UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: Marcu Ariana-Mălina
Grupa: 30232/2

Facultatea de Automatica
si Calculatoare

7 Decembrie 2023

Cuprins

1 Uninformed search **2**
1.1 Question 1 P1 - Depth-first search 2
1.2 Question 2 P1 - Breadth-first search 2
1.3 Question 3 P1 - Uniform-cost search 3

2 Informed search **4**
2.1 Question 4 P1 - A* search algorithm 4
2.2 Question 5 P1 - Finding All the Corners 4
2.3 Question 6 P1 - Corners Problem: Heuristic 5
2.4 Question 7 P1 - Eating All The Dots 7
2.5 Question 8 P1 - Suboptimal Search 7

3 Adversarial search **7**
3.1 Question 1 P3 - Improve the ReflexAgent 7

1 Uninformed search

1.1 Question 1 P1 - Depth-first search

Având de a face cu o problemă de căutare, primul algoritm folosit în aplicația noastră este "Căutarea în adâncime" - DFS, algoritm pe care l-am implementat la laborator și după care m-am ghidat și pentru următorii. Acesta are ca utilitate găsirea unui punct fix oarecare de către pacman. Căutarea în adâncime merge pe principiul expandării celui mai adânc nod neexpandat și se folosește de o stivă (last in first out) pentru a ține evidența frontierei și o listă pentru a stoca nodurile explorate. După ce se inițializează stiva și lista și se adaugă în listă starea inițială, căutarea continuă atâta timp cât există noduri în frontieră. Dacă starea curentă este GoalState, se returnează drumul până la această stare, altfel se adaugă la listă și se expandează succesorii.

În cel mai rău caz, DFS explorează toate nodurile arborelui de căutare, până își atinge obiectivul sau până tot arborele e explorat. Așadar, complexitatea în timp a acestui algoritm este $O(b^m)$, unde m este adâncimea maximă a oricărui nod și b este numărul maxim de succesori ai fiecărui nod. Complexitatea în spațiu e determinată de numărul maxim de noduri stocate în stivă, precum și de nodurile explorate, deci în worst case, vom avea $O(bm)$.

Este important de precizat că algoritmul este ineficient in cazuri anume, în special dacă arborele are o adâncime mare, deoarece explorează în adâncime, înainte să facă backtracking.

```
1  def depthFirstSearch(problem: SearchProblem):
2      frontier = util.Stack()
3      explored = []
4      frontier.push((problem.getStartState(), []))
5      while not frontier.isEmpty():
6          current, path = frontier.pop()
7          if problem.isGoalState(current):
8              return path
9          else:
10             explored.append(current)
11             for ns, a, _ in problem.getSuccessors(current):
12                 if ns not in explored:
13                     frontier.push((ns, path+[a]))
```

1.2 Question 2 P1 - Breadth-first search

Folosind aceeași structură ca la DFS, am implementat și algoritmul de căutare în lățime, care rezolvă de asemenea o problemă de căutare, din nou, evitând să expandeze nodurile deja vizitate.

Inițial, am creat o coadă goală pentru a ține nodurile ce urmau a fi explorate, pentru a păstra ordinea în care sunt adăugate. Bucla va rula atâta timp cât există noduri în coadă. Dacă starea curentă nu a fost explorată, se va adăuga în lista de noduri explorate. Se iterează prin toți succesorii stării curente și dacă cel nou nu e deja în coadă, i se va da push. Dacă nu se găsește nicio cale către obiectiv, se va returna o listă goală.

Ideea de BFS constă în explorarea nodurilor în ordinea lor de adăugare în coadă, asigurându-se că se explorează mai întâi nodurile mai apropiate de start înainte de a trece la cele mai îndepărtate.

În cel mai rău caz, BFS va explora fiecare nod până la adâncimea d, și în fiecare adâncime, va explora toți succesorii posibili, astfel complexitatea în timp fiind $O(b^d)$. Toate nodurile de

pe un anumit nivel trebuie păstrate în coadă înainte de a trece la nivelul următor, la un anumit moment fiind maxim b^d noduri în coadă. Complexitatea spațială este deci de asemenea $O(b^d)$.

```
def breadthFirstSearch(problem: SearchProblem):
    frontier = util.Queue()
    explored = []
    frontier.push((problem.getStartState(), []))
    while not frontier.isEmpty():
        current, path = frontier.pop()
        if current not in explored:
            explored.append(current)
            if problem.isGoalState(current):
                return path
            for ns, a, _ in problem.getSuccessors(current):
                if ns not in explored and ns not in [state for state, _ in frontier.list]:
                    frontier.push((ns, path+[a]))
    return []
```

1.3 Question 3 P1 - Uniform-cost search

Dacă BFS caută calea cea mai scurtă spre goal din punct de vedere al acțiunilor, dacă ne dorim să găsim calea potrivită și din alte puncte de vedere, putem folosi căutarea cu cost uniform, adică de exemplu, expandarea nodului cu cel mai mic cost. For-ul final iterează peste succesorii stării curente și pentru fiecare dintre cei neexplorați, se va calcula costul total al căii, tupla (succesor, cale, cost) fiind aici adăugată în coadă.

Se începe cu inițializarea unei cozi de priorități, a unui set care să țină stările care au fost deja explorate, iar starea inițială este așezată în coadă cu prioritatea zero. Asemenea celorlalți algoritmi, bucla principală se realizează până ce coada este goală, la fiecare iterație scoțând din coada de priorități, starea cu cel mai mic cost.

Completitudinea algoritmului este garantată dacă, costul fiecarei acțiuni depășește 0, iar complexitatea spațiului și a timpului poate fi mai mare decât cea a căutării în adâncime.

```
def uniformCostSearch(problem: SearchProblem):
    frontier = util.PriorityQueue()
    explored = set()
    frontier.push((problem.getStartState(), [], 0), 0)
    while not frontier.isEmpty():
        current, path, cost = frontier.pop()
        if current in explored:
            continue
        explored.add(current)
        if problem.isGoalState(current):
            return path
        for s, a, c in problem.getSuccessors(current):
            frontier.push((s, path+[a], cost+c), cost+c)
    return []
```

2 Informed search

2.1 Question 4 P1 - A* search algorithm

Trecând la strategiile de căutare informată, A* primește o funcție euristică, ca argument. Euristicile primesc două argumente: un stadiu în problema de căutare și problema în sine. Problema originală vine de la a găsi o cale printr-un labirint până la o poziție fixă, folosind euristica Manhatten implementată în **searchAgents.py**.

Metoda ce stă la baza acestui algoritm este evitarea expandării căilor care sunt deja cu cost mare și deci găsirea celui mai scurt drum dintre două puncte dintr-un graf ponderat. Scopul este de a minimiza funcția de cost total, care reprezintă suma dintre costul real al drumului parcurs până în acel moment și o estimare euristică a costului rămas până la final.

În implementare, am început cu un nod de start, pe care l-am adăugat în coada de priorități. Costul inițial este setat pe zero, iar funcția de evaluare este suma dintre costul și euristica nodului de start. Se continuă până când coada e goală sau nodul e atins. În fiecare iterație, la fel ca până acum, extrag nodul cu cea mai mică valoare a funcției de evaluare și se adaugă succesorii neexplorați în coadă, cu funcția de evaluare actualizată. După cum spune cursul, A* tree-search este optimal cu euristică admisibilă și este complet dacă nu există o infinitate de noduri. Complexitatea în timp este exponențială în cel mai rău caz, ceea ce înseamnă că poate deveni ineficient pentru probleme cu spații mari de căutare. Complexitatea în spațiu e dominată de stocarea în memorie a tuturor nodurilor.

```
1  def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
2      frontier = util.PriorityQueue()
3      explored = []
4      frontier.push((problem.getStartState(),[],0),heuristic(problem.getStartState(), problem)
5      while not frontier.isEmpty():
6          current, path, cost = frontier.pop()
7          if not current in explored:
8              explored.append(current)
9              if problem.isGoalState(current):
10                 return path
11             for s, a, c in problem.getSuccessors(current):
12                 frontier.push((s, path+[a], cost+c), cost+c+heuristic(s,problem))
13     return []
```

2.2 Question 5 P1 - Finding All the Corners

Această problemă de căutare găsește căile către toate cele 4 colțuti ale jocului. În primul rând avem constructorul clasei **CornersProblem**, care primește un obiect ce reprezintă starea de pornire a jocului, deja implementat. Apoi am modificat metoda getStartState, care returnează starea inițială a problemei de căutare, aici reprezentată de poziția inițială a lui pacman și o listă goală visited care va conține colțurile vizitate. Se ajunge la GoalState când toate cele 4 colțuri au fost vizitate, verificare realizată în următoarea funcție isGoalState.

În continuare, metoda getSuccessors, pentru fiecare direcție posibilă (nord, sud, est, vest), se verifică dacă următoarea poziție este în interiorul labirintului și nu conține un perete.Dacă următoarea poziție este un colț și nu a fost vizitat anterior, adaugă noul colț în lista de colțuri vizitate și construiește o nouă stare succesoară. Altfel, adaugă următoarea poziție și colțurile existente în lista de stări succesoare. În general, complexitatea temporală a metodei getSuccessors

este dominată de verificările pentru existența unui perete și de operațiile legate de gestionarea listei de colțuri vizitate, per total fiind O(1).

```
class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState: pacman.GameState):
        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

    def getStartState(self):
        visited = []
        return (self.startingPosition, visited)

    def isGoalState(self, state: Any):
        if len(state[1]) == 4:
            return len(state[1])

    def getSuccessors(self, state: Any):
        successors = []
        PosCurrent = state[0]
        corners = state[1]
        for action in [Directions.NORTH,Directions.SOUTH,Directions.EAST,Directions.WEST]:
            x,y = PosCurrent
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                if (nextx, nexty) in self.corners:
                    if (nextx, nexty) not in corners:
                        visited = corners + [(nextx, nexty)]
                        successors.append((((nextx, nexty), visited), action, 1))
                else:
                    successors.append((((nextx, nexty), corners), action, 1))
        self._expanded += 1
        return successors
```

2.3 Question 6 P1 - Corners Problem: Heuristic

În cazul acestei probleme, se poate observa mai jos o clasă mai lungă, formată din mai multe metode, dintre care, cea în care s-au realizat schimbările importante este **cornersHeuristic**. Clasa CornersProblem definește o problemă de căutare în care agentul trebuie să viziteze toate cele patru colțuri ale unei grile. Funcția euristică estimează costul de la starea curentă la o stare scop, calculând distanța Manhattan până la cel mai apropiat colț neexplorat și actualizând ite-

rativ poziția curentă. Euristicul este admisibil, deoarece nu supraestimează costul real. Această implementare este potrivită pentru algoritmi de căutare care utilizează euristici, cum ar fi în cazul nostru căutarea A*. Metoda isGoalState verifică dacă starea curentă este o stare scop. În acest caz, o stare este considerată scop dacă au fost vizitate toate cele patru colțuri. La această implementare, mi-am dat seama că funcția de la Q4 trebuie sa fie compatibilă și să funcționeze corect așa încât să pot rezolva și problema aceasta a colțurilor.

```
class CornersProblem(search.SearchProblem):
    def getStartState(self):
        visited = []
        return (self.startingPosition, visited)
    def isGoalState(self, state: Any):
        if len(state[1]) == 4: return len(state[1])
    def getSuccessors(self, state: Any):
        successors = []
        PosCurrent = state[0]
        corners = state[1]
        for action in [Directions.NORTH,Directions.SOUTH,Directions.EAST,Directions.WEST]:
            x,y = PosCurrent
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                if (nextx, nexty) in self.corners:
                    if (nextx, nexty) not in corners:
                        visited = corners + [(nextx, nexty)]
                        successors.append((((nextx, nexty), visited), action, 1))
                else:
                    successors.append((((nextx, nexty), corners), action, 1))
        self._expanded += 1
        return successors

def cornersHeuristic(state: Any, problem: CornersProblem):
    corners = problem.corners
    walls = problem.walls
    current = state[0]
    explored = state[1]
    h = 0
    notExplored = []
    for c in corners:
        if not c in explored:
            notExplored.append(c)
    while notExplored:
        distance, corner = min([(util.manhattanDistance(current, corner), corner)
                                for corner in notExplored])
        current = corner
        h = h + distance
        notExplored.remove(corner)
    return h
```

2.4 Question 7 P1 - Eating All The Dots

Acum vom rezolva o problemă mai dificilă de căutare: să fie mâncată toată hrana din jocul pacman în cât mai puține mișcări posibile. Pentru aceasta, vom avea nevoie de o nouă definiție a problemei de căutare care formalizează problema căutării hranei FoodSearchProblem. O soluție este definită ca o cale care colectează toată hrana din lumea pacman. Desface tupla state în două variabile separate, position și foodGrid. Se creează o listă "food" care conține toate pozițiile de hrană din labirintul foodGrid. Dacă nu este necesar să se facă nicio mișcare, am calculat distanța (heuristică) de la poziția curentă (position) la fiecare poziție de hrană (f) folosind funcția mazeDistance. Aceste distanțe sunt stocate în lista distances.Voi returna distanța maximă dintre poziția curentă și oricare dintre pozițiile de hrană, reprezentând astfel o euristică pentru cât de departe este cea mai îndepărtată hrană.

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem:FoodSearchProblem):
    position, foodGrid = state
    food = foodGrid.asList()
    if not food: return 0
    distances = [mazeDistance(position, f, problem.startingGameState) for f in food]
    return max(distances)
```

2.5 Question 8 P1 - Suboptimal Search

Ultima, dar nu cea din urmă din această categorie, o reprezintă căutarea suboptimală. Uneori, chiar și cu algoritmul A* și o euristică bună, găsirea căii optime prin toate punctele este dificilă. În aceste cazuri, ne-am dori totuși să găsim rapid o cale destul de bună. În această secțiune, se va descrie un agent care mănâncă întotdeauna punctul cel mai apropiat. ClosestDotSearchAgent este implementat în searchAgents.py, dar lipsește funcția cheie care găsește o cale către cel mai apropiat punct. După ce am completat metoda isGoalState cu linia de cod "return self.food[x][y]", am completat clasa cu continuarea metodei **findPathToClosestDot**. Acest cod implementează funcția care are ca scop să returneze o cale (o listă de acțiuni) către punctul cel mai apropiat, începând de la starea jocului. Am folosit funcția astar pentru a găsi calea către cea mai apropiată mâncare.

```
    def findPathToClosestDot(self, gameState: pacman.GameState):
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)
        path = search.aStarSearch(problem)
        return path
```

3 Adversarial search

3.1 Question 1 P3 - Improve the ReflexAgent

În această parte a proiectului, se îmbunătățește în primul rând clasa **ReflexAgent** din multiAgents.py. Un agent reflex alege de fapt o acțiune la fiecare punct de decizie, prin examinarea alternativelor sale realizate printr-o funcție de evaluare a stării. Funcția primește "the current and proposed successor GameStates" și returnează un număr, unde numerele mai mari sunt cele bune. Pentru început, codul de mai jos extrage informații utile din stări, cum ar fi mâncarea

rămasă(mewFood) sau poziția lui pacman dupa ce se mișcă(newPos). Scopulfuncției este să atribuie un scor unei stări de joc particulare, indicând cât de bună sau rea este aceea pentru pacman. Prin **arr**, se convertește distribuția rămasă a hranei într-o listă pentru a itera mai ușor.

Prima buclă iterează prin fiecare poziție a hranei, iar pentru fiecare distanță Manhattan nenulă între noua poziție a lui pacman si o bucată de mâncare, adaugă un scor în plus. A doua buclă iterează prin stările fiecărei fantome, iar scorul fiind invers proporțional cu distanța Manhatten, înseamnă că fantomele mai apropiate contribuie la mai mult scor. Complexitatea de timp a acestei funcții este în general O(N), unde N este numărul de bucați de mâncare și numărul de fantome în starea succesoare. Această funcție de evaluare nu oferă întotdeauna o soluție optimă, dar este concepută pentru a fi eficientă în timp și spațiu, adesea facilitând procesul de luare a deciziilor în timp real.

```
class ReflexAgent(Agent):

    def getAction(self, gameState: GameState):
        legalMoves = gameState.getLegalActions()
        scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
        chosenIndex = random.choice(bestIndices)
        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState: GameState, action):
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        arr = newFood.asList()
        scor = successorGameState.getScore()
        for i in arr:
            if (util.manhattanDistance(i, newPos)) != 0:
                scor += (1.0/util.manhattanDistance(i, newPos))
        for ghost in newGhostStates:
            pos = ghost.getPosition()
            g = util.manhattanDistance(pos, newPos)
            if (manhattanDistance(newPos, pos)) > 1:
                scor += (1.0/g)
```