باسمه تعالی

پروژه ۱ درس مبانی هوش مصنوعی – استاد جوانمردی

سید امیرمهدی میرشریفی- ۹۸۳۱۱۰۵

سوال: کاربرد کلاس **SearchProblems** در فایل **search.py** را به همراه متو دهای آن توضیح دهید.
همچنین به اختصار کاربرد هر یک از کلا سهای **Agent, Directions, Configuration, AgentState, Grid** را که در فایل **game.py** قرار دارند، بیان کنید.

پاسخ:

کلاس searchProblem کلاسی است که مسئله ما شیئی از آن است و همچنین حاوی توابعی است که اطلاعات مربوط به مسئله و همچنین خدماتی را ارائه میدهد که در ادامه توضیح داده میشود.

توابع این کلاس :

getStartState : موقعیت ابتدایی عامل را بر می گرداند.

isGoalState: بررسی می کند که آیا به وضعیت هدف رسیده ایم یا خیر

getSuccessors: یک لیست از سه تایی موقعیت بعدی ، عمل مورد نیاز و هزینه آن را بر می گرداند.

getCostOfAction: بابت دریافت دنباله ای از اعمال هزینه آن را بر میگرداند. همچنین این حرکات باید مجاز باشند.

کلاس Grid: این کلاس کلاسی است که یک نقشه دوبعدی که به صورت یک لیست حاوی تعدادی لیست است را نگه می دارد و در ابتدا به ازای مقدار اولیه ای که به آن داده می شود ، مقدار هر خانه را برابر آن مقدار می گذارد.

کلاس AgentState : این کلاس حالت های عامل ما که حاوی تنظیمات و سرعت یا حالت ترس را نگه داری میکند.

کلاس Agent: در این کلاس که معلق به عامل است یک تابع تعریف میشود به نام getAction عامل متناسب با وضعیتی که دریافت می کند یک حرکت از بین حرکت های تعریف شده بر میگرداند.

کلاس Directions :در این کلاس جهت ها و همچنین اعمالی که یک عامل می تواند انجام دهد تعریف شده است.

کلاس Configuration: این کلاس مختصات عامل و همچنین جهتی که به سمت آن است را نگهداری میکند و همچنین با گرفتن یک حرکت می تواند این مختصات را به روز رسانی بکند.

پیاده سازی: متود **update** از کلاس **PriorityQueue** را در فایل **util.py** کامل کنید. (توضیحات کمکی جهت کامل کردن این متود در کنار این متود در کد قرار داده شده است) .

در ابتدا داخل هرم خود چک میکنیم که آیا آیتم مورد نظر موجود است یا خیر. اگر موجود بود شماره ایندکس آن را به دست آورده و چک می کنیم که اولیت جدید اگر شرط به روزرسانی را داشت ، به روز رسانی کنیم و در خیر این صورت کاری نکنیم. اگر هم موجود نبود آن را اضافه خواهیم کرد.

```
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.

    index = [i for i, v in enumerate(self.heap) if v == item]
    if len[index] ==0:
        self.push(item, priority)
        return
    index=index[0]
    (prev_priority, prev_count) = self.heap[index]
    if prev_priority <= priority:
        return


    self.heap[index] = (priority, prev_count, item)
    heapq.heapify(self.heap)
```

تابع search:

برای پیاده سازی توابع جست وجو یک تابع مبنا نوشته شد که الگوریتم یکسانی را برای تمام الگوریتم ها جست و جو فراهم می کند و در مقادیری مانند نوع fringe و این که آیا اولویت دار است یا خیر و هیوریستیک و غیره متفاوت است.

```
def search(problem,fringe,is_priority_fringe=False,heuristic=None,breadth_limit=None):
```

در ابتدا بررسی می کنیم که آیا اولویت مهم است یا خیر. اگر مهم بود یعنی می بایست هزینه ها هم همراه موقعیت و اعمال داخل fringe وارد شوند و در غیر این صورت هزینه مهم نیست. سپس حالت اولیه را داخل fringe وارد میکنیم.

```
if is_priority_fringe:
    fringe.push((problem.getStartState(),[]),0)
else:
    fringe.push((problem.getStartState(),[]))
```

سپس تا زمانی که fringe خالی نشده است گره ها را گسترش می دهیم و بنابر این که آیا چک کردن هدف زمان گسترش گره است یا زمان اضافه کردن آن دو حالت را در نظر می گیریم.

```
if state not in visited_nodes:
    if is_priority_fringe:
        if problem.isGoalState(state):
                return action


        successors=problem.getSuccessors(state)
        for successor in successors:
                new_state=successor[0]
                new_action=successor[1]
                new_cost=problem.getCostOfActions(action+[new_action])
                hcost=heuristic(new_state,problem)

                fringe.push((new_state,action+[new_action]),new_cost+hcost)
```

همانطور که در تصویر بالا می بینید ابتدا اگر گره در بین گره های بسط داده شده نباشد، و اگر اولیت دار باشد بررسی می کند که آیا هدف است یا خیر و سپس با استفاده از تابع پسین حالات دیگر را به fringe اضافه می کند. این در حالی است که اگر جست و جو اولویت دار نباشد در هنگام اضافه کردن گره ها بررسی هدف بودن را انجام می دهد (کد آن موجود است که در تصاویر بالا آورده نشده است).

بدین ترتیب تمام الگوریتم های جست و جویی که قرار است در ادامه پیاده سازی شود را می شود با تغییراتی در ورودی این تابع به راحتی پیاده سازی کرد.

پیاده سازی DFS:

طبق مطالبی که در بالا گفته شده است صرفا نیاز است تابع search را با ptoblem و یک پشته که در این الگوریتم نیاز است مقدار دهی بکنیم و actions را بر گردانیم.

```
def depthFirstSearch(problem):
    return search(problem,util.Stack())
```

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

پاسخ:

الگوریتم IDS یا Iterative deeping depth first search الگوریتمی ترکیبی از الگوریتمDFS & BFS است با این تفاوت که برای جست و جو محدودیت عمق می گذارد و هر بار از عمق یک تا عمق داده شده و در هر مرحله یک بار الگوریتم جست و جوی DFS را اجرا می کند.

کد زیر کدی است که برای پیاده سازی الگوریتم IDS به کار رفته است.

```
def iterative_deepening_depthFirstSearch(problem,level):
    for i in range(1, level+1):
        action=search(problem,util.Stack(),is_priority_fringe=False,breadth_limit=i)
        if len(action)is not 0:
            return action
```

چون از DFS به صورت مستقیم استفاده می کنیم بنابراین نوع fringe یک پشته است که اولیت ندارد و عمق آن هم در هر محله یکی افزایش پیدا میکند.

همچین این محدودیت عمق در تابع search در نظر گرفته شده است

```
#this is for ids search
if breadth_limit is not None:
    if len(action+[new_action])>breadth_limit-1:
        visited_nodes.add(state)
        continue
# finish
```

پیاده سازی BFS:

همانند DFS با این تفاوت که نوع fringe صف است.

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    return search(problem,util.Queue())
```

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

BBFS یک جست و جوی گرافی است که بجای این که از یک جهت شروع به جست و جو کند، از ریشه و هدف به صورت همزمان، یکی به صورت رو به جلو (ریشه) و دیگری رو به عقب (هدف) جست و جو می کنند تا در یک گره به یکدیگر برسند.

کد زیر شبه کدی از این الگوریتم است:

```
def bidirectional_search(src, dest):

      src_queue.append(src)
      src_visited[src] = True
      src_parent[src] = -1
      self.dest_queue.append(dest)
      self.dest_visited[dest] = True
      dest_parent[dest] = -1
      while self.src_queue and self.dest_queue:

        fbfs=bbs(direction = 'forward')
        backbfs=bbs(direction = 'backward')
        intersecting_node = self.is_intersecting()
          if intersecting_node != -1:
              print(f"Path exists between {src} and {dest}")
              print(f"Intersection at : {intersecting_node}")
              print_path(intersecting_node,
                                src, dest
      return -1
```

می توانیم بین هر هدف و ریشه یک جست و جوی bbfs را انجام دهیم و راه بهینه را اعلام کنیم.

پیاده سازی UCS:

```
def uniformCostSearch(problem,heuristic=nullHeuristic):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    return search(problem,util.PriorityQueue(),True,heuristic=heuristic)
```

سوال: آیا ممکن است که با مشخص کردن یک تابع هزینه مشخص برای الگو ریتم UCS ، به الگو ریتمBFS و یا DFS برسیم؟ در صورت امکان برای هرکدام از الگو ریتمهای BFS و یا DFS ، تابع هزینه مشخص شده را با تغییر کد خود توضیح دهید)نیاز به پیاده سازی کد جدیدی نیست؛ صرفا تغییراتی را که باید به کد خود اعمال کنید را ذکر نمایید.)

تفاوت UCS و DFS در این است که صف USC اولیت دار است . همچنین UCS هنگام گسترش گره چک می کند که آن هدف است یا نه و DFS هنگام اضافه کردن آن. برای همین اگر در تابع UCS هنگام اضافه کردن هزینه، همه را با هزینه صفر وارد کنیم و چک کردن این که هدف است یا خیر هم در هنگام اضافه کردن گره انجام دهیم به DFS تبدیل می شود.

برای تبدیل USC به BFS همان کار بالا با این تفاوت که بجای صف می بایست یک پشته به این تابع به عنوان fringe بدهیم.

پیادهسازی: ابتدا، متود دو هیوریستیک منهتن و اقلیدسی را که در توابع manhattanHeuristic و euclideanHeuristicدر فایل searchAgents.py قرار دارند، کامل کنید.

```
def manhattanHeuristic(position, problem, info={}):
    "The Manhattan distance heuristic for a PositionSearchProblem"
    startx, starty = position
    goalx, goaly = problem.goal
    return abs(goalx - startx) + abs(goaly - starty)
def euclideanHeuristic(position, problem, info={}):
    "The Euclidean distance heuristic for a PositionSearchProblem"
    xdistance = abs(problem.goal[0]-position[0])
    ydistance = abs(problem.goal[1]-position[1])
    return math.sqrt(xdistance**2 + ydistance**2)
```

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.)

پس از آن که این دو هیورستیک را روی گره اعمال می کنیم تفاوت به وجود آمده را می توان در بین گره های بسط داده شده مشاهده کرد. برای مثال در هیوریتسک منهتن تعداد گره ها برابر ۴۴۹ بود و در اقلیدسی برابر ۴۵۷ که نشان از اثر هیورستیک دارد.

پیدا کردن همه گوشه ها

در این بخش تنها کافی است دو تابع getStartState & isGoalState که می بایست در اینجا متفاوت پیاده سازی شود را تغییر داد. در این توابع شرط اصلی گذر عامل از گوشه ها است که در کد زیر قابل مشاهده است:

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    corner_visite_condition = (False,False,False,False)

    # Gotta check if we started in a corner.
    for i in range(4):
        if self.startingPosition==self.corners[i]:
            corner_visite_condition[i]=True



    return (self.startingPosition, corner_visite_condition)

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"
    # Have to validate all four.
    one, two, three, four = state[1]
    return one and two and three and four
```

cornersHeuristic

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

در این بخش منطق هیوریستیک ما این است که فاصله منهتن را از موقعیت عامل تا ۴ گوشه حساب می کند و پس ماکسیمم آن ها را بر می گرداند.

این تابع قطعا پذیرفته شده است زیرا فاصله منهتن تا هر نقطه قطعا کمتر از فاصله اصلی است(بخاطر وجود دیوار). همچنین سازگار است. زیرا اختلاف هیورستیک بین زمانی که میخواهی مستقیم تخمین فاصله تا یک گوشه بزنی یا از تخمین از یک گوشه به گوشه ای دیگر باز هم کمتر از هزینه واقعی است. اختلاف این دو هیوریستیک. فرض کنید عامل در نقطه (x,y) قرار دارد. یک گوشه مختصات (0,0) و دگیر مختصات (x+z,0). بنابر این تخمین هزینه از وضعیت عامل از گوشه اول برابر x+y و از گوشه دوم برابر z+y است.

همچنین تخمین هزینه گوشه اول از گوشه دوم برابر x+z است.بنابراین تخمین هزینه از نقطه عامل تا گوشه اول برابر است با : z+y – (z+x) =y-x<x+y<main cost

```
def cornersHeuristic(state, problem):
    corners = problem.corners  # These are the corner coordinates
    # These are the walls of the maze, as a Grid (game.py)
    walls = problem.walls

    "*** YOUR CODE HERE ***"
    current_position = state[0]
    cornersStatus = state[1]
    heuristic = 0
    if problem.isGoalState(state):
        return heuristic

    distancesFromGoals = []
    for index, item in enumerate(cornersStatus):
        if item == False:
            distancesFromGoals.append(util.manhattanDistance(
                current_position, corners[index]))
    heuristic = max(distancesFromGoals)
    return heuristic
```

foodHeuristic

سوال:هیوریستیک خود را توضیح دهید و سازگاری آن را استدلال کنید.
سوال: پیاه سازی هیوریستیک خودتان در این بخش و در بخش قبلی را با یکدیگر مقایسه و تفاوتها را بیان کنید.

در این بخش هیورستیکی که به کار رفته است این است که فاصله واقعی از موقعیت عامل تا دورترین نقطه غذا را به واسطه تابع mazedestination به عنوان مقدار هیوریستیک بر میگردانیم.

این هیورستیک قابل قبول است زیرا هزینه آن قطعا از هزینه واقعی که می بایست مسیر های دیگری برای خوردن دیگر غذا ها طی شود کمتر است. همچنین سازگار است زیرا فرق دو هیورستیک برای دو گره همسایه در اختلاف فاصله آن ها از دورترین غذا است که که بازهم برابر هزینه واقعی است.

بنابراین سازگار است.

تفاوت این هیورستیک با هیورستیک قبلی اولا در تابعی است که با آن فاصله را حساب می کنیم و درثانی در هیورستیک قبلی صرفا ماکسیمم فاصله تا ۴ گوشه مد نظر است و در این هیورستیک ماکسیمم فاصله تا غذا.

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    if  not  foodGrid.asList():
        return 0
    each_food_distance=[]
    for i in foodGrid.asList():
        each_food_distance.append(mazeDistance(position,i,problem.startingGameState))

    return max(each_food_distance)
```

findPathToClosestDot

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

در این تابع ما با استفاده از الگوریتم جست و جوی IDS غذایی را پیدا می کنیم که لزوما کوتاه ترین مسیر نیست. زیرا الگوریتم IDS لزوما جواب بهینه نمی دهد.