گزارش تمرین عملی ۱ هوش مصنوعی (الگوریتم A^*)

آریا جلالی
۹۸۱۰۵۶۶۵

۱ کتابخانه‌های استفاده شده:

```
import xml.etree.ElementTree as ET
from queue import PriorityQueue
import pygame
import numpy as np
```

از ET برای خواندن محیط مسئله از فایل xml استفاده شده است و از کتابخانه‌ی $PriorityQueue$ برای نگه داشتن $frontier$ الگوریتم‌های پیاده‌سازی شده استفاده شده است. زیرا نیاز داریم در زمان مناسب خانه با کمترین مقدار f پیدا کنیم. از کتابخانه‌ی $pygame$ برای نمایش گرافیکی فعالیت لحظه‌ای الگوریتم استفاده شده است و در نهایت از کتابخانه‌ی $numpy$ برای استفاده از تابع‌های سریع برای کارکردن با آرایه‌ها و ماتریس‌ها استفاده شده است.

۲ خواندن نقشه از فایل و ورودی گرفتن:

```
# Define some colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
BLUE = (0, 0, 255)
VIOLET = (255, 0, 190)
PEACHPUFF = (255, 218, 185)
GOLD = (255, 215, 0)

# Directions
dRow = [1, 0, -1, 0]
dCol = [0, 1, 0, -1]
directions = ['U', 'L', 'D', 'R']
```

در تکه کد بالا ابتدا برای بخش گرافیکی مسئله چند رنگ را با استفاده از مقادیر RGB آن‌ها تعریف کرده و در ادامه چند جهت برای پیدا کردن همسایه‌های هر خانه قرار داده‌ایم. دقت کنید لیست $directions$ برای مشخص کردن مسیر پس از پیدا کردن هدف قرار داده شده است.

```
name = input("Please enter the name of the xml file you want to
    open.\nIf you don't have an xml file you can ignore "
                "this.\n")

try:
    xml = open(name + ".xml", "r")
    path = xml.read()
    doc = ET.fromstring(path)
    row = len(doc)
    col = len(doc[0])
```

```
    grid = np.zeros((row, col))
    grid_dir = np.full((row, col), 'N')
except FileNotFoundError:
    pass

pygame.init()
```

در این بخش در صورت وجود داشتن فایل xml از کاربر نام آن را میپرسیم و در اگر نام معتبر بود فایل را باز میکنیم و تعداد سطر و ستون‌های نقشه را بدست می‌اوریم. در غیر اینصورت از این مرحله گذر کرده. در نهایت با استفاده از تابع $pygame.init()$ بخش گرافیکی را $initiate$ میکنیم.

```
def read_from_xml():
    for i in range(row):
        for j in range(col):
            if doc[i][j].text == 'robot':
                x_start, y_start = i, j
                grid[i][j] = 1
            elif doc[i][j].text == 'Battery':
                grid[i][j] = 2
                x_goal, y_goal = i, j
            elif doc[i][j].text == 'obstacle':
                grid[i][j] = 3
            else:
                grid[i][j] = 0
    return grid, (x_start, y_start), (x_goal, y_goal)


map_type = int(input("To customize your own map, type 1.\nTo
    read the map from the xml file, type 2.\n"))
if map_type == 1:
    (x_start, y_start) = map(int,
                             input("Please enter the coordinates
    for the robot, separated by space.\n").split(' '))
    (x_goal, y_goal) = map(int, input("Please enter the
    coordinates for the battery, separated by space.\n").split('
    '))
    (row, col) = map(int, input("Please enter the width and
    height of your map, separated by space.\n").split(' '))
    grid = np.zeros((row, col))
    grid_dir = np.full((row, col), 'N')
    grid[x_start][y_start] = 1
    grid[x_goal][y_goal] = 2
else:
    grid, (x_start, y_start), (x_goal, y_goal) = read_from_xml()
```

در تابع $read\ from\ xml$ نقشه را از فایل xml خوانده و آن را در ماتریس $grid$ قرار میدهیم که مقادیر درون آن بنا بر نوشته‌ی هر $cell$ در فایل xml مشخص میشود که در ادامه معنی هر عدد را خواهیم فهمید.

دقت کنید کاربر امکان ساخت نقشه‌ی $custom$ خود را نیز دارد و تابع $read\ from\ xml$ در صورتی صدا زده میشود که کاربر بخواهد نقشه را از فایل بخواند و در غیر این صورت اطلاعات نقشه از خود کاربر گرفته میشود.

```
def save_as_XML(name, grid):
    map = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n"
    map += "<rows>\n"
    for i in range(row):
        map += "<row>\n"
        for j in range(col):
            if grid[i][j] == 1:
                map += "<cell>robot</cell>\n"
            elif grid[i][j] == 2:
                map += "<cell>Battery</cell>\n"
            elif grid[i][j] == 3:
                map += "<cell>obstacle</cell>\n"
            else:
                map += "<cell>empty</cell>\n"
        map += "</row>\n"
    map += "</rows>"
    text_file = open(name, "w")
    text_file.write(map)
    text_file.close()
```

از تابع $save\ as\ xml$ برای ذخیره کردن نقشه‌ی نهایی استفاده خواهیم کرد. زیرا کاربر میتواند تغییرات دلخواه در نقشه اعمال کند و بخواهد آن‌ها را در فایل xml ذخیره کند.

۳ مقداردهی‌های اولیه

```
# This sets the margin between each cell
MARGIN = 1

# This sets the WIDTH and HEIGHT of each grid location
WIDTH, HEIGHT = 40, 40

if WIDTH * col + (col + 1) * MARGIN > 720 or HEIGHT * row + (row
    + 1) * MARGIN > 720:
    HEIGHT = 720 // (row)
    WIDTH = 720 // (col)

frontier = PriorityQueue()
explored_set = []
cost_set = np.full((row, col), -1)
cost_set[x_start][y_start] = abs(x_start - x_goal) + abs(y_start
    - y_goal)
```

```
15 frontier.put((abs(x_start - x_goal) + abs(y_start - y_goal),
       x_start, y_start, 0))
```

در این بخش مقادیر هر بلوک را در بخش گرافیکی مشخص میکنیم که مقادیر $default$ آن‌ها 40 برای طول و عرض و 1 برای فاصله‌ی بین هر دو بلوک میباشد. ولی در ادامه در صورت بزرگ شدن اندازه‌ی نقشه به اندازه‌ی زیاد این مقادیر را $scale$ میکنیم.

در ادامه با $instance$ گرفتن از کلاس $PriorityQueue$ $frontier$ را تبدیل به یک $min\ queue$ میکنیم. و در ادامه $explored\ set$ را که نشان‌دهنده‌ی خانه‌هایی است که قبلا آن‌ها را بازدید کرده‌ایم برابر با یک لیست خالی قرار میدهیم و در ادامه از ماتریس $cost\ set$ برای مشخص کردن هزینه‌ی تمام خانه‌ها استفاده میکنیم و اولین مقدار آن را برای خانه‌ی اول برابر با فاصله‌ی $manhattan$ آن از خانه‌ی هدف قرار میدهیم.
در نهایت با قرار دادن خانه‌ی اول در $frontier$ مقداردهی اولیه تمام میشود.

۴ پیدا کردن همسایه‌های هر خانه و چاپ مسیر

```
1  def isValid(x_cord, y_cord, cost):
2      if x_cord >= row or x_cord < 0 or y_cord >= col or y_cord <
       0 or ((x_cord, y_cord) in explored_set) \
3              or grid[x_cord][y_cord] == 3 or (cost_set[x_cord][
       y_cord] < cost and cost_set[x_cord][y_cord] != -1):
4          return False
5      return True
6
7
8  def print_path():
9      cur_x = x_goal
10     cur_y = y_goal
11     length = 0
12     while cur_x != x_start or cur_y != y_start:
13         length += 1
14         grid[cur_x][cur_y] = 6
15         print(cur_x, cur_y)
16         cur_dir = grid_dir[cur_x][cur_y]
17         if cur_dir == 'U':
18             cur_x -= 1
19         elif cur_dir == 'D':
20             cur_x += 1
21         elif cur_dir == 'R':
22             cur_y += 1
23         else:
24             cur_y -= 1
25         pygame.event.get()
26         Draw()
27     grid[x_goal][y_goal] = 2
28     grid[x_start][y_start] = 1
```

```
    print(x_start, y_start)
    print('length = ', length)
```

در این بخش با استفاده از تابع $isValid$ قابل دسترس بودن خانه‌های همسایه را بررسی میکنیم تا در صورت قابل دسترس بودن آن‌ها را در $frontier$ قرار دهیم. شروط این بخش خارج از نقشه بودن نقاط یا $obstacle$ بودن یا قرار داشتن آن‌ها در $explored\ set$ را بررسی میکنند و آخرین شرط بررسی میکند که در صورت آمدن از مسیر با هزینه‌ی بیشتر از هزینه‌ی فعلی آن را در $frontier$ قرار ندهیم و $direction$ آن را عوض نکنیم.

در تابع $print\ path$ مسیر را با استفاده از مقادیر قرار داده شده در ماتریس $grid\ dir$ از خانه‌ی هدف تا خانه‌ی شروع چاپ میکنیم و مقدار خانه‌ها را برابر با 6 قرار میدهیم. و در نهایت طول مسیر را چاپ میکنیم.

۵ تابع $search$ و تابع $draw$

```
def Search(h, scale):
    q = frontier.get()
    x, y, cost = q[1], q[2], q[3]
    if (x, y) in explored_set:
        return
    if x == x_goal and y == y_goal:
        print_path()
        frontier.queue.clear()
        return
    for i in range(4):
        x += dRow[i]
        y += dCol[i]
        if isValid(x, y, h(x, y) + (cost + 1) * scale):
            neighbor = ((cost + 1) * scale + h(x, y), x, y, cost
    + 1)
            cost_set[x][y] = (cost + 1) * scale + h(x, y)
            grid_dir[x][y] = directions[i]
            frontier.put(neighbor)
            grid[x][y] = 5
        x = q[1]
        y = q[2]
        grid[x][y] = 4
        explored_set.append((q[1], q[2]))


def Draw():
    # Set the screen background
    screen.fill(BLACK)
    for i in range(row):
        for j in range(col):
            color = WHITE
```

```
            if grid[i][j] == 1:
                color = GREEN
            elif grid[i][j] == 2:
                color = RED
            elif grid[i][j] == 3:
                color = BLUE
            elif grid[i][j] == 4:
                color = VIOLET
            elif grid[i][j] == 5:
                color = PEACHPUFF
            elif grid[i][j] == 6:
                color = GOLD
            pygame.draw.rect(screen,
                             color,
                             [(MARGIN + WIDTH) * j + MARGIN,
                              (MARGIN + HEIGHT) * i + MARGIN,
                              WIDTH,
                              HEIGHT])

    # Limit to 120 frames per second
    clock.tick(120)

    # Go ahead and update the screen with what we've drawn.
    pygame.display.flip()
```

در تابع $search$ در هر مرحله با گرفتن خانه با کوچکترین مقدار f بودن آن خانه در set $explored$ را بررسی میکند و در صورت بودن $return$ میکند و در صورت نبودن به شرط خانه‌ی هدف نبودن همسایه‌های آن را پیدا میکند و در صورت $isValid$ بودن همسایه آن در $frontier$ قرار داده میشود و جهت خانه‌ای که از آن به آن رسیده‌ایم در ماتریس dir $grid$ قرار داده میشود و در نهایت مقدار همسایه‌ها برابر با 5 و خانه‌ی بررسی شده برابر با 4 قرار داده میشود.

در تابع $Draw$ با هر خانه از $grid$ با توجه به مقدار درون آن رنگ آمیزی میشود و در نهایت با 120 fps جدول کشیده میشود و به کاربر نشان داده میشود.

۶ $loop$ نهایی

```
# Set the HEIGHT and WIDTH of the screen
WINDOW_SIZE = [WIDTH * col + (col + 1) * MARGIN, HEIGHT * row +
    (row + 1) * MARGIN]
screen = pygame.display.set_mode(WINDOW_SIZE)

# Set title of screen
pygame.display.set_caption("Search Algorithms")

# Loop until the user clicks the close button.
```

```
done = False
search_finish = False

# Used to manage how fast the screen updates
clock = pygame.time.Clock()


def search_helper():
    message = "For A* with random heuristic, type 1.\nFor greedy best search, type 2.\nFor A* with Manhattan heuristic, type 3.\nfor uniform cost search, type 4.\n"
    type_of_search = int(input(message))
    if type_of_search == 4:
        h = lambda x, y: 0
        scale = 1
    elif type_of_search == 3:
        h = lambda x, y: abs(x_goal - x) + abs(y_goal - y)
        scale = 1
    elif type_of_search == 2:
        h = lambda x, y: abs(x_goal - x) + abs(y_goal - y)
        scale = 0
    elif type_of_search == 1:
        h = lambda x, y: np.random.randint(100,size = 1)[0]
        scale = 1
    while not frontier.empty():
        pygame.event.get()
        Search(h, scale)
        Draw()


# -------- Main Program Loop -----------
while not done:
    for event in pygame.event.get():  # User did something
        if event.type == pygame.QUIT:  # If user clicked close
            done = True  # Flag that we are done so we exit this loop
        elif event.type == pygame.MOUSEBUTTONDOWN:
            # User clicks the mouse. Get the position
            pos = pygame.mouse.get_pos()
            # Change the x/y screen coordinates to grid coordinates
            new_column = pos[0] // (WIDTH + MARGIN)
            new_row = pos[1] // (HEIGHT + MARGIN)

            if search_finish and new_column == y_start and new_row == x_start:
                search_finish = False
                grid_dir = np.full((row, col), 'N')
                grid[np.logical_and(grid != 3, grid != 1, grid
```

```
    != 2)] = 0
53                  grid[x_start, y_start] = 1
54                  grid[x_goal][y_goal] = 2
55              elif not search_finish and new_column == y_start and
     new_row == x_start:
56                  search_finish = True
57                  explored_set = []
58                  cost_set = np.full((row, col), -1)
59                  cost_set[x_start][y_start] = abs(x_start -
    x_goal) + abs(y_start - y_goal)
60                  search_helper()
61                  frontier.put((abs(x_start - x_goal) + abs(
    y_start - y_goal), x_start, y_start, 0))
62              elif grid[new_row][new_column] == 3:
63                  grid[new_row][new_column] = 0
64              elif new_row != x_goal or new_column != y_goal:
65                  grid[new_row][new_column] = 3
66
67      Draw()
68
69  # Be IDLE friendly. If you forget this line, the program will '
    hang'
70  # on exit.
71  pygame.quit()
72
73  save = int(input("Type 1 to save the map as an xml file.\nType 2
     to skip this process.\n"))
74
75  if save == 1:
76      save_as_XML(input("Please enter the file's name.\n") + ".xml
    ", grid)
```

در بخش بالا نیز تابع *search helper* ابتدا با ورودی گرفتن از کاربر و پرسیدن نوع جستجو از او تابع *heuristic* را مشخص میکند و آن را به تابع *search* میدهد و هر بار در *while loop* تابع‌های *Search* و *Draw* صدا زده میشوند. در *while loop* نهایی نیز با کلیک کردن کاربر روی خانه‌ی شروع مقداردهی‌های اولیه از اول شروع میشوند و خانه‌های *grid* همه به جز *obstacle* ها پاک میشوند تا بتوان با دوباره فشردن خانه‌ی شروع جستجو را از آغاز شروع کرد.
در نهایت نیز در صورت تمایل کاربر نقشه‌ی طراحی شده‌ی او در یک فایل *xml* ذخیره میشود.

پاسخ به سوالات:

سوال ۱:

در کد نوشته شده از 2 هیوریستیک $Manhattan$ و $Random$ استفاده شده است. انتخاب هیوریستیک $Manhattan$ به دلیل سازگار بودن و در نتیجه قابل‌قبول بودن آن انتخاب شده است که این خواص در مقابل اثبات شده‌اند:

$$h^*(x,y) = |x - x_{goal}| + |y - y_{goal}| \rightarrow h(x,y) = h^*(x,y) \rightarrow h(x,y) \leq h^*(x,y)$$

که در نابرابری بالا $h^*(x,y)$ فاصله‌ی واقعی نقطه‌ی (x,y) از هدف است. حال به اثبات سازگاری این هیوریستیک میپردازیم:

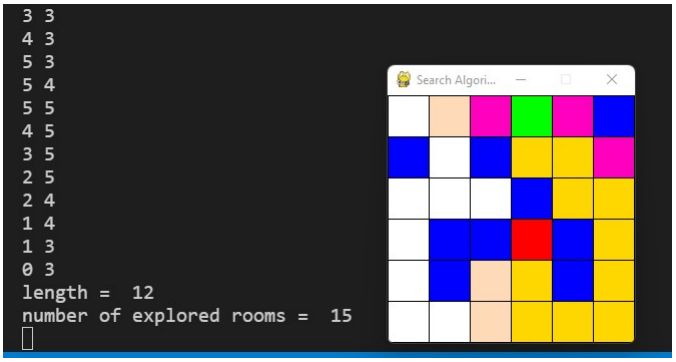
$$c(n,n') = 1 \rightarrow h(x,y) \leq 1 + h(x',y')$$

دقت کنید نابرابری بالا به این دلیل برقرار است که مجاز به حرکت قطری نیستیم و در بهترین حالت تابع هیوریستیک با یک حرکت یک مقدار کم میشود که با جمع کردن آن با 1 مقدار آن تغییری نمیکند.
هیوریستیک $Random$ به دلیل غیرقابل قبول بودن و ناسازگار بودن آن انتخاب شده است تا با هیوریستیک $manhattan$ مقایسه شود. دقت کنید غیرقابل قبول بودن این هیوریستیک به این دلیل است که هر عددی مهم نیست چقدر بزرگ ممکن است انتخاب شود و مقدار تابع هیوریستیک بزرگتر از فاصله‌ی کنونی از هدف شود.
از طرفی دقت کنید غیرقابل قبول بودن این هیوریستیک ناسازگاری آن را نتیجه میدهد زیرا اگر سازگار بود قابل‌قبول بودن نیز نتیجه میشد.

سوال ۲:

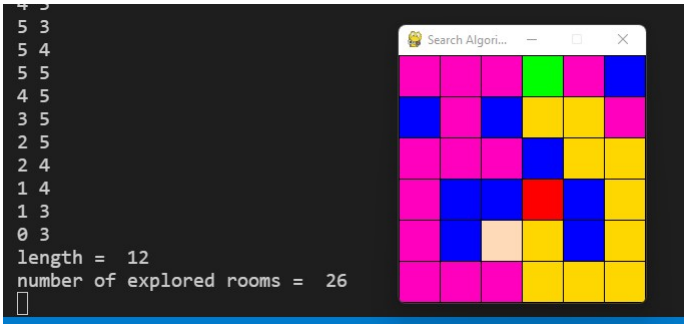
مسیر و طول آن و تعداد خانه‌های جستجو شده برای چند الگوریتم سرچ را میتوانید در عکس‌های مقابل مشاهده کنید.

شکل ۱: نتیجه‌ی اجرا با هیوریستیک منهتن



شکل ۲: نتیجه‌ی اجرا با الگوریتم حریصانه

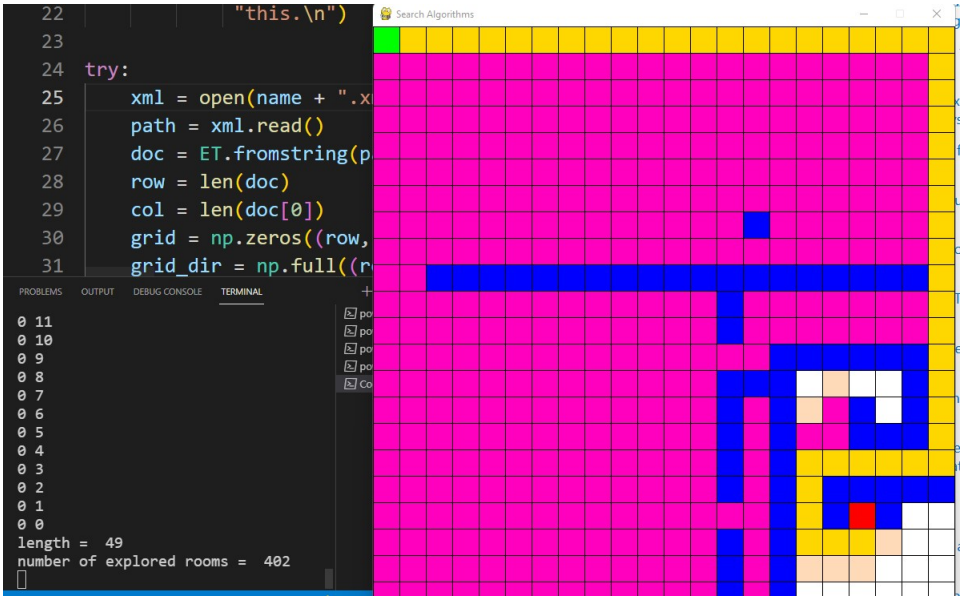
شکل ۳: نتیجه‌ی اجرا با هیوریستیک رندوم



شکل ۴: نتیجه‌ی اجرا با الگوریتم یکنواخت

سوال‌های ۳ و ۴:

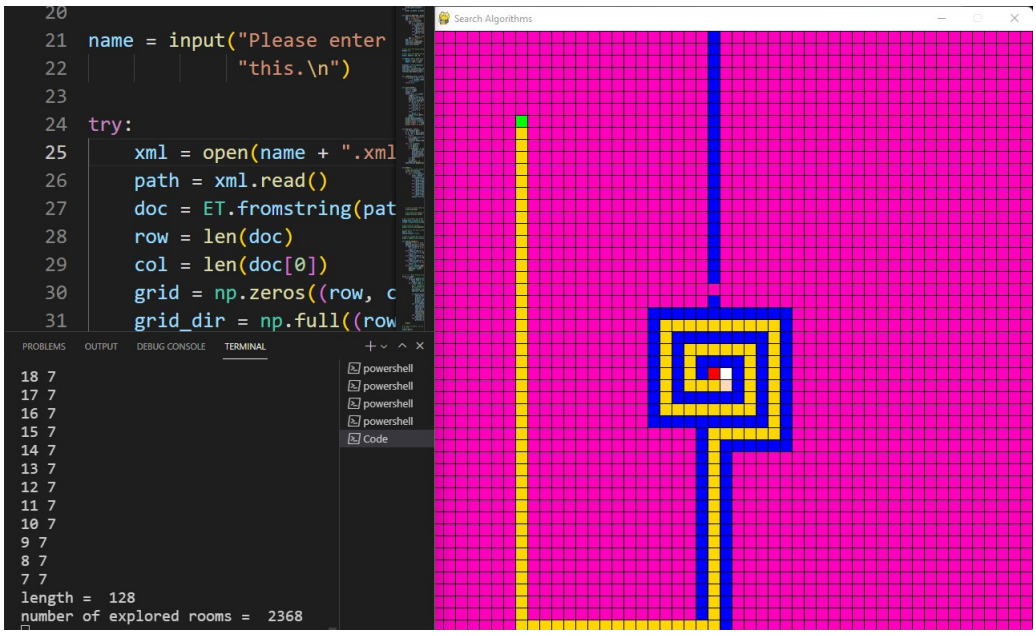
دقت کنید برای آنالیز کردن الگوریتم‌های مختلف از عکس‌های مقابل که نتیجه‌ی اجرای کد روی نقشه‌های مختلف هستند کمک میگیرم:



شکل ۵: نتیجه‌ی اجرا با هیوریستیک منهتن

شکل ۶: نتیجه‌ی اجرا بدون هیوریستیک

شکل ۷: نتیجه‌ی اجرا با هیوریستیک منهتن



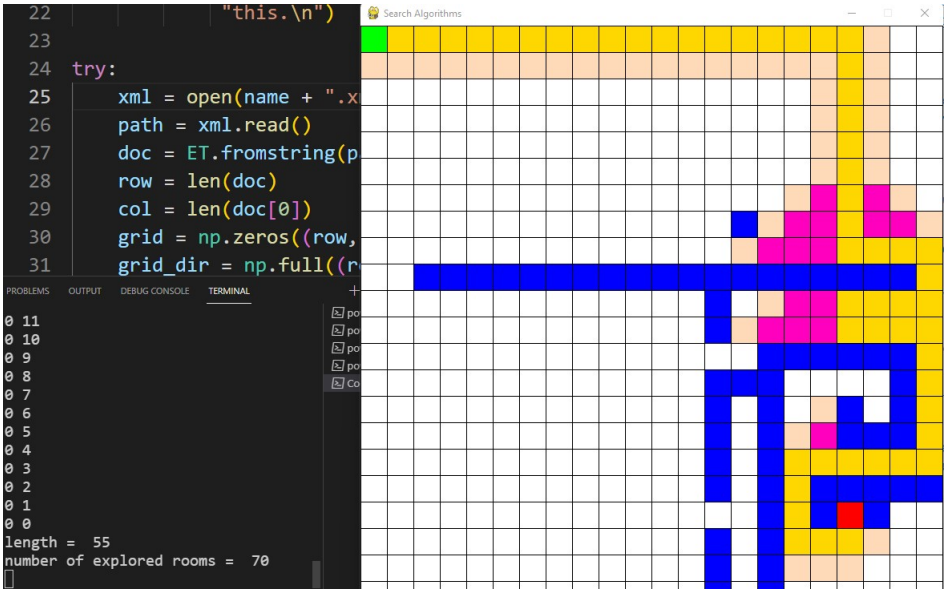
شکل ۸: نتیجه‌ی اجرا بدون هیوریستیک

همانطور که در تمرین تئوری نیز ثابت شد در یک مستطیل X در Y هیوریستیک منهتن

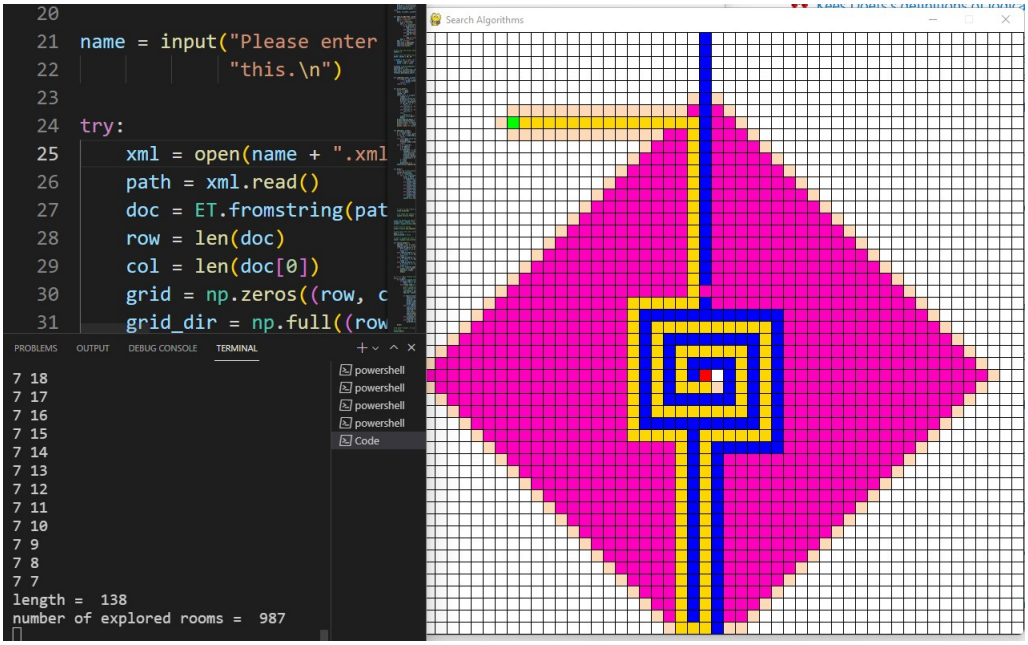
با وجود پیدا کردن مسیر بهینه در بدترین حالت مجبور به چک کردن تمام خانه‌ها میشود که این حقیقت در عکس‌های بالا قابل مشاهده است. البته لازم به توجه است که هنوز چند خانه مانده‌اند که جستجو نشده‌اند.
با حذف هیوریستیک الگوریتم ما تبدیل به الگوریتم سرچ یکنواخت میشود که همانطور در کلاس اثبات شد بهینه و کامل است و کوتاه‌ترین مسیر را میدهد. ولی تفاوت آن با الگوریتم A^* که شاید در عکس‌های بالا به دلیل بودن نقشه در یک آرایه 2 بعدی مشخص نباشد این است که الگوریتم یکنواخت به مراتب خانه‌های بیشتری را نسبت به الگوریتم A^* سرچ میکند و برخلاف آن یک الگوریتم سرچ ناآگاهانه است.
دقت کنید اگر هیوریستیک ما ایده آل باشد کانتور آن الگوریتم به صورت یک خط خواهد بود و این یعنی یک الگوریتم A^* با هیوریستیک ایده آل بدون چک کردن خانه‌ی اضافه‌ای مستقیم به هدف میرود.

سوال ۵

در این بخش الگوریتم سرچ بهترین حریصانه را بررسی میکنیم که فرق آن با الگوریتم A^* در این است که موقع انتخاب نود برای بسط دادن تنها به مقدار هیوریستیک آن نگاه میکنیم و هزینه‌ی آن تا اینجا اهمیتی ندارد و این امر همانطور که در کلاس نیز اثبات شد باعث میشود الگوریتم حریصانه بهینه نباشد.



شکل ۹: نتیجه‌ی اجرا با الگوریتم حریصانه



شکل ۱۰: نتیجه‌ی اجرا با الگوریتم حریصانه

همانطور که از عکس‌های بالا نیز مشخص است، این الگوریتم به مراتب خانه‌های کمتری را بررسی میکند ولی با اینکار ممکن است مسیر بهینه را پیدا نکند که هر دو این موارد در عکس‌های بالا قابل مشاهده است.
در بعضی مواقع ممکن است الگوریتم حریصانه برای جستجوی سریعترش نسبت به $A*$ انتخاب شود و طول مسیر فدای هزینه‌ی زمان اجرا شود.
دقت کنید تمام عکس‌ها و کد و گزارش و نقشه‌های استفاده شده در این لینک قابل مشاهده است.