UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Pac-Man

Inteligenta Artificiala

Autori: Schiop Adrian-Marian si Mocoi Ioan Victor
Grupa: 30234

Facultatea de Automatica
si Calculatoare

17 Noiembrie 2022

Cuprins

1 Uninformed search

1.1 Question 1.1 - Depth-first search

Pentru a cauta cel mai adanc nod in graf trebuie sa utilizam algoritmul DFS. Pentru acest algoritm folosim o stiva in care vom adauga pe rand succesorii nodului curent. Ne folosim de proprietatea stivei, LIFO. Pentru inceput in stiva va fi pus nodul de *Start* si va fi marcat ca vizitat. In momentul cand scoatem din stiva un nod verificam daca acesta a fost expandat sau nu. In cazul in care nu a fost expandat se expandeaza si se continua algoritmul. Cand scoatem din stiva, verificam daca nodul curent nu este *goal state*. In caz afirmativ, se returneaza lista cu calea pana la nodul curent, altfel se continua algoritmul.

```
def depthFirstSearch(problem: SearchProblem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:
"""
    "*** YOUR CODE HERE ***"
    stack =  util.Stack()
    start_node = problem. getStartState()
    visited_nodes = []
    stack.push((start_node, []))
    if problem.isGoalState(start_node):
        return []

    while not stack.isEmpty():
        node, path = stack.pop()
        if node not in visited_nodes:
            visited_nodes.append(node)
            if problem.isGoalState(node):
                return path
            for next_node, action, stepCost in problem.getSuccessors(node):
                stack.push((next_node, path + [action]))

    util.raiseNotDefined()
```

Figura 1: DFS Algorithm

1.2 Question 1.2 - Breadth-first search

Pentru a cauta cel mai superficial nod trebuie sa utilizam algoritmul BFS. Pentru acest algoritm folosim o coada in care vom adauga pe rand succesorii nodului curent. Ne folosim de proprietatea cozii, FIFO. Pentru inceput in stiva va fi pus nodul de *Start* si va fi marcat ca vizitat. In momentul cand scoatem din stiva un nod verificam daca acesta a fost expandat sau nu. In cazul in care nu a fost expandat se expandeaza si se continua algoritmul. Cand scoatem din stiva, verificam daca nodul curent nu este *goal state*. In caz afirmativ, se returneaza lista cu calea pana la nodul curent, altfel se continua algoritmul.

```python
def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    queue =  util.Queue()
    visited_nodes = []
    start_node = problem. getStartState()
    queue.push((start_node, []))
    if problem.isGoalState(start_node):
        return []
    while not queue.isEmpty():
        node, path = queue.pop()
        if node not in visited_nodes:
            visited_nodes.append(node)
            if problem.isGoalState(node):
                return path
            for next_node, action, stepCost in problem.getSuccessors(node):
                queue.push((next_node, path + [action]))

    util.raiseNotDefined()
```

Figura 2: BFS Algorithm

1.3 Question 1.3 - Uniform-cost search

Cu ajutorul algoritmului Uniform Cost Search aflam nodul cu costul minim. Pentru acest algoritm folosim o coada de prioritati. Daca pana acum in structura de date, coada sau stiva adaugam nodul si calea catre nod, acum aduagam si costul pana la nodul curent. Pentru inceput, se aduaga nodul de *start* cu costul 0. Se parcurge coada de prioritati dupa nodul cu sotul cel mai mic. Daca nodul care este extras din coada de prioritati nu a mai fost vizitat, se marcheaza ca vizitat si se adauga succesorii acestuia in coada de prioritati. Algoritmul se finalizeaza in momentul in care *Goal State-ul* corespunde cu nodul curent.

```python
def uniformCostSearch(problem: SearchProblem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    queue =  util.PriorityQueue()
    visited_nodes = []
    start_node = problem. getStartState()
    queue.push((start_node, []), 0)
    if problem.isGoalState(start_node):
        return []
    while not queue.isEmpty():
        node, path = queue.pop()
        if node not in visited_nodes:
            visited_nodes.append(node)
            if problem.isGoalState(node):
                return path
            for next_node, action, stepCost in problem.getSuccessors(node):
                queue.push((next_node, path + [action]), problem.getCostOfActions(path + [action]))

    util.raiseNotDefined()
```

Figura 3: UCS Algorithm

2 Informed search

2.1 Question 1.4 - A* search algorithm

Algoritmul A* determina nodul ce are cel mai mic cost si cea mai buna aproximmare a distantei fata de nodul *Goal State*. La acest algoritm se foloseste o coada de prioritati. Diferenta dintre coada de prioritati de la UCS si A* este data de valoarea dupa care se extrage nodul din coada.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"


    queue =  util.PriorityQueue()
    start_node = problem. getStartState()
    visited_nodes = []
    queue.push((start_node, []), 0)

    if problem.isGoalState(start_node):
        return []
    while not queue.isEmpty():
        node, path = queue.pop()
        if node not in visited_nodes:
            visited_nodes.append(node)
            if problem.isGoalState(node):
                return path
            for next_node, action, stepCost in problem.getSuccessors(node):
                queue.push((next_node, path + [action]), problem.getCostOfActions(path + [action]) + heuristic(next_node, problem


    util.raiseNotDefined()
```

Figura 4: A* Algorithm

3 Adversarial search

3.1 Question 2.1- Improve the ReflexAgent

Aceasta parte din program are rolul de a evalua starea urmatoare a jocului si pentru a decide ce actiune ar trebuie sa execute Pacman in continuare. Functia responsabula pentru acest lucru ia in considerare mai multi factori: *pozitia curenta, mancarea ramasa, starea fantomelor, timpul ramas pana cand fantomele vor fi din nou eligibile*. Functia calculeaza distanta minima de la Pacman la macare si distanta de la Pacman la fantome. Scorul final returnat de funcție este calculat ca suma dintre scorul jocului curent și inversul distanței minime până la mâncare, minus inversul distanței totale până la fantome și numărul de fantome din proximitate. Acest scor este conceput astfel încât un scor mai mare este mai bun. Pacman va încerca să maximizeze acest scor prin alegerea acțiunii care duce la starea de joc cu cel mai mare scor.

```
successorGameState = currentGameState.generatePacmanSuccessor(action)
newPos = successorGameState.getPacmanPosition()
newFood = successorGameState.getFood()
newGhostStates = successorGameState.getGhostStates()
newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
"*** YOUR CODE HERE ***"
"distanta de la pacman la mancare"
minFoodDistance = -1
newFoodList = newFood.asList()
for food in newFoodList:
    distance = util.manhattanDistance(newPos, food)
    if minFoodDistance >= distance or minFoodDistance == -1:
        minFoodDistance = distance
"""distanta de la pacman la fantome"""
distancesGhosts = 1
proximityGhosts = 0
for ghost_state in successorGameState.getGhostPositions():
    distance = util.manhattanDistance(newPos, ghost_state)
    distancesGhosts += distance
    if distance <= 1:
        proximityGhosts += 1
return successorGameState.getScore() + (1 / float(minFoodDistance)) - (1 / float(distancesGhosts)) - proximityGhosts
```

Figura 5: ReflexAgent function

3.2 Question 2.2 - Minimax algorithm

Algoritmul MiniMax este folosit adeseori in teoria jocurilor pentru a determina cea mai buna miscare intr-un joc cu doi jucatori care se joaca pe rand. Acest algoritm este compus din doua functii ajutatoare *minFunc* si *maxFunc* ce au rolul de a implementa partile de minimizare respectiv maximizare ale algoritmului.

Funcția maxFunc este responsabilă pentru alegerea acțiunii care maximizează valoarea funcției de evaluare. Aceasta generează toate acțiunile legale pentru Pacman (agentID=0) și calculează valoarea funcției de evaluare pentru fiecare acțiune folosind funcția minFunc. Apoi, alege acțiunea care are valoarea maximă.

```
def maxFunc(gameState,depth):
    actions=gameState.getLegalActions(0)
    if len(actions)==0 or gameState.isWin() or gameState.isLose() or depth==self.depth:
        return(self.evaluationFunction(gameState),None)
    maxVal=-999999999
    actiune=None
    for act in actions:
        sucsValue=minFunc(gameState.generateSuccessor(0,act),1,depth)
        sucsValue=sucsValue[0]
        if(sucsValue>maxVal):
            maxVal,actiune=sucsValue,act
    return(maxVal,actiune)


maxFunc=maxFunc(gameState,0)[1]
return maxFunc
```

Figura 6: MinMax Algorithm maximisation

Funcția minFunc este responsabilă pentru alegerea acțiunii care minimizează valoarea funcției de evaluare. Aceasta generează toate acțiunile legale pentru fantome (agentIndex¿=1) și calculează valoarea funcției de evaluare pentru fiecare acțiune folosind funcția maxFunc sau minFunc, în funcție de agentul curent. Apoi, alege acțiunea care are valoarea minimă.

```
def minFunc(gameState,agentID,depth):
    actions=gameState.getLegalActions(agentID)
    if len(actions) == 0:
        return(self.evaluationFunction(gameState),None)
    minVal=999999
    actiune=None
    for act in actions:
        if(agentID==gameState.getNumAgents() -1):
            sucsValue=maxFunc(gameState.generateSuccessor(agentID,act),depth + 1)
        else:
            sucsValue=minFunc(gameState.generateSuccessor(agentID,act),agentID+1,depth)
        sucsValue=sucsValue[0]
        if(sucsValue<minVal):
            minVal,actiune=sucsValue,act
    return(minVal,actiune)
```

Figura 7: MinMax Algorithm minimisation

3.3 Question 2.3 - Alpha-Beta Pruning algorithm

Algoritmul Alpha Beta Pruning este o tehnica de optimizare al algoritmului MiniMax, ce reduce numarul de noduri ce trebuie sa fie evaluate. Functia ajutatoare *PacmanOrAgent* determina pentru ce tip de jucator se aplica algoritmul, Pacman sau fantoma. Daca functia returneaza agentul cu indexul=0 atunci agentul este Pacman si se merge pe maximizare, altfel este vorba de o fantoma si se continua cu minimizare. Functia *maxFunc* realizeaza taietura alpha-beta. Daca valoarea maxima curenta este mai mare decat beta, atunci functia se intoarce imediat cu valoarea maxima, taind astfel ramurile ramase de explorat.

```
def PacmanOrAgent(self, gameState, alpha, beta, depth, agent):
    if agent >= gameState.getNumAgents():
        depth    = depth + 1
        agent = 0
    if (depth == self.depth or gameState.isWin() or gameState.isLose()):
        return self.evaluationFunction(gameState)
    if agent == 0:
        return self.maxFunc(gameState, alpha, beta , depth, agent) #pacman
    else:
        return self.minFunc(gameState, alpha, beta,depth, agent )  #fantoma

def maxFunc(self, gameState, alpha, beta, depth, agent):
    maxVal = -999999
    legalActions = gameState.getLegalActions(agent)
    for action in legalActions:
        successorGameState = gameState.generateSuccessor(agent, action)
        maxVal = max(maxVal, self.PacmanOrAgent(successorGameState, alpha, beta, depth, agent + 1))
        if maxVal > beta:
          return maxVal
        alpha = max(alpha, maxVal)
    return maxVal
```

Figura 8: Alpha Beta Pruning

Functia *minFunc* realizeaza taietura alpha-beta. Daca valoarea minima curenta este mai mica decat alpha, atunci functia se intoarce imediat cu valoarea minima, taind astfel ramurile de explorat. Functia getAxtion returneaza actiunea care are valoarea maxima calculata de functia *PacmanOrAgent*.

```
def minFunc(self, gameState, alpha, beta, depth, agent):
    minVal = 999999
    legalActions = gameState.getLegalActions(agent)
    for action in legalActions:
        successorGameState = gameState.generateSuccessor(agent, action)
        minVal = min(minVal,self.PacmanOrAgent(successorGameState, alpha, beta, depth, agent + 1))
        if minVal < alpha:
            return minVal
        beta = min(beta, minVal)
    return minVal

def getAction(self, gameState):
    alpha = -999999
    beta = 999999
    agent = 0
    legalActions = gameState.getLegalActions(agent)
    for action in legalActions:
        successorGameState = gameState.generateSuccessor(agent, action)
        val = self.PacmanOrAgent(successorGameState, alpha, beta, 0, 1)
        if val > alpha:
            alpha = val
            best_action = action
    return best_action
```

Figura 9: Alpha Beta Pruning