پیچیدگی زمانی الگوریتم BFS برابر با O(b^d) است که d عمق کم عمق ترین جواب

پیچیدگی زمانی الگوریتم DFS برابر با O(b^m) است که m عمق طولانی ترین مسیر درخت است

پس نتیجه میشود که DFS پیچیدگی زمانی بیشتری دارد .

```
class Tile:

    def __init__(self, x, y):
        self.parent = None
        self.x = x
        self.y = y
        self.h = math.sqrt((13 - x)*(13 - x) + y * y)
```

در کلاس Tile یکسری اتریبیوت تعریف میکنیم

پدر نود فعلی و فاصله مختصاتی تا نود هدف

```python
def move(self, direction, color):
    # make your next move based on your perception
    # check if the move destination is not blocked
    # if not blocked:
    # use red color to show visited tiles.
    # something like :
    # current_pos = self.get_position()
    x, y = direction[0], direction[1]
    self.board.colorize(x, y, color)

    # then move to destination - set new position
    # something like :
    # self.set_position(direction)
    pass
```

در تابع move نود داده شده را رنگ میکنیم

```python
def get_actions(self, cord, o_list, c_list, flag):
    actions = []
    x = cord[0]
    y = cord[1]
    o_list_temp = list()
    if flag:
        while (not o_list.empty()):
            o_list_temp.append(o_list.get())
    else:
        o_list_temp = o_list

    if x - 1 >= 0:
        if not self.current_state[x - 1][y].is_blocked():
            cord = (x - 1, y)
            if (cord not in o_list_temp):
                if (cord not in c_list):
                    actions.append(cord)

    if y - 1 >= 0:
        if not self.current_state[x][y - 1].is_blocked():
            cord = (x, y - 1)
            if (cord not in o_list_temp):
                if (cord not in c_list):
                    actions.append(cord)

    if x + 1 < 13:
        if not self.current_state[x + 1][y].is_blocked():
            cord = (x + 1, y)
            if (cord not in o_list_temp):
                if (cord not in c_list):
                    actions.append(cord)

    if y + 1 < 13:
        if not self.current_state[x][y + 1].is_blocked():
            cord = (x,  y + 1)
            if (cord not in o_list_temp):
                if (cord not in c_list):
                    actions.append(cord)
    # returns a list of valid actions

    if flag:
        for i in range(len(o_list_temp)):
            o_list.put(o_list_temp[i])

    return actions
```

در تابع get action چک میکنیم که به کدام نود ها میتوانیم برویم

ابتدا در خط اول شرط ها بررسی میشود که از صفحه بیرون نزنیم

سپس چک میشود که نود مجاور بلاک نباشد

بعد چک میشود که نود در لیست بسته ما قرار نداشته باشد

همچنین اگر در لیست باز هم موجود باشد نیازی نیست آن را مجدد اضافه کنیم و بررسی کنیم چون در صف بررسی ما قرار دارد

یک فلگ داریم که لیست ارسالی لیست است یا صف برای تفاوت bfs , dfs

اگر bfs باشد صف است و اگر dfs باشد لیست است

الگوریتم bfs

```
def bfs(self, environment):
    # now go on !
    self.percept(environment)
    o_list = Queue()
    c_list = list()
    agent_tile = self.position
    o_list.put(agent_tile)
    while (not o_list.empty()):
        tile = o_list.get()
        if self.current_state[tile[0]][tile[1]].isGoal:
            break
        self.move(tile, colors.red)
        c_list.append(tile)
        actions = self.get_actions(tile, o_list, c_list, True)
        for element in actions:
            o_list.put(element)
        for i in range(len(actions)):
            self.current_state[actions[i][0]][actions[i][1]].parent = tile

    itr = (12, 0)
    while self.current_state[itr[0]][itr[1]].parent != None:
        self.move(itr, colors.green)
        itr = self.current_state[itr[0]][itr[1]].parent
    self.move(itr, colors.green)
    # use green color and colorize method to show the path.
    pass
```

یک لیست باز و بسته تعریف میکنیم که لیست باز ما مثل صف fifo عمل میکند

و هر نودی زودتر آمده زودتر بررسی میشود

سپس نقطه شروع را ست میکنیم و تا زمانی که لیست باز خالی نشده به جستجو در حلقه ادامه میدهیم

در صورتی که به نود هدف برسیم از حلقه خارج میشویم

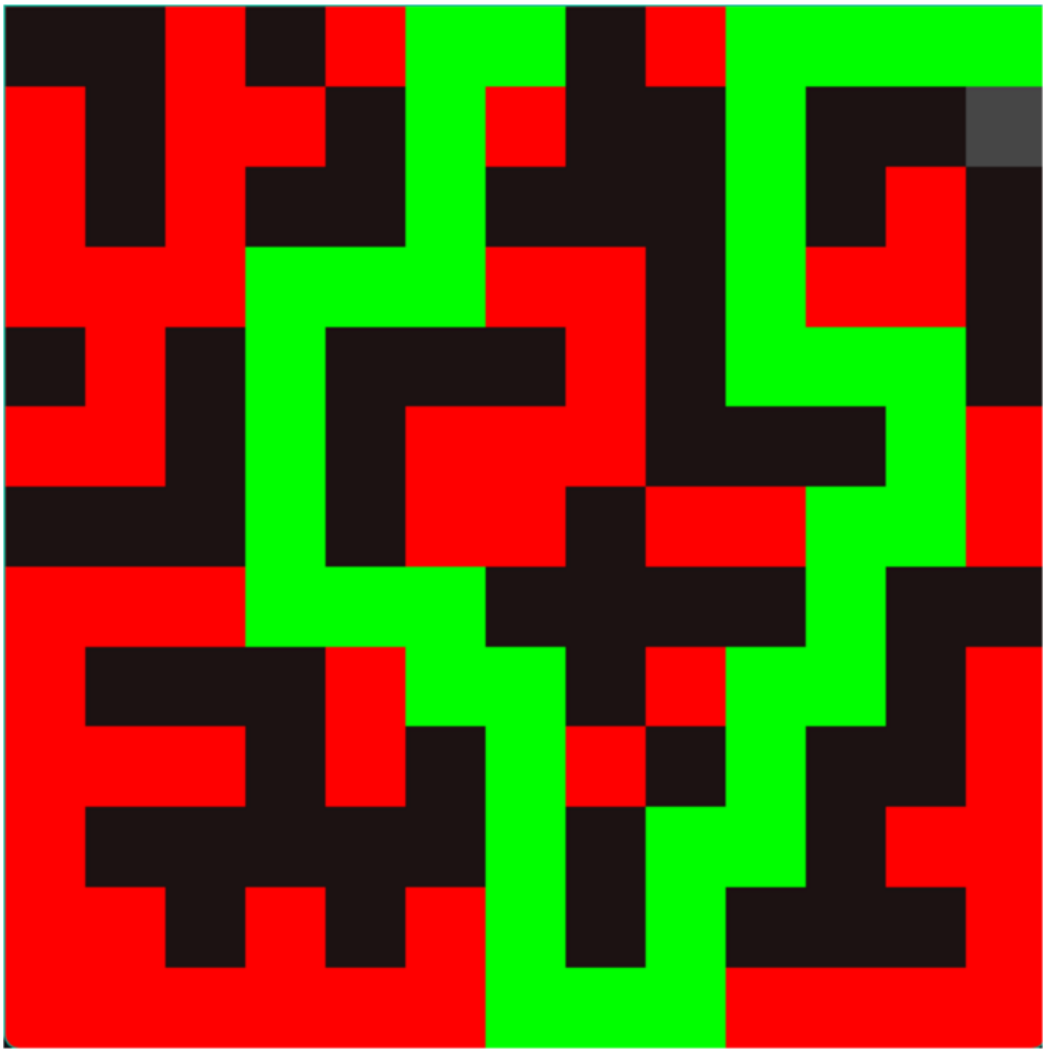
در غیر اینصورت اکشن های مجاز نود را پیدا کرده و آن ها را گسترش میدهیم

به هر نود که برسیم قرمز میکنیم

پس از رسیدن به هدف از نود هدف به صورت بازگشتی به عقب برمیگردیم و مسیر طی شده را سبز میکنیم

(از اتریبیوت اضافه شده parent استفاده میکنیم)

خروجی



الگوریتم dfs

```
def dfs(self, environment):
    self.percept(environment)
    o_list = list()
    c_list = list()
    agent_tile = self.position
    o_list.append(agent_tile)
    while (len(o_list) != 0):
        tile = o_list[len(o_list) - 1]
        if self.current_state[tile[0]][tile[1]].isGoal:
            break
        self.move(tile, colors.red)
        c_list.append(tile)
        o_list.pop()
        actions = self.get_actions(tile, o_list, c_list, False)
        o_list.extend(actions)
        for i in range(len(actions)):
            self.current_state[actions[i][0]][actions[i][1]].parent = tile

    itr = (12, 0)
    while self.current_state[itr[0]][itr[1]].parent != None:
        self.move(itr, colors.green)
        itr = self.current_state[itr[0]][itr[1]].parent
    self.move(itr, colors.green)
```

در اینجا لیست باز و بسته لیست هستند

سپس نقطه شروع را ست میکنیم و تا زمانی که لیست باز خالی نشده به جستجو در حلقه ادامه میدهیم

در صورتی که به نود هدف برسیم از حلقه خارج میشویم

در غیر اینصورت اکشن های مجاز نود را پیدا کرده و آن ها را گسترش میدهیم

جهت انتخاب نود از لیست باز مانند استک عمل میکنیم و از سر لیست نود را برمیداریم

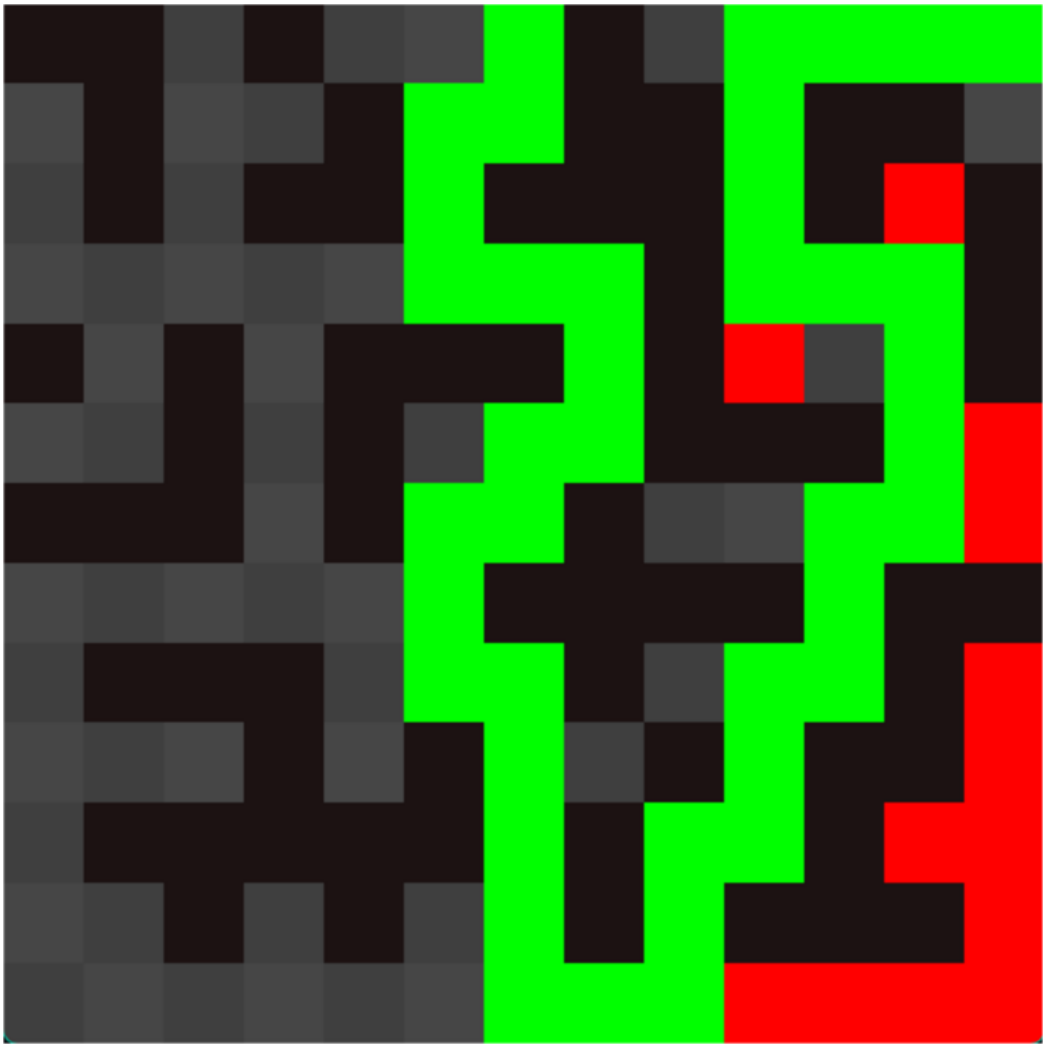
به هر نود که برسیم قرمز میکنیم

پس از انتقال نود به لیست بسته آن را ار لیست باز حذف میکنیم

پس از رسیدن به هدف از نود هدف به صورت بازگشتی به عقب برمیگردیم و مسیر طی شده را سبز میکنیم

(از اتریبیوت اضافه شده parent استفاده میکنیم)

خروجی :



الگوریتم A*

```
def a_star(self, environment):
    self.percept(environment)
    o_list = list()
    c_list = list()
    agent_tile = self.position
    o_list.append(agent_tile)
    while (len(o_list) != 0):
        n = len(o_list)
        for i in range(n - 1):
            for j in range(0, n - i - 1):
                if self.current_state[o_list[j][0]][o_list[j][1]].h > self.current_state[o_list[j + 1][0]][o_list[j + 1][1]].h:
                    o_list[j], o_list[j + 1] = o_list[j + 1], o_list[j]

        tile = o_list[0]
        if self.current_state[tile[0]][tile[1]].isGoal:
            break
        self.move(tile, colors.red)
        c_list.append(tile)
        o_list.remove(tile)
        actions = self.get_actions(tile, o_list, c_list, False)
        o_list.extend(actions)
        for i in range(len(actions)):
            self.current_state[actions[i][0]][actions[i][1]].parent = tile

    itr = (12, 0)
    while self.current_state[itr[0]][itr[1]].parent != None:
        self.move(itr, colors.green)
        itr = self.current_state[itr[0]][itr[1]].parent
    self.move(itr, colors.green)
```

در اینجا لیست باز و بسته لیست هستند و لیست باز در طول کد سورت میشود

سپس نقطه شروع را ست میکنیم و تا زمانی که لیست باز خالی نشده به جستجو در حلقه ادامه میدهیم

در صورتی که به نود هدف برسیم از حلقه خارج میشویم

در غیر اینصورت اکشن های مجاز نود را پیدا کرده و آن ها را گسترش میدهیم

جهت انتخاب نود از لیست باز لیست را بر اساس فاصله مختصاتی آن تا نود هدف سورت میکنیم (بابل سورت)

و نود با فاصله کمتر را ابتدا انتخاب میکنیم

به هر نود که برسیم قرمز میکنیم

پس از انتقال نود به لیست بسته آن را ار لیست باز حذف میکنیم

پس از رسیدن به هدف از نود هدف به صورت بازگشتی به عقب برمیگردیم و مسیر طی شده را سبز میکنیم

(از اتریبیوت اضافه شده parent استفاده میکنیم)

خروجی :