Project 7 无线信道冲突避免（1）

项目报告

摘要

在本项目中，基于 CSMA/CA 协议，我们使用 python 搭建了一个无线通信系统，并实现了对 CSMA/CA 协议内容的模拟与仿真。我们将多智能体强化学习（MARL）算法 MADDPG 引入通信系统中，还尝试了将多智能体强化学习降维为单智能体强化学习，在通信系统中实现了 Q-learning、Sarsa 等算法，使系统性能在传输延迟、丢包率、重传率等多个方面得到提升。

关键词：CSMA/CA，强化学习（RL），多智能体强化学习（MARL）

Abstract

In this project, based on csma/ca protocol, we use Python to build a wireless communication system, and realize the simulation and Simulation of csma/ca protocol content. We introduced the Multi-Agent Reinforcement Learning (marl) algorithm maddpg into the communication system, and also tried to reduce the dimension of Multi-Agent Reinforcement Learning to single agent reinforcement learning. We implemented Q-learning, sarsa and other algorithms in the communication system, which improved the system performance in many aspects, such as transmission delay, packet loss rate, retransmission rate and so on.

Key words: CSMA/CA, Reinforcement Learning (RL), Multi-Agent Reinforcement Learning (MARL)

1 项目介绍

1.1 项目背景

无线局域网为了避免通信设备在传输数据时发生冲突，通常采用 CSMA/CA 协议以协调各设备的传输时间。然而，CSMA/CA 协议并不能完全避免冲突的发生，而且由于要在各设备传输数据之前设置等待窗口而降低了通信效率。目前，已经有研究将多智能体强化学习应用于信道规划中，其以提高整体系统的通信效率为目标，各设备将学习到各自最佳的传输时间。本项目将利用模拟仿真的方式搭建一套通信系统，分别利用 CSMA/CA 协议与多智能体强化学习模型达到避免设备间发生冲突的目的，并进一步比较两者对通信系统整体性能的提升程度。

1.2 项目目标

① 搭建通信系统（多个设备，发送的消息具有不同的大小与优先级等），将 CSMA/CA 协议应用于该系统并进行性能测试。

② 搭建多智能体强化学习模型，将其用于①中搭建的通信系统并进行性能测试。

③ 与 Project 8 合作，探究在假定各设备能较为准确地预测未来信道占用情况的前提下，多智能体强化学习模型的效果是否能进一步提升。

2 CSMA/CA

2.1 协议介绍

CSMA/CA（Carrier Sense Multiple Access with Collision Avoid，带有冲突避免机制的载波侦听多路访问）是一种数据传输时避免各站点之间数据传输冲突的算法，其特点是发送包的同时不能检测到信道上有无冲突，只能尽量“避免”。802.11 局域网在使用 CSMA/CA 的同时，还使用停止等待协议。这是因为无线信道的通信质量远不如有线信道，因此无线站点每通过无线局域网发送完一帧后，要等到收到对方的确认帧后才能继续发送下一帧，该操作称为链路层确认。

2.2 二进制指数退避

二进制指数退避技术（Binary Exponential Back off），指在遇到重复的冲突时，站点将重复传输，但在每一次冲突之后，等待时延的平均值将加倍。二进制指数退避算法提供了一种处理重负荷的方法。尝试传输的重复失败导致更长的退避时间，这将有助于负荷的平滑。如果没有这样的退避，以下状况可能发生：两个或多站点同时尝试传输，这将导致冲突，之后这些站点又立即尝试重传，导致一个新冲突的发生。

2.3 ACK

ACK（Acknowledge character），即确认字符，是在数据通信过程中接收端发给发送端的一种传输类控制字符，表示发来的数据已确认接收无误。接收端对所收到的数据包进行检查，若未发现错误，便向发送端发出确认回答 ACK，表明信息已被正确接收，并准备好接收下一个数据包。该控制字符可由中心结点发送，也可由远地结点发送。

通常，CSMA/CA 利用 ACK 信号来避免冲突的发生，也就是说，只有当客户端收到网络上返回的 ACK 信号后，才确认送出的数据已经正确到达目的地。

2.4 RTS/CTS 协议

RTS/CTS 协议（Request To Send/Clear To Send），即请求发送/允许发送协议，相当于一种握手协议，主要用来解决“隐藏终端”问题。“隐藏终端”问题经常发生在大容量文件传输过程中，这将导致效率损失，并需要错误恢复机制。对该问题，IEEE802.11 提供了如下解决方案。

在参数配置中，若使用 RTS/CTS 协议，同时设置传输字节数上限——一旦要传输的数据大于该上限值，即启动 RTS/CTS 协议：

首先，A 向 B 发送 RTS 信号，表示 A 要向 B 发送若干数据。B 在接收到 RTS 后，向所有基站发送 CTS 信号，表示其已准备就绪，A 可以开始发送，而其余欲向 B 发送数据的基站则暂停发送。双方在成功交换 RTS/CTS 信号（即握手完成）后才开始真正的数据传输，确保了当多个互不可见的发送站点同时向同一接收站点发送信号时，实际只能是从接收站点接收到 CTS 响应的站点才能进行信号发送，避免了冲突发生。

3 强化学习和多智能体强化学习

3.1 强化学习（RL）

强化学习（Reinforcement Learning, RL），机器学习的范式和方法论之一，用于描述和解决智能体（agent）在与环境的交互过程中通过学习策略以达成回报最大化或实现特定目标的问题 。不同于有监督学习，在强化学习中，我们对不同的行为进行不同的奖励，而不会为这些行为标上“对”与“错”。智能体以“试错”的方式进行学习，使得奖赏最大化。强化学习的常见模型是标准的马尔可夫决策过程，图 3-1 展示了马尔可夫决策过程的基本建模框架。

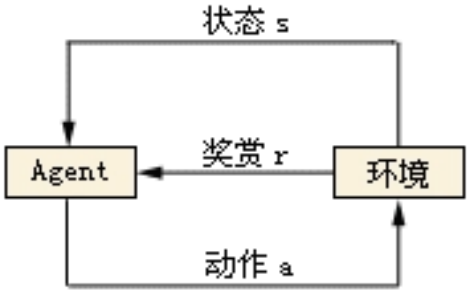

图 3-1 马尔可夫决策过程建模框架

3.2 多智能体强化学习（MARL）

多智能体强化学习（Multi-Agent Reinforcement Learning，MARL）是由 RL 和多智能体系统结合而成的新领域。一组具有自我控制能力、能够相互作用的智能体，在同一环境下通过感知器、执行器操作，进而形成完全合作性、完全竞争性或者混合类型的多智能体系统，每个多智能体的奖励都会受到其他智能体动作的影响。MARL 的目标在于学习到一种策略使得系统达到均衡稳态。多智能体强化学习的常见模型是马尔可夫博弈过程，图 3-2 展示了马尔可夫博弈过程的基本建模框架。

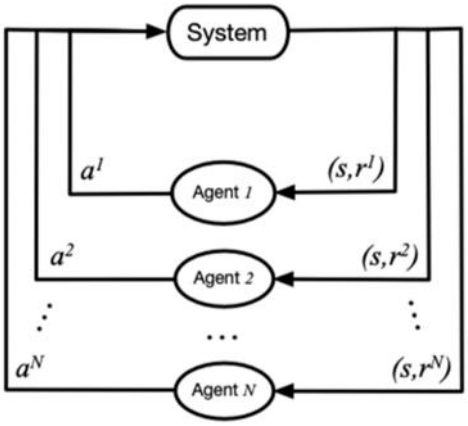

图 3-2 马尔可夫博弈过程建模框架

4 CSMA/CA 通信系统实现

4.1 CSMA/CA 基本流程

CSMA/CA 协议原理如图 4-1 所示，基本流程如图 4-2 所示。对于每个待发送的数据帧，当监听到信道空闲时，发送节点会在等待 DIFS 时间后发送 RTS 信号以预约信道；此后如果在超时之前收到由接收端发送的 CTS 信号，则发送端将在等待 SIFS 时间后开始发送数据帧；若在发送完数据帧后一段时间内收到 ACK 信号，则认为此次发送成功；否则进入重传规则，重新计算退避时间后继续监听信道，重复以上过程。若重传次数大于设定阈值，则放弃此数

据包的传输，并归类为传输失败。

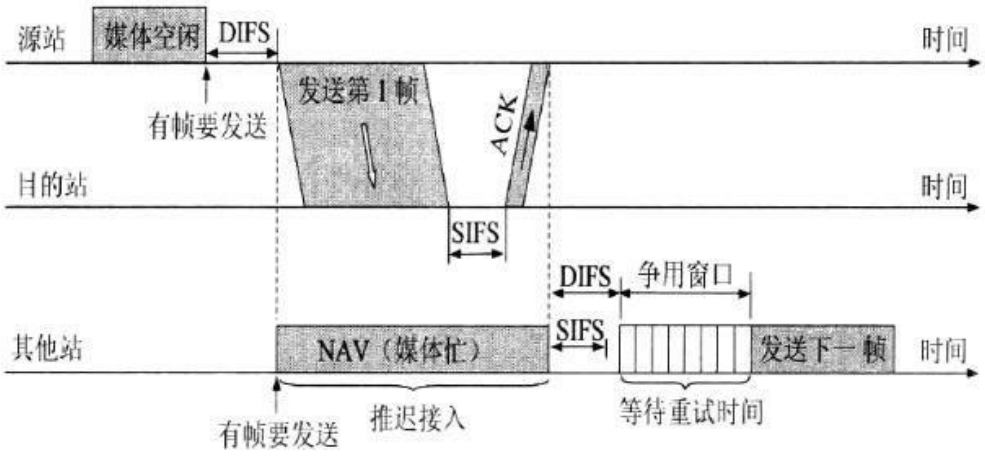

图 4-1 CASA/CA 协议原理图

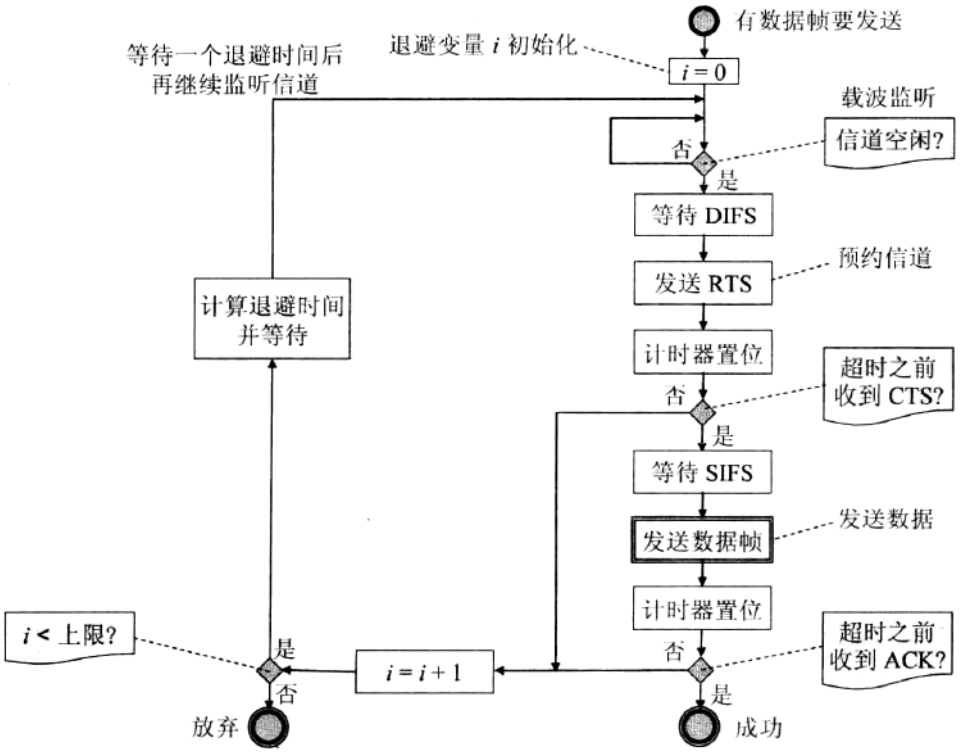

图 4-2 CSMA/CA 基本流程图

4.2 代码整体结构

“CSMA-CA-仿真代码”文件夹下的代码结构如表 4-1 所示：

表 4-1 “CSMA-CA-仿真代码”代码整体结构

Python 文件名	作用
parameters.py	参数配置程序。关键参数：仿真时间（纳秒），节点数，比特率，丢包率，最大重传次数等等
ether.py	信道环境的定义、距离计算、传输延迟和信号衰减的引入
mac.py	MAC 层行为定义
macpacket.py	MAC 层封包定义
phy.py	PHY 层行为定义
phypacket.py	PHY 层封包定义
node.py	节点类定义
stats.py	作图程序代码
main.py	主程序代码

程序输出在文件夹 results_yyyy_mm_dd-hh-mm-ss 中，输出内容列举在表 4-2 中：

表 4-2 程序输出内容

文件名	内容
simulation.log	程序的控制台输出（重定向到 log 文件中）
position.pdf	节点位置图
delays.pdf	收发包的延迟情况
packets.pdf	收发包的数量情况
retransmissions.pdf	重传包的数量累计
failed_packets.pdf	传输失败包的数量累计

4.3 关键代码解释

4.3.1 关键参数配置：parameters.py

用于配置程序运行中的关键参数。其中，关键参数如下：

-SIM_TIME=2*1e9 单位为纳秒，也就是 2 秒。此参数不能调整至 1e9（1 秒）以下，会导致程序报错。

-NUMBER_OF_NODES=4 节点数暂时设定为 4，调大此参数会导致程序的运行时间增加。程序默认每个节点都同时具有收发功能。

-PACKET_LOSS_RATE 丢包率，此处设定为 1%。

-BITRATE 比特率。若想程序运行的更快，可以减小比特率，相当于提升每个事件的持续时间与间隔时间，减小总事件数，在客观上可以加速程序运行。可用于程序的调试过程。

-NOISE_FLOOR 底噪。噪声基底（Noise floor）指的是接收信噪比为 0dB 时，接收机能够感知的最小信号强度。

-FREQUENCY 频率。由于频率会影响波长，因此调整此参数会影响到传输中信号功率的衰

减（采用 FSPL 传输功率衰减模型），从而影响传输失败的几率。
-MAX_RETRANSMITION_TIME 最大重传次数。此处设定为 10。
-NODE_POSITION 节点位置指定。默认值为 None，即节点横纵坐标均为（0，40）内的随机数。

别的一些参数大部分是 CSMA-CA 协议设定的或是行业内常用参数，几乎不需要调整。

4.3.2 生成节点事件：KeepSendingIncreasing 函数

Keepsendingincreasing 函数如图 4-3 所示。此函数为节点类的函数。在 main 主程序生成节点的同时调用此函数，函数将持续生成每个节点需要发送的数据包。数据包的产生时间、目的节点、包的有效数据长度均为随机生成。

虽然各数据包间的间隔时间应满足泊松分布，但为了简化仿真过程，我们假定数据包是均匀产生的，即 finalRate=startingRate=5, increasingSpeed=0。

```
def keepSendingIncreasing(self, startingRate, finalRate, destinationNodes):
    rate = startingRate
    increasingSpeed = (finalRate - startingRate) / parameters.SIM_TIME
    while True:
        yield self.env.timeout(round(random.expovariate(startingRate + increasingSpeed * self.env.now) * 1e9))

        destination = destinationNodes[random.randint(0, len(destinationNodes)-1)]
        length = random.randint(0, parameters.MAX_MAC_PAYLOAD_LENGTH)       # PAYLOAD_LENGTH
        id = str(self.env.now) + '_' + self.name + '_' + destination
        if parameters.PRINT_LOGS:
            print('Time %d: %s sends %s to %s' % (self.env.now, self.name, id, destination))
        yield self.env.process(self.mac.send(destination, length, id))
```

图 4-3 KeepSendingIncreasing 函数

4.3.3 数据包发送流程

在 KeepSendingIncreasing 函数的末尾调用了 MAC 层的 send 函数，即在 MAC 层准备发送信号。send 函数中首先检查 PHY 层是否有信号正在发送，若有则需等待发送完后再进一步处理，如图 4-4 所示。

```
def send(self, destination, payloadLength, id):
    length = payloadLength + parameters.MAC_HEADER_LENGTH
    macPkt = macPacket.MacPacket(self.name, destination, length, id, False)
    self.stats.logGeneratedPacket(id, self.env.now)
    self.retransmissionCounter[macPkt.id] = 0

    # sensing phase
    if self.phy.isSending:   # I cannot sense while sending
        yield self.phy.transmission   # wait for my phy to finish sending other packets

    if self.isSensing: # I'm sensing for another packet, I wait
        yield self.sensing

    self.sensing = self.env.process(self.waitIdleAndSend(macPkt))
    yield self.sensing
```

图 4-4 MAC 层的 send 函数

若 PHY 层目前空闲，则进入争用窗口，计算等待时间，由等待时间最短者调用 PHY 层的 send 函数准备开始发送，其余数据包则计算得到新的等待时间，如图 4-5、图 4-6 所示。PHY 层的 send 函数会打断当前节点对信道的监听，即 listen 函数，并调用封装与传输函数 encapsulateAndTransmit 准备开始发送数据包，如图 4-7 所示。

```
def waitIdleAndSend(self, macPkt):
    # 闲置等待并发送
    self.isSensing = True
    timeout = parameters.DIFS_DURATION
    backoff = 0
    if self.retransmissionCounter[macPkt.id] != 0:  # add backoff in case of retransmission
        backoff = random.randint(0, min(pow(2,self.retransmissionCounter[macPkt.id])-1)*parameters.CW_MIN, parameters.CW_M
        timeout += backoff
    while True:
        try:
            while timeout > 0:
                yield self.env.timeout(1)
                timeout -= 1

                # sensing phase
                if self.phy.isSending:   # I cannot sense while sending
                    yield self.phy.transmission   # wait for my phy to finish sending other packets
                    timeout = parameters.DIFS_DURATION + backoff    # if a trasmission occours during the sensing I resta

            self.phy.send(macPkt)
            self.pendingPackets[macPkt.id] = self.env.process(self.waitAck(macPkt))
            self.isSensing = False
            return
        except simpy.Interrupt:
            if backoff == 0:    # need to add backoff, even if this is not a retransmission
                backoff = random.randint(0, parameters.CW_MIN-1) * parameters.SLOT_DURATION
                timeout = parameters.DIFS_DURATION + backoff
            elif timeout > backoff: # backoff has not been consumed, new timeout is DIFS + backoff
                timeout = parameters.DIFS_DURATION + backoff
            else:   # backoff has been partially consumed, new timeout is DIFS + remaining backoff
                backoff = timeout
                timeout = parameters.DIFS_DURATION + backoff
            continue
```

图 4-5 MAC 层的 waitidleandsend 函数

```
def send(self, macPkt):
    if not self.isSending:  # I do not send if I'm already sending
        self.listen.interrupt(macPkt)
```

图 4-6 PHY 层的 send 函数

```
def listen(self):
    self.inChannel = self.ether.getInChannel(self)
    yield self.env.timeout(parameters.RADIO_SWITCHING_TIME) # simulate time of radio switching
    if parameters.PRINT_LOGS:
        print('Time %d: %s starts listening' % (self.env.now, self.name))
    while True:
        try:
            (phyPkt, beginOfPacket, endOfPacket) = yield self.inChannel.get()
            # the signal just received will interfere with other signals I'm receiving (and vice versa)
            for receivingPkt in self.receivingPackets:
                if receivingPkt != phyPkt:
                    receivingPkt.interferingSignals[phyPkt.macPkt.id] = phyPkt.power
                    phyPkt.interferingSignals[receivingPkt.vmacPkt.id] = receivingPkt.power
            if endOfPacket and phyPkt.macPkt.id.split('_')[-1]==self.name:
                print('Time %d: %s receives signal %s from %s with power %e' % (self.env.now, self.name, phyPkt.macPkt.id, phyPkt.macPkt.s
            if self.mac.isSensing:  # interrupt mac if it is sensing for idle channel
                self.mac.sensing.interrupt()
            if phyPkt.power > parameters.RADIO_SENSITIVITY: # decodable signal
                if beginOfPacket:  # begin of packet
                    self.receivingPackets.append(phyPkt)
                elif endOfPacket:   # end of packet
                    if phyPkt in self.receivingPackets:
                        self.receivingPackets.remove(phyPkt)
                        if not phyPkt.corrupted:
                            sinr = self.computeSinr(phyPkt)
                            if sinr > 10:    # signal greater than noise and inteference
                                self.env.process(self.mac.handleReceivedPacket(phyPkt.macPkt))

        except simpy.Interrupt as macPkt:       # listening can be interrupted by a message sending
            self.isSending = True
            self.transmission = self.env.process(self.encapsulateAndTransmit(macPkt.cause))
            yield self.transmission
            self.isSending = False
```

图 4-7 PHY 层的 listen 函数

在封装与传输函数 encapsulateAndTransmit 中，发射节点在切换到传输模式后首先发送 RTS 信号。若在规定时间内收到了 CTS 信号，则开始数据包的发送，即 ether 中的 transmit 函数，在每一个时隙中均发送数据，直到数据发送完成为止，如图 4-8、图 4-9 所示。

```
def encapsulateAndTransmit(self, macPkt):   # 封装和传输
    self.receivingPackets.clear() # I switch to transmitting mode, so I drop all ongoing receptions
    yield self.env.timeout(parameters.RADIO_SWITCHING_TIME)
    self.ether.removeInChannel(self.inChannel, self)
    yield self.env.timeout(parameters.RADIO_SWITCHING_TIME) # simulate time of radio switching
    if parameters.PRINT_LOGS:
        print('Time %d: %s stops listening' % (self.env.now, self.name))
    phyPkt = phyPacket.PhyPacket(parameters.TRANSMITTING_POWER, False, macPkt) # start of packet

    if not macPkt.ack:
        if parameters.PRINT_LOGS:
            print('Time %d: %s sends RTS signal to %s' % (self.env.now, self.name, phyPkt.macPkt.id.split('_')[-1]))
        yield self.env.timeout(parameters.RADIO_SWITCHING_TIME) # simulate time of radio switching
        if parameters.PRINT_LOGS:
            print('Time %d: %s receives CTS signal from %s' % (self.env.now, self.name, phyPkt.macPkt.id.split('_')[-1]))

    if macPkt.ack:
        if parameters.PRINT_LOGS:
            print('Time %d: %s PHY starts transmission of %s ACK' % (self.env.now, self.name, phyPkt.macPkt.id))
    else:
        if parameters.PRINT_LOGS:
            print('Time %d: %s PHY starts transmission of %s' % (self.env.now, self.name, phyPkt.macPkt.id))
    self.ether.transmit(phyPkt, self.latitude, self.longitude, True, False) # beginOfPacket=True, endOfPacket=False

    duration = macPkt.length * parameters.BIT_TRANSMISSION_TIME + parameters.PHY_HEADER_LENGTH
```

图 4-8 PHY 层的 encapsulateAndTransmit 函数（1）

```
while True:
    if duration < parameters.SLOT_DURATION:
        yield self.env.timeout(duration)    # wait only remaining time
        break
    yield self.env.timeout(parameters.SLOT_DURATION) # send a signal every slot
    self.ether.transmit(phyPkt, self.latitude, self.longitude, False, False)  # beginOfPacket=False, endOfPacket=Fals
    duration -= parameters.SLOT_DURATION

self.ether.transmit(phyPkt, self.latitude, self.longitude, False, True)  # beginOfPacket=False, endOfPacket=True
if macPkt.ack:
    if parameters.PRINT_LOGS:
        print('Time %d: %s PHY ends transmission of %s ACK' % (self.env.now, self.name, phyPkt.macPkt.id))
else:
    if parameters.PRINT_LOGS:
        print('Time %d: %s PHY ends transmission of %s' % (self.env.now, self.name, phyPkt.macPkt.id))

self.inChannel = self.ether.getInChannel(self)
yield self.env.timeout(parameters.RADIO_SWITCHING_TIME) # simulate time of radio switching
if parameters.PRINT_LOGS:
    print('Time %d: %s starts listening' % (self.env.now, self.name))
```

图 4-9 PHY 层的 encapsulateAndTransmit 函数（2）

接下来，对每个时隙中发送的数据，ether 会计算两节点间的距离、由 FSPL 模型得到信道的衰减、接收功率与丢包情况，并回传到 PHY 层的 listen 函数中，如图 4-10 所示。listen 函数对接收到的所有数据包判断是否成功接收，判断指标包括是否丢包、接收功率是否大于门限、是否能从干扰与噪声中恢复数据等，如图 4-7 所示。

```
def computeDistance(self, senderLatitude, senderLongitude, receiverLatitude, receiverLongitude):
    return math.sqrt(pow(senderLatitude - receiverLatitude, 2) + pow(senderLongitude - receiverLongitude, 2))


def latencyAndAttenuation(self, phyPkt, sourceLatitude, sourceLongitude, destinationChannel, destinationNode, beginOfPack
    # 延迟和衰减
    distance = self.computeDistance(sourceLatitude, sourceLongitude, destinationNode.latitude, destinationNode.longitude)
    delay = round((distance / c) * pow(10, 9), 0)
    yield self.env.timeout(delay)
    receivingPower = parameters.TRANSMITTING_POWER * pow(parameters.WAVELENGTH/(4 * pi * distance), 2) # NB. used FSPL pr
    phyPkt.power = receivingPower

    if endOfPacket:
        if random.randint(0,100) < parameters.PACKET_LOSS_RATE * 100:
            phyPkt.corrupted = True

    return destinationChannel.put((phyPkt, beginOfPacket, endOfPacket))


def transmit(self, phyPkt, sourceLatitude, sourceLongitude, beginOfPacket, endOfPacket):
    events = [self.env.process(self.latencyAndAttenuation(phyPkt, sourceLatitude, sourceLongitude, destinationChannel, de
    return self.env.all_of(events)
```

图 4-10 computeDistance、latencyAndAttenuation 和 transmit 函数

若成功接收到数据包，则会将信息返回到 MAC 层的 handleReceivedPacket 函数，并由接收方向发送方发送 ACK 确认帧，如图 4-11 所示。ACK 确认帧将会经过与数据帧同样的上述流程。若 ACK 帧被成功接收，则数据包发送成功，流程结束。若发送方未在规定时间内接收到由接收方发送的 ACK 帧，且已有重传次数小于所设定的最大重传次数，则会调用

MAC 层中的 waitACK 函数进行重传，如图 4-12 所示。否则认为此数据包传输失败，放弃重传。

至此，一个数据包的全部传输流程结束。

```
def handleReceivedPacket(self, macPkt):
    # 成功收到信息
    if macPkt.destination == self.name and not macPkt.ack:  # send ack to normal packets
        if parameters.PRINT_LOGS:
            print('Time %d: %s MAC receives packet %s from %s and sends ACK' % (self.env.now, self.name, macPkt.id, macPkt
        self.node.receive(macPkt.id, macPkt.source)
        self.stats.logDeliveredPacket(macPkt.id, self.env.now)
        ack = macPacket.MacPacket(self.name, macPkt.source, parameters.ACK_LENGTH, macPkt.id, True)
        yield self.env.timeout(parameters.SIFS_DURATION)
        self.phy.send(ack)
    elif macPkt.destination == self.name:
        if parameters.PRINT_LOGS:
            print('Time %d: %s MAC receives ACK %s from %s' % (self.env.now, self.name, macPkt.id, macPkt.source))
        if macPkt.id in self.pendingPackets:    # packet could not be in pendingPackets if timeout has expired but ack st
            self.pendingPackets[macPkt.id].interrupt()
```

图 4-11 MAC 层的 handleReceivedPacket 函数

```
def waitAck(self, macPkt): # 重传
    try:
        yield self.env.timeout(parameters.ACK_TIMEOUT)
        # timeout expired, resend
        if macPkt.retransmitiontimes > parameters.MAX_RETRANSMITION_TIME:
            if parameters.PRINT_LOGS:
                self.stats.logfailedRetransmission(macPkt.id,self.env.now)
                print('Time %d: %s %s transmit to %s fails finally' % (self.env.now, self.name, macPkt.id, macPkt.destinat
        else:
            macPkt.retransmitiontimes += 1
            if parameters.PRINT_LOGS:
                print('Time %d: %s %s transmit timeout without ACK from %s' % (self.env.now, self.name, macPkt.id, macPkt
                print('Time %d: %s MAC retransmit %s to %s' % (self.env.now, self.name, macPkt.id, macPkt.destination))
            self.pendingPackets.pop(macPkt.id)
            self.retransmissionCounter[macPkt.id] += 1

            # sensing phase
            if self.phy.isSending:   # I cannot sense while sending
                yield self.phy.transmission   # wait for my phy to finish sending other packets
            if self.isSensing: # I'm sensing for another packet, I wait
                yield self.sensing

            self.stats.logRetransmission(self.name,self.env.now)
            self.sensing = self.env.process(self.waitIdleAndSend(macPkt))
    except simpy.Interrupt:
        # ack received
        self.pendingPackets.pop(macPkt.id)
        self.retransmissionCounter.pop(macPkt.id)
```

图 4-12 MAC 层的 waitAck 函数

5 MARL 在通信系统中的实现

5.1 基本定义

① 幕：

原定义一个包的完整传输为一幕，但我们认为单个包的传输质量和前后相邻包的传输有关， 如果前一个包在很近的时间内以很短的传输延迟传输成功了，那么当前的包就可以尝试用更大胆的短延迟去传输。所以定义本项目为持续性任务。

② 马尔可夫过程：

定义帧的一次重传为一个马尔可夫过程。从 MAC 层的传输/重传开始，直至 ACK 到达或者 ACK 超时结束。

③ 状态：

目标节点的 ID（不同的通信路线有不同的信号质量和延迟）；

当前帧的重传次数（重传次数多，则退避时间更长）；

包长度（若本包已经成功传输了很多帧，则可以尝试用更短的等待时长来尝试传输）

④ 动作：

退避时间的长度。由于退避时长是一个从 0 到数十万的大区间，是一个很大的动作空间，收敛起来会极其漫长，我们决定通过取 log2 的方式对动作空间进行降维。

也可以加入 DIFS、SIFS 和 AckTimeout 等数值量作为动作空间的一部分，但这样的话动作空间将由一维列向量变为二维矩阵，很难在实际算法中实现，除非对于每一个类型的动

作新分配一个强化学习任务，极大提高代码的运算量，故暂时对它们不予考虑。

⑤ 收益（reward）：

若 ack 超时，帧未被收到，则定义收益为-10；

若 ack 成功收到，则收益为$\frac{k}{AoI}$或$\frac{k}{AoS}$，也可以是$\frac{k}{AoI*(AckTimeout-Acktime)}$ 。

5.2 多智能体强化学习算法——MADDPG

5.2.1 算法介绍

MADDPG 算法是在深度确定策略梯度（Deep Deterministic Policy Gradient，DDPG）方法的基础上、对其中涉及到的 actor-critic 框架进行改进，使用集中式训练、分布式执行的机制（centralized training and decentralized execution），为解决多智能体问题提供了一种比较通用的思路。

MADDPG 为每个智能体都建立了一个中心化的 critic，它能够获取全局信息（包括全局状态和所有智能体的动作）并给出对应的值函数 $Q_i(x,a_1,...,a_n)$，这在一定程度上能够缓解多智能体系统环境不稳定的问题。另一方面，每个智能体的 actor 则只需要根据局部的观测信息做出决策，这能够实现对多智能体的分布式控制。

MADDPG 方法的网络模型图如图 5-1 所示。其中，绿色部分为中心化的 Q 值学习，红褐色部分为分布式的策略执行。Q 值获取所有智能体的观测信息 o 和动作 a，策略 π 根据个体的观测信息来输出个体动作。MADDPG 算法的伪代码如图 5-2 所示。

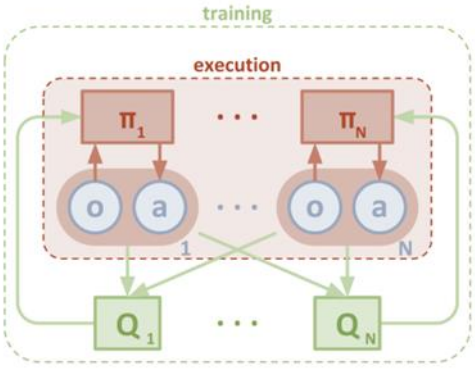

图 5-1　MADDPG 网络模型图

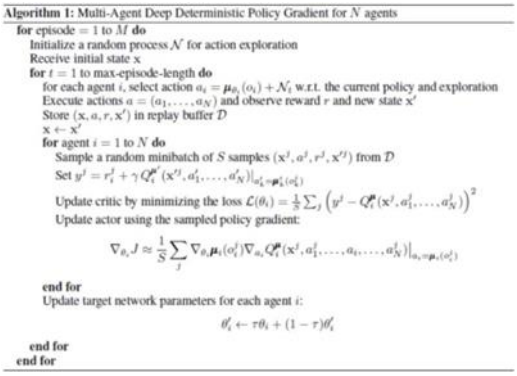

图 5-2　MADDPG 算法伪代码

5.2.2 算法实现

首先，我们搭建好 actor-critic 网络模型。其中 actor 网络与 critic 网络均为三层网络结构，且均为 Linear 层，输入均为状态、动作和隐藏层大小，隐藏层的加入可以使得网络拥有更好的评分能力或者概率生成能力。这两个网络结构的前两层的激活函数均为 relu 函数，不同之处在于 actor 网络第三层要经过 softmax 函数激活后输出，使得 actor 的输出值对应为满足概率的特性。而 critic 网络则是直接输出第三层的结果，即对行为进行“打分”的结果。

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_layer_size):
        super(ActorNet, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_layer_size)
        self.fc2 = nn.Linear(hidden_layer_size,hidden_layer_size)
        self.fc3 = nn.Linear(hidden_layer_size, action_dim)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x), dim=1)
        return x
```

图 5-3　actor 网络

```python
class CriticNet(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_layer_size):
        super(CriticNet, self).__init__()
        self.fc1 = nn.Linear(state_dim + action_dim, hidden_layer_size)
        self.fc2 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.fc3 = nn.Linear(hidden_layer_size, 1)

    def forward(self, state, action):
        state_action = torch.cat([state, action], 1)
        x = F.relu(self.fc1(state_action))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

图 5-4　critic 网络

随后我们构建多智能体的类。为了充分与 CSMA/CA 协议适配，我们将每一个 node 定义为一个智能体，CSMA/CA 协议中 node.py 文件中的函数封装为智能体的函数，智能体的状态与动作的定义如 5.1 节中所述。在运行过程中，critic 网络会输入所有智能体的动作与全局的状态，进而更新此智能体的 Q 值。Actor 网络则根据本智能体的状态和动作进行决策，决策依据为：若概率大于 0.8，则退避时间长度保持不变；若概率小于 0.8 而大于 0.2，则退避时间为退避时长/概率；若概率小于 0.2，则退避时间为 2^最大重传数。

```
qs = self.critic.forward(torch.cat((obs_t_flat, acts_t_flat), dim=1))
# print(qs)
# print(next_acts_t.size())
next_qs = self.critic_target.forward(torch.cat((next_obs_t_flat, next_acts_t_flat), dim=1))

# print("Next qs", next_qs)
# print("rew", rew)
targets = rew_t + self.gamma * next_qs * (1 - done_t)
# print(targets)

critic_loss = self.critic_criterion(qs, targets.detach())
# print(critic_loss)
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()

# print(acts.shape)
for b in range(batch_size):
    acts_t[b, self.id] = self.actor.forward(obs_t[b, self.id])
act_t_flat = acts_t.view(batch_size, -1)
actor_loss = -torch.mean(self.critic.forward(torch.cat((obs_t_flat, act_t_flat), dim=1)))
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()

self._soft_update_target_nets()
```

图 5.5　actor-critic 更新

5.3 将多智能体学习问题降维为单智能体学习问题

5.3.1 MARL 的缺点

① 维度爆炸：联结动作随智能体数量指数增长，因此多智能体系统维度非常大，计算复杂。

② 目标奖励确定困难：多智能体系统中每个智能体的任务可能不同，但是彼此之间又相互耦合影响。

③ 不稳定性：在多智能体系统中，多个智能体是同时学习的。当同伴的策略改变时，每个智能体自身的最优策略也可能会变化，这将对算法的收敛性带来影响。

④ 探索-利用：每个智能体的探索都可能对同伴智能体的策略产生影响，这将使算法很难稳定，学习速度慢。

5.3.2 强化学习算法

童乐等人[2]提到：“基于多智能体强化学习的动态频谱分配算法对 Q 表进行拆解，将多智能体学习问题降维为单智能体学习问题。在迭代过程中，每个智能体单独更新其 Q 值，频谱分配最优策略可以通过所有虚拟智能体的 Q 值的和来表征。”由于在 CSMA/CA 协议中，每个智能体间为竞争的关系，我们只需对每个智能体采用普通的 Q 学习方法，并且都采取贪心的方式、即最大化各自的 Q 值，这样就很容易收敛到均衡状态。由此，我们将此拆解方法使用到我们的算法中。

On-policy Sarsa:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Off-policy Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

总的 Q 为：

$$Q(s,a) = \sum_{n=1}^{N} Q^{(n)}(s,a)$$

其中，Q 表示期望收益，R 表示 t 时刻的 reward（越大越好），A 为 t 时刻的动作，α 为学习率，γ 为折扣因子。公式表示每一次马尔可夫过程结束之后，对强化学习 Q 表的更新情况。Sarsa 算法相对而言收敛慢，但收敛结束后更稳定。Q-learning 收敛快，但是收敛完成后不稳定。经过测试，Sarsa 更适合本次项目。

由于本项目会尝试多种算法，所以我们将算法与智能体隔离开来，分别设置为 Agent 类和 RL 类，在 Agent 中统合不同的算法，为代码主要部分提供接口。

由于 Agent 类是在 Mac 层中实例化的，所以 init 函数将 Mac 对象包含在了 Agent 对象中。Agent 代码结构、RL 算法代码结构分别如图 5-6、图 5-7 所示。

```python
import RL

class agent:
    def __init__(self, *args) -> None:
        self.mac = mac
        self.RL = RL.Qlearning(*args)

    def action_choosing(self, *args):
        return self.RL.choose_action(*args)

    def learn(self, *args):
        self.RL.learn(*args)
```

图 5-6　Agent 代码结构

```python
class RL(object):
    def __init__(self, *args):
        pass

    def check_state_exist(self, *args):
        pass

    def choose_action(self, *args):
        pass
    def learn(self, *args):
        pass
```

图 5-7　RL 算法代码结构

在 Mac 中，需要初始化 backout 时，从强化学习智能体中进行动作选择。Ack 相关语句结束后便进行本轮学习。对 Q 表进行更新。如图 5-8 所示。

```
27  action = self.agent.action_choosing(str(state))
28  backoff = (parameters.SLOT_DURATION/2) * int(parameters.action_space[action])
29
30  self.agent.learn(str(state), action, reward, str(new_state), self.agent.action_choosing(str(new_state)))
```

图 5-8　RL 算法代码结构

RL 相关参数定义在 parameters.py 中，如图 5-9 所示。

```
32  # RL
33  action_space = ['0','1','2','4','8','16','32','64','128','256']#,'512']
34  learning_rate = 0.01
35  reward_decay = 0.9
36  e_greedy = 0.9
37  REWARD_of_ackfailing = -5
38  REWARD_of_transmissionfailing = -20
```

图 5-9　RL 相关参数定义

6 仿真结果

6.1 应用 CSMA/CA 协议通信系统的性能

6.1.1 控制台 log 输出

一次传输的典型 log 输出如图 6-1 所示。

在本次传输中，首先由节点生成传输事件并在信道空闲后开始第一次传输。然而第一次传输结束后并未收到由接收端发送的 ACK 信号，因此开始重传流程。在重传也即第二次传输中，接收方成功收到数据包后发送了 ACK 信号，发送方此后也成功收到了 ACK 信号。

至此本数据包的传输结束，发送方重新开始对信道的监听。

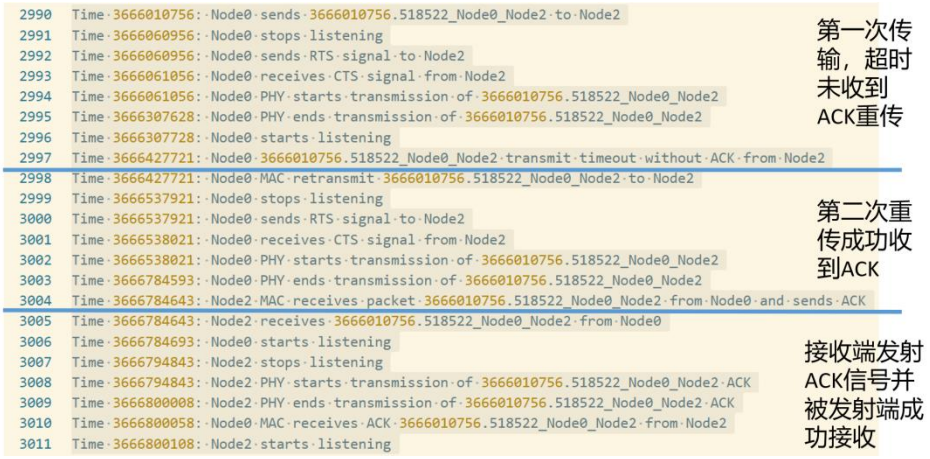

图 6-1 一次传输的典型 log 输出

6.1.2 运行结果

仿真条件：设定节点数为 4，最大重传次数为 10。数据包大小随机，节点位置随机。仿真时间设为 10 秒。

程序的一次运行结果如图 6-2 中数据所示。收发端延迟、收发包数量、传输失败包的数量、重传包数量随运行时间的变化如图 6-3 所示。

```
Total number of generated packets: 187
Total number of delivered packets: 131
Average delay (ms): 0.39236629007640283
Standard deviation of delay: 0.9353839621664811
Minimum delay (ms): 0.05590496296326819
Maximum delay (ms): 9.036940296296962
Total number of retransmissions: 563
Total number failed packets: 55
```

图 6-2 运行结果输出

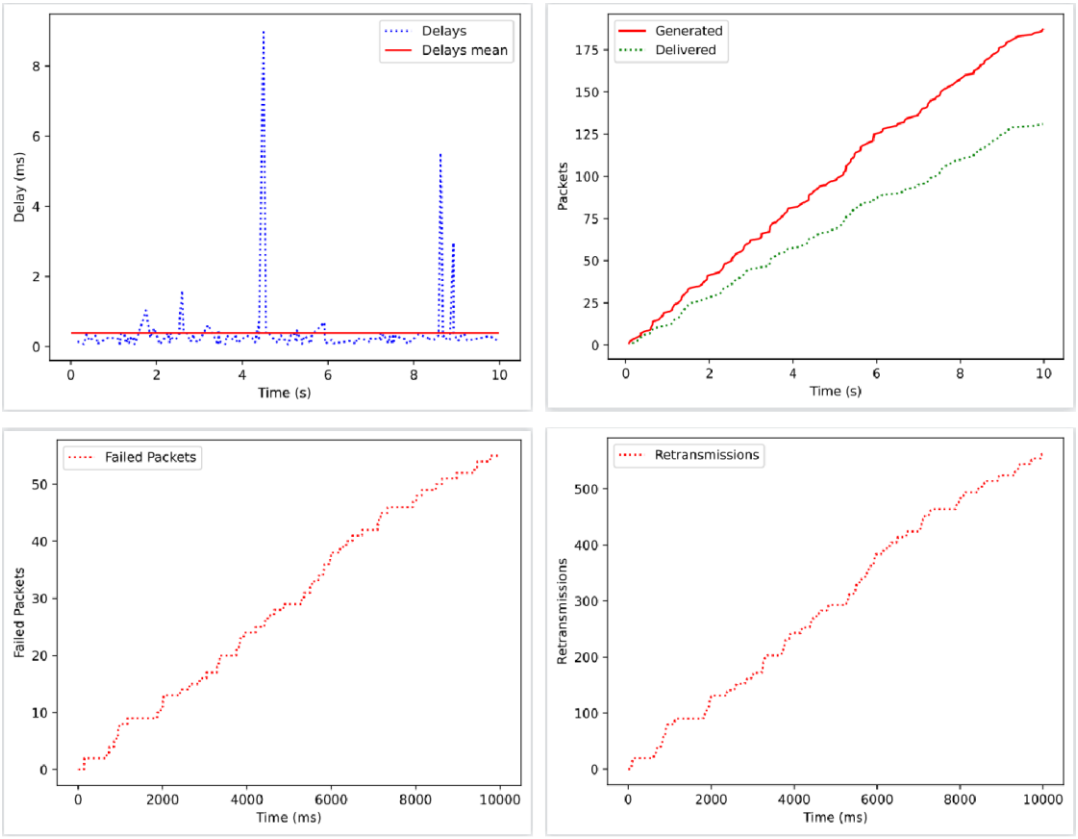

图 6-3 收发端延迟（左上）、收发包数量（右上）、
传输失败包的数量（左下）、重传包数量（右下）图像

结合运行结果输出与 PDF 文件输出，通信系统的收发端平均延迟约为 0.3924 ms，除了在少数时间点上会出现延迟很大的情况之外，在大部分时间里，延迟均保持略低于平均延迟的水平。发送包数量、接收包数量、传输失败包的数量（发送包数量-接收包数量）以及重传包数量均以近似线性的趋势随时间增长，证明系统性能较为稳定，能较好地对无线通信系统进行模拟。

6.1.3 仿真性能分析

首先，我们成功仿真模拟了 CSMA/CA 协议，并得到了有效数据。程序重传次数比较多体现了 CSMA/CA 协议本身存在降低通信效率的问题，这在后面强化学习的引入中得到了较好地解决。数据传输的时延均值不到 0.5 毫秒，最大值也仅为 9 毫秒，在时延方面 CSMA/CA 的性能让人还算满意，强化学习的引入也将对时延进行进一步的优化。

在程序性能方面，程序的运行时间比较久（1 个小时），但这是因为受限于所用的离散事件仿真库 simpy。在 simpy 库中，若在某个时间节点未查询到事件，则会一直调用其内置的 step 函数完成时间递增，这部分属于库自身的内容，我们暂时无法优化。

需要说明的是，发送包与接收包的数量差距并没有数据体现的那么大。这是因为到达设定的最大运行时间（此处为 10 秒）时程序自动终止了，此时仍在排队、发送或重传的包会被认定为未成功接收。

6.2 通信系统性能对比

由于受限于离散事件仿真库 simpy，CSMA/CA 通信系统的运行时间比较长（1 个小时）。

考虑到数据包大小与节点位置都是随机地进行初始化，且收发包数量、传输失败包的数量和重传包数量都以近似线性的趋势随时间增长，故取运行时间为 100 秒，并以该时间下的系统性能代表整个 CSMA/CA 通信系统的性能。

为了确保强化学习算法能够取到训练效果，我们允许其有较长的训练时间。在此取运行时间为 1000 秒，并以该时间下的系统性能代表整个 RL 通信系统的性能。

6.2.1 应用 MADDPG 的通信系统与应用 CSMA/CA 协议的通信系统的性能对比

仿真条件：两种系统均设定节点数为 4，最大重传次数为 10。数据包大小随机，节点位置随机。

两种系统的收发端延迟图像如图 6-4 所示，程序运行结果输出如表 6-1 所示。两种系统的收发包数量图像如图 6-5 所示，程序运行结果输出如表 6-2 所示。

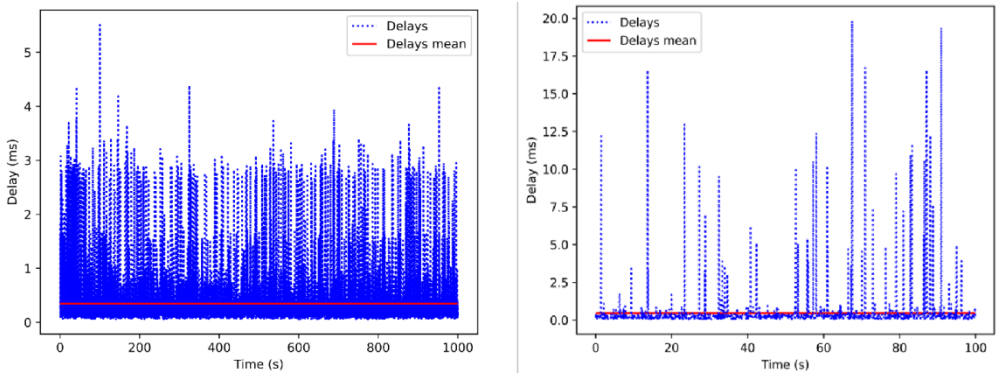

图 6-4 应用 MADDPG 的通信系统（左）与应用 CSMA/CA 协议的通信系统（右）的收发端延迟图像对比

表 6-1 应用 MADDPG 的通信系统（左）与应用 CSMA/CA 协议的通信系统（右）的收发端延迟运行结果对比（单位：ms，保留四位小数）

	MADDPG	CSMA/CA
最小延迟	0.0555	0.0555
最大延迟	5.5106	19.8121
平均延迟	0.3454	0.4637
延迟标准差	0.3949	1.4950

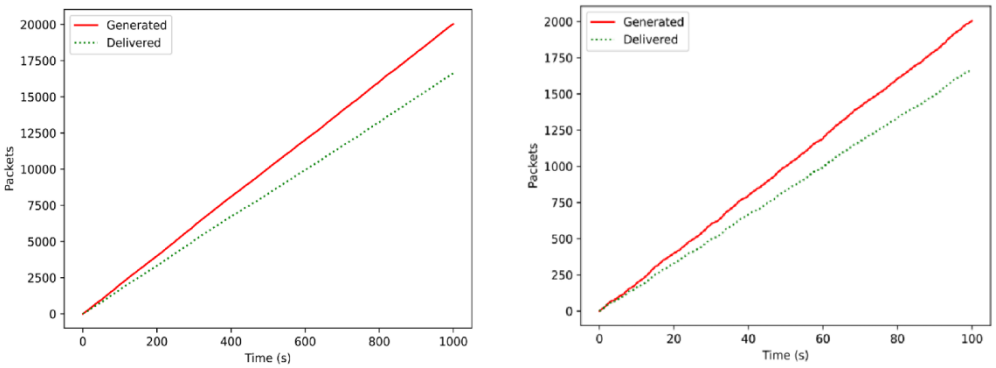

图 6-5 应用 MADDPG 的通信系统（左）与应用 CSMA/CA 协议的通信系统（右）的收发包数量图像对比

表 6-2 应用 MADDPG 的通信系统（左）与应用 CSMA/CA 协议的通信系统（右）的收发包总数量运行结果对比

	MADDPG	CSMA/CA
发送包总数量	20036	2006
接收包总数量	16622	1672
传输失败包总数量	3414	333
丢包率	17.04%	16.60%

综合来看，MADDPG 通信系统的收发端延迟在迭代后期显著收敛，趋于稳定。与 CSMA/CA 通信系统相比，使用 MADDPG 可以将平均延迟降低 25.51%，延迟标准差降低 73.59%，最大延迟也大幅度减小，系统收发延迟性能得到提升。两种系统的丢包率基本一样。因此，在丢包率方面，难以使用 MADDPG 学习到一个优化的策略。

6.2.2 应用 Sarsa 的通信系统与应用 CSMA/CA 协议的通信系统的性能对比

① 仿真条件：两种系统均设定节点数为 4，最大重传次数为 10。数据包大小随机，节点位置随机。

两种系统的收发端延迟图像如图 6-6 所示，程序运行结果输出如表 6-3 所示。两种系统的收发包数量运行结果输出如表 6-4 所示。

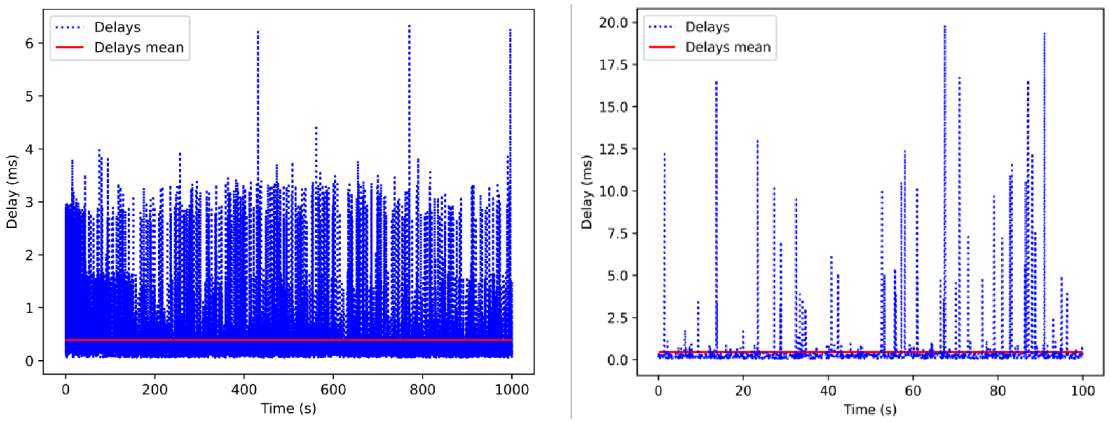

图 6-6 应用 Sarsa 的通信系统（左）与应用 CSMA/CA 协议的通信系统（右）的收发端延迟图像对比

表 6-3 应用 MADDPG 的通信系统（左）与应用 CSMA/CA 协议的通信系统（右）的收发端延迟运行结果对比（单位：ms，保留四位小数）

	Sarsa	CSMA/CA
最小延迟	0.0554	0.0555
最大延迟	6.3266	19.8121
平均延迟	0.3874	0.4637
延迟标准差	0.4810	1.4950

表 6-4 应用 Sarsa 的通信系统（左）与应用 CSMA/CA 协议的通信系统（右）的收发包总数量运行结果对比

	Sarsa	CSMA/CA
发送包总数量	20300	2006
接收包总数量	20300	1672
传输失败包总数量	0	333
丢包率	0%	16.60%

综合来看，使用 Sarsa 可以将平均延迟降低 26.45%，延迟标准差降低 67.83%，最大延迟也大幅度减小，系统收发延迟性能同样得到提升，提升幅度略小于 MADDPG。同时，在最大重传次数为 10 的前提条件下，使用 Sarsa 可以实现丢包率为 0，这对系统传输性能是极大的提升。

② 仿真条件：两种系统均设定节点数为 4。数据包大小随机，节点位置随机。为了避免偶然性，我们将 Sarsa 系统的最大重传次数重新设置为 5，并与最大重传次数为 10 的 CSMA/CA 系统作性能对比实验。

两种系统的收发端延迟图像如图 6-7 所示，程序运行结果输出如表 6-5 所示。两种系统的收发包数量图像如图 6-8 所示，运行结果输出如表 6-6 所示。两种系统的重传包总数量运行结果输出如表 6-7 所示。

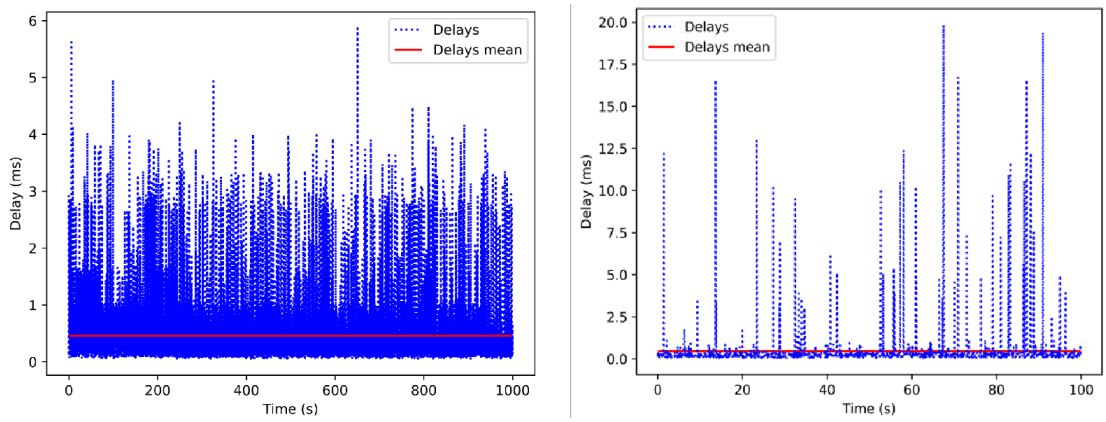

图 6-7 应用 Sarsa 的通信系统（左，最大重传次数为 5）与应用 CSMA/CA 协议的通信系统（右，最大重传次数为 10）的收发端延迟图像对比

表 6-5 应用 Sarsa 的通信系统（左，最大重传次数为 5）与应用 CSMA/CA 协议的通信系统（右，最大重传次数为 10）的收发端延迟运行结果对比（单位：ms，保留四位小数）

	Sarsa	CSMA/CA
最小延迟	0.0555	0.0555
最大延迟	5.8626	19.8121
平均延迟	0.4599	0.4637
延迟标准差	0.4896	1.4950

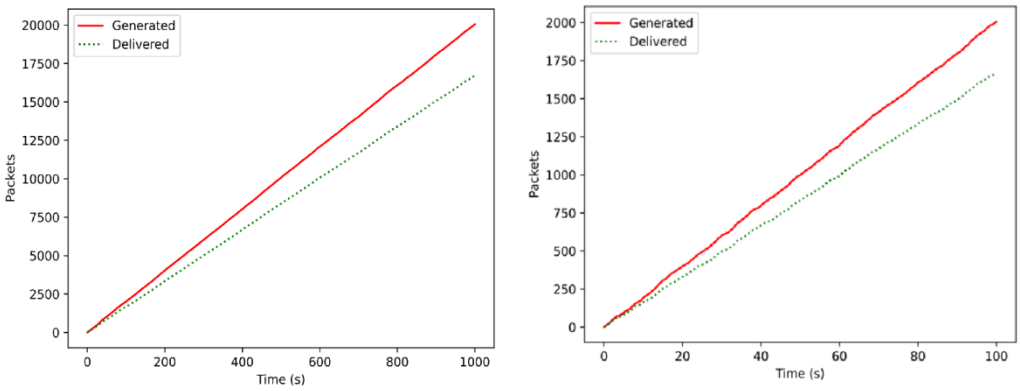

图 6-8 应用 Sarsa 的通信系统（左，最大重传次数为 5）与应用 CSMA/CA 协议的通信系统（右，最大重传次数为 10）的收发包数量图像对比

表 6-6 应用 Sarsa 的通信系统（左，最大重传次数为 5）与应用 CSMA/CA 协议的通信系统（右，最大重传次数为 10）的收发包总数量运行结果对比

	Sarsa	CSMA/CA
发送包总数量	20055	2006
接收包总数量	16724	1672
传输失败包总数量	3331	333
丢包率	16.61%	16.60%

表 6-7 应用 Sarsa 的通信系统（左，最大重传次数为 5）与应用 CSMA/CA 协议的通信系统（右，最大重传次数为 10）的重传包总数量运行结果对比

	Sarsa	CSMA/CA
重传包总数量	17252	3401
重传率 （重传包总数量/发送包总数量）	86.02%	169.54%

综合来看，当把 Sarsa 通信系统的最大重传次数调整为 5 时，相比最大重传次数为 10 的 CSMA/CA 系统，系统性能有所变化。Sarsa 系统的平均延迟降低 0.82%，延迟标准差降低 67.25%，最大延迟大幅度减小。因此，系统的收发延迟性能仍有提升，但提升幅度不大。在丢包率方面，两种系统几乎相同，无明显差别。值得一提的是，Sarsa 通信系统的重传率约为 CSMA/CA 通信系统的一半，证明 Sarsa 算法能学习到一个较优的传输策略来减少数据包之间的碰撞。

7 工作评价

7.1 贡献与创新

① 构建了一个完整应用 CSMA/CA 协议的通信系统，且该通信系统绝大数参数可以由使用者更改。

② 复现了多智能体强化学习算法 MADDPG，并验证了其在改进通信系统性能上的可行性。

③ 自行尝试了将多智能体强化学习模型降维为单智能体模型，并验证了该方法在此项目中的可行性，结果显示该方法较大幅度地改善了通信系统性能。

7.2 不足与展望

① 通信系统仿真运行效率不够高，在 12700 CPU 下运行一次 100 秒仿真需要五十分钟左右的时间。

② 由于时间较为紧张，且对大规模节点的仿真较为耗时，因此在对 CSMA/CA 的仿真中，我们均设定节点数为 4。在后续的实验中，可以考虑将节点数调整至 10 或以上，观察 CSMA/CA 对大规模节点的仿真情况并进行性能对比。

③ 由于计算资源的限制，现有的多智能体强化学习算法只尝试了 MADDPG 这一种 policy-base 的算法，且并未尝试 VDN、QMIX 等 value-base 算法。

④ 对于大规模群体问题，有 MFMARL（基于平均场理论的多智能体强化学习）可以进行简化，但由于我们未上升到大规模系统，故并未进行尝试。

（相关论文链接：https://arxiv.org/abs/1802.05438v4）

8 参考文献

[1]Zhiyuan Jiang,Member,IEEE,Yan Liu,Jernej Hribar,Luiz A. DaSilva,Fellow,IEEE, Sheng Zhou,Member,IEEE,and Zhisheng Niu,Fellow,IEEE. SMART: Situationally-Aware Multi-Agent Reinforcement Learning-Based Transmissions. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9386228

[2] 童 乐，梁 涛，张 余 ，钱鹏智. 基于多智能体强化学习的动态频谱分配方法. http://www.opticsjournal.net/Articles/OJ6e19483e60fcd2d9/Abstract

[3] Riccardomicheletto, Computer Science student at University of Trento, Discrete event simulator for the 802.11 CSMA/CA DCF. https://github.com/riccardomicheletto/SPEproject-802.11DCFsim