گزارش پروژه اول هوش مصنوعی

آوا میرمحمدمهدی 810199501

نحوه‌ی کلی مدل کردن مساله در هر سه الگوریتم:

در این مساله گرافی که برای حل مساله تشکیل می‌دهیم با گراف ورودی مساله متفاوت است زیرا حالات زیادی وجود دارد که در یک نود خاص در گراف ورودی قرار داشته باشیم ولی وضعیت ما متفاوت باشد، این تفاوت می‌تواند در تعداد مریدهای راضی‌شده، تعداد دستورپخت‌های دیده شده، تعداد عبور از هر راس صعب العبور و یا مقدار هزینه‌ی صرف شده در رسیدن به آن نود باشد. ما از initial state که در ادامه به توضیح آن می‌پردازیم شروع می‌کنیم و با توجه به اکشن‌هایی که در هر state می‌توانیم انجام دهیم، اقدام به ساختن استیت‌های پیش رو می‌کنیم. هر استیت ما، یک نمونه از کلاسی (با نام State) است که در آن تعداد عبور از راس‌های صعب العبور، دستورپخت‌های دیده شده، مریدهای راضی شده، لوکیشنی که روی آن قرارداریم و پدر آن استیت (یعنی استیتی که پیش از استیت فعلی در آن بودیم) نگهداری می‌شود؛ در این کلاس متغیری به نام remainingTime نیز وجود دارد که به صورت پیش فرض مقدار آن صفر است و در صورتی که در استیتی که در آن اس صعب العبور قرار دارد باشیم، مقدار آن برابر با تعداد دفعات دیده شدن آن راس صعب العبور می‌شود. کاربرد این متغیر نیز در ادامه ذکر می‌شود. در ابتدا تعداد دیده شدن هر یک از راس‌های صعب العبور 0 است و برای هر دستورپخت نیز مقدار False قرار دارد. (اگر دستورپختی دیده شود مقدار آن True می‌شود)؛ همچنین مقدار همه‌ی مریدها نیز False است؛ مقدار یک مرید درصورتی True می‌شود که همه‌ی دستورپخت‌های آن مرید دیده شده باشند.

```
class State:

    def __init__(self, n, loc, morids):
        self.saaboloburVisited = [0 for x in range(n)]
        self.morids = morids
        self.recipesVisited = [False for x in range(n)]
        self.loc = loc
        self.father = None
        self.remainingTime = 0
        self.cost = 0
```

Initial state: لوکیشن این state همان مقداری است که ورودی مساله به عنوان مکان ابتدایی سید می‌دهد؛ یعنی در ابتدا ما یک نمونه از کلاس State می‌سازیم و مقدار متغیرهای آن را درصورت لزوم آپدیت می‌کنیم.

نحوه آپدیت کردن یک استیت: اگر در لوکیشن استیتی که در آن قرار داریم، راس صعب العبور باشد، تعداد دیده شدن آن راس صعب العبور را در آن استیت اضافه می‌کنیم و همچنین remainingTime آن را برابر با تعداد دیده شدن آن راس صعب العبور قرار می‌دهیم؛ اگر در لوکیشن استیتی که در آن قرار داریم، دستورپخت باشد، مقدار لیست دستورپخت آن استیت را به ازای آن لوکیشن True می‌کنیم؛ و در نهایت اگر در لوکیشن استیت فعلی، مرید باشد، درصورتی که تا کنون تمامی دستورپخت‌های آن مرید را دیده باشیم، مقدار آن را True می‌کنیم.

```
if (saabolobur[startLoc]):
    startState.updateSaabolobur(startLoc)
if (recipes[startLoc]):
    startState.updateRecipes(startLoc)
if (moridCountInEachLoc[startLoc] > 0):
    startState.updateMorids(startLoc, moridRecipes[startLoc]['recipes'])
```

Action: تعریف اکشن در این مساله درواقع رفتن به همسایه های یک استیت است که در ورودی مساله با عنوان یال به ما داده می‌شود؛ وقتی در استیت A قرار داریم، تعداد اکشن‌های ما برای خارج شدن از آن استیت برابر با تعداد همسایه‌های استیت A است. به ازای هر همسایه، یک نمونه از کلاس State می‌سازیم، پدر این استیت، استیت A است. فرزند A، ویژگی‌های A از جمله مقدار هزینه‌ی صرف شده برای رسیدن به آن، تعداد دیده شدن هر راس صعب العبور، دستورپخت‌های دیده شده و مریدهای راضی شده را از A به ارث می‌برد و هزینه‌ی رسیدن به آن را نیز به علاوه یک می‌کنیم، سپس با توجه به لوکیشنی که در آن قرار دارد، همانطور که در بالا ذکر شد درصورت لزوم، آپدیت می‌شود.

```
for childLoc in edges[currentState.loc]:
    childState = State(n, childLoc, morids)
    childState.inheritFromFather(currentState)
    childState.increaseCost()
    visitedStatesCount += 1
    if (saabolobur[childLoc]):
        childState.updateSaabolobur(childLoc)
    if (recipes[childLoc]):
        childState.updateRecipes(childLoc)
    if (moridCountInEachLoc[childLoc] > 0):
        childState.updateMorids(childLoc, moridRecipes[childLoc]['recipes'])
```

Goal State: اگر در استیتی تمام مریدها راضی شده باشند، به هدف رسیده‌ایم پس در goal test اگر به ازای تمامی مریدها، همه‌ی رسپی‌های آنها دیده شده بود، به هدف رسیده‌ایم.

```
def goalTest(self):
    for x in self.morids.keys():
        for morid in self.morids[x]:
            if morid == False:
                return False
    return True
```

توضیح الگوریتم BFS:

در این الگوریتم مجموعه frontier ما یک صف است. ابتدا initial state را به این صف اضافه می‌کنیم و سپس تا زمانی که frontier ما خالی نشده است مراحل ذکر شده را انجام می‌دهیم: ابتدا یک عضو از مجموعه frontier، pop می‌کنیم، درصورتی که remainingTime این عضو برابر با 0 نبود یعنی در مکان یک راس صعب العبور قرار داریم؛ برای اینکه به اندازه‌ی کافی در این استیت بمانیم، یک واحد از remainingTime آن کم می‌کنیم و به هزینه‌ی رسیدن به آن، یک واحد اضافه می‌کنیم و مجددا آن را به مجموعه frontier اضافه می‌کنیم، به این‌کار درواقع به جای انجام اکشن (اضافه کردن استیت‌های همسایه)، با اضافه کردن دوباره‌ی آن به مجموعه‌ی frontier باعث می‌شویم تا در عمق پایین تری از درخت BFS به بسط دادن آن برسیم و در نتیجه هزینه‌ی استیت هایی که بعد از این استیت به آن ها می‌رویم نیز بیشتر می‌شود. درصورتی که remainingTime یک استیت برابر با 0 بود، (در آن راس صعب العبور وجود نداشت و یا به اندازه‌ی کافی در آن در انتظار مانده باشیم)، همسایه‌های آن نود را بررسی می‌کنیم و ویژگی‌های پدر را برای هر یک به ارث می‌گذاریم و آن را آپدیت می‌کنیم؛ در اینجا تست goal نیز انجام می‌شود و درصورتی که به هدف رسیده باشیم استیت فعلی به همراه تعداد استیت‌های دیده شده را برمی‌گردانیم.

```
while (len(frontier)):
    currentState = frontier.popleft()
    if (currentState.remainingTime == 0):
        for childLoc in edges[currentState.loc]:
            childState = State(n, childLoc, morids)
            childState.inheritFromFather(currentState)
            childState.increaseCost()
            visitedStatesCount += 1
            if (saabolobur[childLoc]):
                childState.updateSaabolobur(childLoc)
            if (recipes[childLoc]):
                childState.updateRecipes(childLoc)
            if (moridCountInEachLoc[childLoc] > 0):
                childState.updateMorids(childLoc, moridRecipes[childLoc]['recipes'])
            if (childState.goalTest() == True):
                return childState, visitedStatesCount
            if (isInVisited(childState, visited)== False):
                visited.append(childState)
                frontier.append(childState)
    else:
        currentState.reduceRemainingTime()
        currentState.increaseCost()
        frontier.append(currentState)
```

برای کاهش زمان اجرای برنامه یک لیست از استیت‌های دیده شده نگهداری می‌کنیم و درصورتی که استیت فرزند از نظر لوکیشن، دستورپخت‌های دیده شده، مریدهای راضی‌شده و تعداد دفعات عبور از راس‌های صعب العبور با یکی از استیت‌های موجود در لیست استیت‌های دیده شده برابر بود، آن را به مجموعه‌ی frontier اضافه نمی‌کنیم و در غیر این‌صورت این استیت به مجموعه استیت‌های دیده شده و مجموعه frontier اضافه می‌شود. الگوریتم BFS پاسخ بهینه را به ما می‌دهد.

```
def isSameState(self , curState):
    if (self.morids == curState.morids and self.remainingTime == curState.remainingTime and self.loc == curState.loc and
        self.saaboloburVisited == curState.saaboloburVisited and self.recipesVisited == curState.recipesVisited ):
        return True
    return False
```

پاسخ تست1:

```
BFS time:  0.002716064453125
The numbers of visited states is:   150
Final Cost is:   8
Path:   1 -> 3 -> 4 -> 5 -> 7 -> 10 -> 11 -> 9 -> 8
```

پاسخ تست2:

```
BFS time: 26.749454259872437
The numbers of visited states is:  40194
Final Cost is:  12
Path:  28 -> 19 -> 13 -> 3 -> 11 -> 24 -> 9 -> 23 -> 28 -> 23 -> 5 -> 7 -> 29
```

پاسخ تست3:

```
BFS time: 33.369579553604126
The numbers of visited states is:  32462
Final Cost is:  21
Path:  40 -> 42 -> 38 -> 24 -> 31 -> 45 -> 30 -> 48 -> 41 -> 18 -> 1 -> 19 -> 43 -> 49 -> 47 -> 49 -> 9 -> 34 -> 25 -> 50 -> 12 -> 16
```

توضیح الگوریتم IDS:

در این الگوریتم از عمق 1 شروع می‌کنیم و DFS را با این عمق اجرا می‌کنیم و درصورتی که به جواب برسیم آن را برمی‌گردانیم و در غیر این صورت مجددا DFS را با عمق یک واحد بیشتر فراخوانی می‌کنیم.

```
while (True):
    startStateCpy = deepcopy(startState)
    findAnswer, goal, visitedStatesCount = DFS(startStateCpy, n, edges, saabolobur, recipes, morids , moridRecipes, depth, moridCountInEachLoc)
    if findAnswer:
        return goal, visitedStatesCount
    depth += 1
```

در تابع DFS مجموعه frontier ما یک stack است که تا زمانی که خالی نشده از آن pop می‌کنیم (اطلاعات در این استک به صورت یک جفت از عمق استیت و خود استیت نگهداری می‌شود). اگر عمق استیت بیشتر از حد تعیین شده برای تابع DFS بود آن را در نظر نمی‌گیریم. در ادامه مانند BFS و با همان استدلال درصورتی که مقدار remainingTime برابر با 0 نبود مجددا آن استیت را این بار با عمق بیشتر (و کم کردن از remainingTime) به استک اضافه می‌کنیم و درصورتی که remainingTime برابر با 0 باشد، مانند BFS همسایه‌های آن استیت

را در نظر می‌گیریم و تست goal را روی آنها انجام می‌دهیم و آن را به استک اضافه می‌کنیم. الگوریتم DFS به تنهایی ممکن است پاسخ بهینه را تولید نکند (چون ممکن است ابتدا عمق طولانی تری را بررسی کند و جواب موجود در آن عمق را برگرداند درصورتی که جواب بهینه در عمق کمتر موجود باشد) ولی در اینجا با استفاده از IDS و تعیین عمق در هر مرحله، جواب بهینه تولید می‌شود.

```
stack = deque()
stack.append((startState, 0))
visitedStatesCount = 1
while (len(stack)):
    currentState, curDepth = stack.pop()
    if (curDepth > depth):
        continue
    if (currentState.remainingTime == 0):
        for childLoc in edges[currentState.loc]:
            childState = State(n, childLoc, morids)
            childState.inheritFromFather(currentState)
            childState.increaseCost()
            visitedStatesCount += 1
            if (saabolobur[childLoc]):
                childState.updateSaabolobur(childLoc)
            if (recipes[childLoc]):
                childState.updateRecipes(childLoc)
            if (moridCountInEachLoc[childLoc] > 0):
                childState.updateMorids(childLoc, moridRecipes[childLoc]['recipes'])
            if (childState.goalTest() == True):
                return True, childState, visitedStatesCount
            stack.append((childState, curDepth + 1))
    else:
        currentState.reduceRemainingTime()
        stack.append((currentState, curDepth + 1))
return False, None, visitedStatesCount
```

پاسخ تست1:

```
IDS time: 0.012642383575439453
The numbers of visited states is:  1163
Final Cost is:  8
Path:  1 -> 3 -> 4 -> 5 -> 7 -> 10 -> 11 -> 9 -> 8
```

پاسخ تست2:

```
IDS time: 0.024193763732910156
The numbers of visited states is:  999
Final Cost is:  8
Path:  9 -> 10 -> 2 -> 4 -> 12 -> 3 -> 7 -> 5 -> 8
```

پاسخ تست3:

```
IDS time: 0.8447091579437256
The numbers of visited states is:  62147
Final Cost is:  13
Path:  13 -> 11 -> 10 -> 3 -> 2 -> 6 -> 12 -> 5 -> 9 -> 4 -> 1 -> 13 -> 11 -> 10
```

توضیح الگوریتم *A:

توضیح heuristic انتخاب شده: heuristic تعریف شده از مجموع تعداد مریدهای راضی نشده و تعداد دستورپخت‌های دیده نشده بدست می‌آید؛ توجه شود که در صورت که در یک لوکیشن، هم دستورپخت دیده نشده و هم مرید راضی نشده وجود داشته باشد، فقط یک واحد به مقدار heuristic ما اضافه می‌شود. heuristic تعریف شده consistent است چون در بهترین حالت دستورپخت‌های دیده نشده و همچنین مریدهای راضی نشده در استیت همسایه‌ی استیت فعلی قرار دارد که هزینه‌ی رسیدن به هرکدام ار همسایه‌ها به اندازه یک واحد است و همچنین در این heuristic اگر مرید و دستورپخت مربوط به آن، هردو دیده نشده باشند و در استیت همسایه باشند، چون تنها موجب اضافه شدن یک واحد به heuristic شده‌اند پس همچنان هزینه‌ی صرف شده‌ی واقعی بزرگتر یا مساوی مقدار هزینه‌ی تعریف شده با heuristic است. در حالت دیگر اگر یکی ار نوادگان A را B بنامیم به طوری که هزینه رسیدن از A به استیت هدف a و هزینه ی رسیدن از B به استیت هدف b باشد؛ می‌دانیم طبق تعریف هیوریستیک ما، a >= x (x مجموع مریدان راضی نشده و دستور پخت‌های دیده نشده در استیت A است) و همینطور b >= y (y مجموع مریدان راضی نشده و دستور پخت‌های دیده نشده در استیت B است) پس هزینه ی رسیدن از A به B برابر x – y + alfa می‌شود. طبق تعریف هیوریستیک h(A) = x و h(B) = y است و نتیجه می‌گیریم cost(A to B) >= h(A) – h(B) است پس هیوریستیک ما consistent است.

```python
def caclHeuristic(self,recipes, moridCountInEachLoc):
    h = 0
    for loc in self.morids.keys():
        for morid in self.morids[loc]:
            if morid == False:
                h += 1
                break
    for rec in recipes:
        if rec == True and self.recipesVisited[rec] == False and moridCountInEachLoc[rec] == 0:
            h += 1
    return h
```

در اینجا مجموعه frontier به صورت یک priority queue است که با توجه به مقدار evaluation function اولویت بندی صورت می‌گیرد و درصورت برابر بودن این مقدار برای دو استیت، اولویت بر اساس هزینه‌ی رسیدن از استیت آغازین تا استیت فعلی است.

```python
def __lt__(self, other):
    return self.cost < other.cost
```

تست goal در زمان pop کردن از صف اولویت انجام می‌شود و برای اضافه کردن یک استیت به صف الویت و لیست استیت‌های دیده شده، درصورتی که این استیت در قبل با مقدار evaluation function بیشتری وجود داشت، آن را با مقدار جدید اضافه می‌کنیم. الگوریتم *A جواب بهینه را تولید می‌کند زیرا تست goal در محل مناسبی انجام می‌شود و همچنین heuristic در نظر گرفته شده، consistent است.

```
while (len(heapList)):
    currentF, currentState = heapq.heappop(heapList)
    if (currentState.goalTest() == True):
        return currentState, visitedStatesCount
    if (currentState.remainingTime == 0):
        for childLoc in edges[currentState.loc]:
            childState = State(n, childLoc, morids)
            childState.inheritFromFather(currentState)
            childState.increaseCost()
            visitedStatesCount += 1
            if (saabolobur[childLoc]):
                childState.updateSaabolobur(childLoc)
            if (recipes[childLoc]):
                childState.updateRecipes(childLoc)
            if (moridCountInEachLoc[childLoc] > 0):
                childState.updateMorids(childLoc, moridRecipes[childLoc]['recipes'])
            childF = childState.calcFfunction(recipes, moridCountInEachLoc)
            if (isInVisited(childState, visited) == False):
                visited.append(childState)
                heapq.heappush(heapList, (childF, childState))
    else:
        currentState.reduceRemainingTime()
        currentState.increaseCost()
        currentF =  currentState.caclHeuristic(recipes, moridCountInEachLoc) + currentState.cost
        heapq.heappush(heapList, (currentF, currentState))
```

```
def isSameState(self , curState):
    if (self.morids == curState.morids and self.remainingTime == curState.remainingTime and self.loc == curState.loc and
        self.saaboloburVisited == curState.saaboloburVisited and self.recipesVisited == curState.recipesVisited):
        if (self.F > curState.F):
            return False
        return True
    return False
```

پاسخ تست1 با A*:

```
A* time: 0.0031085014343261 72
The numbers of visited states is:  202
Final Cost is:  8
Path:  1 -> 3 -> 4 -> 5 -> 7 -> 10 -> 11 -> 9 -> 8
```

پاسخ تست2 با A*:

```
A* time: 19.67703151702881
The numbers of visited states is:  33684
Final Cost is:  12
Path:  28 -> 19 -> 13 -> 3 -> 11 -> 24 -> 9 -> 22 -> 28 -> 22 -> 5 -> 7 -> 29
```

پاسخ تست3 با A*:

```
A* time: 24.09947443008423
The numbers of visited states is:  27440
Final Cost is:  21
Path:  40 -> 42 -> 38 -> 24 -> 31 -> 45 -> 30 -> 48 -> 41 -> 18 -> 1 -> 19 -> 43 -> 49 -> 47 -> 49 -> 9 -> 34 -> 25 -> 50 -> 12 -> 16
```

الگوریتم Weighted A*:

چون هیوریستیک در نظر گرفته شده، consistent است پس هزینه‌ی تخمین زده همواره کوچکتر یا مساوی هزینه واقعی است؛ ما در اینجا با ضرب کردن یک ضریب در مقدار هیوریستیک سعی در نزدیک تر کردن آن به مقدار واقعی داریم. با ضرب یک مقدار در هیوریستیک و تغییر آن، هیوریستیک دیگر consistent نیست و در نتیجه الگوریتم Weighted A* لزوما جواب بهینه تولید نمی‌کند؛ البته در مثال های زیر که ضریب در نظر گرفته شده برابر با 1.2 و 1.7 است جواب تولید شده صحیح است. اگرچه این الگوریتم لزوما جواب بهینه تولید نمی‌کند ولی زمان

اجرای آن می‌تواند نسبت به A* کاهش چشمگیری داشته باشد که در مقادیر ضریب داده شده این موضوع با ضریب 1.7 کاملا مشخص است. همچنین تعداد استیت‌های دیده شده نیز کمتر است.

پاسخ تست1 (ALPHA = 1,2) A*:

```
A* time: 0.0019562244415283203
The numbers of visited states is:  153
Final Cost is:  8
Path:  1 -> 3 -> 4 -> 5 -> 7 -> 10 -> 11 -> 9 -> 8
```

پاسخ تست2 (ALPHA = 1,2) A*:

```
A* time: 12.22197699546814
The numbers of visited states is:  24313
Final Cost is:  12
Path:  28 -> 30 -> 9 -> 24 -> 11 -> 3 -> 13 -> 23 -> 5 -> 7 -> 29 -> 22 -> 28
```

پاسخ تست3 (ALPHA = 1,2) A*:

```
A* time: 19.116877555847168
The numbers of visited states is:  23990
Final Cost is:  21
Path:  40 -> 42 -> 38 -> 24 -> 31 -> 45 -> 30 -> 48 -> 41 -> 18 -> 1 -> 19 -> 43 -> 49 -> 47 -> 49 -> 9 -> 34 -> 25
-> 50 -> 12 -> 16
```

پاسخ تست1 (ALPHA = 1.7) A*:

```
A* time: 0.0037622451782226562
The numbers of visited states is:  177
Final Cost is:  8
Path:  1 -> 3 -> 4 -> 5 -> 7 -> 10 -> 11 -> 9 -> 8
```

پاسخ تست2(ALPHA = 1.7) A*:

```
A* time: 12.046502351760864
The numbers of visited states is:  23652
Final Cost is:  12
Path:  28 -> 30 -> 9 -> 24 -> 11 -> 3 -> 13 -> 23 -> 5 -> 7 -> 29 -> 22 -> 28
```

پاسخ تست3 (ALPHA = 1.7) A*:

```
A* time: 15.630030393600464
The numbers of visited states is:  20933
Final Cost is:  21
Path:  40 -> 42 -> 38 -> 24 -> 31 -> 45 -> 30 -> 48 -> 41 -> 18 -> 1 -> 19 -> 43 -> 49 -> 47 -> 49 -> 9 -> 34 -> 25
-> 50 -> 12 -> 16
```

علاوه بر نکاتی که در هر قسمت ذکر شد، الگوریتم IDS درصورتی که پاسخ در عمق‌های کم مساله پیدا شود، مناسب است ولی در این مساله معمولا اینطور نیست و فراخوانی DFS با عمق‌های متفاوت و به صورت پی در پی باعث بالا رفتن زمان برنامه می‌شود. از لحاظ مصرف حافظه، الگوریتم IDS خطی و برای بقیه‌ی الگوریتم‌ها به صورت نمایی است.

تست1	پاسخ مساله (حداقل زمان لازم برای رساندن دیزی‌ها)	تعداد استیت‌های دیده شده	میانگین زمان اجرا
BFS	8	150	0.00169
IDS	8	1163	0.0135
A*	8	202	0.0038
Weighted A* 1 **ALPHA = 1.2**	8	153	0.0023
Weighted A* 2 **ALPHA = 1.7**	8	177	0.0022

تست2	پاسخ مساله (حداقل زمان لازم برای رساندن دیزی‌ها)	تعداد استیت‌های دیده شده	میانگین زمان اجرا
BFS	12	40194	26.855
IDS (easy test case)	8	999	0.0244
A*	12	33684	20.442
Weighted A* 1 **ALPHA = 1.2**	12	24313	12.787
Weighted A* 2 **ALPHA = 1.7**	12	23652	11.065

تست3	پاسخ مساله (حداقل زمان لازم برای رساندن دیزی‌ها)	تعداد استیت‌های دیده شده	میانگین زمان اجرا
BFS	21	32462	32.545
IDS (easy test case)	13	62147	0.906
A*	21	27440	23.485
Weighted A* 1 **ALPHA = 1.2**	21	23990	22.250
Weighted A* 2 **ALPHA = 1.7**	21	20933	14.250

برای قسمت weighted A*، آلفا (وزن قسمت هیوریستیک) را یک بار برابر با 1.2 و یک بار برابر با 1.7 در نظر می‌گیریم که در هر دو حالت میانگین زمان اجرا از A* کمتر است ولی با ALPHA = 1.7 زمان اجرا کمتر از ALPHA = 1.2 است.