Artificial intelligence - Project 1
- Search problems -

Sandru Diana - Teodora

2/11/2020

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      print("Start:", problem.getStartState())
4      print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
5      print("Start's successors:", problem.getSuccessors(problem.getStartState()))
6
7      stiva = util.Stack()
8      start = problem.getStartState()
9      visitate = []
10     tupla = (start, [])
11     stiva.push(tupla)
12     while not stiva.isEmpty():
13         stare, actiune = stiva.pop()
14         if problem.isGoalState(stare):
15             return actiune
16         visitate.append(stare)
17         copii = problem.getSuccessors(stare)
18         for copil in copii:
19             stareCopil = copil[0]
20             directieCopil = copil[1]
21             if not stareCopil in visitate:
22                 stiva.push((stareCopil, actiune + [directieCopil]))
23
24     util.raiseNotDefined()
```

Explanation:

- DFS este un algoritm pentru traversarea sau căutarea structurilor de date: arbore sau graf. Algoritmul începe de la nodul rădăcină (selectând un nod arbitrar ca nod rădăcină în cazul unui graf) și explorează cât mai mult posibil de-a lungul fiecărei ramuri.
- Incepem prin a initializa o stiva, intrucat DFS lucreaza cu stiva. Se declara o lista a nodurilor vizitate/expandate, iar in acea stiva pe prima pozitie adaugam nodul(starea de start) a Pacman-ului. Atata timp cat lista nu e goala, extragem pe rand din stiva cate-un nod (stare). Daca acea stare este starea scop (finalul, mancarea) atunci algoritmul returneaza setul de actiuni pe care le face Pacman-ului ca sa ajunga din starea initiala(start) in starea care s-a dovedit a fi staree scop. Daca nodul nu e nod scop se marcheaza ca vizitat si se exploreaza succesorii lui(copiii) prin functia *getSuccessors()*, punandu-se si ei, la randul lor, 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 nu aduce o solutie optima intrucat nu gaseste nodul cu adancimea minima. DFS nu exploreaza toti vecinii unui nod, doar daca nu gaseste solutia va reveni si va exploata vecinii nodului, pe rand(daca este cazul). Astfel, nu se garanteaza solutia optima.
Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: 3/3

1.1.3 Personal observations and notes

Problemele in timpul rezolvarii task-ului au fost legate de sintaxa noua de python nu neaparat de ceea ce trebuia implementat

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      "*** YOUR CODE HERE ***"
4      frontier = util.Queue()
5      state = problem.getStartState()
6      expanded = []
7      current = Node(state, None, None, 0)
8      frontier.push(current)
9      while (not frontier.isEmpty()):
10         current = frontier.pop()
11         expanded += [current.getState()]
12         if problem.isGoalState(current.getState()):
```

```
            solution = []
            while (current.getParent()):
                solution += [current.getAction()]
                current = current.getParent()
            solution.reverse()
            return solution
        for (state, action, _) in problem.getSuccessors(current.getState()):
            newNode = Node(state, current, action, 0)
            if state not in expanded and newNode not in frontier.list:
                frontier.push(newNode)

    util.raiseNotDefined()
```

Explanation:

- BFS (Căutarea în lățime) este un algoritm pentru parcurgerea sau căutarea într-o structură de date de tip arbore sau graf. Aceasta începe cu rădăcina arborelui și explorează nodurile mai întâi nodurile vecine acestuia, înainte de a trece la vecinii de pe nivelul următor.
- BFS utilizeaza o coada, spre deosebire de DFS unde am vazut ca utilizeaza o stiva.
- Nodurile vor fi exploatate in acea ordine in care au fost adugate in coada.
- Incepem prin a declara o coada goala si o lista de nosuri expandate tot goala in aceasta etapa de intializare. In coada se pune o prima stare, un nod de start (de unde porneste Pacman-ul). Cat timp coada nu e goala, se extrage din ea starea prin apelul lui pop(). Daca starea extrasa e staree scop algoritmul de cautare se termina aici, dacac nu nodul asociat starii se marcheaza ca vizitat dupa care se continua cu exploatarea succesorilor (nodurilor copii).

Commands:

- python pacman . py -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 de BFS gaseste solutia optima, asta doar in cazul in care nu se tine cont de costuri (adica este un algorimt "neinformat"). Spre deosebire de DFS, gaseste nodul scop cel mai apropiat de nodul de inceput intrucat BFS expandeaza pe nivele de noduri..
Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: 3/3

1.2.3 Personal observations and notes

La acest algoritm mi-a fost destul de greu de gasit o logica de implementare pana cand am creat noua clasa Node care mi-a facilitat munca cu acest algoritm.

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 ***"
    coadaP = util.PriorityQueue()
    start = problem.getStartState()
    visitedNodes = []
    coadaP.push((start, []), 0)
    while not coadaP.isEmpty():
        stare, actions = coadaP.pop()
        if stare not in visitedNodes:
            if problem.isGoalState(stare):
                return actions
            copii = problem.getSuccessors(stare)
            for copil in copii:
                stareCopil = copil[0]
                directieCopil = copil[1]
                if stareCopil not in visitedNodes:
                    visitedNodes.append(stare)
                    cost = problem.getCostOfActions(actions + [directieCopil])
                    coadaP.push((stareCopil, actions + [directieCopil]), cost)

    util.raiseNotDefined()
```

Explanation:

- UCS este un algoritm de căutare utilizat pentru parcurgerea unui arbore sau a unui graf ponderat (cu costuri). Acest algoritm intervine atunci când este disponibil un cost diferit pentru fiecare margine/muchie. Scopul principal al căutării cu costuri uniforme este de a găsi o cale către nodul-obiectiv care are cel mai mic cost cumulat. Căutarea cu costuri uniforme extinde nodurile în funcție de costurile căii lor din nodul rădăcină. Poate fi folosit pentru a rezolva orice graf / arbore în care costul optim este solicitat. Un algoritm de căutare cu costuri uniforme este implementat de coada de prioritate. Oferă prioritate maximă celui mai mic cost cumulat. Căutarea uniformă a costurilor este echivalentă cu algoritmul BFS dacă costul căii tuturor marginilor este același.
- Avantaje:

  UCS este un algoritm optim, deoarece în fiecare stare se alege calea cu cel mai mic cost.
- Dezavantaje:

  Nu-i pasă de numărul de pași implicați în căutare și este preocupat doar de costul căii. Datorită acestui aspect, algoritmul poate fi blocat într-o buclă infinită.
- Dupa cum se poate observa din implementare, UCS este implementat exact ca BFS cu diferenta ca aici avem de a face cu o coada de prioritate, care pune costul ca prioritate si face urmatoarele alegeri de deplasare in functie de acest cost.

Commands:

- python pacman . py -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: Daca comparam cei doi algoritmi de cauatare, obtinem rezultate diferite. Numarul de noduri expandate de UCS este mai mare decat numarul de noduri expandate de DFS. UCS este un algortim optim, gaseste solutia cu cost minim cea mai apropiata de nosul sursa. Asta inseamna ca incepanad de la nodul sursa pana la nodul scop UCS incearca mai multe variante pana o gaseste pe cea optima, adica cu cost minim, pe cand DFS nu gaseste starea scop prin cea mai "putin" adanca cale. Nodurile expandate de UCS sunt evident mai multe decat cele expandate de DFS.

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: Se utilizeaza StayWestAgent pentru a mentine Pacmanul sa parcurga doar pozitii cu cost minim. Costul este asociat agentului StayEastAgent intrucat costul pasirii intr-o zona ilegala (x, y) este (1/2)**x.

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

A3: 3/3

1.3.3 Personal observations and notes

UCS este un algoritm optim atunci cand, in primul rand, suntem interesati de un cost total minim, nu neaparat de calea parcursa de la sursa la scop.

1.4 References

https://stackoverflow.com/questions/21082771/uniform-cost-search-vs-depth-first-search
https://www.researchgate.net/publication/333262471$_{C}omparative_{A}nalysis_{o}f_{S}earch_{A}lgorithms$
$https://www.wikipedia.org/$

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      "*** YOUR CODE HERE ***"
4      coadaP = util.PriorityQueue()
5      start = problem.getStartState()
6      visitedNodes = []
7      coadaP.push((start, []), 0)
8      while not coadaP.isEmpty():
9          stare, actions = coadaP.pop()
10         if stare not in visitedNodes:
11             if problem.isGoalState(stare):
12                 return actions
13             copii = problem.getSuccessors(stare)
14             for copil in copii:
15                 stareCopil = copil[0]
16                 directieCopil = copil[1]
17                 if stareCopil not in visitedNodes:
18                     visitedNodes.append(stare)
19                     g = problem.getCostOfActions(actions + [directieCopil])
20                     h = heuristic(stareCopil, problem)
21                     prioritate = h + g
22                     coadaP.push((stareCopil, actions + [directieCopil]), prioritate)
23
24     util.raiseNotDefined()
```

Listing 1: Solution for the A* algorithm.

Explanation:

- A * este un algoritm de parcurgere a graficului și de căutare a căilor, care este adesea utilizat în multe domenii ale informaticii datorită completitudinii, optimității și eficienței sale optime. Un dezavantaj practic important este complexitatea sa spațială, deoarece stochează toate nodurile generate în memorie.
- Și, de asemenea, merită menționat faptul că multe jocuri și hărți bazate pe web folosesc acest algoritm pentru a găsi cea mai scurtă cale in mod eficient.

- Ceea ce face un algoritm de căutare este că la fiecare pas alege nodul urmator în funcție de o valoare f care este un parametru egal cu suma altor doi parametri - „g" și „h". La fiecare pas, alege nodul / celula având cel mai mic „f" și procesează acel nod / celulă.
- g = costul de mișcare pentru a trece de la punctul de plecare la un pătrat dat de pe grilă, urmând calea generată pentru a ajunge acolo.
- h = costul de mișcare estimat pentru a trece de la acel pătrat dat pe grilă la destinația finală. Aceasta este adesea denumită euristică, care nu este altceva decât un fel de "ghicire" inteligentă. Cand nu cunoaștem distanța reală până când nu găsim calea, deoarece tot felul de lucruri pot fi în cale (pereți, fantome etc.). Pot fi multe modalități de a calcula acest „h".

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: UCS este un caz special de A*. In mometul in care euristica considerata (h) este euristica nula, atunci se tine cont doar de g (cost) la fel cum se tine cont in cazul UCS. Asadar, putem spune ca A* si UCS pot gasi aceleasi solutii in unele cazuri. A* este mai eficient incat se foloseste de euristica, mai ales daca euristica este admisibila si consistenta.
Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2: A* gaseste solutia reusind sa expandeze mai putine noduri, intrucat se foloseste de doua functii, euristica (care estimeaza distanta nodului fata de scop) si functia de cost pentru a fi implementat. Daca avem o euristica "mai eficienta", adica consistenta de preferat, se vor extinde mai putine noduri.
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

A* este un algoritm foarte eficient care se bazeaza simultan pe doua aspecte: costul de la starea sursa la starea scop si euristica (care prevede existenta unor obstacole). Este folosit in majoritatea aplicatiilor web.

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):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor 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
        "*** YOUR CODE HERE ***"

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        #corners = (False, False, False, False)
        #start = (self.startingPosition, corners)
        stateStart = self.startingPosition, []
        return stateStart
        util.raiseNotDefined()

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        pos, cornersV = state
        if pos in self.corners:
            if pos not in cornersV:
                cornersV.append(pos)
        if len(cornersV) == 4:
            return True
        return False

```

```
49
50          util.raiseNotDefined()
51
52      def getSuccessors(self, state):
53          """
54          Returns successor states, the actions they require, and a cost of 1.
55
56           As noted in search.py:
57              For a given state, this should return a list of triples, (successor,
58              action, stepCost), where 'successor' is a successor to the current
59              state, 'action' is the action required to get there, and 'stepCost'
60              is the incremental cost of expanding to that successor
61          """
62
63          successors = []
64          for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
65              # Add a successor state to the successor list if the action is legal
66              # Here's a code snippet for figuring out whether a new position hits a wall:
67              currentPosition, cornersV = state
68              x, y = currentPosition
69              dx, dy = Actions.directionToVector(action)
70              nextx, nexty = int(x + dx), int(y + dy)
71              hitsWall = self.walls[nextx][nexty]
72              if not hitsWall:
73                  nextPosition = nextx, nexty
74                  cornersVSuc= list (cornersV)
75                  if nextPosition in self.corners:
76                      if nextPosition not in cornersVSuc:
77                          cornersVSuc.append(nextPosition)
78                  nextState = nextPosition, cornersVSuc
79                  cost = 1
80                  successors.append((nextState, action, cost))
81
82              "*** YOUR CODE HERE ***"
83
84          self._expanded += 1  # DO NOT CHANGE
85          return successors
86
87
88      def getCostOfActions(self, actions):
89          """
90          Returns the cost of a particular sequence of actions.  If those actions
91          include an illegal move, return 999999.  This is implemented for you.
92          """
93          if actions == None: return 999999
94          x,y= self.startingPosition
95          for action in actions:
96              dx, dy = Actions.directionToVector(action)
97              x, y = int(x + dx), int(y + dy)
98              if self.walls[x][y]: return 999999
99          return len(actions)
```

Explanation:

- Acest task presupune vizitarea tuturor colturilor, chiar daca acolo exista sau nu mancare(food dot).

- Am completat functia getStartState() prin simplul fapt ca apelul acestei acum functii va returna nu doar nodul in care ne aflam in starea curenta, ci si o lista de colturi din cele patru care au fost traversate pana la acel moment de timp.
- O stare este stare scop daca am ajuns in punctul potrivit si daca calea de a ajunge acolo a presupus traversarea tuturor celor patru colturi. Astfel, daca lungimea listei care tine colturile ce au fost traversate este 4, atunci ne aflam cu adevarat in starea scop. Totodata, poate starea scop sa fie chiar un colt, atunci adaugam coltul respectiv in loista de colturi vizitate si afirmam starea ca fiind scop.
- Pentru a afla succesorii in astfel de conditii, avem nevoie sa stim daca acesti succesori trec sau nu prin colturi, sa cream o noua lista in care sa adaugam colturile vechi deja traversate si potentialele noi colturi.

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: A* expandeaza mai putin noduri, in jur de 900 (846).

2.2.3 Personal observations and notes

Mi s-a parut o maniera foarte utila de intelegere a codului deja existent prin crearea propriei noastre "probleme" in ceea ce priveste Pacman-ul.

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
3     "*** YOUR CODE HERE ***"
4
5
```

```
    currentPoint, cornersV = state
    cornersnV = []
    heuristicL = []
    cost = 0
    curPoint, crns = state
    totalCost = 0;

    for c in problem.corners:
        if c not in cornersV:
            cornersnV.append(c)

    while cornersnV:
        for cor in cornersnV:
            cost, cornerCost = (util.manhattanDistance(curPoint, cor), cor)
            tupla = cost, cornerCost
            heuristicL.append(tupla)
        # print ("cor ramase:")
        # print(cornersnV)
        # print(heuristicL)
        heuristic_cost, corM = min(heuristicL)
        heuristicL = []
        # print("heuristica dupa stergere:")
        # print(heuristicL)
        # print(heuristic_cost)
        # print(corM)
        cornersnV.remove(corM)
        curPoint = corM
        totalCost += heuristic_cost
    return totalCost
```

Explanation:

- O euristică este o tehnică pentru a rezolva o problemă mai rapid decât metodele clasice sau pentru a găsi o soluție aproximativă atunci când metodele clasice nu pot. Acesta este un fel de comandă rapidă, una de optimitate, completitudine, acuratețe sau precizie pentru viteză. O euristică (sau o funcție euristică) aruncă o privire asupra algoritmilor de căutare. La fiecare pas de ramificare, evaluează informațiile disponibile și ia o decizie cu privire la ce ramură să urmeze. O face prin clasarea alternativelor.
- Pentru acest task va trebui sa concepem o euristica prin care toate cele patru colturi sa fie vizitate cat mai optim si totdata, toate bucatile de mancare sa fie mancate.
- In primul rand, vom pune intr-o lista toate colturile care pana la momentul de fata nu au fost vizitate. Vom afla distanata Manhattan dintre pozitia curenta si fiecare dintre colturile ramase inca nevizitate. Apoi vom alege sa vizitam coltul pentru care distanta Manhattan este cea mai mica, iar pe urma vom actualiza pozitia curenta cu noul colt in care am migrat si vom scoate din lista colturilor nevizitate coltul respectiv.

Commands:

- -l mediumCorners -p SearchAgent -a fn = aStarSearch , prob = CornersProblem , heuristic = cornersHeuristic
  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: 1407

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     position, foodGrid = state
4     "*** YOUR CODE HERE ***"
5     return 0
```

Explanation:

-

Commands:

-

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:

2.4.3 Personal observations and notes

2.5 References

3 Adversarial search

3.1 Question 8 - Improve the ReflexAgent

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

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

3.1.1 Code implementation

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

Code:

```
def evaluationFunction(self, currentGameState, action):
    """
    Design a better evaluation function here.

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

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

    Print out these variables to see what you're getting, then combine them
    to create a masterful evaluation function.
    """
    # Useful information you can extract from a GameState (pacman.py)
    successorGameState = currentGameState.generatePacmanSuccessor(action)

    # newFood = successorGameState.getFood()
    # newGhostStates = successorGameState.getGhostStates()
    # newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

    "*** YOUR CODE HERE ***"

    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood().asList()
    FoodL = []
    minFood = -1
    newGhostStates = successorGameState.getGhostStates()

    for food in newFood:
        distPacFood = manhattanDistance(newPos, food)
        if distPacFood <= minFood or minFood == -1:
            minFood = distPacFood

    # for food in newFood:
```

```
37      # minFoodist = min(minFoodist, manhattanDistance(newPos, food))
38
39      # avoid ghost if too close
40      newGhostPosition = successorGameState.getGhostPositions()
41      for ghost in newGhostPosition:
42          distPacGhost = manhattanDistance(newPos, ghost)
43          if (distPacGhost < 2):
44              return -float('inf')
45      # reciprocal
46      return successorGameState.getScore() + 1.0 / minFood
```

Explanation:

- Incepem prin a afla noua posibila pozitie a Pacman-ului si noile posibile bucati de mancare ca lista.
- Aflam cu distanta Manhattan care este cea mai apropiata bucata de mancare de Pacman.
- Aflam tot cu distanta Manhattan care este distanat dintre Pacman si fantoma. Daca distanta este mai mica decat 2 atunci, am pierdut.

Commands:

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

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  def getAction(self, gameState):
2      """
```

```
      Returns the minimax action from the current gameState using self.depth
      and self.evaluationFunction.

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

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

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

      gameState.getNumAgents():
        Returns the total number of agents in the game
    """
    "*** YOUR CODE HERE ***"
    _, action = self.max_value(gameState, 0)
    return action


def max_value(self, gameState, depth):
    legalM = gameState.getLegalActions(0)
    if len(legalM) == 0 or depth == self.depth:
        return gameState.getScore(), ""
    max_value = None
    max_action = ""
    for action in legalM:
        successor = gameState.generateSuccessor(0, action)
        value, _ = self.min_value(successor, 1, depth)

        if max_value is None or value > max_value:
            max_value = value
            max_action = action
    return max_value, max_action


def min_value(self, gameState, index, depth):
    legalM = gameState.getLegalActions(index)
    if len(legalM) == 0 or depth == self.depth:
        return gameState.getScore(), None
    min_value = None
    min_action = ""
    next_index = index + 1
    if next_index == gameState.getNumAgents():
        next_index = 0

    for action in legalM:
        successor = gameState.generateSuccessor(index, action)
        if next_index == 0:
            value, _ = self.max_value(successor, depth + 1)
        else:
            value, _ = self.min_value(successor, next_index, depth)

        if min_value is None or value < min_value:
```

```
57              min_value = value
58              min_action = action
59      return min_value, min_action
```

Explanation:

- Pacmanul, care are indexul 0, va alege actiunea cu scorul cel mai mare, pe cand fantomele vor alege actiunea cu scorul minim (in defavoarea noastra, a Pacman-ului).
- Depth reprezinta numarul nivelului la care ne aflam
- In functia minvalue avem un index transmis ca argument intrucat putem opera cu una sau mai multe fantome.
- Totodata, in fiecare funtie exista un return intr-un if care returneaza scorul in cazul in care nu mai putem face actiuni legale sau in cazul in care am ajuns la nivelul (depth) cerut.

Commands:

- python pacman . py -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: Pacmanul actioneaza in acest fel intrucat obtine cel mai mare scor. Prima data se concentreaza pe obtinerea scorului, abia apoi pe fentome.

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
```

Explanation:

-

Commands:

-

3.3.2 Questions

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

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

3.3.3 Personal observations and notes

3.4 References

4 Personal contribution

4.1 Question 11 - Define and solve your own problem.

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

4.1.1 Code implementation

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

Code:

```
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