UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Bogdan Tudor
Grupa: 30239

Facultatea de Automatica
si Calculatoare

12 Decembrie 2024

Cuprins

1 Introducere

1.1 Context

Pac-Man, numit inițial Puck Man în Japonia, este un joc video de acțiune labirint din 1980 dezvoltat și lansat de Namco pentru arcade. În America de Nord, jocul a fost lansat de Midway Manufacturing ca parte a acordului de licențiere cu Namco America. Jucătorul controlează pe Pac-Man, care trebuie să mănânce toate punctele înăuntrul unui labirint închis în timp ce evită patru fantome colorate. Mâncând puncte mari intermitente numite "Peleți de putere" fac ca fantomele să devină temporar albastre, permițând lui Pac-Man să le mănânce pentru puncte bonus.

1.1.1 Istorie

Dezvoltarea jocului a pornit la începutul lui 1979, regizat de Toru Iwatani cu o echipă de nouă oameni. Iwatani a vrut să creeze un joc care să le placă atât femeilor cât și bărbaților, deoarece majoritatea jocurilor de pe vremea aceea aveau teme fie de război, fie de sport.[2][3] Deși inspirația pentru personajul Pac-Man a fost o imagine a unei pizza cu o felie tăiată, Iwatani a spus că a rotunjit cuvântul japonez pentru gură, kuchi. Personajele din joc au fost făcute să fie drăguțe și colorate pentru a atrage jucătorii mai tineri. Titlul original japonez Puck Man a fost derivat din fraza japoneză paku paku taberu, care se referă la a înghiți ceva; titlul a fost schimbat în Pac-Man pentru lansarea nord-americană.

1.2 Elemente de baza

Proiectul își propune să dezvolte un agent inteligent Pac-Man capabil să navigheze eficient printr-un labirint, consumând toate punctele și evitând întâlnirile cu fantomele într-un număr minim de pași. Pentru a realiza acest lucru, am implementat algoritmi de căutare neinformate (DFS, BFS, UCS) și informate (A*), care se diferențiază prin gradul de informație despre problemă utilizat în procesul de decizie. De asemenea, am explorat comportamentul mai multor tipuri de agenți, inclusiv ReflexAgent, Minimax și Alpha-Beta, astfel încât Pac-Man să poată lua decizii strategice, să evite pericolele și să finalizeze cu succes sarcina.

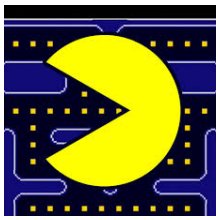

Figura 1: Pacman

2 Uninformed search

2.1 Question 1 - Depth-first search

Cautarea in adancime incearca intotdeauna sa extinda nodul cel mai profund din stiva arborelui de cautare. Acest algoritm se bazeaza pe o structura de date de tip stiva (LIFO - ultimul

intrat, primul iesit). Prin intermediul stivei, nodul generat cel mai recent este selectat pentru expansiune.

Funcția `depthFirstSearch` implementează algoritmul de căutare în adâncime pentru rezolvarea unei probleme date. Pașii algoritmului sunt următorii:

1. Se creează o structură de tip stivă (*state stack*) pentru gestionarea stărilor.
2. Starea inițială este adăugată în stivă, împreună cu un traseu gol (*path*).
3. Un set (*visited*) este inițializat pentru a reține stările deja explorate.
4. Cât timp stiva nu este goală:
   - Se extrage cel mai recent element (*popped element*) din stivă.
   - Se obțin starea curentă și traseul asociat acesteia.
   - Dacă starea curentă coincide cu starea scop, se returnează traseul calculat până în acel moment.
   - În cazul în care starea curentă nu a fost vizitată anterior:
     - Se marchează starea ca vizitată.
     - Se generează succesorii pentru starea curentă și se iterează prin aceștia.
     - Pentru fiecare succesor nevizitat, se adaugă în stivă împreună cu traseul actualizat (*path + [direction]*).

```
75  def depthFirstSearch(problem):
76      """
77      Search the deepest nodes in the search tree first.
78
79      Your search algorithm needs to return a list of actions that reaches the
80      goal. Make sure to implement a graph search algorithm.
81
82      To get started, you might want to try some of these simple commands to
83      understand the search problem that is being passed in:
84
85      print "Start:", problem.getStartState()
86      print "Is the start a goal?", problem.isGoalState(problem.getStartState())
87      print "Start's successors:", problem.getSuccessors(problem.getStartState())
88      """
89      "*** YOUR CODE HERE ***"
90      state_stack = util.Stack()
91      start = problem.getStartState()
92      state_stack.push((start, []))
93      visited = set()
94
95      while not state_stack.isEmpty():
96          popped_element = state_stack.pop()
97          current_state = popped_element[0]
98          path = popped_element[1]
99          if problem.isGoalState(current_state):
100             return path
101         if current_state not in visited:
102             visited.add(current_state)
103             successors = problem.getSuccessors(current_state)
104             for next_state, direction, _ in successors:
```

```
105                 if next_state not in visited:
106                     state_stack.push((next_state, path + [direction]))
```

2.2 Question 2 - Breadth-first search

Căutarea (parcurgerea) în lățime (BFS) este un algoritm pentru parcurgerea sau căutarea într-o structură de date de tip arbore sau graf. Aceasta începe cu rădăcina arborelui (sau cu un nod arbitrar dintr-un graf, uneori denumit „cheie de căutare") și explorează nodurile mai întâi nodurile vecine acestuia, înainte de a trece la vecinii de pe nivelul următor (vecinii vecinilor).

Funcția `breadthFirstSearch` implementează algoritmul de căutare în lățime (BFS) pentru a rezolva o problemă dată. Pașii acestui algoritm sunt următorii:

1. Se inițializează o coadă (*state queue*) care va fi utilizată pentru gestionarea stărilor.
2. Starea inițială este introdusă în coadă, împreună cu un traseu gol (*path*) asociat.
3. Un set (*visited*) este inițializat pentru a urmări stările care au fost deja explorate.
4. Cât timp coada nu este goală:
    - Se scoate primul element din coadă (*popped element*).
    - Se extrag starea curentă și traseul aferent acesteia.
    - Dacă starea curentă corespunde stării scop, se returnează traseul construit până în acel moment.
    - Dacă starea curentă nu a fost încă vizitată:
        - Se marchează starea ca fiind vizitată.
        - Se obțin succesorii stării curente și se iterează prin aceștia.
        - Pentru fiecare succesor nevizitat, acesta este adăugat în coadă împreună cu traseul actualizat (*path + [direction]*).

```
110 def breadthFirstSearch(problem):
111     """Search the shallowest nodes in the search tree first."""
112     "*** YOUR CODE HERE ***"
113     state_queue = util.Queue()
114     start = problem.getStartState()
115     state_queue.push((start, []))
116     visited = set()
117 
118     while not state_queue.isEmpty():
119         popped_element = state_queue.pop()
120         current_state = popped_element[0]
121         path = popped_element[1]
122         if problem.isGoalState(current_state):
123             return path
124         if current_state not in visited:
125             visited.add(current_state)
126             successors = problem.getSuccessors(current_state)
127             for next_state, direction, _ in successors:
128                 if next_state not in visited:
129                     state_queue.push((next_state, path + [direction]))
```

2.3 Question 3 - Uniform cost search

Uniform Cost Search (UCS) este un algoritm de căutare popular utilizat în inteligența artificială (IA) pentru a găsi calea cu costul cel mai mic într-un graf. Este o variantă a algoritmului lui Dijkstra și este deosebit de util atunci când toate muchiile grafului au greutăți diferite, iar scopul este de a identifica traseul cu costul total minim de la un nod de start la un nod de scop.

Funcția `uniformCostSearch` implementează algoritmul de căutare cu cost uniform (UCS), care găsește calea cu costul total minim pentru a rezolva o problemă dată. Etapele de execuție sunt următoarele:

1. Se inițializează o coadă de priorități (*state queue*), care va gestiona stările pe baza costului total al fiecărei căi.
2. Starea inițială este adăugată în coadă, împreună cu o listă goală de acțiuni (*path*) și un cost inițial de zero.
3. Un set (*visited*) este utilizat pentru a reține stările deja explorate.
4. Cât timp coada nu este goală:
   - Se extrage starea cu costul total cel mai mic din coadă.
   - Se obțin starea curentă, traseul (*path*) asociat acesteia și costul acumulat.
   - Dacă starea curentă este starea scop, algoritmul returnează traseul calculat.
   - Dacă starea curentă nu a fost deja vizitată:
     - Se adaugă starea curentă în setul de stări vizitate.
     - Se obțin succesorii stării curente, fiecare având asociată o direcție și un cost.
     - Pentru fiecare succesor care nu a fost vizitat:
       * Se calculează costul total al drumului către acel succesor.
       * Succesorul este adăugat în coada de priorități, împreună cu traseul actualizat (*path + [direction]*) și costul total.

```
132 def uniformCostSearch(problem):
133     """Search the node of least total cost first."""
134     "*** YOUR CODE HERE ***"
135     state_queue = util.PriorityQueue()
136     start = problem.getStartState()
137     state_queue.push((start, [], 0), 0)
138     visited = set()
139 
140     while not state_queue.isEmpty():
141         popped_element = state_queue.pop()
142         current_state = popped_element[0]
143         path = popped_element[1]
144         cost = popped_element[2]
145         if problem.isGoalState(current_state):
146             return path
147         if current_state not in visited:
148             visited.add(current_state)
149             successors = problem.getSuccessors(current_state)
150             for next_state, direction, state_cost in successors:
151                 if next_state not in visited:
152                     total_cost = cost + state_cost
153                     state_queue.push((next_state, path + [direction], total_cost), total_cos
```

3 Informed search

3.1 Question 4 - A* search algorithm

Algoritmul `aStarSearch` implementează căutarea A* pentru rezolvarea unei probleme date, având ca obiectiv găsirea celui mai scurt drum către starea scop, folosind o combinație de costul drumului parcurs și o estimare a distanței până la scop (heuristică). Pașii algoritmului sunt următorii:

1. Se creează o structură de tip coadă de priorități (*state queue*) pentru gestionarea stărilor, unde fiecare element conține starea curentă, traseul asociat și costul până la acea stare.
2. Se obține starea de start și se adaugă în coada de priorități împreună cu un traseu gol și costul inițial 0.
3. Se creează un set (*visited*) pentru a urmări stările deja vizitate, pentru a evita procesarea lor de mai multe ori.
4. Cât timp coada de priorități nu este goală:
   - Se extrage elementul cu cel mai mic cost total (*popped element*) din coada de priorități.
   - Se obțin starea curentă, traseul asociat și costul acumulat până în acest punct.
   - Dacă starea curentă este starea scop, se returnează traseul curent, care reprezintă soluția.
   - Dacă starea curentă nu a fost vizitată anterior:
     - Se marchează starea curentă ca vizitată.
     - Se generează succesorii stării curente și se iterează prin aceștia.
     - Pentru fiecare succesor nevizitat, se calculează costul total (*costul actual + costul pasului curent + estimarea heuristică*) și se adaugă succesorul în coada de priorități cu costul total ca prioritate.

```
163 def aStarSearch(problem, heuristic=nullHeuristic):
164     """Search the node that has the lowest combined cost and heuristic first."""
165     "*** YOUR CODE HERE ***"
166     """Search the node that has the lowest combined cost and heuristic first."""
167     "*** YOUR CODE HERE ***"
168     state_queue = util.PriorityQueue()
169     start = problem.getStartState()
170     state_queue.push((start, [], 0), 0)
171     visited = set()
172 
173     while not state_queue.isEmpty():
174         popped_element = state_queue.pop()
175         current_state = popped_element[0]
176         path = popped_element[1]
177         cost = popped_element[2]
178         if problem.isGoalState(current_state):
179             return path
180         if current_state not in visited:
181             visited.add(current_state)
182             successors = problem.getSuccessors(current_state)
183             for next_state, direction, state_cost in successors:
184                 if next_state not in visited:
185                     new_cost = cost + state_cost
```

```
186                     total_cost = new_cost + heuristic(next_state, problem)
187                     state_queue.push((next_state, path + [direction], new_cost), total_cost)
```

4 Adversarial search

4.1 Question 1 - Reflex Agent

Algoritmul ReflexAgent implementează un agent reflexiv care ia o decizie la fiecare pas, alegând acțiunea care maximizează o funcție de evaluare a stării. Funcția de evaluare ia în considerare atât poziția curentă a lui Pacman, cât și caracteristicile mediului înconjurător, precum locațiile alimentelor, fantomele și capsulele. Pașii algoritmului sunt următorii:

1. Se colectează mișcările legale posibile pentru Pacman (*legalMoves*).
2. Pentru fiecare mișcare posibilă, se calculează scorul de evaluare asociat folosind funcția `evaluationFunction`.
3. Se selectează acțiunea care are cel mai mare scor de evaluare. Dacă mai multe acțiuni au același scor, se alege aleator un index din aceste acțiuni.
4. Funcția de evaluare (`evaluationFunction`) calculează o valoare care reflectă calitatea unei stări posibile după efectuarea unei acțiuni, folosind următoarele informații:
   - Poziția lui Pacman (*newPos*).
   - Harta alimentelor (*newFood*) și distanța față de cele mai apropiate alimente.
   - Pozițiile fantomelor (*newGhostStates*) și distanța față de cea mai apropiată fantomă.
   - Distanța față de capsulele (*capsules*) care pot îmbunătăți scorul.
5. Funcția de evaluare este construită astfel încât să maximizeze distanța față de fantome (pentru a evita coliziunile) și să minimizeze distanța față de alimente pentru a colecta mai multe. De asemenea, un bonus semnificativ este acordat când Pacman ajunge la o capsulă.
6. Dacă o fantomă se află prea aproape (distanța ¡= 1), se reduce scorul pentru a reflecta pericolul, în timp ce distanțele mai mari față de fantome și alimentele mai apropiate sunt favorizate.

```
77          evaluation = successorGameState.getScore()
78          capsules = successorGameState.getCapsules()
79          closestFoodDistance = float('inf')
80          closestGhostDistance = float('inf')
81          for food in newFood.asList():
82              distance = util.manhattanDistance(newPos, food)
83              if distance < closestFoodDistance:
84                  closestFoodDistance = distance
85          for ghostState in newGhostStates:
86              ghostPos = ghostState.getPosition()
87              distance = util.manhattanDistance(newPos, ghostPos)
88              if distance < closestGhostDistance:
89                  closestGhostDistance = distance
90          evaluation = successorGameState.getScore()
91          if closestGhostDistance <= 1:
92              evaluation -= 500
93          else:
94              evaluation += 1.0 / (closestFoodDistance + 1)
95          if newPos in capsules:
96              evaluation += 500
```

```
97          return evaluation
98          #return successorGameState.getScore()
```

4.2 Question 2 - Minmax Algorithm

Algoritmul `Minimax` este utilizat pentru a lua decizii într-un joc cu doi sau mai mulți agenți, cum ar fi jocul Pacman. Algoritmul caută să maximizeze scorul pentru un agent (de obicei Pacman), în timp ce minimizează scorul pentru adversari (fantomele), pe baza unei funcții de evaluare. Pașii algoritmului sunt următorii:

1. Se definește o funcție recursivă `minimax`, care explorează toate stările posibile ale jocului până la o adâncire predefinită (numărul de pași care pot fi efectuați) sau până când se atinge o stare terminală (când jocul se încheie cu o victorie sau o pierdere).
2. Algoritmul are trei parametri principali:
   - *state* – starea curentă a jocului.
   - *depth* – adâncimea maximă de căutare (adică, numărul de mișcări rămase).
   - *agentIndex* – indexul agentului care urmează să ia o decizie (Pacman este agentul 0, iar fantomele sunt agenți 1, 2, etc.).
3. Algoritmul funcționează în două moduri, în funcție de agentul care face mișcarea:
   - Dacă agentul curent este Pacman (agentul 0), atunci scopul său este de a maximiza valoarea scorului. Astfel, se aleg acțiunile care duc la cele mai bune scoruri posibile.
   - Dacă agentul curent este o fantomă (orice agent diferit de 0), scopul său este de a minimiza valoarea scorului, adică de a reduce șansele de câștig ale lui Pacman.
4. La fiecare nivel de recursiune, algoritmul iterează prin toate acțiunile legale disponibile pentru agentul curent și generează stările succesorilor (stările rezultate după aplicarea acțiunilor).
5. Funcția `minimax` se oprește în cazul în care:
   - Se atinge adâncimea maximă (*depth == 0*).
   - Starea curentă este o stare terminală (jocul s-a încheiat, *state.isWin()* sau *state.isLose()*).
6. Pentru fiecare stare succesor, funcția `minimax` calculează valoarea acesteia și o compară cu valoarea maximă sau minimă, în funcție de agentul curent.
7. În final, se returnează valoarea optimă și acțiunea corespunzătoare pentru agentul curent, alegând acțiunea care conduce la valoarea maximă (pentru Pacman) sau minimă (pentru fantome).
8. Când recursiunea ajunge la adâncimea dorită (sau la o stare terminală), se evaluează starea curentă utilizând o funcție de evaluare (`evaluationFunction`).

```
154         def minimax(state, depth, agentIndex):
155             if depth == 0 or state.isWin() or state.isLose():
156                 return self.evaluationFunction(state), None
157             if agentIndex == 0:  # Pacman
158                 bestValue = float("-inf")
159             else:  # Ghost
160                 bestValue = float("inf")
161             bestAction = None
162             legalActions = state.getLegalActions(agentIndex)
163             for action in legalActions:
164                 successor = state.generateSuccessor(agentIndex, action)
165                 value, _ = minimax(successor, depth - 1, (agentIndex + 1) % state.getNumAgen
166                 if (agentIndex == 0 and value > bestValue) or (agentIndex != 0 and value < b
```

```
                    bestValue = value
                    bestAction = action
            return bestValue, bestAction

        _, bestAction = minimax(gameState, self.depth * gameState.getNumAgents(), 0)
        return bestAction
```

4.3 Question 3 - Alpta-Beta Pruning Algorithm

Algoritmul `Alpha-Beta Pruning` este o îmbunătățire a algoritmului `Minimax` care reduce semnificativ numărul de noduri evaluate într-un arbore de decizie, prin eliminarea ramurilor care nu vor afecta rezultatul final. Acesta funcționează prin menținerea a două valori: *alpha* și *beta*, care reprezintă limitele minime și maxime ale valorilor posibile ale nodurilor. Pașii algoritmului sunt următorii:

1. Se definește o funcție recursivă `alphaBeta` care caută în adâncime arborele de decizie, similar cu algoritmul `Minimax`, dar cu utilizarea prunerii.
2. Algoritmul are patru parametri principali:
   - *state* – starea curentă a jocului.
   - *depth* – adâncimea maximă de căutare.
   - *alpha* – valoarea minimă pe care o poate obține un agent maximizator (Pacman).
   - *beta* – valoarea maximă pe care o poate obține un agent minimizator (fantomele).
   - *agentIndex* – indexul agentului curent.
3. Dacă adâncimea maximă (*depth*) este atinsă sau jocul a ajuns într-o stare terminală (victorie sau înfrângere), se returnează valoarea de evaluare a stării curente.
4. Algoritmul alternează între agenții care maximizează (Pacman) și agenții care minimizează (fantomele):
   - Dacă agentul curent este Pacman (agentul 0), atunci acesta va maximiza valoarea (alegerea celei mai mari valori posibile pentru scor).
   - Dacă agentul curent este o fantomă (agentul 1 sau altul), atunci acesta va minimiza valoarea (alegerea celei mai mici valori posibile pentru scor).
5. La fiecare pas, algoritmul îmbunătățește valorile *alpha* și *beta*:
   - *Alpha* reprezintă valoarea maximă pe care o poate obține agentul maximizator.
   - *Beta* reprezintă valoarea minimă pe care o poate obține agentul minimizator.
6. Dacă la un pas valoarea curentă depășește valoarea *beta* (pentru maximizator) sau dacă este mai mică decât valoarea *alpha* (pentru minimizator), se oprește recursiunea pentru acea ramură (pruning).
7. Dacă nu se face pruning, algoritmul continuă să exploreze ramurile următoare ale arborelui de decizie.
8. În final, funcția returnează valoarea optimă și acțiunea corespunzătoare.

1. Algoritmul recursiv funcționează astfel:
   - Dacă agentul curent este Pacman (agent 0), se încearcă să se maximizeze valoarea, iar valoarea *alpha* este actualizată cu valoarea maximă obținută.
   - Dacă agentul curent este o fantomă, se încearcă să se minimizeze valoarea, iar valoarea *beta* este actualizată cu valoarea minimă obținută.
   - Pruning-ul este realizat atunci când valoarea curentă depășește limita *alpha* sau *beta*.

`Alpha-Beta Pruning` reduce numărul de noduri care trebuie explorate prin eliminarea ramurilor care nu pot afecta rezultatul final, îmbunătățind astfel performanța algoritmului `Minimax`.

```
185
186         def alphaBeta(state, depth, alpha, beta, agentIndex):
187             if depth == 0 or state.isWin() or state.isLose():
188                 return self.evaluationFunction(state), None
189             if agentIndex == 0:  # Pacman
190                 bestValue = float("-inf")
191                 for action in state.getLegalActions(agentIndex):
192                     successor = state.generateSuccessor(agentIndex, action)
193                     value, _ = alphaBeta(successor, depth - 1, alpha, beta, (agentIndex + 1)
194                     if value > bestValue:
195                         bestValue = value
196                         bestAction = action
197                     if bestValue > beta:
198                         return bestValue, bestAction
199                     alpha = max(alpha, bestValue)
200                 return bestValue, bestAction
201             else:  # Ghost
202                 bestValue = float("inf")
203                 for action in state.getLegalActions(agentIndex):
204                     successor = state.generateSuccessor(agentIndex, action)
205                     value, _ = alphaBeta(successor, depth - 1, alpha, beta, (agentIndex + 1)
206                     if value < bestValue:
207                         bestValue = value
208                         bestAction = action
209                     if bestValue < alpha:
210                         return bestValue, bestAction
211                     beta = min(beta, bestValue)
212                 return bestValue, bestAction
213
214         _, bestAction = alphaBeta(gameState, self.depth * gameState.getNumAgents(), float("-
215         return bestAction
```