1 问题重述

光纤通信是以光波作为信息载体，以光纤作为传输媒介的一种通信方式。自1966年英籍华人高锟教授提出光纤通信的概念后，经历了几十年的研究与开发，光纤以其传输频带宽、抗干扰性高和信号衰减小，而远优于电缆、微波通信的传输，成为了今天大容量、长距离信息传输的主要方式，遍布全球，在全球的信息交流、信息流通方面扮演着重要角色。

光纤通信是以光纤为信道、光波为信息的载体进行二进制的编码传输。发射机进行编码调制后进行传输，不同的调制方式可以表示为不同的“星座图”。传输过程中光功率会发生衰减，所以每经过一段距离后，需要通过光放大器对光功率进行补偿放大。同时，来自放大器和光纤传输的噪声可能会导致产生误码。接收机接收到信号后再进行解调，从而建立起两地之间的一条连接进行通信传输。在相同技术条件下，传送容量会随着传输距离的增加而减小，在建立全国范围的光传送网时，还需要综合考虑传输距离、传输容量和网络拓扑等多种因素，获得最大化的网络价值。

本题需要通过建立相应的数学模型和算法，解决以下问题：

（1） 问题 1 中的子问题 1 需要分别利用 4 进制、8 进制和 16 进制，即 QPSK、8QAM 和 16QAM 三种调制方式，经过信道叠加噪声和接收机解调后，在不同信噪比（SNR）的情况下计算误码率（BER），得到不同调制方式的信噪比与误码率的关系曲线，并计算出 BER=0.02 时的 SNR 容限点。子问题 2 在单跨传输距离为变为 80km 和 100km 时，计算三种不同调制方式的最远传输距离。

（2） 问题 2 中的子问题 1 选定一种 16 条连接数的网络规划，将连接数扩展到 33 条，计算出其网络价值，并找到最大的网络价值。子问题 2 可以利用中间节点建立起两个节点之间的多个连接，从而对中间节点的传输容量进行分配得到最大网络价值，将市扩大到省之后，分析网络价值的变化情况。子问题 3 则在兼顾运营商的收入和均衡发展的情况下调整目标函数，并与之前的目标函数进行对比分析。

（3） 问题 3 在保证信息熵为 3bit 的情况下，改变 16QAM 的星座点位置、数量或者每个点的概率，使得新的 16QAM 方案的比 8QAM 具有更低的 SNR 容限点。

由于本题的 3 个问题相对独立，且 3 个问题的思路和模型差别大，为保证读者在阅读的过程中思维的连贯，所以本文在行文过程中将按照 3 个问题的顺序进行展开。

2 问题 1

2.1 问题分析

对于子问题 1，在不考虑信噪比变化，即信号功率和噪声功率不变的情况下，可以通过随机比特序列，分别利用 QPSK、8QAM、16QAM 三种调制方式对比特序列进行调制，在传输的过程中加入噪声，再对含有噪声的信号进行解调，得到多组对应的 SNR 容限值和 BER，将多组 BER-SNR 对应值进行拟合，即可得到 BER 与 SNR 的关系曲线。同时，可以利用多项式对曲线进行拟合，设定拟合误差值，确定出拟合多项式的阶数，即可得到 BER 与 SNR 的函数关系。通过函数关系，可以确定在 BER=0.02 时 SNR 容限点。

对于子问题 2，在传输过程中，在每跨开头叠加非线性噪声，SNR 值发生变化，传输过程中光和噪声都发生衰减，这个过程中 SNR 值不变，在经过放大器之后，放大器将光功率补偿至初始功率，且在末尾叠加放大器噪声，SNR 再次变化。每跨传输末尾的 SNR 值相比每跨开始的 SNR 值变小。在经过多跨传输之后，在某跨的结尾的 SNR 值对应的 BER 值将超过 0.02 的门限值，此时得到的就是传输最远的跨段数量，即可得到最远的传输距离。

2.2 模型假设

（1） 在本次建模中，为了简化模型，只考虑光信号本身的功率。
（2） 每跨开始阶段的光功率保持相同，均为最开始的入纤光功率。
（3） 由于题目中未给定入纤光功率，取通信工程中常用的 1mW 作为问题 1 子问题 2 的入纤光功率进行计算。
（4） 放大器放大的目标时将光功率放大到初始功率，并将噪声功率同比例放大。
（5） 假设放大器先将光功率放大到初始功率，再叠加放大器噪声。

2.3 变量说明

表 1 问题 1 变量说明

符号	意义	单位	备注
f_c	光波振动频率	Hz	
p_{nf}	放大器噪声功率	mw	
p_{ng}	非线性噪声功率	mw	
p_{nt}	单跨噪声总功率	mw	
h	普朗克常熟	J·s	
B	带宽	GHz	
f	光波频率	THz	
NF	噪声指数		

2.4 模型的建立与求解

2.4.1 理论基础

（1） 调制

QPSK 调制是利用载波的四种不同相位差来表征输入的数字信息，它规定了

四种载波相位，调制器输入的数据是二进制数字序列，为了能和四进制的载波相位配合起来，则需要把二进制数据变换为四进制数据，这就是说需要把二进制数字序列中每两个比特分成一组，共有四种组合，即 00，01，10，11，其中每一组称为双比特码元。每一个双比特码元是由两位二进制信息比特组成，它们分别代表四进制四个符号中的一个符号。可传输 2 个信息比特，这些信息比特是通过载波的四种相位来传递的。QPSK 中每次调制可传输 2 个信息比特，这些信息比特是通过载波的四种相位来传递的。解调器根据星座图及接收到的载波信号的相位来判断发送端发送的信息比特。[1]

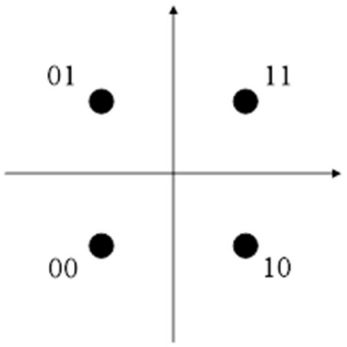

图 1 QPSK 星座图

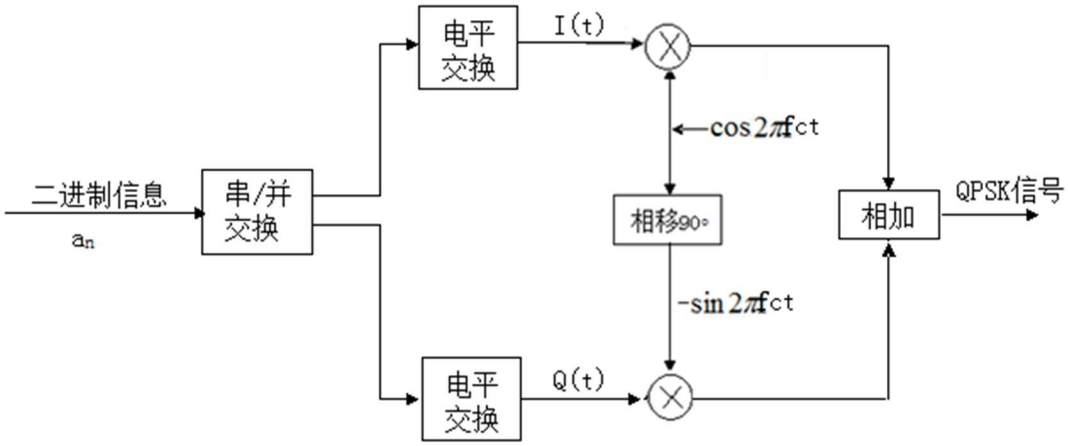

图 2 QPSK 调制框图

QPSK 调制框图如图 2 所示，其中串并转换模块是将码元序列进行 I/Q 分离，转换规则可以设定为奇数位为 I，偶数位为 Q。以 1011001001 为例，则 I 路为 11010，Q 路为 01001。

电平转换模块是将 1 转换成幅度为 A 的电平，0 转换成幅度为-A 的电平。例如：以 π/4 QPSK 信号来分析，由星座图图 1 和调制框图图 2 可以看出，当输入的数字信息为“00”码元时，则输出载波：

$$-\mathrm{A}\cos(2\pi f_c t) + A\sin(2\pi f_c t) = \sqrt{2}A\cos\left(2\pi f_c t + \frac{5\pi}{4}\right) \quad (1)$$

当输入的数字信息为“11”码元时，则输出载波：

$$\mathrm{A}\cos(2\pi f_c t) - A\sin(2\pi f_c t) = \sqrt{2}A\cos\left(2\pi f_c t + \frac{\pi}{4}\right) \quad (2)$$

当输入的数字信息为“01”码元时，则输出载波：

$$-\mathrm{A}\cos(2\pi f_c t) - A\sin(2\pi f_c t) = \sqrt{2}\mathrm{A}\cos\left(2\pi f_c t + \frac{3\pi}{4}\right) \quad (3)$$

当输入的数字信息为“10”码元时，则输出载波：

$$\mathrm{A}\cos(2\pi f_c t) + A\sin(2\pi f_c t) = \sqrt{2}\mathrm{A}\cos\left(2\pi f_c t + \frac{7\pi}{4}\right) \quad (4)$$

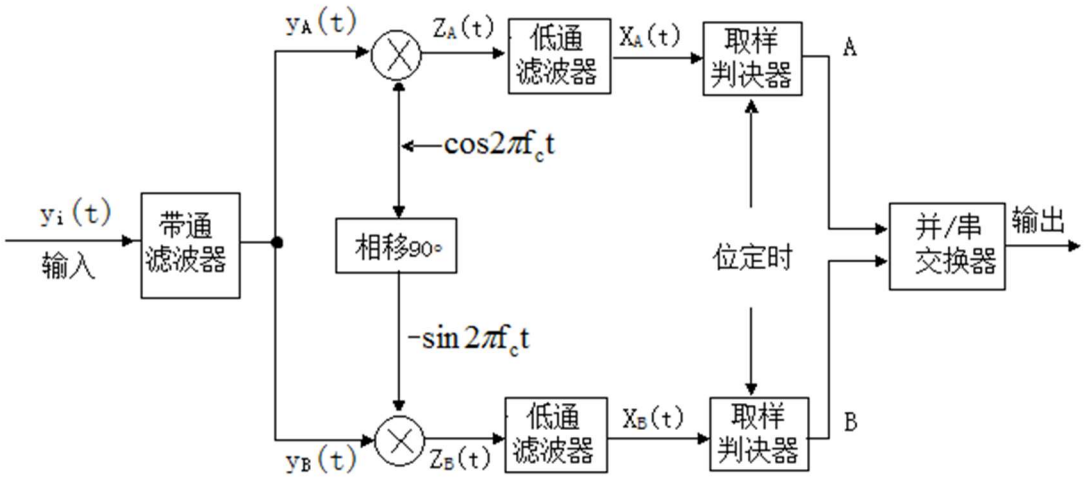

图 3 QPSK 解调框图

（2） 解调

接收机收到某一码元的 QPSK 信号可表示为：

$$y_i(\mathrm{t}) = \mathrm{a}\cos(2\pi f_c t + \varphi_n)，\ \varphi_n = \frac{\pi}{4}, \frac{3\pi}{4}, \frac{5\pi}{4}, \frac{7\pi}{4} \quad (5)$$

由 QPSK 的解调框图得到：

$$y_A(t) = y_A(t) = y_i(t) = a\cos(2\pi f_c t + \varphi_n) \quad (6)$$

$$z_A(t) = a\cos(2\pi f_c t + \varphi_n)\cos 2\pi f_c t = \frac{a}{2}\cos(4\pi f_c t + \varphi_n) + \frac{a}{2}cos\varphi_n \quad (7)$$

$$z_B(t) = a\cos(2\pi f_c t + \varphi_n)\cos(2\pi f_c t + \frac{\pi}{2} = \frac{-a}{2}\sin(4\pi f_c t + \varphi_n) + \frac{a}{2}sin\varphi_n \quad (8)$$

$$x_A(t) = \frac{a}{2}\cos\varphi_n，\ x_B(t) = \frac{a}{2}\sin\varphi_n \quad (9)$$

（3） 噪声

白噪声序列，是指白噪声过程的样本实称，简称白噪声。白噪声是在较宽的频率范围内，各等带宽的频带所含的噪声能量相等的噪声，是一种功率频谱密度为常数的随机信号或随机过程，也就是说，此信号在各个频段上的功率是一样的。

高斯噪声指的是它的概率密度函数服从正态分布的噪声。高斯分布记为 $N(\mu,\sigma^2)$，其中 μ 为高斯分布的均值，σ^2 为高斯分布的方差，当 μ=0，σ^2=1 时，该分布称为标准正态分布。高斯分布的一维概率密度可表示为式：

$$\mathrm{P(x)} = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (10)$$

在通信信道中，一般噪声的均值 μ=0。，那么可以得知当噪声的均值是零的时候，噪声的平均功率等于其方差。由于高斯白噪声能够反映实际通信信道中的噪声情况，能够比较真实的反映信道噪声的一些特性，并且可以用具体的数学表

达式表示，适合分析、计算系统的抗噪声性能，所以广泛应用于通信系统的理论分析。

在本题中，取高斯白噪声作为信道噪声。

（4） 误码率

由于信道过程中的噪声和接收机的处理，接收端的星座图不再是理想的四个点，而是发生了扩散。在噪声过大的情况下，接收到的符号可能被判错从而产生误码。

8QAM 与 16QAM 的调制与解调原理与 QPSK 类似，此处不再展开。

2.4.2 模型建立

（1） 子问题 1

首先建立 QPSK 调制的模型[2]：

1) 随机生成一个长度为 100000 的二进制比特流；
2) 将二进制的比特流转换为对应的四进制信号；
3) 生成 QPSK 调制器，通过对信号进行调制并画出信号的星座图；
4) 给一特定值的 SNR 值，在信道中加入高斯白噪声，可以得到通过信道后的星座图和眼图；
5) 生成 QPSK 解调器，并将四进制转化成二进制比特流信息；
6) 用得到的比特流信息除以原始发送的比特流信息来计算得到误码率；
7) 利用蒙特卡洛仿真，计算在该信噪比的情况下 100 次的误码率，计算出平均值，即可得到一组对应的 SNR-BER 值；
8) 改变特定的 SNR 值，重复 1）—8）的过程，得到多组对应的 SNR-BER 值；
9) 将多组 SNR-BER 进行拟合，即可得到 BER 与 SNR 的关系曲线，采用最小二乘法对曲线进行多项式拟合，可以得到 BER 与 SNR 的函数关系。

通过 BER 与 SNR 的函数关系或 BER 与 SNR 的关系曲线，可以得到当 BER=0.02 时 SNR 的容限点。

8QAM 调制与 16QAM 调制模型建立的过程与 QPSK 模型建立过程基本一致，只需要将原本的 4 进制改为 8 进制，就可以得到 8QAM 调制下的 BER 与 SNR 的函数关系，将将原本的 4 进制改为 16 进制，就可以得到 16QAM 调制下的 BER 与 SNR 的函数关系。

（2） 子问题 2

基于上面的假设，将传输过程可以简化为图 4，以发射机发射 1mW 功率的信号（不含噪声）为例进行求解。

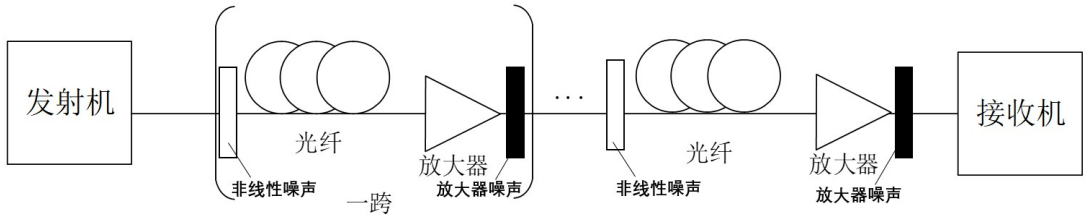

图 4 传输过程简化示意图

根据子问题 1 的结果可知，随着噪声的增加及信号在传输过程中的衰减，信噪比会减小，从而使得误码率增加，因此当误码率达到门限值时，可以根据图中

曲线找到 BER=0.02 时对应的横坐标 SNR 的值，如表 2 所示。

表 2 不同调制格式的容限点

调制格式	QPSK	8QAM	16QAM
SNR 容限点（dB）	6.62	11.34	13.48

根据信噪比 SNR 公式：

$$SNR = 10\lg\frac{p_s}{p_n} \tag{11}$$

解得当输入信号功率 p_s 为 1mW 时各个调制格式对应的临界噪声功率 p_n，如 表 3 所示。

表 3 不同调制格式的临界噪声功率

调制格式	QPSK	8QAM	16QAM
临界噪声功率（mW）	0.22	0.07	0.04

2.4.3 模型求解

（1） 子问题 1

使用 Matlab 软件对上述过程进行建模与求解，得到接收信号后的星座图，以 QPSK 调制为例，如图 5 所示：

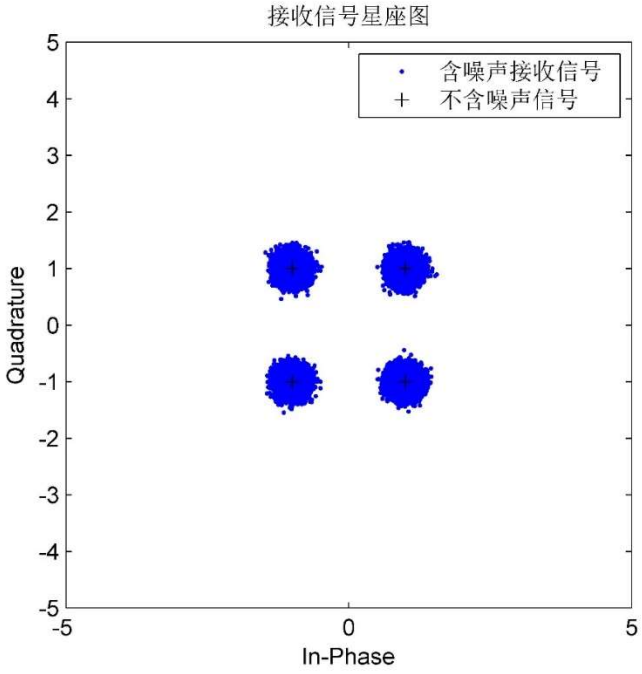

图 5 接收信号星座图

进一步可以得到 BER 与 SNR 的关系曲线如图 6 所示：

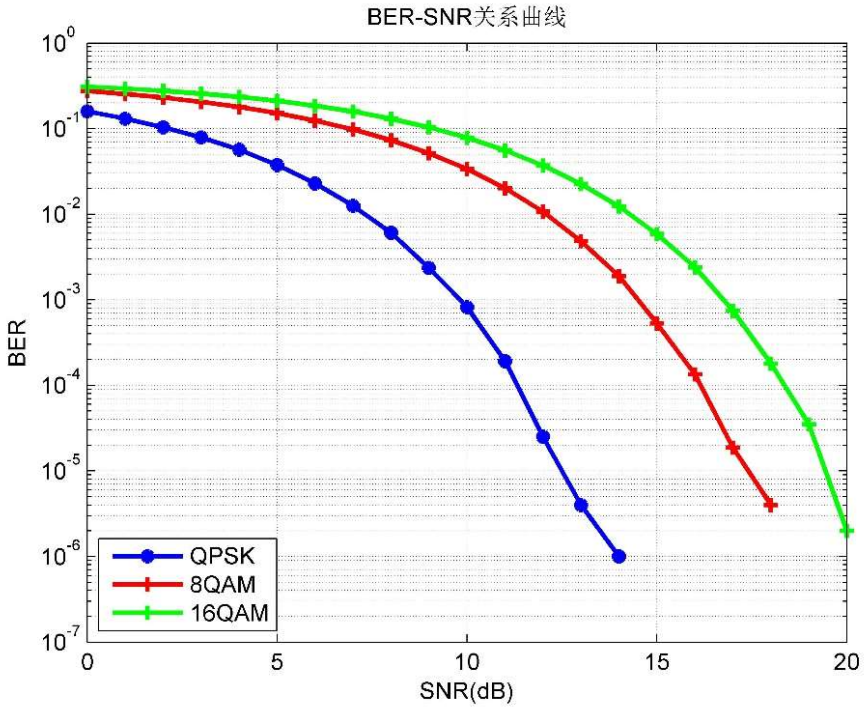

图 6 BER-SNR 关系曲线

使用 Matlab 对 QPSK、8QAM 和 16QAM 三种调制方式利用最小二乘法多项式拟合，以保证拟合误差值在 1%左右为目标确定出拟合多项式的阶数。

得到 QPSK 调制的拟合阶数为 4 阶，拟合得到的函数关系为：

$$BER = 1.324 \times 10^{-6} SNR^4 - 1.068 \times 10^{-4} SNR^3 + 0.003 SNR^2 - 0.038 SNR + 0.164$$

得到 8QAM 调制的拟合阶数为 5 阶，拟合得到的函数关系为：

$$BER = 1.922 \times 10^{-7} SNR^5 - 1.562 \times 10^{-5} SNR^4 + 4.301 \times 10^{-4} SNR^3 - 0.0039 SNR^2 - 0.143 SNR + 0.274$$

得到 16QAM 调制的拟合阶数为 6 阶，拟合得到的函数关系为：

$$BER = 7.845 \times 10^{-9} SNR^6 - 5.4233 \times 10^{-7} SNR^5 + 8.963 \times 10^{-6} SNR^4 + 1.109 \times 10^{-4} SNR^3 - 0.0029 SNR^2 - 0.00082 SNR + 0.306$$

根据 BER 与 SNR 关系曲线可得，在 BER=0.02 时，QPSK 调制的 SNR 容限点为 6.62dB，8QAM 调制的 SNR 容限点为 11.34dB，16QAM 调制的 SNR 容限点为 13.48dB。

（2） 子问题 2

1) 当单跨传输距离为 80km 时：

因为信号每传输 15km，光功率衰减一半，所以在 80km 单跨中光功率衰减为$(\frac{1}{2})^{\frac{16}{3}}$，因此放大器的放大增益（Gain）为 $2^{\frac{16}{3}}$，再将题目中的各个系数值代入下式求解放大器噪声 p_{nf} ：

$$p_{nf} = 2\pi hfB(NF - \frac{1}{Gain})$$

求得 $p_{nf} = 1.5\times10^{-4} mw$。

同时非线性噪声与放大器噪声成正比例关系，非线性噪声约等于单个放大器噪声的 2/3，因此非线性噪声 $p_{ng} = 1.0\times10^{-4} mw$。

单跨的噪声 $p_{nt} = 2.5\times10^{-4} mw$。

在计算最后一跨时，由于放大器噪声和非线性噪声是加性噪声，因此最后只需要取舍最后整跨段，最终求得个调制方式的最远传输距离的结果如表 4 所示。

表 4 每跨 80km 时的最远传输距离

调制格式	QPSK	8QAM	16QAM
最远传输距离（km）	70400	22400	12800
跨段数量	880	280	160

2) 当单跨传输距离为 100km 时：

因为信号每传输 15km，光功率衰减一半，所以在 100km 单跨中光功率衰减为$(\frac{1}{2})^{\frac{20}{3}}$，因此放大器的放大增益（Gain）为 $2^{\frac{20}{3}}$。

$$p_{nf} = 2\pi hfB(NF - \frac{1}{Gain})$$

求得 $p_{nf} = 1.6\times10^{-4} mw$。

同时非线性噪声与放大器噪声成正比例关系，非线性噪声约等于单个放大器噪声的 2/3，因此非线性噪声 $p_{ng} = 1.07\times10^{-4} mw$。

单跨的噪声 $p_{nt} = 2.67\times10^{-4} mw$。

在计算最后一跨时，由于放大器噪声和非线性噪声是加性噪声，因此最后只需要取舍最后整跨段，最终求得个调制方式的最远传输距离的结果如表 5 所示：

表 5 每跨 100km 时的最远传输距离

调制格式	QPSK	8QAM	16QAM
最远传输距离（km）	82400	26200	15000
跨段数量	824	262	150

2.4.4 结果分析

对于子问题 1，可以发现越复杂的调制方式，在 BER=0.02 时对应的 SNR 容限点越大。

对于子问题 2 可以发现，单跨距离长，信号传输的最远距离会增加，其中因为该问题将噪声简化为两端的加性噪声且中间传输过程信噪比不变的较为理想情况，所以增加跨段传输距离可以增加信号传输距离。

此外还发现，越长位数的调制格式传输的最远距离越小，这是因为调制格式位数越多，在编码中出现解码错误的概率越大，反映在 SNR-BER 图中也是同一信噪比下，更多位数的调制方式误码率越大，所以总定性分析的角度，结果也是符合预期的。

2.5 模型的评价与总结

对于子问题 1，蒙特卡洛仿真算法的应用提高了计算的准确性，但是由于是拟合曲线，所以相比真实值还是略有一定的差距。

对于子问题 2，模型建立过程中对于不同的噪声有全面的考虑，但是由于问题的解决应用了较多的假设，在今后还有较大的提升空间。

3 问题 2

3.1 问题分析

对于子问题 1，是一种最小生成树问题。因此首先我们通过国家统计局得到不同城市的人口，利用网络地图，可得到任意 2 座城市间的距离。因此在认为每条连接权重都为 1 的条件下，可以得到任意两个城市之间的网络价值。按照网络价值从大到小取前 16 条连接，分析这 16 条连接经过的城市，对于未经过的城市，依次向下取新的连接来代替原来 16 条连接中的网络价值较小且该网络连接所连接的城市有不止一条连接，直到 12 座城市都有连接经过。此时的 16 条连接的网络价值之和即为最高的网络价值。在剩余的 50 条连接中，取网络价值最高的 17 条连接加入到原本的网络规划中，与原来的 16 条连接共同得到一个 33 条连接数的网络拓扑，且此时的网络规划为 33 条连接网络规划中的最大方案。

对于子问题 2，我们从线性规划的角度去求解，其中考虑到了最短路径算法得到不同城市之间的连接。首先根据子问题 1 中所选的 12 个城市，需要规划 16 条路径，计算出任意一个城市与其他城市的直线连接数量，根据连接数量从小到大对城市进行排序，这样可选择出与其他城市连接数量最少的城市，然后找到该城市与其他城市的所有连接，并将各个连接设置不同的初始权重，当该城市与其他城市的所有连接都遍历结束之后，将该城市从图中删除，继续选取一个连接数量最小的城市进行遍历，直到图中的城市数目为空为止。这样可以得到所有城市之间需要分得的单波传输容量数目，进而可以设计得到所有城市的网络价值总和，多个路线之间分得的带宽的总容量作为多个网络带宽限制条件。然后对网络价值总和进行约束，通过线性规划算法得到最优解；同理，对于 33 个城市连接，也可以在子问题 1 的基础上按照上述方法进行求解。如果扩大为省，人口数量会存在一定的变化，而且网络中节点数目变多，但是计算方法不会改变，只是需要处理的数据点更多，不同分支的网络带宽容量需要重新进行调整。

对于子问题 3，是子问题 2 基础上增加了条件约束，并修正了目标函数，仍然基于线性规划的思想求解目标函数。从运营商的角度考虑，设置了代价系数，代价系数与城市之间的距离成正比，距离越远所要耗费的资源代价越大，新的目标函数需要在原来问题 2 的基础上减去代价系数矩阵乘两个网络之间人口容量的值，如果需要考虑其他因素，同样可以在此基础上增加系数和权重，比如，若考虑地区之间的平衡发展，可以先通过聚类算法，对区域的经济发展水平进行聚类，找到经济最好的区域的坐标作为中心点，然后计算不同城市到达中心点的距离，距离越近，说明该地区的经济形势较好，距离越远，说明经济欠发达，然后设置一个经济发展系数矩阵，原来的目标函数要加上经济发展系数乘两个地区之间的人口容量，以此作为目标函数重新进行求解，这样便可以兼顾地区的经济发展水平。

3.2 模型假设

（1） 在问题 2 中未提及 12 个城市是否都要在网络规划中，从保障全国通信的角度出发，假设要求所有城市都需要进行网络规划。

（2） 对于两座城市之间的距离，认为光纤可以进行直线布局，故假设两座城市之间的距离为两座城市市政府之间的直线距离[1]。

[1] 数据来源高德地图网页版

3.3 变量说明

表 6 问题 2 变量说明

符号	意义	单位	备注
V_i	城市节点 i		
E(i,j)	城市 V_i 与城市 V_j 之间的连接		
connect(V_i,V_j)	城市 V_i 与城市 V_j 之间的带宽容量	Tb/s	
dist(V_i,V_j)	城市节点 V_i, V_j 之间的距离	km	
popu(Vi, Vj)	城市 V_i 与城市 V_j 之间的人口容量	m(百万人)	
pep(i)	城市 V_i 的人口数目	m(百万人)	
value(Vi, Vj)	城市 V_i 与城市 V_j 之间的网络价值	mTb/s	
K	城市之间的最大连接数	个	
N	城市数目	个	
X	带宽容量矩阵		
A	人口权重矩阵		
F	目标函数		
Cap(Vi, Vj)	最大容量	Tb/s	
λ	距离权重因子		
α	目标函数调节系数		
eco(V_i)	城市 V_i 的经济发展水平		
diseco(V_i,V_j)	城市 V_i 到城市 V_j 之间的经济发展差异		
Y	耗材代价矩阵		
Z	经济发展差异矩阵		
β	经济发展水平权重因子		

3.4 模型的建立与求解

3.4.1 理论基础

（1） 子问题 1 采用改进的最小生成树算法

基于改进的最小生成树算法（Kruskal 算法），用于选取价值较高的城市连接；克鲁斯卡尔算法是一种用来寻找最小生成树的算法。在剩下的所有未选取的边中，找最小边，如果和已选取的边构成回路，则放弃，选取次小边。先构造一个只含 n 个顶点、而边集为空的子图，把子图中各个顶点看成各棵树上的根结点，之后，从网的边集 E 中选取一条权值最小的边，若该条边的两个顶点分属不同的树，则将其加入子图，即把两棵树合成一棵树，反之，若该条边的两个顶点已落在同一棵树上，则不可取，而应该取下一条权值最小的边再试之。依次类推，直到森林中只有一棵树，也即子图中含有 n-1 条边为止。

（2）子问题 2 采用 Dijkstra 算法求解最短距离

Dijkstra 算法是由荷兰计算机科学家 Dijkstra 于 1959 年提出的。是从一个顶点到其余各顶点的最短路径算法，解决的是有向图中最短路径问题。迪杰斯特拉算法主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。

按路径长度递增次序产生算法：

把顶点集合 V 分成两组：

1) S：已求出的顶点的集合（初始时只含有源点 V0）
2) V-S=T：尚未确定的顶点集合

将 T 中顶点按递增的次序加入到 S 中，保证：

1) 从源点 V0 到 S 中其他各顶点的长度都不大于从 V0 到 T 中任何顶点的最短路径长度；
2) 每个顶点对应一个距离值。

S 中顶点：从 V0 到此顶点的长度。

T 中顶点：从 V0 到此顶点的只包括 S 中顶点作中间顶点的最短路径长度

依据：可以证明 V0 到 T 中顶点 Vk 的，或是从 V0 到 Vk 的直接路径的权值；或是从 V0 经 S 中顶点到 Vk 的路径权值之和。

（3）子问题 2，3 中采用 linprog 算法求解目标函数

linprog 是求解线性规划问题，f,x,b,beq,lb,ub 为向量,A,Aeq 为矩阵。功能：最小化带有参数项的线性规划问题。其中 options 可以使用 optimset 来设置。

功能：对 problem 求最小值，其中 problem 是一个结构体。

3.4.2 模型建立

该模型基于数据结构中的图论知识，城市用图中的节点表示，记为 V，其中 V_1-V_{12} 分别表示各个城市；城市之间的连接用边 E 表示，比如 E(i,j)表示城市 V_i 与城市 V_j 之间的连接；每条边有不同的权重表示城市之间的带宽大小，这里我们用 connect(V_i,V_j)表示带宽容量，城市之间的连接不考虑顺序（比如“北京-上海”和“上海-北京”可视为同一条边），因此可以看做带权无向图问题进行求解。

（1） 子问题 1

针对问题 2-1，首先获取到城市的原始数据如表 7 所示，实际编程中我们用图的邻接矩阵代表不同的连接关系，两个城市节点 V_i, V_j 之间的距离用 dist(V_i,V_j)表示。

表 7 不同城市间距离统计

	哈尔滨	北京&天津	乌鲁木齐	上海	郑州	西安	武汉	重庆	成都	拉萨	广州&深圳	昆明
哈尔滨	-	1060	3086	1702	1048	1979	1998	2510	2580	3554	2800	3120
北京&天津	1060	-	2098	1117	625	906	1066	1480	1540	2560	1890	2100
乌鲁木齐	3086	2098	-	3270	2450	2114	2770	2310	2060	1610	3290	2520
上海	1702	1117	3270	-	830	1225	688	1445	1660	2900	1215	1960
郑州	1048	625	2450	830	-	430	465	880	1005	2180	1295	1510
西安	1979	906	2114	1225	430	-	655	580	620	1755	1720	1210
武汉	1998	1066	2770	688	465	655	-	886	990	2230	840	1300
重庆	2510	1480	2310	1445	880	580	886	-	265	1490	980	635
成都	2580	1540	2060	1660	1005	620	990	265	-	1250	1280	645
拉萨	3554	2560	1610	2900	2180	1755	2230	1490	1250	-	2370	1270
广州&深圳	2800	1890	3290	1215	1295	1720	840	980	1280	2370	-	1080
昆明	3120	2100	2520	1960	1510	1210	1300	635	645	1270	1080	-

根据表 7 以及表 8，可以得到两个城市 i，j 之间的最大带宽 connect(Vi, Vj)。

表 8 不同传输格式的传输距离

单波传输容量	最大传输距离	总容量
100 Gb/s	3000 km	8 Tb/s
200 Gb/s	1200 km	16 Tb/s
400 Gb/s	600 km	32 Tb/s

具体计算公式如公式（12）所示：

$$\text{connect}(V_i,V_j)=\begin{cases}32Tb/s & 0<dist(V_i,V_j)\le 600km\\ 16Tb/s & 600km<dist(V_i,V_j)\le 1200km\\ 8Tb/s & 1200km<dist(V_i,V_j)\le 3000km\\ 0 & 3000km<dist(V_i,V_j)\end{cases} \quad (12)$$

通过查找不同城市之间的人口数量，得到如下信息，如表 9 所示。

表 9 不同城市人口[1]

	哈尔滨	北京&天津	乌鲁木齐	上海	郑州	西安	武汉	重庆	成都	拉萨	广州&深圳	昆明
人口数目（m）	1064	3735	268	2420	972	883	1071	3048	1717	55	2595	673

每条直接连接两个城市/区域的链路当做 1 个连接，每个连接的价值定义为传输的容量与连接区域人口数的乘积，本文用 pep(i)表示城市 V_i 的人口数目，城市 i，j 之间的人口容量用 popu(Vi, Vj)表示，可以得到人口容量的计算公式如（13）所示：

$$popu(V_i,V_j)=\sqrt{pep(i)*pep(j)},i\neq j \quad (13)$$

根据两个城市之间的人口容量 popu(Vi, Vj)和带宽 connect(Vi, Vj)，问题 1 中的链路权值为 1，进而可以得到两个城市之间的网络价值 value(Vi, Vj)为：

$$value(V_i,V_j)=popu(V_i,V_j)*\text{connect}(V_i,V_j) \quad (14)$$

按照如下改进的最小生成树算法（严格意义上来说，论文中的算法最终生成的已经不是一棵树而且图的形式，但是计算思想与之相似，所以命名为改进的最小生成树算法）对问题 1 进行求解，假设连接条数为 k:

1) 根据两个城市之间的网络价值 value(Vi, Vj)大小，从大到小对城市连接 E(i,j)进行排序；
2) 选取前 k 个网络价值较大的城市连接按照顺序放入集合 Con1 中，剩余的连接按照顺序放入集合 Con2；

[1] 数据来源：国家统计局 2016 年数据

3) 定义城市集合 V1,将所有的城市放入集合 V1 中，遍历 Con1 中各个出现城市 V_i，如果在城市集合 V1 中出现，那么从城市 V1 集合中将该城市剔除，如果最终解得城市集合 V1 为空，那么算法结束，得到所有的城市连接；如果城市集合不为空，那么继续执行步骤 4);
4) 将连接 Con2 集合中的城市按照网络价值大小从高到低排序，将连接 Con1 集合中的元素按照网络价值从低到高排序，首先从城市集合 V1 中选择一个剩余元素 V_j, 从 Con2 中选择含有城市 V_j，并且价值最高的连接 C_m，然后从 Con1 集合中选择价值较低的连接 C_n，并且 C_n 满足该连接的任意一个城市在集合 V1 中出现的次数大于 1，然后将 C_n 从集合 Con1 中剔除，将 C_m 放入集合 Con1 中，此时需要将城市集合 V 中的元素 V_j 剔除，重复上述步骤，直到城市集合 V 中的元素为空即可。

（2） 子问题 2

1) 根据子问题 1 可以得到 N 个城市的列表为 V；
2) 计算列表中的各个城市与其他城市的连接条数，并按照连接数从小到大进行排序，按照顺序存储在城市列表 V 中；
3) 首先选择排序最前面的城市 Vi, Vi 表示当前集合中连接数目最少的城市，进而可以根据 Vi 得到与其他各个城市的最短路径(这里的最短路径算法采用狄更斯特拉最短路径算法求解)，用邻接矩阵表示保存各个节点之间的连接，然后从邻接矩阵中删除城市 Vi ，继续执行步骤 3，直到所有的城市被删除为止，在求解的过程中记录路径的最大连接数为 K；
4) 定义一个带宽容量矩阵 X，其中城市数目用 N 表示，矩阵的维度为 N*N*K，其中 K 为步骤(3)所求的最大连接数目，带宽容量矩阵中的 Xi,j,k 表示从城市 Vi 到城市 Vj, 中间第 k 条连接的带宽值，重新执行步骤 3）生成带宽容量矩阵 X，执行过程中如果 Vi 到城市 Vj 有网络连接的那么 Xi,j,k 随机设置一个初始值，没有网络连接的设置为 0 即可，城市人口的权重矩阵为 A, Ai,j 表示城市 Vi 到城市 Vj 的人口数目，可以通过 popu(Vi, Vj)进行计算，那么两个城市之间的网络价值 value(Vi, Vj)可以表示为：k 为 K 个网络带宽容量中的任意一个，且保证 $X_{i,j,k} \neq 0$

$$value(Vi,\ Vj) = A_{i,j} * X_{i,j,k}$$

5) 进而可以得到整个网络的目标函数 F 为：

$$F = \sum_{i}^{N}\sum_{j}^{N} value(V_i, V_j) = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{i,j} * X_{i,j,k}, (k \in \forall K, X_{i,j,k} \neq 0) \quad (15)$$

6) 定义函数的约束条件，其中对于每两个节点之间的连接 E(i,j)中间最多有 K 个连接，Xi,j,k 表示节点 Vi 到节点 Vj 的第 k 个网络流量，但是总容量不超过每条节点之间的最大容量 Cap(Vi, Vj)，Xi,j,k 需要满足以下条件：

$$connect(V_i, V_j) = \sum_{k=1}^{K} X_{i,j,k} = \sum_{k=1}^{K} connect(V_i, V_j)_k \leq Cap(V_i, V_j)$$

7) 最终通过线性规划的方法根据上述目标函数和约束条件进行求解；采用 linprog 算法模型带入约束条件进行求解，目标函数为 F，约束条件为（15）

其中由市扩大为省（区）影响其实是在考虑人口数目的变化以及城市数量的增多，会对题目造成什么样的影响，本质上仍然可以采取上述算法进行解决，具有一定的可扩展性。

（3） 子问题 3

1) 光传送网络价值有多个侧面，需要在子问题 2 基础上重新定义目标函数，比如考虑耗材代价，可以定义目标函数，根据距离计算代价，引入耗材代价矩阵 Y，对应的各个城市之间的连接，其中矩阵 Y 的参数跟每个城市之间的距离有关，与距离成正比，距离越大，造价越高；例如城市 Vi 到城市 Vj 之间的距离为 dist(Vi,Vj)，那么 Yi,j 表示两个城市之间的耗材代价:（为了简化问题，我们近似认为城市 Vi 到城市 Vj 之间不同带宽的耗材代价相等，λ 表示距离权重因子）

$$Y_{i,j,k} = Y_{i,j} = \lambda * dist(i,j)$$

2) 然后目标函数 F 需要定义为（α_1 表示调节系数）：

$$F = \sum_{i}^{N}\sum_{j}^{N} value(V_i,V_j) = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{i,j} * (X_{i,j,k} - \alpha_1 Y_{i,j,k}), (k \in \forall K, X_{i,j,k} \neq 0) \quad (16)$$

3) 经济发展问题可以在问题 2-2 的基础上更新目标函数，首先我们可以通过聚类算法得到不同城市之间的经济发展热度，然后选取每个中心点，进而可以得到经济最发达的地区，那么可以认为离经济最发达的地区距离越近，表示经济发展水平越高，距离最发达的地区越远，经济发展水平越低；如果连接欠发达的城市，从政府的角度保障了发展相对滞后地区的通信要求，eco(Vi)表示城市 Vi 的经济发展水平，城市 Vi 到城市 Vj 之间的经济发展差异为 diseco(Vi,Vj),计算方法如下：

$$di\sec o\left(V_i,V_j\right) = |\, eco(V_i) - eco(V_j)\,| \quad (17)$$

4) 这里引入经济发展差异矩阵 Z，其中 Zi,j,k 表示城市 Vi 到城市 Vj 之间的经济发展差异，可通过 diseco(Vi,Vj)进行计算，其中经济发展差异越大，连接这条线路会对目标函数有正面的影响，反之，会阻碍欠发达地区的通信，Zi,j,k 计算方法如下：（为了简化问题，我们近似认为城市 Vi 到城市 Vj 之间不同带宽的经济发展差异相同，β 表示经济发展的权重因子）

$$Z_{i,j,k} = Z_{i,j} = \beta * di\sec o(V_i,V_j) \quad (18)$$

5) 目标函数 F 需要定义为（α_2 表示调节系数）：

$$F = \sum_{i}^{N}\sum_{j}^{N} value(V_i,V_j) = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{i,j} * (X_{i,j,k} + \alpha_2 Z_{i,j,k}), (k \in \forall K, X_{i,j,k} \neq 0)$$

6) 其中步骤 1）为负面的代价，步骤 2）为正面的代价，如果有新的代价，可以按照步骤 1）、2）的方法设计目标函数，重新进行计算即可，同样也要考

虑引入不同的代价带来的约束条件的变化。

7) 子问题 3 本质上是考察引入不同的参数会对目标函数带来哪些影响，首选需要引入的参数对于目标函数而言，是正面的还是负面的，然后在按照合适的方法对前面的原目标函数进行修正，还要注意对约束条件的更新。

3.4.3 模型求解

（1） 子问题 1

由于子问题 1 不考虑中间节点，由上述公式可以计算出两个城市之间的网络价值，当有 16 条连接时，选择的连接如表 10 所示(注：为了方便编程，表中的城市编号从 0-11，代表实际中的城市 1-城市 12)，计算过程如附件 6.1.6 所示。

表 10 16 条连接时网络价值情况

编号	城市连接	网络价值（mTb/s）
1	(7,8)	73205.31390548093
2	(5,7)	52497.40199286056
3	(1,3)	48103.089297881896
4	(7,10)	44998.30396803862
5	(4,6)	32649.598588650366
6	(1,6)	32000.73999144395
7	(0,1)	31895.99097065335
8	(1,4)	30485.85770484406
9	(4,5)	29645.819671582703
10	(1,5)	29056.656380251323
11	(6,7)	28908.27646194079
12	(4,7)	27539.788234479944
13	(1,7)	26992.478952478596
14	(7,11)	22915.79856780034
15	(1,2)	8003.919040070308
16	(1,9)	3625.906783137151

最优的网络价值连接情况如图 7 所示。

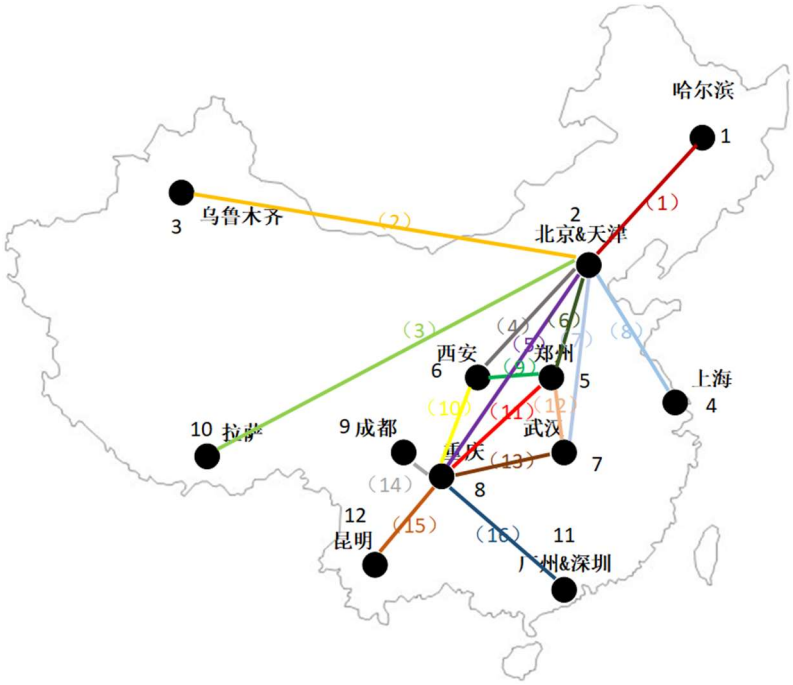

图 7 16 条连接时的最优规划

当有 33 条连接时，选择的连接如表 11 33 条连接时网络价值情况表 11 所示，计算过程如附件 6.1.6 所示。

表 11 33 条连接时网络价值情况

编号	城市连接	网络价值（mTb/s）
1	(7,8)	73205.31390548093
2	(5,7)	52497.40199286056
3	(1,3)	48103.089297881896
4	(7,10)	44998.30396803862
5	(4,6)	32649.598588650366
6	(1,6)	32000.73999144395
7	(0,1)	31895.99097065335
8	(1,4)	30485.85770484406
9	(4,5)	29645.819671582703
10	(1,5)	29056.656380251323
11	(6,7)	28908.27646194079
12	(4,7)	27539.788234479944
13	(1,7)	26992.478952478596
14	(6,10)	26673.708403594726
15	(3,6)	25758.60865807779
16	(1,10)	24905.99927728257
17	(3,4)	24539.222481570192
18	(7,11)	22915.79856780034
19	(3,7)	21727.26950171144

20	(6,8)	21697.008826103196
21	(10,11)	21144.44040404002
22	(4,8)	20669.894629629827
23	(1,8)	20259.113504790876
24	(3,10)	20047.78291981435
25	(5,8)	19700.873483173276
26	(8,11)	17199.374872360913
27	(8,10)	16886.66219239314
28	(3,8)	16307.328413937092
29	(0,4)	16271.362819383015
30	(5,6)	15559.47968281716
31	(0,7)	14406.825049260508
32	(1,2)	8003.919040070308
33	(1,9)	3625.906783137151

（2） 子问题 2

1) 根据附件 6.1.1 代码，计算得到的城市之间的距离矩阵（单位：km）。

表 12 城市之间的距离矩阵

0	1060	3086	1702	1048	1979	1998	2510	2580	3554	2800	3120
0	0	2098	1117	625	906	1066	1480	1540	2560	1890	2100
0	0	0	3270	2450	2114	2770	2310	2060	1610	3290	2520
0	0	0	0	830	1225	688	1445	1660	2900	1215	1960
0	0	0	0	0	430	465	880	1005	2180	1295	1510
0	0	0	0	0	0	655	580	620	1755	1720	1210
0	0	0	0	0	0	0	886	990	2230	840	1300
0	0	0	0	0	0	0	0	265	1490	980	635
0	0	0	0	0	0	0	0	0	1250	1280	645
0	0	0	0	0	0	0	0	0	0	2370	1270
0	0	0	0	0	0	0	0	0	0	0	1080
0	0	0	0	0	0	0	0	0	0	0	0

2) 根据附件 6.1.2 代码计算得到的城市之间的带宽总容量矩阵(单位：Tb/s)。

表 13 城市之间的带宽总量矩阵

0	16	0	8	16	8	8	8	8	0	8	0
0	0	8	16	16	16	16	8	8	8	8	8
0	0	0	0	8	8	8	8	8	8	0	8
0	0	0	0	16	8	16	8	8	8	8	8
0	0	0	0	0	32	32	16	16	8	8	8
0	0	0	0	0	0	16	32	16	8	8	8
0	0	0	0	0	0	0	16	16	8	16	8

0	0	0	0	0	0	0	0	32	8	16	16
0	0	0	0	0	0	0	0	0	8	8	16
0	0	0	0	0	0	0	0	0	0	8	8
0	0	0	0	0	0	0	0	0	0	0	16
0	0	0	0	0	0	0	0	0	0	0	0

3) 根据附件 6.1.3 代码计算得到的城市之间的人口数矩阵(单位：m)。

表 14 城市之间的人口数矩阵

0	1993	534	1605	1017	969	1067	1801	1352	242	1662	846
0	0	1000	3006	1905	1816	2000	3374	2532	453	3113	1585
0	0	0	805	510	486	536	904	678	121	834	425
0	0	0	0	1534	1462	1610	2716	2038	365	2506	1276
0	0	0	0	0	926	1020	1721	1292	231	1588	809
0	0	0	0	0	0	972	1641	1231	220	1514	771
0	0	0	0	0	0	0	1807	1356	243	1667	849
0	0	0	0	0	0	0	0	2288	409	2812	1432
0	0	0	0	0	0	0	0	0	307	2111	1075
0	0	0	0	0	0	0	0	0	0	378	192
0	0	0	0	0	0	0	0	0	0	0	1322
0	0	0	0	0	0	0	0	0	0	0	0

4) 根据附件 6.1.4 代码计算得到城市之间的网络价值矩阵（单位：mTb/s）。

表 15 城市之间的网络价值矩阵

·	31896	0	12837	16271	7754	8540	14407	10813	0	13293	0
0	0	8004	48103	30486	29057	32001	26992	20259	3626	24906	12684
0	0	0	0	4083	3892	4286	7230	5427	971	0	3398
0	0	0	0	24539	11694	25759	21727	16307	2919	20048	10210
0	0	0	0	0	29646	32650	27540	20670	1850	12706	6470
0	0	0	0	0	0	15559	52497	19701	1763	12110	6167
0	0	0	0	0	0	0	28908	21697	1942	26674	6792
0	0	0	0	0	0	0	0	73205	3276	44998	22916
0	0	0	0	0	0	0	0	0	2458	16887	17199
0	0	0	0	0	0	0	0	0	0	3022	1539
0	0	0	0	0	0	0	0	0	0	0	21144
0	0	0	0	0	0	0	0	0	0	0	0

5) 根据附件 6.2.4 代码计算得到的邻接矩阵，用于算法 6.2.8 求解节点之间的最短路径（1 表示两个节点之间有路径，0 或者∞表示两点之间不可达）。

表 16 节点之间的邻接矩阵

0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	1	1	1	1	1	∞	1	∞	∞
∞	1	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	1	∞	0	∞	∞	∞	∞	∞	∞	∞	∞
∞	1	∞	∞	0	1	1	1	∞	∞	∞	∞
∞	1	∞	∞	1	0	∞	1	∞	∞	∞	∞
∞	1	∞	∞	1	∞	0	1	∞	∞	∞	∞
∞	1	∞	∞	1	1	1	0	1	∞	1	1
∞	∞	∞	∞	∞	∞	∞	1	0	∞	∞	∞
∞	1	∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	1	∞	∞	0	∞
∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	0

3.4.4 结果分析

子问题 1 根据网络价值的计算公式，在权重为 1 的情况下，人口数量与网络容量的计算结果是可以穷举的，因为根据计算结果进行排序，那么取值较大的进行求和会得到较大的网络价值，但是要兼顾各个城市都要有连接，因此需要找到没有连接的城市，从剩余连接中仍然选权重较大的，而且从前面剔除出现城市数目较多的连接，具有一定的合理性；

子问题 2 通过采用网络带宽系数矩阵与人口容量乘积的设计目标函数去求解，进而转换成线程方程组的形式去求解，极大简化了计算过程，而且容易进行拓展，求解最短路径的过程中，采用 Dijkstra 算法，寻找从当前节点出发到其他节点的最短路径，跟实际查找情况一致，具有一定的合理性

子问题 3 是在子问题 2 基础上的拓展，可以通过分析所加的限制条件与目标函数的关系，进行扩展，扩展的方法也很简单，在原来系数的基础上，根据权重关系，对目标函数进行更新和拓展，不过对于一些特殊条件，需要注意限制条件的更新，通过矩阵之间的运算，减少了计算量。问题答案符合实际情况。

3.5 模型的评价与总结

子问题 1 采用的改进的最小生成树算法 Kruskal 算法，因为会产生环行结构，构成图，这里面思想还是一致的，选取价值较大的边放入其中，改进的地方在于对已有连接进行剔除，从而保证各个城市都能连接；算法具有一定的可扩展性，只需要传入节点的数目，即可生成最优解；

子问题 2 将原问题进行拆分，转换成了多个条件下的线性规划问题，该模型采用 Dijkstra 算法选区当前城市到其他城市的最短路径，减少了流量的浪费，而且根据带宽权重与城市之间的人口容量进行叠加，将原问题转换成多维向量的形式进行表示，具有一定的可扩展性和可计算性

子问题 3 在子问题 2 的模型基础上进行拓展，增加了限制条件，更新了目标函数，同样采用 向量的形式对原来的函数进行修正，简单易于计算，拓展性强

总体来看，问题 2 用于根据不同的场景对网络权重进行划分，从简单到复杂，子问题 1 不考虑中间节点的情况下，解决方案相对容易，可以得到较为固定的最优解，子问题 2 在考虑中间节点的情况下，我们将原问题简化，不断缩小图中节

点的数目和连接，防止重复计算，最后采用线性方程组的形式对问题进行求解，把参数权重定义在一个多维矩阵中，从子问题 3 的计算中也可以看出，该问题可扩展性好，设计合理。

4 问题 3

4.1 问题分析

问题 3 需要对 16QAM 方案进行改进，设计一种优化的调制解调方案达到比题目中的图五 8QAM 调制具有更低的 SNR 容限点的调制方案。在设计的过程中，最关键的原则是要保证调制格式的信息熵为 3bit。[3]

其中一类优化方案是三维空间的星座图方案如图 8 所示，将 16QAM 星座图的 16 个符号 S_i 在三维坐标轴 xyz 构成的空间中排布，再通过将图 8 中内部正方体逆时针旋转 π/4，如图 9 所示，进一步使符号空间分布合理，使符号点间欧式距离更大，且相邻信号点的距离相同，以 S_5 为例，假设 $S_{x,6} = b$, $S_{y,6} = 0$, $S_{z,6} = b$, $S_{x,5} = a$, $S_{y,5} = a$, $S_{z,5} = a$, $S_{x,4} = 0$, $S_{y,4} = b$, and $S_{z,4} = b$，同时根据条件 S_5 到 S_6 和 S_5 到 S_4 的距离都为：

$$\sqrt{a^2+(a-b)^2+(a-b)^2}=\sqrt{b^2+b^2}\ ,\ a>0,\ b>0 \qquad (19)$$

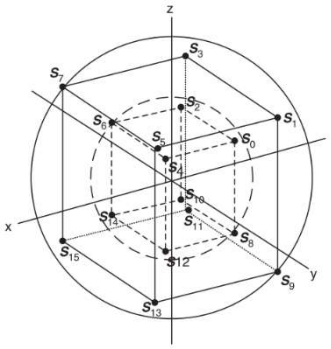

图 8 三维空间星座图

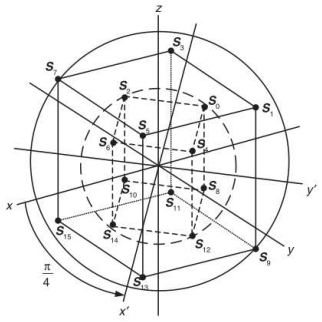

图 9 优化后的三维空间星座图

同时，假设符号在星座图的映射归一为单位能量，则用公式表达为：

$$\sqrt{\frac{8(a^2+a^2+a^2)+8(b^2+b)^2}{16}}=1 \qquad (20)$$

基于该改星座图映射进行仿真计算可以得到信噪比（SNR）与误码率（SEP）系曲线，如图 10 所示。可以看出，信号传输相对于题目中 16QAM 的二维平面排布有了很大的提高，具有更低的 SNR 容限点。

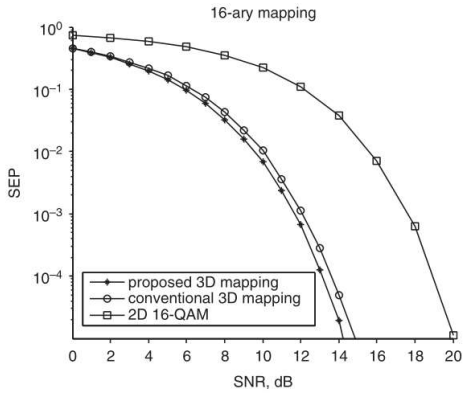

图 10 改进前后误码率比较

改进后的星座图空间排布方案由于依旧保留了 16 个信息符号，且每个信息符号的概率相同，均为 p_i=1/16，因此该种方案的信息熵为：

$$\Omega=\sum_{i=1}^{16} p_i \log_2 p_i = 4\text{bit} \qquad (21)$$

可以发现传统的等概映射方式显然不能够解决问题，为了保证调制方案的信息熵格式为 3bit，需要进一步减省信号符号的个数，并且调整每个信号点的概率，因此需要考虑新的不等概星座映射，达到低信息熵传输的目标。[4]

4.2 模型的建立与求解

4.2.1 理论基础

星座成形可通过寻找最佳星座点的位置或是利用不等间隔的信号星座来获得成形增益。概率成型的基本原理是使能量较低的信号被使用的概率高于能量较高的信号。因能量较高的信号被使用的概率降低，信号的平均功率就会降低，从而可以节省发射功率。概率成形可采用在信号空间叠加的方法来实现不等概的输入分布，也可利用成形码使能量较低的信号被使用的概率大于能量高的信号。[5]

不等概星座在在实际星座图中的表现形式为，某些星座点只对应一个标号(label)，而有的星座点对应了多个标号。不等概的 Gallager 映射可以做到这一点。

假设$\{P(x)|x\in A_x\}$ 是一个与前述定义星座图 A_x 有关的离散概率分布函数，其中，$x=M(v')$，概率 P(x)代表星座图中某一个星座点被使用的概率。对于传统的等概星座来说，每一个星座点在调制的时候被使用的概率是相等的，对应的概率分布表达式记为 P(x) = 1/M。例如 QPSK 调制，每个星座点被使用的概率为 $P(x)=1/4$。定义一个映射函数 $MP(x)(v')$：$\{0,1\}^T\rightarrow A_x$，该映射函数表示将长度为 T 的采样序列 v′映射到星座图 A_x 中的某一个星座点 x 上。显然，映射到某一个星座点 x 上的调制符号个数为 $2TP(x)$。[6]

为了简便，将映射函数 $MP(x)(v'):\{0,1\}^T\rightarrow A_x$ 记为 $M(v')$。

从上述定义可以看出，P(x)并没有规定一定是等概的。这里，将引入不等概映射的概念，也就是星座图上每一个星座点在调制的时候被使用的概率是不相等的。因为不等概映射的概念最早由 Gallager 提出，故通常被后人称为 Gallager 映射。

4.2.2 模型建立

只要合理设计每个星座点的使用概率，完全能够使发射的信号变量近似服从离散高斯分布，从而产生概率成形增益，使通讯系统能够更好逼近信道容量。通过相关文献的查阅，为了使信号在传输过程中出现的错误尽可能的少，应该在方案设计中按照以下设计优化准则：

（1） 准则 1: 信号星座图中的星座点使用概率要尽量逼近离散高斯分布。

（2） 准则 2: 对于同一组内的不同标号 (label)，设计的时候尽量最小化标号之间的最大汉明距离。

（3） 准则 3: 对于不同组内的不同标号，设计的时候同样尽量最小化标号之间的最大汉明距离[1]。

为了尽量达到传输中同一信噪比水平下误码率尽可能低，需要优化设计星座图以寻找最佳的星座形状并给出符号分配方案，也即确定保留星座上的哪些信号点决定了星座的形状，而在信号点上各分配多少符号决定了成形映射下各信号点的使用概率。

为了尽量减少信息熵，从减少星座图的信息符号的思路出发，首先将 16QAM 的信息符号中 16 个信息符号减少到 8 个，之后要解决的问题是 8 个信息符号在星座图的布局，以及 16 个信息符号对应 8 个星座图的映射关系的确定。[7]

题中给出的 8QAM 的星座图排列方案是在二维图中较优的一种设计，相邻各星座点之间距离相同，因此在此星座图排布的基础上进行进一步的优化设计。

4.2.3 解决方案

在问题分析的部分中已经提到设计准则，在确定映射关系的时候在星座图一个映射点对应多个信号时，要尽量最小化组内信号的汉明距离，这样在解码时由于噪声干扰，产生译码错误后，也不会有过多位数出错。

据此，得到新的解决方案，如图 11 所示：

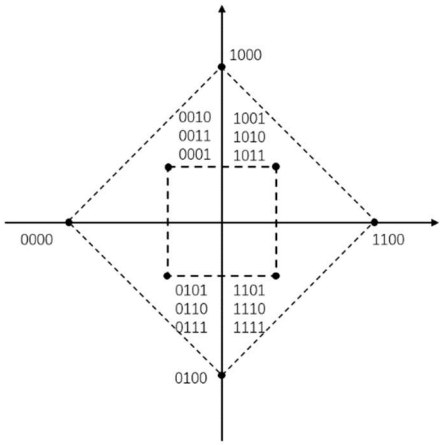

图 11 解决方案的星座图

图 12 是 Gallager 映射模型的信号调制与解调方案。

在发射端，系统包括 LDPC 码编码器，交织器以及 Gallager 映射器三部分。令 LDPC 码编码器的输入为一个二进制序列 $u=(u_0, u_1, ..., u_{n-1})$，其中 $u_i \in \{0, 1\}$，即 u 为信息序列，经过编码器，输出的码字序列记为 $c=(c_0, c_1, ..., c_{n-1})$。

编码后的码字序列 c 送入比特交织器进行交织操作，交织器产生的比特序列

[1]汉明距离（Hamming Distance）是指它表示两个（相同长度）字对应位不同的数量。

记为 v，$v = (v_0, v_1, ..., v_{n-1})$。序列 v 将被送入 Gallager 映射器进行比特序列到调制符号的对应转换。这里定义一个二维星座 $A_x = \{a_i \mid 0 \leqslant i \leqslant M-1, a_i \in c\}$，其中 M 为星座点的个数。Gallager 映射器把比特序列映射为星座符号序列。

假设映射器输出的信号符号序列为 $x = (x_0, x_1, ..., x_j, ...)$，其中 $x_j = M(v'_j)$，M( ) 代表 Gallager 映射器的映射规则，$v'_j = (v'_{j,1}, v'_{j,2}, ..., v'_{j,T})$ 代表从序列 v 中的 T 比特长度采样。显然，一个 v'_j 符号标号 (label) 对应星座图上的一个星座点 a_i。

接下来介绍系统模型的接收端，接收端包括解映射器 (解调器)、交织器、解交织器以及 LDPC 码译码器等部分。解映射器首先利用信道接收值，将其作为先验信息(a priori information) 计算每个比特的对数似然比 (log-likelihood ratio, LLR)，并将先验对数似然比记为 La。

解映射器首先计算每一个接收符号与星座点之间的欧氏距离，求出判决为某一个星座点符号的概率。之后，根据输入的先验信息 La，求出每一个信号符号的概率。然后，解映射器根据符号和比特之间的相应转化关系，利用最大后验概率 (MAP)算法计算出每一个比特的 LLR 值，即为解映射器的输出 LLR 序列。将该输出信息减去解映射器输入的先验信息 La，则得到解映射器输出的有用外信息序列，记为 Le。

映射器输出的外信息 Le 将在解交织后当作先验信息 La 输入 LDPC 码译码器，译码器迭代译码之后的输出 LLR 在减去输入的先验 La 后得到 LDPC 码译码器的输出外信息 Le，即 Le= LLR − La。

该外信息在经过交织器后再次送入解映射器，作为解映射器的先验信息。简而言之，解映射器和 LDPC 码译码器作为一对软输入软输出 (SISO) 的译码器进行联合迭代，直到满足设置的条件退出为止。

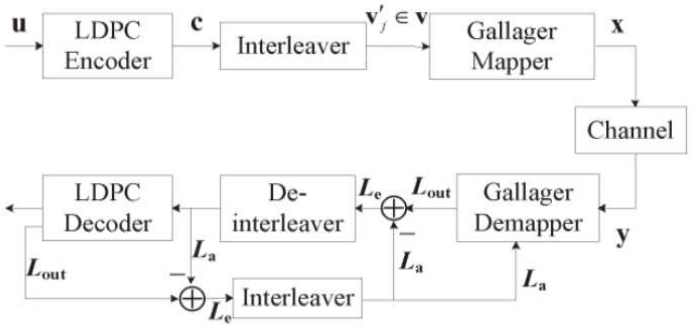

图 12 调制与解调方案

4.3 模型进一步优化的探讨

根据之前对于 16QAM 三维星座图优化方法的探讨，信息点在三维空间的排布更有利于增各个点之间的欧式距离，进而进一步降低误码率。在此设想的基础上，我们提出了三维空间的星座图映射优化方案的可能性，依旧采用星座图中的一个点对应多编码的映射形式，将 8 个点在立体空间进行排序，方案如图 13 所示。

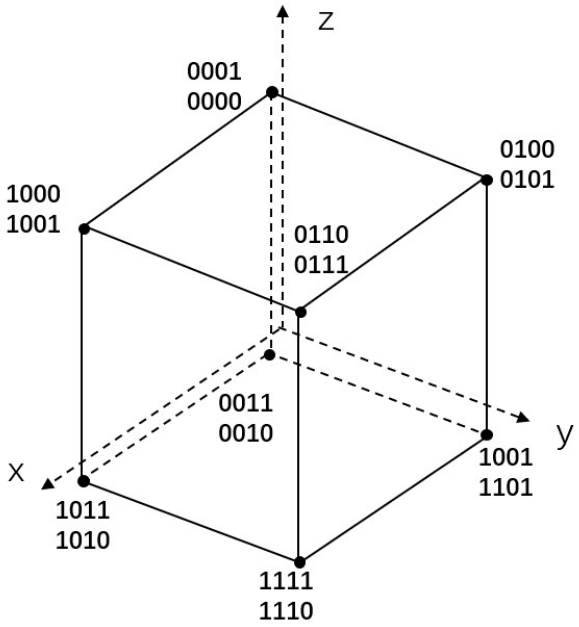

图 13 改进方案的空间星座图

在信号码分配时依旧遵照前文提到的设计准则 2 和设计准则 3，在同一个符号点的映射符号的汉明距离最小，相邻符号点间也尽量保证最小。

4.4 模型的评价与总结

本文采用的解决方案从几何构型和概率构型两种思路出发，采用 Gallager 映射方法，对 16QAM 传统方案进行优化，但是在模型的调制以及解调的过程中仍需要进一步考虑完善 Gallager 解映射的方案，进而达到比 8QAM 具有更低 SNR 容限点的目标。

5 参考文献

[1]杨英杰 赵小兰，光纤通信原理及应用，北京：电子工业出版社,257-262,2011

[2]DreamerZhang123，QAM 调制解调的原理，https://blog.csdn.net/zhangfuliang123/article/details/78486790，2018/09/19

[3]S. Cho, S.K. Park, Improved 16-ary constellation mapping for three-dimensional OFDM systems[J]. ELECTRONICS LETTERS, 2012

[4]张海英，一种三维极化幅度及其星座映射的研究[M],2015

[5]周林，LDPC 码高效编码调制研究[M],2011

[6]Dan Feng, Qi Li , Baoming Bai , Xiao Ma, Gallager Mapping Based Constellation Shaping for LDPC-Coded Modulation Systems[J], IEEE, 2015

[7]Lin Zhou, Weicheng Huang, Shengliang Peng, Yan Chen and Yucheng He, An Improved Design of Gallager Mapping for LDPC-coded BICM-ID System[J]. ELECTRONICS, 2016

6 附录

6.1 问题 2 子问题 1 代码

6.1.1 # 从文件中读取距离数据，转换成二维向量

```
def process_value():
    fr = open('../data/city_dist.txt', 'r')
    city_dist = dict()
    city_name = ['heb', 'bj&tj', 'wlmq', 'sh', 'zz', 'xa', 'wh', 'cq', 'cd', 'ls', 'gz&sz', 'km']
    city_num = len(city_name)
    city_city_array = []
    i = 0
    for line in fr.readlines():
        j = 0
        city_array = []
        for dist in line.strip('\n').split('     '):
            if i <= j:
                city_array.append(float(dist))
                city_dist['(' + str(i) + ',' + str(j) + ')'] = float(dist)
            else:
                city_array.append(float(0))
                city_dist['(' + str(i) + ',' + str(j) + ')'] = float(0)
            j += 1
        city_city_array.append(city_array)
        i += 1
    fr.close()
    return city_city_array
```

6.1.2 # 计算城市的带宽

```
def cal_city_value(city_city_array):
    # city_city_array = process_value()
    city_city_value_array = []
    for city_city in city_city_array:
        city_value_array = []
        for city in city_city:
            if (city <= 600) & (city != 0):
                city_value = 32
            elif (city >= 600) & (city <= 1200):
                city_value = 16
            elif (city >= 1200) & (city <= 3000):
                city_value = 8
            elif city >= 3000 or city == 0:
                city_value = 0
            city_value_array.append(city_value)
        city_city_value_array.append(city_value_array)
```

```
        return city_city_value_array
```

6.1.3 # 计算城市的人口系数

```
    def cal_population():
        population = [1064, 3735, 268, 2420, 972, 883, 1071, 3048, 1717, 55, 2595, 673]
        popu_popu_array = []
        i = j = 0
        for popu_i in population:
            i += 1
            popu_array = []
            for popu_j in population:
                j += 1
                if i < j:
                    popu_array.append(math.sqrt(popu_i*popu_j))
                else:
                    popu_array.append(0.0)
            j = 0
            popu_popu_array.append(popu_array)
        return popu_popu_array
```

6.1.4 # 计算城市人口系数*人口容量

```
    def array_mul_array(array1, array2):
        array_array_result = []
        for i in range(len(array1)):
            array_result = []
            for j in range(len(array1[i])):
                array_result.append(array1[i][j]*array2[i][j])
            array_array_result.append(array_result)
        return array_array_result
```

6.1.5 # 按照系数大小排序

```
    def value_sort(city_mul_popu_value):
        city_value = dict()
        i = 0
        for city_mul_popus in city_mul_popu_value:
            # print line
            j = 0
            for city_mul_popu in city_mul_popus:
                print(city_mul_popu),
                # if i <= j:
                city_value[str(i)+','+str(j)] = float(city_mul_popu)
                j += 1
            i += 1
            print
        city_value_sort = collections.OrderedDict(sorted(city_value.items(), key=lambda x: x[1],
reverse=True))
        for k, v in city_value_sort.items():
```

```
            print(k, v)
        return city_value_sort
```

6.1.6 # 最小生成树改进算法选择分数较高的城市

```
    def select_top_city(city_value_sort, city_num):
        result_dict = dict()
        city_label = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
        city_label_dict = dict()
        for city in city_label:
            city_label_dict[city] = 0
        count = 0
        for k, v in city_value_sort.items():
            count += 1
            if count > city_num:
                break
            city_1 = int(k.split(',')[0])
            city_2 = int(k.split(',')[1])
            city_label_dict[city_1] = city_label_dict[city_1] + 1
            city_label_dict[city_2] = city_label_dict[city_2] + 1
            if city_1 in city_label:
                city_label.remove(city_1)
            if city_2 in city_label:
                city_label.remove(city_2)
            print(str(city_1)+','+str(city_2))
            result_dict[k] = v
            city_value_sort.pop(k)
        # city_value_sort 表示剩余元素
        # result_dict 表示最后要的元素
        result_dict_sort = collections.OrderedDict(sorted(result_dict.items(), key=lambda x: x[1],
reverse=True))
        residual = city_num - len(city_label)
        count = residual
        for i in range(residual):
            if count > city_num or not city_label:
                break
            for k, v in city_value_sort.items():
                city_1 = int(k.split(',')[0])
                city_2 = int(k.split(',')[1])
                # 如果找到了含有当前 city_label 剩余元素的元素
                if city_1 in city_label or city_2 in city_label:
                    if city_1 in city_label:
                        city_label_dict[city_1] = city_label_dict[city_1] + 1
                        city_label.remove(city_1)
                    if city_2 in city_label:
```

```
                    city_label_dict[city_2] = city_label_dict[city_2] + 1
                    city_label.remove(city_2)
                print(str(city_1) + ',' + str(city_2))
                # 删除元素部分
                # 首先把之前排好序的字典倒排序，系数小的在前，系数大的在后
                selected_dict_sort = collections.OrderedDict(sorted(result_dict.items(), key=lambda
x: x[1], reverse=False))
                # selected_dict_sort 表示已经排好的元素
                # 去已经排好的字典中查找元素
                for k1, v1 in selected_dict_sort.items():
                    city_1 = int(k1.split(',')[0])
                    city_2 = int(k1.split(',')[1])
                    # 如果在数量都大于一次，那么把它剔除
                    if city_label_dict[city_1] > 1 and city_label_dict[city_2] > 1:
                        selected_dict_sort.pop(k1)
                        city_label_dict[city_1] = city_label_dict[city_1] - 1
                        city_label_dict[city_2] = city_label_dict[city_2] - 1
                        print(k1+"已经删除！")
                        break
                        # 相应的地方也要剔除
                        city_value_sort.pop(k)
                # 新元素加入其中
                # result_dict_sort.popitem()  # 去掉最后一个元素
                result_dict = selected_dict_sort
                result_dict[k] = v
                print(k+"已经添加！")
            count += 1
        result_dict_sort = collections.OrderedDict(sorted(result_dict.items(), key=lambda x: x[1],
reverse=True))
        print("最终筛选结果为：")
        for k, v in result_dict_sort.items():
            print(k, v)
    return result_dict_sort
```

6.2 问题 2 子问题 2 代码

6.2.1

```
# 字典中的元素放入到数组里面
def dict_to_array(result_dict, city_mul_popu_value):
    # 创建一个大小一样的空数组
    city_mul_popu_value_new = []
    city_mul_popu_value_key = []
    city_mul_popu_value_isnull = []
    for city_mul_popus in city_mul_popu_value:
        city_mul_popu_value_new_sub = []
```

```
            city_mul_popu_value_key_sub = []
            city_mul_popu_value_isnull_sub = []
            for city_mul_popu in city_mul_popus:
                city_mul_popu_value_new_sub.append(0)
                city_mul_popu_value_key_sub.append(0)
                city_mul_popu_value_isnull_sub.append(0)
            city_mul_popu_value_new.append(city_mul_popu_value_new_sub)
            city_mul_popu_value_key.append(city_mul_popu_value_key_sub)
            city_mul_popu_value_isnull.append(city_mul_popu_value_isnull_sub)
        for k, v in result_dict.items():
            print(k, v)
            city_1 = int(k.split(',')[0])
            city_2 = int(k.split(',')[1])
            city_mul_popu_value_new[city_1][city_2] = v
            city_mul_popu_value_key[city_1][city_2] = k
            city_mul_popu_value_isnull[city_1][city_2] = 1
        return city_mul_popu_value_key, city_mul_popu_value_new, city_mul_popu_value_isnull
```

6.2.2 # 三角矩阵转换成邻接矩阵

```
def triangle2rectangle(city_mul_popu_value_new):
    city_mul_popu_value_rec = []
    i = 0
    for city_mul_popu_value in city_mul_popu_value_new:
        i += 1
        city_mul_popu_value_rec_sub = []
        for count in range(i):
            city_mul_popu_value_rec_sub.append(0)
        for city_mul_popu in city_mul_popu_value:
            city_mul_popu_value_rec_sub.append(city_mul_popu)
        city_mul_popu_value_rec.append(city_mul_popu_value_rec_sub)
    '''city_mul_popu_value_rec_sub = []
    for count in range(i):
        city_mul_popu_value_rec_sub.append(0)
    city_mul_popu_value_rec.append(city_mul_popu_value_rec_sub)'''
    return city_mul_popu_value_rec
```

6.2.3 # 转换成带权值的邻接矩阵

```
def to_adjacency_matrix(city_mul_popu_isnull_rec):
    city_mul_popu_isnull_adg = []
    for city_mul_popu_isnull_rec_sub in city_mul_popu_isnull_rec:
        city_mul_popu_isnull_adg_sub = []
        for items in city_mul_popu_isnull_rec_sub:
            if items == 0:
                items = 999999
```

```
                    city_mul_popu_isnull_adg_sub.append(items)
                city_mul_popu_isnull_adg.append(city_mul_popu_isnull_adg_sub)
            return city_mul_popu_isnull_adg
```

6.2.4 # 转换成带权值的对称邻接矩阵

```
    def to_adjacency_symmetry_matrix(city_mul_popu_isnull_rec):
        symmetry_array = []
        for city_mul_popu_isnull_rec_sub in city_mul_popu_isnull_rec:
            symmetry_array_sub = []
            for items in city_mul_popu_isnull_rec_sub:
                symmetry_array_sub.append(0)
            symmetry_array.append(symmetry_array_sub)
        i = 0
        for city_mul_popu_isnull_rec_sub in city_mul_popu_isnull_rec:
            j = 0
            for items in city_mul_popu_isnull_rec_sub:
                if i == j:
                    symmetry_array[i][j] = 0
                elif i > j:
                    symmetry_array[i][j] = city_mul_popu_isnull_rec[j][i]
                else:
                    symmetry_array[i][j] = city_mul_popu_isnull_rec[i][j]
                j += 1
            i += 1
        return symmetry_array
```

6.2.5 # 计算每个点的直接连接数目

```
    def cal_connection_count(symmetry_array):
        city_label = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
        city_connect = dict()
        city_label_count = 0
        for city_sub in symmetry_array:
            count = 0
            for city in city_sub:
                if city == 1:
                    count += 1
            city_connect[city_label[city_label_count]] = count
            city_label_count += 1
        city_connect_sort = collections.OrderedDict(sorted(city_connect.items(), key=lambda x: x[1],
reverse=True))
        for k, v in city_connect_sort.items():
            print(k, v)
        return city_connect_sort
```

6.2.6 # 路径系数表

```
def cal_coefficient(city_connect_sort, city_value_array, symmetry_array, max_length):
    # 生成一维最大长度为 max_length 的数组
    max_length_array = []
    for i in range(max_length):
        max_length_array.append(0)
    # path_array, path_result = dijkstra_all_minpath(key, symmetry_array)
    # 定义一个系数空数组
    connect_coefficient = []
    for city_value_array_sub in city_value_array:
        connect_coefficient_sub = []
        for items in city_value_array_sub:
            connect_coefficient_sub.append([])
        connect_coefficient.append(connect_coefficient_sub)
    print(connect_coefficient)
    update_symmetry_array = symmetry_array
    # 逐一筛选，选出元素并填充到数组
    for iterator in range(12):
        key = city_connect_sort.popitem()
        path_array, path_result = dijkstra_all_minpath(key[0], symmetry_array)
        for path in path_result:
            print(path)
            length = len(path)-1
            count = 0
            for path_items in path:
                if count < length:
                    print(path_items)
                    connect_coefficient[int(path[count])][int(path[count+1])].append(int(1))
                count += 1
        update_symmetry_array = update_city_value(update_symmetry_array, key[0])
        print(str(key[0]) + "已经被计算，图被更新！")
        print(path_result)
    print(connect_coefficient)
    return connect_coefficient
```

6.2.7 # 路径最大长度查询

```
def get_max_length(city_connect_sort, symmetry_array):
    update_symmetry_array = symmetry_array
    max_length = 0
    for i in range(12):
        key = city_connect_sort.popitem()
        path_array, path_result = dijkstra_all_minpath(key[0], update_symmetry_array)
        for path in path_result:
            if max_length < len(path):
```

```
                    max_length = len(path)
            update_symmetry_array = update_city_value(update_symmetry_array, key[0])
            print(str(key[0]) + "已经被计算，图被更新！")
            print(path_result)
        return max_length


    # 更新一个节点的系数
    def update_city_value(symmetry_array, key):
        i = 0
        for symmetry_array_sub in symmetry_array:
            j = 0
            for items in symmetry_array_sub:
                if i == key or j == key:
                    symmetry_array[i][j] = 999999
                j += 1
            i += 1
    return symmetry_array
```

6.2.8

```
    # 计算最短路径
    def dijkstra_all_minpath(start, matrix):
        length = len(matrix)  # 该图的节点数
        path_array = []
        temp_array = []
        path_array.extend(matrix[start])
        temp_array.extend(matrix[start])
        temp_array[start] = inf  # 临时数组会把处理过的节点的值变成 inf，表示不是最小权值的节点
了
        already_traversal = [start]  # start 已处理
        path_parent = [start] * length  # 用于画路径，记录此路径中该节点的父节点
        path_result = []
        while len(already_traversal) < length:
            i = temp_array.index(min(temp_array))  # 找最小权值的节点的坐标
            temp_array[i] = inf
            path = [str(i)]  # 用于画路径
            k = i
            while path_parent[k] != start:  # 找该节点的父节点添加到 path，直到父节点是 start
                path.append(str(path_parent[k]))
                k = path_parent[k]
            path.append(str(start))
            path.reverse()  # path 反序产生路径
            print(str(i) + ':', '->'.join(path))  # 打印路径
            if path not in path_result:
```

```
                path_result.append(path)
            already_traversal.append(i)  # 该索引已经处理了
            for j in range(length):
                if j not in already_traversal:
                    if (path_array[i] + matrix[i][j]) < path_array[j]:
                        path_array[j] = temp_array[j] = path_array[i] + matrix[i][j]
                        path_parent[j] = i  # 说明父节点是 i
return path_array, path_result
```