Artificial intelligence - Project 1
- Search problems -

Gavra Anamaria

03/11/2021

1 Uninformed search

1.1 Question 1 - Depth-first search

In this section the solution for the following problem will be presented:

*"In search.py, implement **Depth-First search(DFS) algorithm** in function depthFirstSearch. Don't forget that DFS graph search is graph-search with the frontier as a LIFO queue(Stack)."*.

1.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def depthFirstSearch(problem):

    expanded = []
    stack = util.Stack()
    stack.push((problem.getStartState(), [])) #adaugam pe lista prima stare

    while not stack.isEmpty(): #cattimp nu am explorattoate starile
        (parent, path) = stack.pop() #scoatem ultima stare adaugata din stiva
        if problem.isGoalState(parent): #verificam daca e scop
            return path
        expanded = expanded + [parent] #il adaugam la lista nodurilor expandate

        for children, action, cost in problem.expand(parent): #ii parcurgem succesorii
            if children not in expanded: #daca nu au fost expandati
                stack.push((children, path + [action])) #ii punem pe stiva
```

Explanation:

- Se utilizeaza o stiva pentru a adauga starile si o cale de la pozitia de inceput catre acea stare. Am utilizat stiva pentru a le putea scoate din lista in ordinea inversa a adaugarii. Se expandeaza fiecare nod din stiva adaugand vecinii sai si se verifica daca este scop inainte de eliminare.

Commands:

- python3 pacman.py -l binMaze -z .5 -p searchAgent

1.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Is the found solution optimal? Explain your answer.
A1: nu este optima deoarece nu ia in considerare costul.
Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: Solutia este corecta.

1.1.3 Personal observations and notes

1.2 Question 2 - Breadth-first search

In this section the solution for the following problem will be presented:

*"In **search.py**, implement the **Breadth-First search** algorithm in function breadthFirstSearch."*.

1.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def breadthFirstSearch(problem):

    queue = util.Queue()
    visited = []

    queue.push((problem.getStartState(), [])) #adaugam in coada primul nod
    visited = visited + [problem.getStartState()] #il vizitam

    while not queue.isEmpty():
        (parent, path) = queue.pop() #extragem primul nod adaugat din lista
        if problem.isGoalState(parent): #verificam daca e scop
            return path #in caz afirmativ, returnam drumul spre acel nod

        for children, action, cost in problem.expand(parent): #expandam nodul
            if children not in visited:
                queue.push((children, path + [action])) #adaugam copiii care nu au fost vizitati in coa
            visited = visited + [children] #ii marcam ca vizitati
```

Explanation:

pentru BFS este asemanatoare cu cea pentru DFS, diferenta fiind faptul ca la BFS se utilizeaza o coada in locul stivei, starile fiindexpandate in ordinea in care au fost introduse on coada.

Commands:

- python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs

1.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Is the found solution optimal? Explain your answer.
A1:
Da.
Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2:
Punctajul este maxim.

1.2.3 Personal observations and notes

1.3 Question 3 - Uniform-cost search

In this section the solution for the following problem will be presented:

> *"In search.py, implement* ***Uniform-cost graph search*** *algorithm in uniformCostSearchfunction"*

1.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def UCS(problem):
    queue = util.PriorityQueue()
    visited = []

    queue.push((problem.getStartState(), []), 0) #adaugam in coada de prioritati prima stare si costul=
    visited = visited + [problem.getStartState()] #il marcam ca vizitat

    while not queue.isEmpty():
        parent, path = queue.pop() #extragem din coada nodul care are costul minim
        visited = visited + [parent] #marcam nodul ca vizitat
        if problem.isGoalState(parent):#daca este scop
            return path #returnam calea

        for children, action, cost in problem.expand(parent): #expandam nodul
            if children not in visited:
                queue.push((children, path + [action]), problem.getCostOfActionSequence(path + [action])
            visited = visited + [children] #ii marcam ca vizitati
            if problem.isGoalState(children):  #verificam daca copilul este scop
                queue.push((children, path + [action]), problem.getCostOfActionSequence(path + [action])

    return None
    # util.raiseNotDefined()
```

Explanation:

- Algoritmul implementat este asemanator cu bfs, diferenta fiind ca UCS ia in calcul si costul drumului de la sursa la starea data. Este utilizata o coada de prioritati, nodurile fiind parcurse in ordine crescatoare a costului.

Commands:

- python3 pacman.py -l tinyMaze -p SearchAgent -a fn=ucs

1.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Compare the results to the ones obtained with DFS. Are the solutions different? Is the number of extended (explored) states smaller? Explain your answer.

A1:

Numarul de stari expandate este mai mic decat la dfs.

Q2: Consider that some positions are more desirable than others. This can be modeled by a cost function which sets different values for the actions of stepping into positions. Identify in **searchAgents.py** the description of agents StayEastSearchAgent and StayWestSearchAgent and analyze the cost function. Why the cost .5 ** x for stepping into (x,y) is associated to StayWestAgen.

A2:

Q3: Run autograder *python autograder.py* and write the points for Question 3.

A3:

Nu avem test pentru ucs.

1.3.3 Personal observations and notes

1.4 References

2 Informed search

2.1 Question 4 - A* search algorithm

In this section the solution for the following problem will be presented:

*"Go to aStarSearch in search.py and implement **A* search algorithm**. A* is graphs search with the frontier as a priorityQueue, where the priority is given bythe function g=f+h".*

2.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def aStarSearch(problem, heuristic=nullHeuristic):

    queue = util.PriorityQueue()
    visited = []

    queue.push((problem.getStartState(), []), heuristic(problem.getStartState(),problem)) #adaugam in c
    visited = visited + [problem.getStartState()] #il marcam ca vizitat

    while not queue.isEmpty():
        parent, path = queue.pop() #extragem din coada nodul care are suma dintre cost si euristica min
        visited = visited + [parent] #marcam nodul ca vizitat
        if problem.isGoalState(parent):#daca este scop
            return path #returnam calea

        for children, action, cost in problem.expand(parent): #expandam nodul
            if children not in visited:
                queue.push((children, path + [action]), problem.getCostOfActionSequence(path + [action])
            visited = visited + [children] #ii marcam ca vizitati
            if problem.isGoalState(children):  #verificam daca copilul este scop
                queue.push((children, path + [action]), problem.getCostOfActionSequence(path + [action])

    return None
    # util.raiseNotDefined()
```

Listing 1: Solution for the A* algorithm.

Explanation:

- A* are o implementare asemanatoare cu cea a algoritmului BFS, diferentele fiind faptul ca in coada, pe langa pozitie si drumul spre acea pozitie din starea initiala, se mai adauga si valoarea unei functii, elementele scotandu se din coada in functie de valoarea acestui parametru. Functia se calculeaza adunand costul distantelor de la pozitia de start la pozitia respectiva si valoarea unei euristici in acel punct, aceasta reprezentand o aproximare a distantei fata de scop.

Commands:

- python3 pacman.py -l bigMaze -p SearchAgent -a fn=astar

2.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Does A* and UCS find the same solution or they are different?
A1: Sunt diferite intrucat A* nu ia in considerare doar costul drumului pana la o pozitie, ci si valoarea euristicii in acea pozitie.
Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2:
Da, deoarece functia dupa care de alege ordinea expandarii este mai buna.
Q3: Does A* finds the solution with fewer expanded nodes than UCS?
A3:

Q4: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).
A4:4P.

2.1.3 Personal observations and notes

Pentru anumite teste se luau mai multe stari schiar si dupa gasirea solutiei. Pentru rezolvarea acestei probleme am verificat daca vreuna din starile urmatoare este scop, dupa expandarea parintelui.

2.2 Question 5 - Find all corners - problem implementation

In this section the solution for the following problem will be presented:

*"Pacman needs to find the shortest path to visit all the corners,regardless there is food dot there or not. Go to **CornersProblem** in searchAgents.py and propose a representation of the state of this search problem. It might help to look at the existing implementation for PositionSearchProblem. The representation should include only the information necessary to reach the goal. Read carefully the comments inside the class CornersProblem."*.

2.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and child function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
```

```
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        return (self.startingPosition, False, False, False, False) #pozitia + 4 vatiabile booleene=>tru
        util.raiseNotDefined()

    def isGoalState(self, state):
        return state[1] and state[2] and state[3] and state[4] #toate vatiabilele booleene au valoarea
        util.raiseNotDefined()


    def expand(self, state):

        children = []
        for action in self.getActions(state):
            nextState = self.getNextState(state, action) #starea copilului
            cost = self.getActionCost(state, action, nextState) #costul deplasarii de la parinte la fiu
            children.append((nextState, action, cost)) #atasam succesorul
        self._expanded += 1 # DO NOT CHANGE
        return children

    def getActions(self, state):

        possible_directions = [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
        valid_actions_from_state = []
        for action in possible_directions:
            x, y = state[0]  #pozitia
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy) #pozitia succestorului
            if not self.walls[nextx][nexty]: #daca nu este perete
                valid_actions_from_state.append(action) #se adauga la lista actiunilor valide
        return valid_actions_from_state

    def getActionCost(self, state, action, next_state):
        assert next_state == self.getNextState(state, action), (
            "Invalid next state passed to getActionCost().")
        return 1

    def getNextState(self, state, action):
        assert action in self.getActions(state), ( #validam datele de intrare
            "Invalid action passed to getActionCost().")

        x, y = state[0] #pozitia
        dx, dy = Actions.directionToVector(action) #directia
```

```
        children = int(x + dx), int(y + dy)  #pozitia succesorului = pozitia initiala+deplasarea

        corners_state = list(state[1:])
        if children in self.corners: #daca succesorul contine mancare
            corners_state[self.corners.index(children)] = True #se modifica variabila corespunzatoare c
        return (children, corners_state[0], corners_state[1], corners_state[2], corners_state[3]) #actu
        util.raiseNotDefined()


    def getCostOfActionSequence(self, actions):
        """
        Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999.  This is implemented for you.
        """
        if actions == None: return 999999
        x, y = self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)
```

Explanation:

Aceasta cerinte presupune implementarea mai multor metode din clasa CornersProblem.Im netoda next State Se transmite urmatoarea pozitie, dar se si verifica daca pozitia nu esteunul dintre colturile-labitintului, caz in care, parametrul transmis, state, isi modifica valoarea de pe pozitia egala cu numarul coltului in TRUE.Problema este rezolcata atunci cand pe toate pozitiile se afla valoarea TRUE.

Commands:

- python3 pacman.py -l mediumCorners -p AStarCornersAgent

2.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: For mediumCorners, BFS expands a big number - around 2000 search nodes. It's time to see that A* with an admissible heuristic is able to reduce this number. Please provide your results on this matter. (Number of searched nodes).
A1: 1658 noduri expandate.

2.2.3 Personal observations and notes

2.3 Question 6 - Find all corners - Heuristic definition

In this section the solution for the following problem will be presented:

*"Implement a consistent heuristic for CornersProblem. Go to the function **cornersHeuristic** in searchAgent.py."*.

2.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def cornersHeuristic(state, problem):

    dist = []
    for corners_left in problem.corners: #calculam distanta manhattan dintre pozitie si fiecare bucata 
        dist.append(abs(state[0][0] - corners_left[0]) + abs(state[0][1] - corners_left[1])) #

    if not dist:
        return 0
    return min(dist) #returnam minimul acestei distante , astfel, valorile euristicii pe pozitiile care
```

Explanation:

- Euristica implementata este euristica Manhattan, Calculata prin adunarea valorilor absolute ale diferentelor cordonatelor starii si ale colturilor. Am ales aceasta euristica deoarece am expandat mai putine noduri decat daca foloseam euristica euclidiana.

Commands:

- python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

2.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test with on the mediumMaze layout. What is your number of expanded nodes?
A1:1658

2.3.3 Personal observations and notes

2.4 Question 7 - Eat all food dots - Heuristic definition

In this section the solution for the following problem will be presented:

"Propose a heuristic for the problem of eating all the food-dots. The problem of eating all food-dots is already implemented in ***FoodSearchProblem*** *in searchAgents.py."*.

2.4.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    min_position = position #pozitia celei mai apropiate bucati de mancare
    min_dist = 999999
```

```
    for food in foodGrid.asList(): #pentru fiecare bucatica de mancare
        dist = (abs(position[0] - food[0]) + abs(position[1] - food[1])) #calculam distanta
        if min_dist >= dist: #alegem distanta minima
            min_dist = dist
            min_position = food #actualizam pozitia celei mai apropiate bucati de mancare

    return mazeDistance(position, min_position, problem.startingGameState) #returnam distanta de la poz
    if not dist:
        return 0
    return min(dist)
    return 0



#suboptimal search
def isGoalState(self, state):

        return state in self.food.asList() #retunreaza true atunci cand pe pozitia data se afla o bucat

        util.raiseNotDefined()

 def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        problem = AnyFoodSearchProblem(gameState)

        return search.breadthFirstSearch(problem) # calea catre cea mai apropiata bucatica de mancare e
        util.raiseNotDefined()
```

Explanation:

- Algoritmul ales calculeaza mancarea cu distanta minima data de locul in care ne aflam su returneaza distanta de la coordonatele acesteia la pozitia a carei euristici vrem sa o determinam.

Commands:

- python3 pacman.py -l trickySearch -p AStarFoodSearchAgent

2.4.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test with autograder *python autograder.py*. Your score depends on the number of expanded states by A* with your heuristic. What is that number?
A1:7300

2.5 Question 7 - Closest Dot

Code:

```
def isGoalState(self, state):

        return state in self.food.asList() #retunreaza true atunci cand pe pozitia data se afla o bucat
```

```

        util.raiseNotDefined()

    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        problem = AnyFoodSearchProblem(gameState)

        return search.breadthFirstSearch(problem) # calea catre cea mai apropiata bucatica de mancare e
        util.raiseNotDefined()
```

Explanation:

- Functia Is goal state verifica daca pe pozitie se afla vreo bucata de mancare, caz in care returneaza TRUE. Functia FindPathToClosestDot returneaza drumul pana la cea mai apropiata bucata de mancare, apeland functia de cautare BFS implementata la Q2.

Commands:

- python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

2.5.1 Personal observations and notes

2.6 References

3 Adversarial search

3.1 Question 8 - Improve the ReflexAgent

In this section the solution for the following problem will be presented:

"Improve the ReflexAgent such that it selects a better action. Include in the score food locations and ghost locations. The layout testClassic should be solved more often.".

3.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
 def getAction(self, gameState):
     legalMoves = gameState.getLegalActions()

     # Choose one of the best actions
     scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
     bestScore = max(scores)
     bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
     chosenIndex = random.choice(bestIndices) # Pick randomly among the best

     "Add more of your code here if you want to"

     return legalMoves[chosenIndex]

 def evaluationFunction(self, currentGameState, action):
     childGameState = currentGameState.getPacmanNextState(action)
     newPos = childGameState.getPacmanPosition()
     newFood = childGameState.getFood()
     newGhostStates = childGameState.getGhostStates()
     newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

     dist = []
     for food in currentGameState.getFood().asList():
         dist.append(manhattanDistance(food, newPos)) #calculam distanta manhattan de la pozitia cop

     if action == 'Stop': #oprire
         return -9999999

     for ghost in newGhostStates: #daca intalnim vreo fantoma
         if ghost.getPosition() == newPos:
             return -9999999  #returnam cel mai mic nr negativ
```

Explanation:

- Se verifica daca actiunea este 'STOP' sau daca am intalnir o fantoma, cazuri in care se returneaza cel mai mic numar negativ. In restul cazurilor se returneaza minimul dintre distntele manhattan dintre fiecare pozitie pe care se afla mancarea si pozitia urmatoare cu semn schimbat.

Commands:

- python3 pacman.py -p ReflexAgent -l testClassic

3.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test your agent on the openClassic layout. Given a number of 10 consecutive tests, how many types did your agent win? What is your average score (points)?
A1: De fiecare data. Scorul mediu = 562

3.1.3 Personal observations and notes

3.2 Question 9 - H-Minimax algorithm

In this section the solution for the following problem will be presented:

" Implement H-Minimax algorithm in MinimaxAgentclass from multiAgents.py. Since it can be more than one ghost, for each max layer there are one ormore min layers.".

3.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """

    def getAction(self, gameState):
        val_min = -999999
        for act in gameState.getLegalActions(0): #pentru fiecare actiune
            minim = self.minim(gameState.getNextState(0, act), 0, gameState.getNumAgents() - 1) #apelam
            if minim > val_min:
                val_min, action = minim, act #salvam valoarea minima si actiunea

        return action

    def maxim(self, state, depth):

        depth += 1 #creste adancimea

        if self.depth == depth or state.isWin() or state.isLose(): #executie incheiata
            return self.evaluationFunction(state)

        maxi = -999999
```

```

        for action in state.getLegalActions(0): #pentru orice actiune legala
            children = state.getNextState(0, action)
            maxi = max(maxi, self.minim(children, depth, state.getNumAgents() - 1)) #calculam maximul d
        return maxi


    def minim(self, state, depth, ghosts):

        if self.depth == depth or state.isWin() or state.isLose(): #executie incheiata
            return self.evaluationFunction(state)

        ghostNr = state.getNumAgents() - ghosts  #nr de identificare a fantomei
        mini = 9999999
        for action in state.getLegalActions(ghostNr):#pentru fiecare mutare care nu este in zid
            children = state.getNextState(ghostNr, action) #trecem in starea urmatoare
            if ghosts == 1: #daca a mai ramas o singura fantoma
                mini = min(mini, self.maxim(children, depth)) # calculam minimul dintre vechiul minim s
            if ghosts > 1: #daca numarul de fantome este mai mare decat 1
                mini = min(mini, self.minim(children, depth, ghosts - 1)) # salvam minimul dintre vechi
        return mini
```

Explanation:

- Acest Algoritm este folosit pentru cazul in care avem mai multi agenti inamici, unde, un agent este min, iar celalalt max. Pentru fiecare actiune a lui Max, min executa o actiune, functiile de min si max apelandu-se succesiv una pe cealalta.

Commands:

- python3 pacman.py -p MiniMaxAgent -l TrappedClassic

3.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test Pacman on trappedClassic layout and try to explain its behaviour. Why Pacman rushes to the ghost?
A1: Pacman reuseste sa manance aproape toata mancarea, dar, la final, este prins de una dintre fantome, indiferent de decizia pe care o ia.

3.2.3 Personal observations and notes

3.3 Question 10 - Use $\alpha - \beta$ pruning in AlphaBetaAgent

In this section the solution for the following problem will be presented:

*" Use alpha-beta prunning in **AlphaBetaAgent** from multiagents.py for a more efficient exploration of minimax tree.".*

3.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  class AlphaBetaAgent(MultiAgentSearchAgent):
2      """
3      Your minimax agent with alpha-beta pruning (question 3)
4      """
5  
6      def getAction(self, gameState):
7  
8          mini = -999999
9  
10         a = -999999
11 
12         for action in gameState.getLegalActions(0):
13             new_state = gameState.getNextState(0, action)
14             m = self.minim(new_state, 0, gameState.getNumAgents()-1, a, 99999)
15             if m > mini:
16                 mini, actMin = m, action
17 
18             a = max(a, mini)
19         return actMin
20 
21     def maxim(self, state, depth, a, b):
22         depth += 1
23         maxi = -99999999
24         if self.depth == depth or state.isWin() or state.isLose():
25             return self.evaluationFunction(state)
26         else:
27             for action in state.getLegalActions(0):
28                 children = state.getNextState(0, action)
29                 maxi = max(maxi, self.minim(children, depth, state.getNumAgents()-1, a, b))
30 
31                 if maxi > b:
32                     return maxi
33                 a = max(a, maxi)
34         return maxi
35 
36     def minim(self, state, depth, ghosts_left, a, b):
37 
38         if self.depth == depth or state.isWin() or state.isLose():
39             return self.evaluationFunction(state)
40 
41         ghost_id = state.getNumAgents() - ghosts_left
42         mini = 9999999
43         for action in state.getLegalActions(ghost_id):
44             children= state.getNextState(ghost_id, action)
45             if ghosts_left == 1:
46                 mini = min(mini, self.maxim(children, depth, a, b))
47             if ghosts_left > 1:
48                 mini = min(mini, self.minim(children, depth, ghosts_left-1, a, b))
```

```
            if mini < a:
                return mini
            b = min(b, mini)
        return mini
```

Explanation:

- Functiile sunt asemanatoare cu cele implementate la MiniMax doar ca aici se folosesc doua variabile, alfa in care se salveaza maximul dintre valoarea curenta a lui alfa si maximul calculat in functia maxi si beta care este utilizata pentru a salva minimul dintre valoarea curenta a lui beta si minimul calculat in functia mini, atunci cand alfa este mai mic sau egal cu minimul. Variabilele alfa si beta au scopul de a limita numarul de stari ale jocului, alfa reprezentand cea mai buna alegere pentru max, iar bet, cea mai buna alegere pentru min.

Commands:

- python3 pacman.py -p AlphaBetaAgent -l smallClassic

3.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test your implementation with autograder **python autograder.py** for Question 3. What are your results?
A1: Pacman este prins in continuare de fantome in acelasi punct.

3.3.3 Personal observations and notes

3.4 References

a

4 Personal contribution

4.1 Question 11 - Define and solve your own problem.

In this section the solution for the following problem will be presented:

4.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

Explanation:

-

Commands:

-

4.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

4.1.3 Personal observations and notes

4.2 References