توابع پیاده شده در زیر به همراه توضیحات آمده است

BFS

```
def bfs(self):
    way_out = False
    open_list = []
    visited = []
    open_list.append(Tree_node(None,self.position))
    while len(open_list)!=0:
        father = open_list[0]
        open_list.pop(0)
        visited.append(father.data)
        if father.data == self.board.goal_pos:
            way_out = self.found(Tree_node(father,new_point))
            break
        self.paint(father.data,colors.red)
        actions = self.get_actions(father.data)
        for i in actions:
            new_point  = (i[0]+father.data[0], i[1]+father.data[1])
            if new_point in visited:
                continue
            open_list.append(Tree_node(father,new_point))
            self.paint(new_point,colors.black)
    if way_out==False:
        print("no way out")
        return
```

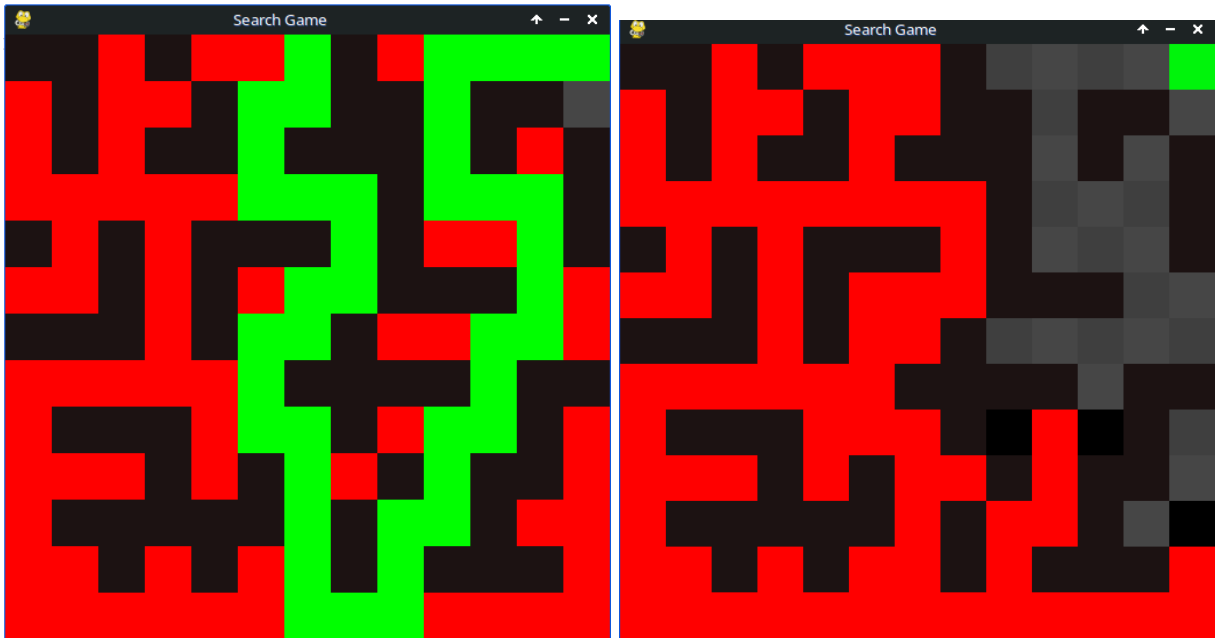
همونطور که در کد بالا میبینید این تابع به کمک دو لیست open_list پیاده شده است در open_list نود هایی پیاده شده است که هنوز گسترش پیدا نشده اند اگر برنامه را ران کنید خانه های قرمز خانه هایی هستند که گسترش پیدا کرده اند و خانه های مشکی خانه هایی هستند که در open_list هستند.

خانه های سبز هم مربوط به مسیری است که از آن میتوان به مقصد رسید

TreeNode نیز برای پیدا کردن این مسیر استفاده شده است و برای این استفاده شده است که درختی که پیموده میشود را شبیه سازی کند

همونطور که میبینید خانه هایی که قبلا پیمایش شده اند دیگر پیمایش نمیشوند این خصوصیت به خاطر حضور لیست visited به وجود آمده است

تصاویر اجرای برنامه در زیر آمده است:

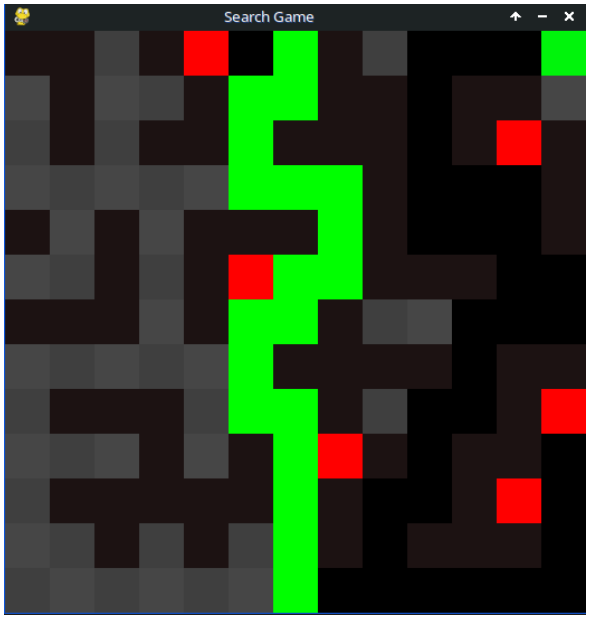


DFS

```
    def dfs(self):
        visited_list = []
        way_out = []
        way_out.append(self.position)
        self.paint(self.position,colors.black)
        if self.dfs_recurcive(way_out,visited_list,self.position) == True:
            for i in way_out:
                self.paint(i,colors.green)
        else:
            print("no way out")

    def dfs_recurcive(self,way,visited,point):
        if point == self.board.goal_pos:
            return True
        visited.append(point)
        self.paint(point,colors.red)
        available_moves = self.get_actions(point)
        for i in available_moves:
            new_pos = (point[0]+i[0],point[1]+i[1])
            if new_pos in visited:
                continue
            self.paint(point,colors.black)
```

```
            visited.append(new_pos)
            way.append(new_pos)
            if self.dfs_recurcive(way,visited,new_pos):
                return True
            way.pop()
        return False
```

دی اف اس به صورت تابع بازگشتی پیاده سازی شده است به همین دلیل دیگر نیازی به ایجاد درخت برای ذخیره مسیر ها نیست.
مشکلی که این روش پیاده سازی دارد ممکن است کوتاه ترین مسیر را پیشنهاد ندهد اما با کمی بهبود این مشکل حل میشود
تابع bfs خود بازگشتی نیست اما تابع dfs_recursive کاملا بازگشتی است
خانه هایی که به درون آن ها رفته و سپس از مسیر حذف میشوند قرمز و سیاه نشان داده شده اند و مسیر با سبز نشان داده شده است.
همونطور که میبینید کوتاه ترین مسیر ممکن پیشنهاد نشده است اما فضای خیلی کم تری اشغال شده است(خانه هایی که رنگشان عوض نشده است اصلا در حافظه ذخیره نشده اند و این برای سیستم ها با حافظه کم خیلی مفید است.
عکس های اجرا در زیر آمده است:

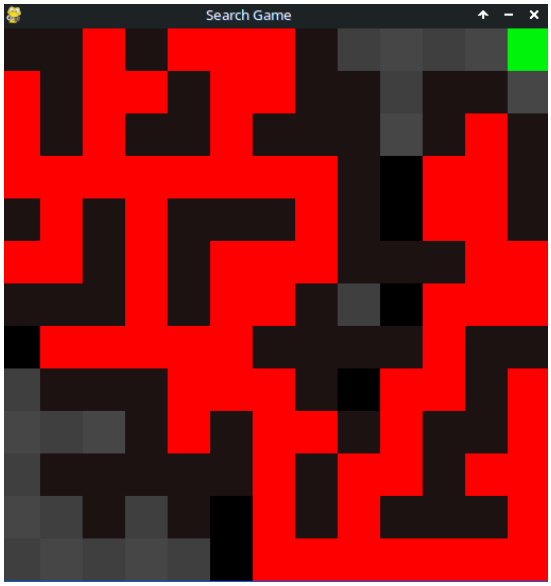


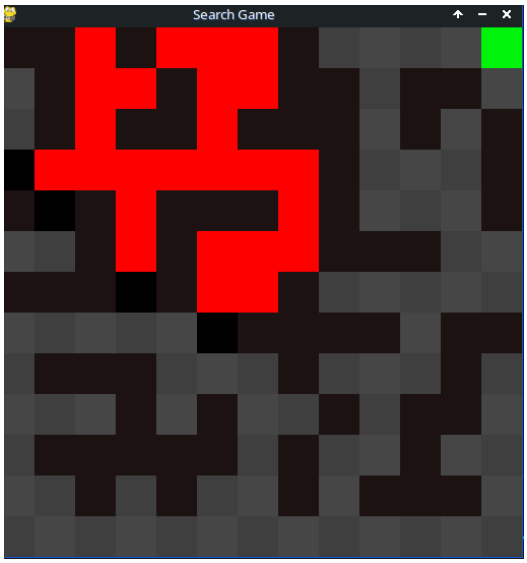
حال به سراغ a-star میرویم:

```
    def a_star(self):
        way_out = False
        open_list = []
        visited = []

open_list.append((Tree_node(None,self.position),self.heuristic(self.positi
on)))
        while len(open_list)!=0:
            father = open_list[0][0]
            open_list.pop(0)
            visited.append(father.data)
```

```
            if father.data == self.board.goal_pos:
                way_out = self.found(Tree_node(father,new_point))
                break
            self.paint(father.data,colors.red)
            actions = self.get_actions(father.data)
            for i in actions:
                new_point  = (i[0]+father.data[0], i[1]+father.data[1])
                if new_point in visited:
                    continue

open_list.append((Tree_node(father,new_point),self.heuristic(new_point)))
                self.paint(new_point,colors.black)
            open_list.sort(key= lambda num : num[1])
        if way_out==False:
            print("no way out")
            return
```

همونطور که میبینید این تابع به صورت غیر بازگشتی پیاده سازی شده است. اول تلاش کردم به صورت بازگشتی آن را پیاده سازی کنم اما موفق نشدم. کد حالت بازگشتی هنوز به صورت کامنت در فایل ایجنت هست. مشکل آن حالت این بود که چیزی برای ذخیره بخش هایی که قبلا دیده شده نداشم و این باعث ایجاد لوپ تولانی میشد. برای پیاده سازی به آن روش باید تابع هیوریستیک را جوری تغییر داد که بتواند خود را تغییر دهد و از لوپ جلو گیری کند.
تابع غیر بازگشتی آن دو لیست دارد یکی لیست نود هایی که آن ها برای گسترش چک میشوند. و لیست ویزیتد هم تمام نود هایی را نگه داری میکند که قبلا گسترش داده شده است
همونطور که میبینید باز هم حافظه زیادی میگیرد اما برتری که دارد هم سریع تر به مقصد میرسد هم خانه های کم تری را نسبت به bfs چک میکند.
در عکس های این مدل پیاده سازی بخش های قرمز خانه هایی هستند که باز شده و گسترش پیدا کرده اند بخش های سیاه خانه هایی هستند که در لیست گسترش هستند و منتظر هستند که در صورت ممکن نوبت آن ها شود برای گسترش. خط سبز نیز نشانه مسیر ما به سمت هدف است.
همونطور که در کد میبینید در پایان لوپ لیست باز مرتب میشود بر اساس مقادیر تابع هیوریستیک.
این تابع فاصله رو به صوورت چند خانه بالا و چند خانه چپ رفتن حساب میکند.
میتوانید با آن کامنت کردن کد هر تابع سرچ آن را تست کنید


Search Game


Search Game


Search Game