<Name : Arpit Savarkar>

==========
README.md
==========
For Brevity, all the necesaary inputs can be changed in YAML files linked to each program setup for Workspace 1 (5 obstacles), Workspace 2 (9 obstacles) and Workspace 3 (2 obstacles).

Rather than taking input from the user the config files (yaml) can be updated to get the required setup which one needs to look into.

PRM
Execution Steps

- For Workspace 1:
    - python prm_WO1.py
    - Following Output will be seen :
        - **Config Files need to be updated for different results**
        - Enter 0 for Defualt Single setup or 1 for Benchmarking:

- For Workspace 2:
    - python prm_WO2.py
    - Following Output will be seen :
        - **Config Files need to be updated for different results**
        - Enter 0 for Defualt Single setup or 1 for Benchmarking:

- For Workspace 3:
    - python prm_WO3.py
    - Following Output will be seen :
        - **Config Files need to be updated for different results**
        - Enter 0 for Defualt Single setup or 1 for Benchmarking:

RRT
Execution Steps

- For Workspace 1:
    - python rrt_W01.py
    - Following Output will be seen :
        - **Config Files need to be updated for different results**
        - Enter 0 for Defualt Single setup or 1 for Benchmarking:

- For Workspace 2:
    - python rrt_W02.py
    - Following Output will be seen :
        - **Config Files need to be updated for different results**
        - Enter 0 for Defualt Single setup or 1 for Benchmarking:

- For Workspace 3:
    - python rrt_W03.py
    - Following Output will be seen :
        - **Config Files need to be updated for different results**
        - Enter 0 for Defualt Single setup or 1 for Benchmarking:
==========
prm_WO1.py
==========
```
import matplotlib.pyplot as plt
```

```
import numpy as np
from scipy import spatial
# https://www.geeksforgeeks.org/union-find/
from networkx.utils import UnionFind
from shapely.geometry import Point, LineString, Polygon, \
    MultiPolygon
import random
import copy
from graph import Graph
from workspaces import config
import yaml
import itertools
from timeit import default_timer as timer
import pandas as pd
import seaborn as sns
import os

WORKSPACE_CONFIG = config()

class PRM:

    def __init__(self, name = None, n = None, r = None, smoothing = None):
        with open(name, 'r') as stream:
            configData = yaml.load(stream, Loader=yaml.Loader)

        # self.n = configData['n']
        # self.r = configData['r']
        # self.usePathSmoothing = [False, True]
        self.n = n
        self.r = r
        self.usePathSmoothing = smoothing
        self.minBounds = configData['minBounds']
        self.maxBounds = configData['maxBounds']
        self.iterations = 1000

    def check_for_obs_collission(self, startState, goalState):

        start = tuple(startState.flatten())
        goal = tuple(goalState.flatten())

        line = LineString([start, goal])

        obstacles = WORKSPACE_CONFIG['WO1']

        collide_flag = False
        for obstacle in obstacles:
            if obstacle.intersects(line):

                collide_flag = True

        return collide_flag


    def checkConnectivity(self, data_structure, currLabel, nbrLabel):

        currComponent = data_structure[currLabel]
        newComponent  = data_structure[nbrLabel]

        flag = (currComponent != newComponent)
```

```
        return flag

    def admissible_heuristic_dist(self, point, dest, distNorm=2):
        p1 = np.reshape(point, (len(point), 1))
        p2 = np.reshape(dest, (len(dest), 1))

        distance = np.linalg.norm(p2 - p1, ord=distNorm)

        return distance

    def smoothPathInGraph(self, graph, path, goal_node_idx, pathLength,
                          shouldBenchmark):

        developed_path = copy.deepcopy(path)

        numEdgesToSmooth = round(len(developed_path) / 5)

        for i in range(0, numEdgesToSmooth):

            # only allow sampling from the middle of the path
            rNodes = tuple(self.replace_sample(developed_path[1:-1], 2))
            start_node_idx = rNodes[0]
            end_node_idx = rNodes[1]

            # skip the sampled nodes if they're already directly connected
            nodeBeforeEnd = graph.getter_helper(end_node_idx, 'prev')
            if nodeBeforeEnd == start_node_idx:
                continue

            # obtain the collision free samples
            startNodePos = graph.getter_helper(start_node_idx, 'pos')
            endNodePos = graph.getter_helper(end_node_idx, 'pos')

            collided = True
            itr = 0
            flag = False

            while collided and itr <= self.iterations:

                potentialSample = np.random.uniform(low=self.minBounds,
high=self.maxBounds, size=(1, len(self.minBounds)))
                potentialSample = potentialSample.flatten()

                collided = self.check_for_obs_collission(startNodePos,
potentialSample)\
                    or self.check_for_obs_collission(potentialSample, endNodePos)

                itr += 1

            if not collided:
                flag = True

            if not flag:
                continue

            # add the node to the PRM graph
            new_node_idx = goal_node_idx + i + 1
            goal_node_pos = graph.getter_helper(goal_node_idx, 'pos')
            updated_heuristic = self.admissible_heuristic_dist(potentialSample,
```

```
goal_node_pos)
            graph.add_node(new_node_idx,
                           heuristic=updated_heuristic,
                           prev=start_node_idx,
                           dist=0, priority=0, pos=potentialSample)

            # connect it to the graph
            graph.add_edge(start_node_idx, new_node_idx,
weight=self.admissible_heuristic_dist(startNodePos, potentialSample))

            graph.add_edge(new_node_idx, end_node_idx,
weight=self.admissible_heuristic_dist(potentialSample, endNodePos))

            # remove in-between nodes on the path
            currNode = end_node_idx
            prev_node = graph.getter_helper(currNode, 'prev')

            while prev_node != start_node_idx:

                prevPrevNode = graph.getter_helper(prev_node, 'prev')
                developed_path.remove(prev_node)

                # need to update prev now in order to continue proper traversal
                graph.setter_helper(currNode, 'prev', prevPrevNode)

                # now set the linked list pointers
                prev_node = graph.getter_helper(prev_node, 'prev')

            # now insert the new node into its place
            endNodeIDX = developed_path.index(end_node_idx)
            developed_path.insert(endNodeIDX, new_node_idx)
            graph.setter_helper(end_node_idx, 'prev', new_node_idx)

        # compute new path length
        newPathEdges = graph.getPathEdges(developed_path)
        newPathLength = 0
        for edge in newPathEdges:
            newPathLength += graph.edges[edge]['weight']

        # only return the smoothed path if its shorter
        if newPathLength > pathLength:

            if not shouldBenchmark:
                print('smoothing failed, using unsmoothed path')

            return path, pathLength

        else:

            return developed_path, newPathLength

    def replace_sample(self, seq, sampleSize):

        totalElems = len(seq)

        picksRemaining = sampleSize
        for elemsSeen, element in enumerate(seq):
            elemsRemaining = totalElems - elemsSeen
            prob = picksRemaining / elemsRemaining
```

```
            if random.random() < prob:
                yield element
                picksRemaining -= 1

    def computePRM(self, startState, goalState, n, r, usePathSmoothing,
                   shouldBenchmark):

        routes = []

        samples = np.random.uniform(low=self.minBounds, high=self.maxBounds,
                                    size=(n, len(self.minBounds)))

        # put them in a K-D tree to allow for easy connectivity queries
        kdTree = spatial.cKDTree(samples)

        # add all start, goal, and sampled nodes
        if not shouldBenchmark:
            print('Initializing PRM...')

        graph = Graph()


        # start_node_idx = n + 1
        startState = np.asarray(startState)
        goalState = np.asarray(goalState)
        graph.add_node(n+1,
                       heuristic=self.admissible_heuristic_dist(startState,
goalState),
                       prev=None, dist=0, priority=0, pos=startState.flatten())

        # goal_node_idx = n + 2
        graph.add_node(n+2,
                       heuristic=0, prev=None, dist=np.inf,
                       priority=np.inf, pos=goalState.flatten())

        # now initialize the sampled nodes of the underlying PRM graph
        for sample in range(0, n):

            pos = samples[sample, :]
            heuristic = self.admissible_heuristic_dist(pos, goalState)
            graph.add_node(sample,
                           heuristic=heuristic,
                           prev=None, dist=np.inf,
                           priority=np.inf, pos=pos.flatten())

        (graph, start_node_idx, goal_node_idx) =  (graph, n+1, n+2)

        # now connect all of the samples within radius r of each other
        if not shouldBenchmark:
            print('Connecting PRM...')

        # keep a union-find data structure to improve search performance by not
        # allowing cycles in the graph
        split_graph = UnionFind()

        for curr_node_index, curr_node_dat in list(graph.nodes(data=True)):

            curr_pos = curr_node_dat['pos']
```

```
            # search for all nodes in radius of the current node in question
            nbrs = kdTree.query_ball_point(curr_pos.flatten(),r)

            # adding all NEW edges that don't collide to the graph
            for nbrIndex in nbrs:

                gaol_xy = graph.getter_helper(nbrIndex, 'pos')

                collides = self.check_for_obs_collission(curr_pos, gaol_xy)
                check_comp = self.checkConnectivity(split_graph,
                                                    curr_node_index,
                                                    nbrIndex)

                if (not collides) and check_comp:

                    weight = self.admissible_heuristic_dist(curr_pos, gaol_xy)
                    graph.add_edge(curr_node_index, nbrIndex,
                                   weight=weight)

                    # need to update union-find data with the new edge
                    split_graph.union(curr_node_index, nbrIndex)

        if not shouldBenchmark:
            print('Finding path through PRM...')

        (shortestPath,
         pathLength, _) = graph.get_path(start_node_idx,
                                         goal_node_idx,
                                         algo='A star')
        foundPath = (shortestPath is not None)

        # only start smoothing if desired
        if foundPath and usePathSmoothing:

            if not shouldBenchmark:
                print('Smoothing path found through PRM...')

            (shortestPath,
             pathLength) = self.smoothPathInGraph(graph, shortestPath,
                                                  goal_node_idx, pathLength,
                                                  shouldBenchmark)

        # run robot through whole path
        if foundPath:
            for node in shortestPath:

                currPos = graph.getter_helper(node, 'pos')
                routes.append(currPos)
                # self.robot.updateRobotState(currPos)

        return (graph, shortestPath, pathLength, foundPath, routes)

    def findPathToGoal(self, startState, goalState, plannerConfigData,
                       plotConfigData, shouldBenchmark):

        # # allow the user to overide the settings in the config file
        plannerConfigData = None

        # n = self.n[0]
```

```
        # r = self.r[0]
        # usePathSmoothing = self.usePathSmoothing[0]
        n = self.n
        r = self.r
        usePathSmoothing = self.usePathSmoothing

        start = timer()
        (graph,
         shortestPath,
         pathLength, foundPath, routes) = self.computePRM(startState, goalState, n,
r,
                                                          usePathSmoothing,
                                                          shouldBenchmark)
        finish = timer()
        computationTime = finish - start

        # plot the resulting path over the PRM computation
        shouldPlot = plotConfigData['shouldPlot']

        if(pathLength == None):
            pathLength = 0
            computationTime = 0

        if shouldPlot:
            if not pathLength:
                pathLength = np.nan
            title = 'PRM - path length = %0.3g  n = %0.3g  r = %0.3g' \
                % (pathLength, n, r)
            plotConfigData['plotTitle'] += title
            self.plot(graph, startState, goalState, plotConfigData,
                      routes, path=shortestPath)

        # print("Path Length" , pathLength)

        return (computationTime, pathLength, foundPath)

    def plot(self, graph, startState, goalState,
             plotConfigData, routes, path=None):

        fig = plt.figure()
        ax = fig.add_subplot(111)

        # plot the graph and its shortest path
        fig, ax = graph.plot(path=path, fig=fig, showLabels=False,
                             showEdgeWeights=False)

        # unpack dictionary
        plotTitle = plotConfigData['plotTitle']
        xlabel = plotConfigData['xlabel']
        ylabel = plotConfigData['ylabel']
        shouldPlotCSpaceDiscretizationGrid = False
        shouldPlotObstacles = True


        # plot grid lines BEHIND the data
        ax.set_axisbelow(True)

        plt.grid()
```

```
        # plotting all the obstacles
        if shouldPlotObstacles:
            obstacles = WORKSPACE_CONFIG['WO1']
            for obst in obstacles:
                x,y = obst.exterior.xy
                ax.fill(x,y, alpha=0.5, fc='k',ec='none')

        # plotting the robot's motion
        # if robot is not None:
        robotPath = routes
        # robotPath = path

        # plotting the robot origin's path through cspace
        x = [state[0] for state in robotPath]
        y = [state[1] for state in robotPath]
        plt.plot(x, y, color='red', marker='*', linestyle='none',
                    linewidth=4, markersize=3,
                    label='Robot path')

        # plotting the start / end location of the robot
        plt.plot(startState[0], startState[1],
                 color='green', marker='o', linestyle='none',
                 linewidth=2, markersize=16,
                 label='Starting State')

        plt.plot(goalState[0], goalState[1],
                 color='red', marker='x', linestyle='none',
                 linewidth=4, markersize=16,
                 label='Goal State')

        ax.set_aspect('equal')
        plt.title(plotTitle)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        ax.axes.get_xaxis().set_visible(True)
        ax.axes.get_yaxis().set_visible(True)
        ax.set_xlim(self.minBounds[0], self.maxBounds[0])
        ax.set_ylim(self.minBounds[1], self.maxBounds[1])
        fig.legend(loc='upper left')


def savePlot(fig, shouldSavePlots, baseSaveFName, plotTitle,
             useTightLayout=True):
    print("Saving fig: ", plotTitle)

    if shouldSavePlots:
        saveFName = baseSaveFName + '-' + plotTitle + '.png'
        if useTightLayout:
            plt.tight_layout()
        plt.savefig(saveFName, dpi=500)

        print('wrote figure to: ', saveFName)
        # plt.show()
        plt.close(fig)

def plotStatistics(benchMarkingDF, pathValidityDF, benchParams, baseSaveFName,
plotTitle):

    print("Entering Plotting Stastics")
```

```
    ##
    # Plotting boxplots
    ##
    boxPlotsToMake = ['computationTimeInSeconds', 'pathLength']

    # need to create a new, merged categorical data for boxplots
    mergedParamsName = ', '.join(benchParams)
    benchMarkingDF[mergedParamsName] = benchMarkingDF[benchParams].apply(
        lambda x: ', '.join(x.astype(str)), axis=1)
    pathValidityDF[mergedParamsName] = pathValidityDF[
        benchParams].apply(lambda x: ', '.join(x.astype(str)), axis=1)

    # Usual boxplot for each variable that was benchmarked
    for plotVar in boxPlotsToMake:

        # make it wider for the insanse length of xticklabels
        fig = plt.figure(figsize=(20, 10))

        plt.style.use("seaborn-darkgrid")
        bp = sns.boxplot(data=benchMarkingDF,
                         x=mergedParamsName, y=plotVar)
        sns.swarmplot(x=mergedParamsName, y=plotVar, data=benchMarkingDF,
                      color="grey")

        # for readability of axis labels
        bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

        newPlotTitle = plotVar + '-' + plotTitle
        plt.title('Benchmarking of Sampled Planner ' + plotVar)
        savePlot(fig=fig, shouldSavePlots=True,
                 baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

    # number of times a valid path was found
    fig = plt.figure()

    plt.style.use('seaborn-darkgrid')
    bp = sns.barplot(x=mergedParamsName, y='numValidPaths',
                     data=pathValidityDF)
    plt.title('Number of Valid Paths Found for Each Parameter Combination')

    # for readability of axis labels
    bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

    newPlotTitle = 'numPaths' + '-' + plotTitle
    savePlot(fig=fig, shouldSavePlots=True,
                baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

def main():
    print(" Config Files need to be updated for different results ")
    val = input(" Enter 0 for Defualt Single setup or 1 for Benchmarking: ")
    val = int(val)

    if val == 0:
        name = 'prm_w01_backup.yaml'
        with open('prm_w01_backup.yaml', 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        # prm = PRM()
```

```
        numRunsOfPlannerPerSetting = configData['numRunsOfPlannerPerSetting']
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        print("Running Experimenents")
        print(experiments)

        for experiment in experiments:
            print("Currently Running Experiment")
            print(experiment)
            prm = None
            prm = PRM(name=name, n=experiment['n'], r=experiment['r'],
smoothing=experiment['smoothing'])
            plotConfigData = {'shouldPlot': True,
                              'plotTitle': '',
                              'xlabel': 'x',
                              'ylabel': 'y',
                              'plotObstacles': True,
                              'plotGrid': False}
            print(experiment)
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, numRunsOfPlannerPerSetting)):
                print(idx)
                (computationTime,
                pathLength,
                fp) = prm.findPathToGoal(startState=configData['startState'],
                                         goalState=configData['goalState'],
                                         plotConfigData=plotConfigData,
                                         plannerConfigData=experiment,
                                         shouldBenchmark=True)

        plt.show()

    elif val == 1:
        name = 'prm_w01.yaml'
        with open('prm_w01.yaml', 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        # prm = PRM()

        numRunsOfPlannerPerSetting = configData['numRunsOfPlannerPerSetting']
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []
```

```
        print("Running Experimenents")
        print(experiments)

        for experiment in experiments:
            print("Currently Running Experiment")
            print(experiment)
            prm = None
            prm = PRM(name=name, n=experiment['n'], r=experiment['r'],
smoothing=experiment['smoothing'])
            plotConfigData = {'shouldPlot': True,
                              'plotTitle': '',
                              'xlabel': 'x',
                              'ylabel': 'y',
                              'plotObstacles': True,
                              'plotGrid': False}
            print(experiment)
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, numRunsOfPlannerPerSetting)):
                print(idx)
                (computationTime,
                pathLength,
                fp) = prm.findPathToGoal(startState=configData['startState'],
                                         goalState=configData['goalState'],
                                         plotConfigData=plotConfigData,
                                         plannerConfigData=experiment,
                                         shouldBenchmark=True)

                # dat = None
                dat = {'computationTimeInSeconds': computationTime, 'pathLength':
pathLength}
                # bencmarkingInfo = None
                # fp = None
                bencmarkingInfo = {**dat, **experiment}
                # benchmarkingInfo = None
                # foundPath = None
                (benchmarkingInfo, foundPath) = (bencmarkingInfo, fp)
                print(foundPath)

                benchmarkingInfo.update(experiment)
                data.append(benchmarkingInfo)

                # print(foundPath)

                if foundPath:
                    numValidPaths += 1



            runInfo['numValidPaths'] = copy.deepcopy(numValidPaths)
            runInfo['numTimesRun'] = numRunsOfPlannerPerSetting
            runInfo.update(copy.deepcopy(experiment))
            pathValidityData.append(runInfo)

            print(runInfo)
```

```
        benchMarkingDF = pd.DataFrame(data)
        pathValidityDF = pd.DataFrame(pathValidityData)


benchMarkingDF.to_csv('/home/arpit/studies/motion/Assignment4/benchMarkingDF.csv',h
eader=True)

pathValidityDF.to_csv('/home/arpit/studies/motion/Assignment4/pathValidityDF.csv',h
eader=True)

        (benchMarkingDF, pathValidityDF, benchParams) = (benchMarkingDF,
pathValidityDF, parametersToVary)

        plotTitle = 'PRM' + '_stats'

        my_path = os.path.abspath(__file__) + '\plots'

        plotStatistics(benchMarkingDF=benchMarkingDF,
                       pathValidityDF=pathValidityDF,
                       benchParams=benchParams,
                       baseSaveFName=my_path,
                       plotTitle=plotTitle)



if __name__ == '__main__':
    main()
==================================
prm_WO2.py
==================================
import matplotlib.pyplot as plt
import numpy as np
from scipy import spatial
# https://www.geeksforgeeks.org/union-find/
from networkx.utils import UnionFind
from shapely.geometry import Point, LineString, Polygon, \
    MultiPolygon
import random
import copy
from graph import Graph
from workspaces import config
import yaml
import itertools
from timeit import default_timer as timer
import pandas as pd
import seaborn as sns
import os

WORKSPACE_CONFIG = config()

class PRM:

    def __init__(self, name = None, n = None, r = None, smoothing = None):
        with open(name, 'r') as stream:
            configData = yaml.load(stream, Loader=yaml.Loader)

        # self.n = configData['n']
        # self.r = configData['r']
```

```
    # self.usePathSmoothing = [False, True]
    self.n = n
    self.r = r
    self.usePathSmoothing = smoothing
    self.minBounds = configData['minBounds']
    self.maxBounds = configData['maxBounds']
    self.iterations = 1000

def check_for_obs_collission(self, startState, goalState):

    start = tuple(startState.flatten())
    goal = tuple(goalState.flatten())

    line = LineString([start, goal])

    obstacles = WORKSPACE_CONFIG['WO2']

    collide_flag = False
    for obstacle in obstacles:
        if obstacle.intersects(line):

            collide_flag = True

    return collide_flag


def checkConnectivity(self, data_structure, currLabel, nbrLabel):

    currComponent = data_structure[currLabel]
    newComponent  = data_structure[nbrLabel]

    flag = (currComponent != newComponent)
    return flag

def admissible_heuristic_dist(self, point, dest, distNorm=2):
    p1 = np.reshape(point, (len(point), 1))
    p2 = np.reshape(dest, (len(dest), 1))

    distance = np.linalg.norm(p2 - p1, ord=distNorm)

    return distance

def smoothPathInGraph(self, graph, path, goal_node_idx, pathLength,
                      shouldBenchmark):

    developed_path = copy.deepcopy(path)

    numEdgesToSmooth = round(len(developed_path) / 5)

    for i in range(0, numEdgesToSmooth):

        # only allow sampling from the middle of the path
        rNodes = tuple(self.replace_sample(developed_path[1:-1], 2))
        start_node_idx = rNodes[0]
        end_node_idx = rNodes[1]

        # skip the sampled nodes if they're already directly connected
        nodeBeforeEnd = graph.getter_helper(end_node_idx, 'prev')
        if nodeBeforeEnd == start_node_idx:
```

```
                continue

            # obtain the collision free samples
            startNodePos = graph.getter_helper(start_node_idx, 'pos')
            endNodePos = graph.getter_helper(end_node_idx, 'pos')

            collided = True
            itr = 0
            flag = False

            while collided and itr <= self.iterations:

                potentialSample = np.random.uniform(low=self.minBounds,
high=self.maxBounds, size=(1, len(self.minBounds)))
                potentialSample = potentialSample.flatten()

                collided = self.check_for_obs_collission(startNodePos,
potentialSample)\
                    or self.check_for_obs_collission(potentialSample, endNodePos)

                itr += 1

            if not collided:
                flag = True

            if not flag:
                continue

            # add the node to the PRM graph
            new_node_idx = goal_node_idx + i + 1
            goal_node_pos = graph.getter_helper(goal_node_idx, 'pos')
            updated_heuristic = self.admissible_heuristic_dist(potentialSample,
goal_node_pos)
            graph.add_node(new_node_idx,
                           heuristic=updated_heuristic,
                           prev=start_node_idx,
                           dist=0, priority=0, pos=potentialSample)

            # connect it to the graph
            graph.add_edge(start_node_idx, new_node_idx,
weight=self.admissible_heuristic_dist(startNodePos, potentialSample))

            graph.add_edge(new_node_idx, end_node_idx,
weight=self.admissible_heuristic_dist(potentialSample, endNodePos))

            # remove in-between nodes on the path
            currNode = end_node_idx
            prev_node = graph.getter_helper(currNode, 'prev')

            while prev_node != start_node_idx:

                prevPrevNode = graph.getter_helper(prev_node, 'prev')
                developed_path.remove(prev_node)

                # need to update prev now in order to continue proper traversal
                graph.setter_helper(currNode, 'prev', prevPrevNode)

                # now set the linked list pointers
                prev_node = graph.getter_helper(prev_node, 'prev')
```

```
            # now insert the new node into its place
            endNodeIDX = developed_path.index(end_node_idx)
            developed_path.insert(endNodeIDX, new_node_idx)
            graph.setter_helper(end_node_idx, 'prev', new_node_idx)

        # compute new path length
        newPathEdges = graph.getPathEdges(developed_path)
        newPathLength = 0
        for edge in newPathEdges:
            newPathLength += graph.edges[edge]['weight']

        # only return the smoothed path if its shorter
        if newPathLength > pathLength:

            if not shouldBenchmark:
                print('smoothing failed, using unsmoothed path')

            return path, pathLength

        else:

            return developed_path, newPathLength

    def replace_sample(self, seq, sampleSize):

        totalElems = len(seq)

        picksRemaining = sampleSize
        for elemsSeen, element in enumerate(seq):
            elemsRemaining = totalElems - elemsSeen
            prob = picksRemaining / elemsRemaining
            if random.random() < prob:
                yield element
                picksRemaining -= 1

    def computePRM(self, startState, goalState, n, r, usePathSmoothing,
                   shouldBenchmark):

        routes = []

        samples = np.random.uniform(low=self.minBounds, high=self.maxBounds,
                                    size=(n, len(self.minBounds)))

        # put them in a K-D tree to allow for easy connectivity queries
        kdTree = spatial.cKDTree(samples)

        # add all start, goal, and sampled nodes
        if not shouldBenchmark:
            print('Initializing PRM...')

        graph = Graph()


        # start_node_idx = n + 1
        startState = np.asarray(startState)
        goalState = np.asarray(goalState)
        graph.add_node(n+1,
                       heuristic=self.admissible_heuristic_dist(startState,
```

```
goalState),
                        prev=None, dist=0, priority=0, pos=startState.flatten())

        # goal_node_idx = n + 2
        graph.add_node(n+2,
                        heuristic=0, prev=None, dist=np.inf,
                        priority=np.inf, pos=goalState.flatten())

        # now initialize the sampled nodes of the underlying PRM graph
        for sample in range(0, n):

            pos = samples[sample, :]
            heuristic = self.admissible_heuristic_dist(pos, goalState)
            graph.add_node(sample,
                           heuristic=heuristic,
                           prev=None, dist=np.inf,
                           priority=np.inf, pos=pos.flatten())

        (graph, start_node_idx, goal_node_idx) =  (graph, n+1, n+2)

        # now connect all of the samples within radius r of each other
        if not shouldBenchmark:
            print('Connecting PRM...')

        # keep a union-find data structure to improve search performance by not
        # allowing cycles in the graph
        split_graph = UnionFind()

        for curr_node_index, curr_node_dat in list(graph.nodes(data=True)):

            curr_pos = curr_node_dat['pos']

            # search for all nodes in radius of the current node in question
            nbrs = kdTree.query_ball_point(curr_pos.flatten(),r)

            # adding all NEW edges that don't collide to the graph
            for nbrIndex in nbrs:

                gaol_xy = graph.getter_helper(nbrIndex, 'pos')

                collides = self.check_for_obs_collission(curr_pos, gaol_xy)
                check_comp = self.checkConnectivity(split_graph,
                                                    curr_node_index,
                                                    nbrIndex)

                if (not collides) and check_comp:

                    weight = self.admissible_heuristic_dist(curr_pos, gaol_xy)
                    graph.add_edge(curr_node_index, nbrIndex,
                                   weight=weight)

                    # need to update union-find data with the new edge
                    split_graph.union(curr_node_index, nbrIndex)

        if not shouldBenchmark:
            print('Finding path through PRM...')

        (shortestPath,
         pathLength, _) = graph.get_path(start_node_idx,
```

```
                                            goal_node_idx,
                                            algo='A star')
        foundPath = (shortestPath is not None)

        # only start smoothing if desired
        if foundPath and usePathSmoothing:

            if not shouldBenchmark:
                print('Smoothing path found through PRM...')

            (shortestPath,
             pathLength) = self.smoothPathInGraph(graph, shortestPath,
                                                  goal_node_idx, pathLength,
                                                  shouldBenchmark)

        # run robot through whole path
        if foundPath:
            for node in shortestPath:

                currPos = graph.getter_helper(node, 'pos')
                routes.append(currPos)
                # self.robot.updateRobotState(currPos)

        return (graph, shortestPath, pathLength, foundPath, routes)

    def findPathToGoal(self, startState, goalState, plannerConfigData,
                       plotConfigData, shouldBenchmark):

        # # allow the user to overide the settings in the config file
        plannerConfigData = None

        # n = self.n[0]
        # r = self.r[0]
        # usePathSmoothing = self.usePathSmoothing[0]
        n = self.n
        r = self.r
        usePathSmoothing = self.usePathSmoothing

        start = timer()
        (graph,
         shortestPath,
         pathLength, foundPath, routes) = self.computePRM(startState, goalState, n,
r,
                                                  usePathSmoothing,
                                                  shouldBenchmark)
        finish = timer()
        computationTime = finish - start

        # plot the resulting path over the PRM computation
        shouldPlot = plotConfigData['shouldPlot']

        if(pathLength == None):
            pathLength = 0
            computationTime = 0

        if shouldPlot:
            if not pathLength:
                pathLength = np.nan
            title = 'PRM - path length = %0.3g  n = %0.3g  r = %0.3g' \
```

```
            % (pathLength, n, r)
        plotConfigData['plotTitle'] += title
        self.plot(graph, startState, goalState, plotConfigData,
                  routes, path=shortestPath)

    # print("Path Length" , pathLength)

    return (computationTime, pathLength, foundPath)

def plot(self, graph, startState, goalState,
         plotConfigData, routes, path=None):

    fig = plt.figure()
    ax = fig.add_subplot(111)

    # plot the graph and its shortest path
    fig, ax = graph.plot(path=path, fig=fig, showLabels=False,
                         showEdgeWeights=False)

    # unpack dictionary
    plotTitle = plotConfigData['plotTitle']
    xlabel = plotConfigData['xlabel']
    ylabel = plotConfigData['ylabel']
    shouldPlotCSpaceDiscretizationGrid = False
    shouldPlotObstacles = True


    # plot grid lines BEHIND the data
    ax.set_axisbelow(True)

    plt.grid()

    # plotting all the obstacles
    if shouldPlotObstacles:
        obstacles = WORKSPACE_CONFIG['WO2']
        for obst in obstacles:
            x,y = obst.exterior.xy
            ax.fill(x,y, alpha=0.5, fc='k',ec='none')

    # plotting the robot's motion
    # if robot is not None:
    robotPath = routes
    # robotPath = path

    # plotting the robot origin's path through cspace
    x = [state[0] for state in robotPath]
    y = [state[1] for state in robotPath]
    plt.plot(x, y, color='red', marker='*', linestyle='none',
                linewidth=4, markersize=3,
                label='Robot path')

    # plotting the start / end location of the robot
    plt.plot(startState[0], startState[1],
             color='green', marker='o', linestyle='none',
             linewidth=2, markersize=16,
             label='Starting State')

    plt.plot(goalState[0], goalState[1],
             color='red', marker='x', linestyle='none',
```

```
                linewidth=4, markersize=16,
                label='Goal State')

        ax.set_aspect('equal')
        plt.title(plotTitle)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        ax.axes.get_xaxis().set_visible(True)
        ax.axes.get_yaxis().set_visible(True)
        ax.set_xlim(self.minBounds[0], self.maxBounds[0])
        ax.set_ylim(self.minBounds[1], self.maxBounds[1])
        fig.legend(loc='upper left')


def savePlot(fig, shouldSavePlots, baseSaveFName, plotTitle,
             useTightLayout=True):
    print("Saving fig: ", plotTitle)

    if shouldSavePlots:
        saveFName = baseSaveFName + '-' + plotTitle + '.png'
        if useTightLayout:
            plt.tight_layout()
        plt.savefig(saveFName, dpi=500)

        print('wrote figure to: ', saveFName)
        # plt.show()
        plt.close(fig)

def plotStatistics(benchMarkingDF, pathValidityDF, benchParams, baseSaveFName,
plotTitle):

    print("Entering Plotting Stastics")
    ##
    # Plotting boxplots
    ##
    boxPlotsToMake = ['computationTimeInSeconds', 'pathLength']

    # need to create a new, merged categorical data for boxplots
    mergedParamsName = ', '.join(benchParams)
    benchMarkingDF[mergedParamsName] = benchMarkingDF[benchParams].apply(
        lambda x: ', '.join(x.astype(str)), axis=1)
    pathValidityDF[mergedParamsName] = pathValidityDF[
        benchParams].apply(lambda x: ', '.join(x.astype(str)), axis=1)

    # Usual boxplot for each variable that was benchmarked
    for plotVar in boxPlotsToMake:

        # make it wider for the insanse length of xticklabels
        fig = plt.figure(figsize=(20, 10))

        plt.style.use("seaborn-darkgrid")
        bp = sns.boxplot(data=benchMarkingDF,
                         x=mergedParamsName, y=plotVar)
        sns.swarmplot(x=mergedParamsName, y=plotVar, data=benchMarkingDF,
                      color="grey")

        # for readability of axis labels
        bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')
```

```
        newPlotTitle = plotVar + '-' + plotTitle
        plt.title('Benchmarking of Sampled Planner ' + plotVar)
        savePlot(fig=fig, shouldSavePlots=True,
                    baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

    # number of times a valid path was found
    fig = plt.figure()

    plt.style.use('seaborn-darkgrid')
    bp = sns.barplot(x=mergedParamsName, y='numValidPaths',
                         data=pathValidityDF)
    plt.title('Number of Valid Paths Found for Each Parameter Combination')

    # for readability of axis labels
    bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

    newPlotTitle = 'numPaths' + '-' + plotTitle
    savePlot(fig=fig, shouldSavePlots=True,
                baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

def main():
    print(" Config Files need to be updated for different results ")
    val = input(" Enter 0 for Defualt Single setup or 1 for Benchmarking: ")
    val = int(val)

    if val == 0:
        name = 'prm_w02_backup.yaml'
        with open('prm_w02_backup.yaml', 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        # prm = PRM()

        numRunsOfPlannerPerSetting = configData['numRunsOfPlannerPerSetting']
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        print("Running Experimenents")
        print(experiments)

        for experiment in experiments:
            print("Currently Running Experiment")
            print(experiment)
            prm = None
            prm = PRM(name=name, n=experiment['n'], r=experiment['r'],
smoothing=experiment['smoothing'])
            plotConfigData = {'shouldPlot': True,
                              'plotTitle': '',
                              'xlabel': 'x',
                              'ylabel': 'y',
                              'plotObstacles': True,
                              'plotGrid': False}
            print(experiment)
```

```
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, numRunsOfPlannerPerSetting)):
                print(idx)
                (computationTime,
                pathLength,
                fp) = prm.findPathToGoal(startState=configData['startState'],
                                         goalState=configData['goalState'],
                                         plotConfigData=plotConfigData,
                                         plannerConfigData=experiment,
                                         shouldBenchmark=True)

        plt.show()

    elif val == 1:
        name = 'prm_w02.yaml'
        with open('prm_w02.yaml', 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        # prm = PRM()

        numRunsOfPlannerPerSetting = configData['numRunsOfPlannerPerSetting']
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        print("Running Experimenents")
        print(experiments)

        for experiment in experiments:
            print("Currently Running Experiment")
            print(experiment)
            prm = None
            prm = PRM(name=name, n=experiment['n'], r=experiment['r'],
smoothing=experiment['smoothing'])
            plotConfigData = {'shouldPlot': True,
                              'plotTitle': '',
                              'xlabel': 'x',
                              'ylabel': 'y',
                              'plotObstacles': True,
                              'plotGrid': False}
            print(experiment)
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, numRunsOfPlannerPerSetting)):
                print(idx)
                (computationTime,
                pathLength,
                fp) = prm.findPathToGoal(startState=configData['startState'],
                                         goalState=configData['goalState'],
                                         plotConfigData=plotConfigData,
```

```
                                            plannerConfigData=experiment,
                                            shouldBenchmark=True)

                # dat = None
                dat = {'computationTimeInSeconds': computationTime, 'pathLength':
pathLength}
                # bencmarkingInfo = None
                # fp = None
                bencmarkingInfo = {**dat, **experiment}
                # benchmarkingInfo = None
                # foundPath = None
                (benchmarkingInfo, foundPath) = (bencmarkingInfo, fp)
                print(foundPath)

                benchmarkingInfo.update(experiment)
                data.append(benchmarkingInfo)

                # print(foundPath)

                if foundPath:
                    numValidPaths += 1



            runInfo['numValidPaths'] = copy.deepcopy(numValidPaths)
            runInfo['numTimesRun'] = numRunsOfPlannerPerSetting
            runInfo.update(copy.deepcopy(experiment))
            pathValidityData.append(runInfo)

            print(runInfo)


        benchMarkingDF = pd.DataFrame(data)
        pathValidityDF = pd.DataFrame(pathValidityData)


benchMarkingDF.to_csv('/home/arpit/studies/motion/Assignment4/benchMarkingDF.csv',h
eader=True)

pathValidityDF.to_csv('/home/arpit/studies/motion/Assignment4/pathValidityDF.csv',h
eader=True)

        (benchMarkingDF, pathValidityDF, benchParams) = (benchMarkingDF,
pathValidityDF, parametersToVary)

        plotTitle = 'PRM' + '_stats'

        my_path = os.path.abspath(__file__) + '\plots'

        plotStatistics(benchMarkingDF=benchMarkingDF,
                       pathValidityDF=pathValidityDF,
                       benchParams=benchParams,
                       baseSaveFName=my_path,
                       plotTitle=plotTitle)



if __name__ == '__main__':
```

```
    main()

==================================
prm_WO3.py
==================================
import matplotlib.pyplot as plt
import numpy as np
from scipy import spatial
# https://www.geeksforgeeks.org/union-find/
from networkx.utils import UnionFind
from shapely.geometry import Point, LineString, Polygon, \
    MultiPolygon
import random
import copy
from graph import Graph
from workspaces import config
import yaml
import itertools
from timeit import default_timer as timer
import pandas as pd
import seaborn as sns
import os

WORKSPACE_CONFIG = config()

class PRM:

    def __init__(self, name = None, n = None, r = None, smoothing = None):
        with open(name, 'r') as stream:
            configData = yaml.load(stream, Loader=yaml.Loader)

        # self.n = configData['n']
        # self.r = configData['r']
        # self.usePathSmoothing = [False, True]
        self.n = n
        self.r = r
        self.usePathSmoothing = smoothing
        self.minBounds = configData['minBounds']
        self.maxBounds = configData['maxBounds']
        self.iterations = 1000

    def check_for_obs_collission(self, startState, goalState):

        start = tuple(startState.flatten())
        goal = tuple(goalState.flatten())

        line = LineString([start, goal])

        obstacles = WORKSPACE_CONFIG['WO3']

        collide_flag = False
        for obstacle in obstacles:
            if obstacle.intersects(line):

                collide_flag = True

        return collide_flag
```

```
    def checkConnectivity(self, data_structure, currLabel, nbrLabel):

        currComponent = data_structure[currLabel]
        newComponent  = data_structure[nbrLabel]

        flag = (currComponent != newComponent)
        return flag

    def admissible_heuristic_dist(self, point, dest, distNorm=2):
        p1 = np.reshape(point, (len(point), 1))
        p2 = np.reshape(dest, (len(dest), 1))

        distance = np.linalg.norm(p2 - p1, ord=distNorm)

        return distance

    def smoothPathInGraph(self, graph, path, goal_node_idx, pathLength,
                          shouldBenchmark):

        developed_path = copy.deepcopy(path)

        numEdgesToSmooth = round(len(developed_path) / 5)

        for i in range(0, numEdgesToSmooth):

            # only allow sampling from the middle of the path
            rNodes = tuple(self.replace_sample(developed_path[1:-1], 2))
            start_node_idx = rNodes[0]
            end_node_idx = rNodes[1]

            # skip the sampled nodes if they're already directly connected
            nodeBeforeEnd = graph.getter_helper(end_node_idx, 'prev')
            if nodeBeforeEnd == start_node_idx:
                continue

            # obtain the collision free samples
            startNodePos = graph.getter_helper(start_node_idx, 'pos')
            endNodePos = graph.getter_helper(end_node_idx, 'pos')

            collided = True
            itr = 0
            flag = False

            while collided and itr <= self.iterations:

                potentialSample = np.random.uniform(low=self.minBounds,
high=self.maxBounds, size=(1, len(self.minBounds)))
                potentialSample = potentialSample.flatten()

                collided = self.check_for_obs_collission(startNodePos,
potentialSample)\
                    or self.check_for_obs_collission(potentialSample, endNodePos)

                itr += 1

            if not collided:
                flag = True

            if not flag:
```

```
                continue

            # add the node to the PRM graph
            new_node_idx = goal_node_idx + i + 1
            goal_node_pos = graph.getter_helper(goal_node_idx, 'pos')
            updated_heuristic = self.admissible_heuristic_dist(potentialSample,
goal_node_pos)
            graph.add_node(new_node_idx,
                           heuristic=updated_heuristic,
                           prev=start_node_idx,
                           dist=0, priority=0, pos=potentialSample)

            # connect it to the graph
            graph.add_edge(start_node_idx, new_node_idx,
weight=self.admissible_heuristic_dist(startNodePos, potentialSample))

            graph.add_edge(new_node_idx, end_node_idx,
weight=self.admissible_heuristic_dist(potentialSample, endNodePos))

            # remove in-between nodes on the path
            currNode = end_node_idx
            prev_node = graph.getter_helper(currNode, 'prev')

            while prev_node != start_node_idx:

                prevPrevNode = graph.getter_helper(prev_node, 'prev')
                developed_path.remove(prev_node)

                # need to update prev now in order to continue proper traversal
                graph.setter_helper(currNode, 'prev', prevPrevNode)

                # now set the linked list pointers
                prev_node = graph.getter_helper(prev_node, 'prev')

            # now insert the new node into its place
            endNodeIDX = developed_path.index(end_node_idx)
            developed_path.insert(endNodeIDX, new_node_idx)
            graph.setter_helper(end_node_idx, 'prev', new_node_idx)

        # compute new path length
        newPathEdges = graph.getPathEdges(developed_path)
        newPathLength = 0
        for edge in newPathEdges:
            newPathLength += graph.edges[edge]['weight']

        # only return the smoothed path if its shorter
        if newPathLength > pathLength:

            if not shouldBenchmark:
                print('smoothing failed, using unsmoothed path')

            return path, pathLength

        else:

            return developed_path, newPathLength

    def replace_sample(self, seq, sampleSize):
```

```
        totalElems = len(seq)

        picksRemaining = sampleSize
        for elemsSeen, element in enumerate(seq):
            elemsRemaining = totalElems - elemsSeen
            prob = picksRemaining / elemsRemaining
            if random.random() < prob:
                yield element
                picksRemaining -= 1

    def computePRM(self, startState, goalState, n, r, usePathSmoothing,
                   shouldBenchmark):

        routes = []

        samples = np.random.uniform(low=self.minBounds, high=self.maxBounds,
                                    size=(n, len(self.minBounds)))

        # put them in a K-D tree to allow for easy connectivity queries
        kdTree = spatial.cKDTree(samples)

        # add all start, goal, and sampled nodes
        if not shouldBenchmark:
            print('Initializing PRM...')

        graph = Graph()


        # start_node_idx = n + 1
        startState = np.asarray(startState)
        goalState = np.asarray(goalState)
        graph.add_node(n+1,
                       heuristic=self.admissible_heuristic_dist(startState,
goalState),
                       prev=None, dist=0, priority=0, pos=startState.flatten())

        # goal_node_idx = n + 2
        graph.add_node(n+2,
                       heuristic=0, prev=None, dist=np.inf,
                       priority=np.inf, pos=goalState.flatten())

        # now initialize the sampled nodes of the underlying PRM graph
        for sample in range(0, n):

            pos = samples[sample, :]
            heuristic = self.admissible_heuristic_dist(pos, goalState)
            graph.add_node(sample,
                           heuristic=heuristic,
                           prev=None, dist=np.inf,
                           priority=np.inf, pos=pos.flatten())

        (graph, start_node_idx, goal_node_idx) =  (graph, n+1, n+2)

        # now connect all of the samples within radius r of each other
        if not shouldBenchmark:
            print('Connecting PRM...')

        # keep a union-find data structure to improve search performance by not
        # allowing cycles in the graph
```

```
    split_graph = UnionFind()

    for curr_node_index, curr_node_dat in list(graph.nodes(data=True)):

        curr_pos = curr_node_dat['pos']

        # search for all nodes in radius of the current node in question
        nbrs = kdTree.query_ball_point(curr_pos.flatten(),r)

        # adding all NEW edges that don't collide to the graph
        for nbrIndex in nbrs:

            gaol_xy = graph.getter_helper(nbrIndex, 'pos')

            collides = self.check_for_obs_collission(curr_pos, gaol_xy)
            check_comp = self.checkConnectivity(split_graph,
                                                curr_node_index,
                                                nbrIndex)

            if (not collides) and check_comp:

                weight = self.admissible_heuristic_dist(curr_pos, gaol_xy)
                graph.add_edge(curr_node_index, nbrIndex,
                               weight=weight)

                # need to update union-find data with the new edge
                split_graph.union(curr_node_index, nbrIndex)

    if not shouldBenchmark:
        print('Finding path through PRM...')

    (shortestPath,
     pathLength, _) = graph.get_path(start_node_idx,
                                     goal_node_idx,
                                     algo='A star')
    foundPath = (shortestPath is not None)

    # only start smoothing if desired
    if foundPath and usePathSmoothing:

        if not shouldBenchmark:
            print('Smoothing path found through PRM...')

        (shortestPath,
         pathLength) = self.smoothPathInGraph(graph, shortestPath,
                                              goal_node_idx, pathLength,
                                              shouldBenchmark)

    # run robot through whole path
    if foundPath:
        for node in shortestPath:

            currPos = graph.getter_helper(node, 'pos')
            routes.append(currPos)
            # self.robot.updateRobotState(currPos)

    return (graph, shortestPath, pathLength, foundPath, routes)

def findPathToGoal(self, startState, goalState, plannerConfigData,
```

```
                    plotConfigData, shouldBenchmark):

        # # allow the user to overide the settings in the config file
        plannerConfigData = None

        # n = self.n[0]
        # r = self.r[0]
        # usePathSmoothing = self.usePathSmoothing[0]
        n = self.n
        r = self.r
        usePathSmoothing = self.usePathSmoothing

        start = timer()
        (graph,
         shortestPath,
         pathLength, foundPath, routes) = self.computePRM(startState, goalState, n,
r,
                                                          usePathSmoothing,
                                                          shouldBenchmark)
        finish = timer()
        computationTime = finish - start

        # plot the resulting path over the PRM computation
        shouldPlot = plotConfigData['shouldPlot']

        if(pathLength == None):
            pathLength = 0
            computationTime = 0

        if shouldPlot:
            if not pathLength:
                pathLength = np.nan
            title = 'PRM - path length = %0.3g  n = %0.3g  r = %0.3g' \
                % (pathLength, n, r)
            plotConfigData['plotTitle'] += title
            self.plot(graph, startState, goalState, plotConfigData,
                      routes, path=shortestPath)

        # print("Path Length" , pathLength)

        return (computationTime, pathLength, foundPath)

    def plot(self, graph, startState, goalState,
             plotConfigData, routes, path=None):

        fig = plt.figure()
        ax = fig.add_subplot(111)

        # plot the graph and its shortest path
        fig, ax = graph.plot(path=path, fig=fig, showLabels=False,
                             showEdgeWeights=False)

        # unpack dictionary
        plotTitle = plotConfigData['plotTitle']
        xlabel = plotConfigData['xlabel']
        ylabel = plotConfigData['ylabel']
        shouldPlotCSpaceDiscretizationGrid = False
        shouldPlotObstacles = True
```

```
        # plot grid lines BEHIND the data
        ax.set_axisbelow(True)

        plt.grid()

        # plotting all the obstacles
        if shouldPlotObstacles:
            obstacles = WORKSPACE_CONFIG['WO3']
            for obst in obstacles:
                x,y = obst.exterior.xy
                ax.fill(x,y, alpha=0.5, fc='k',ec='none')

        # plotting the robot's motion
        # if robot is not None:
        robotPath = routes
        # robotPath = path

        # plotting the robot origin's path through cspace
        x = [state[0] for state in robotPath]
        y = [state[1] for state in robotPath]
        plt.plot(x, y, color='red', marker='*', linestyle='none',
                    linewidth=4, markersize=3,
                    label='Robot path')

        # plotting the start / end location of the robot
        plt.plot(startState[0], startState[1],
                 color='green', marker='o', linestyle='none',
                 linewidth=2, markersize=16,
                 label='Starting State')

        plt.plot(goalState[0], goalState[1],
                 color='red', marker='x', linestyle='none',
                 linewidth=4, markersize=16,
                 label='Goal State')

        ax.set_aspect('equal')
        plt.title(plotTitle)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        ax.axes.get_xaxis().set_visible(True)
        ax.axes.get_yaxis().set_visible(True)
        ax.set_xlim(self.minBounds[0], self.maxBounds[0])
        ax.set_ylim(self.minBounds[1], self.maxBounds[1])
        fig.legend(loc='upper left')


def savePlot(fig, shouldSavePlots, baseSaveFName, plotTitle,
             useTightLayout=True):
    print("Saving fig: ", plotTitle)

    if shouldSavePlots:
        saveFName = baseSaveFName + '-' + plotTitle + '.png'
        if useTightLayout:
            plt.tight_layout()
        plt.savefig(saveFName, dpi=500)

        print('wrote figure to: ', saveFName)
        # plt.show()
```

```
        plt.close(fig)

def plotStatistics(benchMarkingDF, pathValidityDF, benchParams, baseSaveFName,
plotTitle):

    print("Entering Plotting Stastics")
    ##
    # Plotting boxplots
    ##
    boxPlotsToMake = ['computationTimeInSeconds', 'pathLength']

    # need to create a new, merged categorical data for boxplots
    mergedParamsName = ', '.join(benchParams)
    benchMarkingDF[mergedParamsName] = benchMarkingDF[benchParams].apply(
        lambda x: ', '.join(x.astype(str)), axis=1)
    pathValidityDF[mergedParamsName] = pathValidityDF[
        benchParams].apply(lambda x: ', '.join(x.astype(str)), axis=1)

    # Usual boxplot for each variable that was benchmarked
    for plotVar in boxPlotsToMake:

        # make it wider for the insanse length of xticklabels
        fig = plt.figure(figsize=(20, 10))

        plt.style.use("seaborn-darkgrid")
        bp = sns.boxplot(data=benchMarkingDF,
                         x=mergedParamsName, y=plotVar)
        sns.swarmplot(x=mergedParamsName, y=plotVar, data=benchMarkingDF,
                      color="grey")

        # for readability of axis labels
        bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

        newPlotTitle = plotVar + '-' + plotTitle
        plt.title('Benchmarking of Sampled Planner ' + plotVar)
        savePlot(fig=fig, shouldSavePlots=True,
                 baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

    # number of times a valid path was found
    fig = plt.figure()

    plt.style.use('seaborn-darkgrid')
    bp = sns.barplot(x=mergedParamsName, y='numValidPaths',
                     data=pathValidityDF)
    plt.title('Number of Valid Paths Found for Each Parameter Combination')

    # for readability of axis labels
    bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

    newPlotTitle = 'numPaths' + '-' + plotTitle
    savePlot(fig=fig, shouldSavePlots=True,
             baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

def main():
    print(" Config Files need to be updated for different results ")
    val = input(" Enter 0 for Defualt Single setup or 1 for Benchmarking: ")
    val = int(val)

    if val == 0:
```

```
        name = 'prm_w03_backup.yaml'
        with open('prm_w03_backup.yaml', 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        # prm = PRM()

        numRunsOfPlannerPerSetting = configData['numRunsOfPlannerPerSetting']
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        print("Running Experimenents")
        print(experiments)

        for experiment in experiments:
            print("Currently Running Experiment")
            print(experiment)
            prm = None
            prm = PRM(name=name, n=experiment['n'], r=experiment['r'],
smoothing=experiment['smoothing'])
            plotConfigData = {'shouldPlot': True,
                              'plotTitle': '',
                              'xlabel': 'x',
                              'ylabel': 'y',
                              'plotObstacles': True,
                              'plotGrid': False}
            print(experiment)
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, numRunsOfPlannerPerSetting)):
                print(idx)
                (computationTime,
                pathLength,
                fp) = prm.findPathToGoal(startState=configData['startState'],
                                         goalState=configData['goalState'],
                                         plotConfigData=plotConfigData,
                                         plannerConfigData=experiment,
                                         shouldBenchmark=True)

        #         # dat = None
        #         dat = {'computationTimeInSeconds': computationTime, 'pathLength':
pathLength}
        #         # bencmarkingInfo = None
        #         # fp = None
        #         bencmarkingInfo = {**dat, **experiment}
        #         # benchmarkingInfo = None
        #         # foundPath = None
        #         (benchmarkingInfo, foundPath) = (bencmarkingInfo, fp)
        #         print(foundPath)

        #         benchmarkingInfo.update(experiment)
        #         data.append(benchmarkingInfo)
```

```
        #         # print(foundPath)

        #         if foundPath:
        #             numValidPaths += 1



        #     runInfo['numValidPaths'] = copy.deepcopy(numValidPaths)
        #     runInfo['numTimesRun'] = numRunsOfPlannerPerSetting
        #     runInfo.update(copy.deepcopy(experiment))
        #     pathValidityData.append(runInfo)

        #     print(runInfo)


        # benchMarkingDF = pd.DataFrame(data)
        # pathValidityDF = pd.DataFrame(pathValidityData)

        #
benchMarkingDF.to_csv('/home/arpit/studies/motion/Assignment4/benchMarkingDF.csv',h
eader=True)
        #
pathValidityDF.to_csv('/home/arpit/studies/motion/Assignment4/pathValidityDF.csv',h
eader=True)

        # (benchMarkingDF, pathValidityDF, benchParams) = (benchMarkingDF,
pathValidityDF, parametersToVary)

        # plotTitle = 'PRM' + '_stats'

        # my_path = os.path.abspath(__file__) + '\plots'

        # # plotStatistics(benchMarkingDF=benchMarkingDF,
        # #                 pathValidityDF=pathValidityDF,
        # #                 benchParams=benchParams,
        # #                 baseSaveFName=my_path,
        # #                 plotTitle=plotTitle)

        plt.show()

    elif val == 1:
        name = 'prm_w03.yaml'
        with open('prm_w03.yaml', 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        # prm = PRM()

        numRunsOfPlannerPerSetting = configData['numRunsOfPlannerPerSetting']
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []
```

```
        print("Running Experimenents")
        print(experiments)

        for experiment in experiments:
            print("Currently Running Experiment")
            print(experiment)
            prm = None
            prm = PRM(name=name, n=experiment['n'], r=experiment['r'],
smoothing=experiment['smoothing'])
            plotConfigData = {'shouldPlot': True,
                              'plotTitle': '',
                              'xlabel': 'x',
                              'ylabel': 'y',
                              'plotObstacles': True,
                              'plotGrid': False}
            print(experiment)
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, numRunsOfPlannerPerSetting)):
                print(idx)
                (computationTime,
                pathLength,
                fp) = prm.findPathToGoal(startState=configData['startState'],
                                         goalState=configData['goalState'],
                                         plotConfigData=plotConfigData,
                                         plannerConfigData=experiment,
                                         shouldBenchmark=True)

                # dat = None
                dat = {'computationTimeInSeconds': computationTime, 'pathLength':
pathLength}
                # bencmarkingInfo = None
                # fp = None
                bencmarkingInfo = {**dat, **experiment}
                # benchmarkingInfo = None
                # foundPath = None
                (benchmarkingInfo, foundPath) = (bencmarkingInfo, fp)
                print(foundPath)

                benchmarkingInfo.update(experiment)
                data.append(benchmarkingInfo)

                # print(foundPath)

                if foundPath:
                    numValidPaths += 1



            runInfo['numValidPaths'] = copy.deepcopy(numValidPaths)
            runInfo['numTimesRun'] = numRunsOfPlannerPerSetting
            runInfo.update(copy.deepcopy(experiment))
            pathValidityData.append(runInfo)

            print(runInfo)


        benchMarkingDF = pd.DataFrame(data)
```

```
        pathValidityDF = pd.DataFrame(pathValidityData)


benchMarkingDF.to_csv('/home/arpit/studies/motion/Assignment4/benchMarkingDF.csv',h
eader=True)

pathValidityDF.to_csv('/home/arpit/studies/motion/Assignment4/pathValidityDF.csv',h
eader=True)

        (benchMarkingDF, pathValidityDF, benchParams) = (benchMarkingDF,
pathValidityDF, parametersToVary)

        plotTitle = 'PRM' + '_stats'

        my_path = os.path.abspath(__file__) + '\plots'

        plotStatistics(benchMarkingDF=benchMarkingDF,
                       pathValidityDF=pathValidityDF,
                       benchParams=benchParams,
                       baseSaveFName=my_path,
                       plotTitle=plotTitle)



if __name__ == '__main__':
    main()

==================================
rrt_WO1.py
==================================
import math
import random

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.pyplot as plt
import numpy as np
from scipy import spatial
# https://www.geeksforgeeks.org/union-find/
from networkx.utils import UnionFind
from shapely.geometry import Point, LineString, Polygon, \
    MultiPolygon
import random
import copy
# from graph import Graph
from workspaces import config
import yaml
import itertools
from timeit import default_timer as timer
import pandas as pd
import seaborn as sns
import os
import random
import math

WORKSPACE_CONFIG = config()
```

```
show_animation = True


class RRT:
    """
    Class for RRT planning
    """

    class Node:
        """
        RRT Node
        """

        def __init__(self, x, y):
            self.x = x
            self.y = y
            self.path_x = []
            self.path_y = []
            self.parent = None

    def __init__(self):

        name = 'rrt_WO1.yaml'
        with open(name, 'r') as stream:
            configData = yaml.load(stream, Loader=yaml.Loader)

        self.startState = configData['startState']
        self.goalState = configData['goalState']
        self.minBounds = configData['minBounds']
        self.maxBounds = configData['maxBounds']

        self.start = self.Node(configData['startState'][0],
configData['startState'][1])
        self.end = self.Node(configData['goalState'][0], configData['goalState']
[1])
        self.radius = configData['radius']
        self.grid_size = configData['grid_size']
        self.goal_sample_rate = configData['goal_sample_rate']
        self.max_iter = configData['max_iter']
        self.graph = []

    def planning(self, animation=True):
        pathLength = 0
        computationTime = 0
        start = timer()
        path_generated = False
        self.graph = [self.start]
        for i in range(self.max_iter):
            sample = self.get_random_node()

            inter_node_dist_list = [(node.x - sample.x)**2 + (node.y - sample.y)**2
for node in self.graph]
            temp = inter_node_dist_list.index(min(inter_node_dist_list))
            next_node = self.graph[temp]

            upgraded_node = self.directional_growth(next_node, sample, self.radius)

            if self.check_collision(upgraded_node):
                self.graph.append(upgraded_node)
```

```
            if animation and i % 5 == 0:
                self.plotGraph(sample)

            if self.GAOLdist(self.graph[-1].x, self.graph[-1].y) <= self.radius:
                final_node = self.directional_growth(self.graph[-1], self.end,
self.radius)
                if self.check_collision(final_node):
                    _path = [[self.end.x, self.end.y]]
                    _node = self.graph[(len(self.graph) - 1)]
                    while _node.parent is not None:
                        _path.append([_node.x, _node.y])
                        _node = _node.parent
                    _path.append([_node.x, _node.y])
                    path = _path

            if animation and i % 5:
                self.plotGraph(sample)

        finish = timer()
        computationTime = finish - start

        if path is not None:
            start_x = 0
            start_y = 0
            pathLength = 0
            path_generated = True
            for (x, y) in path:
                dx, dy = (start_x - x), (start_y - y)
                d = math.hypot(dx, dy)
                start_x = x
                start_y = y
                pathLength += d

            return (computationTime, path, pathLength, True)

        return (None, None, None, False)  # cannot find path

    def directional_growth(self, src, dest, grow_len=float("inf")):

        upgraded_node = self.Node(src.x, src.y)
        dx = (dest.x - upgraded_node.x)
        dy = (dest.y - upgraded_node.y)
        euclid_dist = math.hypot(dx, dy)
        theta = math.atan2(dy, dx)

        upgraded_node.path_x, upgraded_node.path_y = [upgraded_node.x],
[upgraded_node.y]

        if grow_len > euclid_dist:
            grow_len = euclid_dist

        for _ in range(math.floor(grow_len / self.grid_size)):
            upgraded_node.x += self.grid_size * math.cos(theta)
            upgraded_node.y += self.grid_size * math.sin(theta)
            upgraded_node.path_x.append(upgraded_node.x)
            upgraded_node.path_y.append(upgraded_node.y)

        dx, dy = (dest.x - upgraded_node.x), (dest.y - upgraded_node.y)
```

```
        # dy = dest.y - upgraded_node.y
        euclid_dist = math.hypot(dx, dy)
        if euclid_dist <= self.grid_size:
            upgraded_node.path_x.append(dest.x)
            upgraded_node.path_y.append(dest.y)
            upgraded_node.x = dest.x
            upgraded_node.y = dest.y

        upgraded_node.parent = src

        return upgraded_node

    def GAOLdist(self, x, y):
        dx = x - self.end.x
        dy = y - self.end.y
        return math.hypot(dx, dy)

    def get_random_node(self):
        if random.randint(0, 100) > self.goal_sample_rate:
            rnd = self.Node(
                random.uniform(self.minBounds[0], self.maxBounds[0]),
                random.uniform(self.minBounds[1], self.maxBounds[1]))
        else:  # goal point sampling
            rnd = self.Node(self.end.x, self.end.y)
        return rnd

    def plotGraph(self, rnd=None):
        plt.clf()
        # for stopping simulation with the esc key.
        if rnd is not None:
            plt.plot(rnd.x, rnd.y, "^k")
        for node in self.graph:
            if node.parent:
                plt.plot(node.path_x, node.path_y, "-g")

        obstacles = WORKSPACE_CONFIG['WO1']
        for obst in obstacles:
            x,y = obst.exterior.xy
            plt.fill(x,y, alpha=0.5, fc='c',ec='none')

        plt.plot(self.startState[0], self.startState[1], "xr")
        plt.plot(self.goalState[0], self.goalState[1], "xr")
        plt.axis("equal")
        plt.axis([self.minBounds[0], self.maxBounds[0], self.minBounds[1],
self.maxBounds[1]])
        plt.grid(True)
        # plt.pause(0.01)

    def check_collision(self, node):

        obstacles = WORKSPACE_CONFIG['WO1']
        p = Point(node.x, node.y)
        collide_flag = False
        for obstacle in obstacles:
            if p.within(obstacle):

                collide_flag = True
                break
```

```
        if collide_flag:
            return False
        else:
            return True


def savePlot(fig, shouldSavePlots, baseSaveFName, plotTitle,
             useTightLayout=True):
    print("Saving fig: ", plotTitle)

    if shouldSavePlots:
        saveFName = baseSaveFName + '-' + plotTitle + '.png'
        if useTightLayout:
            plt.tight_layout()
        plt.savefig(saveFName, dpi=500)

        print('wrote figure to: ', saveFName)
        # plt.show()
        plt.close(fig)

def plotStatistics(benchMarkingDF, pathValidityDF, benchParams, baseSaveFName,
plotTitle):

    print("Entering Plotting Stastics")
    ##
    # Plotting boxplots
    ##
    boxPlotsToMake = ['computationTimeInSeconds', 'pathLength']

    # need to create a new, merged categorical data for boxplots
    mergedParamsName = ', '.join(benchParams)
    benchMarkingDF[mergedParamsName] = benchMarkingDF[benchParams].apply(
        lambda x: ', '.join(x.astype(str)), axis=1)
    pathValidityDF[mergedParamsName] = pathValidityDF[
        benchParams].apply(lambda x: ', '.join(x.astype(str)), axis=1)

    # Usual boxplot for each variable that was benchmarked
    for plotVar in boxPlotsToMake:

        # make it wider for the insanse length of xticklabels
        fig = plt.figure(figsize=(20, 10))

        plt.style.use("seaborn-darkgrid")
        bp = sns.boxplot(data=benchMarkingDF,
                         x=mergedParamsName, y=plotVar)
        sns.swarmplot(x=mergedParamsName, y=plotVar, data=benchMarkingDF,
                      color="grey")

        # for readability of axis labels
        bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

        newPlotTitle = plotVar + '-' + plotTitle
        plt.title('Benchmarking of Sampled Planner ' + plotVar)
        savePlot(fig=fig, shouldSavePlots=True,
                 baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

    # number of times a valid path was found
    fig = plt.figure()
```

```
    plt.style.use('seaborn-darkgrid')
    bp = sns.barplot(x=mergedParamsName, y='numValidPaths',
                     data=pathValidityDF)
    plt.title('Number of Valid Paths Found for Each Parameter Combination')

    # for readability of axis labels
    bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

    newPlotTitle = 'numPaths' + '-' + plotTitle
    savePlot(fig=fig, shouldSavePlots=True,
                baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)


def main():

    val = input("Enter 0 to get a single run, Enter 1 for Benchmarking Plot: ")
    val = int(val)

    if val == 0:

        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.set_axisbelow(True)

        name = 'rrt_WO1.yaml'
        with open(name, 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        minBounds = configData['minBounds']
        maxBounds = configData['maxBounds']

        distance = 0

        numRunsOfPlannerPerSetting = 100
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        plotConfigData = {'shouldPlot': True,
                          'plotTitle': '',
                          'xlabel': 'x',
                          'ylabel': 'y',
                          'plotObstacles': True,
                          'plotGrid': False}

        rrt = RRT()

        (computationTime, path, pathLength, path_generated) =
rrt.planning(animation=False)

        if path is None:
            print("Algorithm convergence failed in the specified number of
iterations")
```

```
    else:
        print("You Bet: GOT A PATH")
        # # Draw final path
        if show_animation:
            rrt.plotGraph()
            plt.plot([x for (x, y) in path], [y for (x, y) in path], '-r')
            plt.grid(True)

    start_x = 0
    start_y = 0
    pathLength = 0

    for (x, y) in path:
        dx = start_x - x
        dy = start_y - y
        d = math.hypot(dx, dy)
        start_x = x
        start_y = y
        pathLength += d

    r = 0.5

    plotTitle = 'RRT - path length = %0.3g r = %0.3g' % (pathLength, r)

    ax.set_aspect('equal')
    plt.title(plotTitle)
    ax.axes.get_xaxis().set_visible(True)
    ax.axes.get_yaxis().set_visible(True)
    ax.set_xlim(minBounds[0], maxBounds[0])
    ax.set_ylim(minBounds[1], maxBounds[1])
    fig.legend(loc='upper left')

    plt.show()

if val == 1:
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_axisbelow(True)

    name = 'rrt_WO1.yaml'
    with open(name, 'r') as stream:
            configData = yaml.load(stream, Loader=yaml.Loader)

    minBounds = configData['minBounds']
    maxBounds = configData['maxBounds']

    distance = 0

    numRunsOfPlannerPerSetting = 100
    parametersToVary = configData['paramterNamesToVary']
    allParams = dict((var, configData[var]) for var in parametersToVary)

    keys, values = zip(*allParams.items())
    experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
    print(experiments)
    data = []
    pathValidityData = []

    plotConfigData = {'shouldPlot': True,
```

```
                            'plotTitle': '',
                            'xlabel': 'x',
                            'ylabel': 'y',
                            'plotObstacles': True,
                            'plotGrid': False}

        for experiment in experiments:
            rrt = None
            rrt = RRT()
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, 100)):
                (computationTime, path, pathLength, path_generated) =
rrt.planning(animation=False)

                dat = {'computationTimeInSeconds': computationTime, 'pathLength':
pathLength}
                bencmarkingInfo = {**dat, **experiment}
                (benchmarkingInfo, path_generated) = (bencmarkingInfo,
path_generated)

                benchmarkingInfo.update(experiment)
                data.append(benchmarkingInfo)

                if path_generated:
                    numValidPaths += 1


            runInfo['numValidPaths'] = copy.deepcopy(numValidPaths)
            runInfo['numTimesRun'] = numRunsOfPlannerPerSetting
            runInfo.update(copy.deepcopy(experiment))
            pathValidityData.append(runInfo)

            print(runInfo)


        benchMarkingDF = pd.DataFrame(data)
        pathValidityDF = pd.DataFrame(pathValidityData)


benchMarkingDF.to_csv('/home/arpit/studies/motion/Assignment4/benchMarkingDF.csv',h
eader=True)

pathValidityDF.to_csv('/home/arpit/studies/motion/Assignment4/pathValidityDF.csv',h
eader=True)

        (benchMarkingDF, pathValidityDF, benchParams) = (benchMarkingDF,
pathValidityDF, parametersToVary)

        plotTitle = 'PRM' + '_stats'

        my_path = os.path.abspath(__file__) + '\plots'

        plotStatistics(benchMarkingDF=benchMarkingDF,
                       pathValidityDF=pathValidityDF,
                       benchParams=benchParams,
                       baseSaveFName=my_path,
                       plotTitle=plotTitle)
```

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

==================================
rrt_WO2.py
==================================
import math
import random

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.pyplot as plt
import numpy as np
from scipy import spatial
# https://www.geeksforgeeks.org/union-find/
from networkx.utils import UnionFind
from shapely.geometry import Point, LineString, Polygon, \
    MultiPolygon
import random
import copy
# from graph import Graph
from workspaces import config
import yaml
import itertools
from timeit import default_timer as timer
import pandas as pd
import seaborn as sns
import os
import random
import math

WORKSPACE_CONFIG = config()

show_animation = True


class RRT:
    """
    Class for RRT planning
    """

    class Node:
        """
        RRT Node
        """

        def __init__(self, x, y):
            self.x = x
            self.y = y
            self.path_x = []
            self.path_y = []
            self.parent = None

    def __init__(self):

        name = 'rrt_WO2.yaml'
```

```
        with open(name, 'r') as stream:
            configData = yaml.load(stream, Loader=yaml.Loader)

        self.startState = configData['startState']
        self.goalState = configData['goalState']
        self.minBounds = configData['minBounds']
        self.maxBounds = configData['maxBounds']

        self.start = self.Node(configData['startState'][0],
configData['startState'][1])
        self.end = self.Node(configData['goalState'][0], configData['goalState']
[1])
        self.radius = configData['radius']
        self.grid_size = configData['grid_size']
        self.goal_sample_rate = configData['goal_sample_rate']
        self.max_iter = configData['max_iter']
        self.graph = []

    def planning(self, animation=True):
        pathLength = 0
        computationTime = 0
        start = timer()
        path_generated = False
        self.graph = [self.start]
        for i in range(self.max_iter):
            sample = self.get_random_node()

            inter_node_dist_list = [(node.x - sample.x)**2 + (node.y - sample.y)**2
for node in self.graph]
            temp = inter_node_dist_list.index(min(inter_node_dist_list))
            next_node = self.graph[temp]

            upgraded_node = self.directional_growth(next_node, sample, self.radius)

            if self.check_collision(upgraded_node):
                self.graph.append(upgraded_node)

            if animation and i % 5 == 0:
                self.plotGraph(sample)

            if self.GAOLdist(self.graph[-1].x, self.graph[-1].y) <= self.radius:
                final_node = self.directional_growth(self.graph[-1], self.end,
self.radius)
                if self.check_collision(final_node):
                    _path = [[self.end.x, self.end.y]]
                    _node = self.graph[(len(self.graph) - 1)]
                    while _node.parent is not None:
                        _path.append([_node.x, _node.y])
                        _node = _node.parent
                    _path.append([_node.x, _node.y])
                    path = _path

            if animation and i % 5:
                self.plotGraph(sample)

        finish = timer()
        computationTime = finish - start

        if path is not None:
```

```
            start_x = 0
            start_y = 0
            pathLength = 0
            path_generated = True
            for (x, y) in path:
                dx, dy = (start_x - x), (start_y - y)
                d = math.hypot(dx, dy)
                start_x = x
                start_y = y
                pathLength += d

            return (computationTime, path, pathLength, True)

        return (None, None, None, False)  # cannot find path

    def directional_growth(self, src, dest, grow_len=float("inf")):

        upgraded_node = self.Node(src.x, src.y)
        dx = (dest.x - upgraded_node.x)
        dy = (dest.y - upgraded_node.y)
        euclid_dist = math.hypot(dx, dy)
        theta = math.atan2(dy, dx)

        upgraded_node.path_x, upgraded_node.path_y = [upgraded_node.x],
[upgraded_node.y]

        if grow_len > euclid_dist:
            grow_len = euclid_dist

        for _ in range(math.floor(grow_len / self.grid_size)):
            upgraded_node.x += self.grid_size * math.cos(theta)
            upgraded_node.y += self.grid_size * math.sin(theta)
            upgraded_node.path_x.append(upgraded_node.x)
            upgraded_node.path_y.append(upgraded_node.y)

        dx, dy = (dest.x - upgraded_node.x), (dest.y - upgraded_node.y)
        # dy = dest.y - upgraded_node.y
        euclid_dist = math.hypot(dx, dy)
        if euclid_dist <= self.grid_size:
            upgraded_node.path_x.append(dest.x)
            upgraded_node.path_y.append(dest.y)
            upgraded_node.x = dest.x
            upgraded_node.y = dest.y

        upgraded_node.parent = src

        return upgraded_node

    def GAOLdist(self, x, y):
        dx = x - self.end.x
        dy = y - self.end.y
        return math.hypot(dx, dy)

    def get_random_node(self):
        if random.randint(0, 100) > self.goal_sample_rate:
            rnd = self.Node(
                random.uniform(self.minBounds[0], self.maxBounds[0]),
                random.uniform(self.minBounds[1], self.maxBounds[1]))
        else:  # goal point sampling
```

```
            rnd = self.Node(self.end.x, self.end.y)
        return rnd

    def plotGraph(self, rnd=None):
        plt.clf()
        # for stopping simulation with the esc key.
        if rnd is not None:
            plt.plot(rnd.x, rnd.y, "^k")
        for node in self.graph:
            if node.parent:
                plt.plot(node.path_x, node.path_y, "-g")

        obstacles = WORKSPACE_CONFIG['WO2']
        for obst in obstacles:
            x,y = obst.exterior.xy
            plt.fill(x,y, alpha=0.5, fc='c',ec='none')

        plt.plot(self.startState[0], self.startState[1], "xr")
        plt.plot(self.goalState[0], self.goalState[1], "xr")
        plt.axis("equal")
        plt.axis([self.minBounds[0], self.maxBounds[0], self.minBounds[1],
self.maxBounds[1]])
        plt.grid(True)
        # plt.pause(0.01)

    def check_collision(self, node):

        obstacles = WORKSPACE_CONFIG['WO2']
        p = Point(node.x, node.y)
        collide_flag = False
        for obstacle in obstacles:
            if p.within(obstacle):

                collide_flag = True
                break

        if collide_flag:
            return False
        else:
            return True


def savePlot(fig, shouldSavePlots, baseSaveFName, plotTitle,
             useTightLayout=True):
    print("Saving fig: ", plotTitle)

    if shouldSavePlots:
        saveFName = baseSaveFName + '-' + plotTitle + '.png'
        if useTightLayout:
            plt.tight_layout()
        plt.savefig(saveFName, dpi=500)

        print('wrote figure to: ', saveFName)
        # plt.show()
        plt.close(fig)

def plotStatistics(benchMarkingDF, pathValidityDF, benchParams, baseSaveFName,
plotTitle):
```

```
    print("Entering Plotting Stastics")
    ##
    # Plotting boxplots
    ##
    boxPlotsToMake = ['computationTimeInSeconds', 'pathLength']

    # need to create a new, merged categorical data for boxplots
    mergedParamsName = ', '.join(benchParams)
    benchMarkingDF[mergedParamsName] = benchMarkingDF[benchParams].apply(
        lambda x: ', '.join(x.astype(str)), axis=1)
    pathValidityDF[mergedParamsName] = pathValidityDF[
        benchParams].apply(lambda x: ', '.join(x.astype(str)), axis=1)

    # Usual boxplot for each variable that was benchmarked
    for plotVar in boxPlotsToMake:

        # make it wider for the insanse length of xticklabels
        fig = plt.figure(figsize=(20, 10))

        plt.style.use("seaborn-darkgrid")
        bp = sns.boxplot(data=benchMarkingDF,
                         x=mergedParamsName, y=plotVar)
        sns.swarmplot(x=mergedParamsName, y=plotVar, data=benchMarkingDF,
                      color="grey")

        # for readability of axis labels
        bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

        newPlotTitle = plotVar + '-' + plotTitle
        plt.title('Benchmarking of Sampled Planner ' + plotVar)
        savePlot(fig=fig, shouldSavePlots=True,
                 baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

    # number of times a valid path was found
    fig = plt.figure()

    plt.style.use('seaborn-darkgrid')
    bp = sns.barplot(x=mergedParamsName, y='numValidPaths',
                     data=pathValidityDF)
    plt.title('Number of Valid Paths Found for Each Parameter Combination')

    # for readability of axis labels
    bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

    newPlotTitle = 'numPaths' + '-' + plotTitle
    savePlot(fig=fig, shouldSavePlots=True,
                baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)


def main():

    val = input("Enter 0 to get a single run, Enter 1 for Benchmarking Plot: ")
    val = int(val)

    if val == 0:

        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.set_axisbelow(True)
```

```
        name = 'rrt_WO2.yaml'
        with open(name, 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        minBounds = configData['minBounds']
        maxBounds = configData['maxBounds']

        distance = 0

        numRunsOfPlannerPerSetting = 100
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        plotConfigData = {'shouldPlot': True,
                          'plotTitle': '',
                          'xlabel': 'x',
                          'ylabel': 'y',
                          'plotObstacles': True,
                          'plotGrid': False}

        rrt = RRT()

        (computationTime, path, pathLength, path_generated) =
rrt.planning(animation=False)

        if path is None:
            print("Algorithm convergence failed in the specified number of
iterations")
        else:
            print("You Bet: GOT A PATH")
            # # Draw final path
            if show_animation:
                rrt.plotGraph()
                plt.plot([x for (x, y) in path], [y for (x, y) in path], '-r')
                plt.grid(True)

        start_x = 0
        start_y = 0
        pathLength = 0

        for (x, y) in path:
            dx = start_x - x
            dy = start_y - y
            d = math.hypot(dx, dy)
            start_x = x
            start_y = y
            pathLength += d

        r = 0.5

        plotTitle = 'RRT - path length = %0.3g r = %0.3g' % (pathLength, r)
```

```
        ax.set_aspect('equal')
        plt.title(plotTitle)
        ax.axes.get_xaxis().set_visible(True)
        ax.axes.get_yaxis().set_visible(True)
        ax.set_xlim(minBounds[0], maxBounds[0])
        ax.set_ylim(minBounds[1], maxBounds[1])
        fig.legend(loc='upper left')

        plt.show()

    if val == 1:
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.set_axisbelow(True)

        name = 'rrt_WO2.yaml'
        with open(name, 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        minBounds = configData['minBounds']
        maxBounds = configData['maxBounds']

        distance = 0

        numRunsOfPlannerPerSetting = 100
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        plotConfigData = {'shouldPlot': True,
                          'plotTitle': '',
                          'xlabel': 'x',
                          'ylabel': 'y',
                          'plotObstacles': True,
                          'plotGrid': False}

        for experiment in experiments:
            rrt = None
            rrt = RRT()
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, 100)):
                (computationTime, path, pathLength, path_generated) =
rrt.planning(animation=False)

                dat = {'computationTimeInSeconds': computationTime, 'pathLength':
pathLength}
                bencmarkingInfo = {**dat, **experiment}
                (benchmarkingInfo, path_generated) = (bencmarkingInfo,
path_generated)

                benchmarkingInfo.update(experiment)
```

```
                data.append(benchmarkingInfo)

                if path_generated:
                    numValidPaths += 1


            runInfo['numValidPaths'] = copy.deepcopy(numValidPaths)
            runInfo['numTimesRun'] = numRunsOfPlannerPerSetting
            runInfo.update(copy.deepcopy(experiment))
            pathValidityData.append(runInfo)

            print(runInfo)


        benchMarkingDF = pd.DataFrame(data)
        pathValidityDF = pd.DataFrame(pathValidityData)


benchMarkingDF.to_csv('/home/arpit/studies/motion/Assignment4/benchMarkingDF.csv',h
eader=True)

pathValidityDF.to_csv('/home/arpit/studies/motion/Assignment4/pathValidityDF.csv',h
eader=True)

        (benchMarkingDF, pathValidityDF, benchParams) = (benchMarkingDF,
pathValidityDF, parametersToVary)

        plotTitle = 'PRM' + '_stats'

        my_path = os.path.abspath(__file__) + '\plots'

        plotStatistics(benchMarkingDF=benchMarkingDF,
                       pathValidityDF=pathValidityDF,
                       benchParams=benchParams,
                       baseSaveFName=my_path,
                       plotTitle=plotTitle)


if __name__ == '__main__':
    main()

==================================
rrt_WO3.py
==================================
import math
import random

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.pyplot as plt
import numpy as np
from scipy import spatial
# https://www.geeksforgeeks.org/union-find/
from networkx.utils import UnionFind
from shapely.geometry import Point, LineString, Polygon, \
    MultiPolygon
import random
import copy
```

```
# from graph import Graph
from workspaces import config
import yaml
import itertools
from timeit import default_timer as timer
import pandas as pd
import seaborn as sns
import os
import random
import math

WORKSPACE_CONFIG = config()

show_animation = True


class RRT:
    """
    Class for RRT planning
    """

    class Node:
        """
        RRT Node
        """

        def __init__(self, x, y):
            self.x = x
            self.y = y
            self.path_x = []
            self.path_y = []
            self.parent = None

    def __init__(self):

        name = 'rrt_WO3.yaml'
        with open(name, 'r') as stream:
            configData = yaml.load(stream, Loader=yaml.Loader)

        self.startState = configData['startState']
        self.goalState = configData['goalState']
        self.minBounds = configData['minBounds']
        self.maxBounds = configData['maxBounds']

        self.start = self.Node(configData['startState'][0],
configData['startState'][1])
        self.end = self.Node(configData['goalState'][0], configData['goalState']
[1])
        self.radius = configData['radius']
        self.grid_size = configData['grid_size']
        self.goal_sample_rate = configData['goal_sample_rate']
        self.max_iter = configData['max_iter']
        self.graph = []

    def planning(self, animation=True):
        pathLength = 0
        computationTime = 0
        start = timer()
        path_generated = False
```

```
        self.graph = [self.start]
        for i in range(self.max_iter):
            sample = self.get_random_node()

            inter_node_dist_list = [(node.x - sample.x)**2 + (node.y - sample.y)**2
for node in self.graph]
            temp = inter_node_dist_list.index(min(inter_node_dist_list))
            next_node = self.graph[temp]

            upgraded_node = self.directional_growth(next_node, sample, self.radius)

            if self.check_collision(upgraded_node):
                self.graph.append(upgraded_node)

            if animation and i % 5 == 0:
                self.plotGraph(sample)

            if self.GAOLdist(self.graph[-1].x, self.graph[-1].y) <= self.radius:
                final_node = self.directional_growth(self.graph[-1], self.end,
self.radius)
                if self.check_collision(final_node):
                    _path = [[self.end.x, self.end.y]]
                    _node = self.graph[(len(self.graph) - 1)]
                    while _node.parent is not None:
                        _path.append([_node.x, _node.y])
                        _node = _node.parent
                    _path.append([_node.x, _node.y])
                    path = _path

            if animation and i % 5:
                self.plotGraph(sample)

        finish = timer()
        computationTime = finish - start

        if path is not None:
            start_x = 0
            start_y = 0
            pathLength = 0
            path_generated = True
            for (x, y) in path:
                dx, dy = (start_x - x), (start_y - y)
                d = math.hypot(dx, dy)
                start_x = x
                start_y = y
                pathLength += d

            return (computationTime, path, pathLength, True)

        return (None, None, None, False)  # cannot find path

    def directional_growth(self, src, dest, grow_len=float("inf")):

        upgraded_node = self.Node(src.x, src.y)
        dx = (dest.x - upgraded_node.x)
        dy = (dest.y - upgraded_node.y)
        euclid_dist = math.hypot(dx, dy)
        theta = math.atan2(dy, dx)
```

```
        upgraded_node.path_x, upgraded_node.path_y = [upgraded_node.x],
[upgraded_node.y]

        if grow_len > euclid_dist:
            grow_len = euclid_dist

        for _ in range(math.floor(grow_len / self.grid_size)):
            upgraded_node.x += self.grid_size * math.cos(theta)
            upgraded_node.y += self.grid_size * math.sin(theta)
            upgraded_node.path_x.append(upgraded_node.x)
            upgraded_node.path_y.append(upgraded_node.y)

        dx, dy = (dest.x - upgraded_node.x), (dest.y - upgraded_node.y)
        # dy = dest.y - upgraded_node.y
        euclid_dist = math.hypot(dx, dy)
        if euclid_dist <= self.grid_size:
            upgraded_node.path_x.append(dest.x)
            upgraded_node.path_y.append(dest.y)
            upgraded_node.x = dest.x
            upgraded_node.y = dest.y

        upgraded_node.parent = src

        return upgraded_node

    def GAOLdist(self, x, y):
        dx = x - self.end.x
        dy = y - self.end.y
        return math.hypot(dx, dy)

    def get_random_node(self):
        if random.randint(0, 100) > self.goal_sample_rate:
            rnd = self.Node(
                random.uniform(self.minBounds[0], self.maxBounds[0]),
                random.uniform(self.minBounds[1], self.maxBounds[1]))
        else:  # goal point sampling
            rnd = self.Node(self.end.x, self.end.y)
        return rnd

    def plotGraph(self, rnd=None):
        plt.clf()
        # for stopping simulation with the esc key.
        if rnd is not None:
            plt.plot(rnd.x, rnd.y, "^k")
        for node in self.graph:
            if node.parent:
                plt.plot(node.path_x, node.path_y, "-g")

        obstacles = WORKSPACE_CONFIG['WO3']
        for obst in obstacles:
            x,y = obst.exterior.xy
            plt.fill(x,y, alpha=0.5, fc='c',ec='none')

        plt.plot(self.startState[0], self.startState[1], "xr")
        plt.plot(self.goalState[0], self.goalState[1], "xr")
        plt.axis("equal")
        plt.axis([self.minBounds[0], self.maxBounds[0], self.minBounds[1],
self.maxBounds[1]])
        plt.grid(True)
```

```
        # plt.pause(0.01)

    def check_collision(self, node):

        obstacles = WORKSPACE_CONFIG['WO3']
        p = Point(node.x, node.y)
        collide_flag = False
        for obstacle in obstacles:
            if p.within(obstacle):

                collide_flag = True
                break

        if collide_flag:
            return False
        else:
            return True


def savePlot(fig, shouldSavePlots, baseSaveFName, plotTitle,
             useTightLayout=True):
    print("Saving fig: ", plotTitle)

    if shouldSavePlots:
        saveFName = baseSaveFName + '-' + plotTitle + '.png'
        if useTightLayout:
            plt.tight_layout()
        plt.savefig(saveFName, dpi=500)

        print('wrote figure to: ', saveFName)
        # plt.show()
        plt.close(fig)

def plotStatistics(benchMarkingDF, pathValidityDF, benchParams, baseSaveFName,
plotTitle):

    print("Entering Plotting Stastics")
    ##
    # Plotting boxplots
    ##
    boxPlotsToMake = ['computationTimeInSeconds', 'pathLength']

    # need to create a new, merged categorical data for boxplots
    mergedParamsName = ', '.join(benchParams)
    benchMarkingDF[mergedParamsName] = benchMarkingDF[benchParams].apply(
        lambda x: ', '.join(x.astype(str)), axis=1)
    pathValidityDF[mergedParamsName] = pathValidityDF[
        benchParams].apply(lambda x: ', '.join(x.astype(str)), axis=1)

    # Usual boxplot for each variable that was benchmarked
    for plotVar in boxPlotsToMake:

        # make it wider for the insanse length of xticklabels
        fig = plt.figure(figsize=(20, 10))

        plt.style.use("seaborn-darkgrid")
        bp = sns.boxplot(data=benchMarkingDF,
                         x=mergedParamsName, y=plotVar)
        sns.swarmplot(x=mergedParamsName, y=plotVar, data=benchMarkingDF,
```

```
                        color="grey")

        # for readability of axis labels
        bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

        newPlotTitle = plotVar + '-' + plotTitle
        plt.title('Benchmarking of Sampled Planner ' + plotVar)
        savePlot(fig=fig, shouldSavePlots=True,
                    baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)

    # number of times a valid path was found
    fig = plt.figure()

    plt.style.use('seaborn-darkgrid')
    bp = sns.barplot(x=mergedParamsName, y='numValidPaths',
                        data=pathValidityDF)
    plt.title('Number of Valid Paths Found for Each Parameter Combination')

    # for readability of axis labels
    bp.set_xticklabels(bp.get_xticklabels(), rotation=45, ha='right')

    newPlotTitle = 'numPaths' + '-' + plotTitle
    savePlot(fig=fig, shouldSavePlots=True,
                baseSaveFName=baseSaveFName, plotTitle=newPlotTitle)


def main():

    val = input("Enter 0 to get a single run, Enter 1 for Benchmarking Plot: ")
    val = int(val)

    if val == 0:

        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.set_axisbelow(True)

        name = 'rrt_WO3.yaml'
        with open(name, 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        minBounds = configData['minBounds']
        maxBounds = configData['maxBounds']

        distance = 0

        numRunsOfPlannerPerSetting = 50
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)
        print(allParams)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        plotConfigData = {'shouldPlot': True,
                          'plotTitle': '',
```

```
                            'xlabel': 'x',
                            'ylabel': 'y',
                            'plotObstacles': True,
                            'plotGrid': False}

        rrt = RRT()

        (computationTime, path, pathLength, path_generated) =
rrt.planning(animation=False)

        if path is None:
            print("Algorithm convergence failed in the specified number of
iterations")
        else:
            print("You Bet: GOT A PATH")
            # # Draw final path
            if show_animation:
                rrt.plotGraph()
                plt.plot([x for (x, y) in path], [y for (x, y) in path], '-r')
                plt.grid(True)

        start_x = 0
        start_y = 0
        pathLength = 0

        for (x, y) in path:
            dx = start_x - x
            dy = start_y - y
            d = math.hypot(dx, dy)
            start_x = x
            start_y = y
            pathLength += d

        r = 0.5

        plotTitle = 'RRT - path length = %0.3g r = %0.3g' % (pathLength, r)

        ax.set_aspect('equal')
        plt.title(plotTitle)
        ax.axes.get_xaxis().set_visible(True)
        ax.axes.get_yaxis().set_visible(True)
        ax.set_xlim(minBounds[0], maxBounds[0])
        ax.set_ylim(minBounds[1], maxBounds[1])
        fig.legend(loc='upper left')

        plt.show()

    if val == 1:
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.set_axisbelow(True)

        name = 'rrt_WO3.yaml'
        with open(name, 'r') as stream:
                configData = yaml.load(stream, Loader=yaml.Loader)

        minBounds = configData['minBounds']
        maxBounds = configData['maxBounds']
```

```
        distance = 0

        numRunsOfPlannerPerSetting = 100
        parametersToVary = configData['paramterNamesToVary']
        allParams = dict((var, configData[var]) for var in parametersToVary)

        keys, values = zip(*allParams.items())
        experiments = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(experiments)
        data = []
        pathValidityData = []

        plotConfigData = {'shouldPlot': True,
                          'plotTitle': '',
                          'xlabel': 'x',
                          'ylabel': 'y',
                          'plotObstacles': True,
                          'plotGrid': False}

        for experiment in experiments:
            rrt = None
            rrt = RRT()
            numValidPaths = 1
            runInfo = {}

            for idx, i in enumerate(range(0, 100)):
                (computationTime, path, pathLength, path_generated) =
rrt.planning(animation=False)

                dat = {'computationTimeInSeconds': computationTime, 'pathLength':
pathLength}
                bencmarkingInfo = {**dat, **experiment}
                (benchmarkingInfo, path_generated) = (bencmarkingInfo,
path_generated)

                benchmarkingInfo.update(experiment)
                data.append(benchmarkingInfo)

                if path_generated:
                    numValidPaths += 1


            runInfo['numValidPaths'] = copy.deepcopy(numValidPaths)
            runInfo['numTimesRun'] = numRunsOfPlannerPerSetting
            runInfo.update(copy.deepcopy(experiment))
            pathValidityData.append(runInfo)

            print(runInfo)


        benchMarkingDF = pd.DataFrame(data)
        pathValidityDF = pd.DataFrame(pathValidityData)


benchMarkingDF.to_csv('/home/arpit/studies/motion/Assignment4/benchMarkingDF.csv',h
eader=True)

pathValidityDF.to_csv('/home/arpit/studies/motion/Assignment4/pathValidityDF.csv',h
eader=True)
```

```
        (benchMarkingDF, pathValidityDF, benchParams) = (benchMarkingDF,
pathValidityDF, parametersToVary)

        plotTitle = 'PRM' + '_stats'

        my_path = os.path.abspath(__file__) + '\plots'

        plotStatistics(benchMarkingDF=benchMarkingDF,
                       pathValidityDF=pathValidityDF,
                       benchParams=benchParams,
                       baseSaveFName=my_path,
                       plotTitle=plotTitle)


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

```
==================================
prm_w01_backup.yaml
==================================
# Start State
startState: [0,0]

# Goal State
goalState: [10, 10]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -1.0
    - -1.0

maxBounds:
    - 13.0
    - 13.0

# List of sample numbers to try
n:
    - 200

# List of Radii of sampling for neighbour analysis
r:
    - 2.0

# Smoothing
smoothing:
    - False

# Statistical Analysis
numRunsOfPlannerPerSetting: 1

# Params to vary
paramterNamesToVary:
    - n
    - r
    - smoothing
```

```
==================================
prm_w02_backup.yaml
==================================

# Start State
startState: [0,0]

# Goal State
goalState: [35, 0]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -6.0
    - -6.0

maxBounds:
    - 36.0
    - 6.0

# List of sample numbers to try
n:
    - 500

# List of Radii of sampling for neighbour analysis
r:
    - 2.0

# Smoothing
smoothing:
    - False

# Statistical Analysis
numRunsOfPlannerPerSetting: 1

# Params to vary
paramterNamesToVary:
    - n
    - r
    - smoothing
==================================
prm_w03_backup.yaml
==================================

# Start State
startState: [0,0]

# Goal State
goalState: [10, 0]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -1.0
    - -3.0
```

```
maxBounds:
    - 11.0
    - 3.0

# List of sample numbers to try
n:
    - 200

# List of Radii of sampling for neighbour analysis
r:
    - 1.0

# Smoothing
smoothing:
    - True

# Statistical Analysis
numRunsOfPlannerPerSetting: 1

# Params to vary
paramterNamesToVary:
    - n
    - r
    - smoothing
==================================
prm_w01.yaml
==================================
# Start State
startState: [0,0]

# Goal State
goalState: [10, 10]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -1.0
    - -1.0

maxBounds:
    - 13.0
    - 13.0

# List of sample numbers to try
n:
    - 200
    - 500
    - 1000

# List of Radii of sampling for neighbour analysis
r:
    - 1.0
    - 2.0

# Smoothing
smoothing:
```

```
    - False
    - True

# Statistical Analysis
numRunsOfPlannerPerSetting: 100

# Params to vary
paramterNamesToVary:
    - n
    - r
    - smoothing

==================================
prm_w02.yaml
==================================

# Start State
startState: [0,0]

# Goal State
goalState: [35, 0]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -6.0
    - -6.0

maxBounds:
    - 36.0
    - 6.0

# List of sample numbers to try
n:
    - 200
    - 500
    - 1000

# List of Radii of sampling for neighbour analysis
r:
    - 1.0
    - 2.0

# Smoothing
smoothing:
    - False
    - True

# Statistical Analysis
numRunsOfPlannerPerSetting: 100

# Params to vary
paramterNamesToVary:
    - n
    - r
    - smoothing
==================================
```

prm_w03.yaml
==========

```
# Start State
startState: [0,0]

# Goal State
goalState: [10, 0]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -1.0
    - -3.0

maxBounds:
    - 11.0
    - 3.0

# List of sample numbers to try
n:
    - 200
    - 500

# List of Radii of sampling for neighbour analysis
r:
    - 2.0

# Smoothing
smoothing:
    - False
    - True

# Statistical Analysis
numRunsOfPlannerPerSetting: 100

# Params to vary
paramterNamesToVary:
    - n
    - r
    - smoothing
```

==========

rrt_WO1.yaml

==========

```
# Start State
startState: [0,0]

# Goal State
goalState: [10, 10]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -1.0
    - -1.0
```

```
maxBounds:
    - 13.0
    - 13.0

# List of sample numbers to try
n:
    - 5000

# List of Radii of sampling for neighbour analysis
r:
    - 0.5

# Statistical Analysis
numRunsOfPlannerPerSetting: 100

# Params to vary
paramterNamesToVary:
    - n
    - r

# Expand Distance
radius: 0.5

# Path Resolution
grid_size: 0.5

# Goal Sample Rate
goal_sample_rate: 5

# Max iteratons
max_iter: 5000

# Node List initially empty
node_list: []

==================================
rrt_WO2.yaml
==================================

# Start State
startState: [0,0]

# Goal State
goalState: [35, 0]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -6.0
    - -6.0

maxBounds:
    - 36.0
    - 6.0

# List of sample numbers to try
```

```
n:
    - 5000

# List of Radii of sampling for neighbour analysis
r:
    - 0.5

# Smoothing
smoothing:
    - True

# Statistical Analysis
numRunsOfPlannerPerSetting: 100

# Params to vary
paramterNamesToVary:
    - n
    - r

# Expand Distance
radius: 0.5

# Path Resolution
grid_size: 0.5

# Goal Sample Rate
goal_sample_rate: 5

# Max iteratons
max_iter: 5000

# Node List initially empty
node_list: []

==================================
rrt_WO3.yaml
==================================

# Start State
startState: [0,0]

# Goal State
goalState: [10, 0]

# Discretization Density
gridDensity: 3

# Bounds
minBounds:
    - -1.0
    - -3.0

maxBounds:
    - 11.0
    - 3.0

# List of sample numbers to try
n:
    - 5000
```

```
# List of Radii of sampling for neighbour analysis
r:
    - 0.5

# Smoothing
smoothing:
    - True

# Statistical Analysis
numRunsOfPlannerPerSetting: 100

# Params to vary
paramterNamesToVary:
    - n
    - r

# Expand Distance
radius: 0.5

# Path Resolution
grid_size: 0.5

# Goal Sample Rate
goal_sample_rate: 5

# Max iteratons
max_iter: 5000

# Node List initially empty
node_list: []
```

```
==================================
graph.py
==================================
"""
Arpit Savarkar
Implemenatation of Graph Search
for Dijkstra and A*

"""

import matplotlib.pyplot as plt
import yaml
import networkx as nx
import heapq
import queue
import numpy as np
import copy


class Graph(nx.Graph):
    """
    Graph Class for handy functions
    """

    def __init__(self, nodes=[], edges=[]):

        # networkx initialization is also required
```

```
        super().__init__()

        if nodes:
            self.add_nodes_from(nodes)

            if edges:
                self.add_edges_from(edges)

            self.nodeProperties = set([k for n in self.nodes
                                       for k in self.nodes[n].keys()])

    def get_path(self, start, goal, algo):
        """
        Returns the Path From the nodes based on Graph
        PARAMETERS
        ----------
        start : Start Node
        goal : Goal Node
        algo : String - "A star" or "Dijkstra"

        RETURNS
        -------
        path: A list of the nodes
        pathLength: Integer - Length of the Algorithamic plan
        itrs: Number of iterations it took to find the result
        """
        path = None
        pathLength = None
        itrs = 0

        if start == goal:
            path = [start]
            return (path, pathLength, itrs)

        # Node is the structure that holds the priority, prev node and the distance
        for node in self.nodes:

            self.setter_helper(node, 'distance', np.inf)
            self.setter_helper(node, 'priority', np.inf)
            self.setter_helper(node, 'prev', None)

        # Sets the hueristic distance to 0 for the start node
        self.setter_helper(start, 'priority', 0)
        self.setter_helper(start, 'distance', 0)

        # Priority Queue implementation
        Q = PrioQueue()

        # Pushes onto the FIFO(Queue)
        Q.put(self.getPriorityTuple(start))

        while not Q.empty():

            currPriority, currNode = Q.get()
            itrs += 1

            # Success
            if self.searchStop(currNode, goal, algo):
```

```
                pathLength = self.getter_helper(currNode, 'distance')
                path = self.revv(start, goal)
                break

            # Checks the Neighhbour for the heuristic with weight additions
            for neighbor in self.adj[currNode]:

                edg = (currNode, neighbor)

                src = edg[0]
                dest = edg[1]
                weight = self.edges[edg]['weight']

                srcDist = self.getter_helper(src, 'distance')
                srcPriority = self.getter_helper(src, 'priority')

                distance_to_label_dest = self.getter_helper(dest, 'distance')
                destPriority = self.getter_helper(dest, 'priority')

                # Constraint Condition
                if not (self.getter_helper(src, 'prev') == dest):
                    flag = (srcDist + weight) < destPriority
                else:
                    flag = False

                if flag:
                    # Updates the path accordingly, push onto the priority_queue
                    self.update_to_shorter_route(edg, algo)
                    Q.put(self.getPriorityTuple(neighbor))

        return (path, pathLength, itrs)


    def update_to_shorter_route(self, edgeLabel, algo):
        """
        Update step to find a route shorter than the existing calculated route
        PARAMETERS
        ----------
        edgeLabel: Edge Index
        algo: String - "A star" or "Dijkstra"

        RETURNS
        -------
        distance_to_label_dest: Distance Count
        """
        src = edgeLabel[0]
        dest = edgeLabel[1]
        weight = self.edges[edgeLabel]['weight']

        srcDist = self.getter_helper(src, 'distance')
        srcPriority = self.getter_helper(src, 'priority')

        distance_to_label_dest = self.getter_helper(dest, 'distance')
        destPriority = self.getter_helper(dest, 'priority')

        if algo == 'A star':

            distance_to_label_dest = copy.deepcopy(self.getter_helper(src,
'distance')) + weight
```

```
            destPriority = copy.deepcopy(distance_to_label_dest) + \
                self.getter_helper(dest, 'heuristic')

        elif algo == 'Dijkstra':

            distance_to_label_dest = copy.deepcopy(self.getter_helper(src,
'distance')) + weight
            destPriority = copy.deepcopy(distance_to_label_dest)


        self.setter_helper(dest, 'distance', distance_to_label_dest)
        self.setter_helper(dest, 'priority', destPriority)

        if not (self.getter_helper(src, 'prev') == dest):
            self.setter_helper(dest, 'prev', src)

        return distance_to_label_dest

    def searchStop(self, currNode, goal, algo):
        """
        Success Condition
        PARAMETERS
        ----------
        currNode : Node under consideration
        goal: Goal Node
        algo : String - "A star" or "Dijkstra"

        RETURNS
        -------
        atGoal: Boolean . True/False
        """

        atGoal = (currNode == goal)

        if algo == 'A star':

            if atGoal:

                prevNode = self.getter_helper(currNode, 'prev')
                currDist = self.getter_helper(currNode, 'distance')
                prevPriority = self.getter_helper(prevNode, 'priority')

                return (currDist <= prevPriority)

        elif algo == 'Dijkstra':

            return atGoal

    def getPathEdges(self, path):
        """
        PARAMETERS
        ----------
        path: List of Nodes

        RETURNS
        -------
        path_edges: Path_Edges
        """
```

```
        path_edges = [(v1, v2) for v1, v2 in zip(path, path[1:])]

        return path_edges

    def revv(self, start, goal):
        """
        Reverse the Path, gives if found from start to goal
        PARAMETERS
        ---------
        start: Start Node
        goal: Goal Node
        """

        currNode = goal
        path = [currNode]

        while currNode != start:

            currNode = self.getter_helper(currNode, 'prev')
            path.append(currNode)

        path.reverse()

        return path

    def getPriorityTuple(self, node):
        """
        Returns the priority
        PARAMETERS
        ----------
        Node : Node
        """

        return (self.getter_helper(node, 'priority'), node)

    def getter_helper(self, nodeLabel, dataKey):
        """
        GETTER FUNCTION

        PARAMETERS
        ---------
        nodeLabel: Node Index
        dataKey:
        """

        nodeData = self.nodes.data()

        return nodeData[nodeLabel][dataKey]

    def setter_helper(self, nodeLabel, dataKey, data):
        """
        SETTER Function

        PARAMETERS
        ----------
        nodeLabel: Node Index
        data: Dataset to set
        dataKey:
```

```
    """

    nodeData = self.nodes.data()
    nodeData[nodeLabel][dataKey] = data

def print_edges(self):
    """
    Prints the Edges

    PARAMETERS
    ----------
    Segmented Print statement
    """

    for n, nbrs in self.adj.items():
        for nbr, eattr in nbrs.items():

            wt = eattr['weight']
            print('(%s, %s, %0.3g)' % (str(n), str(nbr), wt))

def dispNodes(self):
    """
    Prints Node in the graph established
    """
    for node in self.nodes(data=True):
        print(node)

def plot(self, path=None, fig=None, plotTitle=None,
         baseSize=400, node_size=10, showLabels=True,
         showEdgeWeights=True, showAxes=True):
    """
    Plotting Function
    PARAMETERS
    ----------
    path : List of nodes
    fig: plt.figure()
    Title: Plot title
    baseSize: Size of the figure
    node_size: Size of the nodes to be displayed (scaled)
    showLabels: Boolean

    """
    if not fig:
        fig = plt.figure()

    # scale node sizes by string length only if all node labels are strings
    allStrs = bool(self.nodes()) and all(isinstance(elem, str)
                                         for elem in self.nodes())
    pos = nx.get_node_attributes(self, 'pos')
    if allStrs:
        node_size = [len(v) * baseSize for v in self.nodes()]
        nx.draw_networkx(self, pos=pos,
                         with_labels=showLabels, node_size=node_size)
    else:
        nx.draw_networkx(self, pos=pos, with_labels=showLabels,
                         node_size=node_size,
                         cmap=plt.get_cmap('jet'))

    # show edge weights as well
```

```
        if showEdgeWeights:
            labels = nx.get_edge_attributes(self, 'weight')
            nx.draw_networkx_edge_labels(self, pos, edge_labels=labels)

        # draw path through the graph if it exists
        if path:

            nx.draw_networkx_nodes(self, pos, nodelist=path, node_color='r',
                                   node_size=node_size)

            path_edges = self.getPathEdges(path)
            nx.draw_networkx_edges(self, pos, edgelist=path_edges,
                                   edge_color='r', width=4)

        # Axes settings
        ax = plt.gca()
        ax.set_title(plotTitle)
        if showAxes:
            ax.tick_params(left=True, bottom=True,
                           labelleft=True, labelbottom=True)
        else:
            [sp.set_visible(False) for sp in ax.spines.values()]
            ax.set_xticks([])
            ax.set_yticks([])

        return (fig, ax)

# https://stackoverflow.com/questions/5997189/how-can-i-make-a-unique-value-
priority-queue-in-python

class PrioQueue(queue.Queue):
    """
    Class for Setting up the priority queue
    """

    def _init(self, maxsize):
        self.queue = []
        self.REMOVED = '<removed-task>'
        self.entry_finder = {}

    def _put(self, item, heappush=heapq.heappush):
        item = list(item)
        priority, task = item

        if task in self.entry_finder:
            # Do not add new item.
            pass
        else:
            self.entry_finder[task] = item
            heappush(self.queue, item)

    def _qsize(self, len=len):

        return len(self.entry_finder)

    def is_empty(self):
        while self.pq:
            if self.queue[0][1] != self.REMOVED:
                return False
```

```
            else:
                _, _, element = heapq.heappop(self.pq)
                if element in self.element_finder:
                    del self.element_finder[element]
        return True

    def _get(self, heappop=heapq.heappop):
        while self.queue:

            item = heappop(self.queue)
            _, task = item

            if task is not self.REMOVED:

                del self.entry_finder[task]
                return item




def print_helper(algo, pathLength, itr):
    print('|********  ', algo, '  *********|')
    if pathLength:
        print('Path Length:', pathLength)
    else:
        print('No Path Exists')
    print("Iterations: ", itr)

def plotPath(algo, graph, path, pathLength, itr, flag, flag2):
    print("**********SHORTEST PATH************")
    print(path)
    if path:
        plotTitle = 'Shortest Path (length = ' + str(pathLength) + \
                            ') Found with ' + algo + ' - nIter: ' + \
                            str(itr)
    else:
        plotTitle = 'No Path Found with ' + algo + ' - nIter: ' + \
                            str(itr)


def main():
    with open('graphConfig.yaml', 'r') as stream:
            configData = yaml.load(stream, Loader=yaml.Loader)

    nodes = configData['nodes']
    adjList = configData['edges']
    startNode = configData['startLabel']
    goalNode = configData['goalLabel']

    edgeList = []
    for srcEdge, dat in adjList.items():
        if dat:
            newEdges = [(srcEdge, data[0], data[1]) for data in dat]
            edgeList.extend(newEdges)

    G = Graph(nodes.items(), edges=edgeList)

    path = {}
    pathLength = {}
```

```
    num_itrs = {}

    val = input("Enter 0 for A_star and 1 for Dijkstra: ")
    val = int(val)

    if val == 0:
        algo = 'A star'
        (path[algo],
        pathLength[algo],
        num_itrs[algo]) = G.get_path(start = startNode,
                                     goal = goalNode,
                                     algo = algo)

        print_helper(algo, pathLength[algo], num_itrs[algo])
        plotPath(algo, G, path[algo], pathLength[algo],
                    num_itrs[algo], True, True)
    elif val == 1:

        algo = 'Dijkstra'
        (path[algo],
            pathLength[algo],
            num_itrs[algo]) = G.get_path(start=startNode,
                                                goal=goalNode,
                                                algo=algo)

        print_helper(algo, pathLength[algo], num_itrs[algo])
        plotPath(algo, G, path[algo], pathLength[algo],
                    num_itrs[algo], True, True)


if __name__ == '__main__':
    main()

==================================
graphConfig.yaml
==================================
# Start Node Label
startLabel: 'START'

# End Node Label
goalLabel: 'GOAL'

nodes:

    'START':
        heuristic: 0.0
        prev: null
        distance: .inf
        priority: .inf
        pos: [0, 10.0]

    'A':
        heuristic: 3.0
        prev: null
        distance: .inf
        priority: .inf
        pos: [-1.0, 9.0]

    'B':
```

```
    heuristic: 2.0
    prev: null
    distance: .inf
    priority: .inf
    pos: [0.0, 8.0]

'C':
    heuristic: 3.0
    prev: null
    distance: .inf
    priority: .inf
    pos: [1.0, 9.0]

'D':
    heuristic: 3.0
    prev: null
    distance: .inf
    priority: .inf
    pos: [-2.0, 9.0]

'E':
    heuristic: 1.0
    prev: null
    distance: .inf
    priority: .inf
    pos: [-2.0, 5.0]

'F':
    heuristic: 3.0
    prev: null
    distance: .inf
    priority: .inf
    pos: [-1.0, 7.0]

'G':
    heuristic: 2.0
    prev: null
    distance: .inf
    priority: .inf
    pos: [-1.0, 5.0]

'H':
    heuristic: 1.0
    prev: null
    distance: .inf
    priority: .inf
    pos: [0.0, 5.0]

'I':
    heuristic: 2.0
    prev: null
    distance: .inf
    priority: .inf
    pos: [1.0, 5.0]

'J':
    heuristic: 3.0
    prev: null
    distance: .inf
```

```
        priority: .inf
        pos: [1.0, 7.0]

    'K':
        heuristic: 2.0
        prev: null
        distance: .inf
        priority: .inf
        pos: [2.0, 5.0]

    'L':
        heuristic: 3.0
        prev: null
        distanceance: .inf
        priority: .inf
        pos: [2.0, 9.0]

    'GOAL':
        heuristic: 0.0
        prev: null
        distanceance: .inf
        priority: .inf
        pos: [0.0, 3.0]

# Edges
edges:
    'START':
        - ['A', weight: 1.0]
        - ['B', weight: 1.0]
        - ['C', weight: 1.0]

    'A':
        - ['D', weight: 1.0]
        - ['E', weight: 1.0]
        - ['F', weight: 3.0]

    'B':
        - ['G', weight: 4.0]
        - ['H', weight: 1.0]
        - ['I', weight: 2.0]

    'C':
        - ['J', weight: 1.0]
        - ['K', weight: 1.0]
        - ['L', weight: 1.0]

    'D':

    'E':
        - ['GOAL', weight: 3.0]

    'F':

    'G':
        - ['GOAL', weight: 3.0]

    'H':

    'I':
```

```
    - ['GOAL', weight: 3.0]

'J':

'K':
    - ['GOAL', weight: 2.0]

'L':

'GOAL':
```