UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Darius-Iulian Stan
Grupa: 30231

Facultatea de Automatica
si Calculatoare

2 Decembrie 2023

Cuprins

1 Introducere

1.1 Motivație

Ca și scop, implementarea proiectul oferă posibilitatea punerii în aplicare a cunoștințele dobândite pe parcursul semestrului din domeniul inteligenței artificiale. Mai exact, prin intermediul jocului de Pacman, implementăm diverse variante de căutare, atât pentru cazul în care doar un agent e implicat în joc (Single Agent Search), cât și pentru cazul în care trebuie să ținem cont de comportamentul altor agenți implicați în joc (Multi Agent Search).

1.2 Obiective

Presupunând că regulile și scopul jocului de Pacman sunt cunoscute, pentru a facilita implementarea diverselor concepte ce țin de cele două tipuri de căutări menționate mai sus, avem premisa că pe tabla de joc avem doar un agent (PacMan) care are o poziție de start și o poziție unde trebuie să ajungă (goal). Pentru a găsi această poziție, am implementat următorii alogirmi de căutare:

1. căutarea în adâncime (DFS)
2. căutarea în lungime (BFS)
3. căutarea pe baza unui cost (UCS)
4. căutarea pe baza unei euristici date (A* Search)
   În continuare, presupunem că putem avea mai multe poziții de pe tabla de joc unde pacman trebuie să ajungă, astfel că avem următoarele task-uri:
5. vizitarea tuturor colțurilor tăblii de joc (4 targeturi)
6. euristică pentru optimizarea căutării tuturor colțurilor
7. euristică astfel încât Pacman să ajungă viziteze o mulțime dată de poziții din labirint
8. Căutare suboptimală

În a doua parte a proiectului, ne focusăm pe situațiile în care apar și alți agenți (adversari) - fantomele. Astfel, fiind dată o listă de target-uri unde pacman trebuie să ajungă, o poziție inițială, și presupunând că fantomele sunt oponenți absoluți, avem următoarele sarcini:

1. pentru un agent reflex, care ține cont doar de datele din prezent, să construim o funcție de evaluare a acestora.
2. implementarea algoritmului MiniMax Search
3. optimizarea MiniMax Search-ului cu Alpha-Beta-prunning

1.3 Tutorial elemente de baza

O propozitie normala. Daca vrem sa adaugam text in **bold**, *italic* sau ***bold italic***.
Astfel se poate utiliza o **lista numerotata**:

2 Implementare

Pentru primele patru subcapitole, tabla de joc conține un singur agent (PacMan) și un singur target (poziția cu mâncare). Trebuie să găsim un drum de la Pacman la target, având ca și restricții faptul ca agentul poate face 4 tipuri de mișcări (Nord, Sud, Est, Vest), iar unele căsuțe pot conține un zid, astfel că agentul nu are dreptul să le viziteze.

2.1 Căutare în adâncime - DFS

2.1.1 Prezentare algoritm

Având ca și date de intrare poziția de start și poziția de finish, DFS are următoarea strategie:

- verifică dacă start-ul coincide cu finish-ul, în caz afirmativ, problema a fost deja rezolvată. Dacă nu, execută pașii următori
- path - inițializează o listă în care vom pune acțiunile, în ordinea de la start la finish, corespunzătoare drumului găsit
- explored - inițializează un set, în care trecem fiecare nod pe care l-am vizitat, astfel încât să nu-l mai vizităm
- frontier - crează o stivă în care ținem evidența nodurilor pe care trebuie să le vizităm. Punem în ea nodul de start
- cât timp frontiera nu este goală (mai avem noduri de vizitat):
- luăm nodul din vârful frontierei, îl marcăm ca și vizitat
- fiecare vecin al nodului curent este adăugat în frontieră
- se repetă pașii 5-7 până când condiția nu e adevărată. Dacă nu a fost găsit un drum, înseamnă că nu există cale de la start la finish

2.1.2 Implementare în mediul de lucru

```
def depthFirstSearch(problem: SearchProblem):
    currentState = problem.getStartState()
    path = list()
    explored = set()
    frontier = util.Stack()
    if problem.isGoalState(currentState):
        return path
    frontier.push((currentState, path))
    while not frontier.isEmpty():
        (currentState, path) = frontier.pop()
        explored.add(currentState)
        if problem.isGoalState(currentState):
            return path
        for successor, action, _ in problem.getSuccessors(currentState):
            if successor not in explored:
                frontier.push((successor, path + [action]))
    return []
```

Pentru a lua datele din joc, folosim metodele .getStartState(), care returnează poziția de start a lui Pacman, .isGoalState, care returnează True dacă poziția transmisă ca parametru este poziția de finish și .getSuccessors, care returnează o lisă de tipul (pozitie următoare, acțiune, cost).

2.2 Căutare în lungime - BFS

2.2.1 Prezentare algoritm

Având ca și date de intrare poziția de start și poziția de finish, BFS are următoarea strategie:

- verifică dacă start-ul coincide cu finish-ul, în caz afirmativ, problema a fost deja rezolvată. Dacă nu, execută pașii următori
- path - inițializează o listă în care vom pune acțiunile, în ordinea de la start la finish, corespunzătoare drumului găsit
- explored - inițializează un set, în care trecem fiecare nod pe care l-am vizitat, astfel încât să nu-l mai vizităm
- frontier - crează o coadă în care ținem evidența nodurilor pe care trebuie să le vizităm. Punem în ea nodul de start
- cât timp frontiera nu este goală (mai avem noduri de vizitat):
- luăm nodul din vârful frontierei, îl marcăm ca și vizitat
- fiecare vecin al nodului curent, dacă nu a fost deja vizitat și nu se află în frontieră, este adăugat în frontieră
- se repetă pașii 5-7 până când condiția nu e adevărată. Dacă nu a fost găsit un drum, înseamnă că nu există cale de la start la finish

Se poate observa ca bfs are aceeași strategie cu dfs, însă, diferența majoră o reprezintă faptul că frontiera este o coadă, astfel că explorăm, cu întâietate, nodurile cele mai apropiate de nodul de start. Astfel, bfs garantează că returnează cea mai scurtă distanță dintre start și finish.

2.2.2 Implementare în mediul de lucru

```
def breadthFirstSearch(problem: SearchProblem):
    currentState = problem.getStartState()
    path = list()
    explored = set()
    frontier = util.Queue()
    if problem.isGoalState(currentState):
        return path
    frontier.push((currentState, path))
    while not frontier.isEmpty():
        (currentState, path) = frontier.pop()
        explored.add(currentState)
        if problem.isGoalState(currentState):
            return path
        for successor, action, _ in problem.getSuccessors(currentState):
            if successor not in explored and successor not in [s[0] for s in frontier.list]:
                frontier.push((successor, path + [action]))
    return []
```

2.3 Căutare cu cost uniform - UCS

În contextul jocului de Pacman, când luăm o decizie (alegem o poziție nouă pentru Pacman), trebuie să ținem cont de mai mulți parametii, cum ar fi: cât de aproape sunt fantomele de poziția respectivă, dacă este sau nu aproape de target etc. Astfel, putem crea o funție care ține cont, în difierite procentaje, de mai mulți parametrii, deciziile fiind luate pe baza acestei funcții, fiind alese deciziile cu cel mai mare scor (valoare a funției). Astfel, UCS, în algoritm, este identic cu BFS, însă frontiera este o coadă de prioritate (implementată cu heap-uri), astfel că ordinea în care vizităm nodurile este dată de funcția care calculează prioritatea (scorul fiind relevat pentru calea de la start până la nodul pentru care calculăm prioritatea). Mai mult, apare cazul în care dăm peste un nod vizitat deja, însă costul de a vizita acel nod, ar putea să fie altul decât

cel din momentul în care îl vizitasem (pot să difere căile). Astfel suntem nevoiți să actualizăm costul nodului respectiv. Așadar, UCS ține cont de costul căii pe care o va returna, când BFS returnează calea cea mai scurtă.

2.3.1 Prezentare algoritm

Având ca și date de intrare poziția de start și poziția de finish, UCS are următoarea strategie:

- verifică dacă start-ul coincide cu finish-ul, în caz afirmativ, problema a fost deja rezolvată. Dacă nu, execută pașii următori
- path - inițializează o listă în care vom pune acțiunile, în ordinea de la start la finish, corespunzătoare drumului găsit
- explored - inițializează un set, în care trecem fiecare nod pe care l-am vizitat, astfel încât să nu-l mai vizităm
- frontier - crează o coadă de priorități în care ținem evidența nodurilor pe care trebuie să le vizităm. Punem în ea nodul de start
- cât timp frontiera nu este goală (mai avem noduri de vizitat):
- luăm nodul din vârful frontierei, îl marcăm ca și vizitat
- fiecare vecin al nodului curent, dacă nu a fost deja vizitat și nu se află în frontieră, este adăugat în frontieră
- altfel, verificăm dacă costul nodului care se află deja în frontieră nu diferă în momentul actual, în caz afirmativ, îl actualizăm
- se repetă pașii 5-8 până când condiția nu e adevărată. Dacă nu a fost găsit un drum, înseamnă că nu există cale de la start la finish

2.3.2 Implementare în mediul de lucru

```
def uniformCostSearch(problem: SearchProblem):
    currentState = problem.getStartState()
    path = list()
    frontier = util.PriorityQueue()
    explored = set()
    """bfs dar cu cozi de prioritate basically"""
    if problem.isGoalState(currentState):
        return path
    frontier.push((currentState,path), 0)
    while not frontier.isEmpty():
        currentState, path = frontier.pop()
        explored.add(currentState)
        if problem.isGoalState(currentState):
            return path
        statesInFrontier = [entry[2] for entry in frontier.heap]
        for successor, action, cost in problem.getSuccessors(currentState):
            successorPath = path + [action]
            newPotentialCost = problem.getCostOfActions(successorPath)
            if successor not in explored
               and successor not in [s for s, v in statesInFrontier]:
                frontier.push((successor, successorPath), newPotentialCost)
            else:
                for index, existingState in enumerate(statesInFrontier):
```

```
                    if existingState[0] == successor:
                        currentCost = frontier.heap[index][0]
                        if currentCost > newPotentialCost:
                            frontier.heap[index] = (currentCost, frontier.heap[index][1],
                                                    (successor, successorPath))
                            frontier.update((successor, successorPath), newPotentialCost)
    return []
```

2.4 Căutare cu ajutorul unei euristici- A* Search

În principiu, A* are ca strategie UCS, însă, nu se ține cont doar de costul căii de până acum, ci și de un cost aproximativ al drumului rămas de parcurs până la target dacă va fi aleasă acțiunea respectivă, introducându-se, astfel, conceptul de euristică.

2.4.1 Implementare în mediul de lucru

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    currentState = problem.getStartState()
    path = list()
    frontier = util.PriorityQueue()
    explored = set()
    if problem.isGoalState(currentState):
        return path
    frontier.push((currentState, path), heuristic(currentState, problem))
    while not frontier.isEmpty():
        currentState, path = frontier.pop()
        explored.add(currentState)
        if problem.isGoalState(currentState):
            return path
        statesInFrontier = [entry[2] for entry in frontier.heap]
        for successor, action, cost in problem.getSuccessors(currentState):
            successorPath = path + [action]
            newPotentialCost = problem.getCostOfActions(successorPath) +
                               heuristic(successor, problem)
            if successor not in explored and
               successor not in [s for s, v in statesInFrontier]:
                frontier.push((successor, successorPath), newPotentialCost)
            else:
                for index, existingState in enumerate(statesInFrontier):
                    if existingState[0] == successor:
                        currentCost = frontier.heap[index][0]
                        if currentCost > newPotentialCost:
                            frontier.heap[index] = (currentCost, frontier.heap[index][1],
                                                    (successor, successorPath))
                            frontier.update((successor, successorPath), newPotentialCost)
    return []
```

Astfel, se observă în implementare că singurul lucru care diferă la A* față de UCS este faptul că, în frontieră (coadă de priorități), costul cu care este marcat nodul este suma dintre costul

drumului de la start până la nodul curent (path) și valoarea calculată de euristica aleasă pentru restul drumului până la target.

2.5 Găsirea tuturor colțurilor

Dată fiind poziția pe tabla de joc a lui PacMan, trebuie să găsim un drum astfel încât să fie vizitate toate cele patru colțuri ale tablei de joc.

În acest task, trebuie să creăm metode pentru clasa CornersProblem (care extinde clasa SearchProblem, clasă a cărei instante o primeam ca parametru în problemele de căutare anterioare).

Astfel, avem următoarele metode:

- **getStartState** - returnează starea de început a agentului nostru. Reținem doar informațiile strict necesare, și anume: poziția din labirint din care pornim (self.startingPoing) - o tupă de forma (X, Y), o tuplă cu patru valori Bool, fiecare valoare de pe poziția I marcând dacă colțul de pe poziția I a fost vizitat sau nu. La început,bineînțeles, niciun colț nu a fost vizitat, deci toate falorile sunt False.

```
def getStartState(self):
        visited = [False, False, False, False]
        startState = (self.startingPosition, tuple(visited))
        return startState
```

- **isGoalState** - returnează True sau False, în funcție dacă toate colțurile au fost vizitate sau nu.

```
    def isGoalState(self, state: Any):
        _, visited = state
        if False in visited:
            return False
        return True
```

- **getSuccessors** - Returnează toți succesorii succesorii stării primite ca și parametru, creând o listă de tipul valorii retunate la getStartState (tuplă de două tuple, prima cu poziția în labirint și a doua cu cele patru valori pentru colțurile labirintului (actualizând-o dacă e cazul)

```
    def getSuccessors(self, state: Any):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
            x, y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                succesorCornersState = list(state[1])
                if (nextx, nexty) in self.corners:
                    succesorCornersState[self.corners.index((nextx, nexty))]=True
                successorState = ((nextx, nexty), tuple(succesorCornersState))
                successors += [(successorState, action, 1)]

```

```
15
16          self._expanded += 1 # DO NOT CHANGE
17          return successors
```

2.6 Euristică pentru vizitarea celor patru colțuri

Putem aplica orice algoritm de căutare implementat în secțiunile 2.1 - 2.4 pentru problema celor patru colțuri, date fiind metodele enumerate mai sus. Însă, pentru a aplica A* Search, mai avem nevoie de o funție Euristică, care să estimeze costul de la nodul pe care îl putem alege în momentul curent, până la target (target-ul fiind, de fapt, vizitarea tuturor celor patru poziții). Astfel, ca și euristică, am decis ca pentru fiecare nod, costul estimat să fie maximul dintre distanțele de la el până la colțurile ce au rămas de vizitat.

```
1  def cornersHeuristic(state: Any, problem: CornersProblem):
2
3      corners = problem.corners
4      walls = problem.walls
5
6      if problem.isGoalState(state):
7          return 0
8      maxdist = max([util.manhattanDistance(state[0], corner)
9                     for index, corner in enumerate(problem.corners)
10                    if not list(state[1])[index]])
11     return maxdist
```

2.7 Euristică pentru Vizitarea tuturor nodurilor dintr-o mulțime dată

Dată fiind o listă de poziții pe care dorim să le viziteze Pacman, implementăm, pentru a putea folosi algoritmul A* Search: **foodHeuristic** - euristica pentru aproximarea costului drumului rămas până la atingerea obiectivului. Dată fiind poziția la care ne aflăm și lista de poziții ce trebuie vizitate (position, respectiv food), luăm cea mai mare distanță manhatten (suma diferenței dintre coordonatele X ale celor două poziții și ce a coordonatelor Y) dintre două poztii ce trebuie vizitate. Mai apoi, adăugăm la ea și minimul dintre distanțele de la poziția curentă la cele două poziții pentru care am găsit rezultatul anterior.

```
1  def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
2      if problem.isGoalState(state):
3          return 0
4
5      position, foodGrid = state
6      food = foodGrid.asList()
7      maxDistance = 0
8
9      first = food[0]
10     second = food[0]
11     for index, food1 in enumerate(food):
12         for food2 in (food[index:]):
13             dist = util.manhattanDistance(food1, food2)
14             if dist > maxDistance:
15                 maxDistance = dist
```

```
                first = food1
                second = food2

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

Căutare suboptimală pentru vizitarea tuturor pozițiilor dintr-o mulțime dată

Pentru cazul în care chiar dacă euristica aleasă este una consistentă și admisibilă, datorită numărului mare de poziții pe care trebuie să le vizităm, calcularea pentru fiecare potențial nod a costului până la target poate încetini procesul de căutare propriu-zis. Spre exemplu, în secțiunea anterioară, dacă folosim euristica implementată, pentru n poziții de vizitat trebuie să calculăm (n-1)(n-2)/2 + 2 distanțe. Pentru mulțimi cu număr mare de poziții, euristica nu este rentabilă. În acest caz, intervine căutarea suboptimală. Ea are următorul algoritm:

- Cât timp nu s-au vizitat toate pozițiile din mulțimea dată, execută
- aplică bfs din punctul curent până la orice poziție din mulțime.
- când a fost găsită o poziție din mulțime, o vom vizita, parcurgând calea returnată de bfs.

2.8 Agentul Reflex

Când apar în joc și fantomele, automat PacMan trebuie să țină cont și de poziția acestora, astfel, pentru Agentul Reflex, care ține cont doar de pozițiile fantomelor și target-urile unde trebuie să ajungă, am implementat următoarea funcție de evaluare:

```
    def evaluationFunction(self, currentGameState: GameState, action):
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostPositions = successorGameState.getGhostPositions()
        distmin = -1
        for food in newFood.asList():
            dist = util.manhattanDistance(newPos, food)
            if distmin > dist or distmin==-1:
                distmin = dist

       distGhosts = list()
       for ghostPos in newGhostPositions:
            dist = util.manhattanDistance(ghostPos, newPos)
            distGhosts.append(dist)

        if 0 in distGhosts or 1 in distGhosts:
            return -1 * float("inf")

        return successorGameState.getScore() +
               (10 / float(distmin)) - (1 / float(sum(distGhosts)))
```

Calculăm, pentru poziția newPos a lui Pacman, cea mai mică distanță dintre el și un target (food). De asemeni, calculăm și distanțele dintre Pacman și fiecare fantomă, suma acestor distanțe fiind relevantă în calcularea scorului final. Dacă Pacman este lângă o fantomă (distanța este 1) sau dacă l-a prins o fantomă (distanța e 0), atunci scorul va fi cel mai mic posibil (minus infimit). În caz contrar, Folosind inversul lui distMin și inversul sumei distanțelor distGhost,

returnăm, ca și scor, diferența dintre acestea. Dat fiind că cu cât distanța e mai mică, cu atât dorim ca scorul să fie mai mare, e optim să folosim inversul acesteia, mai mult, pentru că are o relevanță mai mare decât distanța până la fantome, inversul lui distMin are coeficientul 10. Cât despre distGhost, cu cât e mai mare, cu atât dorim ca scorul să fie mai mic, așadar folosim inversul acesteia. Folosim operația de scădere tocmai pentru a marca faptul că termenul generat de distMin are un impact pozitiv asupra scorului, iar termenul generat de distGhost unul negativ.

2.9 Agentul MiniMax

2.9.1 Descrierea algoritmului

Presupunând că avem un adversar (sau mai mulți) perfecți, dorim să le putem anticipa mișcările. Astfel MiniMax presupune că există un scor bine definit pentru fiecare mutare, și că agentul nostru dorește maximizarea scorului total al jocului, iar restul agenților minimizarea lui. Algoritmul are, de asemeni, o limită maximă de analizare a potențialelor decizii. Mai exact, dacă limita este 3, atunci poate să decidă ce alegere să facă agentul nostru strict pe baza următoarelor trei mutări ale tuturor fantomelor.

Astfel, el are următoarea strategie:

- cât timp nu s-a atins limita maximă, execută:
- Pentru fiecare mutare posibilă, calculează scorul obținut din anticiparea următoarelor mișcări
- Dacă agentul curent este agentul nostru, selectează maximul dintre scoruri
- Dacă agentul este altul, selectează scorul minim
- Trimite scorul obținut Mutării anterioare, care are nevoie de el
- Crește adâncimea când Agentul nostru cere o altă iterație

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState: GameState):

        def maxLevel(gameState, depth):
            if gameState.isWin() or gameState.isLose() or depth == self.depth - 1:
                return self.evaluationFunction(gameState)
            maxvalue = -1 * (float("inf"))
            for action in gameState.getLegalActions(0):
                successor = gameState.generateSuccessor(0, action)
                maxvalue = max(maxvalue, minLevel(successor, depth + 1, 1))
            return maxvalue

        def minLevel(gameState, depth, index):
            if gameState.isWin() or gameState.isLose():
                return self.evaluationFunction(gameState)
            minvalue = float("inf")
            for action in gameState.getLegalActions(index):
                successor = gameState.generateSuccessor(index, action)
                if index == gameState.getNumAgents() - 1:
                    minvalue = min(minvalue, maxLevel(successor, depth))
                else:
```

```
                minvalue = min(minvalue, minLevel(successor, depth, index + 1))
        return minvalue

    bestscore = -1 * float("inf")
    bestOption = ''
    for action in gameState.getLegalActions(0):
        score = minLevel(gameState.generateSuccessor(0, action), 0, 1)
        if score > bestscore:
            bestscore = score
            bestOption = action
    return bestOption
```

2.10 Agentul MiniMax- Alpha-Beta-Prunning

Algoritmul este bazat pe MiniMax, însă aduce o îmbunătățire mai ales situațiilor în care putem alege dintr-o mulțime mai mare de mutări (ex: șah). Mai exact, Avem două variabile, alfa și beta, care, inițial, vor avea valoarea minus infinit, respectiv infinit. În alpha, vom reține cel mai mare scor obținut până acum, din mutările pe care le-am verificat deja, iar beta cel mai mic. Aceste două valori folosesc la eliminarea mutărilor care știm deja că nu o să aibe un efect asupra scorului final al stării în care ne aflăm.

Mai precis, dacă verificăm pentru mutarea m1, care dorește scorul maxim, având mutarea m2, care dorește scorul minim, pentru potențialele mutări m21, m22 ... m2n, nu le vom explora pe cele care au, cel puțin, un scor mai mic decât alpha de la m1, întrucât la m2 vom alege minimul, deci acest minim va fi mai mic sau egal cu scorul găsit (mai mic decat alfa), iar m1 va lua scorul maxim, care e mai mare sau egal decat alfa. Prin deducție, scor final m1 ¿= alfa ¿ scor găsit la o mutare a succesoare lui m2. Așadar, nu are sens să mai verificăm alte mutări succesoare lui m2, întrucât scorul final al lui m2 va fi mai mic sau egal cu scorul deja gasi, care oricum e mai mic decât alfa. Analogic funcíonează și optimizarea cu ajutorul parametrului beta.

```
    def maxLevel(gameState, depth, alpha, beta):
        if gameState.isWin() or gameState.isLose() or depth == self.depth - 1:
            return self.evaluationFunction(gameState)
        maxvalue = -1 * (float("inf"))
        for action in gameState.getLegalActions(0):
            successor = gameState.generateSuccessor(0, action)
            maxvalue = max(maxvalue, minLevel(successor, depth + 1, 1, alpha, beta))
            if maxvalue > beta:
                return maxvalue
            alpha = max(maxvalue, alpha)
        return maxvalue

    def minLevel(gameState, depth, index, alpha, beta):
        if gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)
        minvalue = float("inf")
        for action in gameState.getLegalActions(index):
            successor = gameState.generateSuccessor(index, action)
            if index == gameState.getNumAgents() - 1:
                minvalue = min(minvalue, maxLevel(successor, depth, alpha, beta))
```

```
                else:
                    minvalue = min(minvalue, minLevel(successor, depth, index + 1,
                                   alpha, beta))
                beta = min(minvalue, beta)
                if minvalue < alpha:
                    return minvalue
            return minvalue

        bestscore = -1 * float("inf")
        bestOption = ''
        alpha = -1 * float("inf")
        beta = float("inf")
        for action in gameState.getLegalActions(0):
            score = minLevel(gameState.generateSuccessor(0, action), 0, 1, alpha, beta)
            alpha = max(alpha, score)
            if score > bestscore:
                bestscore = score
                bestOption = action
        return bestOption
```