Artificial intelligence - Assignment 1

Chereji Iulia - 30434

November 5, 2021

Contents

P1: Search

Question 1: Finding a Fixed Food Dot using Depth First Search

For this question I implemented the graph search algorithm:

```
Algorithm: GRAPH SEARCH:
frontier = {startNode}
expanded = {}
while frontier is not empty:
    node = frontier.pop()
    if isGoal(node):
        return path_to_node
    if node not in expanded:
        expanded.add(node)
        for each child of node's children:
            frontier.push(child)
return failed
```

I wrote code in the depthFirstSearch function in the search.py file. For the frontier I used a Stack data structure taken from util.py file.

I created a Node data structure to add to the frontier, containing the state int which we arrived, the parent (where we came from, another Node), the action that got us here (which is the actual thing that we will need to return from the method) and the cost of the action which in this problem is always 1. I also override the eq method to check for equality when we check if the node is in the expanded list (only the state has to be checked).

```
class Node:
    def __init__(self, state, parent, action, cost):
        self.state=state
        self.parent=parent
        self.action=action
        self.cost=cost
    def __eq__(self, other):
        if other == None: return False
        return self.state==other.state
```

My solution for the depthFirstSearch:

```
def depthFirstSearch(problem):
"*** YOUR CODE HERE ***"
    frontier = util.Stack()
    frontier.push(Node(problem.getStartState(),None,None,0))
    expanded=[]
    while(not frontier.isEmpty()):
        currentNode=frontier.pop()
        if(problem.isGoalState(currentNode.state)):
```

```
            #if we reached the goal state start constructing the list of actions that got us here
            listToReturn=[]
            node1=currentNode
            while(node1.action!=None):
                listToReturn.insert(0,node1.action)
                node1=node1.parent
            return listToReturn
        if(currentNode not in expanded):
            expanded.append(currentNode)
            children=problem.expand(currentNode.state)
            [frontier.push(Node(x[0], currentNode, x[1], x[2])) for x in children if (
                        (Node(x[0], currentNode, x[1], x[2]) not in expanded))]
            #we add the current node in the expanded list and add all its children (states we can get to from the current state to the frontier if they were not yet expanded
    return []
```

The autograder graded this question with 4/4 points.

Question 2: Breadth First Search

For this question I implemented the same graph search algorithm in the breadthFirstSearch method in the search.py file, but I used a Queue data structure for the frontier.

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    frontier = util.Queue()
    frontier.push(Node(problem.getStartState(), None, None, 0))
    expanded = []
    while (not frontier.isEmpty()):
        currentNode = frontier.pop()

        if (problem.isGoalState(currentNode.state)):
            listToReturn = []
            node1 = currentNode
            while (node1.action != None):
                listToReturn.insert(0, node1.action)
                node1 = node1.parent
            return listToReturn
        if (currentNode not in expanded):
            expanded.append(currentNode)
            children = problem.expand(currentNode.state)
            [frontier.push(Node(x[0], currentNode, x[1], x[2])) for x in children if (
                (Node(x[0], currentNode, x[1], x[2]) not in expanded))]
    return []
```

The autograder graded this question with 4/4 points.

Question 3: A* search

A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information).

For this question I implemented the same graph search algorithm in the aStarSearch method in the search.py file, but I used a PriorityQueue data structure for the frontier. When pushing an item in the priority queue the item itself is the Node and for the priority I used the cost so far of the state from the node + the heuristic function applied to the state. Instead of using the push method of the PriorityQueue I used the update method.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic
    first."""
    "*** YOUR CODE HERE ***"
    frontier = util.PriorityQueue()
    frontier.push(Node(problem.getStartState(), None, None, 0),heuristic(
    problem.getStartState(),problem))
    expanded = []
    while (not frontier.isEmpty()):
        currentNode = frontier.pop()
        if (problem.isGoalState(currentNode.state)):
            listToReturn = []
            node1 = currentNode
            while (node1.action != None):
                listToReturn.insert(0, node1.action)
                node1 = node1.parent
            return listToReturn
        if (currentNode not in expanded):
            expanded.append(currentNode)
            children = problem.expand(currentNode.state)
                        for child in children:
                nod=Node(child[0],currentNode,child[1], child[2] +
    currentNode.cost)
                if nod not in expanded :
                    frontier.update(nod,nod.cost+heuristic(nod.state,
    problem))
    return []
```

The autograder graded this question with 4/4 points.

Question 4: Finding All the Corners

The problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not).

To encode the information I chose a state representation of the form (position, remaining corners), where position is a touple (x,y) representing the coordinates in the maze, and

remaining corners is an array containing the indices in the problem's corners array, of the corners that were not yet reached.

```
    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        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 # DO NOT CHANGE; Number of search nodes
    expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"
        self.startState = (self.startingPosition, [0,1,2,3])
        """an array with the indexes of the corners in self.corners that
    have not been visited"""

    def getStartState(self):
        "*** YOUR CODE HERE ***"
        return self.startState
```

Initially the start state contains the indices 0,1,2 and 3 because no corner has been reached, and then each corner's index will be removed when pacman reaches it.

Reaching the goal state means that the last corner has been found and eliminated from the array, so the length of the array is 0.

```
    def isGoalState(self, state):
        "*** YOUR CODE HERE ***"
        return len(state[1])==0
```

The expand function returs a list of triples, (child, action, stepCost), where 'child' is a child to the current state, 'action' is the action required to get there, and 'stepCost' is the incremental cost of expanding to that child.

```
    def expand(self, state):
        children = []
        for action in self.getActions(state):
            "*** YOUR CODE HERE ***"
            nextState = self.getNextState(state, action)
            cost = self.getActionCost(state,action,nextState)
            children.append( (nextState,action,cost) )
        return children
```

The getNextState function returns the new state in which the pacman will arrive if he takes an action from the current state. For that we copy the current state's array of

unreached corners and check if the new state's position is a corner, and if so, and if it hasn't been reached, we eliminate it from the new state's array.

```
    def getNextState(self, state, action):
        assert action in self.getActions(state), (
            "Invalid action passed to getActionCost().")
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        "*** YOUR CODE HERE ***"
        arr = state[1].copy()
        for i in range(4):
            if nextx == self.corners[i][0] and nexty == self.corners[i
    ][1] and i in arr:
                arr.remove(i)
                """we got to a corner"""
        return ((nextx,nexty), arr)
```

The autograder graded this question with 3/3 points.

Question 5: Corners Problem: Heuristic

For this question I implemented a non-trivial, consistent heuristic for the CornersProblem. I computed the manhattan distance from the current position to all the remaining corners and returned the greatest one.

The heuristic is admissible because the value is the lower bound on the actual shortest path cost (the real cost will never be smaller than the heuristic approximation), and is non-negative. It is also consistent because if an action has cost c, then taking that action can only cause a drop in heuristic of at most c. The heuristic always returns 0 at a goal state.

```
    def cornersHeuristic(state, problem):
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (
    game.py)

    "*** YOUR CODE HERE ***"
    #max of manhattan
    x1, y1 = state[0]
    max=0
    for i in range(len(state[1])):
        x2, y2 = corners[state[1][i]]
        val = abs(x1 - x2) + abs(y1 - y2)
        if val>max: max=val
    return max
```

The autograder graded this question with 3/3 points.

Question 6: Eating All The Dots

The FoodSearchProblem means eating all the food in the maze. For this I implemented the method foodHeuristic. I used the maximum of the manhattan distance from pacman to all the dots.

```
    def foodHeuristic(state, problem):
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    listOfFood=foodGrid.asList()
    "*** max of manhattan ***"
    x1, y1 = position
    max = 0
    for i in range(len(listOfFood)):
        x2, y2 = listOfFood[i]
        val = abs(x1 - x2) + abs(y1 - y2)
        if val > max: max = val
    return max
```

However this works for all the mazes except for the trikySearch board.

I tried to use the PositionSearchProblem. To create a problem for each remaining piece of food in which the starting position is the pacman's position and the goal state is the piece of food, and use BFS to find the real distance to it; then return the maximum.

```
    maximum=0
    for i in range(len(listOfFood)):
        probl=PositionSearchProblem(gameState=problem.startingGameState,
    goal=listOfFood[i],start=position,warn=False,visualize=False)
        dist=len(search.bfs(probl))
        if dist>maximum:
            maximum=dist
    return maximum
```

I couldn't get it to work because I couldn't initialize the problems :(

The autograder graded this question with 0/4 points. The first 17 test cases pass but the tricky test does not and it causes the autograder to run out of time.

Question 7: Suboptimal Search

For this problem we will complete an agent that always greedily eats the closest dot. It is missing the implementation of the findPathToClosestDot method. To do this we will complete the AnyFoodSearchProblem, which is missing its goal test then solve it with A* search and the null heuristic.

```
    def findPathToClosestDot(self, gameState):
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

```

```
        "*** YOUR CODE HERE ***"
        return search.aStarSearch(problem)
```

For the isGoalState in AnyFoodSearchProblem, we will simply check if there is a piece of food at the current state; and if so we return true, otherwise we return false.

```
    def isGoalState(self, state):
        x,y = state

        "*** YOUR CODE HERE ***"
        if self.food[x][y]==True: return True
        else: return False
```

The autograder graded this question with 3/3 points.

P2: Multiagent

Question 1: Reflex Agent

A reflex agent chooses an action at each choice point by examining its alternatives via a state evaluation function. The evaluation function that we have to write takes in the current game state and a proposed action to be taken and returns a number, where higher numbers are better.

The first approach that I tried (and failed) was to calculate the manhattan distance from pacman and the closest piece of food in distFood, and the manhattan distance from pacman and the closest ghost in distGhost and then add weights to them based on how critical it would be for pacman to run from the ghost. These would be weightGhost and weightFood, where weightGhost + weightFood = 1. Then the value returned by the evaluationFunction would be weightGhost*distGhost - weightFood*distFood; meaning that if the ghost is far and the food is close then the returned value would be big and we would take this action. To compute weightGhost I set a thresHold of 1 meaning that if the ghost is at distance ¡=1 then weightGhost=0 so that action is really bad and we should run away.

```
    def evaluationFunction(self, currentGameState, action):
        childGameState = currentGameState.getPacmanNextState(action)
        newPos = childGameState.getPacmanPosition()
        newFood = childGameState.getFood()
        newGhostStates = childGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in
    newGhostStates]

        "*** YOUR CODE HERE ***"
        listFood=newFood.asList()
        distFood=-1
        lenListFood=len(listFood)
        for i in range(lenListFood):
            md=util.manhattanDistance(newPos,listFood[i])
```

```
            if distFood==-1:
                distFood=md
            else:
                if md<distFood:
                    distFood=md

        listGhosts=childGameState.getGhostPositions()
        distGhost = -1
        lenLisGhost = len(listGhosts)
        for i in range(lenLisGhost):
            mdg = util.manhattanDistance(newPos, listGhosts[i])
            if distGhost == -1:
                distGhost = mdg
            else:
                if mdg < distGhost:
                    distGhost = mdg

        if distGhost<=1: weightGhost=0
        else:
            if distGhost<=3: weightGhost=0.5
            else:
                if distGhost<=6: weightGhost=0.6
                else: weightGhost=0.7
        weightFood=1-weightGhost

        return weightGhost*distGhost - weightFood*distFood
```

The problem is that pacman only behaves good if the ghost is close, otherwise it will just move around close to food without going to eat it and without making any progress, so pacman is good at not dying but not good at winning.

The second approach I tried was to just check if there will be a ghost at the new pacman position and in that case (and if the scared timer is equal to zero) return a very small number (meaning very bad choice of action). Otherwise, we take the list of the remaining food at the current moment (before taking the action), calculate the manhattan distance to each piece of food and take the smallest distance. We return that distance negated because the higher the returned value of the evaluationFunction the better the action. So the function will always return a negative value and the greater one (smallest distance) will be chosen.

```
    def evaluationFunction(self, currentGameState, action):
        childGameState = currentGameState.getPacmanNextState(action)
        newPos = childGameState.getPacmanPosition()
        newFood = childGameState.getFood()
        newGhostStates = childGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in
    newGhostStates]

        foodNow = currentGameState.getFood()
        foodNowList = foodNow.asList()
        "*** YOUR CODE HERE ***"
        for ghost in newGhostStates:
```

```
12              if ghost.getPosition() == tuple(newPos) and (ghost.
    scaredTimer == 0):
13                  return -10000000
14
15          maxDist = -10000000
16          for food in foodNowList:
17              dist = util.manhattanDistance(newPos, food) * -1
18              if (dist > maxDist):
19                  maxDist = dist
20
21          return maxDist
```

The autograder graded this question with 4/4 points.

Question 2: Minimax

For this question it is required to write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py. The minimax agent should work with any number of ghosts, so the minimax tree will have multiple min layers (one for each ghost) for every max layer. The method getAction has to be implemented; it returns the minimax action from the current gameState. The depth of the minimax tree is variable of the problem and the score of the leaves of the tree will be calculated using the provided function evaluationFunction.

In this problem we're evaluating states rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.

To solve the problem I first created an inner function called miniMax(gameState, howDeep, agent), where gameState is the state of the agent in GameState representation, howDeep represents the deepness at which we arrived (what layer in the tree are we at) and agent is an integer in range 0 (pacman) .. number of ghosts. From the getActions method we will just call minimax(gameState, 0, 0) meaning that we are at deepness 0 and pacman is the first to move. The method returns a dictionary of the form action: ..., value: ..., where value is the score that we will reach if we take the respective action.

The method then calls another 2 methods to help expand the tree: maxHelper if pacman is to move and miniHelper if it is a ghost's turn. These functions will recursively call minimax again and so on.

From minimax the recursion ends if we reached a winning state, a losing state, or we reached the problem's depth, in this case we return the action none and the evaluationFunction of the current state. In maxHelper and minHelper the recursion ends if there are no more actions that can be made from the current state, case in which we return none action and the score of the state, or for each possible action, we save the agent's next state and call minimax for the next agent.

Then for maxHelper we choose the greatest value of the possibilities and return it to-

gether with the action that takes us there; and for minHelper we choose the smallest value.

So from the returned dictionary we need the value key for the minimax logic to calculate the best option, and the action key because that is what we will need to return from the getActions method.

```
    def getAction(self, gameState):
        "*** YOUR CODE HERE ***"

        def miniMax(gameState, howDeep, agent):
            if agent >= gameState.getNumAgents():
                #we completed new level in the tree (pacman and all the ghosts), so we increase depth and start over with pacman (agent 0)
                agent = 0
                howDeep = howDeep + 1
            if (gameState.isWin() or gameState.isLose() or howDeep == self.depth):
                #we reached a terminal leaf (win or lost) that can't be expanded no more or we reached the maximum depth given to the problem
                #so we take no more actions and return the current state's score
                return {"action":"none", "value":self.evaluationFunction(gameState)}
            else:
                #we expand further
                if (agent == 0):
                    #it is pacman's turn so we use the maxHelper function
                    return maxHelper(gameState, howDeep, agent)
                else:
                    #it is a ghost's turn so we use the miniHelper function#
                    return miniHelper(gameState, howDeep, agent)

        def maxHelper(gameState, howDeep, agent):
            pacmanActions = gameState.getLegalActions(agent)
            if len(pacmanActions)==0:
                #there are no more actions that can be made from this state so we return the state's score
                return {"action":"none", "value":self.evaluationFunction(gameState)}

            output={"action":"none", "value":-10000000}
            for action in pacmanActions:
                #expand the tree starting from each of the next actions, for the other agents (ghosts), and the deepness does not change
                nextState = gameState.getNextState(agent, action)
                currentValue = miniMax(nextState, howDeep, agent + 1)
                #since we are on pacman's turn (max), we choose the greatest value of the possibilities and return it together with the action that
                #takes us there
                if currentValue["value"]>output["value"]:
```

```
                    output["action"]=action
                    output["value"]=currentValue["value"]
            return output

        def miniHelper(gameState, howDeep, agent):
            ghostActions = gameState.getLegalActions(agent)
            if len(ghostActions)==0:
                #there are no more actions that can be made from this state so we return the state's score
                return {"action":"none", "value":self.evaluationFunction(gameState)}

            output = {"action": "none", "value": 10000000}
            for action in ghostActions:
                #expand the tree starting from each of the next actions, for the other agents (the remaining ghosts), and the deepness does not change
                nextState = gameState.getNextState(agent, action)
                currentValue = miniMax(nextState, howDeep, agent + 1)
                #since we are on a ghost's turn (mini), we choose the smallest value of the possibilities and return it together with the action that
                #takes us there
                if currentValue["value"]<output["value"]:
                    output["action"]=action
                    output["value"]=currentValue["value"]
            return output

        outputFinal=miniMax(gameState, 0, 0)
        return outputFinal["action"]
```

The autograder graded this question with 5/5 points.

Question 3: Alpha-Beta Pruning

For this question we have to make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree.

I used the same approach (with helper functions) as for the question 2, but with some modifications in order to fit the pseudocode fot alpha-beta pruning:

```
def max-value(state, alpha, beta):
    initialize v=-infinity
    for each sucessor of state:
        v=max(v, value(successor,alpha,beta))
        if v>beta return v
        alpha=max(alpha,v)
    return v
```

```
def min-value(state, alpha, beta):
    initialize v=+infinity
    for each sucessor of state:
        v=min(v, value(successor,alpha,beta))
        if v<alpha return v
        beta=min(beta,v)
    return v
```

Where alpha is max's best option on path to root and beta is min's best option on path to root.

The minimax function becomes miniMax(gameState, howDeep, agent, maximum, minimum) where maximum is alpha and minimum is beta, and the function is called from getAction as miniMax(gameState, 0, 0, -10000000, 10000000).

The maxHelper repesents the max-value from the pseudocode, minHelper: min-value, output is v and the call to "value" means call to minimax.

The line "if v¡alpha return v" form min-value represents the actual optimization (pruning). It means that if the current value from this subtree is smaller than the so far maximum (possibly from another branch) then we don't need to expand the other subtrees because this means that this whole branch will not be chosen for the solution so it is an optimization to return here and to not work further in vain.

The reciprocal is the line "if v¿beta return v" from max-value which means the opposite for max.

```
1     def getAction(self, gameState):
2         "*** YOUR CODE HERE ***"
3
4         def miniMax(gameState, howDeep, agent, maximum, minimum):
5             if agent >= gameState.getNumAgents():
6                 #we completed new level in the tree (pacman and all the
    ghosts), so we increase depth and start over with pacman (agent 0)
7                 agent = 0
8                 howDeep = howDeep + 1
9             if (gameState.isWin() or gameState.isLose() or howDeep ==
    self.depth):
10                #we reached a terminal leaf (win or lost) that can't be
    expanded no more or we reached the maximum depth given to the problem
11                #so we take no more actions and return the current state'
    s score
12                return {"action": "none", "value": self.
    evaluationFunction(gameState)}
13            else:
14                #we expand further
15                if (agent == 0):
16                    #it is pacman's turn so we use the maxHelper function
17                    return maxHelper(gameState, howDeep, agent, maximum,
    minimum)
18                else:
```

```
19                  #it is a ghost's turn so we use the miniHelper
    function
20                  return miniHelper(gameState, howDeep, agent, maximum,
     minimum)
21
22          def maxHelper(gameState, howDeep, agent, maximum, minimum):
23              pacmanActions = gameState.getLegalActions(agent)
24              if len(pacmanActions) == 0:
25                  #there are no more actions that can be made from this
    state so we return the state's score
26                  return {"action": "none", "value": self.
    evaluationFunction(gameState)}
27
28              "initialize v as minus infinity"
29              output = {"action": "none", "value": -10000000}
30              for action in pacmanActions:
31                  #expand the tree starting from each of the next actions,
    for the other agents (ghosts), and the deepness does not change
32                  nextState = gameState.getNextState(agent, action)
33                  currentValue = miniMax(nextState, howDeep, agent + 1,
    maximum, minimum)
34                  #since we are on pacman's turn (max), we choose the
    greatest value of the possibilities and return it together with the
    action that
35                  #takes us there
36                  "v=max(v, value(successor,alpha, beta))"
37                  if currentValue["value"] > output["value"]:
38                      output["action"] = action
39                      output["value"] = currentValue["value"]
40
41                  #if the current value from this subtree is greater than
    the so far minimum (possibly from another branch) then we don't need
42                  #to expand the other subtrees because this means that
    this whole branch will not be chosen for the solution so it is an
    optimization
43                  #to return here and to not work further in vain
44                  "if v>beta return v"
45                  if output["value"]>minimum:
46                      return output
47
48                  #else if the current value is greater than the so far
    maximum then the maximum gets the value of the current value
49                  "alpha = max(alpha,v)"
50                  if output["value"]>maximum:
51                      maximum=output["value"]
52
53              return output
54
55          def miniHelper(gameState, howDeep, agent, maximum, minimum):
56              ghostActions = gameState.getLegalActions(agent)
57              if len(ghostActions) == 0:
58                  #there are no more actions that can be made from this
    state so we return the state's score
```

```
59                  return {"action": "none", "value": self.
    evaluationFunction(gameState)}
60
61              "initialize v as infinity"
62              output = {"action": "none", "value": 10000000}
63              for action in ghostActions:
64                  #expand the tree starting from each of the next actions,
    for the other agents (the remaining ghosts), and the deepness does not
     change
65                  nextState = gameState.getNextState(agent, action)
66                  currentValue = miniMax(nextState, howDeep, agent + 1,
    maximum, minimum)
67                  #since we are on a ghost's turn (mini), we choose the
    smallest value of the possibilities and return it together with the
    action that
68                  #takes us there
69                  "v=min(v, value(successor,alpha, beta))"
70                  if currentValue["value"] < output["value"]:
71                      output["action"] = action
72                      output["value"] = currentValue["value"]
73
74                  #if the current value from this subtree is smaller than
    the so far maximum (possibly from another branch) then we don't need
    to
75                  #expand the other subtrees because this means that this
    whole branch will not be chosen for the solution so it is an
    optimization
76                  #to return here and to not work further in vain
77                  "if v<alpha return v"
78                  if output["value"]<maximum:
79                      return output
80
81                  #else if the current value is smaller than the so far
    minimum then the minimum gets the value of the current value
82                  "beta = min(beta,v)"
83                  if output["value"]<minimum:
84                      minimum=output["value"]
85
86              return output
87
88          outputFinal = miniMax(gameState, 0, 0, -10000000, 10000000)
89          return outputFinal["action"]
```

The autograder graded this question with 5/5 points.