UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: Adriana Mihnea
Grupa: 30236

Facultatea de Automatica
si Calculatoare

3 Decembrie 2023

Cuprins

1 Descriere generală a proiectului

Acest proiect reprezintă o implementare a primelor două proiecte de inteligență artificială puse la dispoziție de UC Berkely (Search și Multiagent search).

In search.py se gasesc implementarile pentru intrebarile 1-4, in searchAgents.py se gasesc implementarile pentru intrebarile 5-8, iar in multiAgents.py se gasesc implementarile pentru intrebarile 1-3 de la al doilea proiect (intrebarile 2 si 3 de la al doilea proiect nu sunt implementate).

2 Uninformed search

2.1 Question 1 - Depth-first search

Pentru următorii algoritmi, nodurile vizitate vor fi ținute într-un set. Ca și structură de date va fi folosită o stivă deja definită in util.py. În stivă va fi reținută poziția lui Pacman de la un anumit moment și soluția până la acea poziție (de forma [South, North, ...])

Algoritmul adaugă nodul de start în stivă, împreună cu o listă goală de acțiuni (doarece din acel nod se pornește). Se va inițializa o listă goală solution, care va ajuta la construirea soluției unui succesor, sau va fi returnată în cazul în care stiva se golește. Atâta timp cât stiva nu este goală, se va scoate primul element din ea. Dacă poziția pe care se află Pacman coincide cu poziția pe care se află mâncarea, se va returna soluția de la start până la acea poziție (action). Altfel, în caz că poziția nu a fost deja vizitată, se va marca ca fiind vizitată. Se va itera lista de succesori a acestei poziții (pozițiile pe care Pacman poate face următoarea mișcare), iar pentru fiecare succesor va fi adăugată la acțiunea sa soluția prin care s-a ajuns până la el (acțiunea nodului părinte). În cele din urmă, în stivă se va adăuga poziția succesorului și noua soluție.

```
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:
    """
    visited = set()     #keep track of the visited nodes in the graph
    my_stack = util.Stack()         #initialise an empty stack
    start = problem.getStartState()
    #every node in the stack: position + solution until that position
    my_stack.push((start, []))

    solution = []       # solution will be something like: "North", "West",
    #... (for now it's empty cause we didn't go anywhere)
    while not my_stack.isEmpty():
        state, action = my_stack.pop()
        if(problem.isGoalState(state)):      #we arrived at the food
            return action
        if(state not in visited):
```

```
            visited.add(state)
            successors = problem.getSuccessors(state)
            for successor in successors:
                succ_state, succ_action, succ_cost = successor
                solution = action + [succ_action]
                my_stack.push((succ_state, solution))

    return solution
```

Probleme întâmpinate: În momentul când se dă push pe stivă, se dă push la o tuplă care conține state și action, nu la două obiecte diferite, iar mesajele de eroare nu au fost cele mai evidente atunci când vine vorba de numărul de paranteze de care are nevoie metoda pentru a continua. De asemenea, am marcat un anumit nod ca fiind vizitat prea repede la început, ceea ce a dus la crearea unui ciclu infinit.

2.2 Question 2 - Breadth-first search

Algoritmul este identic cu cel de depth first search, singura diferenta fiind structura de date folosita. Aici se foloseste o coada in loc de stiva, ceea ce face ca ordinea nodurilor sa fie luata diferit iar algoritmul sa mearga in adancime, prelucrand toti succesorii unui nod inainte de a trece la succesorii acestora

```
def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    visited = set()     #keep track of the visited nodes in the graph
    my_queue = util.Queue()         #initialise an empty queue
    start = problem.getStartState()
    #every node in the queue: position + solution until that position
    my_queue.push((start, []))

    solution = []       # solution will be something like: "North", "West",
    #... (for now it's empty cause we didn't go anywhere)
    while not my_queue.isEmpty():
        state, action = my_queue.pop()
        if(problem.isGoalState(state)):      #we arrived at the food
            # visited = set()
            return action
        if(state not in visited):
            visited.add(state)
            successors = problem.getSuccessors(state)
            for successor in successors:
                succ_state, succ_action, succ_cost = successor
                solution = action + [succ_action]
                my_queue.push((succ_state, solution))

    return solution
```

2.3 Question 3 - Uniform cost search

Acest algoritm ia in considerare si costul de a ajunge la o pozitie la alta, basically este un algoritm bfs care tine cont si de cost. In loc de o coada normala se foloseste o coada de prioritati, unde fiecare nod are asignat un anumit cost. In momentul cand se adauga primul nod in coada, costul este 0. Solutia va fi adaugata in coada in acelasi fel ca si pana acum, insa la fiecare pas se va calcula costul acestei solutii, care se va retine in coada alaturi de solutie. Problema alegerii costului minim este deja rezolvata datorita structurii de date folosite.

```
def uniformCostSearch(problem: SearchProblem):
    """Search the node of least total cost first."""
    visited = set()  # keep track of the visited nodes in the graph
    priority_queue = util.PriorityQueue()  # initialise an empty priority
    ↪  queue
    start = problem.getStartState()
    # every node in the priority queue: item(position + solution until that
    #position) & cost
    priority_queue.push((start, []), 0)

    solution = []  # solution will be something like: "North", "West", ... (for
    ↪
    #now it's empty cause we didn't go anywhere)
    while not priority_queue.isEmpty():
        (state, action) = priority_queue.pop()      #returns only the item
        ↪  without cost
        if (problem.isGoalState(state)):  # we arrived at the food
            return action
        if (state not in visited):
            visited.add(state)
            successors = problem.getSuccessors(state)
            for successor in successors:
                succ_state, succ_action, cost = successor
                solution = action + [succ_action]
                priority_queue.push((succ_state, solution),
                ↪  problem.getCostOfActions(solution))

    return solution
```

Probleme intampinate: La inceput am adaugat variabila cost in coada, insa acesta reprezinta doar costul asignat nodului succesor. Pentru ca algortmul sa functioneze corect, in coada trebuie sa se afle costul solutiei pana la acea positie, care se poate afla folosind metoda getCostOfActions().

3 Informed search

3.1 Question 4 - A* search algorithm

Acest algoritm este identic cu cele prezentate pana acum, insa in acest caz structura de date folosita este o coada de prioritati cu functie. Functia data este cea corespunzatoare cu formula

algoritmului: f(n) = g(n) + h(n). G(n) resprezinta costul actiunilor pana la pozitia curenta, iar h(n) reprezinta o euristica (o estimare a costului de la pozitia curenta pana la capat) Euristica este data ca parametru in metoda, default functionaza nullEuristic ce este de fapt un algoritm ucs (neluand in vedere estimarea de la nodul curent pana la final).

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic
    ↪  first."""
    visited = set()  # keep track of the visited nodes in the graph
    function_priority_queue = util.PriorityQueueWithFunction(lambda x:
    ↪  problem.getCostOfActions(x[1]) + heuristic(x[0], problem))
    start = problem.getStartState()
    # every node in the queue: position + solution until that position
    function_priority_queue.push((start, []))

    solution = []  # solution will be something like: "North", "West", ... (for
    ↪
    # now it's empty cause we didn't go anywhere)
    while not function_priority_queue.isEmpty():
        state, action = function_priority_queue.pop()
        if (problem.isGoalState(state)):  # we arrived at the food
            return action
        if (state not in visited):
            visited.add(state)
            successors = problem.getSuccessors(state)
            for successor in successors:
                succ_state, succ_action, succ_cost = successor
                solution = action + [succ_action]
                function_priority_queue.push((succ_state, solution))
    return solution
```

Probleme intampinate: Mi-a luat ceva sa inteleg cum ar trebui sa scriu functia din coada de prioritati. x reprezinta un nod din spatiul de cautare, care, la fel ca pana acum, este o tupla de forma (state, actions). x[0] reprezinta pozitia curenta (state), iar x[1] reprezinta solutia pana la pozitia curenta (actions). De aceea se calculeaza costul corespunzator acestei solutii si se aduna cu euristica ce ia ca parametru pozitia curenta.

3.2 Question 5 - Finding All the Corners

In aceasta problema, un state va fi definit de pozitia curenta a lui Pacman si de colturile care au ramas sa fie atinse. La inceput, getStartState() va contine toate colturile, pentru ca inca nu s-a ajuns in vreunul. Functia de getSuccessors() initializeaza pentru fiecare succesor o lista goala de colturi, itereaza lista anterioara de colturi, iar daca vreunul de ele este pozitia curenta, nu il mai adauga in noua lista de colturi, care va fi returnata ca fiind parte din state-ul succesorului. Functia de isGoalState() verifica momentul in care jocul este castigat, atunci cand lista de colturi este goala, adica s-au atins toate colturile.

```
def getStartState(self):
    """
```

```
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    # the state is defined by current pacman position + corners left to visit
    # at start, all corners are left to be visited
    remaining_corners = self.corners
    return (self.startingPosition, remaining_corners)
###
def isGoalState(self, state: Any):
    """
    Returns whether this search state is a goal state of the problem.
    """
    pacman_position, remaining_corners = state
    return len(remaining_corners) == 0
###
def getSuccessors(self, state: Any):
    """
    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 = []
    pacman_position, remaining_corners = state
    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 = pacman_position
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]

        if not hitsWall:
            next_position = (nextx, nexty)
            next_corners = tuple()

            # Populate next_corners using a loop
            for corner in remaining_corners:
                if corner != next_position:
                    next_corners = next_corners + (corner,)

            next_state = (next_position, next_corners)
            cost = 1
```

```
            successors.append((next_state, action, cost))

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

Probleme intampinate: La inceput, am incercat sa retin colturile intr-un dictionar si sa le marchez cu True in momentul in care unul dintre ele este atins. Am lasat comentate cateva dintre incercarile mele. Pe langa faptul ca m-am confruntat cu probleme legate de mutabilitatea dictionarului fiindca nu intotdeauna mi se seta coltul pe care voiam sa il setez, cand am rezolvat asta am realizat ca logica mea presupunea sa mai trec inca o data prin nodurile deja vizitate, ceea ce insemna fie sa resetez din exterior setul de visited, fie sa gasesc o modalitate sa apelez bfs de mai multe ori. Mi s-a parut prea complicat, insa am gasit aceasta idee de a include in state si colturile ramase neatinse, idee la care nu ma gandissem anterior. Aceasta idee rezolva problemele pe care le aveam folosind logica mea, folosind putine linii de cod.

3.3 Question 6 - Corners Problem: Heuristic

Aceasta euristica are un principiu simplu: calculeaza distanta Manhattan de la Pacman pana la fiecare colt nevizitat si o alege pe cea maxima. Pentru un maze de marime medie, algoritmul expandeaza 1136 de noduri si trece toate testele.

```
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)

    pacman_position, remaining_corners = state

    if not remaining_corners:
        return 0  # All corners have been visited

    # Initialize the largest distance to the first corner
    largest_distance = util.manhattanDistance(pacman_position,
    ↪  remaining_corners[0])

    # Calculate the largest Manhattan distance to any unvisited corner
    for corner in remaining_corners[1:]:
```

```
        distance = util.manhattanDistance(pacman_position, corner)
        if distance > largest_distance:
            largest_distance = distance

    return largest_distance
```

3.4 Question 7 - Eating All The Dots

Pentru aceasta euristica, am folosit logica de la intrebarea anterioara, inlocuind colturile cu pozitiile unde se afla mancarea. Metoda calculeaza distanta de la Pacman la fiecare food dot si o returneaza pe cea maxima. Pentru trickySearch, algoritmul expandeaza 9551 de noduri, trecand 3/4 teste.

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    position, foodGrid = state
    pacman_x, pacman_y = position

    # Get the list of food coordinates
    food_coordinates = foodGrid.asList()

    if not food_coordinates:
        # If there is no food left, the heuristic is 0
        return 0

    # Calculate the Manhattan distance to every food dot and find the maximum
    max_distance = max(
        util.manhattanDistance(position, food_pos)
        for food_pos in food_coordinates
    )

    return max_distance
```

3.5 Question 8 - Suboptimal Search

Dupa cum sugereaza cei de la Berkley, cea mai usoara modalitate de a rezolva aceasta problema este de a rezolva goal state-ul din problema any food search. Se verifica daca exista mancare pe pozitia pe care se afla Pacman, caz in care jocul s-a terminat. Drumul pana la cea mai apropiata mancare se gaseste apeland algoritmul A* pe aceasta problema.

```
class AnyFoodSearchProblem(PositionSearchProblem):
    def isGoalState(self, state: Tuple[int, int]):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x,y = state
    current_food = self.food
    return current_food[x][y]

```

```
class ClosestDotSearchAgent(SearchAgent):
    def findPathToClosestDot(self, gameState: pacman.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)

        path = search.aStarSearch(problem)
        return path
```

4 Adversarial search

4.1 Question 9 - Improve the ReflexAgent

Aceasta functie de evaluare ia in evidenta distanta pana la mancare si pana la fantome. Se calculeaza distanta pana la cea mai apropiata mancare si distanta pana la cea mai apropiata fantoma speriata (luand minimul dintre toate distantele). Se vor folosi reciprocele acestor distante, deoarece o distanta mai mica trebuie sa rezulte intr-un scor mai mare. Rezultatul final va fi scorul, la care se va aduna reciproca distantei pana la mancare si se va scadea reciptroca distantei pana la fantoma. Un lucru important de luat in vedere: distanta minima pana la mancare nu trebuie sa fie 0, altfel se va incerca impartirea la 0 iar jocul va da crash.

```
def evaluationFunction(self, currentGameState: GameState, action):
    # Useful information you can extract from a GameState (pacman.py)
    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

    # Calculate distances to food
    foodDistances = [util.manhattanDistance(newPos, food) for food in
    ↪  newFood.asList()]
    closestFoodDistance = min(foodDistances) if foodDistances else 1  # Avoid
    ↪  division by zero
    # without adding division by zero check, the game crashes

    # Calculate distances to scared ghosts
    scaredGhostDistances = [util.manhattanDistance(newPos, ghost.getPosition())
    ↪  for ghost in newGhostStates if
                            ghost.scaredTimer > 0]
    closestScaredGhostDistance = min(scaredGhostDistances) if
    ↪  scaredGhostDistances else float('inf')

```

```
19      # reciprocal = the smaller the distance -> the higher the value
20      # addition for food, substraction for ghosts
21      evaluationScore = successorGameState.getScore() + 1 / closestFoodDistance -
        ↪  1 / closestScaredGhostDistance
22
23      return evaluationScore
```

4.2 Question 10 - Minimax

Neimplementat.

4.3 Question 11 - Alpha-Beta Pruning

Neimplementat.

5 Bibliografie

- Introduction to Artificial Intelligence by Adrian Groza, Radu Razvan Slavescu and Anca Marginean (suportul de laborator)
- Proiect 1 de la Berkley: http://ai.berkeley.edu/search.html
- Proiect 2 de la Berkley: http://ai.berkeley.edu/multiagent.html
- https://rshcaroline.github.io/research/resources/pacman-report.pdf
- https://chat.openai.com/ for helping me fix errors in my code