无线传感网实验 实验报告

学院：计算机科学与技术学院

指导教师：李瑞、蒋志平

项目成员：

- 罗阳豪 16130120191
- 方浩杰 16130120201

1. 实验背景与题目要求

在无线多跳自组织网络环境下，高效地将关键数据从网络的一端扩散至全网是多跳自组织网络中非 常重要的重要之一，广泛用于网络控制、公共数据广播、时间同步等，这种数据广播协议一般称为 数据分发协议。

分发协议本质是一种广播协议，目的是让环境中所有的节点都收到消息。但分发协议在无线多跳网 络中存在多方面的权衡与设计挑战，例如：

- 应实现总体节能，分发协议应尽可能减少无线发送的次数，延长网络的工作寿命；
- 应实现个体节能 ，应降低对某个特定节点的发送次数，当此节点失效时，可能会影响整个网络的工作；
- 应尽量减小网络跳数，网络跳数过多，传输成功率会下降，同时传输时延会上升

题目：

1. 在 MATLAB 或 Python 中模拟 N(N > 100) 个节点的多跳传感网络，该 N 个节点随机分布在 100m * 100m 的正方形 2 维平面。每个节点的通信半径符合正态分布 $r \sim N(\mu, \sigma^2)$ ；进一步，假设当两个节点距离为 d 时，通信成功率 $t = 1 - \frac{d^2}{r_1 r_2}$ 。每个节点一些基础物理信息：id，总电量，单次发射耗电量；
2. 设计自己的数据分发协议：
    1. 基本要求：在不考虑电量的情况下，可实现从任意选定的一点将信息分发至全网
    2. 进阶要求：在考虑电量的情况下，实现从任意选定的一点将信息分发至全网
    3. 高阶要求：在考虑电量的情况下，提出一种最优的数据分发策略
3. 可视化多跳网络，并通过简单的过程动画展示分发过程

2. 实验思路与过程

2.1 模拟无线传感网并将其可视化

该项目使用 Python3 实现，使用数学库 `numpy` 生成所需的随机数和进行一些节点坐标的运算，使用 `matplotlib` 进行数据可视化，使用 `PyQt5` 作为数据可视化的图形引擎，并使用 `coloredlogs` 为打印在终端的 log 提供一些人类友好的色彩。

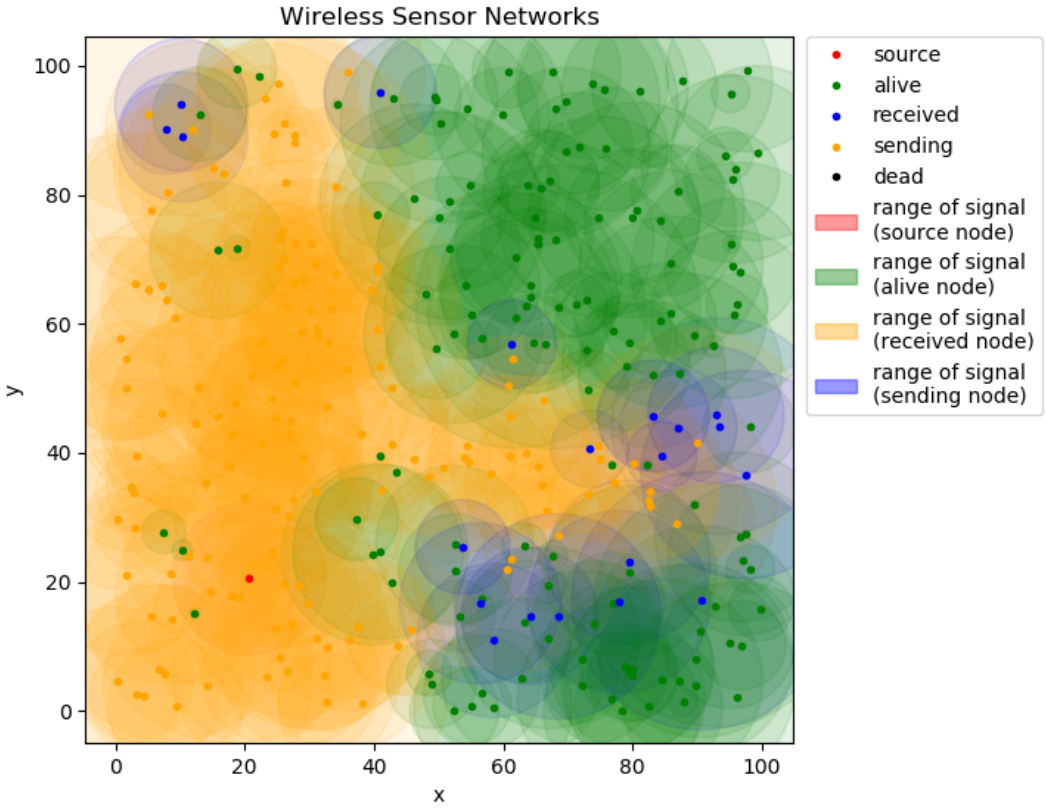
无线传感网是由若干孤立的节点构成的，它们一般是具有通信能力的小型嵌入式设备，它们可以独立地收、发消息。所以应该将其实现为一个节点类 `WsnNode` ，这样一个类包含坐标、电量、发送消息队列、接收消息队列等属性，并实现有至少收发消息的方法。并且由于这些节点是独立运作的，所以很适合将其运行放在独立线程中，一个节点一个线程。所以这样一个线程也是 `WsnNode` 类的属性。

对于一个无线传感网来说，节点应该能够被添加、删除，并且为了实验可控，它还应该能够控制节点的开启和关闭。所以将这些对节点的管理和操作可以打包为一个节点管理者类 `WsnNodeManager` 。通过节点管理者，我们可以批量创建节点、同时将这些节点开启（启动这些节点的节点线程）、同时将它们关闭（通知这些节点的节点线程退出）。

节点及其管理者并不是一个无线传感网的全部，在真实环境下，空间也是无线传感网的一部分，空间作为无线传感网节点间通信的“介质”，能够容纳无线电信号，通过一些物理过程（比如随着传播距离增加而削弱信号强度），并且最终带着被物理规律过滤过的信号，概率地送达可能的接收者。所以再实现一个 `WsnMedium` 类，以模拟这个过程，一个无线传感网应该只有一个介质对象，它是被所有节点共享的。介质类提供“传播”的方法，接收消息作为参数，用数学方法模拟物理过程，然后概率地将消息放到可能的目标节点的接收消息队列。其实放置消息到接收消息队列原则上应该是节点的网络接口卡做的事情，但是为了简化设计，把这一部分交给了介质。

作为通信协议的一部分，消息除了内容本身，还带有其它的信息，以达到协调传输过程的目的。在真实情况下这些额外信息和消息内容本身会被按照一定的格式包装在一个或多个分组中，并编码。但是编解码、封包解包并不是这个实验的重点，所以我们用基类 `BaseMessage` 及其各种子类来表示，根据协议需要，类除了消息内容本身，还会有额外的属性。所有消息都会是 `BaseMessage` 的对象或其子类的对象，它们能在节点与介质之间交换。

节点、节点管理器、介质、消息，构成了一个无线传感网。将它们初始化，让它们跑起来，就能模拟一个无线传感网的工作。但是如何将其工作过程可视化呢，除了无线传感网工作时打的成吨的 log ，我们还使用 `matplotlib` 在一个平面坐标系上绘制一些点和圆来表示节点和节点的通信半径，并用不同的颜色来区分节点的不同状态。



我们在一个独立线程上不停地检查网络上各个节点的状态，并且将变化更新到图像上。通过图像表示的节点状态的变化，我们可以直观地了解这个网络的工作情况。这些能力被我们打包在旁观者类 `Bystander` 中。它除了能在网络工作时实时呈现网络状态，还会在网络停止后将记录的每一帧结果以 `.gif` （取决于系统支持）和 `.html` 、 `.png` 文件的形式持久化。

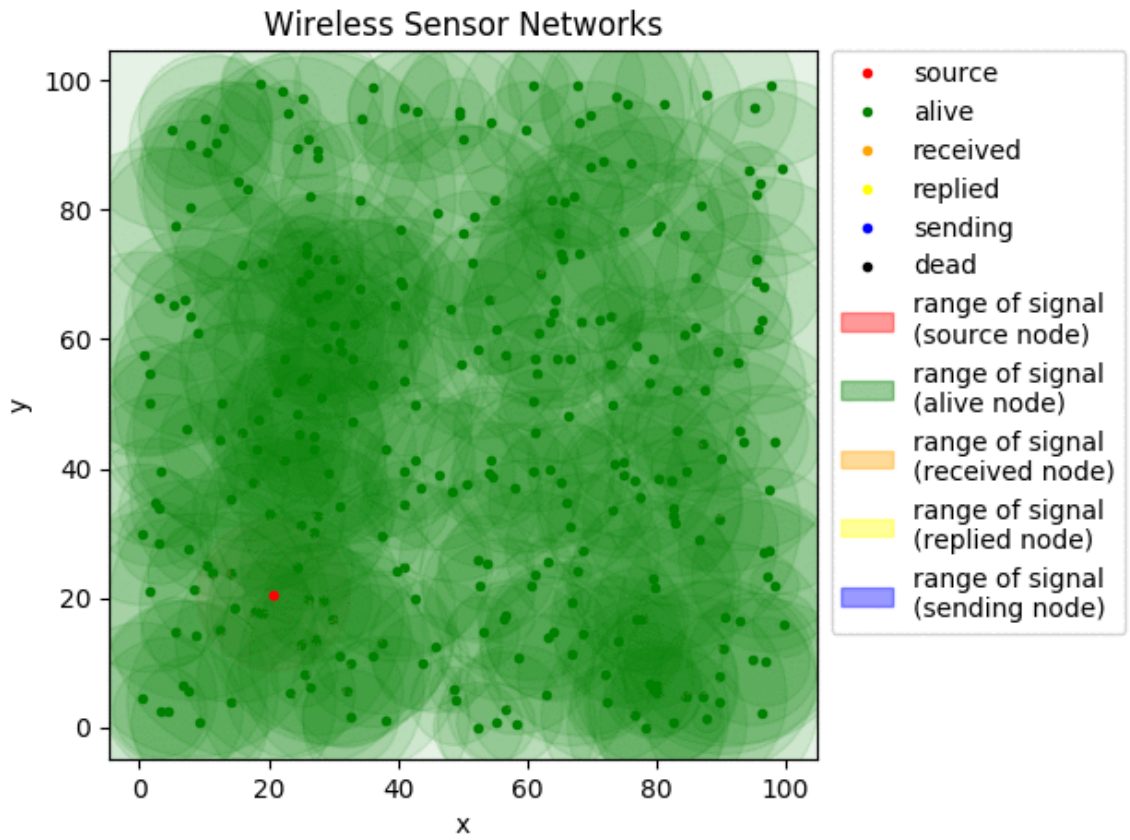
由此，我们搭建了一套实验框架，接下来只需要去实现和调整这里面节点的行为，就能够进行实验验证。

以下示例中，尽管我们使用了不同的算法、节点的行为、消息的传播都有其随机性，但是在生成地图时使用了一样的随机数种子，使得 **生成的地图和发送消息的源节点都是一样的** ，以方便比较。

2.2 实现基本要求

> 基本要求：在不考虑电量的情况下，可实现从任意选定的一点将信息分发至全网

这个要求其实基本就是没有要求，只要求消息送达。我们很容易能够想到一个能够迅速传播消息的方法“广播风暴”。就是，源头节点不停地重复发送同一个消息，收到这个消息的节点也不停重复地转发这个消息。它的效果如下图所示



图中，绿色表示普通的节点，蓝色表示接收到消息并器在发送的节点。可以看出来，消息扩散的速度非常快，而且只要时间足够长，所有能够与源节点之间存在通信链路的节点最终都会接收到消息。

这种方案只有一个问题，就是 **耗电量无穷大** 。因为不管消息覆盖情况如何，没有一种机制让这些节点能够自发停下来。在这个实验中，是观察者统计到绝大部分节点都接收到了，于是通知主线程停止，但是在实际中，不会有一个上帝视角来观察消息的发送情况，所以节点也只能无休止地重复广播。不过作为参考，在外部介入的情况下，完成上图中的数据分发，一共消耗了 9578 点电力（一个节点发送一次消耗 1 点）。

2.3 尝试进阶要求

> 进阶要求：在考虑电量的情况下，实现从任意选定的一点将信息分发至全网

2.3.1 初始方案：有限重发 + 单次转发 + 要求回应

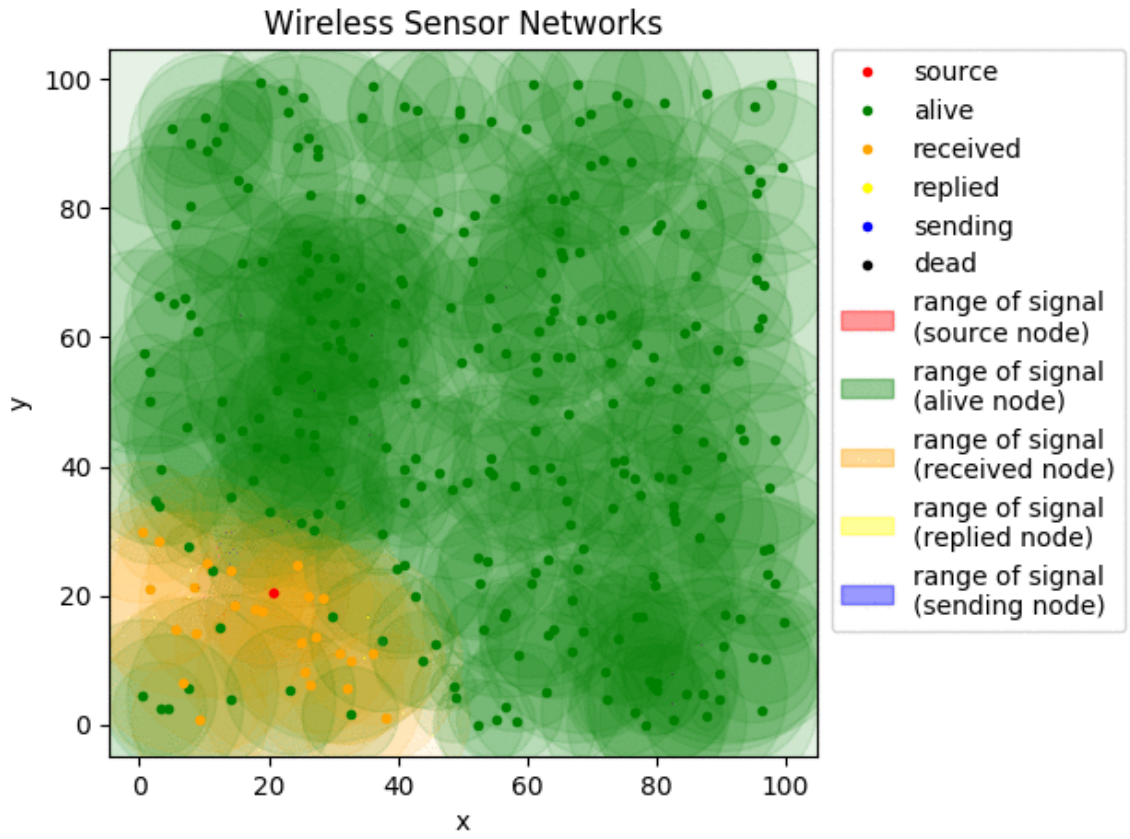
由于上一个方案耗电量无穷大的原因就是节点不知道何时停止，但是网络要求确保送达，只能不停重发。所以解决方法就是接收到消息需要回应。每一个消息带一个 UUID ，这个消息的所有重发、转发、回应都使用同样的 UUID 。如果要让每个中间节点都停止，就需要确保每个节点都接收到其它所有节点的回应，那么如何传播回应，如何停止回应的重发和转发又是一个问题，如果这样讨论下去将会没完没了。就好像你难以让一个沸腾的班级安静下来。

所以最好就是控制消息的传播，只有消息源可以重发消息，其它中间节点只能跟着转发一次，而且每次转发会在消息中记录上自己的 ID ，如果是收到一个自己转发过的消息就丢弃，以确保网络中不会形成转发的环路。这样，只要源头停止重发，整个网络就会停下来。

但是源头如何停止重发呢，就是根据回应。每个节点收到一个非回应消息，就要发送一个对应的回应消息，回应消息也是通过一样的方式可以被中间节点转发一次，但是不需要对回应消息回应。源头节点一边重发，一边接收回应消息，当它发现网络中所有节点都给它回应了，它就停止。

这样做还有最后一个问题，就是源头节点如何知道网络中有哪些节点？我们假设网络中的节点是一次性投放的，而且它们是有共同目标的，所以它们 ID 的编号是彼此关联的。比如一个编号 `2/300` 表示编号总范围是 1-300 ，它自己编号是 2 ，这样每个节点就能够知道网络中其它节点的编号。所以当源头节点收集到所有其它节点的回应后，停止重发消息，整个网络上对这个消息的转发和回应和回应的转发，很快都会停止。

传播情况如下图



途中橙色的表示接收到消息的节点，黄色表示接收到消息且源节点收到了它们的回应的节点。

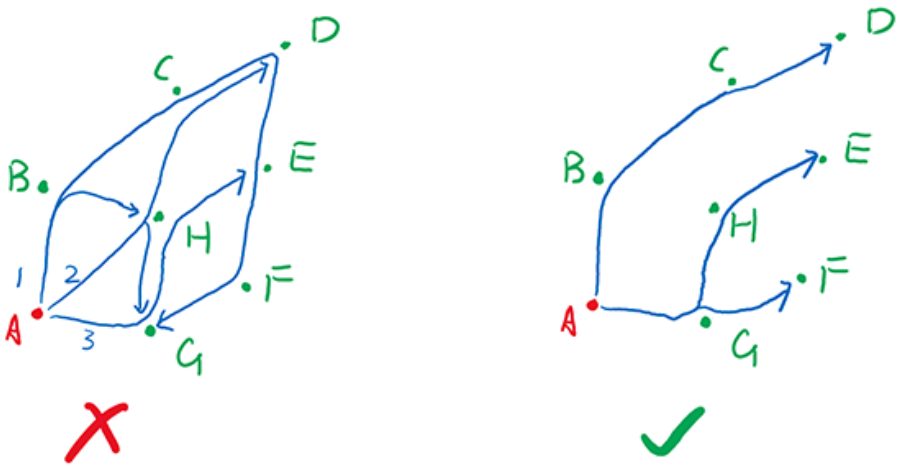
可以看到，由于只有一个源头在重发，沿着传播链，越往后消息在转发过程中丢失的概率就越大，回应被丢失的概率也越大，所以传播比较慢。而且由于有一些节点通信半径极小，没法与其它节点建立通信，所以无论如何也无法接收到消息。但是源头节点不能知道哪些节点是这种情况，所以设定了 90% 的阈值，只要收到了 90% 的节点的回应，就判定为已经送达全网。

这个方法是可行的，它没有消耗无穷大的电力，但是显然它比第一种方法浪费的能量多，在上图所示的实验中共消耗 601684 点电力，总节点数是 300 ，也就是说平均每个节点要耗电 2000 多点，而且实际上越靠近源头节点，耗电越大。

所以需要改进一下

2.3.2 改进1：使用最常用路径

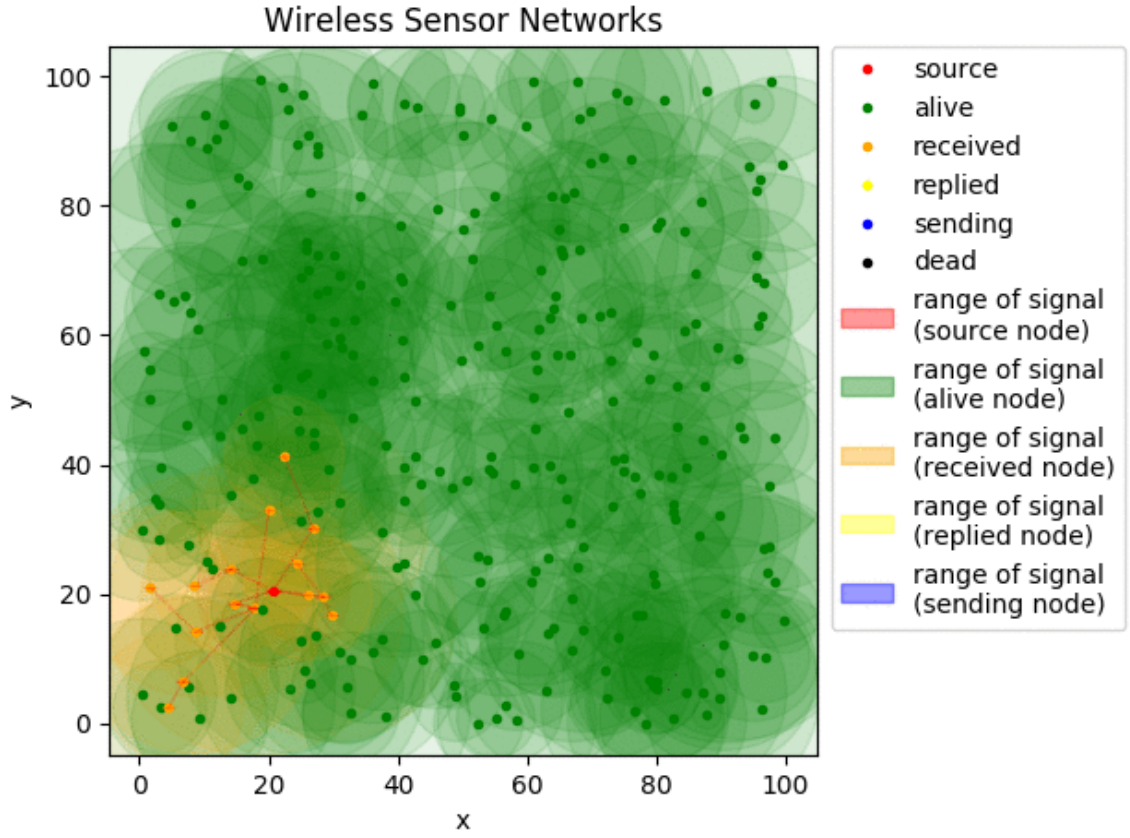
上一个方案解决了广播风暴无法停止的问题，但是耗电量仍然惊人。如果分析这些节点打出的收发消息的日志，很容易能够发现一个问题。上一个方案中，我们使用在转发的消息中带上自己的id，以避免自己重复转发自己转发过的消息，也就是 **避免转发链成环** 。但是只要稍微多想一步就能发现，即使不会成环，也会走很多弯路。举个例子：



在下面这两个草图中，上一个方案明显更接近左边草图所示的情况，所有转发链都没有成环，但是有大量没有意义的转发链。最好的方案应该是类似左侧，消息沿着一个生成树传播。

但是由于节点都是孤立的，节点不知道其它节点的位置，所以自然也不知道这样一棵树是怎样的。但是，节点可以根据接收到的消息，猜测网络的状态。每一个消息都带有沿途每一个节点的 ID （因为每个节点转发之前会在消息中加上自己 ID ）。所以如果统计收到的这些消息，可以发现某一些路径出现的概率比其它更高，依靠这些信息可以设计一个转发策略。

实际操作上，因为一个节点没法决定它前面的路径，只能影响它后面的路径。而且前面很多跳之外的路径其实也很难参考，统计起来开销也很大。所以每个节点只统计一个消息的源头和直接前驱节点。比如，一个消息通过 `A -> B -> C ->` 传到 D 手里，那么 D 就记录 `'A -> ... -> C' += 1` 就不关注 A 和 C 之间的路径。当一个消息来，节点检查其是否是从相同源的所有路径中最常用的路径传过来的，如果是就转发一次，否则丢弃。这样做的理由就是，一个最常用的路径一般意味着最大的传输成功率，所以通过转发激励这条路径的发展，阻止其它低效的路径的发展。这样调整之后效果如下



可以看出来，传播速率依然很快。但是更加高效、节能，上图所示过程花费了 17747 点电力。图中橙色的线表示以消息源为起点的最常用路径。

上图中只演示了消息发送，但是我们知道消息源还必须收到节点们的回应才能停止。但是我们没有将回应消息和普通消息一样处理，我们使用了不一样的处理逻辑。

2.3.3 改进2：回应原路返回 + 逐级确认

我们没有使用和发出消息一样的方法来发回回应。理由很简单，因为上一种改进中使用的发出消息的方法更适合于一对多，但是 **消息的回应是一对一的** ，只需要消息源收到回应即可。一对一的消息如果也要传播至整个网络，花销就太大了点，上一节的演示花费了 17747 点电力，如果回传也用这种方法，那 300 个节点也许需要 50 万点电力。

所以我们使用另一种方法，就是原路返回。因为在消息发出过程中，网络中已经产生了一个类似生成树的结构，所以消息送达路径也许就是比较好的路径，那回应也就原路返回就好了。

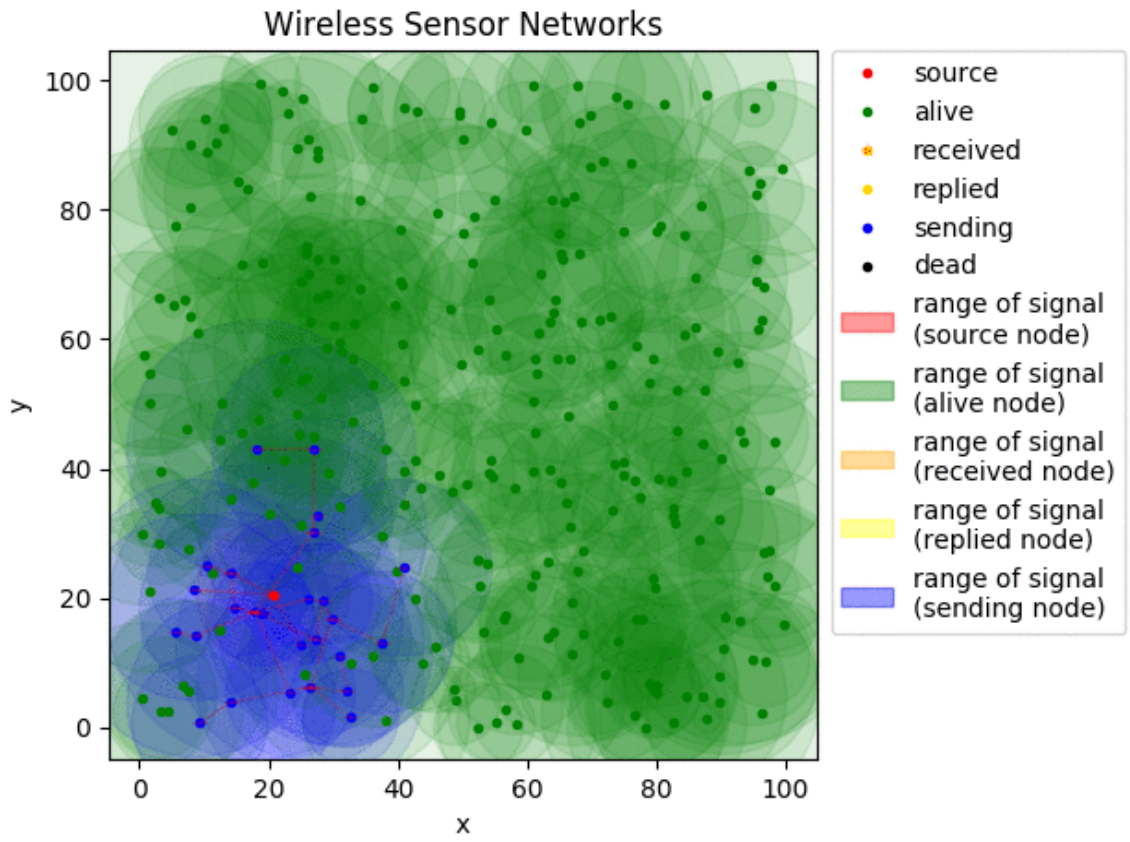
那如何原路返回呢？我们把消息中记录的路径反转过来，写在回应包中，比如一个消息通过 `A -> B -> C ->` 传达 D ，那么 D 的回应消息中就写下 `D -> C -> B -> A` 。D 将它广播出去，如果 C 收到了， C 发现自己 ID 在回应包路径第二位，所以 C 将自己 ID 删除，转发一次 `D -> B -> A` 随后是 B ， A ，沿途节点逐一删除自己的名字然后转发一次。这样这个回应消息最终会传达到源头。过程基本上类似于传出过程的逆过程。

但是回应也有可能会在路途中被丢掉，所以需要一些重发机制。一种方法是每次收到消息就发一个对应的回应，但是一个消息送达就已经很艰难了，回程有和去程一样的失败率，所以最终成功率是单程成功率的平方。就需要非常非常多次重复才能成功。

所以我们换一个思路，既然我们都知道回程的路线了，所以我们为什么不设计一种回程的逐级确认的机制呢？如果一个节点要回应，它就不停重发回应，直到收到下一个节点的确认，然后重发的职责就交给了下一个节点。

具体操作是这样的，如果一个节点要回应，那么它就不停重发回应。直到它发现有另一个节点在转发它的回应（这说明回应已经传达到了下一级）。那么此时它就停止重发。中间逐级的发送和确认都是一样的方法。

在改进1的基础上加上改进2的效果如下：



图中增加了一些蓝色节点，蓝色节点是正在发送回应消息的节点。可以看出一些边缘节点收到消息后逐渐将回应消息的发送职责交给了靠近源的节点，蓝色节点收缩的发展情况类似于橙色节点扩散时的逆过程。

以上全过程共消耗电力 68168 点，平均一个节点经历了数百次发送，比该方案未经改进过的情况耗电少了几个数量级。

2.3 高阶要求

> 高阶要求：在考虑电量的情况下，提出一种最优的数据分发策略

由于我们能力实在有限，实在是不能设计出一种 **“最优”** 的数据分发策略。

...

在思考上一节的两个改进的过程中，我们觉得我们的设计中已经包含了一些“较优”的思想。避免丢包只能通过重传，要知道重传何时停止就只能够通过消息确认。消息发出时，因为对网络情况一无所知，所以稍微粗放一点，但是在过程中产生节点对于网络状态的理解。回传确认的时候就通过一个已知的较好路径，并且尽量利用已知信息（比如回程路径）增加节点对于传输结果的把握，减少重发次数。

但是要如何做到“最优”，我们实在是没有什么思路，因为虽然我们处于上帝视角可以很容易规划出一个“最优路径”。但是作为网络中对网络一无所知的节点，只能通过大量的摸索，建立对网络的认识。这个摸索的方法是否最优是很难判断的，因为我们甚至不知道一个“最优”的探索方法会具有怎样的指标或者模式。探索之后节点就会产生一些对网络的认识，可以利用这些认识减少无意义的工作。很显然，节点获得的信息越多，就能够越高效，但是探索网络和交流对网络的认知是需要额外开销的，在这两方面如何能够达到一个“最优”的平衡也是未知的。

因为我们对于“最优”缺少认识，所以自然也不能给出最优的设计。

3. 实验总结

通过这个课程实验，我们除了了解了 Python 的科学计算和数据可视化工具 numpy 和 matplotlib ，丰富了我程序设计经验。更重要的，还加深了我们对于无限传感网络的认识。

在一个无限传感网络中，每个节点是孤立的、无知的，被随机地放置在网络中。但是不代表它们不能协同工作，其核心就是网络通信。通过网络，节点之间可以共享信息，使得每一个节点都对网络整体有一部分认识。通过这些认知，节点的行为能够变得越发高效。而如何设计这种协同行为是十分有挑战的。