تغییرات در کلاس های دیگر

- Environment
  - تابع set_agent_pos()

در این تابع نوع متغیر new_pos برابر با tuple در نظر گرفته شد بنابراین جای x و y از 0 و 1 استفاده کردیم:

```
def set_agent_pos(self, new_pos):

    self.agent_pos = new_pos[0], new_pos[1]
    (self.boardArray[new_pos[0]][new_pos[1]]).set_player_here()
```

- Tile
  - ویژگی is_visited برای توابع dfs و bfs و تشخیص اینکه آیا نودی ویزیت شده و الان در closed list قرار دارد یا خیر.
  - ویژگی parent برای چاپ شاخه ای که به مقصد منتهی میشود.
  - ویژگی dist برای پیدا کردن مینیمم طول مسیر از مبدا به کاشی مد نظر در الگوریتم a_star.
  - ویژگی a_star_func که جمع تابع هیوریستیک هر کاشی با dist آن است که در طول الگوریتم آن را نیز به دست می آوریم.

```
# here are my additions
self.is_visited = False
self.parent = None
self.dist = math.inf
self.a_star_func = math.inf
```

- تابع pathColor()

  در این تابع از مقصد شروع کرده و تا زمانی که به مبدا نرسیدیم، والد هر کاشی را پیدا و آن را رنگ میکنیم تا شاخه ای که به مقصد منتهی میشود، رنگ آمیزی گردد.

نحوه اجرای بازی

تابع main به این صورت است در لوپ بی نهایت میخواهیم یکبار، یک الگوریتم جستجو را انجام دهیم، بنابراین کانتری میگذاریم که اگر بیشتر از 0 شد دیگر بلوک اجرا نشود و در بلوک از کاربر میخواهیم که تابع مورد نظر خود را حساب کند:

```
if counter < 1:
    ask = input("Choose your desired function to solve the maze:
    if ask == "1":
        agent.bfs(gameBoard)
    elif ask == "2":
        agent.dfs(gameBoard)
    else:
        agent.a_star(gameBoard)

counter = counter + 1
```

BFS

در این تابع ابتدا مبدا را پیدا میکنیم و ان در لیست بسته قرار میدهیم. حال صفی میسازیم که نود هارا در خود نگه میدارد و اولین نودی که نگه میدارد، خود مبدا است.

سپس تا زمانی که صف خالی نشده است، اولین نودی که وارد صف شده بود را پاپ میکنیم، آزمون هدف را بر روی آن انجام میدهیم و در صورت هدف نبودن،

بازیکن را بر روی آن نود قرار میدهیم و سپس فرزندان آن را گسترش میدهیم و به صف اضافه میکنیم(در تابع check_neighbor).

متغیر expand_limit برای این است که در محیط گرافیکی گسترش به طور کامل انجام نشود:

```
def bfs(self, board):
    self.percept(board)
    source = self.find_source(board)
    queue = []
    source.is_visited = True
    queue.append(source)
    expand_limit = 0
    while queue:

        s = queue.pop(0)

        down_tile, left_tile, right_tile, up_tile = self.get_actions(board, s)

        if s.isGoal:
            s.pathColor()
            return
```

```
        if expand_limit < 34:
            self.move(board, (s.get_coordinates()[0], s.get_coordinates()[1]))
        self.check_neighbor(board, expand_limit, queue, right_tile, s)
        self.check_neighbor(board, expand_limit, queue, left_tile, s)
        self.check_neighbor(board, expand_limit, queue, down_tile, s)
        self.check_neighbor(board, expand_limit, queue, up_tile, s)
        expand_limit = expand_limit + 1

def check_neighbor(self, board, expand_limit, queue, right_tile, s):
    if right_tile.get_coordinates()[0] < 13 and not right_tile.is_visited \
            and not right_tile.is_blocked():
        queue.append(right_tile)
        right_tile.is_visited = True
        right_tile.parent = s
```

DFS

ابتدا در تابع اولیه dfs مبدا را پیدا میکنیم و سپس تابع بازگشتی را صدا میزنیم.
حال در تابع بازگشتی هر بار نود جدید انتخاب شده را در لیست بسته قرار میدهیم و سپس در جهت آن حرکت میکنیم. حال اگر این نود برابر با هدف باشد که هیچ و اگر نبود تابع بازگشتی را بر روی فرزندان این کاشی(نود) صدا میزنیم.

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

    source = self.find_source(board)

    self.dfs_recursive(0, source, board)

def dfs_recursive(self, expand_limit, new_tile, board):
    new_tile.is_visited = True
    expand_limit += 1
    if expand_limit < 25:
        self.move(board,
                  (new_tile.get_coordinates()[0], new_tile.get_coordinates()[1]))
    down_tile, left_tile, right_tile, up_tile = self.get_actions(board, new_tile)
```

```
    if new_tile.isGoal:
        new_tile.pathColor()
        self.find_source(board).color = colors.startColor
        self.find_dest(board).color = colors.endColor
        return

    self.call_dfs(board, new_tile, right_tile, expand_limit)
    self.call_dfs(board, new_tile, left_tile, expand_limit)
    self.call_dfs(board, new_tile, down_tile, expand_limit)
    self.call_dfs(board, new_tile, up_tile, expand_limit)

def call_dfs(self, board, new_tile, right_tile, expand_limit):
    if right_tile.get_coordinates()[0] < 13 and not right_tile.is_visited \
            and not right_tile.is_blocked():
        right_tile.parent = new_tile
        self.dfs_recursive(expand_limit, right_tile, board)
```

A*

در الگوریتم ای استار دو لیست باز و بسته داریم و ابتدا مبدا را در لیست بسته قرار میدهیم و dist آن را صفر میکنیم(مقدار اولیه dist بی نهایت است). سپس فرزندان مبدا را گسترش میدهیم و به لیست باز اضافه میکنیم تا بررسی شوند. حال شبیه الگوریتم دایجسترا یک فور به تعداد کل کاشی ها میزنیم و هر بار تابع f برای نودهای لیست باز را آپدیت میکنیم. تابع هیوریستیک استفاده شده فاصله منهتنی بین دو نقطه است. پس از اپدیت لیست، مینیمم نود را پیدا میکنیم و از لیست باز برداشته و به لیست بسته اضافه میکنیم. آزمون هدف بر روی این نود اجرا و اگر هدف بود، مسیر رسیدن به هدف را رنگ میکنیم و در غیر این صورت فرزندان آن را گسترش میدهیم.

```
def a_star(self, board):
    self.percept(board)
    open_list = []
    closed_list = []
    source = self.find_source(board)

    dest = self.find_dest(board)
    source.dist = 0
    closed_list.append(source)

    down_tile, left_tile, right_tile, up_tile = self.get_actions(board, source)

    self.check_neighbor_aStar(open_list, right_tile, source)
    self.check_neighbor_aStar(open_list, left_tile, source)
    self.check_neighbor_aStar(open_list, down_tile, source)
    self.check_neighbor_aStar(open_list, up_tile, source)
```

```
expand_limit = 0
tiles_length = rows * cols
for _ in range(tiles_length):
    for j in open_list:
        j.a_star_func = j.dist + abs(j.get_coordinates()[0] - dest.get_coordinates()[0])
                        + abs(j.get_coordinates()[1] - dest.get_coordinates()[1])
    min = math.inf
    min_idx = None
    for j in open_list:
        if j not in closed_list and not j.is_blocked():
            if j.a_star_func < min:
                min = j.a_star_func
                min_idx = j
```

```
if min_idx is not None:
    expand_limit += 1
    closed_list.append(min_idx)
    open_list.remove(min_idx)
    if expand_limit < 34:
        self.move(board, (min_idx.get_coordinates()[0], min_idx.get_coordinates()[1]))
    if min_idx.isGoal:

        min_idx.pathColor()
        self.find_source(board).color = colors.startColor
        self.find_dest(board).color = colors.endColor
        return

    down_tile, left_tile, right_tile, up_tile = self.get_actions(board, min_idx)
```

```
        self.expand(closed_list, min_idx, open_list, right_tile)
        self.expand(closed_list, min_idx, open_list, left_tile)
        self.expand(closed_list, min_idx, open_list, down_tile)
        self.expand(closed_list, min_idx, open_list, up_tile)

find_dest(self, board):
dest = None
for i in range(rows):
    for j in range(cols):
        if board.boardArray[i][j].isGoal:
            dest = board.boardArray[i][j]
return dest
```

```python
@staticmethod
def expand(closed_list, min_idx, open_list, right_tile):
    if right_tile.get_coordinates()[0] < 13 \
            and right_tile not in closed_list \
            and min_idx.dist + 1 < right_tile.dist \
            and not right_tile.is_blocked():
        right_tile.dist = min_idx.dist + 1
        right_tile.parent = min_idx
        open_list.append(right_tile)

@staticmethod
def check_neighbor_aStar(open_list, right_tile, source):
    if right_tile.get_coordinates()[0] < 13:
        right_tile.dist = 1
        open_list.append(right_tile)
        right_tile.parent = source
```

get_actions

به این گونه تغییرش دادیم که فرزندان را به ما بدهد به این شرط که در جدول جا شوند و وجود داشته باشند، در غیر این صورت کاشی ای رندوم را برمیگردانیم و در تابعها چک میکنیم که آیا این کاشی موجود است یا خیر.

```python
@staticmethod
def get_actions(board, s):
    position_x = s.get_coordinates()[0]
    position_y = s.get_coordinates()[1]
    right_tile = Tile(13, 13)
    left_tile = Tile(13, 13)
    up_tile = Tile(13, 13)
    down_tile = Tile(13, 13)
    if position_x + 1 < cols:
        right_tile = board.boardArray[position_x + 1][position_y]
    if position_x > 0:
        left_tile = board.boardArray[position_x - 1][position_y]
    if position_y > 0:
        up_tile = board.boardArray[position_x][position_y - 1]
    if position_y + 1 < rows:
        down_tile = board.boardArray[position_x][position_y + 1]
    return down_tile, left_tile, right_tile, up_tile
```

Move

تابع move کمی تغییر کرده است و ابتدا جایی که بازیکن قبلا در آن بود قرمز و سپس جای جدیدی که قرار است باشد با new_pos نمایش داده شده و با رنگ بازیکن رنگ میشود(با استفاده از تابع set_player_here در کلاس Tile).

```
def move(self, board, new_pos):
    current_pos = self.get_position()
    x, y = current_pos[0], current_pos[1]
    board.colorize(x, y, colors.red)

    self.set_position(new_pos, board)
```

مقایسه پیچیدگی زمانی الگوریتم ها

پیچیدگی زمانی bfs برابر با b^d میباشد زیرا در بدترین حالت باید کل درخت را پیمایش کنیم تا به هدف برسیم(هدف در اخرین نود از اخرین سطح باشد).

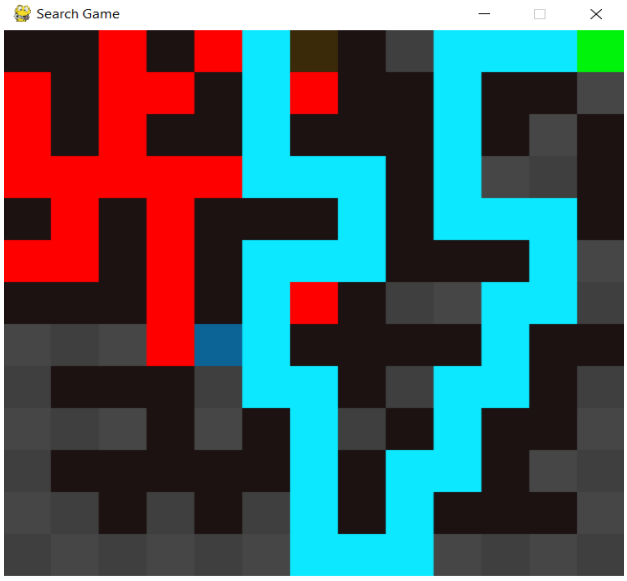
پیچیدگی زمانی dfs نیز با bfs برابر است و باید در بدترین زمان ممکن کل درخت را پیمایش کنیم. (البته که bfs در مورد گراف های با وزن برابر 1 بهینه است و کوتاه ترین مسیر را پیدا میکند در حالی که در dfs اینطور نیست).

الگوریتم ای استار پیچیدگی زمانی مشخصی ندارد و در واقع اگر تابع هیوریستیکیکه انتخاب شده است به تابع شهودی واقعی نزدیک تر باشد، پیچیدگی زمانی بهتری داریم و نود های کمتری بررسی میشوند.

اجرای بازی

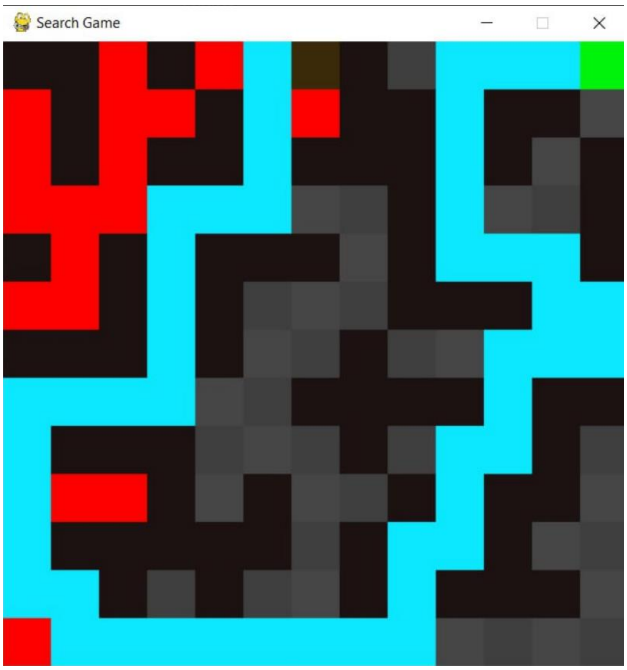
BFS

همانطور که مشخص است bfs چون در هر سطح کوتاه ترین مسیر هارا میکند پس مسیری بهینه نیز پیدا کرده است.



DFS

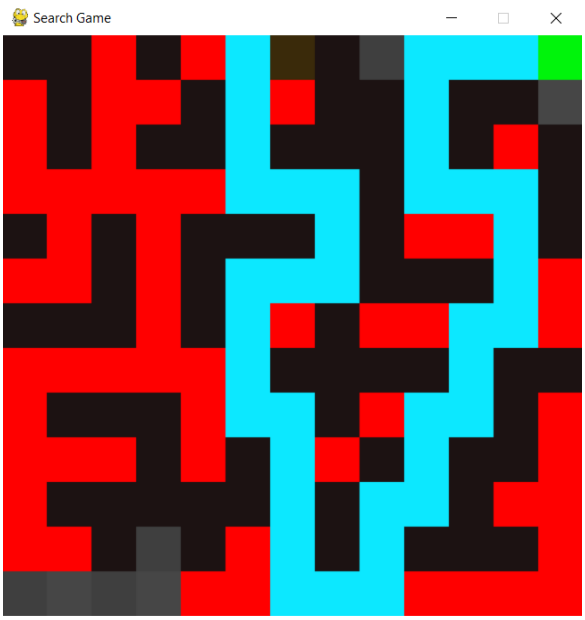
در dfs اما ابتدا یک عمق بررسی میشود پس dfs اصلا به بهینه بودن جستجو کاری ندارد و اگر در شکل نیز ببینیم در گسترش به سمت چپ مایل شده است و از هدف دور شده است.

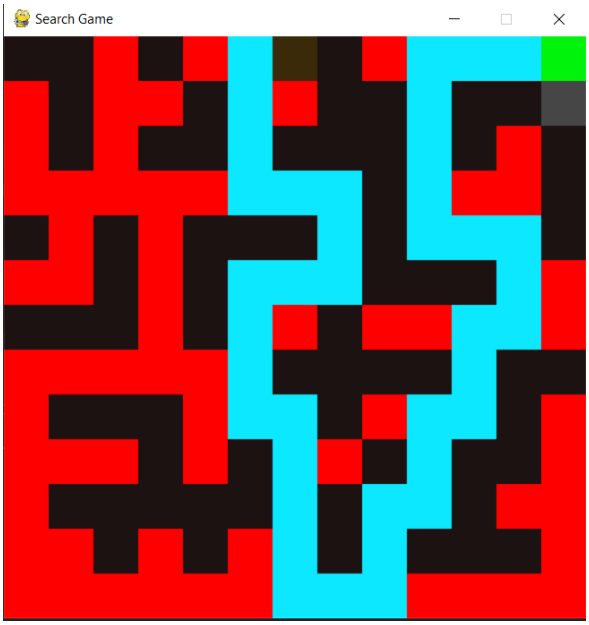


A*

تا حدودی شبیه به bfs است ولی پایین تر مشخص میشود که چرا ای استار بهتر است.

A* vs BFS





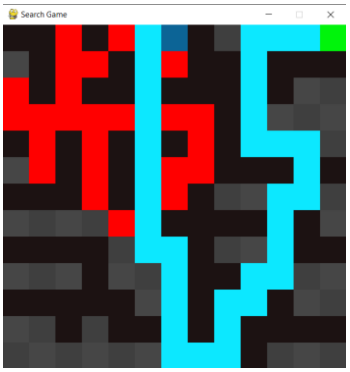
سمت چپی ای استار است و در گسترش کامل میبینیم که بعصی از نودهارا به علت وجود تابع هیوریستیک لازم به بررسی ندانسته است اما تابع bfs جستجویی نا آگاهانه است و این نود هارا نیز بررسی کرده است(عکس سمت راست).

جنراتور

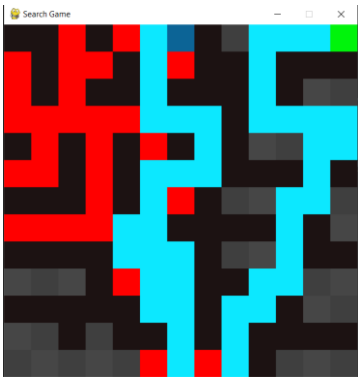
سه ماز جدید در فولدر Mazes موجود هستند.

New_maze3 را ببینیم:

BFS



DFS :))))



A*