UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Muresan Ioana Danina si Radu Vlad
Grupa: 30236

Facultatea de Automatica
si Calculatoare

08 Decembrie 2022

Cuprins

1 Pac-Man

1.1 Introducere

Pac-man este un joc cu mai multe entitati: Pacman, fantome, food-dots si power-pellets. Jucatorul il controleaza pe Pacman cu scopul de a manca toate food-dots-urile. Pacman moare in cazul in care este mancat de o fantoma, dar daca Pacman mananca o power-pellet, acesta va avea o abilitate temporara de a manca fantome. Scopul nostru este de a construi agenti care sa il controleze pe Pacman si de a castiga. Actiunile posibile sunt Nord, Sud, Est, Vest, Stop, depinzand de prezenta peretilor. Cu fiecare pas, Pacman pierde 1 punct, pentru fiecare food-dot mancat primeste 10 puncte, iar pentru terminarea jocului primeste 500 puncte.

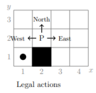


Figura 1: Legal actions

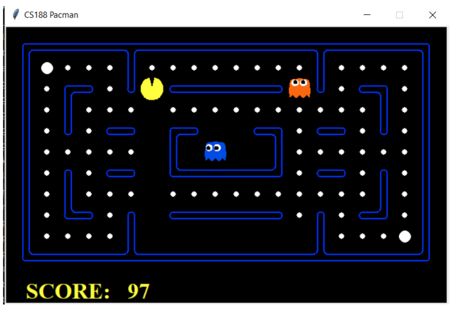


Figura 2: Pac-Man

2 Uninformed search

2.1 Question 1 - Depth-first search

2.1.1 Definirea cerintei

"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)."

2.1.2 Implementare

cod:

```
def depthFirstSearch(problem: SearchProblem):

    class SearchNode:
        """
            Creates node: <state, action, parent_node>
        """
        def __init__(self, state, action=None, parent=None):
            self.state = state
```

```python
            self.action = action
            self.parent = parent

        def extract_solution(self):
            """ Gets complete path from goal state to parent node """
            action_path = []
            search_node = self
            while search_node:
                if search_node.action:
                    action_path.append(search_node.action)
                search_node = search_node.parent
            return list(reversed(action_path))

    start_node = SearchNode(problem.getStartState())

    if problem.isGoalState(start_node.state):
        return start_node.extract_solution()

    frontier = util.Stack()
    explored = set()
    frontier.push(start_node)

    #rulez pana cand stiva este goala
    while not frontier.isEmpty():
        node = frontier.pop()  # aleg nodul din varful stivei
        explored.add(node.state) # il adaug in lista de noduri vizitate

        if problem.isGoalState(node.state):
            return node.extract_solution()

        # adaug in stiva nodurile vecine nevizitate
        successors = problem.getSuccessors(node.state)

        for succ in successors:
            # daca nodul nu a fost vizitat
            child_node = SearchNode(succ[0], succ[1], node)
            if child_node.state not in explored:
                frontier.push(child_node)

    # nu am gasit solutie
    util.raiseNotDefined()
```

2.1.3 Explicatie

• Se utilizeaza o stiva pentru a adauga starile si o cale de la pozitia de inceput catre acea stare. Am utilizat stiva pentru a le putea scoate din lista in ordinea inversa a adaugarii. Se expandeaza fiecare nod din stiva adaugand vecinii sai si se verifica daca este scop inainte de eliminare.

2.2 Question 2 - Breadth First Search

2.2.1 Definirea cerintei

In this section the solution for the following problem will be presented: ”In search.py, implement the Breadth-First search algorithm in function breadthFirstSearch.”

2.2.2 Implementare

cod:

```python
def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    class SearchNode:
        """
            Creates node: <state, action, parent_node>
        """
        def __init__(self, state, action=None, parent=None):
            self.state = state
            self.action = action
            self.parent = parent

        def extract_solution(self):
            """ Gets complete path from goal state to parent node """
            action_path = []
            search_node = self
            while search_node:
                if search_node.action:
                    action_path.append(search_node.action)
                search_node = search_node.parent
            return list(reversed(action_path))

        def is_in_frontier(self, data_structure):
            for n in data_structure.list:
                if n.state == self.state:
                    return True
            return False


    start_node = SearchNode(problem.getStartState())

    if problem.isGoalState(start_node.state):
        return start_node.extract_solution()

    frontier = util.Queue() #coada
    frontier.push(start_node)
    explored = set()

    while not frontier.isEmpty():
```

```
        node = frontier.pop()  # choose the shallowest node in frontier
        explored.add(node.state)

        if problem.isGoalState(node.state):
            return node.extract_solution()

        successors = problem.getSuccessors(node.state)
        for succ in successors:
            child_node = SearchNode(succ[0], succ[1], node)
            if child_node.state not in explored and\
                not child_node.is_in_frontier(frontier):
                frontier.push(child_node)

    # no solution
    util.raiseNotDefined()
```

2.2.3 Explicatie

- Implementarea pentru BFS este asemanatoare cu cea pentru DFS, diferenta fiind faptul ca la BFS se utilizeaza o coada in locul stivei, starile fiind expandate in ordinea in care au fost introduse on coada.

2.3 Question 3 - Varying the Cost Function

2.3.1 Definirea cerintei

In this section the solution for the following problem will be presented: ”In search.py, implement Uniform-cost graph search algorithm in uniformCostSearchfunction”

2.3.2 Implementare

cod:

```
def uniformCostSearch(problem: SearchProblem):

    class SearchNode:
        """
            Creates node: <state, action, cost, parent_node>
        """
        def __init__(self, state, action=None, path_cost = 0, parent=None):
            self.state = state
            self.action = action
            self.parent = parent
            # costul pana la nodul curent
            if parent:
                self.path_cost = path_cost + parent.path_cost
            else:
                self.path_cost = path_cost

        def extract_solution(self):
```

```python
            """ Gets complete path from goal state to parent node """
            action_path = []
            search_node = self
            while search_node:
                if search_node.action:
                    action_path.append(search_node.action)
                search_node = search_node.parent
            return list(reversed(action_path))

        def is_in_priority_queue(self, priority_queue):
            """ Check if the node is already in the priority queue """
            for index, (p, c, i) in enumerate(priority_queue.heap):
                if i.state == self.state:
                    return True
            else:
                return False

    start_node = SearchNode(problem.getStartState())

    if problem.isGoalState(start_node.state):
        return start_node.extract_solution()

    frontier = util.PriorityQueue()  # FIFO
    frontier.push(start_node, start_node.path_cost)
    explored = set()

    while not frontier.isEmpty():
        node = frontier.pop()  #aleg nodul cu costul minim


        if problem.isGoalState(node.state):
            return node.extract_solution()

        if node.state not in explored:
            explored.add(node.state)

            successors = problem.getSuccessors(node.state)

            for succ in successors:
                child_node = SearchNode(succ[0], succ[1], succ[2], node)
                frontier.update(child_node, child_node.path_cost)

    #nu e solutie
    util.raiseNotDefined()
```

2.3.3 Explicatie

• Algoritmul implementat este asemanator cu bfs, diferenta fiind ca UCS ia in calcul si costul drumului de la sursa la starea data. Este utilizata o coada de prioritati, nodurile fiind parcurse in ordine crescatoare a costului.

3 Informed search

3.1 Question 4 - A* search

3.1.1 Definirea cerintei

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".

3.1.2 Explicatie

• A* are o implementare asemanatoare cu cea a algoritmului BFS, diferentele fiind faptul ca in coada, pe langa pozitie si drumul spre acea pozitie din starea initiala, se mai adauga si valoarea unei functii, elementele scotandu se din coada in functie de valoarea acestui parametru. Functia se calculeaza adunand costul distantelor de la pozitia de start la pozitia respectiva si valoarea unei euristici in acel punct, aceasta reprezentand o aproximare a distantei fata de scop

3.2 Question 5 - Finding All the Corners

3.2.1 Definirea cerintei

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.".

3.2.2 Implementare

cod:

```python
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.
    You must select a suitable state space and 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 ***"
        return self.startingPosition,self.corners

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        position, corners = state
        if position in corners and len(corners) == 1:
            return True
        return False

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.
        As noted in search.py:
            For a given state, this should return a list of triples, (successor,
            action, stepCost), where 'successor' is a successor to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that successor
        """

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]

```

```
            "*** YOUR CODE HERE ***"
            pos, corners = state
            x,y = pos
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                if pos not in corners:
                    nextState = ((nextx, nexty), corners)
                else:
                    newcorners=[]
                    for corner in corners:
                        if corner != pos:
                            newcorners.append(corner)
                    nextState = ((nextx, nexty), tuple(newcorners))
                successors.append( ( nextState, action, 1) )


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

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999.  This is implemented for you.
        """
        if actions == None: return 999999
        x,y= self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)
```

3.2.3 Explicatie

- Aceasta cerinta presupune implementarea mai multor metode din clasa CornersProblem astfel incat a ajunge in toate cele 4 colturi.

3.3 Question 6 - Corners Problem: Heuristic

3.3.1 Definirea cerintei

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.".

3.3.2 Implementare

cod:

```
def cornersHeuristic(state, problem):

    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
    pos, corners = state
    x1,y1 = pos
    distances = []
    for corner in corners:
        x2,y2 = corner
        distance=abs(x1-x2)+abs(y1-y2)
        distances.append(distance)
    heuristic = max(distances)
    if problem.isGoalState(state):
        return 0
    return heuristic
```

3.3.3 Explicatie

• Euristica implementata este euristica Manhattan, Calculata prin adunarea valorilor absolute ale diferentelor coordonatelor starii si ale colturilor. Am ales aceasta euristica deoarece am expandat mai putine noduri decat daca foloseam euristica euclidiana.

3.4 Question 7 - Eating All The Dots

3.4.1 Definirea cerintei

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.”.

3.4.2 Implementare

cod:

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    foodlist = foodGrid.asList()
    distances =[]
    if problem.isGoalState(state):
        return 0
    for food in foodlist:
        newproblem = PositionSearchProblem(problem.startingGameState, start=position, goal=f
        distance = len(search.bfs(newproblem))
        distances.append(distance)
    heuristic = max(distances)
    return heuristic

```

3.4.3 Explicatie

• Algoritmul ales calculeaza mancarea cu distanta minima data de locul in care ne aflam su returneaza distanta de la coordonatele acesteia la pozitia a carei euristici vrem sa o determinam.

3.5 Question 8 - Suboptimal Search

3.5.1 Implementare

cod:

```

def findPathToClosestDot(self, gameState):

        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)


        return search.breadthFirstSearch(problem)
```

3.5.2 Explicatie

• Functia FindPathToClosestDot returneaza drumul pana la cea mai apropiata bucata de mancare, apeland functia de cautare BFS implementata la Q2.

4 Adversarial search

4.1 Definirea problemei

Pentru a dezvolta jocul Pac-man, ne-am propus sa il imbunatatim prin implementarea unor algoritmi multi-agent-search precum ReflexAgent.

Prin termenul multi-agent ne referim la un mediu competitiv in care actiunile agentilor sunt in conflict. In jocul nostru, mediul multi-agent este definit prin faptul ca fantomele din joc definesc fiecare cate un agent care isi "face planuri" impotriva agentului principal: Pacman. Fiecare agent isi alege actiunea curenta bazata pe propria perceptie, computand miscarea optima pana cand unul castiga.

Un reflex agent ia toate actiunile legale posibile, calculeaza scorul starilor accesibile cu aceste actiuni si selecteaza starile care rezulta intr-o stare cu scor maxim. In cazul in care mai multe stari au scor maxim, se alege random una dintre ele. Agentul inca pierde de multe ori. Adversarul agentului functioneaza pe principiul privirii inainte tinand cont de miscarile oponentului.

Intr-un mediu multi-agent precum cel specificat mai sus, putem folosi algoritmul de cautare Minimax, algoritm de cautare limitata in adancime. Plecand de la pozitia curenta, generam multimea de pozitii succesoare posibile. Un agent este numit MAX, iar celalalt MIN. Actiunile agentului MAX se adauga primele, apoi pentru fiecare stare rezultata se adauga actiunea MINurilor si asa mai departe. In acest scop, structura de date folosita este arborele. Se aplica functia de evaluare si se alege cea mai buna stare. Valoarea minimax asigura strategia optima pentru MAX.

4.2 Question1: Reflex Agent

Dupa cum s-a mentionat mai sus (v. Definirea problemei), un reflex agent ia toate actiunile legale posibile, calculeaza scorul starilor accesibile cu aceste actiuni si selecteaza starile care rezulta intr-o stare cu scor maxim. Pentru a imbunatati calitatile acestui agent in asa fel incat sa selecteze o actiune mai buna, am inclus in valoarea returnata de catre fiecare stare si locatia mancarii si cea a fantomelor. Astfel, acum luam in considerare distanta celui mai apropiat aliment fata de pozitia pe care o are Pacman in starea actuala, cat si pozitia fantomelor fata de pozitia lui Pacman. Toate aceste modificari au fost facute in cadrul fuctiei evaluationFunction din clasa ReflexAgent.

4.2.1 Testare

Testarea pe testClassic:

——————————-

python pacman.py -p ReflexAgent -l testClassic

Rezultatul obtinut:

Pacman emerges victorious! Score: 480

Average Score: 480.0

Scores: 480.0

Win Rate: 1/1 (1.00)

Record: Win

——————————-

Cu o singura fantoma:

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

Rezultatul obtinut:

Pacman emerges victorious! Score: 1066

4

Average Score: 1066.0

Scores: 1066.0

Win Rate: 1/1 (1.00)

Record: Win

——————————-

Cu doua fantome:

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

Rezultatul obtinut:

Pacman died! Score: 246

Average Score: 246.0

Scores: 246.0

Win Rate: 0/1 (0.00)

Record: Loss

——————————-

Deoarece este o functie de evaluare medie, agentul va pierde in majoritatea cazurilor in care exista doua fantome.

4.3 Question2: Minimax

4.3.1 Implementare

cod:

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState: GameState):

        numberOfGhosts = gameState.getNumAgents() - 1

        # Folosit doar pt agentul Pacman deoarece agentindex este mereu 0
        def maxLevel(gameState, depth):
            currDepth = depth + 1
            if gameState.isWin() or gameState.isLose() or currDepth == self.depth:  # Termin
                return self.evaluationFunction(gameState)
            maxvalue = -999999
            actions = gameState.getLegalActions(0)
            for action in actions:
                successor = gameState.generateSuccessor(0, action)
                maxvalue = max(maxvalue, minLevel(successor, currDepth, 1))
            return maxvalue

        # Pentru toate fantomele
        def minLevel(gameState, depth, agentIndex):
            minvalue = 999999
            if gameState.isWin() or gameState.isLose():  # Terminal Test
                return self.evaluationFunction(gameState)
            actions = gameState.getLegalActions(agentIndex)
            for action in actions:
                successor = gameState.generateSuccessor(agentIndex, action)
                if agentIndex == (gameState.getNumAgents() - 1):
                    minvalue = min(minvalue, maxLevel(successor, depth))
                else:
                    minvalue = min(minvalue, minLevel(successor, depth, agentIndex + 1))
            return minvalue

        # Root level action
        actions = gameState.getLegalActions(0)
        currentScore = -999999
        returnAction = ''
        for action in actions:
            nextState = gameState.generateSuccessor(0, action)
            # Urmatorul nivel este un nivel MIN => aplicam MIN pt succesorii radacinii
            score = minLevel(nextState, 0, 1)
            # Alege actiunea alegand Maximum dintre succesori
            if score > currentScore:
                returnAction = action
                currentScore = score
```

```
        return returnAction
```

4.3.2 Explicatie

- Acest Algoritm este folosit pentru cazul in care avem mai multi agenti inamici, unde, un agent este min, iar celalalt max. Pentru fiecare actiune a lui Max, min executa o actiune, functiile de min si max apelandu-se succesiv una pe cealalta.

4.4 Question3: Alpha-Beta Pruning

4.4.1 Implementare

cod:

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        "*** YOUR CODE HERE ***"
        # Used only for pacman agent hence agentindex is always 0.
        def maxLevel(gameState, depth, alpha, beta):
            currDepth = depth + 1
            if gameState.isWin() or gameState.isLose() or currDepth == self.depth:   # Termin
                return self.evaluationFunction(gameState)
            maxvalue = -999999
            actions = gameState.getLegalActions(0)
            alpha1 = alpha
            for action in actions:
                successor = gameState.generateSuccessor(0, action)
                maxvalue = max(maxvalue, minLevel(successor, currDepth, 1, alpha1, beta))
                if maxvalue > beta:
                    return maxvalue
                alpha1 = max(alpha1, maxvalue)
            return maxvalue

        # For all ghosts.
        def minLevel(gameState, depth, agentIndex, alpha, beta):
            minvalue = 999999
            if gameState.isWin() or gameState.isLose():   # Terminal Test
                return self.evaluationFunction(gameState)
            actions = gameState.getLegalActions(agentIndex)
            beta1 = beta
            for action in actions:
                successor = gameState.generateSuccessor(agentIndex, action)
```

```
                if agentIndex == (gameState.getNumAgents() - 1):
                    minvalue = min(minvalue, maxLevel(successor, depth, alpha, beta1))
                    if minvalue < alpha:
                        return minvalue
                    beta1 = min(beta1, minvalue)
                else:
                    minvalue = min(minvalue, minLevel(successor, depth, agentIndex + 1, alph
                    if minvalue < alpha:
                        return minvalue
                    beta1 = min(beta1, minvalue)
            return minvalue

        # Alpha-Beta Pruning
        actions = gameState.getLegalActions(0)
        currentScore = -999999
        returnAction = ''
        alpha = -999999
        beta = 999999
        for action in actions:
            nextState = gameState.generateSuccessor(0, action)
            # Next level is a min level. Hence calling min for successors of the root.
            score = minLevel(nextState, 0, 1, alpha, beta)
            # Choosing the action which is Maximum of the successors.
            if score > currentScore:
                returnAction = action
                currentScore = score
            # Updating alpha value at root.
            if score > beta:
                return returnAction
            alpha = max(alpha, score)
        return returnAction
```

4.4.2 Explicatie

• Functiile sunt asemanatoare cu cele implementate la MiniMax doar ca aici se folosesc doua variabile, alfa in care se salveaza maximul dintre valoarea curenta a lui alfa si maximul calculat in functia maxi si beta care este utilizata pentru a salva minimul dintre valoarea curenta a lui beta si minimul calculat in functia mini, atunci cand alfa este mai mic sau egal cu minimul. Variabilele alfa si beta au scopul de a limita numarul de stari ale jocului, alfa reprezentand cea mai buna alegere pentru max, iar bet, cea mai buna alegere pentru min.

5 Dezvoltare

Pentru a dezvolta proiectul,am ales sa implementam o functie noua de cautare,si anume **Iterative Deepening Search(IDS)**.

IDS (sau aprofundarea iterativă a căutării în profunzime) este o strategie generală, adesea folosită în combinație cu căutarea limitată în profunzime, care găsește cea mai bună limită de adâncime. Ea face acest lucru prin creșterea treptată a limitei - mai întâi 0, apoi 1, apoi 2 și așa mai departe - până când se găsește un obiectiv. Acest lucru se va întâmpla atunci când limita de adâncime ajunge la d, adâncimea celui mai puțin adânc nod țintă.

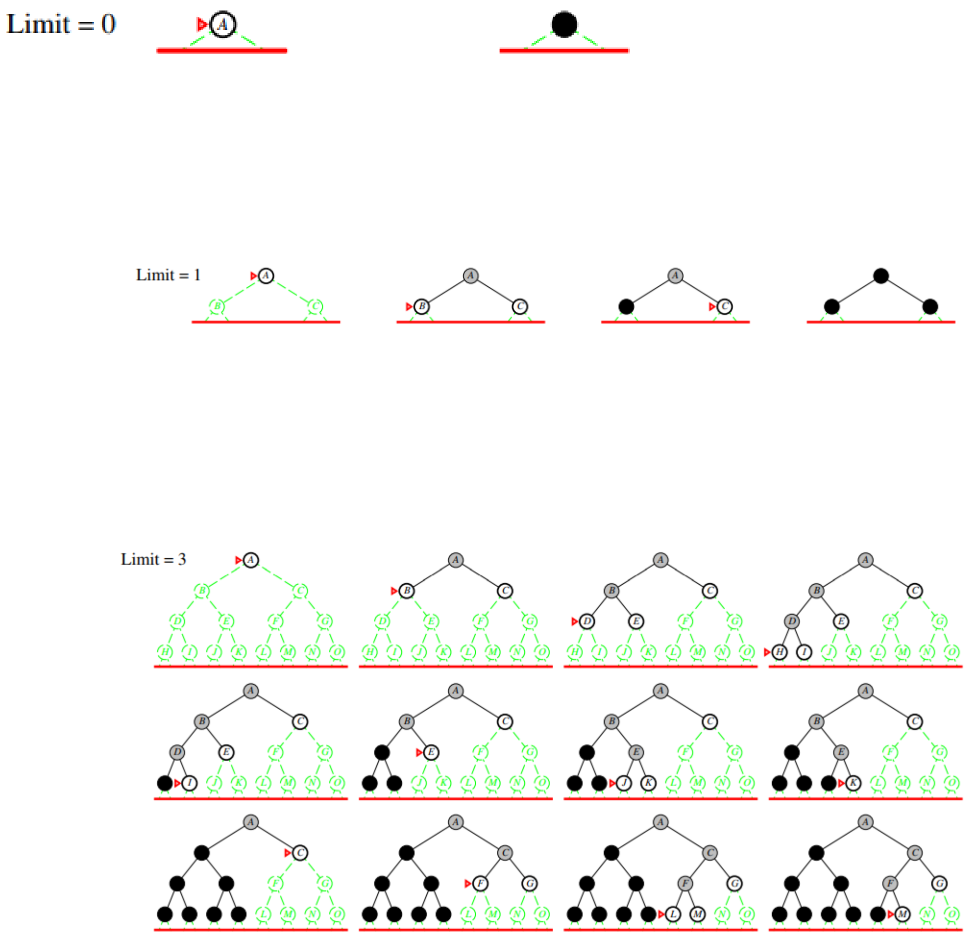


Figura 3: Limit

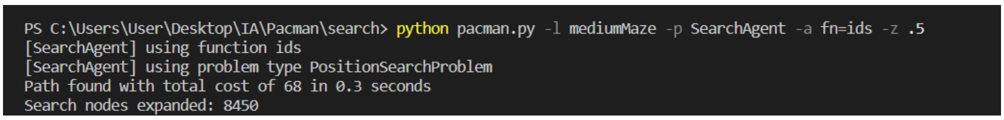
5.1 Testare

python pacman.py -l tinyMaze -p SearchAgent -a fn=ids

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

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

python pacman.py -l openMaze -p SearchAgent -a fn=ids -z .5



```
PS C:\Users\User\Desktop\IA\Pacman\search> python pacman.py -l mediumMaze -p SearchAgent -a fn=ids -z .5
[SearchAgent] using function ids
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.3 seconds
Search nodes expanded: 8450
```

Figura 4: Testare

Un aspect important al acestei cautari este faptul ca ajunge sa viziteze nodurile de la nivelurile mai inalte de mai multe ori. Desi acest lucru poate parea foarte costisitor, ın practica nu este chiar asa deoarece ıntr-un arbore, nodurile de la nivelurile de jos sunt cele mai numeroase. Asadar, faptul ca nodurile de la nivelurile mai inalte sunt vizitate de mai multe ori nu are un impact atat de mare. Nodurile de pe ultimul nivel sunt generate o singura data, cele de pe penultimul nivel sunt generate de 2 ori si asa mai departe pana la radacina. Se ajuge sa fie generate N = (d)b + (d — 1)b2 + . . . + (1)b noduri. Acest lucru duce la o complexitate de O(b), egala cu cea de la BFS.