UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: Colceriu Mihai Bogdan
Grupa: 30233

Facultatea de Automatica
si Calculatoare

04 Noiembrie 2024

Cuprins

1 Uninformed search **2**
1.1 Question 1 - Depth-first search 2
1.2 Question 2 - Breadth-first search 3
1.3 Question 3 - Uniform Cost Search 4

2 Informed search **5**
2.1 Question 4 - A* Search Algorithm 5
2.2 Question 5 - Finding All the Corners 7
2.3 Question 6 - Corners Problem: Heuristic 9
2.4 Question 7 - Food Search Heuristic 10
2.5 Question 8 - Suboptimal Search: Closest Dot Search Agent 11

3 Adversarial search **12**
3.1 Question 1 - ReflexAgent 12
3.2 Question 2 - MinimaxAgent 14
3.3 Question 3 - AlphaBetaAgent 15
3.4 Implementarea suplimentară pentru punctul în plus - algoritmului Greedy Best-First Search (GBFS) + euristică bazată pe MST 17
3.5 Implementarea euristicii bazate pe MST 18

1 Uninformed search

1.1 Question 1 - Depth-first search

În fișierul search e rezolvarea conform cerințelor proiectului, iar în search cu punct bonus e implementarea GBFS + euristica MST ca și componente adăugate de mine.

Definirea cerinței: Implementarea unui algoritm de căutare în adâncime (DFS) pentru explorarea unui spațiu de stări în mod neinformat. Algoritmul trebuie să respecte cerința de a nu vizita stări deja explorate.

Prezentare algoritm: Algoritmul DFS utilizează o stivă pentru a gestiona stările de explorat. Pe scurt, pașii sunt:

1. Inițializarea stivei cu starea inițială.
2. Repetarea procesului:
   - Extragerea unei stări din stivă.
   - Marcarea acesteia ca fiind vizitată.
   - Adăugarea succesorilor neexplorați în stivă.
3. Oprirea procesului dacă se atinge starea-țintă.

Cod:

```
def depthFirstSearch(problem):
    stack = util.Stack()
    visited = set()
    stack.push((problem.getStartState(), []))

    while not stack.isEmpty():
        state, actions = stack.pop()
        if state in visited:
            continue
        visited.add(state)

        if problem.isGoalState(state):
            return actions

        for successor, action, _ in problem.getSuccessors(state):
            if successor not in visited:
                stack.push((successor, actions + [action]))
```

Explicații:

- `util.Stack()` este folosită pentru a implementa structura LIFO (last-in, first-out).
- Setul `visited` păstrează stările deja explorate pentru a evita expansiunile repetate.
- Succesorii sunt extrași utilizând metoda `getSuccessors()`, iar acțiunile sunt acumulate pentru a construi drumul către starea finală.

Performanță:

- DFS explorează profund spațiul de stări, ceea ce poate duce la cicluri infinite în cazul în care nu se păstrează un set de stări vizitate.
- Complexitatea temporală este $O(b^m)$, unde b este factorul de ramificare, iar m este adâncimea maximă a arborelui de căutare.
- Complexitatea spațială este $O(b \times m)$, datorită stivei folosite.

Rezultate:

- Testele pentru DFS au fost rulate cu comanda: `python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs`.
- Algoritmul a găsit o soluție rapid, însă nu întotdeauna cea mai scurtă cale, datorită naturii sale de căutare neinformată.

1.2 Question 2 - Breadth-first search

Definirea cerinței: Implementarea unui algoritm de căutare în lățime (BFS) pentru explorarea unui spațiu de stări în mod neinformat. Algoritmul trebuie să respecte cerința de a nu vizita stări deja explorate și să găsească întotdeauna soluția cu cel mai mic număr de pași.

Prezentare algoritm: Căutarea în lățime utilizează o coadă pentru gestionarea stărilor, explorând mai întâi toate nodurile de pe un nivel înainte de a trece la următorul. Pașii principali ai algoritmului sunt:

1. Inițializarea cozii cu starea inițială.
2. Repetarea procesului:
    - Extragerea unei stări din coadă.
    - Verificarea dacă starea este o stare-țintă.
    - Marcarea acesteia ca fiind vizitată.
    - Adăugarea succesorilor neexplorați în coadă.
3. Oprirea procesului la găsirea stării-țintă sau când toate stările au fost explorate.

Codul:

```
def breadthFirstSearch(problem):
    queue = util.Queue()
    visited = set()
    queue.push((problem.getStartState(), []))

    while not queue.isEmpty():
        state, actions = queue.pop()

        if state in visited:
            continue
        visited.add(state)

        if problem.isGoalState(state):
            return actions

        for successor, action, _ in problem.getSuccessors(state):
            if successor not in visited:
                queue.push((successor, actions + [action]))
```

Explicații:

- `util.Queue()` este utilizată pentru a implementa structura FIFO, caracteristică BFS.
- Similar cu DFS, setul `visited` previne explorarea repetată a acelorași stări.
- Succesorii sunt extrași și adăugați la coadă împreună cu drumul acumulat până la acea stare.
- BFS prioritizează stările mai apropiate de rădăcina arborelui de căutare, garantând găsirea soluției cu cel mai mic număr de pași.

Performanță:

- **Complexitatea temporală:** $O(b^d)$, unde b este factorul de ramificare și d este adâncimea soluției.
- **Complexitatea spațială:** $O(b^d)$, deoarece toate nodurile de pe un nivel trebuie stocate în memorie.
- Performanța este limitată de spațiu, deoarece BFS necesită stocarea simultană a tuturor nodurilor de pe un nivel.

Rezultate:

- Testele pentru BFS au fost rulate cu comanda:
  - `python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs`
  - `python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z 0.5`
- BFS a găsit soluții corecte și optime în termeni de număr de pași pentru ambele cazuri.
- Pentru labirintul `bigMaze`, algoritmul a explorat 620 noduri, iar soluția găsită este mai rapidă decât DFS.

Comparație cu DFS:

- Spre deosebire de DFS, BFS găsește întotdeauna soluția optimă (cu cel mai mic număr de pași).
- Totuși, BFS este mai costisitor din punct de vedere al memoriei, deoarece păstrează simultan mai multe noduri în memorie.

Concluzie:

- BFS este o alegere mai potrivită decât DFS atunci când soluția optimă este necesară.
- Pentru spații de căutare mari, este recomandat să fie utilizat împreună cu euristici, cum ar fi în algoritmul A*.

1.3 Question 3 - Uniform Cost Search

Problema constă în implementarea algoritmului **Uniform Cost Search (UCS)**, un algoritm de căutare bazat pe cost care extinde nodurile în funcție de costul cumulat al traseului.

Descriere problemă:

- UCS este o variantă a algoritmului de căutare în lățime (*Breadth-First Search*), dar prioritatea este determinată de costul traseului în loc de nivelul de adâncime.
- Este ideal pentru probleme unde costurile între noduri pot varia.

Codul: Algoritmul a fost implementat în funcția `uniformCostSearch` din `search.py`. Codul Python utilizat este următorul:

```
1  def uniformCostSearch(problem):
2      from util import PriorityQueue
3
4      frontier = PriorityQueue()
5      visited = set()
6
7      start_state = (problem.getStartState(), [], 0)
8      frontier.push(start_state, 0)
9
10     while not frontier.isEmpty():
11         current_state, actions, current_cost = frontier.pop()
12
13         if current_state in visited:
```

```
            continue

        visited.add(current_state)

        if problem.isGoalState(current_state):
            return actions

        for successor, action, step_cost in problem.getSuccessors(current_state):
            if successor not in visited:
                total_cost = current_cost + step_cost
                frontier.push((successor, actions + [action], total_cost), total_cost)

    return []
```

Explicații:

- O coadă de priorități (`PriorityQueue`) este utilizată pentru a gestiona stările pe baza costului cumulativ.
- Nodurile vizitate sunt păstrate într-un set (`visited`) pentru a preveni re-expansiunea inutilă.
- La fiecare expansiune:
  - Se extrage nodul cu cel mai mic cost din coadă.
  - Dacă nodul este starea finală, se returnează calea (`actions`).
  - Dacă nu, succesorii nodului sunt adăugați în frontieră, împreună cu costurile actualizate.

Testare și rezultate: Pentru a valida implementarea, au fost utilizate următoarele comenzi:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Rezultate:

- Algoritmul a găsit soluții optime pentru toate configurațiile testate.
- Comportamentul agenților diferă în funcție de funcția de cost utilizată:
  - `StayEastSearchAgent` încurajează deplasarea spre est, prin costuri exponențial mai mari spre vest.
  - `StayWestSearchAgent` încurajează deplasarea spre vest, utilizând o strategie similară.

Concluzii: Implementarea algoritmului `uniformCostSearch` a demonstrat eficiență și corectitudine, garantând soluții optime pentru probleme de căutare cu costuri variabile. De asemenea, comportamentul agenților `StayEast` și `StayWest` evidențiază flexibilitatea UCS în adaptarea la funcții de cost personalizate.

2 Informed search

2.1 Question 4 - A* Search Algorithm

Algoritmul **A*** este o extensie a căutării uniforme (*Uniform Cost Search*), care utilizează o funcție euristică pentru a estima costul rămas până la atingerea scopului. Acesta combină costul

de la începutul căutării până la un nod (`g(n)`) cu estimarea costului rămas (`h(n)`), oferind o prioritate generală definită de `f(n) = g(n) + h(n)`.

Descriere problemă:

- A* este un algoritm euristic optim și complet dacă funcția euristică utilizată este admisibilă (nu supraestimează costul real).
- Implementarea sa este utilizată pentru a găsi soluții eficiente în labirinturi și alte probleme de căutare.

Cod: Algoritmul a fost implementat în funcția `aStarSearch` din `search.py`. Codul Python utilizat este următorul:

```
def aStarSearch(problem, heuristic=nullHeuristic):
    from util import PriorityQueue

    frontier = PriorityQueue()
    visited = {}

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

    while not frontier.isEmpty():
        current_state, actions, current_cost = frontier.pop()

        if current_state in visited and visited[current_state] <= current_cost:
            continue

        visited[current_state] = current_cost

        if problem.isGoalState(current_state):
            return actions

        for successor, action, step_cost in problem.getSuccessors(current_state):
            total_cost = current_cost + step_cost
            f_cost = total_cost + heuristic(successor, problem)

            if successor not in visited or visited[successor] > total_cost:
                frontier.push((successor, actions + [action], total_cost), f_cost)

    return []
```

Explicații:

- O coadă de priorități (`PriorityQueue`) gestionează stările pe baza funcției `f(n) = g(n) + h(n)`.
- Nodurile vizitate sunt păstrate într-un dicționar (`visited`) împreună cu costul minim găsit până la acel nod.
- La fiecare expansiune:
  - Se extrage nodul cu cel mai mic cost `f(n)` din frontieră.
  - Dacă nodul este starea finală, se returnează calea (`actions`).
  - Succesorii nodului sunt calculați și adăugați în frontieră cu costurile actualizate.

Rezultate: Pentru a valida implementarea, au fost utilizate următoarele comenzi:

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

Rezultatele obținute:

- Algoritmul a găsit soluții optime pentru toate testele.
- A* a depășit algoritmul `Uniform Cost Search` în timp de execuție și număr de noduri expandate, datorită utilizării euristicii.

Observații:

- Pentru funcția `manhattanHeuristic`, numărul de noduri expandate a fost semnificativ redus comparativ cu UCS.
- În configurația `bigMaze`, A* a extins aproximativ 549 noduri, în timp ce UCS a extins peste 620 noduri.
- Performanța A* depinde de calitatea euristicii utilizate:
  - O euristică admisibilă (precum distanța Manhattan) garantează soluții optime.
  - O euristică neadmisibilă poate compromite corectitudinea rezultatului.

Concluzii: Implementarea algoritmului `aStarSearch` a demonstrat eficiență în găsirea soluțiilor optime utilizând euristici, reducând numărul de noduri expandate. Este un algoritm versatil și puternic pentru rezolvarea problemelor complexe de căutare.

2.2 Question 5 - Finding All the Corners

Problema constă în determinarea celei mai scurte căi printr-un labirint, astfel încât Pacman să treacă prin toate cele patru colțuri ale labirintului. Această problemă introduce un nou tip de reprezentare a stărilor și utilizează algoritmi de căutare pentru a rezolva problema optim.

Descriere problemă:

- Trebuie să implementăm o clasă `CornersProblem` care definește starea inițială, succesorii posibili, condiția de atingere a scopului și costurile de tranziție.
- O stare trebuie să conțină poziția actuală a lui Pacman și informația despre care colțuri au fost deja vizitate.

Implementare: Codul pentru clasa `CornersProblem` și metoda `getSuccessors` este următorul:

```
1  class CornersProblem(search.SearchProblem):
2
3      def __init__(self, startingGameState):
4
5          self.startingPosition = startingGameState.getPacmanPosition()
6          self.corners = ((1, 1), (1, startingGameState.data.layout.height - 2),
7                          (startingGameState.data.layout.width - 2, 1),
8                          (startingGameState.data.layout.width - 2, startingGameState.data.lay
9          self.walls = startingGameState.getWalls()
10
11
12         for corner in self.corners:
13             if not startingGameState.hasFood(*corner):
14                 raise Exception('Corner %s is not reachable' % str(corner))
15
16         self._expanded = 0
```

```
17
18      def getStartState(self):
19
20          return (self.startingPosition, (False, False, False, False))
21
22      def isGoalState(self, state):
23
24          _, visitedCorners = state
25          return all(visitedCorners)
26
27      def getSuccessors(self, state):
28
29          successors = []
30          currentPosition, visitedCorners = state
31
32          for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
33              x, y = currentPosition
34              dx, dy = Actions.directionToVector(action)
35              nextPosition = (int(x + dx), int(y + dy))
36
37              if not self.walls[nextPosition[0]][nextPosition[1]]:
38                  newVisitedCorners = list(visitedCorners)
39                  if nextPosition in self.corners:
40                      index = self.corners.index(nextPosition)
41                      newVisitedCorners[index] = True
42
43                  successors.append(((nextPosition, tuple(newVisitedCorners)), action, 1))
44
45          self._expanded += 1
46          return successors
47
48      def getCostOfActions(self, actions):
49
50          if actions == None: return 999999
51          x, y = self.startingPosition
52          for action in actions:
53              dx, dy = Actions.directionToVector(action)
54              x, y = int(x + dx), int(y + dy)
55              if self.walls[x][y]:
56                  return 999999
57          return len(actions)
```

Explicații:

- `__init__`: Stabilește poziția de start a lui Pacman, pozițiile colțurilor și pereții labirintului.
- `getStartState`: Returnează starea inițială, care include poziția de start și un tuplu care indică ce colțuri au fost vizitate.
- `isGoalState`: Verifică dacă toate colțurile au fost vizitate.
- `getSuccessors`: Generează succesorii stării curente, împreună cu acțiunile necesare și costurile.

- `getCostOfActions`: Calculează costul unui set de acțiuni, verificând dacă acestea trec prin pereți.

Rezultate: Pentru testarea soluției, au fost utilizate următoarele comenzi:

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

Rezultatele obținute:

- Pe `tinyCorners`, algoritmul a extins aproximativ 252 de noduri.
- Pe `mediumCorners`, algoritmul a extins aproximativ 2000 de noduri utilizând BFS.

Observații privind performanța:

- Reprezentarea stărilor este esențială pentru eficiența algoritmului. Utilizarea unui tuplu pentru colțuri vizitate reduce redundanța și accelerează căutarea.
- Heuristica pentru această problemă este discutată în detaliu la Q6.

Concluzii: Implementarea clasei `CornersProblem` demonstrează cum pot fi abordate probleme complexe de căutare prin definirea corectă a stărilor, tranzițiilor și scopurilor. Aceasta servește ca bază pentru optimizări ulterioare prin utilizarea euristicilor, discutate în întrebările următoare.

2.3 Question 6 - Corners Problem: Heuristic

Cerință: Se cere implementarea unei euristici care să estimeze costul minim rămas pentru a atinge toate cele patru colțuri ale labirintului. Euristica trebuie să fie:

- **Non-trivială:**
- **Admisibilă:**
- **Consistentă:**

Strategia:

- Calculăm distanța Manhattan între poziția curentă și cel mai apropiat colț care nu a fost vizitat.
- Adăugăm la această distanță estimarea minimă a drumului pentru a atinge toate colțurile rămase.
- Acest lucru garantează că estimarea este admisibilă și utilă.

Implementare euristică: Codul implementat pentru funcția `cornersHeuristic` este următorul:

```
1  def cornersHeuristic(state, problem):
2
3      position, visitedCorners = state
4      unvisitedCorners = [corner for i, corner in enumerate(problem.corners) if not visitedCor
5
6      if not unvisitedCorners:
7          return 0
8
9      distances = [util.manhattanDistance(position, corner) for corner in unvisitedCorners]
10
11     min_distance = min(distances)
12
13     estimated_cost = min_distance
14
15     return estimated_cost
```

Explicații cod:

- `state`: Starea curentă include poziția lui Pacman și un tuplu de valori booleene pentru colțurile vizitate.
- `unvisitedCorners`: Extragem colțurile care nu au fost vizitate din starea curentă.
- `distances`: Calculăm distanțele Manhattan de la poziția curentă la fiecare colț nevizitat.
- `min_distance`: Selectăm distanța minimă, corespunzătoare celui mai apropiat colț nevizitat.
- `estimated_cost`: Returnăm costul estimat, adică distanța minimă. Acesta poate fi extins pentru a considera și alte colțuri într-o ordine optimă.

Testare și rezultate: Pentru testarea funcției `cornersHeuristic`, au fost utilizate următoarele comenzi:

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

Rezultatele obținute:

- Pe `mediumCorners`, numărul de noduri extinse este mai mic de 2000, ceea ce indică o euristică admisibilă și utilă.
- Pe `bigCorners`, euristica permite soluționarea problemei în timp rezonabil, reducând semnificativ numărul de noduri extinse comparativ cu UCS.

Observații privind performanța:

- Euristica este **admisibilă**, deoarece distanța Manhattan nu supraestimează costul real al deplasării.
- Este **non-trivială**, deoarece reduce numărul de noduri extinse comparativ cu euristica trivială `h(n) = 0`.

2.4 Question 7 - Food Search Heuristic

Am rezolvat problema îndeplinind și condiția de a expanda mai puțin de 7000 de noduri și am primit punctul în plus adică 5/4. În această secțiune, scopul este să dezvoltăm o euristică pentru `FoodSearchProblem`, care să permită utilizarea algoritmului A* pentru a găsi traseul optim care colectează toate punctele de mâncare din labirint.

Cerință:

- Implementarea unei funcții euristice în `foodHeuristic`.
- Euristica trebuie să fie admisibilă (nu supraestimează costul real) și să reducă numărul de noduri expandate.

Strategie: Implementarea funcției `foodHeuristic` utilizează distanța de labirint (`mazeDistance`) pentru a calcula costul maxim către cel mai îndepărtat punct de mâncare. Această abordare oferă o estimare mai realistă decât distanța Manhattan, îmbunătățind performanța algoritmului.

Cod:

```
1  def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
2
3      position, foodGrid = state
4      foodList = foodGrid.asList()
5
6      if not foodList:
7          return 0
8
```

```
    max_distance = 0

    for food in foodList:
        distance = mazeDistance(position, food, problem.startingGameState)
        max_distance = max(max_distance, distance)

    return max_distance
```

Explicații:

* `state`: starea curentă, reprezentată prin poziția lui Pacman și grila cu mâncare.
* `problem`: instanța problemei `FoodSearchProblem`.
* Funția va da costul estimat de la starea curentă până la obiectiv (toate punctele de mâncare colectate).

– **Algoritm:**

1. Dacă nu există puncte de mâncare (`foodList` este gol), returnăm 0.
2. Iterăm prin toate punctele de mâncare din `foodList` și calculăm distanța de labirint (`mazeDistance`) dintre poziția curentă și fiecare punct.
3. Determinăm cea mai mare dintre aceste distanțe (`max_distance`) și o folosim ca euristică.

Rezultate: Codul a fost testat folosind:

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

– Agentul a găsit soluția optimă cu un cost de traseu minim.
– Numarul de noduri expandate este semnificativ redus comparativ cu euristica trivială.
– Timpul de execuție este unul bun, chiar și pentru hărți complexe.

Concluzii:

– Utilizarea distanței de labirint ca bază pentru euristică îmbunătățește precizia estimării, reducând numărul de noduri expandate.
– Această abordare oferă un echilibru între complexitatea calculului euristic și reducerea căutării în spațiul stărilor.

2.5 Question 8 - Suboptimal Search: Closest Dot Search Agent

În această cerință, trebuie să implementăm o strategie suboptimă pentru Pacman, care prioritizează găsirea celui mai apropiat punct de mâncare, fără a ține cont de traseul optim global. Agentul `ClosestDotSearchAgent` este deja definit în `searchAgents.py`, însă funcția `findPathToClosestDot`, responsabilă cu determinarea traseului către cel mai apropiat punct, trebuie implementată.

Strategie:

– Definim problema `AnyFoodSearchProblem`, care extinde `PositionSearchProblem`.
– Stabilim criteriul de oprire în funcția `isGoalState`, pentru a determina dacă poziția curentă conține mâncare.
– Utilizăm o funcție de căutare, cum ar fi BFS, pentru a găsi cel mai scurt traseu către cel mai apropiat punct.

Cod:

```
def findPathToClosestDot(gameState):

    problem = AnyFoodSearchProblem(gameState)

    return search.bfs(problem)
```

Clasa `AnyFoodSearchProblem`:

```
class AnyFoodSearchProblem(PositionSearchProblem):

    def __init__(self, gameState):
        self.food = gameState.getFood()
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        super().__init__(gameState, costFn=self.costFn, goal=None)

    def isGoalState(self, state):

        x, y = state
        return self.food[x][y]

```

Explicații:

- `findPathToClosestDot`:
  - Creează o instanță a problemei `AnyFoodSearchProblem`, care conține toate detaliile despre starea curentă a jocului.
  - Utilizează funcția `bfs` (Breadth-First Search) din `search.py` pentru a găsi drumul către cel mai apropiat punct.
- `AnyFoodSearchProblem`:
  - Extinde `PositionSearchProblem`, dar redefinește funcția `isGoalState` pentru a verifica dacă poziția curentă conține mâncare.
  - Utilizează `foodGrid` pentru a determina dacă starea curentă este o stare de obiectiv.

Testare și rezultate: Agentul a fost testat utilizând comanda:

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

Rezultate observate:

- Agentul găsește traseul către toate punctele de mâncare în mai puțin de o secundă.
- Costul total al traseului este suboptimal (aproximativ 350 pentru `bigSearch`), deoarece agentul urmărește doar punctele cele mai apropiate, fără să ia în considerare costul global.

Concluzii:

- Soluția este rapidă, dar suboptimală, deoarece agentul poate alege trasee care implică revenirea pe aceeași porțiune a hărții.

3 Adversarial search

3.1 Question 1 - ReflexAgent

Descriere cerință: În această întrebare, a fost necesar să îmbunătățim agentul ReflexAgent din fișierul `multiAgents.py`. Agentul trebuia să ia decizii eficiente bazate pe pozițiile mâncării, fantomelor și altor factori relevanți.

Algoritmul ReflexAgent îmbunătățit: Agentul a fost modificat astfel încât să evalueze în detaliu fiecare acțiune posibilă, utilizând o funcție de evaluare personalizată. Această funcție ia în considerare următorii factori:

- Distanța Manhattan până la cea mai apropiată bucată de mâncare (pentru a încuraja agentul să mănânce mâncare rapid).
- Distanța până la fantomele aflate în apropiere (pentru a evita moartea).
- Stările de frică (`newScaredTimes`) ale fantomelor, pentru a decide dacă să le evite sau să le vâneze.
- Numărul total de bucăți de mâncare rămase pe hartă, pentru a penaliza stările de indecizie.

Funcția de evaluare combină acești factori pentru a atribui un scor fiecărei stări de joc, iar ReflexAgent alege acțiunea care maximizează acest scor.

Codul:

```
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]

    foodDistances = [manhattanDistance(newPos, foodPos) for foodPos in newFood.asList()
    minFoodDistance = min(foodDistances) if foodDistances else 1

    ghostDistances = [manhattanDistance(newPos, ghostState.getPosition()) for ghostStat

    ghostPenalty = 0
    for i, ghostDistance in enumerate(ghostDistances):
        if ghostDistance > 0:
            ghostPenalty += (1 / ghostDistance) if newScaredTimes[i] == 0 else 0

    score = successorGameState.getScore()
    score += 10 / minFoodDistance
    score -= ghostPenalty * 10

    return score
```

Rezultate obținute: Agentul îmbunătățit a demonstrat performanțe semnificativ mai bune:

- A reușit să finalizeze harta `testClassic`.
- Pe harta `mediumClassic`, agentul s-a descurcat bine cu 1 fantomă și a avut performanțe mixte cu 2 fantome.
- Pentru testare, am folosit comenzile:

```
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
```

Observații: Agentul a fost capabil să optimizeze deciziile în funcție de distanța până la mâncare și de prezența fantomelor. Totuși, performanța sa este limitată de faptul că funcția de evaluare nu anticipează mai mulți pași în avans.

Testare: Am verificat performanța agentului folosind comanda:

```
python autograder.py -q q1 --no-graphics
```

Rezultatele obținute au fost satisfăcătoare, agentul îndeplinind criteriile specificate în enunț.

3.2 Question 2 - MinimaxAgent

Cerință: În această întrebare, am implementat un agent care utilizează algoritmul *Minimax* pentru a explora arborele de decizie al jocului. Agentul Minimax trebuie să funcționeze cu un număr arbitrar de fantome și să fie capabil să evalueze stările până la o adâncime specificată. Fiecare strat de adâncime din arbore reprezintă o mișcare a lui Pacman urmată de răspunsurile fantomelor.
Algoritmul Minimax: Algoritmul Minimax caută să maximizeze scorul lui Pacman în timp ce minimizează riscurile introduse de fantome. Agentul Pacman este un maximizator, în timp ce fiecare fantomă este un minimizator. La fiecare strat al arborelui de decizie:

- Straturile de `max` reprezintă deciziile lui Pacman (agentul 0).
- Straturile de `min` reprezintă deciziile fantomelor (agenți `>= 1`).

Algoritmul evaluează frunzele arborelui folosind funcția `self.evaluationFunction`.
Implementare: Codul implementat pentru agentul Minimax este prezentat mai jos:

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState: GameState):

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

            if agentIndex == 0:  #max - pacman
                return maxValue(agentIndex, depth, gameState)
            else:  #min - fantome
                return minValue(agentIndex, depth, gameState)

        def maxValue(agentIndex, depth, gameState):
            v = float('-inf')
            for action in gameState.getLegalActions(agentIndex):
                successor = gameState.generateSuccessor(agentIndex, action)
                v = max(v, minimax(1, depth, successor))
            return v

        def minValue(agentIndex, depth, gameState):
            v = float('inf')
            nextAgent = (agentIndex + 1) % gameState.getNumAgents()
            nextDepth = depth + 1 if nextAgent == 0 else depth

            for action in gameState.getLegalActions(agentIndex):
                successor = gameState.generateSuccessor(agentIndex, action)
                v = min(v, minimax(nextAgent, nextDepth, successor))
            return v

        actions = gameState.getLegalActions(0)
        scores = [minimax(1, 0, gameState.generateSuccessor(0, action)) for action in a
```

```
        bestScore = max(scores)
        bestIndex = scores.index(bestScore)
        return actions[bestIndex]
```

Descriere detaliată:

- Funcția `minimax(agentIndex, depth, gameState)` implementează recursiv algoritmul Minimax. În funcție de agentul curent:
  - Dacă este Pacman (`agentIndex == 0`), funcția apelează `maxValue`.
  - Dacă este o fantomă (`agentIndex >= 1`), funcția apelează `minValue`.
- Funcția `maxValue` calculează valoarea maximă a unui nod, iterând prin toate acțiunile posibile ale lui Pacman și generând succesorii.
- Funcția `minValue` calculează valoarea minimă a unui nod, iterând prin toate acțiunile posibile ale fantomelor și generând succesorii. Dacă fantoma curentă este ultima, următorul agent este Pacman, iar adâncimea crește cu 1.
- Funcția `getAction` determină cea mai bună acțiune pentru Pacman, evaluând toate succesorii posibili și alegând acțiunea cu cel mai mare scor.

Testare și rezultate: Agentul a fost testat folosind următoarele comenzi:

```
python pacman.py -p MinimaxAgent -a depth=3 -l smallClassic
python autograder.py -q q2
```

Rezultatele obținute indică faptul că algoritmul Minimax funcționează corect:

- Pe harta `minimaxClassic`, valorile Minimax pentru starea inițială au fost corecte: 9, 8, 7 și -492 pentru adâncimi de 1, 2, 3 și respectiv 4.
- Agentul a fost capabil să gestioneze mai mulți agenți (Pacman și fantome) și să ia decizii optime în majoritatea cazurilor.

Observații:

- Performanța agentului depinde de adâncimea arborelui de căutare. La adâncimi mari, timpul de execuție crește considerabil.

3.3 Question 3 - AlphaBetaAgent

Descriere cerință: În această întrebare, am implementat un agent care utilizează algoritmul *Alpha-Beta Pruning* pentru a optimiza căutarea Minimax. Prin aplicarea tăierii Alpha-Beta, agentul explorează doar succesorii relevanți, reducând numărul de stări evaluate fără a afecta rezultatul final. Agentul trebuie să gestioneze mai mulți agenți (Pacman și fantome) și să lucreze corect pentru o adâncime specificată.

Algoritmul Alpha-Beta Pruning: Alpha-Beta Pruning îmbunătățește Minimax prin eliminarea succesorilor care nu pot influența rezultatul final.

- α: cea mai bună valoare maximă de-a lungul căii curente.
- β: cea mai bună valoare minimă de-a lungul căii curente.

Implementare: Codul implementat pentru agentul Alpha-Beta este prezentat mai jos:

```
class AlphaBetaAgent(MultiAgentSearchAgent):

    def getAction(self, gameState: GameState):

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

```
        if agentIndex == 0:  #max - pacman
            return maxValue(agentIndex, depth, gameState, alpha, beta)
        else:  #min - fantome
            return minValue(agentIndex, depth, gameState, alpha, beta)

    def maxValue(agentIndex, depth, gameState, alpha, beta):
        v = float('-inf')
        for action in gameState.getLegalActions(agentIndex):
            successor = gameState.generateSuccessor(agentIndex, action)
            v = max(v, alphaBeta(1, depth, successor, alpha, beta))
            if v > beta:
                return v
            alpha = max(alpha, v)
        return v

    def minValue(agentIndex, depth, gameState, alpha, beta):
        v = float('inf')
        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        nextDepth = depth + 1 if nextAgent == 0 else depth

        for action in gameState.getLegalActions(agentIndex):
            successor = gameState.generateSuccessor(agentIndex, action)
            v = min(v, alphaBeta(nextAgent, nextDepth, successor, alpha, beta))
            if v < alpha:
                return v
            beta = min(beta, v)
        return v

    actions = gameState.getLegalActions(0)
    alpha, beta = float('-inf'), float('inf')
    scores = []
    for action in actions:
        scores.append(alphaBeta(1, 0, gameState.generateSuccessor(0, action), alpha
        alpha = max(alpha, scores[-1])

    bestScore = max(scores)
    bestIndex = scores.index(bestScore)
    return actions[bestIndex]
```

- Funcția `alphaBeta(agentIndex, depth, gameState, alpha, beta)` implementează recursiv algoritmul Alpha-Beta.
  - Dacă agentul este Pacman (`agentIndex == 0`), funcția apelează `maxValue`.
  - Dacă agentul este o fantomă (`agentIndex >= 1`), funcția apelează `minValue`.
- Funcția `maxValue` calculează valoarea maximă a unui nod folosind α și β pentru a tăia succesorii irelevanți.
- Funcția `minValue` calculează valoarea minimă a unui nod folosind α și β pentru a tăia succesorii irelevanți.
- Funcția `getAction` determină cea mai bună acțiune pentru Pacman, evaluând toți succesorii și folosind Alpha-Beta pentru optimizare.

Testare și rezultate: Agentul a fost testat folosind următoarele comenzi:

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
python autograder.py -q q3
```

Rezultatele obținute indică faptul că algoritmul Alpha-Beta funcționează corect:

- Valorile Minimax calculate cu Alpha-Beta au fost identice cu cele obținute prin Minimax simplu.
- Numărul de stări explorate a fost semnificativ redus datorită tăierii Alpha-Beta.
- Performanța agentului a fost mai bună decât în cazul algoritmului Minimax, iar timpul de execuție a fost redus semnificativ.

Observații:

- Agentul Alpha-Beta are același comportament ca agentul Minimax, dar este mai eficient datorită reducerii spațiului de căutare.
- Alpha-Beta pruning este implementat fără reordonarea succesorilor, respectând cerințele autograderului.

3.4 Implementarea suplimentară pentru punctul în plus - algoritmului Greedy Best-First Search (GBFS) + euristică bazată pe MST

Descrierea algoritmului: Greedy Best-First Search este un algoritm de căutare informat care selectează, la fiecare pas, nodul cu cea mai mică valoare a funcției euristice `h(n)`. Algoritmul prioritizează nodurile care par a fi cele mai apropiate de starea țintă, fără a lua în considerare costul acumulat până la acel nod.

Implementare: Funcția `gbfs` a fost adăugată în `search.py` și este prezentată mai jos:

```python
def gbfs(problem, heuristic=nullHeuristic):
    from util import PriorityQueue

    frontier = PriorityQueue()
    start_state = problem.getStartState()
    frontier.push((start_state, []), heuristic(start_state, problem))
    visited = set()

    while not frontier.isEmpty():
        state, actions = frontier.pop()

        if state in visited:
            continue
        visited.add(state)

        if problem.isGoalState(state):
            return actions

        for successor, action, _ in problem.getSuccessors(state):
            if successor not in visited:
                new_actions = actions + [action]
                priority = heuristic(successor, problem)
                frontier.push((successor, new_actions), priority)

    return []
```

Explicații cod:

- `frontier`: Coada de priorități care ordonează nodurile pe baza valorii euristicii `h(n)`.
- `visited`: Set care ține evidența stărilor deja explorate pentru a evita ciclurile.
- La fiecare iterație, nodul cu cea mai mică valoare a euristicii este extins.

Observații:

- În urma implementării mele am constatat că este mai rapid decât ceilalți algoritmi de căutare în găsirea unei soluții inițiale, deoarece se concentrează pe nodurile care par cele mai promițătoare, dar nu garantează găsirea soluției optime, deoarece ignoră costul acumulat (`g(n)`).
- Poate să nu funcționeze corespunzător dacă funcția euristică nu este bine aleasă.

Rezultate: Algoritmul a fost testat pe mai multe hărți folosind următoarele comenzi:

```
python pacman.py -l bigMaze -p SearchAgent -a fn=gbfs,heuristic=manhattanHeuristic
python pacman.py -l mediumMaze -p SearchAgent -a fn=gbfs,heuristic=euclideanHeuristic
```

Concluzii:

- GBFS poate fi eficient în găsirea rapidă a unei soluții, dar nu este întotdeauna cea optimă.
- Performanța depinde puternic de funcția euristică utilizată.

3.5 Implementarea euristicii bazate pe MST

Descriere: Pentru a îmbunătăți performanța algoritmului GBFS în `FoodSearchProblem`, am integrat euristica bazată pe Arborele Minim de Acoperire (MST) descrisă anterior. Am făcut și o funcție separată pentru a calcula MST.

Cod:

```
def computeMST(edges, nodes):
    edges.sort()
    parent = {node: node for node in nodes}

    def find(node):
        while parent[node] != node:
            node = parent[node]
        return node

    def union(node1, node2):
        parent[find(node1)] = find(node2)

    mst_cost = 0
    for cost, node1, node2 in edges:
        if find(node1) != find(node2):
            union(node1, node2)
            mst_cost += cost

    return mst_cost

def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    position, foodGrid = state
    foodList = foodGrid.asList()
    nodes = [position] + foodList
```

```
25
26      if not foodList:
27          return 0
28
29      edges = []
30      for i in range(len(nodes)):
31          for j in range(i + 1, len(nodes)):
32              node1 = nodes[i]
33              node2 = nodes[j]
34              distance = util.manhattanDistance(node1, node2)
35              edges.append((distance, node1, node2))
36
37      mst_cost = computeMST(edges, nodes)
38
39      return mst_cost
```

Testare și rezultate: Prin utilizarea euristicii bazate pe MST în GBFS, am observat o îmbunătățire a performanței în termeni de număr de noduri expandate și timp de execuție, deși GBFS nu garantează soluția optimă.
Observații:

- În urma analizei și comparației, am constatat că A* rămâne superior în găsirea soluțiilor optime atunci când este utilizată o euristică admisibilă și consistentă.