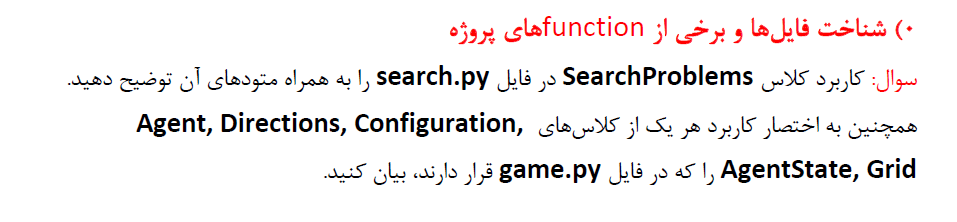
"بسمه تعالی"

مبانی هوش کاربردی – پروژه اول



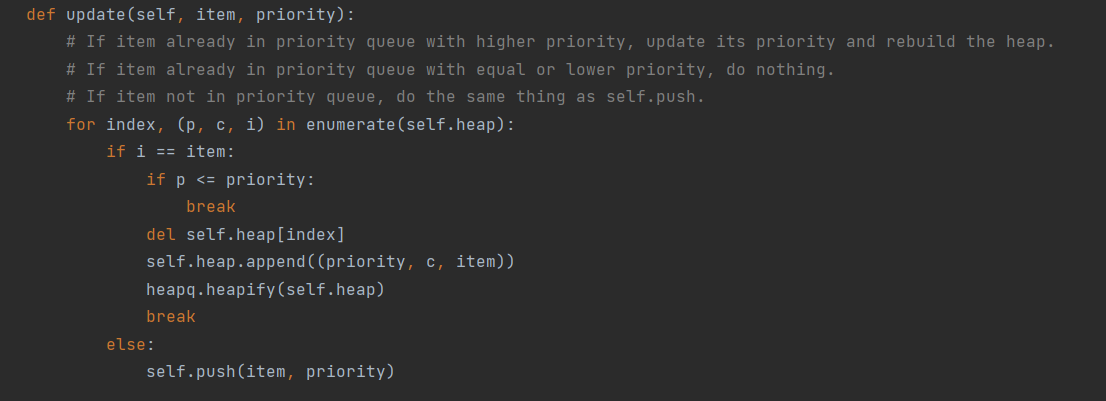
این کلاس بصورت کلی ساختار یک مسئله جستجو را مشخص میکند

* getStartState
* حالت اولیه را برای مسئله جستجو برمیگرداند
* isGoalState
* درصورتی که در حالت قابل قبول هدف باشیم مقدار صحیح را برمیگرداند
* getSuccessors
* یک سه گانه را برمیگرداند که شامل "وارث" یا بعبارت بهتر حالت ممکن بعدی، هزینه رسیدن به این حالت و عملی که باید انجام شود تا به آن برسیم خواهد بود
* getCostOfActions
* براورد هزینه کلی برای مجموعه ای از اعمال را برمیگرداند
* tinyMazeSearch
* دنباله ای از حرکات را که یک ماز کوچک را حل میکنند برمیگرداند، برای باقی ماز ها جواب درستی نخواهد داشت.
* depthFirstSearch
* با الگوریتم DFS عمیق ترین نود هارا در درخت جستجو میکند
* breadthFirstSearch
* با الگوریتم BFS سطح به سطح جستجو میکند
* uniformCostSearch
* با الگوریتم UCS نود ها با کمترین هزینه را جستجو میکند
* nullHeuristic
* heuristic را محاسبه میکند ( هزینه به نزدیک ترین حالت هدف ممکن)
* aStarSearch
* نود ای را جستجو میکند که بصورت برایند کمترین هزینه و اولین heuristic را دارد

**کلاس ها در game.py**

* Agent
* کلاس اجنت متدی تحت عنوان getAction دارد که با دریافت حالت فعلی و عملی برای آن وارد عمل میشود
* Configuration
* موقعیت فعلی را حفظ میکند در عین اینکه حرکت را دنبال میکند
* AgentState
* حالت فعلی یک اجنت را نگه داری میکند
* Directions
* موقعیت های حرکتی را تعریف میکند
* Grid
* موقعیت خانه ها در بازی

0)



در این بخش در یک حلقه روی آیتم های موجود در هیپ حرکت میکنیم، اگر آیتم i در هیپ موجود بود و اولویت کمتر یا مساوی داشت کاری نمیکنیم ولی اگر غیر این بود آن را از هیپ پاک کرده و مجددا هیپ را میسازیم و اگر این آیتم جدید بود آن را پوش میکنیم



برای اعمال الگوریتم DFS نیاز به استفاده از استک داریم پس ازutil استکی به نام forDFS تعریف میکنیم:

#Defining a stack for DFS traverse  
forDFS = util.Stack()

در قدم بعدی موقعیت اولیه پکمن را میگیریم و به شکل یک نود ( نود ریشه) ذخیره میکنیم:

#Getting starting point  
startLocation = problem.getStartState()  
  
# Defining Root Node => (location, path)  
rootNode = (startLocation, [])

حال گره ریشه را به استکی که تعریف کرده بودیم اضافه کرده و ست ای برای ذخیره سازی گره های مشاهده شده تعریف میکنیم:

#Pushing root to stack  
forDFS.push(rootNode)  
  
#Defining a set for visited nodes  
visitedLocations = set()

حال تا زمانی که استک ما خالی نباشد وارد حلقه وایل زیر میشویم، ابتدای کار اخرین گره افزوده شده به استک را به عنوان گره فعلی در نظر میگیریم، منظور از گره یک ارایه دوتایی است که المان اول نشان دهنده موقعیت ان گره و دوم جهت آن نسبت به گره قبلی است.

while not forDFS.isEmpty():  
 # node[0] : location, node[1] : path(NEWS)  
  
 #pop latest node as current node  
 node = forDFS.pop()

گره فعلی را به ست گره های مشاهده شده اضافه میکنیم، سپس چک میکنیم که ایا این گره همان گره هدف ما هست یا نه، اگر بود به نتیجه دلخواه رسیدیم و لازم نیست ادامه دهیم تنها جهت آن را برمیگردانیم.

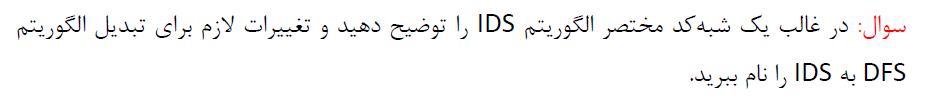
#adding current node to visited ones  
visitedLocations.add(node[0])  
  
#check whether current node is goal or not  
if problem.isGoalState(node[0]):  
 return node[1]

در غیر اینصورت گره های مجاور گره فعلی را درنظر گرفته درصورتی که قبلا مشاهده نشده باشند به صورت گره (موقعیت و جهت( به استک اضافه میکنیم :

#find successors of current node  
successors = problem.getSuccessors(node[0])  
  
for item in successors:  
 #checking whether successor has been visited or not  
 if item[0] in visitedLocations:  
 continue  
 #pushining unvisited ones as nodes to stack  
 forDFS.push((item[0], node[1] + [item[1]]))

این روند تا جایی که پکمن به گره هدف برسد تکرار خواهد شد و اگر هم در راستای یک شاخه به بن بست بخوریم طبق الگوریتم DFS به گره قبلی برگشته و مجددا جستجو در راستای آن شاخه را انجام میدهد تا نهایتا به جواب برسد.

همچنین گره ها علاوه برموقعیت فعلیشان جهت هایی که طی شده تا به آنها برسیم را در خود ذخیره کرده اند و تمامی گره های مشاهده شده هم در ست تعریف شده ذخیره شده اند.



الگوریتم های DFS, BFS برای درخت های بزرگ بخوبی عمل نمیکنند بنابراین میتوان از الگوریتم IDS استفاده کرد که تلفیقی از این دو الگوریتم است، در این الگوریتم بصورت سطح بندی شده از DFS استفاده میکنیم، به این معنی که در هر مرحله DFS اجازه دارد تا عمق معینی را جستجو کند.

// Returns true if target is reachable from

// src within max\_depth

**bool** IDS(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**;

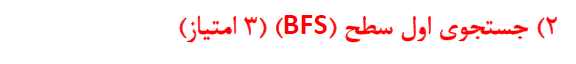
**foreach** adjacent i of src

**if** DLS(i, target, limit?1)

**return** **true**

**return** **false**

همانطور که در شبه کد بالا نشان داده شده باید در هرمرحله اجرای DFS محدودیتی برای عمق کاوش ان تعریف شود و از طرفی باید چندین و چند بار این الگوریتم را برای اعماق مختلف اعمال کرد تا به نتیجه دلخواه برسیم.



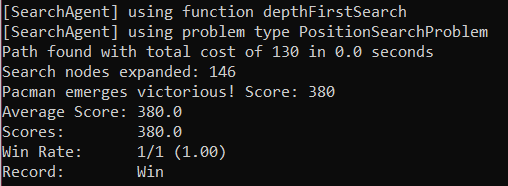
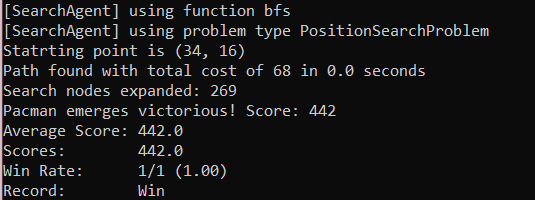
برای پیاده سازی این الگوریتم نیاز به تعریف یک صف داریم که تحت عنوان forBFS تعریف میشود، مشابه الگوریتم قبلی برای نود های مشاهده شده یک ست تعریف میشود و موقعیت گره ریشه به ان و خودش به صف اضافه میشود:

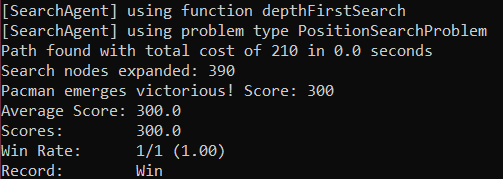
def breadthFirstSearch(problem):  
 #Defining a Queue for applying BFS  
 forBFS = util.Queue()  
  
 # Getting starting node  
 startLocation = problem.getStartState()  
  
 #Setting Root Node  
 rootNode = (startLocation, [])  
  
 #adding root node to queue  
 forBFS.push(rootNode)

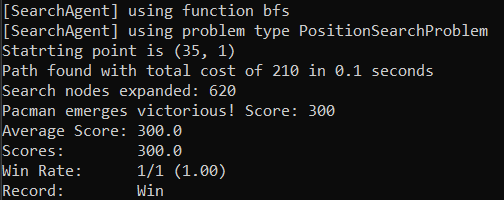
#Defining a set for visited nodes  
 visitedLocations = set()  
  
 #adding starting point to that set  
 visitedLocations.add(startLocation)

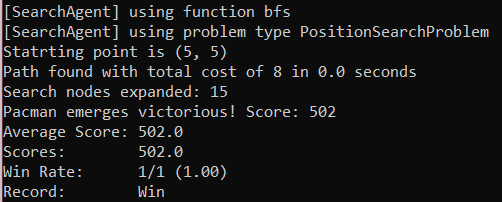
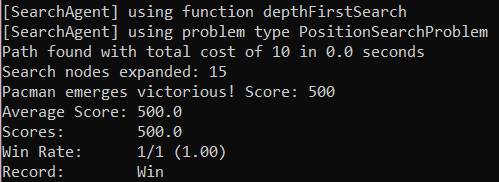
در قدم بعد اخرین گره را به عنوان گره فعلی در نظر میگیریم در صورتی که گره هدف بود ادامه نمیدهیم و درغیر این صورت وارث هارا پیدا کرده و به صف اضافه میکنیم:

while not forBFS.isEmpty():  
 # node[0] : location, node[1] : path (NEWS)  
 #Setting latest node as current one  
 node = forBFS.pop()  
  
 #checking whether current node is goal  
 if problem.isGoalState(node[0]):  
 return node[1]  
  
 #getting current node successors  
 successors = problem.getSuccessors(node[0])  
  
 #adding successors to queue if they are not visited  
 for item in successors:  
 if item[0] in visitedLocations:  
 continue  
 visitedLocations.add(item[0])  
 forBFS.push((item[0], node[1] + [item[1]]))

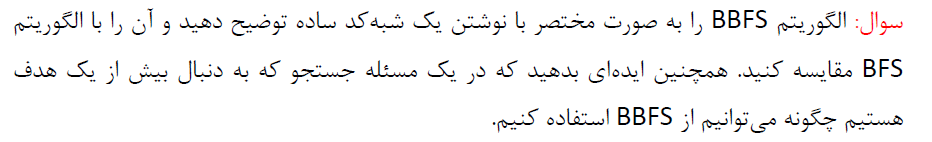
مقایسه این دو الگوریتم برای mediumMaze:

bigMaze:



tinyMaze :

همانطور که مشاهده میشود درمورد tinyMaze,mediumMaze الگوریتم BFS موفق تر از DFS عمل کرده است و درمورد bigMaze مشابه عمل کرده اند که این دور از انتظار نیست چون گفته بودیم هردوی این الگوریتم ها برای مسائل بزرگ خوب عمل نمیکنند.



در الگوریتم های bidirectional به جای اینکه تنها از طرف مبدا به سمت مقصد حرکت کنیم یا به عبارت بهتر از ابتدای مسئله به انتها برویم همزمان جستجو را از انتهای مسئله به سمت گره مبدا نیز آغاز میکنیم، محل تقاطع این دو جستجو مسیری را به ما خواهد داد که دنبالش هستیم. برای این جستجو میتوان از الگوریتم BFS استفاده کرد که در نهایت به ما BBFS را میدهد. شبه کد به صورت زیر خواهد بود:

startq = Queue for BFS from start node

endq = Queue for BFS from end node

parent= Array where startparent[i] is parent of node i

visited= Array where visited[i]=True if node i has been encountered

while startq is not empty and endq is not empty

perform next iteration of BFS for startq (also save the parent of children in parent array)

perform next iteration of BFS for endq

if we have encountered the intersection node

save the intersection node

break

using intersection node, find the path using parent array

برای پیدا کردن چند هدف میتوان چند گره متفاوت را به عنوان نقاط شروع از اخر و اول درخت ها انتخاب و جستجو کرد تا به نتیجه دلخواه برسیم.

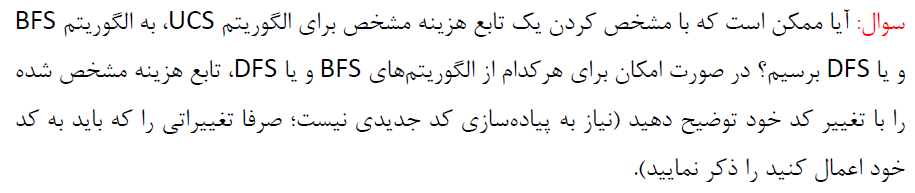


برای پیاده سازی این الگوریتم از صف اولویت استفاده میکنیم که در بخش اول بخش آپدیت آن را تکمیل کرده بودیم، یک صف اولویت به نام forUCS تعریف کرده، گره شروع را گرفته به آن و ست گره های مشاهده شده اضافه میکنیم، در این الگوریتم گره ها علاوه بر موقعیت و جهت هزینه تا رسیدن به آن گره را هم دارند:

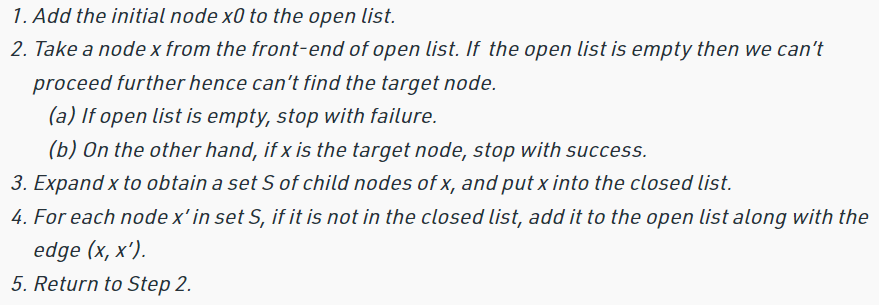
def uniformCostSearch(problem):  
 #Defining a priority queue  
 forUCS = util.PriorityQueue()  
  
 #Getting starting point  
 startLocation = problem.getStartState()  
  
 # (location, path, cost)  
 rootNode = (startLocation, [], 0)  
  
 #adding root to priority queue and visited locations  
 forUCS.push(rootNode, 0)  
 visitedLocations = set()

در قدم بعدی تا زمانی که صف اولویتمان خالی نباشد، گره با اولویت بالاتر را گرفته و به عنوان گره فعلی در نظر میگیریم نهایتا اگر گره هدف بود ادامه نمیدهیم وگرنه آن و وارثینش را (درصورت مشاهده نشده بودن) به ست اضافه میکنیم، در اضافه کردن این گره ها به ست هزینه تا رسیدن به آنها را مجدد حساب کرده و به اطلاعات آن می افزاییم:

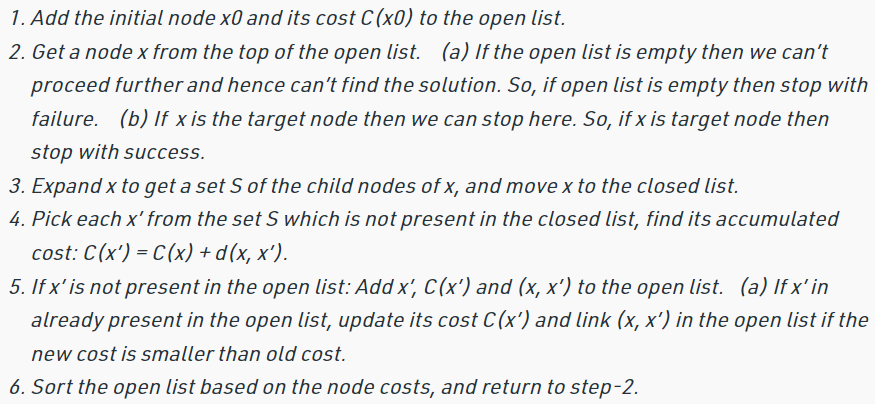
while not forUCS.isEmpty():  
 # node[0] : location, node[1] : path, node[2] : cost  
 #setting latest node as current one  
 node = forUCS.pop()  
  
 #checking whether if current node is goal or not  
 if problem.isGoalState(node[0]):  
 return node[1]  
  
 #adding current node to visited ones and checking for its successors  
 if node[0] not in visitedLocations:  
 visitedLocations.add(node[0])  
 for successor in problem.getSuccessors(node[0]):  
 if successor[0] not in visitedLocations:  
 cost = node[2] + successor[2]  
 forUCS.push((successor[0], node[1] + [successor[1]], cost), cost)



اگر الگوریتم BFS را با الگوی زیر جلو ببریم:



و الگوریتم UCS بصورت زیر باشد:



اگر منظور از d(x,x’) هزینه رسیدن از x به x’ باشد و این هزینه را برای همه نقاط 1 در نظر بگیریم الگوریتم ucs به bfs تبدیل میشود.

اما امکان تبدیل الگوریتم ucs به dfs وجود ندارد



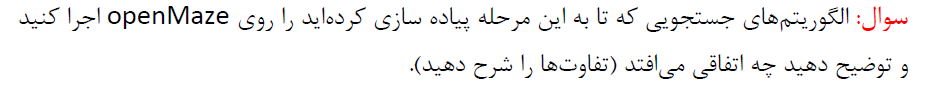
در قدم اول لازم است فاصله منهتن و اقلیدسی را تعریف کنیم:

def manhattanHeuristic(position, problem, info={}):  
 *"The Manhattan distance heuristic for a PositionSearchProblem"* point1 = position  
 point2 = problem.goal  
 return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])

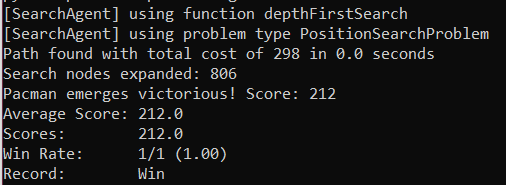
def euclideanHeuristic(position, problem, info={}):  
 *"The Euclidean distance heuristic for a PositionSearchProblem"* point1 = position  
 point2 = problem.goal  
 return ( (point1[0] - point2[0]) \*\* 2 + (point1[1] - point2[1]) \*\* 2 ) \*\* 0.5

برای اعمال الگوریتم یک صف اولویت و یک ست مشابه الگوریتم های قبلی تعریف کرده و روند کلی الگوریتم قبلی را جلو میبریم با این تفاوت که به جای محاسبه هزینه تابع f که شامل هزینه و هیوریستیک است را ذخیره میکنیم که اولویت گره ها را در صف تعیین کند:

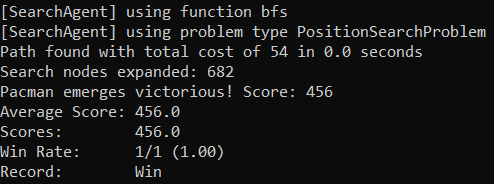
def aStarSearch(problem, heuristic=nullHeuristic):  
 #defining a priority queue  
 forAstar = util.PriorityQueue()  
  
 #Getting start location  
 startLocation = problem.getStartState()  
  
 #Setting root node  
 rootNode = (startLocation, [], 0)  
 forAstar.push(rootNode, 0)  
 visitedLocations = set()  
  
 while not forAstar.isEmpty():  
 # node[0] : location,node[1] : path,node[2] : cumulative cost  
 #current node  
 node = forAstar.pop()  
  
 #checking whether current node is goal or not  
 if problem.isGoalState(node[0]):  
 return node[1]  
  
 #adding to visited nodes  
 if node[0] not in visitedLocations:  
 visitedLocations.add(node[0])  
 for successor in problem.getSuccessors(node[0]):  
 if successor[0] not in visitedLocations:  
 cost = node[2] + successor[2]  
 #f function  
 totalCost = cost + heuristic(successor[0], problem)  
 forAstar.push((successor[0], node[1] + [successor[1]], cost), totalCost)



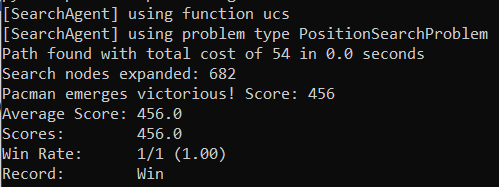
الگوریتم DFS :



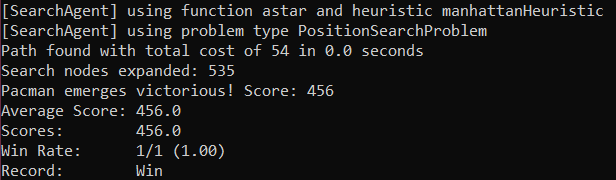
الگوریتم BFS :



الگوریتم UCS:



الگوریتم A\* :



مشاهده میشود که الگوریتم DFS بسیار کند عمل میکند و نهایتا امتیاز پایین تری میگیرد، عملکرد الگوریتم BFS از همه موفق تر و سریع تر است و الگوریتم UCS هم در حالتی که هزینه طی کردن مسیر 1 باشد همانBFS است، همچنین الگوریتم A\* هم نتیجه مشابهی دارد چون در این مسئله هیوریستیک و هزینه ای تعریف نشده است.



در کلاس CornersProblem ابتدا تابع init تعریف شده که ما تغییری نداده ایم. در قدم بعدی تابعی برای برگرداندن استیت اولیه تعریف میشود،، همانطور که مشاهده میکنید اینکه یک تاپل هم به استیت اضافه شده که بعدا برای پیدا کردن کرنر ها استفاده خواهد شد:

def getStartState(self):  
 *"""  
 Returns the start state (in your state space, not the full Pacman state  
 space)  
 """* return (self.startingPosition, (0,1,2,3))

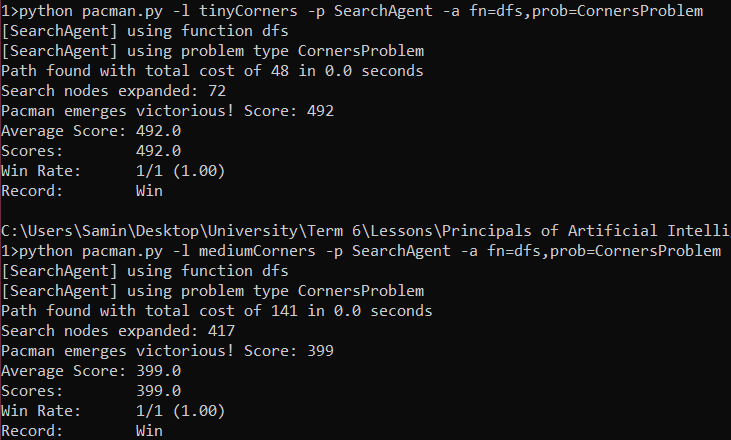
تابع بعدی چک میکند آیا این استیت هدف مسئله هست یا خیر:

def isGoalState(self, state):  
 *"""  
 Returns whether this search state is a goal state of the problem.  
 """* return not state[1]

این تابع وارث های این گره را برمیگرداند، به این صورت که یک لیست برای وارثین تعریف میکنیم، به ازای هرحرکت( شمال، غرب،جنوب، شرق) از موقعیت فعلی چک میکنیم اگر دیوار نبود آن را به عنوان یک وارث اضافه میکنیم، لازم به ذکر است که اگر در یک کرنر باشیم این مورد را چک میکنیم و از لیست کرنر ها مسئله کم میکنیم.

def getSuccessors(self, state):  
 successors = []  
 for action in [Directions.NORTH, Directions.WEST, Directions.SOUTH, Directions.EAST]:  
 x, y = state[0]  
 #delta to next state position  
 dx, dy = Actions.directionToVector(action)  
 #next state  
 nextX, nextY = int(x + dx), int(y + dy)  
  
 #check if next one is a wall or not  
 if not self.walls[nextX][nextY]:  
 # Change state[1] if reaches corner  
 leftCorners = state[1]  
 nextLocation = (nextX, nextY)  
 try:  
 # Find out if the successor is a corner  
 indexOfCorner = self.corners.index(nextLocation)  
 except:  
 pass  
 else:  
 if indexOfCorner in leftCorners:  
 temp = list(leftCorners)  
 temp.remove(indexOfCorner)  
 leftCorners = tuple(temp)  
  
 nextState = (nextLocation, leftCorners)  
 successors.append((nextState, action, 1))  
   
 self.\_expanded += 1 # DO NOT CHANGE  
 return successors

خروجی مسئله ها بصورت زیر است:



همانطور که مشاهده میشود با الگوریتم DFS پکمن موفق به کشف کرنر ها شده است.



برای محاسبه هیوریستیک در این مسئله ابتدا کرنرها و دیوارهای مسئله را در متغیر هایی به همین نام ها ذخیره میکنیم، موقیت فعلی و کرنرهارا هم در متغیر های دیگری ذخیره میکنیم:

def cornersHeuristic(state, problem):  
 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]  
 cornersIndex = state[1]

اگر کرنری موجود نبود صفر برمیگردانیم:

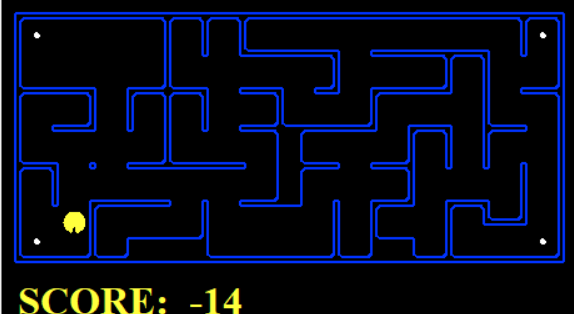
if not cornersIndex:  
 return 0

وگرنه فاصله منهتن گره فعلی را با کرنرهای موجود محاسبه کرده ماکسیموم مقدار را برمیگردانیم:

# max manhattan distance among all remained corners  
return max([util.manhattanDistance(currentPosition,corners[idx]) for idx in cornersIndex])



هیوریستیک ما حداکثر فاصله منهتن موقعیت گره فعلی تا گوشه هاست موقعیت زیر را در نظر بگیرید:



d

c

b

**a**

در این موقعیت فاصله منهتن پکمن تا چهارگوشه به رنگ نارنجی و فاصله اقلیدسی آن به رنگ سفید نشان داده شده است، بنابر نابرابری مثلثی مشخص است که فاصله منهتن تا نقاط گوشه کوچکتر مساوی فاصله واقعی است ( درحالیکه فاصله اقلیدسی چنین نیست و برای همین آن را انتخاب نمیکنیم) پس هیوریستیک ما قابل قبول است، برای سازگار بودن فرض میکنیم بخواهیم به گوشه بالا سمت چپ رفته بعد به گوشه بالا سمت راست برویم در چنین حالتی اگر از مسیر واقعی حرکت کنیم باید از فاصله اقلیدسی نقطه فعلی تا اولین مقصد رفته (c) و بعد مستقیما به مقصد بعدی برویم(a+b ) درصورتی که اگر از فاصله منهتن استفاده کنیم مقدار کمتری خواهیم داشت (e+a+b) زیرا میدانیم مشخصا a,e از c کوچکتر هستند پس دلتای نقاط از حالت واقعی کمتر و سازگار است. علت انتخاب بزرگترین فاصله منهتن این است که با طی کردن این فاصله باقی گوشه هارا هم چک کرده ایم.

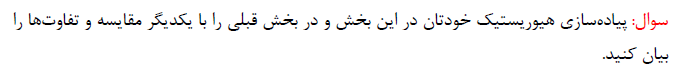


در این بخش ابتدا موقعیت فعلی پکمن و موقعیت غذاهارا میگیریم سپس با استفاده از تابع mazeDistance چک میکنیم ( اگر هیورستیک مشخصی برای آن نقطه که غذا در ان قرار دارد نداشته باشیم از قبل) چقدر فاصله تا آن وجود دارد (فاصله منهتن) و دورترین غذارا به عنوان هیورستیک مسئله برمیگردانیم:

def foodHeuristic(state, problem):  
 #Getting current state and food locations  
 position, foodGrid = state  
 foods = foodGrid.asList()  
 #if there is no food then return 0  
 if not foods:  
 return 0  
  
 farFood = 0  
 for food in foods:  
 key = position + food  
 if key in problem.heuristicInfo:  
 distance = problem.heuristicInfo[key]  
 else:  
 # Use manhattan distance   
 distance = mazeDistance(position, food, problem.startingGameState)  
 problem.heuristicInfo[key] = distance  
  
 if distance > farFood:  
 farFood = distance  
  
 return farFood



در این بخش از فاصله میز استفاده کرده ایم، اینجا باید تمامی غذاها خورده شود و انتخاب دورترین غذا جواب قابل قبولی به ما میدهد زیرا یا باید از این راه رفته ان را بخوریم ( همچنین باقی را) یا از راه های دورتر و از طرفی سازگار هم هست زیرا مقدار تابع f در طی مسیر زیاد میشود.



در قسمت قبلی از فاصله منهتن استفاده کرده بودیم اما اینجا از فاصله میز استفاده کرده ایم و محاسبه هیورستیک برای غذاهای تکراری دوباره انجام نمیشود.