UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Istrate Cristian Andrei si Ivan Alexandru Sofron
Grupa: 30237

Facultatea de Automatica
si Calculatoare

04 Decembrie 2023

Cuprins

1 Uninformed search

1.1 Question 1 - Depth-first search

Ca prima cerinta am avut de realizat implementarea algoritmului DFS (Depth First Search) ca mod de gandire al jocului PAC-MAN pentru a-si gasi mancarea din labirint si a incheia jocul, aplicand algoritmul. DFS este un algoritm de tipul LIFO, adica foloseste o stiva. Este un algoritm pentru parcurgerea sau cautarea intr-o structura de date de tip arbore sau graf. Se incepe de la radacina și se exploreaza cat mai mult posibil de-a lungul fiecarei ramuri inainte de a face pasi inapoi.

```
if problem.isGoalState(problem.getStartState()):
        return []
    visited = set()
    coord = Stack()
    directions = []
    coord.push((problem.getStartState(), []))
    while not coord.isEmpty():
        currTup = coord.pop()
        current = currTup[0]
        path = currTup[1]
        if current not in visited:
            visited.add(current)
            if problem.isGoalState(current):
                directions = path
                break
            successors = problem.getSuccessors(current)
            for succ_coord, succdirections,  in successors:
                coord.push((succ_coord, path + [succ_directions]))
    return directions
```

Am inceput prin a importa clasele necesare relizarii algoritmului, si anume clasa Stack din pachetul util. Ne-am uitat peste pachetul util ca sa vedem ce contine. Ne-am uitat peste algoritmii implementati pe parcursul laboratului, am verificat tot ce poate returna problema de tipul Search ca sa intelegem cu ce avem de lucrat, am inceput sa adaptam pentru codul nostru cum sa realizam parcurgerile. Ne-am confruntat cu urmatoarele probleme: nu stiam ce sa punem in lista de directii, nu stiam ce sa punem in coada, ne-am dat seama ca trebuie sa facem niste path-uri provizorii pentru a ajunge la solutia finala, am desenat mai multe parcurgeri ca sa intelegem mai bine modul de parcurgere al algoritmului. Printre primele rulari pac-manul facea un pas inapoi si ajungea intr-o pozitie interzisa, astfel programul se oprea din compilare si rezulta un cost urias.

1.2 Question 2 - Breadth-first search

Pentru a doua cerinta am avut de realizat implementarea algoritmului BFS (Breadth-first search) tot pentru aceleasi obiective ca si DFS. Cautarea (parcurgerea) in latime este un algoritm pentru parcurgerea sau cautarea intr-o structura de date de tip arbore sau graf. Aceasta incepe cu radacina arborelui si exploreaza nodurile, mai intai nodurile vecine acestuia, inainte de a trece la vecinii de pe nivelul urmator

In ceea ce priveste codul, singura diferenta fata de DFS este folosirea unei cozi in locul stivei.

```
if problem.isGoalState(problem.getStartState()):
        return []
    visited = set()
    coord = Queue()
    directions = []
    coord.push((problem.getStartState(), []))
    while not coord.isEmpty():
        currTup = coord.pop()
        current = currTup[0]
        path = currTup[1]
        if current not in visited:
            visited.add(current)
            if problem.isGoalState(current):
                directions = path
                break
            successors = problem.getSuccessors(current)
            for succ_coord, succdirections,  in successors:
                coord.push((succ_coord, path + [succ_directions]))
    return directions
```

Realizarea BFS-ului nu a necesitat multa munca deoarece implementarea a fost aprope identica cu cea a DFS-ului iar problemele nu au avut de unde sa apara fiind rezolvate deja de la pasul anterior.

1.3 Question 3 - Uniform Costs Search

Pentru a treia cerinta am avut de realizat implementarea algoritmului UCS (Uniform cost search), aceasta tinand seama de costul deplasarilor prin labirint. Algoritmul UCS se ocupa de gasirea drumului cu valoare minima spre nodul destinatie. Foloseste o abordare asemanatoare cu algoritmul Greedy.

Modul de realizare al acestui algoritm este diferit fata de restul implementarilor realizate in momentul de fata pentru ca folosim o structura de date care se modifica in timp ce lucram cu ea si anume **PriorityQueue**, care functioneaza ca o coada dar elementul care este scos prima data din coada este cel cu prioritatea cea mai mare, elementele adaugate putand avea o prioritate mult mai mare fata de cele existente, astfel fiind plasate la inceputul cozii. Prioritatea este stabilita in functie de costul fiecarei actiuni.

```
if problem.isGoalState(problem.getStartState()):
        return []
    visited = set()
    coord = PriorityQueue()
    directions = []
    coord.push((problem.getStartState(), []), 0)
    while not coord.isEmpty():
        currTup = coord.pop()
        current = currTup[0]
        path = currTup[1]
        if current not in visited:
            visited.add(current)
```

```
            if problem.isGoalState(current):
                directions = path
                break
            successors = problem.getSuccessors(current)
            for succ_coord, succdirections,  in successors:
                coord.push((succ_coord, path + [succ_directions]), problem.getCostOfActions(
    return directions
```

Ne-am uitat in util.py ca sa intelegem cum functioneaza clasa de Priority queue, am vazut ca are nevoie de un cost si ca elementele extrase cu functia pop se iau in ordinea costului. Ca sa stabilim costul fiecarui path provizoriu am folosit functia getCostOfActions pentru fiecare path iar la final am returnat path-ul cu cel mai mic cost pe care l-am gasit. Am analizat si clasele StayWest si StayEast pentru a intelege cum se calculeaza costurile in functie de problema fiecarui labirint. Probleme: am crezut ca trebuie sa implementam o functie care sa calculeze costul fiecarei actiuni, mai apoi am gasit in cod functia getCostOfAction gata implementata.

2 Informed search

2.1 Question 4 - A* search algorithm

In continuare a trebuit sa ne ocupam de algoritmul A* (A star). Acesta este tot un algoritm de parcurgere in functie de cost dar ca element nou ne apare euristica.

$$f(n) = g(n) + h(n)$$

Figura 1: Formula algoritmului

In formula algoritmului (Fig. 1), f este costul estimat pana la destinatie, h reprezinta euristica (estimarea costului pana la final, fara sa tina cont de anumite lucruri) iar g costul pana la un anumit punct.

Pentru proiectul nostru am folosit doua tipuri de euristici (Fig. 2). Euristica euclidiana calculeaza distanta cea mai rapida pana la destinatie fara a tine cont de limitarile de miscare a lui PAC-MAN (South, North, West, East). Euristica Manhattan tine cont de miscarile posibile ale lui PAC-MAN, dar nu tine cont de aparitia zidurilor in drumul lui, si am considerat ca este cea mai buna varianta pentru algoritmul nostru.

```
if problem.isGoalState(problem.getStartState()):
        return []
    visited = set()
    coord = PriorityQueue()
    directions = []
    coord.push((problem.getStartState(), []), 0)
    while not coord.isEmpty():
        currTup = coord.pop()
        current = currTup[0]
        path = currTup[1]
        if current not in visited:
```

```
            visited.add(current)
            if problem.isGoalState(current):
                directions = path
                break
            successors = problem.getSuccessors(current)
            for succ_coord, succdirections,  in successors:
                coord.push((succ_coord, path + [succ_directions]), problem.getCostOfActions(
                        + heuristic(succ_coord, problem))
    return directions
```

Pentru algoritmul A star am adaptat algoritmul UCS prin calcularea costului drumului solutiilor nu doar in functie de pasii pe care ii facem dar si in fucntie de o anumita euristica pe care am specificat-o la apelarea functiei. Asadar codul nu se modifica drastic, dar algoritmul are o solutie mai buna, adica nodurile pe care le expandeaza sunt mai putine, fata de o cautare UCS prin faptul ca ne asteptam deja la o aproximare a distantei parcurse, asadar algoritmul nu cauta doar o solutie rapida, ci si una optima in raport cu nodurile pe care le expandeaza.

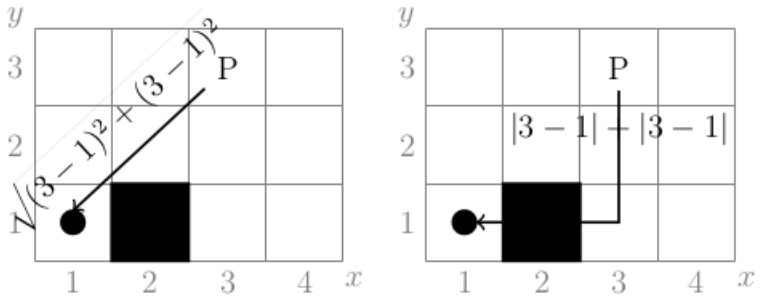

Figura 2: Euristica euclidiana si Manhattan

2.2 Question 5 - Finding all the corners

Pentru urmatoarea cerinta am avut de implementat un mod de gandire prin care PAC-MAN sa gaseasca toate colturile labirintului intr-un mod cat mai eficient, folosind BFS-ul.

In ceea ce priveste codul, am avut de lucrat in 4 functii, cea de initiere, getStartState, isGoalState si getSuccessors. La inceputul jocului pac man se afla in pozitia de start si am construit o lista cu 4 elemente False, semnificand fiecare colt al labirintului. Pentru a ne folosi de state si de lista in care se gasesc colturile asemeni unui dictionar a trebuit sa convertim colturile la o tupla, pentru a vedea statusul fiecarui colt in fiecare pozitie a labirintului in care ne aflam, aceasta fiind singura metoda prin care poti accesa variabila instanta a clasei corner problem pe care am definit-o. In goal state verificam daca in lista colturilor sunt toate elementele adevarate, iar in succesors folosim acelasi model din PositionSearchProblem, modificand doar starea listei de colturi in momentul in care pac-man a ajuns la unul dintre colturi.

Codul adaugat in initiere.

```
self.cornersFound = [False, False, False, False]
```

Codul pentru returnarea starii de start, sub forma de tupla.

```
return (self.startingPosition, tuple(self.cornersFound))
```

Verificarea starii de final.

```
for corner in state[1]:
            if not corner:
                return False
        return True
```

Metoda pentru obtinerea succesorilor si actualizarea colturilor.

```
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)
            if not self.walls[nextx][nexty]:
                nextState = ((nextx, nexty), list(state[1]))
                for i, corner in enumerate(self.corners):
                    if nextState[0] == corner:
                        nextState[1][i] = True
                cost = 1
                nextState = ((nextx, nexty), tuple(nextState[1]))
                successors.append((nextState, action, cost))
        self._expanded += 1 # DO NOT CHANGE
        return successors
```

Pentru rezolvarea unei bune parti din cod ne-am inspirat din clasa PositionSearchProblem care foloseste acelasi model dar una dintre cele mai mari probleme intalnite a fost *TypeError: unhashable type: "list"* care ne impiedica sa trimitem colturile sub forma de lista starii curente in care ne aflam pentru a putea fi accesata de fiecare succesor in parte. Problema fiind rezolvata prin cautarea erorii pe internet.

2.3 Question 6 - Corner problem: Heuristic

S-a cerut rezolvarea problemei colturilor intr-un numar mai mic de pasi prin implementarea unei euristici. Noi stim ca PAC-MAN-ul trebuie sa ajunga de la un colt la altul, ceea ce avem noi de facut este sa estimam aceste drumuri pe care le are de parcurs. Ca o euristica sa fie valida aceasta trebuie sa fie consistenta si admisibila, astfel noi trebuie sa ne gandim la distanta pe care o are de parcurs pe termen lung.

La inceput am incercat sa impartim labirintul in 4 sectiuni iar in functie de sectiunea in care se afla pac-man, acesta sa porneasca calculul in functie de cel mai apropiat colt iar apoi sa continue inspre celelalte colturi. Programul nu functiona deoarece euristica nu era consistenta. Am printat stateurile in care se gasea pac-man ca sa vedem pasii pe care ii face, astfel ne-am dat seama ca putem stabili euristica pentru fiecare colt in parte fara sa conteze din ce sectiune incepe. Pentru solutia finala am folosit functia manhattanDistance din util.py pentru fiecare colt al labirintului. Am stocat toate distantele intr-o lista si ca valoare returnata pentru euristica am ales maximul dintre distantele pentru fiecare colt nedescoperit.

```
if problem.isGoalState(state):
        return 0
    else:
```

```
        distances = []
        for i, corner in enumerate(state[1]):
            if not corner:
                distances.append(manhattanDistance(state[0], corners[i]))
        return max(distances)
```

2.4 Question 7 - Eating All The Dots

Asemenea cerintei anterioare trebuie sa implementam din nou o euristica dar care il va ajuta pe PAC-MAN sa manance toate bucatile de mancare intr-un mod eficient.

Pentru rezolvarea problemei am aplicat acelasi mod de gandire ca la Q6. La inceput am afisat food grid-ul pentru a ne da seama cum este constuit labirintul, dupa care am parcurs labirintul pe linii si coloane pentru a afla daca este bucata de mancare pe pozitia respectiva. La fel ca la Q6 am folosit distanta Manhattan pentru a calcula numarul de pasi pana la fiecare bucata de mancare nedescoperita, iar valoarea euristicii returnate fiind cea mai indepartata bucata de mancare pe care PAC-MAN trebuie sa o manance.

Rezolvarea a fost buna, dar nu suficient de eficienta, astfel ne-am propus sa rezolvam problema zidurilor din euristica Manhattan, si anume verificam daca exista zid intre pozitia lui PAC-MAN si pozitia mancarii, iar daca era cazul adaugam faptul ca zidul trebuie ocolit prin incrementarea euristicii.

```
def manhattanDistanceWithWallSolved(xy1, xy2, walls):
        wallsFound = 0
        x1 = xy1[0]
        y1 = xy1[1]
        x2 = xy2[0]
        y2 = xy2[1]
        if x1 > x2:
            x1, x2 = x2, x1
        if y1 > y2:
            y1, y2 = y2, y1
        for i in range(x1, x2):
            if walls[i][y1]:
                wallsFound+=1
        for i in range(y1, y2):
            if walls[x1][i]:
                wallsFound+=1
        return (abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1]) + wallsFound)
    if problem.isGoalState(state):
        return 0
    else:
        distances = []
        for i,l in enumerate(foodGrid):
            for j,c in enumerate(l):
                if c:
                    distances.append(manhattanDistanceWithWallSolved(position, (i, j),proble
        return max(distances)
```

2.5 Question 8 - Suboptimal Search

Pentru ultima cerinta a trebuit sa gasim drumul catre cea mai apropiata mancare, iar pentru aceasta cerinta nu am avut de implementat decat stabilirea faptului ca am ajuns la o mancare si folosirea unui algoritm de cautare adecvat.

Pentru rezolvarea problemei am printat gridul cu mancare pentru a vedea cum arata labirintul. Am verificat la goal daca in pozitia in care se afla pacman se gaseste mancare si am completat functia findPathToClosestDot apeland bfs-ul pe problema noastra.

3 Adversarial search

3.1 Question 9 - Improve the ReflexAgent

In continuarea proiectului avem o noua problema de abordat, si anume cea a castigarii jocului fara a muri din cauza fantomelor. Astfel, avem de implementat o functie care ne indica daca actiunea PAC-MAN-ului este una buna sau nu. Noi trebuie sa adunam mancarea, pentru a castiga jocul, in timp ce ne ferim de fantome. Siind aceste lucruri avem o variabila care ne indica o valoare numerica in raport cu actiunea pe care dorim sa o facem aceasta fiind direct proportionala cu drumul spre succes a lui PAC-MAN *(valoare mare = castig sigur)*

```
score = successorGameState.getScore()
        curPos = currentGameState.getPacmanPosition()
        curFood = currentGameState.getFood()
        newGhostPos = successorGameState.getGhostPositions()
        curGhostPos = currentGameState.getGhostPositions()
        if curFood[newPos[0]][newPos[1]]:
            score += 1
        else:
            minFood = min(manhattanDistance(newPos, food) for food in newFood.asList())
            score += 1.00 / minFood
        for ghost in newGhostPos:
            if manhattanDistance(newPos, ghost) <= 1:
                score -= 10
        return score
```

Pentru rezolvarea problemei am printat toate variabilele care erau scrise la inceputul functiei pentru a intelege de ce trebuie sa ne folosim. Am verificat toate cazurile de rulare, ce se poate intampla cu pacman. Dupa ce am inteles problema am incercat sa il facem pe pacman sa verifice daca este mancare la urmatorul pas sau daca este fantoma. Am reusit sa verificam daca este mancare pe urmatoare pozitie deoarece mancarea este statica, dar fantomele fiind mereu in miscare a trebuit sa ne gandim la o alta abordare. Nu ne-am mai legat de pozitia actuala si urmatoare a fantomei, ci de distanta lui pacman fata de fantoma, folosind manhattanDistance din util. Am verificat distanta dintre PAC-MAN si fantome, iar la o anumita valoare scadeam scorul pe baza caruia se misca pacman. Apoi a trebuit sa rezolvam gasirea mancarii care nu este pe o pozitie apropiata de noi, asadar am folosit tot manhattanDistance, luand drumul minim, cel mai apropiat de mancare. In functie de urmatoarea actiune stabilim daca este bine sau nu directia in care ne deplasam pentru a gasi mancare.

3.2 Question 10 - Minimax

Pentru aceasta cerinta avem de realizat agentul Minimax care returneaza o actiune folosind algoritmul minmax. Minimax este o regula de decizie care consta in maximizarea castigului minim posibil. Acest algoritm se aplica in momentul in care avem si un inamic care de asemenea cauta sa faca cele mai bune mutari. Conceptul algoritmului consta in 3 functii (vezi Fig. 3).

```
function MINIMAX–DECISION (state) returns an action
   return arg max_{a∈ACTIONS(state)} MIN–VALUE(RESULT(state, a))

function MAX–VALUE(state) returns a utility value
 if TERMINAL–TEST(state) then return UTILITY(state)
 v ← −∞
 for each a in ACTIONS(state) do
   v ← MAX(v, MIN–VALUE(RESULT(state, a)))
return v

function MIN–VALUE(state) returns a utility value
 if TERMINAL–TEST(state) then return UTILITY(state)
 v ← ∞
 for each a in ACTIONS(state) do
   v ← MIN(v,MAX–VALUE(RESULT(state, a)))
return v
```

Figura 3: Pseudocodul algoritmului Minimax

Pentru implementarea acestui algoritm am inceput realizarea celor 3 functii. Prima functie, functia de decizie (minimax), stabileste daca trebuie sa extragem o valoare maxima sau o valoare minima in functie de agentul care o apeleaza (PAC-MAN sau fantome). Aceasta functie in final returnand rezultatul algoritmuli care consta in decizia cea mai buna a agentului. In minimax apelam functiile max_val si min_val.

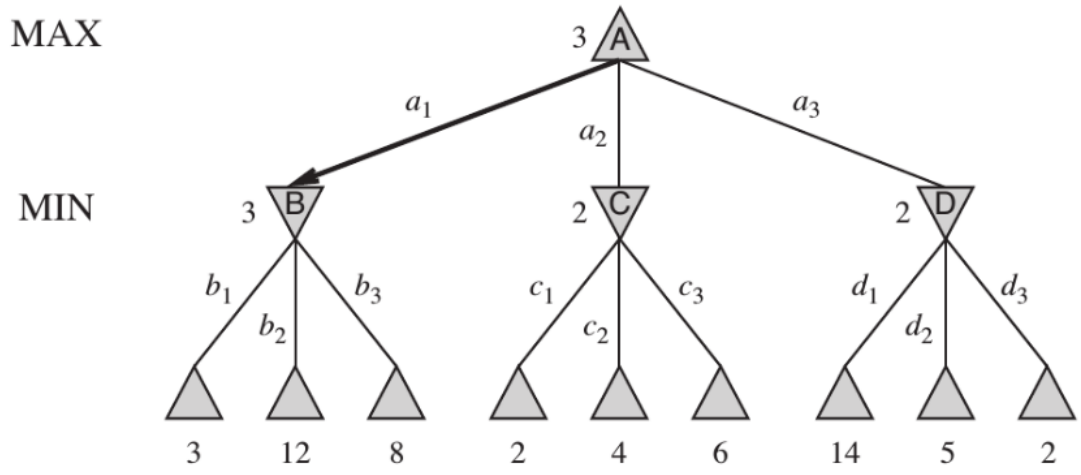

Figura 4: Exemplu arbore Minimax

Functia max_val este functia care tine de PAC-MAN. Ea se ocupa cu stabilirea valorii maxime dintre minimele selectate (adica partea de sus a arborelui). Pe aceasta functie intra doar PAC-MAN-ul de cate ori este nevoie in functie de depht-ul stabilit. Aceasta parcurge actiunile

PAC-MAN-ului, iar pentru fiecare actiune genereaza recursiv cazurile celorlalti agenti, in final returnand valoare maxima dintre toti agentii

Functia min_val este functia care tine de fantome. In aceasta, asemanator functiei max_val luam toate actiunile posibile fiecarei fantome, actiuni in urma carora trebuie sa generam si urmatoarele actiuni posibile si celorlalte fantome, rezultatul final fiind minimul dintre un numar de combinatii posibile al tuturor mutarilor fantomelor. In termen scurt, la fiecare mutare a unei fantome se genereaza si celelalte mutari posibile ale celorlalte fantome (daca este cazul) care si ele fac acelasi lucru recursiv. In final am parcurs un nivel atunci cand ne-am ocuapt de toti agentii, urmatoare mutare revenindu-i lui PAC-MAN.

```
def minimax(gameState, agentIndex ,depth):
            if gameState.isWin() or gameState.isLose() or depth == 0:
                return self.evaluationFunction(gameState)
            if agentIndex == 0:
                return max_val(gameState, depth)
            else:
                return min_val(gameState, agentIndex, depth)

        def max_val(gameState, depth):
            maxVal = float('-inf')
            for action in gameState.getLegalActions(0):
                nextMove = gameState.generateSuccessor(0, action)
                maxVal = max(maxVal, minimax(nextMove, 1, depth))
            return maxVal

        def min_val(gameState, agentIndex, depth):
            if agentIndex == gameState.getNumAgents():
                return minimax(gameState, 0, depth - 1)
            minVal = float('inf')
            for action in gameState.getLegalActions(agentIndex):
                nextMove = gameState.generateSuccessor(agentIndex, action)
                minVal = min(minVal, minimax(nextMove, (agentIndex + 1), depth))
            return minVal


        maxVal = float('-inf')
        resAction = gameState.getLegalActions(0)[0]
        for action in gameState.getLegalActions(0):
            nextMove = gameState.generateSuccessor(0, action)
            val = minimax(nextMove, 1, self.depth)
            if val > maxVal:
                resAction = action
                maxVal = val
        return resAction
```

Ceea ce facem noi este sa generam un arbore minmax pe mai multe nivele datorita numarului de fantome ridicat si nu doar una(doi agenti, PAC-MAN si fantoma), cum este prezentat in materialul de laborator. Algoritmul fiind recursiv este destul de greu de urmarit, dar ne putem

imagina ca fiecare ramura b1 b2 b3 este constituita la randul ei din mai multe ramuri dar care se afla pe acelasi nivel.

In final noi avem valoare mutarii pe care PAC-MAN trebuie sa o faca dar nu cunoastem tipul actiunii pe care acesta trebuie sa o faca. Astfel primul pas al algoritmului minimax il facem in afara functiilor pentru a retine denumirea actiunii pe care trebuie sa o faca.

3.3 Question 11 - Alpha-Beta Pruning

In urma implementarii algoritmului Minimax am constatat ca putem optimiza acest algoritm prin adaugarea a doua variabile alpha si beta, alpha fiind cea mai buna alegere la momentul actual pentru valoarea maxima iar beta fiind cea mai buna alegere la momentul actual pentru valoarea minima.

Cu ajutorul acestor variabile putem scurta pasii algoritmului nefiind nevoie sa mai cautam dupa valori care stim sigur ca nu pot lua locul celor deja existente. Algorimtul este unul partial identic cu cel al Minimax-ului, doar cu completarea pseudocodului din Fig. 5

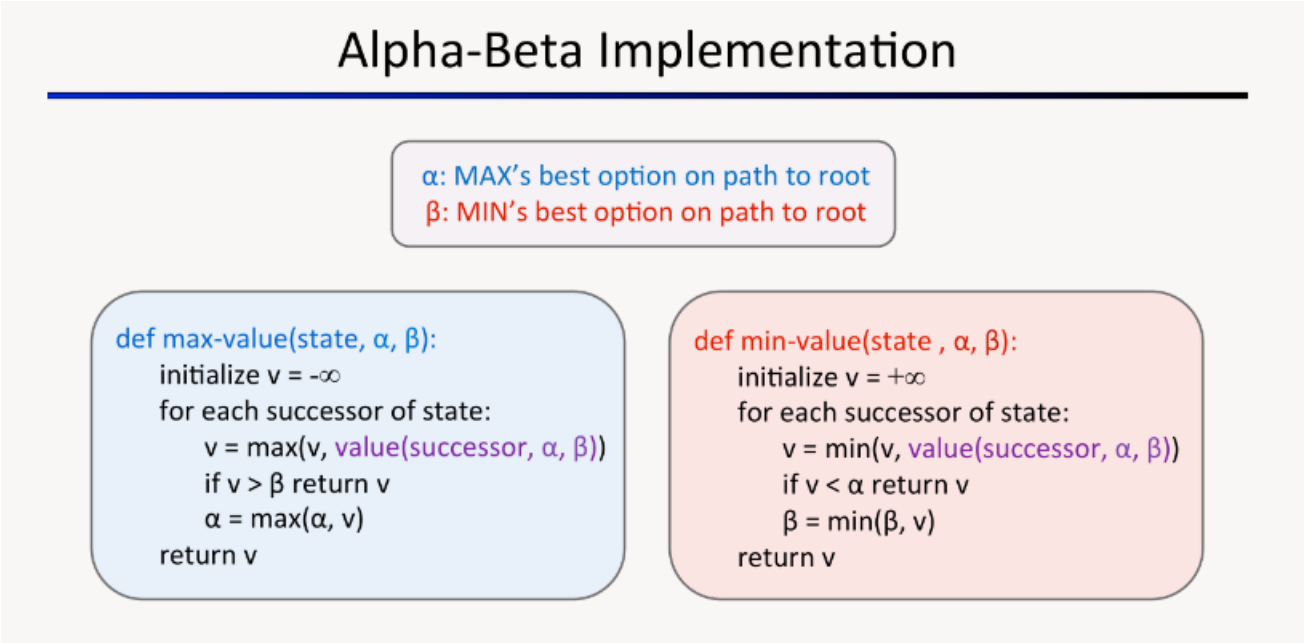

Figura 5: Pseudocod algoritm Alpha-Beta Pruning