UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Oprisor Paul si Turda Sorin
Grupa: 30232

Facultatea de Automatica
si Calculatoare

4 Decembrie 2024

Cuprins

1 Uninformed search

1.1 Question 1 - Finding a Fixed Food Dot using Depth-First Search

Gasirea unui punct unde se afla mancare folosind Depth-First Search

Depth-First Search este un algoritm utilizat pentru explorarea grafurilor sau arborilor. Scopul acestuia este de a traversa toate nodurile unui graf, urmărind o cale cât mai adâncă înainte de a reveni și a explora căile neexplorate.

Algorithm 1 Depth-First Search

```
1: function DFS(problem)
2:     stack ← Stack()
3:     visited ← ∅
4:     node ← problem.getStartState()
5:     stack.push((node, [ ]))
6:     while not stack.isEmpty() do
7:         position, path ← stack.pop()
8:         if position ∉ visited then
9:             visited.add(position)
10:            if problem.isGoalState(position) then return path
11:            end if
12:            for (successor, direction, cost) in problem.getSuccessors(position) do
13:                if successor ∉ visited then
14:                    stack.push((successor, path + [direction]))
15:                end if
16:            end for
17:        end if
18:    end while
19: return [ ]
20: end function
```

Complexitatea algoritmului:

- Timp: $O(b^d)$ unde b este factorul de ramificare (numarul mediu de succesori) si d este adancimea maxima a arborelui
- Spațiu: $O(b*d)$ - trebuie sa stocam nodurile de pe calea curenta plus nodurile de pe acelasi nivel

Avantaje:

- Implementare simpla si intuitiva
- Necesita mai putina memorie decat BFS deoarece exploreaza in adancime
- Poate gasi rapid o solutie daca aceasta se afla pe o ramura explorata devreme

Dezavantaje:

- Nu garanteaza gasirea celui mai scurt drum
- Poate ramane blocata explorând cai foarte lungi/infinite daca nu se implementeaza detectia ciclurilor

1.2 Question 2 - Breadth-first search

Breadth-first search este un algoritm utilizat pentru traversarea și căutarea în grafuri sau arbori. Spre deosebire de DFS, BFS explorează nodurile pe niveluri, adică parcurge mai întâi toate nodurile aflate la o anumită distanță de nodul de start înainte de a trece la nodurile mai îndepărtate. BFS este implementat utilizând o coadă pentru a gestiona ordinea vizitării nodurilor.

Algorithm 2 Breadth-First Search

```
1:  function BFS(problem)
2:      queue ← Queue()
3:      visited ← ∅
4:      node ← problem.getStartState()
5:      queue.push((node, [ ]))
6:      while not queue.isEmpty() do
7:          position, path ← queue.pop()
8:          if position ∉ visited then
9:              visited.add(position)
10:             if problem.isGoalState(position) then return path
11:             end if
12:             for (successor, direction, cost) in problem.getSuccessors(position) do
13:                 if successor ∉ visited then
14:                     queue.push((successor, path + [direction]))
15:                 end if
16:             end for
17:         end if
18:     end while
19: return [ ]
20: end function
```

Complexitatea algoritmului:

- Timp: $O(b^{d+1})$ unde b este factorul de ramificare și d este adâncimea la care se găsește prima soluție
- Spațiu: $O(b^d)$ - trebuie să stocheze toate nodurile de pe nivelul curent și următorul nivel

BFS este optimal, spațiul fiind o mare problemă

Avantaje:

- Garantează găsirea celei mai scurte căi până la soluție
- Explorează sistematic toate nodurile de pe un nivel înainte de a trece la următorul
- Potrivit pentru spații de căutare cu factor de ramificare mic și soluții la adâncimi mici

Dezavantaje:

- Consumă mai multă memorie decât DFS deoarece trebuie să stocheze toate nodurile de pe un nivel
- Nu este potrivit pentru probleme cu ramuri infinite
- Poate fi ineficient pentru spații de căutare mari cu soluții la adâncimi mari

1.3 Question 3 - Uniform Cost Search

Algoritmul Uniform Cost Search aplicat pe un graf găsește calea cu cel mai mic cost între un nod inițial și un nod țintă. Este o variantă a algoritmului Breadth-First Search, dar ia în considerare costurile asociate cu muchiile, fiind o metodă de cautare informată.

Algorithm 3 Uniform Cost Search

```
1: function UniformCostSearch(problem)
2:     pq ← PriorityQueue()
3:     visited ← ∅
4:     node ← problem.getStartState()
5:     pq.update((node, [ ], 0), 0)
6:     while not pq.isEmpty() do
7:         position, path, total_cost ← pq.pop()
8:         if position ∉ visited then
9:             visited.add(position)
10:            if problem.isGoalState(position) then return path
11:            end if
12:            for (successor, direction, cost) in problem.getSuccessors(position) do
13:                if successor ∉ visited then
14:                    pq.update((successor, path + [direction], total_cost + cost), total_cost +
    cost)
15:                end if
16:            end for
17:        end if
18:    end while
19: return [ ]
20: end function
```

Implementare:

- Foloseste o cu priorități pentru alegerea frontierei
- Intr-un set se pastrează toate nodurile vizitate

Complexitatea spatiului si a timpului paote fi mai mare decat cea a cautarii in adancime (DFS). Algoritmul este optimal, iar completitudinea este garantata daca costul fiecarei actiuni este strict pozitiv.

2 Informed search

2.1 Question 4 - A* search algorithm

A* este un algoritm de căutare informată/euristică Combină căutarea bazată pe cost (ca în UCS) cu o estimare euristică a distanței până la destinație. Este asemenea dependent de euristica folosită.

Algorithm 4 A* Search

```
1: function AStarSearch(problem, heuristic)
2:     start ← problem.getStartState()
3:     queue ← PriorityQueue()
4:     bestCost ← {start: 0}
5:     queue.push((start, []), 0)
6:     while not queue.isEmpty() do
7:         pos, path ← queue.pop()
8:         if problem.isGoalState(pos) then return path
9:         end if
10:        for (successor, direction, cost) in problem.getSuccessors(pos) do
11:            if bestCost[pos] + cost < bestCost.setdefault(successor, ∞) then
12:                bestCost[successor] ← cost + bestCost[pos]
13:                newCost ← bestCost[successor] + heuristic(successor, problem)
14:                queue.update((successor, path + [direction]), newCost)
15:            end if
16:        end for
17:    end while
18: return [ ]
19: end function
```

Implementare si Structuri de Date:

- Coadă cu priorități (PriorityQueue) pentru selectarea nodului cu cost minim
- Menține un dicționar pentru costurile minime până la fiecare nod
- Funcția de prioritate f(n) = g(n) + h(n), unde:
  - g(n) este costul real până la nodul n
  - h(n) este euristica manhattan pentru distanța estimată până la țintă

Euristica Manhattan:

- Calculează $|x_1 - x_2| + |y_1 - y_2|$ între poziția curentă și pozitia urmatoare
- Este admisibilă (nu supraestimeaza costul real) si consistenta (respecta inegalitatea triunghiului)

Complexitate:

- Timp si spațiu: $O(b^d)$ - trebuie să păstreze în memorie nodurile explorate

Avantaje:

- Garantează găsirea drumului optim când euristica este admisibilă
- Mai eficient decât BFS/DFS prin folosirea euristicii pentru ghidarea căutării
- Explorează mai puține noduri decât algoritmii de căutare neinformată

Dezavantaje:

- Spatiul utilizat
- Implementarea este mai complexă decât BFS/DFS

2.2 Question 5 - Găsirea tuturor colțurilor

Pentru fiecare colț care nu a fost parcurs se calculeaza distanta manhattan pană acolo. Se alege distanța minimă, prin urmare colțul cu acea distanță se va parcurge.

Algorithm 5 getStartState

```
function getStartState
    return (startingPosition, (False, False, False, False))
end function
```

Inițial, niciun colț nu este vizitat. Tupla returnata de functia **getStartState** specifică acest lucru.

Algorithm 6 isGoalState

```
function isGoalState(state)
    position, visitedCorners ← state
    return visitedCorners = (True, True, True, True)
end function
```

isGoalState verifică dacă toate colțurile au fost parcurse de Pacman

Algorithm 7 getSuccessors

```
function getSuccessors(state)
    successors ← empty list
    position, visitedCorners ← state
    x, y ← position
    for action in [NORTH, SOUTH, EAST, WEST] do
        dx, dy ← directionToVector(action)
        nextX ← x + dx
        nextY ← y + dy
        nextPos ← (nextX, nextY)
        if not walls[nextX][nextY] then
            newVisitedCorners ← list(visitedCorners)
            for i in range(4) do
                if corners[i] = nextPos then
                    newVisitedCorners[i] ← True
                end if
            end for
            cost ← 1
            Add ((nextPos, tuple(newVisitedCorners)), action, cost) to successors
        end if
    end for
    return successors
end function
```

2.3 Question 6 - Corners Problem: Heuristic

Algorithm 8 cornersHeuristic

```
1: function cornersHeuristic(state, problem)
2:     position, visitedCorners ← state
3:     if visitedCorners = (True, True, True, True) then
4:         return 0
5:     end if
6:     manhattanDistances ← empty list
7:     for i in range(4) do
8:         if not visitedCorners[i] then
9:             distance ← manhattanDistance(corners[i], position)
10:            Add distance to manhattanDistances
11:        end if
12:    end for
13:    return max(manhattanDistances)
14: end function
```

Pentru fiecare colț nevizitat, se calculează distanța manhattan si se alege cea mai mare dintre toate.

2.4 Question 7 - Eating All The Dots

Algorithm 9 foodHeuristic

```
1: function FOODHEURISTIC(state, problem)
2:     position, foodGrid ← state
3:     foodPositions ← foodGrid.asList()
4:     totalDistance ← 0
5:     if length(foodPositions) = 0 then
6:         return 0
7:     end if
8:     if length(foodPositions) = 1 then
9:         return manhattanDistance(position, foodPositions[0])
10:    end if
11:    closestFoodDistance ← ∞
12:    for food in foodPositions do
13:        closestFoodDistance ← min(closestFoodDistance, manhattanDistance(position, food))
14:    end for
15:    closestFood ← foodPositions[0]
16:    while foodPositions is not empty do
17:        nextClosestFood ← foodPositions[0]
18:        nextMinDistance ← ∞
19:        for food in foodPositions do
20:            currentDistance ← manhattanDistance(closestFood, food)
21:            if currentDistance < nextMinDistance then
22:                nextMinDistance ← currentDistance
23:                nextClosestFood ← food
24:            end if
25:        end for
26:        closestFood ← nextClosestFood
27:        totalDistance ← totalDistance + nextMinDistance
28:        Remove nextClosestFood from foodPositions
29:    end while
30:    return closestFoodDistance + totalDistance
31: end function
```

La început se află distanța minimă până la cea mai apropiată mâncare folosind distanța manhattan. Apoi, se estimează o distanță totală pentru colectarea restului de mâncare. Cu alte cuvinte se construieste un drum minim ce leagă toate punctele cu mâncare:

- Se caută următoarea cea mai apropiată bucată de mâncare rămasă
- Se adaugă distanța până la aceasta la suma totală
- Se elimină bucata de mâncare găsită din lista de căutare
- Procesul continuă până când s-au găsit toate punctele cu mâncare

Rezultatul final este distanța până la cea mai apropiată mâncare + suma distanțelor minime între toate bucățile de mâncare. Euristica este admisibilă pentru că nu supraestimează niciodată costul real.

2.5 Question 8 - Suboptimal Search

Algorithm 10 findPathToClosestDot

```
1: function findPathToClosestDot(gameState)
2:     problem ← AnyFoodSearchProblem(gameState)
3:     return breadthFirstSearch(problem)
4: end function
```

Funcția găseste cel mai scurt drum până la un punct cu mâncare. BFS va găsi garantat drumul cel mai scurt până la cea mai apropiată bucată de mâncare.

3 Adversarial search

3.1 Question 9 - Improve the ReflexAgent

Funcția de evaluare alege optim următoarea mâncare spre care va avansa Pacman în funcție de poziția fantomelor. Se află distanța manhattan pentru mâncare si fantomă. Dacă distanța până la fantomă e foarte aproape (distanță de o mutare), Pacman nu se va apropia de fantomă (se returnează o valoare foarte mică e.g. $-\infty$

Pacman va căuta tot timpul mâncarea cea mai apropiată de el. Se returnează $1/distanta$ deoarece scorul este invers proporțional cu distanța până la mâncare.

Algorithm 11 Evaluation Function

```
1: procedure EVALUATIONFUNCTION(self, currentGameState, action)
2:     minFoodDist ← ∞
3:     for food ∈ newFood.asList() do
4:         minFoodDist ← min(minFoodDist, manhattanDistance(food, newPos))
5:     end for
6:     for ghostState ∈ newGhostStates do
7:         if manhattanDistance(ghostState.getPosition(), newPos) < 2 then return −∞
8:         end if
9:     end for
10:    return 1/minFoodDist + successorGameState.getScore()
11: end procedure
```

3.2 Question 10 - Minimax

Algorithm 12 getAction

```
1: function GETACTION(gameState)
2:     maximum ← −∞
3:     bestAction ← SOUTH
4:     for action in gameState.getLegalActions(0) do
5:         currentScire ← MINIMAX(1, 0, gameState.generateSuccessor(0, action))
6:         if currentScire > maximum then
7:             maximum ← currentScire
8:             bestAction ← action
9:         end if
10:    end for
11:    return bestAction
12: end function
```

Algoritmul Minimax este o metodă clasică pentru luarea deciziilor în jocurile cu sumă nulă și informații perfecte, în cazul de față: Pacman.

Complexitatea algoritmului:

- Timp: $O(b^d)$
- Spațiu: $O(b * d)$

Avantaje:

- Găsește întotdeauna cea mai bună mișcare până la adâncimea dată
- Joacă perfect împotriva fantomelor care joacă optimal

Factori care influențează complexitatea

- Numărul de fantome (crește factorul de ramificare b)
- Dimensiunea labirintului

Algorithm 13 minimax

```
1: function MINIMAX(agent, depth, gameState)
2:     if gameState.isLose() or gameState.isWin() or depth = self.depth then
3:         return evaluationFunction(gameState)
4:     end if
5:     if agent = 0 then                                  ▷ Pacman's turn (MAX player)
6:         maxValue ← −∞
7:         for action in gameState.getLegalActions(agent) do
8:             successor ← gameState.generateSuccessor(agent, action)
9:             value ← MINIMAX(1, depth, successor)
10:            maxValue ← max(maxValue, value)
11:        end for
12:        return maxValue
13:    else                                               ▷ Ghost's turn (MIN player)
14:        minValue ← ∞
15:        nextAgent ← (agent + 1) mod gameState.getNumAgents()
16:        if nextAgent = 0 then                          ▷ Back to Pacman
17:            depth ← depth + 1
18:        end if
19:        for action in gameState.getLegalActions(agent) do
20:            successor ← gameState.generateSuccessor(agent, action)
21:            value ← MINIMAX(nextAgent, depth, successor)
22:            minValue ← min(minValue, value)
23:        end for
24:        return minValue
25:    end if
26: end function
```

3.3 Question 11 - $\alpha - \beta$ Pruning

Algorithm 14 getAction with Alpha-Beta Pruning

```
1: function GETACTION(gameState)
2:     maximum ← −∞
3:     α, β ← −∞, ∞
4:     bestAction ← SOUTH
5:     for action in gameState.getLegalActions(0) do
6:         currentScore ← ALFABETA(1, 0, gameState.generateSuccessor(0, action), α, β)
7:         if currentScore > maximum then
8:             maximum ← currentScore
9:             bestAction ← action
10:        end if
11:        α ← max(α, maximum)
12:    end for
13:    return bestAction
14: end function
```

Algoritmul **Alpha-Beta Pruning** este o optimizare a algoritmului Minimax, care reduce numărul de stări evaluate fără a afecta rezultatul final. În loc să evalueze toate nodurile arborelui, Alpha-Beta ignoră ramurile care nu influențează decizia finală. Performanța optimă este atinsă atunci când nodurile sunt explorate într-o ordine bună. Dacă ordinea explorării este ideală, complexitatea se reduce la $O(b^{d/2})$. In caz contrar, performanța este similară cu Minimax.
Alpha-Beta produce același rezultat ca Minimax fară să influențeze calitatea deciziilor.

Algorithm 15 alfabeta

```
1: function ALFABETA(agent, depth, gameState, α, β)
2:     if gameState.isLose() or gameState.isWin() or depth = self.depth then
3:         return evaluationFunction(gameState)
4:     end if
5:     if agent = 0 then                                   ▷ Pacman's turn (MAX player)
6:         value ← −∞
7:         for action in gameState.getLegalActions(agent) do
8:             successor ← gameState.generateSuccessor(agent, action)
9:             value ← max(value, ALFABETA(1, depth, successor, α, β))
10:            if value > β then
11:                return value                                          ▷ Beta cutoff
12:            end if
13:            α ← max(α, value)
14:        end for
15:        return value
16:    else                                                ▷ Ghost's turn (MIN player)
17:        value ← ∞
18:        nextAgent ← (agent + 1) mod gameState.getNumAgents()
19:        if nextAgent = 0 then
20:            depth ← depth + 1
21:        end if
22:        for action in gameState.getLegalActions(agent) do
23:            successor ← gameState.generateSuccessor(agent, action)
24:            value ← min(value, ALFABETA(nextAgent, depth, successor, α, β))
25:            if value < α then
26:                return value                                          ▷ Alpha cutoff
27:            end if
28:            β ← min(β, value)
29:        end for
30:        return value
31:    end if
32: end function
```