UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: Miruna-Roxana Bacisor
Grupa: 30236

Facultatea de Automatica
si Calculatoare

1 Decembrie 2024

Cuprins

1 Tutorial

1.1 Scurta istorie a jocului

Pac-Man (cunoscut în Japonia inițial sub numele de Puck Man) este un joc video de acțiune tip labirint lansat în 1980. Jucătorul îl controlează pe Pac-Man, care trebuie să consume toate punctele dintr-un labirint închis, evitând în același timp fantomele colorate. Consumul punctelor mari intermitente, numite „Peleți de putere", transformă temporar fantomele în adversari albi, pe care Pac-Man îi poate mânca pentru a câștiga puncte bonus.

1.1.1 Pacman si Inteligenta Artificiala

În cadrul acestui proiect, am implementat algoritmi clasici de căutare și euristici pentru a rezolva probleme complexe în jocul Pac-Man. Algoritmii de căutare reprezintă o componentă fundamentală în Inteligența Artificială, fiind utilizați pentru rezolvarea problemelor care implică explorarea spațiu de stări posibile.

În contextul jocului Pacman:

- **Spațiul de stări** este reprezentat de poziția lui Pac-Man și configurația labirintului (colțuri, mâncare, fantome, pereți).
- **Obiectivul** variaza de la colectarea tuturor punctelor de mâncare la atingerea colțurilor specificate în cel mai scurt timp sau atingerea unor pereti specificati.

1.2 Tutorial elemente de baza

Pe scurt, **pasii** pe care trebuie sa ii urmam pentru a rula jocul fara strategiile implementate sunt:

1. Deschidem proiectul în **PyCharm** și rulăm fișierul `pacman.py`.
   - Navigăm în directorul proiectului și localizăm fișierul `pacman.py`.
   - Apăsăm butonul `Run` pentru a rula programul.

2. Din combinațiile de taste **Sus-Jos-Stânga-Dreapta**, ghidăm Pac-Man prin labirint.
   - **Sus**: Navighează în sus pe harta.
   - **Jos**: Coboară pe hartă.
   - **Stânga**: Se deplasează spre stânga.
   - **Dreapta**: Avansează spre dreapta.

3. Jocul se încheie cu succes atunci când Pac-Man a reușit să culeagă toată mâncarea și nu a fost prins de fantome.
   - Dacă toate punctele de mâncare au fost colectate, jocul afișează mesajul de victorie.
   - Dacă Pacman este prins de fantome, jocul afișează mesajul de înfrângere.
   - Scorul este afișat la final, masurandu-se astfel performanta jocului.

Algoritmi implementati:

- Depth First Search (DFS)
- Breath First Search (BFS)
- Uniform Cost Search (UCS)
- AStar Search (A*)

Mai jos se poate vizualiza cum arata jocul in momentul rularii 1:

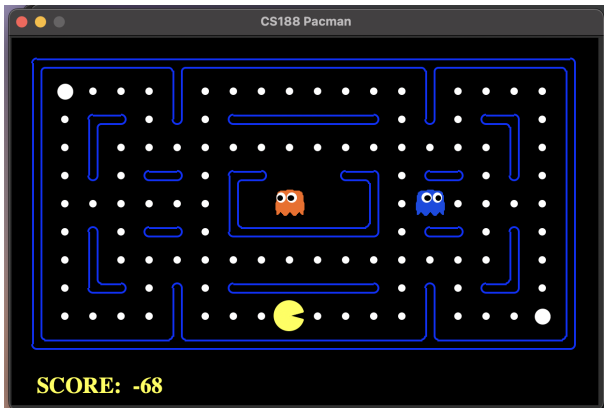

Figura 1: Pac-Man Game a short preview

2 Uninformed search

2.1 Question 1 - Depth-first search

Depth-First Search (DFS) este un algoritm de căutare neinformat care explorează in adancime, cât mai mult posibil de-a lungul fiecărei ramuri înainte de a reveni înapoi. În cadrul proiectului Pac-Man, sarcina pentru această cerinta a fost să implementăm DFS pentru a ajuta agentul Pac-Man să găsească un drum către punctul de mâncare fix in labirint.

2.1.1 Definirea cerinței

A fost nevoie sa implementez funcția `depthFirstSearch` în fișierul `search.py`, astfel încât algoritmul să returneze o listă de acțiuni care conduc Pac-Man de la poziția de start la mâncarea țintă.
Cerințele principale:

- Implementarea algoritmului DFS folosind structura de date `Stack` din `util.py`.
- Evitarea re-explorării stărilor deja vizitate pentru a ne feri de cicluri in program.
- Returnarea unui traseu format din acțiuni valide, care sa nu fie prin peretii labirintului.

2.1.2 Prezentarea algoritmului

Algoritmul DFS explorează cât mai profund posibil fiecare ramură înainte de a reveni pentru a explora alte opțiuni. Structura folosită pentru a gestiona frontiera este o stivă (`Stack`), care permite accesarea ultimului nod adăugat (*LIFO*). Pseudocodul de bază al algoritmului este următorul:

1. Inițializează stiva cu starea de start și traseul gol.
2. Cât timp avem elemente in stiva:
    - Extrage nodul curent din stivă.
    - Daca am ajuns in nodul de final, returnează traseul.
    - Dacă nodul curent nu a fost deja vizitat:
        - Il adaugam la setul de stări vizitate.

- Adaugă succesorii nodului curent în stivă împreună cu traseul actualizat, pentru ca avem nevoie de succesiunea de actiuni de la punctul de start la punctul de final.

3. Dacă stiva devine goală fără a găsi soluția, returnează o listă goală, ceea ce inseamna ca nu putem ajunge la punctul de final.

Cod:

```
def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """
    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()))
    """
    # dupa pseudocodul din cursul 2
    noduriVizitate = set() #folosim set ca sa evitam ciclurile
    frontiera = Stack()
    frontiera.push((problem.getStartState(), []))

    while not frontiera.isEmpty():
        stare, cale = frontiera.pop()

        if problem.isGoalState(stare):#daca am ajuns unde ne doream returnam calea
            return cale

        if stare not in noduriVizitate: #daca starea nu e in set, o adaugam si o marcam ca j
            noduriVizitate.add(stare)
            noduriSuccesoare = problem.getSuccessors(stare)
            for succesor, mutare, cost in noduriSuccesoare: #aici folosim si cost doar pentr
                if succesor not in noduriVizitate:
                    frontiera.push((succesor, cale + [mutare]))  #adaugam in stiva, succesor
    return []
```

2.1.3 Explicații suplimentare cod

- **Structura** `Stack`: Aceastra structura este esentiala pentru DFS, deoarece gestionează ordinea de explorare a nodurilor.
- **Setul de stări vizitate**: Ne ajuta sa evitam ciclurile in explorarea nodurilor.
- **Traseul** (`calea`): Este actualizat la fiecare pas pentru a păstra acțiunile care duc la nodul curent. Astfel in final o sa avem stocate actiunile luate de Pac-Man in traseul sau de la nodul de start la nodul de final.

2.1.4 Comentarii și observații

Performanță:

- DFS explorează profund fără să țină cont de costurile drumurilor, ceea ce înseamnă că poate produce soluții care nu sunt optime din punctul de vedere al costului.
- Numărul de noduri expandate depinde de ordinea succesorilor.

Complexitate:

- **Timp**: $O(b^m)$, unde b este factorul de ramificare și m este adâncimea maximă a arborelui.
- **Spațiu**: $O(b \cdot m)$, deoarece stiva stochează până la $b \cdot m$ noduri în cel mai rău caz.

Rezultate și concluzii:

DFS funcționează corect pentru toate labirinturile testate (`tinyMaze`, `mediumMaze`, `bigMaze`), iar pentru `mediumMaze` produce o soluție de lungime 130, ceea ce confirmă că ordinea succesorilor afectează lungimea soluției.

2.2 Question 2 - Breadth-first search

Breadth-First Search (BFS) este un algoritm de căutare neinformat care explorează toate nodurile de pe același nivel înainte de a trece la nivelurile următoare. În cadrul proiectului Pac-Mac, sarcina pentru această întrebare a fost să implementăm BFS pentru a găsi cel mai scurt traseu, în termeni de număr de pași, de la poziția de start la un punct de final.

2.3 Definirea cerintei

Avem de implementat functia breadthFirstSearch din fisierul search.py. La fel ca si la DFS, algoritmul trebuie sa returneze o lista de actiuni care conduc Pac-Man in labirint pentru a colecta mancarea si a castiga puncte pana sa ajunga la punctul de final. Pseudocodul de baza al algoritmului:

1. Inițializează coada cu starea de start și traseul gol.
2. Cât timp mai avem elemente in coada:
   - Extrage nodul curent din coada.
   - Dacă nodul curent este punctul de final, returnează traseul.
   - Dacă nodul curent nu a fost deja vizitat:
     - Il adaugam la setul de stări vizitate.
     - Adaugă succesorii nodului curent în coada împreună cu traseul actualizat.
3. Dacă coada devine goală fără a găsi soluția, returnează o listă goală, ceea ce inseamna ca nu putem ajunge la punctul de final.

Cod:

```
def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
    #exact aceeasi idee ca si la dfs, doar ca structura de date folosita este alta
    noduriVizitate = set()
    frontiera = Queue()
    frontiera.push((problem.getStartState(), []))

    while not frontiera.isEmpty():
        stare, cale = frontiera.pop()
```

```
10
11          if stare in noduriVizitate:
12              continue
13
14          noduriVizitate.add(stare)
15
16          if problem.isGoalState(stare): #daca am ajuns unde ne doream returnam calea
17              return cale
18
19          succesori = problem.getSuccessors(stare) #se parcurge graful pe nivele
20          for successor, mutare, cost in succesori: #nici aici nu avem nevoie de cost, puteam
21              if successor not in noduriVizitate:
22                  frontiera.push((successor, cale + [mutare]))
23      return []
```

2.3.1 Explicații suplimentare cod

- **Structura `Queue`**: Folosind aceasta structura se asigură faptul că nodurile sunt explorate în ordinea în care au fost descoperite, păstrând o strategie de explorare pe niveluri.
- **Setul de stări vizitate**: Previne re-explorarea nodurilor, reducând astfel numărul de noduri expandate și asigurând eficiența.
- **Traseul (`path`)**: Este actualizat la fiecare pas pentru a păstra acțiunile care duc la nodul curent, la final avand lista de actiuni de la nodul de start la nodul final.

2.3.2 Comentarii și observații

Performanță:

- Numărul de noduri expandate crește odată cu mărimea spațiului de căutare, ceea ce poate duce la un consum mare de memorie pentru labirinturi de dimensiuni mai mari.
- BFS garantează găsirea unui traseu cu numărul minim de pași către obiectiv, ceea ce îl face ideal pentru labirinturi mai simple.

Complexitate:

- **Timp**: $O(b^m)$, cu b este factorul de ramificare și m este adâncimea maximă a arborelui.
- **Spațiu**: $O(b \cdot m)$, deoarece stiva stochează până la $b \cdot m$ noduri în cel mai rău caz.

Rezultate și concluzii:

DFS funcționează corect pentru toate labirinturile testate (`tinyMaze`, `mediumMaze`, `bigMaze`), iar pentru `mediumMaze` produce o soluție de lungime 68, ceea ce confirmă că ordinea succesorilor afectează lungimea soluției.

2.3.3 Question 3 - Uniform Cost Search

Definirea cerinței

Uniform-Cost Search (UCS) este un algoritm de căutare neinformat care extinde întotdeauna nodul cu cel mai mic cost total. Spre deosebire de BFS, care se concentrează doar pe numărul de pași, UCS ia în considerare și costurile asociate fiecărui pas, ceea ce îl face potrivit pentru probleme în care există costuri variabile pentru acțiuni.
Sarcina pentru această întrebare a fost să implementăm funcția uniformCostSearch în fișierul search.py, astfel încât algoritmul să returneze o listă de acțiuni care minimizează costul total

necesar pentru a ajunge la obiectiv, la nodul final.

Prezentarea algoritmului

Pentru acest algoritm se utilizeaza o coadă cu priorități pentru a gestiona frontiera, ordonând nodurile pe baza costului total acumulat. Pseudocodul algoritmului este urmatorul:

1. Inițializează coada cu priorități cu starea de start, traseul gol și costul 0.
2. Cât timp mai avem elemente in coada de prioritati:
   (a) Extrage nodul cu costul cel mai mic din coadă.
   (b) Dacă nodul este cel final, returnează traseul.
   (c) Dacă nodul nu a fost deja vizitat sau are un cost mai mic decât cel înregistrat:
      i. Se adauga în setul de noduri vizitate cu costul său actualizat.
      ii. Adaugă succesorii nodului în coadă împreună cu traseul și costul actualizat.
3. Dacă coada devine goală fără a găsi soluția, returnează o listă goală, ceea ce inseamna ca nu exista solutie pentru traseul de la nodul se start la nodul final.

Cod:

```
def uniformCostSearch(problem: SearchProblem) -> List[Directions]: #calea cea mai ieftina co
    """Search the node of least total cost first."""
    #o extindere a bfs cu folosirea unei cozi de prioritati si stocarea costului pe parcurs
    #prioritatea o face costulNou calculat
    start = problem.getStartState()
    frontiera = PriorityQueue() #seamana cu bfs dar folosesc un dictionar pentru a tine cont
    frontiera.push((start, [], 0), 0)

    noduriVizitate = {} #dicționar folosit pentru a ține minte cel mai mic cost al drumului

    while not frontiera.isEmpty():
        stare, cale, cost = frontiera.pop() #tupla (stare, cale cost)
        if stare in noduriVizitate and noduriVizitate[stare] <= cost:
            continue

        noduriVizitate[stare] = cost

        if problem.isGoalState(stare): #daca am ajuns in punctul de stop atunci returnam cal
            return cale

        for successor, mutare, costCurent in problem.getSuccessors(stare): #Expansiunea nodu
            costNou = cost + costCurent
            if successor not in noduriVizitate or noduriVizitate[successor] > costNou:
                frontiera.push((successor, cale + [mutare], costNou), costNou) # dacă am găs
    return []
```

2.4 Explicații suplimentare

Coadă cu priorități: Se foloseste pentru a selecta întotdeauna nodul cu cel mai mic cost total.

Dicționarul de noduri vizitate: Reține costurile pentru fiecare nod deja vizitat, evitând astfel re-explorarea sau permitând înlocuirea nodurilor cu costuri mai mici.

Calculul costului total: Costul total pentru fiecare nod este suma costului traseului până la acel nod și a costului incremental al acțiunii care a dus la el.

2.5 Comentarii și observații

Performanță:

- Numărul de noduri expandate depinde de funcția de cost și de structura problemei.
- UCS garantează găsirea unei soluții optime, chiar și în prezența costurilor variabile.

Complexitate:

- **Timp:** $O(b^d)$, unde b este factorul de ramificare și d este adâncimea soluției.
- **Spațiu:** $O(b^d)$, datorită cozii care poate conține toate nodurile de pe un nivel.

Rezultate și concluzii:

- UCS funcționează corect pentru toate labirinturile testate (*tinyMaze*, *mediumMaze*, *bigMaze*).
- Comparativ cu BFS, UCS poate genera soluții mai scurte sau mai lungi în termeni de pași, dar întotdeauna cu cost minim, ceea ce îl recomanda în probleme cu costuri variabile.

3 Informed search

3.1 Question 4 - A* search algorithm

A* Search este un algoritm informat de căutare care utilizează o funcție de cost combinată pentru a găsi cea mai scurtă cale până la nodul țintă. Acest algoritm este o îmbunătățire față de UCS prin utilizarea, in plus unei funcții euristice pentru a ghida procesul de căutare in labirint a nodului de final.

3.1.1 Definirea cerinței

Implementarea funcției `aStarSearch` în fișierul `search.py` cu scopul de a returna o listă de acțiuni care conduc Pac-Man de la poziția de start la țintă. Algoritmul utilizeaza structura `PriorityQueue` și să integreze o euristică pentru a reduce numărul de noduri expandate.

Pseudocodul de bază al algoritmului este:

1. Inițializează coada cu priorități cu starea de start, traseul gol și costul 0.
2. Cât timp exista elemente in coada de prioritati:
   (a) Extrage nodul cu cel mai mic cost total (cost + euristică).
   (b) Dacă nodul este nodul de final, returnează traseul.
   (c) Dacă nodul nu a fost deja vizitat sau are un cost mai mic decât cel înregistrat:
      i. Se adauga în setul de noduri vizitate cu costul său actualizat.
      ii. Adaugă succesorii nodului în coadă împreună cu traseul și costul actualizat.
3. Dacă coada devine goală fără a găsi soluția, returnează o listă goală, ceea ce inseamna ca nu exista traseu optim de la nodul de start la nodul final.

Cod:

```python
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) -> List[Directions]:
    """Search the node that has the lowest combined cost and heuristic first."""
    # eureistica euclidiana : manhattan
    # inițializează coada cu priorități A* și adaugă nodul de start împreună cu drumul parcu
    # in plus fata de UCS tine minte cel mai bun cost si drumul parcurs pana in acel moment
    stareStart = problem.getStartState()
    vizitate = {}
    frontiera = PriorityQueue()
    frontiera.push((stareStart, [], 0), heuristic(stareStart, problem))

    while not frontiera.isEmpty():
        stare, cale, costCurent = frontiera.pop()

        if stare in vizitate and vizitate[stare] <= costCurent:
            continue

        vizitate[stare] = costCurent

        if problem.isGoalState(stare): #am ajuns in punctul in care doream
            return cale

        for succesor, mutare, cost in problem.getSuccessors(stare):
            costNou = costCurent + cost #h(n)
            prioritate = costNou + heuristic(succesor, problem) # f(n) = h(n) + g(n), unde g

            if succesor in vizitate and costNou >= vizitate[succesor]:
                continue
            frontiera.push((succesor, cale + [mutare], costNou), prioritate) # punem succes
    return []
```

3.1.2 Explicații suplimentare

Coadă cu priorități: Este utilizată pentru a selecta întotdeauna nodul cu cel mai mic cost (cost + euristică).
f(n) =h(n)+g(n)

Euristica: Este o funcție care estimează costul de la nodul curent la țintă. Alegerea unei euristici admisibile și consistente este esențială pentru ca algoritmul să garanteze o soluție optimă.

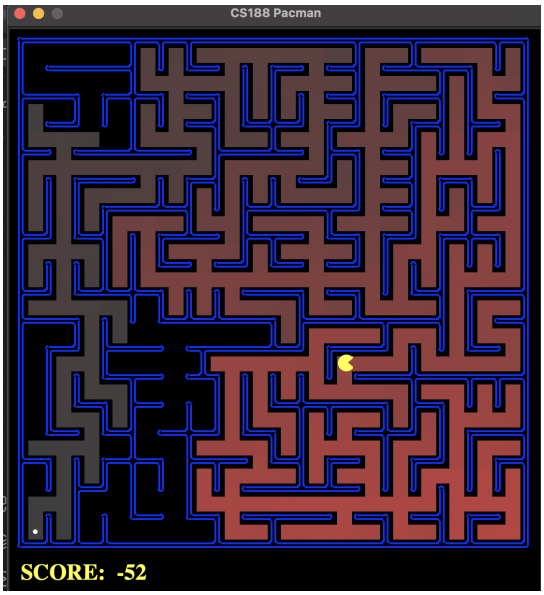

Figura 2: A*

3.1.3 Comentarii

Performanță:

- A* garantează găsirea unei soluții optime dacă euristica este admisibilă (nu supraestimează) și consistentă.
- Numărul de noduri expandate este influențat de calitatea euristicii; o euristică mai informată reduce numărul de noduri expandate.

Complexitate:

- **Timp:** $O(b^d)$, unde b este factorul de ramificare și d este adâncimea soluției.
- **Spațiu:** $O(b^d)$, deoarece coada poate conține toate nodurile explorate.

Rezultate și concluzii: A* funcționează corect pentru toate labirinturile testate și produce soluții optime în termeni de cost total. In comparatie cu UCS, A* este mai eficient datorită ghidării prin euristică, reducând numărul de noduri expandate, astfel ca ii permite lui Pac-Man sa mearga aproximativ doar in directia punctului de final.

Pentru bonus* Am implementat pe langa cerintele de baza o noua modalitate de parcurgere a labirintului pentru Pac-Man Ca si implementare am urmarit ideea de a ajunge in toate cele 4 colturi, doar ca am impus restrictia sa manance toata mancarea din labirint si sa atinga macar o data grid-ul de sus si cel de jos al labirintului. M-am folosit de parcurgerea BFS. **Cod**:

```
class FoodAndWalls(PositionSearchProblem):
    """
    Functionalitatea pe care doresc sa o adaug este traversarea lui Pacman pentru a colecta
    cu conditia sa atinga atat peretele de sus cat si peretele de jos al labirintului in cel
    """

    def __init__(self, gameState):
        "Stochează informațiile din gameState."
```

```
        self.food = gameState.getFood()
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0  # DO NOT CHANGE

    def isGoalState(self, state: Tuple[int, int]):
        """
        Verifică dacă poziția dată este o poziție de mâncare.
        """
        x, y = state
        return self.food[x][y]

    def getSuccessors(self, state: Tuple[int, int]):
        """
        Returnează succesorii unui nod, respectând restricțiile pentru peretele de sus și de
        """
        successors = []
        x, y = state
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:  # Asigură-te că nu colidează cu un perete
                successors.append(((nextx, nexty), action, 1))

                # Adaugă condițiile pentru atingerea peretelui de sus și de jos
                # Poți verifica dacă Pacman a atins o poziție de pe marginea labirintului
                if nextx == 0 or nextx == self.walls.width - 1:  # Peretele de sus și de jos
                    pass

        return successors

    def getStartState(self):
        """
        Returnează starea de start (poziția lui Pacman).
        """
        return self.startState

class FoodWallsSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"

    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        while (currentState.getFood().count() > 0):
            nextPathSegment = self.findFoodAndWalls(currentState)
            self.actions += nextPathSegment
            for action in nextPathSegment:
```

```
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception('findFoodAndWalls returned an illegal move: %s!\n%s' % t
                currentState = currentState.generateSuccessor(0, action)
        self.actionIndex = 0
        print('Path found with cost %d.' % len(self.actions))

    def findFoodAndWalls(self, gameState: pacman.GameState):
        """
        Returnează un drum către toată mâncarea, astfel incat sa fim siguri ca atinge cel pu
        """
        problem = FoodAndWalls(gameState)
        return search.bfs(problem)
```

4 Adversarial Search

4.1 Reflex Agent

Un *reflex agent* alege o acțiune la fiecare punct de decizie prin examinarea alternativelor sale folosind o funcție de evaluare a stării. Implementarea sa presupune alegerea acțiunii celei mai bune dintre opțiunile legale disponibile, pe baza scorului calculat de funcția de evaluare.

Codul de mai jos implementează un agent reflexiv simplu, folosind o funcție de evaluare bazată pe distanța față de mâncare și față de fantome.

```
class ReflexAgent(Agent):
    """
    A reflex agent chooses an action at each choice point by examining
    its alternatives via a state evaluation function.

    The code below is provided as a guide.  You are welcome to change
    it in any way you see fit, so long as you don't touch our method
    headers.
    """


    def getAction(self, gameState: GameState):
        """
        You do not need to change this method, but you're welcome to.

        getAction chooses among the best options according to the evaluation function.

        Just like in the previous project, getAction takes a GameState and returns
        some Directions.X for some X in the set {NORTH, SOUTH, WEST, EAST, STOP}
        """
        # Collect legal moves and successor states
        legalMoves = gameState.getLegalActions()

        # Choose one of the best actions
```

```
        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)  # Pick randomly among the best

        "Add more of your code here if you want to"

        return legalMoves[chosenIndex]

    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).
        newTimeScared 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()
        newStateGhost = successorGameState.getGhostStates()
        newTimeScared = [ghostState.scaredTimer for ghostState in newStateGhost]

        #Calculul scorului pentru mâncare
        foodList = newFood.asList()
        distFood = [manhattanDistance(newPos, food) for food in foodList]
        distMinToFood = min(distFood) if distFood else 0
        scoreFood = 1.0 / (distMinToFood + 1) #scoreFood: Cu cât mâncarea este mai aproape,

        #aici avem calculul scorului pentru fantome
        scoreForGhosts = 0
        for stateGhost, timeScared in zip(newStateGhost, newTimeScared):
            distGhost = manhattanDistance(newPos, stateGhost.getPosition())
            if timeScared <= 0:
                scoreForGhosts -= 1.0 / (distGhost + 1)
            else:
                scoreForGhosts += 1.0 / (distGhost + 1)

        if action == Directions.STOP:
            stopScore = -300
```

```
73          else:
74              stopScore = 0
75
76          return successorGameState.getScore() + scoreFood + scoreForGhosts + stopScore
```

Această implementare utilizează două concepte principale:

- **Evaluarea mâncării**: Se acordă un scor pozitiv în funcție de distanța față de mâncare, cu cât aceasta este mai aproape, cu atât scorul este mai mare.
- **Evaluarea fantomelor**: Se acordă un scor negativ pentru fantomele care nu sunt speriate și un scor pozitiv pentru cele care sunt speriate. De asemenea, distanța față de fantome este luată în considerare, cu cât fantoma este mai aproape, cu atât scorul este mai mic (dacă nu este speriată).
- **Penalizare pentru pericolul fantomelor**: Se adaugă o penalizare semnificativă atunci când o fantomă se află prea aproape de Pacman (distanța mai mică de 2 unități), ceea ce face ca agentul să prefere evitarea fantomelor periculoase.
- **Evita acțiunea de a sta pe loc**: Se introduce o penalizare suplimentară pentru acțiunea de a opri agentul (direcția STOP), pentru a încuraja mișcarea continuă.

4.2 Minimax Agent

Agentul Minimax utilizează algoritmul de căutare Minimax pentru a determina acțiunea optimă, având în vedere adâncimea căutării și evaluările stărilor succesive. Algoritmul se bazează pe alternarea maximizării scorului pentru agentul Pac-Man și minimizării scorului pentru fantome.

Codul de mai jos implementează agentul Minimax folosind o funcție recursivă pentru maximizarea și minimizarea valorii stărilor.

```
1  class MinimaxAgent(MultiAgentSearchAgent):
2      """
3      Your minimax agent (question 2)
4      """
5
6      def getAction(self, gameState: GameState):
7          """
8          Returns the minimax actiune from the current gameState using self.depth
9          and self.evaluationFunction.
10
11         Here are some method calls that might be useful when implementing minimax.
12
13         gameState.getLegalActions(agentIndex):
14         Returns a list of legal actions for an agent
15         agentIndex=0 means Pacman, ghosts are >= 1
16
17         gameState.generateSuccessor(agentIndex, actiune):
18         Returns the successor game state after an agent takes an actiune
19
20         gameState.getNumAgents():
21         Returns the total number of agents in the game
22
```

```
        gameState.isWin():
        Returns whether or not the game state is a winning state

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        "*** YOUR CODE HERE ***"
        #folosesc doua functii auxiliare de ajutor
        def maxValue(state, depth):
            if depth == self.depth or state.isWin() or state.isLose():
                return self.evaluationFunction(state)

            legalActions = state.getLegalActions(0)
            if not legalActions:
                return self.evaluationFunction(state)

            maxScore = float('-inf')

            for action in legalActions:
                successorState = state.generateSuccessor(0, action)
                score = minValue(successorState, depth, 1)
                if score > maxScore:
                    maxScore = score

            return maxScore

        def minValue(state, depth, agentIndex):
            if state.isWin() or state.isLose():
                return self.evaluationFunction(state)

            legalActions = state.getLegalActions(agentIndex)
            if not legalActions:
                return self.evaluationFunction(state)

            scoreMin = float('inf')
            if agentIndex < state.getNumAgents() - 1:
                nextAgent = agentIndex + 1
            else:
                nextAgent = 0

            if nextAgent == 0:
                depthNext = depth + 1
            else:
                depthNext = depth

            for actiune in legalActions:
                successorState = state.generateSuccessor(agentIndex, actiune)
                if nextAgent == 0:
```

```
                    score = maxValue(successorState, depthNext)
                else:
                    score = minValue(successorState, depthNext, nextAgent)
                scoreMin = min(scoreMin, score)

            return scoreMin

        def actionScore(action):
            stateSuccesor = gameState.generateSuccessor(0, action)
            return minValue(stateSuccesor, 0, 1)

        legalActions = gameState.getLegalActions(0)

        best = None
        scoreBest = float('-inf')
        for actiune in legalActions:
            score = actionScore(actiune)
            if score > scoreBest:
                scoreBest = score
                best = actiune

        return best
```

4.3 AlphaBeta Agent

Agentul AlphaBeta este o variantă optimizată a agentului Minimax, folosind tehnica *pruning* (tăierea ramurilor) pentru a îmbunătăți eficiența algoritmului. Aceasta presupune eliminarea ramurilor care nu pot afecta rezultatul căutării, economisind astfel timp de calcul.

Codul de mai jos implementează agentul AlphaBeta:

```
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 ***"

        def alphaBeta(state, depth, agentIndex, alpha, beta):
            if depth == self.depth or state.isWin() or state.isLose():
                return self.evaluationFunction(state)

            if agentIndex == 0:
                value = float('-inf')
                for action in state.getLegalActions(agentIndex):
                    successor = state.generateSuccessor(agentIndex, action)
                    value = max(value, alphaBeta(successor, depth, 1, alpha, beta))
```

```
                    if value > beta:
                        return value
                    #update pentru alpha
                    alpha = max(alpha, value)
                return value
            else:
                value = float('inf')
                nextAgent = agentIndex + 1
                if agentIndex == state.getNumAgents() - 1:
                    depth += 1
                    nextAgent = 0
                for action in state.getLegalActions(agentIndex):
                    successor = state.generateSuccessor(agentIndex, action)
                    value = min(value, alphaBeta(successor, depth, nextAgent, alpha, beta))
                    if value < alpha:
                        return value
                    #update pentru beta
                    beta = min(beta, value)
                return value

        alpha = float('-inf')
        beta = float('inf')
        bestAction = None
        for action in gameState.getLegalActions(0):
            successor = gameState.generateSuccessor(0, action)
            valoare = alphaBeta(successor, 0, 1, alpha, beta)
            if valoare > alpha:
                alpha = valoare
                bestAction = action

        return bestAction
```

4.4 Concluzie

Implementarea agenților Reflex, Minimax și AlphaBeta ilustrează abordări diferite ale căutării într-un joc adversarial. ReflexAgent ia decizii pe baza unei funcții de evaluare, MinimaxAgent folosește o căutare completă a arborelui de decizie, iar AlphaBetaAgent îmbunătățește performanța prin eliminarea ramurilor nepromițătoare folosind pruning.

4.5 Imbunatariri ReflexAgent

1. Adăugarea unei funcții de evaluare mai complexe: În loc să folosim doar distanța față de mâncare și fantome, putem lua în considerare mai mulți factori, cum ar fi numărul de bonusuri rămase pe hartă, distanța față de pericole (cum ar fi fantomele care nu sunt speriate)
2. Îmbunătățirea deciziilor în fața fantomelor: Se poate ajusta comportamentul agentului Reflex pentru a se comporta mai inteligent în prezența fantomelor.
3. Îmbunătățirea gestionării resurselor: Agentul ar putea să prioriteze acțiuni care maximizează scorul pe termen lung, nu doar scorul imediat.