امیرعلی بلباسی 9831109

سوال 0) کلاس SearchProblem یک کلاس abstract است و کاربردش این است که ساختار یک مسئله جستجو را شکل می‌دهد. به همین جهت فقط هدر متودها را داریم و پیاده‌سازی‌ای در این کلاس صورت نمی‌گیرد (پیاده‌سازی در کلاس‌های دیگری که آن را به ارث می‌برند صورت می‌گیرد). متود getStartState حالت شروع مسئله جستجو ما را برمی‌گرداند. متود isGoalState مقدار true برمی‌گرداند اگر و تنها اگر حالت فعلی مسئله جستجو ما، حالت goal باشد. متود getSuccessors لیستی از سه‌تایی‌هایی برمی‌گرداند. هر سه‌تایی شامل حالت جانشینی برای حالت فعلی ما، عملی که برای رسیدن به آن حالت از حالت فعلی لازم است و هزینه گسترش از حالت فعلی به آن حالت است. متود getCostOfActions یک لیستی از عمل‌ها را می‌گیرد و مجموع هزینه انجام آنها را برمی‌گرداند.

کلاس Configuration مختصات x و y و جهت حرکت شخصیت‌ها (روح یا پکمن) را نگه‌داری می‌کند. استاندارد قراردادی x و y هم به صورت نموداری است (0 و 0 چپ پایین قرار دارد). کلاس Directions شامل سه دیکشنری LEFT، RIGHT و REVERSE می‌باشد که بسته به ورودی (key) به ترتیب جهت حرکت جایگزین به صورت پادساعت‌گرد، ساعت‌گرد و 180 درجه معکوس برمی‌گرداند (value) (اگر ایستاده هم باشد همچنان ایستاده می‌ماند). کلاس Agent یک کلاس abstract شامل متود abstract، getAction است. کاری که این متود می‌کند این است که بسته به آنکه کجا دارد پیاده‌سازی می‌شود (کدام کلاس دارد کلاس Agent را به ارث می‌برد) آنجا با توجه به ماهیت و هدفی که آن کلاس دارد (مثلا کلاس عامل روح) و با توجه به حالت فعلی، باید یکی از عمل‌های شمال، جنوب، شرق، غرب یا ایست را برگرداند. کلاس Grid یک آرایه دوبعدی از آبجکت‌ها و یک لیست از لیست‌ها دارد که نقشه بازی پکمن را نمایندگی می‌کنند (0 و 0 چپ پایین قرار دارد). طول و عرض آرایه نیز قابل مقداردهی است (در __init__ هنگام ساخت آبجکت مقداردهی می‌شود). کلاس AgentState حالات عوامل را نگه می‌دارد. مانند مختصات و جهت، ترسیده بودن، سرعت و پکمن یا روح بودن.

پیاده‌سازی 0)

```
"*** YOUR CODE HERE ***"
foundItem = 0
for (n, (p, c, i)) in enumerate(self.heap):
    if i == item:
        foundItem = 1
        if p > priority:
            self.heap[n] = (priority, c, i)
            heapq.heapify(self.heap)
if foundItem == 0: self.push(item, priority)
```

یک حلقه روی heap می‌زنیم. اگر item ما پیدا شد با توجه به مقدار priority داده شده و priority پیداشده، یا مقدار priority آن item را update می‌کنیم و سپس heapify می‌کنیم تا دوباره یک heap داشته‌باشیم (وقتی که مقدار priority داده شده کوچک‌تر از priority پیداشده است) یا هیچ‌کاری نمی‌کنیم (مقدار priority داده شده بزرگ‌تر یا مساوی priority پیداشده است). در آخر هم با توجه به flag، اگر item ای نتوانستیم پیدا کنیم، آن item را با priority داده‌شده به push heap می‌کنیم.

پیاده‌سازی 1)

```
"*** YOUR CODE HERE ***"
fringe = util.Stack()
explored = []
fringe.push(([problem.getStartState()], []))
while not fringe.isEmpty():
    expanded = fringe.pop()
    if problem.isGoalState(expanded[0][-1]): return expanded[1]
    explored.append(expanded[0][-1])
    successors = problem.getSuccessors(expanded[0][-1])
    for successor, action, stepCost in successors:
        if successor not in explored:
            tmp0 = expanded[0].copy()
            tmp1 = expanded[1].copy()
            tmp0.append(successor)
            tmp1.append(action)
            fringe.push((tmp0, tmp1))
util.raiseNotDefined()
```

Fringe از نوع استک ایجاد می‌کنیم و یک لیست برای گره‌های explore شده (زیرا جستجو گرافی داریم انجام می‌دهیم). حالت شروع را پوش می‌کنیم سپس در یک حلقه تا وقتی fringe خالی نشده، گره از fringe پاپ کرده و سپس تست گل را انجام می‌دهیم. آن گره را به لیست explore شده ها اضافه کرده و سپس گره های جایگزین را با تابع getSuccessors می‌گیریم و در آن حلقه می‌زنیم و آنهایی که قبلا explore نشده‌اند را تک تک به fringe اضافه می‌کنیم. حلقه while تا وقتی که fringe خالی نشده‌باشد یا به حالت گل رسیده‌باشیم، ادامه پیدا می‌کند (ساختمان داده هر گره هم به این صورت است که یک tuple دوتایی است که عنصر اول آن یک لیست از مختصات های مسیر طی شده در گره ماست و عنصر دوم آن نیز یک لیست از اکشن‌هایی است که مارا به ترتیب به آن گره رسانده)

سوال 1)

تا وقتی گل پیدا نشده یا فرینج خالی نشده:

Dfs+ را انجام بده

تفاوت dfs+ با dfs این است که یک آرگومان برای عمق حداکثر نیز می‌گیرد و تنها حداکثر تا آن عمق جستجو می‌کند

برای تبدیل dfs به ids نیز باید به جای اینکه یکسره تا بیشترین عمق ممکن پیش برویم، در iterate های مختلف تا حداکثر یک عمق پیش می‌رویم.

پیاده‌سازی 2)

```
"*** YOUR CODE HERE ***"
fringe = util.Queue()
explored = []
fringe.push(([problem.getStartState()], []))
while not fringe.isEmpty():
    expanded = fringe.pop()
    if problem.isGoalState(expanded[0][-1]): return expanded[1]
    explored.append(expanded[0][-1])
    successors = problem.getSuccessors(expanded[0][-1])
    for successor, action, stepCost in successors:
        if successor not in explored:
            flag = 0
            for route, actions in fringe.list:
                if successor == route[-1]: flag = 1
            if flag == 0:
                tmp0 = expanded[0].copy()
                tmp1 = expanded[1].copy()
                tmp0.append(successor)
                tmp1.append(action)
                fringe.push((tmp0, tmp1))
util.raiseNotDefined()
```

تنها تفاوت آن با کد قبل این است که fringe از ساختمان داده صف استفاده می‌کند و همچنین علاوه بر چک کردن اینکه گره‌های جدید برای اضافه شدن به fringe، قبلا explore نشده‌اند، این را نیز چک می‌کند که گره در خود fringe هم از قبل نباشد. این کار به کمک یک flag و جستجو در fringe انجام می‌شود.

سوال 2)

Bfs به ازای حالت مبدا و Bfs به ازای حالت مقصد را اجرا میکنیم:

هرجا در fringe به یک گره یکسان دست پیدا کردیم ==> مسیر پیدا شده (مسیر از مبدا + مسیر از مقصد = مسیر نهایی)

تفاوت آن با bfs این است که به صورت موازی دوتا bfs در آن اجرا می‌شود و از bfs نیز سریع‌تر است زیرا هرچه bfs جلوتر می‌رود باید شعاع بزرگ‌تری را برای حالت هدف بگردد ولی در bbfs در شعاعی نصف، دو الگوریتم به همدیگر می‌رسند.

برای مسئله بیش از یک هدف، می‌توانیم از هر هدف یک bfs استارت بزنیم و یکی نیز از مبدا، هرجا bfs مبدا به هرکدام از bfs های مقصدها رسید، مسیری از مبدا به آن هدف را یافته‌ایم. اینقدر این کار را ادامه می‌دهیم تا الگوریتم مبدا به الگوریتم آخرین هدف هم برسد.

پیاده‌سازی 3)

```
"*** YOUR CODE HERE ***"
fringe = util.PriorityQueue()
explored = []
fringe.push(([problem.getStartState()], [], 0), 0)
while not fringe.isEmpty():
    expanded = fringe.pop()
    if problem.isGoalState(expanded[0][-1]): return expanded[1]
    explored.append(expanded[0][-1])
    successors = problem.getSuccessors(expanded[0][-1])
    for successor, action, stepCost in successors:
        if successor not in explored:
            flag = 0
            for (_, _, item) in fringe.heap:
                if successor == item[0][-1]:
                    if expanded[2] + stepCost > item[2]: flag = 1
                    elif expanded[2] + stepCost < item[2]:
                        fringe.update(item, -1)
                        fringe.pop()
            if flag == 0:
                tmp0 = expanded[0].copy()
                tmp1 = expanded[1].copy()
                tmp0.append(successor)
                tmp1.append(action)
                fringe.push((tmp0, tmp1, expanded[2] + stepCost), expanded[2] + stepCost)
util.raiseNotDefined()
```

تفاوت این با کد قبلی تنها در این است که ساختمان‌داده fringe از نوع صف اولویت می‌باشد و ساختمان‌داده هر گره نیز تغییر کرده است. Tuple سه‌تایی می‌شود. دو عنصر اول مشابه قبل است و عنصر سوم هزینه مسیری است که تا آنجا هزینه شده تا به آن گره برسیم. تغییر دیگر نیز این است که اگر گره‌ای که می‌خواهیم به fringe اضافه کنیم از قبل در fringe وجود داشته‌باشد، مقایسه‌ای بین هزینه آن دو گره می‌کنیم، اگر هزینه گره جدید بیشتر از گره‌ای که در fringe از قبل بود باشد، گره جدید را اضافه نمی‌کنیم ولی اگر کمتر بود، گره قدیمی را حذف می‌کنیم (این کار بدین صورت انجام می‌شود که با استفاده از تابع update که قبلا پیاده‌سازی کردیم، به آن گره اولویت 1- میدهیم (با این کار گره به آخر هیپ می‌آید) و سپس با پاپ کردن، آن گره از صف اولویت حذف می‌شود) و سپس گره جدید اضافه می‌شود.

سوال 3) گره با کمترین هزینه در ucs اول گسترش می‌یابد بنابراین اگر تابع هزینه ما بین هردو گره، 1 باشد (یا هر عدد ثابتی) در نتیجه آن مسیری که از گره‌های کمتری عبور کرده‌است، اول گسترش می‌یابد درنتیجه الگوریتم bfs را خواهیم داشت (stepCost = 1)

اگر همچنان stepCost = 1 باشد و مقدار عنصر سوم tuple نیز تغییر نکند و برابر هزینه داده‌شده برای رسیدن به آن گره باشد، ولی آرگومان دوم تابع پوش به fringe را (که مقدار priority ای است که صف اولویت با توجه به آن درخت هیپ را تشکیل می‌دهد و با هر گره اولویت کمتری داشته باشد انتها درخت هیپ می‌رود و با دستور پاپ، حذف می‌شود) بدین صورت تعریف کنیم:

1/node[3]

بدین ترتیب هر گره‌ای که هزینه بیشتری داشته باشد (چون stepCost = 1 یعنی آن گره‌ای که مسیراش طولانی‌تر است)، priority کمتری خواهد داشت و زودتر گسترش پیدا می‌کند پس در نتیجه الگوریتم dfs را خواهیم داشت.

پیاده‌سازی 4)

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

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

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

    "*** YOUR CODE HERE ***"
    res = math.sqrt(math.pow((position[0] - problem.goal[0]), 2) + math.pow((position[1] - problem.goal[1]), 2))
    return res
```

هیوریستیک منهتن جمع قدرمطلق مقادیر حرکات افقی و عمودی را می‌دهد.

هیوریستیک اقلیدسی ریشه دوم مثبت حاصل جمع مربعات مقادیر حرکات افقی و عمودی را می‌دهد (وتر).

```python
"*** YOUR CODE HERE ***"
fringe = util.PriorityQueue()
explored = []
fringe.push(([problem.getStartState()], [], 0), 0)
while not fringe.isEmpty():
    expanded = fringe.pop()
    if problem.isGoalState(expanded[0][-1]): return expanded[1]
    explored.append(expanded[0][-1])
    successors = problem.getSuccessors(expanded[0][-1])
    for successor, action, stepCost in successors:
        if successor not in explored:
            flag = 0
            for (_, _, item) in fringe.heap:
                if successor == item[0][-1]:
                    if expanded[2] + stepCost > item[2]: flag = 1
                    elif expanded[2] + stepCost < item[2]:
                        fringe.update(item, -1)
                        fringe.pop()
            if flag == 0:
                tmp0 = expanded[0].copy()
                tmp1 = expanded[1].copy()
                tmp0.append(successor)
                tmp1.append(action)
                fringe.push((tmp0, tmp1, expanded[2] + stepCost), expanded[2] + stepCost + heuristic(successor, problem))
util.raiseNotDefined()
```

تفاوت این کد با قبلی تنها در این است که مقدار priority هنگام پوش کردن به fringe علاوه بر تابع هزینه، مقدار تابع هیوریستیک نیز با آن جمع می‌شود.

سوال 4)

DFS: مسیر غیربهینه در تعداد گسترش زیاد.

BFS: مسیر بهینه در تعداد گسترش زیاد.

UCS: در اینجا مانند BFS عمل می‌کند.

*A: مسیر بهینه در تعداد گسترش کمتر.

پیاده‌سازی 5)

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    stp = self.startingPosition
    return (stp, [])
    util.raiseNotDefined()

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"
    if state[0] in self.corners:
        if state[0] not in state[1]: state[1].append(state[0])
        if len(state[1]) == 4: return True
        else: return False
    return False
    util.raiseNotDefined()
```

ساختمان داده هر گره ما به صورت یک tuple دوتایی است که اولین عنصر مختصات و دومی لیستی از گوشه‌هایی که گره ما تا الان بازدید کرده‌است می‌باشد. تابع getStartState حالت شروع را برمی‌گرداند. تابع isGoalState در صورتی که حالت گل را بگیرد مقدار True برمی‌گرداند. بدین صورت کار می‌کند که برسی می‌کند مختصات ما در لیست گوشه‌ها هست یا نه. اگر نبود در نتیجه در حالت هدف نیستیم و False برمی‌گرداند. اگر بود: اگر گوشه ما از قبل در لیست گوشه‌های بازدیدشده نبود، آن را به لیست گوشه‌های بازدیدشده اضافه می‌کنیم. حال اگر لیست گوشه‌های بازدیدشده ما، 4 عضو داشته باشد یعنی تمام گوشه‌ها را بازدید کرده‌ایم و این همان حالت گل مدنظر ماست و درنتیجه مقدار True برمی‌گردانیم. در غیر این صورت باز هم False برمی‌گردانیم زیرا یعنی هنوز گوشه‌ای هست که بازدید نکرده‌ایم.

```python
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]

        "*** YOUR CODE HERE ***"
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            exploredCorners = state[1].copy()
            if (nextx, nexty) in self.corners and (nextx, nexty) not in exploredCorners: exploredCorners.append((nextx, nexty))
            successors.append((((nextx, nexty), exploredCorners), action, 1))

    self._expanded += 1 # DO NOT CHANGE
    return successors
```

این تابع باید گره‌هایی که می‌توان از حالت فعلی به آنها رفت را برگرداند. در هرکدام از 4 جهت حرکت ممکن بررسی می‌کنیم مختصات جدید دیوار هست یا خیر. اگر نبود: اگر مختصات جدید مدنظر یکی از گوشه‌ها باشد و قبلا بازدید نشده باشد، در گره جدیدی که ایجاد می‌کنیم و به لیست successors اضافه می‌کنیم، آن گوشه را به لیست گوشه‌های بازدیدشده آن گره اضافه می‌کنیم (این کار روی یک کپی رخ می‌دهد زیرا نمی‌خواهیم تغییرات روی آن، بر روی لیست گوشه‌های بازدیدشده گره گسترش داده شده منعکس شود). پس از بررسی هر 4 جهت، در نهایت لیست successors را برمی‌گردانیم.

پیاده‌سازی 6)

```
"*** YOUR CODE HERE ***"
cornersToExplore = []
for corner in corners:
    if corner not in state[1]: cornersToExplore.append(corner)
from itertools import permutations
possibleRoutes = list(permutations(cornersToExplore))
possibleRoutesCost = []
for r in possibleRoutes:
    currentPosition = state[0]
    tmp = list(r)
    tmpRes = 0
    for i in tmp:
        tmpRes += util.manhattanDistance(currentPosition, i)
        currentPosition = i
    possibleRoutesCost.append(tmpRes)
return min(possibleRoutesCost)
```

ابتدا در یک حلقه یک لیست از گوشه‌هایی که هنوز بازدید نکرده‌ایم را درست می‌کنیم. سپس یک لیست از تمام جایگشت‌های ممکن آن تشکیل می‌دهیم. یک لیست هم برای نگه‌داری هزینه مسیرها. سپس هزینه حداقلی منهتن از حالت فعلی به اولین عنصر جایگشت، سپس از اولین عنصر به احیانا دومی و... را حساب میکنیم و جمع می‌کنیم. با این کار برای هر جایگشت هزینه حداقلی ممکن محاسبه می‌شود. سپس از بین این هزینه‌ها، کمترین آنها را انتخاب کرده و به عنوان مقدار هیوریستیک بازگردانده.

سوال 6) هیوریستیک بالا توضیح داده‌شد. هیوریستیک هنگامی سازگار نیست که با یک حرکت از حالتی به حالت دیگر (که در اینجا همواره هزینه 1 دارد رفتن به خانه مجاور)، مقدار هیوریستیک بیش از 1 در حالت جدید کمتر شود. که به‌هیچ وجه چنین چیزی ممکن نیست زیرا حرکتی که می‌کنیم در خوشبینانه‌ترین حالت در جهت مسیر حداقلی‌ای است که هیوریستیک آن را در نظر گرفته‌است و در این حالت نیز که هیوریستیک بیشترین کاهش را خواهد داشت، فقط به مقدار 1 کاهش می‌یابد نه بیش از آن.

پیاده‌سازی 7)

```
"*** YOUR CODE HERE ***"
return len(foodGrid.asList())
```

مقدار هیوریستیک ما، تعداد غذاهای موجود در صفحه را برمی‌گرداند. حالاتی که غذاهای کمتری در صفحه هستند به 0 نزدیک‌تر اند در نتیجه، الگوریتم جستجو ما به سمت هدف که خوردن تمام غذا هاست، هدایت می‌شود.

سوال 7) هیوریستیک بالا توضیح داده‌شد. سازگار هم می‌باشد زیرا در یک حرکت حداکثر یک غذا می‌توان خورد و درنتیجه هیوریستیک از حالتی به حالت دیگر، حداکثر به مقدار 1 کم می‌شود نه بیشتر.

سوال 7) هیوریستیک قسمت (6) به مقدار واقعی هزینه نزدیک‌تر است تا قسمت (7)، بنابراین به میزان بهتری پکمن را به سمت غذاها هدایت می‌کند. هیوریستیک قسمت (7) به ازای تعداد غذا یکسان همواره یک مقدار را می‌دهد که این یعنی فقط وقتی که پکمن در همسایگی یک غذا است، آن را به سمت خوردن غذا تشویق می‌کند و هنگامی که پکمن در همسایگی غذایی نباشد، در حقیقت هیوریستیک ما کاری نمی‌کند و الگوریتم ما UCS می‌شود. ولی هیوریستیک (6) اینطور نیست و پکمن را به سمت غذا هدایت می‌کند.

پیاده‌سازی 8)

```
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.ucs(problem)
    util.raiseNotDefined()
```

با UCS کوتاه‌ترین مسیر تا غذا را پیدا می‌کند.

سوال 8) نزدیک‌ترین غذا به پکمن در انتها یک بن‌بست قرار دارد. بدین ترتیب پکمن اول می‌رود و آن را می‌خورد و سپس برمی‌گردد تا بقیه را بخورد و این رفت و برگشت در بن‌بست بهینه نیست. در مسیر بهینه غذا در بن‌بست را باید آخر بخورد.