Artificial intelligence - Project 1
- Search problems -

Moldovan Adelina, Vorniceanu Iuliana

4/11/2020

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:

```
1  class Node:
2      def __init__(self, state, parent, action, path_cost):
3          self.state = state
4          self.parent = parent
5          self.action = action
6          self.path_cost = path_cost
7
8      def getState(self):
9          return self.state
10
11     def getParent(self):
12         return self.parent
13
14     def getAction(self):
15         return self.action
16
17     def getPathCost(self):
18         return self.path_cost
19
20     def setPathCost(self, cost):
21         self.path_cost = cost
22
23
24 def depthFirstSearch(problem):
25     node = Node(problem.getStartState(), None, None, 0)
26
27     frontier = util.Stack()
28     frontier.push(node)
29     explored = []
30
31     while not frontier.isEmpty():
32         node = frontier.pop()
33         if problem.isGoalState(node.getState()):
34             solution = []
35             while node.getParent() is not None:
36                 solution.append(node.getAction())
```

```
37                 node = node.getParent()
38             solution.reverse()
39             return solution
40 
41         for successor in problem.getSuccessors(node.getState()):
42             if successor[0] not in explored:
43                 child = Node(successor[0], node, successor[1], successor[2])
44                 frontier.push(child)
45         explored.append(node.getState())
```

Explanation:

- Liniile 1-21 : am definit o clasa Node cu ajutorul careia am stocat informatii utile pentru nodurile expandate de algoritm
- Linia 25 : am creat un obiect de tipul Node pentru nodul de start
- Linia 29 : am declarat o lista in care vom stoca nodurile explorate
- Liniile 31-32: cat timp avem noduri in frontiera, scoatem cate unul in scopul de a-l expanda
- Liniile 33-39 : daca am ajuns la GoalState, construim solutia folosind atributele Parent si Action din clasa Node, dupa care o returnam
- Liniile 41-44: am generat succesorii nodului curent, pe care i-am adaugat in frontiera daca nu au fost deja explorati
- Linia 45: dupa ce am vizitat succesorii unui nod, adaugam nodul respectiv in vectorul explored

Commands:

- -l tinyMaze -p SearchAgent -a fn=dfs
- -l mediumyMaze -p SearchAgent -a fn=dfs
- -l bigMaze -z .5 -p SearchAgent -a fn=dfs

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: Algoritmul de căutare DFS nu este optim, deoarece poate expanda un numar mare de noduri sau poate genera un cost ridicat pentru a ajunge la solutie.
Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: 3/3

1.1.3 Personal observations and notes

1.2 Question 2 - Breadth-first search

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

*"In **search.py**, implement the **Breadth-First search** algorithm in function breadthFirstSearch.".*

1.2.1 Code implementation

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

Code:

```
1  def breadthFirstSearch(problem):
2      node = Node(problem.getStartState(), None, None, 0)
3      frontier = util.Queue()
4      frontier.push(node)
5      explored = []
6      nodesInQueue = [node.getState()]
7      while not frontier.isEmpty():
8          node = frontier.pop()
9          if problem.isGoalState(node.getState()):
10             solution = []
11             while node.getParent() is not None:
12                 solution.append(node.getAction())
13                 node = node.getParent()
14             solution.reverse()
15             return solution
16         for successor in problem.getSuccessors(node.getState()):
17             child = Node(successor[0], node, successor[1], successor[2])
18             if child.getState() not in explored and child.getState() not in nodesInQueue:
19                 frontier.push(child)
20                 nodesInQueue.append(child.getState())
21         explored.append(node.getState())
```

Explanation:

- Ne folosim de structura de Node folosita si definita mai sus la DFS.
- Linia 2 : am creat un obiect de tipul Node pentru nodul de start
- Linia 5 : am declarat o lista in care vom stoca nodurile explorate
- Linia 6 : am creat o lista in care vom salva nodurile care au fost adaugate in coada
- Liniile 7-8 : cat timp avem noduri in frontiera, scoatem cate unul in scopul de a-l expanda
- Liniile 9-15 : daca am ajuns la GoalState, construim solutia folosind atributele Parent si Action din clasa Node, dupa care o returnam
- Liniile 16-20 : am generat succesorii nodului curent, pe care i-am adaugat in frontiera daca nu au fost deja explorati sau adaugati in frontiera ca succesori pentru alt nod
- Linia 21: dupa ce am vizitat succesorii unui nod, adaugam nodul respectiv in vectorul explored

Commands:

- -l mediumMaze -p SearchAgent -a fn=bfs

1.2.2 Questions

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

Q1: Is the found solution optimal? Explain your answer.

A1: Algoritmul BFS este optim deoarece incepe sa caute de la nodul radacina si expandeaza toti succesorii nodului curent inainte sa treaca la nodul urmator astfel va genera solutia cu cel mai mic numar de pasi.

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

A2: 3/3

1.2.3 Personal observations and notes

1.3 Question 3 - Uniform-cost search

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

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

1.3.1 Code implementation

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

Code:

```
def uniformCostSearch(problem):
    node = Node(problem.getStartState(), None, None, 0)
    frontier = util.PriorityQueue()
    frontier.push(node, 0)
    explored = []
    nodesInQueue = [node.getState()]

    cost_so_far = dict()
    cost_so_far[node.getState()] = 0

    while not frontier.isEmpty():
        node = frontier.pop()
        if problem.isGoalState(node.getState()):
            solution = []
            while node.getParent() is not None:
                solution.append(node.getAction())
                node = node.getParent()
            solution.reverse()
            return solution
        for successor in problem.getSuccessors(node.getState()):
            child = Node(successor[0], node, successor[1], successor[2])
            new_cost = cost_so_far[node.getState()] + successor[2]
            if child.getState() not in explored and child.getState() not in nodesInQueue:
                frontier.push(child, new_cost)
                cost_so_far[child.getState()] = new_cost
                nodesInQueue.append(child.getState())
            if child.getState() in cost_so_far and new_cost < cost_so_far[child.getState()]:
                cost_so_far[child.getState()] = new_cost
                frontier.update(child, new_cost)
        explored.append(node.getState())
```

Explanation:

- Ne folosim de structura de Node folosita si definita mai sus la DFS.
- Linia 2 : am creat un obiect de tipul Node pentru nodul de start
- Linia 5 : am declarat o lista in care vom stoca nodurile explorate
- Linia 6 : am creat o lista in care vom salva nodurile care au fost adaugate in coada
- Liniile 8-9 : am creat un dictionar in care cheia este reprezentata de state-ul unui nod, iar valorea este data de costul solutiei de la nodul de start pana la nodul dat ca cheie
- Liniile 11-12 : cat timp avem noduri in frontiera, scoatem cate unul in scopul de a-l expanda
- Liniile 13-19 : daca am ajuns la GoalState, construim solutia folosind atributele Parent si Action din clasa Node, dupa care returnam solutia
- Liniile 20-26 : am generat succesorii nodului curent, pe care i-am adaugat in frontiera, in dictionar si in nodesInQueue, daca nu au fost deja explorati sau adaugati in frontiera ca succesori pentru alt nod
- Linia 22 : calculam costul drumului pentru succesorii nodului curent
- Liniile 27-29 : daca un succesor a fost deja adaugat in dictionar, verificam daca ii putem actualiza costul drumului cu o valoare mai mica, in caz afirmativ actualizam si valoarea din dictionar si cea din frontiera
- Linia 30: dupa ce am vizitat succesorii unui nod, adaugam nodul respectiv in vectorul explored

Commands:

- -l mediumMaze -p SearchAgent -a fn=ucs

1.3.2 Questions

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

Q1: Compare the results to the ones obtained with DFS. Are the solutions different? Is the number of extended (explored) states smaller? Explain your answer.
A1: Solutiile sunt diferite, UCS expandeaza mai multe noduri dar cu un cost mai redus fata de DFS care expandeaza mai putine noduri, dar cu un cost mai mare. Scopul UCS-ului este de a gasi cel mai scurt drum pana la solutie si nu tine cont de numarul de pasi, spre deosebire de DFS care intotdeauna expandeaza cel mai adanc nod din frontiera.
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: 3/3

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:

```
1  def aStarSearch(problem, heuristic=nullHeuristic):
2      node = Node(problem.getStartState(), None, None, 0)
3      frontier = util.PriorityQueue()
4      frontier.push(node, 0)
5      explored = []
6      nodsInQueue = [node.getState()]
7
8      cost_so_far = dict()
9      cost_so_far[node.getState()] = 0
10
11     while not problem.isGoalState(node.getState()):
12         node = frontier.pop()
13         if problem.isGoalState(node.getState()):
14             solution = []
15             while node.getParent() is not None:
16                 solution.append(node.getAction())
17                 node = node.getParent()
18             solution.reverse()
19             return solution
20
21         for successor in problem.getSuccessors(node.getState()):
22             child = Node(successor[0], node, successor[1], successor[2])
23             new_cost = cost_so_far[node.getState()] + successor[2]
24             if child.getState() not in explored and child.getState() not in nodsInQueue:
25                 frontier.push(child, new_cost + heuristic(child.getState(), problem))
26                 cost_so_far[child.getState()] = new_cost
27                 nodsInQueue.append(child.getState())
28             if child.getState() in cost_so_far and new_cost < cost_so_far[child.getState()]:
29                 cost_so_far[child.getState()] = new_cost
30                 frontier.update(child, new_cost)
31         explored.append(node.getState())
```

Listing 1: Solution for the A* algorithm.

Explanation:

- Ne folosim de structura de Node folosita si definita mai sus la DFS.
- Linia 2 : am creat un obiect de tipul Node pentru nodul de start
- Linia 5 : am declarat o lista in care vom stoca nodurile explorate
- Linia 6 : am creat o lista in care vom salva nodurile care au fost adaugate in coada
- Liniile 8-9 : am creat un dictionar in care cheia este reprezentata de state-ul unui nod, iar valorea este data de costul solutiei de la nodul de start pana la nodul dat ca cheie
- Liniile 11-12 : cat timp avem noduri in frontiera, scoatem cate unul in scopul de a-l expanda
- Liniile 13-19 : daca am ajuns la GoalState, construim solutia folosind atributele Parent si Action din clasa Node, dupa care returnam solutia
- Liniile 21-27 : am generat succesorii nodului curent, pe care i-am adaugat in frontiera, in dictionar si in nodesInQueue, daca nu au fost deja explorati sau adaugati in frontiera ca succesori pentru alt nod
- Linia 23 : calculam costul drumului pentru succesorii nodului curent
- Linia 25 : cand dam push in coada unui nod, path costul acestuia va fi suma dintre costul de la nodul de start pana la nodul curent si distanta de la nodul curent la goalState, aproximata cu ajutorul unei euristici
- Liniile 28-30 : daca un succesor a fost deja adaugat in dictionar, verificam daca ii putem actualiza costul drumului cu o valoare mai mica, in caz afirmativ actualizam si valoarea din dictionar si cea din frontiera
- Linia 31: dupa ce am vizitat succesorii unui nod, adaugam nodul respectiv in vectorul explored

Commands:

- -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
- -l mediumMaze -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: A* si UCS gasesc aceiasi solutie, diferenta este data de faptul ca A* expandeaza mai putine noduri decat UCS.
Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2: Da, A* gaseste solutia expandand mai putine noduri decat UCS, deoarece A* estimeaza costul drumului de la nodul curent pana la solutie si le expandaeza pe cele cu costul mai mic.
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: 3/3

Am realizat o comparatie intre algoritmi pe baza costului drumului si numarului de noduri expandate

	DFS	BFS	UCS	ASTAR
tinyMaze	10,15	8,15	8,15	8,14
mediumMaze	130,146	69,269	68,269	68,222
bigMaze	210,390	210,620	210,620	210,549

Concluzii:

-DFS gaseste un drum cu cost foarte mare, dar expandeaza un numar mic de noduri

-BFS gaseste un drum scurt, dar expandeaza un numar mare de noduri

-UCS gaseste drumul cu cel mai mic cost

-UCS si BFS au aceleasi rezultate in ceea ce priveste numarul de noduri expandate deoarece costul de la un nod la orice succesor al sau are aceiasi valoare pentru toate nodurile din graf

-A* se comporta ca un UCS, dar foloseste o euristica pentru a aproxima costul drumului pana la goalState, prioritatea find data de lungimea drumului de la nodul de start pana la nodul curent(la fel ca la UCS) plus valoarea returnata de euristica

-daca vrem sa gasim solutia cat mai repede folosim DFS

-daca vrem sa gasim solutia cu cel mai scurt drum fara sa tinem cont de costul muchiilor folosim BFS

-daca vrem sa gasim o solutie cu cel mai scurt drum si vrem sa tinem cont de costul muchiilor folosim UCS

-daca vrem sa gasim o solutie cu cel mai scurt drum, tinand cont de costul muchiilor dar vrem sa fie si un timp mai scurt, folosim A*

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 CornerState:
    def __init__(self, coord, visitedCorners):
        self.coord = coord
        self.visitedCorners = visitedCorners

    def __eq__(self, other):
        return self.getCoord() == other.getCoord() and self.getVisitedCorners() == other.getVisitedCorne

    def __hash__(self):
        return (str(self.coord) + str(self.visitedCorners)).__hash__()

    def getCoord(self):
        return self.coord

    def getVisitedCorners(self):
        return self.visitedCorners

class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState):

        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

    def getStartState(self):
        return CornerState(self.startingPosition, [])

    def isGoalState(self, state):
        if len(state.getVisitedCorners()) == 4:
            return True
        return False

    def getSuccessors(self, state):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            x, y = state.getCoord()
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                nextState = (nextx, nexty)
                if nextState in self.corners and nextState not in state.getVisitedCorners():
                    successors.append((CornerState(nextState, state.getVisitedCorners() + [nextState]),
                else:
                    successors.append((CornerState(nextState, state.getVisitedCorners()), action, 1))
        self._expanded += 1
        return successors

```

```
55      def getCostOfActions(self, actions):
56          if actions == None: return 999999
57          x, y = self.startingPosition
58          for action in actions:
59              dx, dy = Actions.directionToVector(action)
60              x, y = int(x + dx), int(y + dy)
61              if self.walls[x][y]: return 999999
62          return len(actions)
```

Explanation:

- Liniile 1-16 : am creat clasa CornerState care are doua atribute, coord care reprezinta coordonatele unui punct si visitedCorners, o lista care va contine nodurile vizitate
- Liniile 31-32 : functia getStartState() va returna un obiect de tipul CornerState, unde coord va avea valoarea punctului de plecare si visitedCorners va primi o lista goala
- Liniile 34-37 : functia isGoalState() care va returna TRUE daca in lista visitedCorners se vor afla 4 obiecte, si anume cele 4 colturi
- Liniile 39-53 : functia getSuccessors() care returneaza lista succesorilor
- Liniile 41-44 : parcurgem lista urmatoarelor stari posibile si calculam coordonatele acestora
- Liniile 45-46 : verificam daca starea urmatoare nu este un perete, in caz afirmativ trecem la pasul urmator
- Liniile 48-51 : daca starea urmatoare este un corner si nu se afla in lista de visitedCorners, la lista de succesori adaugam aceasta stare, dar introducem si acest corner in lista de visitedCorners, in caz contrar doar o adaugam in lista de succesori

Commands:

- -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- -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: 1360

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:

```
1  def cornersHeuristic(state, problem):
2      visited = state.getVisitedCorners()
3      node = state.getCoord()
4      heuristic = 0
5      corners = problem.corners
6      unvisited = []
7      if problem.isGoalState(state):
8          return 0
9      else:
10         for corner in corners:
11             if corner not in visited:
12                 unvisited.append(corner)
13         for corner in unvisited:
14             distance = abs(corner[0] - node[0]) + abs(corner[1] - node[1])
15             heuristic = max(distance, heuristic)
16         return heuristic
```

Explanation:

- Liniile 7-8 : daca am ajuns la goalState returnam 0
- Liniile 10-12 : parcurgem lista de noduri vizitate pentru a obtine nodurile nevizitate
- Liniile 13-15 : parcurgem nodurile nevizitate si calculam distanta Manhattan de la nodul curent la toate aceste colturi, returnand maximul dintre aceste distance, adica distanta pana la coltul cel mai apropiat

Commands:

- -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
- -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: 1360

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:

```
1  def foodHeuristic(state, problem):
2      position, foodGrid = state
3      foodGrid = foodGrid.asList()
4      if problem.isGoalState(state):
5          return 0
6      else:
7          manhattan = 0
8          for food in foodGrid:
9              distance = abs(position[0] - food[0]) + abs(position[1] - food[1])
10             manhattan = max(manhattan, distance)
11         return manhattan
```

Explanation:

- Liniile 4-5 : daca am ajuns la goalState returnam 0
- Liniile 8-11 : parcurgem foodDoturile ramase si calculam distanta Manhattan de la nodul curent la toate foodDoturile, returnand maximul dintre aceste distance, adica distanta pana la foodDotul cel mai apropiat

Commands:

- -l testSearch -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: 9928

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:

```
1  class ReflexAgent(Agent):
2
3      def getAction(self, gameState):
4          legalMoves = gameState.getLegalActions()
5
6          scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
7          bestScore = max(scores)
8          bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
9          chosenIndex = random.choice(bestIndices)  # Pick randomly among the best
10         return legalMoves[chosenIndex]
11
12     def findNext(self, pacman, matrix):
13         dist = 10000000000
14         for i, matrix_i in enumerate(matrix):
15             for j, value in enumerate(matrix_i):
16                 if value:
17                     aux = abs(pacman[0] - i) + abs(pacman[1] - j)
18                     if aux < dist:
19                         dist = aux
20         return dist
21
22     def evaluationFunction(self, currentGameState, action):
23         successorGameState = currentGameState.generatePacmanSuccessor(action)
24         newPos = successorGameState.getPacmanPosition()
25         newFood = successorGameState.getFood()
26         newGhostStates = successorGameState.getGhostStates()
27         ghostPosition = newGhostStates[0].getPosition()
28
29         ghostPosiblePositions = []
30         ghostPosiblePositions.append((ghostPosition[0], ghostPosition[1]))
31         ghostPosiblePositions.append((ghostPosition[0], ghostPosition[1] + 1))
32         ghostPosiblePositions.append((ghostPosition[0], ghostPosition[1] - 1))
33         ghostPosiblePositions.append((ghostPosition[0] + 1, ghostPosition[1]))
34         ghostPosiblePositions.append((ghostPosition[0] - 1, ghostPosition[1]))
35
36         pacmanPos = currentGameState.getPacmanPosition()
```

```
37          if newPos in ghostPosiblePositions:
38              return -99999
39          initialDistance = self.findNext(pacmanPos, newFood)
40          newDistance = self.findNext(newPos, newFood)
41          if initialDistance > newDistance:
42              return successorGameState.getScore()
43          return successorGameState.getScore() - 1
```

Explanation:

- Linia 4: gasim actiunile posibile din pozitia actuala.
- Liniile 6-7: calculam punctajele pe care le putem obtine pentru fiecare actiune iar dupa calculam maximul dintre acestea.
- Liiile 8-9: calculam pe ce indice este punctajul maxim si daca acesta este obtinut de mai multe actiuni alegem un indice random.
- Linia 10: returnam din vectorul de actiuni calculat la linia 4 valoarea aflata pe indicele calculat anterior.
- Linia 12: declararea functiei findNext care returneaza cea mai mica distanta dintre o pozitie si cel mai apropiat foodDot. Functia primeste ca parmetru o pozitie si o matrice in care se afla valoarea true pe pozitiile de foodDot.
- Liniile 14-19: parcurgem matricea de valori de true si false si calculam cu ajutorul distantei manhattan care este cea mai mica distanta dintre pozitia primita ca si parametru si foodDot-uri
- Linia 20: returnam valoarea distantei obtinute.
- Linia 22: declararea functiei care calculeaza punctajele. Aceasta primeste ca parametru o stare a jocului si o actiune.
- Liniile 23-24: calculam noua pozitie obtinuta cu ajutorul actiunii aplicata lui pacman.
- Linia 25: obtinerea matricei pentru pozitiile foodDot-urilor.
- Liniile 26-27: obtinerea pozitiei fantomei.
- Liniile 29-34: crearea unei liste cu pozitiile posibile ale fantomei.
- Linia 36: obtinerea pozitiei initiale a lui pacman.
- Liniile 37-38: verificam daca noua pozitie a lui pacman este una din posibilele pozitii ale fantomei si returnam o valoare foarte mica astfel incat sa semnalam ca aceasta mutare nu este una buna.
- Liniile 39-40: calculam cu ajutorul functiei findNext distanta de la pacman si de la noua pozitie a lui pacman pana la cel mai apropiat foodDot.
- Liniile 41-42: daca distanta de la pacman pana la foodDot este mai mare decat distanta de la noua pozitie a lui pacman pana la mancare atunci returnam valoarea functiei getScore.
- Linia 43: semnalam ca aceasta actiune nu este cea mai buna pentru pacman deoarece daca este aplicata atunci pacman nu se apropie de mancare.

Commands:

- -p ReflexAgent -l testClassic
- – frameTime 0 -p ReflexAgent -k 1 -l mediumClassic

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 la toate cele 10 jocuri. Average score este 1239.2

4/4 points

3.1.3 Personal observations and notes

3.2 Question 9 - H-Minimax algorithm

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

" Implement H-Minimax algorithm in MinimaxAgentclass from multiAgents.py. Since it can be more than one ghost, for each max layer there are one ormore min layers.".

3.2.1 Code implementation

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

Code:

```
1  class MinimaxAgent(MultiAgentSearchAgent):
2
3      def getAction(self, gameState):
4          return self.max_value(0, gameState)
5
6      def max_value(self, currentDepth, gameState):
7          if self.depth == currentDepth or gameState.isLose() or gameState.isWin():
8              return self.evaluationFunction(gameState)
9          val = float('-inf')
10         actionF = ""
11         for action in gameState.getLegalActions():
12             suc = gameState.generateSuccessor(0, action)
13             mini = self.min_value(currentDepth, suc, 1)
14             if mini > val:
15                 val = mini
16                 actionF = action
17         return actionF if currentDepth == 0 else val
18
19     def min_value(self, currentDepth, gameState, ghostIndex):
20         if self.depth == currentDepth or gameState.isLose() or gameState.isWin():
21             return self.evaluationFunction(gameState)
22         val = float('inf')
23         for action in gameState.getLegalActions(ghostIndex):
24             suc = gameState.generateSuccessor(ghostIndex, action)
25             if ghostIndex == gameState.getNumAgents() - 1:
26                 max_value = self.max_value(currentDepth + 1, suc)
27             else:
28                 max_value = self.min_value(currentDepth, suc, ghostIndex + 1)
29             val = min(val, max_value)
30         return val
```

Explanation:

- Linia 4: apelarea functiei de max-alue.
- Linia 6: definirea functiei de max-value

- Liniile 7-8: verificam daca suntem intr-un caz de stop si returnam valoarea functiei evaluationFunction.
- Liniile 9-10: instantiem o variabila cu valoarea -infinit si o actiuneAuxiliara cu nimic.
- Liniile 11-13: pentru fiecare actiune pe care o putem face calculam succesorul si apelam functia min-value pentru acel succesor, adancimea la care am ajuns si 1(valoarea din vectorul de agenti pentru prima fantoma)
- Liniile 14-16: comparam variabila instantiata cu infinit cu valoarea returnata de functia min-value si daca cea dea doua este mai mare atunci ii atribuim variabilei valoarea functiei iar actiuneiAuxiliare ii este atribuita valoarea actiunii pentru care am obtinut valoare maxime
- Linia 17:dupa ce am parcurs toti succesorii daca suntem in cazul in care adancimea este 0 atunci returnam actiuneaAuxiliara iar daca nu atunci returnam valoarea variabilei
- Linia 19: definirea functiei de min-value
- Liniile 20-21: verificam daca suntem intr-un caz de stop si returnam valoarea functiei evaluationFunction.
- Linia 22: instantiem o variabila cu valoarea infinit.
- Liniile 23-24: pentru fiecare actiune pe care o putem face calculam succesorul si apelam functia min-value pentru acel succesor, adancimea la care am ajuns si 1(valoarea din vectorul de agenti pentru prima fantoma).
- Liniile 25-28: verificam daca suntem in cazul in care parametru functiei-ghostIndex este egal cu indexul ultimei fantome; in caz afirmativ atunci se apeleaza functia max-value cu parametrii depth+1 si succesor iar in caz negativ se apeleaza functia min-value cu ghostIndex+1.
- Linia 29: variabilei instantieate ii atribuim minimul dintre ea si valoarea functiei apelate mai sus.
- Linia 30: dupa ce am parcurs toti succesorii returnam valoarea variabilei.

Commands:

- -p MinimaxAgent -l minimaxClassic -a depth =4

3.2.2 Questions

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

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

A1: Algoritmul pentru minimax nu are rolul de a gasi soutia prin care pacman sa castige jocul mereu ci are ca scop obtinerea celui mai bun punctaj posibil. In cazul in care se observa ca pacman nu are cale de scapare acesta incearca sa ajunga la cea mai apopiata fantoma astfel incat sa moara cat mai repede pentru a nu pierde prea multe puncte.

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:

```
1  class AlphaBetaAgent(MultiAgentSearchAgent):
2
3      def getAction(self, gameState):
4          return self.max_value(0, gameState,float('-inf'),float('inf'))
5
6      def max_value(self, currentDepth, gameState,alpha,beta):
7          if self.depth == currentDepth or gameState.isLose() or gameState.isWin():
8              return self.evaluationFunction(gameState)
9          val = float('-inf')
10         actionF = ""
11         for action in gameState.getLegalActions():
12             suc = gameState.generateSuccessor(0, action)
13             mini = self.min_value(currentDepth, suc, 1,alpha,beta)
14             if mini > val:
15                 val = mini
16                 actionF = action
17             alpha = max(alpha , mini)
18             if beta < alpha:
19                 break
20         return actionF if currentDepth == 0 else val
21
22     def min_value(self, currentDepth, gameState, ghostIndex,alpha,beta):
23         if self.depth == currentDepth or gameState.isLose() or gameState.isWin():
24             return self.evaluationFunction(gameState)
25         val = float('inf')
26         for action in gameState.getLegalActions(ghostIndex):
27             suc = gameState.generateSuccessor(ghostIndex, action)
28             if ghostIndex == gameState.getNumAgents() - 1:
29                 max_value = self.max_value(currentDepth + 1, suc,alpha,beta)
30             else:
31                 max_value = self.min_value(currentDepth, suc, ghostIndex + 1,alpha,beta)
32             val = min(val, max_value)
33             beta=min(beta , max_value)
34             if beta < alpha:
35                 break
36         return val
```

Explanation:

- Linia 4: apelarea functiei de max-alue.
- Linia 6: definirea functiei de max-value
- Liniile 7-8: verificam daca suntem intr-un caz de stop si returnam valoarea functiei evaluationFunction.
- Liniile 9-10: instantiem o variabila cu valoarea -infinit si o actiuneAuxiliara cu nimic.

- Liniile 11-13: pentru fiecare actiune pe care o putem face calculam succesorul si apelam functia min-value pentru acel succesor, adancimea la care am ajuns si 1(valoarea din vectorul de agenti pentru prima fantoma)
- Liniile 14-16: comparam variabila instantiata cu infinit cu valoarea returnata de functia min-value si daca cea dea doua este mai mare atunci ii atribuim variabilei valoarea functiei iar actiuneiAuxiliare ii este atribuita valoarea actiunii pentru care am obtinut valoare maxime
- Linia 17: parametrului alpha ii se atribuie maximul dintre alpha si valoarea calculata mai sus
- Linile 18-19: verificam daca beta este mai mic decat alpha iar in caz pozitiv atunci iesim fortat din for-ul pentru actiuni
- Linia 20:dupa ce am parcurs toti succesorii daca suntem in cazul in care adancimea este 0 atunci returnam actiuneaAuxiliara iar daca nu atunci returnam valoarea variabilei
- Linia 22: definirea functiei de min-value
- Liniile 23-24: verificam daca suntem intr-un caz de stop si returnam valoarea functiei evaluationFunction.
- Linia 25: instantiem o variabila cu valoarea infinit.
- Liniile 26-27: pentru fiecare actiune pe care o putem face calculam succesorul si apelam functia min-value pentru acel succesor, adancimea la care am ajuns si 1(valoarea din vectorul de agenti pentru prima fantoma).
- Liniile 28-31: verificam daca suntem in cazul in care parametru functiei-ghostIndex este egal cu indexul ultimei fantome; in caz afirmativ atunci se apeleaza functia max-value cu parametrii depth+1 si succesor iar in caz negativ se apeleaza functia min-value cu ghostIndex+1.
- Linia 32: variabilei instantieate ii atribuim minimul dintre ea si valoarea functiei apelate mai sus.
- Linia 33: lui beta ii se atribuie minimul dintre beta si valoarea calculata mai sus
- Liniile 34-35:verificam daca beta este mai mic decat alpha iar in caz pozitiv atunci iesim fortat din for-ul pentru actiuni
- Linia 30: dupa ce am parcurs toti succesorii returnam valoarea variabilei.

Commands:

- -p AlphaBetaAgent -a depth =3 -l smallClassic

3.3.2 Questions

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

Q1: Test your implementation with autograder **python autograder.py** for Question 3. What are your results?
A1: 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