T. Shanmuga Ganesh

mouse: "left" click - Choose, "Right" click - remove } Block

keyboard: "C" - Clear Board, "Space" - run

1.A* Path Finding Algorithm

(Informed search)

```
# import
import pygame   # (for Visualization)
import math
from queue import PriorityQueue   # (minHeap (or) MaxHeap)
WIDTH = 800;WIN = pygame.display.set_mode((WIDTH, WIDTH))   # || Set the Frame   (600 crossed out, 800 written in)
pygame.display.set_caption("A* Path Finding Algorithm")   # || title on that Frame
# color
RED = (255, 0, 0);GREEN = (0, 255, 0);BLUE = (0, 255, 0);YELLOW = (255, 255, 0)
WHITE = (255, 255, 255);BLACK = (0, 0, 0);PURPLE = (128, 0, 128);ORANGE = (255, 165 ,0)
GREY = (128, 128, 128);TURQUOISE = (64, 224, 208)
class Spot:   # || it keep track of each block, what color, position/location
        def __init__(self, row, col, width, total_rows):
                self.row = row   # width of each cube is 800/50 = 16
                self.col = col
                self.x = row * width   # ] Position (drawing Borders for cube), Actual co-ord pos. -> [Length]
                self.y = col * width
                self.color = WHITE   # || Starting color
                self.neighbors = []
                self.width = width
                self.total_rows = total_rows
        # is
        def get_pos(self): return self.row, self.col   # || Get the current position. (5,6) rxc
        def is_closed(self): return self.color == RED   # || we are not looking at it anymore
        def is_open(self): return self.color == GREEN   # || open_set
        def is_barrier(self): return self.color == BLACK   # || No considerable / obstacle
        def is_start(self): return self.color == ORANGE   # || Start point
        def is_end(self): return self.color == TURQUOISE   # || End Node
        # make
        def reset(self): self.color = WHITE   # || reset
        def make_start(self): self.color = ORANGE
        def make_closed(self): self.color = RED
        def make_open(self): self.color = GREEN
```

Spot → [square] 16x16, x, y

```
def make_barrier(self): self.color = BLACK
def make_end(self): self.color = TURQUOISE || Life Blue
def make_path(self): self.color = PURPLE || Path
```

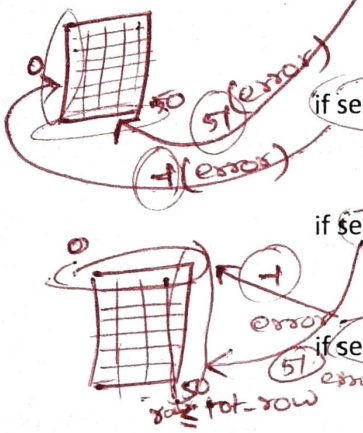
Actual co-ord Pos.

```
def draw(self, win):
    pygame.draw.rect(win, self.color, (self.x, self.y, self.width, self.width))
```

width, Heigh

```
def update_neighbors(self, grid):  || Check neighbour is not Barrier
    self.neighbors = []
    if self.row < self.total_rows - 1 and not grid[self.row + 1][self.col].is_barrier(): # DOWN
        self.neighbors.append(grid[self.row + 1][self.col])
    if self.row > 0 and not grid[self.row - 1][self.col].is_barrier(): # UP
        self.neighbors.append(grid[self.row - 1][self.col])
    if self.col < self.total_rows - 1 and not grid[self.row][self.col + 1].is_barrier(): # RIGHT
        self.neighbors.append(grid[self.row][self.col + 1])
    if self.col > 0 and not grid[self.row][self.col - 1].is_barrier(): # LEFT
        self.neighbors.append(grid[self.row][self.col - 1])
```

(Valid Neighbor)



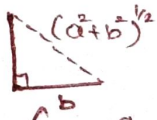
down, up, right, left

Recent Node/spot

```
def __lt__(self, other):  || lt - lessthan ; we cmp 2 spot object.
    return False
```

Class ↔ Ends

```
def h(p1, p2):  || H-Factor → Heuristic function
    x1, y1 = p1;x2, y2 = p2  || P1 - Point 1 , P2 - Point 2
    return abs(x1 - x2) + abs(y1 - y2)  || Manhattan dist / l-dist. (or) (taxi-cab dist)
```




```
def reconstruct_path(came_from, current, draw):  || to draw the path that we found!
    while current in came_from:  || in table
        current = came_from[current]  || traversing the prev Node!
        current.make_path()  || purple - path
        draw()  || draw / paint in screen.
```

back track

```
def algorithm(draw, grid, start, end):  || A* Algorithm
    count = 0  || keep track of Added Nodes to open set
    open_set = PriorityQueue()  || our set (open set) → "return smallest every time"
    open_set.put((0, count, start))  || put / add / push the start Node / spot.
    came_from = {}  || dictionary - (last) → Back Tracking
```

f score, track, node

count is used for; when the f score is same then (count) which is added 1st returns. (No need to update)

for row in grid:
for spot in row:

```
g_score = {spot: float("inf") for row in grid for spot in row}   (∞)
g_score[start] = 0   || for Start Node
f_score = {spot: float("inf") for row in grid for spot in row}   (∞)
f_score[start] = h(start.get_pos(), end.get_pos())   || Abs b/w Start & end Node
open_set_hash = {start}   || add the start to set (dict)
while not open_set.empty():
    for event in pygame.event.get():
        if event.type == pygame.QUIT:   → ["x" on top of frame]
            pygame.quit()   || exit
    current = open_set.get()[2]   → 2nd Index  {(0, 0, A)} openset  (Priority Queue) - Min Value by (Heap sort)
    open_set_hash.remove(current)   || pop cur
    if current == end:
        reconstruct_path(came_from, end, draw)
        end.make_end()
        return True
    for neighbor in current.neighbors:   || Consider all the neighbors of curr Node
        temp_g_score = g_score[current] + 1   || assume Node cost is 1. (it's valid)
        if temp_g_score < g_score[neighbor]:   || Better way/path
            came_from[neighbor] = current   || update/add
            g_score[neighbor] = temp_g_score   || g-score
            f_score[neighbor] = temp_g_score + h(neighbor.get_pos(),
end.get_pos())   || f score
            // f(n) = g(n) + h(n)
            if neighbor not in open_set_hash: count += 1   || not in neighbor
                open_set.put((f_score[neighbor], count, neighbor))   || New Neighbor of Better path
                open_set_hash.add(neighbor)
                neighbor.make_open()   || green
    draw()   || lambda function
    if current != start: current.make_closed()   || red
return False   || we donot find a path
def make_grid(rows, width):   || it's the Data Structure that stores/Holds all of the spots so that we can Manipulate/traverse them
    grid = []   || clear
```

Our Algm runs this.

exit

[cur == end] we found the shortest path

[g-score create] Cmp update

add in Table

```
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            spot = Spot(i, j, gap, rows)
            grid[i].append(spot)
    return grid
def draw_grid(win, rows, width):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(win, GREY, (0, i * gap), (width, i * gap))
        for j in range(rows):
            pygame.draw.line(win, GREY, (j * gap, 0), (j * gap, width))
def draw(win, grid, rows, width):
    win.fill(WHITE)
    for row in grid:
        for spot in row: spot.draw(win)
    draw_grid(win, rows, width)
    pygame.display.update()
def get_clicked_pos(pos, rows, width):
    gap = width // rows
    y, x = pos
    row = y // gap
    col = x // gap
    return row, col
def main(win, width):
    ROWS = 50
    grid = make_grid(ROWS, width)
    start = None;= None;run = True
    while run:
        draw(win, grid, ROWS, width)
```

- Quotient (16)
- gap = width // rows (800 // 50) → through this gap we will draw Border to form as grid; 800 // 50 = 0 ← Gap length
- grid.append([]) → (Spot1, Spot2 ...) [[ ], [ ], [ ], ...] (50 * 2) 2D List
- spot = Spot(i, j, gap, rows): row, col, width, total_rows
- def draw_grid: || to draw grid
- pygame.draw.line(win, GREY, (0, i * gap), (width, i * gap)): color; Start pos.; end Pos. — Horizontal line for each one
- pygame.draw.line(win, GREY, (j * gap, 0), (j * gap, width)): x, y — Vertical line
- def draw: || Paint; 1st win.fill(WHITE); 2nd for row in grid; 3rd draw_grid
- for spot in row: spot.draw(win) → draw cube on own color
- pygame.display.update() || update the display
- def get_clicked_pos(pos, ...): mouse pos. || Mouse position translate into row, col → || representing the cube that we actually clicked on.
- row = y // gap → width of each cube
- def main(win, width): || Main program
- ROWS = 50 || total No. of rows in each 50.
- grid = make_grid(ROWS, width): 50, 800 [[ ] [ ] ...]
- start = None; end = None
- draw(win, grid, ROWS, width): 50, 800 || draw each time

Event ← (Interupt) keypress

```
for event in pygame.event.get():                    → "X" Button on top
    if event.type == pygame.QUIT: run = False
    if pygame.mouse.get_pressed()[0]: # LEFT        click
        pos = pygame.mouse.get_pos()                || get pos. (x & y) ← tuple
        row, col = get_clicked_pos(pos, ROWS, width)   || get row, col
        spot = grid[row][col]
        if not start and spot != end:               || Not start choosen
            start = spot;.make_start()              ↙ Make it (spot) as start
        elif not end and spot != start:             || Not end choosen
            end = spot;end.make_end()               ↙ Make it (spot) as end
        elif spot != end and spot != start:         || if start & end are choosen
            spot.make_barrier()                     ↙ it must be a Barrier.
    elif pygame.mouse.get_pressed()[2]: # RIGHT     click
        pos = pygame.mouse.get_pos()                || get Pos. (x & y)
        row, col = get_clicked_pos(pos, ROWS, width)
        spot = grid[row][col]
        spot.reset()                                ← reset (white)
        if spot == start: start = None              } reset start & end
        elif spot == end: end = None
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_SPACE and start and end:   || Start & end must Be in Board.
            for row in grid:                        → [ ] [ ] ...
                for spot in row:                    → [...]
                    spot.update_neighbors(grid)     || update Neighbors
            algorithm(lambda: draw(win, grid, ROWS, width), grid, start, end)
                                                    || Calling the Algorithm
        if event.key == pygame.K_c:
            start = None;end = None
            grid = make_grid(ROWS, width)
pygame.quit()                                       || end when loop (after)
main(WIN, WIDTH)                                    || Calling the main Method
```

(Set Spot)

Mouse (left click)

doesn't Allow start & end on same spot

(re-Set spot)

Mouse Right click

(KeyBoard pressed)

(space)

~~end)~~

(c)

|| lambda ⇒ is that returns a function (anonymous function)

—— X ——

Program Ends

↗ A* Algm — it is a Informal Search (Optimal) → {Not Brute Force}.

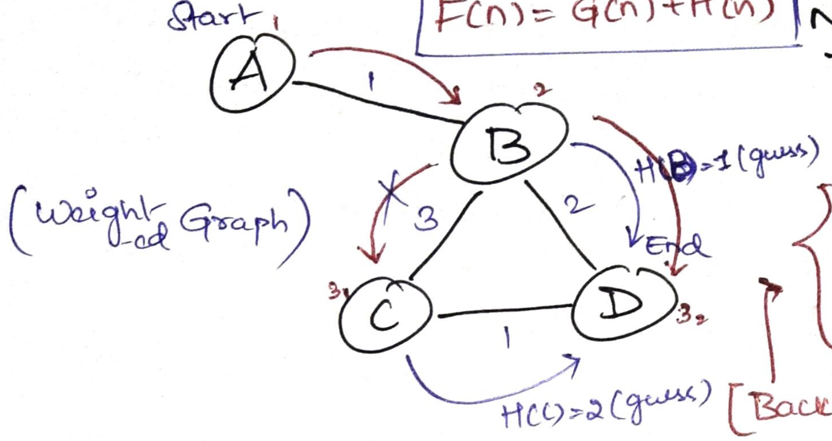
[Heuristic search] - knowledge Based search

Open = {(0, A), (2, B), (6, C), (3/D)} // it will keep track of Node's next yet to visit.

↑ [Priority Queue] — 1st ↑ Last, F-Value (Pop & use it), 2nd step, 3rd step, due to D

$F(n) = G(n) + H(n)$

Start,



Node	F	G	H	Last [we came from] (at same Node)
A	0	0	0	-
B	~~0~~ 2	~~0~~ 1	~~0~~ 1	A
C	~~0~~ 6	~~0~~ 4 (1+3)	~~0~~ 2	B
D	~~0~~ 3	~~0~~ 3 (1+2)	~~0~~ 0 (end)	B

[Back track]

H(n) (Heuristic guess) — it gives us an estimate of the Node 'n' to the end Node.

↳ eg:- A to D (No direct path) ⇒ from A to D, 'x' is the distance Based on ABsolute distance.

distance formula (Guess), (i) Euclidean (ii) ManHattan (We are gonna use this in our pgm)

G(n) (dist. travelled ↳ shortest) — it gives us the current shortest distance from current Node to start Node.

↳ eg:- G(C) is 1, from A to C ⇒ 1 (from Start Node till this Node (path)).

F(n) (estimate, How far away we from end Node) — it gives us the estimate from 'n' nodes to end node.

↳ eg:- from 'C', it gives approx. / guess as 1.

Procedure :- 1) We Had our table set ready & open set's (A) is popped for next step.

2) find A's Neighbours, edge cost is cmp with G score, if less update G score.

↳ find H(B) = 1 (guess), update H score & [F(n) = G(n) + H(n)] thus F score is 2.

↳ then push 'B' in open set "(2, B)".

3) Pop (B) from open set & repeat the above steps

↳ till popped out Element/Node is the End.

—x—