from \_\_future\_\_ import print\_function

import rospy

import random

import numpy as np

import robot\_global

import human

import motor

import laser

import time\_step

import math

import tf

import sys

from time import gmtime, strftime

import copy

import time

from ShortestPathFindAlgorithm import spfa

from ShortestPathFindAlgorithm import unit

from std\_srvs.srv import Empty

from sensor\_msgs.msg import LaserScan

from std\_msgs.msg import String

from std\_msgs.msg import String

from geometry\_msgs.msg import Twist

from std\_msgs.msg import Bool

from geometry\_msgs.msg import Quaternion

from geometry\_msgs.msg import Vector3

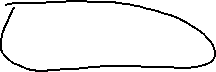
#ACTION\_LOG\_FILE = open("my\_net18/actionLog\_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt", 'a+')

actionoutPath = "my\_net22/actionLog\_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt"

PI=3.1415926

RSTATEDIS = 10

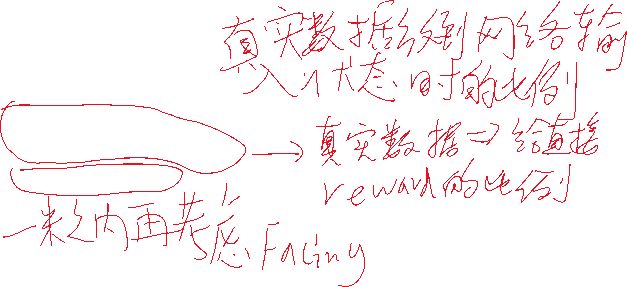
RSTATEANGLE = 10



RSTATEFACINGANGLE = 10

RREWARDDIS = 20\*5

RREWARDANGLE = 20



RREWARDFACINGANGLE = 20\*4

FACINGDIS\_RANGE = 1

MAX\_LASER\_RANGE = 3

MIN\_LASER\_RANGE = 0

MINNUM = 0.1

FAILREWARD = -300

WINREWARD = 500

ARRIVEDIS = 0.8

ARRIVEANGLE = 0.5

POSITIVEDIS = 0.8

DISFROMSTARTTOEND = 4

MINDISFROMSTARTTOEND = 0.5

my\_case\_robotx = [3, 1, 3, 2, 2, 1]

my\_case\_roboty = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]

my\_case\_robotz = [2, 2, 2.5, 2.5, 2.5, 7]

my\_case\_robot\_rotat\_x = [0, 0, 0, 0, 0, 0]

my\_case\_robot\_rotat\_y = [1, 1, 1, 1, 1, 1]

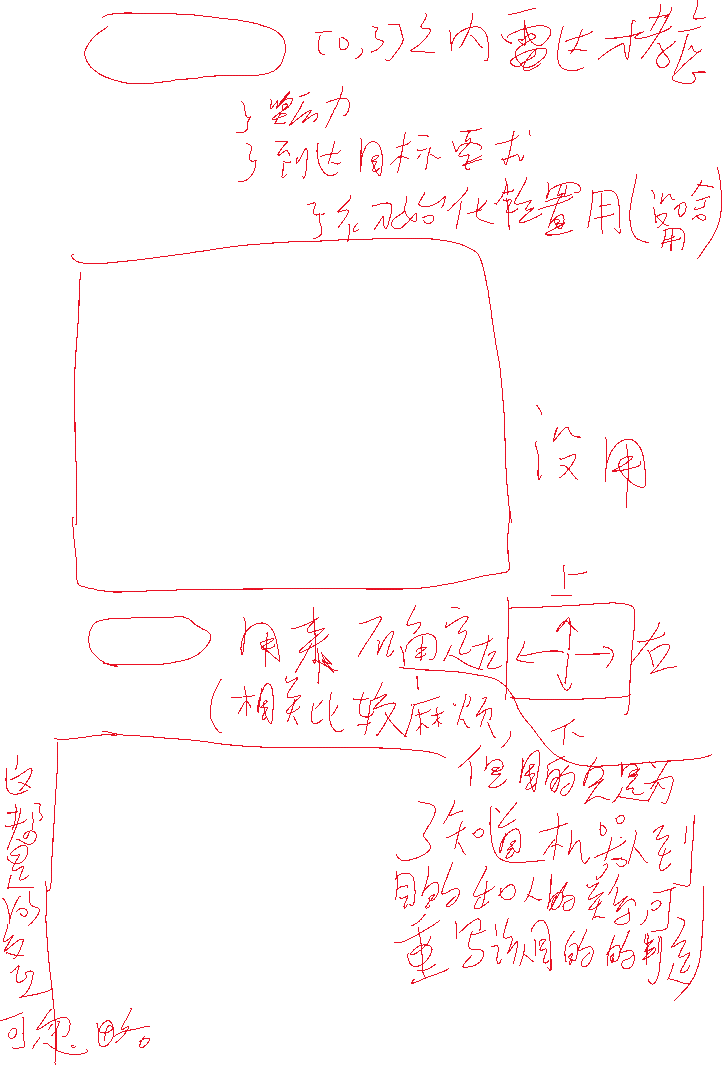
my\_case\_robot\_rotat\_z = [0, 0, 0, 0, 0, 0]

my\_case\_robot\_rotat\_w = [PI, 0, -PI, -PI / 4, -PI \* 3 / 4, -PI / 4]

my\_case\_humanx = [3, 3, 3, 2, 2, 3]

my\_case\_humany = [1.27, 1.27, 1.27, 1.27, 1.27, 1.27]

my\_case\_humanz = [8, 8, 6, 7, 7, 3.5]



opposite\_x = [0, 0, 0, 2, 2, 0]

opposite\_z = [-1.5, -1.5, 1.5, 0, 0, -1.5]

tmp\_out\_pos = 0

global\_rn = 0

def get\_human\_position\_res\_callback(position\_value):

human.position = position\_value

human.get\_position\_res\_flag = True

def pub\_get\_human\_position\_req():

value = Bool()

value.data = True

human.PubGetPositionReq.publish(value)

human.get\_position\_res\_flag = False

def get\_human\_rotation\_res\_callback(rotation\_value):

human.rotation = rotation\_value

human.get\_rotation\_res\_flag = True

def pub\_get\_human\_rotation\_req():

value = Bool()

value.data = True

human.PubGetRotationReq.publish(value)

human.get\_rotation\_res\_flag = False

def get\_position\_res\_callback(position\_value):

robot\_global.position = position\_value

robot\_global.get\_position\_res\_flag = True

def pub\_get\_position\_req():

value = Bool()

value.data = True

robot\_global.PubGetPositionReq.publish(value)

robot\_global.get\_position\_res\_flag = False

def set\_position\_res\_callback(value):

robot\_global.set\_position\_res\_flag = value.data

## new add for human

def set\_human\_pose\_res\_callback(value):

human.set\_human\_pose\_res\_flag = value.data

def pub\_set\_human\_pose\_req(position\_value):

human.PubSetPositionReq.publish(position\_value)

human.set\_human\_pose\_res\_flag = False

def pub\_set\_position\_req(position\_value):

robot\_global.PubSetPositionReq.publish(position\_value)

robot\_global.set\_position\_res\_flag = False

def set\_rotation\_res\_callback(value):

robot\_global.set\_rotation\_res\_flag = value.data

def pub\_set\_rotation\_req(rotation\_value):

robot\_global.PubSetRotationReq.publish(rotation\_value)

robot\_global.set\_rotation\_res\_flag = False

def get\_rotation\_res\_callback(rotation\_value):

robot\_global.rotation = rotation\_value

robot\_global.get\_rotation\_res\_flag = True

def pub\_get\_rotation\_req():

value = Bool()

value.data = True

robot\_global.PubGetRotationReq.publish(value)

robot\_global.get\_rotation\_res\_flag = False

def reset\_node\_physics\_res\_callback(value):

robot\_global.reset\_node\_physics\_res\_flag = value.data

def pub\_reset\_node\_physics\_req():

value = Bool()

value.data = True

robot\_global.PubResetNodePhsicsReq.publish(value)

robot\_global.reset\_node\_physics\_res\_flag = False

def human\_connect():

rospy.Subscriber("/human\_name", String)

# connect

model\_name = None

while model\_name is None:

try:

model\_name = rospy.wait\_for\_message('/human\_name', String, timeout=5)

except:

pass

print("human %s connect success" % model\_name.data)

return model\_name.data

def robot\_connect():

rospy.Subscriber("/model\_name", String)

# connect

model\_name = None

while model\_name is None:

try:

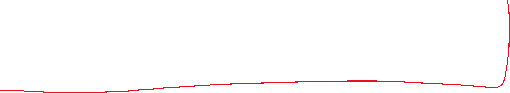
model\_name = rospy.wait\_for\_message('/model\_name', String, timeout=5)

except:

pass

print("robot %s connect success" % model\_name.data)

return model\_name.data

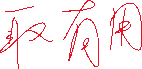


#change webots laser and dist to our focus data

def discretize\_observation(laser\_data, laser\_dim, collision\_threshold):



sum = 0.0

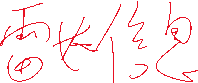


new\_laser\_data = []



for i, item in enumerate(laser\_data.ranges):

sum += laser\_data.ranges[i]



if ((i + 1) % 3 == 0):

new\_laser\_data.append(sum / 3.0)

sum = 0

collision = False

mod = len(new\_laser\_data) / laser\_dim

discretized\_ranges = []

###just for habit..

HJKINF = 1000000

tmp\_cnt = 0

min\_val = HJKINF

for i, item in enumerate(new\_laser\_data):

####add

if (i % mod == mod - 1 and tmp\_cnt + 1 <= laser\_dim):

min\_val = min(min\_val, new\_laser\_data[i])

tmp\_cnt = tmp\_cnt + 1

if min\_val > MAX\_LASER\_RANGE:

discretized\_ranges.append(MAX\_LASER\_RANGE)

elif min\_val < MIN\_LASER\_RANGE:

discretized\_ranges.append(MIN\_LASER\_RANGE)

else:

discretized\_ranges.append(min\_val)

####init

if (i % mod == 0):

min\_val = HJKINF

min\_val = new\_laser\_data[i]

#####update

min\_val = min(min\_val, new\_laser\_data[i])

if (new\_laser\_data[i] < collision\_threshold):

collision = True

print("hjk--- check dim of laser: ", len(discretized\_ranges))

#for x in enumerate(discretized\_ranges):

# print("\*\*\*check\*\*: ", x)

return discretized\_ranges, collision

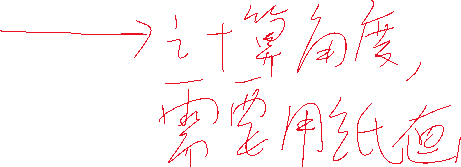
#get angle from point 1 to point 2

#angle is rotate y-axis(+) to vector<1=>2>, reverse clock is +(0, PI), clock is -(0, PI)

def getangle(x1, y1, x2, y2):

if y1 > y2 + MINNUM:

rtang = math.atan((x1-x2)/(y1-y2))



elif math.fabs(y1-y2) < MINNUM:

if x1 > x2+MINNUM:

rtang = PI/2



elif x1 < x2-MINNUM:

rtang = -PI/2

else:



rtang = 0

else:

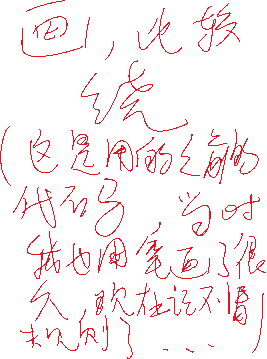
if x1 > x2:

rtang = PI+math.atan((x1-x2)/(y1-y2))

else:

rtang = -PI+math.atan((x1-x2)/(y1-y2))

return rtang



#get feature between point 1 and point 2: angle from point 1 to interleave angle, angle from interleave angle to point 2

def getanglefea(x1, y1, w1, x2, y2, w2):

rtang = getangle(x1, y1, x2, y2)

ang1 = rtang - w1

if ang1 > PI:

ang1 = ang1-2\*PI

elif ang1 <= -PI:

ang1 = ang1 + 2\*PI

ang2 = w2 - w1

if ang2 > PI:

ang2 = ang2 - 2\*PI

elif ang2 <= -PI:

ang2 = ang2 + 2\*PI

return rtang, ang1, ang2

#now for hjk, just calcu dist.

def getDisXZ(robotpos\_x, robotpos\_z, robotrot\_w, rightpos\_x, rightpos\_z, rightrot\_w):

distx = robotpos\_x - rightpos\_x

distz = robotpos\_z - rightpos\_z

dist1 = math.sqrt(distx \* distx + distz \* distz)

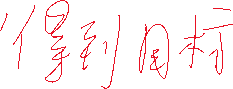
return dist1

def getopposite(position, rotation, my\_case = -1):

rpos = Vector3()

rpos.x = position.x

rpos.y = position.y



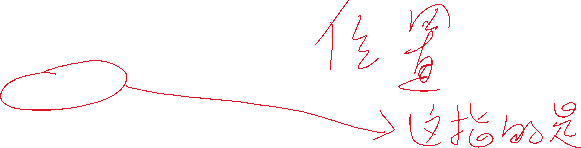
rpos.z = position.z #- 0.5

global tmp\_out\_pos

tmp\_out\_pos = 0

global global\_rn

if my\_case == -1:



print("hjktest---- getopposit this")

#rn = random.random()

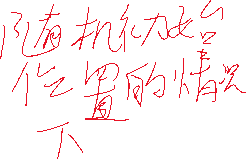
if global\_rn < 0.25:

rpos.x = position.x - POSITIVEDIS

tmp\_out\_pos = 1

elif global\_rn < 0.5:

rpos.x = position.x + POSITIVEDIS



tmp\_out\_pos = 3

elif global\_rn < 0.75:

rpos.z = position.z + POSITIVEDIS

tmp\_out\_pos = 2

else:

rpos.z = position.z - POSITIVEDIS

tmp\_out\_pos = 4

else:

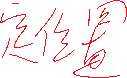
rpos.x = rpos.x + opposite\_x[my\_case]



rpos.z = rpos.z + opposite\_z[my\_case]



if opposite\_x[my\_case] > 0.01:



tmp\_out\_pos = 3



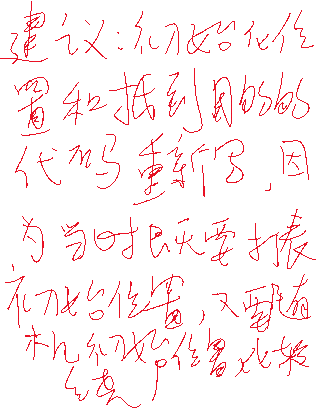
if opposite\_x[my\_case] < -0.01:

tmp\_out\_pos = 1

if opposite\_z[my\_case] > 0.01:

tmp\_out\_pos = 2

if opposite\_z[my\_case] < -0.01:



tmp\_out\_pos = 4

print("GGGGGGGGGg\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* goal is : ", tmp\_out\_pos)

#rpos.z = position.z + 3 # - 0.5

rrot = Quaternion()

rrot.x = rotation.x

rrot.y = rotation.y

rrot.z = rotation.z

rrot.w = PI

if tmp\_out\_pos == 1:

rrot.w = -PI/ 2

elif tmp\_out\_pos == 2:

rrot.w = 0

elif tmp\_out\_pos == 3:

rrot.w = PI / 2

elif tmp\_out\_pos == 4:

rrot.w = PI

# if rrot.w > PI:

# rrot.w = rrot.w - PI\*2

return rpos, rrot

def changeStateFromEnvToNetwork(laser\_state, dist1, rot1, rot2, action):

print("hjk--- rot1 is : ", rot1)

print("hjk--- dist1 is : ", dist1)



# no need to add do something to laser\_state. Becuase laser\_state has its limit !!!!!!!!!!!!!!!!!!!

limitLaser\_state = laser\_state

state = limitLaser\_state + [dist1 \* RSTATEDIS, rot1 \* RSTATEANGLE, rot2 \* RSTATEFACINGANGLE, action]

return state

class WebotsLidarNnEnv(): #why not object?? hjk

def \_\_init\_\_(self, laser\_dim, collision\_threshold):



self.my\_case = -1

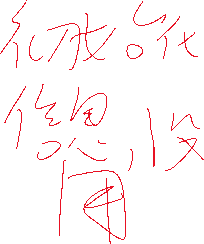
global global\_rn



global\_rn = random.random()



self.laser\_dim = laser\_dim



self.collision\_threshold = collision\_threshold

rospy.init\_node('webots\_env', anonymous=True)

robot\_global.robot\_name = robot\_connect()

human.human\_name = human\_connect()

robot\_global.PubSetPositionReq = rospy.Publisher('/simulation\_set\_position\_req', Vector3, queue\_size=1)

robot\_global.SubSetPositionRes = rospy.Subscriber('/simulation\_set\_position\_res', Bool,

set\_position\_res\_callback)

###new add for human\_pose

human.PubSetPositionReq = rospy.Publisher('/simulation\_set\_human\_pose\_req', Vector3, queue\_size=1)

human.SubSetPositionRes = rospy.Subscriber('/simulation\_set\_human\_pose\_res', Bool,

set\_human\_pose\_res\_callback)

# human.PubSetRotationReq = rospy.Publisher('/simulation\_set\_rotation\_req', Quaternion, queue\_size=1)

#human.SubSetRotationRes = rospy.Subscriber('/simulation\_set\_rotation\_res', Bool,

# set\_rotation\_res\_callback)

robot\_global.PubSetRotationReq = rospy.Publisher('/simulation\_set\_rotation\_req', Quaternion, queue\_size=1)

robot\_global.SubSetRotationRes = rospy.Subscriber('/simulation\_set\_rotation\_res', Bool,

set\_rotation\_res\_callback)

robot\_global.PubResetNodePhsicsReq = rospy.Publisher('/simulation\_reset\_node\_physics\_req', Bool, queue\_size=1)

robot\_global.SubResetNodePhsicsRes = rospy.Subscriber('/simulation\_reset\_node\_physics\_res', Bool,

reset\_node\_physics\_res\_callback)

robot\_global.PubGetPositionReq = rospy.Publisher('/simulation\_get\_position\_req', Bool, queue\_size=1)

robot\_global.SubGetPositionRes = rospy.Subscriber('/simulation\_get\_position\_res', Vector3,

get\_position\_res\_callback)

robot\_global.PubGetRotationReq = rospy.Publisher('/simulation\_get\_rotation\_req', Bool, queue\_size=1)

robot\_global.SubGetRotationRes = rospy.Subscriber('/simulation\_get\_rotation\_res', Quaternion,

get\_rotation\_res\_callback)

human.PubGetPositionReq = rospy.Publisher('/simulation\_get\_human\_position\_req', Bool, queue\_size=1)

human.SubGetPositionRes = rospy.Subscriber('/simulation\_get\_human\_position\_res', Vector3,

get\_human\_position\_res\_callback)

human.PubGetRotationReq = rospy.Publisher('/simulation\_get\_human\_rotation\_req', Bool, queue\_size=1)

human.SubGetRotationRes = rospy.Subscriber('/simulation\_get\_human\_rotation\_res', Quaternion,

get\_human\_rotation\_res\_callback)

for i in range(0, 5):

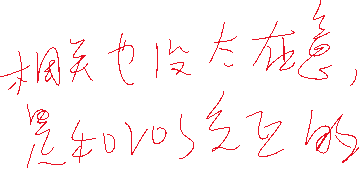
time\_step.time\_step\_call() #zuoyong? hjk

motor.init()

motor.set\_velocity(0, 0, 0, 0)

time\_step.time\_step\_call()

time\_step.time\_step\_call()



time\_step.time\_step\_call()

laser.init()

#laser.get\_laser\_scan\_data()

self.reward\_range = (-np.inf, np.inf)

self.action\_history1 = 0

self.action\_history2 = 0

self.action\_history3 = 0

for i in range(0, 5):

time\_step.time\_step\_call()

#get robot pose

pub\_get\_position\_req()

while robot\_global.get\_position\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_rotation\_req()

while robot\_global.get\_rotation\_res\_flag is False:

time\_step.time\_step\_call()

#get human pose

pub\_get\_human\_position\_req()

while human.get\_position\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_human\_rotation\_req()

while human.get\_rotation\_res\_flag is False:

time\_step.time\_step\_call()

rpos, rrot = getopposite(human.position, human.rotation, self.my\_case)

dist1 = getDisXZ(robot\_global.position.x, robot\_global.position.z, robot\_global.rotation.w, rpos.x, rpos.z, rrot.w)

rtang, rot1, rot2 = getanglefea(robot\_global.position.x, robot\_global.position.z, robot\_global.rotation.w,

rpos.x, rpos.z, rrot.w)

self.distp = dist1 #past distance

self.rot1p =rot1

self.rot2p = rot2

self.init\_dist\_pos = 0

self.wintimes\_dist\_key = 10

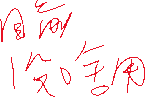
self.wintimes\_all = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

self.wintimes\_win = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

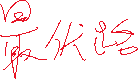
self.wintimes = 0

self.collisiontimes = 0

self.N = int(6.6 \* unit + 10)



self.M = int(9.9 \* unit + 10)



self.mp = self.init\_map\_hjktest()



self.lenp = 0

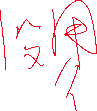
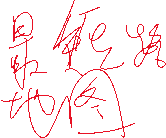
self.old\_robot\_postion = copy.deepcopy(robot\_global.position)

self.old\_robot\_rotation = copy.deepcopy(robot\_global.rotation)

def init\_map\_hjktest(self):

mp = np.zeros((self.N, self.M))

for i in range(int(4.5 \* unit), int(6.60 \* unit + 1)):



for j in range(int(0 \* unit), int(1.90 \* unit + 1)):



mp[i][j] = 1



for i in range(int(6.00 \* unit), int(6.60 \* unit + 1)):

for j in range(int(2.80 \* unit), int(5.00 \* unit + 1)):

mp[i][j] = 1

for i in range(int(2.30 \* unit), int(3.70 \* unit + 1)):

for j in range(int(3.15 \* unit), int(3.45 \* unit + 1)):

mp[i][j] = 1

for i in range(int(3.70 \* unit), int(4.10 \* unit + 1)):

for j in range(int(9.10 \* unit), int(9.90 \* unit + 1)):

mp[i][j] = 1

for i in range(int(1.60 \* unit), int(3.40 \* unit + 1)):

for j in range(int(8.50 \* unit), int(9.90 \* unit + 1)):

mp[i][j] = 1

for i in range(int(0 \* unit), int(1.00 \* unit + 1)):

for j in range(int(0 \* unit), int(3.30 \* unit + 1)):

mp[i][j] = 1

for i in range(int(0 \* unit), int(1.00 \* unit + 1)):

for j in range(int(4.40 \* unit), int(5.80 \* unit + 1)):

mp[i][j] = 1

for i in range(int(0 \* unit), int(1.40 \* unit + 1)):

for j in range(int(6.10 \* unit), int(7.90 \* unit + 1)):

mp[i][j] = 1

return mp

def distance(self, p1, p2):

dist = math.sqrt((p1.x-p2.x)\*(p1.x-p2.x)+(p1.y-p2.y)\*(p1.y-p2.y)+(p1.z-p2.z)\*(p1.z-p2.z))

return dist

def step(self, action):

if action == 0: # FORWARD

if self.action\_history1 != 0:

motor.set\_velocity(0, 0, 0, 0)

pub\_reset\_node\_physics\_req()

while robot\_global.reset\_node\_physics\_res\_flag is False:

time\_step.time\_step\_call()

motor.set\_velocity(5.0, 5.0, 5.0, 5.0)

elif action == 1: # LEFT forward

if self.action\_history1 != 3:

motor.set\_velocity(0, 0, 0, 0)

pub\_reset\_node\_physics\_req()

while robot\_global.reset\_node\_physics\_res\_flag is False:

time\_step.time\_step\_call()

motor.set\_velocity(4.0, 4.0, 7.0, 7.0)

elif action == 2: # RIGHT forward

if self.action\_history1 != 4:

motor.set\_velocity(0, 0, 0, 0)

pub\_reset\_node\_physics\_req()

while robot\_global.reset\_node\_physics\_res\_flag is False:

time\_step.time\_step\_call()

motor.set\_velocity(7.0, 7.0, 4.0, 4.0)

elif action == 3: # TURN LEFT

if self.action\_history1 != 1:

motor.set\_velocity(0, 0, 0, 0)

pub\_reset\_node\_physics\_req()

while robot\_global.reset\_node\_physics\_res\_flag is False:

time\_step.time\_step\_call()

motor.set\_velocity(-3.0, -3.0, 3.0, 3.0)

elif action == 4: # TURN RIGHT

if self.action\_history1 != 2:

motor.set\_velocity(0, 0, 0, 0)

pub\_reset\_node\_physics\_req()

while robot\_global.reset\_node\_physics\_res\_flag is False:

time\_step.time\_step\_call()

motor.set\_velocity(3.0, 3.0, -3.0, -3.0)

self.action\_history3 = self.action\_history2

self.action\_history2 = self.action\_history1

self.action\_history1 = action

for i in range(0, 4):

time\_step.time\_step\_call()

laser\_data, done = laser.get\_laser\_scan\_data()

while done is False:

laser\_data, done = laser.get\_laser\_scan\_data()

time\_step.time\_step\_call()

laser\_state, is\_collision = discretize\_observation(laser\_data, self.laser\_dim, self.collision\_threshold)

# get robot pose

pub\_get\_position\_req()

while robot\_global.get\_position\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_rotation\_req()

while robot\_global.get\_rotation\_res\_flag is False:

time\_step.time\_step\_call()

# get human pose

pub\_get\_human\_position\_req()

while human.get\_position\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_human\_rotation\_req()

while human.get\_rotation\_res\_flag is False:

time\_step.time\_step\_call()

# get right opposite pose to human

rpos, rrot = getopposite(human.position, human.rotation, self.my\_case)

rtang, rot1, rot2 = getanglefea(robot\_global.position.x, robot\_global.position.z, robot\_global.rotation.w, rpos.x, rpos.z, rrot.w)

dist1 = getDisXZ(robot\_global.position.x, robot\_global.position.z, robot\_global.rotation.w, rpos.x, rpos.z,

rrot.w)

# state = laser\_state + [dist1, rot1, rot2] + action\_history

#print ("from robot to person: ", robot\_global.position.x, robot\_global.position.z, robot\_global.rotation.w,

# human.position.x, human.position.z, human.rotation.w, dist1, rot1)

# print ("State : " + str(state) + " Action : " + str(action))

state = changeStateFromEnvToNetwork(laser\_state, dist1, rot1, rot2, action)

'''

distance = getDisXZ(robot\_global.position.x, robot\_global.position.z, robot\_global.rotation.w, self.old\_robot.position.x, self.old\_robot.position.z, self.old\_robot.rotation.w)

#distance = getDisXZ(robot\_global.position.x, robot\_global.position.z, self.old\_robot.position.x, self.old\_robot.position.z, self.old\_robot.rotation.w)

ACTION\_LOG\_FILE = open("my\_net18/actionLog\_" + strftime("%Y-%m-%d-%H-%M-%S", gmtime()) + ".txt", 'a+')

#print("???", file = ACTION\_LOG\_FILE)

print("postion change : ", distance, file=ACTION\_LOG\_FILE)

doubletmp = 1.0 \* (robot\_global.rotation.w - self.old\_robot.rotation.w)

if doubletmp >= 2.0 \* PI - 0.000001:

doubletmp = 0

if doubletmp <= -2.0 \* PI + 0.000001:

doubletmp = 0

print("rotation value change : ", doubletmp, file=ACTION\_LOG\_FILE)

sys.stdout.flush()

self.old\_robot\_x = copy.deepcopy(robot\_global)

'''

done = is\_collision

if done is True :

reward = FAILREWARD

self.collisiontimes += 1

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

print("----------------------------------")

print("----------------------------------")

print ("-----NO! You collide it!------------")

print("----------------------------------")

print("----------------------------------")

print("----------------------------------")

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

print("------------------------------------------------------------------------------")

else :

reward = 0

tmp\_par = dist1

if tmp\_par > FACINGDIS\_RANGE:

tmp\_par = FACINGDIS\_RANGE

print("hjk--- robot: ", robot\_global.rotation.w, rrot.w)

print("hjk--- delta dis last - now : ", self.distp - dist1)

print("hjk--- delta angle last - now : ", math.fabs(self.rot1p), math.fabs(rot1), math.fabs(self.rot1p) - math.fabs(rot1))



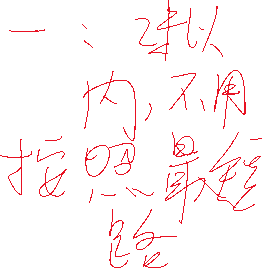
print("hjk--- 2222222 delta angle last - now : ", math.fabs(self.rot2p), math.fabs(rot2), math.fabs(self.rot2p) - math.fabs(rot2))



if dist1 < 2.00:



reward = tmp\_par \* ((self.distp - dist1) \* RREWARDDIS + (math.fabs(self.rot1p) - math.fabs(rot1)) \* RREWARDANGLE) + (FACINGDIS\_RANGE - tmp\_par) \* RREWARDFACINGANGLE \* (math.fabs(self.rot2p) - math.fabs(rot2)) # 0 + (self.difap - difa1)\*10



print ("hjk--reward: ", tmp\_par \* ((self.distp - dist1) \* RREWARDDIS + (math.fabs(self.rot1p) - math.fabs(rot1)) \* RREWARDANGLE), (FACINGDIS\_RANGE - tmp\_par) \* RREWARDFACINGANGLE \* (math.fabs(self.rot2p) - math.fabs(rot2)))

# print ("distance of past and this frame: " + str(self.distp) + " " + str(dist1))

# print ("angle difference of past and this frame: " + str(self.difap) + " " + str(difa1))

print ("Rewards: " + str(reward))

self.lenp = dist1

else:

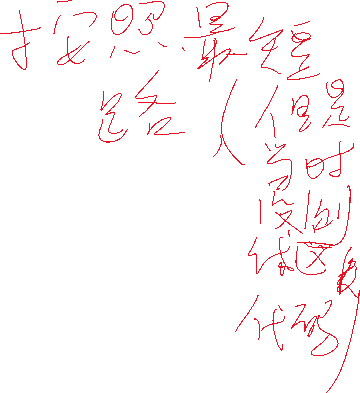
myspfa = spfa(self.mp, self.N, self.M, robot\_global.position.x, robot\_global.position.z, rpos.x, rpos.z)



now\_l, now\_g\_x, now\_g\_z = myspfa.getKey()

ang, \_, \_ = getanglefea(robot\_global.position.x, robot\_global.position.z, robot\_global.rotation.w, now\_g\_x, now\_g\_z, 0)

reward = (self.lenp - now\_l) \* RREWARDDIS + (PI / 2.0 - math.fabs(ang)) \* RREWARDANGLE



print("hjk--reward: ", (self.lenp - now\_l) \* RREWARDDIS, (PI / 2.0 - math.fabs(ang)) \* RREWARDANGLE, reward)

self.lenp = now\_l

# time.sleep(10000000)

actionout = open(actionoutPath, 'a+')

print("action", action, "postionchange:", getDisXZ(robot\_global.position.x, robot\_global.position.z, 0, self.old\_robot\_postion.x, self.old\_robot\_postion.z, 0), "rotationchange:", np.fabs(robot\_global.rotation.w - self.old\_robot\_rotation.w), file=actionout)

sys.stdout.flush()

actionout.close()

self.old\_robot\_postion = copy.deepcopy(robot\_global.position)

self.old\_robot\_rotation = copy.deepcopy(robot\_global.rotation)

self.distp = dist1

self.rot1p = rot1

self.rot2p = rot2

if dist1 < ARRIVEDIS and math.fabs(rot2) < ARRIVEANGLE:

reward = WINREWARD

print (dist1)

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print ("\*\*\*\*\*OK! You shoot it!\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

self.wintimes\_win[self.init\_dist\_pos] = self.wintimes\_win[self.init\_dist\_pos] + 1

self.wintimes = self.wintimes + 1

done = True

#exit()

print("\*\*\*\*\*\*\*\*\*\*\* hjk-- already shot win for all episode: ", self.wintimes, "collision: ", self.collisiontimes, self.wintimes\_all, self.wintimes\_win)

return np.asarray(state), reward, done, {}

def reset(self, testxml = 0, xml\_human\_x = -1, xml\_human\_y = -1, xml\_human\_rotation\_z = -1, xml\_robot\_x = -1, xml\_robot\_y = -1, xml\_robot\_rotation\_z = -1):

if testxml == 0:

for random\_iter in range(20):

if self.my\_case == -1:

print("hjktest---- reset this", random\_iter)

randomrobotINT = random.randint(0, 9)

if randomrobotINT == 0:

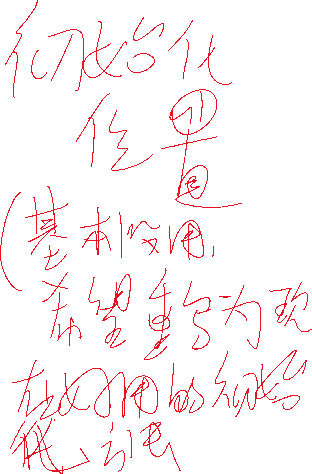
robotx = 1.5

roboty = 0.1

robotz = 1.5

elif randomrobotINT == 1:

robotx = 2



roboty = 0.1

robotz = 1.5

elif randomrobotINT == 2:

robotx = 3

roboty = 0.1

robotz = 1.5

else:

robotx = random.random() \* (5.5 - 2) + 2

roboty = 0.1

robotz = random.random() \* (8 - 4) + 4

#robotx = random.random() \* (5 - 1.5) + 1.5

#roboty = 0.1

#robotz = random.random() \* (8 - 1.5) + 1.5

'''

robotx = random.random() \* (5.5 - 3.8) + 3.8

roboty = 0.1

robotz = random.random() \* (4.5 - 2.7) + 2.7

'''

robot\_rotat\_x = 0

robot\_rotat\_y = 1

robot\_rotat\_z = 0

robot\_rotat\_w = random.uniform(-PI, PI)#-3.14, 3.14) #-PI\*4/5

randomINT =random.randint(0, 4)

if randomINT == 0:

humanx = 3

humany = 1.27

humanz = 6

elif randomINT == 1:

humanx = 4.5

humany = 1.27

humanz = 7.5

elif randomINT >= 2 and randomINT <= 4:

humanx = 2.5

humany = 1.27

humanz = 2

else:

humanx = random.random()\*(4 - 2.5) + 2.5

humany = 1.27

humanz = random.random()\*(7.5 - 2.5) + 2.5

else:

robotx = my\_case\_robotx[self.my\_case]

roboty = my\_case\_roboty[self.my\_case]

robotz = my\_case\_robotz[self.my\_case]

robot\_rotat\_x = my\_case\_robot\_rotat\_x[self.my\_case]

robot\_rotat\_y = my\_case\_robot\_rotat\_y[self.my\_case]

robot\_rotat\_z = my\_case\_robot\_rotat\_z[self.my\_case]

robot\_rotat\_w = my\_case\_robot\_rotat\_w[self.my\_case]

humanx = my\_case\_humanx[self.my\_case]

humany = my\_case\_humany[self.my\_case]

humanz = my\_case\_humanz[self.my\_case]

###important!!!!!

global global\_rn

global\_rn = random.random()

### room

position = Vector3()

position.x = robotx

position.y = roboty

position.z = robotz

rotation = Quaternion()

rotation.x = robot\_rotat\_x

#rotation.y = random.uniform(-3.14, 3.14)

rotation.y = robot\_rotat\_y

rotation.z = robot\_rotat\_z

rotation.w = robot\_rotat\_w

human\_pose = Vector3()

human\_pose.x = humanx

human\_pose.y = humany

human\_pose.z = humanz

rpos, rrot = getopposite(human\_pose, human.rotation, self.my\_case)

dist1 = getDisXZ(position.x, position.z, rotation.w, rpos.x, rpos.z,

rrot.w)

distRtoH = getDisXZ(position.x, position.z, rotation.w, human\_pose.x, human\_pose.z, 0)

if self.my\_case != -1 or (dist1 < DISFROMSTARTTOEND and distRtoH > MINDISFROMSTARTTOEND):

break

elif testxml == 1:

position = Vector3()

position.x = xml\_robot\_x

position.y = 0.1

position.z = xml\_robot\_y

rotation = Quaternion()

rotation.x = 0

# rotation.y = random.uniform(-3.14, 3.14)

rotation.y = 1

rotation.z = 0

rotation.w = xml\_robot\_rotation\_z

human\_pose = Vector3()

human\_pose.x = xml\_human\_x

human\_pose.y = 1.27

human\_pose.z = xml\_human\_y

global global\_rn

if xml\_human\_rotation\_z == 1:

global\_rn = 0

elif xml\_human\_rotation\_z == 2:

global\_rn = 0.6

elif xml\_human\_rotation\_z == 3:

global\_rn = 0.2

elif xml\_human\_rotation\_z == 4:

global\_rn = 0.9

done = motor.set\_velocity(0, 0, 0, 0)

while done is False:

time\_step.time\_step\_call()

done = motor.set\_velocity(0, 0, 0, 0)

pub\_reset\_node\_physics\_req()

while robot\_global.reset\_node\_physics\_res\_flag is False:

time\_step.time\_step\_call()

pub\_set\_position\_req(position)

while robot\_global.set\_position\_res\_flag is False:

time\_step.time\_step\_call()

pub\_set\_rotation\_req(rotation)

while robot\_global.set\_rotation\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_human\_position\_req()

while human.get\_position\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_human\_rotation\_req()

while human.get\_rotation\_res\_flag is False:

time\_step.time\_step\_call()

#print('hjk--- not set random :', human.position.x, human.position.y, human.position.z, human.rotation.x, human.rotation.y, human.rotation.z, human.rotation.w)

pub\_set\_human\_pose\_req(human\_pose)

print("hjk--lll :", human.set\_human\_pose\_res\_flag)

while human.set\_human\_pose\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_human\_position\_req()

while human.get\_position\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_human\_rotation\_req()

while human.get\_rotation\_res\_flag is False:

time\_step.time\_step\_call()

print('hjk--- have set random :', human.position.x, human.position.y, human.position.z)

laser\_data, done = laser.get\_laser\_scan\_data()

while done is False:

laser\_data, done = laser.get\_laser\_scan\_data()

time\_step.time\_step\_call()

laser\_state, is\_collision = discretize\_observation(laser\_data, self.laser\_dim, self.collision\_threshold)

for i in range(0, 10):

time\_step.time\_step\_call()

###############

self.action\_history1 = 0

self.action\_history2 = 0

self.action\_history3 = 0

#get human pose

pub\_get\_human\_position\_req()

while human.get\_position\_res\_flag is False:

time\_step.time\_step\_call()

pub\_get\_human\_rotation\_req()

while human.get\_rotation\_res\_flag is False:

time\_step.time\_step\_call()

# self.mp = np.zeros((self.N,self.M))

rpos, rrot = getopposite(human.position, human.rotation, self.my\_case)

dist1 = getDisXZ(robot\_global.position.x, robot\_global.position.z, robot\_global.rotation.w, rpos.x, rpos.z, rrot.w)

rtang, rot1, rot2 = getanglefea(position.x, position.z, rotation.w, rpos.x, rpos.z, rrot.w)

print("11111 state dis : ", dist1 \* 10)

state = changeStateFromEnvToNetwork(laser\_state, dist1, rot1, rot2, -1)

self.distp = dist1

self.lenp = dist1

self.rot1p = rot1

self.rot2p = rot2

# self.old\_robot = robot\_global

self.old\_robot\_postion = copy.deepcopy(robot\_global.position)

self.old\_robot\_rotation = copy.deepcopy(robot\_global.rotation)

self.init\_dist\_pos = int(self.distp / 0.5)

#self.init\_dist\_pos = self.init\_dist / 0.5

if self.init\_dist\_pos > self.wintimes\_dist\_key:

self.init\_dist\_pos = self.wintimes\_dist\_key

print("wwwww????? : ", self.init\_dist\_pos, self.distp)

self.wintimes\_all[self.init\_dist\_pos] = self.wintimes\_all[self.init\_dist\_pos] + 1

return np.asarray(state)