Project Report: Robot Navigation Using Rapidly-Exploring Random Tree

Problem Statement:

This is a Robot Navigation problem. In this problem, we have a Robot who needs to go from a certain point A(Start) to point B (Goal). The robot does not know in which direction or at what position the goal is located. Also, the path determined by the robot also needs to be such that it avoids all the obstacles present in the map. This is also a very common problem faced in game design, where a bot needs to go from location A to B by avoiding all obstacles. Also, the space is not discrete and hence many graph traversal algorithms like A* are hard to implement. We need to first construct a graph or have a discrete space like a grid and then apply A* algorithm.

Assumption:

- The shape of the map and its boundaries are known.
- The Goal is not known.
- The obstacles are polygon (Both concave and convex)
- Goal is to just find a path from start to end rather than to find optimized path

Rapidly-exploring random tree:

One solution to this problem is rapidly exploring random tree.

The Official page of RRT provides below explanation of RRT:

“A Rapidly-exploring Random Tree (RRT) is a data structure and algorithm that is designed for efficiently searching nonconvex high-dimensional spaces. RRTs are constructed 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 (nonholonomic or Kino dynamic). 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.” (source: http://msl.cs.uiuc.edu/rrt/about.html)

Key Advantages of RRT are:

1. The expansion of RRT is heavily biased towards unexplored portion of state space.
2. The distribution vertices in an RRT approaches the sampling distribution, leading to consistent behavior.
3. An RRT is probabilistically complete under very general conditions
4. The RRT algorithm is relatively simple which facilitates performance analysis (this is also preferred feature of probabilistic road maps)
5. An RRT always remains connected, even though the number of edges is minimal.
6. An RRT can be considered as a path planning module, which can be adapted and incorporated into wide variety of planning systems.
7. Entire path planning algorithm can be constructed without requiring the ability to steer the system between two prescribed states which greatly broadens applicability of RRTs (Source: Rapidly-Exploring Random tree: A New tool for Path Planning)

Algorithm:

The Algorithm can be briefly stated as:

```
G.init(qinit)
 for k = 1 to K
  qrand ← RAND_CONF()
  qnear ← NEAREST_VERTEX(qrand, G)
  qnew ← NEW_CONF(qnear, qrand, Δq)
  G.add_vertex(qnew)
  G.add_edge(qnear, qnew)
```

Description of above algorithm:

1. Step 1 is to initialize the starting configuration or point on map. This point is specified by the user.

qinit/Start Node

Figure 1

2. Step 2 is to Generate Random Configuration. This means to sample a random point which lies inside our configuration space or in this case the rectangular map in which robot navigates. If we have obstacles in our map, then we use Random_free_config and sample in C_free space. That is, we sample multiple points and reject all those points which lie inside obstacle. For this I, have used collision detection algorithm Ray casting.

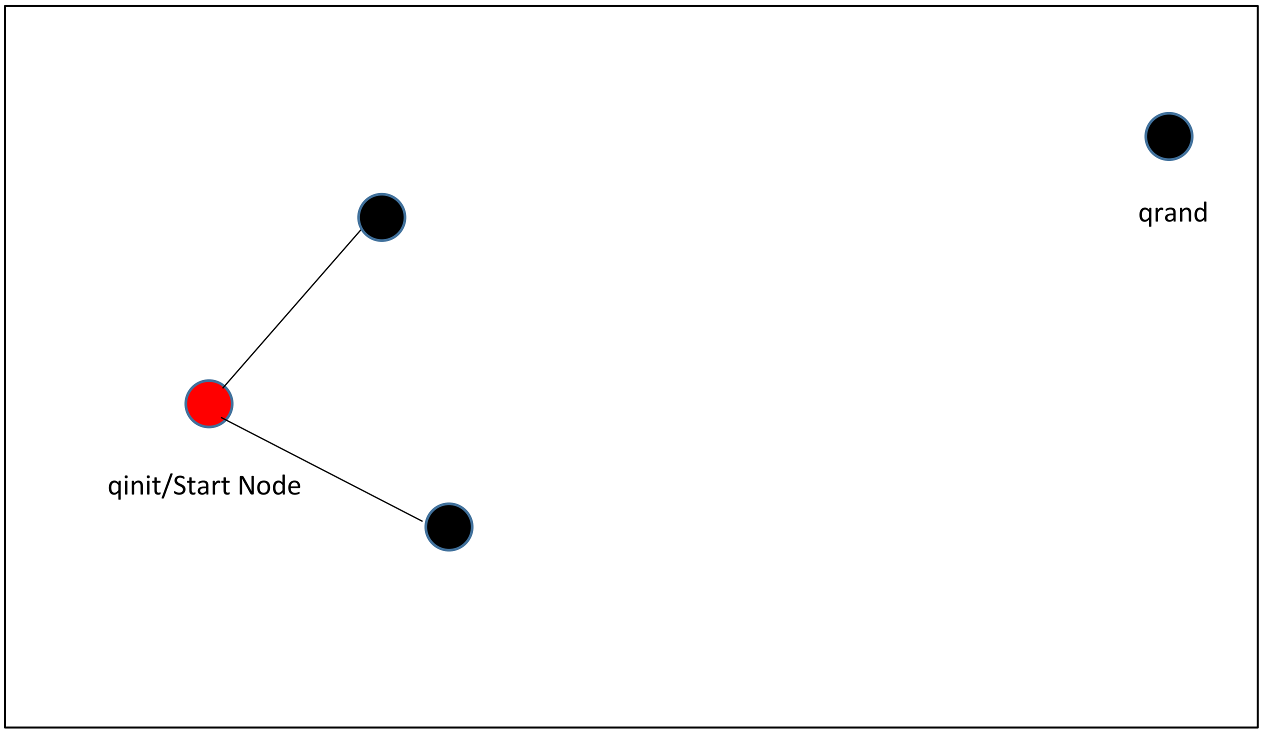


Figure 2

3. Select a Neighbor such that it is nearest to the random config or has minimum distance. In this Project I have used Euclidean distance metric to find the qnear (nearest node to the sampled qrand node.)

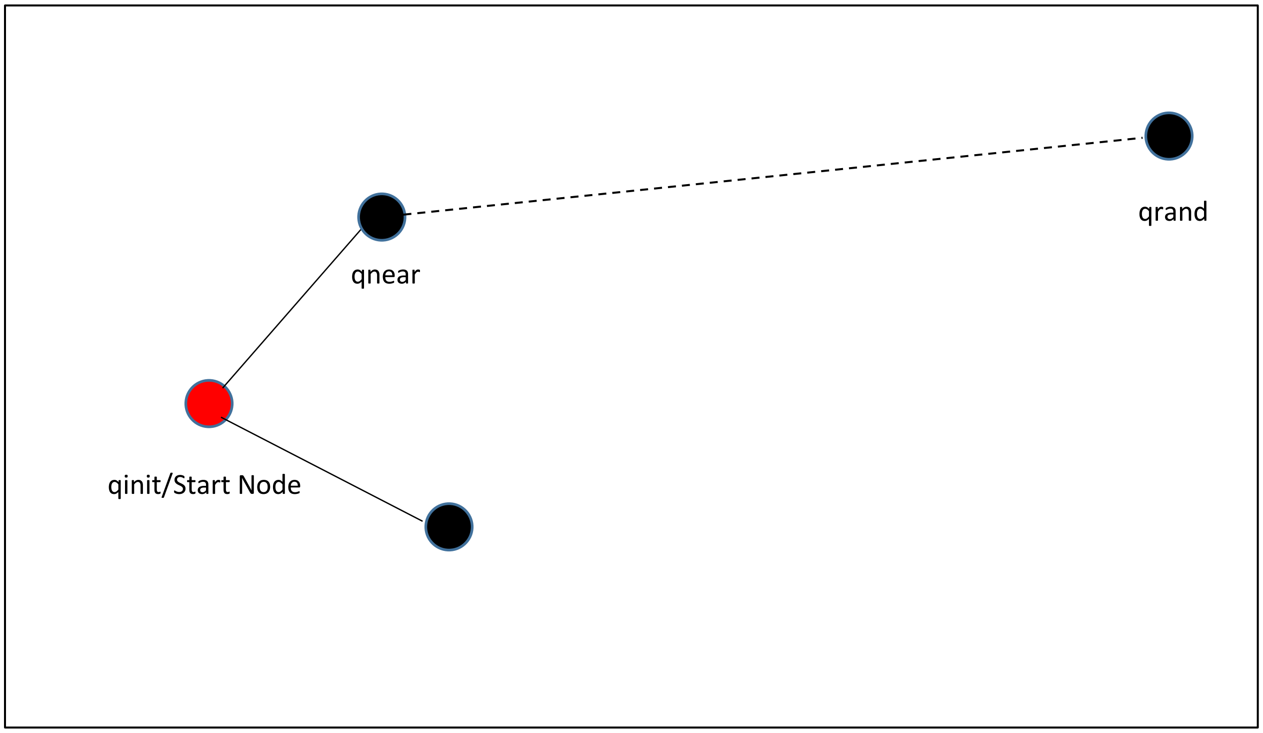


Figure 3

4. Take a Delta step from qnear in direction of the qrand. The Delta is step size and is predefined. Finally, a new node qnew is added to our vertex list. Also, this assumes that the robot can move in any direction. In case of specific constrain in robot motion we need to apply those in this step to generate a new node.

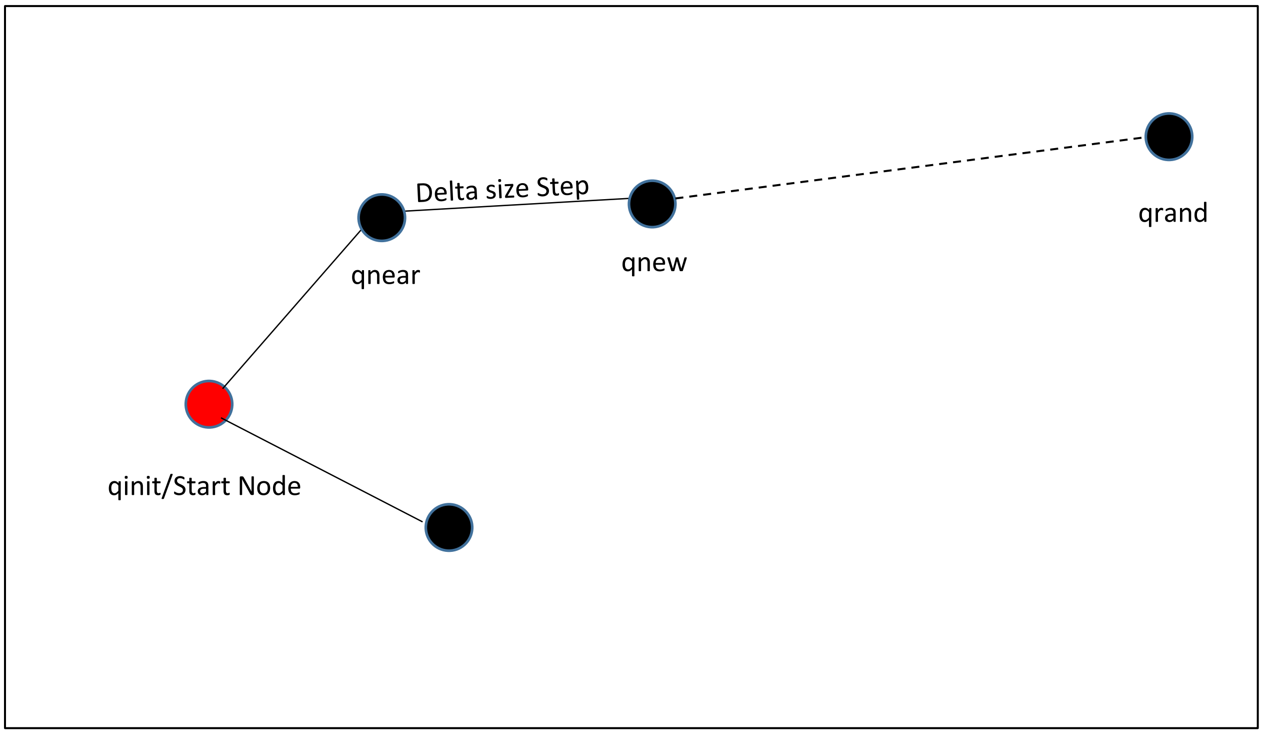


Figure 4

5. Check whether the newly added node is close to the goal node. If that is true than stop the expansion of RRT and draw a path from qnew to goal node. Draw a path from the current qnew node to the start node.

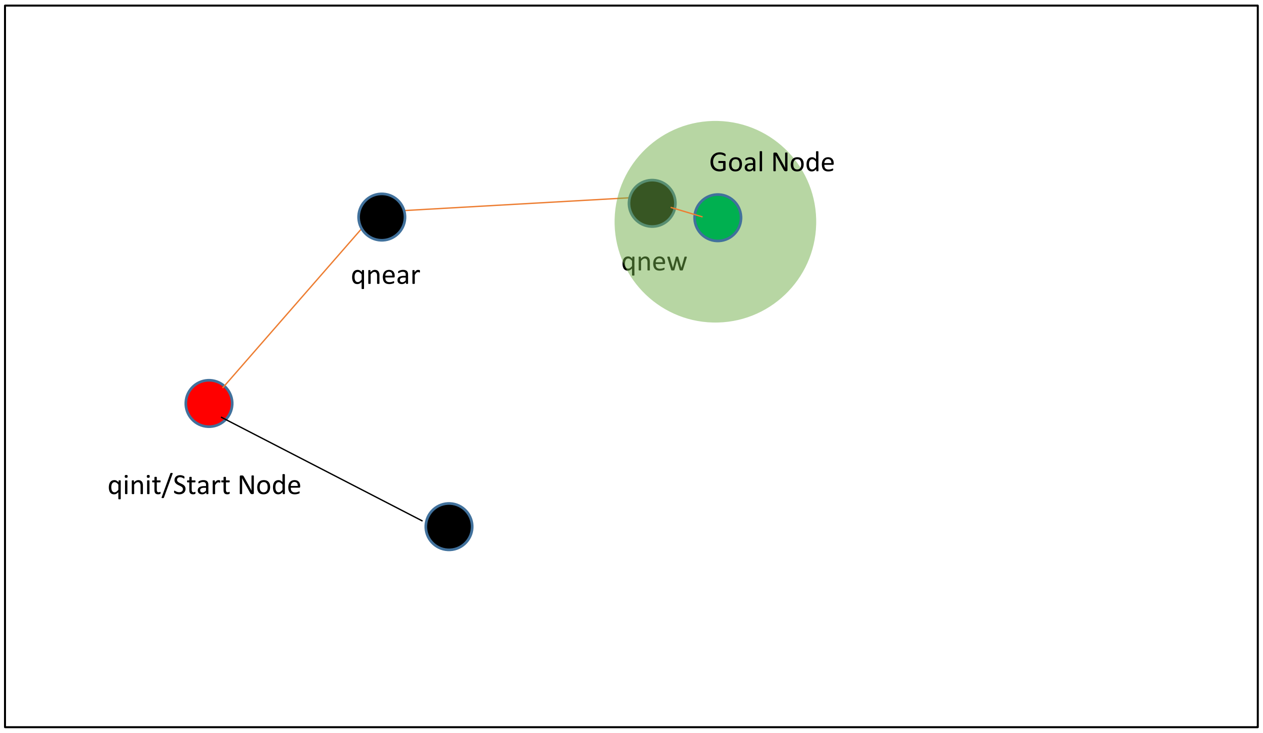


Figure 5

Collision Detection:

During generation of Random configuration, we sample multiple points. Than we check each point if it lies inside a obstacle or not. If the point passes this test, then we use that point. To detect whether that point lies inside a obstacle or not we need a collision checking system. In this project the obstacles are polygons. There are concave as well as convex polygons present inside map as obstacles. So, for collision detection I use Ray casting algorithm.

Ray Casting:

One simple way of finding whether the point is inside or outside a simple polygon is to test how many times a ray, starting from the point and going in any fixed direction, intersects the edges of the polygon. If the point is on the outside of the polygon the ray will intersect its edge an even number of times. If the point is on the inside of the polygon, then it will intersect the edge an odd number of times. Unfortunately, this method won't work if the point is on the edge of the polygon. Hence when a point is on the edge we shift it by a very small value.

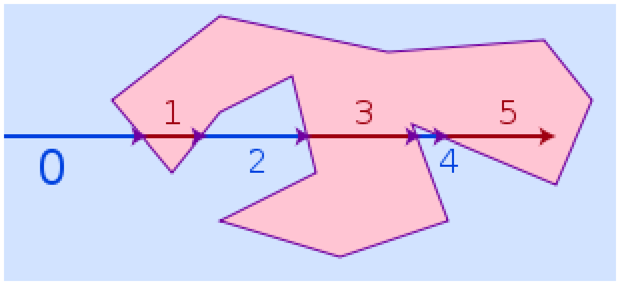


Figure 6 (Source: By Melchoir [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons)

Algorithm for Ray Casting:

1. Given a point for which we need to check whether that point lies inside the polygon or not and list of vertices which form the polygon.
2. For every edge in the polygon check whether the given point intersects the edge.
3. If it intersects the edge than count it.
4. If it does not intersect the edges, then do not count it.
5. Repeat above steps for all edges in the polygon.
6. If the count it even, then point does not collide with polygon.
7. If count is odd, then point collides with the polygon

Algorithm to check whether a ray from point intersects a line segment:

1. We are given the point for which we need to check whether it intersects a line segment. As shown in below diagram we have four points and Line segment AB.
2. As shown in diagram below:

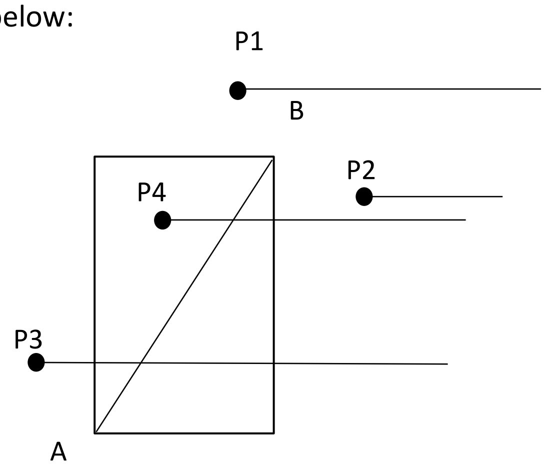


Figure 7

   Points P1 and P2 can easily be checked by checking if their y-coordinate is greater than A or less than B. If so then they do not collide with Polygon.
3. Point P3 can also be checked by checking whether a point lies beyond max(Ax, Bx) or behind min(Ax, Bx).
4. For point P4 we need to check whether it lies to left side of line segment or right side of the line segment. If it lies to left side, then it intersects the line segment. If it lies on right side than it does not intersect the line segment AB.
5. To Check the position of the point we check the slope of line formed by joining P4 & A and B &A.
6. If the Slope of line P4-A is greater than line B-A than it lies on left side else it lies on right side of line.

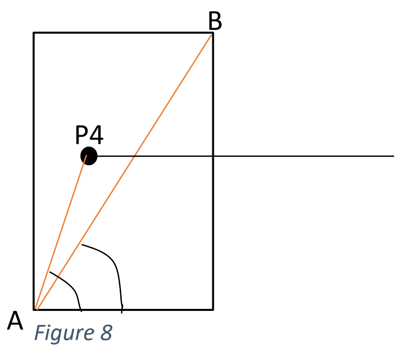
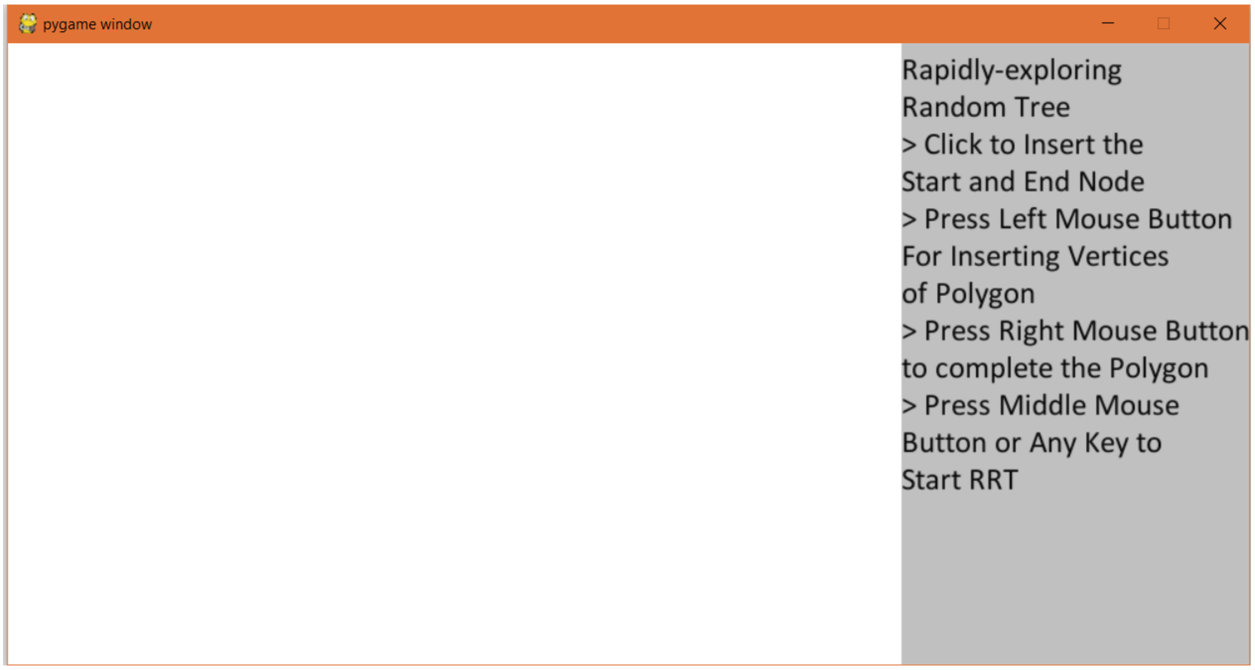


Figure 8

Graphical User Interface:



Rapidly-exploring
Random Tree
> Click to Insert the
Start and End Node
> Press Left Mouse Button
For Inserting Vertices
of Polygon
> Press Right Mouse Button
to complete the Polygon
> Press Middle Mouse
Button or Any Key to
Start RRT

Figure 9

- The GUI for project is very simple. The white area is where the RRT is constructed. On right side are the instruction on How to use this program.
- To place the start and end Node just do mouse click.
- First the initial node is placed and then the final node is place at location where mouse is clicked.
- After this polygon mode is started. To draw a polygon just do left click to place vertices and once enough vertices (more than 2) are placed then press right mouse button to complete the polygon.
- Once done with polygons just press any key or middle mouse button to begin the RRT.
- Once RRT has been started the right-side changes to display information like time in seconds and Number of nodes currently explored.

Program Structure:

Class:

```python
class Node(object):
    def __init__(self,point,Parent_node):  #Constructor Function
        super(Node, self).__init__()
        self.parent_pointer = Parent_node  #Node class has two variable :
        self.point = point   #1. Pointer to Parent 2. Point holding the X & Y
Cordinates of that Node
```

Global Variables:

```python
XDIM = 1030   #Window Size X
YDIM = 500    #Window Size Y
BORDER = 720  #Border size of the RRT region/White Region
windowsize = [XDIM,YDIM]  #Variable to Hold Window Size
delta = 10.0  #Delta Step Size
RADIUS = 10   #Radius to for Displaying Start and End Node
SMALL_RADIUS = 2  #Radius for Displaying vertices
fpsClock = pygame.time.Clock()  #Clock variable to control Display Speed
screen = pygame.display.set_mode(windowsize)  #Screen variable used for all Display
white = 255, 255, 255  #Color Values
black = 0, 0, 0  #Color Values
StartNodeColor = 255, 0, 0  #Color Values
blue = 0, 0, 255  #Color Values
GoalNodeColor = 0, 255, 0  #Color Values
lineColor = 0, 180, 105  #Color Values
gray = 128, 128, 128  #Color Values
maroon = 192, 192, 192  #Color Values
GoalPathColor = 0, 0, 0  #Color Values
node = []  #List Containing Nodes in RRT
goalNode = Node(None,None)  #Node Holding the Goal
initialNode = Node(None,None)  #Node Holding Initial Node
polygon_list = [] #List holding all the Polygons
```

Functions:

```python
def reset():  #Function to Reset and initialize all the varibles and also to display
instructions


def generate_rrt():  #This Function return a new Node and Displays the edge to Display


def check_collision_polygon(point):  #Function to Check Collison between point and all
the polygon



def ray_casting(point,vertex):  #Function to check collision between a point and a
polygon



def ray_intersect(P,A,B):  #Function to check whether a single point intersects a edge.


def draw_path(u):  #Function to draw path from Node u to initial Node



def check_goal(u):  #Function to check whether u is Goal node



def Check_start_state_collide(u):  #Function to check whether point collides with starting Node



def Random_config():  #Function to generate Random points which are constrain by XDIM and Border varible



def select_state(x_rand,near_node):  #Function to generate new node which is at distance of Delta from near_node and in direction of x_rand



def nearest_neighbour(x_rand):  #Function to select node which has minimum Euclidean distance from x_rand



def dist(p1,p2):  #Function to calculate Euclidean distance between two points


def border_check(p): #Check wheter point p crosses the border.

```

Program Code:

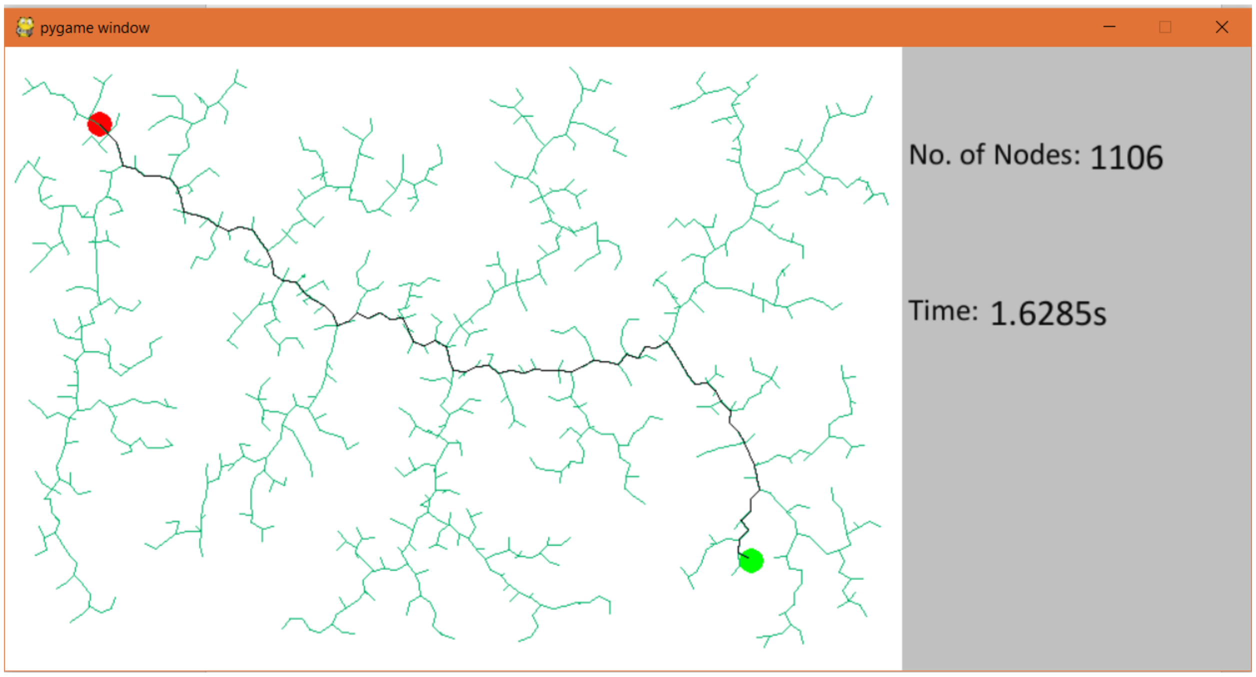
```
import math, sys, pygame, random
from math import *
from pygame import *
import time
class Node(object):
    def __init__(self,point,Parent_node):
        super(Node, self).__init__()
        self.parent_pointer = Parent_node
        self.point = point


def reset():
    global node
    screen.fill(white)
    node = []
    pygame.draw.polygon(screen, maroon, PointList, 0)
    text = font2.render("Rapidly-exploring", 1, (10, 10, 10))
    screen.blit(text, (BORDER,10))
    text = font2.render("Random Tree", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 40))
    text = font2.render("> Click to Insert the", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 70))
    text = font2.render("Start and End Node", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 100))
    text = font2.render("> Press Left Mouse Button", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 130))
    text = font2.render("For Inserting Vertices ", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 160))
    text = font2.render("of Polygon", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 190))
    text = font2.render("> Press Right Mouse Button", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 220))
    text = font2.render("to complete the Polygon", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 250))
    text = font2.render("> Press Middle Mouse ", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 280))
    text = font2.render("Button or Any Key to", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 310))
    text = font2.render("Start RRT", 1, (10, 10, 10))
    screen.blit(text, (BORDER, 340))


def generate_rrt():
    global node
    while True:
        x_rand = Random_config()
        x_near = nearest_neighbour(x_rand)
        u = select_state(x_rand,x_near)
        if check_collision_polygon(u.point) is None:
            pygame.draw.line(screen, lineColor, x_near.point, u.point)
            pygame.draw.polygon(screen, maroon, PointList, 0)
            text = font2.render('No. of Nodes:', 1, (10, 10, 10))
            screen.blit(text, (BORDER+5, 0.15*YDIM))
            text = font.render(str(len(node)), 1, (10, 10, 10))
            screen.blit(text, (BORDER+150, 0.15 * YDIM))
            text = font2.render('Time:', 1, (10, 10, 10))
            screen.blit(text, (BORDER+5, 0.4 * YDIM))
            text = font.render(str(((float("{0:.4f}".format(time.time() - start_time)))
)), 1, (10, 10, 10))
            screen.blit(text, (BORDER+70, 0.4*YDIM))
            text = font.render('s', 1, (10, 10, 10))
            screen.blit(text, (BORDER + 170, 0.4 * YDIM))
            return u


def check_collision_polygon(point):
    for i in range(0,len(polygon_list)):
        vertex = polygon_list[i]
        if ray_casting(point,vertex)==1:
            return 1


def ray_casting(point,vertex):
    count = 0
    for i in range(0,len(vertex)):
        if i != (len(vertex)-1):
            if vertex[i][1] < vertex[i+1][1]:
                count += ray_intersect(point,vertex[i],vertex[i+1])
            else:
                count += ray_intersect(point, vertex[i+1], vertex[i])
        else:
            if vertex[i][1] < vertex[0][1]:
                count += ray_intersect(point,vertex[i],vertex[0])
```

```python
            else:
                count += ray_intersect(point, vertex[0], vertex[i])
    if count % 2 == 0:
        return 0
    else:
        return 1


def ray_intersect(P,A,B):
    if P[1] == A[1] or P[1] == B[1]:
        P[1] += 10
    if P[1] < A[1] or P[1] > B[1]:
        return 0
    elif P[0] > max(A[0],B[0]):
        return 0
    else:
        if P[0] < min(A[0],B[0]):
            return 1
        else:
            if A[0] != B[0]:
                segment_slope = (B[1]-A[1])/(B[0]-A[0])
            else:
                segment_slope = inf
            if A[0]!=P[0]:
                ray_slope = (P[1] - A[1])/(P[0] - A[0])
            else:
                ray_slope = inf
            if segment_slope < ray_slope:
                return 1
            else:
                return 0


def draw_path(u):
    while u.parent_pointer!=None:
        pygame.draw.line(screen, GoalPathColor,u.point,u.parent_pointer.point)
        u = u.parent_pointer


def check_goal(u):
    if dist(u.point,goalNode.point) <= RADIUS:
        return True
    else:
        return False


def Check_start_state_collide(u):
    if dist(initialNode.point,u) <= RADIUS:
        return True
    else:
        return False


def Random_config():
    while True:
        p = random.random()*BORDER, random.random()*YDIM
        if Check_start_state_collide(p)!=True:
            break
    return p

    def select_state(x_rand,near_node):
        if dist(x_rand,near_node.point) <= delta:
            newNode = Node(x_rand,near_node)
            return newNode
        theta = atan2((x_rand[1]-
near_node.point[1]),(x_rand[0] - near_node.point[0]) )
        point = cos(theta)*delta + near_node.point[0], sin(theta)*delta + near_node.
point[1]
        newNode = Node(point,near_node)
        return newNode


    def nearest_neighbour(x_rand):
        min_dist = dist(node[0].point,x_rand)
        near_node = node[0]
        for p in node:
            if dist(p.point,x_rand)<=min_dist:
                min_dist = dist(p.point,x_rand)
                near_node = p
        return near_node


    def dist(p1,p2):
        return sqrt((p1[1] - p2[1])**2 + (p1[0] - p2[0])**2)


    def border_check(p):
        if p[0] < BORDER:
            return True
        else:
            return False


    XDIM = 1030
    YDIM = 500
    BORDER = 720
    windowsize = [XDIM,YDIM]
    delta = 10.0
    RADIUS = 10
    SMALL_RADIUS = 2
    pygame.init()
    fpsClock = pygame.time.Clock()
    screen = pygame.display.set_mode(windowsize)
    white = 255, 255, 255
    black = 0, 0, 0
    StartNodeColor = 255, 0, 0
    blue = 0, 0, 255
    GoalNodeColor = 0, 255, 0
    lineColor = 0, 180, 105
    gray = 128, 128, 128
    maroon = 192, 192, 192
    GoalPathColor = 0, 0, 0
    node = []
    goalNode = Node(None,None)
    initialNode = Node(None,None)
    polygon_list = []
    font = pygame.font.Font('C:\Windows\Fonts\Calibri.ttf', 30)
    font2 = pygame.font.Font('C:\Windows\Fonts\Calibri.ttf', 25)

    PointList = []
    PointList.append((BORDER, 0))
```

```python
PointList.append((XDIM, 0))
PointList.append((XDIM, YDIM))
PointList.append((BORDER, YDIM))
start_time = 0

def main():
    reset()
    K = 5000
    done_flag=0
    global goalNode
    global initialNode
    global start_time
    currentstate = 'init'
    initPose = False
    finalPose = False
    vertices = []
    global polygon_list
    while True:
        if currentstate == 'StartRRT':
            if K > 0:
                u = generate_rrt()
                K-=1
                node.append(u)
                if check_goal(u):
                    draw_path(u)
                    currentstate = "Found RRT"
                    print("Found")
            else:
                if done_flag==0:
                    text = font.render('Uggh ran out of nodes!!', 1, (10, 10, 10))
                    screen.blit(text, (BORDER + 10 , 0.6 * YDIM))
                    done_flag = 1
        for e in pygame.event.get():
            if e.type == QUIT or (e.type == KEYUP and e.key == K_ESCAPE):
                sys.exit("Exiting")
            elif e.type == KEYUP and (currentstate == 'SetPolygon'):
                start_time = time.time()
                if not vertices:
                    currentstate = 'StartRRT'
                else:
                    polygon_list.append(vertices)
                    pygame.draw.polygon(screen, gray, vertices, 0)
                    pygame.display.update()
                    currentstate = 'StartRRT'
            if e.type == MOUSEBUTTONDOWN:
                if currentstate == 'init':
                    if initPose == False:
                        if border_check(e.pos)==True:
                            initialNode = Node(e.pos,None)
                            pygame.draw.circle(screen,StartNodeColor,initialNode.point,RADIUS)
                            initPose = True
                            node.append(initialNode)
                    elif finalPose == False:
                        if border_check(e.pos)==True:
                            goalNode = Node(e.pos,None)
                            pygame.draw.circle(screen,GoalNodeColor,goalNode.point,RADIUS)
                            currentstate = 'SetPolygon'
                            finalPose = True
                elif currentstate == 'SetPolygon' or currentstate=='check':
                    button = pygame.mouse.get_pressed()
                    if button[0] == 1:
                        if border_check(e.pos)==True:
                            vertices.append(e.pos)
                            pygame.draw.circle(screen, gray, e.pos, SMALL_RADIUS)
                    elif button[2] == 1 :
                        if vertices:
                            polygon_list.append(vertices)
                            pygame.draw.polygon(screen,gray,vertices,0)
                            vertices = []
                    elif button[1] == 1:
                        start_time = time.time()
                        if not vertices:
                            currentstate = 'StartRRT'
                        else:
                            polygon_list.append(vertices)
                            pygame.draw.polygon(screen, gray, vertices, 0)
                            vertices = []
                            currentstate = 'StartRRT'
                else:
                    currentstate = 'init'
                    initPose = False
                    finalPose = False
                    vertices = []
                    K = 5000
                    done_flag = 0
                    polygon_list = []
                    reset()
        pygame.display.update()
        fpsClock.tick(1000)


if __name__ == '__main__':
    main()

```

Result:

Case # 1:

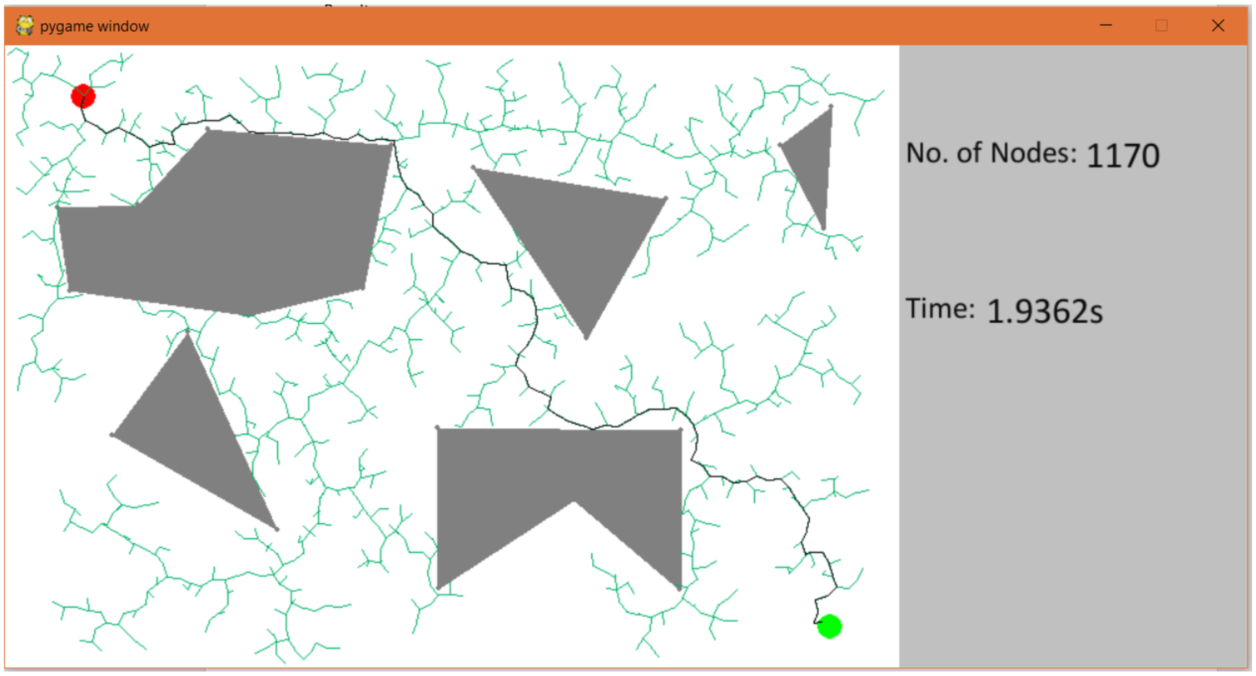


No. of Nodes: 1106

Time: 1.6285s

Figure 10

Case #2:

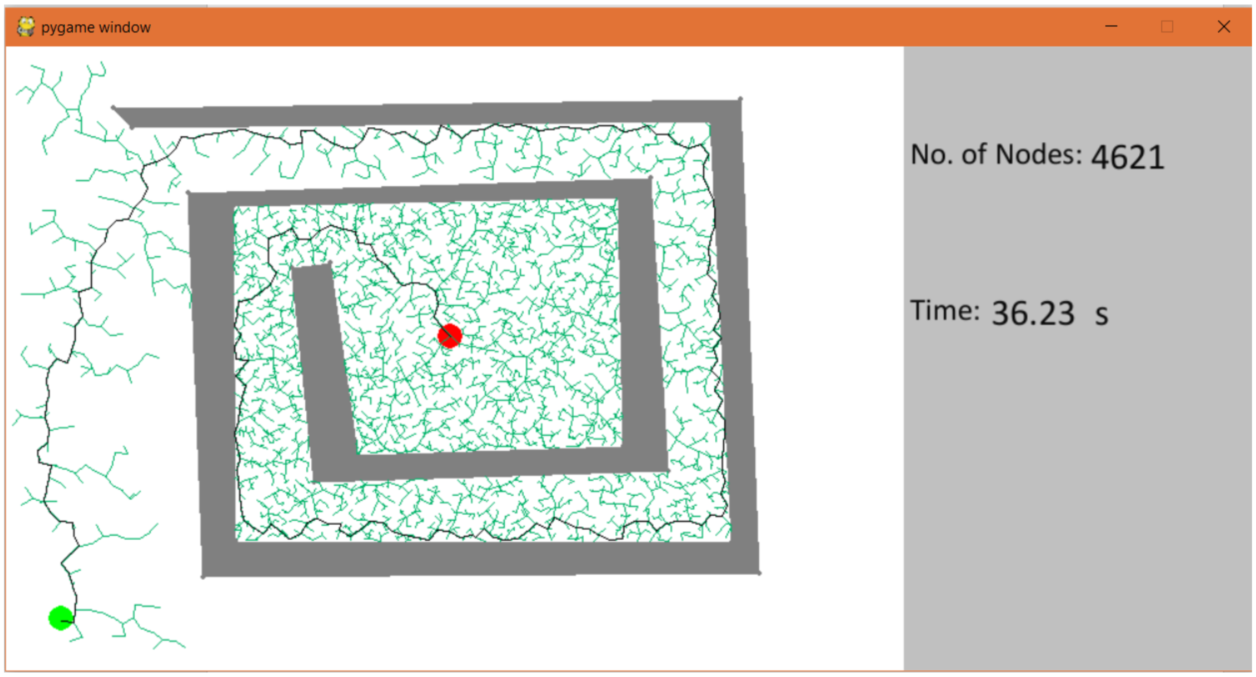


No. of Nodes: 1170

Time: 1.9362s

Figure 11

Case #3:

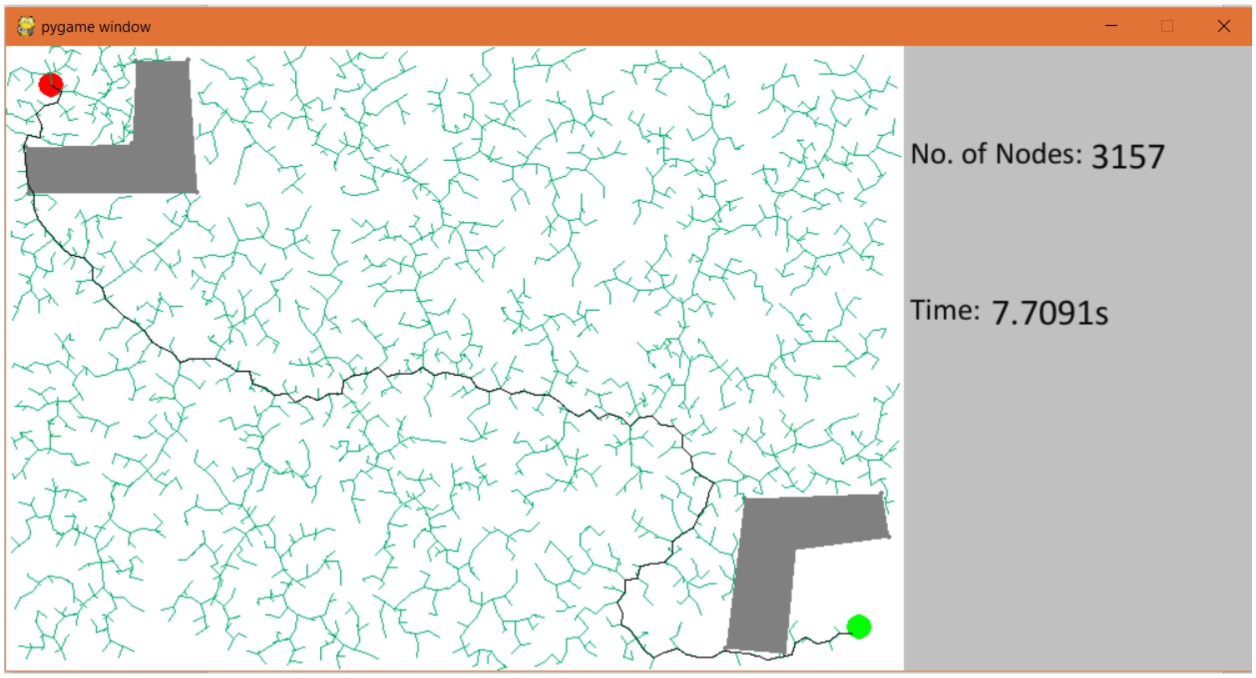


No. of Nodes: 4621

Time: 36.23 s

Figure 12

Case #4:

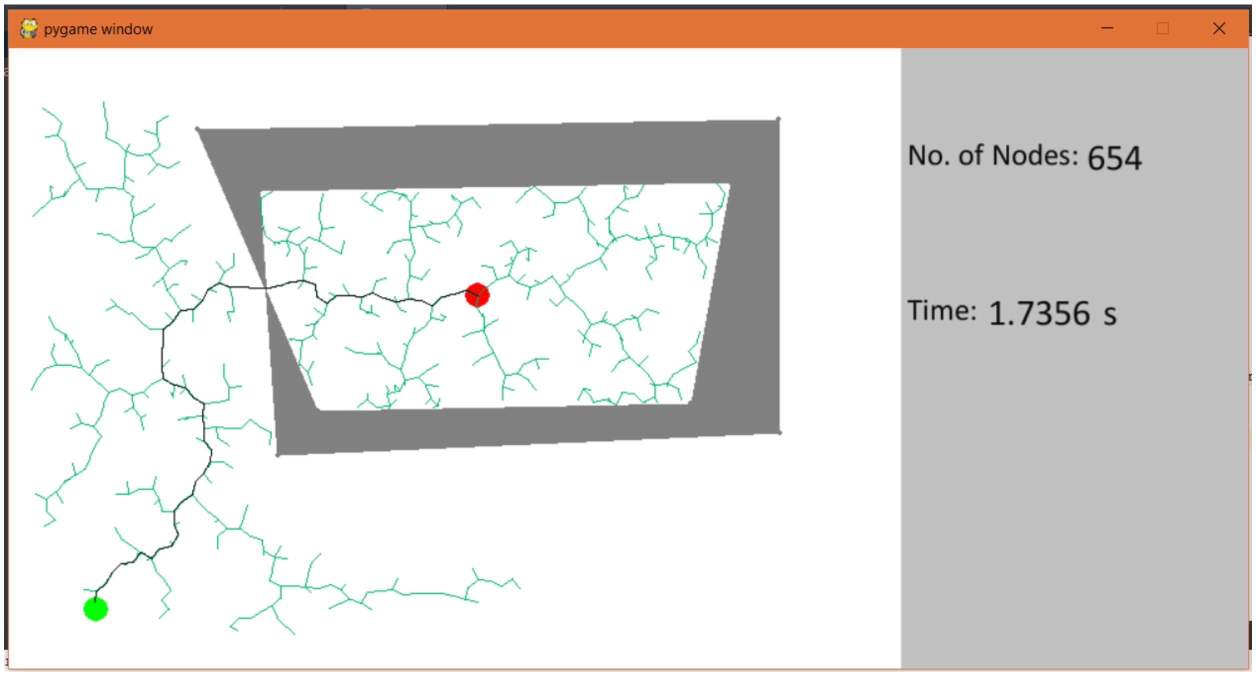


No. of Nodes: 3157

Time: 7.7091s

Figure 13

Case #5:



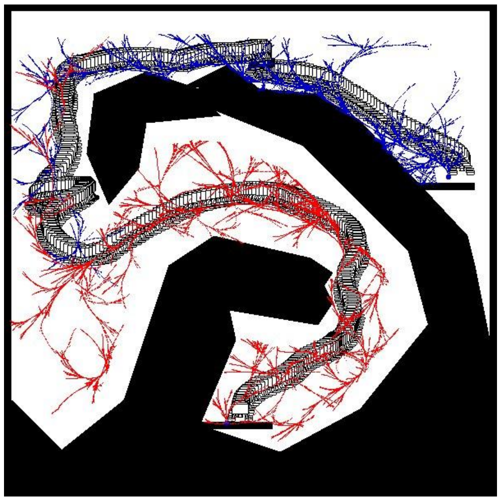
No. of Nodes: 654

Time: 1.7356 s

Figure 14

Application:

1. Landing a Spacecraft Under Lunar Gravity:
   This example involves a rigid planar body with two small side thrusters, and a larger lower thruster. The goal is to navigate and softly "land" the craft by firing thrusters, in spite of gravity. The initial state places the craft at rest on the pad in the upper right. The goal state places the craft at rest on the pad in the lower left. RRTs were grown from the initial and goal states, respectively, until they met each other. A 2D projection of the RRTs is shown below, along with a computed trajectory. The RRT from the initial



   state is shown in blue, and the RRT from the goal is shown in red.

Figure 15

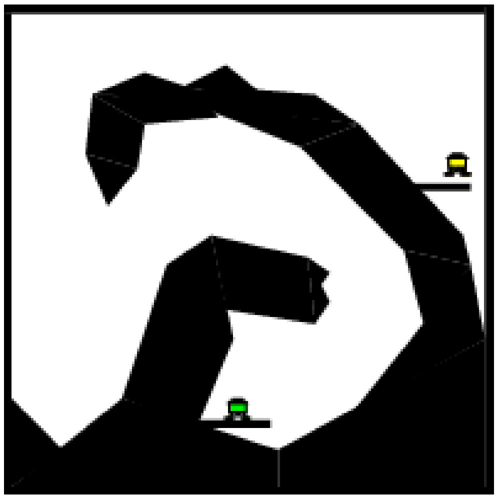


Figure 16

2. A Hovercraft that Translates in a Planar World:
   The task is to determine how to fire thrusters to navigate a hovercraft from an initial state to a goal state. The image below shows the RRTs (this is a projection from the 4D state space), and the computed trajectory. (Source :This result appears in Randomized Kinodynamic Planning by Steven M. LaValle and James J. Kuffner, Jr., International Journal of Robotics Research, 20(5):378--400, May 2001)

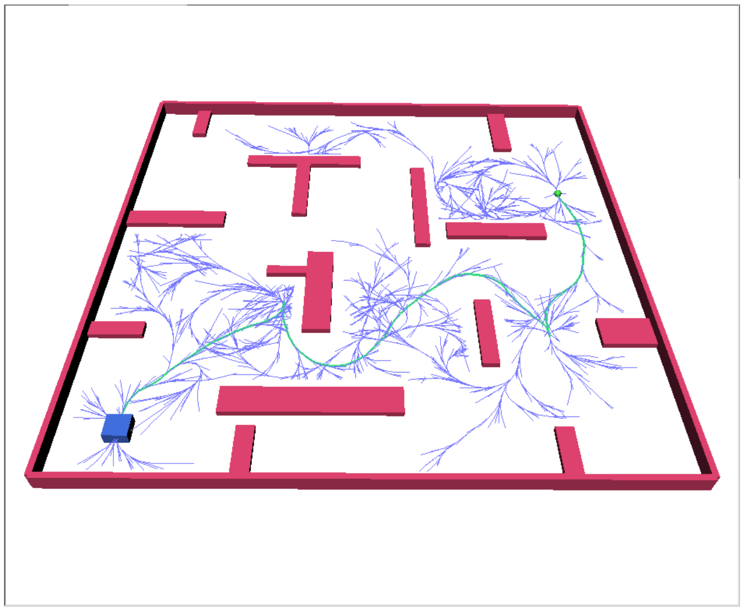


Figure 17

3. Motion Strategies with Moving Obstacles

   Below is a simple example that involves a spacecraft with four thrusters that navigates through a field of moving obstacles. The dynamics of the craft are also considered.

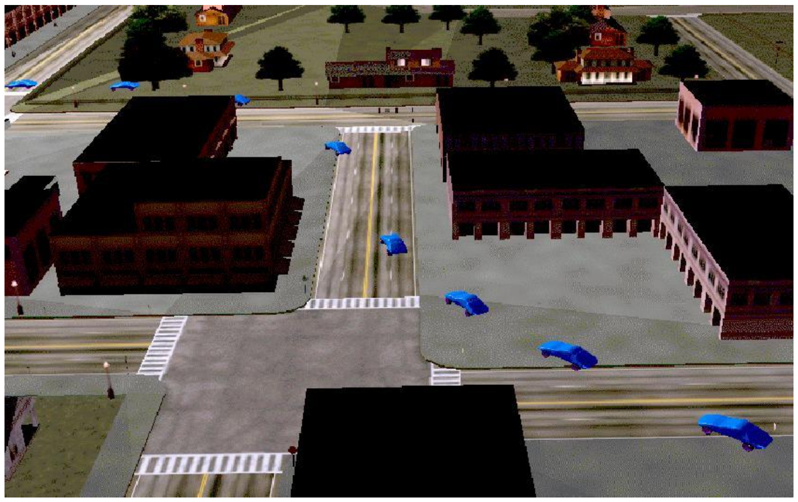


Figure 18

References:

1. LaValle, Steven M. (October 1998). "Rapidly-exploring random trees: A new tool for path planning”. Technical Report. Computer Science Department, Iowa State University (TR 98-11).
2. http://msl.cs.uiuc.edu/rrt/about.html About RRTs, by Steve LaValle
3. http://msl.cs.uiuc.edu/rrt/gallery.html Photo and Animation Gallery, by Steve LaValle
4. https://www.cs.cmu.edu/~motionplanning/lecture/lec20.pdf Robotic Motion Planning: RRT’s, RI 16-735, Howie Choset with slides from James Kuffner
5. Rapidly-Exploring Random Trees: Progress and Prospects (2000), by Steven M. LaValle, James J. Kuffner, Jr. Algorithmic and Computational Robotics: New Directions, http://eprints.kfupm.edu.sa/60786/1/60786.pdf