UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori:Breha Paula si Balint Catalin
Grupa: 30236

Facultatea de Automatica
si Calculatoare

20 Noiembrie 2023

Cuprins

1 Pacman-Project

1.1 Introducere

În acest proiect agentul Pacman va gasi rutele prin labirint utilizând cea mai eficientă cale, dar și a colecta toate bucățile de mancare . Vom folosi diferiți algoritmi de căutare pentru a îndeplini cu succes misiunea având in vedere mai multe scenarii.

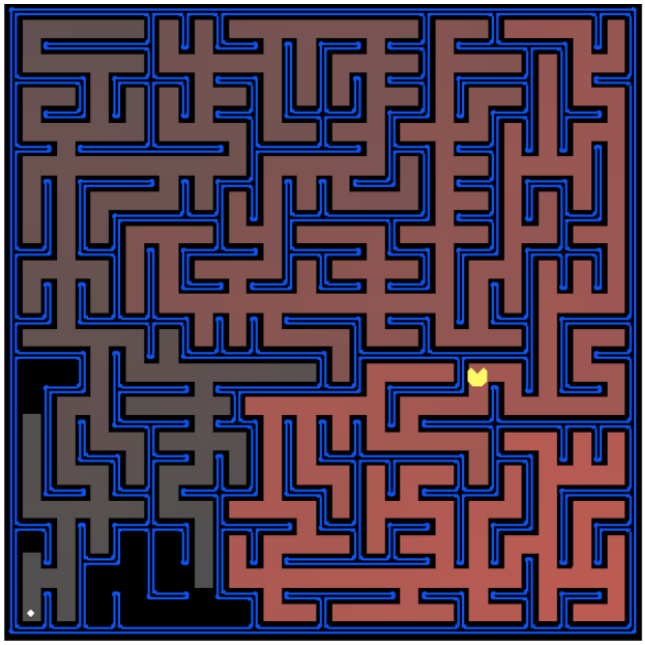

Figura 1: Labirint Pacman

1.2 Ideea de baza

Scopul jocului este ca agentul Pacman să colecteze toate bucățile de mancare (bulinele albe), fără a fii omorât de către fantomițele care il urmaresc pe parcurs.

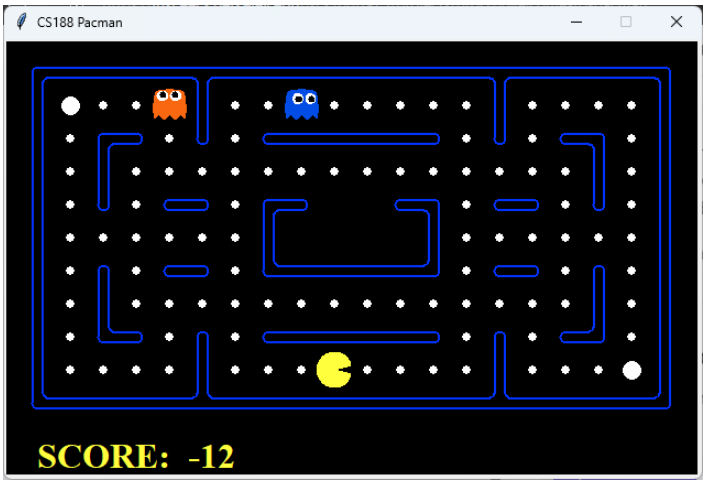

Figura 2: Jocul Pacman

2 Uninformed search

2.1 Question 1 - Depth-first search

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

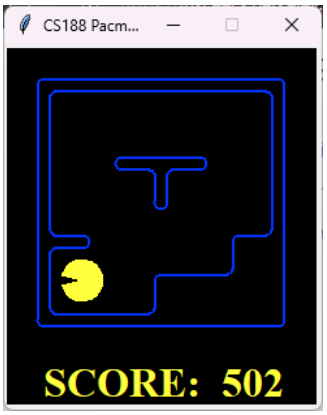

Figura 3: Labirint DFS

2.2 Metoda Depth-first search

```
def depthFirstSearch(problem: SearchProblem):
    stack = util.Stack()
    visited = set()

    start_state = problem.getStartState()
    stack.push((start_state, []))

    while not stack.isEmpty():
        state, actions = stack.pop()
        if state in visited:
            continue
        visited.add(state)
        if problem.isGoalState(state):
            return actions
        successors = problem.getSuccessors(state)
        for successor in successors:
            next_state, action, _ = successor
            if next_state not in visited:
                next_actions = actions + [action]
                stack.push((next_state, next_actions))
    return []
```

Algoritmul explorează cât mai adânc posibil pe o ramură înainte de a se întoarce. Folosește o stivă pentru a ține evidența nodurilor de explorat. În esență, începe de la nodul de start,

explorează pe o ramură cât poate de mult în profunzime și apoi revine și explorează alte ramuri.

2.3 Rezultate/Complexitati DFS

Testarea in terminal pentru **Q1** da rezulatate de **5/5**, trecand de toate testele cu bine.Complexitatea este de **O(nr.muchii + nr.noduri)**.

2.4 Question 2 - Breadth-first search

Implement the breadth-first search **(BFS)** algorithm in the Breath-first search 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.

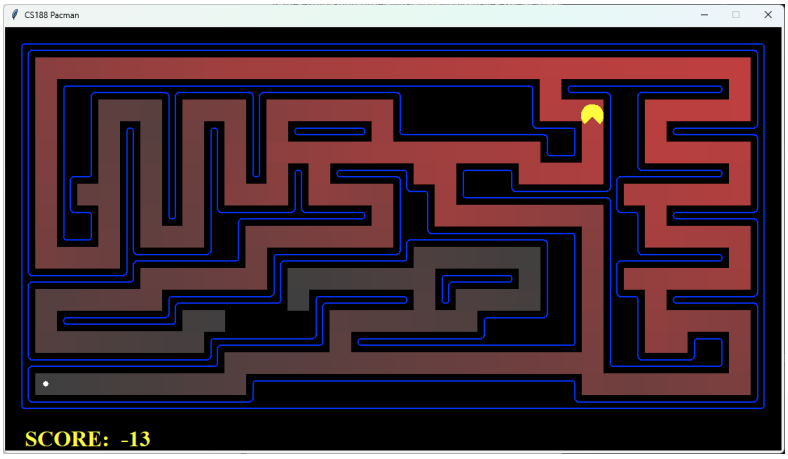

Figura 4: Labirint BFS

2.5 Metoda Breath-first search

```
def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    queue = util.Queue()
    visited = set()

    start_state = problem.getStartState()
    queue.push((start_state, []))

    while not queue.isEmpty():
        state, actions = queue.pop()
        if state in visited:
            continue
        visited.add(state)
        if problem.isGoalState(state):
            return actions
        successors = problem.getSuccessors(state)
        for successor in successors:
            next_state, action, _ = successor
```

```
            if next_state not in visited:
                next_actions = actions + [action]
                queue.push((next_state, next_actions))

    return []
```

Algoritmul explorează un graf începând de la o stare inițială, găsind cel mai scurt drum către o stare finală (dacă există) și returnând lista de acțiuni necesare pentru a ajunge acolo. Se folosește o coadă pentru a gestiona nodurile de explorat și un set pentru a ține evidența nodurilor vizitate.

2.6 Rezulatate/complexitati BFS

Testarea in terminal pentru **Q2** da rezulatate de **3/3**, trecand de toate testele cu bine.Complexitatea este de **O(nr.muchii + nr.noduri)**.

2.7 Q3 - Uniform-Cost function

Implement the uniform-cost graph search algorithm in the Uniform-Cost search function in search.py. We encourage you to look through util.py for some data structures that may be useful in your implementation.

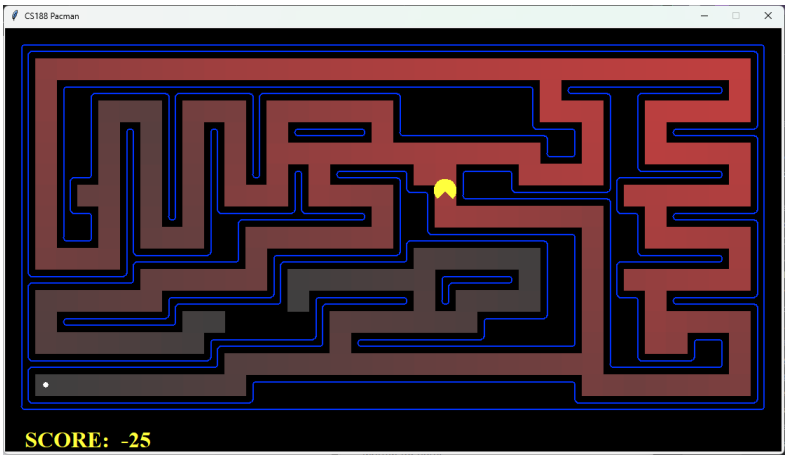

Figura 5: Labirint UCS

2.8 Algoritmul Uniform-Cost

```
def uniformCostSearch(problem: SearchProblem):
    """Search the node of least total cost first."""
    priority_queue = util.PriorityQueue()
    visited = set()

    start_state = problem.getStartState()
    priority_queue.push((start_state, []), 0)

    while not priority_queue.isEmpty():
```

```
        state, actions = priority_queue.pop()

        if state in visited:
            continue
        visited.add(state)
        if problem.isGoalState(state):
            return actions
        successors = problem.getSuccessors(state)
        for successor in successors:
            next_state, action, step_cost = successor
            if next_state not in visited:
                next_actions = actions + [action]
                total_cost = problem.getCostOfActions(next_actions)
                priority_queue.push((next_state, next_actions), total_cost)
    return []

```

Algoritmul Uniform Cost Search (UCS) explorează un graf pentru a găsi cel mai ieftin drum către o stare finală. Folosește o coadă de priorități pentru a extinde nodurile, luând în considerare costurile căilor până la acestea. În fiecare pas, alege nodul cu cel mai mic cost și explorează succesoriile nevizitate, menținând un cost total până la fiecare nod. Algoritmul se oprește atunci când găsește nodul final sau când nu mai există noduri de explorat, întorcând lista de acțiuni pentru cel mai ieftin drum găsit sau o listă vidă în caz contrar.

2.9 Rezulatate/Complexitati UCS

Testarea in terminal pentru **Q3** da rezulatate de **3/3**, trecand de toate testele cu bine.Complexitatea este de **O(nr.muchii + nr.noduri)**.

3 Informed search

3.1 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.

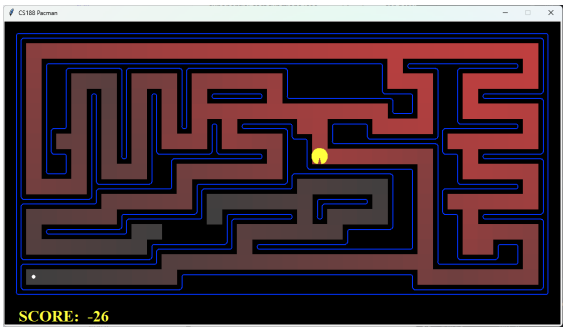

Figura 6: Labirint A*

3.2 Metoda A*

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    priority_queue = util.PriorityQueue()
    visited = set()

    start_state = problem.getStartState()
    priority_queue.push((start_state, []), heuristic(start_state, problem))

    while not priority_queue.isEmpty():
        state, actions = priority_queue.pop()
        if state in visited:
            continue
        visited.add(state)
        if problem.isGoalState(state):
            return actions
        successors = problem.getSuccessors(state)
        for successor in successors:
            next_state, action, step_cost = successor
            if next_state not in visited:
                next_actions = actions + [action]
                g_cost = problem.getCostOfActions(next_actions)
                h_cost = heuristic(next_state, problem)
                f_cost = g_cost + h_cost
                priority_queue.push((next_state, next_actions), f_cost)

    return []
```

Algoritmul A* explorează nodurile pe baza costurilor,dar si a euristicii până la ele, priorizând mereu cele mai ieftine căi. Astfel, asigură găsirea unui drum optim, cel mai ieftin, către destinație într-un graf cu costuri asociate muchiilor.

3.3 Rezulatate/Complexitati A*

Testarea in terminal pentru **Q4** da rezulatate de **3/3**, trecand de toate testele cu bine.Complexitatea este de **O(nr.stari + nr.conexiuni)**.

3.4 Question 5 - Corners-Problem

Implement the Corners-Problem 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.

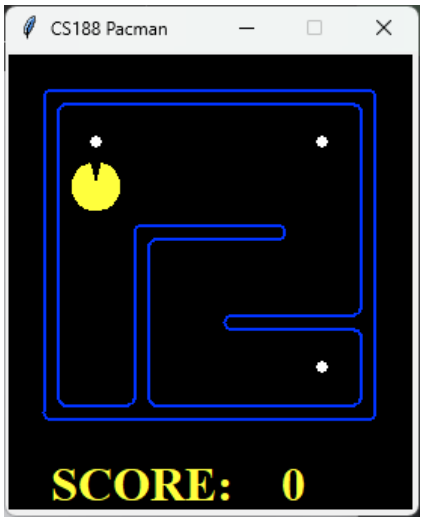

Figura 7: Labirint Corners-Problem

3.5 Algoritmul Finding-all-Corners

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

    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState: pacman.GameState):
        """
        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
        self.visited_corners = set()

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        return (self.startingPosition, tuple())

    def isGoalState(self, state: Any):
        """
```

```
        Returns whether this search state is a goal state of the problem.
        """
        current_position, visited_corners = state
        return set(visited_corners) == set(self.corners)

    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 = []
        current_position, visited_corners = state

        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
            x, y = current_position
            dx, dy = Actions.directionToVector(action)
            next_position = int(x + dx), int(y + dy)

            if not self.walls[next_position[0]][next_position[1]]:
                if next_position in self.corners and next_position not in visited_corners:
                    new_visited_corners = tuple(list(visited_corners) + [next_position])
                    successors.append(((next_position, new_visited_corners), action, 1))
                else:
                    successors.append(((next_position, visited_corners), action, 1))

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

Acest cod definește o problemă de căutare numită CornersProblem care vizează găsirea unui drum care trece prin toate cele patru colțuri ale unui labirint.

- Metoda getStartState() furnizează starea inițială a căutării, care este formată din poziția de start a lui Pacman și o listă goală a colțurilor vizitate;
- Metoda isGoalState() verifică dacă starea dată reprezintă o stare finală, adică dacă toate colțurile au fost vizitate;
- Metoda getSuccessors() returnează stările succesoare pentru o anumită stare, acțiunile necesare pentru a ajunge la aceste stări și costul asociat;
- etoda getCostOfActions() calculează costul total al unei secvențe de acțiuni. Acesta verifică dacă acțiunile duc la o mișcare ilegală și în caz afirmativ, returnează o valoare mare pentru a penaliza acele acțiuni.
  Algoritmul va încerca diferite acțiuni și va evalua costurile acestora pentru a găsi drumul optim care parcurge toate colțurile în cel mai eficient mod posibil.

3.6 Rezulatate/Complexitati Corners-Problem

Testarea in terminal pentru **Q5** da rezulatate de **6/6**, trecand de toate testele cu bine.

3.7 Q6 - Corners-Heuristic

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

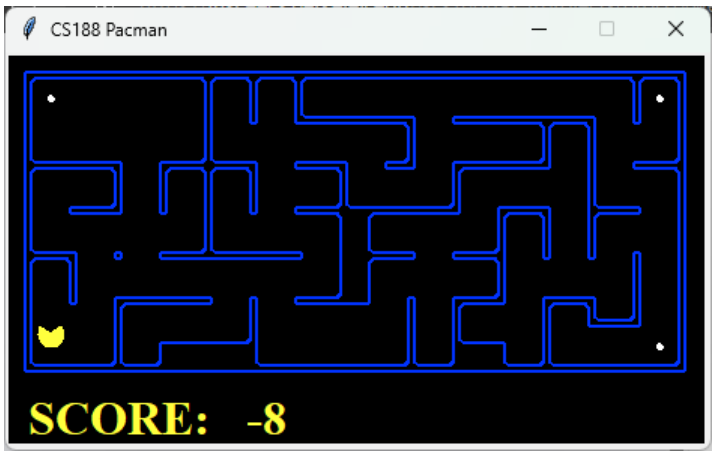

Figura 8: Labirint Corners-Problem

3.8 Algoritmul Corners-Heuristic

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    current_position, visited_corners = state
    corners = problem.corners

    if set(visited_corners) == set(corners):
        return 0

    unvisited_corners = set(corners) - set(visited_corners)

    if unvisited_corners:
        total_distance = sum(util.manhattanDistance(current_position, corner) for corner in
        average_distance = total_distance / len(unvisited_corners)
        max_distance = max(util.manhattanDistance(current_position, corner) for corner in ur
        return max(average_distance, max_distance)
    else:
        return 0
```

Acest algoritm reprezintă o euristică pentru a ghida algoritmul A* în rezolvarea problemei CornersProblem. Scopul ei este să furnizeze o estimare a costului rămas până la finalizarea obiectivului, adică găsirea tuturor colțurilor din labirint. Euristicile sunt utilizate în algoritmi de căutare informați pentru a ghida explorarea către soluții eficiente.

3.9 Rezulatate/Complexitati Corners-Heuristic

Testarea in terminal pentru **Q6** da rezulatate de **5/6**, trecand aproape toate testele.

- Question q4 : 3/3
- Question q6 : 2/3

3.10 Q7 - Eating-All-the-Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: FoodSearchProblem in searchAgents.py (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) 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).

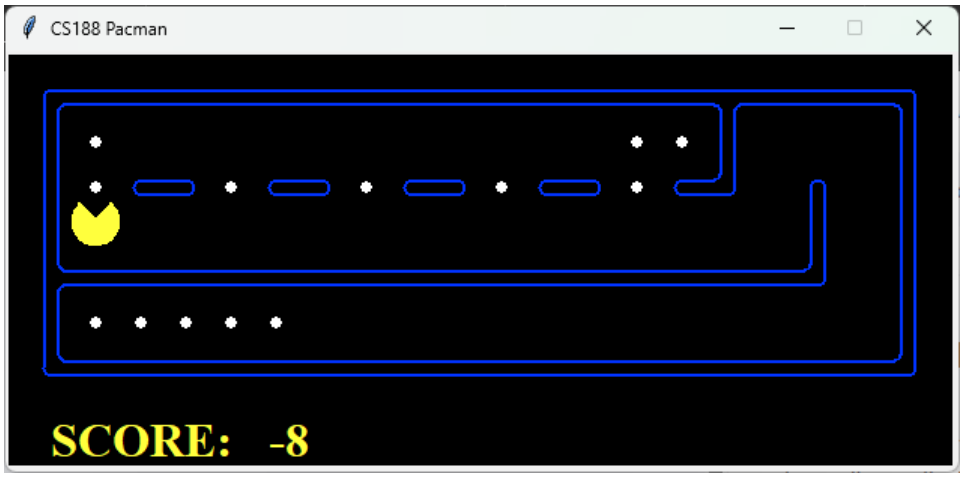

Figura 9: Labirint Eating-All-Dots

3.11 Algoritmul Eating-All-the-Dots

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):

    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    foodList = foodGrid.asList()
    if len(foodList) == 0:
        return 0
    nextFood = NextFood(position, foodList)
    return util.manhattanDistance(position, nextFood)
```

Această implementare utilizează algoritmul A* pentru a găsi cel mai scurt drum pentru ca Pacman să mănânce toată mâncarea din labirint, având o euristică care îl îndrumă către cea mai îndepărtată bucată de mâncare rămasă.

- FoodSearchProblem:Metode pentru a determina starea inițială, starea finală și stările succesoare;
- Agent care utilizează algoritmul A* pentru căutarea în FoodSearchProblem;

- FoodHeuristic:Alegerea următoarei bucati de mâncare pe baza distanței Manhattan între poziția lui Pacman și bucatile de mâncare rămase.

3.12 Rezulatate/Complexitati Corners-Problem

Testarea in terminal pentru **Q7** da rezulatate de **6/7**, trecand aproape toate testele.

- Question 4: 3/3
- Question 7: 2/3

3.13 Q8 - Closest-Dot-Search

Write an agent that always greedily eats the closest dot. Closest-Dot-SearchAgent is implemented for you in searchAgents.py, but it's missing a key function that finds a path to the closest dot.

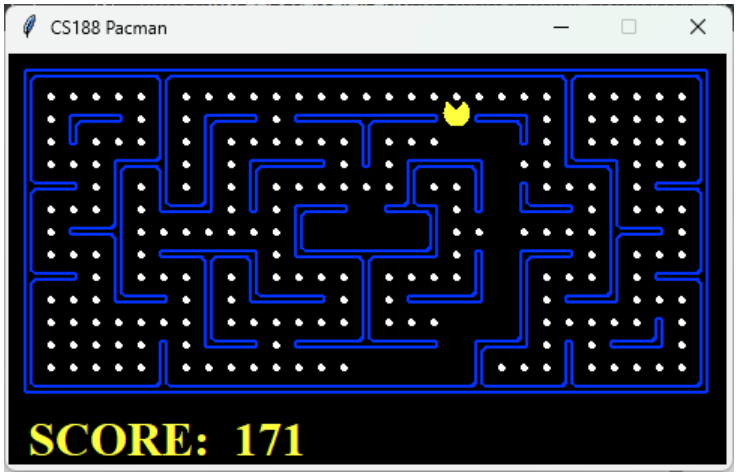

Figura 10: Labirint ClosestDot

3.14 Algoritmul Closest-Dot-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.bfs(problem)
        return path

class AnyFoodSearchProblem(PositionSearchProblem):

    def __init__(self, gameState):
        "Stores information from the gameState.  You don't need to change this."
```

```
18          # Store the food for later reference
19          self.food = gameState.getFood()
20
21          # Store info for the PositionSearchProblem (no need to change this)
22          self.walls = gameState.getWalls()
23          self.startState = gameState.getPacmanPosition()
24          self.costFn = lambda x: 1
25          self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT CHANGE
26
27      def isGoalState(self, state: Tuple[int, int]):
28          x,y = state
29
30          "*** YOUR CODE HERE ***"
31          return self.food[x][y]
```

Acest agent de căutare încearcă să găsească și să urmeze un drum către cea mai apropiată bucată de mâncare folosind algoritmul BFS (Breadth-First Search), iar acest proces este repetat până când toată mâncarea este consumată în labirint.

- 'registerInitialState':Inițializează starea jocului și caută mâncarea până când nu mai există puncte de mâncare;
- 'findPathToClosestDot':Găsește un drum către cea mai apropiată bucată de mâncare.Primește starea curentă a jocului și utilizează algoritmul BFS pentru a găsi drumul către cea mai apropiată bucată de mâncare.Construiește un obiect de tip AnyFoodSearchProblem pentru a descrie problema de căutare.

3.15 Rezulatate/Complexitati Closest-Dot-SearchAgent

Testarea in terminal pentru **Q8** da rezulatate de **3/3**, trecand toate testele cu bine.

4 Adversarial search

4.1 Question 1 - Improve the ReflexAgent

Improve the ReflexAgent in multiAgents.py to play respectably. The provided reflex agent code provides some helpful examples of methods that query the GameState for information

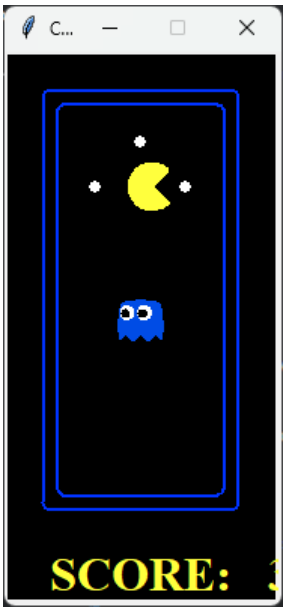

Figura 11: Labirint Reflex-Agent

4.2 Metoda Reflex-Agent

```
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]
    "*** YOUR CODE HERE ***"
    foodlist = newFood.asList()

    position_ghosts = [ghost for ghost in successorGameState.getGhostPositions()]

    distances = [manhattanDistance(newPos, food) for food in foodlist]

    if len(distances) == 0:
        return float('inf')
    extrascore = 0.0

    min_distance = min(distances)
    extrascore = 1 / min_distance * 10

    for index, ghost in enumerate(position_ghosts):
        if newScaredTimes[index] > 0:
            continue
        else:
            if manhattanDistance(ghost, newPos) < 2:
                extrascore = float('-inf')
```

```
29      return successorGameState.getScore() + extrascore
30
```

Codul calculează noua poziție a lui Pac-Man, mâncarea rămasă și pozițiile fantomelor după o acțiune dată. Determină distanța până la cea mai apropiată bucată de mâncare și acordă un scor bazat pe cât de aproape este Pac-Man de aceasta.Verifică apropierea fantomelor de Pac-Man; dacă o fantomă este prea aproape și nu este speriată, evită acea acțiune.

4.3 Rezulatate/Complexitati Reflex-Agent

Testarea in terminal pentru **Q1/proj.2** da rezulatate de **4/4**, avand rezultate maxime la win rate.

4.4 Q2 - MinMax

Write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in lecture. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Figura 12: Labirint MinMax

4.5 Algoritmul MinMax

```
1   class MinimaxAgent(MultiAgentSearchAgent):
2       """
3       Your minimax agent (question 2)
4       """
5
6       def getAction(self, gameState: GameState):
7
8           legalActions = gameState.getLegalActions(0)  # Pacman's legal actions
9           scores = [self.minimaxValue(gameState.generateSuccessor(0, action), 1, 1)
10                    for action in legalActions]
11          bestAction = max(zip(scores, legalActions))[1]
12          return bestAction
13
14      def minimaxValue(self, gameState: GameState, depth: int, agentIndex: int):
```

```
        """
        Returns the minimax value of a gameState.
        """
        if depth > self.depth or gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)

        if agentIndex == 0:
            return max(self.minimaxValue(gameState.generateSuccessor(0, action),
                                         depth, 1) for action in gameState.getLegalActions(0
        else:  # Ghosts' turn (minimizing)
            nextAgentIndex = (agentIndex + 1) % gameState.getNumAgents()
            if nextAgentIndex == 0:
                depth += 1
            return min(self.minimaxValue(gameState.generateSuccessor(agentIndex, action),
                                         depth, nextAgentIndex) for action in gameState.getL

```

Agentul încearcă să aleagă acțiuni care maximizează scorul lui Pac-Man și minimizează scorul fantomelor.La fiecare nivel, verifică dacă s-a atins adâncimea maximă, a fost câștig sau pierdere în joc pentru a opri explorarea recursivă. Pentru fiecare nivel, alternează între maximizarea scorului lui Pac-Man și minimizarea scorului pentru fantome.

4.6 Rezulatate/Complexitati MinMax

Testarea in terminal pentru **Q2/proj.2** da rezulatate de **5/5**, avand rezultate maxime la win rate.

4.7 Q3 - Alfa-Beta Pruning

Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in AlphaBetaAgent. Again, your algorithm will be slightly more general than the pseudocode from lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

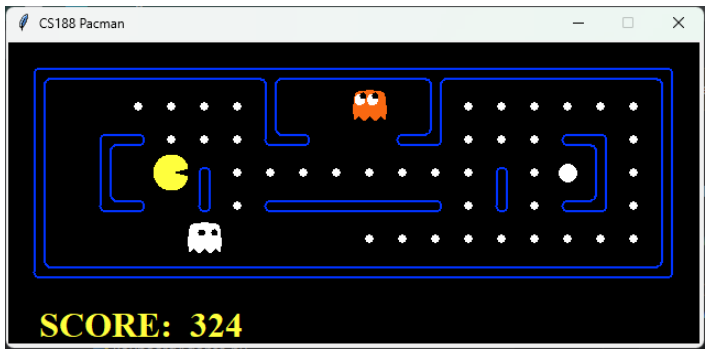

Figura 13: Labirint Alpha-Beta

4.8 Algoritmul Alpha-Beta Pruning

```
def alphabeta_pruning(self, state: GameState, alpha, beta, agent_index, depth):
    """If we reached the depth that we want to search or it's a win state or it's a lose sta
```

```
    if depth == self.depth or state.isWin() or state.isLose():
        return self.evaluationFunction(state)

    if agent_index == 0:
        return self.maximize(state, alpha, beta, agent_index, depth)[0]
    else:
        return self.minimize(state, alpha, beta, agent_index, depth)[0]


def maximize(self, state: GameState, alpha, beta, agent_index, depth):
    maxValue = (float('-inf'),)
    legal_moves = state.getLegalActions(agent_index)

    for action in legal_moves:
        evalue = (
            self.alphabeta_pruning(state.generateSuccessor(agent_index, action), alpha, beta
            action)

        maxValue = self.max_eval(evalue, maxValue)
        alpha = self.max_eval(alpha, evalue)
        if beta[0] < alpha[0]:
            break

    return maxValue


def minimize(self, state: GameState, alpha, beta, agent_index, depth):
    minValue = (float('inf'),)
    legal_moves = state.getLegalActions(agent_index)

    for action in legal_moves:
        if agent_index == state.getNumAgents() - 1:
            agent_index = -1
            depth += 1
        evalue = (
            self.alphabeta_pruning(state.generateSuccessor(agent_index, action), alpha, beta
            action)

        minValue = self.min_eval(evalue, minValue)
        beta = self.min_eval(beta, evalue)
        if beta[0] < alpha[0]:
            break

    return minValue


def max_eval(self, evaluation1, evaluation2):
    if evaluation1[0] > evaluation2[0]:
```

```
        return evaluation1
    else:
        return evaluation2


def min_eval(self, evaluation1, evaluation2):
    if evaluation1[0] < evaluation2[0]:
        return evaluation1
    else:
        return evaluation2
    # util.raiseNotDefined()
```

Algoritmul este folosit pentru a optimiza explorarea arborelui de joc și a reduce numărul de noduri evaluate. Algoritmul încearcă să maximizeze beneficiul propriu și să minimizeze beneficiul adversarului.În esență, algoritmul Alpha-Beta Pruning optimizează algoritmul Minimax, reducând numărul de noduri evaluate în arborele de joc.

Obiectivul său final este de a găsi cea mai bună decizie posibilă, având în vedere adâncimea limitată de căutare.

4.9 Rezulatate/complexitati

Testarea in terminal pentru **Q3** da rezulatate de **5/5**, trecand de toate testele cu bine.

4.10 Q4 - Expectimax

Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case. In this question you will implement the ExpectimaxAgent, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices. As with the search and problems yet to be covered in this class, the beauty of these algorithms is their general applicability. To expedite your own development, we've supplied some test cases based on generic trees.

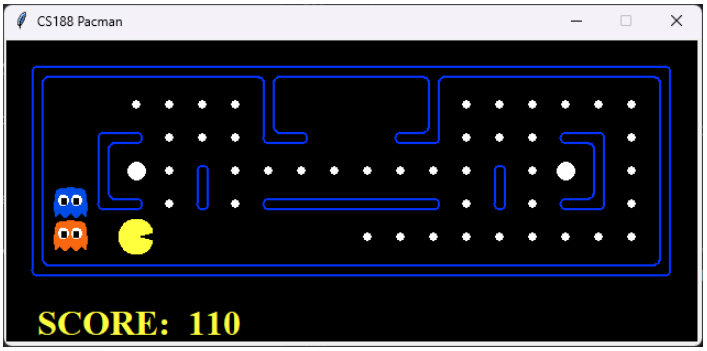

Figura 14: Labirint Expectimax

4.11 Metoda Expectimax

```
class ExpectimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
```

```
3
4           legalActions = gameState.getLegalActions(0)  # Pacman's legal actions
5           expectedValues = [self.expectimaxValue(gameState.generateSuccessor(0, action), 1, 1)
6                             for action in legalActions]
7           bestAction = max(zip(expectedValues, legalActions))[1]
8           return bestAction
9
10      def expectimaxValue(self, gameState: GameState, depth: int, agentIndex: int):
11          """
12          Returns the expectimax value of a gameState.
13          """
14          if depth > self.depth or gameState.isWin() or gameState.isLose():
15              return self.evaluationFunction(gameState)
16
17          if agentIndex == 0:
18              return max(self.expectimaxValue(gameState.generateSuccessor(0, action),
19                                              depth, 1) for action in gameState.getLegalActior
20          else:  # Ghosts' turn
21              nextAgentIndex = (agentIndex + 1) % gameState.getNumAgents()
22              if nextAgentIndex == 0:
23                  depth += 1
24              ghostActions = gameState.getLegalActions(agentIndex)
25              prob = 1.0 / len(ghostActions)
26              return sum(prob * self.expectimaxValue(gameState.generateSuccessor(agentIndex, a
27                                                     depth, nextAgentIndex) for action in ghos
```

Algoritmul Expectimax se bazează pe ideea că atunci când nu există informație despre acțiunile fantomelor, se folosește media ponderată a valorilor așteptate. Pac-Man încearcă să maximizeze valoarea sa în timp ce fantomele sunt considerate a acționa aleatoriu, fără un comportament specific. Astfel, agentul alege acțiunile care îi maximizează așteptările, având în vedere incertitudinea comportamentului adversarilor.

4.12 Rezultate/complexitati

Testarea in terminal pentru **Q4** da rezulatate de **5/5**, trecand de toate testele cu bine.

4.13 Q5 - Evaluation Function

Write a better evaluation function for Pacman in the provided function betterEvaluationFunction. The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did. With depth 2 search, your evaluation function should clear the smallClassic layout with one random ghost more than half the time and still run at a reasonable rate (to get full credit, Pacman should be averaging around 1000 points when he's winning).

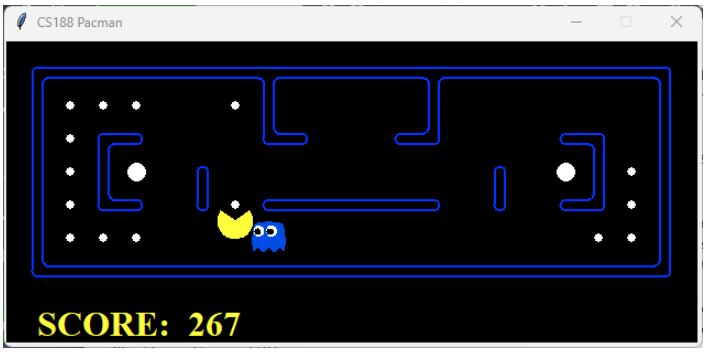

Figura 15: Labirint Evaluation-Function

4.14 Metoda Evaluation-Function

```
def betterEvaluationFunction(currentGameState: GameState):

    pacmanPosition = currentGameState.getPacmanPosition()
    foodList = currentGameState.getFood().asList()
    ghostStates = currentGameState.getGhostStates()


    score = currentGameState.getScore()

    minGhostDistance = min(manhattanDistance(pacmanPosition, ghost.getPosition()) for ghost

    remainingFood = len(foodList)
    evaluation = score - 2 * minGhostDistance - 10 * remainingFood

    return evaluation
```

Această funcție de evaluare combină informații despre scorul curent, distanța până la fantome și numărul de bucăți de mâncare rămase pentru a obține o estimare a valorii unei stări de joc.În concluzie evaluează starea curentă a jocului PacMan.

4.15 Rezultate/complexitati

Testarea in terminal pentru **Q5** da rezulatate de **5/6**, trecand de toate testele cu bine.

5 Concluzii

Proiectul Pacman m-a ajutat să învăț să adaptez partea teoretica învățată la curs pentru cazurile diferite întâlinite pe parcursul fiecărui ”q”. Totodată am mai învățat și cum să mă folosesc de niște funcții gata implementate , pentru a-mi putea rezolva task-urile si a pune în aplicare ideile personale.

6 Bibliografie

- *https://inst.eecs.berkeley.edu/ cs188/fa23/projects/proj1/*
- *https://inst.eecs.berkeley.edu/ cs188/fa23/assets/lectures/cs188-fa23-lec03.pdf*