Artificial intelligence - Project 1
- Search problems -

Baleanu Sorina-Diana
Ionescu Raluca-Ionela

3/11/2020

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  def depthFirstSearch(problem):
2
3   starting_node = problem.getStartState()
4
5      if problem.isGoalState(starting_node):
6          return []
7
8      stack = util.Stack()
9      visited = []
10
11     node = GraphNode(starting_node, None, [], 0)
12     stack.push(node)
13
14     while not stack.isEmpty():
15         current = stack.pop()
16
17         if current.getState() not in visited:
18             visited.append(current.getState())
19
20             if problem.isGoalState(current.getState()):
21                 return current.getAction()
22
23             for next_node, action, cost in problem.getSuccessors(current.getState()):
24                 next_action = current.getAction() + [action]
25                 next_cost = current.getCost() + cost
26                 new_node = GraphNode(next_node, current.getState(), next_action, next_cost)
27                 stack.push(new_node)
28
29     util.raiseNotDefined()
```

Explanation:

- DFS este un algoritm ce se foloseste pentru traversarea unui graf. Se incepe cu alegerea unui nod de start, care in cazul nostru va fi problem.getStartState().In cazul in care nodul de inceput este nodul destinatie, vom returna o lista vida. In caz contrar, initializam o stiva si cream o lista pentru nodurile vizitate. In stiva vom adauga elemente de tipul GraphNode, o structura auxiliara care retine starea curenta, parintele, lista de actiuni si costul. Pentru inceput adaugam nodul initial care nu are parinte, o lista de actiuni nula si costul 0.

Cat timp exista elemente in stiva vom extrage ultimul nod din aceasta, se verifica daca starea curenta se afla in lista de noduri vizitate, iar in cazul in care nu se afla se va adauga in aceasta lista, dupa care se verifica daca starea curenta este destinatia. In cazul in care este goalState, algoritmul o sa returneze lista de actiuni ale nodului curent. In cazul contrar, se parcurg succesorii nodului curent si se construieste noua lista de actiuni formata din actiunile nodului curent si actiunile succesorilor, se calculeaza noul cost (suma costului curent + costul succesorilor) si se creeaza un nou element de tipul GraphNode care se adauga in stiva

Commands:

- python pacman . py -l tinyMaze -p SearchAgent
  python pacman . py -l mediumMaze -p SearchAgent
  python pacman . py -l bigMaze -z .5 -p SearchAgent

1.1.2 Questions

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

Q1: Is the found solution optimal? Explain your answer.
A1: DFS este un algoritm ce evita expandarea nodurilor deja vizitate, insa este unul neinformat. Deoarece exploreaza in adancime, prima solutie gasita nu este de cele mai multe ori si cea mai optima, caci nu ia in calcul costul explorarii altor cai. In concluzie nu este un algorim optim.

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
3       starting_node = problem.getStartState()
4
5      if problem.isGoalState(starting_node):
6          return []
7
8      queue = util.Queue()
9      visited = []
```

```
10
11      node = GraphNode(starting_node, None, [], 0)
12      queue.push(node)
13
14      while not queue.isEmpty():
15
16          current = queue.pop()
17
18          if current.getState() not in visited:
19              visited.append(current.getState())
20
21              if problem.isGoalState(current.getState()):
22                  return current.getAction()
23
24              for next_node, action, cost in problem.getSuccessors(current.getState()):
25                  next_action = current.getAction() + [action]
26                  next_cost = current.getCost() + cost
27                  new_node = GraphNode(next_node, current.getState(), next_action, next_cost)
28                  queue.push(new_node)
29
30      util.raiseNotDefined()
```

Explanation:

- BFS este un algoritm ce se foloseste pentru traversarea unui graf. Se incepe cu alegerea unui nod de start, care in cazul nostru va fi problem.getStartState(). In cazul in care nodul de inceput este nodul destinatie, vom returna o lista vida. In caz contrar initializam o coada si cream o lista pentru nodurile vizitate. In coada vom adauga elemente de tipul GraphNode, o structura auxiliara care retine starea curenta, parintele, lista de actiuni si costul. Pentru inceput adaugam nodul initial care nu are parinte, o lista de actiuni nula si costul 0.
  Cat timp exista elemente in coada vom extrage primul nod din aceasta, se verifica daca starea curenta se afla in lista de noduri vizitate, iar in cazul in care nu se afla se va adauga in aceasta lista si se verifica daca starea curenta este destinatia. In cazul in care este goalState algoritmul o sa returneze lista de actiuni ale nodului curent. In cazul contrar, se parcurg succesorii nodului curent si se construieste noua lista de actiuni formata din actiunile nodului curent si actiunile succesorilor, se calculeaza noul cost (suma costului curent + costul succesorilor) si se creeaza un nou element de tipul GraphNode care se adauga in coada.

Commands:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
  python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

1.2.2 Questions

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

Q1: Is the found solution optimal? Explain your answer.
A1: Cautarea in latime este optima atunci cand costul caii nu scade odata cu cresterea adancimii nodurilor. In aceasta situatie algoritmul BFS este un algoritm optim, deoarece toate costurile actiunilor sunt egale.
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:

```
1  def uniformCostSearch(problem):
2
3       starting_node = problem.getStartState()
4
5      if problem.isGoalState(starting_node):
6          return []
7
8      visited = []
9      queue = util.PriorityQueue()
10
11     node = GraphNode(starting_node, None, [], 0)
12     queue.push(node, 0)
13
14     while not queue.isEmpty():
15
16         current = queue.pop()
17
18         if current.getState() not in visited:
19             visited.append(current.getState())
20
21             if problem.isGoalState(current.getState()):
22                 return current.getAction()
23
24             for next_node, action, cost in problem.getSuccessors(current.getState()):
25                 next_action = current.getAction() + [action]
26                 priority = current.getCost() + cost
27                 new_node = GraphNode(next_node, current, next_action, priority)
28                 queue.push(new_node, priority)
29
30     util.raiseNotDefined()
```

Explanation:

- Uniform-cost Search este un algoritm folosit pentru traversarea unui graf ponderat si pentru gasirea unui drum optim cu cost minim. Algoritmul este implementat folosind o coada de prioritate, fiind foarte asemanator cu BFS-ul, daca am presupune ca toate muchiile au acelasi cost.
  Cat timp in coada exista noduri, se va extrage nodul cu cea mai mica prioritate si se va adauga in lista

de noduri vizitate in cazul in care nu s-a vizitat anterior. Daca nodul curent este goalState, algoritmul o sa returneze lista de actiuni ale nodului curent. Altfel, se parcurg succesorii nodului curent si se construieste noua lista de actiuni formata din actiunile nodului curent si actiunile succesorilor, se calculeaza noul cost (suma costului curent + costul succesorilor) si se creeaza un nou element de tipul GraphNode. Se adauga in coada nodul creat impreuna cu prioritatea egala cu noul cost calculat.

Commands:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
  python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
  python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

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: Algoritmul UCS exploreaza mult mai multe noduri decat DFS, insa drumurile gasite de UCS au un cost mai mic, deci scroul final mai mare. UCS, fata de DFS, ia in calcul mai intai nodul cu cel mai mic cost (utilizant coada cu prioritate)

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: Agentul StayWest calculeaza costul pozitiei urmatoare ridicand (1/2) la puterea coordonatei x. In cazul in care coordonata x creste, costul actiunii o sa scada (cresterea coordonatei x inseamna ca se merge catre est). Astfel calculand mereu un cost mai mic pentru pozitiile dinspre est, se va favorizeaza alegerea acestora.

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

http://ai.berkeley.edu/search.html
https://biblioteca.utcluj.ro/files/carti-online-cu-coperta/290-8.pdf
https://www.geeksforgeeks.org/search-algorithms-in-ai/

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
3  starting_node = problem.getStartState()
4
5      if problem.isGoalState(starting_node):
6          return []
7
8      visited = []
9      queue = util.PriorityQueue()
10
11     node = GraphNode(starting_node, None, [], 0)
12     queue.push(node, 0)
13
14     while not queue.isEmpty():
15
16         current = queue.pop()
17
18         if current.getState() not in visited:
19             visited.append(current.getState())
20
21             if problem.isGoalState(current.getState()):
22                 return current.getAction()
23
24             for next_node, action, cost in problem.getSuccessors(current.getState()):
25                 next_action = current.getAction() + [action]
26                 next_cost = current.getCost() + cost
27                 heuristic_cost = next_cost + heuristic(next_node, problem)
28                 new_node = GraphNode(next_node, current, next_action, next_cost)
29                 queue.push(new_node, heuristic_cost)
30
31     util.raiseNotDefined()
```

Listing 1: Solution for the A* algorithm.

Explanation:

- A * este un algoritm de cautare informat. Algoritmul combina punctele forte ale UCS-ului și a algoritmului de cautare Greedy. In această cautare se foloseste euristica ce reprezinta insumarea costului UCS si o estimare a distantei dintre nodul curent si nodul destinatie. Precum la UCS algoritmul este implementat folosind o coada de prioritate.
  Cat timp in coada exista noduri, se va extrage nodul cu cea mai mica prioritate si se va adauga in lista de noduri vizitate in cazul in care nu s-a vizitat anterior. Daca nodul curent este goalState algoritmul o sa returneze lista de actiuni ale nodului curent. Altfel, se parcurg succesorii nodului curent si se construieste noua lista de actiuni formata din actiunile nodului curent si actiunile succesorilor, se calculeaza noul cost, pe baza costurilor succesorilor si pe baza unei euristici si se creeaza un nou element de tipul GraphNode care se adauga in coada pe baza prioritatii (euristica +costul actiunilor).

Commands:

- python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

2.1.2 Questions

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

Q1: Does A* and UCS find the same solution or they are different?
A1: Atat A* cat si UCS gasesc solutia optima. Pentru fiecare tip de labirint cei doi algoritmi gasesc aceeasi solutie.

Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2: Deoarece spre deosebire de UCS, A* este un algoritm de cautare informat, acesta reuseste sa gaseasca solutia optima expandand mai putine noduri , prin utilizarea unei euristici care estimeaza cat de aproape este nodul curent de starea finala.

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

2.1.3 Personal observations and notes

2.2 Question 5 - Find all corners - problem implementation

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

"Pacman needs to find the shortest path to visit all the corners,regardless there is food dot there or not. Go to ***CornersProblem*** *in searchAgents.py and propose a representation of the state of this search problem. It might help to look at the existing implementation for PositionSearchProblem. The representation should include only the information necessary to reach the goal. Read carefully the comments inside the class CornersProblem.".*

2.2.1 Code implementation

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

the implementation or testing process (please fill in *just the arguments*).

Code:

```
class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState):
        """
        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

        self.flags = [0, 0, 0, 0]

    def getStartState(self):

        return self.startingPosition, self.flags

        util.raiseNotDefined()

    def isGoalState(self, state):

        for flag in state[1]:
            if flag == 0:
                return False

        return True

        util.raiseNotDefined()

    def getSuccessors(self, state):

        successors = []

        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

            x,y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            flags = state[1][:]

            if not hitsWall:

                if (nextx, nexty) in self.corners:
```

```
                    flags[self.corners.index((nextx, nexty))] = 1

                successors.append(((nextx, nexty), flags), action, 1))

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

Explanation:

- Pentru implementarea problemei de vizitare a celor 4 colturi din labrint, am introdus in constructorul clasei CornersProblem o lista de 4 flaguri, folosita pentru marcarea colturilor ca vizitat (1) sau nevizitate (0).
  Daca starea curenta este goal state atunci inseamna ca toate flagurile din lista self.flags sunt setate pe 1. In caz contrar, stim ca nu s-a atins scopul problemei.
  Pentru obtinerea succesorilor starii curente, se parcurg toate actiunile posibile (nord,sud,est,vest) si se verifica ulterior daca sunt legale sau nu. Se determina noua pozitie, adunand la coordonatele actuale coordonatele actiunii corespunzatoare. In cazul in care nu se intra intr-un perete in urma actiunii (actiunea este legala), se verifica daca ne aflam intr-unul din colturi pentru a seta flagul corespunzator coltului pe 1.De asemenea, se adauga in lista de succesori un nod (format din coordonatele pozitei noi si lista de flaguri), actiunea, si costul egal cu 1.

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: 1968

2.2.3 Personal observations and notes

2.3 Question 6 - Find all corners - Heuristic definition

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

"Implement a consistent heuristic for CornersProblem. Go to the function ***cornersHeuristic*** *in searchAgent.py.".*

2.3.1 Code implementation

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

Code:

```python
def cornersHeuristic(state, problem):

    from util import manhattanDistance
    corners = problem.corners
    if problem.isGoalState(state):
        return 0

    unvisited = []
    flags = state[1][:]

    for i, flag in enumerate(flags):
        if flag == 0:
            unvisited.append(corners[i])

    heuristic_cost = 0
    current = state[0]

    while unvisited:
        distance, corner = min(
            [(manhattanDistance(current, corner), corner) for corner in unvisited])
        unvisited.remove(corner)
        current = corner
        heuristic_cost += distance

    return heuristic_cost
```

Explanation:

- Functia cornersHeuristic descrie o euristica folosita de A* pentru rezolvarea problemei vizitarii celor 4 colturi in mod optim si cu expandarea a cator mai putine noduri.
  Se incepe prin a crea o lista cu colturile nevizitate, parcurgand vectorul de flaguri si adaugand colturile ale caror flag-uri sunt egale cu 0. Se va initializa o variabila ce estimeaza totalul distantelor ce trebuiesc parcurse pentru a atinge scopul problemei. Se va initializa aceasta variabila cu 0. Intr-o bucla vom calcula distanta aproximativa de la punctul curent la toate colturile nevizitate, folosind distanta manhattan. Dupa care vom adauga la variabila ce estimeaza totalul, distanta la cel mai apropiat colt. Se va actualiza punctul curent la cel mai apropiat colt si vom scoate coltul din lista de colturi nevizitate. Bucla se repeta pana cand lista de colturi nevizitate este goala. La final se va returna suma distantelor gasite.

Commands:

- python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

2.3.2 Questions

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

Q1: Test with on the mediumMaze layout. What is your number of expanded nodes?
A1: 692

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
3      from util import manhattanDistance
4
5      position, foodGrid = state
6      food_dots = foodGrid.asList()
7
8      if problem.isGoalState(state):
9          return 0
10
11     closest_dot = min( [(manhattanDistance(position, food_dot), food_dot) for food_dot in food_dots])[1]
12
13     distance = mazeDistance(position, closest_dot, problem.startingGameState)
14
15     return distance + len(food_dots) -1
16
```

Explanation:

- Functia foodHeuristic descrie o euristica folosita de A* pentru gasirea unei solutii optime (un drum in care Pacman colecteaza toata mancarea).
  In cazul in care state-ul actual este goalState se va returna 0, iar in caz contrar se calculeaza, din lista ce contine coordonatele punctelor de mancare, cel mai apropiat food dot, folosind manhattanDistance. Se va calcula si distanta reala dintre pozitia curenta a pacman-ului si cel mai apropiat food dot folosind mazeDistance. Euristica noastra va returna o valoare care resprezinta suma distantei reale dintre pozitia curenta si cel mai apropiat food dot si food dot-urile ramase care nu au fost mancate de catre pacman.

Commands:

- python pacman . py -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: 5/4

2.4.3 Personal observations and notes

2.5 References

http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html
https://www.redblobgames.com/pathfinding/a-star/implementation.html

3 Adversarial search

3.1 Question 8 - Improve the ReflexAgent

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

"Improve the ReflexAgent such that it selects a better action. Include in the score food locations and ghost locations. The layout testClassic should be solved more often.".

3.1.1 Code implementation

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

Code:

```
1  def evaluationFunction(self, currentGameState, action):
2          successorGameState = currentGameState.generatePacmanSuccessor(action)
3          newPos = successorGameState.getPacmanPosition()
4          newFood = successorGameState.getFood()
5          ghost_positions = currentGameState.getGhostPositions()
6
7          distToGhosts = [manhattanDistance(newPos, ghostPos) for ghostPos in ghost_positions]
8          distToFood = [manhattanDistance(newPos, foodPos) for foodPos in newFood.asList()]
9
10         d1 = min(distToFood) if len(distToFood) > 0 else float("inf")
11         d2 = min(distToGhosts)
12         if d1 == 0:
13             d1 = float("inf")
14         if d2 == 0:
15             d2 = float("inf")
16
17         for ghost in successorGameState.getGhostPositions():
18             if (manhattanDistance(newPos, ghost) == 1):
19                 return -float("inf")
20
21         return successorGameState.getScore() + 1./d1 - 1./d2
```

Explanation:

- Pentru a ne asigura ca pacman va selecta cea mai buna actiune pentru a avea un scor cat mai mare si a nu muri, se va lua in considerare distanta pana la cel mai apropiat food dot, distanta pana la fantome, dar si fantomele care sunt foarte aproape de pacman.
  Se determina distanta catre mancare si fantome cu ajutorul distantei manhattan, calculandu-se apoi minimul dintre ele. Pentru ca pacman alege mereu actiunea care duce la cel mai bun scor, in cazul in care exista fantome ce se afla la o distanta manhattan egala cu 1 fata de pacman, functia va returna un numar extrem de mic (- inf). Altfel, se returneaza scorul obtinut adunat cu valorile reciproce ale distantelor catre cel mai apropiat food dot si cea mai apropiata fantoma.

Commands:

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

3.1.2 Questions

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

Q1: Test your agent on the openClassic layout. Given a number of 10 consecutive tests, how many types did your agent win? What is your average score (points)?
A1: 10/10
Avergae score: 1420.9

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:

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):

        def minimax_decision(current, next_state, depth, agent):

            if agent < current.getNumAgents() - 1:
                return min_value(next_state, depth, agent + 1)
            else:
                return max_value(next_state, depth + 1, 0)

        def max_value(current, depth, agent):

            actions = current.getLegalActions(agent)

            if current.isWin() or current.isLose() or depth == self.depth:
                return self.evaluationFunction(current), None

            max_val = -float("inf")
            max_action = None

            for action in actions:

                next_state = current.generateSuccessor(agent, action)
                value = minimax_decision(current, next_state, depth, agent)[0]
```

```
26
27                 if value > max_val:
28                     max_val = value
29                     max_action = action
30
31             return max_val, max_action
32
33         def min_value(current, depth, agent):
34
35             actions = current.getLegalActions(agent)
36
37             if current.isWin() or current.isLose() or depth == self.depth:
38                 return self.evaluationFunction(current), None
39
40             min_val = float("inf")
41             min_action = None
42
43             for action in actions:
44
45                 next_state = current.generateSuccessor(agent, action)
46                 value = minimax_decision(current, next_state, depth, agent)[0]
47
48                 if value < min_val:
49                     min_val = value
50                     min_action = action
51
52             return min_val, min_action
53
54         return max_value(gameState, 0, 0)[1]
55
56         util.raiseNotDefined()
```

Explanation:

- Minimax este asemanator cu un algoritm de backtracking care este utilizat in luarea deciziilor si în teoria jocurilor pentru a gasi miscarea optima pentru un jucator, presupunand ca și adversarul tau joaca optim.
  In Minimax, exista doua tipuri de jucatori, anume maximizator și minimizator. Maximizatorul încearcă să obțină cel mai mare scor posibil, în timp ce minimizatorul încearcă să facă contrariul și să obțină cel mai mic scor posibil. In cazul jocului de pacman , maximizatorul este pacman, iar fantomele sunt minimizatori.
  Am implementat acest algoritm cu ajutorul a trei functii: minimax decision, min value si max value.
  Functia minimax decision verifica daca suntem pe un agent care doreste maximizarea sau pe un agent care doreste minimzarea. In cazul in care agentul pe care a fost apelata functia este mai mic decat numarul total de agenti - 1 (numerotarea agentilor incepe de la 0) atunci inseamna ca urmatorul agent este o fantoma care doreste minimizarea, deci se apeleaza functia min value cu aceeasi adancime si cu incrementarea agentului curent , in caz contrar inseamna ca am ajuns la ultima fantoma si urmatorul agent va a fi pacman (care este agentul 0) si se va apela functia max value cu incrementarea adancimii si reinitializarea agentului la 0.
  Functia max value este apelata doar de catre agentul 0, reprezentat de pacman. Aceasta functie va returna valoarea maxima care poate fi obtinuta si o actiune corespunzatoare acestei valori. In primul rand functia va verifica daca se afla intr-un stadiu terminal (sfarsit de joc sau ajungerea la adancimea maxima pe care se face cautarea), in acest caz va returna rezultatul functiei de evaluare din starea curenta si nicio actiune. In cazul in care nu suntem intr-o stare terminala , vom initializa valoarea maxima la - infinit si vom initializa actiunea cu none . Vom parcurge lista cu actiunile legale din starea

curenta si pentru fiecare actiune se va apela minimax decision si se va verifica daca valoarea obtinuta este sau nu maxim. In final vom returna maximul si actiunea.
Functia min value se comporta asemanator functiei max value, singura diferenta fiind aceea ca se va calcula si se va returna minimul in loc de maxim.

Commands:

- python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
  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: Deoarece Pacman este blocat intre doua fantome, moartea sa este inevitabila, asa ca va încerca sa puna capat jocului cat mai curand posibil deoarece scorul scade cu cat este mai mult in viata.

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
4      def getAction(self, gameState):
5
6          def alphabeta_search(current, next_state, depth, agent, alpha, beta):
7
8              if agent < current.getNumAgents() - 1:
9                  return min_value(next_state, depth, agent + 1, alpha, beta)
10             else:
11                 return max_value(next_state, depth + 1, 0, alpha, beta)
12
13         def max_value(current, depth, agent, alpha, beta):
14
```

```
            actions = current.getLegalActions(agent)
            max_val = -float("inf")

            max_action = None

            if current.isWin() or current.isLose() or depth == self.depth:
                return self.evaluationFunction(current), None

            for action in actions:
                next_state = current.generateSuccessor(agent, action)
                value = alphabeta_search(current, next_state, depth, agent, alpha, beta)[0]
                if value > max_val:
                    max_val = value
                    max_action = action

                alpha = max(alpha, max_val)

                if alpha > beta:
                    return max_val, max_action

            return max_val, max_action

        def min_value(current, depth, agent, alpha, beta):
            actions = current.getLegalActions(agent)

            if current.isWin() or current.isLose() or depth == self.depth:
                return self.evaluationFunction(current), None

            min_val = float("inf")
            min_action = None

            for action in actions:
                newState = current.generateSuccessor(agent, action)
                value = alphabeta_search(current, newState, depth, agent, alpha, beta)[0]

                if value < min_val:
                    min_val = value
                    min_action = action

                beta = min(beta, min_val)

                if beta < alpha:
                    return min_val, min_action



            return min_val, min_action

        alpha = -float("inf")
        beta = float("inf")

        return max_value(gameState, 0, 0, alpha, beta)[1]
```

Explanation:

- Alfa-beta prunning este un algoritm de căutare care urmarește scaderea numarului de noduri care sunt evaluate de algoritmul minimax din arborele sau de căutare. .Opreste evaluarea unei mutari atunci cand a fost gasită cel puțin o posibilitate care dovedeste că mutarea este mai rea decât o mutare examinata anterior. Astfel de miscari nu trebuie evaluate în continuare. Atunci cand este aplicat unui arbore standard de minimax, acesta returneaza aceeasi miscare ca și minimax, dar elimină ramurile care nu pot influenta decizia finala.

  Am implementat acest algoritm cu ajutorul a trei functii foarte asemanatoare cu cele implementate pentru algoritmul minimax: alphabeta search, min value si max value.
  Functia alphabeta search este identica cu functia minimax decision , doar ca se adauga la parametrii functiei argumentele alpha si beta.
  Functia max value este apelata doar de catre agentul 0, reprezentat de pacman. Aceasta functie va returna valoarea maxima care poate fi obtinuta si o actiune corespunzatoare acestei valori. In primul rand functia va verifica daca se afla intr-un stadiu terminal (sfarsit de joc sau ajungerea la adancimea maxima pe care se face cautarea), in acest caz va returna rezultatul functiei de evaluare din starea curenta si nicio actiune. In cazul in care nu suntem intr-o stare terminala , vom initializa valoarea maxima la - infinit si vom initializa actiunea cu none . Vom parcurge lista cu actiunile legale din starea curenta si pentru fiecare actiune se va apela minimax decision si se va verifica daca valoarea obtinuta este sau nu maxim. La fiecare pas in bucla se recalculeaza valoarea parametrului alpha, care este maximul dintre maximul calculat anterior si valoarea lui beta. Daca valoarea lui alpha este mai mare decat cea a lui beta,inseamna ca nu mai are sens sa exploram restul actiunilor si se va returna tupla cu valoarea maxima si actiunea corespunzatoare valorii. In cazul in care nu se indeplineste conditia mentionata anterior se va continua evaluarea actiuniilor si functia returneaza in final maximul si actiunea gasita.
  Functia min value se comporta asemanator functiei max value, singura diferenta fiind ca se va calcula si se va returna minimul in loc de maxim. Se va actualiza la fiecare actiune valoarea parametrului beta care este minimul dintre valoarea minima din momentul respectiv si beta, se va verifica daca beta este mai mic ca alfa si daca aceasta conditie este indeplinita nu va mai analiza restul actiuniilor. Daca nu se indeplineste aceasta conditie pana la final se vor returna minimul gasit si actiunea corespunzatoare.

Commands:

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

3.3.2 Questions

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

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

A1: 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