Artificial intelligence - Project 1
- Search problems -

Dunca Denisa Mihaela

2/11/2021

1 Uninformed search

1.1 Question 1 - Depth-first search

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

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

1.1.1 Code implementation

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

Code:

```
1  def depthFirstSearch(problem):
2  
3      # calea pe care va merge pacman
4      path = {}
5      # in stiva tinem nodurile neexplorate
6      frontier = util.Stack()
7      # se pune in stiva state-ul de start
8      frontier.push(problem.getStartState())
9      # o lista de noduri explorate pentru a ne asigura ca nu le parcurgem de mai multe ori
10     explored = []
11     path[problem.getStartState()] = []
12     # daca stiva e goala si nu am gasit finish ul inseamna ca nu exista goal
13     while not (frontier.isEmpty()):
14         # scoatem primul nod din stiva si il punem il lista de noduri explorate
15         current = frontier.pop()
16         explored.append(current)
17         # verificam daca am ajuns la goal
18         if problem.isGoalState(current):
19             return path[current]
20         # adaugam celelalte noduri neexplorate in stiva (succesorii)
21         for (state, action, cost) in problem.expand(current):
22             if state not in explored:
23                 frontier.push(state)
24                 path[state] = path[current] + [action]
25     return path[current]
26     #util.raiseNotDefined()
```

Explanation:

- La linia 4 am declarat un path care reprezinta calea pe care o va parcurge pacman
- In stiva frontier tinem nodurile care nu au fost inca explorate si o initializam cu primul state, cel de start
- De asemenea, am adaugat si o lista de noduri explorate, numita explored, pentru a nu parcurge de mai multe ori acelasi nod La linia 13 verificam dac alista este goala, in cazul in care stiva este goala si nu am gasit inca goal state-ul, se scoate primul nod din stiva si il punem in lista de noduri explorate
- La linia 18 se face o verificare daca nodul curent este cel de finish, daca este, returnam calea pana la el, daca nu este atunci la linia 21 adaugam celelalte noduri neexplorate in stiva, adica succesorii

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: DFS nu este optimal deoarece parcurge in lungime si pentru ca ia prima cale care ajunge la goal, care nu este neaparat cea mai buna solutie.
Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: 4/4

1.1.3 Personal observations and notes

In ultimul if trebuia doar sa verific daca nodul a fost sau nu explorat, initial verificam si daca nodul se afla in stiva si aveam erori, am rezolvat problema cautand doar in lista cu noduri deja explorate.

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):

    # calea pe care va merge pacman
    path = {}
    # in coada tinem nodurile neexplorate
    frontier = util.Queue()
    # se pune in coada state-ul de start
    frontier.push(problem.getStartState())
    # o lista de noduri explorate pentru a ne asigura ca nu le parcurgem de mai multe ori
    explored = []
    path[problem.getStartState()] = []
    explored.append(problem.getStartState())
    while not (frontier.isEmpty()):
        # scoatem primul nod din coada
        current = frontier.pop()
```

```
16          # verificam daca este goal-ul
17          if problem.isGoalState(current):
18              return path[current]
19          # adaugam toti succesorii neexplorati in coada
20          for (state, action, cost) in problem.expand(current):
21              if state not in explored and state not in frontier.list:
22                  frontier.push(state)
23                  path[state] = path[current] + [action]
24                  explored.append(state)
25      return path[current]
26      #util.raiseNotDefined()
```

Explanation:

- La linia 4 declarat un path care reprezinta calea pe care o va parcurge pacman
- In coada frontier tinem nodurile care nu au fost inca explorate si o initializam cu primul state, cel de start
- De asemenea, am adaugat si o lista de noduri explorate, numita explored, pentru a nu parcurge de mai multe ori acelasi nod La linia 13 verificam daca este goala coada, in cazul in care coada este goala si nu am gasit inca goal state-ul, se scoate primul nod din coada
- La linia 17 se face o verificare daca nodul curent este cel de finish, daca este, returnam calea pana la el, daca nu este atunci la linia 20 adaugam celelalte noduri neexplorate in coada, adica succesorii
- Adaugam apoi fiecare succesor explorat in lista cu noduri explorate la linia 24

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: BFS este optimal pentru ca parcurge pe nivel si poate sa afle cea mai buna solutie, aceasta fiind cea gasita la nivelul cel mai mic.
Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: 4/4

1.2.3 Personal observations and notes

Nu am intampinat probleme la indeplinirea acestui task.

1.3 References

https://cs.stanford.edu/people/abisee/tutorial/bfsdfs.html

2 Informed search

2.1 Question 3 - 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:

```
1  def aStarSearch(problem, heuristic=nullHeuristic):
2  
3     # folosim PriorityQueue pentru ca este importanta ordinea elementelor in queue in functie de h
4      frontier = util.PriorityQueue()
5      # o lista de noduri explorate pentru a ne asigura ca nu le parcurgem de mai multe ori
6      explored = []
7      # tine o tupla de state uri si path ul ca sa ajungem intr-un state
8      startingTuple = (problem.getStartState(), [], 0)
9      frontier.push(startingTuple, 0)
10     while not frontier.isEmpty():
11         # dau pop la nodul cu euristica cea mai mica ( primul din PriorityQueue)
12         current = frontier.pop()
13         state = current[0]
14         path = current[1]
15         cost = current[2]
16         # nu exploram stari deja explorate
17         if state in explored:
18             continue
19         explored.append(state)
20         # vedem daca starea este goal
21         if problem.isGoalState(state):
22             return path
23         #adaugam toti succesorii neexplorati in coada
24         for child in problem.expand(state):
25             childState = child[0]
26             newAction = child[1]
27             newCost = child[2]
28             if childState not in explored:
29                 # schimbam itemul din coada cu unul care are o prioritate mai mica
30                 frontier.update((childState, path + [newAction], cost + newCost),
31                                 cost + newCost + heuristic(childState, problem))
32     return []
33     #util.raiseNotDefined()
```

Listing 1: Solution for the A* algorithm.

Explanation:

- La linia 3 declaram variabila frontier care reprezinta un priority queue deoarece aici este importanta ordinea elementelor in coada in funtie de euristica data
- Am declarat si o lista de noduri explorate astfel in cat sa nu parcurgem aceleasi noduri de mai multe ori
- La linia 8 am declarat o tupla care tine un state si path-ul nostru si am initializat-o cu starea de start
- Se face un while in care se verifica de fiecare data daca priority queue-ul nu este empty, iar la linia 11 se da pop out la primul nod care este si cel cu euristica cea mai mica
- La linia 17 verificam daca state-ul in care ne aflam a fost sau nu explorat, iar daca nu a fost explorat il adaugam la lista explored
- La linia 21 se verifica daca stat-ul la care suntem este sau nu goal state, in cazul in care este atunci returnam path-ul, iar daca nu este atunci verificam fiecare succesor neexplorat in forul dintre liniile 24-30
- In acest for, la linia 28 se verifica daca succesorul a fost explorat, iar daca nu a fost atunci schimbam nodul din coada cu unul care are o prioritate mai mica, queue-ul isi face update : starea devine starea succesorului, pathul se prelungeste cu un pas dat de actiunea pe care o face pacman, iar costului i se adauga costul succesorului
- De asemenea, costul total creste cu noul cost si cu heuristica pe care o are starea succesorului.

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: Solutiile sunt diferite, deoarece rezultatul de la Astar este influentat si de euristica, pe cand UCS, fiind un algoritm neinformat, nu are euristicae
Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2: Da, Astar incearca sa aiba mai putine noduri expandate (pathCost + heuristic function)
Q3: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).
A3: 4/4

2.1.3 Personal observations and notes

Nu am intampinat probleme la aceast exercitiu.

2.2 Question 4 - 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:

```
 1 class CornersProblem(search.SearchProblem):
 2 
 3     def __init__(self, startingGameState):
 4 
 5         self.walls = startingGameState.getWalls()
 6         self.startingPosition = startingGameState.getPacmanPosition()
 7         top, right = self.walls.height-2, self.walls.width-2
 8         self.corners = ((1,1), (1,top), (right, 1), (right, top))
 9         for corner in self.corners:
10             if not startingGameState.hasFood(*corner):
11                 print('Warning: no food in corner ' + str(corner))
12         self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
13         # Please add any code here which you would like to use
14         # in initializing the problem
15         "*** YOUR CODE HERE ***"
16 
17 
18     def getStartState(self):
19 
20         "*** YOUR CODE HERE ***"
21         return(self.startingPosition, ())
22         #util.raiseNotDefined()
23 
24     def isGoalState(self, state):
25 
26         "*** YOUR CODE HERE ***"
27         # daca oricare dintre colturi nu este vizitat, se returneaza fals
28         for i in self.corners:
29             if i not in state[1]:
30                 return False
31         return True
32         #util.raiseNotDefined()
33 
34 
35     def expand(self, state):
36 
37             "*** YOUR CODE HERE ***"
38         children = []
39         for action in self.getActions(state):
40             child = self.getNextState(state, action)
41             stepCost = self.getActionCost(state, action, child)
42             children.append((child, action, stepCost))
43 
44         self._expanded += 1 # DO NOT CHANGE
45         return children
```

```
46
47      def getActions(self, state):
48          possible_directions = [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
49          valid_actions_from_state = []
50          for action in possible_directions:
51              x, y = state[0]
52              dx, dy = Actions.directionToVector(action)
53              nextx, nexty = int(x + dx), int(y + dy)
54              if not self.walls[nextx][nexty]:
55                  valid_actions_from_state.append(action)
56          return valid_actions_from_state
57
58      def getActionCost(self, state, action, next_state):
59          assert next_state == self.getNextState(state, action), (
60              "Invalid next state passed to getActionCost().")
61          return 1
62
63      def getNextState(self, state, action):
64
65          assert action in self.getActions(state), (
66              "Invalid action passed to getActionCost().")
67          x, y = state[0]
68          dx, dy = Actions.directionToVector(action)
69          nextx, nexty = int(x + dx), int(y + dy)
70          "*** YOUR CODE HERE ***"
71          # verificam daca next state intra intr-un zid
72          if not self.walls[nextx][nexty]:
73              visited = state[1]
74              # daca next state este un corner si nu a fost vizitat, il adaugam la nodurile vizitate
75              if (nextx, nexty) in self.corners and (nextx, nexty) not in visited:
76                  visited += ((nextx, nexty),)
77              # adaugam next state-ul la lista de succesori
78              children = ((nextx, nexty), visited)
79              return children
80          return 0
81          #util.raiseNotDefined()
82
83      def getCostOfActions(self, actions):
84          if actions == None: return 999999
85          x,y= self.startingPosition
86          for action in actions:
87              dx, dy = Actions.directionToVector(action)
88              x, y = int(x + dx), int(y + dy)
89              if self.walls[x][y]: return 999999
90          return len(actions)
91
```

Explanation:

- La linia 21 in functia getStartState am aduagat un return care returneaza pozitia de start
- In isGoalState la liniile 28-31 am adaugat un for care parcurge fiecre dintre cele patru colturi si verifica daca a fost sau nu verificat
- In fucnctia de expand se calculeaza succesorii, intr-un for care contine toate actiunile care se pot face dintr-un anumit state, adaugam la lista de succesori : starea urmatoare, actiunea pe care o face si costul actiunii

- In functia getNextState intre liniile 72- 80 am implementat o verificare daca next state intra intr-un zid, in cazul in care nu este zid, verificam daca este un corner si daca nu a fost vizitat. In caz afirmativ adaugam next state-ul la nodurile vizitate, iar apoi la lista de succesori si returnam aceasta lista

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 nodes: 2448

2.2.3 Personal observations and notes

Am intampinat oarecum cateva probleme la implementarea functiei de nextState deoarece nu implementam si rezolvarea problemei cu zidurile, de asemenea la inceput pacaman manca doar o bucata si apoi se oprea, am descoperit ca nu rezovasem bine problema din functia isGoalState, si de aceea se oprea dupa prima bucata. Am rezolvat problema printr-un for care sa parcurga toate cele patru colturi.

2.3 Question 5 - 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:

```
def cornersHeuristic(state, 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 ***"
    currentState = state[0]
    visited = state[1]
    total = 0
    # daca colturile nu sunt vizitate le adaugam in lista de colturi
    for i in problem.corners:
        if i not in visited:
```

```
12             corners.append(i)
13     # cat timp exista un colt in lista de colturi
14     while len(corners) > 0:
15         mini = 100000
16         nextCorner = currentState
17         # parcurgem lista de colturi
18         for i in corners:
19             # calculam distanta manhattan de la starea curenta la fiecare colt
20             distance = util.manhattanDistance(i, currentState)
21             # cautam distanta minima
22             if mini > distance:
23                 mini = distance
24                 nextCorner = i
25         currentState = nextCorner
26         # scoatem din lista  coltul pentru care am calculat distanta minima
27         corners.remove(nextCorner)
28         # adunam distanta minima la un total si il returnam
29         total += mini
30     return total
31
```

Explanation:

- La linia 10 verificam daca fiecare colt nu este vizitat si le adaugam in lista de colturi in cazul in care nu sunt
- In while-ul ce incepe de la linia 14 verificam daca lista de colturi nu este goala, cat timp exista macar un element vom parcurge aceasta lista
- Pentru fiecare element din lista calculam distanta manhattan de la el la state-ul unde ne aflam
- Intre liniile 22-24 comparam distantele intre ele astfel incat sa gasim distanta minima
- Apoi scoatem din lista coltul pentru care am calculat distanta minima si adunam aceasta distanta la total pentru a-l returna la final

Commands:

- 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: Number of expanded nodes: 901

2.3.3 Personal observations and notes

Nu am intampinat probleme in timpul rezolvarii acestui task.

2.4 Question 6 - 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):

     position, foodGrid = state
    "*** YOUR CODE HERE ***"
    # lista cu locurile unde se afla mancarea in grid
    food = foodGrid.asList()
    cost = 0
    # pentru fiecare bucata de mancare din lista
    for i in food:
        # calculam distanta cu mazeDistance
        distance = (mazeDistance(position, i, problem.startingGameState))
        # cautam distanta maxima si o punem in variabila cost iar apoi o returnam
        cost = max(cost, distance)
    return cost
```

Explanation:

- In primul rand am creat o lista numita food care contine toata mancarea ce urmeaza sa fie mancata
- Pentru fiecare bucata de mancare din lista cu mancare se calculeaza distanta mazeDistance intre pozitia in care ne aflam si mancare, iar apoi facem maximul dintre toate distantele posibile si le adaugam la cost pentru a-l returna la final.

Commands:

- python pacman.py -l testSearch -p AStarFoodSearchAgent
- -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: Number of expanded nodes: 4137

2.4.3 Personal observations and notes

Am intampinat probeleme la calculul maximului dintre distante, dar am reusit sa le rezolv folosind functia max().

2.5 Question 7 - Suboptimal Search

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

"In this section, you'll write an agent that always greedily eats the closest dot. ***ClosestDotSearchAgent*** *is implemented for you in* ***searchAgents.py****, but it's missing a key function that finds a path to the closest dot. Implement the function* ***findPathToClosestDot*** *in* ***searchAgents.py****"*

2.5.1 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 AnyFoodSearchProblem(PositionSearchProblem):
2  
3      def isGoalState(self, state):
4          """
5          The state is Pacman's position. Fill this in with a goal test that will
6          complete the problem definition.
7          """
8          x, y = state
9  
10         "*** YOUR CODE HERE ***"
11         if (x, y) in self.food.asList() or len(self.food.asList()) == 0:
12             return True
13         return False
14         #util.raiseNotDefined()
15 
16 class ClosestDotSearchAgent(SearchAgent):
17 
18     def findPathToClosestDot(self, gameState):
19         problem = AnyFoodSearchProblem(gameState)
20         "*** YOUR CODE HERE ***"
21         return search.breadthFirstSearch(problem)
```

Explanation:

- La linia 11 se verifica daca x si y fac parte din starile unde se afla mancare in lista cu mancare sau daca lungimea listei de mancare este 0 astfel incat sa returneze true daca toata mancarea a fost mancata si false in caz contrar.
- La linia 21 am pus si ce am returnat in ClosestDotSearchAgent si anume am facut BFS pe probelma data.

Commands:

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

2.5.2 Questions

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

Q1: What is the cost for the path found?
A1: Cost: 350

2.6 References

https://stackoverflow.com/questions/44151713/what-is-the-difference-between-uniform-cost-search-and-best-first-search-methods

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   def evaluationFunction(self, currentGameState, action):
2
3           from util import manhattanDistance
4           # Useful information you can extract from a GameState (pacman.py)
5           childGameState = currentGameState.getPacmanNextState(action)
6           newPos = childGameState.getPacmanPosition()
7           newFood = childGameState.getFood()
8           newGhostStates = childGameState.getGhostStates()
9           newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
10
11          "*** YOUR CODE HERE ***"
12          # verificam daca newPos ne duce prea aproape de o fantoma sau nu
13          closeGhost = 0
14          for state in newGhostStates:
15              if manhattanDistance(state.getPosition(), newPos) < 2:
16                  closeGhost = -1
17          # se calculeaza distanta medie pana la toata mancarea
18          distance = []
19          for i in range(newFood.width):
20              for j in range(newFood.height):
21                  if newFood.data[i][j]:
22                      distance.append(manhattanDistance(newPos, (i, j)))
23
24          if len(distance) == 0:
25              avg = 0
26          else:
27              avg = sum(distance)/len(distance)
28
29          return childGameState.getScore() + closeGhost - avg
```

Explanation:

- La liniile 14-16 verificam daca newPos nu ne duce prea aproape de o fantoma folosind manhattanDistance
- La liniile 19-22 calculam distanta de la pozitia unde ne aflam la fiecare bucatica de mancare
- La liniile 24-27 calculam distanta medie

- La final returnam scorul din care scadem distanta medie parcursa.

Commands:

- python pacman.py -p ReflexAgent -l testClassic
- python pacman.py –frameTime 0 -p ReflexAgent -k 1
- python pacman.py –frameTime 0 -p ReflexAgent -k 2

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: In medie scorul este de 549 de puncte.

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:

Explanation:

-

Commands:

-

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:

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:

Explanation:

-

Commands:

-

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:

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:

```
29
```

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