Artificial intelligence - Project 1
- Search problems -

Birlutiu Claudiu-Andrei

29/10/2021

1 Uninformed search

1.1 Question 1 - Depth-first search

In aceasta sectiune va fi prezentatata implementarea problemei 2.14 din laborator, al carei enunt este:

*"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

In continuare se regaseste codul propus pentru implemetarea acestui **algoritm de cuatare**. In incercarea de implementare si apoi testare a solutiei propuse am apelat mai multe comenzi din terminal pentru a vedea evolutia agentului *pacman* in procesul de cautare a solutiei sale.

Code:

```
def depthFirstSearch(problem):
    from game import Directions
    from util import Stack  # este nevoie de o stiva pentru a implementa iterativ algoritmul
    frontier = Stack()
    # starile parcurse
    reached = []
    node = Node(problem.getStartState(), None, [], 0)
    frontier.push(node)
    # se intra in iteratie; cat timp nu mai exista noduri candidat in frontiera
    while not frontier.isEmpty():
        node = frontier.pop()

        # daca nodul a fost explorat inainte, nu va mai fi expandats
        if node in reached:
            continue
        if problem.isGoalState(node.getState()):
            return construct_path(node)
        # se adauga la noduri explorate
        reached.append(node.getState())
        for (state, action, cost) in problem.expand(node.getState()):  # se cauta in lista de succesori
            childNode = Node(state, node, action, node.getCost() + cost)
            if state not in reached:
                # se adauga nodul copil la frontiera
                frontier.push(childNode)

    return construct_path(node)

def construct_path(node):
    solution = []
    while node:
        if node.getParent():
            solution.append(node.getAction())
        node = node.getParent()
    solution.reverse()
    return solution
```

Explanation:

- la linia 4 am importat structura de date Stack implementata in achetul util; aceasta structura permite adaugarea unui element, scoaterea elementului cel mai recent adaugat si permite verificarea starii acesteia (daca stiva e goala)
- la linia 4 ne declaram o variabila de tipul Stack, in care se vor adauga nodurile din frontiera in momentul expandarii grafului de cautare
- la linia 6 ne declaram o lista ce va contine nodurile deja explorate
- la lina 7 ne declaram nodul de start; Pentru aceasta s-a crea o clasa ce reprezinta structura de date a unui nod. Atributele acestei clase sunt: starea (pozitia din grid), nodul parinte, action (actiunea prin care s-a ajuns la el) si costul total al caii de la radacina
- la linia 8 se adauga in frontiera nodul de start, iar apoi se intra in bucla while a carei conditii de terminare il reprezinta testul pe frontiera. In cazul in care nu va mai exista nciun nod in frontiera va arata ca nu mai este niciun nod care sa fie expandat
- incepand cu linia 11 pana la linia 24 se va descrie ideea algorimului dfs; intotdeauna se va lua nodul cel mai recent adaugat in frontiera: daca acesta este scopul problemei de cautare, atunci algorimul se va opri si va returna solutia, iar in caz contrar va continua prin expandarea succesorilor acestui nod si adaugarea lor in frontiera daca nu au fost deja adaugati. Deoarece avem acest test de verificare a nodurilor (daca sunt in lista de noduri explorate) inainte de a fi adaugate in frontiera, va determina o parcurgere de tip **graph search**
- la linia 14 se verifica daca nodul extras din frontiera a fost deja explorat, in caz afirmativ va fi ignorat. Teoretic acest nod a fost adaugat de mai multe ori in frontiera deaorece nu s-a putut face testul de apartenenta pe structura Stack
- pentru reconstructia actiunilor prin care s-a ajuns la nodul scop din nodul start s-a implementat functia **construct_path(node)**, functie ce reconstruieste calea pe baza relatiei copil parinte
- valoarea returnata de functie va fi o lista cu actiunile ce urmeaza sa le execute agentul din pozitia initiala pana la scop

Commands:

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

1.1.2 Questions

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

A1: Nu gaseste solutia optima, deoarece algorimul nu tine seama de costul drumului pe care il alege. Cum ii zice si numele, el va cauta cat mai mult in adancime, astfel va ajunge sa exploreze o ramura pana la nivelul cel mai de jos si apoi daca nu va reusi sa gaseasca solutia va cauta si pe celelalte ramuri la un nivel mai apropiat de radacina si unde s-ar putea sa fie solutia

Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: Question q1: 4/4

1.1.3 Personal observations and notes

Pentru bigMaze costul total al caii este de 210, iar numarul de noduri explorate este de 390.Pentru mediumMaze costul total al caii este de 130, iar numarul de noduri explorate este de 146.Se vor face niste comparatii cu celelalte rezultate obtinute pentru ceilalti algoritmi implementati la fiecare subsectiune.

1.2 Question 2 - Breadth-first search

In continuare va fi descris exercitiul 2.15 din laborator:

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

1.2.1 Code implementation

In continuare se regaseste codul propus pentru implemetarea acestui **algoritm de cuatare**. In incercarea de implementare si apoi testare a solutiei propuse am apelat mai multe comenzi din terminal pentru a vedea evolutia agentului *pacman* in procesul de cautare a solutiei sale.

Code:

```
1  def breadthFirstSearch(problem):
2      """Search the shallowest nodes in the search tree first."""
3      "*** YOUR CODE HERE ***"
4      # se bazeaza pe codul din AIMA
5      from util import Queue
6
7      # va stoca nodurilor parcurse
8      reached = []
9      # frontiera de explorat
10     frontier = Queue()
11
12     node = Node(problem.getStartState(), None, [], 0)  # radacina
13     frontier.push(node)  # se pune in coada nodul parinte
14
15     # se intra in iteratie; cat timp nu mai exista noduri candidat in frontiera
16     while not frontier.isEmpty():
17         node = frontier.pop()
18         # daca noudul a fost vizitat, se va continua iteratia si nu se va expanda din nou
19         if node.getState() in reached:
20             continue
21         if problem.isGoalState(node.getState()):
22             return construct_path(node)
23         # se adauga nodul in noduri explorate
24         reached.append(node.getState())
25         for (state, action, cost) in problem.expand(node.getState()):  # se cauta in lista de succesori
26             if state not in reached:
27                 childNode = Node(state, node, action, node.getCost() + cost)
28                 frontier.push(childNode)
29
30     return construct_path(node)
```

Explanation:

- la linia 5 se importa structura de date Queue implementatata in util; ceasta structura permite adaugarea unui element, scoaterea elementului cel mai vechi adaugat si permite verificarea starii acesteia (daca e goala sau nu)
- la linia 8 se declara o lista vida in care se vor adauga nodurile explorate
- in cazul acestei solutii va fi nevoie de o coada pentru a descrie frontiera
- la liniile 12 si 13 va fi creat nodul de start (care are aceeasi structura descrisa la cerinta anterioara) si va fi adaugat in frontiera

- ideea acestui algoritm consta in scoaterea nodurilor din frontiera in ordinea in care au fost aduagate si expandarea acestora prin determinarea succesorilor si adaugarea lor in frontiera. Deoarece si in acest caz se merge pe o abordare de tip **graph search**, ianinte de a adauga in fronitera un nod se va verifica daca acesta nu a fost explorat deja.
- deoarece si in cazul cozii implementate in util nu putem sa determinam daca un element e deja in coada, am mers pe aceeasi idee de a adauga duplicate, iar cand se vor extrage se vor ignora - la linia 19 e tratat acest caz
- in cazul in care se ajunge la nodul scop se va opri cautarea si se va returna solutia problemei
- pentru a construi calea de la nodul de start la nodul scop cu actiunile necesare se va apela functia **construct_path(node)**

Commands:

- -l bigMaze -z .5 -p SearchAgent -a fn=bfs
- -l mediumMaze -z .5 -p SearchAgent -a fn=bfs
- -l tinyMaze -z .5 -p SearchAgent -a fn=bfs

1.2.2 Questions

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

Q1: Is the found solution optimal? Explain your answer.
A1: Da, bfs va gasi intotdeuana solutia optima daca costul tranzitiilor este acelasi. In cazul lui pacman costul este de 1 pentru orice tranzitie dintr-o stare in alta. De fapt, bfs va gasi calea cea mai scurta pana la nodul scop, deoarece el parcurge graful nivel cu nivel. Cu alte cuvinte, cand se va explora un nod de la un nivel, toate nodurile din nivelele mai apropoiate de radacina au fost explorate.
Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: Question q2: 4/4

1.2.3 Personal observations and notes

Pentru bigMaze costul total al caii este de 210, iar numarul de noduri explorate este de 620.Pentru mediumMaze costul total al caii este de 68, iar numarul de noduri explorate este de 269. Diferentele ce se observa intre dfs si bfs sunt la nivelul nodurilor explorate si la costul caii gasite. Observam ca bfs gaseste mereu calea optima, dar va explora mult mai multe noduri decat o face dfs.

1.3 References

R. Stuart, N. Peter, Artificial Intelligence: A Modern Approach, 4th US ed., capitol 3, [online]

2 Informed search

2.1 Question 4 - A* search algorithm

In sectiunea urmatoare va fi descris exercitiul 3.2 din laborator

*"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

In continuare se regaseste codul propus pentru implemetarea acestui algoritm de cuatare. In incercarea de implementare si apoi testare a solutiei propuse s-au apelat mai multe comenzi din terminal pentru a vedea evolutia agentului pacman in procesul de cautare a solutiei sale

Code:

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    """from util import PriorityQueue"""

    from util import PriorityQueue

    # va stoca nodurilor parcurse
    reached = []
    # frontiera de explorat
    frontier = PriorityQueue()

    node = Node(problem.getStartState(), None, [], 0)  # radacina
    frontier.push(node, heuristic(node.getState(), problem))  # se pune in coada nodul parinte

    # se intra in iteratie; cat timp nu mai exista noduri candidat in frontiera
    while not frontier.isEmpty():
        node = frontier.pop()
        # daca noudul a fost vizitat, se va continua iteratia; aceasta e cazul in care e pus de mai mul
        if node.getState() in reached:
            continue
        if problem.isGoalState(node.getState()):
            return construct_path(node)
        reached.append(node.getState())
        for (state, action, cost) in problem.expand(node.getState()):  # se cauta in lista de succesori
            # daca nodul nu a fost deja extras si expandat din frontiera
            if state not in reached:
                childNode = Node(state, node, action, node.getCost() + cost)
                frontier.update(childNode, heuristic(childNode.getState(), problem) + childNode.getCost
    return construct_path(node)
```

Listing 1: Solution for the A* algorithm.

Explanation:

- in cadrul acestui algoritm este nevoie de o structura de date speciala pentru frontiera. Aceasta structura va fi PriorityQueue, iar prioritatea nodurilor ce se vor afla in frontiera va fi data de costul drumului

pana in acel nod si de estimarea costului pana la scop(determinata prin intermediul unei euristici). La linia 11 este creata o astfel de structura pe care o gasim implementata in util

- pentru nod-uri s-a folosit aceeasi structura folosita si in cadrul solutiei pentru dfs si bfs
- ideea acestui algoritm este de a alege din frontiera nodul cel mai bun ca si candidat pentru solutia optima, nod ce va fi testat daca e scop si in caz afirmativ se va returna solutia sau in caz contrar ii vor expandati succesorii si adaugati in frontiera daca nu au fost expandati inainte.
- linia 29 este esentiala in cadrul acestei solutii, deoarece daca pentru un nod se gaseste un copil ce se afla in frontiera, prioritatea acetuia se va modifica daca functia de evaluare a costului in aceasta conjuctura e mai mica decat cea de dinainte. Aceasta va determina intotdeuana o solutie optima a algoritmului.
- pentru a construi calea de la nodul de start la nodul scop cu actiunile necesare se va apela functia **construct_path(node)**

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: Depinde de euristica folosita pentru estimarea costului pana la nodul scop. Daca euristica este constana 0, atunci A* star este identinc cu UCS, iar daca euristica e buna (admisibila si consistenta) A* ar trebui sa ofere aceeasi solutie ca UCS, solutie care e si optima de altfel.
Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2: Cu cat euristica se apropie de adevar, cu atat algoritmul va expanda mai putine noduri, deaorece el se va duce pe o abordare de tip Greedy. Deoarece A* are in logica sa si o abordare de tip Greedy, el va expanda mai putine noduri decat UCS.
Q3: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).
A3: Question q3: 4/4

2.1.3 Personal observations and notes

Pentru bigMaze costul total al caii este de 210, iar numarul de noduri explorate este de 549.Pentru mediumMaze costul total al caii este de 68, iar numarul de noduri explorate este de 221.S-a folosit pentru aceste teste euritstica *manhattan* Observam ca avem solutia optima pentru cele doua probleme, iar numarul de noduri expandate este mai putin decat la bfs.

2.2 Question 5 - Find all corners - problem implementation

In urmatoare subsectiune va fi prezentat exercitiul 3.3 din laborator al carui enunt e:

"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

In continuare se regaseste codul propus pentru implemetarea acestei clase ce descrie o noua problema. In incercarea de implementare si apoi testare a solutiei propuse s-au apelat mai multe comenzi din terminal pentru a vedea evolutia agentului pacman in procesul de cautare a solutiei sale

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 child function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height - 2, self.walls.width - 2
        self.corners = ((1, 1), (1, top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0  # DO NOT CHANGE; Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        """*** YOUR CODE HERE ***"""

        # initializam un dictionar ce contine informatia de corner; tine evidenta colturilor vizitate
        self.visitedCorners = {self.corners[0]: False,
                               self.corners[1]: False,
                               self.corners[2]: False,
                               self.corners[3]: False}

        # verificam pentru fiecare colt daca e pozitia de start sau e zid in locul respectiv si nu va pt
        for corner in self.corners:
            if self.startingPosition == corner or corner in self.walls:
                self.visitedCorners[corner] = True

        # setam pozitia de start ca o tupla ce contine pozitia de start si dictionarul cu ccolturile
        # initial am creat o clasa CornerState care imi descria structura pentru o stare din aceasta pr
        # apoi am vazut ca se doreste folosirea unei tuple, deoarece cand se prelua pozitia se folosea
        self.currentState = (self.startingPosition, self.visitedCorners)

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

    def isGoalState(self, state):
```

```
48          """
49          Returns whether this search state is a goal state of the problem.
50          """
51          "*** YOUR CODE HERE ***"
52          # se ia dictionarul cu colturile si se verifica daca toate colturile au fost vizitate
53          for values in state[1].values():
54              if not values:
55                  return False
56          return True
57
58      def expand(self, state):
59          """
60          Returns child states, the actions they require, and a cost of 1.
61
62           As noted in search.py:
63              For a given state, this should return a list of triples, (child,
64              action, stepCost), where 'child' is a child to the current
65              state, 'action' is the action required to get there, and 'stepCost'
66              is the incremental cost of expanding to that child
67          """
68
69          children = []
70          for action in self.getActions(state):
71              # Add a child state to the child list if the action is legal
72              # You should call getActions, getActionCost, and getNextState.
73              "*** YOUR CODE HERE ***"
74              nextState = self.getNextState(state, action)
75              cost = self.getActionCost(state, action, nextState)
76              children.append((nextState, action, cost))
77
78          self._expanded += 1  # DO NOT CHANGE
79          return children
80
81      def getActions(self, state):
82          possible_directions = [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
83          valid_actions_from_state = []
84          for action in possible_directions:
85              x, y = state[0]
86              dx, dy = Actions.directionToVector(action)
87              nextx, nexty = int(x + dx), int(y + dy)
88              if not self.walls[nextx][nexty]:
89                  valid_actions_from_state.append(action)
90          return valid_actions_from_state
91
92      def getActionCost(self, state, action, next_state):
93          assert next_state == self.getNextState(state, action), (
94              "Invalid next state passed to getActionCost().")
95          return 1
96
97      def getNextState(self, state, action):
98          assert action in self.getActions(state), (
99              "Invalid action passed to getActionCost().")
100         x, y = state[0]
101         dx, dy = Actions.directionToVector(action)
```

```
102         nextx, nexty = int(x + dx), int(y + dy)
103         "*** YOUR CODE HERE ***"
104         # se ia un dictionar pentru colturi si se actualizeaza
105         visitedCorners = {}
106         for corner in state[1]:
107             if corner[0] == nextx and corner[1] == nexty:
108                 visitedCorners[corner] = True
109             else:
110                 visitedCorners[corner] = state[1][corner]
111         # se va returna pozitia si dictionarul cu colturile vizitate
112         return (nextx, nexty), visitedCorners
113     def getCostOfActionSequence(self, actions):
114         """
115         Returns the cost of a particular sequence of actions.  If those actions
116         include an illegal move, return 999999.  This is implemented for you.
117         """
118         if actions == None: return 999999
119         x, y = self.startingPosition
120         for action in actions:
121             dx, dy = Actions.directionToVector(action)
122             x, y = int(x + dx), int(y + dy)
123             if self.walls[x][y]: return 999999
124         return len(actions)
```

Explanation:

- la linia 24 este declarat un atribut al *problemei* ce se refera la starea colturilor. Va fi declarat un dictionar de forma **colt:vizitat?**- unde prin *vizitat?* se intelege o valoare buleana (True sau False). Acest dictionar va contine informatia despre toate colturile.
- la liniile 30 -32 se verifica pentru fiecare colt daca e pozitia de start sau in acel loc e un zid si in caz afirmativ il vom marca vzitat in dictionarul atribut
- la linia 37 se va lua o tupla pentru pozitia de start, care va contine pozitia in grid si dictionarul cu informatia despre colturi
- metoda **getStartState** va returna tupla formata din starea initiala si starea colturilor la momentul initial
- scopul problemei de fata este de a vizita toate colturile. Se ajunge la scop in momentul in care toate valorile din dictionar sunt true. Metoda **isGoalstate(self,state)** va fi implementata printr-o simpla parcurgere a valorilor din dictionar si daca se intalneste o valoare de False va returna False, iar daca nu, va returna True
- liniile 58-79: metoda **expand** este similara cu cea de la *PositionSearchProblem*
- liniile 105-122 ceea ce lipseste in metoda **getnextState** este determinarea noului dictionar al colturilor asociat starii.

Commands:

- -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- -l mediumCorners -p SearchAgent -a fn=astar,heuristic=cornersHeuristic,prob=CornersProblem
- -l mediumCorners -p SearchAgent -a fn=astar,prob=CornersProblem

2.2.2 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: Bfs: 1966 noduri expandate; O euristica admisibila pentru aceasta problema poate fi nullEuristic. Pentru aceasta euristica numarul de noduri expandate a ramas 1966. Pentru cazul in care am apelat cu o eurirstica consistenta si admisibila (cornersHeuristic -implementatata in sectiunea urmatoare), numarul s-a redus la 783 de noduri expandate.

2.2.3 Personal observations and notes

Initial am creat o clasa **CornerState** care imi descria structura pentru o stare din aceasta problema apoi am vazut ca se doreste folosirea unei tuple, deoarece cand se prelua pozitia se dorea indexarea (state[0] -linia 85)

2.3 Question 6 - Find all corners - Heuristic definition

In sectiunea aceasta este explicat si rezolvat exercitiul 3.4 din laborator:

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

2.3.1 Code implementation

In continuare se regaseste codul propus pentru implemetarea acestei metode ce descrie o euristica consistenta pentru problema CornersProblem. In incercarea de implementare si apoi testare a solutiei propuse s-au apelat mai multe comenzi din terminal pentru a vedea evolutia agentului pacman in procesul de cautare a solutiei sale

Code:

```
1  def cornersHeuristic(state, problem):
2      """
3      A heuristic for the CornersProblem that you defined.
4
5        state:   The current search state
6                 (a data structure you chose in your search problem)
7
8        problem: The CornersProblem instance for this layout.
9
10     This function should always return a number that is a lower bound on the
11     shortest path from the state to a goal of the problem; i.e.  it should be
12     admissible (as well as consistent).
13     """
14     corners = problem.corners  # These are the corner coordinates
15     walls = problem.walls  # These are the walls of the maze, as a Grid (game.py)
16     "*** YOUR CODE HERE ***"
17     # se va construi o lista cu colturile neexplorate
18     unvisitedCorners = []
19     for corner in state[1].keys():
20         if not state[1][corner]:
21             unvisitedCorners.append(corner)
22     # goal test - se va returna 0 pentru aceasta heuristica
23     if not unvisitedCorners:
24         return 0
```

```
25
26      # se va calcula distanta Manhatten pana la coltul neexplorat cel mai apropiat si se va determina si
27      (closeCorner, dist1) = min([(corner, util.manhattanDistance(state[0], corner)) for corner in unvisit
28                                  key=lambda t: t[1])
29
30      # se calculeaza distanta de la coltul (cel mai apropiat de locatie) la coltul cel mai indepartat de
31      dist2 = max([util.manhattanDistance(closeCorner, corner) for corner in unvisitedCorners])
32
33      # euristica va fi suma dintre cele doua distante
34      h = dist1 + dist2
35      return h
```

Explanation:

- pentru ca o euristica sa fie consistenta si admisibila ea trbuie sa indeplineasca niste conditii:
  admisibilitate: h(n) <= h*(n) , unde h* este costul real pana la cel mai apropiat scop; in acest caz se va elimina supraestimarea costului
  consistenta: se poate determina cu regula triunghiului;
  - fie n si n' doua stari din spatiul starilor problemei.
  - h(n) - h(n') <= c(n,a,n');
  - h(n)-costul estimat din n pana la scop,
  - h(n')- costul estimatdin n' ,
  - iar c(n,a,n') - este costul real de tranzitie din starea n in n' prin actiunea a
- o euristica consistenta e si admisibila
- euristica pe care am propus-o se descrie astfel: Se calculeaza distanta din punctul curent la coltul cel mai apropiat de el, iar apoi se calculeaza distanta de la acel colt la coltul cel mai indepartat de acel colt. Prin distanta ma refer la distanta manhattan. Euristica va fi suma dintre cele doua distante calculate

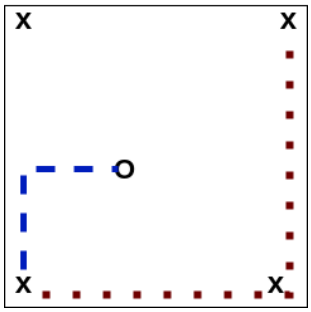

Figure 1: O posibila stare initiala

-
- ideea consta in faptul ca daca agentul alege sa mearga la cel mai apropiat colt de el, va suporta consecinta de a merge din acel colt la cel mai indepartat colt de coltul respectiv
- liniile 18-21: se extrag colturile nevizitate cand agentul e in starea *state*
- liniile 23-24: se rezolva situatia in care starea e scop si atunci euristica trbeuie sa returneze valoarea 0
- linia 27: calculare distanta cea mai scurta catre un colt cu manhattan facandu-se abstractie de pereti
- linia 31: calculare distanta cea mai lunga catre un colt din coltul determinat la instructiunea precedenta
- 34 : euristica va fi suma dintre aceste doua distante

Commands:

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

2.3.2 Questions

Q1: Test with on the mediumMaze layout. What is your number of expanded nodes?
A1: 783 numarul total de noduri expandate, iar costul e de 106

2.3.3 Personal observations and notes

2.4 Question 7 - Eat all food dots - Heuristic definition

In urmatoarele paragrafe va fi prezentata rezolvarea exercitiului 3.5 dib laborator:

"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

In continuare se regaseste codul propus pentru implemetarea acestei metode ce descrie o euristica consistenta pentru problema CornersProblem. In incercarea de implementare si apoi testare a solutiei propuse s-au apelat mai multe comenzi din terminal pentru a vedea evolutia agentului pacman in procesul de cautare a solutiei sale

Code:

```
1  def foodHeuristic(state, problem):
2      position, foodGrid = state
3      "*** YOUR CODE HERE ***"
4      foodList = foodGrid.asList()
5      if not foodList:
6          return 0
7
8      # ideea pentru heuristica: asemanatoare cu cea de la corner
9      # o sa gasesc calea cea mai scurta pana la mancare iar apoi de la acel punct o sa gasesc distanta m
10     (foodPos, distMin) = min([(food, util.manhattanDistance(position, food)) for food in foodList], key=
11     (foodMax, distMax) = max([(food, util.manhattanDistance(foodPos, food)) for food in foodList], key=l
12     h1 = distMin + distMax
13     return h1
```

Explanation:

- ideile prezentate in sectiunea anterioara sun valabile si pentru aceasta subsectiune
- ideea este asemnatoare cu cea de la cornerHeuristic
- la liniile 5-6: se rezolva cazul in care starea curenta e scop, caz in care euristica trebuie sa fie 0
- la linia 10: se calculeaza distanta de la agent pana la cea mai apropiata bucata de mancare, si se determina si pozitia acesteia
- la linia 11: se calculeaza distanta de la bucata de mancare determinata anterior pana la cea mai indepartata bucata de mancare fata de aceasta.
- euristica va fi data de suma acestor doua distante

- ideea de la care am plecat a fost ca daca agentul alege calea cea mai usoara (adica se indreapta spre cea mai apropiata bucata de mancare), el va suporta consecinta de a merge de la acea bucata de mancare pana la cea mai indeparatata bucata de mancare

Commands:

- -l testSearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic

2.4.2 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: Numarul de noduri expandate: 8178

2.5 Question 8 - Suboptimal Search

In aceasta sectiune este prezentata o metoda suboptimala:

"Implement the function findPathToClosestDot in searchAgents.py. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:"

2.5.1 Code implementation

In continuare se regaseste codul propus pentru implemetarea clasei **AnyFoodSearchProblem** care decrie o problema ce se poate rezolva suboptimal. In incercarea de implementare si apoi testare a solutiei propuse s-au apelat mai multe comenzi din terminal pentru a vedea evolutia agentului pacman in procesul de cautare a solutiei sale

Code:

```
class ClosestDotSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"

    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        while (currentState.getFood().count() > 0):
            nextPathSegment = self.findPathToClosestDot(currentState)  # The missing piece
            self.actions += nextPathSegment
            for action in nextPathSegment:
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception('findPathToClosestDot returned an illegal move: %s!\n%s' % t)
                currentState = currentState.generateChild(0, action)
        self.actionIndex = 0
        print('Path found with cost %d.' % len(self.actions))

    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
```

```
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        "*** YOUR CODE HERE ***"
        #se foloseste breadthFirstSearch pentru a lua cea mai scurta distanta
        #problema noua va a avea ca scop orice bucata de mancare pe care o va gasi
        actions = search.breadthFirstSearch(problem)
        return actions



class AnyFoodSearchProblem(PositionSearchProblem):
    """
    A search problem for finding a path to any food.

    This search problem is just like the PositionSearchProblem, but has a
    different goal test, which you need to fill in below.  The state space and
    child function do not need to be changed.

    The class definition above, AnyFoodSearchProblem(PositionSearchProblem),
    inherits the methods of the PositionSearchProblem.

    You can use this search problem to help you fill in the findPathToClosestDot
    method.
    """

    def __init__(self, gameState):
        "Stores information from the gameState.  You don't need to change this."
        # Store the food for later reference
        self.food = gameState.getFood()

        # Store info for the PositionSearchProblem (no need to change this)
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0  # DO NOT CHANGE

    def isGoalState(self, state):
        """
        The state is Pacman's position. Fill this in with a goal test that will
        complete the problem definition.
        """
        x, y = state
        "*** YOUR CODE HERE ***"
        if (x, y) in self.food.asList():
            return True
        return False
```

Explanation:

- ideea consta in faptul ca agentul mananca de fiecare data cea mai apropiata bucata de mancare, ceea ce ii confera o logica greedy de rezolvare a solutiei
- in aceste conditii, metoda **isGoalState(self, state)** a problemei *AnyFoodSearchProblem* este foarte simpla si se rezuma la verificarea daca starea curenta prezinta o pozitie cu mamncare sau nu. In cazul

in care e pozitie cu mancare se va returna True.

- in metoda **findPathToClosestDot**, determinarea actiunilor se va face cu ajutorul parcurgerii bfs pe problema *AnyFoodSearchProblem*, aceasta parcurgere radiala va duce la cea mai scurta cale spre scop

Commands:

- -l bigSearch -p ClosestDotSearchAgent -z .5

2.6 References

R. Stuart, N. Peter, Artificial Intelligence: A Modern Approach, 4th US ed., capitol 3, [online]
Curs Inteligenta Artficiala, Universitatea Tehnica din Cluj Napoca, furnizat: moodle.cs.utcluj.ro

3 Adversarial search

3.1 Question 8 - Improve the ReflexAgent

In aceasta sectiune va fi descris exercitiul 4.8 din laborator:

"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

In continuare se regaseste descrirerea perceptiei de moment a agentului. In incercarea de implementare si apoi testare a solutiei propuse s-au apelat mai multe comenzi din terminal pentru a vedea evolutia agentului pacman in procesul de cautare a solutiei sale

Code:

```
1   def evaluationFunction(self, currentGameState, action):
2       # Useful information you can extract from a GameState (pacman.py)
3       childGameState = currentGameState.getPacmanNextState(action)
4       newPos = childGameState.getPacmanPosition()
5       newFood = childGameState.getFood()
6       newGhostStates = childGameState.getGhostStates()
7       newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
8
9       "*** YOUR CODE HERE ***"
10      foodList = newFood.asList()
11      score = sum([1 / util.manhattanDistance(newPos, food) if foodList else 0 for food in newFood.asl
12
13      score += sum([util.manhattanDistance(newPos, ghost) / 100 if util.manhattanDistance(newPos,
14                                                                                          ghost) / 10
15                    for ghost in childGameState.getGhostPositions()])
16      score += sum([time / 100 if newScaredTimes else 0 for time in newScaredTimes])
17
18      return childGameState.getScore() + score
```

Explanation:

- am aplicat mai multe aporturi pentru perceptia de moment a agentului; in primul rand am luat in calcul cat de aproape se afla de mancare, cat de departe se afla de fantome si cate miscari sunt valabile in momentul in care vor fi inghetate fantomele
- distanta fata de mancare trebuie sa contribuie invers proportional la perceptia totala, deoarece cu cat mancarea e mai departe, cu atat nu ii convine agentului situatia. De aceea score-ului final o sa ii adun inversul distantelor fata de mancare. Cu cat distanta e mai mica, cu ata aportul la rezultatul final e mai mare. (linia 11)
- cu cat fantomele sunt mai departe, cu atat e mai bine agentului, de aceea o sa adun score-ului a suta parte din distantele fata de fantome (linia 13)
- numarul de miscari cand fantomele sunt inghetate va contribui cu a 100 parte la rezultatul final (linia 16)

Commands:

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

3.1.2 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: 9/10 meciuri castigate. O medie de aproximativ 1230 de puncte

3.1.3 Personal observations and notes

In cele 10 incercari, agentul nu a mancat acea bucata de mancare care ar fi inghetat fantoma

3.2 Question 9 - H-Minimax algorithm

Aceasta sectiune este dedicata exercitiului 4.9 din laborator si se cere:

" 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

In continuare este implementat si explicat algoritmul minimax. In incercarea de implementare si apoi testare a solutiei propuse s-au apelat mai multe comenzi din terminal pentru a vedea evolutia agentului pacman in procesul de cautare a solutiei sale

Code:

```
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """
    def getAction(self, gameState):
        "*** YOUR CODE HERE ***"
        (action, value) = self.h_minimax(gameState, 0, 0)
        return action

    # functie auxiliara care verifica daca se va incheia apelul lui h_minmax;
    # acesta se va termina in momentul in care se ajunge la o stare in care pacman castiga sau pacman p
    # la adancimea setata pana la care se face evaluarea
    def _cut_off_test(self, state, depth):
        if state.isWin() or state.isLose() or depth == self.depth:
            return True
        else:
            return False

    def h_minimax(self, state, agentIndex, depth):
        # adancimea pana la care se ajunge sa se evalueze depinde de cate
        # ori pacman (MAX) are de facut o decizie; deoarece sunt mai multe fantome
        # care vor fi evaluate secvential
       #Daca e randul lui MAX sa ia o dezcizie, atunci se incrementeaza adancimea doarece s-a ajuns
        # ca Max sa ia o noua decizie
        (agentIndex, depth) = ((0, depth + 1) if agentIndex == state.getNumAgents() else (agentIndex, de

        # se va evalua testul de cut_off
        if self._cut_off_test(state, depth):
```

```
29              return [], self.evaluationFunction(state)
30          # se va decide al carui agent este randul
31          # daca indexul este egal cu 0 atunci randul e a lui MAX si se va determina valoarea maxima
32          if agentIndex == 0:
33              return self.max_value(state, agentIndex, depth)
34          else:
35              # in caz contrar, e o fantoma si se calculeaza valoarea minima
36              return self.min_value(state, agentIndex, depth)
37
38      def max_value(self, state, agentIndex, depth):
39          legalActions = state.getLegalActions(agentIndex)
40          # daca nu sunt valori legale inseamna ca e punct terminal si se va termina si se va intoarce eva
41          # in acel punct
42          if not legalActions:
43              return [], self.evaluationFunction(state)
44
45          (action, value) = max(
46              [(action, self.h_minimax(state.getNextState(agentIndex, action), agentIndex + 1, depth)[1])
47               for action in legalActions], key=lambda t: t[1])
48          return action, value
49
50      def min_value(self, state, agentIndex, depth):
51          legalActions = state.getLegalActions(agentIndex)
52          # daca nu sunt valori legale inseamna ca e punct terminal si se va intoarce evaloarea in acel pu
53          if not legalActions:
54              return [], self.evaluationFunction(state)
55          # se va intoarece valoarea minima
56          (action, value) = min(
57              [(action, self.h_minimax(state.getNextState(agentIndex, action), agentIndex + 1, depth)[1])
58               for action in legalActions], key=lambda t: t[1])
59          return action, value
```

Explanation:

- acest algoritm este de tipul **adversial search** in care doi sau mai multi agenti au scopuri diferite si se formeaza un mediu competitiv; in cazul jocului pacman avem doua tipuri de agenti: pacman care este agentul MAX si fantomele care sunt agentii MIN.
- algoritmul de fata este insipirat din [1] si a fost adaptat pentru cazul in care sunt mai multi adversari, iar atunci exista mai multe nivele pentru MIN (fantome)
- o alta cosntrangere e adancimea pana la care sa se calculeze evaluarea starilor. Din punct de vedere al timpului de executie, este extrem de greu sa se parcurga intreg **game tree-ul**
- ideea consta in faptul ca valoarea returnata de **h_minimax** reprezinta utilitatea pentru agentul MAX de a fi in acea stare. MAX prefera o stare cu o valoare maxima, pe cand MIN prefera o stare cu valoare minima
- deoarece avem mai multi agenti de tip MIN, atunci trebuie sa se formeze mai multe nivele cu acestia; se considera actiunile fantomelor secvential
- algoritmul propus tine cont si de adancimea maxima la care se va parcurge game tree-ul si este reprezentata de variabila **depth**
- liniile 13-17: este implementata o functie care face testul de cut-off; algoritmul de cautare a unei stari optime se termina in momentul in care starea respectiva duce la pierderea sau castigul jocului sau se afla la adancimea limita pe care am impus-o pentru parcurgerea game tree-ului
- functia **h_minimax** va fi functia care se va apela de MAX si MIN. In momentul in care agentul este MAX (s-a ajuns la numarul maxim de agenti ai jocului- linia 23), atunci s-a terminat un nivel de

cautare si se incrementeaza variabila *depth*. In functie de randul agentului, se va merge pe una dintre cele doua ramuri de la liniile 32-36

- metoda **max_value** va fi apelata de MAX si ea va returna o actiune nula si evaluarea in starea respectiva daca nu exista alternative de plecare din starea aceea, sau va lua valoarea maxima a stariilor succesoare cu actiunea aferenta
- metoda **min_value** va fi apelata de MIN si ea va returna o actiune nula si evaluarea in starea respectiva daca nu exista alternative de plecare din starea aceea, sau va lua valoarea minima dintre starile succesoare cu actiunea aferenta
- s-a folosit list comprehension

Commands:

- -p MinimaxAgent -l minimaxClassic -a depth=4
- -p MinimaxAgent -l trappedClassic -a depth=3

3.2.2 Questions

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

A1: Deoarece sansele lui Pacman de a manca bucatile de mancare sunt foarte mici, el alege varianta de a muri cat mai repede, deoarece functia de evaluare a starilor tine cont si de scorul jocului. Astfel cu cat moare mai repede in asemnea situatie el va castiga mai multe puncte. De aceea se grabeste catre fantoma. Aici joaca un rol important adancimea cu care se apeleaza h_minimax.

3.3 Question 10 - Use $\alpha - \beta$ pruning in AlphaBetaAgent

Aceasta sectiune este dedicata exercitiului 4.11 din laborator si se cere:

*" Use alpha-beta prunning in **AlphaBetaAgent** from multiagents.py for a more efficient exploration of minimax tree.".*

3.3.1 Code implementation

In continuare este implementat si explicat algoritmul minimax cu $\alpha - \beta$ prunning. In incercarea de implementare si apoi testare a solutiei propuse s-au apelat mai multe comenzi din terminal pentru a vedea evolutia agentului pacman in procesul de cautare a solutiei sale

Code:

```
1  class AlphaBetaAgent(MultiAgentSearchAgent):
2      """
3      Your minimax agent with alpha-beta pruning (question 3)
4      """
5
6      def getAction(self, gameState):
7          """
8          Returns the minimax action using self.depth and self.evaluationFunction
9          """
10         "*** YOUR CODE HERE ***"
11
12         (action, value) = self.h_alpha_beta_prunning(gameState, 0, 0, -math.inf, math.inf)
13         return action
```

```
14
15      # functie auxiliara care verifica daca se va incheia apelul lui h_minmax;
16      # acesta se va termina in momentul in care se ajunge la o stare in care pacman castiga sau pacman p
17      # la adancimea setata pana la care se face evaluarea
18      def _cut_off_test(self, state, depth):
19          if state.isWin() or state.isLose() or depth == self.depth:
20              return True
21          else:
22              return False
23
24      def h_alpha_beta_prunning(self, state, agentIndex, depth, alpha, beta):
25          # adancimea pana la care se ajunge sa se evalueze depinde de cate
26          # ori pacman (MAX) are de facut o decizie; deoarece sunt mai multe fantome
27          # care vor fi evaluate secvential
28          #Daca e randul lui MAX sa ia o dezcizie, atunci se incrementeaza adancimea doarece s-a ajuns
29          # ca Max sa ia o noua decizie
30          (agentIndex, depth) = ((0, depth + 1) if agentIndex == state.getNumAgents() else (agentIndex, de
31
32
33          # se va evalua testul de cut_off
34          if self._cut_off_test(state, depth):
35              return [], self.evaluationFunction(state)
36          # se va decide al carui agent este randul
37          # daca indexul este egal cu 0 atunci randul e a lui MAX si se va determina valoarea maxima
38          if agentIndex == 0:
39              return self.max_value(state, agentIndex, depth, alpha, beta)
40          else:
41              # in caz contrar, e o fantoma si se calculeaza valoarea minima
42              return self.min_value(state, agentIndex, depth, alpha, beta)
43
44      def max_value(self, state, agentIndex, depth, alpha, beta):
45          legalActions = state.getLegalActions(agentIndex)
46          # daca nu sunt valori legale inseamna ca e punct terminal si se va termina si se va intoarce ev
47          # in acel punct
48          if not legalActions:
49              return [], self.evaluationFunction(state)
50          # initializam o valoare foarte mica
51          v = -math.inf
52          move = []
53          for action in legalActions:
54              (action2, value2) = self.h_alpha_beta_prunning(state.getNextState(agentIndex, action), agent
55                                                             depth, alpha, beta)
56              if value2 > v:
57                  v, move = value2, action
58                  alpha = max(alpha, v)
59              if v > beta:
60                  return move, v
61          return move, v
62
63      def min_value(self, state, agentIndex, depth, alpha, beta):
64          legalActions = state.getLegalActions(agentIndex)
65          # daca nu sunt valori legale inseamna ca e punct terminal si se va intoarce evaloarea in acel p
66          if not legalActions:
67              return [], self.evaluationFunction(state)
```

```
68
69          # se va initializa valoarea minima la infinit
70          v = math.inf
71          move = []
72          for action in legalActions:
73              action2, value2 = self.h_alpha_beta_prunning(state.getNextState(agentIndex, action), agentIn
74                                                           depth, alpha, beta)
75              if value2 < v:
76                  v, move = value2, action
77                  beta = min(beta, v)
78              if v < alpha:
79                  return move, v
80          return move, v
```

Explanation:

- acesta algoritm vine pentru a reduce numarul de stari explorate din arborele jocului;
  - α - reprezinta valoarea celei mai bune alegeri dintre punctele de decizie de-a lungul caii pentru MAX, iar
  - β -reprezinta valoarea cea mai buna (cea mai mica) a unei alegeri pe care o face MIN de-a lungul caii sale decizionale
- codul repectiv a fost inspirat din [1]
- prin aceasta tehnica o mare parte din arbore nu va fi explorata daca nu va influenta cu nimic iesirea
- terminarea algoritmului este data de aceeasi functie de cut_offtest, iar metoda **h_alpha_beta_prunning** este simimara cu **h_minmax** numai ca are doi parametri in plus (alfa si beta)
- in cadrul metodei **max_value** pe care o va executa agentul MAX, se va lua fiecare stare succesoare si se va calcula evaloarea acetei stari de decizie. Daca valoarea curent obtinua e mai buna decat cea pentru starea anterioara atunci se va actualiza starea in care va merge agentul, si se va actualiza si valoarea alpha (linia 58) daca valoarea obtinuta este mai buna decat cea pe care am avut-o inainte. Ideea de prunning o regasim la linia 59, unde daca valoarea obtinuta pentru o stare este mai mare decat beta, atunci nu mai are rost sa se continue cu expandarea succesorilor
- aceeasi logica este aplicata si pentru **min_value**

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: Question q3: 5/5

3.3.3 Personal observations and notes

3.4 References

[1]R. Stuart, N. Peter, Artificial Intelligence: A Modern Approach, 4th US ed., capitol 3, [online]
[2]Curs Inteligenta Artficiala, Universitatea Tehnica din Cluj Napoca, furnizat: moodle.cs.utcluj.ro

4 Personal contribution

4.1 Question 11 - Define and solve your own problem.

In aceasta sectiune va fi descris un algoritm de cautare pe care l-am propus:

" Algorimul propus pentru implementare este bidirectional search pe A , algoritm care in mod simultan face doua cautari: forward -de la starea initiala la scop si backward- de la scop la starea initiala. Speranta acestei abordari consta in coliziunea dintre cele doua cai, si apoi constructia caii finale. Principala motivatie pentru un astfel de algoritm este de a obtine un timp de cautare mai bun, astfel problema se poate imparti in 2, fiecare cu o complexitate de $b^{(d/2)}$. In consecinta, de la oproblema de complexitate $O(b^d$ vom ajunge la o complexitate de $O(\ b^{(d/2)} + b^{(d/2)}) = O(b^{(d/2)})$ ".*

4.1.1 Code implementation

In aceasta subsectiune se va explica implementarea algoritmului propus si a testelor ce au fost realizate pentru verificarea corectitudinii solutiei propuse.

Code:

```
# problema propusa este bidirectional best search
def bidirectional_best_first_search(problem, terminated):
    from util import PriorityQueue

    startNode = problem.getStartState()
    nodeF = Node(startNode, None, [], 0)
    # deoarece ne punem problema inversa, nodul de start pentru back este goal-ul problemei
    nodeB = Node(problem.goal, None, [], 0)
    # definim problema inversa pentru situatia in care mergem invers
    problem2 = copy.copy(problem)
    problem2.goal = problem.getStartState()

    frontierF = PriorityQueue()
    frontierB = PriorityQueue()
    # deoarece este implementata pentru ucs, vom alege ca si prioritate costul
    frontierF.push(nodeF, function(nodeF, problem))
    frontierB.push(nodeB, function(nodeB, problem2))

    # crearea unor dictionare state : node
    reachedF = {nodeF.getState(): nodeF}
    reachedB = {nodeB.getState(): nodeB}

    solution = Node('failure', None, [], math.inf)
    while (not frontierF.isEmpty()) and (not frontierB.isEmpty()) and not terminated(solution, frontierF
                                                                                      frontierB, problem
        # o metoda prin care sa extrag fara a sterge top-ul din cozile de prioritati
        top_f = frontierF.pop()
        frontierF.push(top_f, function(nodeF, problem))
        top_g = frontierB.pop()
        frontierB.push(top_g, function(nodeB, problem2))
        # se va decide pe care dintre probleme se va continua: forward sau backward in functie de evalu
        # respective
        if function(top_f, problem) < function(top_g, problem2):
            solution = proceed('F', problem, frontierF, reachedF, reachedB, solution)
        else:
```

```
36              solution = proceed('B', problem, frontierB, reachedB, reachedF, solution)
37      return solution
38
39
40  def join_nodes(nf, nb):
41      """Se va face joinul intre cele doua cai gasite; se va atasa caii nf, calea nb dar inversata"""
42      entirePath = nf
43      while nb.parent is not None:
44          cost = entirePath.getCost() + nb.getCost() - nb.parent.getCost()
45          if nb.getAction() == game.Directions.WEST:
46              action = game.Directions.EAST
47          elif nb.getAction() == game.Directions.EAST:
48              action = game.Directions.WEST
49          elif nb.getAction() == game.Directions.SOUTH:
50              action = game.Directions.NORTH
51          elif nb.getAction() == game.Directions.NORTH:
52              action = game.Directions.SOUTH
53          else:
54              action = nb.getAction()
55          entirePath = Node(nb.parent.getState(), entirePath, action, cost)
56          nb = nb.parent
57      return entirePath
58
59
60  def proceed(dir, problem, frontier, reached, reached2, solution):
61      """Aceasta metoda va determina succesorii nodului din frontiera"""
62      node = frontier.pop()
63      for (state, action, cost) in problem.expand(node.getState()):
64          childNode = Node(state, node, action, node.getCost() + cost)
65          if state not in reached or childNode.getCost() < reached[state].getCost():
66              frontier.push(childNode, function(childNode, problem))
67              reached[state] = childNode
68              # se verifica daca frontierele colizioneaza ceea ce va duce la o noua solutie si se va actu
69              # finala daca este mai buna cea nou obtinuta
70              if state in reached2:
71                  if dir == 'F':
72                      solution2 = join_nodes(childNode, reached2[state])
73                  else:
74                      solution2 = join_nodes(reached2[state], childNode)
75                  if solution2.getCost() < solution.getCost():
76                      solution = solution2
77      return solution
78
79  #se va decide cand se va termina iteratia din while
80  def terminated(solution, frontier_f, frontier_b, problem, problem2):
81      nodeF, nodeB = frontier_f.pop(), frontier_b.pop()
82      frontier_f.push(nodeF, function(nodeF, problem))
83      frontier_b.push(nodeB, function(nodeB, problem2))
84      return nodeF.getCost() + nodeB.getCost() > solution.getCost()
85
86
87  # evaluarea costului unui nod in functie de problema si euirstica
88  def function(node, problem, heuristic=util.manhattanDistance):
89      return node.getCost() + heuristic(node.getState(), problem.goal)
```

```
90
91
92  def a_star_bidirectional(problem_f):
93      solution = construct_path(bidirectional_best_first_search(problem_f, terminated))
94      print(solution)
95      return solution
```

Explanation:

- codul este dupa pseudocodul din [1]
- pentru implementarea acestui algoritm s-a folosit structura de date **PriorityQueue** din util pentru a delimita frontierele
- metoda **function** va calcula evaluarea unui nod in functie de costul caii din starea initiala la el si o estimare a costului pana la scop. Estimarea se va face printr-o euristica ce va fi ca mod default distanta manhattan
- metoda **bidirectional_best_first_search** va descrie logica de parcurgere a grafului. Pentru structura unui nod s-a folosit aceeasi clasa ca la celelalte solutii propuse anterior (bfs, dfs etc.).
  - la liniile 6 si 10 se vor crea nodurile de start pentru cele dua parcurgeri : forward si backward. Pentru nodul de start al parcurgerii backward se va lua scopul problemei
  - liniile 10-11: se creeaza o noua problema pentru backward in care se seteaza ca scopul starea initiala a problemei pentru forward
  - se creeza doua frontiere pentru cele 2 parcurgeri, si aceseat vor fi cozi de prioritati, iar prioritatea este data prin functia **function**
  - liniile 20-21: se creeaza doua dictionare de forma stare:nod (stare si nodul asociat ei). care vor fi populate cu nodurile explorate
  - la linia 23 se initializeaza o solutie care are ca si cost o valoare foarte mare
  - in bucla de parcuregere a grafului va fi conditionata de existanta de noduri in cele doua frontiere si de conditia de terminare data de metoda **terminated**. In cazul in care suma costurilor primelor noduri din cele doua frontiere e mai mare decat ceea ce avem in solutia curenta, atunci nu mai continuam parcurgerea grafului si intoarcem solutia.
  - linia 33: se evalueaza cele doua noduri fruntase din cele doua frontiere pentru a decide pe care dintre ele sa le expandam (sa le determinam succesorii). Expandarea succesorilor se va face in metoda **proceed** . Daca starea nodului succesor nu se afla in reached, sau daca pentru aceasta situatie costul de a ajunge in starea asta e mai mic decat costul pe care il aveam deja din alta cale, se va actualiza valoarea la lnia 67 (de aceea am ales implementarea lui **reached** cu dictionar). Aceasta ultima remarca va duce la gasirea caii optime in toate cazurile pentru acet algoritm. La linia 70 se va verifica coliziunea dintre parcurgerea forward si backward si in caz afirmativ se va face concatenarea celor doua cai (se tine cont de cadrul in care s-a realizat coliziunea prin *dir*)
- metoda **join_nodes** va concatena cele doua cai forward si backward. nf- va fi ultimul nod din calea de forward (calea se obtine prin relatia parinte) si nb - va fi ultimul nod din calea backward. Ideea consta in refacerea legaturilor parinte. Deoarece pentru parcurgerea backward avem cumva oglinda pentru cea forward, trebuie inversata si logica actiunilor (liniile 45-54).
- metoda **a_star_bidirectional** va apela *bidirectional_best_first_search* si prin apelul metodei construct_path (metoda prezentata la dfs) va construi lista de actiuni din relatiile copil parinte

Commands:

- -l tinyMaze -p SearchAgent -a fn=bis
- -l bigMaze -p SearchAgent -a fn=bis -z .5
- -l mediumMaze -p SearchAgent -a fn=bis -z .5

4.1.2 Personal observations and notes

Algoritmul este optimal deoarece tot timpul returneaza solutia de cost minim. mediumMaze: 68 costul, 250 noduri expandate; bigMaze: 210 costul, 594 noduri expandate ; tinyMaze: 8 costul, 16 noduri expandate

4.2 References

[1]R. Stuart, N. Peter, Artificial Intelligence: A Modern Approach, 4th US ed., capitol 3, [online]
[2] https://notebook.community/aimacode/aima-python/search4e