بسم الله الرحمن الرحیم

گزارش کار پروژه دوم هوش مصنوعی

مهدی فرهنگ

روش پیاده سازی مسئله:

مطمئن ترین راهی که برای مدل سازی این مسئله به مسئله ی search به نظر میرسد، اینطور است: فضای نمونه خود را به گونه ای تعریف میکنیم که تمامی حالاتی که میتوان با چیدن وزیر ها روی تخته به آن رسید فضای ماست. البته ما در حل تمامی فضای نمونه را طی نمیکنیم و در هر کدام از راه حل ها، تعداد خاصی از حالات فضای نمونه را بررسی میکنیم.

هدف این است که بتوانیم به گونه ای وزیر ها را بچینیم که هیچ کدام از وزیر ها همدیگر را تهدید نکنند.

برای این کار از الگوریتم های مختلفی استفاده میکنیم. توضیح تمام بخش های code ای که نوشته شده را در پایین میاوریم

توضیح الگوریتم ها:

پیاده سازی مجموعه:

ابتدا یک توضیحی درباره پیاده سازی تخته شطرنج و جایگاه وزیر ها میدهیم. وزیر ها را به صورت لیستی هشت تایی از لیست های دوتایی در نظر میگیریم. هر وزیر خود یک لیست است که شامل دو عنصر است که به ترتیب x و y (سطر و ستون) آنرا به ما میدهد.

پیاده سازی توابع مختلف مورد نیاز:

توابعی مورد نیاز ماست برای حل این مسئله:

۱. تابعی برای محاسبه ی تعداد تهدید هایی که وزیرها نسبت به یکدیگر دارند. این تابع در حالت کلی فقط برای informed search به کار میرود، اما برای uninformed search ها نیز نیاز داریم که

بدانیم آیا مسئله به اتمام رسیده یا نه، یعنی نیاز داریم بفهمیم که آیا در حالتی هستیم که هیچ دو وزیری همدیگر را تهدید میکنند یا خیر. برای این کار، از همین تابع استفاده میکنیم.

دو پیاده سازی مختلف برای این تابع به کار برده ایم.

1.1. num_of_threats(data)

این تابع، تهدید ها را میشمرد، اما ویژگی ای که دارد این است که تهدید ها را به گونه ای میشمرد که اگر چند وزیر همدیگر را به صورت دوری تهدید کنند (وزیر ۱ وزیر ۲ را تهدید کند، وزیر ۲ وزیر ۳ را تهدید کند، وزیر ۳ وزیر ۱ را تهدید کند - هر سه آنها در یک ردیف باشند)، تهدید های تکراری را نمیشمرد. ینی تهدید ها را به صورت مستقل از هم میشمرد.

```
5  def num_of_column_and_row_threats(my_data):
6      column_threats = 0
7      row_threats = 0
8      for i in range(n_queens):
9          num_of_queens_column = 0
10         num_of_queens_row = 0
11         for j in range(n_queens):
12             if (my_data[j][0] == i + 1):
13                 num_of_queens_row += 1
14             if (my_data[j][1] == i + 1):
15                 num_of_queens_column += 1
16         if (num_of_queens_row > 1):
17             row_threats += num_of_queens_row - 1
18         if (num_of_queens_column > 1):
19             column_threats += num_of_queens_column - 1
20     return column_threats + row_threats
21
22 # we could count number of rows or columns without queens and the answer would have been the
23
24 def num_of_diameter_threats(my_data):
25     threats = 0
26     blacklist = []
27     for i in range(n_queens):
28         for j in range(i + 1, n_queens):
29             if (j in blacklist):
30                 continue
31             if (abs(my_data[i][0] - my_data[j][0]) == abs(my_data[i][1] - my_data[j][1])):
32                 threats += 1
33                 blacklist.append(j)
34 # if the abs wasn't there, we could get num of main diameter threats
35 # if the abs wasn't there and a minus was behind the right part of equation, we could get nu
36     return threats
37
38 def num_of_threats(my_data):
39     return num_of_diameter_threats(my_data) + num_of_column_and_row_threats(my_data)
```

تابع به گونه ای پیاده سازی شده که ابتدا در یک تابع تهدید های سطری و ستونی و در تابعی دیگر تهدید های قطری را محاسبه میکند. شیوه ی انجام آن در متن واضح است.

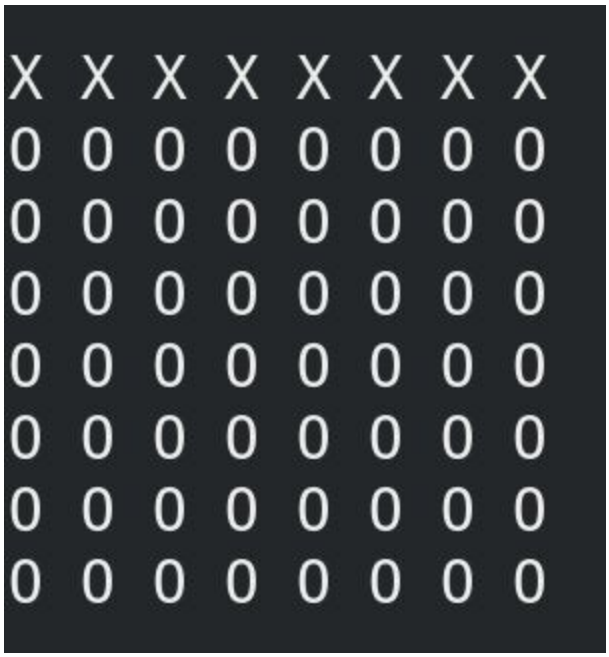
2.2. danger(data)

این تابع دقیقا بررسی میکند که هر وزیر چند بار تهدید میشود و همه را با هم جمع میکند.

```
41 def danger(my_data):
42     dangers = 0
43     for i in range(n_queens):
44         for j in range(i + 1, n_queens):
45             if(my_data[i][0] == my_data[j][0] or my_data[i][1] == my_data[j][1] or abs(my_data[i][0] - my_data[j][0]) == abs(my_data[i][1] - my_data[j][1
46                 dangers += 1
47     return int(dangers)
48
```

پیاده سازی آن نیز به گونه ای است که برای هر وزیر، تعداد تهدید ها را میشمرد. فقط معلوم است که تکراری ها را نمیشماریم

برای مشاهده فرق این دو تابع، به مثال زیر توجه کنید:



در این مثال،توابع بالا جواب های زیر را میدهند:

num_of_threats(data) = 7

danger(data) = 28

در واقع در مقایسه ی دو حالت مختلف، هر دو تابع به درستی بررسی میکنند که تعداد تهدید های کدام یک بیشتر است و فرقی نمیکند که از کدام یک استفاده کنیم.

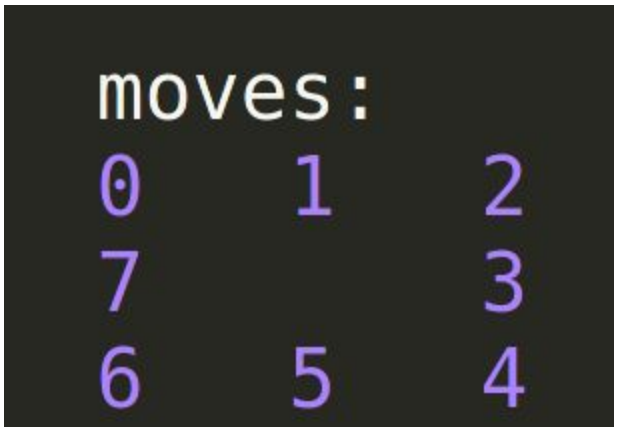
۲. تابعی برای نمایش جایگاه وزیر ها روی تخته

این تابع به سادگی پیاده سازی میشود و توضیح خاصی نیاز ندارد. در پایین مشاهده میکنیم که با دو حلقه تو در تو این تابع کار ما را انجام میدهد.

```
136
137 def print_grid(my_data):
138     for i in range(n_queens):
139         for j in range(n_queens):
140             flag = False
141             for k in range(n_queens):
142                 if (my_data[k][0] == i + 1 and my_data[k][1] == j + 1):
143                     print('X', end = ' ')
144                     flag = True
145                     break
146             if (flag == False):
147                 print('0', end = ' ')
148         print()
149
```

۳. توابع مورد نیاز برای حرکت دادن وزیر ها:

برای این کار به چند تابع نیاز داریم. ابتدا حرکت های مختلف وزیر ها را تعریف میکنیم. هر وزیر ۸ حرکت مختلف میتواند انجام دهد.که در شکل زیر میبینیم:



در نظر داریم که وزیر در وسط ایستاده است.

حال برای پیاده سازی این تابع باید بررسی کنیم که آیا حرکت مورد نظر مجاز است یا خیر. در دو صورت حرکت مجاز نخواهد بود، یک این که وزیر با حرکت مورد نظر از صفحه شطرنج خارج شود، و دوم آنکه حرکت مورد نظر باعث شود دو وزیر در یک خانه قرار گیرند. پس یک تابع مینویسیم که ابتدا بررسی میکند که آیا حرکت مورد نظر مجاز است یا خیر، و اگر مجاز بود،حرکت را انجام میدهد.

پیاده سازی تابع (move_if_possible) دقیقا ۸ حالت دارد. یک حالت برای هر کدام از حرکت های گفته شده. در پایین یکی از این ها را مشاهده میکنید. ۷ حالت دیگر به حالت مشابه به دست آمده است.

```
elif (move_no == 2):
    my_data[queen_no] = [my_data[queen_no][0] - 1, my_data[queen_no][1] + 1]
    if (my_data[queen_no][0] < 1 or my_data[queen_no][1] > num_of_moves or is_duplicate_element(my_data, my_data[queen_no]
        my_data[queen_no] = temp
        return False
    return True
```

در پیاده سازی این تابع نیاز داریم که بفهمیم آیا از یک عنصر مورد نظر در یک لیست بیش از یک عنصر داریم یا خیر؟ تابع is_duplicate_element این کار را برای ما انجام میدهد

```
55  def is_duplicate_element(my_data, element):
56      num_of_existed = 0
57      for i in my_data:
58          if (i == element):
59              if (num_of_existed > 0):
60                  return True
61              num_of_existed += 1
62      return False
```

در پیاده سازی الگوریتم ها، به جایی برمیخوریم که میدانیم حرکت مجاز است. با توجه به این که زمان اجرای برنامه برای ما بسیار مهم است، تابعی مینویسیم که بدون درنظر گرفتن مجاز بودن حرکت،وزیر را جابجا کند. این تابع move نام دارد

```
115 def move(my_data, queen_no, move_no):
116     if (move_no == 1):
117         my_data[queen_no] = [my_data[queen_no][0] - 1, my_data[queen_no][1]]
118     elif (move_no == 3):
119         my_data[queen_no] = [my_data[queen_no][0], my_data[queen_no][1] + 1]
120     elif (move_no == 5):
121         my_data[queen_no] = [my_data[queen_no][0] + 1, my_data[queen_no][1]]
122     elif (move_no == 7):
123         my_data[queen_no] = [my_data[queen_no][0], my_data[queen_no][1] - 1]
124     elif (move_no == 0):
125         my_data[queen_no] = [my_data[queen_no][0] - 1, my_data[queen_no][1] - 1]
126     elif (move_no == 2):
127         my_data[queen_no] = [my_data[queen_no][0] - 1, my_data[queen_no][1] + 1]
128     elif (move_no == 4):
129         my_data[queen_no] = [my_data[queen_no][0] + 1, my_data[queen_no][1] + 1]
130     elif (move_no == 6):
131         my_data[queen_no] = [my_data[queen_no][0] + 1, my_data[queen_no][1] - 1]
132
```

۴. تابع تولید فرزندان

همانطور که اشاره خواهیم کرد، بسیار نیاز داریم که بدانیم در هر کدام از حالت هایی که هستیم، با یک حرکت میتوانیم به چه حالت هایی برسیم. تابع generate_next_children این کار را برای ما میکند.

```
151 def copy_board(my_data):
152     temp = []
153     for i in range(n_queens):
154         temp.append([int(my_data[i][0]), int(my_data[i][1])])
155     return temp
156 def generate_next_children(my_data):
157     children = []
158     for i in range(n_queens):
159         for j in range(num_of_moves):
160             if(move_if_possible(my_data, i, j)):
161                 children.append(copy_board(my_data))
162                 move(my_data, i, (j + 4) % 8)
163     return children
```

این تابع با بررسی کردن تک تک حرکات بر روی تمامی وزیر ها، یک حرکت انجام میدهد و جواب را ذخیره میکند. طبیعتا باید حرکت بازگشت داده شود، و این که میدانیم این حرکت مجاز است به ما در سرعت اجرای برنامه کمک میکند.

تابع copy_board نیز در حین ساخت این تابع نیاز میشود. این تابع یک copy از وضعیت فعلی ما را ساخته و برمیگرداند.

توضیح الگوریتم های استفاده شده:

1. BFS:

در این الگوریتم، ابتدا تمامی حالت های با فاصله یک از حالت اولیه در نظر میگیریم، یعنی تمامی وزیر ها تمامی حرکت هایی که میتوانند انجام دهند را هر کدام یک حالت در نظر بگیریم. حداکثر ۶۴ حالت خواهیم داشت. (تابع generate_next_children این حالت ها را برای ما تولید میکند)

پس از حالت های با فاصله ۱، به حالت های با فاصله ۲ میرسیم. و به همین ترتیب پیش میرویم تا به جواب مورد نظر برسیم.

در زیر پیاده سازی این الگوریتم را میبینیم

```
213 def BFS(root_grid):
214     nodes = [root_grid]
215     v = {}
216     levels = [0]
217     moves = 0
218     while(True):
219         moves += 1
220         node = nodes.pop(0)
221         level = levels.pop(0)
222         if (str(node) not in v):
223             if (num_of_threats(node) == 0):
224                 return node, level, moves
225             else:
226                 v[str(node)] = 0
227             temp = generate_next_children(node)
228             for i in temp:
229                 if (str(i) not in v):
230                     nodes.append(i)
231                     levels.append(level + 1)
```

یک queue ی بسیار بزرگ به نام nodes داریم، ابتدا حالت اولیه را وارد صف خود میکنیم. سپس به ازای تک تک اعضای صف، تا جایی که به جواب برسیم،بررسی میکنیم که آیا جواب ما هست یا خیر. اگر نبود، تمامی فرزندان آن را به انتهای صف اضافه میکنیم. با این روش، الگوریتم BFS را پیاده سازی کرده ایم.

2. IDS:

ابتدا الگوریتم DFS با وجود یک محدودیت ارتفاع را توضیح میدهیم. یک DFS با محدودیت ارتفاع الگوریتمی است که ابتدا تا ارتفاع مجاز برای یک برگ پیش میرود. سپس برادر های آن را چک میکند. سپس یک مرحله بالا میاید، و همین کار را نسبت به فرزندانِ فرزند دوم پدربزرگ همان گروه اول انجام میدهد. و این کار را انقدر تکرار میکند که همه حالات را طی کند.

حال الگوریتم IDS میاید و برای ما به ترتیب DFS های مختلف با محدودیت از یک تا بینهایت را چک میکند تا به جواب برسد.

```
all_moves = 0
def IDS(root_grid):
    i = 1
    global all_moves
    all_moves = 0
    while (True):
        result = DFS(root_grid, i)
        if (result != None):
            return result, i, all_moves
        i += 1

def DFS(root_grid, limit):
    if limit == 0:
        if num_of_threats(root_grid) == 0:
            return root_grid
        else:
            return None
    for i in range(n_queens):
        for j in range(num_of_moves):
            if(move_if_possible(root_grid, i, j)):
                global all_moves
                all_moves += 1
                temp = DFS(root_grid,limit - 1)
                if temp is not None:
                    return temp
                move(root_grid, i, (j + 4) % 8)
    return None
```

به صورت بازگشتی، وارد تابع میشویم. اگر به انتهای محدودیت خود در تابع نرسیده بودیم، همین نود را چک کرده و به نود فرزند وارد میشویم. اگر به انتها رسیدیم، شروع میکنیم به چک کردن برادر های آن.

توضیح این تابع در گزارش کار دشوار است، اما میتوانیم مطمئن باشیم که به جواب درست میرسیم !!

3. A*

این تابع نشان میدهد که informed searsh ها بسیار بهتر از جستجو های نامعین عمل میکنند. در هر حالتی که هستیم، با توجه به این که چقد از وضعیت اصلی دور هستیم که تابع num_of_threats که در بالا توضیح آن آمده است این مقدار را به ما میگوید، تلاش میکنیم که راهی را انتخاب کنیم که بهینه تر است. اگر اشتباه کرده بودیم نیز مشکلی پیش نمی آید و تلاش میکنیم که با بازگشت مقداری از مسیر به راه بهتر دست یابیم.

```
194 def A_star(root_grid):
195     nodes = [root_grid]
196     levels = [0]
197     measures = [0 + num_of_threats(root_grid)]
198     moves = 0
199     while(True):
200         moves += 1
201         min_index = measures.index(min(measures))
202         node = nodes.pop(min_index)
203         level = levels.pop(min_index)
204         measure = measures.pop(min_index)
205         if (num_of_threats(node) == 0):
206             return node, level, moves
207         temp = generate_next_children(node)
208         for i in range(len((temp))):
209             nodes.append(temp[i])
210             levels.append(level + 1)
211             measures.append(level + 1 + num_of_threats(temp[i]))
212
```

این تابع، سه مجموعه را به طور موازی نگه میدارد. یکی برای node ها، یکی برای فاصله آن ها از نوک درخت، و آخری برای محاسبه تابع $f(x) = heuristic(x) + g(x)a$

سپس بر اساس این که تابع f کدام عنصر از بقیه کمتر است، آن نود را انتخاب میکنیم. اگر به جواب رسیده بودیم خارج میشویم و اگر نرسیده بودیم فرزندان این نود را به لیست خود اضافه میکنیم. حال بین نود های فعلی این کار را تکرار میکنیم.

مقایسه الگوریتم ها:

الگوریتم DFS نسبت به BFS حافظه بسیار کمتری نگه میدارد، اما در لایه های پایین سرعت آن کمتر از BFS است

الگوریتم A* از همه الگوریتم ها از نظر سرعت بسیار بهتر است چون حالت های مناسب را فقط بررسی میکند و بسیار محاسبات کمتری را انجام میدهد.

الگوریتم BFS از DFS در ارتفاع های کم در زمان بیشتری اجرا میشود، علاوه بر این که حافظه بیشتری نیز مصرف میکند.

محاسبه جواب ها:

مقایسه عمق محاسبه (تعداد جابجایی های لازم)

	Test a	Test b	Test c	In 1	In 2	In 3
IDS	3	4	-	-	-	-
BFS	3	4	-	-	-	-
A*	3	4	5	6	7	7

مقایسه تعداد حرکات انجام شده تا رسیده به جواب

	Test a	Test b	Test c	In 1	In 2	In 3
IDS	69099	4675728	-	-	-	-
BFS	5857	347100	-	-	-	-
A*	5	12	388	1456	5804	1169

مقایسه زمان رسیدن به جواب

	Test a	Test b	Test c	In 1	In 2	In 3
IDS	1.361	94.226	-	-	-	-
BFS	1.674	1221				
A*	0.005	0.013	0.571	2.685	23.138	2.305