Universitatea Tehnică din Cluj-Napoca
Facultatea de Automatică și Calculatoare

Documentație

Proiect Laborator PACMAN
Project 1 + Project 2

Inteligență Artificială

Student: Maria-Magdalena Creț

Grupa 30223

Anul Universitar 2024-2025

Cuprins

1 Introducere

Pentru început (Project 1), se dorește implementarea jocului Pacman, unde propriul agent Pacman va găsi trasee prin lumea sa labirintică, atât pentru a ajunge la o anumită locație, cât și pentru a colecta mâncare într-un mod eficient. Scopul principal e acela de a construi și înțelege algoritmi de căutare generală, ce vor fi aplicați în scenarii specifice Pacman.

Proiectul total (Project 1 + Project 2) include un autograder pentru a se putea realiza evaluăriile proprii direct pe computerul personal. Și se poate rula cu comanda : *python autograder.py*

Totodată proiectul este împărțit pe mai multe secțiuni. Acestea sunt următoarele:

- Search
- Multi-Agent Search

Secțiunea de *Search* conține mai multe cerințe de îndeplinit (algoritmi de implementat, respectiv și euristici):

1. Q1: Finding a Fixed Food Dot using Depth First Search
2. Q2: Breadth First Search
3. Q3: Varying the Cost Function
4. Q4: A* search
5. Q5: Finding All the Corners
6. Q6: Corners Problem: Heuristic
7. Q7: Eating All The Dots
8. Q8: Suboptimal Search

Secțiunea de *Multi-Agent Search* face referire la Project 2, unde se concep agenți pentru versiunea clasică a jocului Pacman, inclusiv fantomele. Aici se implementează atât căutarea minimax, cât și căutarea expectimax, respectiv se creează funcții de evaluare. Această secțiune are următoarele cerințe:

1. Q1: Reflex Agent
2. Q2: Minimax
3. Q3: Alpha-Beta Pruning
4. Q4: Expectimax
5. Q5: Evaluation Function

2 Search

2.1 Q1: Finding a Fixed Food Dot using Depth First Search

Căutarea în adâncime (Depth-First Search) explorează un graf prin vizitarea unui nod și parcurgerea în mod recursiv a vecinilor săi nevizitați, pătrunzând mai adânc în graf până când nu mai există noduri nevizitate de-a lungul traseului curent. Apoi, revine înapoi și continuă explorarea altor ramuri. DFS este adesea implementat folosind o structură de date de tip stivă și este potrivită pentru căutarea în grafuri sau arbori mari și denși.

Pentru început se inițializează elementele dinamice ale algoritmului DFS. **nodes** este lista care urmărește toate nodurile vizitate până în acel moment și este inițializată cu poziția de start a personajului Pacman. Totodată **path** este o listă goală care va fi completată cu traseul de la nodul de start până la obiectiv, iar **node_to_start** este nodul curent și reprezintă nodul de start al lui Pacman. **stack_dfs** este o stivă, care este structura de date principală utilizată în DFS pentru a menține nodurile care nu au fost încă explorate. S-a utilizat *list comprehension* pentru a se obține succesorii nodului curent și pentru a se adăuga fiecare succesor împreună cu calea sa extinsă în stiva DFS.

Observație: Pentru DFS, complexitatea timpului este $O(N+M)$, iar complexitatea spațială este $O(N)$. Se presupune că există N noduri și M muchii.

Codul are următoarea implementare:

```
def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
    path = []
    nodes = []
    node_to_start = problem.getStartState()
    stack_dfs = util.Stack()
    stack_dfs.push((node_to_start, path))
    while True and not problem.isGoalState(node_to_start):
        if stack_dfs.isEmpty():
            break
        node, path = stack_dfs.pop()

      if node not in nodes:
        if problem.isGoalState(node):  return path
            nodes.append(node)
            successors = problem.getSuccessors(node)
            pathc_dfs = [(s[0], path + [s[1]]) for s in successors]
            for item in pathc_dfs:
                stack_dfs.push(item)
    return []
    util.raiseNotDefined()
```

Exemplul de mai jos are loc pentru comanda:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
```

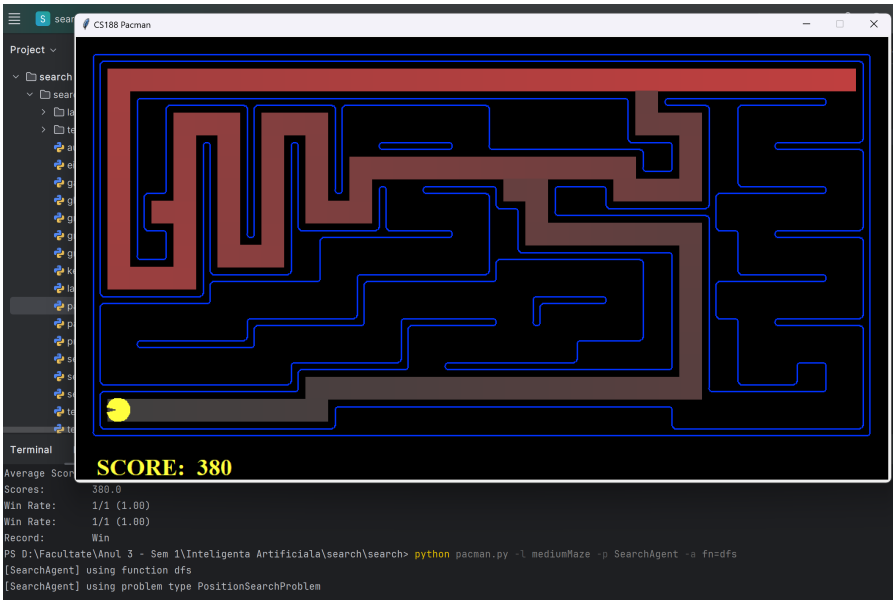


Figura 2.1.1: Depth First Search Algorithm

2.2 Q2: Breadth First Search

Căutarea în lățime (Breadth-First Search) explorează un graf vizitând toți vecinii unui nod înainte de a trece la vecinii acestora. BFS este, de obicei, implementată folosind o structură de date de tip coadă și este ideală pentru căutarea în grafuri sau arbori mari și rarefiati.

Algoritmul de căutare în lățime începe tot cu inițializarea. **queue_bfs** este coada care conține nodurile ce urmează să fie explorate. Totodată **path** este o listă goală care va fi completată cu traseul de la nodul de start până la obiectiv, iar **node_to_start** este nodul curent și reprezintă nodul de start al lui Pacman. Aceasta este inițializată cu poziția de start a personajului Pacman. **Nodes** este lista care urmărește toate nodurile vizitate până în acel moment. S-a utilizat *list comprehension* pentru a se obține succesorii nodului curent și pentru a se adăuga fiecare succesor împreună cu calea sa extinsă în coada BFS.

Observație: Complexitatea timpului pentru algoritmul BFS este $O(N + M)$, iar complexitatea spațială este $O(N)$. Se presupune că există N noduri și M muchii.

Codul are următoarea implementare:

```
def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    path = []
    nodes = []
    node_to_start = problem.getStartState()
    queue_bfs = util.Queue()
    queue_bfs.push((node_to_start, path))
    while True and not problem.isGoalState(node_to_start):
        if queue_bfs.isEmpty():
            break
        node, path = queue_bfs.pop()
```

```
        if node not in nodes:
            if problem.isGoalState(node):  return path
            nodes.append(node)
            successor = problem.getSuccessors(node)
            pathc_bfs = [(s[0], path + [s[1]]) for s in successor]
            for item in pathc_bfs:
                queue_bfs.push(item)
    return []
    util.raiseNotDefined()
```

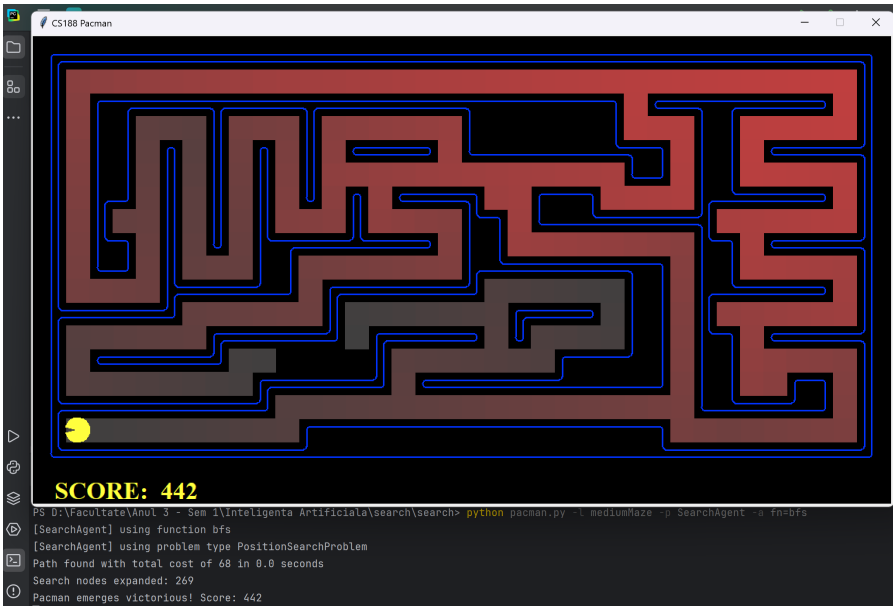


Figura 2.2.1: Breadth First Search Algorithm

Exemplul de mai sus are loc pentru comanda:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

2.3 Q3: Varying the Cost Function

Dacă Pacman este într-un labirint cu diferite drumuri, se cunoaște că unele drumuri sunt mai sigure decât altele. Pacman folosește algoritmul UCS pentru a găsi drumul care îl costă cel mai puțin dintre toate cele existente pentru a ajunge la țintă. Cu toate că este posibil ca acesta să nu fie cel mai scurt. Mai precis, Pacman folosește UCS pentru a alege drumul care îl costă cel mai puțin în funcție de riscuri și recompense.

De exemplu:

1. Dacă drumul ales să fie parcurs este sigur, costul este mai mic.
2. Dacă drumul ales este plin de fantome, costul este mai mare.

Căutarea cu cost uniform (Uniform-Cost Search) este o variantă a algoritmului lui Dijkstra. Aici, în loc să se insereze toate vârfurile într-o coadă de prioritate, se inserează doar sursa și apoi, câte unul, pe măsură ce este necesar. La fiecare pas, se verifică dacă elementul se află deja în coada de prioritate.

Algoritmul funcționează astfel:

a. Se inițializează o coadă de priorități cu starea de start și costul 0.

b. Se extrage nodul cu cel mai mic cost total din coadă.

c. Dacă nodul extras este ținta, se returnează drumul.

d. Dacă nu este ținta, se adaugă succesorii nodului în coadă, respectiv costurile cumulate a acestor noduri.

e. Se repetă pașii până când se găsește drumul optim sau coada devine goală.

În acest algoritm, din starea inițială, se vizitează stările adiacente și se alege starea cu cel mai mic cost, apoi se alege următoarea starea cu cel mai mic cost dintre toate stările nevizitate și adiacente stărilor vizitate. În acest mod, se încerca să se ajungă la starea țintă, chiar dacă se ajunge la starea țintă, se continua căutarea altor posibile trasee (în cazul în care există mai multe astfel de trasee). În coada de prioritate se reține starea cu cel mai mic cost dintre toate stările adiacente celor deja vizitate.

S-a utilizat *list comprehension* în codul de mai jos, prelucând o listă de succesiuni urmând să se genereze o listă de elemente cu cost total și calculând valoarea heuristică pentru fiecare element. Apoi adaugă aceste elemente într-o coadă de priorități **queue_ucs** pe baza valorii euristice calculate.

Codul are următoarea implementare:

```
def uniformCostSearch(problem: SearchProblem) -> List[Directions]:
    path = []
    nodes = []
    node_to_start = problem.getStartState()
    queue_ucs = util.PriorityQueue()

    queue_ucs.push((node_to_start, path, 0), 0)
    while True and not problem.isGoalState(node_to_start):
        if queue_ucs.isEmpty():
            break
        node, path, cost = queue_ucs.pop()

        if node not in nodes:
            if problem.isGoalState(node):  return path

            nodes.append(node)
            successor = problem.getSuccessors(node)
            pathc_ucs = [((s[0], path + [s[1]], cost + s[2]),
            cost + s[2]) for s in successor]
```

```
            for item, priority in pathc_ucs:
                queue_ucs.push(item, priority)

    return []
    util.raiseNotDefined()
```

Exemplul de mai jos are loc pentru comanda:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

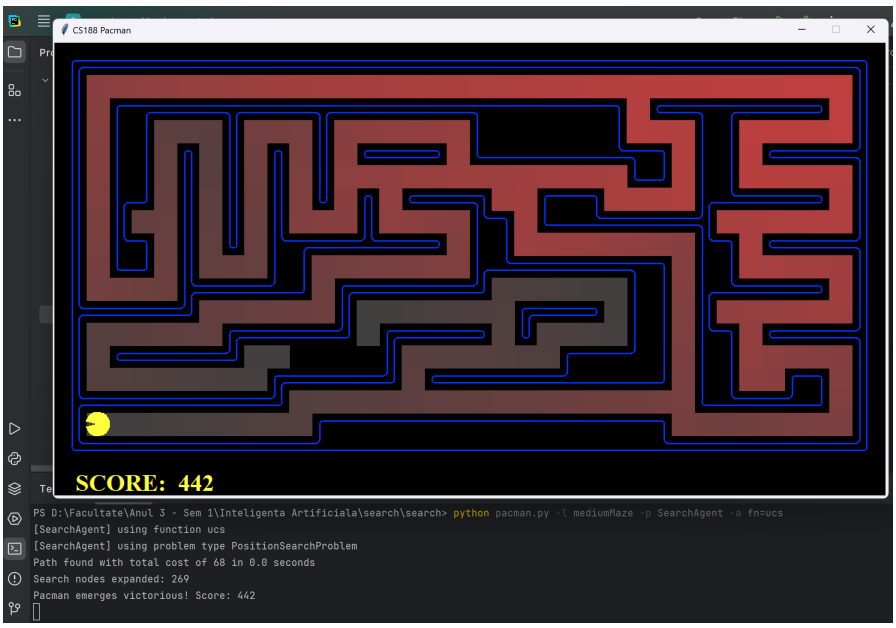


Figura 2.3.1: Uniform-cost graph Search Algorithm

Observație: Complexitatea timpului în cazul UCS depinde de numărul de noduri și de numărul de muchii din graf. Dacă presupunem că există N noduri și M muchii, complexitatea timpului este $O((N + M) \log N)$, având în vedere că pentru fiecare nod și muchie se poate face o operație de inserare/ștergere într-o coadă de priorități, care are complexitate $O(\log N)$.

Complexitatea spațială este dominată de stocarea nodurilor în coada de priorități și a celor vizitate, astfel încât complexitatea spațială este $O(N)$.

2.4 Q4: A* search

Dacă Pac-Man este într-un labirint și trebuie să ajungă la o țintă. În acest labirint, Pac-Man vrea să găsească cel mai scurt drum pentru a ajunge la țintă, lucru care trebuie realizabil fără pierderi sau ocoluri care nu au sens.

În utilizarea agoritmului A*, Pacman alege direcția care pare cea mai favorabilă, ținând cont, totodată, de distanța pe care o mai are de parcurs. Pacman analizează traseele disponibile și îl alege pe cel care i se pare cel mai rapid. Dacă există un drum mai lung, dar care oferă

o soluție mai rapidă pe termen lung, Pacman îl va alege, știind că va fi mai eficient pe tot parcursul călătoriei.

Algoritmul funcționează astfel:

a. Se inițializează o coadă de priorități cu starea de start.

b. Se extrage nodul cu valoarea f(n) cea mai mică din coadă.

c. Dacă nodul extras este ținta, se returnează drumul.

d. Dacă nu este ținta, se adaugă succesorii nodului în coadă cu costul f(n) actualizat.

e. Se repetă pașii până când se găsește drumul optim sau coada devine goală.

BFS și DFS sunt metode de parcurgere a tuturor nodurilor din întregul graf, fără o „destinație" exactă. Dijkstra face același lucru, dar destinația lui este reprezentată de fiecare nod.

Codul are următoarea implementare:

S-a implementat o funcție de reconstruire a drumului de la un nod final (goal) la un nod de început (start) pe baza unei structuri visited.

Metoda urmărește nodurile părinte ale fiecărui nod din structura `visited` pentru a reconstrui traseul parcurs. Acest traseu este compus din acțiunile efectuate (ex. *stânga*, *dreapta*, *sus*, *jos*) pentru a ajunge de la `start` la `goal`.

```
def reconstructpath_function(visited, start, goal):
    path = []
    while goal != start:
        parent, action = visited[goal][1], visited[goal][2]
        path.append(action)
        goal = parent
    return path[::-1]
```

Implementare algoritmului A* este următoarea:

```
start = problem.getStartState()
visited = dict()
visited[start] = (0, None, '')
startQueue = util.PriorityQueue()
startQueue.push(start, 0)
open = []
closed = []
listA = {}
open.append(start)
listA[start] = 0

while not startQueue.isEmpty():
    node = startQueue.pop()

    if problem.isGoalState(node):
        return reconstructpath_function(visited, start, node)
```

```
        for neighbor in problem.getSuccessors(node):
            neighborPosition, direction, cost = neighbor

            # costul total
            costT = visited[node][0] + cost
            heuristic_value = heuristic(neighborPosition, problem)
            a = costT + heuristic_value

            if neighborPosition not in listA
                            or listA[neighborPosition] > a:
                listA[neighborPosition] = a
                startQueue.update(neighborPosition, a)
                visited[neighborPosition] = (costT, node, direction)
            closed.append(node)
```

Afișarea jocului Pacman are loc pentru comanda:

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a
↪  fn=astar,heuristic=manhattanHeuristic
```

Figura 2.4.1: A* search Algorithm

Observație: Complexitățiile sunt similare cu UCS. Timp: $O((N+M)\log N)$ Spațiu: $O(N)$. Se presupune că există N noduri și M muchii.

2.5 Q5: Finding All the Corners

Problema **CornersProblem** presupune ca scop colectarea a patru puncte de mâncare situate în colțurile labirintului, în contrast cu A*, unde obiectivul era colectarea unui singur punct pe cel mai scurt drum.

O provocare importantă constă în determinarea momentului în care toate cele patru colțuri au fost vizitate (sau, altfel spus, când Pacman ajunge în starea finală) într-o reprezentare abstractă a stării. Pentru a rezolva acest aspect, am ales să reprezentăm stările printr-un tuplu imbricat, de forma:

$$((x, y), (\text{colțuri\_de\_vizitat}))$$

În această structură, (colțuri_de_vizitat) este un tuplu imbricat care indică ce colțuri mai sunt de vizitat, iar (x, y) reprezintă poziția curentă a lui Pacman. Fiecare stare este o combinație între poziția lui Pacman în labirint și starea (vizitat/nevizitat) fiecărui colț.

Comenziile de afișare sunt:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

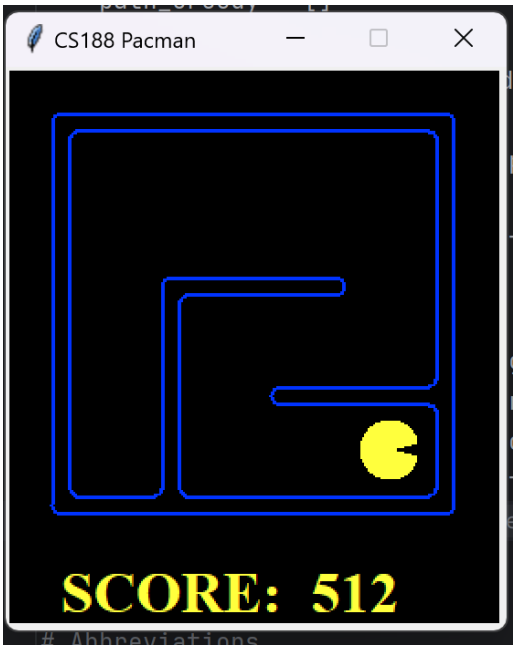


Figura 2.5.1: Finding All the Corners Algorithm

Din punct de vedere al codului, s-a implementat metoda de obținere a succesorului unui nod pentru utilizarea sa în algoritmul de Finding All the Corners, în următorul mod:

```
def getSuccessors(self, state: Any):

    successors = []
     directions = [(Directions.NORTH, (0, 1)),
     (Directions.SOUTH, (0, -1)),
     (Directions.EAST, (1, 0)),
     (Directions.WEST, (-1, 0))]

    currentPosition, visitedCorners = state
    for action, (dx, dy) in directions:
            "*** YOUR CODE HERE ***"
```

```
x, y = currentPosition
nextx, nexty = int(x + dx), int(y + dy)

if not self.walls[nextx][nexty]:
    nextPosition = (nextx, nexty)
    newVisitedCorners = list(visitedCorners)

if nextPosition in self.corners:
    cornerIndex = self.corners.index(nextPosition)
    newVisitedCorners[cornerIndex] = True

successors.append(((nextPosition, tuple(newVisitedCorners)),
action, 1))

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

2.6 Q6: Corners Problem: Heuristic

Se va dezvolta propria funcție euristică pentru a economisi mai mult timp în timpul căutării. Deoarece este o problemă de căutare pe grafuri, trebuie să se proiecteze o euristică nu doar admisibilă, ci și consistentă.

Euristica s-a construit în următorul mod:

- Se utilizează distanța Manhattan între două poziții, ignorând toți pereții din hartă.
- Dacă s-a ajuns deja într-un colț, trebuie doar să ne deplasăm de-a lungul marginilor hărții pentru a atinge toate colțurile nevizitate. În cea mai ideală condiție, se parcurge mai întâi latura scurtă a hărții pentru a atinge al doilea colț nevizitat, apoi se parcurge latura lungă pentru al treilea colț, iar în final ne deplasăm pe latura scurtă pentru a ajunge la ultimul colț nevizitat. Se concentrează doar asupra numărului colțurilor nevizitate, nu pe pozițiile lor.
- Se calculează distanțele Manhattan dintre poziția curentă și colțurile nevizitate pentru a găsi cea mai mică distanță, care se adaugă la valoarea euristicii. După actualizarea valorii euristicii, se înlocuiește poziția curentă cu cel mai apropiat colț nevizitat și se repetă procesul până când toate colțurile sunt vizitate.

Procesul iterativ pentru găsirea colțului cel mai apropiat: În timp ce există colțuri nevizitate, algoritmul alege colțul cel mai apropiat (în termeni de distanță Manhattan) și actualizează poziția curentă pentru a continua căutarea.

Sumarea distanțelor către colțurile nevizitate: Heuristica nu presupune neapărat traseul complet optim, ci oferă o aproximare admisibilă adunând distanțele până la colțurile nevizitate în ordine.

Comanda de afișare este:

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

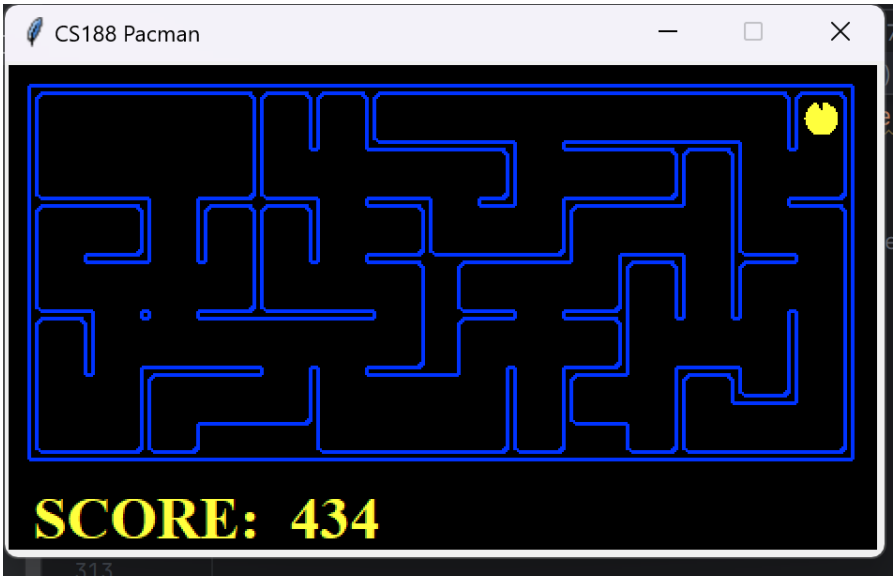


Figura 2.6.1: Corners Heuristic Problem

2.7 Q7: Eating All The Dots

Pentru a complica și mai mult situația, Pacman trebuie acum să colecteze toate punctele din labirint, nu doar să ajungă în colțuri. În această situație, efectul căutării A* devine și mai evident.

S-a implementat o euristică care combină două caracteristici pentru a estima costul rămas al problemei de colectare a mâncării:

1. *Cea mai lungă distanță între două puncte de mâncare:* Codul calculează toate distanțele Manhattan dintre perechi de puncte de mâncare (folosind două bucle `for`) și reține perechea de puncte care are cea mai mare distanță. Aceasta estimează costul minim necesar pentru a traversa punctele cele mai îndepărtate.
2. *Distanța minimă dintre poziția curentă și unul dintre punctele de mâncare cele mai îndepărtate:* După ce determină perechea de puncte de mâncare cele mai îndepărtate, codul calculează distanța Manhattan dintre poziția curentă a lui Pacman și aceste două puncte. Dintre aceste două valori, se alege cea mai mică, reprezentând costul minim pentru a începe să traverseze zona alimentelor.

Această euristică prezintă următoarele aspecte:

- Utilizează distanța cea mai lungă între două puncte de mâncare pentru a aproxima costul final.
- Adaugă distanța minimă de la poziția curentă a lui Pacman la unul dintre aceste puncte.

Admisibilitate: Este admisibilă, deoarece calculează o limită inferioară realistă a costului rămas, fără a supraestima costurile reale.

Consistență: Se bazează pe distanța Manhattan, care respectă inegalitatea triunghiului, aspect care o face să fie consistentă.

Implementarea funției foodHeuristic este următoarea:

```
"*** YOUR CODE HERE ***"
  if problem.isGoalState(state):
      return 0

  foodList = foodGrid.asList()
  maxDistance = 0

  first = foodList[0]
  second = foodList[0]
  for i in range(len(foodList)):
      for j in range(i + 1, len(foodList)):
          dist = util.manhattanDistance(foodList[i], foodList[j])
          if dist > maxDistance:
              maxDistance = dist

              first = foodList[i]
              second = foodList[j]

  return maxDistance + min((util.manhattanDistance(position, first),
  util.manhattanDistance(position, second)))
```

Comanda de afișare este:

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

2.8 Q8: Suboptimal Search

Este de precizat că până și o euristică bună nu ar reuși să găsească traseul optim într-un timp scurt. Așadar, este mai realistă ideea de a se găsi un traseu rezonabil, chiar dacă nu ar fi la fel de bun ca cel optim. Acest lucru trebuind să se realizeze într-un timp scurt.

Un astfel de agent este cel care mănâncă întotdeauna punctul cel mai apropiat de Pacman. Testarea stării de obiectiv a problemei `AnyFoodSearchProblem` poate fi realizată folosind deja funcția definită `gameState.getFood()`. În plus, funcția `findPathToClosestDot` poate fi construită simplu utilizând algoritmii de căutare (de exemplu, BFS, UCS, A*) implementați anterior. S-a ales implementarea cu BFS pentru această funcție. Acest lucru se poate oberva în codul de mai jos:

```
def findPathToClosestDot(self, gameState: pacman.GameState):

    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    return search.bfs(problem)
```

Totodată, implementarea acestui algoritm se realizează prin testarea stării de obiectiv a problemei `AnyFoodSearchProblem`,așa cum s-a menționat mai sus:

```
def isGoalState(self, state: Tuple[int, int]):

    x,y = state

    "*** YOUR CODE HERE ***"
        if (self.food[x][y]):
            return True
        return False
```

Comanda de afișare este:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

2.9 Q suplimentar: Greedy Best First Search

Acest algoritm încearcă să expandeze, la fiecare pas, nodul care pare cel mai aproape de starea finală (*goal*), cu scopul de a găsi rapid o soluție. Evaluarea nodurilor se bazează exclusiv pe funcția euristică $f(n) = h(n)$, fără a lua în calcul costul drumului parcurs până la nod.

Diferența majoră între *Greedy Best First Search* și A constă în faptul că algoritmul A include și costul acumulat până la nod ($g(n)$) în funcția de evaluare:

$$f(n) = g(n) + h(n).$$

Algoritmul funcționează astfel: inițial, se expandează nodul de start, se calculează valoarea euristicii pentru fiecare succesor, iar aceștia sunt adăugați într-o coadă de priorități. Procesul continuă prin extragerea și expansiunea nodului cu cea mai mare prioritate (cel mai mic $h(n)$). În cazul cel mai defavorabil (*worst case*), complexitatea algoritmului este $O(b^m)$, unde b reprezintă factorul de ramificare, iar m adâncimea arborelui.

Codul are următoarea implementare:

S-a utilizat o listă pentru nodurile explorate *exploredNodes* pentru a ține evidența nodurilor explorate. Când un nod nu a fost explorat, acesta se adaugă la lista menționată folosind *explored.append(node)*.

```
def greedyBestFirstSearch(problem, heuristic=nullHeuristic):
    startNode = problem.getStartState()

    if problem.isGoalState(startNode):
        return []

    frontier = util.PriorityQueue()
    path_Greedy = []
    exploredNodes = []
    frontier.push((startNode, path_Greedy), heuristic(startNode, problem))
    while not frontier.isEmpty():
        node, path = frontier.pop()

        if node not in exploredNodes:
```

```
        if problem.isGoalState(node):
            return path

        exploredNodes.append(node)

        successors = problem.getSuccessors(node)

        for child, action, cost in successors:

            heuristic_value = heuristic(child, problem)
            frontier.push((child, path + [action]), heuristic_value)
```

Exemplu de comandă de rulare(afișare a jocului) este:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=gbfs
```

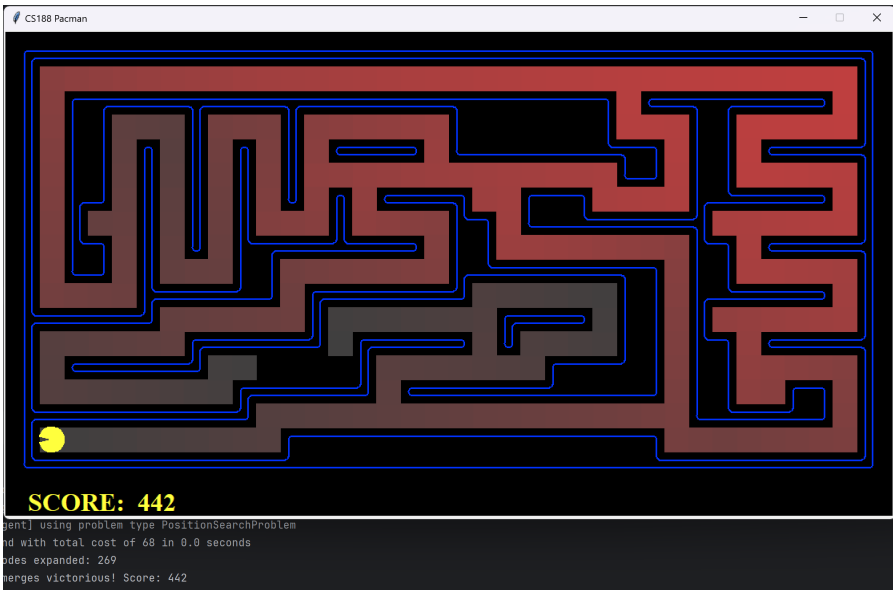


Figura 2.9.1: Greedy Best First Search Algorithm

3 Multi-Agent Search

3.1 Q1: Reflex Agent

ReflexAgent este un agent simplu care reacționează direct la mediul înconjurător, fără a planifica în avans. Sarcina constă în îmbunătățirea ReflexAgent astfel încât să navigheze mai bine labirintul, luând în considerare pozițiile mâncării și ale fantomelor. Acest lucru se realizează prin proiectarea unei funcții de evaluare care determină cât de bună este o pereche stare-acțiune.

Comenziile de afișare sunt:

```
python pacman.py -p ReflexAgent -l testClassic
python pacman.py --frameTime 0 -p ReflexAgent -k 1
```

```
python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

Pentru cea de-a doua comandă menționată, rezolvarea jocului PacMan cu Multi-Agent are următoare vizualizare pentru implementarea Reflex Agent:

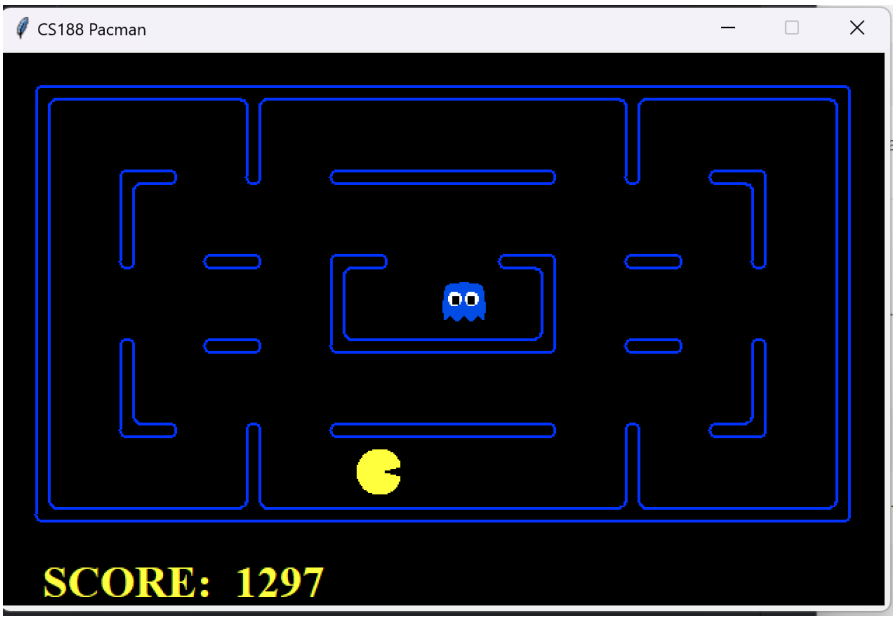


Figura 3.1.1: Reflex Agent

Codul are următoarea implementare:

```
def evaluationFunction(self, currentGameState: GameState, action):

    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in
                      newGhostStates]

    "*** YOUR CODE HERE ***"

    score = successorGameState.getScore()

    for ghostState, scaredTime in zip(newGhostStates, newScaredTimes):
        ghostPos = ghostState.getPosition()
        distToGhost = manhattanDistance(newPos, ghostPos)
        if scaredTime > 0:
            score = score + 200
            score = score / (distToGhost + 1)
        elif distToGhost < 2:
            score = score - 1000

    foodList = newFood.asList()
```

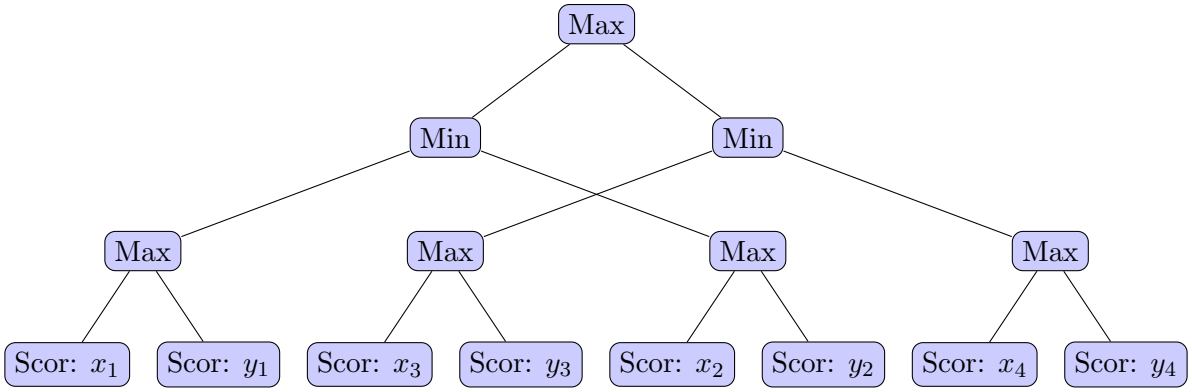
```
if foodList:
    dist = []
    for food in foodList:
        dist.append(manhattanDistance(newPos, food))
    closestFoodDist = min(dist)
    score += 10 / (closestFoodDist + 1)

if action == Directions.STOP:
    score = score - 50
 foodEaten = currentGameState.getFood().count() - newFood.count()

if foodEaten > 0:
     score = score + 100

return score
```

3.2 Q2: Minimax

MinimaxAgent implementează un algoritm de căutare adversarial, generalizat pentru mai multe niveluri de minimizare (câte un nivel pentru fiecare fantomă). Sarcina este să se creeze un agent care caută în arborele de stări la o adâncime arbitrară și să calculeze valorile minimax ale stărilor finale. Algoritmul Minimax este utilizat pentru a calcula mișcarea optimă a unui agent (Pacman) având în vedere evaluarea stării jocului pe mai multe niveluri și funcționează în următorul mod:

1. Construirea arborelui de decizie:
   - Fiecare nod reprezintă o stare a jocului.
   - Nodurile de nivel par reprezintă mutările jucătorului maxim.
   - Nodurile de nivel impar reprezintă mutările jucătorului minim.
2. La frunzele arborelui se atribuie un scor folosind o funcție de evaluare care reflectă cât de favorabilă este starea jocului pentru jucătorul maxim.
3. Nodurile de nivel impar (min) vor alege valoarea minimă dintre copiii lor, iar nodurile de nivel par (max) vor alege valoarea maximă dintre copiii lor.
4. La rădăcina arborelui, jucătorul maxim va alege mișcarea cu valoarea maximă.

Arborele de decizie pentru un joc de două persoane poate fi reprezentat astfel (exemplu):



Complexitatea algoritmului Minimax este exponențială și depinde de adâncimea arborelui și de numărul de ramuri de la fiecare nod. Dacă b este numărul de ramuri ale fiecărui nod și d este adâncimea arborelui, complexitatea algoritmului Minimax este: $O(b^d)$

Aceasta înseamnă că timpul de execuție crește rapid pe măsură ce crește numărul de posibile acțiuni și adâncimea arborelui de decizie.

Comanda de afișare este:

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

Figura 3.2.1: Minimax

3.3 Q3: Alpha-Beta Pruning

Alpha-Beta pruning este o variantă optimizată a algoritmului Minimax, care reduce numărul de stări explorate prin tăierea ramurilor irelevante din arborele de căutare. Este o tehnică de optimizare care reduce timpul de calcul, "tăind" ramuri inutile din arborele de decizie care nu influențează rezultatul final.

Aceasta este funcția principală care este apelată pentru a obține acțiunea optimă a agentului Pacman. Metoda utilizează algoritmul Minimax cu Alpha-Beta Pruning pentru a evalua toate posibilele acțiuni disponibile pentru Pacman.

- **Acțiuni legale:** `actions = gameState.getLegalActions(0)` obține toate acțiunile legale disponibile pentru Pacman (agentul cu indexul 0).
- **Inițializarea valorilor Alpha și Beta:**
  - `a = float('-inf')` - Alpha începe de la minus infinit, indicând că agentul maxim (Pacman) va accepta orice valoare mai mare decât acest prag.
  - `b = float('inf')` - Beta începe de la infinit, semnificând că agentul minim (fantomele) va accepta orice valoare mai mică decât acest prag.
- **Iterarea prin acțiuni:** Pentru fiecare acțiune posibilă, se generează un succesor al stării jocului, iar starea respectivă este evaluată folosind funcțiile `minValue()` sau `maxValue()` pentru a calcula valorile minime și maxime în arborele de decizie.

- **Actualizarea valorilor:** Dacă un rezultat este mai mare decât valoarea maximă curentă, se actualizează valoarea maximă și se selectează acțiunea respectivă.
- **Returnarea celei mai bune acțiuni:** La final, acțiunea care a generat cea mai bună evaluare este aleasă ca acțiune optimă.

Cum funcționează Alpha-Beta Pruning?

- **Alpha (a)** reprezintă cel mai bun rezultat pe care agentul maxim (Pacman) îl poate garanta până la acel moment.
- **Beta (b)** reprezintă cel mai bun rezultat pe care agentul minim (fantomele) îl poate garanta până la acel moment.
- **Alpha-Beta Pruning** ajută la "tăierea" ramurilor din arborele de decizie care nu pot influența rezultatul final al jocului, economisind astfel timp de calcul. De exemplu:
  - Dacă valoarea unui nod depășește Beta, ramura respectivă nu va fi explorată mai departe.
  - Dacă valoarea unui nod este mai mică decât Alpha, ramura respectivă este eliminată.

Comanda de afișare este:

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

3.4 Q4: Expectimax

Expectimax extinde algoritmul Minimax pentru a modela comportamentul probabilistic al adversarilor (fantome care aleg aleatoriu între acțiuni). În loc de minimizare, agentul calculează valoarea așteptată a unei stări.

Cum funcționează Expectimax?

Maxim (decizia agentului nostru): La fel ca în Minimax, la fiecare pas, agentul nostru (de exemplu, Pacman) va alege acțiunea care maximizează scorul său așteptat. Acest lucru este similar cu pasul de Maxim din Minimax, dar diferența este că acum ia în calcul probabilitățile de succes ale acțiunilor adversarilor (de exemplu, fantomele).

Minim (decizia unui adversar determinist): În Minimax, acești adversari sunt modelați printr-o alegere care minimizează scorul jucătorului. În Expectimax, acest pas este înlocuit cu o alegere de medie probabilistică (expectanță). De exemplu, dacă fantomele se mișcă aleator, algoritmul Expectimax va calcula media scorului pe baza tuturor posibilelor mișcări ale fantomelor, ponderată cu probabilitățile lor.

Calculul probabilităților: Dacă în Minimax adversarii fac mișcări care reduc la minimum scorul (maxim pentru ei, minim pentru noi), în Expectimax adversarii sunt modelați prin probabilități. De exemplu, pentru fiecare mișcare posibilă a fantomelor, Expectimax va calcula scorul mediu pentru acea mișcare, ținând cont de distribuțiile de probabilitate.

Codul are următoarea implementare:

```
def getAction(self, gameState: GameState):

     "*** YOUR CODE HERE ***"
    actions = gameState.getLegalActions(0)
    max_result = float('-inf')
    for a in actions:
        successor = gameState.generateSuccessor(0, a)
        result_iterm = self.chanceExpect(successor, 0, 1)
        if result_iterm > max_result:
            max_result = result_iterm
            max_action_pacman = a
        return max_action_pacman

    def getMaxAction(self, gameState):
        actions = gameState.getLegalActions(0)
        if not actions:
            return 'Stop'

    def maxExpect(self, gameState, depth):
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)

        actions = gameState.getLegalActions(0)
        successors = list(map(lambda a:
            gameState.generateSuccessor(0, a), actions))
        return max(self.chanceExpect(s, depth, 1) for s in successors)

    def chanceExpect(self, gameState, depth, agent_current):
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
             return self.evaluationFunction(gameState)

        actions = gameState.getLegalActions(agent_current)
        successors = list(map(lambda a:
            gameState.generateSuccessor(agent_current, a), actions))

        if agent_current < gameState.getNumAgents() - 1:
            return sum(self.chanceExpect(s, depth, agent_current + 1)
                for s in successors) / len(successors)
        else:
            return sum(self.maxExpect(s, depth + 1)
                for s in successors) / len(successors)
```

Comanda de afișare este:

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

Pentru a doua comandă se afișează următoarele rezultate în testarea din terminal:

```
PS D:\Facultate\Anul 3 - Sem 1\Inteligenta Artificiala\PROIECT2\multiagent\multiagent> python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Average Score: 118.4
Scores:        -502.0, -502.0, -502.0, -502.0, 532.0, 532.0, 532.0, 532.0, 532.0, 532.0
Win Rate:      6/10 (0.60)
Record:        Loss, Loss, Loss, Loss, Win, Win, Win, Win, Win, Win
```

Figura 3.4.1: Expectimax

3.5 Q5: Evaluation Function

Sarcina constă în scrierea unei funcții mai bune de evaluare pentru stări, betterEvaluationFunction. Aceasta trebuie să fie mai sofisticată decât funcțiile folosite anterior.

Se realizează evaluarea acestei cerințe, prin comanda:

```
python autograder.py -q q5
```

Scopul Pacman: să mănânce cât mai repede mâncarea și capsulele, să evite fantomele "inamic" și să profite de fantomele speriate.

Penalizările din scor reflectă costurile asociate cu:

- mâncarea rămasă,
- distanța până la mâncare,
- capsule,
- fantomele apropiate și altele.

Scorul final oferă o evaluare numerică a stării jocului, unde un scor mai mare indică o stare mai favorabilă.

Codul are următoarea implementare:

```
"*** YOUR CODE HERE ***"
    position= currentGameState.getPacmanPosition()

    food_positions = currentGameState.getFood().asList()
    ghost_positions = currentGameState.getGhostPositions()

    capsules_positions = currentGameState.getCapsules()

    state_of_ghost = currentGameState.getGhostStates()
    scared_g = [ghost.scaredTimer for ghost in state_of_ghost]
```

```
food_remaining = len(food_positions)
capsules_remaining = len(capsules_positions)
scared= list()
enemy = list()

enemy_position = list()
scared_position = list()

eval = currentGameState.getScore()

distance_from_food = [manhattanDistance(position, food_position)
                      for food_position in food_positions]
if len(distance_from_food) is not 0:
    closest_food = min(distance_from_food)
    eval -= 1.0 * closest_food

enemy = [g for g in state_of_ghost if g.scaredTimer]
scared = [g for g in state_of_ghost if not g.scaredTimer]

for s in scared:
    scared_position.append(s.getPosition())

for e in enemy:
    enemy_position.append(e.getPosition())

if len(scared_position):
    distance_from_scared_ghost = list(map(lambda
         scared_ghost_position: manhattanDistance(position,
         scared_ghost_position), scared_position))

    scared_close = min(distance_from_scared_ghost)
    eval = eval - 3.0 * scared_close


if len(enemy_position):
    distance_enemy = list(map(lambda e:
         manhattanDistance(position, e), enemy_position))

    enemy_close = min(distance_enemy)
    eval -= 2.0 * (1 / enemy_close)

penalty = 20.0 * capsules_remaining + 4.0 * food_remaining
eval -= penalty
return eval
```

Penalizări suplimentare

Penalizează scorul pentru numărul de capsule rămase: `eval -= 20.0 * capsules_remaining`.

Penalizează scorul pentru numărul de bucăți de mâncare rămase: `eval -= 4.0 * food_remaining`.

În cel mai rău caz, Pacman trebuie să proceseze toate elementele:

- F bucăți de mâncare,
- C capsule,
- G fantome.

Operațiile dominante (calculul distanțelor și găsirea minimelor) oferă o complexitate aproximativă:

$$O(F + G_e + G_s),$$

unde F este numărul de bucăți de mâncare, G_e numărul de fantome inamice și G_s numărul de fantome speriate.

4 Concluzii

Realizarea jocului Pacman folosind algoritmii precizați în capitolele anterioare, oferă o viziune practică asupra modului în care algoritmii AI pot fi aplicați pentru a rezolva probleme complexe de luare a deciziilor într-un mediu dinamic. În acest context, astfel de algoritmi sunt esențiali pentru modelarea deciziilor într-un cadru competitiv. Mai mult, crearea unui evaluator performant ajută la îmbunătățirea comportamentului agentului, permițându-i să ia decizii optime sau aproape optime.

În domeniul Inteligenței Artificiale, dezvoltarea și aplicarea acestor algoritmi în jocuri ca Pacman este esențială pentru înțelegerea și implementarea tehnicilor de *căutare adversarială*, *optimizare* și *decizii probabilistice*. Algoritmii utilizați în acest joc ilustrează modul în care AI poate imita comportamente complexe ale agenților.

Bibliografie

[1] https://stackoverflow.com/questions/29141501/how-to-implement-bfs-algorithm-for-pacman

[2] https://en.wikipedia.org/wiki/A*_search_algorithm

[3] https://medium.com/nerd-for-tech/graph-traversal-in-python-a-algorithm-27c30d67e0d0

[4] https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs

[5] https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs

[6] https://rshcaroline.github.io/research/resources/pacman-report.pdf