Rapidly Growing Random Trees to Find the path out of Maze without collision

Shama Zabeen Shaik
University of North Carolina Charlotte
Email: sshaik3@uncc.edu

Abstract -- A Rapidly- exploring Random Tree (RRT) is a data structure and algorithm that is designed for effectively searching non-convex high-dimensional spaces. RRTs are connected incrementally in a way that quickly reduces the expected distance of a randomly-chosen point to the tree. RRTs are particularly suited for path planning problems that involve obstacles and differential constraints like non-holonomic or Kino-dynamic constraints. RRTs can be considered as a technique for generating open-loop trajectories for nonlinear systems with state constraints. An RRT can be intuitively considered as a Monte-Carlo way of biasing search into largest Voronoi regions. Some variations can be considered as stochastic fractals. Usually, an RRT alone is insufficient to solve a planning problem. Thus, it can be considered as a component that can be incorporated into the development of a variety of different planning algorithms.

I. Introduction

In this project, I have made use of one of the path planning algorithms namely, Rapidly Growing Random Trees also called as RRTs. An RRT is an algorithm designed to efficiently search non-convex, high-dimensional spaces by randomly building a space-filling tree which is constructed incrementally from samples drawn randomly from search space and are biased to grow towards the large unsearched areas of the configuration space. The RRTs can also be considered as a technique to generate open-loop trajectories for non-linear systems with state constraints.

Considering the time constraint, I have implemented to basic level of RRT which involves a 2-D work space. I have considered a point robot whose position at the start and the goal state could be defined by the user. The robot is free to start its path at any point inside the boundaries that are pre-defined. Once the start and the goal states are set, the user can see all the possible ways the robot can move with the help of green tree branch like structures that grow continuously till the final goal point is reached. In order to minimize the work on the user's path of which the shortest path is, this algorithm I have implemented, highlights the shortest path in red.

II. Problem Statement

To implement a path planning algorithm that would help the user to find a path from the start configuration to the goal configuration without getting stuck at any point in the environment and not colliding with any obstacles or the boundaries of the obstacles. The algorithm is to be implemented with minimum time and space complexity with a detailed note on what tools and technologies are being used.

III. BACKGROUND

A Rapidly- exploring Random Tree (RRT) is a data structure and algorithm that is designed for effectively searching non-convex high-dimensional spaces. RRTs are connected incrementally in a way that quickly reduces the expected distance of a randomly-chosen point to the tree. RRTs are particularly suited for path planning problems that involve obstacles and differential constraints like non-holonomic or Kino-dynamic constraints. RRTs can be considered as a technique for generating open-loop trajectories for nonlinear systems with state constraints. An RRT can be intuitively considered as a Monte-Carlo way of biasing search into largest Voronoi regions. Some variations can be considered as stochastic fractals. Usually, an RRT alone is insufficient to solve a planning problem. Thus, it can be considered as a component that can be incorporated into the development of a variety of different planning algorithms.

The RRT quickly expands in a few directions to quickly explore the four corners of the workspace. Although the construction method is simple, it is no easy task to find a method that yields such desirable behavior. Consider, for example, a naive random tree that is constructed incrementally by selecting a vertex at random, an input at random, and then applying the input to generate a new vertex. Although one might intuitively expect the tree to ``randomly" explore the space, there is actually a very strong bias toward places already explored.

A random walk also suffers from a bias toward places already visited. An RRT works in the opposite manner by being biased toward places not yet visited. This can be seen by considering the Voronoi diagram of the RRT vertices. Larger Voronoi regions occur on the ``frontier" of the tree. Since vertex selection is based on nearest neighbors, this implies that vertices with large Voronoi regions are more likely to be selected for expansion. On average, an RRT is constructed by iteratively breaking large Voronoi regions into smaller ones.

IV. C-Space and the RRT Algorithm Brief description

For this Project, I have made use of a 2-D configuration space and a point holonomic robot that is free to move in all directions possible.

Considering an RRT that starts at q_{init} and having N vertices uses:

Pseudo code:

```
BUILD_RRT ($q_{init,}$ N, Delta(q))
  1. G.init ($q_{init}$);
  2. For N= 1 to N
  3. $q_{rand} \leftarrow$ RAND_CONF();
  4. $q_{near} \leftarrow$ Nearest_VERTEX ($q_{rand}$, G);
  5. $q_{new} \leftarrow$ NEW_CONF($q_{near}$, Delta(q));
  6. G.add_vertex($q_{new}$);
  7. G.add_edge($q_{near,}q_{new}$);
  8. Return G

NEAREST_VERTEX(q,G)
  1. D $\leftarrow \infty$
  2. for each v $\in$ V
  3. if p(q,v) $\leftarrow$d then
  4. $v_{new}$ = v; d $\leftarrow$ p(q,v);
  5. Return q;
```

V. Robot State and Type and configurations

I have considered a discrete space depicting the shape of a simpler maze. The robot as specified, is a holonomic point robot. Since the environment is a maze, every boundary of the maze is considered an obstacle and whenever an obstacle is reached, the robot traces back its path and updates the path again towards the goal. I have implement collision avoidance myself which ensured that the body is stable and undamaged during the process. This has been implemented in the python environment.

VI. EXPLINATION OF THE ALGORITHM IMPLEMENTED

Python Libraries used: Methods from the following libraries have been called during implementation

1. **Math –** Provides access to the mathematical functions defined by the C standard.
2. **Sys –** System Specific parameters and Functions. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
3. **Random –** Generate pseudo-random numbers. This module implements pseudo-random number generators for various distributions.
4. **Pygame –** This library of python is developed for gaming and animations.

VII. Functions Defined

The following functions are defined in the implementation of the RRT algorithm

There is only one class namely Node () is defined. The class Node () has the following functions:

1. **– name—:**
   This function when called, calls the main function which is main()

2. **main() :**
   This function starts with global count which is a function to increment the number of nodes each time new node is being added to the tree. The main function defines the initial and the goal node positions to be set and sets the current state to init. The init in turn calls the init() function defined. There is a while loop that has been executed which checks for the current and the goal state being set or not, if both the start and the goal states are set, this phase tests if the current state is in build tree, if it is, then the pygame library is called to perform RRT that has been defined.
   While implementing the RRT, this algorithm is checking for the distance between the midpoint of the current point and the point to be traversed to. If there is any collision, then it is detecting that this path has a collision in build tree stage and would wait until the goal state has been reached. If there is a collision occurring, the tree would grow in the other possible directions but not go in the same path beyond the point of collision.

If the tree node reaches the final point to be reached, it would print that the goal is reached and end the growth of the tree.

The selecting of the start and the goal points is enables with the help of mousebuttondown. And the instances are recorded whenever the mouseclickup is observed.

3. **Reset():**

   This function keeps track of the global count of the number of nodes traversed, and sets the count of the nodes to zero at the beginning of eat new RRT execution and whenever the number of nodes exceed 5000, which is defined as the maximum number of iterations possible, it ends the environment and resents the nodes traversed to be zero after displaying maximum limit reached.

4. **init_obstacles():**

   In this function, All the obstacles are being defined in the order of (left, top, width, height).

5. **Get_random_clear():**

   Checks if there are any obstacles and if there are any collisions occurring. If there are no collisions in the collides function, then it says there are no collisions and prints the point

6. **Collides(p):**

   Checks if the point is colliding with the obstacle or not with the rectangular obstacle. The special case handled here is when the random sample generated in not in the collision boundary, the new random sample is generated by calculating the midpoint of the previously generated random sample and the new sample generated.

7. **Point_circle_collision():**

   Checks if the distance between the two points is less than the radius or not. If it is, then it returns collision point. If not, it returns no collision. When the start and the goal point is the same, the room mean square distance of these two points is calculated to update the new point to be selected as the intermediate node between the start and the final goal points.

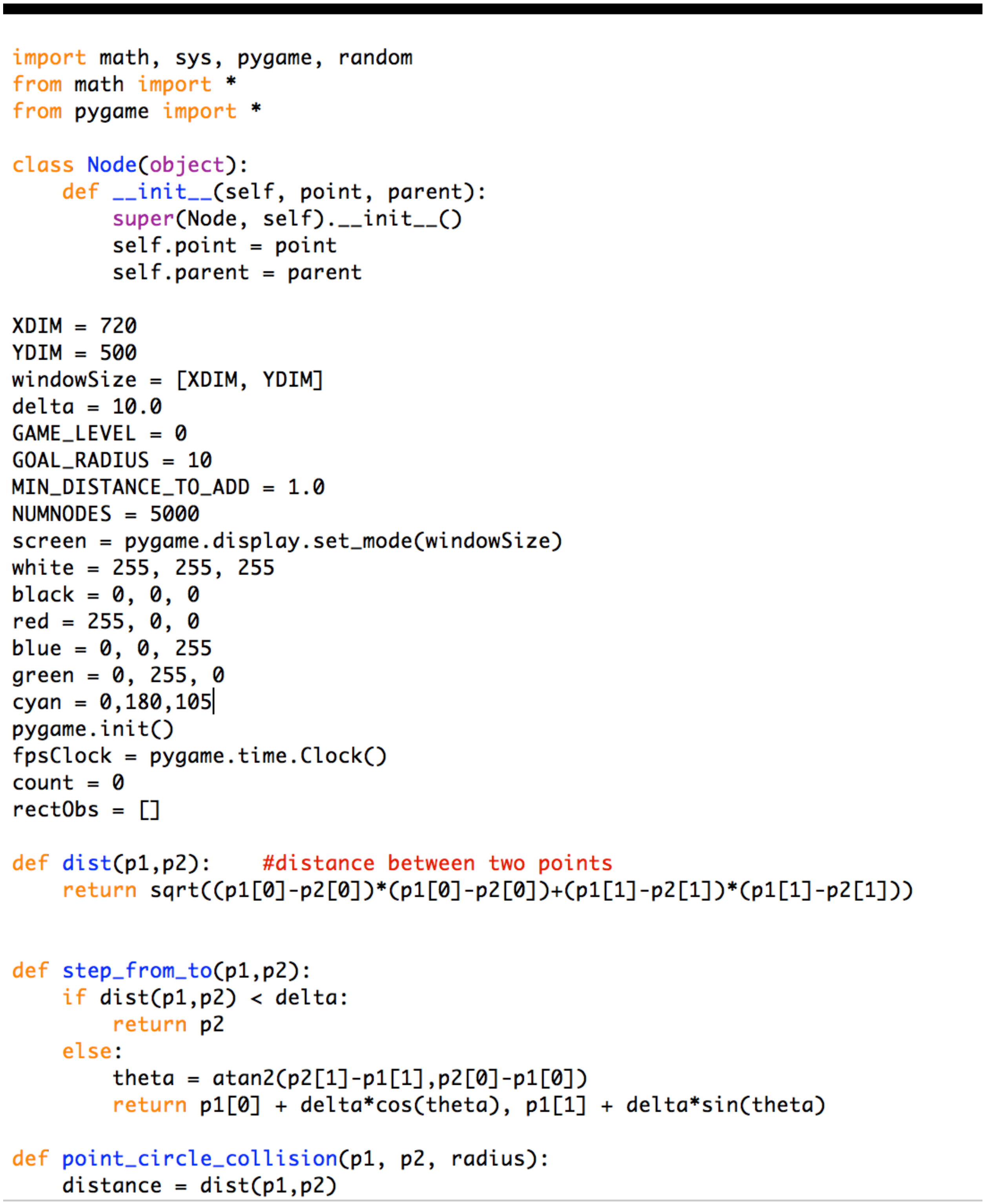
8. **Step_from_to():**

   Makes use of the mathematical functions to check for the point to be traversed next.

9. **Dist():**

   Calculates the distance between two points.

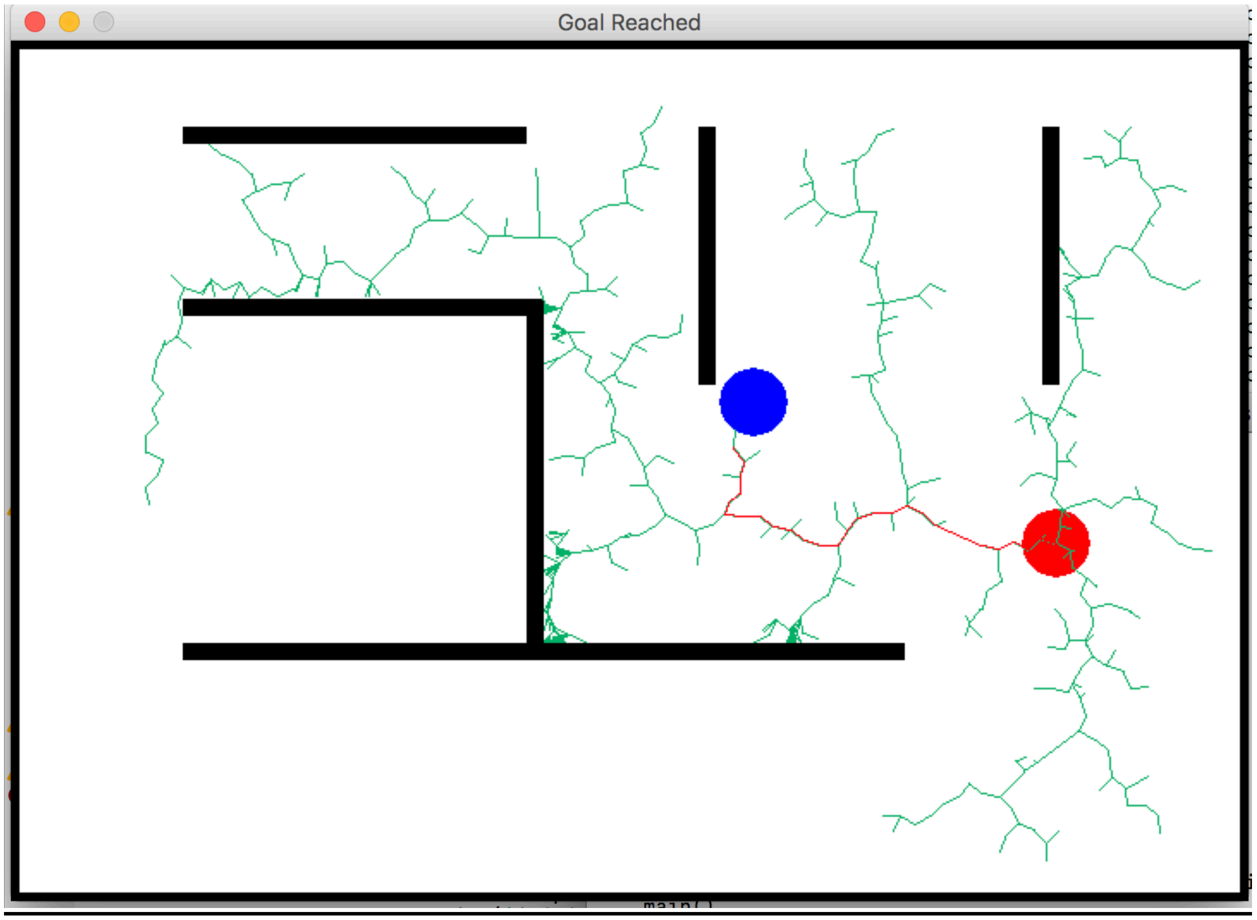
VIII. Code Snippet



```
import math, sys, pygame, random
from math import *
from pygame import *

class Node(object):
    def __init__(self, point, parent):
        super(Node, self).__init__()
        self.point = point
        self.parent = parent

XDIM = 720
YDIM = 500
windowSize = [XDIM, YDIM]
delta = 10.0
GAME_LEVEL = 0
GOAL_RADIUS = 10
MIN_DISTANCE_TO_ADD = 1.0
NUMNODES = 5000
screen = pygame.display.set_mode(windowSize)
white = 255, 255, 255
black = 0, 0, 0
red = 255, 0, 0
blue = 0, 0, 255
green = 0, 255, 0
cyan = 0,180,105
pygame.init()
fpsClock = pygame.time.Clock()
count = 0
rectObs = []

def dist(p1,p2):    #distance between two points
    return sqrt((p1[0]-p2[0])*(p1[0]-p2[0])+(p1[1]-p2[1])*(p1[1]-p2[1]))


def step_from_to(p1,p2):
    if dist(p1,p2) < delta:
        return p2
    else:
        theta = atan2(p2[1]-p1[1],p2[0]-p1[0])
        return p1[0] + delta*cos(theta), p1[1] + delta*sin(theta)

def point_circle_collision(p1, p2, radius):
    distance = dist(p1,p2)
```

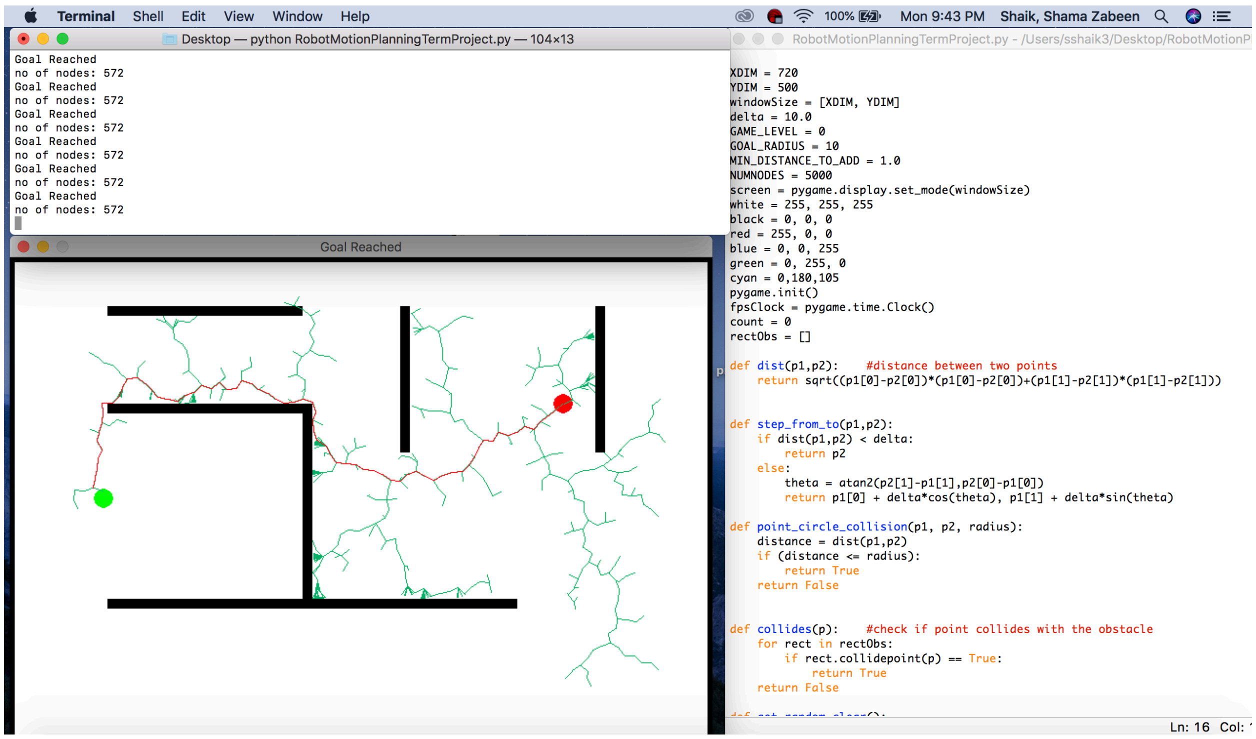
```python
def point_circle_collision(p1, p2, radius):
    distance = dist(p1,p2)
    if (distance <= radius):
        return True
    return False


def collides(p):     #check if point collides with the obstacle
    for rect in rectObs:
        if rect.collidepoint(p) == True:
            return True
    return False

def get_random_clear():
    while True:
        p = random.random()*XDIM, random.random()*YDIM
        noCollision = collides(p)
        if noCollision == False:
            return p


def init_obstacles(configNum):  #initialized the obstacle
    global rectObs
    rectObs = []
    print("config "+ str(configNum))
    if (configNum == 0):
        rectObs.append(pygame.Rect((XDIM / 2.0 - 60, YDIM / 2.0 - 100),(10,200)))
        rectObs.append(pygame.Rect((0,0),(XDIM, 5)))
        rectObs.append(pygame.Rect((0,0),(5, YDIM)))

        rectObs.append(pygame.Rect((XDIM - 5,0),(5, YDIM)))
        rectObs.append(pygame.Rect((0,YDIM-5),(XDIM, 5)))

        rectObs.append(pygame.Rect((100,50),(200,10)))

        rectObs.append(pygame.Rect((100,150),(200,10)))
        rectObs.append(pygame.Rect((100,350),(210,10)))

        rectObs.append(pygame.Rect((310,350),(210,10)))

        rectObs.append(pygame.Rect((400,50),(10,150)))
        rectObs.append(pygame.Rect((600,50),(10,150)))
        print(XDIM)

    if (configNum == 1):
        rectObs.append(pygame.Rect((100,50),(200,150)))
        rectObs.append(pygame.Rect((400,200),(200,100)))
    if (configNum == 2):
        rectObs.append(pygame.Rect((100,50),(200,150)))
    if (configNum == 3):
        rectObs.append(pygame.Rect((100,50),(200,150)))

    for rect in rectObs:
        pygame.draw.rect(screen, black, rect)


def reset():
    global count
    screen.fill(white)
    init_obstacles(GAME_LEVEL)
    count = 0

def main():
    global count

    initPoseSet = False
    initialPoint = Node(None, None)
    goalPoseSet = False
    goalPoint = Node(None, None)
    currentState = 'init'

    nodes = []
    reset()
    myGoalFound = False

    while True:
        if currentState == 'init':
            # print('goal point not yet First set')
            pygame.display.set_caption('First Select Starting Point and then Goal Point')
            #fpsClock.tick(10)
        elif currentState == 'goalFound':
            currNode = goalNode.parent
            pygame.display.set_caption('Goal Reached')
            print("Goal Reached")
            print("no of nodes: "+str(count))
    while currNode.parent != None:
        pygame.draw.line(screen,red,currNode.point,currNode.parent.point)
        currNode = currNode.parent
    optimizePhase = True
elif currentState == 'optimize':
    #fpsClock.tick(0.5)
    pass
elif currentState == 'buildTree':
    count = count+1
    pygame.display.set_caption('Performing RRT')
    if count < NUMNODES:
        foundNext = False
        while foundNext == False:
            rand = get_random_clear()
            parentNode = nodes[0]
            for p in nodes:
                if dist(p.point,rand) <= dist(parentNode.point,rand):
                    newPoint = step_from_to(p.point,rand)
                    if collides(newPoint) == False:
                        parentNode = p
                        foundNext = True

        newnode = step_from_to(parentNode.point,rand)
        nodes.append(Node(newnode, parentNode))
        pygame.draw.line(screen,cyan,parentNode.point,newnode)

        if point_circle_collision(newnode, goalPoint.point, GOAL_RADIUS):
            currentState = 'goalFound'

            goalNode = nodes[len(nodes)-1]


    else:
        print("Ran out of nodes... :(")
        return;

#handle events
for e in pygame.event.get():
    if e.type == QUIT or (e.type == KEYUP and e.key == K_ESCAPE):
        sys.exit("Exiting")
    if e.type == MOUSEBUTTONUP:
        print('mouse down')
        if currentState == 'init':
                return;

        #handle events
        for e in pygame.event.get():
            if e.type == QUIT or (e.type == KEYUP and e.key == K_ESCAPE):
                sys.exit("Exiting")
            if e.type == MOUSEBUTTONUP:
                print('mouse down')
                if currentState == 'init':
                    if initPoseSet == False:
                        nodes = []
                        if collides(e.pos) == False:
                            print('initiale point set: '+str(e.pos))

                            initialPoint = Node(e.pos, None)
                            nodes.append(initialPoint) # Start in the center
                            initPoseSet = True
                            pygame.draw.circle(screen, red, initialPoint.point, GOAL_RADIUS)
                    elif goalPoseSet == False:
                        print('goal point set: '+str(e.pos))
                        if collides(e.pos) == False:
                            goalPoint = Node(e.pos,None)
                            goalPoseSet = True
                            pygame.draw.circle(screen, green, goalPoint.point, GOAL_RADIUS)
                            currentState = 'buildTree'
                else:
                    currentState = 'init'
                    initPoseSet = False
                    goalPoseSet = False
                    reset()

        pygame.display.update()
        # fpsClock.tick(1000)

if __name__ == '__main__':
    main()
```

IX. Snap Shots of the Results



X.

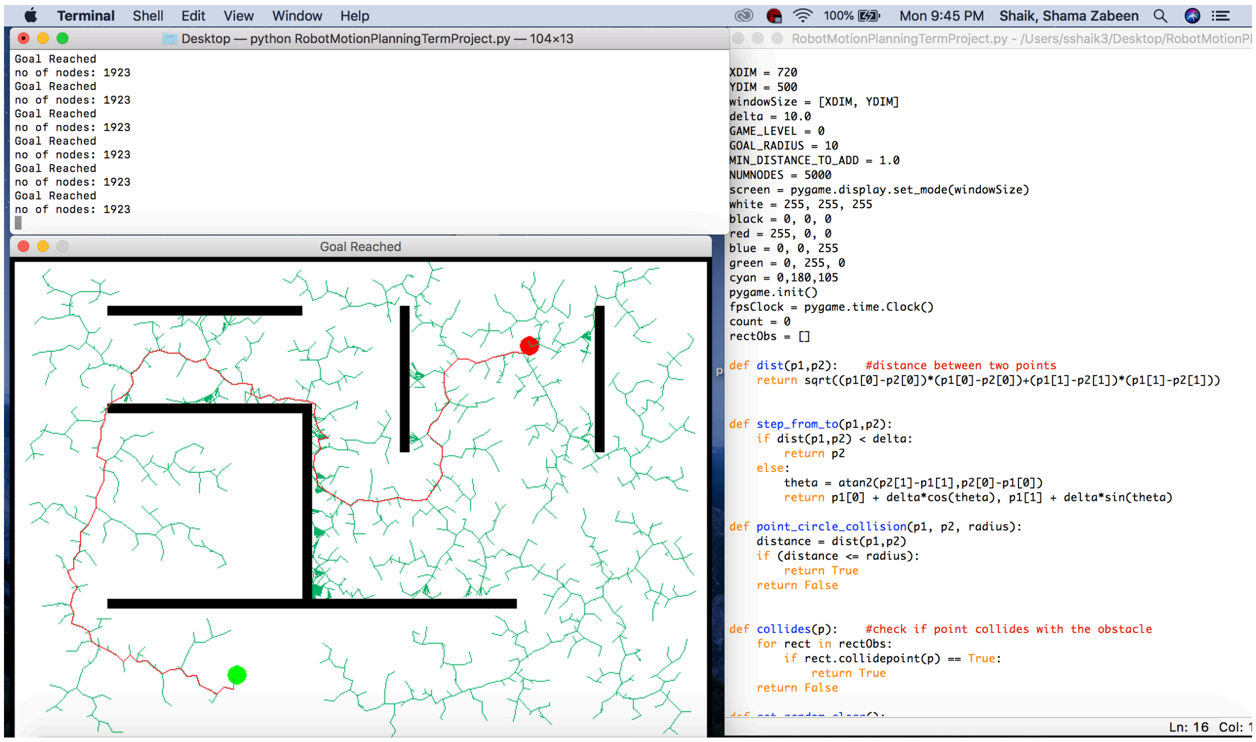
Figure 1: output1


Terminal Shell Edit View Window Help
100% Mon 9:43 PM Shaik, Shama Zabeen
Desktop — python RobotMotionPlanningTermProject.py — 104×13
RobotMotionPlanningTermProject.py - /Users/sshaik3/Desktop/RobotMotionP
Goal Reached
Ln: 16 Col:

```
Goal Reached
no of nodes: 572
Goal Reached
no of nodes: 572
Goal Reached
no of nodes: 572
Goal Reached
no of nodes: 572
Goal Reached
no of nodes: 572
Goal Reached
no of nodes: 572
```

```
XDIM = 720
YDIM = 500
windowSize = [XDIM, YDIM]
delta = 10.0
GAME_LEVEL = 0
GOAL_RADIUS = 10
MIN_DISTANCE_TO_ADD = 1.0
NUMNODES = 5000
screen = pygame.display.set_mode(windowSize)
white = 255, 255, 255
black = 0, 0, 0
red = 255, 0, 0
blue = 0, 0, 255
green = 0, 255, 0
cyan = 0,180,105
pygame.init()
fpsClock = pygame.time.Clock()
count = 0
rectObs = []

def dist(p1,p2):    #distance between two points
    return sqrt((p1[0]-p2[0])*(p1[0]-p2[0])+(p1[1]-p2[1])*(p1[1]-p2[1]))

def step_from_to(p1,p2):
    if dist(p1,p2) < delta:
        return p2
    else:
        theta = atan2(p2[1]-p1[1],p2[0]-p1[0])
        return p1[0] + delta*cos(theta), p1[1] + delta*sin(theta)

def point_circle_collision(p1, p2, radius):
    distance = dist(p1,p2)
    if (distance <= radius):
        return True
    return False

def collides(p):    #check if point collides with the obstacle
    for rect in rectObs:
        if rect.collidepoint(p) == True:
            return True
    return False
```

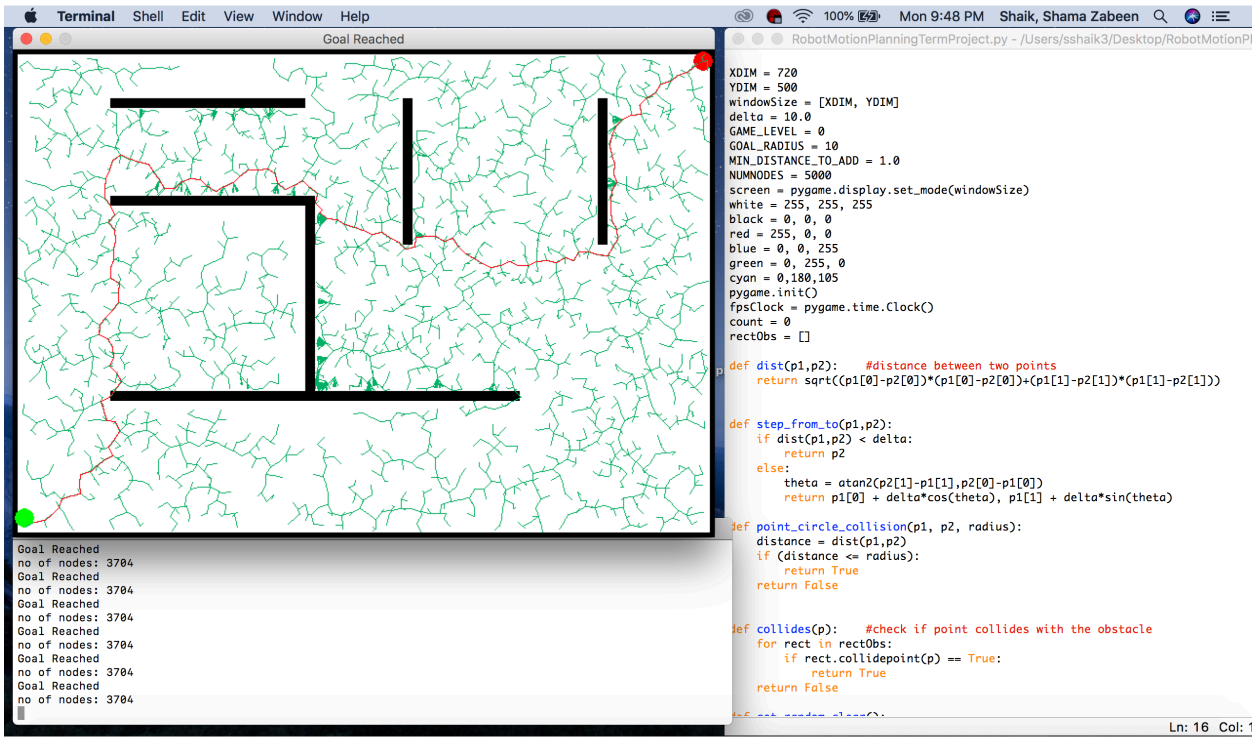
Figure 2: output2


Terminal Shell Edit View Window Help
100% Mon 9:45 PM Shaik, Shama Zabeen
Desktop — python RobotMotionPlanningTermProject.py — 104×13
RobotMotionPlanningTermProject.py - /Users/sshaik3/Desktop/RobotMotionP
Goal Reached
Ln: 16 Col: 1

```
Goal Reached
no of nodes: 1923
Goal Reached
no of nodes: 1923
Goal Reached
no of nodes: 1923
Goal Reached
no of nodes: 1923
Goal Reached
no of nodes: 1923
Goal Reached
no of nodes: 1923
```

```
XDIM = 720
YDIM = 500
windowSize = [XDIM, YDIM]
delta = 10.0
GAME_LEVEL = 0
GOAL_RADIUS = 10
MIN_DISTANCE_TO_ADD = 1.0
NUMNODES = 5000
screen = pygame.display.set_mode(windowSize)
white = 255, 255, 255
black = 0, 0, 0
red = 255, 0, 0
blue = 0, 0, 255
green = 0, 255, 0
cyan = 0,180,105
pygame.init()
fpsClock = pygame.time.Clock()
count = 0
rectObs = []

def dist(p1,p2):    #distance between two points
    return sqrt((p1[0]-p2[0])*(p1[0]-p2[0])+(p1[1]-p2[1])*(p1[1]-p2[1]))

def step_from_to(p1,p2):
    if dist(p1,p2) < delta:
        return p2
    else:
        theta = atan2(p2[1]-p1[1],p2[0]-p1[0])
        return p1[0] + delta*cos(theta), p1[1] + delta*sin(theta)

def point_circle_collision(p1, p2, radius):
    distance = dist(p1,p2)
    if (distance <= radius):
        return True
    return False

def collides(p):    #check if point collides with the obstacle
    for rect in rectObs:
        if rect.collidepoint(p) == True:
            return True
    return False
```

Figure 3:output3



```
XDIM = 720
YDIM = 500
windowSize = [XDIM, YDIM]
delta = 10.0
GAME_LEVEL = 0
GOAL_RADIUS = 10
MIN_DISTANCE_TO_ADD = 1.0
NUMNODES = 5000
screen = pygame.display.set_mode(windowSize)
white = 255, 255, 255
black = 0, 0, 0
red = 255, 0, 0
blue = 0, 0, 255
green = 0, 255, 0
cyan = 0,180,105
pygame.init()
fpsClock = pygame.time.Clock()
count = 0
rectObs = []

def dist(p1,p2):    #distance between two points
    return sqrt((p1[0]-p2[0])*(p1[0]-p2[0])+(p1[1]-p2[1])*(p1[1]-p2[1]))


def step_from_to(p1,p2):
    if dist(p1,p2) < delta:
        return p2
    else:
        theta = atan2(p2[1]-p1[1],p2[0]-p1[0])
        return p1[0] + delta*cos(theta), p1[1] + delta*sin(theta)

def point_circle_collision(p1, p2, radius):
    distance = dist(p1,p2)
    if (distance <= radius):
        return True
    return False


def collides(p):    #check if point collides with the obstacle
    for rect in rectObs:
        if rect.collidepoint(p) == True:
            return True
    return False
```

```
Goal Reached
no of nodes: 3704
Goal Reached
no of nodes: 3704
Goal Reached
no of nodes: 3704
Goal Reached
no of nodes: 3704
Goal Reached
no of nodes: 3704
Goal Reached
no of nodes: 3704
```

Figure 4:Output4

X. References

1. https://docs.python.org/2/library/random.html
2. https://www.pygame.org/docs/ref/event.html#pygame.event.clear
3. https://www.pygame.org/wiki/CommandDispatch
4. https://www.reddit.com/r/pygame/comments/5upg5y/pygamerectcollidepoint_and_pong_help/
5. https://ieeexplore.ieee.org/abstract/document/1308895/
6. http://www.robotplatform.com/knowledge/Classification_of_Robots/Holonomic_and_Non-Holonomic_drive.html
7. http://msl.cs.uiuc.edu/rrt/about.html
8. https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree
9. https://www.youtube.com/watch?v=65XBl7y9mSM