Artificial intelligence - Project 1
- Search problems -

Malaescu Bianca

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:

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:
"""

    #print("Start:", problem.getStartState())
    #print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    #print("Initial state is ",problem.getStartState())


    #print("Start's successors:",problem.getSuccessors(problem.getStartState()))
    #folosim o stiva
    frontier = util.Stack();
    expanded = []
    initial_state = problem.getStartState()
    frontier.push((initial_state,[])) #starea initiala ; nu s-au facut inca actiuni

    while(not frontier.isEmpty()):
        popped_element = frontier.pop()
        current_state,actions = popped_element

        if(current_state not in expanded): #verificam sa nu fie deja expandata starea
            expanded.append(current_state)
            if(problem.isGoalState(current_state)):#daca e goal returnam actiunile facute sa ajungem in

                return actions

            for successor in problem.expand(current_state): #pt fiecare stare urmatoare
                next_pos, next_action, cost = successor
                frontier.push((next_pos,actions+[next_action]))
```

```
                    #adaugam in frontiera pozitia urmatoare si actiunile pana ajungem la pozitia urmatoare
    util.raiseNotDefined()
```

Explanation:

- Algoritmul este unul de cautare in adancime. Pentru Depth First Search vom folosi o structura de stiva ca sa modelam frontiera. Incepem de la starea initiala a lui PacMan, a carei liste de actiuni este goala. O vom adauga in frontiera. Vom parcurge intr-un while elementele din frontiera pe rand, eliminandu-le din frontiera cand am ajuns la ele. Daca starile in care am ajuns nu au fost deja expandate (verificate), vom face acest lucru. Verificam daca nu cumva am ajuns la goal, caz in care returnam lista de actiuni facute pentru a ajunge aici. Fiecare pozitie urmatoare pe care o putem accesa o vom adauga in frontiera, alaturi de actiunile facute pentru a ajunge la ea si costul actualizat al acestor actiuni.

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: Solutia gasita folosind DFS nu este optima, drumul parcurs de PacMan fiind foarte lung, iar costul foarte mare.

Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: 4/4

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:

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "* YOUR CODE HERE *"

    #se foloseste o coada
```

```
    #frontiera contine pozitia unde se afla acum si action-urile facute ca sa se ajunga acolo
    frontier = util.Queue();
    expanded = []
    initial_state = problem.getStartState()
    frontier.push((initial_state, []))

    while (not frontier.isEmpty()):
        popped_element = frontier.pop()
        current_state, actions = popped_element

        if (current_state not in expanded):  #daca nu a fost expandat se expandeaza
            expanded.append(current_state)
            if (problem.isGoalState(current_state)): #verificam daca nu e goal
                return actions

            for successor in problem.expand(current_state): #o lista cu toate starile viitoare posibile
                next_pos, next_action, cost = successor
                frontier.push((next_pos, actions + [next_action]))
                #se adauga in coada urmatoarea pozitie si actiunile de pana acum + actiunea pt a ajunge
```

Explanation:

- Breadth-first search este un algoritm de cautare in latime. Structura folosita pentru a modela frontierea este o coada (FIFO). Vom incepe de la starea initiala a lui PacMan, a carei liste de actiuni este goala. O vom adauga in frontiera. Cat timp frontiera nu este goala, elementele acesteia le vom parcurge pe rand, eliminandu-le din frontiera cand am ajuns la ele. Daca starile in care am ajuns nu au fost deja expandate (verificate), vom face acest lucru. Verificam daca nu cumva am ajuns la goal, caz in care returnam lista de actiuni facute pentru a ajunge aici. Fiecare pozitie urmatoare pe care o putem accesa o vom adauga in frontiera, alaturi de actiunile facute pentru a ajunge la ea si costul actualizat al acestor actiuni.

Commands:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
- python pacman.py -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: Solutia gasita ruland BFS este optima din punct de vedere al distantei minime, dar nu si din punct de vedere al costului.

Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: 4/4

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):

    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

Explanation:

-

Commands:

-

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:

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:

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

1.3.3 Personal observations and notes

1.4 References

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):
    """Search the node that has the lowest combined cost and heuristic first."""
    "* YOUR CODE HERE *"
    frontier = util.PriorityQueue()
    expanded = []

    initial_state = problem.getStartState()
    frontier.push((initial_state,[],0 ),0)

    while(not frontier.isEmpty()):
        popped_element = frontier.pop()
        current_state,actions,total_cost = popped_element

        if current_state not in expanded:
            expanded.append(current_state)

            if(problem.isGoalState(current_state)):
                return actions
            for successors in problem.expand(current_state):
                next_pos, next_action, next_cost = successors
                new_cost = total_cost + next_cost

                # f(successor) = g(successor) + h(successor)
                # g(s) = costul total
                # h(s) = valoarea euristicii in nodul s

                f = new_cost + heuristic(next_pos,problem)
                frontier.push((next_pos,actions + [next_action], new_cost),f)
```

Listing 1: Solution for the A* algorithm.

Explanation:

- A* este un algoritm de cautare folosit pentru a gasi path-ul de cost minim. Distanta de la starea curenta pana la goal este estimata printr-o functie euristica (f(n) = g(n) + h (n)). g(n) reprezinta costul deplasarii de la starea initiala pana la starea curenta, iar h(n) reprezinta costul estimat al deplasarii de la starea curenta la goal. Algoritmul foloseste ca structura pentru frontiera o coada de prioritati.Fiecare stare introdusa are o prioritate asociata acesteia, prioritate avand starea cu cea mai mica valoare a prioritatii. Incepem algoritmul de la pozitia initiala, care nu are o lista de actiuni iar costul ei este 0 si o adaugam in coada. Cat timp coada de prioritati nu este goala, scoatem cate o stare din aceasta si o expandam in cazul in care nu a fost deja expandata. Verificam daca nu am ajuns la goal, caz in care returnam actiunile. Fiecare stare urmatoare pe care o putem accesa va fi adaugata in coada de prioritati impreuna cu euristica sa calculata.

Commands:

- python autograder.py -q q3

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:

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: 4/4

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):
```

```python

        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 ***"


    def getStartState(self):

        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()

    def isGoalState(self, state):

        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()


    def getSuccessors(self, state):
        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 ***"

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


    def getCostOfActions(self, actions):
        if actions == 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:

-

Commands:

•

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:

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):

    "*** YOUR CODE HERE ***"
    return 0 # Default to trivial solution
```

Explanation:

•

Commands:

•

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:

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:

```
def foodHeuristic(state, problem):
    """
    Your heuristic for the FoodSearchProblem goes here.

    This heuristic must be consistent to ensure correctness.  First, try to come
    up with an admissible heuristic; almost all admissible heuristics will be
    consistent as well.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your heuristic is *not* consistent, and probably not admissible!  On the
    other hand, inadmissible or inconsistent heuristics may find optimal
    solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
    (see game.py) of either True or False. You can call foodGrid.asList() to get
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can query the
    problem.  For example, problem.walls gives you a Grid of where the walls
    are.

    If you want to *store* information to be reused in other calls to the
    heuristic, there is a dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls once and store that
    value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    # fara mazeDistance

    #euristica aleasa: distanta manhattan pana la cea mai apropiata mancare + distanta
    #manhattan pana la ea pana la cea mai indepartata mancare de ea  - consistente
    heuristic = 0
    foodCoordinates = foodGrid.asList() #lista cu coordonatele mancarii
```

```
    if len(foodCoordinates) == 0: #daca nu mai e mancare returnez 0
        return 0

    #doresc sa calculez distanta pana la cea mai apropiata mancare
    closestDistance = None
    posNearestFood = None

    for foodPos in foodCoordinates:
        currentDistance = util.manhattanDistance(position, foodPos)
        if (closestDistance == None or currentDistance < closestDistance) and currentDistance != 0:
            closestDistance = currentDistance
            posNearestFood = foodPos
    #actualizez pozitia pe pozitia celei mai apropiate bucati de mancare
    #aflu distanta manhattan pana la cea mai apropiata mancare si o adaug la euristica
    heuristic += closestDistance

    farthestDistance = None
    for foodPos in foodCoordinates:
        currentDistance = util.manhattanDistance(posNearestFood, foodPos)
        if (farthestDistance == None or currentDistance > farthestDistance):
            farthestDistance = currentDistance

    # aflu distanta manhattan pana la cea mai indepartata mancare (de la mancarea precedenta) si o adaug
    heuristic += farthestDistance

    return heuristic
```

Explanation:

- Euristica aleasa consta din suma dintre distanta manhattan pana la cea mai apropiata bucata de mancare si distanta manhattan de la aceasta la cea mai indeparta bucata de mancare de ea. Cazul particular este cel in care nu mai sunt bucati de mancare in joc si se returneaza 0. Pentru celelalte cazuri, in primul rand se calculeaza disanta manhattan pana la cea mai apropiata bucata de mancare. Se calculeaza distanta manhattan de la pozitia curenta la fiecare bucata de mancare si se compara aceste valori, ramanand salvata cea mai mica dintre acestea. Pozitia curenta se actualizeaza cu pozitia bucatii de mancare cele mai apropiate, iar distanta minima gasita se adauga la euristica. De la noua pozitie se calculeaza distantele manhattan pana la toate celelalte bucati de mancare si se memoreaza cea mai mare, care va fi adaugata la euristica.

Commands:

- python pacman.py -l testSearch -p AStarFoodSearchAgent
- python pacman.py -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?

A1:8178

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):
        """
        Design a better evaluation function here.

        The evaluation function takes in the current and proposed child
        GameStates (pacman.py) and returns a number, where higher numbers are better.

        The code below extracts some useful information from the state, like the
        remaining food (newFood) and Pacman position after moving (newPos).
        newScaredTimes holds the number of moves that each ghost will remain
        scared because of Pacman having eaten a power pellet.

        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)
        childGameState = currentGameState.getPacmanNextState(action) #STAREA URMATOARE
        newPos = childGameState.getPacmanPosition()
        newFood = childGameState.getFood()
        newGhostStates = childGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

        "*** YOUR CODE HERE ***"
        #return childGameState.getScore()
        score = childGameState.getScore() - currentGameState.getScore()  #vaoarea scorului este diferen

        ghostManhattanDist = []  #Distanta manhattan de la urmatoarea stare pana toate fantomele
        for ghost in newGhostStates:
            ghostManhattanDist.append(util.manhattanDistance(newPos, ghost.getPosition()))

        foodManhattanDist = []  #Distanta manhattan de la urmatoarea stare pana toate bucatile de manca
        for foodPosition in newFood.asList():
            foodManhattanDist.append(util.manhattanDistance(newPos, foodPosition))

        if (newPos == currentGameState.getPacmanPosition()):  # Nu vreau ca pacman sa stea pe loc
            score = score - 100
```

```python

        #cu cat pacman este mai departe de o fantoma, cu atat ar trebui sa ii creasca scorul, asa ca ad
        for distance in ghostManhattanDist:
            score = score + distance

        if len(foodManhattanDist) > 0: #verific daca mai exista bucati de mancare in joc
            score = score - min(foodManhattanDist)  #Cu cat pacman este mai departe de o bucata de manc
        else: #daca nu mai sunt le-a mancat pacan pe toate si ii cresc scorul
            score = score + 1000

        return score
```

Explanation:

- Pentru a calcula scorul lui Pacman m-am folosit de distantele Manhattan pana la toate fantomele, respectiv pana la toate bucatile de mancare. Cu cat pacman este mai deprte de o fantoma, cu atat ar trebi sa-i creasca scorul, iar cu cat Pacman este mai aproape de o bucata de mancare, cu atat scorul ii scade mai putin, astel din scor am scazut distanta manhattan pana la cea mai apropiata mancare. Pentru ca Pacman sa se miste permanent, am ales sa-i scad puncte in cazul in care urmatoarea stare aleasa este aceeasi cu starea curenta.

Commands:

- python pacman.py -p ReflexAgent -l testClassic
- python python pacman.py –frameTime 0 -p ReflexAgent -k 1
- python autograder.py -q q1

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: A castigat de 10/10 ori, iar scorul mediu este 1096.5

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:

```python
    def getAction(self, gameState):
        """
        Returns the minimax action from the current gameState using self.depth
        and self.evaluationFunction.

        Here are some method calls that might be useful when implementing minimax.

        gameState.getLegalActions(agentIndex):
        Returns a list of legal actions for an agent
        agentIndex=0 means Pacman, ghosts are >= 1

        gameState.getNextState(agentIndex, action):
        Returns the child game state after an agent takes an action

        gameState.getNumAgents():
        Returns the total number of agents in the game

        gameState.isWin():
        Returns whether or not the game state is a winning state

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        "*** YOUR CODE HERE ***"
        #util.raiseNotDefined()
        action, score = self.minimax(0, 0, gameState)  # Primeste actiunile si scorul pt pacman
        return action  #Returneaza actiunea ce trebuie facuta

    def minimax(self, curr_depth, agent_index, gameState):
        #Pentru pacman, cel mai bun scor e cel maxim, iar pentru fantome cel minim

        #Daca toti agentii si-au terminat miscarea dintr-o tura
        if agent_index >= gameState.getNumAgents():
            agent_index = 0 #revin la primul agent si maresc adancimea
            curr_depth += 1

        # Returneaza rezultatul cand se atinge adancimea maxima
        if curr_depth == self.depth:
            return None, self.evaluationFunction(gameState)

        #O sa pastsrez best_action si best_score
        best_score, best_action = None, None

        if agent_index == 0:  #Pentru randul lui pacman
            for action in gameState.getLegalActions(agent_index):  #Parcurgem fiecare actiune legala a
                # Calculam scorul urmatorilor agenti, adica al tuturor fantomelor
                next_game_state = gameState.getNextState(agent_index, action)
                _, score = self.minimax(curr_depth, agent_index + 1, next_game_state)
                #Daca score e mai mare decat best_score curent, il actualizam
                if best_score is None or score > best_score:
                    best_score = score
                    best_action = action
        else:  #Pentru randul fantomelor
            for action in gameState.getLegalActions(agent_index):  # Parcurgem fiecare actiune legala a
```

```
                # Calculam scorul urmatorului agent
                next_game_state = gameState.getNextState(agent_index, action)
                _, score = self.minimax(curr_depth, agent_index + 1, next_game_state)
                #alegem scorul minim
                if best_score is None or score < best_score:
                    best_score = score
                    best_action = action

        # Daca nu mai avem stari urmatoare posibile, returnam valoare de la evaluationFunction
        if best_score is None:
            return None, self.evaluationFunction(gameState)

        return best_action, best_score  # Returnam best_action si best_score
```

Explanation:

- Am implementat functia minimax, care are ca parametri self, adancimea curenta, indexul aentului si starea jocului. Am inceput cu o reinitializarea indexului agentului si a adancimii, in cazul in care la pasul precedent s-au parcurs toti agentii. Rezultatul functiei il returnam cand se atinge adancimea maxima si nu mai sunt stari urmatoare de evaluat.
- Daca agentul curent e pacman, apelam recursiv functia minimax pentru urmatorul agent. Daca scorul rezultat este mai mare decat best score, actualizam best score si best action.
- Daca agentul curent e fantoma, apelam recursiv functia minimax pentru urmatorul agent. Daca scorul rezultat este mai mic decat best score, actualizam best score si best action.
- Daca best score ramane pe None, inseamna ca nu mai sunt stari urmatoare de explorat si returnam rezultatul obtinut.

Commands:

- python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
- python autograder.py -q q2
- python autograder.py -q q2 –no-graphics

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: Cand pacman este prins intre 2 fantome, se indreapta catre fantoma cea mai apropiata. El isi da seama ca moarte ii este inevitabila, asa ca alege varianta de drum cel mai scurt pana la fantoma, ca sa nu fie penalizat pentru miscarile facute daca mai ramanea in joc.

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):
    """
    Returns the minimax action using self.depth and self.evaluationFunction
    """
    "*** YOUR CODE HERE ***"
    #util.raiseNotDefined()
    inf = float('inf')
    action, score = self.alpha_beta(0, 0, gameState, -inf, inf)  #pe alfa si beta le am luat -inf s
    return action  # Returnam doar actiunea gasita in urma algoritmului

def alpha_beta(self, curr_depth, agent_index, gameState, alpha, beta):
    #daca alpha > beta putem sa ne oprim din generat stari urmatoare posibile si sa "taiem" arborel

    # Daca toti agentii si-au terminat miscarea dintr-o tura
    if agent_index >= gameState.getNumAgents(): #revin la primul agent si maresc adancimea
        agent_index = 0
        curr_depth += 1

    # Returneaza rezultatul cand se atinge adancimea maxima
    if curr_depth == self.depth:
        return None, self.evaluationFunction(gameState)

    #O sa pastsrez best_action si best_score
    best_score, best_action = None, None

    if agent_index == 0:  #daca e randul lui pacman
        for action in gameState.getLegalActions(agent_index):
            # Parcurgem fiecare actiune legala a lui pacman
            # Calculam scorul urmatorilor agenti, adica al tuturor fantomelor
            next_game_state = gameState.getNextState(agent_index, action)
            _, score = self.alpha_beta(curr_depth, agent_index + 1, next_game_state, alpha, beta)


            #Daca score e mai mare ca best_score, il actalizam
            if best_score is None or score > best_score:
                best_score = score
                best_action = action

            # Actualizam valoarea pt alpha
            alpha = max(alpha, score)

            # Daca alpha ajungem mai mare ca beta\, nu mai continuam si "taiem" arborele
            if alpha > beta:
                break
    else:  #Daca e randul fantomelor
```

```
            for action in gameState.getLegalActions(agent_index):  # Parcurgem fiecare actiune legala a
                # Calculam scorul urmatorului agent
                next_game_state = gameState.getNextState(agent_index, action)
                _, score = self.alpha_beta(curr_depth, agent_index + 1, next_game_state, alpha, beta)

                #Daca score e mai mic ca best_score, il actualizam pe best_score
                if best_score is None or score < best_score:
                    best_score = score
                    best_action = action

                # Actualizam valoarea entru beta
                beta = min(beta, score)

                # Daca alpha ajungem mai mare ca beta, nu mai continuam si "taiem" arborele
                if beta < alpha:
                    break

        # Daca nu mai avem stari urmatoare posibile, returnam valoare de la evaluationFunction
        if best_score is None:
            return None, self.evaluationFunction(gameState)
        return best_action, best_score  # Returnam best_action si best_score
```

Explanation:

- Singura diferenta fata de acest algoritm si mini-max este faptul ca se mai adauga 2 parametri, alpha si beta. Alpha e initializata la inceput cu -inf, iar beta cu +inf. Alpha se actualizeaza mereu cu scorul maxim, iar beta cu scorul minim. Cand alpha ajunge sa fie mai mare ca beta, inseamna ca nu mai are rost sa exploram stărileurmatoare, si putem "taia" acea parte a arborelui. In acest moment putem returna rezultatul.

Commands:

- 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

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:

```
1
```

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