به نام خدا

درس : مبانی و کاربردهای هوش مصنوعی

استاد: جوانمردی

دانشجو: مهدی منصوری خواه

شماره دانشجویی: 9931056

گزارش پروژه – اول(پکمن)

کلاس SearchProblem

این کلاس طرح کلی مسئله جست و جو را نشان میدهد.

مسئله جست و جو شامل : فضای حالت – تابع پسین – حالت شروع و آزمون هدف

```
def getStartState(self)
```

این تابع حالت شروع را برمیگرداند

```
def isGoalState(self, state)
```

این تابع چک میکند که آیا در حالت (state) هدف هستیم یا خیر

```
getSuccessors(self, state)
```

این تابع وظیفه دارد که با گرفتن وضعیت فعلی، مقادیر action وSuccessor. stepcost را برگرداند و همچنین باید مشخص کند از این حالت با action مربوطه به کدام حالت می رویم.

```
def getCostOfActions(self, actions)
```

این روش هزینه کل اقدامات را برمی گرداند

کلاس Agent

به طور کلی این کلاس از سایر کلاس ها مثل pacman.py اطلاعاتی رو دریافت می کند و عملیات مورد نیاز (stop,south,east,...) را انجام می دهد

کلاس Directions

این کلاس به طور کلی جهت را مشخص می کند و همچنین کار revers هم انجام می دهد یعنی مثلا میتواند جهت شمال را به جنوب و جهت غرب را به شرق تبدیل کند و بلعکس.

کلاس Configuration

مختصات (x,y) یک کاراکتر را به همراه جهت حرکت آن نگه می‌دارد.

کلاس AgentState

وضعیت عامل را نگه میدارد(configuration, speed, scared, etc)

کلاس Grid

موقعیت عامل ما را به صورت یک آرایه نگه میدارد و نقطه (0و0) گوشه پایین سمت چپ است.

متود update از کلاس PriorityQueue

```python
def update(self, item, priority):
    # If item already in priority queue with higher priority, update its priority and rebuild the heap.
    # If item already in priority queue with equal or lower priority, do nothing.
    # If item not in priority queue, do the same thing as self.push.
    for index, (p, c, i) in enumerate(self.heap):
        if i == item:
            if p <= priority:
                break
            del self.heap[index]
            self.heap.append((priority, c, item))
            heapq.heapify(self.heap)
            break
    else:
        self.push(item, priority)
```

این متد صف را با توجه به اولویت آیتم مرتب می کند اگر آیتم در صف وجود نداشت با توجه به اولویتش پوش میکند اگر مورد قبلا در صف اولویت بوده باشه و در حال حاضر اولویتش بالاتر رفته باشد در این صورت دوباره صف بروزرسانی می شود و اگر مورد از قبل در صف اولویت با اولویت مساوی یا کمتر وجود داشت، کاری انجام نمی دهد.

بروزرسانی صف با استفاده از heapify انجام میشود.

پیدا کردن یک نقطه ثابت غذا با استفاده از جستجوی اول عمق (DFS)

```python
def depthFirstSearch(problem):
    open_stack = util.Stack()
    open_stack.push((problem.getStartState(),'origin',0))
    closed_stack = util.Stack()
    backtrack_checkpoints = util.Stack()
    visited_list = []
    while not open_stack.isEmpty():
        X = open_stack.pop()
        visited_list.append(X[0])
        if problem.isGoalState(X[0]):
            closed_stack.push(X)
            break
        else:
            children_of_X = problem.getSuccessors(X[0])
            closed_stack.push(X)
            alreadyVisitedChildren = 0
            for each_child in children_of_X:
                if (each_child[0] in visited_list):
                    alreadyVisitedChildren +=1
                else:
                    open_stack.push(each_child)
            if (len(children_of_X) - alreadyVisitedChildren) > 1:
                backtrack_checkpoints.push(X)
                if len(children_of_X) == 4:
                    backtrack_checkpoints.push(X)
```

```
        if alreadyVisitedChildren == len(children_of_X):
            return_point = backtrack_checkpoints.pop()
            temp_point = closed_stack.pop()
            while return_point[0] != temp_point[0]:
                temp_point = closed_stack.pop()
            closed_stack.push(temp_point)
    actions = []
    while not closed_stack.isEmpty():
        dir = closed_stack.pop()[1]
        if dir!='origin':
            actions.append(dir)
    actions.reverse()
    return actions
    # util.raiseNotDefined()
```

DFS یک جستوجوی عمقی هست و یک شاخه رو دنبال میکند تا به جواب برسد و اگر به جواب نرسد شاخه را برمیگردد و یک شاخه دیگر را دنبال می کند تا به جواب برسد در اینجا تا زمانی که استک خالی نشده باشد یک نود برمیدارد و آن را به لیست نودهای اضافه شده (visited_list) اضافه می کند سپس در else با استفاده از getSuccessors فرزندهایش رو اضافه می کند.
این فرایند تا جایی ادامه دارد(isGoalState) تا به جواب برسیم وبا دستور breake از while بیرون می آییم.
در آرایه action مسیر هدف ذخیره می شود سپس action را برعکس میکنیم (actions.reverse()) تا مسیر از ابتدا تا انتها مرتب شود.

سوال: در غالب یک شبه کد مختصر الگوریتم IDS را توضیح دهید و تغییرات لازم برای تبدیل الگوریتم DFS به IDS را نام ببرید.

```
// Returns true if target is reachable from
// src within max_depth
bool IDDFS(src, target, max_depth)
    for limit from 0 to max_depth
       if DLS(src, target, limit) == true
           return true
    return false

bool DLS(src, target, limit)
    if (src == target)
        return true;

    // If reached the maximum depth,
    // stop recursing.
    if (limit <= 0)
        return false;

    foreach adjacent i of src
```

```
        if DLS(i, target, limit?1)
            return true

    return false
```

الگوریتم جست و جوی عمیق تکراریIterative DFS Search Deepening ، یک ترکیبی از الگوریتم های DFS و BFSاست. این الگوریتم برمبنای جستوجوی عمق اول اما با رویکرد محدودیت در عمق کار میکند. در ابتدا یک عمق را مشخص میکند و تا آن عمق با استفاده از الگوریتم DFS جستوجو را انجام میدهد و این جست وجو مزیت فضای DFS را با مزایای زمان/ راه حل کم عمق BFS همزمان دارد.

برای تبدیل الگوریتم DFS به IDS کافی است که برای dfs یک محدودیت در درخت جست وجو قرار دهیم و اجازه ندیم که تا آخرین برگ را پیمایش کند و تا سطحی از درخت را پیمایش کند و این کار را با یک عمق شروع کنیم و با هر بار جست و جو مقدار depth_max را اضافه می کنیم.

جستجوی اول سطح (BFS)

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    open_queue = util.Queue()
    open_queue.push((problem.getStartState(), []))
    visited_list = []
    while not open_queue.isEmpty():
        X, actions = open_queue.pop()
        if X not in visited_list:
            visited_list.append(X)
            if problem.isGoalState(X):
                return actions
            else:
                children_of_X = problem.getSuccessors(X)
                for each_child in children_of_X:
                    open_queue.push((each_child[0], actions + [each_child[1]]))
    return []
```

BFS به صورت سطحی پیمایش می کند یعنی ابتدا زمانی که نودی رو پیمایش میکند فرزند های آن را به فرینج اضافه میکند سپس سراغ همسایه خود میرود و همینجور ادامه دارد در اینجا هم به همین صورت است ابتدا نودهایی که ویزیت نشده اند را ویزیت می کنیم و آن را به آرایه visited_list اضافه می کنیم سپس فرزند های یک نود را با استفاده از getSuccessors پیدا می کنیم و یک for میزنیم و آن را به صف اضافه می کنیم این روند تا زمانی ادامه دارد که به هدف برسیم(isGoalState)

سوال: الگوریتم BBFS را به صورت مختصر با نوشتن یک شبه کد ساده توضیح دهید و آن را با الگوریتم BFSمقایسه کنید. همچنین ایدهای بدهید که در یک مسئله جستجو که به دنبال بیش از یک هدف هستیم چگونه میتوانیم از BBFS استفاده کنیم؟

این الگوریتم در واقع bfs دوطرفه است که یکی از کاربرهاش حل مسئله 8 پازل است و در این مورد بهینه تر از جست وجوی سطح اول(bfs) کار میکند. BBFS برای زمانی خوب است که امکان شروع حل مسئله از 2 جهت را بخواهیم مثلا مسئله 8 پازل که برای یک خانه خالی، 2 تا 4 گزینه برای حرکت(برای تغییر وضعیت)وجود دارد.

این الگوریتم با رویکرد روبه جلو و روبه عقب میتواند کار کند. اگر دوشاخه ای از گراف که بررسی می شود دارای نقطه اشتراک باشند و بهم برخورد کنند بررسی گراف متوقف میشود.

و برای تغییر میتوانیم مثلا دو صف و دو آرایه برای ویزت شدن نود ها در نظر بگیریم که هر کدام مسیر جداگانه رو طی می کنند تا به هدف برسند.

```
struct Graph {
   int V;
   LinkedList<Integer>[] adj;
}
void addEdge(int u, int v)
{
   adj[u].add(v);
   adj[v].add(u);
}

void BFS();

Boolean is Intersecting(Boolean[] s_visited, Boolean[] t_visited)
{
   for (int i = 0; i < V; i++) {
   if (s_visited[i] && t_visited[i])
   return i;
}
   return -1;
```

```
}
Boolean biDirSearch(int s, int t)
{
   Boolean[] s_visited = new Boolean[V];
   Boolean[] t_visited = new Boolean[V];
   int[] s_parent = new int[V];
   int[] t_parent = new int[V];
   Queue<Integer> s_queue = new LinkedList<Integer>();
   Queue<Integer> t_queue = new LinkedList<Integer>();
   int intersectNode = -1;
   for (int i = 0; i < V; i++) {
   s_visited[i] = false;
   t_visited[i] = false;
   }
   s_queue.add(s);
   s_visited[s] = true;
   s_parent[s] = -1;
   t_queue.add(t);
   t_visited[t] = true;
   t_parent[t] = -1;
   while (!s_queue.isEmpty() && !t_queue.isEmpty()) {
   bfs(s_queue, s_visited, s_parent);
   bfs(t_queue, t_visited, t_parent);
   intersectNode = isIntersecting(s_visited, t_visited);
   if (intersectNode != False) {
   System.out.printf("Path exist between %d and %d\n", s, t);
   System.out.printf("Intersection at: %d\n", intersectNode);
   printPath();
   return True;
```

}

}

return False;

}

}

تغییر تابع هزینه

```python
def uniformCostSearch(problem):
    open_queue = util.PriorityQueue()
    open_queue.push((problem.getStartState(), [], 0), 0)
    visited_set = set()
    closed_list = []
    while not open_queue.isEmpty():
        X, actions, cost = open_queue.pop()
        visited_set.add(X)
        if problem.isGoalState(X):
            return actions
        else:
            # generate successors of X
            if (X not in closed_list):
                children_of_X = problem.getSuccessors(X)
                closed_list.append(X)

                for each_child in children_of_X:
                    if (each_child[0] in visited_set):
                        pass
                    else:
                        open_queue.update((each_child[0], actions + [each_child[1]], each_child[2] + cost), each_child[2] + cost)
    return []
```

الگوریتم USC میاد ارزان ترین شاخه را گسترش میدهد USC یک صف اولویت دارد که اولویت آن بر حسب هزینه تجمعی هست و هر بار هزینه ها را حساب میکند و صف را آپدیت می کند.USC به نوعی BFS می باشد با این تفاوت که در BFS هزینه هر یال یک است برای همین کد هر دو شبیه به هم است. در این کد هزینه هر یال (cost در کد بالا) در نظر گرفته می شود و با بقیه مراحل قبل جمع زده می شود و هر بار صف آپدیت میشود.

سوال: آیا ممکن است که با مشخص کردن یک تابع هزینه مشخص برای الگوریتم UCS ،به الگور یتم BFS و یا DFS برسیم؟ در صورت امکان برای هرکدام از الگور یتمهای BFS و یا DFS ،تابع هزینه مشخص شده را با تغییر کد خود توضیح دهید.

اگر که هزینه هر انتقال برابر با 1 باشد، میتوان گفت که UCS شکل دیگری از BFS است ومتوان از UCS به BFS برسیم. در قطعه کد بالا آنجایی که cost هست را برابر یک قرار می دهیم.

اما نمی توان از USC با تغییر تابع هزینه به DFS برسیم زیرا DFS به عمق نگاه می کند و به هزینه اصلا کاری ندارد.

جستجوی A استار(A*)

```python
def manhattanHeuristic(position, problem, info={}):
    "The Manhattan distance heuristic for a PositionSearchProblem"

    "*** YOUR CODE HERE ***"
    xy1 = position
    xy2 = problem.goal
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])
```

```python
def euclideanHeuristic(position, problem, info={}):
    "The Euclidean distance heuristic for a PositionSearchProblem"

    "*** YOUR CODE HERE ***"
    xy1 = position
    xy2 = problem.goal
    return ((xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2) ** 0.5
```

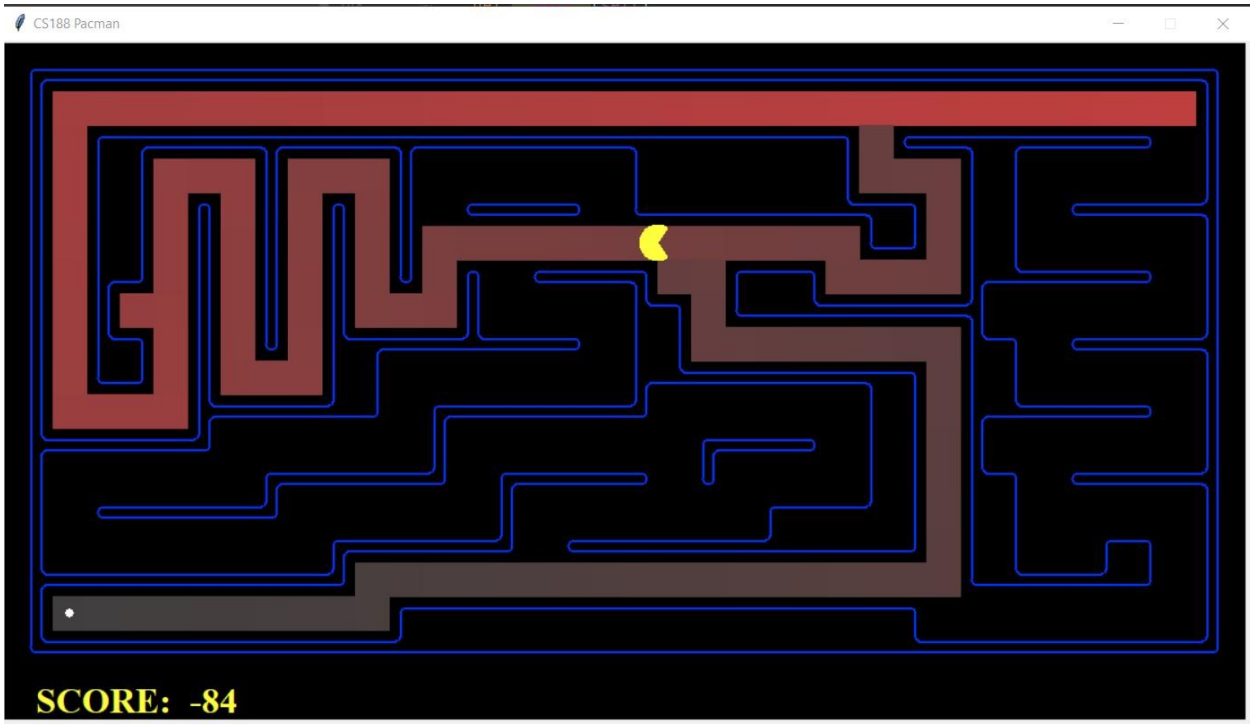
```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    # define a stack for open list
    open_queue = util.PriorityQueue()
    # initialize open stack with start position
    open_queue.push((problem.getStartState(), [], 0), 0)
    visited_set = []
    closed_list = []
    while not open_queue.isEmpty():
        X, actions, cost = open_queue.pop()
        visited_set.append(X)
        if problem.isGoalState(X):
            return actions
        else:
            # generate successors of X
            if (X not in closed_list):
                children_of_X = problem.getSuccessors(X)
                closed_list.append(X)

                for each_child in children_of_X:
                    if (each_child[0] in visited_set):
                        pass
                    else:
                        heuristic_value = heuristic(each_child[0], problem)
                        open_queue.update((each_child[0], actions + [each_child[1]], each_child[2] + cost),
                                          each_child[2] + cost + heuristic_value)
    return []
```

در A استار داریم:

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

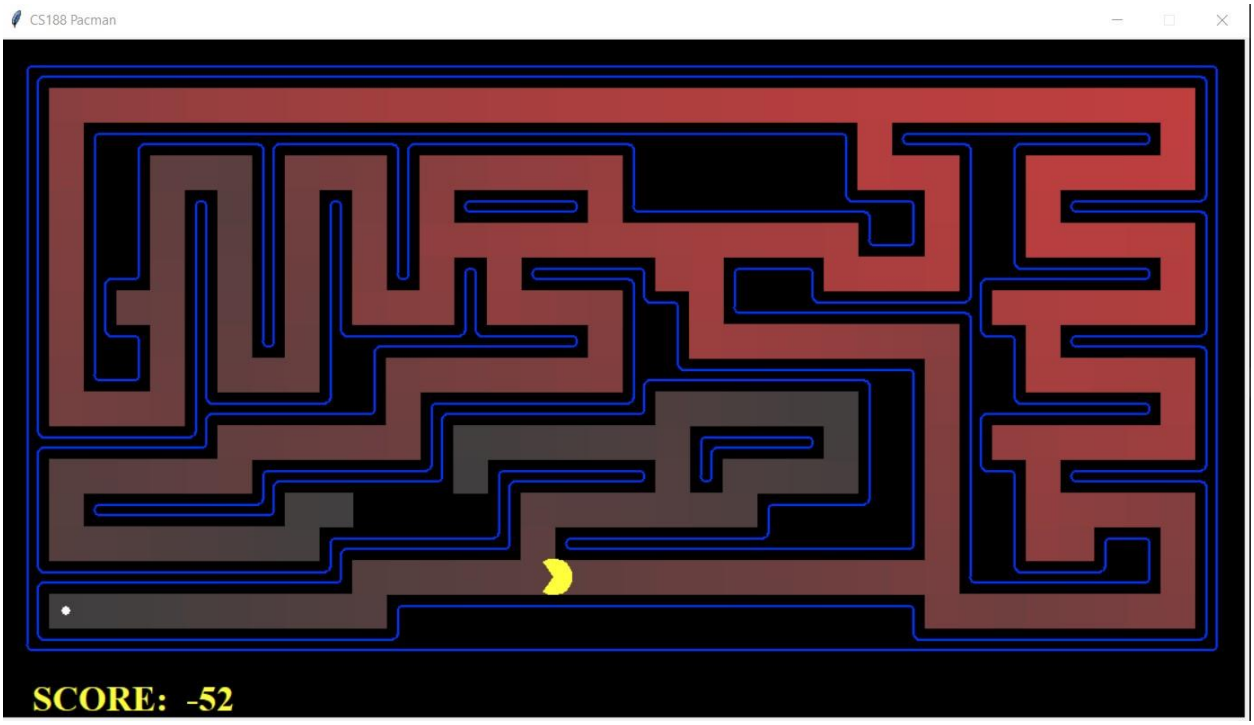
یعنی A استار ترکیبی از الگوریتم حریصانه و USC است و هم هیرویستیک و هم هزینه مسیر رو در نظر می گیرد و هر کدام که حاصل جمع کمتری داشت آن را ادامه میدهد. کد A استار شبیه به USC است با این تفاوت که در اینجا هم هزینه جمع شده می شود و هم هیروستیک و در آخر صف آپدیت می شود.(قسمت بالای کد در قسمت های قبلی توضیح داده شده است برای همین قسمت جدید هر قسمت توضیح داده شده است.)

سوال: الگوریتمهای جستجویی که تا به این مرحله پیاده سازی کرده اید را روی openMaze اجرا کنید و توضیح دهید چه اتفاقی میافتد (تفاوتها را شرح دهید)



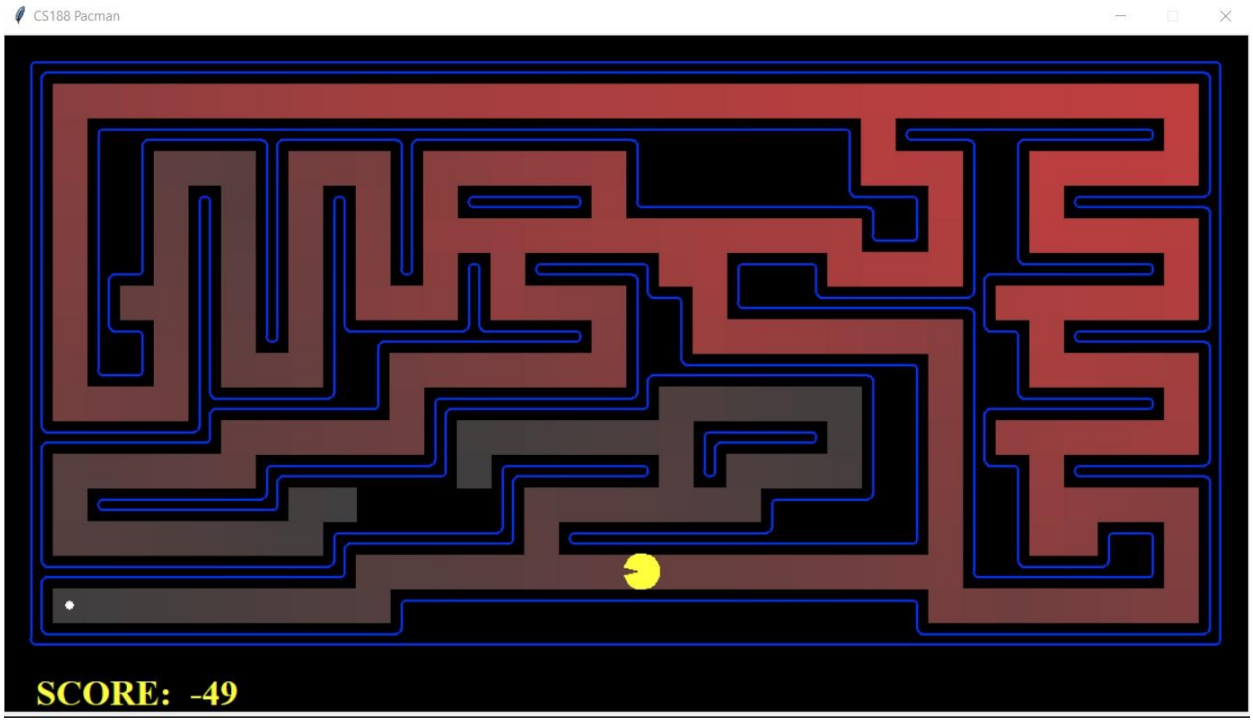

```
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```

در جست و جوی dfs به این صورت است که به صورت عمقی جستوجو می کند و در قسمتی از مسیر که یک حالت بن بست داریم بدون در نظرگرفت آن سرچ را انجام میدهد که سر بار بالایی دارد. در پیداکردن مسیر و طی کردن آن بسیار بد عمل می کند و فقط عمق را نگاه میکند تمام سطر هارا از چپ به راست عمقشان را طی می کند. در نتیجه dfs در اینجا کاربرد خوبی ندارد.



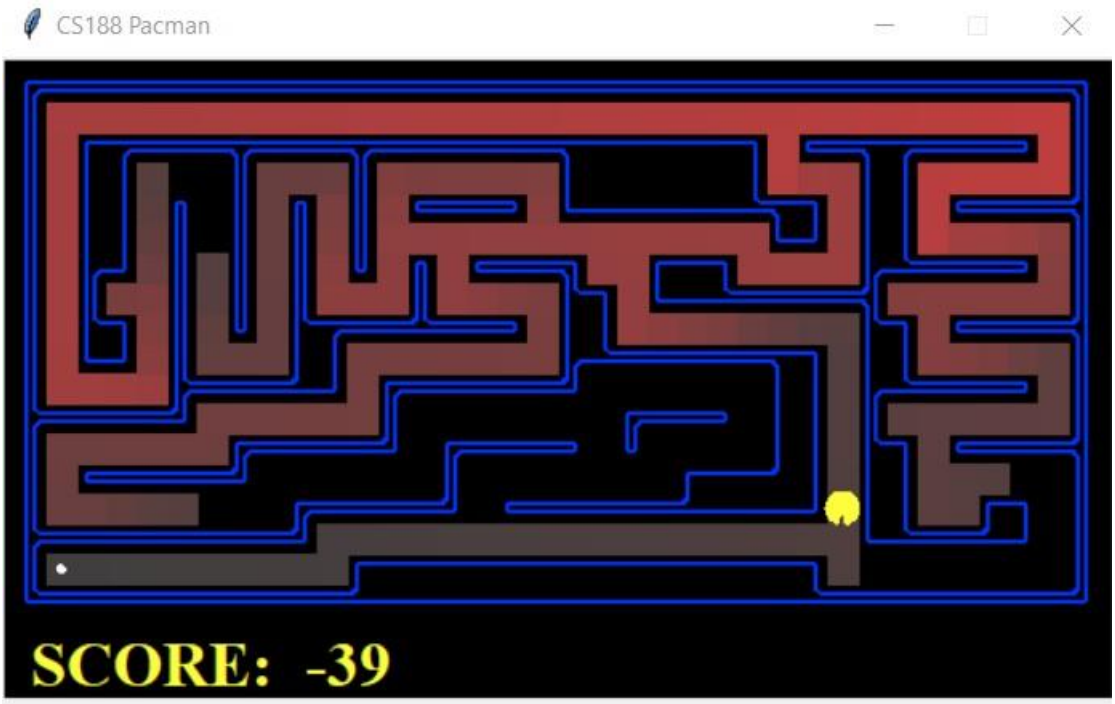

```
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

در bfs که هم زمان حرکت پکمن صرف سرچ کردن و پیداکردن مسیر زیاد است چون فاصله هدف تامبدا طبق فاصله منهتن تقریبا بیشترین فاصله بیشترین فاصله ممکن است بنابراین حتی زمان سرچ کردن آن از dfs بیشتر است چون حالت های بیشتری را بررسی میکند اما مسیر بهتری از dfs را پیدا میکند برای همین امتیاز بالاتری نسبت به dfs دارد.




```
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

چون در اینجا هزینه رسیدن به هر یک از خانه های اطراف با هم برابر بوده است، در نتیجه الگوریتم ucsهمانند bfs عمل کرده است و دقیقا نتایج مشابیه بدست آمده است.




```
Search nodes expanded: 221
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

در اینجا هم چون heuristic ما فاصله منهتن تا هدف بوده است پس در هر مرحله چیزی که در اولویت قرار میدهد خانه ای است که مجموع هزینه رسیدن به آن و مقدار فاصله آن تا هدف کمتر است در اینجا قسمت های پایین سمت راست از هدف دور هستند پس مقدار هیرویستیک آن ها بیشتر است و f(n) افزایش پیدا میکند برای همین این مسیر ها بعدا بررسی میشوند البته در اینجا زودتر به جواب رسیدیم و دیگر آن ها بررسی نشده اند.

پیدا کردن همه گوشه ها

```
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 ***"
        corners_visited = [False, False, False, False]
        return (self.startingPosition, corners_visited)

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        if False in state[1]:
            return False
        else:
            return True

    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
        """

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

```
            goal_visited_list = state[1]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            temp_goal_visited_list = goal_visited_list.copy()
            currentPosition = (nextx, nexty)
            if not hitsWall:
                if currentPosition in self.corners:
                    cornerIndex = self.corners.index(currentPosition)
                    temp_goal_visited_list[cornerIndex] = True
                    successorState = ((currentPosition,
temp_goal_visited_list), action, 1)
                else:
                    successorState = ((currentPosition,
temp_goal_visited_list), action, 1)

                successors.append(successorState)

        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)
```

در اینجا ابتدا مقدار گوشه ها را False میکنیم و هر موقع آن را ویزیت کردیم true میکنیم. هر دفعه با بررسی دیوار ها و عملیات action فاصله پکمن تا گوشه ها را بدست می آوریم و بعلاوه موقعیت فعلی پکمن میکنیم.

هیوریستیک برای مسئله گوشه ها

```python
def mazeDistance(point1, point2, gameState):
    """
    Returns the maze distance between any two points, using the search functions
    you have already built. The gameState can be any game state -- Pacman's
    position in that state is ignored.

    Example usage: mazeDistance( (2,4), (5,6), gameState)

    This might be a useful helper function for your ApproximateSearchAgent.
    """
    x1, y1 = point1
    x2, y2 = point2
    walls = gameState.getWalls()
    assert not walls[x1][y1], 'point1 is a wall: ' + str(point1)
    assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
    prob = PositionSearchProblem(gameState, start=point1, goal=point2, warn=False, visualize=False)
    return len(search.bfs(prob))
```

در این بخش از heuristic ای به نام distance maz استفاده شده است که در py.SearchAgents پیاده سازی شده است. در هر نقطه ای که هستیم فاصله با دورترین گوشه از مازکه تاکنون نقطه که تاکنون explore نشده است را برابر h آن نقطه در نظر می گیریم.(از bfs برای پیدا کردن مقدار بهینه آن استفاده شده است)

از این نظر قابل قبول است که با طی کردن فاصله ای به اندازه مسیر از پوزیشن فعلی به دورترین راس، به تمامی راس ها سر بزنیم و آنها را رصد کنیم، مگر اینکه تنها یک راس دیده نشده داشته باشیم و یا سایر راس ها در مسیر دورترین راس باشند .در این حالت پیشبینی ما دقیقاً همان بیشینه فاصله است. پس بدین صورت اثبات میشود که هیوریستک قابل قبول است.

درباره سازگار بودن هم اینگونه اثبات میکنم که اگر به سمت دورترین راس حرکت نکنیم، و به سراغ راس دیگری برویم، در اینصورت این دور ترین راس تأثیر زیادی بر روی نا دقیق بودن تابع هیوریستیک ما خواهد داشت. اگر به سمت راس دورتر، که از اول تا آخر باید سراغش برویم حرکت کنیم، تابع هیورستیک ما قطعاً در مرحله بعدی عددی بزرگتر از مقدار هیورستیک فعلی نمیدهد حچرا که ما در هر حرکت تنها یک واحد از رئوس دور یا نزدیک میشویم .

پس با این حساب اگر دوباره مقدار هیوریستیک راس قبلی که دورترین فاصله را داشت انتخاب شود، مقدار هیورستیک جدید قطعاً کمتر خواهد شد.

اما اگر راس دیگری این بار به عنوان دورترین راس انتخاب شود، باز هم نگرانی نداریم چرا که صرفاً یک واحد نسبت به قبلی حرکت کردهایم و از دورترین راس فعلی دور یا نزدیک شدیم. اگر نزدیک شده باشیم که قطعاً مقدار هیوریستیک ما کمتر میشود.

اما اگر دور شده باشیم نیز مقدار هیوریستیک فعلی حداکثر با هیوریستیک قبلی برابر میشود و باز هم افزایش مقدار هیوریستیک را نداریم.

بدین صورت در هیچ یک از حالت ها، مقدار تابع هیوریستیک کاهش پیدا نمیکند. پس سازگار است.

خوردن همه غذاها

```python
"*** YOUR CODE HERE ***"
position, foodGrid = state
food_list = foodGrid.asList()
from util import manhattanDistance
if not len(food_list):
    return 0
distances = []
for food_position in food_list[:2]:
    # distances.append(manhattanDistance(position, food_position))
    distances.append(mazeDistance(position, food_position, problem.startingGameState))
return max(distances)
```

سوال:هیوریستیک خود را توضیح دهید و سازگاری آن را استدلال کنید.

سوال: پیاده ساز ی هیوریستیک خودتان در این بخش و در بخش قبلی را با یکدیگر مقایسه و تفاوتها را بیان کنید.

در اینجا استدلالی که برای پیدا کردن گوشه ها در قسمت قبل کردیم را اینجا هم استفاده می کنیم با این تفاوت که اهداف ما در این مساله به جای آنکه گوشه های ماز باشد، نقطه های درون ماز است.

بدین صورت که با استفاده از متدmazeDistance این فاصله دقیق که شامل رد نشد از دیوار ها و غیره را میشود را حساب میکنم .از آنجایی که این هزینه نسبتاً واقعی است، پس تابع هیوریستیک من خیلی به مقدار واقعی هزینه نزدیک میشود و تعداد نود های بسیار کمی اکسپند میکند و باعث میشود تابع هیوریستیک من قابل قبول و سازگار شود و تقریب بسیار خوبی از پوزیشن فعلی تا هدف شود .برای اینکه در زمان اجرای محاسبه فاصله ها صرفه جویی کنم، صرفاً فاصله مازی دو تا از غذا هارا از موقعیت فعلی پکمن حساب کردم.

جستجوی نیمه بهینه

الگوریتم IDS در کدم موجود هست در findPathToClosestDot ولی نتونستم کامل دیباگش کنم برای همین از BFS استفاده کردم.

```
def isGoalState(self, state):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x,y = state

    "*** YOUR CODE HERE ***"
    if state in self.food.asList():
        return True
    return False
```

```
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 ***"
    return search.breadthFirstSearch(problem)
```

سوال : ClosestDotSearchAgent شما، همیشه کوتاهترین مسیر ممکن در مارپیچ را پیدا نخواهد کرد. مطمئن شوید که دلیل آن را درک کرده اید و سعی کنید یک مثال کوچک بیاورید که در آن رفتن مکرر به نزدیکترین نقطه منجر به یافتن کوتاهترین مسیر برای خوردن تمام نقاط نمی شود.

دقیقاً مشکلی که وجود دارد این است که با حرکت به سمت نزدیکترین نقطه، پکمن صرفاً جلوی پایش را میبیند و به آینده فکر نمیکند .ممکن است اگر الان نقطه ای را بخورد ضرر کمی کند اما بعداً از برگشت ها و طی کردنهای مسیر های طولانی دیگر جلوگیری کند .همانطور که در پایین تر مشاهده می کنید، پک من یک نقطه را نمیخورد چون نقطه نزدیکتری پیدا کرده است.

اما با صرفاً رها کردن یک نقطه، مجبور میشود بعد از اینکه تمامی نقاط به جز این نقطه را خورد، مسیر طولانی ای طی کند تا آخرین نقطه باقیمانده را بخورد .درصورتی که میتوانست هنگامی که در نزدیکی آن نقطه قرار دارد آن را بخورد و بعداً خودش را به درد سر نیانداز و زمان کمتری مصرف کند.