שם: דניאל ליבנר

ת.ז: 208271778

מטלה 11:

שאלה 1:

:class Node

מחלקה שמממשת יצוג ל- "Node" כך שיהיה אפשרי לדחוף אותם למבנה הנתונים ולהשתמש בהם בצורה נוחה יותר ושהקוד יהיה קריא יותר. מימוש בלי המחלקה הזו היה מאלץ אותנו לבצע העברות tuples רבות וזה היה מקשה מאוד על כתיבת וקריאת הקוד.

הערכים שנשמרים לכל עצם במחלקה הם:

המצב של הצומת.

הצומת שממנה הגענו לצומת הנוכחית, לצורך חישוב הדרך שבוצעה בהגעה אל הייעד.

הפעולה שבוצעה בשביל להגיע לצומת הנוכחית (הפעולה שנלקחה מהצומת הקודמת בשביל שהמצב הנוכחי יהיה המצב הנוכחי).

והקדימות של הצומת בתור הקדימות (כאשר לא ממומש תור קדימות ערך זה ימולא ע"י 0 וכך זה לא ישפיע על המימושים השונים.

לא בוצע מימוש לצומת העוקבת מכיוון שיכולים להיות מספר רב של צמתים עוקבות וכן מכיוון שמימוש של דבר כזה היה מצריך יכולת חישובית לא סבירה, כמובן שחישוב כזה היה מבטל את הצורך במטלה מכיוון שהיה מבטל את החשיבות ביוריסטיקה במימושים השונים כי זה פשוט שם אחר ל- *H מהצומת.

```
def __init__(self, state, predecessor, action, priority=0):
    self.state = state
    self.predecessor = predecessor
    self.action = action
    self.priority = priority
```

---

def firstSearch(problem: SearchProblem, type_of_search: int):

פונקציה שמבצעת את הליבה של חיפוש bfs ו- dfs.

מכיוון ששתי הפונקציות זהות (חוץ ממבנה הנתונים שבו הן משתמשות) החלטתי לממש אותן ביחד תוך שימוש במבנה נתונים אחר בהתאם לסוג החיפוש שמתבצע.

כאשר dfs ממומש עם שימוש במחסנית (והעברת הפרמטר 1 כמזהה לפונקציה)

וכאשר bfs ממומש עם שימוש בתור (והעברת הפרמטר 2 כמזהה לפונקציה)

```
def firstSearch(problem: SearchProblem, type_of_search: int):
    """
    A function that performs the core of bfs and dfs search.
    Since the two functions are the same (except for the data structure they use) I decided to implement them together
        using a different data structure depending on the type of search
    """
    if type_of_search == 1:
        frontier = util.Stack()
    elif type_of_search == 2:
        frontier = util.Queue()
    explored = dict()
    # add Node(state, predecessor, action) to the data structure according his Style(dfs,bfs)
    frontier.push(Node(problem.getStartState(), None, None))
    while not frontier.isEmpty():
        node = frontier.pop()
        if problem.isGoalState(node.state):
            actions = list()
            while node.action is not None:
                actions.append(node.action)
                node = node.predecessor
            actions.reverse()
            return actions
        if node.state not in explored:
            explored.update({node.state: node})
            for successor in problem.getSuccessors(node.state):
                successor_as_node = Node(successor[0],node ,successor[1])
                frontier.push(successor_as_node)
    return list()
```

מכיוון שאנו רוצים את הדרך מהמצב ההתחלתי אל המטרה שהתהפך

המשך שאלה 1:

שם: דניאל ליבנר

ת.ז: 208271775

כמובן שהפונקציה DFS איננה אופטימלית שהרי DFS
מפתח תמיד את הצומת העמוקה ביותר
שלא פותח (הוכחה) לכן במצב בו יש פתרון בתצורה
שהוא עמוק יותר מפתרון אחר שאינו בתצורה
אז נקבל את הפתרון העמוק שבתצורה

```
def depthFirstSearch(problem: SearchProblem):
    """
    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()))
    """
    return firstSearch(problem, 1)
```

סדר הסריקה הוא לא בדיוק מה שציפיתי מבין שסדר הקריאה
בפונקציית לצומת Successors לא הייתה דפוניה כמו ש
כמה יורשים עם עומק שונה. לכן אם כבר היה
בדרך לא נקבל פתרון אופטימלי כי ישנן דרכים רבות
במבוך הpacman לא.

שם: דניאל ליבנר
ת.ז.: 208271775

שאלה 2:

מכיוון שBFS הפעולות יש אותו מחיר (1) פונקציית המחיר היא פונקציה לא יורדת בעומק הצומת שכן הפתרון אופטימלי.

```
def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    return firstSearch(problem, 2)
```

כאמור לפי מבנה הנתונים עבור BFS הוא תור שכן נשלח לפונקציה "firstSearch" המספר 2.

שאלה 3

def uniformCostSearch(problem: SearchProblem):

המימוש של הפונקציה הוא כמו שמובא בספר בעמוד 84 (3.14) תוך שימוש בתור קדימות מהמחלקה util עבור frontier, ושימוש במילון עבור explored וזאת בשביל למזער את הסיבוכיות של בדיקת הימצאות הצומת ב explored.

```
def uniformCostSearch(problem: SearchProblem):
    """
    Search the node of the least total cost first.
    The realization of the function is as presented in the book.
    Using "priority queue" from "util" for frontier,and using "dictionary" for explored and this is to reduce the
        complexity of node exploration in explored.
    """
    frontier = util.PriorityQueue()
    explored = dict()
    # add Node(state, predecessor, action) to the Priority Queue while the priorty is 0
    frontier.push(Node(problem.getStartState(), None, None), 0)
    while not frontier.isEmpty(): #there are more to Node waiting to checked
        node = frontier.pop()
        if problem.isGoalState(node.state):
            actions = list()
            while node.action is not None: #This node is not the initial state.
                actions.append(node.action)
                node = node.predecessor
            actions.reverse()#Since we added the actions to the list from the destination to the beginning(where the direction of
the action is the way to the destination)
            return actions
        if node.state not in explored:
            explored.update({node.state: node}) #Update as explained in the task file. So if the node is not explored - then the
update will insert it. And if it is in explored then the node will be updated to the better priority value.
            for successor in problem.getSuccessors(node.state):#Adds the sucssesors
                successor_as_node = Node(successor[0], node, successor[1], successor[2]+node.priority)
                frontier.push(successor_as_node, successor_as_node.priority)
    return list() #failure
```

שם: דניאל קיובר
ת.ז: 208271775

שאלה 4:

המימוש של A* דומה למימוש של UCS למעט בדרך החישוב של העדיפות (priority).

חישוב העדיפות של A* יהיה לפי: priority = g(n)+h(n)

כאשר: g מתארת את המרחק הנוכחי של הצומת מההתחלה.
h מתארת את ההערכה היוריסטית של המרחק של n מהמטרה.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """
    Search the node that has the lowest combined cost and heuristic first.
    Similar to the uniformCostSearch function except for the changes as presented in the book on page 112 and as
        explained in Dvir's recorded lecture on the site.
    """
    frontier = util.PriorityQueue()
    explored = dict()
    # add Node(state, predecessor, action) to the Priority Queue while the priorty is 0
    frontier.push(Node(problem.getStartState(), None, None, heuristic(problem.getStartState(),
problem)),heuristic(problem.getStartState(), problem))
    while not frontier.isEmpty(): #there are more to Node waiting to checked
        node = frontier.pop()
        if problem.isGoalState(node.state):
            actions = list()
            while node.action is not None: #This node is not the initial state.
                actions.append(node.action)
                node = node.predecessor
            actions.reverse()#Since we added the actions to the list from the destination to the beginning(where the direction of
the action is the way to the destination)
            return actions
        if node.state not in explored:
            explored.update({node.state: node}) #Update as explained in the task file. So if the node is not explored - then the
update will insert it. And if its is in explored then the node will be updated to the better priority value.
            for successor in problem.getSuccessors(node.state): #Adds the sucssesors
                successor_as_node = Node(successor[0], node, successor[1], successor[2]+node.priority)
                frontier.update(successor_as_node, successor_as_node.priority + heuristic(successor[0], problem))
    return list() #failure
```

עדכון הפריוריטי לפי:
הדומה (המרחק מההתחלה - g)
+ ההערכה היוריסטית של מרחק הצומת מהיעד
= ערך העדיפות

הרצה של האלגוריתמים על openmaze מתאימה לבעיה שעשיתי.
בגלל ש-4g את מדריך התלמידה ועל figure 3.15 ויחד עם figure 3.26

	פרמטר	DFS	BFS	UCS	A*
פתרון למטרה	האם נמצא פתרון	✓	✓	✓	✓
	האם הפתרון אופטימלי	✗	✓	✓	✓
	עלות הפתרון	298	54	54	54
	זמן	0.0	0.1	0.1	0.1
הצמתים שפותחו	כמות	576	682	682	535
[illegible]	ציון במשחק	212	456	456	456
דירוג אלגוריתם לפי כל פרמטר	אורך הנתיב	4	2	2	2
	אופטימליות הפתרון	4	2	2	2
	זמן	1	3	3	3
האלגוריתמים	כמות מצבים שפותחו	2	3.5	3.5	1
	ציון במשחק	2	1	1	1
סכום	ציון	13	11.5	11.5	9
	דירוג	4	2.5	2.5	1

מסקנות מההשוואה:

* ישנה חשיבות גדולה להגדרת ה"מטרה" בצורה מדויקת להתאמת התפיסה שלה.
* ישנם מקרים שבהם שילוב של כמה אופציות חיפוש יכול להפיק פונקציה היוריסטית שתביא יותר טובה מאשר שימוש יחיד

שלב ב'

def getStartState(self):

מחזירה את מצב ההתחלה במרחב הבעיה של Corners, לא את מרחב המצב המלא של פקמן (כנדרש במטלה)

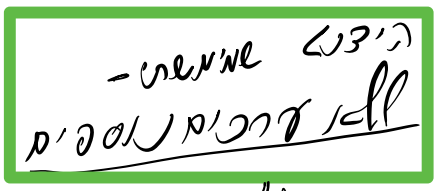
מצב חיפוש בבעיה זו(בעיית פינות) הוא tuple שבו:

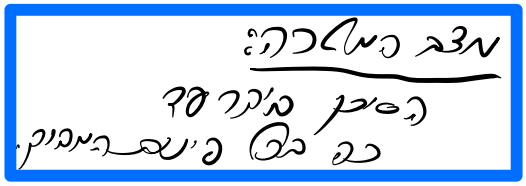
((מצב פינה 4, מצב פינה 3, מצב פינה 2, מצב פינה 1, מיקום פקמן))
((pacmanPosition, corner1status, corner2status, corner3status, corner4status))

pacmanPosition: tuple (x, y) של מספרים שלמים המציינים את מיקומו של Pacman

Corners_Status: אינדיקטורים בוליאנים כאשר אמת פירושו שהסוכן היה בפינה המתאימה ושקר אם לא היה בפינה הנ"ל

```
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: pacman.GameState):
        """
        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
        # As written in the assignment, no more code is required here.

    def getStartState(self):
        """
        Returns the start state in Corners Problem space, not the full Pacman state space (as required in the task)
        A search state in this problem(Corners Problem) is a
            tuple ( pacmanPosition,  corner1status, corner2status, corner3status, corner4status ) where:
                pacmanPosition: a tuple (x,y) of integers specifying Pacman's position
                Corners_Status: a boolean indicator while true means the corner visited in this GameState
        """
        StartState = (self.startingPosition, False, False, False, False)
        return StartState

    def isGoalState(self, state: Any):
        """
        Returns whether this search state is a goal state of the problem.
        Only if all corners have been marked as "visited"
        """
        for cornerStatus in state[1:]:
            if cornerStatus is False: #not visited
                return False
        return True
```





מצב המטרה יהיה כאשר כל 4 האינדיקטורים בקליטה יהיו מצב True

שם: דנעם לייבנר
ת.ז: 208271775

המשך שאלה 5

האינדקסים של הפינות במצב החיפוש גדל ב1 מהאינדקס של הפינה המתאימה ב self.corners וכן ב checkd_bollean.

self.corners = ((1,1), (1,top), (right, 1), (right, top))

A search state = tuple (pacmanPosition, corner1status, corner2status, corner3status, corner4status)

```
def getSuccessors(self, state: Any):
    """
    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
    """

    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]
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]
        if hitsWall is False:
            checkd_bollean = list()
            # checkd_bollean is like the corner_state in places 1-4 - a boolean indicator,
            # while true means the corner visited in this GameState
            for i in range(len(self.corners)):
                if ((nextx, nexty) == self.corners[i]) or state[i + 1]:
                    # if the current position is one of the corners or the corner already checked
                    checkd_bollean.append(True)
                else:
                    # means not visited
                    checkd_bollean.append(False)
            # appending type:(successor((pacmanPosition,corner1status, corner2status, corner3status, corner4status)
            # ,action, stepCost)
            successors.append((((nextx, nexty), checkd_bollean[0], checkd_bollean[1], checkd_bollean[2],
                                checkd_bollean[3]), action, 1))
    self._expanded += 1 # DO NOT CHANGE
    return successors
```

היה ביקור בעבר
או
שכרגע אנחנו בה

שמי: דניאל לייבנר
ת.ז.: 208271773

שאלה 6:

def recursive_path(not_visted_corners, currentPosition, min_path=sys.maxsize):

זוהי פונקציית עזר עבור הפונקציה cornersHeuristic.
מכיוון שמספר הפינות הכולל ה"סטנדרטי" הוא מספר סופי (ולא גדול בכלל), אפשר לפתור בעיה קלה יותר שתהיה עם המרחק האווירי של הפינות שעדיין לא עברו ביקור וללא הגבלות נוספות על האפשרות להגעה אליהן. ומכיוון שזו בעיה נראית במלואה עם סוכן יחיד דטרמניסטית סטטית סדידה וידועה אז אין בעיה לחפש כאן את התוצאה הטובה ביותר ללא שימוש באלגוריתמים מיחדים.

פונקציית העזר:

```
def recursive_path(not_visted_corners, currentPosition, min_path=sys.maxsize):
    """
    This is an help function for the cornersHeuristic function.
    Since the number of corners("standard" corner layout) is a finite number (and not large at all[4]), it is possible
        to solve an easier problem that will be with the Manhattan distance of the corners that have not yet been
        visited and without further restrictions on the possibility of reaching them. And since this is a:
            1.fully observable 2.single agent 3.deterministic 4.episodic 5.static 6.discrete and 7.well-known problem
        then there is no problem to look here for the best result without the use of special algorithms , and because of
        that, this is good heuristic for the real problem.
    And we will find the shortest way from Pacman's current location to all the corners we have not yet visited using
    a direct calculation like in the introduction to cs course.
    """

    if len(not_visted_corners) == 0:
        return 0 # The path of the remaining way is 0 because this is the goal state in this problem
    for corner in not_visted_corners:
        path = util.manhattanDistance(corner, currentPosition)
        if path < min_path: # otherwaise it will not update the min_path
            path = path + recursive_path([c for c in not_visted_corners if c != corner], corner, min_path)
            # souch that it will be the lower bound that can achieved from the state to a goal of the problem
            min_path = min(min_path, path)
    return min_path
```

הפונקציה ההיוריסטית עצמה:

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    """
    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).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
    currentPosition = state[0]
    if state[1:] == (True, True, True, True):  # The path of the remaining way is 0 because this is the goal state in this
problem
        return 0
    not_visted_corners = list()
    for i in range(1, 5, 1):
        if not state[i]: # mean the boolean indicator is false
            not_visted_corners.append(corners[i - 1]) # corners are in 0-3 index and corner_state(inside State) are in 1-4
index's
    return recursive_path(not_visted_corners, currentPosition)
```

מכיוון שכפי שמסביר בלמידה (עמ' 47) מרחק אווירי הוא עקבי. ומכיוון שגם המרחק האווירי היא לא הבעיה שהרי בעולם אמיתי בכלל לא אפשר ללכת בקו אווירי מנקודה לנקודה כי יש קירות וכו' ולכן בהכרח מתקיים: $h(n) \le h^*(n)$
ולכן הפונקציה היוריסטית פה היא קבילה ועקבית

שם: דניאל לייבנר
ת.ז: 208271778

שאלה 7:

הפונקציה ההיוריסטית שמשתמשת היא (בדומה לשאלה הקודמת) עבור בעיה מרוככת כאשר הבעיה המרוככת היא הדרך בין
בשני פתרונות(פקמן) מיקומו הנוכחי שמרחק מנהטן ביניהם הוא המקסימלי
אז שתי הנקודות שנשיג אליהן את המרחק מנהטן המינימלי מבין: (מיקום נוכחי)←A(מיקום נקודה)≤B יהיה גדול מה מרחק
אחר שירתנה הפונקציה ההיוריסטית ונגדיל מה מרחק בין 2 נקודות אוכל

שכן כמו בשאלה 6 הפונקציה הזו היא עקבית מכיוון שמדובר במרחק אוקלידי (שמו 48 המעבדים) וכן היא גם קבילה מכיוון שזה הפתרון יהיה גדול לגבי את המסלול הישיר בין 2 הנקודות(נקודות)
והגעה אליהן תהיה מהמיקום הנוכחי שהpacman נמצא שבה
נבנה: $f(n)=h(n)+g(n)\le f^*(n)$ ו- $h(n)\le h^*(n)$

אם יש נקודת אוכל אחת בלבד שנשארה אז נבדוק מרחק מנהטן מהמיקום הנוכחי אל הנקודה הזו וזו תהיה ההיוריסטיקה.
אם לא נשארו נקודות אוכל-משמע הגענו למטרה ולכן ההיוריסטיקה תהיה 0.

קוד הפונקציה:

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    """
    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
    foodList = foodGrid.asList()
    heuristic = 0
    if len(foodList) == 0: # no more food
        heuristic = 0 # 0 in default
    elif len(foodList) == 1:
        food = foodList.pop(0)
        heuristic = util.manhattanDistance(position, food)
    elif len(foodList) >= 2:
        for firstFood in foodList:
            for secondFood in foodList:
                V12 = util.manhattanDistance(firstFood, secondFood)
                 # The huristic using the relaxed problam - the manhattan distance between two food's that f(n)=h(n)+g(n)<=f*(n)
                if V12 >= heuristic:
                    Vp1 = util.manhattanDistance(position, firstFood)
                    Vp2 = util.manhattanDistance(position, secondFood)
                    heuristic = V12 + min(Vp1, Vp2)
    # using two heuristics ways. for case of non-straight grid for the game. see readme for deeper explanation.
    return max( heuristic, len(foodList))
```

אם אין נקודות אוכל

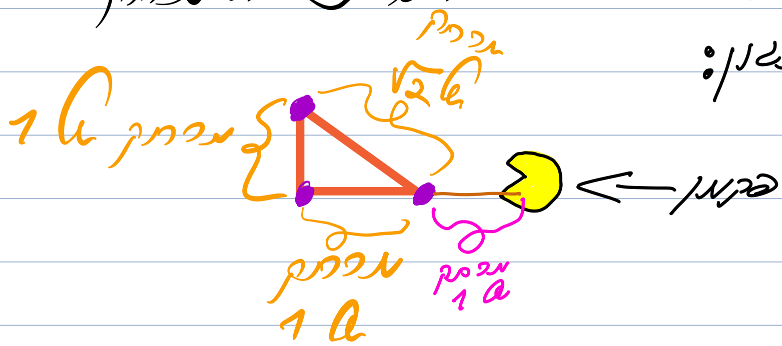
אם יש נקודת אוכל יחידה

אם יש לפחות 2 נקודות אוכל

המשחק שלב 7.

שם: דניאל לייבנר
ת.ז: 208271775

המקרה שבו יש רשת המשחק בצורה "שרה"
למשל מצב כגון:



למשל שהיוריסטיקה שלקחה את המרחק בין כל הנקודות
הכתומות בתוספת המרחק בין פקמן לקרובה שבהן
יהיה קטן מהאמיתי הנקודות שצריך לחשב אז נבדה
לקחת אחת כמות הנקודות כפולה בהיוריסטיקה
מצב כמו שלא בשני הסדר הבריאה הנקודות
המשלב משלב המרחקים. כל מקרה כזה שיתב לו
היא יתקבל נכונה מבחינת שלא הפתרון צריך
לחכות צעדים כמספר הנקודות שיש לאכול.

תשובה:
הפונקציה ההיוריסטית שלי מפתחת 7924 צמתים עבור
trickyPath כאשר המסלול שנמצא הוא באורך 60 ולכן
נשאר ניקוד 27 ששירות ונדיין שהכותב מקבל עבור
trickyPath הוא 570

שם: דניאל ליבנר

ת.ז: 208271775

שאלה 8:

בהתאם להגדרה של המימוש השלמתי את הפונקציה isGoalState כאשר מצב מטרה הוא קיום של נקודת אוכל בנקודה "State"

```python
def isGoalState(self, state: Tuple[int, int]):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x, y = state
    foodGrid = self.food;
    foodList = foodGrid.asList()
    for food in foodList: # finding a path to any food
        fx, fy = food
        if fx == x and fy == y:
            return True
    return False
```

ואז בהתאם המימוש של findPathToClosestDot
יתבצע בקלות ע"י שימוש ב- "UCS" שימצא פתרון
אופטימלי עם מחיר מינימלי.
במימושים של בנוסף שימוש ב-BFS מבטיח שגם
המחלקה שמתחילה הפונקציה יושבת בתוך
"ClosestDotSearchAgent"
ולכן לא של לממש אגב הפעלת BFS
כי נכון שבמצב של יש מחיר קבוע
לכל פעולה אבל לא זה המצב ב pacman
כלו שכן שיכול להיות יקר זה כאילו ישבנו חישוב
הפעלת UCS והגדלת את סביבת המקום
והזמן ריצה $O(b^d)$ לפי ההגדרה בשיעור 3 שבריי מספר
הצעדים בראש הגאו קבוע לכל צעד

סיום הפונקציה!

```python
def findPathToClosestDot(self, gameState: pacman.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)
    # we already solve souch type of problem - using the uniform cost search we made in search.py
    return search.uniformCostSearch(problem)
```