UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Pescaru Silviu-Mihaita, Vlase Rafaella
Grupa: 30235

Facultatea de Automatica
si Calculatoare

17 Noiembrie 2022

Cuprins

1 Uninformed search

1.1 Question 1 - Depth-first search

Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

1.1.1 Prezentare algoritm

DFS este un algoritm de cautare neinformat, el cauta noduri **in adancime**, dupa cum ii spune si numele (**depth-first** search). Cautarea in adancime viziteaza fiecare nod chiar daca vecinii sai nu au fost inca vizitati. Verificam daca fiecare vecin a fost expandat si in caz contrar il expandam (ii vizitam vecinii). Ne vom folosi de un vector *visited* de noduri deja parcurse pentru a nu intra in bucla infinita. Cand se ajunge la solutie returnam un vector de directii de la start la goalState. Structura de date folosita de DFS este *stiva*.

1.1.2 Prezentare cod

```
def depthFirstSearch(problem: SearchProblem):
    start_state = problem.getStartState()
    current_state = start_state
    visited = set()
    stack = util.Stack()
    stack.push((start_state, []))

    while not stack.isEmpty():
        current_state, actions = stack.pop()

        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for successor, action, _ in successors:
                stack.push((successor, actions + [action]))
    return []
```

Comenzi utilizate:

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

1.1.3 Observatii

- **Performanta:** Algoritmul poate găsi soluția la o problemă, dacă există una, deoarece explorează întregul spațiu de căutare. Cu toate acestea, nu garantează găsirea celei mai optime soluții.

- **Optimalitate:** Algoritmul nu garantează găsirea celei mai optime soluții, deoarece explorează în profunzime înainte de a explora alte ramuri, iar soluția găsită prima dată nu este neapărat cea mai optimă.
- **Complexitate:** În cel mai rău caz, algoritmul poate explora fiecare nod în arborele de căutare până la adâncimea maximă, și poate exista un număr exponențial de noduri de explorat. Complexitatea temporală este, prin urmare, în $O(b^d)$, unde "b" reprezintă factorul de ramificare al arborelui (numărul maxim de succesori ai unui nod) și "d" reprezintă adâncimea maximă a arborelui.

1.2 Question 2 - Breadth-first search

Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

1.2.1 Prezentare algoritm

BFS (Breadth-First Search) este un algoritm neinformativ de căutare care explorează în lățime structuri de date de tip arbore sau graf. Pornind de la rădăcina arborelui sau de la un nod arbitrar dintr-un graf, BFS vizitează nodurile vecine ale acestuia înainte de a explora nodurile de pe nivelul următor. Folosește o coadă pentru a gestiona nodurile descoperite, asigurându-se că explorează nodurile pe niveluri și ajutând la identificarea celei mai scurte căi într-un graf, când toate muchiile au aceeași lungime.

1.2.2 Prezentare cod

```
def breadthFirstSearch(problem: SearchProblem):
    start_state = problem.getStartState()

    if problem.isGoalState(start_state):
        return []
    visited = set()
    queue = util.Queue()
    queue.push((start_state, []))

    while not queue.isEmpty():
        current_state, actions = queue.pop()
        if current_state not in visited:
            visited.add(current_state)

            if problem.isGoalState(current_state):
                return actions

            successors = problem.getSuccessors(current_state)
            for successor, action, _ in successors:
                if successor not in visited:
                    queue.push((successor, actions + [action]))

    return []
```

Comenzi utilizate:

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

1.2.3 Observatii

- **Performanta:** Algoritmul este complet și va găsi întotdeauna soluția optimă dacă aceasta există, deoarece explorează întregul spațiu de căutare în ordine de la nodurile cele mai apropiate la cele mai îndepărtate.
- **Optimalitate:** Algoritmul găsește întotdeauna soluția optimă în ceea ce privește numărul minim de acțiuni necesare pentru a ajunge la soluție.
- **Complexitate:** Complexitatea spațială este în general mai mare decât la căutarea în adâncime, deoarece trebuie să păstreze toți succesori la un nivel dat în coadă. Complexitatea spațială este în O(b^d), unde "b" este factorul de ramificare, iar "d" este adâncimea maximă a arborelui.Complexitatea temporală este și ea în O(b^d), unde "b" și "d" au aceleași semnificații ca și în cazul complexității spațiale.

1.3 Question3 - Uniform Cost Search

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider mediumDottedMaze and mediumScaryMaze. By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response. Implement the uniform-cost graph search algorithm in the uniformCostSearch function in search.py.

1.3.1 Prezentare algoritm

Algoritmul Uniform Cost Search (UCS) este un algoritm de căutare neinformată care explorează un graf ponderat pentru a găsi cea mai mică cale între un nod de start și un nod țintă. Acesta atribuie costuri reale fiecărui arc și extinde mereu nodul cu cel mai mic cost total până la acel moment. Uniform Cost Search este garantat să găsească cea mai mică cale între nodul de start și cel țintă.

1.3.2 Prezentare cod

```
1  def uniformCostSearch(problem):
2      """Search the node of least total cost first."""
3      start_state = problem.getStartState()
4      if problem.isGoalState(start_state):
5          return []
6
7      visited = set()
8      pqueue = util.PriorityQueue()
9      pqueue.push((start_state, []), 0)
10
11     while not pqueue.isEmpty():
12         current_state, actions = pqueue.pop()
13
```

```
        if current_state in visited:
            continue

        visited.add(current_state)

        if problem.isGoalState(current_state):
            return actions

        successors = problem.getSuccessors(current_state)
        for successor, action, _ in successors:
            if successor not in visited:
                new_actions = actions + [action]
                pqueue.push((successor, new_actions), problem.getCostOfActions(new_actions))

    return []
```

Comenzi utilizate:

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

1.3.3 Observatii

- **Performanta:** Algoritmul este complet pentru spațiile de căutare finite, asigurând găsirea soluției, dacă există una.
- **Optimalitate:** Uniform Cost Search (UCS) este optimal, deoarece explorează întotdeauna nodurile cu cel mai mic cost până când găsește soluția.
- **Complexitate:** Complexitatea spațială este determinată de numărul de noduri care trebuie memorate în coadă de priorități. În general, este O(b^d), unde "b" este factorul de ramificare, iar "d" este adâncimea maximă a arborelui. Complexitatea in timp este, de asemenea, în general O(b^d).
- **Alte observatii:** UCS este sensibil la costuri și explorează întotdeauna nodurile cu cele mai mici costuri până la momentul respectiv. Acest lucru îl face eficient în găsirea soluțiilor cu cost minim. Algoritmul memorează costul total al acțiunilor până la un anumit nod, ceea ce poate facilita găsirea soluției cu cel mai mic cost.

2 Informed search

2.1 Question 4 - A* search algorithm

Implement A graph search in the empty function aStarSearch in search.py. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function in search.py is a trivial example.*

2.1.1 Prezentare algoritm

Algoritmul A* search este un algoritm de cautare informat folosit pentru găsirea celei mai scurte căi între două noduri pe un graf cu ponderi. Algoritmul A* folosește o coadă de priorități

pentru a explora nodurile în ordinea estimării costului total. Alegerea nodului următor de explorat se face selectând nodul cu cel mai mic cost total estimat. Formula costului total utilizată este: f(n) = g(n) + h(n), unde:

- g(n) este costul real de la început până la nodul curent.
- h(n) este o estimare a costului de la nodul curent până la destinație (heuristică).
- f(n) este costul total estimat.

2.1.2 Prezentare cod

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    start_state = problem.getStartState()
    pqueue = util.PriorityQueue()
    visited = set()
    pqueue.push((start_state, [], 0), 0)

    while not pqueue.isEmpty():
        current_state, actions, cost = pqueue.pop()
        if current_state in visited:
            continue

        visited.add(current_state)

        if problem.isGoalState(current_state):
            return actions

        successors = problem.getSuccessors(current_state)

        for successor in successors:
            next_state, action, intermediate_cost = successor
            new_cost = cost + intermediate_cost
            new_state = (next_state, actions + [action], new_cost)
            priority = new_cost + heuristic(next_state, problem)
            pqueue.push(new_state, priority)
    return []
```

Comenzi utilizate:

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

2.1.3 Observatii

- **Performanta:** A* este complet pentru spațiile de căutare finite, cu condiția ca funcția euristică să fie admisibilă (nu supraestimează costurile). A* este mai eficient decât căutarea în adâncime sau căutarea în lățime, deoarece utilizează o combinație de cost real și estimări euristice pentru a ghida căutarea.
- **Optimalitate:** A* este un algoritm informat și, atunci când este folosit cu o funcție euristică admisibilă, garantează găsirea celei mai bune soluții în ceea ce privește costul.
- **Complexitate:** Complexitatea spațială depinde de numărul de noduri care trebuie să fie memorate în coadă de priorități. În general, este $O(b^d)$, unde "b" este factorul de

ramificare, iar "d" este adâncimea maximă a arborelui. Complexitatea temporală este în general O(b^d).

2.2 Question 5 - Finding All The Corners

Implement the CornersProblem search problem in searchAgents.py. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached.

2.2.1 Prezentare algoritm

Corners Problem are ca scop gasirea unui traseu care sa treaca prin toate cele patru colturi ale labirintului. Sunt utilizate mai multe metode:

- **getStartState** furnizează starea de start a problemei, reprezentată de poziția inițială a lui Pacman și un șir de biți pentru a urmări ce colțuri au fost vizitate
- **isGoalState** verifică dacă in starea curentă toate cele patru colțuri au fost vizitate
- **getSuccessors** furnizează succesorii, acțiunile (directiile) corespunzătoare și un cost de 1 pentru fiecare acțiune validă
- **getCostOfActions** calculează costul total al unui traseu

2.2.2 Prezentare cod

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.
    """

    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

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state space)
        """
        return (self.startingPosition, 0) # initial state and 0 as the bitmask

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
```

```
        """
        return state[1] == 0b1111  # all four corners are visited w 0b1111 as the bitmask

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.
        """
        successors = []
        position, visited_corners = state

        #exploring possible directions/actions

        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            x, y = position
            dx, dy = Actions.directionToVector(action)
            next_x, next_y = int(x + dx), int(y + dy)
            #if the next position is not a wall
            if not self.walls[next_x][next_y]:
                next_position = (next_x, next_y)
                new_visited_corners = visited_corners
                #if the next position is one of the corners
                for i, corner in enumerate(self.corners):
                    if next_position == corner:
                        new_visited_corners |= (1 << i)  # update the visited corners w bitu
                successors.append(((next_position, new_visited_corners), action, 1)) # updat

        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.
        """
        if actions is None:
            return 999999

        x, y = self.startingPosition
        cost = 0
        for action in actions:
            # Check the next state and whether it's legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
            cost += 1
        return cost
```

Comenzi utilizate:

- python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

2.2.3 Observatii

- **Optimalitate:** Algoritmul nu garantează optimizarea costului total al căii. Totuși, poate furniza soluții valide pentru problema specifică de a trece prin toate cele patru colțuri ale labirintului.
- **Complexitate:** Complexitatea este, in general, moderata.

2.3 Question 6 - Corners Problem: Heuristic

Implement a non-trivial, consistent heuristic for the CornersProblem in cornersHeuristic.

2.3.1 Prezentare algoritm

Functia CornersHeuristic propune o euristica pentru problema celor patru colturi (CornersProblem).

2.3.2 Prezentare cod

```
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).
    """

    #the sum of the minimum distances from the current position to each unvisited corner in

    corners = problem.corners  # These are the corner coordinates
    walls = problem.walls  # These are the walls of the maze, as a Grid (game.py)

    #  current state information
    position, visited_corners = state

    # list of unvisited corners - bitwise comparison with the visited_corners bitmask
    unvisited_corners = [corners[i] for i in range(4) if not visited_corners & (1 << i)]
    heuristic = 0

    # if unvisited corners, calculate heuristic
    while unvisited_corners:
```

```
        min_distance = float('inf')

        for corner in unvisited_corners:
            # Manhattan distance from current position to each unvisited corner
            distance = abs(position[0] - corner[0]) + abs(position[1] - corner[1])
            min_distance = min(min_distance, distance) #update the minimum distance

        heuristic += min_distance  # add minimum distance
        # update current position to the closest corner found
        position = min(unvisited_corners, key=lambda x: abs(position[0] - x[0]) + abs(positi
        unvisited_corners.remove(position)  # remove the visited corner from unvisited list

    return heuristic
```

Comenzi utilizate:

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

Note: AStarCornersAgent is a shortcut for

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

2.3.3 Observatii

- Euristica este admisibilă deoarece întotdeauna subestimează costul minim al drumului către un colț necercetat. Este și consistentă, deoarece valoarea euristicii pentru un nod este mai mică sau egală cu suma costului real al drumului de la nodul curent la un succesor și valoarea euristicii pentru acel succesor.

2.4 Question 7 - Eating All The Dots

If you have written your general search methods correctly, A with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to testSearch with no code change on your part (total cost of 7).*

Comenzi utilizate:

- python pacman.py -l testSearch -p AStarFoodSearchAgent

Note: Note: AStarFoodSearchAgent is a shortcut for

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

Fill in foodHeuristic in searchAgents.py with a consistent heuristic for the FoodSearchProblem. Try your agent on the trickySearch board.

2.4.1 Prezentare algoritm

FoodHeuristic implementeaza o euristica pentru FoodSearchProblem, cu scopul de a-l ghida pe Pacman sa gaseasca cea mai eficienta cale pentru a manca toata mancarea prezenta pe harta. Euristica implementată alege să calculeze distanța maximă de la poziția curentă a lui Pacman la cea mai îndepărtată bucățică de hrană rămasă. Acest lucru se realizează prin iterarea prin coordonatele hranei rămase (remaining) și calcularea distanței în labirint (prin funcția mazeDistance) între poziția curentă și fiecare dintre aceste bucăți de hrană. Euristica returnează apoi distanța maximă dintre aceste distanțe.

2.4.2 Prezentare cod

```python
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.

    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']
    """
    pacmanPosition, foodGrid = state
    remaining = foodGrid.asList() # remaining food coordinates

    # if no remaining food, heuristic = 0
    if not remaining:
        return 0

    # the distance to the nearest food
    distances_to_food = []
    for food_poz in remaining:
        distance = mazeDistance(pacmanPosition, food_poz, problem.startingGameState)
        distances_to_food.append(distance)

    return max(distances_to_food) # maximum distance as heuristic value
```

Comenzi utilizate:

- python pacman.py -l trickySearch -p AStarFoodSearchAgent

2.4.3 Observatii

- Euristica este admisibilă, deoarece întotdeauna returnează o valoare care este mai mică sau egală cu costul real al căii celei mai scurte de la starea curentă la o stare finală. Este și consistentă, deoarece valoarea euristicii pentru un nod este mai mică sau egală cu suma costului real al drumului de la nodul curent la un succesor și valoarea euristicii pentru acel succesor.

2.5 Question 8 - Suboptimal Search

In this section, you'll write an agent that always greedily eats the closest dot. ClosestDotSearchAgent is implemented for you in searchAgents.py, but it's missing a key function that finds a path to the closest dot. Implement the function findPathToClosestDot in searchAgents.py.

2.5.1 Prezentare algoritm

Functia findPathToClosestDot are ca scop gasirea drumului catre cel mai apropiat punct de mancare aflat pe harta.

2.5.2 Prezentare cod

```
    def findPathToClosestDot(self, gameState: pacman.GameState):
        startPosition = gameState.getPacmanPosition()
        problem = AnyFoodSearchProblem(gameState)

        # Perform the search to find the path to the closest dot
        "*** YOUR CODE HERE ***"
        # Create a search algorithm instance (e.g., BFS or DFS)
        search_algorithm = search.bfs

        # Find a path using the search algorithm and problem instance
        path = search_algorithm(problem)

        # Return the list of actions representing the path
        return path
```

Comenzi utilizate

- python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5