A Mini Project Report

on

Path Finding Visualizer

In Subject: Design and Analysis of Algorithm

by

Member Name	**Member Roll No**
Siddhant Shinde	372049
Rhishikesh Sonawne	372053
Siddhesh Songire	372054
Sakshi Wani	372062

Department of Artificial Intelligence and Data Science

VIIT

2022-2023

Contents

1.1 Introduction

A pathfinding visualizer is a tool that helps users understand and visualize the different algorithms used for pathfinding. Pathfinding is the process of finding a path from one point to another on a map or grid. This is a common problem in computer science and is often used in video games to navigate NPCs (non-player characters) around the game world.

A pathfinding visualizer typically shows a grid or map, with a starting point and an end point, and allows users to select different algorithms to see how they find the path between the two points. The visualizer will then animate the process of the algorithm as it searches the grid and finds the path.

This can be a helpful tool for learning about the different pathfinding algorithms and how they work, as well as for testing and comparing the performance of different algorithms on different types of maps.

1.2 Requirements

Hardware Requirement for Development of Project: (minimum)

- System Processor: Pentium P4
- Motherboard: Genuine Intel
- Memory: 512 MB of RAM, 1GB recommended.
- Display: 1024x 768 or higher-resolution display with 16 bits colors of androidmobile phone.

Software Requirement for Development of Project: (minimum)

- Operating system: Windows XP or higher.
- Software: Vs code

2.1 A* algorithm

The A* algorithm is a popular algorithm used for pathfinding in video games and other applications. It is an extension of the popular Dijkstra's algorithm, which is used for finding the shortest path in a graph.

The A* algorithm uses a combination of the actual distance from the starting point and the estimated distance to the end point (known as the heuristic) to find the optimal path. This allows it to be more efficient than Dijkstra's algorithm, as it avoids exploring paths that are unlikely to be the best option.

The A* algorithm works by maintaining two lists of nodes: an open list and a closed list. The open list contains nodes that have been discovered but not yet evaluated, and the closed list contains nodes that have been evaluated.

The algorithm starts at the starting point and adds all of its neighbors to the open list. It then selects the node on the open list with the lowest cost (the actual distance from the starting point plus the estimated distance to the end point) and evaluates it. If the node is the end point, the algorithm is finished and the path is found. Otherwise, the algorithm adds the node's neighbors to the open list and repeats the process.

The A* algorithm continues to iterate through the open list until the end point is found or there are no more nodes to evaluate. If the end point is not found, then there is no path between the starting and ending points.

2.2 Working of A*

When A* enters a problem, firstly, it calculates the cost to travel to the neighboring nodes and chooses the node with the lowest cost.

The $f(n)$ denotes the cost function, A* chooses the node with the lowest $f(n)$ value. Here ‘n’ denotes the neighboring nodes.

$$f(n) = g(n) + h(n)$$

$g(n)$ = (path weight)shows the shortest path’s value from the starting node to node n which is simply path weight.

h(n) = (heuristic value). The estimated movement cost to move from that given square on the grid to the destination.

2.3 Algorithm

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)

3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search

        ii) else, compute both g and h for successor
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

          successor.f = successor.g + successor.h

        iii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor

        iV) if a node with the same position as
```

```
            successor  is in the CLOSED list which has
            a lower f than successor, skip this successor
            otherwise, add  the node to the open list
     end (for loop)

    e) push q on the closed list
    end (while loop)
```

2.4 Code and Output:

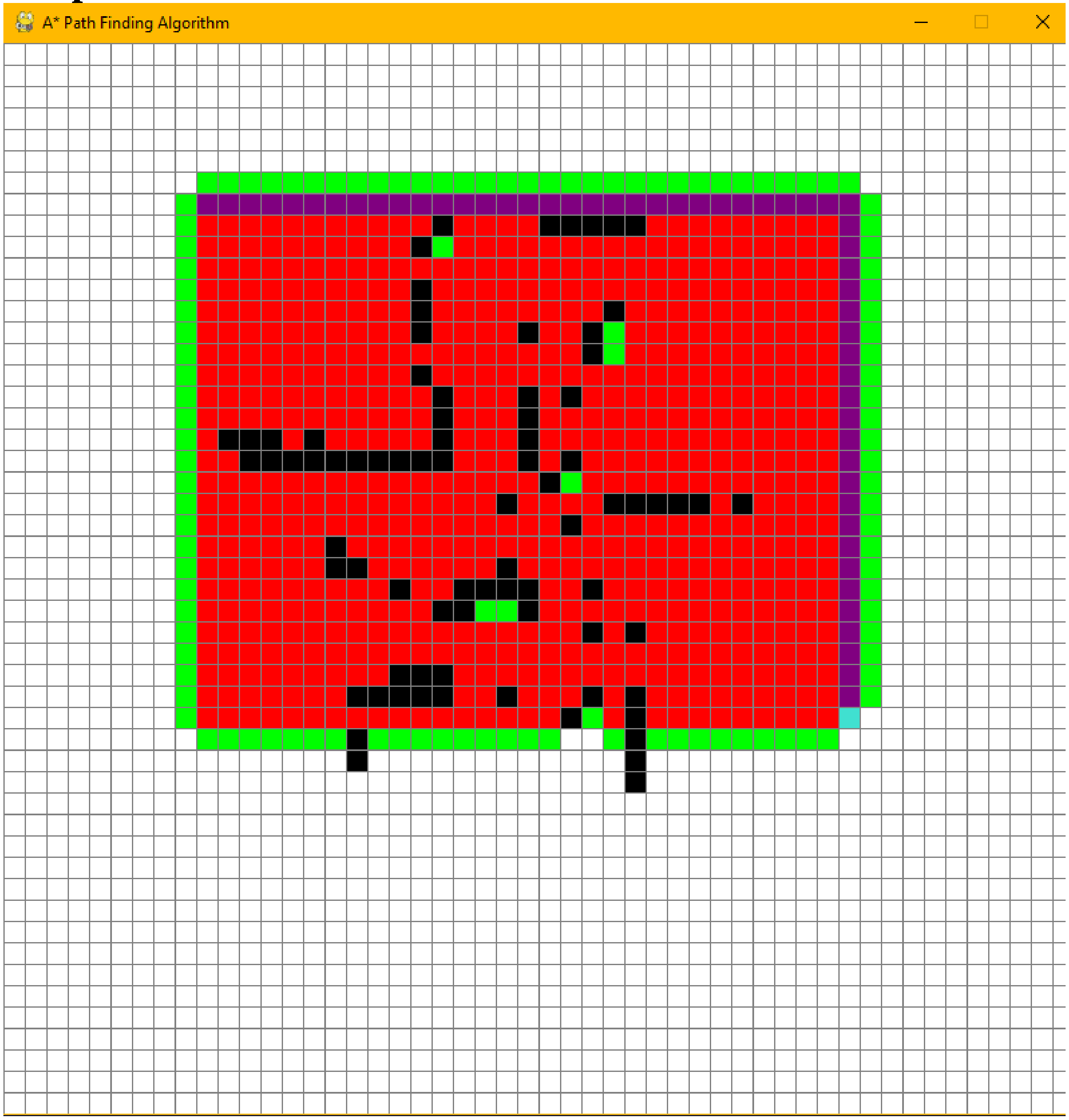
```
# 1 importing modules
import pygame
import math
from queue import PriorityQueue

# 2 size of display windows (dimensions)
WIDTH = 800
WIN = pygame.display.set_mode((WIDTH, WIDTH))
pygame.display.set_caption("A* Path Finding Algorithm") # caption

# 3 defining colous (will be througt)
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)

# 4 defining bunch of functions
# where the node is keep track where the node is
class Spot:
	def __init__(self, row, col, width, total_rows):
		self.row = row
		self.col = col
		self.x = row * width
		self.y = col * width
		self.color = WHITE
		self.neighbors = []
		self.width = width
		self.total_rows = total_rows

	def get_pos(self):
		return self.row, self.col
```

```
        # if visited colour red
        def is_closed(self):
                return self.color == RED

        def is_open(self):
                return self.color == GREEN

        def is_barrier(self):
                return self.color == BLACK

        def is_start(self):
                return self.color == ORANGE

        def is_end(self):
                return self.color == TURQUOISE

        def reset(self):
                self.color = WHITE

        def make_start(self):
                self.color = ORANGE

        def make_closed(self):
                self.color = RED

        def make_open(self):
                self.color = GREEN

        def make_barrier(self):
                self.color = BLACK

        def make_end(self):
                self.color = TURQUOISE

        def make_path(self):
                self.color = PURPLE

# draw a colour on given location
        def draw(self, win):
                pygame.draw.rect(win, self.color, (self.x, self.y, self.width, self.width))

        def update_neighbors(self, grid):
                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])

        def __lt__(self, other):
            return False

# 5 defining heuristic function (manhatten distance)
def h(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)


def reconstruct_path(came_from, current, draw):
    while current in came_from:
        current = came_from[current]
        current.make_path()
        draw()

# last A* algorithm
def algorithm(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue() # defining PQ
    open_set.put((0, count, start)) # add in a start node(f score, count, start)
    came_from = {} # keep track of path
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(), end.get_pos()) # f score of start will herustic

    open_set_hash = {start}

    # while pq is not empty
    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        current = open_set.get()[2] # select a min f score node fro pueue
        open_set_hash.remove(current) # remove it

        if current == end:
            reconstruct_path(came_from, end, draw) # draw path
            end.make_end()
            return True

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1

            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
                if neighbor not in open_set_hash:
```

```
                                                  count += 1 # if f score is same then it will check count
                                                  open_set.put((f_score[neighbor], count, neighbor)) # if we find better
we update

                                                  open_set_hash.add(neighbor)
                                                  neighbor.make_open()

                draw()

                if current != start:
                        current.make_closed()

        return False # if did not find the path

# 6 Create grid so that we can traverse its 2d array which has spots
def make_grid(rows, width):
        grid = []
        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

# 7 draw grid lines (squares)
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)) # horizontal lines
                for j in range(rows):
                        pygame.draw.line(win, GREY, (j * gap, 0), (j * gap, width)) # vertical lines

# 8 main draw function
def draw(win, grid, rows, width):
        win.fill(WHITE) # initially white

        #traverse and draw spots according to grid
        for row in grid:
                for spot in row:
                        spot.draw(win) # defined in class

        draw_grid(win, rows, width)
        pygame.display.update()

# 9 which spot to change colour when clicked
def get_clicked_pos(pos, rows, width):
        gap = width // rows
        y, x = pos

        row = y // gap
        col = x // gap
        return row, col

#*-------------------------------Main logic-------------------------------------------------
# All checks, collision , on space
def main(win, width):
        ROWS = 50
        grid = make_grid(ROWS, width)

        start = None
        end = None

        run = True
        while run:
                draw(win, grid, ROWS, width) # call draw in each loop when we click
                for event in pygame.event.get(): # looping through event
                        if event.type == pygame.QUIT: # if we close
                                run = False

                        if pygame.mouse.get_pressed()[0]: # clickd on LEFT mouse button
                                pos = pygame.mouse.get_pos()
                                row, col = get_clicked_pos(pos, ROWS, width) # gives row and col in grid
                                spot = grid[row][col]
                                if not start and spot != end:
                                        start = spot
                                        start.make_start() #  set color

                                elif not end and spot != start:
                                        end = spot
                                        end.make_end() # set color

                                elif spot != end and spot != start:
                                        spot.make_barrier()

                        elif pygame.mouse.get_pressed()[2]: # RIGHT
                                pos = pygame.mouse.get_pos()
                                row, col = get_clicked_pos(pos, ROWS, width)
                                spot = grid[row][col]
                                spot.reset()    #
                                if spot == start:
                                        start = None
                                elif spot == end:
                                        end = None

# if pressed space
                        if event.type == pygame.KEYDOWN:
                                if event.key == pygame.K_SPACE and start and end:
                                        for row in grid:
                                                for spot in row:
                                                        spot.update_neighbors(grid)

                                        algorithm(lambda: draw(win, grid, ROWS, width), grid, start, end)

                                if event.key == pygame.K_c:
```

```
                    start = None
                    end = None
                    grid = make_grid(ROWS, width)

        pygame.quit()

main(WIN, WIDTH)
```

Output



3 Future Scope and Conclusion

The A* pathfinding algorithm is a widely-used technique in computer science and game development for efficiently finding the shortest path between two points. It is likely that the use of this algorithm will continue to be important in the future, particularly as technology continues to advance and more complex environments and situations arise that require efficient pathfinding.

One potential future application of the A* pathfinding visualizer is in the development of self-driving cars. This technology is becoming increasingly sophisticated and will require algorithms like A* to navigate complex road networks and avoid obstacles in real-time. The visualizer could be used to help engineers and developers test and debug their pathfinding algorithms to ensure that they are functioning correctly and efficiently.

Another potential application of the A* pathfinding visualizer is in the field of robotics. As robots become more advanced and are used in a wider variety of environments, they will need to be able to navigate these environments efficiently and avoid obstacles. The A* visualizer could be used to help designers and engineers develop and test pathfinding algorithms for robots to ensure that they are able to navigate complex environments efficiently.

Overall, the future scope of the A* pathfinding visualizer is likely to be wide-ranging, with applications in a variety of fields including self-driving cars, robotics, and game development. As technology continues to advance and more complex environments and situations arise, efficient pathfinding algorithms like A* will become increasingly important, and tools like the A* visualizer will continue to be useful for developers and engineers working in these fields.

References:

- https://www.geeksforgeeks.org/a-search-
- https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm#:~:text=A*%20Search%20Algorithm%20is%20a,path%20algorithm%20(Dijkstra's%20Algorithm).

- https://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/
- https://www.javatpoint.com/ai-informed-search-algorithms