Artificial intelligence - Project 1
- Search problems -

Dorofte Andrei, Iacob Liviu

7/10/2020

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:

```python
def depthFirstSearch(problem):

    start_node = problem.getStartState()
    if problem.isGoalState(start_node):
        return []
    visited = []
    processing = Stack()
    processing.push((start_node, []))
    while not processing.isEmpty():
        curr = processing.pop()

        if problem.isGoalState(curr[0]):
            return curr[1]
        if curr[0] not in visited:
            visited.append(curr[0])
            for n in problem.getSuccessors(curr[0]):
                if n[0] not in visited:
                    processing.push((n[0], curr[1] + [n[1]]))

    util.raiseNotDefined()
```

Explanation:

- DFS employs a strategy of exploring the deepest nodes of the Pacman maze. Therefore, the stack allows us to push the nodes from closest to the farthest, and then popping and exploring them one by one.
- If we run one of the commands in the terminal,the Pacman board will be displayed an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration).
- When using the DFS algorithm, Pacman looks like a speedster

Commands:

- python pacman.py -l tinyMaze -p SearchAgent
- python pacman.py -l mediumMaze -p SearchAgent
- python pacman.py -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, because it stops when it finds the first solution, even if it is not optimal.

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

A2: Question q1: 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):

    processing = util.Queue()
    processing.push((problem.getStartState(), [], []))
    visited = []
    while not processing.isEmpty():
        node, actions, cost = processing.pop()
        if node not in visited:
            visited.append(node)
            if problem.isGoalState(node):
                return actions
            for child, direction, cost_child in problem.getSuccessors(node):
                processing.push((child, actions + [direction], cost + [cost_child]))
    return []
```

Explanation:

- BFS explores nodes one depth level at a time. It starts from a node, then checks the neighbours of the initial node, then the neighbours of the neighbours and so on, so we can say that it runs in waves. This implementation uses a FIFO queue as an utility to store the nodes before being explored.
- This algorithm is always able to find a solution to a problem, if there is one, but the time complexity is exponential.

Commands:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

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, because it expands all the neighbours of the neighbours until it finds a solution. As a result, the solution will be the shortest path.
Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: Question q2: 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:

```python
def uniformCostSearch(problem):

    processing = util.PriorityQueue()
    processing.push((problem.getStartState(), [], 0), 0)
    visited = []
    while not processing.isEmpty():
        node, actions, cost = processing.pop()
        if node not in visited:
            visited.append(node)
            if problem.isGoalState(node):
                return actions
            for child, direction, cost_child in problem.getSuccessors(node):
                processing.push((child, actions + [direction], cost_child + cost), cost + cost_child)
    return []
```

Explanation:

- Uniform Cost Search is the best algorithm for a search problem, which does not involve the use of heuristic function. UCS finds the optimal path between the two nodes.
- Instead of expanding the shallowest node, uniform-cost search expands the node with the lowest path cost, which makes it a sort of cost-aware BFS. This is done by storing the frontier as a priority queue, which is a sorted FIFO.

Commands:

- python pacman.py -l mediumMaze -p SearchAgent -a 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, because DFS does not find the optimal solution, returning the first solution found, while UCS returns the optimal one. The number of nodes expanded is almost double in the mediumMaze test for UCS comparing to DFS (269 for UCS, 146 for DFS).

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 StayWestAgent.

A2: If we want to make sure that pacman goes first to the west direction, the cost for west must be higher than other directions.

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

A3: Question q3: 3/3

1.3.3 Personal observations and notes

1.4 References

http://aima.cs.berkeley.edu/algorithms.pdf

http://aima.cs.berkeley.edu/contents.html

http://aima.cs.berkeley.edu/figures.pdf

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:

```python
def aStarSearch(problem, heuristic=nullHeuristic):

    processing = util.PriorityQueue()
    processing.push((problem.getStartState(), [], 0), heuristic(problem.getStartState(), problem))
    visited = []
    while not processing.isEmpty():
        node, actions, cost = processing.pop()
        if node not in visited:
            visited.append(node)
            if problem.isGoalState(node):
                return actions
            for child, direction, cost_child in problem.getSuccessors(node):
                g = cost + cost_child
                processing.push((child, actions + [direction], cost + cost_child), g + heuristic(child,
    return []
```

Listing 1: Solution for the A* algorithm.

Explanation:

- A* is almost exactly like UCS algorithm, except that we add in a heuristic. With A*, once we get past the obstacle, it prioritizes the node with the lowest f and the best chance of reaching the goal.
- One important aspect of A* is f = g + h, which is calculated at every step.
- F is the total cost of the node.
- G is the distance between the current node and the start node.
- H is the heuristic — estimated distance from the current node to the end node.
- The heuristic is always an underestimation of the total path, as an overestimation would lead to A* searching through nodes that may not be the 'best' in terms of f value.

Commands:

- python pacman.py -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: Yes, they find the same solution.
Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2: Yes, A* always expands fewer nodes than UCS. (549 for A* on bigMaze and 620 for UCS)
Q3: Does A* finds the solution with fewer expanded nodes than UCS?
A3:

Q4: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).
A4: Question q4: 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):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    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 ***"
        # stanga jos, stanga sus, dreapta jos, dreapta sus
        self.state = (self.startingPosition, [])

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        return self.state
        # util.raiseNotDefined()

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        node = state[0]
        visitedCorners = state[1]

        if node in self.corners:
            if node not in visitedCorners:
                visitedCorners.append(node)
            return len(visitedCorners) == 4
        else:
            return False

        # util.raiseNotDefined()

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.

         As noted in search.py:
            For a given state, this should return a list of triples, (successor,
            action, stepCost), where 'successor' is a successor to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that successor
        """
        x, y = state[0]
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]

            "*** YOUR CODE HERE ***"

            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                succCorners = list(state[1])
                if (nextx, nexty) in self.corners:
                    if (nextx, nexty) not in succCorners:
                        succCorners.append((nextx, nexty))
                nextState = ((nextx, nexty), succCorners)
                successors.append((nextState, action, 1))

        self._expanded += 1  # DO NOT CHANGE
        return successors

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999.  This is implemented for you.
        """
        if actions is None:
            return 999999
        x, y = self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
        return len(actions)
```

Explanation:

- The starting state is represented by the position of Pacman and a list of of visited corners which starts empty.
- When Pacman arrives in a corner position, appends the corner into the list as a mark of visiting it.
- The goal state is achieved when the list with visited corners is full (contains all 4 different corners).

Commands:

- python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

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: Number of search node with BFS: 2448
Number of search node with A*: 901

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:

```python
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 ***"
    #
    visited = state[1]
    cornersLeft = []
    for corner in corners:
        if corner not in visited:
            cornersLeft.append(corner)

    #
    #
    totalCost = 0
    curPoint = state[0]
    while cornersLeft:
        heuristic_cost = min([(util.manhattanDistance(curPoint, corner)) for corner in cornersLeft])
        for corner in cornersLeft:
            if heuristic_cost == util.manhattanDistance(curPoint, corner):
                currentCorner = corner
        cornersLeft.remove(currentCorner)
        curPoint = currentCorner
        totalCost += heuristic_cost
    return totalCost
    #return 0  # Default to trivial solution   ((abs(curPoint[0] - corner[0])) + (abs(curPoint[1] - cor
```

Explanation:

- cornersLeft represents a list of unvisited corners
- The heuristic cost is defined by the distance to the closest corner
- After arriving in the closest corner, removes it from cornersLeft and adds the cost to the totalCost.

Commands:

- python pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuris
- python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

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: Search nodes expanded: 766

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
    "*** YOUR CODE HERE ***"
    return len(foodGrid.asList())
    #
    foodLeft = []
    for food in foodGrid:
        if food not in foodGrid:
            foodLeft.append(food)

    #
    #
    totalCost = 0
    while foodLeft:
        heuristic_cost = min([(util.manhattanDistance(position, food)) for food in foodLeft])
        for food in foodLeft:
            if heuristic_cost == util.manhattanDistance(position, food):
                currentFood = food
        foodLeft.remove(currentFood)
        position = currentFood
        totalCost += heuristic_cost
    return totalCost
```

Explanation:

- Returns the length of the foodGrid, considering the estimated distance to each foodDot 1. Therefor, the added distance will be the length of the list.

Commands:

- python pacman.py -l testSearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic

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: expanded nodes: 12517

2.4.3 Personal observations and notes

Because the heuristic implemented for corners problem didn't make it in time for the autograder, the "unholy" solution found was to consider the distance to each food point equal to 1.

2.5 References

https://github.com/R-Alex95/Project-2-Multi-Agent-Pacman
https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2
http://aima.cs.berkeley.edu/algorithms.pdf
http://aima.cs.berkeley.edu/contents.html
http://aima.cs.berkeley.edu/figures.pdf

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:

```
1  class ReflexAgent(Agent):
2  
3  
4      def getAction(self, gameState):
5          """
6          You do not need to change this method, but you're welcome to.
7  
8          getAction chooses among the best options according to the evaluation function.
9  
10         Just like in the previous project, getAction takes a GameState and returns
11         some Directions.X for some X in the set {NORTH, SOUTH, WEST, EAST, STOP}
12         """
13         # Collect legal moves and successor states
14         legalMoves = gameState.getLegalActions()
15 
16         # Choose one of the best actions
17         scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
18         bestScore = max(scores)
19         bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
20         chosenIndex = random.choice(bestIndices)  # Pick randomly among the best
21 
22         "Add more of your code here if you want to"
23 
24         return legalMoves[chosenIndex]
25 
26     def evaluationFunction(self, currentGameState, action):
27         """
28         Design a better evaluation function here.
29 
30         The evaluation function takes in the current and proposed successor
31         GameStates (pacman.py) and returns a number, where higher numbers are better.
32 
33         The code below extracts some useful information from the state, like the
34         remaining food (newFood) and Pacman position after moving (newPos).
35         newScaredTimes holds the number of moves that each ghost will remain
36         scared because of Pacman having eaten a power pellet.
```

```python

        Print out these variables to see what you're getting, then combine them
        to create a masterful evaluation function.
        """
        # Useful information you can extract from a GameState (pacman.py)
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

        "*** YOUR CODE HERE ***"
        """newFood = successorGameState.getFood().asList()
        minFood = 999999
        for food in newFood:
            minFood = min(minFood, dist(newPos, food))"""

        """ghost_pos = currentGameState.getGhostPositions()
        distToGhosts = [manhattanDistance(newPos, ghost_position) for ghost_position in ghost_pos]

        print("--------------------------------")
        print(currentGameState.getPacmanPosition(), action, newPos)
        print(ghost_pos)
        print(distToGhosts)
        print("--------------------------------")"""

        ghost_pos = currentGameState.getGhostPositions()
        distToGhosts = [manhattanDistance(newPos, ghost_position) for ghost_position in ghost_pos]
        distToFood = [manhattanDistance(newPos, foodPos) for foodPos in newFood.asList()]

        d1 = min(distToGhosts)
        if distToFood:
            d2 = min(distToFood) + 1
        else:
            d2 = 999999
        if d1 < 2:
            d1 = 0.00001
            return successorGameState.getScore() + 1 / d2 - 1 / d1
        else:
            return successorGameState.getScore() + 1 / d2


def scoreEvaluationFunction(currentGameState):
    """
    This default evaluation function just returns the score of the state.
    The score is the same one displayed in the Pacman GUI.

    This evaluation function is meant for use with adversarial search agents
    (not reflex agents).
    """
    return currentGameState.getScore()
```

Explanation:

- d1 represents the manhattanDistance from current position to the nearest ghost.

- If d1 is less than 2, pacman is in danger.
- d2 represents the manhattanDistance from current position to the nearest food point. We add 1 because if we arrive to the food point, the division by 0 is illegal.
- The score returned is added with 1/d2, and if the ghost is close, (Pacman feels the danger) runs away by taking into account the distance to the ghost into the score, keeping at least a distance of 1 by substracting from the score 1/d2.

Commands:

- python pacman.py -p ReflexAgent -l testClassic

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: Our agent won every time and the average score was 1239.9 points

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  class MinimaxAgent(MultiAgentSearchAgent):
2      pacman = 0
3
4      def getAction(self, gameState):
5          "*** YOUR CODE HERE ***"
6
7          max_value = -99999
8          max_action = None
9
10         for action in gameState.getLegalActions(0):
11             action_value = self.Min_Value(gameState.generateSuccessor(0, action), 1, 0)
12             if action_value > max_value:
13                 max_value = action_value
14                 max_action = action
15
```

```
        return max_action

    def Max_Value(self, gameState, depth):

        if (depth == self.depth) or (len(gameState.getLegalActions(0)) == 0):
            return self.evaluationFunction(gameState)

        listOfMin = list(self.Min_Value(gameState.generateSuccessor(0, action), 1, depth) for action in
                         gameState.getLegalActions(0))  # calculam minimul pentru fiecare fantoma

        return max(listOfMin)

    def Min_Value(self, gameState, agentIndex, depth):

        if len(gameState.getLegalActions(
                agentIndex)) == 0:  # daca nu mai sunt actiuni posibile sa calculeze scorul penntru age
            return self.evaluationFunction(gameState)

        if agentIndex < gameState.getNumAgents() - 1:
            listOfMin = list(self.Min_Value(gameState.generateSuccessor(agentIndex, action), agentIndex
                             for action in gameState.getLegalActions(agentIndex))
            return min(listOfMin)

        else:  # decat ultima fantoma trimite "semnal" catre pacman sa calculeze maxim din minimele obt
            decisionList = list(self.Max_Value(gameState.generateSuccessor(agentIndex, action), depth +
                                for action in gameState.getLegalActions(agentIndex))
            return min(decisionList)
```

Explanation:

- The getAction function returns the action with the maximum cost.
- The Max Value function is responsible for the decision taken by Pacman, choosing the action with the maximum points possible (from the minimum score from the actions made by ghosts, considering that the ghosts are trying to make Pacman lose).
- The Min Value function is responsible with the actions made by ghosts, always choosing the one with the lowest score for Pacman (highest score for them).
- The depth information represents how many "steps ahead" should consider.
- In other words, it considers the actions that the ghosts could make for each action that Pacman makes and takes the maximum total output, taking into account that the ghosts will always make the most favorable action for them (the least favorable action for Pacman)

Commands:

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

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 realizes that he will lose, so he chooses the loss with the highest score. (lowest number of moves)

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:

```python
class AlphaBetaAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        alpha = -99999
        beta = 99999

        max_action = None
        for action in gameState.getLegalActions(0):
            action_value = self.Min_Value(gameState.generateSuccessor(0, action), 1, 0, alpha, beta)
            if alpha < action_value:
                alpha = action_value
                max_action = action

        return max_action

    def Min_Value(self, gameState, agentIndex, depth, alpha, beta):

        if len(gameState.getLegalActions(agentIndex)) == 0:
            return self.evaluationFunction(gameState)

        action_value = 99999
        for action in gameState.getLegalActions(agentIndex):
            if agentIndex < gameState.getNumAgents() - 1:
                siblingValue = self.Min_Value(gameState.generateSuccessor(agentIndex, action),
                                              agentIndex + 1, depth, alpha, beta)  # minimul dintre fra
                action_value = min(action_value, siblingValue)
            else:
                fatherValue = self.Max_Value(gameState.generateSuccessor(agentIndex, action),
                                             depth + 1, alpha, beta)  # maximul dintre copii
                action_value = min(action_value, fatherValue)

            if action_value < alpha:
                return action_value
            beta = min(beta, action_value)

        return action_value

    def Max_Value(self, gameState, depth, alpha, beta):

        if depth == self.depth or len(gameState.getLegalActions(0)) == 0:
            return self.evaluationFunction(gameState)

        action_value = -99999
        for action in gameState.getLegalActions(0):
            fatherValue = self.Min_Value(gameState.generateSuccessor(0, action), 1, depth, alpha, beta)
                                # minimul dintre copii
            action_value = max(action_value, fatherValue)

            if action_value > beta:
                return action_value
            alpha = max(alpha, action_value)

        return action_value
```

Explanation:

- Similar to H-MiniMax algorithm, but it does not expand completely a node if it finds a child that is lower than father's brother. A node is fully expanded when alpha and beta are the same value, otherwise alpha and beta will be different.

Commands:

- python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic

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: Question q3: 5/5

3.3.3 Personal observations and notes

3.4 References

http://aima.cs.berkeley.edu/algorithms.pdf
http://aima.cs.berkeley.edu/contents.html
http://aima.cs.berkeley.edu/figures.pdf
https://github.com/srinadhu/adversarialsearch/blob/master/multiagent/multiAgents.py
(underscore between adversarial and search)

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

Not implemented

4.2 References