RRT

July 7, 2021

1 Rapidly exploring Random Trees

The premise of RRT is actually quite straight forward. Points are randomly generated and connected to the closest available node. Each time a vertex is created, a check must be made that the vertex lies outside of an obstacle. Furthermore, chaining the vertex to its closest neighbor must also avoid obstacles. The algorithm ends when a node is generated within the goal region, or a limit is hit.

[1]:
```
import matplotlib.pyplot as plt
import matplotlib.path as mpltPath
import math
import random
from shapely.geometry import Point, MultiPoint
from shapely.geometry import Polygon, MultiPolygon, LineString
```

1.1 RRT Pseudo Code

```
Qgoal #region that identifies success
Counter = 0 #keeps track of iterations
lim = n #number of iterations algorithm should run for
G(V,E) #Graph containing edges and vertices, initialized as empty

While counter < lim:

    Xnew  = RandomPosition()
    if IsInObstacle(Xnew) == True:
        continue
    Xnearest = Nearest(G(V,E),Xnew) //find nearest vertex
    Link = Chain(Xnew,Xnearest)
    G.append(Link)
    if Xnew in Qgoal:
        Return G

Return G
```

So let's get started on our implementation. The first thing we will define is the tree class.

1.2 Trees

A tree is a widely used abstract data type that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

[2]:
```
#tree structure definition
class Tree():

    def __init__(self, data = Point(0,0), children=None, par=None):
        self.data = data
        self.children = []
        if children is not None:
            for child in children:
                self.add_child(child)
        self.par = par

    def add_child(self, node):
        self.children.append(node)
        node.par = self

    def __str__(self, level=0):
        ret = "\t"*level+repr(self.data.x)+" "+repr(self.data.y)+"\n"
        for child in self.children:
            ret += child.__str__(level+1)
        return ret

    def __repr__(self):
        return '<tree node representation>'

    #to trace final path
    def tb(self, n):
        ax = []
        ay = []
        ax.append(n.data.x)
        ay.append(n.data.y)
        while n.data != self.data:
            n = n.par
            ax.append(n.data.x)
            ay.append(n.data.y)
        return ax,ay
```

1.3 Functions useful for implementation

We will first write the functions that will be used in our main RRT implementation code. They are as follows -

1. Define goal region and obstacles
2. Check if randomly generated point is in obstacle
3. Find distance between two points
4. Find nearest node to randomly generated point
5. Link the nearest node and current point

[3]:
```
#defining obstacles as a set of polygons
def obst(arr):
    ans = list()
    for coord in arr:
        m = Polygon(coord)
        ans.append(m)
    t = MultiPolygon(ans)
    return t

#defining goal region as polygon
def dgoal(coord):
    m = Polygon(coord)
    t = MultiPolygon([m])
    return t


#check if point is within polygon
def IsInObstacle(arr,pt):
    for i in arr.geoms:
        if i.contains(pt):
            return True
    return False
```

[4]:
```
#distance between two points
def distance(pt1, pt2):
    ans = math.sqrt((pt1.x-pt2.x)**2 + (pt1.y-pt2.y)**2)
    return ans

#finding nearest neighbour
def nearestNode(node, root, mind):
    d = distance(root.data, node.data)
    if d < mind:
        mind = d

    ans = root
    for i in root.children:
        d = nearestNode(node, i, mind)
        if d[0] < mind:
            mind = d[0]
            ans = d[1]
    return mind, ans
```

```
#linking new point to existing tree
def chain(node1, node2):
    node1.add_child(node2)
    return node2
```

1.4 Implementation of RRT

[5]:
```
def RRT(start,goal,obstacle_list):
    '''
    Your RRT code goes here

    The search space will be a rectangular space defined by
    (0,0),(0,10),(10,0),(10,10)

    Args:
        start(tuple):start point coordinates.
        goal (tuple):end point coordinates.
        obstacle_list (list): list of all obstacles in the envrionment.

    Returns:
        path: You are free to decide what data structure used here.
    '''
    goalr = [(goal[0]-0.3,goal[1]-0.3),(goal[0]+0.3,goal[1]-0.3), (goal[0]+0.
↪3,goal[1]+0.3), (goal[0]-0.3,goal[1]+0.3)]
    Qgoal = dgoal(goalr)

    obstacles = obst(obstacle_list)

    cnt = 0
    graph = Tree(Point(start[0],start[1])) #Graph containing edges and␣
↪vertices, initialized as empty

    while cnt<5000:

        x  = random.random()*10
        y  = random.random()*10
        p = Point(x,y)
        xnew = Tree(p)

        if IsInObstacle(obstacles,p):
            continue

        n = nearestNode(xnew,graph,10**6) #find nearest vertex
        line = LineString([n[1].data,p])
```

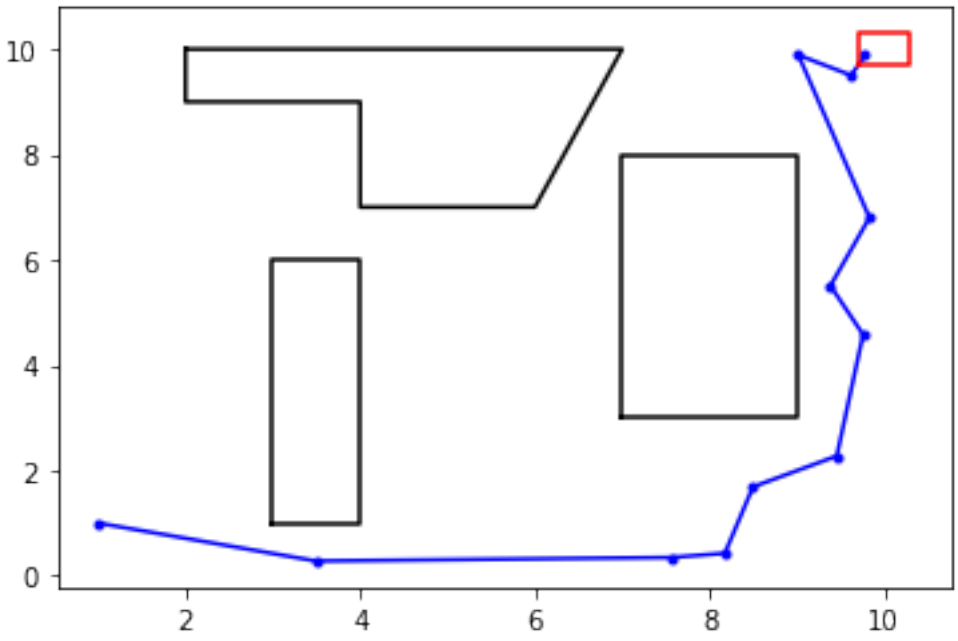
```
        if n[0]>=1:
            p = line.interpolate(1)
            x = p.x
            y = p.y
            line = LineString([n[1].data,p])

        if line.crosses(obstacles):
            continue

        last = chain(n[1],xnew)

        if IsInObstacle(Qgoal,p):
            print("Found it!")
            return graph.tb(last),Qgoal

        cnt+=1

    return graph.tb(last),Qgoal



def visualize(path,obstacle_list,Qgoal):

    '''
    The matplot code required to visulaize both the path and obstacles in
    the environment go here.

    Args:
        path: Same as used in RRT function.
        obstacle_list (list): list of all obstacles in the envrionment.

    Returns:
        None
    '''

    plt.figure()

    plt.plot(path[0],path[1],"b.-")

    m = obst(obstacle_list)
    for i in m:
        x,y = i.exterior.xy
        plt.plot(x,y,"black")

    for i in Qgoal:
        x,y = i.exterior.xy
        plt.plot(x,y,"red")
```

```
    plt.show()
```

Let's test our implementation. In the following cell I will provide my own start and goal region, as well as an obstacle set. The code will give an ouput of the whole region and the path it found.

[6]:
```
def test_rrt():

    obstacle_list = [
            [(2, 10), (7, 10), (6, 7), (4, 7), (4, 9), (2, 9)],
            [(3, 1), (3, 6), (4, 6), (4, 1)],
            [(7, 3), (7, 8), (9, 8), (9, 3)],
        ]

    start = (1, 1)
    goal = (10, 10)

    path = RRT(start,goal,obstacle_list)

    visualize(path[0],obstacle_list,path[1])

test_rrt()
```

```
Found it!
```



The benefit of the algorithm is its speed and implementation. Compared to other path planning

algorithms, RRT is fairly quick. The costliest part of the algorithm is finding its closest neighbor as this process grows depending on the number of vertices that have been generated.

1.5 RRT*

RRT* is an optimized version of RRT. When the number of nodes approaches infinity, the RRT* algorithm will deliver the shortest possible path to the goal. While realistically unfeasible, this statement suggests that the algorithm does work to develop a shortest path. The basic principle of RRT* is the same as RRT, but two key additions to the algorithm result in significantly different results.

First, RRT* records the distance each vertex has traveled relative to its parent vertex. This is referred to as the cost()of the vertex. After the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node are examined. If a node with a cheaper cost() than the proximal node is found, the cheaper node replaces the proximal node. The effect of this feature can be seen with the addition of fan shaped twigs in the tree structure. The cubic structure of RRT is eliminated.

The second difference RRT* adds is the rewiring of the tree. After a vertex has been connected to the cheapest neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. This feature makes the path more smooth.

RRT* Pseudo Code

```
Rad = r
G(V,E) //Graph containing edges and vertices

For itr in range(0…n)
    Xnew = RandomPosition()

    If Obstacle(Xnew) == True, try again

    Xnearest = Nearest(G(V,E),Xnew)
    Cost(Xnew) = Distance(Xnew,Xnearest)
    Xbest,Xneighbors = findNeighbors(G(V,E),Xnew,Rad)
    Link = Chain(Xnew,Xbest)

    For x' in Xneighbors
        If Cost(Xnew) + Distance(Xnew,x') < Cost(x')
            Cost(x') = Cost(Xnew)+Distance(Xnew,x')
            Parent(x') = Xnew
            G += {Xnew,x'}

    G += Link
Return G
```

[7]:
```
def findNeighbours(node, root, rad, parent):
    ans = []
    if distance(root.data, node.data)<rad:
```

```
        ans.append((root, parent))
    else:
        for i in root.children:
            ans += findNeighbours(node, i, rad,root)
#     print(ans)
    return ans
```

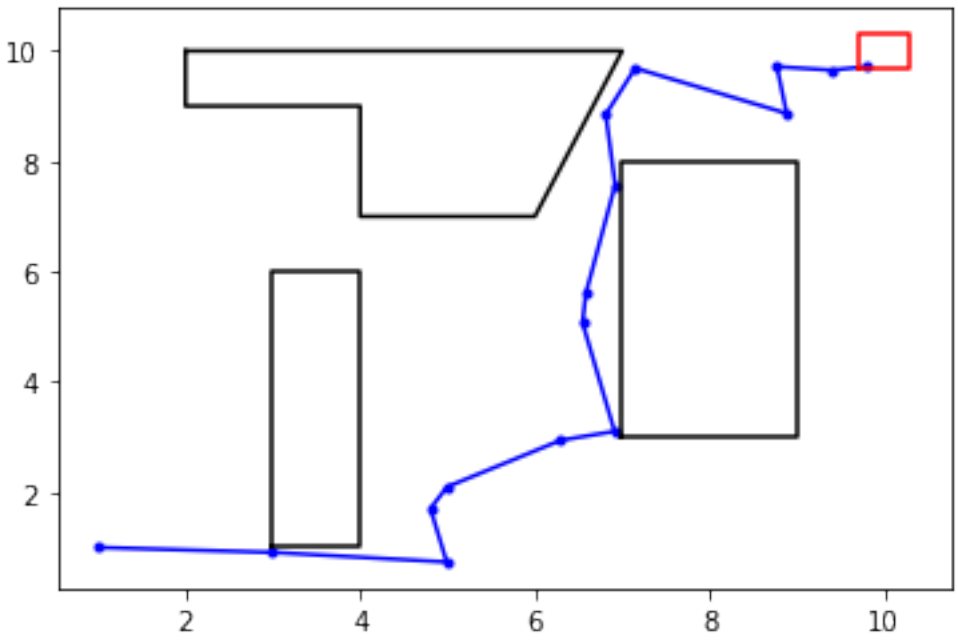
[12]:
```
def RRTstar(start,goal,obstacle_list):
    '''
    Your RRT code goes here

    The search space will be a rectangular space defined by
    (0,0),(0,10),(10,0),(10,10)

    Args:
        start(tuple):start point coordinates.
        goal (tuple):end point coordinates.
        obstacle_list (list): list of all obstacles in the envrionment.

    Returns:
        path: You are free to decide what data structure used here.
    '''
    goalr = [(goal[0]-0.3,goal[1]-0.3),(goal[0]+0.3,goal[1]-0.3), (goal[0]+0.
 ↪3,goal[1]+0.3), (goal[0]-0.3,goal[1]+0.3)]
    Qgoal = dgoal(goalr)

    obstacles = obst(obstacle_list)

    cnt = 0
    graph = Tree(Point(start[0],start[1])) #Graph containing edges and␣
 ↪vertices, initialized as empty
    cost = {}
    cost[graph] = 0

    while cnt<5000:

        x  = random.random()*10
        y  = random.random()*10
        p = Point(x,y)

        xnew = Tree(p)
#         print("Generated point",p)

        if IsInObstacle(obstacles,p):
#             print("Its in an obstacle")
            continue
```

```
        dist, xnearest = nearestNode(xnew, graph, 10**6) #find nearest vertex
#         print("Nearest node", xnearest)

        cost[xnew] = dist
#         print("Assigned cost",cost[xnew])

        line = LineString([xnearest.data,p])

        if cost[xnew] >= 2:
            p = line.interpolate(2)
            x = p.x
            y = p.y
            line = LineString([xnearest.data,p])
            xnew = Tree(Point(x,y))
            cost[xnew] = 2

        if line.crosses(obstacles):
#             print("Its line is in an obstacle")
            continue

        neighbours = findNeighbours(xnew, graph, 0.2, None)
#         print("The neighbours are", neighbours)

        for i,j in neighbours:

            if cost[xnew] + distance(xnew.data,i.data) < cost[i]:
                cost[i] = cost[xnew]+distance(xnew.data,i.data)
                line = LineString([p,i.data])

                if line.crosses(obstacles):
                    continue

                now = chain(xnew, i)
                if j:
                    j.children.remove(i)

#                 print("Connected",xnew.data,i.data)


        last = chain(xnearest,xnew)
#         print("Connected", xnearest.data, p)

        if IsInObstacle(Qgoal,p):
            print("Found it!")
            return graph.tb(last),Qgoal

        cnt+=1
```

```
#         print(cnt)

    return graph.tb(last),Qgoal
```

[13]:
```
def test_rrtstar():

    obstacle_list = [
            [(2, 10), (7, 10), (6, 7), (4, 7), (4, 9), (2, 9)],
            [(3, 1), (3, 6), (4, 6), (4, 1)],
            [(7, 3), (7, 8), (9, 8), (9, 3)],
        ]

    start = (1, 1)
    goal = (10, 10)

    path = RRTstar(start,goal,obstacle_list)

    visualize(path[0],obstacle_list,path[1])

test_rrtstar()
```

```
Found it!
```



1.5.1 References

Tim Chin, Robotic Path Planning: RRT and RRT*, Feb 14, 2019