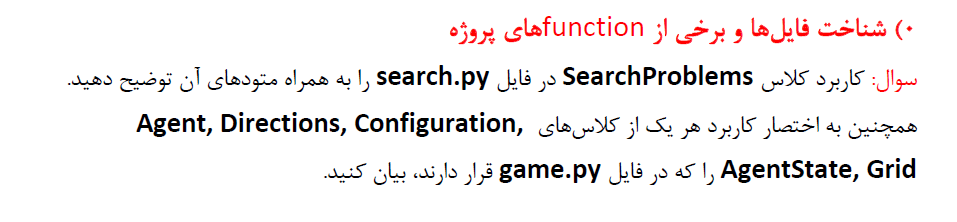
"بسمه تعالی"

مبانی هوش کاربردی – پروژه اول



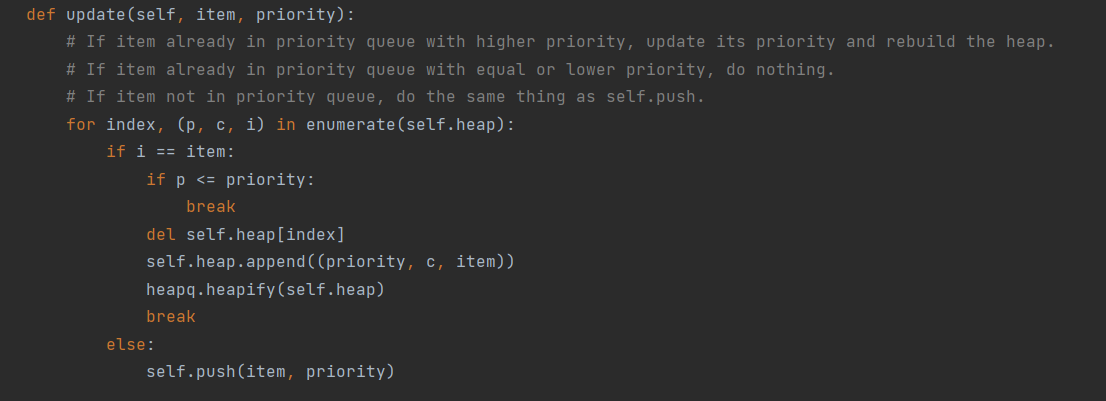
این کلاس بصورت کلی ساختار یک مسئله جستجو را مشخص میکند

* 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.

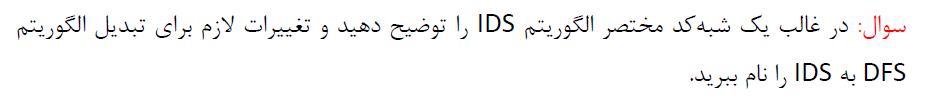
#adding current node to visited ones  
visitedLocations.add(node[0])  
  
#check whether current node is goal or not  
if problem.isGoalState(node[0]):  
 return node[1]

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**;

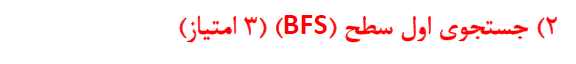
**foreach** adjacent i of src

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

**return** **true**

**return** **false**

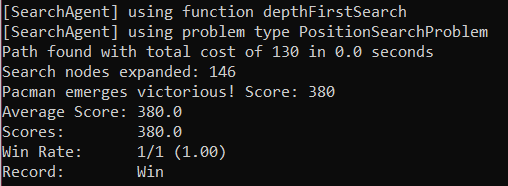
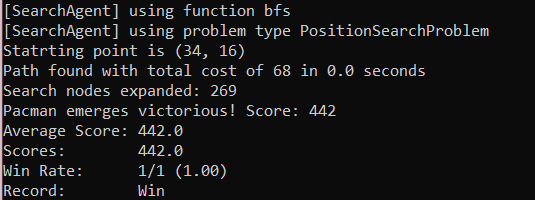
همانطور که در شبه کد بالا نشان داده شده باید در هرمرحله اجرای DFS محدودیتی برای عمق کاوش ان تعریف شود و از طرفی باید چندین و چند بار این الگوریتم را برای اعماق مختلف اعمال کرد تا به نتیجه دلخواه برسیم.

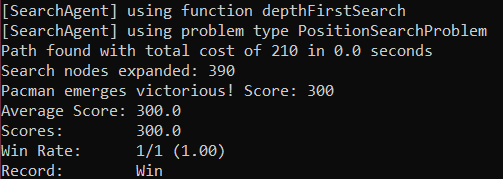


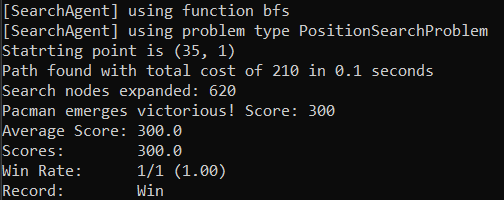
برای پیاده سازی این الگوریتم نیاز به تعریف یک صف داریم که تحت عنوان forBFS تعریف میشود، مشابه الگوریتم قبلی برای نود های مشاهده شده یک ست تعریف میشود و موقعیت گره ریشه به ان و خودش به صف اضافه میشود:

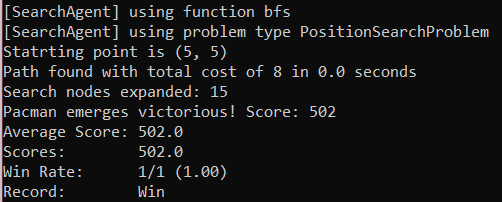
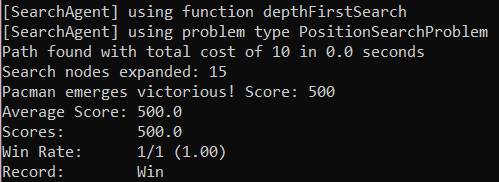
def breadthFirstSearch(problem):  
 #Defining a Queue for applying BFS  
 forBFS = util.Queue()  
  
 # Getting starting node  
 startLocation = problem.getStartState()  
  
 #Setting Root Node  
 rootNode = (startLocation, [])  
  
 #adding root node to queue  
 forBFS.push(rootNode)  
  
 #Defining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

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.