UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Popovici Eusebiu-Ionut si Rus Ionel
Grupa: 30234

Facultatea de Automatica
si Calculatoare

17 Noiembrie 2022

Cuprins

1 Introducere

Această lucrare explorează strategiile, implementările și rezultatele obținute prin utilizarea diferiților algoritmi de căutare pentru rezolvarea problemelor clasice din domeniul inteligenței artificiale. Se folosește șablonul proiectului Pac-Man de la UC Berkeley pentru a investiga și implementa soluții pentru diverse provocări din acest domeniu.

Proiectul se axează în principal pe două obiective principale:

1. **Algoritmii de cautare**:
   - DFS, BFS, UCS, A*
   - Implementarea unor agenti de cautare si a unor euristici pentru eficientizarea cautarilor.
2. **Introducerea in Multiagent**:
   - Implementarile Reflex Agentului, algoritmului MiniMax si al Alpha-Beta-Pruningului.
   - Implementarea algoritmilor Expectimax si cel de Evaluation Function.

2 Uninformed search

2.1 Question 1 - Depth-first search

2.1.1 Definire cerinta

Este timpul sa scriem functii generice de cautare pentru a ajuta Pacman sa isi planifice drumurile! Implementeaza algoritmul de cautare in adancime (DFS) in functia depthFirstSearch din search.py. Pentru a face algoritmul complet, scrie versiunea de cautare in graf a DFS-ului, care evita extinderea oricaror stari deja vizitate.

2.1.2 Prezentare algoritm/metoda

Pentru DFS ca structura de date am folosit o Stiva,iar partea de cod s-a implemtentat un pseudocod pentru un DFS iterativ.

2.1.3 Prezentare Cod

.

```
90      visited = set()
91      stack = util.Stack()
92      path = []
93      begin = problem.getStartState()
94      stack.push((begin, path))
95      while not stack.isEmpty():
96          (node, nodePath) = stack.pop()
97          if node not in visited:
98              visited.add(node)
99              if problem.isGoalState(node):
100                 return nodePath
101             successors = problem.getSuccessors(node)
102             for i, action, _ in successors:
103                 stack.push((i, nodePath + [action]))
104     return None
```

Figura 1: DFS CODE

2.1.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste si a fost destul de usor de implementat.

2.2 Question 2 - Breadth-first search

2.2.1 Definire cerinta

Implementeaza algoritmul de cautare pe latime (BFS) in functia breadthFirstSearch din search.py. Din nou, scrie un algoritm de cautare in graf care evita sa extinda orice stari deja vizitate. Testeaza-ti codul in acelasi mod in care ai facut pentru cautarea in adancime

2.2.2 Prezentare algoritm/metoda

Pentru acest algoritm(BFS) structura de date folosita este o Coada.Codul este similar ca cel al DFS-ului, diferenta fiind structura de date.

2.2.3 Prezentare Cod

.

```python
def breadthFirstSearch(problem: SearchProblem):
    begin = problem.getStartState()
    queue = util.Queue()
    visited = set()
    path = []
    queue.push((begin, path))
    while not queue.isEmpty():
        node, path = queue.pop()

        if problem.isGoalState(node):
            return path

        if node not in visited:
            visited.add(node)

            successors = problem.getSuccessors(node)
            for i in successors:
                next_node, move, _ = i
                new_path = path + [move]
                queue.push((next_node, new_path))
    return []
```

Figura 2: BFS CODE

2.2.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste si a fost destul de usor de implementat.

2.3 Question 3 - Uniform Cost Search

2.3.1 Definire cerinta

In timp ce BFS va gasi un traseu cu cel mai mic numar de actiuni catre scop, s-ar putea sa dorim sa gasim trasee "mai bune" in alte sensuri. Ia in considerare mediumDottedMaze si mediumScaryMaze. Schimband functia de cost, putem incuraja Pacman sa gaseasca cai diferite. De exemplu, putem percepe mai mult pentru pasii periculosi in zonele pline de fantome sau mai putin pentru pas, ii in zone bogate in mancare, iar un agent Pacman rational ar trebui sa-si ajusteze comportamentul in consecinta. Implementeaza algoritmul de cautare uniform-cost in graf in functia uniformCostSearch din search.py. Te incurajam sa consulti util.py pentru unele structuri de date care ar putea fi utile in implementarea ta.

2.3.2 Prezentare algoritm/metoda

Pentru partea de UCS am refolosit codul de la BFS, diferenta fiind data de structura de date care este o coada cu prioritate. Prioritatea este reprezentata de distanta de la start pana la nodul curent.

2.3.3 Prezentare Cod

.

```
129     def uniformCostSearch(problem: SearchProblem):
130         visited = set()
131         stack = util.PriorityQueue()
132         path = []
133         begin = problem.getStartState()
134         stack.push( item: (begin, path, 0),  priority: 0)
135         while not stack.isEmpty():
136             (node, nodePath, nodeCost) = stack.pop()
137             if node not in visited:
138                 visited.add(node)
139                 if problem.isGoalState(node):
140                     return nodePath
141                 successors = problem.getSuccessors(node)
142                 for i, action, cost in successors:
143                     stack.push( item: (i, nodePath + [action], nodeCost + cost), nod
144         return None
```

Figura 3: UNIFORM COST CODE

2.3.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste si a fost destul de usor de implementat.

3 Informed search

3.1 Question 4 - A* search algorithm

3.1.1 Definire cerinta

Implementeaza cautarea in graf A* in functia goala aStarSearch din search.py. A* primeste o functie euristica ca argument. Euristicile ii sunt transmise doua argumente: un stadiu(state) in problema de cautare (argumentul principal) si problema in sine (pentru informatii de referinta). Functia euristica nullHeuristic din search.py este un exemplu trivial. Poti testa implementarea ta A* pe problema initiala de gasire a unui traseu prin labirint catre o pozitie fixa folosind euristica distantei Manhattan (implementata deja ca manhattanHeuristic in searchAgents.py).

3.1.2 Prezentare algoritm/metoda

Se bazeaza pe aceeasi idee ca UCS refolosind codul de la UCS.Diferenta este data de calculul prioritatii fiind dat de suma distantei de la UCS adunata cu o anumita euristica(s-a folosit functia heuristic).

3.1.3 Optional Prezentare Cod

.

```
153  def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
154      visited = set()
155      stack = util.PriorityQueue()
156      path = []
157      begin = problem.getStartState()
158      stack.push( item: (begin, path, 0),  priority: 0)
159      while not stack.isEmpty():
160          (node, nodePath, nodeCost) = stack.pop()
161          if node not in visited:
162              visited.add(node)
163              if problem.isGoalState(node):
164                  return nodePath
165              successors = problem.getSuccessors(node)
166              for i, action, cost in successors:
167                  stack.push( item: (i, nodePath + [action], nodeCost + cost),
168                              problem.getCostOfActions(nodePath + [action]) + heu
169      return None
```

Figura 4: A* CODE

3.1.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.

3.2 Question 5 - Finding All the Corners

3.2.1 Definire cerinta

Puterea reala a algoritmului A* va deveni evidenta doar in fata unei probleme de cautare mai provocatoare. Acum este momentul sa formulam o problema noua si sa proiectam o euristica pentru aceasta. In labirinturile cu colturi, exista patru puncte, cate unul in fiecare colt . Noua noastra problema de cautare este sa gasim cel mai scurt traseu prin labirint care trece prin toate cele patru colturi (indiferent daca labirintul are sau nu hrana acolo). Observam ca pentru unele labirinturi precum tinyCorners, cel mai scurt traseu nu merge intotdeauna prima data la cea mai apropiata hrana! Indicatie: cel mai scurt traseu prin tinyCorners necesita 28 de pasi. Implementeaza problema de cautare CornersProblem in searchAgents.py. Va trebui sa alegi o reprezentare a starii care codifica toate informatiile necesare pentru a detecta daca au fost atinse toate cele patru colturi.

3.2.2 Prezentare algoritm/metoda

Pentru aceasta cerinta s-au adaugat ca atribute ale clasei: visitedCorners si gameState - startingGameState. Obiectivul este realizat atunci cand in atributul visitedCorners contine cele 4 colturi. Pentru obtinerea succesorilor s-a verificat daca potentialul succesor este un colt al mapei. Daca acesta este se adauga in lista cu colturi vizitate

3.2.3 Prezentare Cod

.

```
293     def getStartState(self):
294         "*** YOUR CODE HERE ***"
295         return (self.startingPosition, self.corners)
296
            2 usages (1 dynamic)
297     def isGoalState(self, state: Any):
298         "*** YOUR CODE HERE ***"
299
300         (position, corners) = state
301         if position not in corners:
302             return False
303         else:
304             if len(corners) == 1:
305                 return True
306             else:
307                 return False
308
```

.

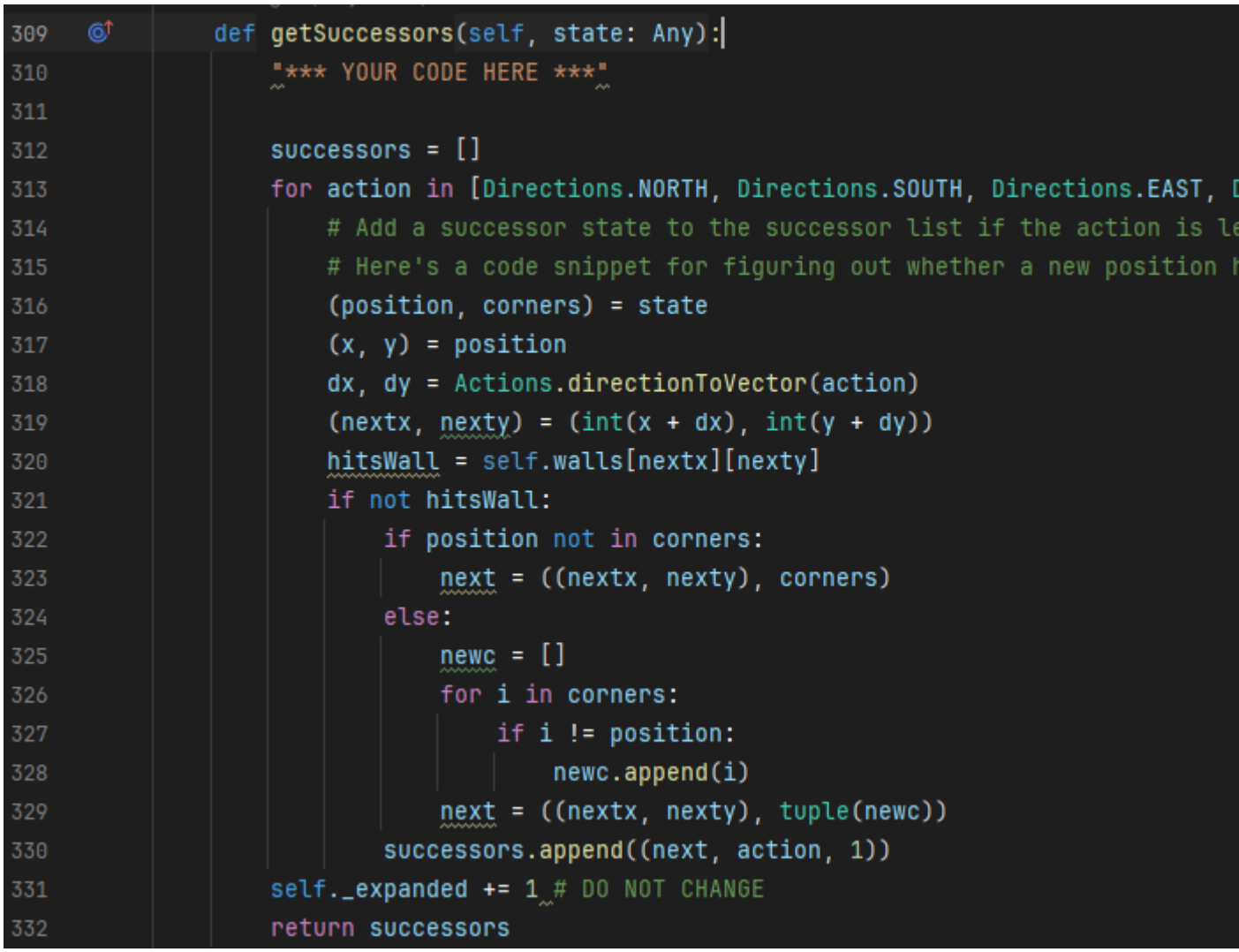

Figura 5: FINDING CORNERS CODE

3.2.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.

3.3 Question 6 - Corners Problem: Heuristic

3.3.1 Definire cerinta

Implementati o euristica non-triviala si consistenta pentru problema CornersProblem in functia cornersHeuristic. Admisibilitate vs. Consistenta: Amintiti-va, euristicile sunt doar functii care iau stari de cautare si returneaza numere care estimeaza costul catre scopul cel mai apropiat. Euristicile mai eficiente vor returna valori mai apropiate de costurile reale ale scopului. Pentru a fi admisibile, valorile euristice trebuie sa fie limite inferioare ale costului real al celui mai scurt traseu catre cel mai apropiat scop (si sa fie non-negative). Pentru a fi consistent, trebuie sa se mentina, in plus, ca daca o actiune are costul c, atunci efectuarea acelei actiuni poate cauza doar o scadere in euristica de cel mult c. Euristici non-triviale: Euristici triviale sunt cele care returneaza zero pretutindeni (pentru UCS) si euristica care calculeaza costul real de finalizare. Prima nu va economisi timp, in timp ce a doua va depasi limita de timp a grilajului automatizat.

Va doriti o euristica care sa reduca timpul total de calcul, desi pentru aceasta sarcina, grilajul automatizat va verifica doar numarul de noduri.

3.3.2 Prezentare algoritm/metoda

Euristica pentru aceasta problema s-a bazat pe distanta catre cel mai indepartat colt. Aceasta distanta fiind calculata prin mazeDistance

3.3.3 Prezentare Cod

.

```
9 usages (8 dynamic)
348 def cornersHeuristic(state: Any, problem: CornersProblem):
349     (pos, corners) = state
350     (x1, y1) = pos
351     distances = set()
352     for i in corners:
353         distance = abs(x1 - i[0]) + abs(y1 - i[1])
354         distances.add(distance)
355     if problem.isGoalState(state):
356         return 0
357     else:
358         return max(distances)
```

Figura 6: CORNERS HEURISTIC CODE

3.3.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.

3.4 Question 7 - Eating All the Dots

3.4.1 Definire cerinta

Acum vom rezolva o problemă de căutare dificilă: consumarea tuturor alimentelor de către Pacman în cât mai puți pași posibili. Pentru aceasta, avem nevoie de o nouă definiție a problemei de căutare care formalizează problema curățării alimentelor: FoodSearchProblem în searchAgents.py (implementată pentru tine). O soluție este definită ca o cale care colectează toate alimentele din lumea Pacman. Pentru proiectul actual, soluțiile nu iau în considerare fantomele sau pastilele de putere; soluțiile depind doar de amplasarea pereților, a alimentelor obținute și a lui Pacman.

3.4.2 Prezentare algoritm/metoda

Euristica pentru aceasta problema se bazeaza pe distanta catre cel mai indepartat punct care ofera un scor.

3.4.3 Prezentare Cod

.

```
422  def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProble
423      position, foodGrid = state
424      "*** YOUR CODE HERE ***"
425      foodGridList = foodGrid.asList()
426      heuristic = []
427      heuristic.append(0)
428      for i in foodGridList:
429          heuristic.append(mazeDistance(position, i, problem.startingGameState
430      return max(heuristic)
431
432
```

Figura 7: FOOD HEURISTIC CODE

3.4.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.

3.5 Question 8 - Suboptimal Search

3.5.1 Definire cerinta

Uneori, chiar și cu A* și o euristică bună, găsirea traseului optim prin toate punctele este dificilă. În aceste cazuri, totuși, ne-am dori să găsim rapid un traseu destul de bun. În această secțiune, vei scrie un agent care mănâncă întotdeauna lacom cel mai apropiat punct. ClosestDotSearchAgent este implementat pentru tine în searchAgents.py, dar lipsește o funcție cheie care găsește un traseu către cel mai apropiat punct. Implementează funcția findPathToClosestDot în searchAgents.py. Agentul nostru rezolvă acest labirint (în mod suboptimal!) în mai puțin de o secundă, cu un cost al traseului de 350.

3.5.2 Prezentare algoritm/metoda

Aceasta cerinta am rezolvat-o cu ajutorul indicatiilor oferite de site-ul Berkeley implementandu-se functia isGoalState si adaugarea in functia findPathToClosestDot a unui apel aStarSearch(problem), rezultatul fiind acelasi si pentru un apel al BFS sau al UCS.

3.5.3 Prezentare Cod

.

```
450     def findPathToClosestDot(self, gameState: pacman.GameState):
451
452         problem = AnyFoodSearchProblem(gameState)
453
454         "*** YOUR CODE HERE ***"
455         from search import aStarSearch
456         return aStarSearch(problem)
457
            1 usage (1 dynamic)
458     def isGoalState(self, state: Tuple[int, int]):
459         x, y = state
460
461         "*** YOUR CODE HERE ***"
462         if (x, y) in self.food.asList():
463             return True
464         else:
465             return False
```

Figura 8: SUBOPTIMAL SEARCH CODE

3.5.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim + 1 pe baza unor anumite teste.

4 Adversarial search

4.1 Question 1 - Improve the ReflexAgent

4.1.1 Definire cerinta

Îmbunătățește ReflexAgentul din multiAgents.py pentru a juca în mod respectabil. Codul furnizat pentru agentul reflex oferă câteva exemple utile de metode care interoghează GameState pentru informații. Un agent reflex capabil va trebui să ia în considerare atât locațiile alimentelor, cât și locațiile fantomelor pentru a performa bine.

4.1.2 Prezentare algoritm/metoda

Pentru abordarea problemei cu agenți adversariali, vom utiliza funcțiile getAction și evaluationFunction. getAction alege cea mai bună opțiune în funcție de ceea ce returnează evaluationFunction. Implementarea noastră pentru o funcție de evaluare mai bună calculează distanța de la Pac-Man până la cel mai apropiat agent (Ghost), folosind distanta absoluta, și distanța până la cea mai apropiată bucată de mâncare. Ambele distanțe sunt calculate în funcție de poziția actuală. Cu o simplă ramură if-elif, evităm să mergem în direcția fantomei și preferăm să ne îndreptăm spre mâncare.

4.1.3 Prezentare Cod

.

```
62          "*** YOUR CODE HERE ***"
63          # Lista de pozitii ale mancarii
64          foodList = newFood.asList()
65
66          # Lista de pozitii ale fantomelor
67          ghostPos = [ghost.getPosition() for ghost in newGhostStates]
68
69          score = 0
70
71          # Calculează distanțele până la hrana și fantomele în bucle
72          for i in foodList:
73              # Calculează distanța până la hrana curentă
74              foodDistance = abs(i[0] - newPos[0]) + abs(i[1] - newPos[1])
75
76              # Ajustează scorul în funcție de distanța până la hrana
77              if foodDistance == 0:
78                  score += 10
79              elif foodDistance == 1:
80                  score += 5
81              elif foodDistance == 2:
82                  score += 3
83              elif foodDistance == 3:
84                  score += 2
```

.

```
85          for i in ghostPos:
86              # Calculează distanța până la fantoma curentă
87              ghostDistance = abs(i[0] - newPos[0]) + abs(i[1] - newPos[1])
88              # Ajustează scorul în funcție de distanța până la fantomă
89              if ghostDistance < 4:
90                  if ghostDistance == 1:
91                      score -= 500
92                  else:
93                      score -= 10
94          # Verifică dacă acțiunea este o oprire și penalizează
95          if action == Directions.STOP:
96              score -= 10
97          # Dacă jocul este câștigat, returnează un scor mare
98          if successorGameState.isWin():
99              return 1000
100         # Adaugă diferența de scor
101         score += successorGameState.getScore() - currentGameState.getScore()
102
103         return score
```

Figura 9: REFLEX AGENT CODE

4.1.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.

4.2 Question 2 - MiniMax Algorithm

4.2.1 Definire cerinta

Acum vei dezvolta un agent de căutare adversarială în șablonul clasei MinimaxAgent furnizat în fișierul multiAgents.py. Agentul tău Minimax trebuie să fie capabil să funcționeze cu oricare număr de fantome, așa că va trebui să elaborezi un algoritm mai general decât ceea ce ai învățat anterior în curs. În special, arborele Minimax pe care îl vei construi va avea mai multe straturi min (unul pentru fiecare fantomă) pentru fiecare strat max.

4.2.2 Prezentare algoritm/metoda

Funcția maxvalue reprezintă rândul lui Pac-Man. Aceasta evaluează valoarea maximă posibilă pentru Pac-Man prin explorarea acțiunilor sale potențiale. Verifică dacă este la adâncimea dorită, dacă Pac-Man a câștigat sau a pierdut, sau dacă nu mai există adâncime de explorat. Dacă una dintre aceste condiții este îndeplinită, returnează evaluarea stării curente. În caz contrar, explorează în mod recursiv acțiunile posibile ale lui Pac-Man și valorile minime ulterioare din rândurile fantomelor.

Funcția minvalue, pe de altă parte, reprezintă rândurile fantomelor. Aceasta evaluează valoarea minimă posibilă pentru fantome prin considerarea acțiunilor lor potențiale. Similar cu

maxvalue, verifică adâncimea, condițiile de câștig/pierdere și efectuează explorarea recursivă a acțiunilor posibile pentru fantome. Dacă există mai multe fantome, apelează minvalue pentru următoarea fantomă; altfel, calculează valoarea maximă pentru Pac-Man.

În final, codul iterează prin acțiunile legale ale lui Pac-Man, apelând minvalue pentru fiecare acțiune pentru a găsi scorurile asociate acestora. Alege acțiunea cu scorul maxim, simulând mutarea lui Pac-Man pe baza valorilor calculate din algoritmul minimax.

Această implementare folosește un algoritm minimax cu limitare de adâncime, evaluând stările jocului până la o anumită adâncime pentru a determina cea mai bună acțiune pentru Pac-Man.

4.2.3 Prezentare Cod

.

```
141  class MinimaxAgent(MultiAgentSearchAgent):
         4 usages (4 dynamic)
142      def getAction(self, gameState: GameState):
143          legalActions = gameState.getLegalActions(0)  # Actiuni legale
144          scores = [self.mini(gameState.generateSuccessor( agentIndex: 0, action), s
145
146          # Alege actiunea cu scorul maxim
147          bestScore = max(scores)
148          bestIndices = [index for index in range(len(scores)) if scores[index] =
149          chosenIndex = random.choice(bestIndices) if bestIndices else 0  # Defau
150          return legalActions[chosenIndex]
151
         1 usage
152      def maxi(self, state, depth):
153          if depth == 0 or state.isWin() or state.isLose():
154              return self.evaluationFunction(state)
155
156          v = float('-inf')
157          for action in state.getLegalActions(0):  # Actiunile lui Pacman
158              successor = state.generateSuccessor(0, action)
159              v = max(v, self.mini(successor, depth,  ghostIndex: 1))
160          return v
161
```

.

```
162     def mini(self, state, depth, ghostIndex):
163         if depth == 0 or state.isWin() or state.isLose():
164             return self.evaluationFunction(state)
165 
166         v = float('inf')
167         for action in state.getLegalActions(ghostIndex):
168             successor = state.generateSuccessor(ghostIndex, action)
169 
170             # Dacă există mai mulți fantome, apelează min_value cu următorul in
171             if ghostIndex < state.getNumAgents() - 1:
172                 v = min(v, self.mini(successor, depth, ghostIndex + 1))
173             else:
174                 v = min(v, self.maxi(successor, depth - 1))  # Tura lui Pacman
175 
176         return v
```

Figura 10: MINIMAX CODE

4.2.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.

4.3 Question 3 - Alpha-Beta Pruning

4.3.1 Definire cerinta

În cadrul provocării, trebuie să creezi un agent nou, AlphaBetaAgent, care să utilizeze alpha-beta pruning pentru a explora mai eficient arborele Minimax. Algoritmul tau trebuie să fie suficient de general încât să funcționeze corespunzător pentru mai mulți agenți și minimizatori. Extinderea logică a tăierii alpha-beta trebuie gestionată în mod corespunzător pentru a accelera explorarea arborelui.

Implementarea eficientă a alpha-beta pruning poate duce la o accelerare semnificativă a algoritmului Minimax, ceea ce poate însemna că o adâncime 3 a alpha-beta poate rula la fel de rapid sau chiar mai rapid decât o adâncime 2 a Minimax. Scopul final este ca adâncimea 3 pe o hartă precum smallClassic să ruleze în doar câteva secunde pe mutare sau chiar mai rapid.

Astfel, provocarea constă în adaptarea corectă a algoritmului Minimax pentru a beneficia de tăierea alpha-beta. Aceasta implică gestionarea corespunzătoare a valorilor alpha și beta, actualizarea acestora pe măsură ce arborele este explorat și tăierea corespunzătoare pentru a elimina secțiuni inutile ale arborelui. O implementare eficientă va optimiza explorarea și va îmbunătăți semnificativ timpul de răspuns al agentului pe hărți mai mari sau la adâncimi mai mari ale arborelui de joc.

4.3.2 Prezentare algoritm/metoda

Alpha-Beta pruning este o optimizare a algoritmului Minimax care elimină anumite noduri din explorarea arborelui, fără a afecta decizia finală, ceea ce face algoritmul mai eficient. În

implementarea acestei optimizări, am extins algoritmul Minimax de la întrebarea anterioară prin adăugarea a câteva modificări.

Funcțiile maxvalue și minvalue acum primesc doi parametri suplimentari, alpha și beta. Acești parametri reprezintă cele mai bune valori găsite pentru jucătorii care maximizează, respectiv minimizează. În timpul traversării arborelui de joc, atunci când se găsește o valoare (v) care depășește valoarea beta în maxvalue sau cade sub valoarea alpha în minvalue, înseamnă că ramura curentă poate fi eliminată. Prin urmare, căutarea în acea ramură poate fi oprită, deoarece nu va afecta decizia finală. În funcția principală getAction, alpha și beta sunt inițializate ca infinit negativ și pozitiv, respectiv, și sunt actualizate în timpul procesului de căutare. Algoritmul ține evidența celei mai bune alegeri găsite până în acel moment și elimină ramurile în consecință, folosind valorile alpha-beta.

4.3.3 Optional Prezentare Cod

.

```
188         def terminalState(state, adancime, apel_max):
189             if apel_max == 1:
190                 if state.isWin() or state.isLose() or adancime == self.depth:
191                     return True
192                 else:
193                     return False
194             else:
195                 if state.isWin() or state.isLose():
196                     return True
197                 else:
198                     return False
199 
200         # folosim pentru pacman deci AgentIndex va fi 0 mereu
201         def maxScor(state, adancime, alpha, beta):
202             if terminalState(state, adancime, apel_max: 1):
203                 return self.evaluationFunction(state), ''
204             else:
205                 scor = -100000
206                 mutare = ''
207                 alpha_curent = alpha
208                 for action in state.getLegalActions(0):
209                     scor_temporar, nimic = minScor(state.generateSuccessor(0, a
210                     if scor_temporar > scor:
211                         scor = scor_temporar
212                         mutare = action
```

.

```
                    if scor > alpha_curent:
                        alpha_curent = scor
                    if scor > beta:
                        return scor, mutare
                return scor, mutare

            # folosim pentru fantome deci vom avea AgentIndex >= 1
            def minScor(state, index_agent, adancime, alpha, beta):
                if terminalState(state, adancime, apel_max: 0):
                    return self.evaluationFunction(state), ''
                else:
                    scor = 100000
                    mutare = ''
                    beta_curent = beta
                    for action in state.getLegalActions(index_agent):
                        scor_temporar = 0
                        if index_agent == (state.getNumAgents() - 1):
                            scor_temporar, nimic = maxScor(state.generateSuccessor
                                                           alpha, beta_curent)
                        else:
                            scor_temporar, nimic = minScor(state.generateSuccessor
                                                           adancime, alpha, beta_c
                        if scor_temporar < scor:
                            scor = scor_temporar
                            mutare = action
```

.

```
                    if beta_curent > scor:
                        beta_curent = scor
                    if scor < alpha:
                        return scor, mutare
                return scor, mutare

            # cel mai de sus apel al algoritmului
            scor, final_action = maxScor(gameState, adancime: 0, -100000, beta: 1000
            return final_action
```

Figura 11: ALPHA-BETA-PRUNING CODE

4.3.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.

4.4 Question 4 - Expectimax

4.4.1 Definire cerinta

Minimax și alpha-beta sunt grozave, dar ambele presupun că joci împotriva unui adversar care ia decizii optime. După cum vă poate spune oricine care a câștigat vreodată tic-tac-toe, acesta nu este întotdeauna cazul. În această întrebare veți implementa ExpectimaxAgent, care este util pentru modelarea comportamentului probabilistic al agenților care pot face alegeri suboptime.

4.4.2 Prezentare algoritm/metoda

Acest algoritm implementează o căutare Expectimax, care este o variantă a algoritmului Minimax, folosit în probleme de decizie și jocuri. În acest caz, pare să fie destinat unui joc, deoarece este menționat un obiect gameState care probabil reprezintă starea jocului.Verifică dacă jocul a fost câștigat, pierdut sau a atins adâncimea maximă.Alege acțiunea care maximizează valoarea dintre acțiunile legale posibile.Calculează valoarea așteptată a stărilor succesive, ponderată în funcție de șansele fiecărei acțiuni.

4.4.3 Optional Prezentare Cod

.

```
257         def expectimax(gameState, agentIndex, depth=0):
258             legalActionList = gameState.getLegalActions(agentIndex)
259             numIndex = gameState.getNumAgents() - 1
260             bestAction = None
261             # If terminal(pos)
262             if (gameState.isLose() or gameState.isWin() or depth == self.depth)
263                 return [self.evaluationFunction(gameState)]
264             elif agentIndex == numIndex:
265                 depth += 1
266                 childAgentIndex = self.index
267             else:
268                 childAgentIndex = agentIndex + 1
269 
270             numAction = len(legalActionList)
271             # if player(pos) == MAX: value = -infinity
272             if agentIndex == self.index:
273                 value = -float("inf")
274             # if player(pos) == CHANCE: value = 0
275             else:
276                 value = 0
277 
278             for legalAction in legalActionList:
279                 successorGameState = gameState.generateSuccessor(agentIndex, le
280                 expectedMax = expectimax(successorGameState, childAgentIndex, d
```

.

```
281                 if agentIndex == self.index:
282                     if expectedMax > value:
283                         value = expectedMax
284                         bestAction = legalAction
285                 else:
286                     value = value + ((1.0 / numAction) * expectedMax)
287             return value, bestAction
288 
289         bestScoreActionPair = expectimax(gameState, self.index)
290         bestScore = bestScoreActionPair[0]
291         bestMove = bestScoreActionPair[1]
292         return bestMove
```

Figura 12: EXPECTIMAX CODE

4.4.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.

4.5 Question 5 - Evaluation Function

4.5.1 Definire cerinta

Scrieți o funcție de evaluare mai bună pentru pacman în funcția furnizată betterEvaluationFunction. Funcția de evaluare ar trebui să evalueze stările, mai degrabă decât acțiunile ca funcția de evaluare a agentului reflex. Cu căutarea adâncime 2, funcția dvs. de evaluare ar trebui să șterge aspectul smallClassic cu o fantomă aleatorie mai mult de jumătate din timp și să ruleze în continuare la o rată rezonabilă (pentru a obține credit complet, Pacman ar trebui să aibă o medie de aproximativ 1000 de puncte când câștigă).

4.5.2 Prezentare algoritm/metoda

În această funcție de evaluare, obiectivul este să atribuim un scor stării curente a jocului, ținând cont de diverse componente și factori relevanți pentru jocul Pac-Man. Vom trece prin pașii principali ai algoritmului:Stabilim ponderi pentru diferite componente ale evaluării, cum ar fi mancarea, fantomele și distanța, calculăm distanța până la cea mai apropiată mâncare și adăugăm la scorul general, pentru fiecare fantomă, calculăm distanța până la ea, iar,dacă este foarte aproape, penalizăm scorul si in final funcția returnează scorul final.

4.5.3 Optional Prezentare Cod

.

```python
def betterEvaluationFunction(currentGamState: GameState):
    pozitiaPacman = currentGamState.getPacmanPosition()
    pozitiiMancare = currentGamState.getFood().asList()
    stariFantomelor = currentGamState.getGhostStates()

    pondereMancare = 10
    pondereFantome = -100
    pondereDistanta = -1
    # Calculeaza scorul bazat pe diferiti factori
    scor = currentGamState.getScore()
    # Evalueaza pozitiile mancarii
    distanteMancare = [manhattanDistance(pozitiaPacman, mancare) for mancare i
    if distanteMancare:
        ceaMaiApropiataDistantaMancare = min(distanteMancare)
        scor += pondereMancare / ceaMaiApropiataDistantaMancare
    # Evalueaza pozitiile fantomelor
    for stareFantoma in stariFantomelor:
        pozitieFantoma = stareFantoma.getPosition()
        distantaFantoma = manhattanDistance(pozitiaPacman, pozitieFantoma)
        if distantaFantoma < 2:
            scor += pondereFantome
    return scor
# Restul codului
better = betterEvaluationFunction
```

Figura 13: BETTER EVALUATION FUNCTION CODE

4.5.4 Comentarii/Observatii

Codul implementat a obtinut punctajul maxim pe baza unor anumite teste.