CHAPTER 1

ABSTRACT

Nowadays, in computer networks, the routing is based on the shortest path problem. This will help in minimizing the overall costs of setting up computer networks. New technologies such as map-related systems are also applying the shortest path problem. This project's main objective is to compare the Dijkstra's Algorithm and A* Algorithm.

Searching is the problem-solving technique in artificial intelligence. There are various search algorithms related to search like Dijkstra, Depth first search, Breadth first search, A*,Hill Climbing, Best first search algorithms and their variants. But most famous algorithms are Dijkstra and A* search algorithms.

Dijkstra algorithm is used to solve the path finding problem. Almost all application is using Dijkstra till now. As technology grows, the speed of vehicles also increased like rocket. That is why a scholar as well as Microsoft, Google like companies starts working on A* algorithm because A* performance is better than Dijkstra.

At present time, even one second has also weightage for Light motor vehicles because of its very high speed. That is why it is necessary to design and develop tools to analysis the performance of Dijkstra, A* and their variants

So the expected result from this mini project is that we can compare the performance of both these two algorithms and determine which is better to find the shortest path for the point robot to pass through and get a visual image of the path that the program is going to undergo with predefined coordinates of the obstacles or the map.

CHAPTER 2

INTRODUCTION

The shortest path problem is a problem of finding the shortest path or route from a starting point to a final destination. Generally, in order to represent the shortest path problem we use graphs. A graph is a mathematical abstract object, which contains sets of vertices and edges. Edges connect pairs of vertices. Along the edges of a graph it is possible to walk by moving from one vertex to other vertices.

Depending on whether or not one can walk along the edges by both sides or by only one side determines if the graph is a directed graph or an undirected graph. In addition, lengths of edges are often called weights, and the weights are normally used for calculating the shortest path from one point to another point. In the real world it is possible to apply the graph theory to different types of scenarios.

For example, in order to represent a map we can use a graph, where vertices represent cities and edges represent routes that connect the cities. If routes are one-way then the graph will be directed; otherwise, it will be undirected. There exist different types of algorithms that solve the shortest path problem.

However, only several of the most popular conventional shortest path algorithms along with one that uses genetic algorithm are going to be discussed

1.Dijkstra's Algorithm

2.A* Algorithm

CHAPTER 3

LITERATURE SURVEY

Kairanbay Magzhan , analysed the Shortest path algorithms like Dijkstra's Algorithm, Floyd-Warshall Algorithm, Bellman-Ford Algorithm, and Genetic Algorithm. Genetic algorithm (GA) provide better result and optimal solution.

Comparative Analysis of Path Finding Algorithms by Shabina Banu Mansuri1, Shiv kumar.

Abhishek Goyal , compared A* and Dijkastra to find shortest path between source and destination based on execution time and proved that A* solves the problem nearly half time lesser then Dijkstra algorithm. Implementation was based on console

Sharwan. Kr. Sharma, B.L. Pal in 2015, "Shortest path Searching for road network using A* Algorithem "they have complete analysis of Dijikstra and A * algo to find the shortest on the basis of obstacle and without obstacle help of bidirectional .The finally show the A* best find path.

In the A comparative study of A-star algorithm for search and rescue in perfect maze, by Daoxing Gong and Xiang Liu they have taken the heuristic function and showed how the math works for the A star search algorithm and then demonstrated The different A star algorithms along with it's heurestics.

CHAPTER 4

SYSTEM REQUIREMENTS

A System Requirements Specification (SRS) (also known as a Software Requirements Specification) is a document or set of documentation that describes the features and behaviour of a system or software application. It includes a variety of elements that attempts to define the intended functionality required by the customer to satisfy their different users.

In addition to specifying how the system should behave, the specification also defines at a high-level the main business processes that will be supported, what simplifying assumptions have been made and what key performance parameters will need to be met by the system.

4.1 SOFTWARE REQUIREMENTS

The software requirements are description of features and functionalities of the target system. Requirements convey the expectations of users from the software product. The requirements can be obvious or hidden, known or unknown, expected or unexpected from client's point of view.

It defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The software requirements in our program are

1. Programming language: Any version of Python.
2. Compiler: It has no compiler it acts as an interpreter
3. Operating System: Ubuntu, Windows or any other distro where python works.
4. Some externally installed python libraries like NUMPY, PYGAME, MATPLOT and the python-tk package to work for matplot library.

4.2 HARDWARE REQUIREMENTS

There are hardware requirements, also known as system requirements, for every OS we are going to use. These requirements include the minimum processor speed, memory, and disk space required to install Windows. In almost all cases, you will want to make sure that your hardware exceeds these requirements to provide adequate performance for the services and applications running on the server. The table below outlines the minimum hardware requirements to execute this project.

1. Processor: Any 3rd Gen+ Intel Processor or even a 1st gen AMD processor.

2. RAM: For the pygame module to run faster 8 gigs of RAM is sufficient but it is going to be very slow more the ram better the speed and lag free run.

3. Graphics: at least 2 if you want to see the pygame module running.

4.3 SHORT DESCRIPTION OF ALL THE LIBRARIES USED

1. **Pygame** is a set of Python modules designed for writing video games. Pygame adds functionality on top of the excellent SDL library. This allows you to create fully featured games and multimedia programs in the python language.

2. **Sys** is a module that provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

3. **NumPy** is the fundamental package needed for scientific computing with python, Besides it's obvious scientific uses, Numpy can also be used as an efficient multi-dimensional container or generic data.

4. **Matplotlib** is a comprehensive library for creating static, animated, and interactive visualizations in Python.

CHAPTER 5

PROBLEM STATEMENT

The project is about implementation of Dijkstra's and algorithm with given obstacles present in the layout. The implementation of the algorithm in python code will help us create a graphical representation of the movement of an object from one point to another which is fed by providing with the co-ordinates of the start and the end points.

5.1 Dijkstra's Algorithm

Explanation and Implementation For each vertex within a graph we assign a label that determines the minimal length from the starting point s to other vertices v of the graph. In a computer we can do it by declaring an array d[]. The algorithm works sequentially, and in each step it tries to decrease the value of the label of the vertices.

The algorithm stops when all vertices have been visited. The label at the starting point s is equal to zero (d[s]=0); however, labels in other vertices v are equal to infinity (d[v]=∞), which means that the length from the starting point s to other vertices is unknown. In a computer we can just use a very big number in order to represent infinity.

In addition, for each vertex v we have to identify whether it has been visited or not. In order to do that, we declare an array of Boolean type called u[v], where initially, all vertices are assigned as unvisited (u[v] = false). The Dijkstra's algorithm consists of n iterations.

If all vertices have been visited, then the algorithm finishes; otherwise, from the list of unvisited vertices we have to choose the vertex which has the minimum (smallest) value at its label (At the beginning, we will choose a starting point s). After that, we will consider all neighbours of this vertex (Neighbours of a vertex are those vertices that have common edges with the initial vertex).

For each unvisited neighbour we will consider a new length, which is equal to the sum of the label's value at the initial vertex v (d[v]) and the length of edge l that connects them. If the resulting value is less than the value at the label, then we have to change the value in that label with the newly obtained value.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path. A minimum priority can be used to efficiently receive the vertex with least path distance. The pseudo code for it is as shown below.

```
function dijkstra(G, S)
        for each vertex V in G
                distance[V] <- infinite
                previous[V] <- NULL
                 If V != S, add V to Priority Queue Q
                distance[S] <- 0
         while Q IS NOT EMPTY
                U <- Extract MIN from Q
                 for each unvisited neighbour V of U
        t                empDistance <- distance[U] + edge_weight(U, V)
                         if tempDistance < distance[V]
                         distance[V] <- tempDistance
                          previous[V] <- U
        return distance[], previous[]
```

5.2 A* Algorithm

A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes where *n* is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from *n* to the goal.

A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

. This idea has been implemented using greedy method technique.In our case the heuristic that we have used is the Euclidean Distance which is given by

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

Dijkstra is a special case of A* Search Algorithm, where h = 0 for all nodes.

CHAPTER 6

SYSTEM DESIGN

The algorithm's have been implemented using python code with the libraries: pygame,.math,sys ,numpy and matplotlib(dicription is given above.

For the implementation of obstacles in the graph we have used the above mentioned additional library files for the calculation of travel and for the representation of the travel of the object from source to destination.

Basic design of our project

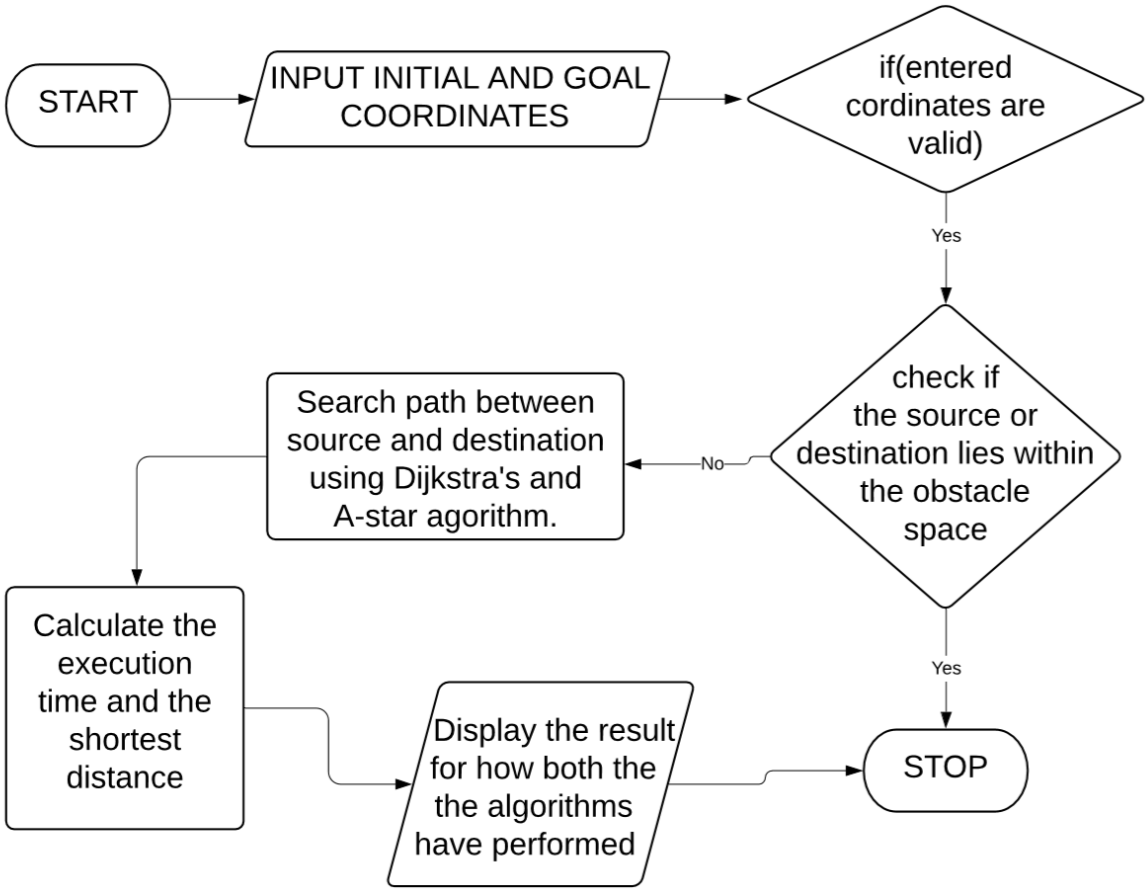

Fig.6.1 Flow chart of the working of the Project

CHAPTER 7

IMPLEMENTATION

At first the user enters the initial coordinates and the final coordinates the resolution of this obstacle map is 230*144. The algorithms and all come into picture only when these initial and the final coordinates are taken.

Code:

```
print("Enter initial node cordinates")
xi=float(input("x =  "))
yi=float(input("y =  "))
init_node=[xi,yi]
print("Enter goal node cordinates (max 230x144 )")
xg=float(input("x =  "))
yg=float(input("y =  "))
goal=[xg,yg]
r=int(input("Enter Resolution (must be an integer value) =  "))
sd=0;
```

On entering these coordinates the program checks if the entered coordinate is within the area of the obstacle map or basically to see if the entered points are valid or not.It is done as shown below.

```
if(obstacle_space(goal[0],goal[1],r)==1 or obstacle_space(init_node[0],init_node[1],r)):
        sys.exit("Either goal node or start node lies inside obstacle ")
if (init_node[0] not in range(0,231) or goal[0] not in range(0,231) or init_node[1] not in range(0,145) or goal[1] not in range(0,145)):
        sys.exit("Entered node cordinates are not integers or outside the workspace or invalid resolution")
```

Now once these coordinates are taken our algorithm starts beginning to search for the destination in the obstacle map shown below.

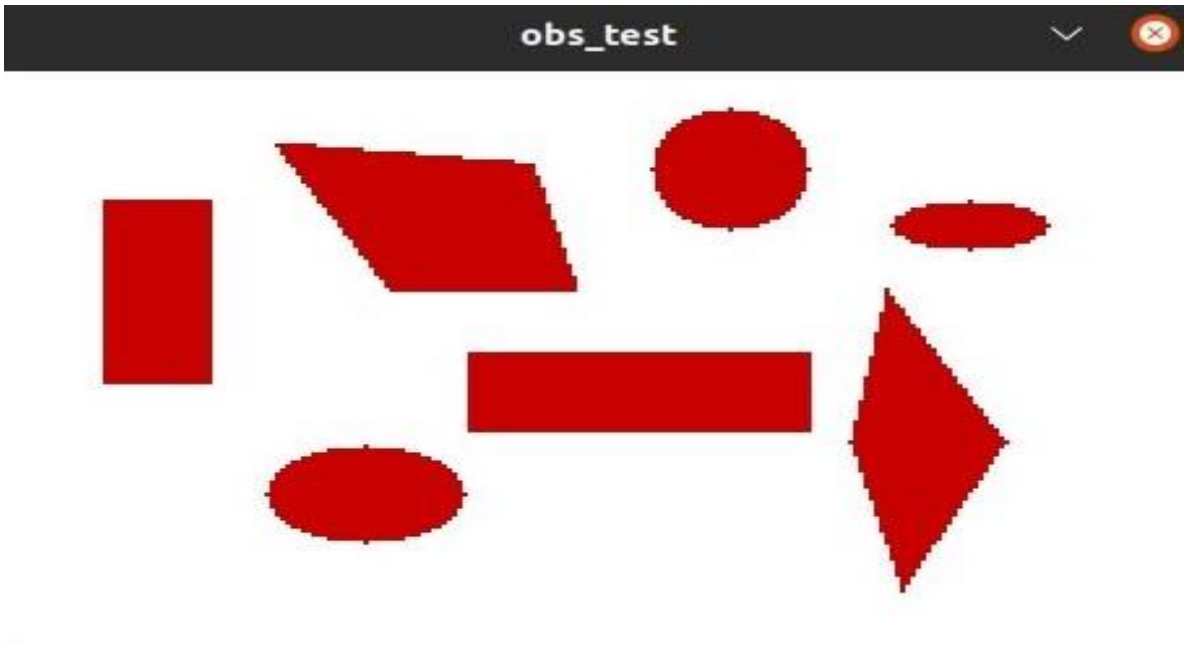

Fig7.1 The designed obstacle map.

The pseudo code for how this algorithm's are going to scan for the destination and move are shown below.

```
def obstacle_space(x,y,r):
    c = 0
    #big circle
    if ((x-math.ceil(140/r))**2+math.ceil(y-(120/r))**2-math.ceil(15/r)**2)<=0:
        c=1
    #rhombus
    if (38*x- 7*y - 5830/r >= 0) and (38*x + 23*y - 8530/r <= 0) and (37*x -20*y -6101/r <= 0) and (37*x +10*y - 6551/r >= 0):
            c=1
    ##parallelogram
    if (2*x + 19*y - 2514/r <= 0) and (41*x+ 25*y -5300/r >= 0) and (y - 90/r>= 0) and (37*x +10*y - 5000/r <= 0):
        c=1
    #rect1
    if (x-math.floor(20/r) >= 0) and (x - math.floor(40/r) <= 0) and (y - math.floor(67.5/r) >= 0) and (y - math.floor(112.5/r) <= 0):
        c=1
```

```
    #rect2
    if (x-math.floor(90/r) >= 0) and (x - math.floor(155/r) <= 0) and (y - math.floor(55/r) >= 0) and (y - math.floor(74/r) <= 0):
        c=1
    #ovaltop
    if ((x-math.ceil(186/r))/math.ceil(15/r))**2 + ((y - math.ceil(106/r))/math.ceil(6/r))**2 - 1 <=0:
        c=1
    #oval bottom
    if ((x-math.ceil(70/r))/math.ceil(19/r))**2 + ((y - math.ceil(39/r))/math.ceil(12/r))**2 - 1 <=0:
        c=1
    return c
```

This is how the obstacles have been defined in our project, the shapes have also been mentioned , now for the movement in the dijkkstra's algorithm it is as shown below to move up i.e in the north or the upward direction.

```
    #goes up
    nd,cost=up(ndx)
    if (nd[1]>=0 and obstacle_space(nd[0],nd[1],r)!=1):
        if nd not in visited_node:
            xl=range(0,len(c_nd))
            xl=xl[::-1]
            check=0
            for cku in xl:
                if(nd == c_nd[cku]):
                    check=1
                    if(cst[cku]>=(cst[x]+cost)):
                        p_nd[cku]=ndx
                        cst[cku]=round((cst[x]+cost),1)
                        break
            if (check!=1):
                p_nd.append(ndx)
```

```
            c_nd.append(nd)
            cst.append(round((cost+cst[x]),1))
def up(current_node):
    next_node=[0,0]
    next_node[0]=current_node[0]
    next_node[1]=current_node[1]-1
    cost=1

    return next_node,cost
```

This is just for the South direction now similarly we do it for all the directions just by changing the x and y values, above since we move in the south direction the following operation is done #N --> North (i, j+1)

Similarly for other directions:-

#W --> West (i-1, j)

#E --> East (i+1, j)

#N --> North (i, j+1)

#S --> South (i, j-1)

#N.W--> North-West (i-1, j+1)

#S.W--> South-West (i-1, j-1)

#N.E--> North-East (i+1, j+1)

#S.E--> South-East (i+1, j-1) is done

This was the scanning and the movement's pseudo code for the dijkstra's algorithm for A* it is going to be almost same but we are just going to add the heuristic function that is the Euclidean distance. The pseudo code is as shown below.

```
def astar(node):
   h = math.sqrt ( (node[0] - goal[0])**2 +  (node[1] - goal[1])**2 )
   return h
```

Now this is the formula to calculate the distance and limit the direction in a particular direction only so that it doesn't waste time like Dijkstra's algorithm to move around in every direction but to just move in the direction of the destination. The same pseudo code mor movement in the north direction is given below.

```
   #North
   nd,cost=up(ndx)
   if (nd[1]>=0 and obstacle_space(nd[0],nd[1],r)!=1):
      if nd not in visited_node:
```

```
        xl=range(0,len(c_nd))
        xl=xl[::-1]
        check=0
        for cku in xl:
            if(nd == c_nd[cku]):
                check=1
                if(cst[cku]>=(cst[x]+cost)):
                    p_nd[cku]=ndx
                    cst[cku]=round((cst[x]+cost),1)
                    break

        if (check!=1):
            p_nd.append(ndx)
            c_nd.append(nd)
            cst.append(round((cost+cst[x]),1))
            h_nd.append(round((cost+cst[x]+astar(nd)),2))
```

So now that we have completed the scanning we need to store everything and find the shortest path from the source to destination , so to this the pseudo code is as shown below.

```
vp_nd.append(p_nd.pop(x))
    visited_node.append(c_nd.pop(x))
    v_cst.append(cst.pop(x))



    if(visited_node[-1]==goal):
        flag=1

    if(flag!=1):
        x=cst.index(min(cst))
        ndx=c_nd[x][:]
```

```
#to find the shortest path and store in the seq numpyarray
seq=[]
seq.append(visited_node[-1])
seq.append(vp_nd[-1])
x=vp_nd[-1]
i=1
while(x!=init_node):
    if(visited_node[-i]==x):
        seq.append(vp_nd[-i])
        x=vp_nd[-i]
        sd+=1
    i=i+1
# In[For the obstacle space ]:
obs_space = []
for i in range(0,231):
    for j in range(0,145):
        q=obstacle_space(i,j,r)
        if q == 1:
          obs_space.append([i,j])
k=2
my_list = np.array(visited_node)
visited_node=my_list*k*r
my_list1 = np.array(seq)
seq=my_list1*k*r
my_list2 = np.array(obs_space)
obs_space = my_list2*k*r
```

So now that we have everything ready and all the computations have been done now all that remains is to take the output of all these and show them visually, so this is where the pygame module comes into the picture, so, the pseudo code for getting the visual representation is as shown below.

```
#defining the colours
Black = [0, 0, 0]          // to show the scanned area
grey = [192, 192, 192]          // to print the shortest path
Blue = [0, 100, 255]
White = [255, 255, 255]         //The background
red= [200, 0, 0]          // colour used to print the obstacle
#length and breadth
SIZE = [230*k+r+r, 144*k+r+r]
screen = pygame.display.set_mode(SIZE)

pygame.display.set_caption("Dijkstra's algorithm")
clock = pygame.time.Clock()
done = False
while not done:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
     screen.fill(White)
#print the obstacles
    for i in obs_space:
        pygame.draw.rect(screen, red, [i[0],144*k-i[1],r*k,r*k])
    pygame.display.flip()
    clock.tick(20)
#print the visited nodes(shows how everything is getting scanned
    for i in visited_node:
        pygame.time.wait(1)
        pygame.draw.rect(screen, Black, [i[0],144*k-i[1],r*k,r*k])
        pygame.display.flip()
#print the path
    for j in seq[::-1]:
        pygame.time.wait(1)
```

```
        pygame.draw.rect(screen, grey, [j[0], 144*k-j[1], r*k,r*k])
        pygame.display.flip()
    pygame.display.flip()

    pygame.time.wait(8000)
    done = True
pygame.quit()
```

Both the algorithms have now been executed and now all we need to do is to take these 2 algorithms and then compare then side by side so that we can get the time taken by both the algorithms and then represent them using a bar graph, so this is where we have used the matplot library and the pseudo code for that as well is shown below.

```
import matplotlib.pyplot as plt
from dijkstr import tt              // Import's total time taken by Dijkstra's Algorithm
dtt=tt
from astr import tt                 // Import's total time taken by A-star algorithm.
att=tt
import numpy as np
alg=['DIJKSTRA','A*']
tt =[dtt, att]
ypos = np.arange(len(alg))
plt.xticks(ypos,alg)
plt.ylabel("Time(s)")
plt.xlabel("Dijkstra vs A*")
plt.title("DAA TEAM 5")
l=plt.bar(ypos,tt)
l[0].set_color('r')
plt.legend()                // Display the bargraph
plt.show()
```

So this is all about the implementation done in our project.

CHAPTER 8

TESTING

The following condition is used to enter size of the object -

(resolution > (gcd (x of origin coordinates, y of origin coordinates, x of goal coordinates, y of goal coordinates) && (input coordinates % resolution == 0))

8.1 TEST CASES

Test Case 1

Goal - Movement from origin (0,0) to farthest point of the graph

Inputs

1. Start Node - (0,0).
2. Goal Node - (230,144).
3. Resolution - 2

Expected Outcome - Visualization of scanning for obstacles and goal node and then presenting the shortest path.

Test Result – Success

Test Case 2

Goal - Movement from origin (0,0) to randomly chosen point that is not present on obstacle.

Inputs

1. Start Node - (0,0).
2. Goal Node - (24,98).
3. Resolution - 2

Expected Outcome - Visualization of scanning for obstacles and goal node and then presenting the shortest path.

Test Result – Success.

Test Case 3

Goal – Movement from origin to goal node outside of the map area boundary

Inputs

1. Start Node –(0,0)
2. Goal Node – (250,210)
3. Resolution – 2

Expected Result – Display appropriate error message.

Test Result – Success.

Test Case 4

Goal- Movement from origin to random point that is present on obstacle

Inputs

1. Start Node - (0,0)
2. Goal Node - (144,32)
3. Resolution – 2

Expected Outcome - Display appropriate error message

Test Result – Success

Test Case 5

Goal- Movement from origin to random point that is not present on obstacle but conditions for size of object is not followed.

Inputs

1. Start Node - (0,0)
2. Goal Node - (144,32)
3. Resolution – 18

Expected Outcome - Display appropriate error message of invalid resolution

Test Result – Success

Test Case 6

Goal- Movement from randomly chosen point to farthest point in the graph.

Inputs

1. Start Node - (24,112)
2. Goal Node - (230,144)
3. Resolution – 2

Expected Outcome - Visualization of scanning for obstacles and goal node and then presenting the shortest path.

Test Result – Success

Test Case 7

Goal- Movement from randomly chosen point to randomly chosen point in the graph.

Inputs

1. Start Node - (24,112)
2. Goal Node - (110,44)
3. Resolution – 2

Expected Outcome - Visualization of scanning for obstacles and goal node and then presenting the shortest path.

Test Result – Success.

Test Case 8

Goal - Movement from randomly chosen to goal node outside of the map area boundary

Inputs

1. Start Node – (24,112)
2. Goal Node – (340,110)
3. Resolution – 2

Expected Outcome - Display appropriate error message

Test Result – Success

Test Case 9

Goal- Movement from randomly chosen point to randomly chosen point that is present on obstacle.

Inputs

1. Start Node - (32,24)
2. Goal Node - (24,98)
3. Resolution – 2

Expected Outcome - Display appropriate error message

Test Result – Success

Test Case 10

Goal- Movement from randomly chosen point to randomly chosen point but conditions for size of object is not followed.

Inputs

1. Start Node - (32,24)
2. Goal Node - (24,98)
3. Resolution – 3

Expected Outcome - Display appropriate error message

Test Result – Success

So, as far as we have tested , we have got all outputs as we wanted and it has succeeded in every test case that we have taken.

CHAPTER 9

RESULTS

9.1 ACCOMPLISHMENTS

We have successfully implemented both the algorithms and we have proved the claim that A Star is faster than Dijkstra's algorithm by showing the results visually and then plotting the bar graph and show how much time both the algorithms are taking individually, we have also learnt more in depth about the python programming language as newbies to an intermediary level and what all can be done, what we also learned is python is slow , when the same was implemented in c++ it ran like 2 times faster but this is much easier to code and get the visual representation and everything as and how we wanted.

9.2 SNAPSHOTS

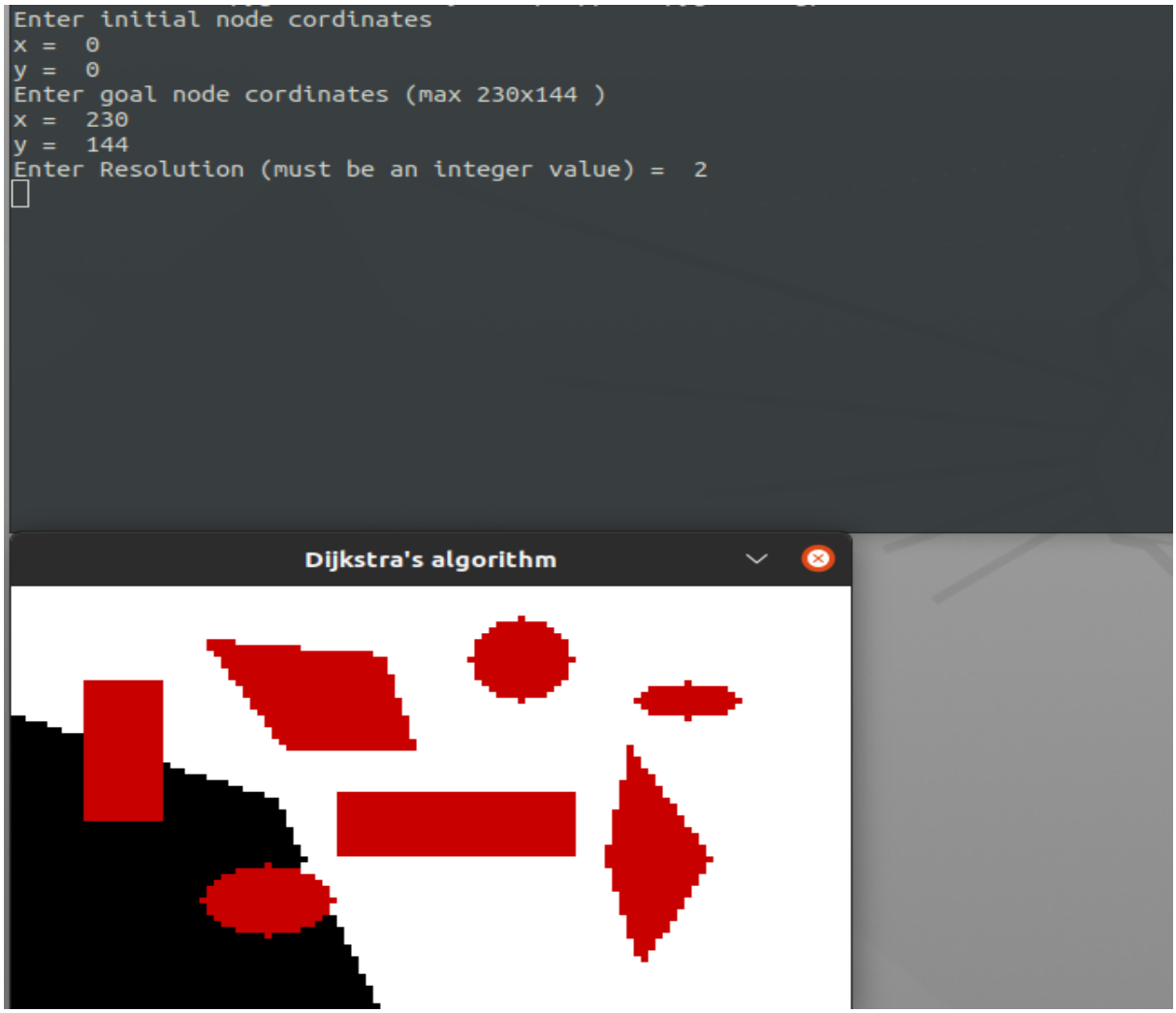

Fig 9.1 Dijkstra's algorithm scanning in progress after taking the inputs

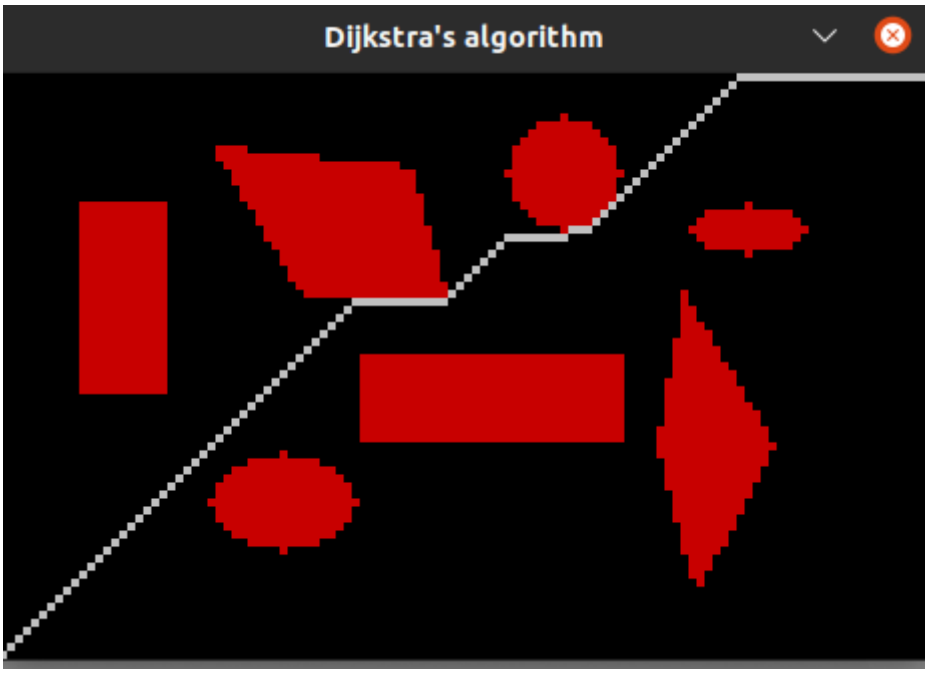

Fig 9.2 The output of the Dijkstra’s algorithm showing the shortest path marked in grey colour.

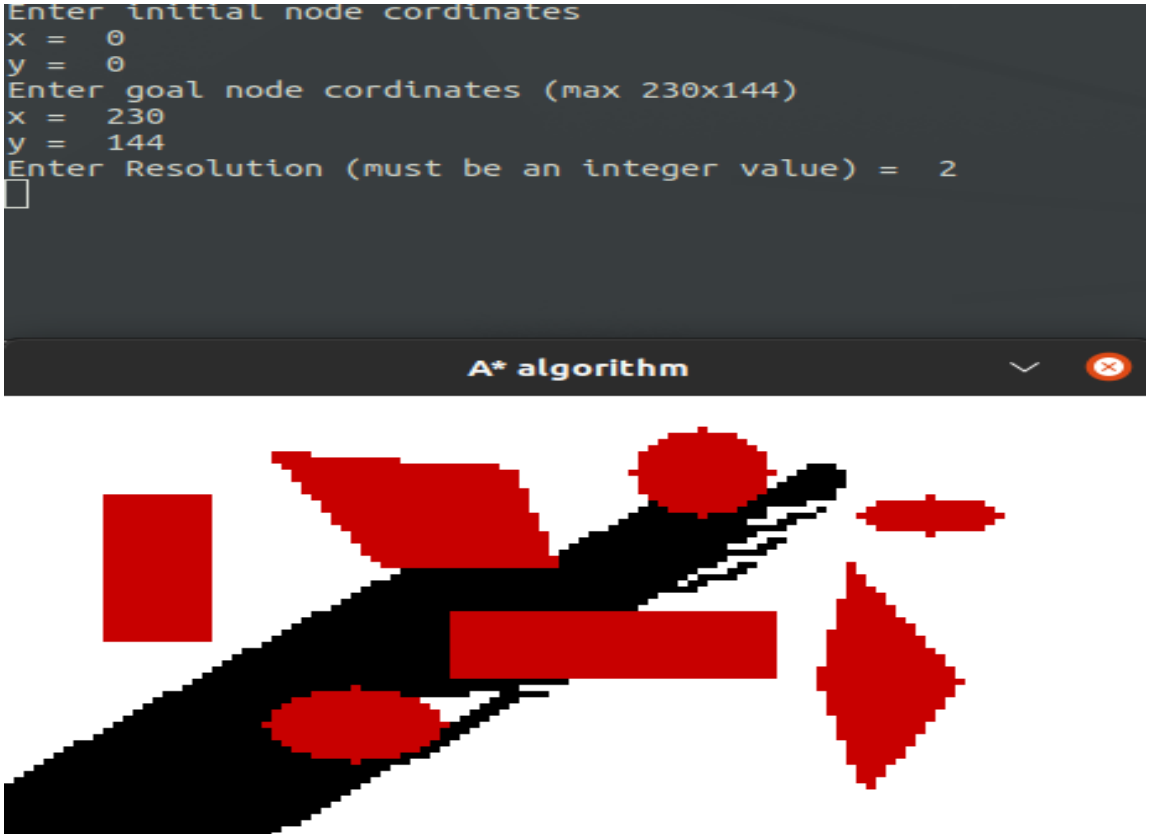

Fig 9.3 A-Star algorithm scanning in progress after taking the inputs.

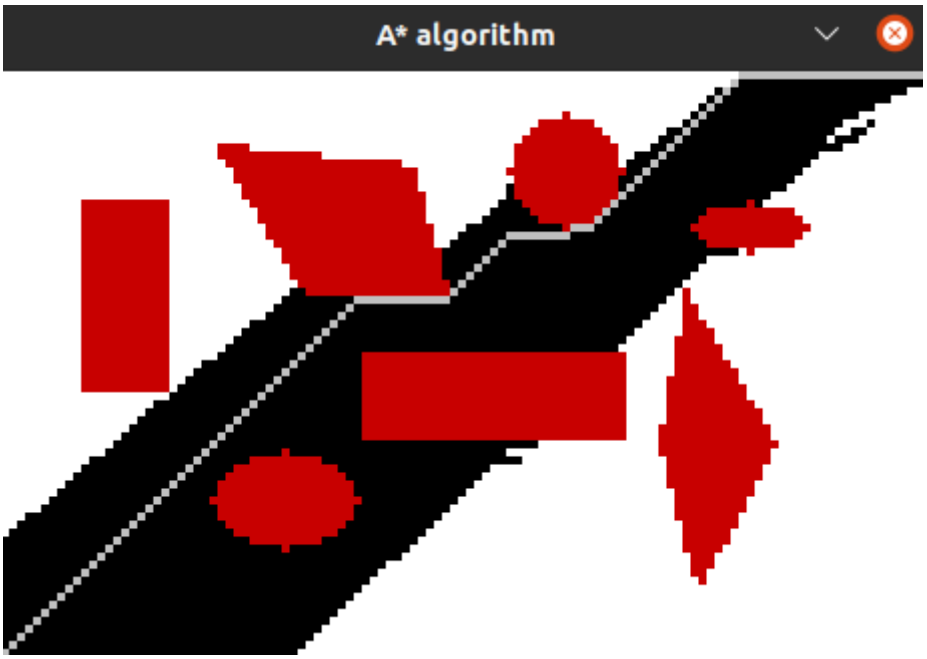

Fig 9.4 The output of the A-star algorithm showing the shortest path marked in grey colour.

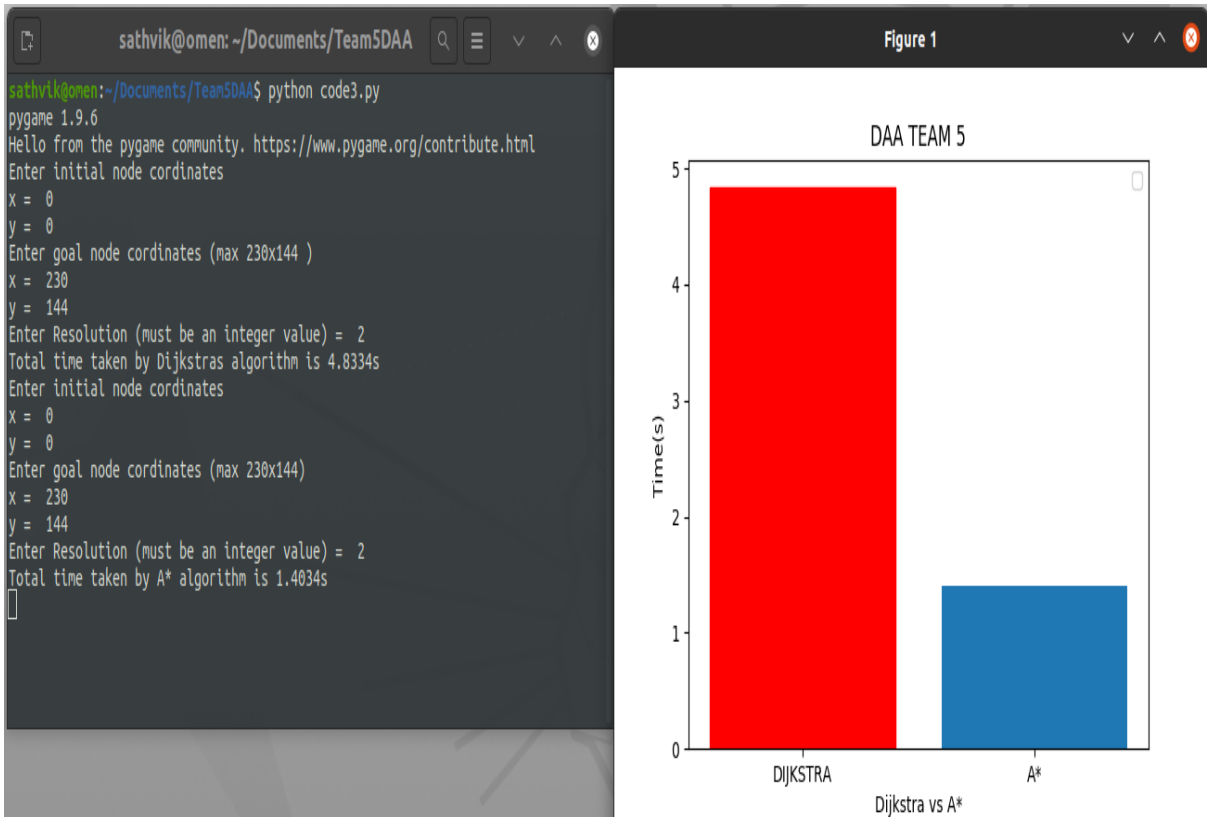

Fig 9.5 The final Output when we get where both the algorithms have been compared against the total time taken by both the algorithms using a single program.

9.3 LIMITATIONS AND ADVANTAGES

1. Search algorithms are mainly used in tower defence games where a character has to move from one point to another point using these search algorithms itself.
2. Coming to the theoretical limitations of the dijkstra's algorithm, it doesn't work with negative edges but since we don't need any negative edges it solves our purpose very well to overcome this disadvantage we actually use the Bellman Ford algorithm where it works with negative edges as well.
3. If were to deal with unweighted graphs then A* algorithm can be used but one limitation it was when we are using graphs is that it always doesn't produce the shortest path.
4. Also google maps uses such search algorithms to find the shortest path from the source to the destination.
5. One main limitation would be with the entered resolution as mentioned before it should be lesser than gcd of the coordinates entered and the mod of all the coordinates should be zero or else we will end up with pointed pixels.

9.4 CONCLUSION AND FUTURE ENHANCEMENT

As we have seen in the above snapshot in the figure 9.5, the total time taken by Dijkstra's algorithm is much slower than that of the A star search algorithm so we can tell that A* algorithm can be more preferred over the Dijkstra's algorithm if time is what we need and we know that time is precious so based on the graph and multiple test cases the above conclusions are drawn.

Coming to the future enhancements, in our project the obstacles are fixed, now what if the obstacles where moving too or there is some blockage in the path something like a road block, so that is what we would like to work on in the future, that is take the obstacle and predict it's next movement and then recalculate the next available shortest path using a technique called as path slicing (so this is just some idea that we got after doing some research).

The github repository link to the complete project is-

https://github.com/sathvikng/Comparison-of-Dijkstra-and-A-star-Algorithm

REFERENCES

1. A comparative study of A-star algorithm for search and rescue in perfect maze, by Daoxing Gong and Xiang Liu
2. Dijkstra's algorithm revisited:the dynamic programming connexion by Moshe Sniedovich [612-614]
3. A Review And Evaluations Of Shortest Path Algorithms byKairanbay Magzhan, Hajar Mat Jani
4. Best Routes Selection Using Dijkstra And Floyd-Warshall Algorithm
5. Comparative Analysis of Path Finding Algorithms ,Shabina Banu Mansuri, Shiv kumar.
6. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/https://stackoverflow.com/questions/982388/how-to-implement-a-linked-list-in-c
7. https://www.geeksforgeeks.org/a-search-algorithm/
8. http://cs231n.github.io/python-numpy-tutorial/
9. https://python101.pythonlibrary.org/chapter20_sys.html
10. https://www.pygame.org/docs/ref/display.html
11. http://aima.cs.berkeley.edu/python/search.py