DAYANANDA SAGAR UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING
DAYANANDA SAGAR UNIVERSITY
KUDLU GATE
BANGALORE - 560068

Project report

ON

*"PATHFINDING VISUALIZATION USING A * ALGORITHM"*

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE & ENGINEERING

Submitted by: **-**

Bharath Kumar N - ENG19CS0064
Chandrashekar V - ENG19CS0070
Chethan B S - ENG19CS0074
Chethan R - ENG19CS0075
Chethan Rao S - ENG19CS0076

Under the supervision of: **-**

PROF. Kavyashree D

CERTIFICATE

This is to certify that the report on ADA project entitled "**Path finding visualization using A* algorithm"** being submitted by Bharath Kumar N - ENG19CS0064 ,Chandrashekar - ENG19CS0070, Chethan BS- ENG19CS0074 ,Chethan R - ENG19CS0075 , Chethan Rao S - ENG19CS0076 to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the IV semester B.Tech CSE of this university during the academic year 2020-2021.

.

Date:_____ _____

Signature of the Faculty in Charge

_____

Signature of the Chairman

DECLARATION

As by the instructions given by you ma'am, we the 2nd team have done a Project entitled- "**Path finding visualization using A* algorithm**", it has not been submitted for the award of any degree, diploma or the seminar of any other college or university.

Bharath Kumar N - ENG19CS0064
Chandrashekar V - ENG19CS0070
Chethan B S - ENG19CS0074
Chethan R - ENG19CS0075
Chethan Rao S - ENG19CS0076

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of task would be incomplete without the mention of the people who made it possible and whose constant guidance and encouragement crown all the efforts with success.

We are especially thankful to our **Chairman Dr.Sanjay Chitnis**, for providing necessary departmental facilities, moral support and encouragement.

We are very much thankful to **PROF. Kavyashree D** for providing help and suggestions in completion of this mini project successfully.

We have received a great deal of guidance and co-operation from our friends and we wish to thank all that have directly or indirectly helped us in the successful completion of this project work.

Bharath Kumar N - ENG19CS0064
Chandrashekar V - ENG19CS0070
Chethan B S - ENG19CS0074
Chethan R - ENG19CS0075
Chethan Rao S - ENG19CS0076

ABSTRACT:

Pathfinding algorithm addresses the problem of finding the shortest path from source to destination and avoiding obstacles. One of the greatest challenges in the design of realistic Artificial Intelligence (AI) in computer application is agent movement. Pathfinding strategies are usually employed as the core of any AI movement system. In this work, A* search algorithm is used to find the shortest path between the source and destination on image that represents a map or a maze. Finding a path through a maze is a basic computer science problem that can take many forms. The A* algorithm is widely used in pathfinding and graph traversal. A famous person algorithm is an informative algorithm in comparison to others that means it's going to handiest use the course which has the possibility of the usage of the shortest and the maximum green course. After finding the course we use Manhattan technique to attract the direction. The algorithm uses the formula f(n)=g(n)+h(n) g(n) = indicates the shortest course's value from the beginning node to node n h(n)=The heuristic approximation of cost of the node. To show how the set of rules runs we will put in force it in python. This will supply us a 2D representation of our version and how the algorithm works.

INTRODUCTION

Pathfinding generally refers to find the shortest route between two end points. Examples of such problems include transit planning, telephone traffic routing, maze navigation and robot path planning. As the importance of industry increases, pathfinding has become a popular and frustrating problem in many industries. Applications like role-playing games and real-time strategy games often have characters sent on missions from their current location to a predetermined or player determined destination. The most common issue of pathfinding in a video game is how to avoid obstacles cleverly and seek out the most efficient path over different terrain. Early solutions to the problem of pathfinding in computer games, such as depth first search, iterative deepening, breadth first search, Dijkstra's algorithm, best first search, A* algorithm, and iterative deepening A*, were soon overwhelmed by the sheer exponential growth in the complexity of the game. More efficient solutions are required so as to be able to solve pathfinding problems on a more complex

environment with limited time and resources. Because of the huge success of A* algorithm in path finding, many researchers are pinning their hopes on speeding up A* so as to satisfy the changing needs of the game. Considerable effort has been made to optimize this algorithm over the past decades and dozens of revised algorithms have been introduced successfully. Examples of such optimizations include improving heuristic methods, optimizing map representations, introducing new data structures and reducing memory requirements.

PROBLEM STATEMENT:

Assume that you are the head of the corporation team that clears the roads in a city like Bengaluru for traffic. It is the time of the year with cyclones causing trees to fall. It is your duty to clear the roads for traffic. The problem here is defined as to find the shortest route that travels through all the roads in an area clearing them. The shortest path includes all the roads but only once. Help your cleaning robot in finding such a path given a starting point and the map. Here the area map is represented as a graph. The algorithm takes the starting point and the graph as the input and produces the shortest path covering all the edges only once. Pathfinding is the plotting, by a computer application, of the shortest route between two points. The task is to build a path finding visualizer tool to visualize the A* pathfinding algorithm that finds the least cost path from a given initial node to goal node.

PROPOSED METHOD:

A Path finding algorithm has been proposed for grid-based graph, that finds the least cost path from a given source node to destination node. The solutions for the path finding algorithm has been analyzed to find the shortest path between two points even with obstacles. Pathfinding requires significant amount of resources especially in movement intensive application. For that reason, an efficient and inexpensive approach is needed. In this work, images from strategy games that represent a map. A* is widely used in pathfinding and graph traversal. This algorithm will be used to find the shortest path between two points on the grid.

DESIGN

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue. What A* Search Algorithm does is that at each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and process that node/cell.
We define '**g**' and '**h**' as simply as possible below
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.

Heuristics

We can calculate **g** but how to calculate h?

We can do things.

A) Either calculate the exact value of h (which is certainly time consuming).
OR
B) Approximate the value of h using some heuristics (less time consuming).
We will discuss both of the methods.

A) **Exact Heuristics –**
We can find exact values of h, but that is generally very time consuming. Below are some of the methods to calculate the exact value of h.
1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm.
2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance.

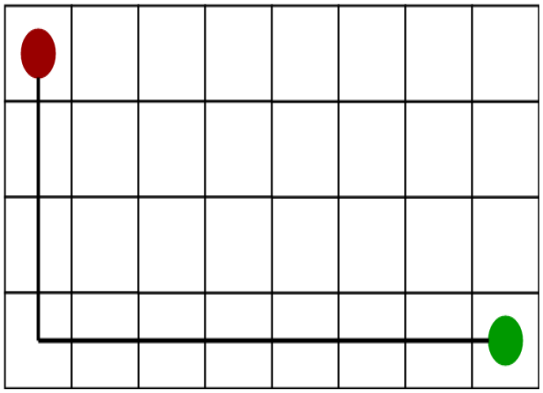
B) Approximation Heuristics –
There are generally three approximation heuristics to calculate h –
1) Manhattan Distance –

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

h = abs (current_cell.x – goal.x) +
abs (current_cell.y – goal.y)

- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).

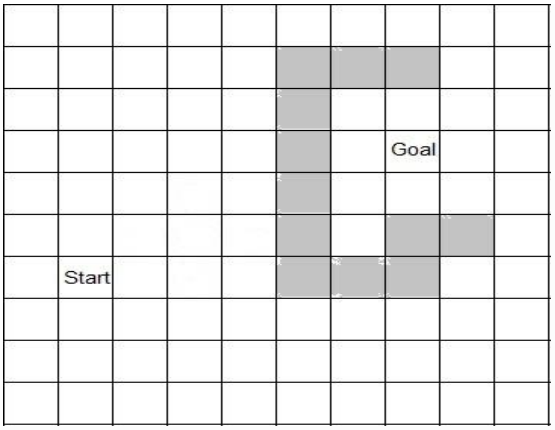


Manhattan distance = |start.x-goal.x| + |start.y-goal.y|

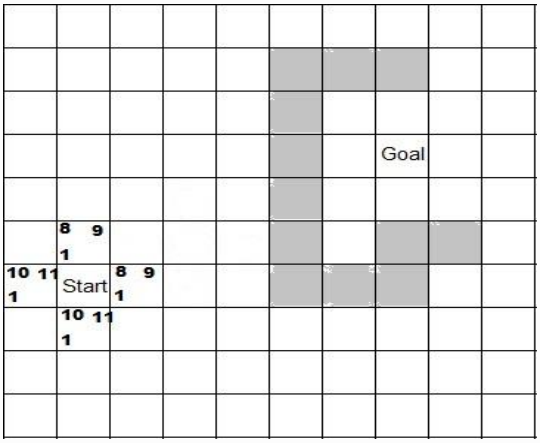
where **start** and **goal** are the two points for which we have to calculate the Manhattan. Thus, for the above figure the Manhattan distance will be 11.

- **Euclidean Heuristics**
- **Octile Heuristics**
- **Chebyshev Heuristics**

So we will understand how pathfinding will work according to the algorithm and using Manhattan heuristics, taking the following instance and watching the progression.

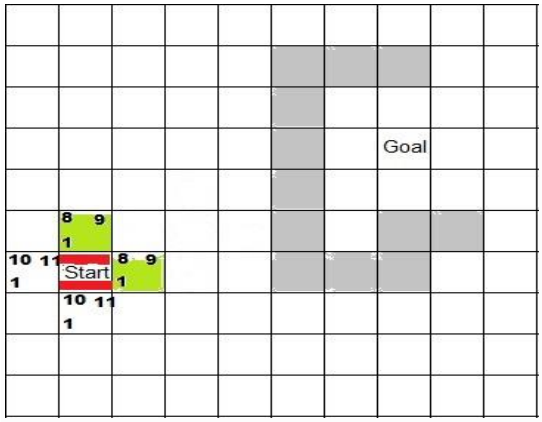


The grey blocks are the obstacles where the object cannot go. So, with our first cycle we get the following



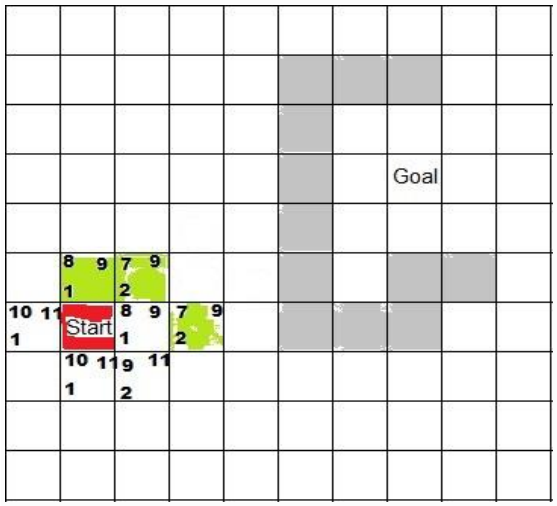
F score = Top right
H score= Top left
G score= Bottom Left.

Now in the next step we have to take the minimum F score according to the algorithm

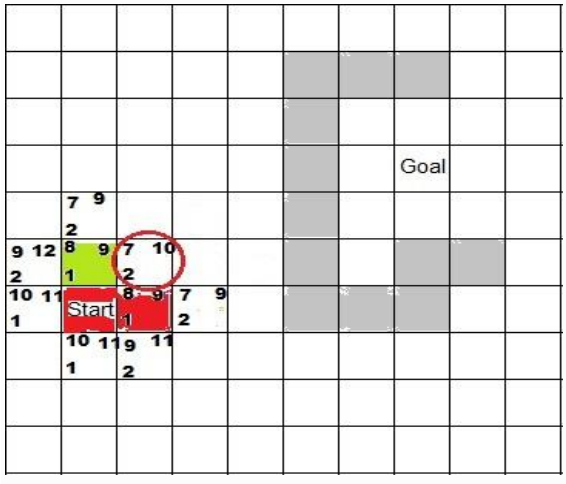


Green block marks the node that can be used for the next iteration. Let's take the tile on the right and move forward with our algorithm. Start is inserted to the closed list and this marked as red.

For the third step we have the following choices



This time let's take the tile on top



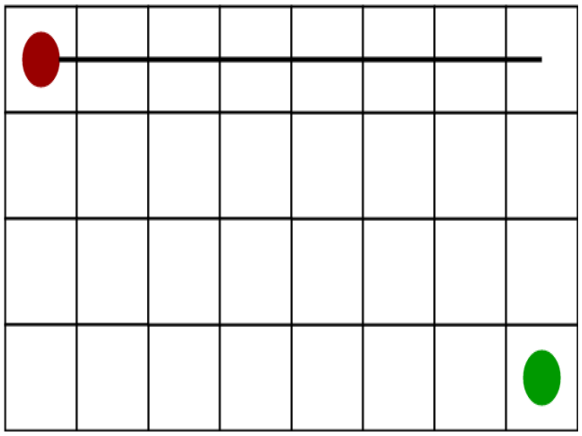
2) Diagonal Distance-

- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

h = max {abs(current_cell.x – goal.x),
abs(current_cell.y – goal.y) }

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



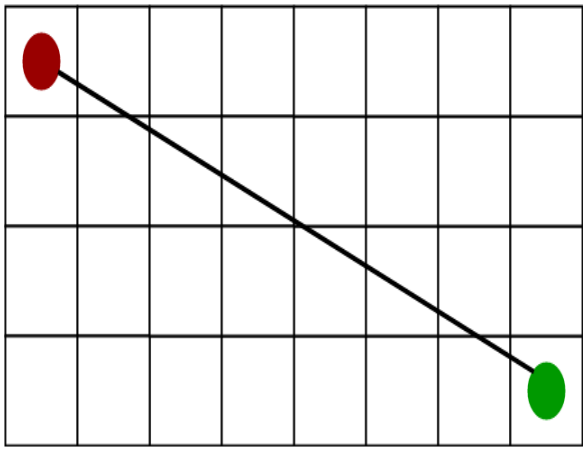
3) Euclidean Distance-

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula.

h = sqrt ((current_cell.x – goal.x)2 +
(current_cell.y – goal.y)2)

- When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



IMPLIMENTATION

This is an extension to the A* in general, and the skeletal algorithm which is common for all implementations.

Introduction

Here we'll be talking about the implementation of the A* algorithm that we have seen so far on pathfinding on a 2D grid. We will be explaining with 4-way movement i.e. the object can move 4 ways, up, down, left and right as it's exactly same as the 8-way movement i.e. 4-way movement plus the diagonal movement.

Working

Let's work on this in the order So, first of all we need to decide what all information is needed to be kept in the node.

- We need the x,y coordinates as column and row information within the nodes.
- We need the individual f,g and h scores to be stored in the nodes.
- We need a pointer that points to the parent.

This is the basic information required to be stored in the node, of course more information can be added as per the requirements of the pathfinding problem.

CODE

```
import pygame
import math
from queue import PriorityQueue

WIDTH = 800
WIN = pygame.display.set_mode((WIDTH, WIDTH))
pygame.display.set_caption("A* Path Finding Algorithm")
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:
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
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

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
def h(p1, p2):
x1, y1 = p1
x2, y2 = p2
```

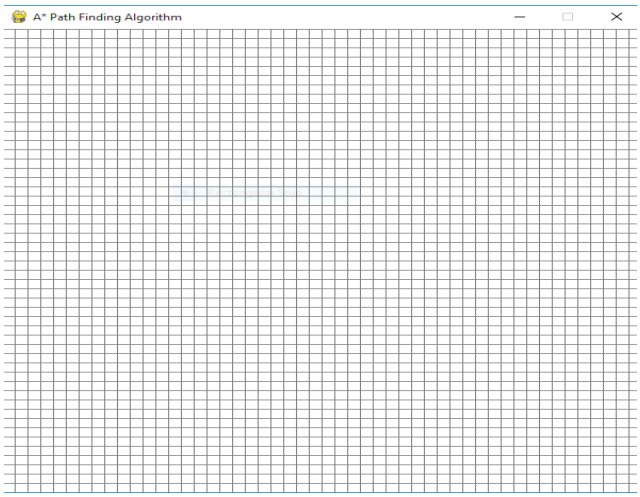
```
return abs(x1 - x2) + abs(y1 - y2)
def reconstruct_path(came_from, current, draw):
while current in came_from:
current = came_from[current]
current.make_path()
draw()

def algorithm(draw, grid, start, end):
count = 0
open_set = PriorityQueue()
open_set.put((0, count, start))
came_from = {}
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())

open_set_hash = {start}

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

current = open_set.get()[2]
open_set_hash.remove(current)

if current == end:
reconstruct_path(came_from, end, draw)
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
open_set.put((f_score[neighbor], count, neighbor))
```

```
open_set_hash.add(neighbor)
neighbor.make_open()
draw()
if current != start:
current.make_closed()
return False
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
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
end = None
run = True
while run:
draw(win, grid, ROWS, width)
```

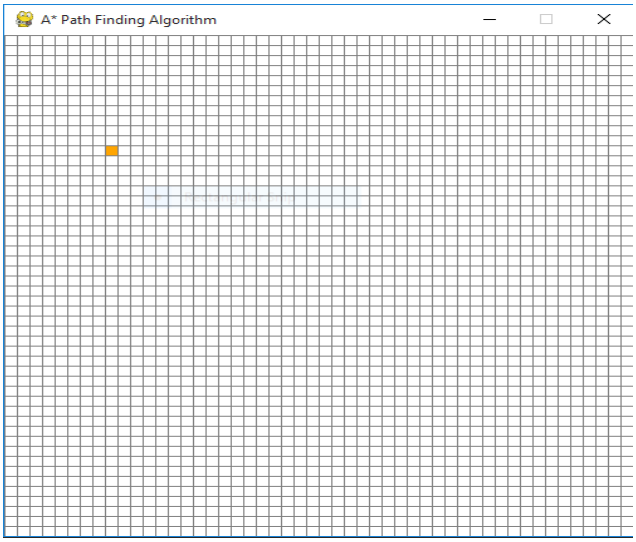
```
for event in pygame.event.get():
if event.type == pygame.QUIT:
run = False
if pygame.mouse.get_pressed()[0]: # LEFT
pos = pygame.mouse.get_pos()
row, col = get_clicked_pos(pos, ROWS, width)
spot = grid[row][col]
if not start and spot != end:
start = spot
start.make_start()
elif not end and spot != start:
end = spot
end.make_end()
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 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).
```

RESULT

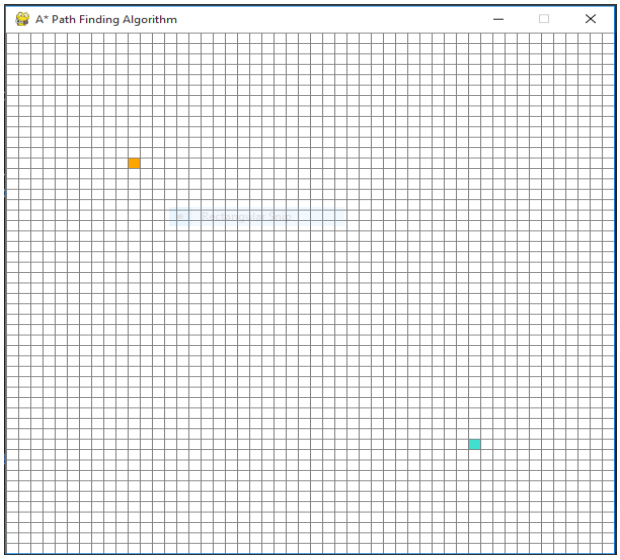
- Initially a 50 by 50 grid (white color) appears .



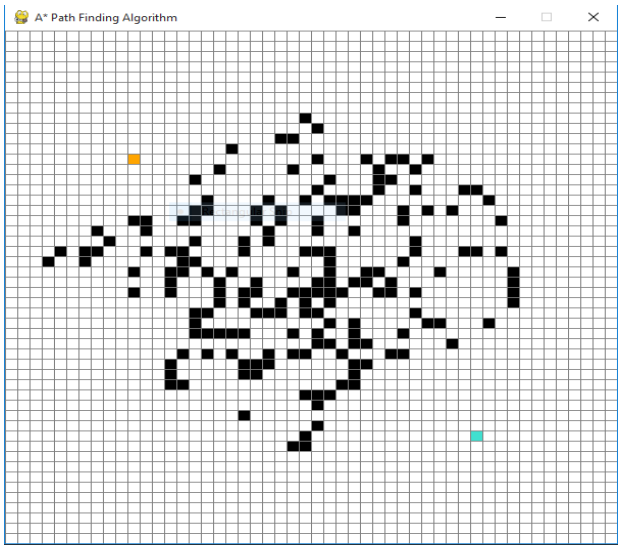
- Source node (Orange color) is marked.



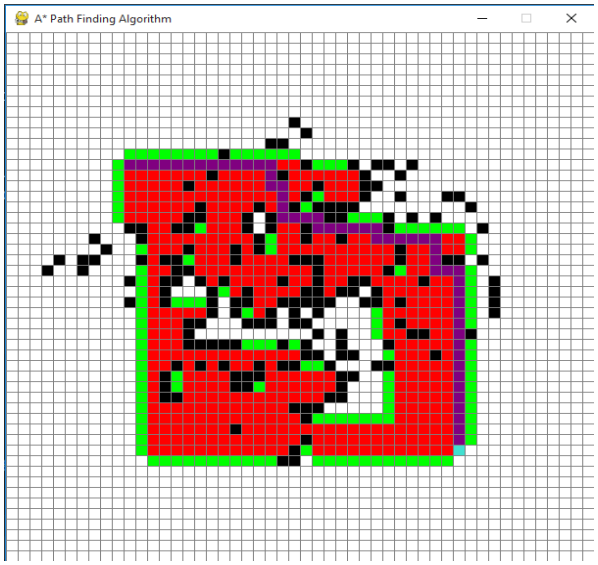
- Destination node (Sky blue color) is marked.



- Barriers (Black color) are created.



- Shortest path(Purple color) from source to destination is found using A* algorithm.



CONCLUSION

By this application it can be helpful in finding out the distance among two points (Source and vacation spot) whether or not it be two towns or states or exclusive locations. In this mini Project, A* search algorithm is implemented to find the shortest path between source and destination. A-star (A*) is a mighty algorithm in Artificial Intelligence with a wide range of usage. However, it is only as good as its heuristic function. A* is the most popular choice for pathfinding because it's reasonably flexible. The basic core of pathfinding algorithm is just a small piece of the puzzle in AI games. The most concern problem is how to use the

algorithm to solve difficult problems. A* algorithm is the most popular algorithm in pathfinding.
The map is converted into three main colours; each map has different correspondence to the three colours, after selecting source and destination points, the system will find the shortest path between the selected points. Other search algorithms can be used and each algorithm has different characteristics.