UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: KIRALY NICOLE ELENA
Grupa: 30237

Facultatea de Automatica
si Calculatoare

30 Noiembrie 2023

Cuprins

1 Introducere

In prima parte a documentatiei ne vom focusa atentia spre algoritmii de cautare neinformati BFS, DFS si UCS, urmand ca mai apoi sa ne indreptam spre algoritmi informati cum ar fi A* care se foloseste de euristici in gasirea unei solutii cat mai optime. Vom analiza CornersProblem si vom gasi o cale cat mai rapida de a atinge toate cele 4 colturi ale maze-ului. In continuare vom incerca sa gasim o solutie folosind un numar minim de pasi pentru ca pacman sa manance toate bucatile de mancare, iar in final vom cauta o solutie de a manca cea mai apropiata bucata de mancare intai. In a doua parte a proiectului vom avea mai multi agenti in joc, lucrurile complicandu-se. Vom construi algoritmii minimax si alpha-beta pruning care se concentreaza pe miscarile din viitor.

2 Uninformed search

2.1 Question 1 - Depth-first search

In Q1, trebuie sa implementam o metoda de a gasi o bucata de mancare fixata, utilizand algoritmul de cautare depth-first search.

Pentru inceput am creat o clasa pentru node, unde am stocat toate datele importante de care ne vom folosi in continuare.

```
class Node:
    def __init__(self, state,action , cost = 0):
        self.state = state
        self.action = action
        self.cost = cost
```

Am implementat algoritmul de cautare DFS, bazat pe clasa Node, dupa cum urmeaza:

```
def depthFirstSearch(problem: SearchProblem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:"""

    # print("Start:", problem.getStartState())
    # print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    # print("Start's successors:", problem.getSuccessors(problem.getStartState()))

    "*** YOUR CODE HERE ***"

    node = Node(problem.getStartState(),[])
    # if problem.isGoalState(node.state):
    #     return []

    frontier = util.Stack()
```

```
    frontier.push(node)

    reached = set()

    while not frontier.isEmpty():
        node = frontier.pop()

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

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

        for child in problem.getSuccessors(node.state):
            child_node = Node(child[0],[],child[2])

            if child_node.state not in reached:
                child_node.action = node.action + [child[1]]
                frontier.push(child_node)
    return []
```

Algoritmul foloseste o stiva pentru frontiera si un set pentru a adauga pozitiile nodurilor vizitate. Cat timp frontiera nu e vida, vom extrage un nod din stiva, verificam daca e pozitia bucatii de mancare pe care o cautam(goalState), adaugam pozitia nodului in set, apoi parcurgem succesorii nodului, calculand pentru fiecare succesor, care nu a fost vizitat, actiunea de a ajunge in acel succesor, adaugandu-l apoi in frontiera. Cand bucata de mancare a fost gasita vom returna succesiunea de actiuni pentru a ajunge din starea pacmanului la acea bucata de mancare.

2.2 Question 2 - Breadth-first search

In Q2, trebuie implementat algoritmul de cautare BFS. Pe acelasi principiu ca si DFS, BFS se foloseste de clasa Node pentru a reprezenta datele importante din algoritm. Spre deosebire de DFS, aici vom folosi o coada ca frontiera, unde vom adauga nodurile pe masura ce le descoperim. Aici am folosit o lista pentru a stoca pozitiile nodurilor vizitate. Spre deosebire de DFS, pozitia nodului vizitat se adauga in lista inaintea adaugarii nodului in frontiera, in timp ce parcurgem succesorii pentru a a asigura vizitarea level by level, prevenind punerea in coada de mai multe ori a aceluiasi nod.

```
def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    # print("Start:", problem.getStartState())
    # print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    # print("Start's successors:", problem.getSuccessors(problem.getStartState()))

    node = Node(problem.getStartState(),[])
    # if problem.isGoalState(node.state):
    #     return []

```

```
    frontier = util.Queue()
    frontier.push(node)

    reached = list()
    reached.append(node.state)

    while not frontier.isEmpty():
        node = frontier.pop()

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


        for child in problem.getSuccessors(node.state):
            child_node = Node(child[0],[],child[2])

            if child_node.state not in reached:
                reached.append(child_node.state)
                child_node.action = node.action + [child[1]]
                frontier.push(child_node)

    return []
```

2.3 Question 3 - Uniform-cost search

In Q3 se adauga costul pentru fiecare actiune suplimentar. UCS, un caz particular al algoritmului Dijkstra, in care vom utiliza o coada de prioritati pentru frontiera, dupa costul total, de fiecare data se parcurge drumul cel mai scurt intai si un set pentru a adauga pozitiile nodurilor vizitate. Difera fata de BFS prin faptul ca se ia in calcul costul fiecarei actiuni.

```
def uniformCostSearch(problem: SearchProblem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    node = Node(problem.getStartState(),[],0)

    frontier = util.PriorityQueue()
    frontier.push(node, node.cost)

    reached = set()

    while not frontier.isEmpty():
        node = frontier.pop()

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

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

```

```
            for child in problem.getSuccessors(node.state):
                child_node=Node(child[0],[])

                if child_node.state not in reached:
                  child_node.action = node.action + [child[1]]
                  child_node.cost = node.cost + child[2]
                  #print(child_node.cost)
                  frontier.push(child_node, child_node.cost)

    return []
```

3 Informed search

Algoritmii prezentati in sectiunea anterioara sunt capabili sa gaseasca o solutie, dar o fac ineficient. In informed search se adauga euristicile pe care vom dezvolta algoritmii, gasind o solutie optima. Apar 2 proprietati noi si foarte importante, acestea fiind **admisibilitatea si consistenta**.

3.1 Question 4 - A* search algorithm

In Q4, trebuie implementata functia astarsearch, care reprezinta o imbunatatire a UCS-ului, calculand costul total pe baza unei euristici. Practic, in cazul A* costul total reprezinta suma dintre costul pana intr-un nod si cat mai avem din acel nod pana intr-un goalState(stiut de dinainte). A* este un algoritm mult mai eficient, acesta expandeaza mai putine noduri. Euristica aduce nou proprietatea de **admisibilitate**, care nu va supraestima goalState-ul, costul dintr-un anumit nod pana la goalState va fi intotdeauna mai mic decat costul normal, si proprietatea de **consistenta**, care evidentiaza reusitele lui Pitagora: **h(A) <= C(A,B) + h(B)**.

De asemenea, observam ca A* ponderat foloseste f(x) = g(x) + wh(x).Daca:
w = 0 –> f(x) = g(x) –> UCS
w = 1 –> f(x) = g(x) –> A*
w foarte mare –> f(x) = h(x) –> Greedy

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

    node = Node(problem.getStartState(), [], 0)

    frontier = util.PriorityQueue()
    frontier.push(node, node.cost)

    reached = list()

    while not frontier.isEmpty():
        node = frontier.pop()

        if problem.isGoalState(node.state):
            return node.action
```

```
17
18          if node.state not in reached:
19              reached.append(node.state)
20
21              for child in problem.getSuccessors(node.state):
22                  child_node = Node(child[0], [])
23
24                  if child_node.state not in reached:
25                      child_node.action = node.action + [child[1]]
26                      child_node.cost = node.cost + child[2]
27                      totalCost = child_node.cost + heuristic(child_node.state,problem)
28                      frontier.push(child_node, totalCost)
29
30      return []
```

3.2 Question 5 - Corners Problem: Representation

In continuare, vom trata o noua problema. In fiecare colt al maze-ului, avem o bucata de mancare. Problema se bazeaza pe gasirea celui mai scurt drum pentru a ajunge in toate cele 4 colturi ale maze-ului si mancarea celor 4 bucati de mancare din acestea.

```
1   def getStartState(self):
2           """
3           Returns the start state (in your state space, not the full Pacman state
4           space)
5           """
6           "*** YOUR CODE HERE ***"
7           #when you start, the tuple of corners is free
8           return (self.startingPosition,())
9           #util.raiseNotDefined()
10
11      def isGoalState(self, state: Any):
12          """
13          Returns whether this search state is a goal state of the problem.
14          """
15          "*** YOUR CODE HERE ***"
16          #if you visited 4 corners then you are done
17          return len(state[1]) == 4
18
19         # util.raiseNotDefined()
20
21      def getSuccessors(self, state: Any):
22          """
23          Returns successor states, the actions they require, and a cost of 1.
24
25           As noted in search.py:
26              For a given state, this should return a list of triples, (successor,
27              action, stepCost), where 'successor' is a successor to the current
28              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 ***"
            x,y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]

            if not hitsWall:
                visited = list(state[1])
                new_state = (nextx,nexty)

                if new_state in self.corners and new_state not in visited:
                    visited.append(new_state)

                successor = ((new_state, visited), action,1)
                successors.append(successor)


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

getStartStart este functia care returneaza o tupla cu pozitia pacmanului si o tupla vida deoarece nu am vizitat inca nici un colt al maze-ului.

isGoalState este functia care returneaza true sau false in functie de cate colturi am vizitat; daca lungimea este 4 am vizitat deci toate cele 4 colturi

getSuccessors returneaza o lista de stari ale succesorilor, actiunea si costul. Pozitia e reprezentata printr-o tupla (x,y), actiunea reprezinta N,S,E,V prin care iteram pentru a actualiza pozitia, daca nu e zid in acea pozitie. In visited, tinem evidenta colturilor vizitate.

3.3 Question 6 - Corners Problem: Heuristic

Pentru problema colturilor, vom folosi distanta Manhattan pentru a calcula cel mai scurt drum de la pozitie la un colt. Pentru aceasta, vom face o functie care calculeaza distanta intre pozitia pacmanului si colturile nevizitate si o va alege pe cea mai mica. Cat timp mai exista colturi nevizitate, extragem costul minim din acea functie si coltul unde ne vom muta pacmanul. Apoi vom scoate din lista coltul vizitat si vom continua procedeul pana cand nu vom mai avea colturi nevizitate, returnand costul total in a atinge toate cele 4 colturi.

```python
def cornersHeuristic(state: Any, problem: CornersProblem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    def min_distance(pos, unvisited):
        min_dist = 99999999999999

        for corner in unvisited:
            if util.manhattanDistance(pos, corner) < min_dist:
                min_dist = util.manhattanDistance(pos,corner)
                result_corner = corner

        return min_dist,result_corner

    "*** YOUR CODE HERE ***"
    pos = state[0]
    visited = list(state[1])
    unvisited = list()

    for corner in corners:
        if corner not in visited:
            unvisited.append(corner)

    totalCost = 0
    while unvisited:
        minCost, corner = min_distance(pos, unvisited)
        pos = corner
        totalCost += minCost
        unvisited.remove(corner)

    return totalCost

    return 0 # Default to trivial solution
```

3.4 Question 7 - Eating all the dots: Heuristic

Pentru Q7, trebuie sa gasim o solutie cat mai eficienta ca pacman sa manance toate bucatile de mancare in cat mai putini pasi, expandand cat mai putine noduri. Ne vom folosi de functia definita in proiect, mazeDistance care calculeaza distanta dintre 2 puncte bazandu-se pe algoritmii deja implementati. Pentru fiecare bucata de mancare, calculam distanta din punctul curent pana la aceasta si returnam maximul dintre aceste distante pentru a ne apropia cat mai mult de costul real al realizarii acestei actiuni si de a manca toate bucatile de mancare.

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    """
    Your heuristic for the FoodSearchProblem goes here.

    This heuristic must be consistent to ensure correctness.  First, try to come
    up with an admissible heuristic; almost all admissible heuristics will be
    consistent as well.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your heuristic is *not* consistent, and probably not admissible!  On the
    other hand, inadmissible or inconsistent heuristics may find optimal
    solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
    (see game.py) of either True or False. You can call foodGrid.asList() to get
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can query the
    problem.  For example, problem.walls gives you a Grid of where the walls
    are.

    If you want to *store* information to be reused in other calls to the
    heuristic, there is a dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls once and store that
    value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state
    "*** YOUR CODE HERE ***"

    foodCoordinates = foodGrid.asList()
    max_distance= -1

    if problem.isGoalState(state):
        return 0

    #we return the max distance as this is the most appropiate to the totalCost
    for food in foodCoordinates:
        distance = mazeDistance(position, food, problem.startingGameState)
```

```
41          if distance > max_distance:
42              max_distance = distance
43
44      return max_distance
```

3.5 Question 8 - Suboptimal Search

Pentru aceasta problema, vom utiliza un algoritm de cautare, care cauta cea mai apropiata bucata de mancare intai. Am utilizat algoritmul de cautare BFS deoarece cauta in nodurile apropiate(closest). GoalState-ul returneaza true sau false in functie de pozitia x, y a pacmanului daca se afla pe o pozitie unde se afla mancare.

```
1   def findPathToClosestDot(self, gameState: pacman.GameState):
2           """
3           Returns a path (a list of actions) to the closest dot, starting from
4           gameState.
5           """
6           # Here are some useful elements of the startState
7           startPosition = gameState.getPacmanPosition()
8           food = gameState.getFood()
9           walls = gameState.getWalls()
10          problem = AnyFoodSearchProblem(gameState)
11
12          "*** YOUR CODE HERE ***"
13          return search.bfs(problem)
14          #util.raiseNotDefined()
15
16  def isGoalState(self, state: Tuple[int, int]):
17          """
18          The state is Pacman's position. Fill this in with a goal test that will
19          complete the problem definition.
20          """
21          x,y = state
22
23          "*** YOUR CODE HERE ***"
24          return self.food[x][y]
```

4 Rezultate algoritmi

Maze	BFS	DFS	UCS	A*
small	92	59	92	53
medium	269	146	269	221
big	620	390	620	549

Tabela 1: Noduri expandate.

Dupa cum observam, pe un maze mic A* castiga din punct de vedere al nodurilor expandate, acesta gasind solutia cel mai rapid si cu un cost minim. In cazul maze-ului mediu, DFS

Maze	BFS	DFS	UCS	A*
small	19	49	19	19
medium	68	130	68	68
big	210	210	210	210

Tabela 2: Cost total.

expandeaza mai putine noduri decat A* deoarece acesta merge in adancime dar costul total in a gasi solutia este mai mic in cazul A*. Pe un maze mare, diferentele nu sunt atat de vizibile DFS expandand mai putine noduri dar costul ramane acelasi pe toti cei 4 algoritmi.

5 Adversarial search

5.1 Question 9 - Improve the ReflexAgent

In partea a doua a proiectului se adauga fantomele care vor complica putin jocul. Daca acestea ajung pe aceeasi pozitie cu a pacmanului jocul e pierdut. In functia evaluationFunction, verificam daca pozitile urmatoare ale fantomei coincid cu pozitia urmatoare a pacmanului, iar in caz afirmativ returnam un scor foarte mic (-500 pentru pierderea jocului). In continuare, iteram prin bucatile de mancare plasate in joc si calculam distantele manhattan intre noua pozitie a lui pacman si fiecare bucata de mancare. Inmultim cu -1 deoarece fiecare pas facut are un scor de -1. Deoarece distantele sunt negative, calculam cea mai apropiata distanta de 0, pe care o returnam

```
  def evaluationFunction(self, currentGameState: GameState, 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)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        food = currentGameState.getFood().asList()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

       # print(successorGameState)
        #print(newPos)
```

```
        # print(newFood.asList())
        # print(newGhostStates)


        "*** YOUR CODE HERE ***"
        for ghostState in newGhostStates:
            if ghostState.getPosition() == newPos:
                return -500

        min_dist = -99999999
        for f in food:
            #-1 for one move N,S,E,V,STOP
            #we return the min distance from pacman to food using manhattanDistance
            dist = util.manhattanDistance(newPos,f) * (-1)
            min_dist = max(min_dist,dist)


        return min_dist

        return successorGameState.getScore()
```

5.2 Question 10 - Minimax algorithm

Minimax este un algoritm care se foloseste la jocurile pe calculator cum ar fi sah, stiind de dinainte miscarile adversarului si incercand sa-l invinga. Anticipand miscarile, jucatorul foloseste maximizarea scorului, in timp ce adversarului trebuie minimizate miscarile. Pe acest principiu, pacman este cel care maximizeaza scorul, iar fantomele realizeaza minimul. Cand jocul este castigat, pierdut sau a ajuns la adancimea maxima, vom returna rezultatul. Pacman are mereu agentIndex 0, iar fantomele au agentIndex mai mare ca 0. Astfel cand agentIndex este 0, apelam recursiv minimax pe fantome, crescand adancimea si indexul pentru a continua cu o fantoma. Se realizeaza maximul scorului, daca adancimea e 0 se va returna actiunea pe care trebuie sa o faca pacman, altfel se va returna maximul scorului. In partea fantomei se realizeaza minimul scorului si se apeleaza recursiv minimax, crescand adancimea si indexul pentru urmatoarea fantoma sau pacman, totul depinde de cati agenti avem in joc, rezultat dedus din functia getNumAgents().

```
    def minimax(self, gameState, depth, agentIndex):
          if depth == self.depth *gameState.getNumAgents() or gameState.isWin()
           or gameState.isLose():
            return self.evaluationFunction(gameState)

          if agentIndex == 0:
            maxEval = float('-inf')
            direction = Directions.STOP
            for action in gameState.getLegalActions(0):
                eval = self.minimax(gameState.generateSuccessor(0, action), depth+1,
                 (agentIndex+1) % gameState.getNumAgents())
                if eval > maxEval:
                    maxEval = eval
                    direction = action
```

```
            if depth == 0:
                 return direction
            return maxEval
          else:
            minEval = float('inf')

            for action in gameState.getLegalActions(agentIndex):
                eval = self.minimax(gameState.generateSuccessor(agentIndex, action), depth+1
                (agentIndex+1) % gameState.getNumAgents())

                if eval < minEval:
                    minEval = eval
            return minEval
```

5.3 Question 11 - Alpha-Beta pruning

Alpha-beta pruning este o optimizare a algoritmului minimax, care evita calcularea unor ramuri de care nu mai avem nevoie. Folosind acest principiu, se adauga 2 variabile alpha si beta pe care le vom folosi in partea calcularii maximului, respectiv in partea calcularii minimului. Daca beta e mai mic ca alpha ne oprim, se face pruning.

```
def alphaBetaPruning(self, gameState, depth, agentIndex, alpha, beta):
        if depth == self.depth * gameState.getNumAgents() or
        gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)

        if agentIndex == 0:
            maxEval = float('-inf')
            direction = Directions.STOP
            for action in gameState.getLegalActions(0):
                eval = self.alphaBetaPruning(gameState.generateSuccessor(0, action),
                 depth+1, (agentIndex+1)%gameState.getNumAgents(),alpha,beta)
                if eval > maxEval:
                    maxEval = eval
                    direction = action
                alpha=max(alpha,eval)
                if beta < alpha:
                    break
            if depth == 0:
                return direction

            return maxEval
        else:
            minEval = float('inf')

            for action in gameState.getLegalActions(agentIndex):
                eval = self.alphaBetaPruning(gameState.generateSuccessor(agentIndex, action)
                 depth + 1, (agentIndex+1) % gameState.getNumAgents(), alpha,beta)

```

```
                if eval < minEval:
                    minEval = eval
                beta = min(beta,eval)
                if beta < alpha:
                    break
            return minEval
```

6 Punctaje obtinute

```
Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 5/4
Question q8: 3/3
------------------
Total: 26/25
```

Figura 1: Search

```
Provisional grades
==================
Question q1: 4/4
Question q2: 5/5
Question q3: 5/5
Question q4: 0/5
Question q5: 0/6
------------------
Total: 14/25
```

Figura 2: Multiagent

7 Probleme intampinate

Ca si probleme intampinate in dezvoltarea algoritmilor, putem mentiona actualizarea nodurilor ca visited, atentia sporita la rezultatele ce trebuie returnate pentru a rula testele, intelegerea intregii structuri a proiectului. Trebuie luat in considerare graful nu arborele, graful putand avea cicluri. Eficienta si optimizarile algoritmilor joaca un rol important.