Artificial intelligence - Project 1
- Search problems -

Ilovan Bianca-Maria & Ioja Sorina-Rodica

7/10/2020

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:

```
1  def depthFirstSearch(problem):
2
3      frontier = problem.getStartState();
4      stack = util.Stack();
5      visitedNodes = []
6
7      if problem.isGoalState(frontier):
8          return []
9
10     node = GraphNode(frontier, None, [], 0)
11     stack.push(node)
12
13     while not stack.isEmpty():
14         node = stack.pop()
15
16         if node.getState() not in visitedNodes:
17             visitedNodes.append(node.getState())
18
19             if problem.isGoalState(node.getState()):
20                 return node.getAction()
21
22             for state, action, cost in problem.getSuccessors(node.getState()):
23                 newAction = node.getAction() + [action]
24                 newCost = cost + node.getCost()
25                 newNode = GraphNode(state, node.getState(), newAction, newCost)
26                 stack.push(newNode)
```

Explanation:

- in line 3 we initialize the frontier which with the start game state
- in line 4 we instantiate a stack that will help us keep the nodes we want to explore
- in line 5 we declare an empty list of visited nodes
- next we return an empty list if we are already in a goal state
- in line 10 we instantiate a node from class GraphNode that has as components: state, parent-node, action, cost (from the initial state to the node), then we add it to the stack as it is the starting node

- as long as the stack is not empty we will pop elements out of the stack and check the visited list which avoids expanding any already visited states
- in line 19 we check if the problem is goal state and return the list of actions if true
- in line 22 we use a loop to add adjacent nodes into the stack by adding the current action and cost

Commands:

- *-l tinyMaze -p SearchAgent -a fn=dfs*
- *-l mediumMaze -p SearchAgent -a fn=dfs*
- *-l bigMaze -z .5 -p SearchAgent -a fn=dfs*

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: Yes, the solution we found is optimal but if we talk about dfs as a search algorithm for our pacman problem, it is neither complete nor optimal. Depth-first search always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end does the search go back and expand nodes at shallower levels. The drawback of depth-first search is that it can get stuck going down the wrong path. Many problems have very deep or even infinite search trees, so depth-first search will never be able to recover from an unlucky choice at one of the nodes near the top of the tree. The time complexity for depth-first search is O(bm), where b is branching factor and m the maximum depth. For problems that have very many solutions, depth-first may actually be faster than breadth-first.

Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: Question q1: 3/3

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:

```
1  def breadthFirstSearch(problem):
2
3      startPoint = problem.getStartState()
4      queue = util.Queue()
5      visitedNodes = []
6      node = GraphNode(startPoint, None, [], 0)
7      queue.push(node)
8
```

```
9       while not queue.isEmpty():
10          node = queue.pop()
11
12          if problem.isGoalState(node.getState()):
13              return node.getAction()
14
15          if node.getState() not in visitedNodes:
16              visitedNodes.append(node.getState())
17
18              for state, action, cost in problem.getSuccessors(node.getState()):
19                  newAction = node.getAction() + [action]
20                  newCost = cost + node.getCost()
21                  newNode = GraphNode(state, node.getState(), newAction, newCost)
22                  queue.push(newNode)
```

Explanation:

- in line 3 we initialize the start point with the start game state
- in line 4 we instantiate a queue instead of stack that will help us keep the nodes we want to explore
- in line 5 we declare an empty list of visited nodes
- in line 6 we instantiate a node from class GraphNode that has as components: state, parent-node, action, cost (from the initial state to the node), then we add it to the queue as it is the starting node
- next, as long as the queue is not empty we will dequeue elements out of the queue and check the visited list which avoids expanding any already visited states
- in line 12 we check if the problem is goal state and return the list of actions if true
- in line 18 we use a loop to add adjacent nodes into the queue by adding the current action and cost

Commands:

- *-l tinyMaze -p SearchAgent -a fn=bfs*
- *-l mediumMaze -p SearchAgent -a fn=bfs*
- *-l bigMaze -z .5 -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: BFS is a level by level search. All the nodes in a particular level are expanded before moving on to the next level. Finding the shallowest goal state, may not always be the least-cost solution for a general path cost function. UCS modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe (as measured by the path cost g(n)), rather than the lowest-depth node. It is easy to see that breadth-first search is just uniform cost search with g(n) = DEPTH(«). So, generally speaking, BFS has no guarantees of optimality unless the actions all have the same cost. In our case, BFS has the same solution as UCS because we have the same cost.

Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: Question q2: 3/3

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:

```
1  def uniformCostSearch(problem):
2
3      startPoint = problem.getStartState()
4      ucs = util.PriorityQueue()
5      dictionary = util.Counter()
6      visitedNodes = []
7      node = GraphNode(startPoint, None, [], 0)
8      ucs.push(node, dictionary[str(startPoint[0])])
9
10     while not ucs.isEmpty():
11         node = ucs.pop()
12         if problem.isGoalState(node.getState()):
13             return node.getAction()
14
15         if node.getState() not in visitedNodes:
16             visitedNodes.append(node.getState())
17
18             for state, action, cost in problem.getSuccessors(node.getState()):
19                 newAction = node.getAction() + [action]
20                 newCost = cost + node.getCost()
21                 newNode = GraphNode(state, node.getState(), newAction, newCost)
22                 dictionary[str(state)] = problem.getCostOfActions(node.getAction() + [action])
23                 ucs.push(newNode, dictionary[str(state)])
```

Explanation:

- in line 3 we get the problem start state
- in line 4 we declare a priority queue from util.py because we will give priority to the node that has the minimum cost of the path
- in line 5 we instantiate a counter from util.py which is an extension of the standard python dictionary type. It is specialized to have number values (integers or floats), and includes a handful of additional functions to ease the task of counting data. In particular, all keys are defaulted to have value 0
- in line 6 we declare the list of visited nodes which will be empty for start
- in the next line we instantiate a node from class GraphNode that has as components: state, parent-node, action, cost (from the initial state to the node), then we add it to the priority queue along with the "priority" kept in dictionary
- lines 11-23 will be compiled as long as the priority queue is not empty
- first thing that we have to do is pop the node from the ucs queue, then check if we can stop, in view of the fact that we are in a goal stat

- in line 16 we update the visited nodes list where needed
- in 18-23 lines we check every successor and push it into the queue along with the state which will be the total cost to reach that node, it will help us pop the cheapest node to reach

Commands:

- *-l mediumMaze -p SearchAgent -a fn=ucs*
- *-l mediumDottedMaze -p StayEastSearchAgent*
- *-l mediumScaryMaze -p StayWestSearchAgent*
- *-l mediumMaze -p StayWestSearchAgent*

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: Comparing to DFS algorithm, using the UCS algorithm we get a smaller cost and we expand almost twice more nodes because this search is always expanding the node with the smallest cost while DFS is expanding the deepest unexplored node reason why the solutions are different.

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 StayWestAgent.
A2: The (1/2)**x cost is associated with stepping into the West side of the board because the value of x is decreasing in that way.

Q3: Run autograder *python autograder.py* and write the points for Question 3.
A3: Question q3: 3/3.

1.4 References

- www.youtube.com/watch?v=ec0IJsiIuWkt=304s
- www.youtube.com/watch?v=CJmlP03ik5g
- www.geeksforgeeks.org/

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:

```
1  def aStarSearch(problem, heuristic=nullHeuristic):
2
3      startPoint = problem.getStartState()
4      astar = util.PriorityQueue()
5      dictionary = util.Counter()
6      visitedNodes = []
7      node = GraphNode(startPoint, None, [], 0)
8      dictionary[str(startPoint[0])] += heuristic(startPoint, problem)
9      astar.push(node, dictionary[str(startPoint[0])])
10
11     while not astar.isEmpty():
12         node = astar.pop()
13         if problem.isGoalState(node.getState()):
14             return node.getAction()
15
16         if node.getState() not in visitedNodes:
17             visitedNodes.append(node.getState())
18
19             for state, action, cost in problem.getSuccessors(node.getState()):
20                 newAction = node.getAction() + [action]
21                 newCost = cost + node.getCost()
22                 newNode = GraphNode(state, node.getState(), newAction, newCost)
23                 dictionary[str(state)] = problem.getCostOfActions(node.getAction() + [action]) + heurist
24                 astar.push(newNode, dictionary[str(state)])
```

Listing 1: Solution for the A* algorithm.

Explanation:

- A* aims to find a path to the given goal node having the smallest cost, maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.
- in line 3 we get the problem start state
- in line 4 we declare a priority queue from util.py because we will give priority to the node that has the minimum cost of the path

- in line 5 we instantiate a counter from util.py which is an extension of the standard python dictionary type. It is specialized to have number values (integers or floats), and includes a handful of additional functions to ease the task of counting data. In particular, all keys are defaulted to have value 0
- in line 6 we declare the list of visited nodes which will be empty for start
- in the next line we instantiate a node from class GraphNode that has as components: state, parent-node, action, cost (from the initial state to the node), then we add it to the priority queue along with the "priority" kept in dictionary
- A* selects the path that minimizes f(n)=g(n)+h(n) where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal.
- lines 11-24 will be compiled as long as the priority queue is not empty
- first thing that we have to do is pop the node from the astar queue, then check if we can stop, in view of the fact that we are in a goal state
- in line 17 we update the visited nodes list where needed
- in 19-24 lines we check every successor and push it into the queue along with the state which will be the total cost to reach that node, summing the heuristic value, it will help us pop the cheapest node to reach

Commands:

- *-l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic*
- *-l mediumMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic*
- *-l openMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic*

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: After taking a look in the test cases for UCS and A*, we find the same solution for both of them.

Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2: A* finds the optimal solution slightly faster than UCS. So, the heuristic should help to reduce the number of nodes expanded. For example, if we test on the openMaze layout UCS expands 682 nodes while A* expands 535. Also, on the bigMaze we get 620 for UCS and 549 for A*.

Q3: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).
A3: Question q4: 3/3

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):

    def __init__(self, startingGameState):

        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):

       return self.startingPosition, self.corners

    def isGoalState(self, state):

        pacman, corners = state
        return len(list(corners)) == 0

    def getSuccessors(self, state):

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

            currentPosition, corners = state
            x, y = currentPosition
            dx, dy = Actions.directionToVector(action)

            next = int(x + dx), int(y + dy)
            nextx, nexty = next
            hitsWall = self.walls[nextx][nexty]
            new_corners = set(corners)

            if next in new_corners:
                new_corners.remove(next)

            if not hitsWall:
                successors.append((((nextx, nexty), tuple(new_corners)), action, 1))

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

```
46
47
48      def getCostOfActions(self, actions):
49          if actions == None: return 999999
50          x,y= self.startingPosition
51          for action in actions:
52              dx, dy = Actions.directionToVector(action)
53              x, y = int(x + dx), int(y + dy)
54              if self.walls[x][y]: return 999999
55          return len(actions)
```

Explanation:

- in line 8 we set the corners
- in line 17 the function will return the starting position, but also the corners position
- in line 23 we set the corners and return true if all the corners were explored so the corners list will be empty
- in the 27-42 loop we explore the four possibilities we have to move north, south, east or west. For each one we calculate the next position from the current point and check if we are in a corner to remove it from the list. In addition we look into the hitsWall if false we append the next position into the successors list with the new corners, the action and cost 1.

Commands:

- *-l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem*
- *-l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem*
- *-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: Search nodes expanded: 693

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:

```
1  def cornersHeuristic(state, problem):
2
3      position, corners = state
4      corners_left = list(corners)
5      total_dist = 0
6      next_point = position
7
8      while corners_left:
9          dists = map(lambda c: (util.manhattanDistance(next_point, c), c), corners_left)
10         sorted_dists = sorted(dists)
11         dist, nearest = sorted_dists[0]
12         total_dist += dist
13         corners_left.remove(nearest)
14         next_point = nearest
15
16     return total_dist
```

Explanation:

- in line 3 we initialize the position and corners with the state
- in line 4 we declare the list of the corners left to be visited
- in the next lines we initialize the total distance to the corners which will be 0 in the beginning and the next position which is the current position for now
- in the while loop we use a lambda expression to map the distances from the point to the corners left, then we sort it to get the minimum distance and add it to the total distance. After this we remove the nearest corners from the corners left lift and set it as the next point
- in the end we return the total distance

Commands:

- *-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: On the mediumMaze with AStarAgent the number of extended nodes is 1728, but if we test on mediumCorners layout the number of extended nodes is 693.

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:

```
1  def foodHeuristic(state, problem):
2
3      position, foodGrid = state
4      foodList = foodGrid.asList()
5      problem.heuristicInfo['wallCount'] = problem.walls.count()
6
7      if problem.isGoalState(state):
8          return 0
9
10     distance = []
11     for item in foodList:
12         distance.append(mazeDistance(position, item, problem.startingGameState))
13         # If we have a difficult maze stop search
14         if problem.heuristicInfo['wallCount'] > 17:
15             break
16
17     return max(distance)
```

Explanation:

- in line 3 we initialize the position and food grid with the state
- in line 4 we declare the list of the food, foodGrid.asList() gives us a list of food coordinates
- next, knowing that problem.walls gives us a Grid of where the walls are, we want to count the walls once and store that value
- if we are already in a goal state, we return 0
- in lines 11-14 for each item in our food list we are trying to find real distances between position and all of the food. Knowing that mazeDistance returns the maze distance between any two points, using the search functions already built, we use mazeDistance with the parameters: position being the first point, item, being the second point and the 3rd parameter, the starting game state
- in lines 14-15 if we have a difficult maze with too many walls, we will stop search
- in the end we return maximum distance

Commands:

- *-l testSearch -p AStarFoodSearchAgent*
- *-l testSearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic*
- *-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: Question q7: 4/4. Expanded nodes: 8676

2.5 References

- en.wikipedia.org/wiki/Admissibleheuristic
- www.youtube.com/watch?v=6TsL96NAZCo
- stackoverflow.com/questions/44151713/what-is-the-difference-between-uniform-cost-search-and-best-first-search-methods
- www.cs.utexas.edu/ mooney/cs343/slide-handouts/heuristic-search.4.pdf

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:

```
1  class ReflexAgent(Agent):
2      def getAction(self, gameState):
3
4          legalMoves = gameState.getLegalActions()
5          scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
6          bestScore = max(scores)
7          bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
8          chosenIndex = random.choice(bestIndices) # Pick randomly among the best
9          return legalMoves[chosenIndex]
10
11     def evaluationFunction(self, currentGameState, action):
12
13         successorGameState = currentGameState.generatePacmanSuccessor(action)
14         newPos = successorGameState.getPacmanPosition()
15         newGhostStates = successorGameState.getGhostStates()
16         newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
17
18         newFood = successorGameState.getFood().asList()
19         minFoodist = float("inf")
20         for food in newFood:
21             minFoodist = min(minFoodist, manhattanDistance(newPos, food))
22
23         for ghost in successorGameState.getGhostPositions():
24             if (manhattanDistance(newPos, ghost) < 2):
25                 return -float('inf')
26
27         return successorGameState.getScore() + 1.0 / minFoodist
```

Explanation:

- the main idea of the function is to make reflex agent to play respectably, meaning that he will have to consider both food locations and ghost locations to perform well
- so, as lines 4-16 were already given, in line 18 we want to get the current food as list, because we have to focus on eating food, and if a ghost is close don't go
- in the next line we initialize a minFoodList to infinity, so that it can be used for comparison
- immediately after, we find the minimum distance to each food in the food list

- we use the manhattan distance to the food as the parameter for calculating the evaluation score.
- in lines 23-25 we want to avoid ghost if too close and then return maximum negative infinity(maximum negative score)

Commands:

- *-p ReflexAgent -l testClassic*
- *-p ReflexAgent -l openClassic*
- *–frameTime 0 -p ReflexAgent -k 1 -l mediumClassic*
- *–frameTime 0 -p ReflexAgent -k 2 -l mediumClassic*

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: After 10 consecutive tests, our agent has won 10 times. Average Score: 1259.0

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:

```
1  class MinimaxAgent(MultiAgentSearchAgent):
2
3      def getAction(self, gameState):
4
5          action, score = self.minimax(gameState, 0, 0)
6          return action
7
8      def minimax(self, gameState, depth, agentIndex = True):
9
10         if gameState.getNumAgents() <= agentIndex:
11             agentIndex = 0
12             depth = depth + 1
13
```

```
14          if depth == self.depth or gameState.isWin():
15              return None, self.evaluationFunction(gameState)
16
17          bestScore, bestAction = None, None
18          actions = gameState.getLegalActions(agentIndex)
19          if agentIndex == False:  # If it is max player's (pacman) turn
20              for action in actions:
21                  _, score = self.minimax(gameState.generateSuccessor(agentIndex, action), depth, agentInd
22                  if bestScore is None or bestScore < score:
23                      bestScore = score
24                      bestAction = action
25          else:  # If it is min player's (ghost) turn
26              for action in actions:
27                  _, score = self.minimax(gameState.generateSuccessor(agentIndex, action), depth, agentInd
28                  if bestScore is None or bestScore > score:
29                      bestScore = score
30                      bestAction = action
31
32          if bestScore is None:
33              return None, self.evaluationFunction(gameState)
34          return bestAction, bestScore
35
```

Explanation:

- param gameState refers to the current state of the game, depth being the current depth of the tree and agentIndex the index of the current agent (pacman or ghost)
- the main idea of the algorithm is that for max player (agentIndex=0), the best score is the maximum score among its successor states and for the min player (agentIndex!=0), the best score is the minimum score among its successor states
- recursion ends if depth equals the maximum depth to be searched or there are no successor states available
- in line 5 we call the minimax algorithm with the parameters: gameState, which refers to the current state of the game, 0, the value of the initial depth, and 0 for the agentIndex, it can be 0, being pacman's turn or other value, meaning it's ghost's turn
- in line 10 we check if all agents finished playing their turn in a move increasing current depth
- in line 14 we check if maximum depth is reached or if the game has a winner and then return the result of evaluation function
- in line 17 we initialize best score and best action with None
- now, if it's max player's turn(agent index = 0), for each legal action of pacman we want to obtain the minimax score of successor
- we increase agent index, meaning it's ghost turn
- in line 22, after seeing whether the best score is not updated yet or the current score is better that the best score found so far, we update the best score and the best action
- in a similar way, in the next lines we treat the case of min player's turn(ghost, with agent index != 0)
- we return the result of evaluation function if the state doesn't have any successors state
- function returns the best score and best action for an agent using the minimax algorithm

Commands:

- *-p MinimaxAgent -l minimaxClassic -a depth=4*
- *-p MinimaxAgent -l trappedClassic -a depth=3*

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: After running the game on trappedClassic layout, we can see that pacman is trapped between the wall and 2 ghosts, so he dies anyway, which is why he chooses to go to the nearest ghost to shorten the duration of the game.

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      def getAction(self, gameState):
4          inf = float('inf')
5          action, score = self.alpha_beta(gameState,0, -inf, inf, 0)
6          return action
7
8      def alpha_beta(self, gameState, depth, alpha, beta , agentIndex = True):
9
10         if gameState.getNumAgents() <= agentIndex:
11             agentIndex = 0
12             depth = depth + 1
13
14         if depth == self.depth or gameState.isWin():
15             return None, self.evaluationFunction(gameState)
16
17         bestScore, bestAction = None, None
18         actions = gameState.getLegalActions(agentIndex)
19         if agentIndex == False:  # If it is max player's (pacman) turn
20             for action in actions:
21                 _, score = self.alpha_beta(gameState.generateSuccessor(agentIndex, action), depth, alpha
22                 if bestScore is None or bestScore < score:
23                     bestScore = score
```

```
24                      bestAction = action
25                  alpha = max(alpha, score)
26                  if alpha > beta:
27                      break;
28          else:  # If it is min player's (ghost) turn
29              for action in actions:
30                  _, score = self.alpha_beta(gameState.generateSuccessor(agentIndex, action), depth, alpha
31                  if bestScore is None or bestScore > score:
32                      bestScore = score
33                      bestAction = action
34                  beta = min(beta, score)
35                  if beta <alpha:
36                      break;
37
38          if bestScore is None:
39              return None, self.evaluationFunction(gameState)
40          return bestAction, bestScore
```

Explanation:

- the main idea of the algorithm is that for max player (agent Index = 0), the best score is the maximum score among its successor states and for the min player (agent Index! = 0), the best score is the minimum score among its successor states, but if alpha > beta, we can stop generating further successors and prune the search tree available
- recursion ends if depth equals the maximum depth to be searched or there are no successor state
- in line 5 we call the alpha beta pruning algorithm with the parameters: gameState, which refers to the current state of the game, 0, the value of the initial depth, -inf for alpha, inf for beta and 0 for the agentIndex, it can be 0 and it's pacman turn or other value and that will be a ghost turn
- in line 10 we check if all agents finished playing their turn in a move increasing current depth
- in line 14 we check if maximum depth is reached or if the game has a winner and then return the result of evaluation function
- in line 17 we initialize best score and best action with None
- now, if it's max player's turn aka pacman, for each legal action we want to obtain the alpha$_b etas core of successor, sowecal$
- we take the maximum value for alpha and prune the tree if alpha is greater than beta
- in a similar way, in the next lines we treat the case of min player's turn, this time beta will be updated with the minimum value and the prune the tree if alpha is greater than beta
- we return the result of evaluation function if the state doesn't have any successors state
- function returns the best score and best action for an agent using the alpha beta pruning algorithm

Commands:

- *-p AlphaBetaAgent -a depth=3 -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: Question q3: 5/5

3.4 References

- www.youtube.com/watch?v=l-hh51ncgDI
- www.youtube.com/watch?v=zp3VMe0Jpf8t=8s
- inst.eecs.berkeley.edu/ cs188/fa20/project2/question-1-4-points-reflex-agent
- Artificial Intelligence A Modern Approach [Stuart J. Russell and Peter Norvig]