze -z .5 -p SearchAgent fn=ucs

1.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Compare the results to the ones obtained with DFS. Are the solutions different? Is the number of extended (explored) states smaller? Explain your answer.
A1: The solutions are different. The explored states is smaller for UCS because is optimal and DFS is not, it will explore more nodes.

Q2: Consider that some positions are more desirable than others. This can be modeled by a cost function which sets different values for the actions of stepping into positions. Identify in **searchAgents.py** the description of agents StayEastSearchAgent and StayWestSearchAgent and analyze the cost function. Why the cost .5 ** x for stepping into (x,y) is associated to StayWestAgen.
A2: For StayWestAgent, if you go up or down the cost will be the same as current state. If you go left the x will decrease and cost will increase. If you go right the cost will decrease. Therefor, it will go west, the same for StayEastAgent but with x ** a where a >= 1

Q3: Run autograder *python autograder.py* and write the points for Question 3.
A3: 3/3

1.3.3 Personal observations and notes

1.4 References

Cormen, T. H., & Cormen, T. H. (2001). Introduction to algorithms. Cambridge, Mass: MIT Press.

2 Informed search

2.1 Question 4 - A* search algorithm

In this section the solution for the following problem will be presented:

*"Go to aStarSearch in search.py and implement **A* search algorithm**. A* is graphs search with the frontier as a priorityQueue, where the priority is given bythe function g=f+h".*

2.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def aStarSearch(problem, heuristic=nullHeuristic):
    u = problem.getStartState()
    queue = util.PriorityQueue()
    mySet = []
    result = []
    queue.push((u, result, 0), 0)
    while not queue.isEmpty():
        u, result, cost = queue.pop()
        if u not in mySet:
            mySet.append(u)
            if problem.isGoalState(u):
                return result
            successors = problem.getSuccessors(u)
            for nextSuc in successors:
                queue.push((nextSuc[0], result + [nextSuc[1]], cost + nextSuc[2]), cost + nextSuc[2] + h
    return []
```

Listing 1: Solution for the A* algorithm.

Explanation:

- Same implementation as UCS but we add a function (heuristic) to the priority

Commands:

- -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

2.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Does A* and UCS find the same solution or they are different?
A1:

Q2: Does A* finds the solution with fewer expanded nodes than UCS?

A2: Yes

Q3: Does A* finds the solution with fewer expanded nodes than UCS?
A3: Yes

Q4: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).
A4: 3/3

2.1.3 Personal observations and notes

2.2 Question 5 - Find all corners - problem implementation

In this section the solution for the following problem will be presented:

"Pacman needs to find the shortest path to visit all the corners,regardless there is food dot there or not. Go to ***CornersProblem*** *in searchAgents.py and propose a representation of the state of this search problem. It might help to look at the existing implementation for PositionSearchProblem. The representation should include only the information necessary to reach the goal. Read carefully the comments inside the class CornersProblem."*.

2.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
class CornersProblem(search.SearchProblem):
    def __init__(self, startingGameState):
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print 'Warning: no food in corner ' + str(corner)
        self._expanded = 0

        self.startState = (self.startingPosition, self.corners)

    def getStartState(self):
        return self.startState

    def isGoalState(self, state):
        isGoal = False

        if len(state[1]) == 0:
            isGoal = True

```

```
23         return isGoal
24
25     def getSuccessors(self, state):
26         successors = []
27         for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
28             ((x, y), cornerTuple) = state
29             dx, dy = Actions.directionToVector(action)
30             nextx, nexty = int(x + dx), int(y + dy)
31             cornerList = list(cornerTuple)
32             if (nextx,nexty) in cornerList:
33                 cornerList.remove((nextx,nexty))
34             if not self.walls[nextx][nexty]:
35                 successors.append((((nextx, nexty), tuple(cornerList)), action, 1))
36         self._expanded += 1
37         return successors
38
39     def getCostOfActions(self, actions):
40         if actions == None: return 999999
41         x,y= self.startingPosition
42         for action in actions:
43             dx, dy = Actions.directionToVector(action)
44             x, y = int(x + dx), int(y + dy)
45             if self.walls[x][y]: return 999999
46         return len(actions)
```

Explanation:

- Line 12: start state is position with corners information
- Line 32-33: if corner in successor remove corner in corner list

Commands:

- -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic

2.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: For mediumCorners, BFS expands a big number - around 2000 search nodes. It's time to see that A* with an admissible heuristic is able to reduce this number. Please provide your results on this matter. (Number of searched nodes).

A1: 692 with corners heuristic implemented in Question 6

2.2.3 Personal observations and notes

2.3 Question 6 - Find all corners - Heuristic definition

In this section the solution for the following problem will be presented:

*"Implement a consistent heuristic for CornersProblem. Go to the function **cornersHeuristic** in searchAgent.py."*.

2.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  def cornersHeuristic(state, problem):
2      corners = problem.corners # These are the corner coordinates
3      walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
4      if len(state[1])==0:
5          return 0
6      distance = 0
7      currentPos = state[0]
8      cornersList = list(state[1])
9      while len(cornersList) != 0:
10         min = ((0,0),999999)
11         for corner in cornersList:
12             manD = (corner, util.manhattanDistance(corner, currentPos))
13             if manD[1] < min[1]:
14                 min = manD
15         distance += min[1]
16         currentPos = min[0]
17         cornersList.remove(currentPos)
18     return distance
```

Explanation:

- Line 7: current position is position of the state
- Line 10-14: compute minimum manhattan distance between corners and current position
- Line 15-17: add minimum to distance and remove element from corners and change in current position
- Line 18: return distance

Commands:

- -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
- -l mediumMaze -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic

2.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test with on the mediumMaze layout. What is your number of expanded nodes?
A1: 692 on mediumCorner layout and 1194 on mediumMaze

2.3.3 Personal observations and notes

2.4 Question 7 - Eat all food dots - Heuristic definition

In this section the solution for the following problem will be presented:

*"Propose a heuristic for the problem of eating all the food-dots. The problem of eating all food-dots is already implemented in **FoodSearchProblem** in searchAgents.py."*.

2.4.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```python
def foodHeuristic(state, problem):
    position, foodGrid = state
    foodList = foodGrid.asList()
    currentPos = position
    distance = 0
    if len(foodList) == 0:
        return distance

    """min = ((0,0),999999)
    for foodDot in foodList:
        manD = (foodDot, (abs(currentPos[0] - foodDot[0]) + abs(currentPos[1] - foodDot[1])))
        if manD[1] < min[1]:
            min = manD
    distance += min[1]
    currentPos = min[0]

    if currentPos in foodList:
        foodList.remove(currentPos)

    if len(foodList) == 0:
        return distance

    max = ((0,0),0)
    for foodDot in foodList:
        manD = (foodDot, (abs(currentPos[0] - foodDot[0]) + abs(currentPos[1] - foodDot[1])))
        if manD[1] > max[1]:
            max = manD

    return distance + max[1]"""
    a, b, c, d = currentPos, currentPos, currentPos, currentPos
    for nx,ny in foodList:
        if (nx <= a[0] and ny >= a[1]): a = (nx,ny)
        if (nx >= b[0] and ny >= b[1]): b = (nx,ny)
        if (nx <= c[0] and ny <= c[1]): c = (nx,ny)
        if (nx >= d[0] and ny <= d[1]): d = (nx,ny)

    distance = 0
    currentPos = position
    cornersList = [a,b,c,d]
    while len(cornersList) != 0:
        min = ((0,0),999999)
        for corner in cornersList:
            manD = (corner, util.manhattanDistance(corner, currentPos))
            if manD[1] < min[1]:
                min = manD
```

```
            distance += min[1]
            currentPos = min[0]
            cornersList.remove(currentPos)
        return distance
```

Explanation:

- First implementation: return sum from minimum manhattan distance from food list and current position and maximum manhattan distance from food list (without minimum) and minimum
- Second implementation: find corners with food from food list and then apply corner heuristic

Commands:

- -l testSearch -p AStarFoodSearchAgent
- -l trickySearch -p AStarFoodSearchAgent

2.4.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test with autograder *python autograder.py*. Your score depends on the number of expanded states by A* with your heuristic. What is that number?
A1: 7769 for using corner heuristic and 8178 for the other implementation

2.4.3 Personal observations and notes

2.5 References

3 Adversarial search

3.1 Question 8 - Improve the ReflexAgent

In this section the solution for the following problem will be presented:

"Improve the ReflexAgent such that it selects a better action. Include in the score food locations and ghost locations. The layout testClassic should be solved more often.".

3.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
    def evaluationFunction(self, currentGameState, action):
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
        score = successorGameState.getScore()
        score += currentGameState.hasFood(newPos[0], newPos[1]) * 100
        d1 = [manhattanDistance(newPos, foodPos) for foodPos in newFood.asList()]
        d2 = [manhattanDistance(newPos, ghostPos) for ghostPos in successorGameState.getGhostPositions()
        distanceGhost = min(d2) if len(d2) > 0 else 0
        distanceFood = min(d1) if len(d1) > 0 else 0
        foodList = newFood.asList()
        currentPos = newPos
        a, b, c, d = currentPos, currentPos, currentPos, currentPos
        for nx, ny in foodList:
            if (nx <= a[0] and ny >= a[1]): a = (nx, ny)
            if (nx >= b[0] and ny >= b[1]): b = (nx, ny)
            if (nx <= c[0] and ny <= c[1]): c = (nx, ny)
            if (nx >= d[0] and ny <= d[1]): d = (nx, ny)

        distance = 0
        cornersList = [a, b, c, d]
        while len(cornersList) != 0:
            min2 = ((0, 0), 999999)
            for corner in cornersList:
                manD = (corner, util.manhattanDistance(corner, currentPos))
                if manD[1] < min2[1]:
                    min2 = manD
            distance += min2[1]
            currentPos = min2[0]
            cornersList.remove(currentPos)
        score -= distance
        score -= sum(d1) / 80
        score -= distanceFood * 2
        score += distanceGhost
```

```
37        if distanceGhost < 2:
38            score = -1e6
39
40        if action == Directions.STOP:
41            score -= 85
42        return score
```

Explanation:

- Line 7: score we add initial score
- Line 13-33: subtract from score the distance to reach all corners
- Line 34: subtract all distances from pacman to all foods
- Line 35: subtract distance to the closest food
- Line 36: add distance to the closest ghost
- Line 37-38: if a ghost is to close, we force pacman not to go in ghost direction
- Line 40-41: avoid stop "direction"

Commands:

- python pacman .py -p RandomAgent -l testClassic
- python pacman .py -p ReflexAgent
- python pacman .py – frameTime 0 -p ReflexAgent -k 1 -l mediumClassic
- python pacman .py – frameTime 0 -p ReflexAgent -k 2 -l mediumClassic
- python autograder .py -q q1
- python autograder .py -q q1 –no - graphics

3.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test your agent on the openClassic layout. Given a number of 10 consecutive tests, how many types did your agent win? What is your average score (points)?
A1:10 wins and 1257.7 average score

3.1.3 Personal observations and notes

3.2 Question 9 - H-Minimax algorithm

In this section the solution for the following problem will be presented:

" Implement H-Minimax algorithm in MinimaxAgentclass from multiAgents.py. Since it can be more than one ghost, for each max layer there are one ormore min layers.".

3.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1   def getAction(self, gameState):
2       return max(((action, self.MinValue(gameState.generateSuccessor(0, action)))
3                   for action in gameState.getLegalActions(0)), key=lambda x: x[1])[0]
4
5   def MaxValue(self, gameState, depth):
6       if self.TerminalTest(depth, gameState):
7           return self.evaluationFunction(gameState)
8
9       return max([self.MinValue(gameState.generateSuccessor(0, action), depth)
10                  for action in gameState.getLegalActions(0)])
11
12  def MinValue(self, gameState, depth=0, agentIndex=1):
13      if self.TerminalTest(depth, gameState, agentIndex):
14          return self.evaluationFunction(gameState)
15      if agentIndex == gameState.getNumAgents() - 1:
16          return min([self.MaxValue(gameState.generateSuccessor(agentIndex, action), depth + 1)
17                      for action in gameState.getLegalActions(agentIndex)])
18      return min([self.MinValue(gameState.generateSuccessor(agentIndex, action), depth, agentIndex + 1
19                  for action in gameState.getLegalActions(agentIndex)])
20
21  def TerminalTest(self, depth, gameState, agentIndex=0):
22      return (depth == self.depth) or len(gameState.getLegalActions(agentIndex)) == 0
```

Explanation:

- the code reflects the pseudocode from the laboratory
- Line 2-3: extract the action that is represented the maximum value from all MinValue of legal actions
- Line 7 and 14:return evaluation that is named utility functions in the pseudocode
- Line 12: in the minvalue function we also call minvalue for each ghost, but last ghost call maxvalue
- Line 16: here we iterate depth, if we iterated in max it will be height and if we iterated in both it will be number of nodes in the tree
- Line 21-22: the terminal test is when we reach the depth or we can not perform moves any more

Commands:

- python pacman .py -p MinimaxAgent -l minimaxClassic -a depth =4
- python autograder .py -q q2
- python pacman .py -p MinimaxAgent -l trappedClassic -a depth =3

3.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test Pacman on trappedClassic layout and try to explain its behaviour. Why Pacman rushes to the ghost?

A1: Pacman rushes to the ghost because it is the best score that the minimax algorithm can find with that depth

3.2.3 Personal observations and notes

3.3 Question 10 - Use $\alpha - \beta$ pruning in AlphaBetaAgent

In this section the solution for the following problem will be presented:

*" Use alpha-beta prunning in **AlphaBetaAgent** from multiagents.py for a more efficient exploration of minimax tree.".*

3.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def getAction(self, gameState):

    alpha = float('-inf')
    beta = float('inf')
    max_action = None
    for action in gameState.getLegalActions(0):
        action_value = self.MinValue(gameState.generateSuccessor(0, action), 1, 0, alpha, beta)
        if alpha < action_value:
            alpha = action_value
            max_action = action

    return max_action

def MaxValue(self, gameState, depth, alpha, beta):
    if self.TerminalTest(depth, gameState):
        return self.evaluationFunction(gameState)

    v = float('-inf')
    for a in gameState.getLegalActions(0):
        v = max(v, self.MinValue(gameState.generateSuccessor(0, a), 1, depth, alpha, beta))
        if v >= beta:
            return v
        alpha = max(alpha, v)

    return v

def MinValue(self, gameState, agentIndex, depth, alpha, beta):
    if self.TerminalTest(depth, gameState, agentIndex):
        return self.evaluationFunction(gameState)

    v = float('inf')
    for a in gameState.getLegalActions(agentIndex):
        if agentIndex == gameState.getNumAgents() - 1:
            v = min(v, self.MaxValue(gameState.generateSuccessor(agentIndex, a), depth + 1, alpha, b
        else:
```

```
36                    v = min(v,
37                            self.MinValue(gameState.generateSuccessor(agentIndex, a), agentIndex + 1, depth
38
39                if v <= alpha:
40                    return v
41                beta = min(beta, v)
42
43            return v
44
45        def TerminalTest(self, depth, gameState, agentIndex=0):
46            return (depth == self.depth) or len(gameState.getLegalActions(agentIndex)) == 0
```

Explanation:

- the code reflects the pseudocode from the laboratory
- terminal test and utility are the same as in minimax implementation
- !!!Wrong!!! Line 21 and 39: need to delete "="
- Line 27-43: in the minvalue function we also call minvalue for each ghost, but last ghost call maxvalue

Commands:

- python pacman .py -p AlphaBetaAgent -a depth =3 -l smallClassic
- python autograder .py -q q3
- python autograder .py -q q3 –no - graphics

3.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test your implementation with autograder **python autograder.py** for Question 3. What are your results?

A1:5/5, but Pacman died! Score: 84

3.3.3 Personal observations and notes

3.4 References

4 Personal contribution

4.1 Question 11 - Define and solve your own problem.

In this section the solution for the following problem will be presented:

4.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

Explanation:

•

Commands:

•

4.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

4.1.3 Personal observations and notes

4.2 References