UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Enache Mihai si Gavrilescu Andreea-Lavinia
Grupa: 30238

Facultatea de Automatica
si Calculatoare

7 Decembrie 2022

Cuprins

1 Uninformed search

Uniform cost search este un tip de cautare care are ca frontiera un *priority queue*, adica expandeaza nodurile cu **cel mai mic** cost al traseului.

1.1 Question 1 - Depth-first search

Cerinta: In **search.py**, implementati algoritmul **Depth-First search** in functia *depthFirstSearch*.

Depth-First search este un *arbore/graf de cautare* care are ca frontiera un **LIFO** (stack) si extinde mereu cel mai adanc nod din frontiera curenta a arborelui de cautare. Cautarea continua la cel mai adanc nivel al arborelui unde nodurile nu au sucesori. Dupa ce aceste noduri sunt expandate, sunt scoase din frontiera, iar cautarea continua cu explorarea sucesorilor urmatorului cel mai adanc nod.

Am inceput prin a initializa variabilele necesare in dezvoltarea algoritmului, dupa cum urmeaza:

1. **stack** = util.Stack(), stiva in care se vor adauga nodurile expandate
2. **visited** = [], lista care va contine nodurile ce au fost deja *vizitate*
3. **initial_node** = (problem.getStartState(), []), **nodul de start** (*pozitia lui*) si **traseul** (**null** la inceput)

Adaugam **nodul de start** pe stiva si vom incepe cautarea. Cautarea se opreste in momentul in care stiva este goala. Fiecare nod va fi extras din stiva si vom verifica daca acesta a fost sau nu vizitat. Daca nu, acesta este adaugat in lista **visited** si vom verifica daca acesta este **goal state-ul** la care vrem sa ajungem. In caz afirmativ, vom returna **paht-ul**, altfel vom parcurge succesorii nodului curent si ii vom adauga in stiva.

Implementarea functiei **depthFirstSearch**:

```python
def depthFirstSearch(problem):
    """
    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:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"
    # dfs uses a stack
    stack = util.Stack()
    # we'll "mark" the nodes that have been visited during the search
    visited = []
    # the start node that has a position and a path
    initial_node = (problem.getStartState(), [])

```

```python
    # push the start node to the stack
    stack.push(initial_node)

    while not stack.isEmpty():
        # every node has a state, a set of legal actions and a cost
        node_state, node_actions = stack.pop()
        # check if the note has been visited or not
        if node_state not in visited:
            # if not, add it to the visited list
            visited.append(node_state)

            # check if the current node is not the goal
            if problem.isGoalState(node_state):
                # return the path
                return node_actions
            else:
                # if not, expand the node
                successors = list(problem.getSuccessors(node_state))
                # successor[0] = state; successor[1] = legal actions
                for successor in successors:
                    # build the path
                    path = node_actions + [successor[1]]
                    new_node = (successor[0], path)
                    # create a node with the state of the successor and the path to it and p
                    stack.push(new_node)
    return node_actions
```

Complexitatea algoritmului este: $\mathbf{O(b^m)}$, unde **b** este *factorul de ramificare* al arborelui de cautare, iar **m** este **adancimea maxima** a oricarui nod.

Pentru **testare** se poate utiliza ***comenzile***:

- python pacman.py -l tinyMaze -p SearchAgent
- python pacman.py -l mediumMaze -p SearchAgent
- python pacman.py -l bigMaze -z .5 -p SearchAgent

1.2 Question 2 - Breadth-first search

Cerinta: In **search.py**, implementeaza algoritmul **Breadth-First search** in functia *breadthFirstSearch*.

Breadth-first search este un *arbore/graf de cautare* care are ca frontiera un **FIFO** (queue) si cauta nodurile aflate mai in adancime.

Am inceput prin a initializa variabilele necesare in dezvoltarea algoritmului, astfel:

1. **queue** = util.Queue(), dupa cum specifica cerinta;
2. **start** = (problem.getStartState(), []) care contine **pozitia** *nodului de start* si **traseul** (nul la inceput)
3. **visited** = [], lista care va contine nodurile ce au fost *deja vizitate*

Apoi se adauga **nodul de start** in *coada*. Pentru a efectua o cautare amanuntita vom parcurge coada cu un *while*. Vom extrage si retine *nodul curent* intr-o variabila pentru viitoare verificari. Intai, daca este **goal state**, vom returna **traseul**. Apoi, daca nodul *nu a fost vizitat*, il *adaugam* in *visited* si initializam o **lista de succesori** ai nodului. Parcurgem aceasta lista si verificam daca **succesorul** respectiv *nu a fost vizitat* pentru *a construi traseul* si a-l *adauga* in coada. **La final**, daca *nu mai avem noduri care trebuie explorate*, vom returna o *lista vida*.

Implementarea functiei **breadthFirstSearch**:

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    queue = util.Queue()  # bfs is using a queue
    start = (problem.getStartState(), [])  # the start node: location, path
    visited = []  # list to check if a node was already visited

    queue.push(start)  # insert the start node in queue

    while not queue.isEmpty():
        current_node = queue.pop()
        # 0 for location, 1 for path

        # if current_node is the goal state, then we return its path
        if problem.isGoalState(current_node[0]):
            return current_node[1]   #

        if current_node[0] not in visited:
            visited.append(current_node[0])  # we add it to the list of visited nodes

            # keep the successors and their paths
            successors = list(problem.getSuccessors(current_node[0]))

            for successor in successors:
                if successor[0] not in visited:
                    path = current_node[1] + [successor[1]]  # we reconstruct the path
                    queue.push((successor[0], path))  # we add them to the queue

    # return empty if we don't have any nodes left to explore
    return []
    util.raiseNotDefined()
```

Complexitatea de timp si spatiu este: $\mathbf{O(b^d)}$, unde **b** este *factorul de ramificare* al arborelui de cautare si **d** este *adancimea* la care este situat cel mai adanc nod.

Pentru **testare** se poate utiliza *comanda*:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

1.3 Question 3 - Uniform-cost search

Cerinta: In **search.py**, implementati algoritmul **Uniform-Cost search** in functia *uniformCostSearch*.

Uniform-Cost search este un *graf de cautare* care are ca frontiera o **Priority queue** si expandeaza nodul *n* cu *path-ul cu costul minim* ***g(n)***. Coada de prioritati este ordonata dupa *g*. **Uniform-cost search** expandeaza nodurile *în ordinea costului lor optim al traseului.*

Am inceput prin a initializa variabilele necesare in dezvoltarea algoritmului, dupa cum urmeaza:

1. **priorityQueue** = util.PriorityQueue(), **coada de prioritati** in care se vor adauga nodurile expandate
2. **visited** = , **lista** care va contine nodurile ce au fost deja *vizitate*
3. **initial_node** = (problem.getStartState(), [], 0), **nodul de start** (*pozitia lui*), **traseul** (**null** la inceput) si **costul** (**0** initial)

Adaugam **nodul de start** in **coada de prioriati** si vom incepe cautarea. Cautarea se opreste in momentul in care coada de prioritati este goala. Fiecare nod va fi extras din coada de prioritati si vom verifica urmatoarele conditii: daca acesta a fost sau nu vizitat sau daca costul nodului curent este mai mic decat costul de pana acum. Daca una din cele doua conditii este indeplinita, nodul curent este adaugat in lista **visited** si vom verifica daca acesta este **goal state-ul** la care vrem sa ajungem. In caz afirmativ, vom returna **paht-ul**, altfel vom parcurge succesorii nodului curent si ii vom adauga in coada de prioritati, tinand cont de a face modificari si legate de cost.

Implementarea functiei **uniformCostSearch**:

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    # ucs uses a priority queue
    priorityQueue = util.PriorityQueue()
    # we'll "mark" the nodes that have been visited during the search
    visited = {}
    # the start node that has a position, a set of legal actions and a cost
    initial_node = (problem.getStartState(), [], 0)

    # push the start node to the priority queue
    priorityQueue.push(initial_node, 0)

    while not priorityQueue.isEmpty():
        # every node has a state, a set of legal actions and a cost
        node_state, node_actions, node_cost = priorityQueue.pop()

        # check if the current node has been visited or not
        # check if we have a lower cost
        if (node_state not in visited) or (node_cost < visited[node_state]):
            # update cost
            visited[node_state] = node_cost
            #check if we reached the goal
            if problem.isGoalState(node_state):
                # return path
                return node_actions
            else:
                # expand current node
                successors = list(problem.getSuccessors(node_state))
                # successor[0] = state; successor[1] = legal actions
                # successor[2] = cost
                for successor in successors:
```

```
                    # build the path
                    path = node_actions + [successor[1]]
                    # update the path's cost
                    total_cost = node_cost + successor[2]
                    new_node = (successor[0], path, total_cost)

                    # create a new node and push it to the priority queue
                    priorityQueue.update(new_node, total_cost)
    # return path
    return node_actions
```

Complexitatea algoritmului este: $\mathbf{O(b^{1+[C^*/e]})}$, unde **b** este *factorul de ramificare* al arborelui de cautare, **C*** este **costul optim al solutiei**, iar **e** ar fi **costul fiecarei actiuni** ([x] - *partea intreaga* a numarului x).

Pentru **testare** se poate utiliza ***comanda***:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

2 Informed search

2.1 Question 4 - A* search algorithm

Cerinta: In **search.py**, implementeaza algoritmul **A* search algorithm** in functia *aStarSearch*.

A* search algorithm este un *arbore/graf de cautare* care are ca frontiera un **priority queue**.

Am inceput prin a initializa variabilele necesare in dezvoltarea algoritmului, astfel:

1. **priority queue** = util.PriorityQueue(), dupa cum specifica cerinta;
2. **start** = (problem.getStartState(), [], 0) care contine **pozitia** *nodului de start*, **traseul** (nul la inceput) si **costul**
3. **visited** = [], lista care va contine nodurile ce au fost *deja vizitate*

Apoi se adauga **nodul de start** in *priority queue*. Pentru a efectua o cautare eficienta vom parcurge coada cu un *while*. Vom extrage si retine *nodul curent* intr-o variabila pentru viitoare verificari. Intai, daca este **goal state**, vom returna **traseul**. Apoi, daca nodul *nu a fost vizitat*, il *adaugam* in *visited* si initializam o **lista de succesori** ai nodului. Parcurgem aceasta lista si verificam daca **succesorul** respectiv *nu a fost vizitat* pentru *a reconstrui traseul*, **costul** care este format din *costul initial* dintre *nodul curent* si *succesor* adunat cu *euristica succesorului*, iar apoi a-l *adauga* in coada. **La final**, daca *nu mai avem noduri care trebuie explorate*, vom returna o *lista vida*.

Implementarea functiei **aStarSearch**:

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

    priority_queue = util.PriorityQueue()  # aStarSearch is using a priority queue
    start = (problem.getStartState(), [], 0)  # the start node: location, path, cost
    visited = []  # list to check if a node was already visited

```

```python
    priority_queue.push(start, 0)  # insert the start node in queue

    while not priority_queue.isEmpty():
        current_node = priority_queue.pop()
        # 0 for location, 1 for path, 2 for cumulative cost

        # if current_node is the goal state, then we return its path
        if problem.isGoalState(current_node[0]):
            return current_node[1]

        if current_node[0] not in visited:
            visited.append(current_node[0])  # we add it to the list of visited nodes

            # keep the successors and their paths
            successors = list(problem.getSuccessors(current_node[0]))

            for successor in successors:
                if successor[0] not in visited:
                    path = current_node[1] + [successor[1]]  # we reconstruct the path
            # the path cost to the current node + the path cost to the current successor
                    initial_cost = current_node[2] + successor[2]
            # the total cost is the sum  of the previous cost and
            # the heuristic of the successor
                    cost = initial_cost + heuristic(successor[0], problem)
            # we add them to the priority queue
                    priority_queue.push((successor[0], path, initial_cost), cost)

    # return empty if we don't have any nodes left to explore
    return []

    util.raiseNotDefined()
```

Complexitatea de spatiu este: $\mathbf{O(b^d)}$, unde **b** este *factorul de ramificare* al arborelui de cautare si **d** este *adancimea* la care este situat cel mai adanc nod.

Complexitatea de timp depinde de euristinca, insa in *cel mai rau caz* este: $\mathbf{O(b^d)}$.

Pentru **testare** se poate utiliza *comanda*:

- python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

2.2 Question 5 - Finding all Corners

Cerinta: Pacman trebuie sa gaseasca cel mai scurt path pentru a vizita toate colturile, indiferent daca este sau nu mancare. In **searchAgents.py**, propuneti o reprezentare a starii acestei probleme.

Pentru aceasta problema, am adaugat cod in functiile **getStartState**, **isGoalState** si **getSuccessors** din clasa **CornersProblem**.

Functia **getStartState** va returna o tupla cu **pozitia de inceput** si o **lista goala** dedicata *colturilor vizitate*.

Functia **isGoalState** se va folosi pentru *a verifica daca toate cele 4 colturi au fost vizitate sau nu*. **Starea** va avea *doua "atribute"*: **pozitia curenta** si **colturile vizitate** pana in acel moment.

Functia **getSuccessors** se va folosi pentru a obtine *succesorii* unei stari. Se porneste din pozitia curenta si se memoreaza cate colturi sunt vizitate. Se incearca toate combinatiile de actiuni posibile (Nord, Sud, Est, Vest), iar pentru acele actiuni valide (urmatorul nod nu trebuie sa fie un zid in urma actiunii alese) se verifica daca nodul este un colt si dupa daca a fost vizitat sau nu. Daca nu a fost vizitat, se adauga la lista de colturi vizitate ale succesorilor. Costul fiecarei actiuni este considerat 1. Functia trebuie sa returneze o lista de forma (succesor, actiune, cost).

Implementare:

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

    You must select a suitable state space and successor 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
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        return (self.startingPosition, [])
        #util.raiseNotDefined()

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        current_position = state[0]
```

```
        visited_corners = state[1]
        # check if the current position is a corner
        if current_position in self.corners:
            # check if the corner has been visited
            if current_position not in visited_corners:
                # if not, we'll "mark" the corner so we know it has been visited
                visited_corners.append(current_position)
            # check if we've visited all corners
            if len(visited_corners) == 4:
                return True
            else:
                # we still have corners to visi
                return False
        else:
            # current position is not a corner
            return False
        #util.raiseNotDefined()

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.

         As noted in search.py:
            For a given state, this should return a list of triples, (successor,
            action, stepCost), where 'successor' is a successor to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that successor
        """

        x,y = state[0] #current position
        visited_corners = state[1]
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]

            "*** YOUR CODE HERE ***"
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x+dx), int(y+dy)
            nextNode = (nextx, nexty)
            hitsWall = self.walls[nextx][nexty]
            # check if successor is not a wall
            if not hitsWall:
                # get the visited corners until now
```

```
                successors_visited_corners = list(visited_corners)
                # check if successor is a corner
                if nextNode in self.corners:
                    # check if the corner has been visited
                    if nextNode not in successors_visited_corners:
                        # if not, we'll "mark" it now
                        successors_visited_corners.append(nextNode)
                successor = ((nextNode, successors_visited_corners), action, 1)
                successors.append(successor)

        self._expanded += 1  # DO NOT CHANGE
        return successors

    def getCostOfActions(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)
```

Pentru **testare** se poate utiliza ***comenzile***:

- python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

2.3 Question 6 - A consistent heuristic for Corners Problem

Cerinta: In **searchAgents.py**, implementeaza o **euristica consistenta** pentru *CornersProblem* in functia *cornersHeuristic*.

Euristicile ajuta la *eficienta* unui program, deoarece cu cat este *mai buna*, cu atat sunt **expandate** mai putine noduri.

Am inceput prin a initializa variabilele necesare in dezvoltarea algoritmului, astfel:

1. **position** = state[0] care contine starea transmisa ca parametru;
2. **visited** = state[1], lista care va contine nodurile ce au fost *deja vizitate*
3. **unvisited** = [], lista care va contine nodurile ce *nu au fost deja vizitate*;
4. **heuristic** = 0, care o sa contina cel mai scurt traseu;

Apoi parcurgem cele **4 colturi posibile**, iar daca *nu a fost vizitat* il adaugam la lista de *unvisited*. Apoi, pentru a efectua o cautare detaliata, vom folosi un *while* pentru a parcurge lista de **unvisited**. Calculam **distanta** folosindu-ne de *distanta manhattan*. Dupa aceea, actualizam *pozitia* folosita in distanta manhattan cu urmatorul colt nevizitat si adunam *distanta* la vechea valoare a *euristicii*. Inainte de a trece la o noua iteratie, **scoatem** *coltul vizitat* din *lista de colturi nevizitate* si **scadem** un element din lungimea listei. La sfarsit, vom returna valoarea **euristicii**.

Implementarea functiei **cornersHeuristic**:

```python
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    position = state[0]  # current possition
    visited = state[1]
    unvisited = []  # unvisited corners
    heuristic = 0  # the shortest path

    # there are 4 possible corners
    for i in range(4):
        if corners[i] not in visited:
            unvisited.append(corners[i])  # we add the corners we didn't visit

    length = len(unvisited)  # the length of the unvisited corners list
    while length:
    # we compute the distance depending on the corner from the list of unvisited corners
        tupleList = []
        for corner in unvisited:
            tupleList.append((manhattanDistance(position, corner), corner))

        dist, corner = min(tupleList)

        # we actualize the current position which is a corner from the unvisited list
        position = corner

        # we add the distance to the path
        heuristic += dist

        unvisited.remove(corner)  # we remove the corner we just visited
        length = length - 1  # and update the length of the list of unvisited corners

    return heuristic  # we return the sum of the shortest path
```

Pentru **testare** se pot utiliza *comenzile*:

- python pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch, prob=CornersProblem, heuristic=cornersHeuristic
- python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

2.4 Question 7 - Eating all food-dots

Cerinta: Propuneti o euristica pentru problema de a manca toate punctele de mancare. Problema este deja implementata in *FoodSearchProblem* in **searchAgents.py**.

Pentru aceasta problema, am ales ca euristica sa fie **calcularea distantei maxime de la pozitia curenta la cel mai indepartat punct de mancare**. Aceasta euristica ne ajuta sa rezolvam problema intr-un **numar redus de pasi** si **eficient**. Pentru calcularea distantei vom alege sa folosim **maze distance (distanta euclidiana)**, nu **manhattan distance** pentru a obtine o *acuratete mai buna a distantei.* Ideea euristicii este de a returna maximul distantelor de la pozitia curenta la punctele de mancare in ideea in care **punctul cel mai indepartat de mancare sa fie vizitat ultimul**.

Implementarea functiei **foodHeuristic**:

```
def foodHeuristic(state, problem):
    """
    Your heuristic for the FoodSearchProblem goes here.

    This heuristic must be consistent to ensure correctness.  First, try to come
    up with an admissible heuristic; almost all admissible heuristics will be
    consistent as well.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your heuristic is *not* consistent, and probably not admissible!  On the
    other hand, inadmissible or inconsistent heuristics may find optimal
    solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
    (see game.py) of either True or False. You can call foodGrid.asList() to get
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can query the
    problem.  For example, problem.walls gives you a Grid of where the walls
    are.

    If you want to *store* information to be reused in other calls to the
    heuristic, there is a dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls once and store that
    value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    start = problem.startingGameState
    foodList = foodGrid.asList()
    distancesToFood = []

    #no food -> return 0
    if len(foodList) == 0:
```

```python
        return 0

    for food in foodList:
        distancesToFood.append(mazeDistance(position, food, start))

    # return maximum distance between the current position and the farthest food
    return max(distancesToFood)
```

Pentru **testare** se poate utiliza *comanda*:

- python pacman.py -l testSearch -p AStarFoodSearchAgent

2.5 Question 8 - Find path to closest dot

Cerinta: In **searchAgents.py**, implementeaza functia *findPathToClosestDot*.

Trebuie doar sa retinem apelul functiei *breadthFirstSearch* intr-o variabila pe care o *returnam*.

Implementarea functiei **findPathToClosestDot**:

```python
def findPathToClosestDot(self, gameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the startState
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    "*** YOUR CODE HERE ***"
    actions = search.bfs(problem)
    return actions  # we are calling the bfs from "search.py"
    util.raiseNotDefined()
```

3 Adversarial search

3.1 Question 9 - Improve the ReflexAgent

Cerinta: In **multiAgents.py**, imnbunatateste *ReflexAgent*, astfel incat *sa aleaga* o actiune **mai buna**. In **scor** ar trebui incluse si *locatiile* mancarii si fantomelor.

Am inceput prin a initializa variabilele necesare in dezvoltarea algoritmului, astfel:

1. **score** = 0 la inceput;
2. **food** = newFood.asList(), lista care va contine *pozitiile* mancarii;
3. **totalFood** = len(food), variabila in care retinem numarul total de mancare;
4. **foodDistance** = math.inf, initializam distanta aceasta cu cea *mai mare* valoare posibila;
5. **totalGhosts** = len(newGhostStates), unde retinem numarul de fantome;

Prima data incepem prin a verifica daca avem *mancare disponibila*, deoarece in caz contrar, *foodDistance* o sa fie *0*. Apoi, atat timp cat avem *mancare disponibila* pe harta, calculam **distanta** catre fiecare mancare folosind *distanta manhattan* pe care o folosim in a determina *mancarea cea mai apropiata* care ar trebui luata prima data cu ajutorul unui *if*, fara sa uitam sa actualizam **scorul**. De asemenea, atat timp cat avem *fantome* pe harta, retinem **pozitia** fiecarei fantome pentru a o folosi in cadrul *distantei manhattan* care determina distanta catre **cea mai apropiata** fantoma. Daca aceasta este **mai mica decat 1**, reactualizam **scorul** cu -*infinit*, deoarece inseamna ca *am pierdut*. La sfarsit, vom returna **scorul final** obtinut in urma tuturor cazurilor posibile care *depind de mancare si fantome*.

Implementarea functiei **evaluationFunction**:

```python
def evaluationFunction(self, currentGameState, action):
    """
    Design a better evaluation function here.

    The evaluation function takes in the current and proposed successor
    GameStates (pacman.py) and returns a number, where higher numbers are better.

    The code below extracts some useful information from the state, like the
    remaining food (newFood) and Pacman position after moving (newPos).
    newScaredTimes holds the number of moves that each ghost will remain
    scared because of Pacman having eaten a power pellet.

    Print out these variables to see what you're getting, then combine them
    to create a masterful evaluation function.
    """
    # Useful information you can extract from a GameState (pacman.py)
    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

    "*** YOUR CODE HERE ***"

    score = 0  # we initialize the score with 0

    # we keep the food positions as a list
    food = newFood.asList()
    # we retain the total number of the food displayed on the grid
    totalFood = len(food)
    # we initialize the foodDistance with the highest value possible
    foodDistance = math.inf

    # we retain the number of ghosts
    totalGhosts = len(newGhostStates)

    # if we don't have any food available
    if totalFood == 0:
        # there are no distances
        foodDistance = 0

    for item in range(totalFood):
```

```python
        # calculate the distance from every food available
        mhFood = manhattanDistance(newPos, food[item])
        # calculate the nearest food available
        nearestFood = 1000 * totalFood + mhFood

        # the closer to food, the better, so we actualize the value
        # of the foodDistance if we find something closer
        if nearestFood < foodDistance:
            foodDistance = nearestFood

        # we add the foodDistance to the score
        score -= foodDistance

    for pos in range(totalGhosts):
        # get the ghost position
        ghostPosition = successorGameState.getGhostPosition(pos + 1)
        # calculate the distance to the nearest ghost
        mhGhost = manhattanDistance(ghostPosition, newPos)

        # it is too close to us, so we died
        if mhGhost <= 1:
            score -= math.inf

    return score
```

Pentru **testare** se pot utiliza *comenzile*:

- python pacman.py -p ReflexAgent -l testClassic
- python pacman.py – frameTime 0 -p ReflexAgent -k 1 -l mediumClassic
- python pacman.py – frameTime 0 -p ReflexAgent -k 2 -l mediumClassic

3.2 Question 10 - Minimax algorithm

Cerinta: Implementati algoritmul **H-Minimax** in **clasa MinimaxAgent** din fisierul **multiAgents.py**.

Minimax este un algoritm care este folosit in **luarea deciziilor** si in **teoria jocului** pentru a gasi **miscarea optima** pentru un jucator, presupunand ca si adversarul sau joaca optim. Acest algoritm calculeaza decizia minimax din starea curenta si foloseste un calcul recursiv simplu al valorilor minimax ale fiecarei stari. Cei doi jucatori sunt numiti **maximizer** (incearca sa obtina cel mai bun scor posibil) si **minimizer** (incearca sa obtina cel mai mic scor posibil).

Pentru implementarea algoritmului, s-a considerat ca **agentul 0** sa fie **Pacman**, iar **fantomele** sa aiba un **index mai mare decat 0**. Trebuie sa tinem cont ca in joc pot fi mai multe fantome, nu doar una.

Am definit **2 functii** pentru a implementa algoritmul:

- **maximizer** - functie dedicata Pacman-ului pentru a gasi actiunile care il pot ajuta sa obtina un scor cat mai mare
- **minimzer** - functie dedicata fantomelor

In functia **maximizer**, am inceput prin a verifica daca jocul s-a incheiat, adica daca agentul este intr-o stare de castig sau de pierdere. Daca da, vom returna scorul. Aceasta functie este dedicata Pacman-ului. Am obtinut actiunile pe care le poate face si am initializat scorul maxim. Am parcurs actiunile posibile pe care le poate face agentul si am cautat acea actiune care ar maximiza scorul.

In functia **minimizer**, am inceput prin a verifica daca jocul s-a incheiat. Avand in vedere ca pot fi mai multe fantome, pentru fiecare dintre ele trebuie sa obtinem actiunile posibile pe care le pot face si pentru fiecare agent sa incercam sa obtinem scorul minim posibil. O conditie suplimentara care apare este in cazul ultimului agent unde trebuie sa avem grija ca dupa acesta urmeaza randul agentului Pacman.

Implementarea algoritmului:

```python
def getAction(self, gameState):
        """
          Returns the minimax action from the current gameState using self.depth
          and self.evaluationFunction.

          Here are some method calls that might be useful when implementing minimax.

          gameState.getLegalActions(agentIndex):
            Returns a list of legal actions for an agent
            agentIndex=0 means Pacman, ghosts are >= 1

          gameState.generateSuccessor(agentIndex, action):
            Returns the successor game state after an agent takes an action

          gameState.getNumAgents():
            Returns the total number of agents in the game
        """
        "*** YOUR CODE HERE ***"
        pacman_agent = 0
        def maximizer(state, depth):
          # verify if Pacman won or not
          # either way, the algorithm stops and we return the score
          if state.isWin() or state.isLose():
            return state.getScore()
          # get all legal actions for Pacman
          actions = state.getLegalActions(pacman_agent)
          # initialize maximum score
          max_score = -math.inf
          score = max_score
          best_action = Directions.STOP
          # we're looking for the action that maximizes the score
          for action in actions:
            score = minimizer(state.generateSuccessor(pacman_agent, action), depth, 1)
            if score > max_score:
              # update score
              max_score = score
              best_action = action
          # we've reached the last level
          if depth == 0:
```

```
        return best_action
      else:
        return max_score

    def minimizer(state, depth, agent):
      # check if the game is over and return the score
      if state.isWin() or state.isLose():
        return state.getScore()
      # compute next agent index (ghost)
      next_agent = agent + 1
      # if it's last ghost's turn, it will be Pacman's next turn
      if agent == state.getNumAgents() - 1:
        next_agent = pacman_agent
      # get all legal actions for the agent
      actions = state.getLegalActions(agent)
      # initialize minimum score
      min_score = math.inf
      score = min_score
      # we're looking for the action that minimizes the score
      for action in actions:
        # check if it's last ghost
        if next_agent == pacman_agent:
          if depth == self.depth - 1:
            score = self.evaluationFunction(state.generateSuccessor(agent, action))
          else:
            score = maximizer(state.generateSuccessor(agent, action), depth+1)
        else:
          score = minimizer(state.generateSuccessor(
              agent, action), depth, next_agent)
        if score < min_score:
          # update score
          min_score = score
      return min_score
    return maximizer(gameState, 0)
```

Complexitatea algoritmului este: $\mathbf{O(b^m)}$, unde **b** reprezinta *numarul de actiuni legale in fiecare punct*, iar **m** este **adancimea maxima a arborelui**.

Pentru **testare**, se poate utiliza **comanda**:

- python pacman.py -p MinimaxAgent minimaxClassic -a depth=4

3.3 Question 11 - Alpha-beta prunning

Cerinta: In **multiAgents.py**, foloseste **alfa-beta prunning** in *AlfaBetaAgent* pentru o *explorare* **mai eficienta** a *arborelui minmax*.

Alpha-beta prunning poate fi aplicat pentru *a limita* numarul de stari ale jocului, atunci cand:

1. α = valoare (cea mai mare) *cea mai buna* de la un moment dat pentru *MAX*
2. β = valoare (cea mai mica) *cea mai buna* de la un moment dat pentru *MIN*

Pentru dezvoltarea acestui algoritm am folsoit ca tehnica de baza **Algoritmul MinMax** prezentat in *sectiunea anterioara*. Astfel:

- *functia* **maximizer** - folosita pentru a-l ajuta pe **pacman** sa gaseasca *actiunile* cu care va obtine un *scor cat mai mare* este imbunatatia cu parametrii α si β care *eficientizeaza* puterea de decizie asupra *urmatoarei actiuni* care va avea **cel mai bun scor**. *Diferenta* semnificativa consta in faptul ca valoarea maxima obtinuta este **comparata** cu β la sfarsitul algoritmului pentru a *concretiza cea mai buna alegere*.
- *functia* **minimizer** - folosita pentru a *identifica* pentru *fiecare* **fantoma** care sunt *actiunile posibile* pe care le pot face *in functie de* **agenti**.**Conditia suplimentara** care *imbunatateste* algoritmul original o reprezinta *compararea scorului* cu α in vederea obtinerii **celui mai bun** *scor minim*.

Implementarea functiei **alfaBeta**:

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        "*** YOUR CODE HERE ***"

        pacman = 0

        def maximizer(state, depth, alfa, beta):
            mx = -math.inf  # the initial maximum value
            nextAction = Directions.STOP  # stop command
            actions = state.getLegalActions(pacman)  # the actions of the agent

            # we terminate the state in either case and return the score
            if state.isWin() or state.isLose():
                return state.getScore()

            # we will try to find the max value for every successor
            # depending on the actions of the agent
            for action in actions:
                # retain the successor
                successor = state.generateSuccessor(
                    pacman, action)

                # we calculate the next value
                nextValue = minimizer(successor, depth, 1, alfa, beta)

                # we find the best value
                if nextValue > mx:
                    mx = nextValue  # keep it
                    nextAction = action  # and actualize the nextAction
```

```
                # we compare the best value with beta
                if mx > beta:
                    return mx

                alfa = max(mx, alfa)  # keep the new alfa value

            # if the algorithm reached the max depth
            if depth == 0:
                return nextAction  # then we stop
            else:
                return mx  # else we return the best value

        def minimizer(state, depth, agent, alfa, beta):

            mn = math.inf  # the initial minimum value
            nextAgent = agent + 1  # nextAgent = ghost
            actions = state.getLegalActions(agent)  # the actions of the agent

            # we terminate the state in either case and return the score
            if state.isLose() or state.isWin():
                return state.getScore()

            if agent == state.getNumAgents() - 1:
                nextAgent = pacman

            # we will try to find the min value for every successor
            # depending on the actions of the agent
            for action in actions:
                # we retain the successor
                successor = state.generateSuccessor(agent, action)

                if nextAgent == pacman:

                    if depth == self.depth - 1:
                        getScore = self.evaluationFunction(successor)
                    else:
                        getScore = maximizer(successor, depth + 1, alfa, beta)

                else:
                    getScore = minimizer(successor, depth, nextAgent, alfa, beta)

                # we try to get the min value
                if getScore < mn:
                    mn = getScore

                # we compare the score with alfa
                if mn < alfa:
```

```python
                    return mn

                beta = min(mn, beta)  # keep the new alfa value

            return mn  # we return the minimum

        return maximizer(gameState, 0, -math.inf, math.inf)
```

Complexitatea de timp este: $\mathbf{O(b^{m/2})}$, unde **b** este *factorul de ramificare* al arborelui de cautare, iar **m** este *adancimea maxima* a acestuia.

Pentru **testare** se poate utiliza *comanda*:

- python pacman.py -p AlphaBetaAgent -a depth =3 -l smallClassic

3.4 Suplimentar pentru nota 10

Cerinta: Propunerea unei **componente originale**: *alt algoritm de cautare*, *alta problema de cautare*, *alte euristici*

Am ales sa implementez un alt **algoritm de cautare**. Algoritmul ales este **Iterative deepening depth-first search**.

Algoritmul **Iterative deepening depth-first search** este o **strategie generala** folosita adesea in combinatie cu **depth-first tree search** care gaseste **cea mai buna limita de adancime**. **Limita** creste treptat pana cand **goal-ul** este atins. Acest lucru se va intampla cand **limita de adancime** ajunge la **d**, *adancimea celui mai apropiat nod de obiectiv* (shallowest goal node). Algoritmul combina beneficiile *algoritmilor de cautare* **Breadth-first search** si **Depth-first search**.

Implementarea algoritmului **Iterative deepening depth-first search**:

```python
def iterativeDeepeningSearch(problem):
    # ids uses a stack
    stack = util.Stack()
    # ids runs dfs algorithm within a depth_bound
    # if the goal is not reached withing the depth_bound
    # we treat it as a dead-end
    depth_bound = 0

    # repeat search until we reach the goal
    while True:
        # list to "mark" the visited nodes
        visited = []
        # push the start node to the stack
        stack.push((problem.getStartState(), [], 0))

        # current_node[0] = node state; current_node[1] = node_actions
        # current_node[2] = cost
        current_node = stack.pop() # extract the node from the stack
        # add it to the visited nodes list
        visited.append(current_node[0])
        # if the current node is not the goal, expand it
        while not problem.isGoalState(current_node[0]):
```

```
            successors = problem.getSuccessors(current_node[0])
            for successor in successors:
                # check if the node has been visited or not
                # check if the current depth is within depth_bound
                if (not successor[0] in visited) and (current_node[2]+successor[2] <= depth_
                    # push successor to the stack
                    stack.push(
                        (successor[0], current_node[1] + [successor[1]], current_node[2]+suc
                    # add successor to the visited nodes list
                    visited.append(successor[0])

            # if the goal is not reached within the current depth, increase the depth and br
            if stack.isEmpty():
                break

            current_node = stack.pop()

        # if the current node is the goal, we return the path
        if problem.isGoalState(current_node[0]):
            return current_node[1]

        # the depth_bound increases when a dead end occurs
        depth_bound += 1
```

Pentru **testare**, se pot utiliza **comenzile**:

- python pacman.py -l tinyMaze -p SearchAgent -a fn=ids
- python pacman.py -l mediumMaze -p SearchAgent -a fn=ids -z .5
- python pacman.py -l bigMaze -p SearchAgent -a fn=ids -z .5 item python pacman.py -l openMaze -p SearchAgent -a fn=ids -z .5

4 Bibliografie

Pentru realizarea acestei **documentatii** s-au folosit urmatoarele materiale:

- Indrumatorul de laborator
- Cursurile: *Probleme de cautare*, *Cautare adversariala*