Artificial intelligence - Project 1
- Search problems -

Pop Ruxandra Maria, Zelenszky Bianca

02/11/2021

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:

```
def depthFirstSearch(problem):

    "*** Date initiale ***"

    solution = [] # lista solutiei
    exploredList = [] # lista nodurilor explorate
    frontier = util.Stack() # frontiera e un stack

    isInFrontier = False  # variabila care indica daca nodul e in frontiera
    isInExploredList = False  # variabila care indica daca nodul e in lista nodurilor explorate
    isNodeAddedPrev = True  # variabila care indica daca nodul a fost adaugat ultima data in frontiera


    "*** Expandam starea initiala *** "
    successors = problem.expand(problem.getStartState()) # succesorii

    for i in range(0, len(successors)):
        node = CustomNode(successors[i][0], successors[i][1], # CustomNode(state, action, cost, parent)
                                successors[i][2], (-1, -1)) # (-1, -1) indica faptul ca nodul nu are p
        frontier.push(node)


    "*** Expandam starile urmatoare ***"
    currentNode = frontier.pop() # scoatem primul nod din frontiera

    while not problem.isGoalState(currentNode.getState()) : # cat timp nu a gasit goal-ul

        " Atunci cand s-a atins adancimea maxima, se scod nodurile pana la nodul curent  "
        if not isNodeAddedPrev :

            while exploredList[-1].parent != currentNode.getParent() :
                exploredList.pop(-1)
                solution.pop(-1)

            exploredList.pop(-1)
            solution.pop(-1)
```

```
37
38          exploredList.append(currentNode) # se adauga in exploredList nodul curent
39          solution.append(currentNode.getAction()) # se adauga in solution actiunea nodului curent
40
41
42          isNodeAddedPrev = False
43          successors = problem.expand(currentNode.getState()) # se expandeaza succesorii nodului curent
44
45          for i in range(0, len(successors)):
46              isInExploredList = False
47              isInFrontier = False
48              node = CustomNode(successors[i][0], successors[i][1],
49                                      successors[i][2], currentNode.getState()) # parintele succesorului c
50
51              if node.getState() != currentNode.getParent() : # nu adauga parintele nodului curent (sa nu
52
53                  if doesStackHaveThisItem(frontier, node) : # verifica daca nodul se afla in frontiera
54                          isInFrontier = True
55
56                  for i in range(0, len(exploredList)) :
57                      if node.state == exploredList[i].state : # verifica daca nodul se afla in lista nod
58                          isInExploredList = True
59
60                  if not isInExploredList and not isInFrontier : # verifica daca nodul nu e in exploredLi
61                      frontier.push(node)  # se adauga nodul in frontiera
62                      isNodeAddedPrev = True
63
64          currentNode = frontier.pop() # se scoate un nod din frontiera
65
66      "*** Solutia ***"
67      solution.append(currentNode.getAction())
68
69      return solution
70
71  def doesStackHaveThisItem(stack, item):
72
73      popped = []
74      ok = False
75      while not stack.isEmpty(): # golim stack-ul
76          popped.append(stack.pop())
77          if item == popped :
78              ok = True
79
80      while len(popped) != 0 : # umplem stack-ul
81          stack.push(popped[-1])
82          popped.pop(-1)
83
84      return ok
```

Explanation:

- la linia 7 declaram frontiera ca o stiva
- la linia 18 si 48 ne folosim de un tip creat de noi, CustomNode, pentru a retine starea, actiunea, costul si parintele unui succesor
- la linia 19 se observa ca parintele pozitiei de start este (-1, -1)

- la linia 39 se observa ca solutia este construita pe parcursul algoritmului prin adaugarea actiunii nodului curent in lista solutiei
- la linia 71 se observa functia doesStackHaveThisItem pe care o apelam in acest algoritm

Commands:

- python pacman.py -l tinyMaze -p SearchAgent
- python pacman.py -l smallMaze -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: Nu este o solutie optima pentru rezolvarea problemei de cautare deoarece este un algoritm DFS, care e mai rapid decat alti algoritmi, dar nu garanteaza solutia optima.
Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: Question q1: 4/4

1.1.3 Personal observations and notes

Pentru a verifica daca succesorul unui nod este deja in frontiera, am creat o functie numita doesStack-HaveThisItem (linia 74), care returneaza True daca a gasit nodul in frontiera si False daca nu l-a gasit.
Pe layout-ul tinyMaze se expandeaza 16 de noduri.
Pe layout-ul smallMaze se expandeaza 59 noduri.
Pe layout-ul mediumMaze se expandeaza 146 noduri.
Pe layout-ul bigMaze se expandeaza 391 noduri.

Algoritmul de cautare:

- nu este complet
- nu este optim
- are complexitatea timpului exponentiala
- are complexitatea spatiului liniara

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):


    "*** Date initiale ***"
    solution = [] # lista de solutii
    exploredList = [] # lista de noduri explorate
    frontier = util.Queue() # frontiera reprezentata ca si coada


    "*** Expandam starea initiala *** "
    successors = problem.expand(problem.getStartState()) # succesorii

    for i in range(0, len(successors)):
        node = CustomNode(successors[i][0], successors[i][1], # CustomNode(state, action, cost, par
                          successors[i][2], (-1, -1))  # (-1, -1) indica faptul ca nodul nu are par
        frontier.push(node)
        exploredList.append(node) # punem in acelasi timp nodurile in lista de stari expandate


    "*** Expandam starile urmatoare ***"
    currentNode = frontier.pop() # scoatem primul nod din frontiera

    while not problem.isGoalState(currentNode.getState()) : # cat timp nu a gasit goal-ul

        successors = problem.expand(currentNode.getState()) # se expandeaza succesorii nodului cur

        for i in range(0, len(successors)):
            isInFrontier = False  # variabila care indica daca nodul e in frontiera
            isInExploredList = False # variabila care indica daca nodul e in exploredList
            node = CustomNode(successors[i][0], successors[i][1],
                              successors[i][2], currentNode.getState()) # parintele succesorului es
            if node.getState() != currentNode.getParent() and node.getState() != problem.getStartSt
                if doesQueueHaveThisItem(frontier, node) : # verifici daca nodul e in frontiera
                    isInFrontier = True
                for i in range(0, len(exploredList)): # verifici daca nodul e in exploredList
                    if node.state == exploredList[i].state:
                        isInExploredList = True

                if not isInExploredList and not isInFrontier : # verifica daca nodul nu e in explor
                    frontier.push(node)
                    exploredList.append(node)  # se adauga in acelasi timp si in exploredList

        currentNode = frontier.pop()
        exploredList.append(currentNode) # se adauga nodul curent in exploredList
```

```
45
46
47      "*** Solutia ***"
48      list = [] # lista care va contine reverse-ul solutiei
49      list.append(exploredList[-1]) # primul element e ultimul element adaugat in exploredList
50
51      while list[-1].getParent() != (-1, -1) : # cat timp nu a ajuns la parintele nodului de start
52          for i in range(len(exploredList) - 1, -1, -1):
53              if list[-1].getParent() == exploredList[i].getState() : # daca parintele ultimului nod
54                  list.append(exploredList[i]) # se adauga nodul in lista
55                  break
56
57      for i in range(len(list) - 1, -1, -1):
58          solution.append(list[i].getAction()) # adaugi la solutie reverse-ul listei
59
60      return solution
61
62
63  def doesQueueHaveThisItem(queue, item):
64      popped = []
65      ok = False
66      while not queue.isEmpty():  # golim coada
67          popped.append(queue.pop())
68          if item == popped:
69              ok = True
70
71      while len(popped) != 0:  # umplem coada
72          queue.push(popped[0])
73          popped.pop(0)
74
75      return ok
```

Explanation:

- la linia 7 declaram frontiera ca o coada
- la linia 14 si 30 ne folosim de o structura de date de tip CurrentNode care retine starea, actiunea, costul si parintele unui nod
- la linia 32, deoarece nu am pus starea initiala in exploredList, verificam ca nodul succesor sa nu fie nodul de start
- la linia 52 se observa ca am parcurs exploredList in ordine inversa, pentru ca sa se realizeze comparatia dintre list si exploredList cat mai repede
- la linia 63 am declarat functia doesQueueHaveThisItem pe care o apelam in algoritm

Commands:

- python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
- python pacman.py -l smallMaze -p SearchAgent -a fn=bfs
- python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
- python pacman.py -l bigMaze -z .5 -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: Da, solutia este optima pentru o problema de cautare, datorita modului de functionare a BFS-ului (cautarea in latime), cu toate ca expandeaza foarte multe noduri si este mai incet decat alti algoritmi de cautare.

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

A2: Question q2: 4/4

1.2.3 Personal observations and notes

Pentru a verifica daca un nod se afla in frontiera am creat o functie numita doesQueueHaveThisItem care returneaza True daca gaseste nodul in coada si False daca nu-l gaseste.
Pentru a grabi gasirea solutiei, am ales sa punem in acelasi timp succesorii in frontiera si in lista de explorat.
Pe layout-ul tinyMaze se expandeaza 15 de noduri.
Pe layout-ul smallMaze se expandeaza 92 noduri.
Pe layout-ul mediumMaze se expandeaza 269 noduri.
Pe layout-ul bigMaze se expandeaza 620 noduri.

Algoritmul de cautare este:

- complet daca spatiul starilor este finit
- optim daca costul este constant intre noduri
- are complexitatea timpului exponentiala
- are complexitatea spatiului exponentiala

1.3 Question 3 - Uniform-cost search

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

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

1.3.1 Code implementation

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

Code:

```
def uniformCostSearch(problem):

    "*** 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):


    "*** Date initiale ***"
    solution = [] # lista solutiei
    exploredList = [] # lista nodurilor explorate
    frontier = util.PriorityQueue() # frontiera e o coada de prioritate


    "*** Expandam starea initiala ***"
    successors = problem.expand(problem.getStartState()) # sucesorii
    for i in range(0, len(successors)):
        position = successors[i][0] # pozitia succesorului
        g = successors[i][2] # costul succesorului
        h = heuristic(position, problem) # euristica problemei
        f = g + h # valoarea prioritatii
        node = CustomNode(successors[i][0], successors[i][1], g, (-1, -1)) # (-1, -1) indica f
        frontier.update(node, f)


    "*** Expandam starile urmatoare ***"
    currentNode = frontier.pop()
    solution = solution + [currentNode.action] # construim solutia

    while not problem.isGoalState(currentNode.state): # cat timp nu a gasit goal-ul

        exploredList.append(currentNode) # se pune nodul curent in lista de noduri explorate

        successors = problem.expand(currentNode.state) # se expandeaza nodul curent
        for i in range(0, len(successors)):
            isInExploredList = False # variabila care indica daca nodul se afla in exploredLis
            node = CustomNode(successors[i][0], successors[i][1],
                              successors[i][2], currentNode.getState())  # parintele succesoru

            if node.getState() != currentNode.getParent() and node.getState() != problem.getSta
```

```
36                  for i in range(0, len(exploredList)):
37                     if node.state == exploredList[i].state: # verifica daca nodul curent e in ex
38                          isInExploredList = True
39
40                  if not isInExploredList: # daca nu e in exploredList
41                      node.action = solution + [node.action] # se actualizeaza solutia
42                      g = problem.getCostOfActionSequence(node.action) # se determina costul de
43                      position = node.state  # se actualizeaza pozitia
44                      h = heuristic(position, problem)  # euristica
45                      f = g + h  # prioritatea
46                      frontier.update(node, f) # se adauga succesorul in frontiera
47
48          isInExploredList = True
49          while isInExploredList and not frontier.isEmpty(): # cat timp gaseste un nod din front
50            isInExploredList = False
51            currentNode = frontier.pop()
52            if currentNode.parent == (-1, -1) : # daca nodul parinte e (-1, -1), se reactualizea
53                solution = [] + [currentNode.action]
54            else :
55                solution = currentNode.action
56            for i in range(0, len(exploredList)):
57              if currentNode.state == exploredList[i].state:
58                  isInExploredList = True
59
60
61      "*** Solutia ***"
62      solution = currentNode.action
63      return solution
```

Explanation:

* la linia 7 se observa ca folosim frontiera ca si o coada de prioritate
* la linia 17 si 32 am folosit o structura de date CustomNode care are ca argumente pozitia starea, actiunea, costul si parintele nodului
* la liniile 18 si 46 se observa folosirea functiei update() din cadrul tipului PriorityQueue pentru a adauga in frontiera nodurile expandate si a actualiza prioritatea in acelasi timp
* la liniile 48 - 58 se scot din frontiera nodurile care se afla in exploredList (pentru a nu explora de doua ori un nod)

Commands:

* python pacman.py -l tinyMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
* python pacman.py -l smallMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
* python pacman.py -l mediumMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
* 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: Cu toate ca nu am facut UCS, solutiile dintre A* si UCS ar trebui sa difere in functie de euristica. Daca euristica la A* e 0, atunci solutia lor este aceeasi, dar daca A* are o alta euristica, solutiile poate sa difere in functie de admisibilitatea si consistenta euristicii. Daca e admisibila si consistenta, atunci A* va avea solutia optima si aceeasi solutie ca UCS, altfel nu.
Q2: Does A* finds the solution with fewer expanded nodes than UCS?

A2: Folosind euristica Manhattan pentru calcularea distantelor dintre starea initiala si celulele din grid, se expandeaza mai putine noduri decat daca as folosi euristica nula.

Q3: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).

A3: Question q3: 4/4

2.1.3 Personal observations and notes

Pentru a rezolva testcase-ul graphmanypaths, am scris liniile de cod 48 - 58, unde, pentru ca am constatat ca se pot pune in frontiera noduri care au fost deja in exploredList, am creat un loop care scoate din frontiera un nod si le compara cu nodurile explorate. De asemenea, valoarea solutiei va fi actiunea curenta daca nodul care a fost scos din frontiera nu e radacina, sau se va crea o solutie noua daca a ajuns sa fie nodul de start.
Pe layout-ul tinyMaze expandeaza 15 de noduri.
Pe layout-ul smallMaze expandeaza 92 noduri.
Pe layout-ul mediumMaze expandeaza 269 noduri.
Pe layout-ul bigMaze expandeaza 620 noduri.

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:

```python
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and child 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

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

        " Tupla ce contine pozitia de start si lista nodurilor vizitate "
        if self.startingPosition in self.corners :
            return (self.startingPosition, [self.startingPosition])
        else:
            return (self.startingPosition, [])

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"

        " Daca a vizitat toate corner-rurile ne oprim "
        if len(state[1]) == 4 : return True
        return False

    def expand(self, state):
        """
        Returns child 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, (child,
            action, stepCost), where 'child' is a child to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that child
        """

        children = []
        for action in self.getActions(state):
            # Add a child state to the child list if the action is legal
            # You should call getActions, getActionCost, and getNextState.
            "*** YOUR CODE HERE ***"

            nextState = self.getNextState(state, action) # calculeaza starea urmatoare
            stepCost = self.getActionCost(state, action, nextState) # calculeaza actiunea urma
            children.append((nextState, action, stepCost)) # calculeaza costul urmator
        self._expanded += 1 # DO NOT CHANGE

        return children
```

```
69
70      def getActions(self, state):
71          possible_directions = [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.
72          valid_actions_from_state = []
73          for action in possible_directions:
74              x, y = state[0]
75              dx, dy = Actions.directionToVector(action)
76              nextx, nexty = int(x + dx), int(y + dy)
77              if not self.walls[nextx][nexty]:
78                  valid_actions_from_state.append(action)
79          return valid_actions_from_state
80
81      def getActionCost(self, state, action, next_state):
82          assert next_state == self.getNextState(state, action), (
83              "Invalid next state passed to getActionCost().")
84          return 1
85
86      def getNextState(self, state, action):
87          assert action in self.getActions(state), (
88              "Invalid action passed to getActionCost().")
89          x, y = state[0]
90          dx, dy = Actions.directionToVector(action)
91          nextx, nexty = int(x + dx), int(y + dy)
92          "*** YOUR CODE HERE ***"
93
94          cornersList = state[1] # lista de corner-uri vizitate
95          nextState = (nextx, nexty)
96          if not self.walls[nextx][nexty]: # daca nu e wall
97
98              " Daca gaseste un corner care nu a fost adaugat in lista de corner-uri " \
99              " se adauga in lista  "
100             for i in range(0, len(self.corners)):
101                 if nextState == self.corners[i] and nextState not in cornersList:
102                     cornersList = cornersList + [nextState]
103
104         return (nextState, cornersList)
105
106     def getCostOfActionSequence(self, actions):
107         """
108         Returns the cost of a particular sequence of actions.  If those actions
109         include an illegal move, return 999999.  This is implemented for you.
110         """
111         if actions == None: return 999999
112         x,y= self.startingPosition
113         for action in actions:
114             dx, dy = Actions.directionToVector(action)
115             x, y = int(x + dx), int(y + dy)
116             if self.walls[x][y]: return 999999
117         return len(actions)
```

Explanation:

- dupa cum se poate vedea la liniile 22, 24 si 104, informatia am retinut-o intr-o tupla de forma (state, cornersList), unde state este starea si cornersList este o lista cu corner-urile vizitate de la nodul de start pana in starea state
- la liniile 100 - 102, adaugam in cornersList corner-urile pe masura ce le gasim

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: Search nodes expanded: 2448 (pentru BFS CornersProblem)
Search nodes expanded: 901 (pentru A* cu cornersHeuristic)

2.2.3 Personal observations and notes

Aici mi-am dat seama de dificultatea implementarii intr-un limbaj de programare pe care nu l-am mai folosit pana acum.
Pe layout-ul tinyCorners se expandeaza 435 noduri.
Pe layout-ul mediumCorners se expandeaza 2448 noduri.

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):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
```

```
13      """
14
15      "*** Date Initiale ***"
16      corners = problem.corners # These are the corner coordinates
17      walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
18
19      currentState = state[0]
20      unvisitedCorners = list(set(corners) - set(state[1])) # lista de corner-uri nevizitate
21      h = 0 # euristica
22
23
24      "*** Euristica ***"
25      if currentState not in walls: # daca starea curenta nu e walls
26          while len(unvisitedCorners) != 0:
27
28              " Se calculeaza distanta manhattan dintre starea curenta si un corner din unvisitec
29              corner = unvisitedCorners[0]
30              mini = util.manhattanDistance(currentState, corner) # distanta minima
31
32              " Se calculeaza distanta dintre starea curenta si celelalte stari, se compara cu di
33              " si se actualizeaza daca este necesar "
34              for i in range(1, len(unvisitedCorners)):
35                  dist = util.manhattanDistance(currentState, unvisitedCorners[i])
36                  if dist < mini:
37                    mini = dist
38                    corner = unvisitedCorners[i]
39
40              h += mini # la euristica se adauga distanta minima gasita
41              currentState = corner # starea curenta devine corner-ul cu distanta cea mai mica
42
43              unvisitedCorners.remove(currentState) # se scoate din unvisitedCorners starea curen
44
45      return h
```

Explanation:

* la linia 20 calculam lista de corner-uri nevizitate ca fiind diferenta dintre lista totala de corners ale problemei si lista de corners vizitate ale starii.
* la liniile 25 - 43 este prezentata euristica pentru rezolvarea CornersProblem. Aceasta functioneaza in felul urmator: se gaseste distanta cea mai mica de la pozitia lui Pacman catre un corner, distanta fiind adunata la euristica, dupa care se schimba pozitia lui Pacman cu pozitia celui mai aproape corner si se scoate din unvisitedCorners acel corner. Acest algoritm are loc cat timp exista corners nevizitate.

Commands:

* python pacman.py -l tinyCorners -p AStarCornersAgent
* 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: 2448 (pentru BFS CornersProblem)
Search nodes expanded: 901 (pentru A* cu cornersHeuristic)

2.3.3 Personal observations and notes

Daca calculez in for toate distantele ca sa aflu cea mai mica dintre ele imi da mai multe noduri expandate decat daca calculez prima distanta separat, si apoi restul distantelor in for, ceea ce nu prea am inteles de ce.
Pe layout-ul tinyCorners se expandeaza 217 noduri.
Pe layout-ul mediumCorners se expandeaza 901 noduri.

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:

```
1  def foodHeuristic(state, problem):
2  
3  
4      "*** Date Initiale***"
5      position, foodGrid = state
6      distances = [] # lista care va tine minte
7                     # distantele de la position la mancare
8  
9      "*** Parcurgerea Mancarii ***"
10     " Se parcurge mancarea, se tin minte distantele, iar la final" \
11     " se caluleaza maximul dintre distante, aceasta fiind euristica "
12     for food in foodGrid.asList():
13         distances.append(mazeDistance(position, food, problem.startingGameState))
14     if len(distances) == 0: return 0
15     return max(distances)
```

Explanation:

- la linia 13 am folosit mazeDistance pentru a calcula distantele dintre pozitia lui Pacman si a mancarii
- la linia 14 returnam 0 daca lungimea listei de distance e 0 (nu mai avem mancare, deci ne-am atins goal-ul)

Commands:

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

2.4.3 Personal observations and notes

Am folosit mazeDistance pentru a calcula distanta dintre pozitia lui Pacman si pozitia mancarii (linia 13) deoarece am observat ca expandeaza mai putine noduri decat cu manhattanDistance (la mazeDistance se foloseste de implementarea noastra de BFS pentru a genera solutia optima).
Pe layout-ul testSearch se expandeaza 10 noduri.
Pe layout-ul tinySearch se expandeaza 2372 noduri.
Pe layout-ul trickySearch se expandeaza 4137 noduri.

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):

        # Useful information you can extract from a GameState (pacman.py)
        childGameState = currentGameState.getPacmanNextState(action)
        newPos = childGameState.getPacmanPosition() #pozitia dupa mutare
        newFood = childGameState.getFood() #mancarea ramasa
        newGhostStates = childGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

        newFoodList=newFood.asList()
        min_food_distance =-1
        for food in newFoodList:
             distance=util.manhattanDistance(newPos,food)
             if min_food_distance >= distance or min_food_distance==-1:
                 min_food_distance=distance
        distance_to_ghost=1
        for ghost in newGhostStates:
            distance = util.manhattanDistance(newPos, ghost.getPosition())

            if distance_to_ghost >= distance and distance!=0:
                distance_to_ghost = distance
        return childGameState.getScore() +(1 / float(min_food_distance)) + (1 / float(distance_
```

Explanation:

- Am imbunatatit ReflexAgent altfel incat sa returneze o actiune mai buna.Am mai adaugat la score si distanta cea mai mica dintre pacman si mancare ,precum si dintre pacman si o fantoma .Aceaste distante le-am calculatat cu distanta euclidiana .Ca si distante initiale am luat min-food-distance=-1 deoarece pacman se poate afla oriunde fata de mancare ,si distance-to-ghost=1 deoarece pacman trebuie sa se afle la cel putin la o distanta fata de fantoma; nu poate sa fie 0 ca atunci ar fi pe aceiasi pozitie si l-ar manca.

Commands:

- python3 pacman.py -p ReflexAgent -l testClassic
- python3 pacman.py –frameTime 0 -p ReflexAgent -k 4 -l mediumClassic
- python3 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: Pacman a castigat 8 din cele 10 teste ,si a pierdut de 2 ori .Average score pentru cele 10 teste este 1010.

3.1.3 Personal observations and notes

Am intaminat o singura problema , nu stiam cum sa aflu pozitia fantomei din lista de fantome,dar dupa am aflat ca exista o metoda implementata pentru asa ceva. In cele 10 incercari, pacman nu a mancat acea bucata de mancare care ar fi speriat fantoma.

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:

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):


        # util.raiseNotDefined()
        #functie care verifica daca se termina apelul lui h_minimax:daca pacman pierde/castiga
        #la adancimea pana la care se face evaluarea
        def cut_off_test(state,depth):
             if state.isLose() or state.isWin() or depth == self.depth:
                 return True
             else:
                 return  False


        def h_minimax(agent, depth, state):
            if cut_off_test(state,depth):
                return self.evaluationFunction(state)
```

```
            #daca nu mai sunt actiuni legale ,se termina
            legalActions=state.getLegalActions(agent)
            if not legalActions:
                return  self.evaluationFunction(state)
            #trebuie sa decidem a cui e randul
            if agent == 0:  #  pt pacman,se determina maximul
                return max(h_minimax(1, depth, state.getNextState(agent, action)) for action in
                           legalActions) #se pune 1 la index deoarece se determina max din age
            # e randul pentru fantoma
            else:
                             # a fost ultima fantoma
                if state.getNumAgents()-1 == agent:
                    nextAgent = 0
                    depth += 1  # trebuie sa ia pacman (max) o decizie
                else:
                   nextAgent = agent + 1
                #  se determina minimul
                return min(h_minimax(nextAgent, depth, state.getNextState(agent, action)) for a
                           legalActions)


        # se incepe de la varf(de la pacman)
        v = float("-inf")
        # vedem ce actiuni poate sa faca pacman
        for action in gameState.getLegalActions(0):
            # pentru fiecare actiune posibila a lui pacman calculam h_minimax
            value = h_minimax(1, 0, gameState.getNextState(0, action))
            # daca minimax da o valoare mai mare decat cea data de noi initial,se schima valoa
            # si pentru a ceea valoarea ,actiunea este optima
            if value > v or v == float("-inf"):
                v = value
                OptimalAction = action
        # returnam actiunea optima pe care am gasit-o
        return OptimalAction
```

Explanation:

* Pentru aceasta intrebare a trebuit sa implementez algoritmul Minimax .Am urmat cu exactitate pseudocodul din laborator ,cel pentru h-minimax.Asftel am creat o functie auxiliara care sa verifice daca se termina apelul lui h-minimax.Daca agentul curent este pacman se determina maxim ul,iar daca agentul este o fantoma se determina minimul.Mai multe exeplicati se pot observa in comentarile marcate pe cod.

Commands:

* python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=1
* python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3

3.2.2 Questions

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

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

A1: In aceasta situatie pacman, decide sa mearga spre fantoma din dreapta desi va fi mancat ,deoarece am ales ca si depth=4 ,si isi da seama ca daca o va lua spre stanga va fi mancat de fantoma albastra (deoarece stie ca va fi acolo) ,dar mergand spre dreapta se gandeste ca poate va reusi sa scape de ambele .Mai bine spus ,pacman alege raul cel mai mic in aceasta situatie .

3.2.3 Personal observations and notes

La inceput nu intelesesem ce simbolizeaza depth ,dar am incercat sa analizez testele din proiect ,si am descoperit dupa

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:

```
class AlphaBetaAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):

        # functie care verifica daca se termina apelul lui h_minimax:daca pacman pierde/castig
        # la adancimea pana la care se face evaluarea
        def terminal_test(state ,depth):
            if state.isLose() or state.isWin() or depth == self.depth:
                return True
            else:
                return False
        #pentru pacman
        def max_value (state,agent,depth,alpha,beta):
            if terminal_test(state, depth):
                 return self.evaluationFunction(state)
            v=float("-inf")

            for action in state.getLegalActions(agent):
                v=max(v,min_value(state.getNextState(agent,action),1,depth,alpha,beta)) #se pu
                                                                                        # maxi
                if v> beta :
                    return v
                alpha=max(alpha,v)
            return v

        #pentru fantome
```

```
        def min_value( state,agent,depth,alpha,beta):

            if terminal_test(state, depth):
                return self.evaluationFunction(state)
            v = float("+inf")
            for action in state.getLegalActions(agent):

                if(agent!=state.getNumAgents()-1):
                    #mai exista fantome
                    v=min(v,min_value(state.getNextState(agent,action),agent+1,depth,alpha,beta
                else:
                   #pacman
                   #de fiecare data cand pacman ia o decizie depth creste
                   v = min(v, max_value(state.getNextState(agent, action),self.index ,depth+1,

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

            return v

        alpha=float("-inf")
        beta = float("+inf")
        # vedem ce actiuni poate sa faca pacman
        for action in gameState.getLegalActions(0):
                 #apelam min_value deoarece mergem pe fantome
                 # daca min_value da o valoare mai mare decat cea data de noi initial,se schim
                 # si pentru a ceea valoarea ,actiunea este optima
            v=min_value(gameState.getNextState(0,action),1,0,alpha,beta)
            if(v>alpha or alpha==float("-inf")):
                 alpha=v
                 max_a=action
        return max_a
```

Explanation:

* Acest algortim este o varianta imbunatatita a lui h-minimax, prin urmare nu difera foarte mult ,decat prin prezenta a doi parametri in plus (alfa si beta),alfa simbolizand cea mai mare valoare(cea mai buna valoare din punct de vedere a MAX),iar beta cea mai mica valoare(cea mai buna valoarea din punct de vedere a MIN).Pentru fiecare fantoma noua se compora valoarea data de min-value cu alfa si daca este mai mare se interschimba valoarea lui alpha.Mai multe informatii se gasesc in comentarile de pe cod.

Commands:

* python3 pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
* python autograder.py -q q3

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: Algoritmul trece de testele pe graf,iar pentru testul facut pe lumea pacman va rezulta un Average Score egal cu 84 si va fi mancat de o fantoma. Question q3: 5/5

3.3.3 Personal observations and notes

In implementarea acestui algoritm nu am intampinat dificultati ,deoarece este foarte asemanator cu h-minimax ,dupa cum se poate observa ,ca l-am si implementat foarte asemanator.

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:

"In search.py, implement ***IterativeDeepeningSearch(IDS) algorithm*** *to solve* ***SearchProblem*** *from searchAgents.py".*

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:

```
def recursiveDLS(node, problem, limit, solution, exploredList):


    if problem.isGoalState(node): # cand s-a gasit nodul de goal returneaza solutia
        return solution
    else:
        if limit == 0: # daca limit e 0 iesi din recursivitate
            return False
        else:

            cutoffOccurred = False # nu se ajunge la un capat unde trebuie sa ne intoarcem

            successors = problem.expand(node) # aflam succesrii nodului
            for i in range(0, len(successors)):
                child = (problem, successors[i][0], successors[i][1])

                if child[1] not in exploredList and child[1] != node: # daca nu e radacina si 

                    # prin apelul recursiv se construieste lista de solutii si de noduri explo
                    result = recursiveDLS(child[1], problem, limit - 1, solution + [child[2]],

                    if result == False: # daca nu a gasit solutia
                        cutoffOccurred = True # se face cutoff
                    else:
                        if result != False: # a gasit solutia
                            return result
            if cutoffOccurred: # se iese din functie
                return False

            return False

def depthLimitedSearch(problem, limit):
    return recursiveDLS(problem.getStartState(), problem, limit, [], [])

def iterativeDeepeningSearch(problem):
```

```
36
37      " Se face o cautare in adancime pana la adancimea depth "
38      " Daca nu gaseste nodul de goal, creste adancimea "
39      depth = 0
40      while True :
41          result = depthLimitedSearch(problem, depth)
42          if result != False:
43              return result
44          depth += 1
```

Explanation:

- Am ales sa implementam iterativeDeepeningSearch, fiind un algoritm de cautare care genereaza solutia optima cu o abordare de tip DFS
- la linia 20, in loc de a folosi o stiva data de noi, am folosit stiva de la apelul functiei, rezolvarea fiind recursiva

Commands:

- python pacman.py -l tinyMaze -p SearchAgent -a fn=ids
- python pacman.py -l smallMaze -p SearchAgent -a fn=ids
- python pacman.py -l mediumMaze -z .5 -p SearchAgent -a fn=ids
- python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=ids

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

Am urmat algoritmul de la curs.
Pe layout-ul tinyMaze se expandeaza 94 de noduri.
Pe layout-ul smallMaze se expandeaza 2295 noduri.
Pe layout-ul mediumMaze se expandeaza 29857 noduri.
Pe layout-ul bigMaze se expandeaza 119140 noduri.

Algoritmul de cautare este:

- complet daca spatiul starilor este finit
- optim daca costul este constant intre noduri
- are complexitatea timpului exponentiala
- are complexitatea spatiului liniara

4.2 References