معصومه پاسبانی 99243022

پیاده سازی BFS

هر نود جستجو را به صورت یک tuple تعیف میکنیم. مولفه اول آن مختصات خانه مورد بررسی، و مولفه دوم آن والد نود میباشد. پس نود اولیه والدی ندارد و به عنوان مختصات خانه فعلی درنظر میگیریم.

در BFS از صف برای لیست باز استفاده میکنیم، زیرا پیچیدگی زمانی بهتری دارد. برای نگه داری لیست بسته، از set استفاده میکنیم. درآخر نود هذف در متغیر solution نگه داری میشود.

تا زمانی که لیست باز ما خالی نشود،جستجو ادامه خواهد داشت. در بدنه حلقه while، از لیست باز نودی که سرصف قرار دارد را برمیداریم و اگر آن نود، دارای حالت هدف باشد به عنوان متغیر solution قرار میگیرد و جستجو تمام میشود. وگرنه با توجه به حالات موجود در نود فعلی، اکشن های ممکن حساب میشوند و و حالتی که میتوان با یک اکشن به ان رفت و در لیست بسته نیستند، به لیست باز اضافه میشوند و گره فعلی به عنوان والد گره افزوده شده خواهد بود.

اقدامات ممکن

در این تابع با دریافت مختصات یکی از خانه ها، مختصات خانه هایی که میتوانیم با یک حرکت به ان ها برویم(در صورتی که از صفحه بیرون نباشد و یا مسدود نباشد) به صورت یک لیست برگردانده میشوند. و اگر جستجو به جوابی رسیده باشد توسط تابع show_solution_bfs نشان داده میشود.

Show solution bfs

```
1 usage
def show_solution_bfs(self, start_solution_node):
    current_node = start_solution_node

    while current_node is not None:
        current_node = current_node

        node_x, node_y = current_node[0]

        current_tile_of_node = self.current_state[node_x][node_y]
        current_tile_of_node.set_color(colors.blue)

        current_node = current_node[1]
```

در این تابع، از نود موجود در آرگومان تابع شروع میکنیم و در هر مرحله خانه مربوزه را رنگ میکنیم و بعد به نود والد میرویم و تاجایی این کار را ادامه میدهیم که نود والدی وجود نداشته باشد.

نتیجه اجرای BFS به صورت زیر خواهد بود:

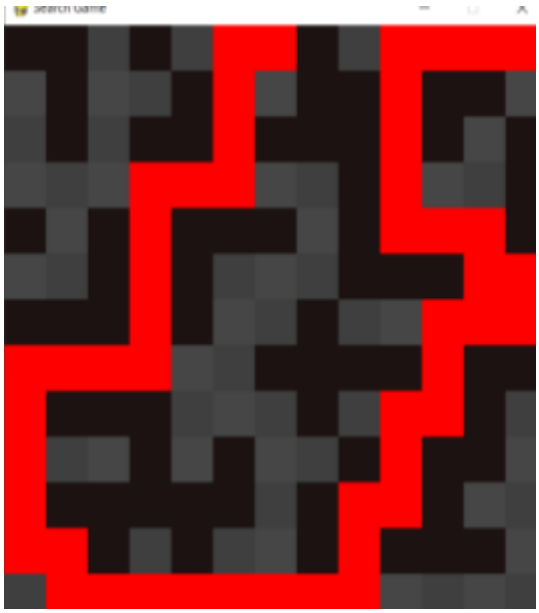
توجه به این که در این جستجو، ضریب انشعاب برابر ۴ است، پیچیدگی زمانی BFS ، $O(4^d)$ خواهد بود که d طول کوتاه ترین مسیر خواهد بود.

DFS

پیاده سازی DFS مشابه BFS میباشد با این تفاوت که به جای صف از stack استفاده میکنیم زیرا در هر مرحله میخواهیم عمیق ترین نود درخت را بررسی کنیم. (list در پایتون همان stack میباشد)

```
def dfs(self, environment):
    self.percept(environment)

    initial = (self.position, None)
    s_list = [initial]
    e_list = set()
    solution = None

    while len(s_list) > 0:
        curr_node = s_list.pop()
        e_list.add(curr_node[0])

        x, y = curr_node[0]
        if self.current_state[x][y].is_goal():
            solution = curr_node
            break

        for action in self.get_actions((x, y)):
            if action not in e_list:
                s_list.append((action, curr_node))

    if solution is not None:
        self.show_solution_dfs(solution)
```

تابع show solution dfs نیز همان تابع show solution bfs میباشد تنها تفاوت ان در رنگ آن ها میباشد.



طیق انتظار ما DFS کوتاه ترین مسیر را پیدا نکرد و پیچیدگی زمانی آن به صورت $O(4^m)$ میباشد که m طول بلندترین مسیر ممکن است. زیرا DFS همه راه های دارای بیشترین طول را بررسی میکند تا به یک جواب نهایی برسد.

A*

برای پیاده سازی این الگوریتم، علاوه بر نود والد و حالت فعلی به مقدار f و g نود نیز نیاز داریم. همچنین برای نگه داری لیست باز از heap استفاده میکنیم. زیرا این DS میتواند نود با کمترین F را برگرداند.

هنگام اضافه کردن نودهای جدید به لیست باز، مقدار f,g جدید را حساب میکنیم و در نود جدید قرار میدهیم. g جدید همان g قبلی به اضافه 1 است زیرا هزینه هر عمل برابر 1 میباشد.

Heuristic

این تابع با توجه به مختصات نود فعلی و نود هدف، تخمینی از هزینه رسیدن به نود را حساب میکند؛ همچنین این تابع مجموع اختلاف xها و yهای این دو نقطه را برمیگرداند.

```
1 usage
@staticmethod
def taxicab_distance(p1: tuple, p2: tuple):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

1 usage
def heuristic(self, node: tuple):
    return Agent.taxicab_distance(node[2], self.goal_pos)
```

پیاده سازی *A به صورت زیر خواهد بود:

```
def a_star(self, environment):
    self.percept(environment)

    initial = (0, 0, self.position, None)
    a_list = [initial]
    c_list = {}
    solution = None

    while len(a_list) > 0:
        curr_node = heapq.heappop(a_list)

        x, y = curr_node[2]
        if self.current_state[x][y].is_goal():
            solution = curr_node
            break

        for action in self.get_actions((x, y)):
            current_g = curr_node[1]
            new_g = current_g + 1
            h = self.heuristic(curr_node)

            if action not in c_list or new_g + h < c_list[action][0]:
                new_node = (new_g + h, new_g, action, curr_node)
                heapq.heappush(a_list, new_node)
                c_list[action] = new_node

    if solution is not None:
        self.show_solution(solution, a_star = True)
```

هنگام مقایسه دو tuple توسطheapq ابتدا عنصر اول دو tuple مقایسه میشوند.بنابراین برای اینکه نود با کمترین f انتخاب شود، مقدار f را به عنوان عنطر اول tuple قرار میدهیم. لیست موجود در این تابع همان لیست باز میباشد که عملیات روی آن توسط heapq انجام میشود.

تابع show solution مانند bfs و dfs میباشد با این تفاوت که مقدار a_star =True میباشد که روی مقدار نهایی متغیر current solution node اثر میگذارد.

برای اینکه بتوانیم نودهایی که قبلا بررسی کرده ایم و اکنون مقدار f کمتری دارند را دوباره بررسی کنیم، از dict استفاده میکنیم.

نتیجه نهایی به صورت زیر خواهد بود:

طبق انتظار A* کوتاه ترین مسیر را پیدا کرد.