南開大學
Nankai University

计算机学院

算法导论大作业实验报告

基于聚类算法的探讨、优化与实际应用

姓名：曹珉浩

学号：213619

专业：计算机科学与技术

2023 年 6 月 1 日

摘要

生活和科学中的很多方面都需要用到聚类，因此开发一个高效的聚类算法十分重要。在本实验中，我们主要实现了基于贪心策略的 Kruskal 聚类算法和基于迭代思想的 K-means 聚类算法，这两种算法各有所长，适合应用在不同的领域。我们详细分析了这两种算法的时空复杂度并对这两种算法进行了不同方法的优化，实验效果显著。最后我们利用 K-means 算法解决了一个实际问题：图像分割。

本实验的所有代码以及测试样例都存放于压缩包中，以及 Github 网站：

https://github.com/MrGuaB1/Introduction-to-Algorithm/tree/main/K-means

关键词：贪心算法、迭代思想、Kruskal、K-means、图像分割、OpenCV、SIMD、并查集

目录

1 问题引入

在生活中，我们常常需要把带有某种相似属性的东西划分到一起：如在将消费者根据其购买行为、偏好和属性进行分组，以便进行市场细分和目标营销；以及对文本数据进行聚类，从大量的文档中发现相似主题或类别，以便进行信息提取、文本分类和推荐等任务；还有图像分割：在计算机视觉和图像处理领域中，常常需要将一幅图像划分为不同的区域或对象，其中每个区域具有相似的特征，用以达到图像分类、物体跟踪、图像重建等目的。

上述问题都可以通过聚类的思想解决：聚类是一种无监督学习方法，用于将相似的数据点分组或归类到同一组别中，算法通过在数据中发现内在的模式和结构，将数据点划分为不同的簇，使得同一簇内的数据点之间具有较高的相似性，而不同簇之间的数据点相似性较低。

算法的形式化定义为：

- 有 n 个 m 维数据 $\{x_1, x_2, ...x_n\}, x_i \in R^m\,(1 \le i \le n)$
- 任意两个 m 维数据之间的欧氏距离为

$$d(x_i, x_j) = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + ... + (x_{in} - x_{jn})^2}$$

  $d(x_i, x_j)$ 值越小，表示 x_i, x_j 越相似，反之越不相似

- 聚类集合数目 K
- 算法目标：将 n 个数据依据它们之间的相似度大小将它们分别聚类到 K 个集合

在课堂上介绍了利用 Kruskal 算法改造的聚类算法的实现思路，我们将首先实现这个算法，分析它的性能并寻找改进点；之后实现无监督学习经典算法 K-means，并利用《并行程序设计》课程所学内容，对算法进行优化。

2 Kruskal 聚类算法

2.1 算法设计

为了找到最大间隔的聚类，我们考虑在结点集上长出一个图，连通分支将是聚类，我们试图把临近的点尽可能地一起带入同一个聚类中，所以我们按照 $d(x_i, x_j)$ 递增的次序不断在点对之间增加边，这个过程刚好就是 Kruskal 算法的执行过程，Kruskal 算法贪心的不断向集合中加入权值最小的边，通过并查集数据结构来维护结点的连通性，当加入了所有结点时算法结束，此时只有一个连通分支。可以看到，如果我们把算法提前 K 步结束，那么我们就可以得到 K 个连通分支，由于贪心算法领先，可知我们产生了一个最优解 (K 聚类)。

算法实现的一个关键就是并查集的实现：并查集的主要构成就是一个前向数组和并函数 unite()、查函数 find()，前向数组记录了每个点的前驱结点是谁，查函数用于查找结点属于哪个集合，并函数用于合并两个集合。这两个关键函数的代码如下：

```
1 int find(int x) { //查函数，判断结点 x 属于哪个集合
2     while(pre[x] != x)
```

```
        x = pre[x];
    return x;
}
void unite(int x, int y) { //并函数，用于合并两个集合
    int fx = find(x);
    int fy = find(y);
    if (fx != fy)
        parent[fx] = fy;
}
```

在测试用例方面，我们采取颜色作为输入数据，原因是自然界中的各种颜色都可以用 RGB 三原色表示，这是一个天生的三标签数据，我们可以很自然地将其划分为三个聚类，同时还可以利用 HTML 使得到的颜色分类结果可视化，验证我们程序的正确性。然后基于本小节对颜色相似性的研究，在后面开发利用 K-means 算法实现图像分割。

算法实现方面，将 n 个数据之间构成完全图，任意两点间的边为它们的欧氏距离，将这些边加入 vector 容器中，并按权值递增排序，接下来改造 Kruskal 算法，当加入了 $n-k$ 个结点之后算法结束，并注意此时要刷新前向数组，当算法结束时，我们就得到了 k 个代表元素，按照这 k 个代表元素，就可以寻找到 k 个聚类。

Algorithm 1 Kruskal 聚类算法伪代码

Input: 元素数目 n，n 行元素标签 (label1,label2,label3)

Output: K 聚类结果

function K_CLUSTER(n)
    construct complete graph and sort all edges
    transfer Kruskal(n), but stop when $n-k$ nodes have been added
    refresh pre[n]
    construct K vectors to record different clusters by traverse refershed pre[n]
end function
traverse K vectors in main() to get our result

2.2 算法分析

- 空间复杂度

算法利用了 n 个元素之间构成的完全图，以及最后记录 n 个元素所在聚类的向量，因此：

$$\text{空间复杂度} = O(C_n^2) + O(n) = O(n^2)$$

- 时间复杂度

算法主要由排序边，遍历边，以及遍历聚类向量三部分组成，边数 $e = C_n^2 = n(n-1)/2$ 。其中，排序边、遍历边时间复杂度 $O(e) = elog\,e = O(n^2 log\,n)$，遍历聚类向量的时间复杂度为 $O(n)$，因此算法的时间复杂度为 $O(n^2 log\,n)$。

2.3 输出结果

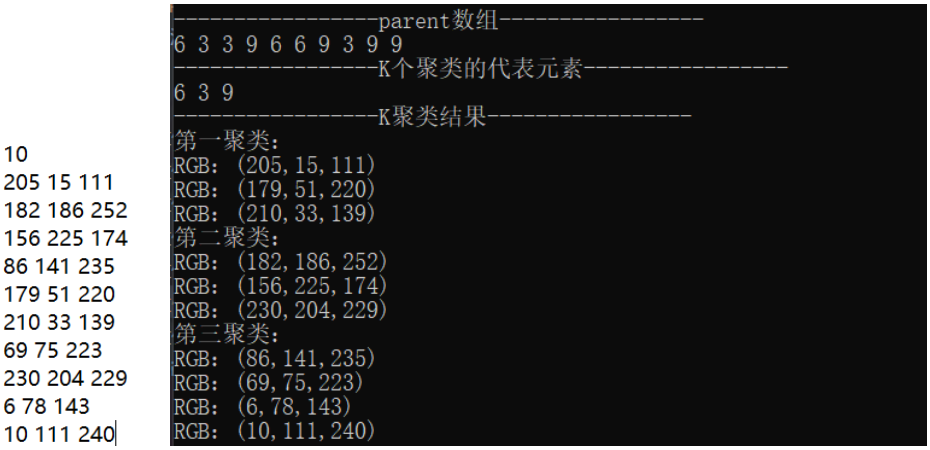

图 2.1: Example1 and command line output

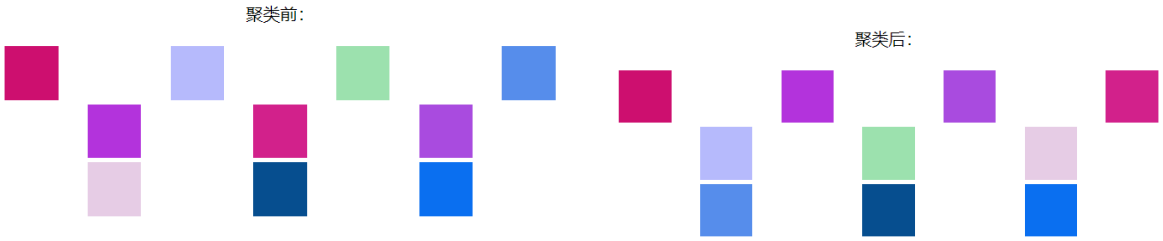

图 2.2: Example1 visualabel output

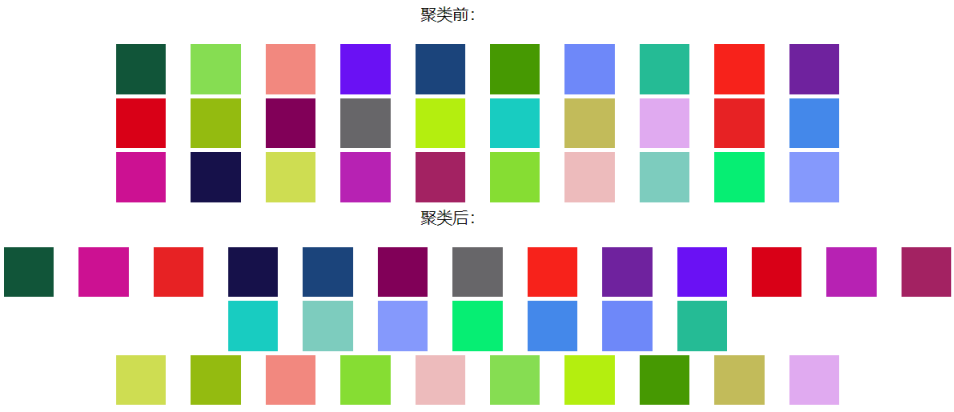

图 2.3: Example2 visualabel output

2.4 算法优化

在上面的并查集中，我们采取的查找策略是不断向前推，直至找到代表元素，这个过程可能产生一个很长的查找路径 (搜索树很高)，如图2.4所示。假设要判断夏侯和许褚是否属于同一阵营，那么就要遍历这两条最长的搜索路径，极大的降低了搜索效率。

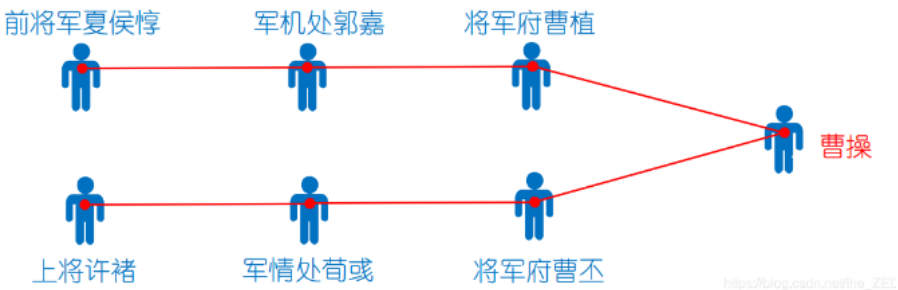

图 2.4: 每一次查找都要逐层向前推，并查集搜索树很高

如果想要避免这种极端的串行情况，我们可以在查找的过程中降低树的高度，一种做法是通过函数递归，找到代表元素后逐层修改每一层的前驱结点，这样树的高度就大大降低了。

```
1 int find_pro(int x) {
2     if (parent[x] != x)
3         parent[x] = find(parent[x]); //在搜索的过程中直接改变上下级关系，降低树高
4     return parent[x];
5 }
```

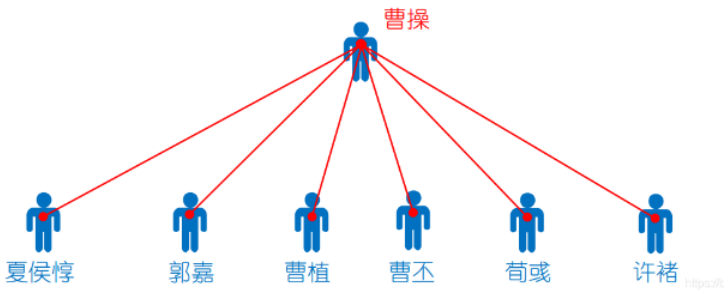

图 2.5: 降低树的高度，保证搜索高效性

问题规模	100	500	1000	2000	4000	10000
优化前/ms	6.74	51.27	105.93	224.66	432.78	1992.01
优化后/ms	6.59	46.17	87.74	175.32	337.06	1071.28

表 1: 算法优化前后时间对比

可见问题规模越大，原先算法可能产生的树高问题就会越严重，优化后的算法效率提升就越明显。虽然仍然不能把聚类问题的时间复杂度降低到强多项式时间，但已经做到了一定程度的改进。

3 K-means 聚类算法

在上面的 Kruskal 聚类算法中我们看到，其时间复杂度和空间复杂度都很高，通常并不是一种常用的聚类算法。K-means(K 均值聚类) 算法是基于另外一种求解思路的算法，它的时间复杂度一般要比 Kruskal 聚类算法要低。

3.1 算法设计

K-means 算法是一种迭代的聚类算法，其主要思路可以概况为：

- 初始化：选择 K 个初始聚类中心点，一般随机选择即可
- 分配：对于每个数据点，计算其与各个聚类中心的距离，并分配到最近的聚类中心所在的簇
- 更新：对每个簇，计算簇内所有数据点的均值，将该均值作为新的聚类中心
- 重复第二步和第三步，直至达到最大迭代次数或簇分配不再变化

算法的目标是最小化簇内数据点与聚类中心的平方距离之和，通过迭代更新聚类中心和重新分配数据点，逐步优化聚类结果。最终得到的聚类结果可以提供对数据集的结构和模式的认识。其中，迭代公式可以抽象为：

$$
\begin{gathered}
d(P_m, clusterCenter_i(index)) = min\{d(P_m, clusterCenter_i(j)), j = 1, 2, ..., k\} \\
clusterAssignment_{i+1}(m) = index \\
clusterCenter_{i+1}(n) = mean(\{P_j | clusterAssignment_{i+1}(j) = n\})
\end{gathered}
$$

在测试用例方面，我们选择每个元素带有两个标签，可以理解为平面上一个点的横纵坐标，便于理解。策略选择方面，我们选择数据流的前 K 个元素作为初始质心，并令两次调用均差和函数之间的差值小于一个给定常数时判定算法结束，则算法关键步骤的代码如下：

```
while (abs(newVar - oldVar) >= eps) {
    for (i = 0; i < K; i++) //更新每个簇的中心点
        means[i] = getMeans(clusters[i]);
    oldVar = newVar;
    newVar = getVar(clusters, means); //计算新的准则函数值
    for (i = 0; i < K; i++) //清空每个簇
        clusters[i].clear();
    for (i = 0; i != tuples.size(); ++i) { //根据新的质心获得新的簇
        label = clusterOfTuple(means, tuples[i]);
        clusters[label].push_back(tuples[i]);
    }
//输出这一轮迭代后当前的簇情况
    for (label = 0; label < K; label++) {
```

```
14          cout << " 第" << label + 1 << " 个簇: " << endl;
15          vector<Tuple> t = clusters[label];
16          for (i = 0; i < t.size(); i++)
17             cout << "(" << t[i].attr1 << "," << t[i].attr2 << ")" << "    ";
18          cout << endl;
19      }
20  }
```

3.2 算法分析

- 时间复杂度

由于上面的代码是基于比较误差值判定算法结束的，这个迭代次数依赖于输入的数据。因此程序何时结束并不能被准确判断。实际上，K-means 算法可以通过规定最大迭代次数来终止算法 (但得到的效果未必有基于误差值比较得到的效果好)，若基于最大迭代次数终止算法，则 K-means 的时间复杂度:

- 分配过程为 $O(N*K*d)$，其中 N 为数据点的数量，K 为聚类数量，d 为数据点的维度
- 更新过程为 $O(N*K*d)$，这个过程需要对每个聚类计算均值
- 最大迭代次数记为 M

  则 K-means 聚类算法的时间复杂度为 $O(N*K*d*M)$

- 空间复杂度

由于 K-means 算法没有像 Kruskal 算法那样的建图过程，算法占用的空间只有读入的数据，保留的中心点以及临时变量，K-means 算法的空间复杂度为 $O(N+K)$。

通过两种聚类算法时空复杂度的对比，可以看到 K-means 算法更加小巧灵活，运行效率更高，因此适合于处理更多的数据，但 K-means 算法相比 Kruskal 聚类算法的缺点在于:

- 初始化聚类中心对结果有影响,不同的初始化结果可能会产生不同的聚类结果,不同于 Kruskal 聚类算法产生唯一最优解
- 受离群值影响很大，如果有一个点距离其他所有点极远，那么 K-means 算法倾向于将这个离群点单独划为一类，不同于基于升序排列的 Kruskal 聚类算法，它会把离群点归为最后一个聚类，因此得到的结果更为均匀
- 对聚类数目 K 需要事先指定，而 Kruskal 聚类算法可以很方便的改为基于动态规划的算法，不需要指定聚类的数目 K

3.3 输出结果

- Sample in：数据文件名，如 data.txt
- Sample out：每一次迭代的每个簇的具体情况

```
11 45
14 19
19 81
12 49
13 24
56 84
12 73
34 20
16 18
20 69
55 55
44 44
81 19
31 13
28 57
47 74
```

```
第1个簇:
(11,45) (12,49) (44,44) (28,57)
第2个簇:
(14,19) (13,24) (34,20) (16,18) (81,19) (31,13)
第3个簇:
(19,81) (56,84) (12,73) (20,69) (55,55) (47,74)
第1个簇:
(11,45) (12,49) (44,44) (28,57)
第2个簇:
(14,19) (13,24) (34,20) (16,18) (81,19) (31,13)
第3个簇:
(19,81) (56,84) (12,73) (20,69) (55,55) (47,74)
第1个簇:
(11,45) (12,49) (44,44) (28,57)
第2个簇:
(14,19) (13,24) (34,20) (16,18) (81,19) (31,13)
第3个簇:
(19,81) (56,84) (12,73) (20,69) (55,55) (47,74)
```

图 3.6: K-means 算法输入输出示例

3.4 算法优化

在 K-means 算法中，包含大量的距离计算函数，这些计算简单，且完全没有任何数据依赖，因此这个过程十分适合并行化，考虑采用 SIMD 技术 (采用 x86 架构，SSE 指令集) 来加速这个过程，设计的思路如下：

- 每次取出多个数据同一维度的数据存储到向量寄存器当中
- 利用向量的减法运算，计算与质心在这一维度上的距离差的平方
- 重复上述过程直至这次拿出的数据的所有维度处理完毕
- 再取出一组新的数据，重复上述过程直至所有数据处理完毕

```
for (int d = 0; d < D; d++){
    for (int i = 0; i < N - N % 4; i += 4) {
        __m128 distance = _mm_loadu_ps(&distanceToCent[i]); //取出每轮累加的距离
        float tmp_centroid_d[4] = { centroids[j][d] }; //质心的数据
        __m128 centroid_d = _mm_loadu_ps(tmp_centroid_d);
        __m128 data_d = _mm_load_ps(&data_align[d][i]);
        __m128 delta = _mm_sub_ps(data_d, centroid_d); //向量减法
        distance = _mm_add_ps(distance, _mm_mul_ps(delta, delta)); // 对每一数据该维度累加距离
        _mm_storeu_ps(&distanceToCent[i], distance); // 存回
    }
}
```

由于文件读写比较耗时，且普通 K-means 算法在函数中有大量的状态输出，因此对比时间不太合理，在这里只列出并行优化后算法的效率，通过前面 Kruskal 聚类算法的间接对比可以看到有很明显的提升。

问题规模	100	500	1000	2000	4000
优化后/ms	3.80	16.40	32.40	70.50	139.10

表 2: 并行优化算法运行时间

3.5 算法应用

如引言所述，K-means 算法在计算机视觉领域发挥了重要的作用，在本小节中，我们将使用 OpenCV 库中提供的 kmeans 函数开发一个实际应用：图像分割。

库函数的形式为：**cv::kmeans(InputArray data, int K, OutputArray labels, TermCriteria criteria, int attempts, int flags, OutputArray centers = noArray())**，其中：

- data：输入的数据集，类型为 cv::InputArray，可以是 cv::Mat 或 cv::Mat_<float>
- K：聚类的簇数。
- labels：输出的标签矩阵，类型为 cv::OutputArray，每个元素表示对应数据点所属的簇索引。
- criteria：终止条件，类型为 cv::TermCriteria，定义了迭代停止的条件，例如最大迭代次数或聚类中心的变化阈值。
- attempts：重复运行 K-means 算法的次数，每次都会使用不同的随机初始聚类中心。
- flags：附加标志，控制 K-means 算法的行为。
- centers：输出的聚类中心矩阵，类型为 cv::OutputArray，每行表示一个聚类中心。

要使用 OpenCV 库,首先要到官网下载安装,并包含头文件 #include <opencv2/opencv.hpp>，算法的实现思路如下：

- 取图像文件，将其存储在 src(cv::Mat 类型) 矩阵中。
- 创建 samples 矩阵，用于存储每个像素点的颜色特征。通过遍历图像的每个像素点，将其 RGB 值存储在 samples 矩阵中的相应位置。
- 调用库函数 kmeans()。
- 创建一个与原始图像大小相同的 new_image 矩阵，用于存储聚类后的图像。
- 遍历图像的每个像素点，根据其所属的簇标签，将对应的聚类中心值赋给 new_image 中的相应像素点。

最关键的 kmeans 函数的调用方式如下，其中 samples 是样本输入数据；clusterCount 是期望的聚类数，labels 是样本的簇标签；TermCriteria 是停止条件，这里是当迭代次数达到最大迭代次数 (10000) 或者聚类中心的更新小于给定的精度 (0.0001) 时，算法将停止迭代；attempts 是尝试次数 (下面实验结果的尝试次数是 5)；KMEANS_PP_CENTERS 是一种指定聚类中心的选择方法，这种方法的效果能更好的初始化簇中心；centers 是输出参数，据此得到生成图像。

```
kmeans(samples, clusterCount, labels, TermCriteria(cv::TermCriteria::COUNT |
    cv::TermCriteria::EPS, 10000, 0.0001), attempts, KMEANS_PP_CENTERS, centers);
```

上述程序进行图像分割的效果如下图所示，从上到下，从左到右依次为原图像、聚类数 K=2 的结果、聚类数 K=4 的结果、聚类数 K=8 的结果。

图 3.7: 图像分割，不同聚类数得到的结果

参考文献

[1] Jon Kleinberg , Éva Tardos **算法设计** 张立昂, 屈婉玲译 清华大学出版社 , 2007.

[2] 吴飞 , **人工智能导论：模型与算法** 高等教育出版社 , 2020.

[3] theSerein 【算法与数据结构】——并查集