基于 OpenCV 建图和最短路算法选择攀岩路线

摘要

室内攀岩是通过攀登人工设计高度、难度不等的岩壁（通常 6—8 米高）在上面装有许多大小不一的岩石点，供人用四肢借助岩点的位置，手攀脚登，来完成攀岩的体验。室内岩壁上布满可以随意改变位置的岩点攀岩者的路线、难易指数，完全可以由改变岩点的位置来进行人为的掌控。攀爬者需要做出特定攀爬动作才能不断向上进发。本文通过建立数学模型，研究分析攀爬最优路线的数学方法，并给出对所选路线好坏的评价方法。

针对问题一，由于需要设计一条合理的攀爬路线，因此问题转化为图论问题，对此，需要将不同颜色的岩石均转为图论问题中的节点，本文使用计算机视觉开源库 OpenCV 对岩石轮廓进行识别，本文使用灰度法，将彩色图像中的三分量的亮度作为三个灰度图像的灰度值之一，得到岩石平面分布图的灰度图。再使用 Canny 边缘检测算法，检测灰度图的边缘。由于岩石轮廓是一个形状不规则的几何体，为简化模型，将每个轮廓均用其质心表示，图论中的节点，使得将问题专注于攀爬路线的设计，而非考虑岩石形状的力学分析。通过查阅相关资料，攀爬时每次向上的高度最好不超过脚底到膝盖的距离，则会难以攀爬并保持平衡，分别确定了以100,90,80,70为每次上升的界限，即当节点之间的欧几里得距离只有小于该界限，才进行连边，由此的到攀爬高度不同时的节点网络图。在使用 Dijkstra 算法求解该模型。将图中边数用$|E|$表示，顶点数用$|V|$表示，对于没有任何优化的戴克斯特拉算法，算法复杂度为$O(|V|^2+|E|)$。可判断该岩石分布图为稀疏图，故使用邻接表来提高节点访问效率并节省内存。

针对问题二，与问题一不同的点在于，问题一的权值是在边上的，而问题二的权值是在节点上的，即使用一次黄色石头减 1 分，使用一次蓝色石头减 2 分等，利用点权建图的难处在于，传统算法如 Dijkstra 并不适用于点权图，并且用多种数据结构维护同一个点集合，导致算法效率低。可将单独一个节点拆成两个节点，即原节点的出点和入点，那么原节点的出点和入点的连边即为节点的权值，至此，即可将点权转化为边权。综上分析，首先将图中所有节点均拆为 2 个节点即入点和出点，那么图中节点的数量就要增大 2 倍，而后再从入点向出点连接一条权值为原节点颜色权值的边，为了保证图的连通性，将在的图上进行，并把原距离权值赋为 0，这里仍然选择编号 1 作为起点，而编号为 111，112，113，114 则作为终点，最后通过题意转化，使得只需再使用 Dijkstra 算法求取最短路，则可得到最高分路径。

针对问题三，本文引用 YDS 作为评价攀岩难度的模型系统，通过查阅相关资料，本文将选取路径总长度以及岩石之间的最大距离差，所攀爬岩石的个数作为评价指标，以评价该模型，并采用 YDS 系统的评分方式。通过 Dijkstra 算法选择的最多得分路径，且与最短路径基本吻合，在 Kmeans 聚类分析分为了两类的情况下，可将这 4 条路线分为 2 条评价等级为优的路线和 2 条评价等级为良的路线。

关键词：OpenCV 轮廓识别　Dijkstra 最短路算法　点权图　Kmeans 聚类

一、问题重述

2.1 问题背景

室内攀岩是一种室内体育活动，参与者在人工设计的高度不等、难度各异的岩壁上攀登，岩壁通常高度在 6 至 8 米之间。这些岩壁上布满了各种大小的岩石点，攀岩者可以借助这些岩点的位置，通过手部和脚部的动作来完成攀岩过程，达到锻炼和体验的目的。在室内岩壁上，岩点的位置可以根据需要进行改变，从而调整攀登者的路线和难度，实现人为的掌控。

对于初次接触室内攀岩的人来说，往往会面临一些挑战。攀岩墙上的众多岩点可能会让人感到眼花缭乱，有些人可能会认为攀岩只需随意抓握岩点，然后就能向上爬行。然而，实际情况是这些岩点是由专业的定线员精心设计的攀爬线路，攀爬者需要采取特定的攀爬动作，才能够不断向上攀升。

2.2 问题重述

第一个问题涉及到一个场地的建立，高度为 10 米，攀爬路线从下边缘开始触碰到上边缘完成。需要基于题目提供的岩点分布平面图建立一个模型，计算出一条合理的攀爬路线。

第二个问题涉及到比赛的计分制度。在这个规则下，每使用一次黄色石头减 1 分，蓝色石头减 2 分，红色石头减 3 分，绿色石头减 4 分。需要规划一条路线，以最大限度地减少总分。

第三个问题则是基于前面的模型，提出了评价所选路线难度系数的方法。

综合考虑以上问题，需要解决场地建模、路线规划以及评价方法等方面的挑战。通过分析岩点分布和计分制度，设计出合适的攀爬路线，以及一种能够客观衡量路线难度的评价方法。

二、 模型假设

本文提出以下合理假设：

- 假设攀爬者身高正常，符合正常人的身体水平
- 假设攀爬者可以根据路线图正常到达节点
- 假设攀爬者在攀爬过程中，可以自由选择攀爬的方向
- 假设岩点的分布密度和位置会影响路线的难度，密集的分布和复杂的位置可能使路线更具挑战性

三、 符号说明

符号	意义
$Gray$	灰度值
R，G，B	红，绿，蓝的数值(0~255)
c	几何体的质心
M_{ji}	图像几何矩
$\|E\|$	边数
$\|V\|$	顶点数
$P(x,y)$	图像上坐标为(x,y)上的灰度值
C_x	质心的x坐标
C_y	质心的y坐标
$dist[i]$	从编号为i的点到源点的权值和
dk_Q	完成键的降序排列时间
em_Q	从优先队列中提取最小键值的时间
YDS	优胜美地十进制系统

四、问题分析

4.1 问题一分析

针对问题 1，在一个高度为 10 米的垂直岩壁上，根据题目给出的岩点分布平面图，建立一个模型来计算出一条合理的攀爬路线。这个问题涉及到路线规划，需要考虑如何从底部开始，通过选择合适的岩点以及移动方向，到达岩壁的顶部。解决这个问题需要考虑岩点的分布、距离、角度和支撑情况，以及攀爬者的身体能力，以便设计出能够克服挑战并完成攀爬的有效路线。

4.2 问题二分析

问题 2 引入了一个比赛计分制度，其中不同颜色的石头使用会导致不同的减分。在这种情况下，需要规划一条攀爬路线，以最大限度地减少总分。这个问题涉及到在限制条件下做出最优决策，即选择使用哪些石头以及在何处使用，以便达到尽可能低的总分。解决这个问题需要综合考虑石头的颜色、位置、分数减少规则，不同石头的位置会影响攀爬者的选择，以及在路线规划中如何合理使用它们，以及如何将这些因素结合到一个高效的路线中。

4.3 问题三分析

问题 3 要求基于之前建立的模型，给出所选路线难度系数的评价方法。这个问题涉及到如何量化路线的难度，以便能够客观地对不同路线进行比较。解决这个问题需要考虑岩点的分布，路线上岩点的密度越高，可能需要更多的移动和支撑动作，增加难度、距离、角度，岩点之间的距离和角度变化越大，攀爬者可能需要更多的技巧和平衡来完成路线。，以及攀爬者在路线上需要进行的动作次数等因素，以便设计出一个能够准确反映难度的评价指标。

综合上述分析，这三个问题涉及了攀爬路线的规划、最优决策和难度评价等多个方面的考虑，需要综合考虑不同因素来解决。

五、问题一的模型建立与求解

5.1 图像预处理

5.1.1 岩石轮廓识别

对于附件岩石平面分布图，由于需要设计一条合理的攀爬路线，因此问题转化为图论问题，对此，需要将不同颜色的岩石均转为图论问题中的节点，本文使用计算机视觉开源库 OpenCV 对岩石轮廓进行识别，以下是数字图像处理步骤。

表 1.1 OpenCV 轮廓识别步骤

（1）	使用 imread 读取图像并转为灰度图像
（2）	使用 Canny 边缘检测找到图像中的边缘
（3）	使用 findContours 查找边缘并遍历每个轮廓
（4）	使用 contourArea 计算轮廓面积，忽略面积过小的轮廓
（5）	使用 drawContours 在图像上绘制轮廓并编号

其中，图像灰度化的目的是为了简化矩阵，提高运算速度。彩色图像中的每个像素颜色由 R、G、B 三个分量来决定，而每个分量的取值范围都在 0-255 之间，对计算机而言，彩色图像的一个像素点就会有 256*256*256=16777216 种颜色的变化范围，而灰度图像是 R、G、B 分量相同的一种特殊彩色图像。

对计算机来说，一个像素点的变化范围只有 0-255 这 256 种。彩色图片的信息含量过大，而进行图片识别时，只需要使用灰度图像里的信息就已经足够。图像灰度化处理主要有以下几种方式：

（1） 分量法

将彩色图像中的三分量的亮度作为三个灰度图像的灰度值，可根据应用需要选取一种灰度图像，同时也是 OpenCV 库的默认方法

$$\begin{cases} Gray_1(i,j) = R(i,j) \\ Gray_2(i,j) = G(i,j) \\ Gray_3(i,j) = B(i,j) \end{cases} \tag{1-1}$$

（2） 最大值法

将彩色图像中的三分量亮度的最大值作为灰度图的灰度值

$$Gray(i,j) = \max R(i,j), G(i,j). B(i,j) \tag{1-2}$$

（3） 平均值法

将彩色图像中的三分量亮度求平均得到一个灰度值

$$Gray(i,j) = \frac{[R(i,j) + G(i,j) + B(i,j)]}{3} \tag{1-3}$$

（4） 加权平均法

由于人眼对绿色的敏感最高，对蓝色敏感最低，因此，按下式对 RGB 三分量进行加权平均能得到较合理的灰度图像

$$Gray(i,j) = 0.299 * R(i,j) + 0.578 * G(i,j) + 0.114 * B(i,j) \tag{1-4}$$

本文使用（1）分量法，将彩色图像中的三分量的亮度作为三个灰度图像的灰度值之一，得到下图 1.1 岩石平面分布图的灰度图。

图 1.1 岩石平面分布图的灰度图

得到灰度图后，需要检测灰度图的边缘，使用 Canny 边缘检测算法，它具有低错误率，检测出的边缘是真正的边缘；良好的定位，检测出的边缘像素点与真正边缘的像素点距离近；对噪声不敏感，噪声不应该标注为边缘。Canny 边缘检测算法有四个步骤：

（1） 降低对噪声的影响，对图像做高斯滤波或中值滤波，过滤噪声。

（2） 使用 Sobel 算子对图像的像素点求梯度大小和方向，以下为 Sobel 算子。

$$dx = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$dy = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

（3） 使用非极大值抑制算法在一组边缘中选取最好的边缘，具体做法是检查每个像素点与附近梯度方向一致的像素点，当前像素点梯度最大，则保留，否则去除。

（4） 使用双阈值（小阈值, 大阈值）确定最终的边缘，像素点梯度高于大的阈值，则保留；像素点低于小的阈值，则忽略；介于两个阈值之间，判断像素点与边缘像素点是否相连。

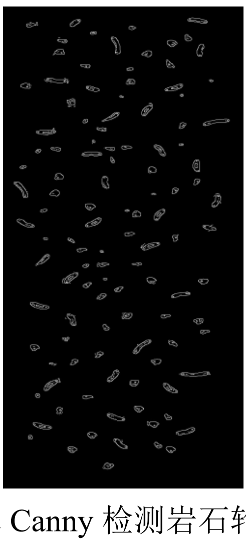

图 1.2 Canny 检测岩石轮廓图

5.1.2 岩石质心表示

由于岩石轮廓是一个形状不规则的几何体，为简化模型，避免过多的几何讨论，将每个轮廓均用其质心表示，将其当作物理学中的质点，图论中的节点，使得将问题专注于攀爬路线的设计，而非考虑岩石形状的力学分析。

几何体的质心是形状中所有点的算术平均值，假设一个形状由以下部分组成n个不同点$x_1, x_2 \ldots x_n$，则质心由下式给出：

$$c = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad (1-5)$$

其中c是几何体的质心，x_i是点在空间中的坐标，在图像处理和计算机视觉的背景下，每个几何体都是由像素组成的，质心只是构成形状的所有像素的加权平均值。

在 OpenCV 中，进行图像操作，使用图像矩找到 blob(机器视觉中指图像中具有相似颜色，纹理等特征所组成的一块连通区域)的中心。图像矩是图像像素值的加权平均值，从而找到图像的一些特定属性，如半径，面积，质心等。为了找到图像的质心，将其二值化然后找到它的质心。质心由下式给出：

$$C_x = \frac{M_{10}}{M_{00}} \qquad (1-6)$$

$$C_y = \frac{M_{01}}{M_{00}} \qquad (1-7)$$

其中C_x是质心的x坐标，C_y是质心的y坐标，M表示图像几何矩，$P(x,y)$表示图像上坐标为(x,y)上的灰度值，几何矩计算由下式给出：

$$M_{ji} = \sum_{x,y}\left(P(x,y)\cdot x^j \cdot y^i\right) \qquad (1-8)$$

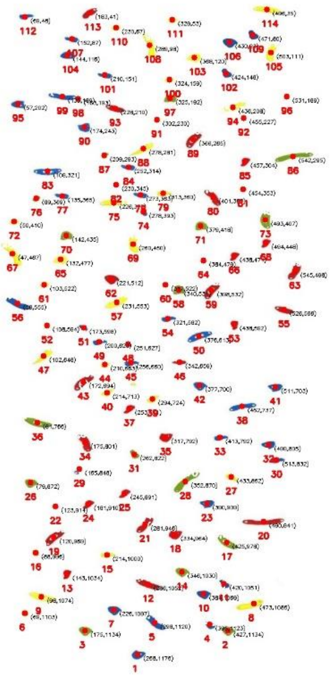

图 1.3 轮廓的质心坐标图及编号

5.1.3 轮廓识别优化算法

当一些岩石轮廓未被正确标出时，可能是由于边缘检测参数的设置不合适，或者轮廓的形状比较复杂，不易被简单的边缘检测方法捕捉到。为了更好地标出岩石轮廓，可采用以下优化步骤：

（1） 调整 Canny 边缘检测参数：Canny 边缘检测的两个阈值参数可以影响边缘检测的结果。尝试调整这两个阈值的值，以获得更好的边缘图像。

（2） 使用更高级的轮廓近似方法： 轮廓近似方法可以帮助更好地捕捉轮廓的形状，特别是当轮廓较复杂时。可以尝试 approxPolyDP 来近似轮廓。

（3） 使用颜色分割： 如果岩石的颜色非常明显，可以使用颜色分割方法，例如阈值化，以便更好地分离岩石和背景。这需要对颜色空间进行一些实验，以找到最适合的颜色通道和阈值。

（4） 形态学操作： 在边缘检测后，使用形态学操作（腐蚀和膨胀）可以消除一些小的孔洞或噪声，使得轮廓更连续。

（5） 调整轮廓面积过滤阈值： 可能需要调整轮廓面积的过滤阈值，以保留更多或更少的轮廓。

5.1.4 节点连线

通过查阅相关资料，攀爬时每次向上的高度最好不超过脚底到膝盖的距离，则会难以攀爬并保持平衡，由于图中整个场地高 10m，经过按比例缩放，分别确定了以100,90,80,70为每次上升的界限，即当节点之间的欧几里得距离只有小于该界限，才进行连边，也就是物理上的可直接从该点通过这条边跨越到另外一个点

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \qquad (1-9)$$

其中假设两点分别为$point_1, point_2$，其坐标分别为$(x_1, y_1), (x_2, y_2)$，那么其距离可由$(1-9)$式计算。

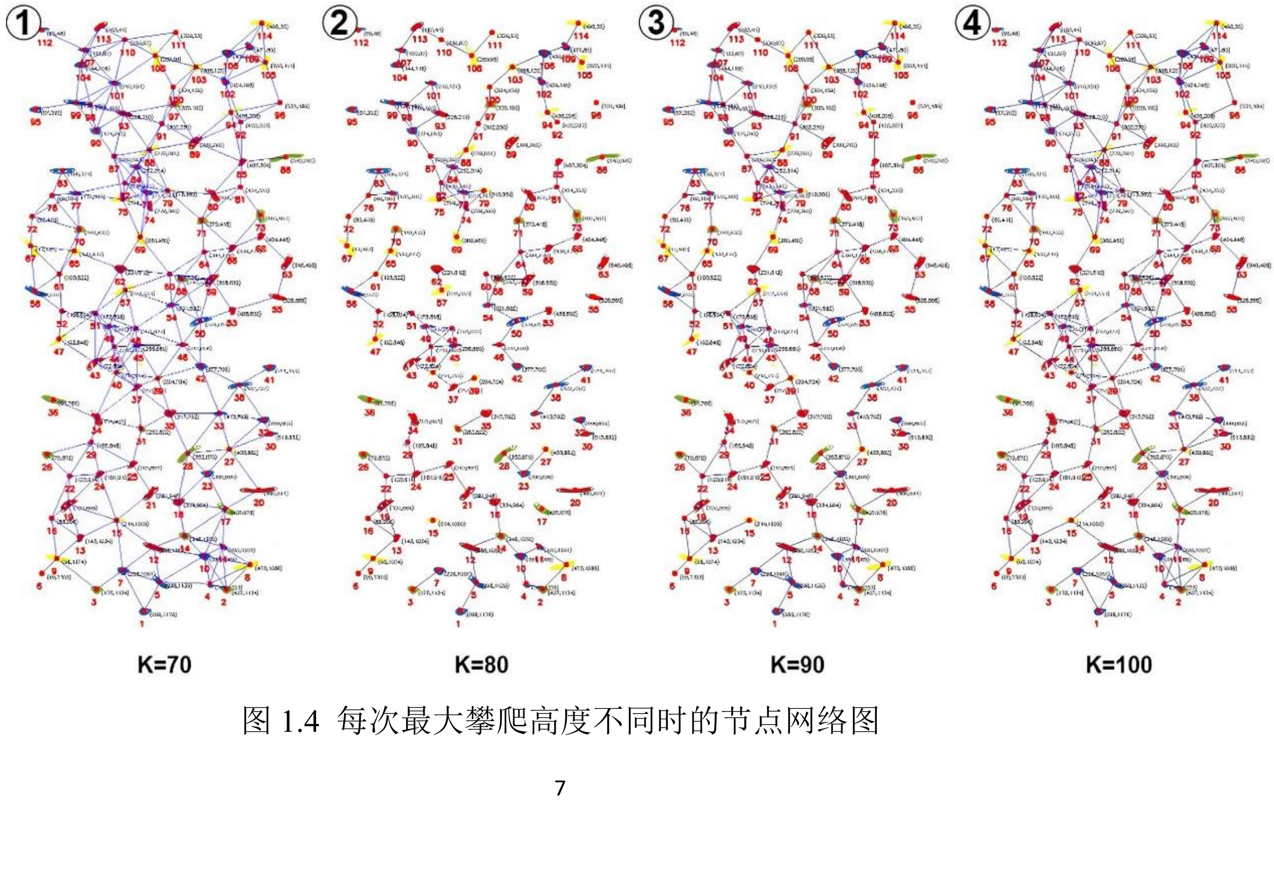

图 1.4 每次最大攀爬高度不同时的节点网络图

5.2 图论模型求解

5.2.1 设定权值

在图论问题中，需要定量地去衡量从出点到入点的价值，这条边的值称为权值，本题中，由于只需要计算一条合理的攀爬路线，故选择路径长度最小的路线，更能节省攀爬者的体力，因此，将权值设置为两点之间的路径长度，可通过式$(1-9)$进行计算。

5.2.2 设置源点终点

从图 1.4 中可以看出，编号为 1 的点为最低点，本文中选取 1 号点作为攀爬者的源点，而处于顶端的位置的共有 4 个点，分别为 111，112，113，114 号点，故分别选取这 4 个点作为终点，由于攀爬者的体型并未规定，故本文分别选取 70，80，90，100 作为攀爬者每次上升的最大高度，并以此为参数建立对应的网络图。

5.2.3 最短路模型求解

本文采用 Dijkstra 算法求解该模型，Dijkstra 算法基于贪心的思想，通过保留目前为止所找到的每个顶点$v \in V$从s到t的最短路径来运行，初始时，原点s的路径权重被赋为 0（即原点到原点的距离为 0），同时把所有其他顶点的路径长度设为无穷大，即表示不知道任何通向这些顶点的路径，当算法结束$dist[v]$中存储的便是从s到t的最短路径，如果路径不存在，则为$dist[v]=inf$。

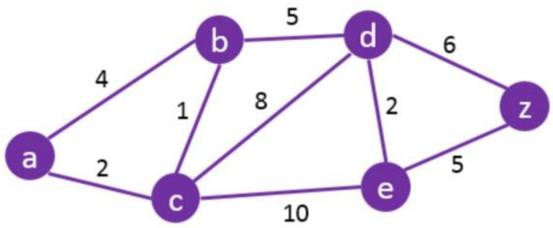

图 1.5 Dijkstra 算法示意图

松弛操作是 Dijkstra 算法的基础操作，如果存在一条从u到v的边，那么从s到v的一条新路径时将边$weight(u,v) \in E$添加到从s到u的路径尾部来拓展一条从s到v的路径，这条路径的长度是$dist[u]+weight(u,v)$，如果这个值比目前已知的$d[v]$的值都要小，那么可以用这个值来替代当前$dist[v]$的值，松弛边的操作一直执行到所有的$dist[v]$都代表从s到v的最短路径的长度值。

其伪代码为：

Function Dijkstra(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)//将原点以外的顶点的$dist[v]$置为无穷大

$dist[s]=0$//将原点到原点的距离设置为 0

$S \leftarrow queue$//Q是顶点V的一个优先队列

$Q \leftarrow s$//以顶点的最短路径估计排序

While($Q \in queue$)

  do $u \leftarrow EXTRACT-MIN(Q)$//选取$u$为$Q$中最短路径估计最小顶点

  $S \leftarrow S \cup u$

  for each vertex $v \in Adj[u]$

    do RELAX(u,v,w)//松弛成功的节点会被加入到队列中

5.2.4 时间复杂度分析与优化

将图中边数用$|E|$表示，顶点数用$|V|$表示，对于任何基于顶点集Q的实现，算法的运行时间是$O\left(|E| \cdot dk_Q + |V| \cdot e_{m_Q}\right)$，其中$dk_Q$和$em_Q$分别表示完成键的降序排列时间和从$Q$中提取最小键值的时间。

对于没有任何优化的戴克斯特拉算法，实际上等价于每次遍历了整个图的所有结点来找到Q中满足条件的元素（即寻找最小的顶点是$O(|V|)$的，此外实际上还需要遍历所有的边一遍，因此算法复杂度为$O(|V|^2 + |E|)$

此外，对于边数$|E|$，如果少于$|V|^2$，则称该图为稀疏图，那么可用邻接表对图进行存储，不仅节省空间，而且能更快地访问节点元素；反之则称为邻接矩阵。

对图 1.4 分别统计，可得到下表，由表中，可判断该岩石分布图为稀疏图，故使用邻接表来提高节点访问效率并节省内存。

表 1.2 不同攀爬最大高度时的点边数统计结果

每次攀爬的最大高度	顶点数	边数
70	114	113
80	114	160
90	114	225
100	114	280

5.2.5 模型求解

综上分析，首先根据 OpenCV 识别出岩石轮廓的中心点集合，并根据每次向上攀爬的最大高度，划分出 4 种不同的网络图，通过权值设置得到了边的集合，而后用邻接表建图，并分析了起点终点归属，再代入 Dijkstra 算法进行求解，最终得了$dist$数组，即起点s距离t的距离为$dist[t]$，得到以下表格。

表 1.3 问题一结果图

每次攀爬最大高度/编号	111	112	113	114
70	inf	inf	inf	inf
80	1337.43	inf	1351.51	1427.8
90	1263.63	inf	1274.58	1327.61
100	1205.84	1254.05	1216.8	1277.48

可以发现，当攀爬高度为 70 时，均无法到达终点，并且选择攀爬终点为 111 号点的距离始终小于其他点，故这里选择以 111 号点为终点，每次攀爬最大高度设为 100，则合理的攀爬路线为：

$$1 \to 7 \to 15 \to 24 \to 29 \to 34 \to 40 \to 49 \to 57 \to 62 \to 69 \to 78 \to 88 \to 91 \to 100 \to 108 \to 111$$

其完整路线图在岩石轮廓图如下图表示

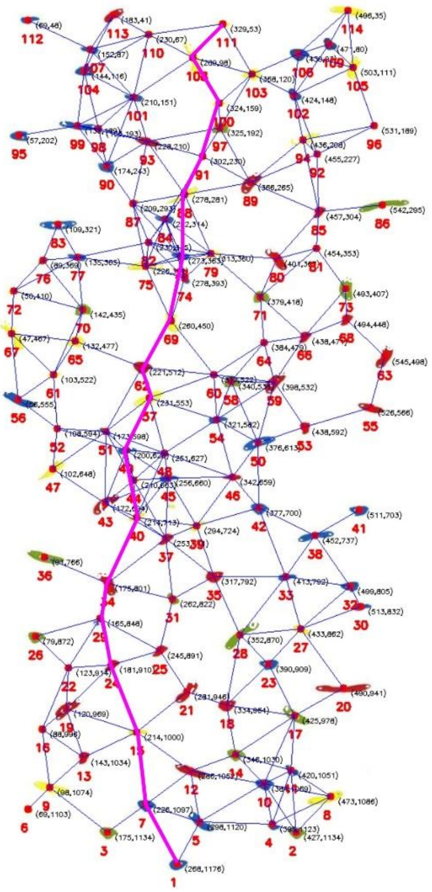

图 1.6 当每次跨越高度最大为 100 的最短路径(粉色)

六、问题二的模型建立与求解

6.1 图论模型建立

6.1.1 拆点

问题二与问题一不同的点在于，问题一的权值是在边上的，而问题二的权值是在节点上的，即使用一次黄色石头减 1 分，使用一次蓝色石头减 2 分等，利用点权建图的难处在于，传统算法如 Dijkstra 并不适用于点权图，并且用多种数据结构维护同一个点集合，会导致代码过于臃肿，导致算法效率低。

对此，可将单独一个节点拆成两个节点，即原节点的出点和入点，那么原节点的出点和入点的连边即为节点的权值，至此，即可将点权转化为边权，示意图如下：

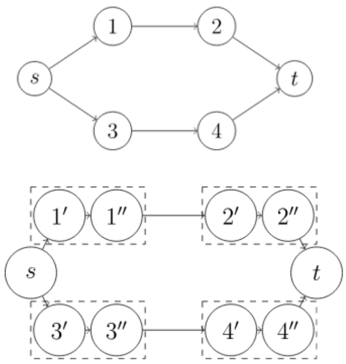

图 2.1 拆点示意图

6.1.2 题意转化

对于该节点与其他节点的连边，仍然采用图 1.4 中的$K = 100$的连线图，以保证图的大部分是连通的，并将该节点的出点与其他节点的入点相连，并设置该边的权值为 0，以避免距离权值会影响到问题二中的颜色权值。

由于题目规定是采取扣分制度，那么可将问题转化为，使用一次黄色石头加 1 分，使用一次蓝色石头加 2 分，使用一次红色石头加 3 分，使用一次绿色石头加 4 分，而后再通过 Dijkstra 算法求取从起点到终点的最短路径，即最少得分路径，而后再通过满分 1000，减去对应的最少得分，则可得到最终的最高得分。

6.2 图论模型求解

综上分析，首先将图中所有节点均拆为 2 个节点即入点和出点，那么图中节点的数量就要增大 2 倍，而后再从入点向出点连接一条权值为原节点颜色权值的边，为了保证图的连通性，将在$K = 100$的图上进行，并把原距离权值赋为 0，这里仍然选择编号 1 作为起点，而编号为 111，112，113，114 则作为终点，最后通过题意转化，使得只需再使用 Dijkstra 算法求取最短路，则可得到最高分路径。

表 2.1 问题二在$K = 100$上的最高得分

每次攀爬最大高度/编号	111	112	113	114
100	969	967	967	969

在 Dijkstra 算法寻找最短路的过程中，只需要用一个数组 prev[N]记录每个节点在最短路径中的前驱节点，而后再通过从终点向起点回溯，将回溯得到的路径依次添加到一个路径数组中，然后反向输出这个路径数组，就能够得到从起点到终点的具体路径，故可得到下表即为从起点编号 1 的节点向，编号为 111，112，113，114 的最高得分路径。

表 2.2 问题二从起点向终点的最高得分路径

终点编号	路径
111	$1 \rightarrow 7 \rightarrow 15 \rightarrow 24 \rightarrow 29 \rightarrow 34 \rightarrow 40 \rightarrow 48 \rightarrow 57 \rightarrow 60 \rightarrow 69 \rightarrow 75 \rightarrow 84 \rightarrow 91 \rightarrow 100 \rightarrow 103 \rightarrow 111$

112	$1 \to 7 \to 15 \to 24 \to 29 \to 34 \to 40 \to 48 \to 57 \to 60 \to 69 \to 78 \to 88 \to 93 \to 101 \to 107 \to 112$
113	$1 \to 7 \to 15 \to 24 \to 29 \to 34 \to 40 \to 48 \to 57 \to 60 \to 69 \to 78 \to 88 \to 93 \to 101 \to 110 \to 113$
114	$1 \to 7 \to 15 \to 24 \to 29 \to 34 \to 40 \to 48 \to 57 \to 60 \to 69 \to 78 \to 88 \to 89 \to 94 \to 96 \to 105 \to 114$

在图上可直观由下图表示：

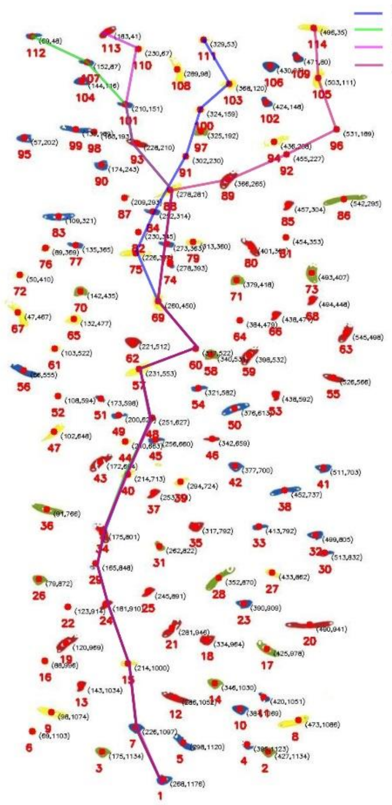

图 2.2 $K = 100$起点向终点最高得分路径图

七、问题三的模型与建立

7.1 攀岩难度评级

7.1.1YDS 系统介绍

本文引用 YDS 作为评价攀岩难度的模型系统，1937 年 Welzenbach 的攀登难度系统由山峦俱乐部引入美国，之后衍生为优胜美地十进制系统(YDS)，它将难度分为 6 个大类。

（1） 徒步

（2） 几乎用不到手的爬行

（3） 需要携带绳子，但是几乎用不到的爬行

（4） 通常需要绳子，很容易找到天然的保护点，脱落是致命的

（5） 技术攀登，要有一定的攀岩技术，绳子和保护是必须的

（6） 必须借助器械才能进行的攀登

其中这里的第五个级别就是指攀岩，也是常说的五级技术攀登，到了 20 世纪 50 年代，五级攀登由于需要被划分成 5.0，5.1，5.2，5.3，5.4，5.5，5.6，5.7，5.8，5.9，5.10，前面的数字 5 仍然代表五级攀登，后面的 0 到 10 代表攀登路线的难易程度，这是个封闭的系统，最简单的线路被定义为 5.0，而最难的线路被定为 5.10。

7.1.2 评价指标选择

通过查阅相关资料，本文将选取路径总长度以及岩石之间的最大距离差，所攀爬岩石的个数作为评价指标，以评价该模型，并采用 YDS 系统的评分方式，以下是问题二路线的指标统计结果。

表 3.1 指标统计结果

终点编号	路径总长度	岩石之间最大距离差	攀爬岩石个数	总减分
111	1205.84	97.7394($7 \rightarrow 15$)	17	31
112	1254.05	97.7394($7 \rightarrow 15$)	17	33
113	1216.8	97.7394($7 \rightarrow 15$)	17	33
114	1277.48	99.0202($28 \rightarrow 33$)	18	31

7.2Kmeans 聚类分析

7.2.1 Kmeans 算法原理

已知数据集$(x_1, x_2, \dots x_n)$，Kmeans 聚类要把这n个数据划分到k个集合中$(k \le n)$，使得组内平方和最小，它的目标是找到使得下式满足的聚类S_i

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{x \in S_i} \left| |x - \mu_i| \right|^2 \qquad (4-1)$$

其中μ_i是S_i中所有点的均值。

7.2.2 Kmeans 算法步骤

（1） 对数据集进行标准化和归一化，避免均值和方差大的数据对聚类产生决定性影响

（2）选择初始化的k个样本作为初始聚类中心$a = a_1, a_2 \dots a_k$

（3）针对数据集中每个样本x_i，计算它到k个聚类中心的距离，并将其分到距离最小的聚类中心所对应的类中

（4）针对每个类别$a_j (j = 1,2..k)$，重新计算它的聚类中心$a_j = \sum_{x \in S_i} x$，即属于该类的所有样本的质心

（5）重复(3)(4)步，知道达到某个中止条件（迭代次数，可允许最小误差等）

其伪代码为：

```
获取数据 n 个 m 维的数据
随机生成 K 个 m 维的点
while(t)
    for(int i=0;i < n;i++)
        for(int j=0;j < k;j++)
```

计算点 i 到类 j 的距离
for(int i=0;i < k;i++)
1. 找出所有属于自己这一类的所有数据点
2. 把自己的坐标修改为这些数据点的中心点坐标
end

7.2.3 Kmeans 模型求解

根据机器学习中的肘部准则，选取斜率最大的$K = 2$的分类个数肘部图，确定了以聚类为 2 能更好地描述评价指标的组合。

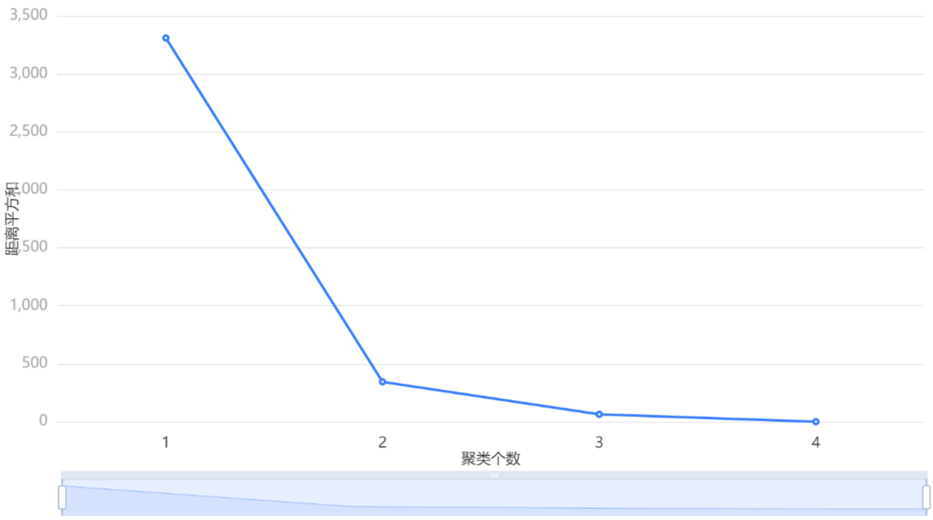

图 3.1 聚类数对比图

表 3.2 数据集聚类标注表

聚类种类	终点编号	路径总长度	岩石之间最大距离差	攀爬岩石个数	总减分
1	111	1205.84	97.7394	17	31
2	112	1254.05	97.7394	17	33
1	113	1216.8	97.7394	17	33
2	114	1277.48	99.0202	18	31

通过 Python 的可视化库 matplotlib 可以直观地展现出聚类散点图，由于变量数大于 2 个，即取主成分分析(PCA)降维后前两个主成分来绘制散点图，在一定程度上可查看聚类效果。

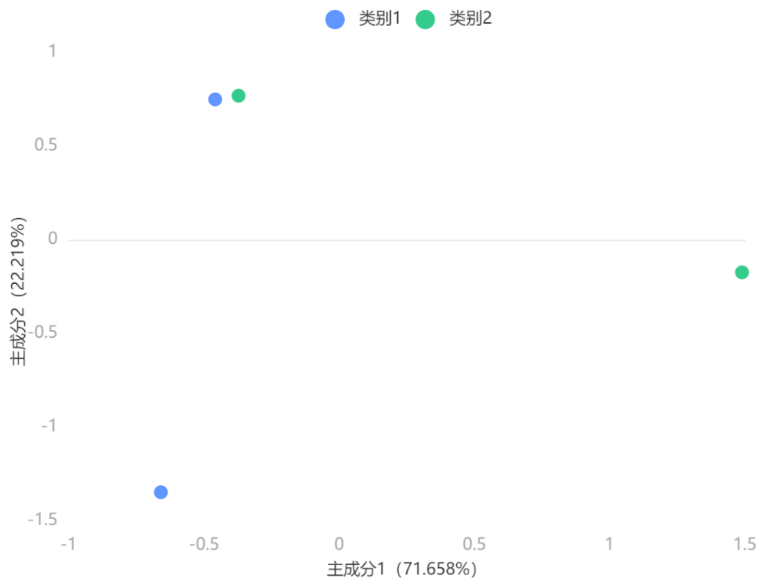

图 3.2 聚类散点图

由于是通过 Dijkstra 算法选择的最多得分路径，且与最短路径基本吻合，在聚类分析分为了两类的情况下，可将这 4 条路线分为 2 条评价等级为优的路线和 2 条评价等级为良的路线，经过映射可得到问题三的最终结果表如下所示：

表 3.3 问题三路线评级

终点编号	路径	评价等级(YDS)
111	$1 \to 7 \to 15 \to 24 \to 29 \to 34 \to 40 \to 48 \to 57 \to 60 \to 69 \to 75 \to 84 \to 91 \to 100 \to 103 \to 111$	5.1
112	$1 \to 7 \to 15 \to 24 \to 29 \to 34 \to 40 \to 48 \to 57 \to 60 \to 69 \to 78 \to 88 \to 93 \to 101 \to 107 \to 112$	5.3
113	$1 \to 7 \to 15 \to 24 \to 29 \to 34 \to 40 \to 48 \to 57 \to 60 \to 69 \to 78 \to 88 \to 93 \to 101 \to 110 \to 113$	5.2
114	$1 \to 7 \to 15 \to 24 \to 29 \to 34 \to 40 \to 48 \to 57 \to 60 \to 69 \to 78 \to 88 \to 89 \to 94 \to 96 \to 105 \to 114$	5.4

八、模型评估和改进

8.1 模型优点

（1）对于问题一本文选择了以岩石轮廓的质心作为图论节点，在一定程度上简化了计算，使得问题聚焦于路线的规划。

（2）对于问题一本文通过枚举每次攀爬者最大上升高度，找到了合适的常数$K = 100$，并以此构建网络图，保证了图的连通性。

（3）对于问题一本文通过分析节点数和边数，选择了以邻接表建图，使得算法运行更高效。

（4）对于问题二本文通过使用拆点算法，将点权转变为了边权，使得可以使用更高效的图论算法，加快了程序的运行效率。

（5）对于问题三本文使用了聚类分析，并通过肘部原则，选取了聚类数为 2，以此更好地从整体方面评价路径。

8.2 模型缺点

（1）本文忽略了岩石轮廓的大小和岩石之间的角度方向考虑，缺少了力学方面的计算分析。

（2）Dijkstra 算法会有其对应的局限性，如不能处理负权边，此时可以使用 SPFA 或 Floyd 算法进行优化。

（3）由于本题数据过少，聚类分析的结果不能很好的统计分析，不能很好地应用在大批量数据。

参考文献

[1] 王伯宇. 攀岩運動型態之分析與比較[J]. 大專體育, 2003 (66): 62-68.

[2] 朱松梅. 攀岩运动力量训练研究[J]. 河南师范大学学报: 自然科学版, 2010, 38(2): 169-171.

[3] 张福浩, 刘纪平, 李青元. 基于 Dijkstra 算法的一种最短路径优化算法[J]. 遥感信息, 2004, 2(4).

[4] 乐阳, 龚健雅. Dijkstra 最短路径算法的一种高效率实现[D]. , 1999.

[5] 侯宾, 张文志, 戴源成, 等. 基于 OpenCV 的目标物体颜色及轮廓的识别方法[J]. 现代电子技术, 2014, 37(24): 76-79.

[6] 王千, 王成, 冯振元, 等. K-means 聚类算法研究综述[J]. 电子设计工程, 2012, 20(7): 21-24.

附录

问题一

OpenCV 轮廓识别代码：

```
import cv2
import numpy as np
import heapq

def euclidean_distance(point1, point2):
    return np.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)

def dijkstra(graph, start):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

title = "using 70"
cv2.putText(output, title, (30, 50), cv2.FONT_HERSHEY_SIMPLEX, 5, (0, 0, 255),
5)

image = cv2.imread("test.jpg")
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, threshold1=20, threshold2=60)
adaptive_thresh = cv2.adaptiveThreshold(gray, 255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                        cv2.THRESH_BINARY, 11, 2)
kernel = np.ones((5, 5), np.uint8)
dilated = cv2.dilate(adaptive_thresh, kernel, iterations=1)
contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

```
output = np.copy(image)
center_points = []

index = 1
for idx, contour in enumerate(contours):
    area = cv2.contourArea(contour)
    if area > 5:
        M = cv2.moments(contour)
        # if M["m00"] != 0:
        cX = int(M["m10"] / M["m00"])
        cY = int(M["m01"] / M["m00"])
        cv2.circle(output, (cX, cY), 5, (0, 0, 255), -1)
        cv2.putText(output, f"({cX},{cY})", (cX - 50, cY - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 2)
        cv2.putText(output, f"({cX},{cY})", (cX + 10, cY + 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.3, (0, 0, 0), 1)
        cv2.putText(output, f"{index}", (cX - 10, cY + 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
        center_points.append((cX, cY))
        index = index + 1

graph = {}
for i in range(len(center_points)):
    graph[i] = {}
    for j in range(i + 1, len(center_points)):
        distance = euclidean_distance(center_points[i], center_points[j])
        if distance <= 100:
            graph[i][j] = distance
            # cv2.line(output, center_points[i], center_points[j], (255, 0, 0), 1)  # 修
改颜色为蓝色

shortest_paths = dijkstra(graph, start=1)

target_points = [181, 174, 182]
for target in target_points:
    if target in shortest_paths:
            print(f"Shortest distance from point 1 to point {target}:
{shortest_paths[target]:.2f}")
    else:
        print(f"No path from point 1 to point {target}")

cv2.imwrite("label.jpg", output)
cv2.imshow("Rock Contours, Centers, and Blue Edges", output)
```

```
cv2.waitKey(0)
cv2.destroyAllWindows()
```

将中心点坐标存入文本文件中：

```
# 将 center_points 写入文本文件中
file_name = "center_points.txt"

# 打开文件以写入模式
with open(file_name, "w") as file:
    for point in center_points:
        # 将每个元组的数据转换为字符串，然后写入文件
        file.write(f"{point[0]} {point[1]}\n")

print("列表已成功写入文本文件。")
```

使用 Dijkstra 算法求解最短路：

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <fstream>
#include <cmath>
#include <queue>
#include <cstring>
#include <map>
using namespace std;

struct Points
{
    int id;
    int x, y;
};

struct Edges
{
    int id1, id2;
    double distance;
};

const int N = 150;
double g[N][N];
double dist[N];
bool st[N];
int s = 1;
```

```
void dijkstra()
{
    int prev[N]; // 用于记录每个节点的前驱节点
    for (int i = 1; i <= N; i ++)
    {
        dist[i] = 10000000000.0;
        prev[i] = -1;
    }
    memset(st, 0, sizeof st);
    dist[s] = 0;

    int n = 115;
    for (int i = 1; i <= n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;
        st[t] = true;
        for (int j = 1; j <= n; j ++ )
        {
            if (dist[j] > dist[t] + g[t][j])
            {
                dist[j] = dist[t] + g[t][j];
                prev[j] = t; // 记录节点 j 的前驱节点是 t
            }
        }
    }

    // 打印从节点 1 到节点 111 的路径
    int current = 114;
    vector<int> path;
    while (current != -1)
    {
        path.push_back(current);
        current = prev[current];
    }
    reverse(path.begin(), path.end());

    cout << "从 1 到 113 的路径经过的节点编号: ";
    for (int node : path)
    {
        cout << node << " ";
```

```
    }
    cout << endl;

    double maxEdgeDistance = 0;
    int maxEdgeId1 = -1, maxEdgeId2 = -1;
    for (int i = 1; i < path.size(); i++)
    {
        int u = path[i - 1];
        int v = path[i];
        double edgeDistance = g[u][v];
        if (edgeDistance > maxEdgeDistance)
        {
            maxEdgeDistance = edgeDistance;
            maxEdgeId1 = u;
            maxEdgeId2 = v;
        }
    }

    cout << "最大距离的边信息：" << endl;
    cout << "起点：" << maxEdgeId1 << " 终点：" << maxEdgeId2 << " 距离：" <<
maxEdgeDistance << endl;
}

int main()
{
    Points center_points[N];
    ifstream file("center_points.txt");
    int point_id = 1;
    while (!file.eof())
    {
        int point_x, point_y;
        file >> point_x >> point_y;
        center_points[point_id] = {point_id, point_x, point_y};
        point_id ++;
    }

    int n = point_id - 1;
    Edges edges[N * N];
    int edge_id = 0;
    for (int i = 1; i <= n; i ++)
    {
        for (int j = i + 1; j <= n; j ++)
        {
            double distance = sqrt(pow(center_points[i].x - center_points[j].x, 2) +
```

```
pow(center_points[i].y - center_points[j].y, 2));
                if (distance <= 100)
                {
                    edges[++edge_id] = {i, j, distance};
                }
            }
        }

    for (int i = 1; i <= N; i ++)
    {
        for (int j = 1; j <= N; j ++)
        {
            g[i][j] = 10000000000.0;
        }
    }

    for (int i = 1; i <= edge_id; i ++)
    {
        g[edges[i].id1][edges[i].id2] = edges[i].distance;
        g[edges[i].id2][edges[i].id1] = edges[i].distance;
    }

    dijkstra();

    return 0;
}
```

问题二

拆点并使用最短路：

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <fstream>
#include <cmath>
#include <queue>
#include <cstring>
#include <map>
using namespace std;

const int N = 300, M = N << 1;
int h[N], e[M], ne[M], w[M], idx;
int dist[N];
bool st[N];
```

```
struct Points
{
    int id;
    int x, y;
    int color_id;
};

struct Edges
{
    int id1, id2;
    double distance;
};

typedef pair<int, int> PII;
int t = 114;
int s = 1, s_n = s + t; //  起点的出点和起点的入点

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
}

void dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    memset(st, 0, sizeof st);
    priority_queue<PII, vector<PII>, greater<>> heap;
    heap.push({0, s});
    dist[s] = 0;

    int prev[N];   //  用于记录路径

    while (!heap.empty())
    {
        PII t = heap.top();
        heap.pop();

        if (st[t.second])   //  跳过已处理的点
            continue;

        st[t.second] = true;

        for (int i = h[t.second]; ~i; i = ne[i])
```

```
        {
            int j = e[i];
            if (dist[j] > dist[t.second] + w[i])
            {
                dist[j] = dist[t.second] + w[i];
                prev[j] = t.second;   //  记录路径

                heap.push({dist[j], j});
            }
        }
    }

    //  打印经过的点的编号
    int current = t + 114;   //  终点编号
    vector<int> path;
    while (current != s)
    {
        path.push_back(current);
        current = prev[current];
    }
    path.push_back(s);

    cout << "114 号经过的点的编号：";
    for (int i = path.size() - 1; i >= 0; i-=2)
    {
        cout << path[i] << " ";
    }
    cout << endl;
}

int main()
{
    Points center_points[N];
    ifstream file("center_points_label.txt");
    int point_id = 0;
    while (!file.eof())
    {
        int point_x, point_y, color_id;
        file >> point_x >> point_y >> color_id;

        if (color_id == 1) color_id = 3;
        else if (color_id == 2) color_id = 4;
        else if (color_id == 3) color_id = 2;
        else if (color_id == 4) color_id = 1;
```

```
        center_points[++ point_id] = {point_id, point_x, point_y, color_id};
        // cout << point_id << " " << point_x << " " << point_y << " " << color_id <<
endl;

        if (point_id == 114) break;
    }

    // cout << point_id << endl;

    int n = point_id;
    Edges edges[N * N];
    int edge_id = 0;
    for (int i = 1; i <= n; i ++)
    {
        for (int j = i + 1; j <= n; j ++)
        {
            double distance = sqrt(pow(center_points[i].x - center_points[j].x, 2) +
pow(center_points[i].y - center_points[j].y, 2));
            if (distance <= 100)
            {
                edges[++edge_id] = {i, j, distance};
            }
        }
    }

    // 建图

    memset(h, -1, sizeof h);

    for (int i = 1; i <= t; i ++)
    {
        add(i, i + t, center_points[i].color_id); // 起点的出点和起点的入点 权值是
颜色值
    }

    for (int i = 1; i <= edge_id; i ++)
    {
        add(edges[i].id1 + t, edges[i].id2, 0); // 该点的出点到另外一个点的入点 权
值是 0
    }

    dijkstra();
```

```
    cout << dist[114 + t] << endl;

    return 0;
}
```

文本文件 center_points_label.txt

```
268 1176 3
427 1134 2
175 1134 2
395 1123 3
298 1120 3
69 1103 2
226 1097 3
473 1086 4
98 1074 4
384 1069 3
420 1051 3
286 1052 1
143 1034 1
346 1030 2
214 1000 4
88 996 2
425 978 2
334 964 1
120 969 1
490 941 1
281 946 1
123 914 1
390 909 3
181 910 1
245 891 1
79 872 2
433 862 4
352 870 2
165 848 3
513 832 3
262 822 2
499 805 3
413 792 3
175 801 1
317 792 1
91 766 2
253 741 1
```

452 737 3
294 724 4
214 713 4
511 703 3
377 700 3
172 694 1
210 663 4
256 660 3
342 659 1
102 648 4
251 627 3
200 623 3
376 613 3
173 598 3
108 594 4
438 592 1
321 582 3
526 566 1
56 555 3
231 553 4
340 531 2
398 532 1
317 522 1
103 522 4
221 512 1
545 498 1
384 479 1
132 477 1
438 471 1
47 467 4
494 448 1
260 450 4
142 435 2
379 418 2
50 410 1
493 407 2
278 393 3
226 377 4
89 369 2
135 365 3
273 363 3
313 360 4
401 367 1
454 353 2

230 345 3
109 321 3
252 314 3
457 304 1
542 295 2
209 293 2
278 281 4
366 265 1
174 243 3
302 230 2
455 227 3
228 210 1
436 208 4
57 202 3
531 189 4
325 192 2
163 193 3
135 189 3
324 159 4
210 151 3
424 148 3
368 120 4
144 116 3
503 111 4
430 93 3
152 87 3
289 98 4
471 80 4
230 67 4
329 53 4
69 48 3
183 41 1
496 35 4