


中国研究生创新实践系列大赛
“华为杯”第十八届中国研究生数学建模竞赛

学　　校	上海交通大学
参赛队号	**21102480004**
队员姓名	**1.** 王祎硕
	2. 赵子恒
	3. 江宁

中国研究生创新实践系列大赛

“华为杯”第十八届中国研究生

数学建模竞赛

题　　目　相关矩阵基于近似矩阵分解概率算法的计算与存储优化

摘　　要：

计算机视觉、相控阵雷达、声呐、射电天文、无线通信等领域的信号通常呈现为矩阵的形式，这一系列的矩阵间通常在某些维度存在一定的关联性，因此数学上可用相关矩阵组表示。本文通过挖掘矩阵内部及矩阵之间的关联性，通过建模及算法优化，降低了计算复杂度，且实现了廉价存储。

首先提出了一种“模方法”去评价不同矩阵的相关程度，并使用“聚类父子节点算法”将其聚类，将矩阵间具有相关关系的矩阵归类为相关矩阵组。利用相关矩阵组中的矩阵进行奇异值分解后右奇异向量的相关性，减少了后续对原始矩阵非必要的奇异值分解。“近似矩阵分解的概率算法—随机奇异值分解”实现了高效查找原始矩阵的低阶近似矩阵，对原矩阵的奇异值分解过程可以转化为对其相应的低阶近似矩阵进行奇异值分解，这极大的提高了计算速度。

其次为了降低矩阵求逆运算的复杂度，首先将 Strassen 算法与矩阵分治的思想相结合，提出了一种基于 Strassen 算法的矩阵求逆算法，降低了计算复杂度，经测试后能够将 1024 维矩阵的求逆运算时间降低 41.5%；接着抓住需求当中 Hermite 阵求逆这一点进行算法优化，提出了一种改进的 Strassen 求逆算法，即将 Strassen 求逆算法中的一些乘法运算替换为简单映射，进一步降低了计算复杂度；最后将 Coppersmith-Winograd 算法引入改进的 Strassen 算法中，替换掉其中的基于 Strassen 算法的矩阵乘法运算，极大地降低了求逆过程的运算复杂度。通过 1024 维矩阵测试后发现，最终使用 Coppersmith-Winograd 算法优化的改进 Strassen 求逆算法的运行时间相较于原始求逆算法降低了 46.9%。最后基于随机奇异值分解提出的“降维分块压缩算法”，实现了在基本还原数据原貌的同时，最大化的进行矩阵压缩。而后的解压过程，只需要通过简单的矩阵乘法及数据升维来实现。

关键字： 相关矩阵　模方法　聚类父子节点算法　随机奇异值分解　改进的 Strassen 求逆算法　降维分块压缩算法

目录

1. 问题重述

1.1 问题的背景

矩阵是一个按照长方阵列排列的复数或实数集合，是高等代数学中的常见工具，也常见于统计分析等应用数学学科中。在物理学中，矩阵于电路学、力学、光学和量子物理中都有应用；计算机科学中，三维动画制作也需要用到矩阵；计算机视觉、相控阵雷达等领域的信号通常呈现为矩阵的形式；视频信号中的单帧图像可视为一个矩阵。这些领域中应用的矩阵通常在某些维度存在一定的关联性，因此数学上可用相关矩阵组表示。矩阵的运算是数值分析领域的重要问题。随着实际需要中矩阵阵列的扩大，对矩阵常规处理算法对计算和存储的需求成倍增长, 从而对处理器件或算法的实现成本和功耗提出了巨大的挑战。对一些应用广泛而形式特殊的矩阵，例如稀疏矩阵和准对角矩阵等，有特定的快速运算算法，可以降低计算速度及存储空间。因此，充分挖掘矩阵内部及相关矩阵之间的关联性，实现低复杂度的计算和和廉价存储，具有重要意义。图1展示了对角矩阵的压缩存储思想。

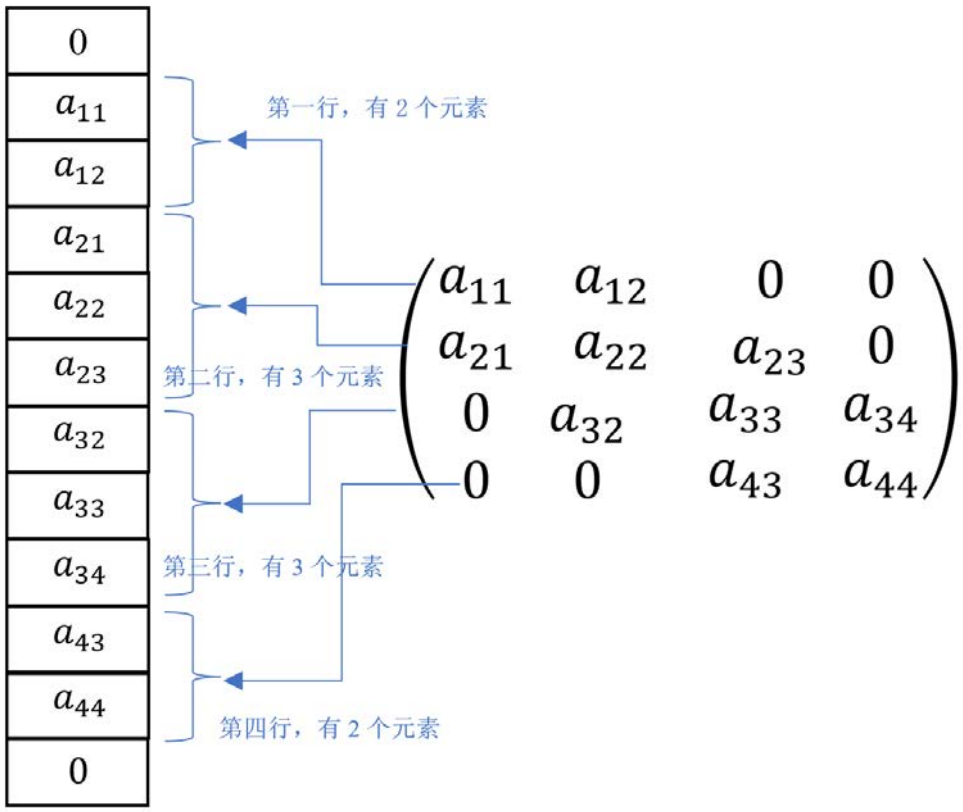

图 1: 对角矩阵的压缩存储

1.2 问题的提出

题目给出 data1—data6 共 6 个数据集，其内部的数据以矩阵的形式表达。要求充分挖掘矩阵内部及矩阵之间的相关性，通过建立相应的数学模型和算法，解决以下问题：

问题 1:

（1）基于数据集中给定的所有矩阵数据 **H**, 分析其数据间的关联性，设计一个近似分析模型 $\hat{\mathbf{V}} = f_1(\mathbf{H})$, 在暂不考虑矩阵数据 **H** 的压缩与解压缩情况下，及在满足 $\rho_{\min}(\mathbf{V}) \geq \rho_{th} = 0.99$ 的前提下，使得根据表格计算的总计算复杂度最低。

（2）基于数据集中给定的所有矩阵数据 **H**, 分析其数据间的关联性，设计一个近似分析模型 $\hat{\mathbf{W}} = f(\mathbf{H})$, 在暂不考虑矩阵数据 **H** 的压缩与解压缩情况下，及在满足 $\rho_{\min}(\mathbf{W}) \geq \rho_{th} = 0.99$ 的前提下，使得根据表格计算的总计算复杂度最低。

问题 2:

基于数据集中给定的所有矩阵数据 **H** 和 **W**, 分析其数据间的关联性，分别设计相应的压缩 $P_1(\cdot)$、$P_2(\cdot)$ 和解压缩 $G_1(\cdot)$、$G_2(\cdot)$ 模型，在满足误差 $err_{\mathbf{H}} \leq E_{th1} = -30dB$、$err_{\mathbf{W}} \leq E_{th2} = -30dB$ 的情况下，同时暂不考虑分析模型 $\hat{\mathbf{W}} = f(\mathbf{H})$ 对输出矩阵组 **W** 的影响，使得存储复杂度和压缩与解压缩的计算复杂度最低。计算复杂度时要考虑压缩和解压缩函数的所有运算过程。

问题 3:

基于数据集中给定的所有矩阵数据 **H**, 分析其数据间的关联性，在满足 $\rho_{\min}(\mathbf{W}) \geq \rho_{th} = 0.99$ 的情况下，设计一个方案，完成从矩阵输入信号 **H** 到近似矩阵输出信号 $\hat{\mathbf{W}}$ 的端到端流程, 且实现低复杂度计算和廉价存储。

2. 问题分析

题目给出 6 个数据集，对于其中数据的处理，常规算法处理和存储方式会造成计算和存储负担。要求分析矩阵关联性，进行建模和提出优化算法，在不同问题所要求的不同的限定条件下，达到低复杂度计算和廉价存储的目的。

2.1 问题 1 分析

（1）题目要求基于给定的矩阵数据 **H**, 设计一个近似分析模型 $\hat{\mathbf{V}} = f_1(\mathbf{H})$。对于常规算法而言，**V** 中的每一个 $\mathbf{V}_{j,k}$ 都是与之对应的 $\mathbf{H}_{j,k}$ 进行标准 SVD 分解后取 $\mathbf{H}_{j,k}$ 的前 **L** 个右奇异向量构成的矩阵。在设计 $\hat{\mathbf{V}} = f_1(\mathbf{H})$ 时，可以从以下两个大方向对上述过程进行改进：

(a) 降低 $\mathbf{H}_{j,k}$ 进行 SVD 分解的计算复杂度。通过挖掘矩阵内部的联系，尽可能地降低 $\mathbf{H}_{j,k}$ 进行 SVD 分解的复杂度。

(b) 存在相关关系的不同矩阵 $\mathbf{H}_{j,k}$ 之间，其 $\mathbf{V}_{j,k}$ 也会存在相关关系。寻找 $\mathbf{H}_{j,k}$ 的相关

矩阵组，挖掘相关关系，进一步尽可能地减少非必要地对 $\mathbf{H}_{j,k}$ 进行 SVD 分解的过程。

同时，该小问还要求 $\rho_{\min}(\mathbf{V}) \geq \rho_{th} = 0.99$，这一点限定了我们进行上面两个大方向时要考虑算法对原始数据损耗率，不同矩阵相关程度的下限。

（2）题目要求基于给定的矩阵数据 $\mathbf{H}$, 设计一个近似分析模型 $\hat{\mathbf{W}} = f(\mathbf{H})$。在（1）的基础上考虑 $\hat{\mathbf{W}} = f_2(\hat{\mathbf{V}})$, 故 $\hat{\mathbf{W}} = f(\mathbf{H}) = f_2(f_1(\mathbf{H}))$。从上面这个思路出发，我们提高计算速率，主要分为以下两个大方向：

(a)$\mathbf{W}_k = \mathbf{V}_k(\mathbf{V}_k{}^H\mathbf{V}_k + \sigma^2\mathbf{I})^{-1}$，降低其中矩阵乘法的复杂度，降低矩阵求逆的复杂度。

(b)$\mathbf{W}_k = \mathbf{V}_k(\mathbf{V}_k{}^H\mathbf{V}_k + \sigma^2\mathbf{I})^{-1}$ 转化为 $(\mathbf{V}_k{}^H\mathbf{V}_k + \sigma^2\mathbf{I})\mathbf{W}_k{}^H = \mathbf{V}_k{}^H$ 来计算，降低解线性方程组的复杂度。

同时，该小问还要求 $\rho_{\min}(\mathbf{W}) \geq \rho_{th} = 0.99$，由于 $\hat{\mathbf{V}}$ 作为得到 $\hat{\mathbf{W}}$ 的中间量，故需要进一步需要考虑（1）中算法对原始数据的损耗率及限定了（1）过程中不同矩阵相关程度的下限。

2.2 问题 2 分析

基于给定的所有矩阵数据 $\mathbf{H}$ 和 $\mathbf{W}$, 注意到，其中的 $\mathbf{W}$ 不是（1）中经过近似分析模型近似后的 $\hat{\mathbf{W}}$。从 $\mathbf{H}$ 和 $\mathbf{W}$ 出发，经过压缩和解压操作，在满足误差 $err_{\mathbf{H}} \leq E_{th1} = -30dB$、$err_{\mathbf{W}} \leq E_{th2} = -30dB$ 的情况下，使得存储复杂度和压缩与解压缩的计算复杂度最低。主要从下面方向考虑：

(a) 改变矩阵中每个元素的底层存储方式。

(b) 分析数据间的关联性，考虑从矩阵内部及不同矩阵之间的联系出发，使得经过压缩和解压缩后能保留原始数据大部分（满足精度需求）的样貌，且尽可能地提高计算速度。

2.3 问题 3 分析

对于问题 3，首先对给定所有的矩阵数据 $\mathbf{H}$ 进行问题 2 中提出的压缩与解压缩操作，解压后为 $\mathbf{H}_r$。对 $\mathbf{H}_r$ 采取问题 1 中的第一小问采用的模型和算法来提高计算速度，得到 $\mathbf{V}_r$。对 $\mathbf{V}_r$ 采用问题 1 第二小问采用的改进算法来提高计算速度，进而得到 $\mathbf{W}_r$。对 $\mathbf{W}_r$ 进行问题 2 中提出的压缩与解压缩操作，得到最终的 $\hat{\mathbf{W}}$。在满足题目所要求的 $\rho_{\min}(\mathbf{W}) \geq \rho_{th} = 0.99$ 的前提下，对上述的一系列过程做限定。

3. 模型的假设

- 假设仅考虑同一行块内部的 $\mathbf{K}$ 个矩阵间的相关性。
- 假设大矩阵进行 SVD 分解后，占据所有奇异值之和大部分的前 n 个奇异值所对应的分解能解释该矩阵大部分的作用。
- 假设高度相似的 $\mathbf{H}_{j,k}$ 具有相似的 $\mathbf{V}_{j,k}$。

4. 符号定义与说明

符号	意义
T	下界
$\underline{\mathbf{A}}$	线性映射
$\mathbf{H}$	给定的矩阵数据
$\mathbf{V}$	对 $\mathbf{H}$ 进行 SVD 分解得到的 $\mathbf{V}$
$\mathbf{W}$	通过 $\mathbf{V}$ 计算得到的矩阵
$\mathcal{W}$	$\mathbf{W}$ 的近似分析模型
$\mathcal{V}$	$\mathbf{V}$ 的近似分析模型
ρ	建模精度
r	相关系数
R	矩阵相关系数

5. 模型建立与求解

5.1 模型一建立与求解

5.2 模型一建立与求解

5.2.1 模方法

由高等代数知识，对于复数域上一个维数为 n 的线性空间 V 与一个维数为 m 的线性空间 V'，分别取定一个标准正交基 $\gamma_1, \gamma_2, ...\gamma_n$ 与 $\eta_1, \eta_2, ...\eta_m$，则 V 到 V' 的线性映射 $\underline{\mathbf{A}}$ 与它在上述基下的矩阵 $\mathbf{A}$ 是线性空间 $\mathbb{C}_{m\times n}$ 到线性空间 $Hom(V, V')$ 的同构映射，从而

$$Hom(V, V') \cong \mathbb{C}_{m\times n} \tag{1}$$

。因此对于任意矩阵 $\mathbf{A} \in \mathbb{C}_{\mathbf{m}\times\mathbf{n}}$，其与一个线性映射 $\underline{\mathbf{A}} \in Hom(V, V')$ 相对应。

模方法用来衡量两个矩阵的相关性。对于矩阵 $\mathbf{B} = (\delta_1, \delta_2, ...\delta_n), \mathbf{C} = (\zeta_1, \zeta_2, ...\zeta_n) \in \mathbb{C}_{\mathbf{m}\times\mathbf{n}}$，先求出矩阵列向量的二范数生成新的矩阵 $\mathbf{B_1} = (\|\delta_1\|_2, \|\delta_2\|_2, ...\|\delta_n\|_2), \mathbf{C_1} = (\|\zeta_1\|_2, \|\zeta_2\|_2, ...\|\zeta_n\|_2) \in \mathbb{C}_{\mathbf{1}\times\mathbf{n}}$，再计算 $\mathbf{B_1'}$ 与 $\mathbf{B_2'}$ 的相关系数：

$$r(\mathbf{B_1'}, \mathbf{B_2'}) = \frac{\mathbf{Cov}(\mathbf{B_1'}, \mathbf{B_2'})}{\sqrt{\mathbf{Var}[\mathbf{B_1'}]\mathbf{Var}[\mathbf{B_2'}]}} \tag{2}$$

，作为 **B** 与 **C** 的相关系数 $R(\mathbf{B},\mathbf{C})$。

对于上述矩阵 **B** 与 **C**，取定上述线性空间 V 与 V' 的标准正交基，则 **B** 与 **C** 可以对应两个线性空间 $Hom(V,V')$ 中的线性变换 $\underline{\mathbf{B}}$ 与 $\underline{\mathbf{C}}$，δ_i 为 $\underline{\mathbf{B}}\gamma_i$ 在 V' 的基 $\eta_1,\eta_2,...\eta_m$ 下的坐标，ζ_i 为 $\underline{\mathbf{C}}\gamma_i$ 在 V' 的基 $\eta_1,\eta_2,...\eta_m$ 下的坐标。如果定义线性空间 V' 的内积为标准内积，那么，V' 成为了一个装备了标准内积的复内积空间，$\|\delta_i\|_2$ 即为 V 中的基向量 γ_i 经过线性映射 $\underline{\mathbf{B}}$ 的作用在 V' 下向量的长度，$\|\zeta_i\|_2$ 即为 V 中的基向量 γ_i 经过线性映射 $\underline{\mathbf{C}}$ 的作用在 V' 下向量的长度。由于线性映射由它在 V 的一个基下的像所唯一决定，所以，$R(\mathbf{B},\mathbf{C})$ 的本质是线性映射 $\underline{\mathbf{B}},\underline{\mathbf{C}}$ 将 V 中的基映射下像长度的相关性，即，不考虑正交变换（旋转变换，镜像反射）之后的相关性（如图2）。

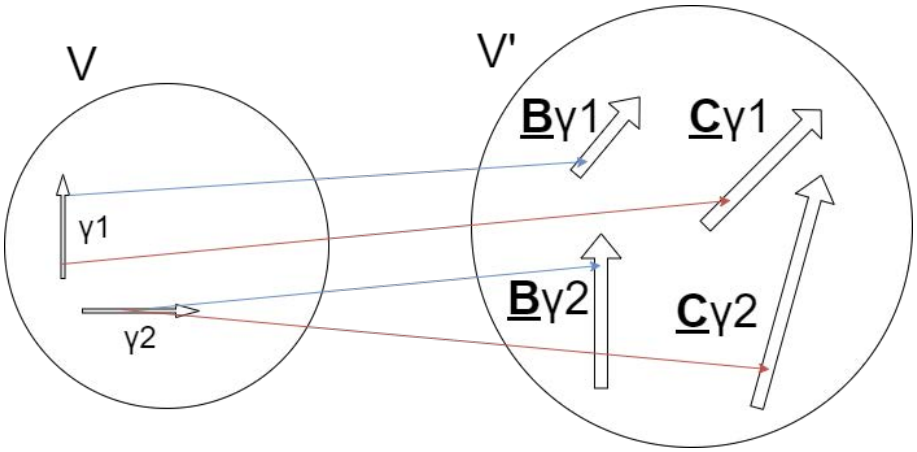

图 2: 模方法图示

5.2.2 聚类父子节点法

由上述模方法，可以求得题目所给 **H** 矩阵每一行中 384 个矩阵 $\mathbf{H_{j,k}}$ 之间的相关系数，筛选出相关系数大于 0.95 的矩阵关系。以 Data1 数据集 **H** 矩阵的第一行为例，表1给出了相关性显著的矩阵编号。可以看出，第 1 个矩阵与第 2 个矩阵具有高度相关性，第 2 个矩阵与第 3 个矩阵具有高度相关性，可是，第 1 个矩阵与第 3 个矩阵不具有高度相关性，由此可以知道，高度相关性不具备传递性，这种关系不是一个等价关系。

综上所述，可以采用聚类父子节点法。聚类父子节点旨在将所有高度相关的矩阵进行聚类，那么这些矩阵对应的 $\mathbf{V_{j,k}}$ 就可以使用其父节点对应的 $\mathbf{V_{p,q}}$ 来代替，从而达到减少计算复杂度的目的。详细见算法 1 和图3。

表 1　相关矩阵表

矩阵的编号	与之相关矩阵的编号
1	1
1	2
2	2
2	3
2	4
3	3
3	4
3	5
4	4
4	5

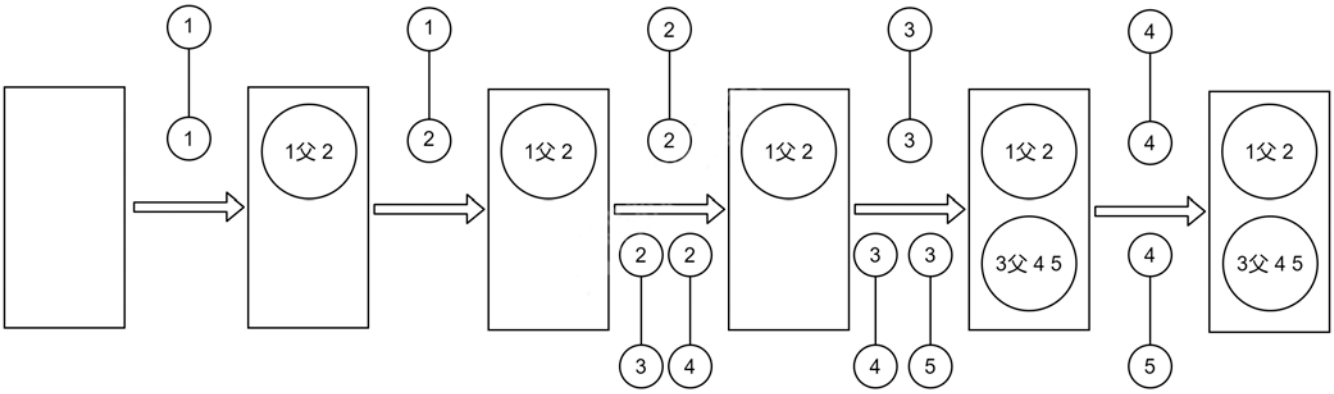

图 3: 表 1 的聚类步骤

5.2.3 近似矩阵分解的概率算法—随机奇异值分解

低秩矩阵逼近：

标准矩阵分解包括旋转 QR 分解、特征值分解和奇异值分解，所有这些都揭示了矩阵

Algorithm 1 聚类父子节点法

Require: 相关矩阵表

Ensure: 容器

1: 容器生产第一个相关类，将 1 加入其中，1 为该相关类的父。定义一个跳跃指针 k, 初始化为 1，然后开始遍历相关矩阵表，每次遍历执行步骤 $2-5$；

2: 比较遍历点与跳跃指针的大小。如果前者小，当前遍历点前进一个点。否则，执行步骤 3；

3: 如果左和右相等，且左在容器中，查找左和右相等，且等于当前左加 1 的位置，令跳跃指针等于该位置。否则，执行步骤 4；

4: 如果左和右相等，且左不在容器中，创造一个新的相关类，将左加入新生的子容器中，左成为新相关类的父。否则执行步骤 5。；

5: 如果左和右不等，且右不在容器中，将右加入最新生成的子容器中，右成为其中父的子；

6: **return** 容器。

的 (数值) 范围。这些因子分解的截断版本通常用于表示给定矩阵的低秩近似:

$$\underset{m\times n}{\mathbf{A}} \approx \underset{m\times k}{\mathbf{B}} \times \underset{k\times n}{\mathbf{C}}, \tag{3}$$

内维 k 有时被称为矩阵的数值秩。当数值秩比 m 或 n 小得多时，像上式这样的因式分解可以廉价地存储矩阵，并快速与向量或其他矩阵相乘。因此找到合理的矩阵低阶近似形式是至关重要的 [1]。

矩阵近似框架：

计算给定矩阵的低秩近似可以自然地分成两个计算阶段。首先是构造一个低维的子空间，捕捉矩阵的作用。第二种是将矩阵限制在子空间，然后计算该简约矩阵的标准因子分解 (QR、SVD 等)。具体步骤细分如下。

Step A：计算输入矩阵 **A** 的近似基。换句话说，我们需要一个矩阵 **Q**，使得，

$$\mathbf{Q}\text{具有正交列且}\mathbf{A}\approx\mathbf{Q}\mathbf{Q}^{*}\mathbf{A}\square$$

我们希望 **Q** 包含尽可能少的列，但更重要的是实现输入矩阵的精确近似。

Step B：给出由 Step A 得出的 **Q** (k 列)，利用 **Q** 来帮助 **A** 的标准因子分解过程（如 QR, SVD 等）。

对于 **A** 的奇异值分解过程，我们希望得到酉矩阵 **U**,**V** 和非负对角阵 Σ，使得 $\mathbf{A}=\mathbf{U}\Sigma\mathbf{V}^{*}$, 这可以利 **Q** 通过以下步骤近似得到。

Step1：构建矩阵 $\mathbf{B}=\mathbf{Q}^{*}\mathbf{A}$，进一步得到 **A** 的低秩近似分解 $\mathbf{A}\approx\mathbf{Q}\mathbf{B}$;

Step2：对矩阵 **B** 进行 SVD 分解：$\mathbf{B}=\widetilde{\mathbf{U}}\Sigma\mathbf{V}^*$；

Step3: $\mathbf{U}=\mathbf{Q}\widetilde{\mathbf{U}}$。

当 **Q** 的列数很少时，这个过程是有效的，因为我们可以很容易地简约矩阵 **B** 并快速计算它的奇异值分解 (图4形象地描绘了这一快速计算过程)。因此接下来的任务是去寻找 **Q**。

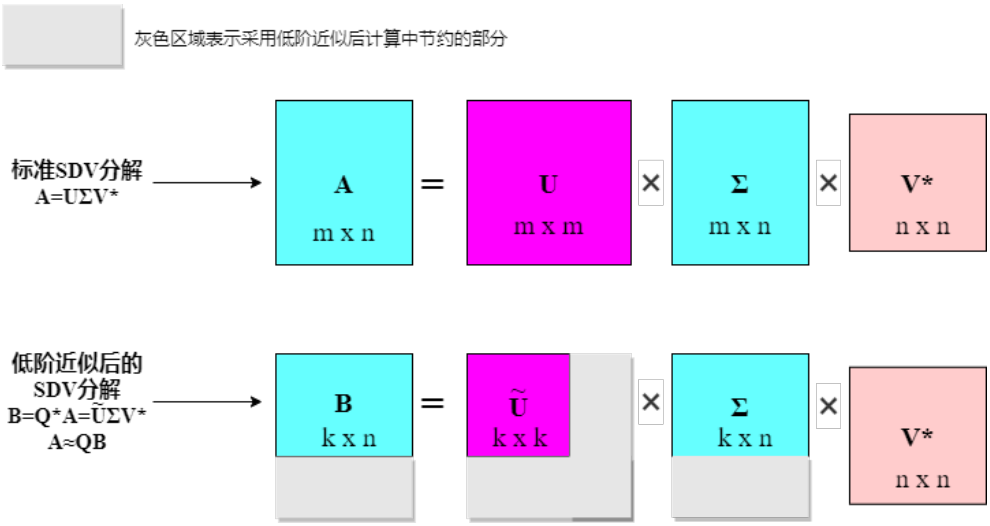

图 4: 简化计算过程示意图

随机化算法：

问题表述：

产生低秩矩阵逼近的基本挑战是固定精度逼近问题。假设给我们一个矩阵 **A** 和一个正的误差限 ε。我们寻找一个具有 k 个正交列的矩阵 **Q**，使得，

$$||\mathbf{A}-\mathbf{Q}\mathbf{Q}^*\mathbf{A}|| \leq \varepsilon, \tag{4}$$

这里 $||\cdot||$ 代表 l_2 算子范数，**Q** 的范围是一个 k 维子空间，它捕捉了 **A** 的大部分作用，且我们希望 k 尽可能的小。

奇异值分解为固定精度问题提供了一个最佳解答。取 δ_j 代表 A 的第 j 个最大奇异值，对于每一个 $j \geq 0$,

$$\min_{rank(\mathbf{X})\leq j} ||\mathbf{A}-\mathbf{X}|| = \sigma_{j+1}, \tag{5}$$

构造极小值地一个方法是选择 $\mathbf{X}=\mathbf{Q}\mathbf{Q}^*\mathbf{A}$, 这里 **Q** 是 **A** 的 k 个主左奇异向量。因此，其中 (4) 成立的最小秩 k 等于超过误差限 ε 的 **A** 的奇异值的数量。

为了简化算法的开发，可以方便地假设期望的秩 k 是预先指定的。我们称由此产生的问题为固定秩近似问题。给定矩阵 **A**、目标秩 k 和过采样参数 p，我们试图构造具有 $k+p$

个正交列的矩阵 **Q**，使得，

$$||\mathbf{A}-\mathbf{Q}\mathbf{Q}^{*}\mathbf{A}|| \approx \min_{rank(\mathbf{X})\leq k} ||\mathbf{A}-\mathbf{X}|| \tag{6}$$

虽然存在一个最小 Q 来解决 p = 0 的固定秩问题，但是使用少量附加列的机会提供了一种灵活性，这对于我们讨论的计算方法的有效性是至关重要的。固定秩问题的算法可以适用于解决固定精度问题，其间的联系是基于这样的观察，即我们可以增量地建立基矩阵 **Q**，并且在计算的任何一点，我们都可以廉价地估计 $||\mathbf{A}-\mathbf{Q}\mathbf{Q}^{*}\mathbf{A}||$ 的残留误差，因此进一步的我们采取下面的方法来得到 **Q**。

增量方式构建 Q：

假定 **A** 是一个 $m \times n$ 的矩阵，ε 是计算容差。我们寻找一个整数 l 和一个 $m \times l$ 的正交矩阵 **Q**，使得，

$$||(\mathbf{I}-\mathbf{Q}^{(l)}(\mathbf{Q}^{(l)})^{*})\mathbf{A}|| \leq \varepsilon \tag{7}$$

基矩阵 **Q** 尺寸 l 通常略大于能实现此 ε 的尺寸最小的基矩阵的尺寸 l。

从空的基矩阵 $\mathbf{Q}^{(0)}$ 开始，以下的方案能产生一个捕捉到 **A** 大部分作用的正交基矩阵:

for i=1,2,3,...

抽取 1 个维度为 $n \times 1$ 的高斯随机矢量 $\mathbf{w}^{(i)}$，令 $\mathbf{y}^{(i)}$=$\mathbf{A}\mathbf{w}^{(i)}$;

计算 $\tilde{\mathbf{q}}^{(i)} = (\mathbf{I}-\mathbf{Q}^{(i-1)}(\mathbf{Q}^{(i-1)})^{*})\mathbf{y}^{(i)}$;

标准化 $\tilde{\mathbf{q}} = \tilde{\mathbf{q}}^{(i)}/||\tilde{\mathbf{q}}^{(i)}||$, 构建 $\mathbf{Q}^{(i)} = [\mathbf{Q}^{(i-1)}q^{(i)}]$。

end

关于上述方案的矩阵近似的精确的近似误差 $||(\mathbf{I}-\mathbf{Q}\mathbf{Q}^{*})\mathbf{A}||$ 可由下式衡量：

$$||(\mathbf{I}-\mathbf{Q}\mathbf{Q}^{*})\mathbf{A}|| \leq 10\sqrt{\frac{2}{\pi}} \max_{i=1,...,r} ||(\mathbf{I}-\mathbf{Q}\mathbf{Q}^{*})\mathbf{A}\mathbf{w}^{(\mathbf{i})}|| \tag{8}$$

其中 $\mathbf{w}^{(\mathbf{i})}$:i=1,2,...,r。

当我们观测 r 个连续的矩阵范数小于 $\varepsilon/(10\sqrt{2/\pi})$ 的 $\tilde{\mathbf{q}}^{(i)}$ 时，跳出增量方式构建基矩阵 **Q** 的循环过程。本小节算法代码见附录。

5.2.4 改进的 Strassen 算法和 Coppersmith-Winograd 算法

参阅历史文献可知，矩阵求逆的计算复杂度与矩阵乘法的计算复杂度在均使用 $O(\bullet)$ 的方式进行度量时是相同的，因此，降低矩阵乘法的计算复杂度可以有效地降低矩阵求逆的计算复杂度。

设 **A** 和 **B** 是两个 $n \times n$ 的矩阵，他们的乘积 C 同样是一个 $n \times n$ 的矩阵，**A** 和 **B** 的乘积矩阵 **C** 中的元素 c_{ij} 定义为:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

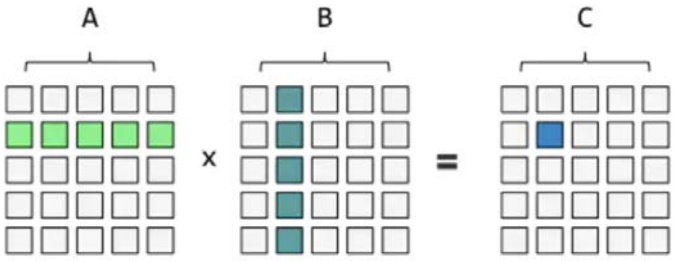

图 5: A 和 B 的矩阵乘法

按照此定义来看，如图5所示计算 **A** 和 **B** 的矩阵乘法，至少需要 n 次乘法与 $n-1$ 次加法，由此可知计算矩阵 **C** 的乘法时间为 $O(n^3)$。

最早的矩阵乘法优化算法，是由德国数学家 Volker Strassen 于 1969 年提出并以其名字命名的 Strassen 算法 [2]。Strassen 矩阵乘法算法采用类似于大数乘法中的分治技术，将计算 2 个 n 阶矩阵乘积所需的时间改进到 $O(n^{log_2 7}) \approx O(n^{2.81})$，但是这个算法有一个前提条件，必须是两个 $n \times n$ 的矩阵相乘，且 n 必须为 2 的幂。这样我们每次都可以把大矩阵分割为 4 个小矩阵，如下所示：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

由上式可知：

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

如果 $n = 2$，两个 2×2 阶矩阵乘法需要 8 次乘法和 4 次加法。当子矩阵的阶数大于 2 时，为求两个子矩阵的乘积，可以继续将矩阵分块，直到子矩阵阶数降为 2，这就是分治降阶的递归算法。按照这个算法，计算 2 个 n 阶矩阵的乘积转化为计算 8 个 $n/2$ 阶矩阵的乘积和 4 个 $n/2$ 阶矩阵的加法。而 2 个 $n/2$ 阶矩阵的加法显然可以在 $O(n^2)$ 时间内完成，由此可知上述算法的时间耗费 $T(n) = O(n^3)$。但是这个方法并没有减少计算所需要的时间，这主要是因为该方法并没有减少矩阵的乘法次数。

由此 Strassen 提出了对于 2 阶矩阵的乘积方法，仅仅使用了如下的 7 次乘法和 18 次加法：

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
\qquad
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

图 6: Strassen 算法核心思想

这样做完以后，它的时间复杂度 $T(n) = O\left(n^{\log_2 7}\right) \approx O\left(n^{2.81}\right)$。

可以使用下面的三维图来更好的理解 Strassen 矩阵算法中高维度的分治算法。

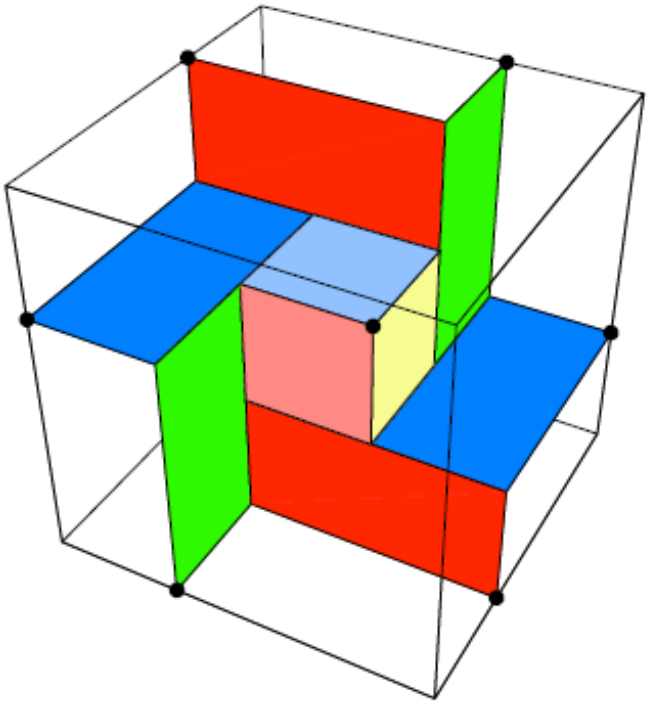

图 7: 三维空间中的分治算法

假设

$$A^{-1}=\begin{bmatrix}[A_{11}]_{n\times n} & [A_{12}]_{n\times m}\\ [A_{21}]_{m\times n} & [A_{22}]_{m\times m}\end{bmatrix}^{-1}=\begin{bmatrix}[C_{11}]_{n\times n} & [C_{12}]_{n\times m}\\ [C_{21}]_{m\times n} & [C_{22}]_{m\times m}\end{bmatrix}$$

那么根据矩阵分块求逆的原理有：

$$C_{11}=\left(A_{11}-A_{12}\times {A_{22}}^{-1}\times A_{21}\right)^{-1}$$

$$C_{12}=-C_{11}\times A_{12}\times {A_{22}}^{-1}$$

$$C_{21}=-{A_{22}}^{-1}\times A_{21}\times C_{11}$$

$$C_{22}=-{A_{12}}^{-1}\times A_{11}\times C_{12}$$

结合上式将 Strassen 算法应用到求逆运算中有如下公式 (详细的数学推导见参考文献[2])：

$$\begin{aligned}
M_1&={A_{11}}^{-1}\\
M_2&=A_{21}\times M_1\\
M_3&=M_1\times A_{12}\\
M_4&=A_{21}\times M_3\\
M_5&=M_4-A_{22}\\
M_6&={M_5}^{-1}\\
C_{12}&=M_3\times M_6\\
C_{21}&=M_6\times M_2\\
M_7&=M_3\times C_{21}\\
C_{11}&=M_1-M_7\\
C_{22}&=-M_6
\end{aligned}\tag{9}$$

在上式中，对于 $N\times N$ 阶矩阵 A 利用一次分块求逆的总的运算量为：

$$T^1(N)=T(m)+T(N)+13m^2n+11mn^2-4mn+3m^2$$

对于 $n>1,m>1$，Strassen 矩阵求逆算法也是利用递归实现的，但因为 Strassen 算法减少了矩阵的乘法次数，所以相比直接分块的常规算法运算量有明显降低。

同时观察到题目所给的 $\left(\mathbf{V}_k^H\mathbf{V}_k+\sigma^2\mathbf{I}\right)$ 矩阵为一个 Hermite 矩阵，那么 $A_{11},A_{22},{A_{22}}^{-1}$ 均为 Hermite 矩阵，且 A_{12}^H=A_{21}，带入到式9中得到 $M_3=M_2^H,C_21=C_12^H$，根据 Hermite 矩阵的共轭对称特性，可将式9变为：

$$
\begin{aligned}
M_1 &= {A_{11}}^{-1} \\
M_2 &= A_{21} \times M_1 \\
M_3 &= M_2^H \\
M_4 &= A_{21} \times M_3 \\
M_5 &= M_4 - A_{22} \\
M_6 &= {M_5}^{-1} \\
C_{12} &= M_3 \times M_6 \\
C_{21} &= C_{12}^H \\
M_7 &= M_2^H \times C_{21} \\
C_{11} &= M_1 - M_7 \\
C_{22} &= -M_6
\end{aligned}
\tag{10}
$$

根据式9，矩阵 A 利用一次分块求逆的总的运算量为：

$$
T^1(N) = T(m) + T(N) + 8m^2n + 8mn^2 + mn
$$

和矩阵直接分块求逆算法相比，新的求逆算法虽然增加了一些加减运算，但是乘法次数降低，对于维数较高的矩阵，有大量的复矩阵运算，其中乘法消耗的运算量将远远大于加减法，而且这个运算量随着矩阵维数增加将有显著增加，因此新算法对于乘法次数的降低将显著改善求逆运算，所以在运算量和算法复杂度上都有明显的降低。下面对两种方法——用于 Hermite 矩阵的改进的基于 Strassen 算法的求逆方法 (以下简称为改进的 Strassen 算法) 和未改进的基于 Strassen 算法的求逆方法 (以下简称 Strassen 算法) 进行研究。

使用 Strassen 算法和传统求逆方法分别去测试计算题目中 $\left(\mathbf{V}_k^H\mathbf{V}_k + \sigma^2\mathbf{I}\right)$ 矩阵的逆矩阵所需时间，同时为了测试算法计算不同维度矩阵的性能，对所给矩阵进行了一定程度的截取和增广，两种算法的矩阵维度和计算耗时如下图所示：

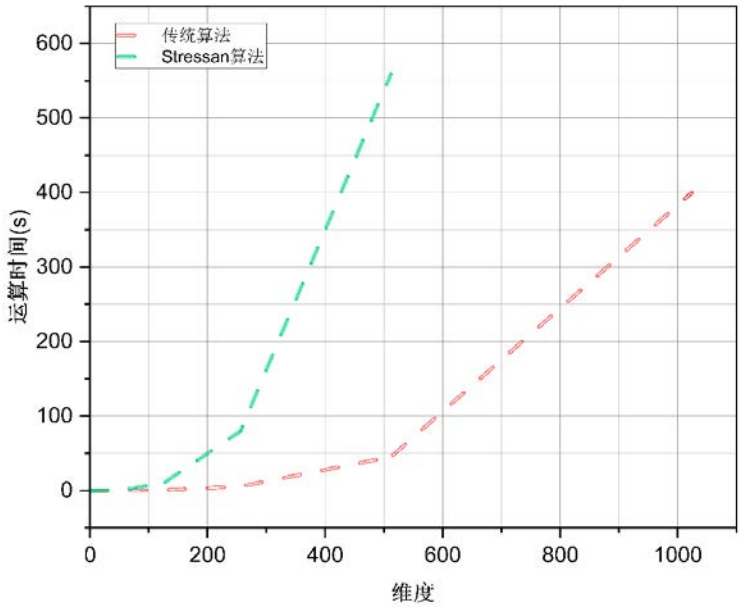

图 8: 不同下界 T 的 Strassen 算法运算时间

可以看到使用 Strassen 算法时，耗时不但没有减少反而剧烈增加，在 $n = 512$ 时计算时间就已无法忍受。在仔细研究后发现，采用 Strassen 算法作递归运算，需要创建大量的动态二维数组，其中分配堆内存空间将占用大量计算时间，从而掩盖了 Strassen 算法的优势。于是对 Strassen 算法做出改进，设定一个界限 T。当 $n < T$ 界限时，使用普通算法计算矩阵，而不继续分治递归。改进后算法优势明显，运算时间大幅下降。之后，针对不同大小的界限进行试验，试验结果如图9所示。经初步试验中发现，当数据规模较小时，尤其是当矩阵维度小于 256 时，下界 S 的大小差别不大，但随着矩阵维度的增加至 512 以上时，带有下界的 Strassen 算法耗时明显低于传统算法，且最优的界限值在 $32 - 128$ 之间，当维度为 1024 维时，下界为 64 的 Strassen 算法求逆时间相较于传统算法降低了 41.5%。

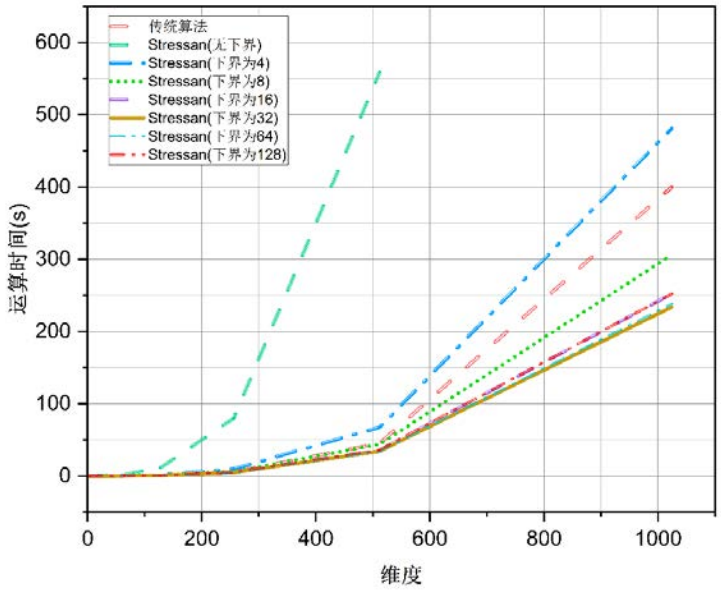

图 9: 不同下界 T 的 Strassen 算法运算时间

接下来选取相同的界限 T 对 Strassen 算法和改进 Strassen 算法的耗时进行研究，如图10所示。可以看到在高维度下改进 Strassen 算法的耗时进一步降低，改进的 Strassen 算法在计算 1024 维矩阵逆矩阵时耗时进一步降低了 3.2%。

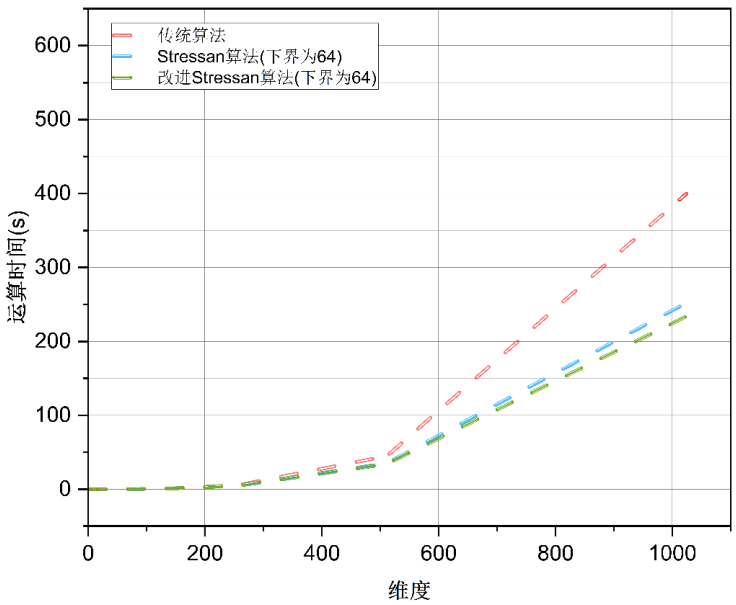

图 10: 相同下界 T 下 Strassen 算法与改进 Strassen 算法对比

将改进 Strassen 算法取不同下界计算不同维度的矩阵所耗费的时间绘制于一张图上可以很清晰地看出界限 T 在不同维度下对计算时间的影响。如图11所示，随着下界的降低

和维度的增大，算法的耗时出现了一个难以接受的峰值，其原因在前一部分也已分析过；在维度较低的时候下界的设置对运算时间几乎没有影响，因为传统算法和改进 Strassen 算法在低维度下的运行效率十分相近；但是随着维度的上升可以发现对于特定的维度存在一个最优的下界。计算三维图的投影得到图11，可以看到在维度接近 800 的时候使用 128 作为下界可以得到意想不到的效果，因此针对不同的计算情况设置不同的下界是很有必要的。

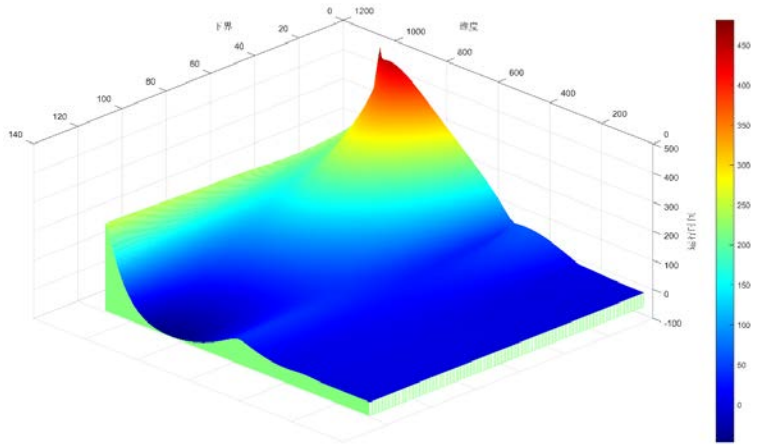

图 11: 改进 Strassen 算法不同维度与下界对运行时间的影响（三维图）

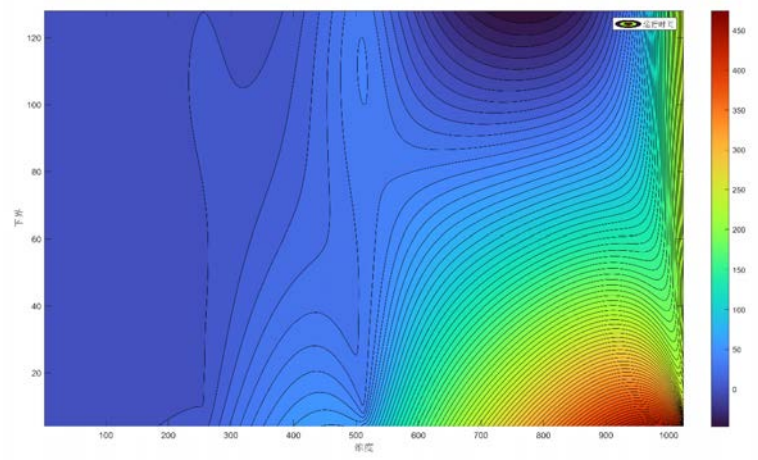

图 12: 改进 Strassen 算法不同维度与下界对运行时间的影响（投影）

更深层次的去考虑 Strassen 算法的底层运算，如下图中左图所示，在我们之前的程序中处理器需要计算完 **M** 矩阵后才会去计算 **C** 矩阵，在计算 **C** 矩阵时先前计算得到的 7 个 **M** 矩阵仍然占用大量内存空间，需要 **C** 矩阵运算完毕之后才会去释放 **M** 矩阵所占用的内存空间，且 **M** 矩阵运算期间没有任何其他运算进行，这种算法的运行效率是较低的。考

虑在此基础上进行一些改进，如下图右图所示，将一些 **C** 矩阵的加减运算放置到 **M** 矩阵计算时运行，**C** 矩阵的运算结束后就可以释放掉不再被需要的 **M** 矩阵的存储空间，虽然这样可能会增加一些加减法运算，但是并行运算的效率提升和内存的占用降低带来的的性能提升要远高于传统运算方法。

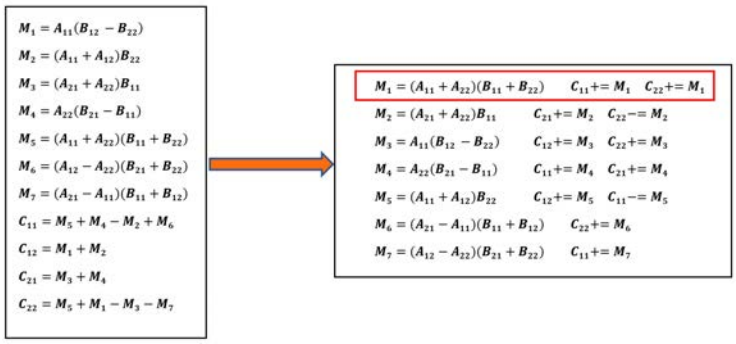

图 13: Strassen 算法运行时的底层改进示意图

更进一步的，我们可以写出这种改进算法的通用格式：

$$
\begin{aligned}
&M = (X + \delta Y)\,(V + \varepsilon W) \\
&C+ = \gamma_0 M \\
&D+ = \gamma_1 M \\
&\gamma_0,\ \gamma_1, \delta, \varepsilon \in \{-1, 0, 1\}
\end{aligned}
\tag{11}
$$

下面流程图14更加详细的讲述改进算法是如何提升运行效率的，从图中可以看到，相较于传统 Strassen 算法，改进的 Strassen 算法在各层循环中均将计算分为两块进行，将 **C** 矩阵的加减运算进一步划分为两部分，虽然一定程度上增加了处理器的运算负担，但是大大降低了运算过程中存储空间的占用。若以比特为单位计算，对于复数矩阵中单个复数元素，其实部和虚部均采用 32 比特单精度浮点表示，可令运算过程中存储器的峰值占用从 $7 \times 64N^2$ bit 降至 $64N^2$ bit。

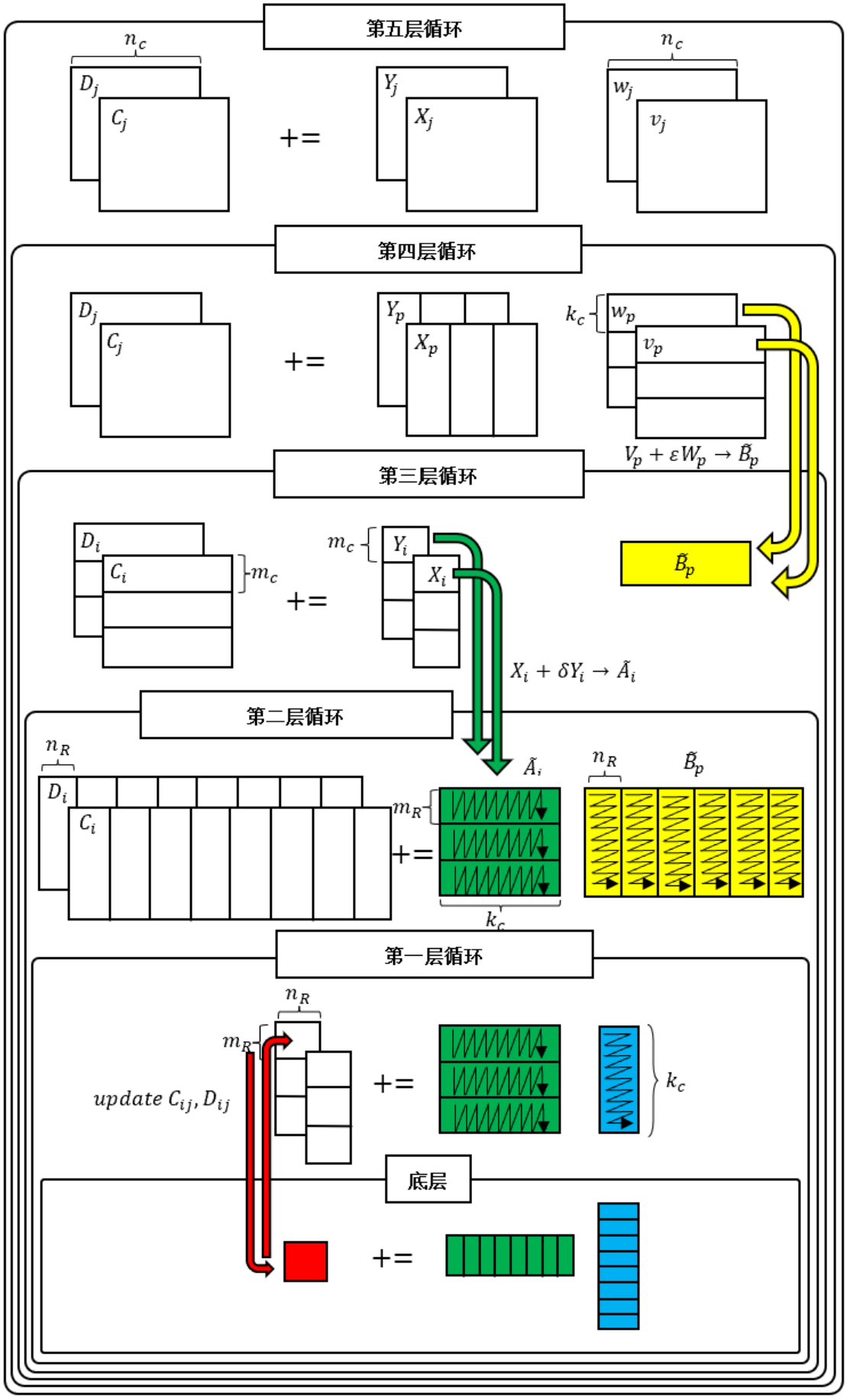
第五层循环
n_c
D_j
C_j
+=
Y_j
X_j
n_c
w_j
v_j
第四层循环
D_j
C_j
+=
Y_p
X_p
k_c
w_p
v_p
$V_p + \varepsilon W_p \to \tilde{B}_p$
第三层循环
D_i
C_i
m_c
+=
m_c
Y_i
X_i
$\tilde{B}_p$
$X_i + \delta Y_i \to \tilde{A}_i$
第二层循环
n_R
D_i
C_i
+=
m_R
$\tilde{A}_i$
k_c
n_R
$\tilde{B}_p$
第一层循环
n_R
m_R
$update\ C_{ij}, D_{ij}$
+=
k_c
底层
+=

图 14: Strassen 算法的改进运算流程图

在 Strassen 算法提出之后很多年，矩阵乘法的复杂度被 Pan[3] 降低至了 $O(n^2.494)$，仅仅 6 年后另一种时间复杂度更低的算法被 Coppersmith 和 Winograd 提出，这种算法也被称为 Coppersmith-Winograd 算法 [4]，该算法将矩阵乘法的运算复杂度降低到了 $O(n^2.376)$，Coppersmith-Winograd 算法在 Strassen 算法的思想上进行了改进，将 2 阶矩阵的乘法运算简化为如下的 7 次乘法运算和 15 次加减法运算:

$$
\begin{array}{llll}
S_1 = A_{21} + A_{22} & T_1 = B_{21} - B_{11} & M_1 = A_{11}B_{11} & M_5 = S_1T_1 \\
S_2 = S_1 - A_{11} & T_2 = B_{22} - T_1 & M_2 = A_{12}B_{21} & M_6 = S_2T_2 \\
S_3 = A_{11} - A_{21} & T_3 = B_{22} - B_{12} & M_3 = S_4B_{22} & M_7 = S_3T_3 \\
S_4 = A_{12} - S_2 & T_4 = T_2 - B_{21} & M_4 = A_{22}T_4 &
\end{array}
$$

$$
\begin{array}{lll}
U_1 = M_1 + M_2 & U_5 = U_4 + M_3 & C_{11} = U_1 \\
U_2 = M_1 + M_6 & U_6 = U_3 - M_4 & C_{12} = U_5 \\
U_3 = U_2 + M_7 & U_7 = U_3 + M_5 & C_{21} = U_6 \\
U_4 = U_2 + M_5 & & C_{22} = U_7
\end{array}
$$

图 15: Coppersmith-Winograd 算法的核心思想

使用 Coppersmith-Winograd 算法去测试计算题目中 $\left(\mathbf{V}_k^H\mathbf{V}_k + \sigma^2\mathbf{I}\right)$ 矩阵求逆过程中的乘法所需时间，同时为了测试算法计算不同维度矩阵的性能，对所给矩阵进行了一定程度的截取和增广，两种算法的矩阵维度和计算耗时如图16所示。可以看到和 Strassen 算法一样，若不设置界限 T 限耗时，在 $n = 512$ 时计算时间就已无法忍受。同样对 Coppersmith-Winograd 算法做出改进，设定一个界限 T。当 $n < T$ 界限时，使用普通算法计算矩阵，而不继续递归。改进后算法优势明显，运算时间大幅下降。之后，针对不同大小的界限进行试验。经初步试验中发现，当数据规模较小时，尤其是当矩阵维度小于 256 时，下界 S 的大小差别不大，但随着矩阵维度的增加至 512 以上时，带有下界的 Coppersmith-Winograd 算法耗时明显低于传统算法，且最优的界限值在 $32 - 128$ 之间。

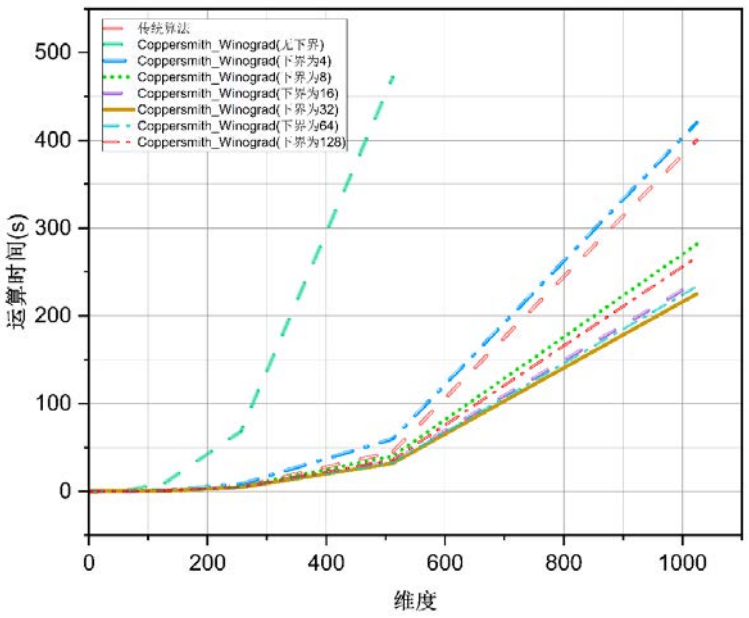

图 16: 不同下界 T 的 Coppersmith-Winograd 算法运算时间

将 Coppersmith-Winograd 算法取不同下界计算不同维度的矩阵所耗费的时间绘制于一张图上可以很清晰地看出界限 T 在不同维度下对计算时间的影响。如图17所示，随着下界的降低和维度的增大，算法的耗时出现了一个难以接受的峰值，其原因在前一部分也已分析过；在维度较低的时候下界的设置对运算时间几乎没有影响，因为传统算法和 Coppersmith-Winograd 算法在低维度下的运行效率十分相近；但是随着维度的上升可以发现对于特定的维度存在一个最优的下界。计算三维图的投影得到图18，同样可以在维度接近 700 时得到意想不到的效果，因此使用 Coppersmith-Winograd 算法时也要设置不同的下界。

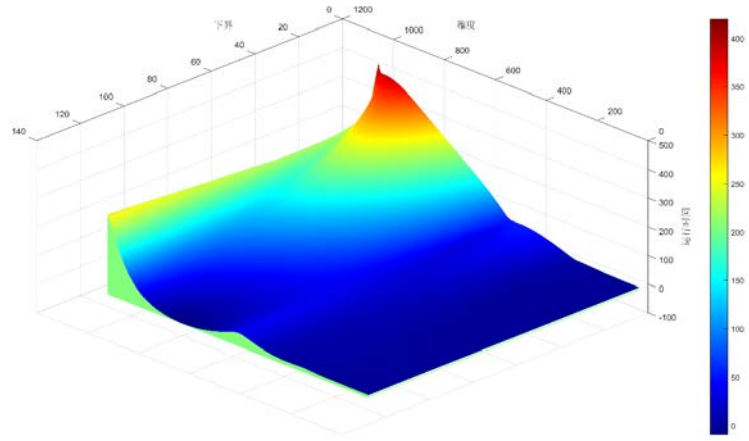

图 17: Coppersmith-Winograd 算法不同维度与下界对运行时间的影响 (三维图)

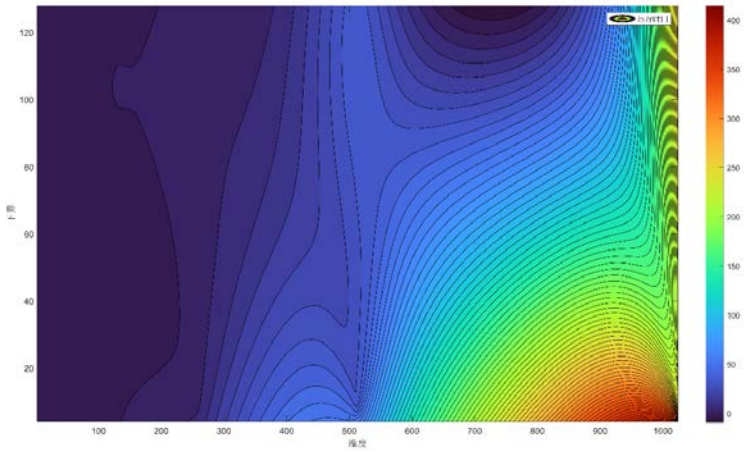

图 18: Coppersmith-Winograd 算法不同维度与下界对运行时间的影响 (投影)

将相同界限 T 下的 Coppersmith-Winograd 算法与改进 Strassen 算法进行对比，对比结果如下图所示，可以看到 Coppersmith-Winograd 算法相较于改进 Strassen 算法在求逆运算过程中效率进一步提升，对 1024 维矩阵求逆的时间进一步降低了 2.2%，其主要原因是因为 Coppersmith-Winograd 算法在矩阵求逆过程中将改进 Strassen 求逆算法中的原本由 Strassen 算法计算的矩阵乘法，使用 Coppersmith-Winograd 算法进行替代，进一步提升了改进 Strassen 算法求逆过程中的矩阵乘法运行效率，从而提升了求逆的效率，因此我们最终选择 Coppersmith-Winograd 算法进行题目中的矩阵求逆计算。

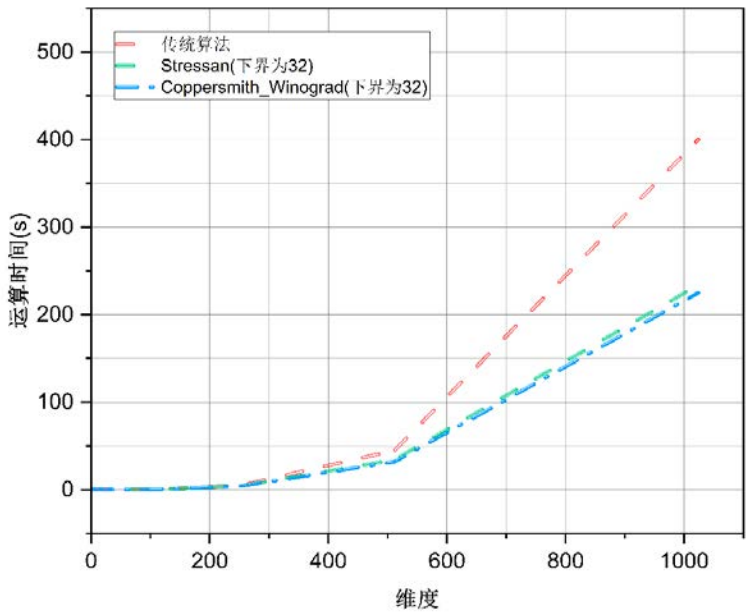

图 19: Coppersmith-Winograd 算法和改进 Strassen 算法运行时间对比

5.2.5 问题 1 求解

首先使用模方法，计算出了每个数据集同一行 $\mathbf{H}_{j,k}$ 之间的相关系数，接着使用聚类父子节点法对具有高度相关性的矩阵进行聚类分析，然后使用随机 SVD 分解方法计算父节点对应的 $\mathbf{V}_{p,q}$，用来代替整个类的 $\mathbf{V}_{j,k}$，以达到简化计算复杂度的目的。图**??**可以看出，满足 $\rho_{min}(\mathbf{V}) \geq \rho_{th} = 0.99$ 的要求。

这里以 Data4H 的第 1 行数据为例，由图20可以看出，384 个矩阵 $\mathbf{H}_{1,k}$ 被分为了 193 个类，每个类的父节点带有一个或者两个子节点，因此可以选取每个父节点对应的 $\mathbf{V}_{p,q}$，所需计算的 $\mathbf{V}_{j,k}$ 数量仅占原先的 50.260%，大大降低了计算复杂度。

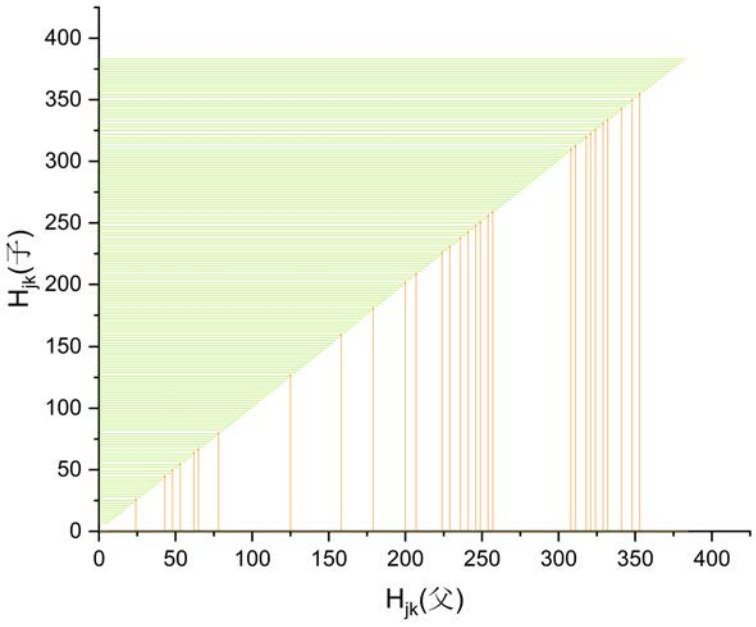

图 20: 聚类父子节点图

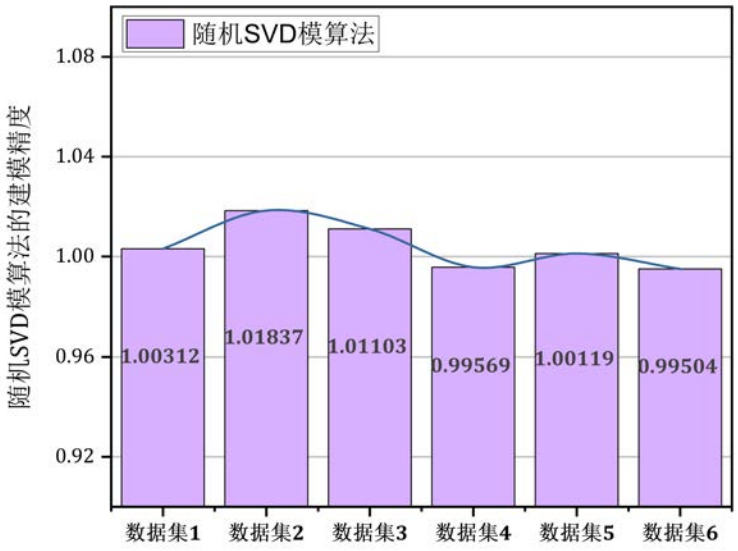

图 21: W 的建模精度

由图21可以看出，计算结果满足 $\rho_{min}(\mathbf{W}) \geq \rho_{th} = 0.99$ 的要求，其中有些值会大于 1，这是由 MATLAB 软件的截断误差导致的。由公式12可知，其分子为一个范数，分母为两个范数相乘，MATLAB 中的 double 类型只有 16 位，上下同时进行截断，就会导致截断误差。

首先使用基于 Strassen 算法的矩阵求逆算法降低了矩阵求逆过程中的复杂度，将 1024 维矩阵的求逆时间降低了 41.5%；继而又结合求逆矩阵为 Hermite 阵的特点提出了一种改进的基于 Strassen 算法的求逆算法，进一步降低了运算复杂度，使 1024 维矩阵的求逆时间又降低了 3.2%；最后将 Coppersmith-Winograd 算法引入到改进的基于 Strassen 算法的求逆算法，进一步降低了改进 Strassen 算法求逆过程中矩阵乘法的计算时间，使 1024 维矩阵的求逆时间在改进算法的基础上进一步降低了 2.2%，从而得到了使用 Coppersmith-Winograd 算法优化的改进 Strassen 求逆算法，相较于传统求逆方法大大降低了算法复杂度，将 1024 维矩阵的求逆时间降低了 46.9%，同时也大大降低了题目中所给的矩阵求逆耗时。

$$\rho_{l,j,k}(\mathbf{W}) = \frac{\|\hat{\mathbf{W}}_{l,j,k}^{H}\mathbf{W}_{l,j,k}\|_2}{\|\hat{\mathbf{W}_{l,j,k}}\|_2\|\mathbf{W}_{l,j,k}\|_2}, l = 1, ..., L \tag{12}$$

5.3 模型二建立与求解

5.3.1 Huffman 编码解码—无损压缩

在计算机科学和信息论中，Huffman 码是一种特殊类型的最优前缀码，通常用于无损数据压缩。查找或使用此类代码的过程通过 Huffman 编码进行 [5]，而 Huffman 编码的首要步骤就是要生成一棵最优带权二叉树—Huffman 树。

Huffman 树:

对于一棵二叉树来说，若叶节点的个数相同且对应权值也相同，而对应权值的叶节点所处的层次不同，则二叉树的带权路径长度可能不相同。但是在所有含 n 个叶结点，并且带有相同权值的二叉树中，必定存在一棵其带权路径长度值为最小的二叉树，也就是最优二叉树。下面我们给出构造其的算法步骤：

假定 n 个叶结点的权值分别为 $\{w_1, w_2, ..., w_n\}$, 则

（1）由已知给定的 n 个权值 $\{w_1, w_2, ..., w_n\}$，构造一棵由 n 棵二叉树所构成的森林 $F = \{T_1, T_2, ..., T_n\}$, 其中每一棵二叉树只有一个根节点，并且根节点的权值分别为 $w_1 \square w_2 \square ... \square w_n$。

（2）在二叉树森林 F 中选取根节点的权值最小和次小的两棵二叉树，分别把他们作为左子树和右子树去构造一棵新的二叉树，新二叉树的根结点权值为其左、右子树根节点的权值之和。

（3）作为新二叉树的左、右子树的两棵两叉树从森林 F 中删除，将新产生的二叉树加入到森林 F 中。

（4）重复步骤（2）和（3），直到森林中只剩下一棵二叉树为止，则这棵二叉树就是所构成的 Huffman 树。

下面给出一个具体是实例，对于一组给定的权值 {5，4，3，2，1}，图22给出了 Huffman 树的构造过程。

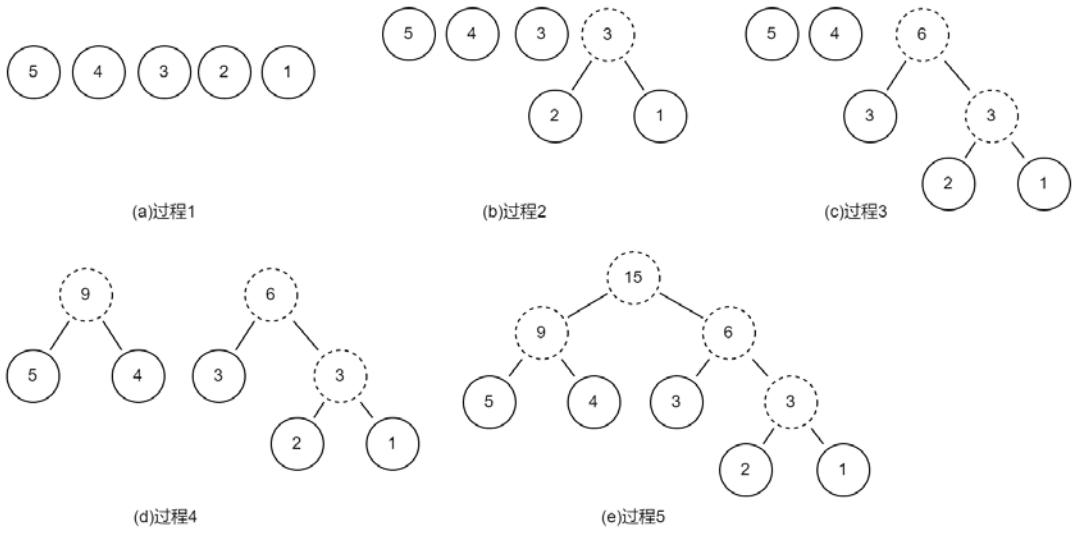

图 22: 构造 Huffman 树的过程

给出 Huffman 树后，我们用 Huffman 树进行编码。

Huffman 编码:

在信息通信领域里，信息传送的速度至关重要，而传送速度由于传送的信息量有关。在信息传送时需要将信息符号转换为二进制组成的符号串，这就需要进行二进制的编码。假设要传送的是由 A、B、C 和 D4 个字符组成的电报文 ABCAABCAD, 在计算机中每个字符在没有压缩的文本文件中由一个字节（例如常见的 ASCII 码）或两个字节（列如 Unicode 码）表示，如果是用这种方案，每个字符都需要 8 个或 16 个位数，但是为了提高存储和传输效率，在信息存储和传送时总是希望传输信息的总长度尽可能短，这就需要设计一套对字符集进行二进制编码的方法，使得采用这种编码方式对信息进行编码时，信息的传输量最小。如果能对每一个字符用不同长度的二进制编码，并且尽可能减少出现次数最多的字符的编码位数，则信息传送的总长度便可以达到最小。

假设将 A、B、C 和 D 分别编码为 0，1，01，10，则电报文 ABCAABCAD 的编码长度只有 12 才能达到最短。然而，在编码序列中，若用起始位组合（或前缀）相同的代码来表示不同的字符，则在不同字符的编码之间必须用分隔符隔开，否则会产生二义性，电文也就无法译码了。为了在字符间省去不必要的分隔符号，这就要求给出的每一个字符的编码必须为前缀编码。所谓前缀编码就是在所有字符的编码中，任何一个字符都不是另一个字符的前缀。

利用 Huffman 树可以构造一种不等长的二进制编码，并且构造所得 Huffman 编码是一种最优前缀编码。

Huffman 编码的构造过程是：用电文中各个字符使用的频度作为叶结点的权，构造一棵具有最小带权路径长度的 Huffman 树，若对树中的每个左分支赋予标记 1，则从根节点到每个叶结点的路径上的标记连接起来就构成了一个二进制串，该二进制串被称为 Huffman 编码。附录中给出了构造 Huffman 树和 Huffman 编码类的描述（java 语言）。

接着，我们给出一个例子，已知一个信息通信联络中使用了 8 个字符：a、b、c、d、e、f、g 和 h，每个字符地使用频度分别为：6、30、8、9、15、24、4 和 12，设计各个字符的 Huffman 编码，图23形象地展示了这一编码过程。

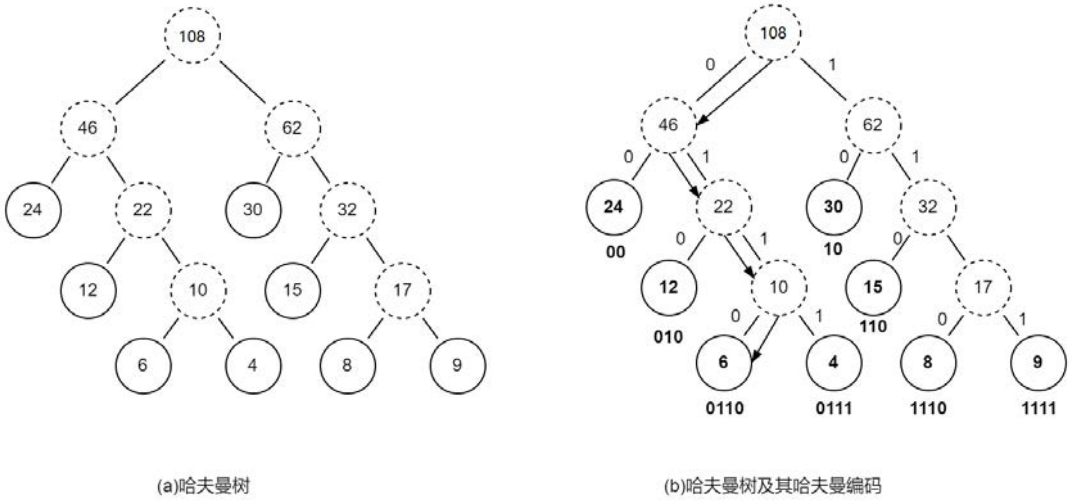

图 23: 例中 Huffman 树和 Huffman 编码

在本例中，得出这 8 个字符的 Huffman 编码后，我们与之前的（假设每个字符需要 8 位数）进行存储比较。Huffman 编码后：$6 \times 4 + 30 \times 2 + 8 \times 4 + 9 \times 4 + 15 \times 3 + 24 \times 2 + 4 \times 4 + 12 \times 3 = 297$; 原:$(6 + 30 + 8 + 9 + 15 + 24 + 4 + 12) \times 8 = 864$。可见节省了 864-297=567 个空间。

编码之后，进行解码操作，就可以完成原始数据的读取。

Huffman 解码:

一般而言，解码过程只是将前缀代码流转换为单个字节值，通常是在从输入流中读取每一位时逐个节点遍历 Huffman 树（到达叶节点必然会终止对该特定字节值的搜索）。在这发生之前，必须以某种方式重建霍夫曼树。Lin 提出的基于递归霍夫曼树的霍夫曼解码快速算法可以加快解码过程 [6]。对于本题而言，可以在 data1-data6 中查找每个元素相同或者相似（实部虚部分别相似，应问题所需精度可定义相似度的衡量参数），然后利用 Huffman 方法进行压缩及解压的操作，这将降低矩阵的存储空间。但问题中所述要求压缩后的每个元素仍然以 32bit 单精度浮点数存储，不考虑位宽的压缩，故这种在解答问题 2 时暂不使用，但不失为一种很好的压缩思想。对于问题 2 的解答，我们采用 5.2.2 中提出的降维分块压缩算法。

5.3.2 分块压缩算法

优化方法 1-降维 SVD 压缩算法

现采用降维 SVD 压缩算法对于 **W** 进行压缩。已知，**W** 是一个 $64 \times 2 \times 4 \times 384$ 的 4 维矩阵，首先把 4 维的矩阵进行降维处理，变成 256×768 的 2 维矩阵，再该矩阵进行 SVD 分解，在保证最终误差的情况下，取出累计贡献率大于 0.974 的前 m 列

$\mathbf{U}(:, 1:m), \mathbf{V}(:, 1:m)$,，前 m 个奇异值进行存储，最后使用它们相乘及升维即可解压(图24)。总共节约的空间为

$$196608 - 1025m \tag{13}$$

个存储单元。

同理，**H** 是一个 $4 \times 64 \times 4 \times 384$ 的 4 维矩阵，首先把 4 维的矩阵进行降维处理，变成 256×1536 的 2 维矩阵，再该矩阵进行 SVD 分解，在保证最终误差的情况下，取出累计贡献率大于 0.947 的前 m 列 $\mathbf{U}(:, \mathbf{1}:\mathbf{m}), \mathbf{V}(:, \mathbf{1}:\mathbf{m})$,，前 m 个奇异值进行存储，最后使用它们相乘及升维即可解压。总共节约的空间为

$$393216 - 1739m \tag{14}$$

个存储单元。

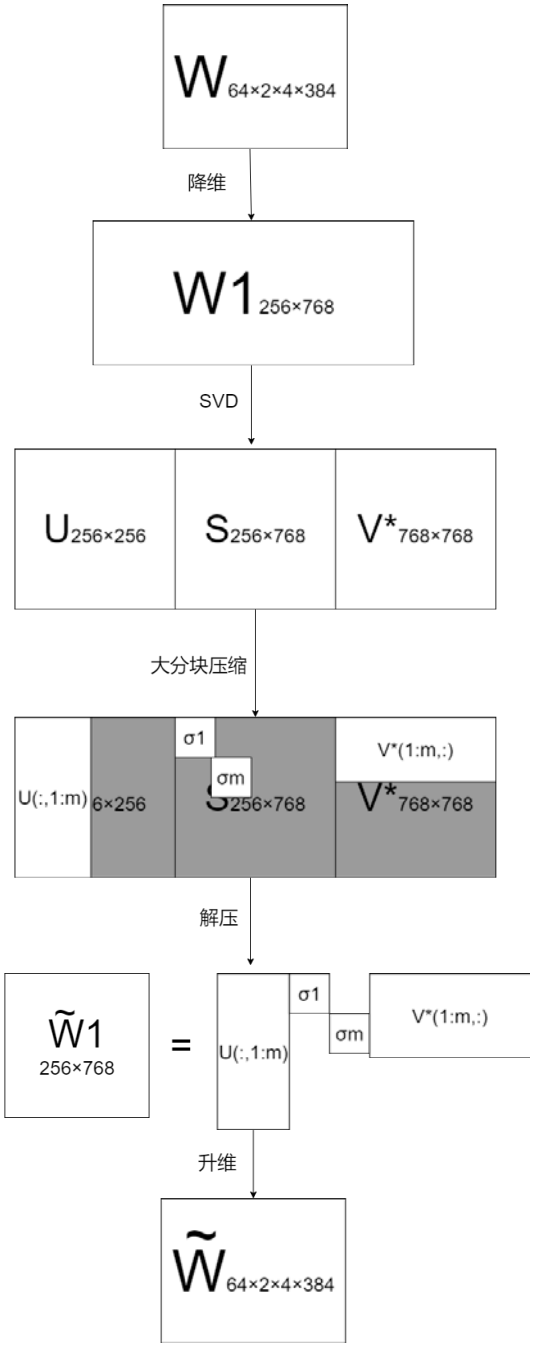

图 24: 优化方法 1

优化方法 2-降维分块 SVD 压缩算法 (大分块)

还可采用大分块压缩对于 **W** 进行压缩。已知，**W** 是一个 $64 \times 2 \times 4 \times 384$ 的 4 维矩阵，首先把 4 维的矩阵进行降维处理，变成 256×768 的 2 维矩阵，再将该 2 维矩阵分为 3 个 256×256 的矩阵，分别对每个矩阵进行 SVD 分解，在保证最终误差的情况下，取出累计贡献率大于 0.966 的前 m 列 $\mathbf{U}(:, \mathbf{1:m}), \mathbf{V}(:, \mathbf{1:m}),$，前 m 个奇异值进行存储，最后使用它们相乘、重组及升维即可解压 (图25)。

设这 3 个 256×256 矩阵分别取前 m_1, m_2, m_3 个奇异值，那么总共节约的空间为

$$196608 - 513(m_1 + m_2 + m_3) \tag{15}$$

个存储单元。

大分块压缩同样可以用于 **H** 的压缩。**H** 是一个 $4 \times 64 \times 4 \times 384$ 矩阵，首先把 4 维的矩阵进行降维处理，为了得到尽可能大的方阵，这里将每个 4×64 矩阵先转置，再降维，成为 256×1536 的 2 维矩阵，再将该 2 维矩阵分为 6 个 256×256 的矩阵，最后仍然使用上述的压缩解压方式 (图25)。

设这 6 个 256×256 矩阵分别取前 $m_1, m_2, m_3, m_4, m_5, m_6$ 个奇异值，那么总共节约的空间为

$$393216 - 513 \sum_{i=1}^{6} m_i \tag{16}$$

个存储单元。

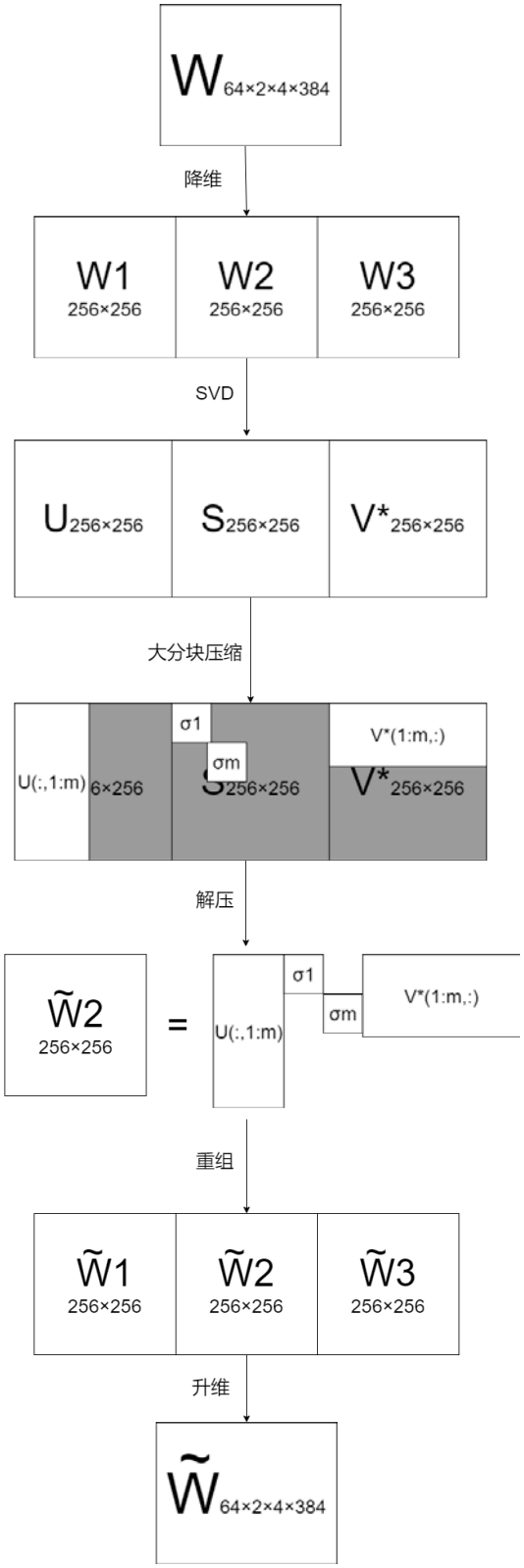

图 25: 优化方法 2

优化方法 3-降维分块 SVD 压缩算法 (小分块)

由于 $\mathbf{H}_{\mathbf{j},\mathbf{k}}$ 是一个 4×64 矩阵，因此可以通过将每个 $\mathbf{H}_{\mathbf{j},\mathbf{k}}$ 或 $\mathbf{W}_{\mathbf{j},\mathbf{k}}$ 分块成为 16 个 4×4

的矩阵，对于每个小矩阵进行压缩处理。

小分块压缩对于每个矩阵的维数进行了要求，由于 $\mathbf{W_{j,k}}$ 是一个 64×2 矩阵，所以不能对其进行小分块压缩，否则存储空间不减反增（$2 * 2 < 2 + 1 + 2$）。因此，该方法仅对 **H** 适用。

对于所有的 4×4 小矩阵，首先检验是否是稀疏矩阵或对称矩阵，如果是则可以按照已有的压缩解压算法（只存储非零元或主对角元以上）进行，若不是则进行 SVD 分解，求出它们的奇异值及左右奇异向量，然后将最大奇异值贡献率超过 74.9% 的矩阵筛选出来，最后将 **U** 的第一列 $\mathbf{U}(:, \mathbf{1})$，$\mathbf{V}(:, \mathbf{1})$ 的第一列及最大奇异值进行存储，可降低存储的复杂度。解压时，将存储的 $\mathbf{U}(:, \mathbf{1})$ 与最大奇异值及 $\mathbf{V}(:, \mathbf{1})^*$ 相乘即可得到原矩阵的近似（图26）。

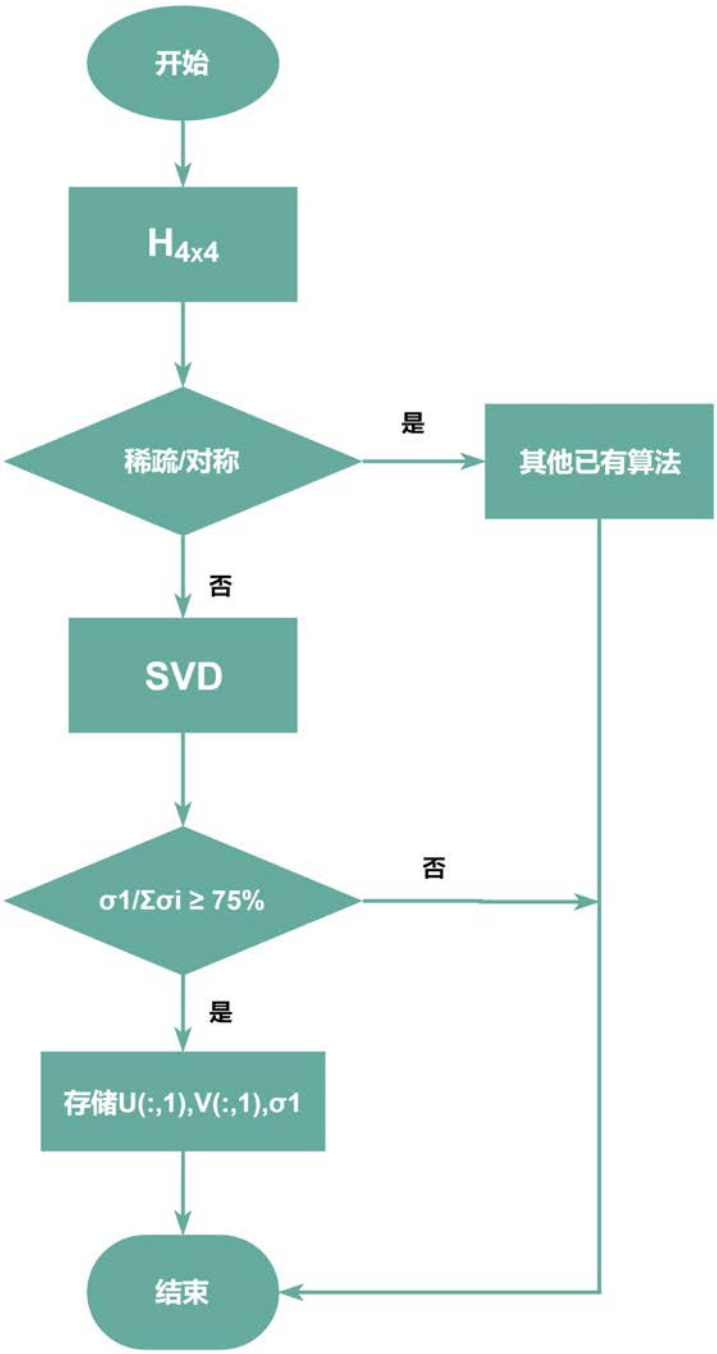

图 26: 优化方法 3 流程图

如图27所示，原来每个小矩阵所占用 $4 \times 4 = 16$ 个存储单元，现在经过压缩仅占用 $4 \times 1 + 1 + 1 \times 4 = 9$ 个存储单元，每个符合贡献率条件的小矩阵可以节约 $16 - 9 = 7$ 个存储单元。

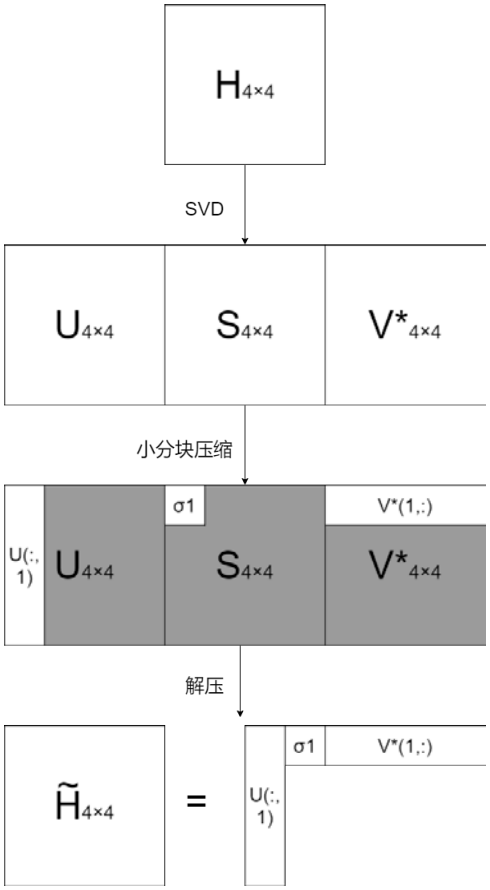

图 27: 优化方法 3

5.3.3 问题 2 求解

通过计算，可以得到 **W**（图33）与 **V**（图34）在上述三个优化方法下节约的空间。在优化方法 3 中没有检测到对称或者稀疏矩阵。

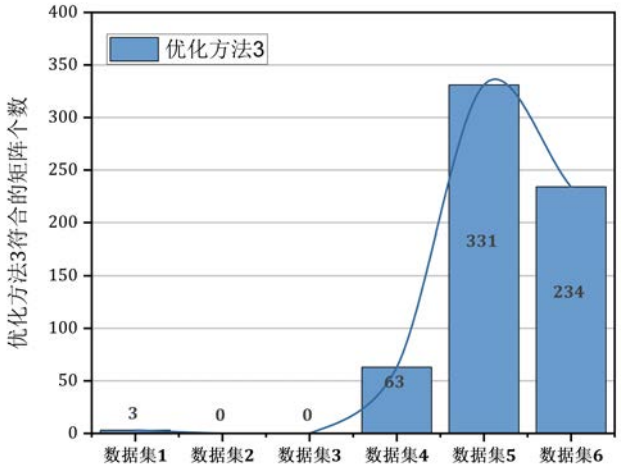

图 28: 优化方法 3 满足条件的矩阵数目

可以看出，优化方法 3 在对 H 进行压缩时表现不佳，一方面因为每个符合压缩条件的 4×4 矩阵仅可以节约 7 个存储单位；另一方面，满足条件的矩阵数目也过少（图28）。优化方法 1 的表现次之，（图29) 展示了所有数据集在该方法下所取的矩阵奇异值数量，可以看到，相比原来的 256 个奇异值，所选取的奇异值数目降低显著，在不失精确度的情况下压缩率升高。优化方法 2 的表现最好，最高的优化达到了 19551552 比特，相比于方法 1 的优化空间平均提升了 65.799%。将方法 1 与方法 2 所取奇异值的数目（图29与图30）带入公式14和16，理论分析与结果结果相一致，证实了优化方法 2 的表现最优。

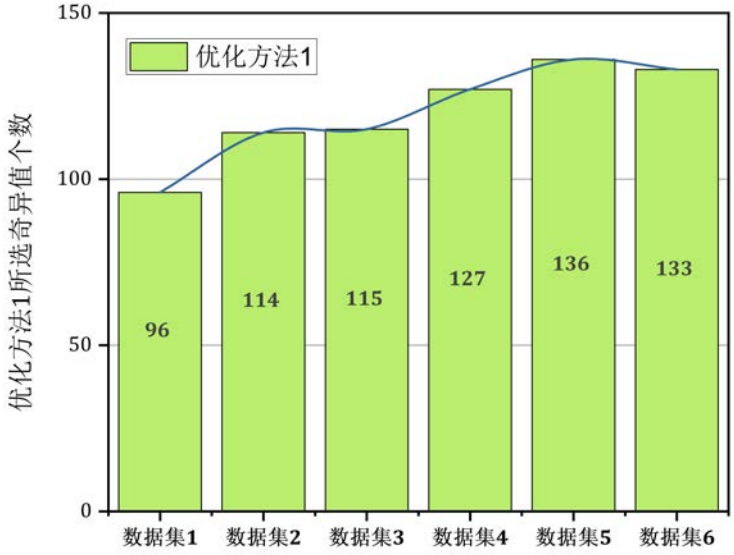

图 29: 优化方法 1 取奇异值数目

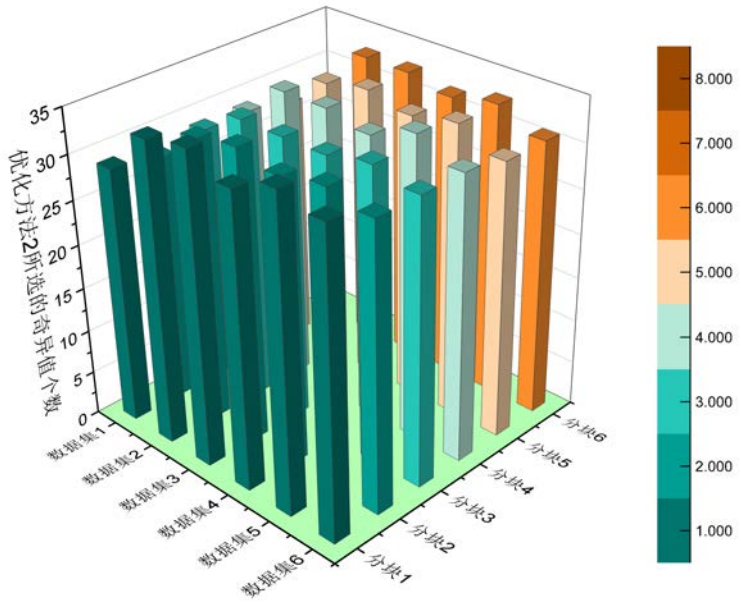

图 30: 优化方法 2 取奇异值数目

优化方法 1 在对 W 进行压缩时表现略差，（图31) 展示了所有数据集在该方法下所取的矩阵奇异值数量。优化方法 2 的表现更好，最高的优化达到了 8150592 比特，相比于方法 1 的优化空间平均提升了 52.190%。将方法 1 与方法 2 所取奇异值的数目（图31与图32）带入公式13和15，理论分析与结果结果相一致，证实了优化方法 2 的表现更优。

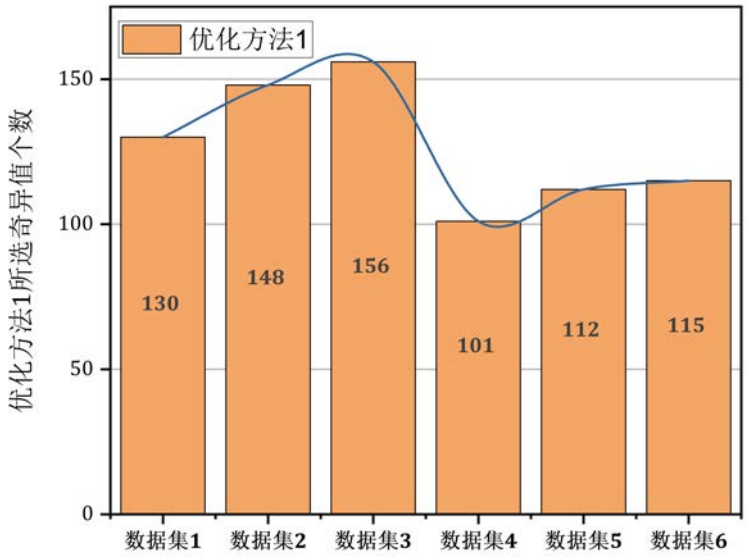

图 31: 优化方法 1 所取奇异值的数量

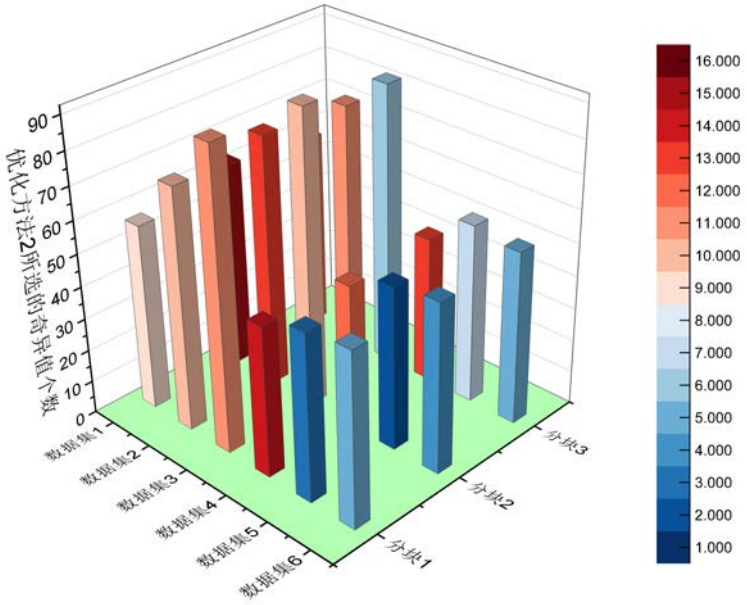

图 32: 优化方法 2 所取奇异值的数量

此外，上述所有算法均满足精确度要求：$err_{\mathbf{H}} \leq E_{th1} = -30dB$、$err_{\mathbf{W}} \leq E_{th2} = -30dB$。

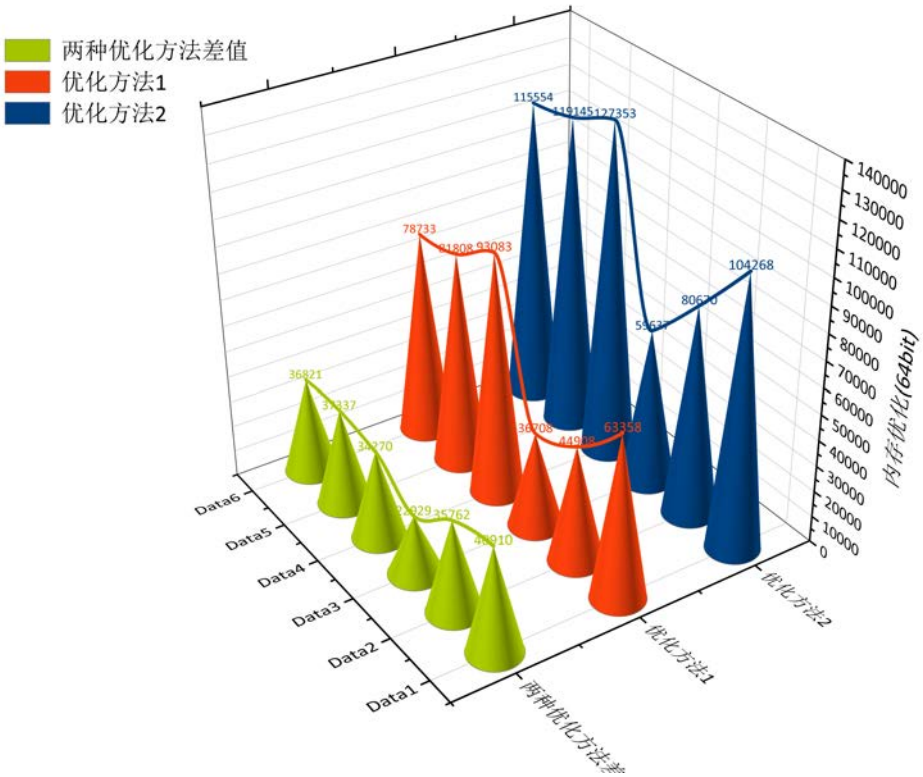

图 33: W 在不同方法下节约的空间

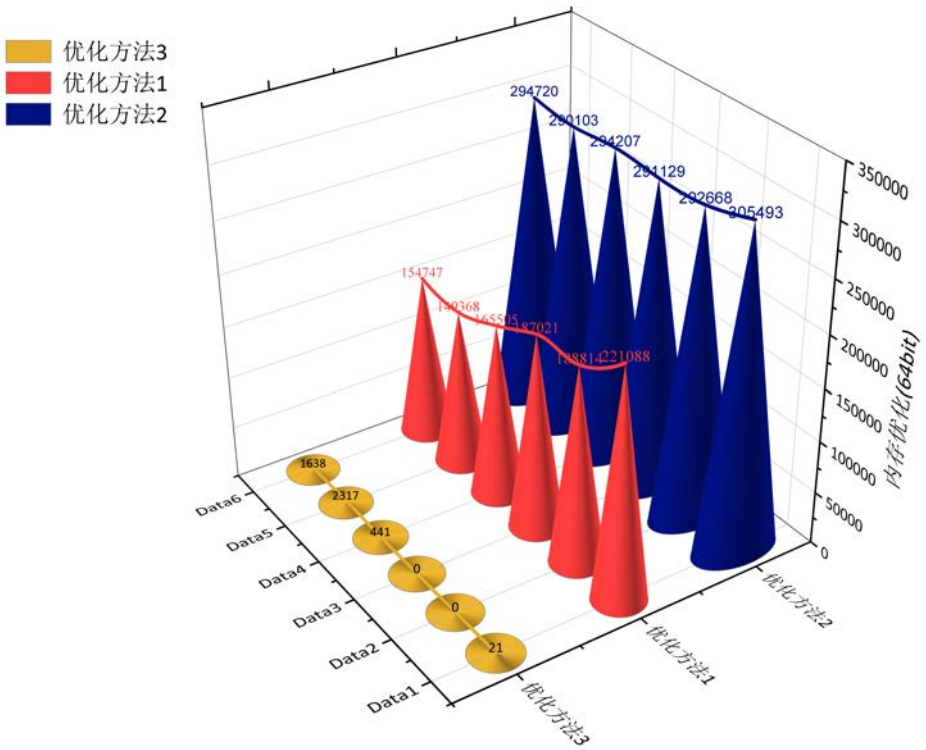

图 34: V 在不同方法下节约的空间

5.4 模型三建立与求解

5.4.1 模型三的建立

模型三将模型一与模型二相结合，既考虑了计算复杂度，又考虑了存储空间复杂度。

首先，基于模型二的求解结果，对于给定的所有数据集的 **H**，应该运用优化方法 2 进行数据压缩，降低储存空间；接着，对于压缩的矩阵 $\mathbf{H}_p$ 进行解压缩，得到解压后的矩阵 $\mathbf{H}_r$；再对 $\mathbf{H}_r$ 使用模方法、聚类父子节点法及随机 SVD 分解算出 $\mathbf{V}_r$；然后，使用基于 Strassen 的改进求逆乘法计算 $\mathbf{W}_r$；最后，基于模型二的求解结果，对 $\mathbf{W}_r$ 使用优化方法 2 进行压缩，可得 $\mathbf{W}_p$，对其解压即可求得模型三所需的 $\hat{\mathbf{W}}$。

5.4.2 问题 3 的求解

经过计算，可得模方法及父子节点法聚类的结果，这里以 Data4H 的第 1 行数据为例，由图37可以看出，384 个矩阵 $\mathbf{H}_{1,k}$ 被分为了 180 个类，每个类的父节点带有一个或者两个子节点，因此可以选取每个父节点对应的 $\mathbf{V}_{p,q}$，所需计算的 $\mathbf{V}_{j,k}$ 数量仅占原先的 46.875%，大大降低了计算复杂度。

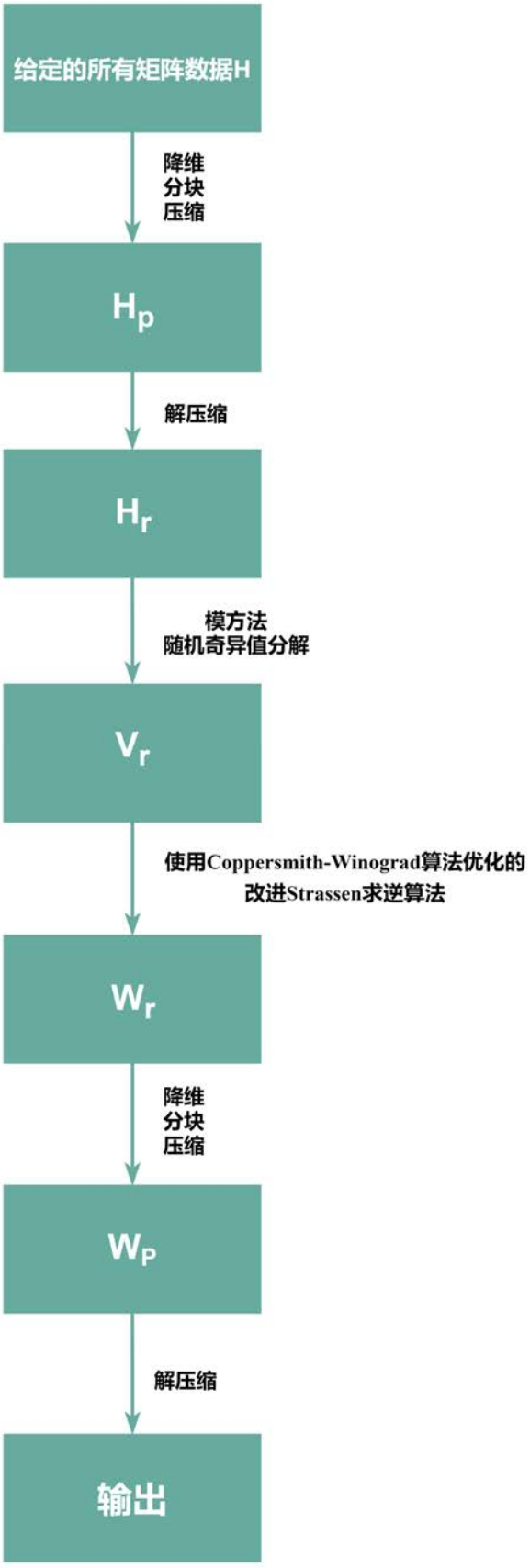
给定的所有矩阵数据H
降维
分块
压缩
H_p
解压缩
H_r
模方法
随机奇异值分解
V_r
使用Coppersmith-Winograd算法优化的
改进Strassen求逆算法
W_r
降维
分块
压缩
W_P
解压缩
输出

图 35: 问题三流程图

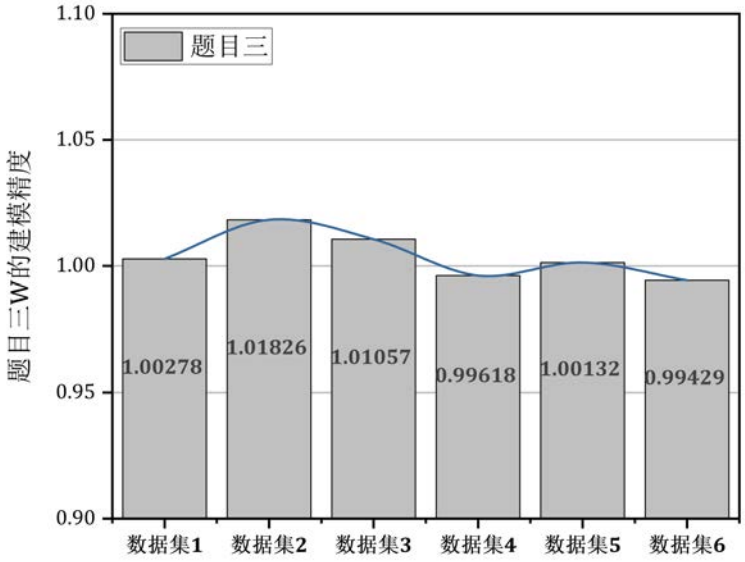

图 36: W 建模精确度

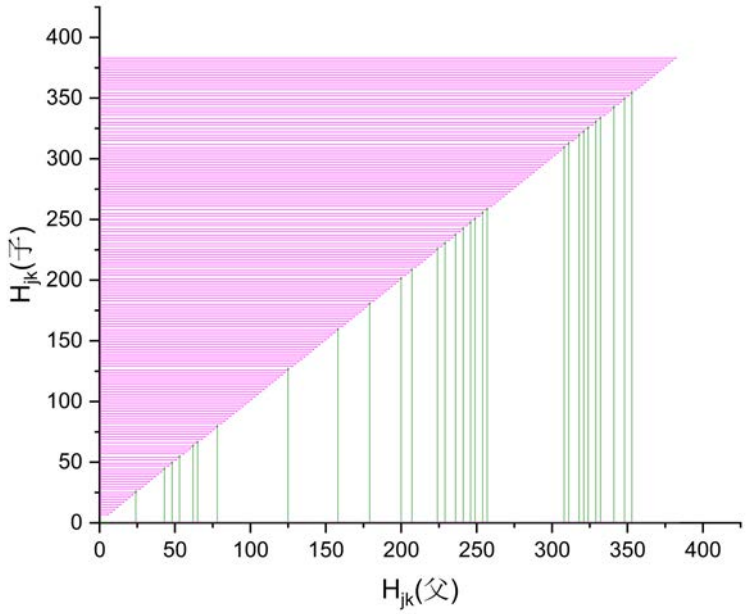

图 37: 聚类父子节点图

由图36可以看出，求解满足 $\rho_{min}(\mathbf{W}) \geq \rho_{th} = 0.99$ 的要求，超过 1 的数字已在之前讨论过。

6. 模型总结与评价

6.1 模型优点

（1）本文在问题 1 求解过程中提出的“模方法”可以有效地捕捉到不同矩阵之间具有相关关系的矩阵。

（2）“聚类父子结点算法”可以通过使用这些具有相关关系的矩阵来构建由不同矩阵组成的相关矩阵组，进而避免进行非必要的奇异值分解过程。

（3）使用 Strassen 求逆算法利用 Strassen 算法和矩阵分治思想降低了矩阵求逆的复杂度；抓住了需求中 Hermite 阵求逆这一要点进行了优化，提出了改进的 Strassen 求逆算法，进一步减少了求逆过程中的矩阵运算次数；在改进的 Strassen 算法的基础上引入 Coppersmith-Winograd 算法降低求逆过程中的矩阵乘法运算的复杂度，进一步降低了整个求逆算法的复杂度。

（4）“近似矩阵分解的概率算法—随机奇异值分解”能高效地查找原始矩阵的低阶近似矩阵, 而后只需对这些低阶近似矩阵做奇异值分解。与标准奇异值分解相比，提高了分解速度，进而降低模型的计算复杂度。

（5）“降维分块压缩算法”利用矩阵奇异值分解后，贡献率较大的前 n 个奇异值及其对应的左、右奇异向量能展示原始矩阵大部分的样貌。同时分块的思想能最大化的利用上述原理来压缩矩阵，进而在保证精度的前提下，实现了廉价存储。

6.2 模型缺点与改进方向

（1）本文需要满足精度要求。在“模方法”中，满足精度要求的相关系数 R 的下限 $R_{critical}$，以及“降维分块压缩算法”中满足精度要求的按贡献度排列的前 n 个奇异值中的 n 的下限 $n_{critical}$, 需要依精度要求手动调节查找。如果时间充裕，可以写一个算法来找到满足精度要求的 $R_{critical}$ 和 $n_{critical}$。

（2）这 6 个数据集中，对 $\mathbf{H}_{j,k}(4 \times 64)$ 进行随机奇异值分解，产生的正交阵 $\mathbf{Q}_{j,k}$（$\mathbf{H}_{j,k} \approx \mathbf{Q}_{j,k}\mathbf{Q}_{j,k}{}^{*}\mathbf{H}_{j,k}$) 有很多维度为 4×4, 这样的 $\mathbf{Q}_{j,k}$ 对于接下来要进行的简约矩阵 $\mathbf{B}_{j,k}(\mathbf{B}_{j,k} = \mathbf{Q}_{j,k}{}^{*}\mathbf{H}_{j,k})$ 的奇异值分解没有起到简化作用。我们可以试图找出能分别产生 4×4 维度的正交阵 $\mathbf{Q}_{j,k}$ 的 $\mathbf{H}_{j,k}$ 和产生 4×3 维度的正交阵 $\mathbf{Q}_{j,k}{}'$ 的 $\mathbf{H}_{j,k}{}'$，将 $\mathbf{H}_{j,k}$ 和 $\mathbf{H}_{j,k}{}'$ 拼接在一起，构建一个维度为 4×128 的矩阵 $\mathbf{H}_{j,k_{fusion}}$, 然后对 $\mathbf{H}_{j,k_{fusion}}$ 进行随机奇异值分解，探究是否能产生一个维度为 3×128 的矩阵 $\mathbf{Q}_{j,k_{fusion}}$（如果产生，将较拼接之前进一步提高计算速度）。但是拼接、探究的过程，也会给计算机增添负担，因此做一个是否拼接的权衡方案是很有必要的。

（3）使用 Coppersmith-Winograd 算法优化的改进 Strassen 求逆算法运行的效率和设置的界限 T 密切相关，需要根据矩阵的维度调整界限 T，不合适的界限 T 甚至会降低算法的运行效率。在后续的工作当中可以将界限 T 作为一个自适应参数在算法中实现，使算

法可以根据当前矩阵维度和运行效率自动调节界限 T。

（4）Strassen 算法和 Coppersmith-Winograd 算法只能计算方阵的乘法，且方阵的维度必须为 2^n，这就使得算法的适用性有所降低，应对算法进行进一步优化，降低其对维度的敏感性。

（5）对于“降维分块压缩算法”，本文中展现了三种分块方式对于压缩存储的表现。如果时间充裕，可以设计一个算法来寻找最优分块方式，以实现最优降维分块压缩存储。

（6）本文中涉及矩阵输入的代码是按照题目给的矩阵维度设计的。如果时间充裕，可以增强代码的可扩展性，使其有能力处理输入任意维度的矩阵，进而将降低计算复杂度和廉价存储的算法应用于现实中遇到的其他问题上。

参考文献

[1] N. Halko, P.G. Martinsson. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. SIAM Rev. 2011;53(2):217-288.

[2] Strassen,Volker. Gaussian elimination is not optimal. Numerische Mathematik 1969;13(4):354–356.

[3] Pan, V. Ya. New combinations of methods for the acceleration of matrix multiplications. Computers & Mathematics with Applications 1981;7(1):73–125.

[4] Coppersmith, Don, and Shmuel Winograd. Matrix multiplication via arithmetic progressions. Proceedings of the nineteenth annual ACM symposium on Theory of computing 1987.

[5] D.A. Huffman. A method for the construction of minimum redundancy codes. In: Proc. IRE 1952;40:1098–1101.

[6] Y.K. Lin, S.C. Huang, and C.H. Yang. A fast algorithm for Huffman decoding based on a recursion Huffman tree. Journal of Systems & Software 2012;85(4):974-980.

A 我的 MATLAB 源程序

```
V2All=zeros(64,2,384,4,6);%存储所有随机加模方法算出来的svdV
containerAll=zeros(384,10,6,4);%最后两个参数表示哪个数据集，哪一行
filepath='C:\Users\wangy\Desktop\A\H';
for q=1:6
temp=load([filepath,'\Data',num2str(q),'_H','.mat']);
H=temp.H;

N=[];
for i=1:4%所有矩阵的二范数相当于压缩了,
   for j=1:384
      A=H(:,:,i,j);
      for k=1:64
         N(i,j,k)=norm(A(:,k));
      end
   end
end
Co=zeros(4,384,384);
for i=1:4
   for j=1:384
      for p=j:384
         temp=corrcoef(N(i,j,:),N(i,p,:));
         Co(i,j,p)=temp(1,2);
      end
   end
end
M=[];
l=1;
for i=1:4
   for j=1:384
      for k=j:384
         if Co(i,j,k)>=0.9985%筛选出来相关系数大于的矩阵关系，存储在0.95中M
            M(l,1)=i;
            M(l,2)=j;
            M(l,3)=k;
            l=l+1;
         end
      end
```

```
    end
end

% 所有的算出来
V2=zeros(64,2,384,4);%是求出的所有矩阵V2V
for k=1:4
    container=julei(M,k);
    containerAll(1:size(container,1),1:size(container,2),q,k)=container;
    column=container(:,1);
    V1=zeros(64,2,size(column,1));
    for i=1:size(column)
        a=column(i);
        tmp=suiji(H(:,:,k,a));
        V1(:,:,i)=tmp(:,1:2);%求随机SVD
    end

    V3=zeros(64,2,384);
    for i=1:size(column)
        for j=1:size(container,2)
            if container(i,j)~=0
                V3(:,:,container(i,j))=V1(:,:,i);
            end
        end
    end

    V2(:,:,:,k)=V3;
end
V2All(:,:,:,:,q)=V2;
% 进行的检验V
temp1=load([filepath,'\Data',num2str(q),'_V','.mat']);
V=temp1.V;
P=zeros(2,4,384);
for i=1:4
    for j=1:384
        for l=1:2
            a=V(:,l,i,j);
            b=V2(:,l,j,i);
            P(l,i,j)=norm(a'*b)/norm(a)*norm(b);
        end
```

```
    end
end
errorAll(1,q)=min(min(min(P)));
end

%% 第一题求解写得到的，写出V.文件给程序运行matPython
%写入文件excel
V2final=zeros(256,768,6);
for k=1:6
    for j=1:384
        for i=1:4
            tmp=V2All(:,:,j,i,k);
            V2final((i-1)*64+1:i*64,(j-1)*2+1:j*2,k)=tmp;%此处的用于下一步的
                对接V
        end
    end
end
data1=V2final(:,:,1);
data2=V2final(:,:,2);
data3=V2final(:,:,3);
data4=V2final(:,:,4);
data5=V2final(:,:,5);
data6=V2final(:,:,6);
for i=1:6
    save([pwd,'\V',num2str(i)],['data',num2str(i)]);
end
%% 问题一的求解再计算W 存储使用计算的的过程pythonW
dataAll=zeros(256,768,6);
for i=1:6
    tmp=load([pwd,'\data',num2str(i),'.mat']);
    dataAll(:,:,i)=tmp.mat;
end

dataAll(:,:,1)=data1;
dataAll(:,:,2)=data2;
dataAll(:,:,3)=data3;
dataAll(:,:,4)=data4;
dataAll(:,:,5)=data5;
dataAll(:,:,6)=data6;
W_new=zeros(64,2,4,384,6);%用来存储所有的数据W
```

```
for i=1:6
   W_new(:,:,:,:,i)=cunchuW(dataAll(:,:,i));%升维处理
end
%% 第一题的解答对算出来的进行检测pythonW
accuracy=zeros(6,1);%检测准确度
filepath='C:\Users\wangy\Desktop\A\W';
for q=1:6
temp1=load([filepath,'\Data',num2str(q),'_W','.mat']);
W=temp1.W;
P=zeros(2,4,384);
for i=1:4
   for j=1:384
      for l=1:2
         a=W(:,l,i,j);
         b=W_new(:,l,i,j,q);
         P(l,i,j)=norm(a'*b)/norm(a)*norm(b);
      end
   end
end
accuracy(q,1)=min(min(min(P)));
end
%% 问题二的求解小分块压缩只能对 (进行处理H)
errorAll=zeros(6,1);
countAll=zeros(6,1);
countSymmetry=0;%寻找对称
filepath='C:\Users\wangy\Desktop\A\H';
for q=1:6
temp=load([filepath,'\Data',num2str(q),'_H','.mat']);
H=temp.H;

%首先先把每个进行分裂H
H2=zeros(4,384,4,4,16);%是存储的所有小矩阵H24*4
for i=1:4
   for j=1:384
      H1=H(:,:,i,j);
      for k=1:16
         H2(i,j,:,:,k)=H1(1:4,(k-1)*4+1:k*4);
      end
   end
```

```
end
%接着算每个小矩阵首元奇异值的贡献率
contribution=zeros(4,384,16);
location=[];
p=1;
for i=1:4
   for j=1:384
      for k=1:16
         H3=H2(i,j,:,:,k);
         tmp=reshape(H3,[4,4]);
         if isequal(tmp,tmp')|| isequal(tmp,tmp.')
            countSymmetry=countSymmetry+1;%寻找每个小矩阵是否有对称矩阵
         end
         [U3,S3,V3]=svd(tmp);
         contribution(i,j,k)=S3(1,1)/(S3(1,1)+S3(2,2)+S3(3,3)+S3(4,4));
         if contribution(i,j,k)>=0.749
            location(p,1)=i;
            location(p,2)=j;
            location(p,3)=k;
            p=p+1;
         end
      end
   end
end
countAll(q,1)=size(location,1);

H6=H;
for i=1:size(location,1)
   H4=H2(location(i,1),location(i,2),:,:,location(i,3));
   tmp=reshape(H4,[4,4]);
   [U,S,V]=svd(tmp);
   V=V';
   H5=U(1:4,1)*S(1,1)*V(1,1:4);
   H6(1:4,(location(i,3)-1)*4+1:location(i,3)*4,
      location(i,1),location(i,2))=H5;%是替换完
      的H6
end

sum1=0;
sum2=0;
```

```
for i=1:4
   for j=1:384
      a=H6(:,:,i,j);
      b=H(:,:,i,j);
      sum1=sum1+(norm(a-b,'fro'))^2;
      sum2=sum2+(norm(b,'fro'))^2;
   end
end
error=10*log10(sum1/sum2);
errorAll(q,1)=error;

end
spareSpace3H=7*countAll;
errorMinH=errorAll;
%% 问题二的求解大分块压缩先对进行处理 W
errorAll=zeros(6,1);
thresAll=zeros(6,3);
filepath='C:\Users\wangy\Desktop\A\W';
for k=1:6
temp=load([filepath,'\Data',num2str(k),'_W','.mat']);
W=temp.W;

W2D=zeros(256,768);%先把四维矩阵展成二维矩阵
for i=1:4
   for j=1:384
      W2D((i-1)*64+1:i*64,(j-1)*2+1:j*2)=W(:,:,i,j);
   end
end
%然后分三个256大块进行分解×256SVD
H2DD=zeros(256,768);%存储解压之后的值
threshold=zeros(1,3);%存储用了前多少列
for i=1:3
   [A,j]=accumCon(W2D(:,(i-1)*256+1:i*256));
   H2DD(:,(i-1)*256+1:i*256)=A;
   threshold(1,i)=j;
end
thresAll(k,:)=threshold;
sum1=0;
sum2=0;%计算误差
```

```
for i=1:4
   for j=1:384
      a=W2D((i-1)*64+1:i*64,(j-1)*2+1:j*2);
      b=H2DD((i-1)*64+1:i*64,(j-1)*2+1:j*2);
      sum1=sum1+(norm(a-b,'fro'))^2;
      sum2=sum2+(norm(b,'fro'))^2;
   end
end
error=10*log10(sum1/sum2);
errorAll(k,1)=error;
end
errorMedium=errorAll;%大分块的误差
thres=zeros(6,1);
for i=1:6
   sum=0;
   for j=1:3
      sum=sum+thresAll(i,j);
   end
   thres(i,1)=sum;
end
thresAllMedium=thresAll;
spareSpace2=ones(6,1)*196608-513*thres;%节省的空间
%% 问题二的求解大分块压缩再对于处理 H
errorAll=zeros(6,1);
thresAll=zeros(6,6);
filepath='C:\Users\wangy\Desktop\A\H';
for k=1:6
temp=load([filepath,'\Data',num2str(k),'_H','.mat']);
H=temp.H;

H2D=zeros(256,1536);%先把四维矩阵展成二维矩阵
for i=1:4
   for j=1:384
      H2D((i-1)*64+1:i*64,(j-1)*4+1:j*4)=H(:,:,i,j).';%要转置
   end
end
%然后分六个256大块进行分解×256SVD
H2DD=zeros(256,1536);%存储解压之后的值
threshold=zeros(1,6);%存储用了前多少列
```

```
for i=1:6
   [A,j]=accumCon(H2D(:,(i-1)*256+1:i*256));
   H2DD(:,(i-1)*256+1:i*256)=A;
   threshold(1,i)=j;
end
thresAll(k,:)=threshold;
sum1=0;
sum2=0;%计算误差
for i=1:4
   for j=1:384
      a=H2D((i-1)*64+1:i*64,(j-1)*4+1:j*4);
      b=H2DD((i-1)*64+1:i*64,(j-1)*4+1:j*4);
      sum1=sum1+(norm(a-b,'fro'))^2;
      sum2=sum2+(norm(b,'fro'))^2;
   end
end
error=10*log10(sum1/sum2);
errorAll(k,1)=error;
end
errorMediumH=errorAll;%大分块的误差H
thres=zeros(6,1);
for i=1:6
   sum=0;
   for j=1:6
      sum=sum+thresAll(i,j);
   end
   thres(i,1)=sum;
end
thresAllMediumH=thresAll;
spareSpace2H=ones(6,1)*393216-513*thres;%节省的空间
%% 问题二的求解不分块处理！W一个大块()
errorAll=zeros(6,1);
thresAll=zeros(6,1);
filepath='C:\Users\wangy\Desktop\A\W';
for k=1:6
temp=load([filepath,'\Data',num2str(k),'_W','.mat']);
W=temp.W;

W2D=zeros(256,768);%先把四维矩阵展成二维矩阵
```

```
for i=1:4
   for j=1:384
      W2D((i-1)*64+1:i*64,(j-1)*2+1:j*2)=W(:,:,i,j);
   end
end
%然后直接进行分解SVD
H2DD=zeros(256,768);%存储解压之后的值
[A,j]=accumCon(W2D);
threshold=j;%存储用了前多少列
thresAll(k,1)=threshold;
H2DD=A;

sum1=0;
sum2=0;%计算误差
for i=1:4
   for j=1:384
      a=W2D((i-1)*64+1:i*64,(j-1)*2+1:j*2);
      b=H2DD((i-1)*64+1:i*64,(j-1)*2+1:j*2);
      sum1=sum1+(norm(a-b,'fro'))^2;
      sum2=sum2+(norm(b,'fro'))^2;
   end
end
error=10*log10(sum1/sum2);
errorAll(k,1)=error;
end
spareSpace1=ones(6,1)*196608-1025*thresAll;
thresAllMax=thresAll;
errorMax=errorAll;
%% 问题二的求解不分块处理！H一个大块()
errorAll=zeros(6,1);
thresAll=zeros(6,1);
filepath='C:\Users\wangy\Desktop\A\H';
for k=1:6
temp=load([filepath,'\Data',num2str(k),'_H','.mat']);
H=temp.H;

H2D=zeros(256,1536);%先把四维矩阵展成二维矩阵
for i=1:4
   for j=1:384
```

```
        H2D((i-1)*64+1:i*64,(j-1)*4+1:j*4)=H(:,:,i,j).';%要转置
    end
end
%直接进行分解SVD
H2DD=zeros(256,1536);%存储解压之后的值
[A,j]=accumCon(H2D);
H2DD=A;
threshold=j;

thresAll(k,1)=threshold;
sum1=0;
sum2=0;%计算误差
for i=1:4
    for j=1:384
        a=H2D((i-1)*64+1:i*64,(j-1)*4+1:j*4);
        b=H2DD((i-1)*64+1:i*64,(j-1)*4+1:j*4);
        sum1=sum1+(norm(a-b,'fro'))^2;
        sum2=sum2+(norm(b,'fro'))^2;
    end
end
error=10*log10(sum1/sum2);
errorAll(k,1)=error;
end
spareSpace1H=ones(6,1)*393216-1793*thresAll;
thresAllMaxH=thresAll;
errorMaxH=errorAll;
%% 问题三的求解先压缩解压矩阵H
%大分块压缩 256*256
errorAll=zeros(6,1);
thresAll=zeros(6,6);
HAll=zeros(4,64,4,384,6);%存储解压缩后的H
filepath='C:\Users\wangy\Desktop\A\H';
for k=1:6
temp=load([filepath,'\Data',num2str(k),'_H','.mat']);
H=temp.H;

H2D=zeros(256,1536);%先把四维矩阵展成二维矩阵
for i=1:4
    for j=1:384
```

```
        H2D((i-1)*64+1:i*64,(j-1)*4+1:j*4)=H(:,:,i,j).';%要转置
    end
end
%然后分六个256大块进行分解×256SVD
H2DD=zeros(256,1536);%存储解压之后的值
for i=1:6
    [A,j]=accumCon(H2D(:,(i-1)*256+1:i*256));
    H2DD(:,(i-1)*256+1:i*256)=A;
end
%接着对压缩后的升维度H
for i=1:4
    for j=1:384
        HAll(:,:,i,j,k)=H2DD((i-1)*64+1:i*64,(j-1)*4+1:j*4).';
    end
end
end

%% 第三题的求解再进行随机及模方法SVD
errorAll=zeros(1,6);
V2All=zeros(64,2,384,4,6);%存储所有随机加模方法算出来的svdV
containerAll=zeros(384,10,6,4);%最后两个参数表示哪个数据集，哪一行
filepath='C:\Users\wangy\Desktop\A\H';
for q=1:6
H=HAll(:,:,:,:,q);%使用解压后的矩阵H

N=[];
for i=1:4%所有矩阵的二范数相当于压缩了,
    for j=1:384
        A=H(:,:,i,j);
        for k=1:64
            N(i,j,k)=norm(A(:,k));
        end
    end
end
Co=zeros(4,384,384);
for i=1:4
    for j=1:384
        for p=j:384
            temp=corrcoef(N(i,j,:),N(i,p,:));
```

```
            Co(i,j,p)=temp(1,2);
        end
    end
end
M=[];
l=1;
for i=1:4
    for j=1:384
        for k=j:384
            if Co(i,j,k)>=0.998%筛选出来相关系数大于的矩阵关系，存储在0.95中M
                M(l,1)=i;
                M(l,2)=j;
                M(l,3)=k;
                l=l+1;
            end
        end
    end
end

% 所有的算出来
V2=zeros(64,2,384,4);%是求出的所有矩阵V2V
for k=1:4
    container=julei(M,k);
    containerAll(1:size(container,1),1:size(container,2),q,k)=container;
    column=container(:,1);
    V1=zeros(64,2,size(column,1));
    for i=1:size(column)
        a=column(i);
        tmp=suiji(H(:,:,k,a));
        V1(:,:,i)=tmp(:,1:2);%求随机SVD
    end

    V3=zeros(64,2,384);
    for i=1:size(column)
        for j=1:size(container,2)
            if container(i,j)~=0
                V3(:,:,container(i,j))=V1(:,:,i);
            end
        end
```

```
    end

    V2(:,:,:,k)=V3;
end
V2All(:,:,:,:,q)=V2;
% 进行的检验V
temp1=load([filepath,'\Data',num2str(q),'_V','.mat']);
V=temp1.V;
P=zeros(2,4,384);
for i=1:4
    for j=1:384
        for l=1:2
            a=V(:,l,i,j);
            b=V2(:,l,j,i);
            P(l,i,j)=norm(a'*b)/norm(a)*norm(b);
        end
    end
end
errorAll(1,q)=min(min(min(P)));
end
%% 第三题求解写得到的，把Vdata1-给程序运行data6Python
%写入文件excel
V2final=zeros(256,768,6);
for k=1:6
    for j=1:384
        for i=1:4
            tmp=V2All(:,:,j,i,k);
            V2final((i-1)*64+1:i*64,(j-1)*2+1:j*2,k)=tmp;
        end
    end
end
data1=V2final(:,:,1);
data2=V2final(:,:,2);
data3=V2final(:,:,3);
data4=V2final(:,:,4);
data5=V2final(:,:,5);
data6=V2final(:,:,6);
for i=1:6
    save([pwd,'\V',num2str(i)],['data',num2str(i)]);
```

```
end
%% 第三题的求解存储使用计算的的过程pythonW
dataAll=zeros(256,768,6);
for i=1:6
   tmp=load([pwd,'\data',num2str(i),'.mat']);
   dataAll(:,:,i)=tmp.mat;
end
W_new=zeros(64,2,4,384,6);%用来存储所有的数据W
for i=1:6
   W_new(:,:,:,:,i)=cunchuW(dataAll(:,:,i));%升维处理
end
%% 第三题的求解再对进行降维压缩W
WAll=zeros(64,2,4,384,6);%存储解压缩后的W
for k=1:6
W=W_new(:,:,:,:,k);

W2D=zeros(256,768);%先把四维矩阵展成二维矩阵
for i=1:4
   for j=1:384
      W2D((i-1)*64+1:i*64,(j-1)*2+1:j*2)=W(:,:,i,j);
   end
end
%然后分三个256大块进行分解×256SVD
H2DD=zeros(256,768);%存储解压之后的值
for i=1:3
   [A,j]=accumCon(W2D(:,(i-1)*256+1:i*256));
   H2DD(:,(i-1)*256+1:i*256)=A;
end
for i=1:4
   for j=1:384
      WAll(:,:,i,j,k)=H2DD((i-1)*64+1:i*64,(j-1)*2+1:j*2);
   end
end
end
%% 第三题的求解对的检验W
accuracy=zeros(6,1);%检测准确度
filepath='C:\Users\wangy\Desktop\A\W';
for q=1:6
temp1=load([filepath,'\Data',num2str(q),'_W','.mat']);
```

```
W=temp1.W;
P=zeros(2,4,384);
for i=1:4
   for j=1:384
      for l=1:2
         a=W(:,l,i,j);
         b=WAll(:,l,i,j,q);
         P(l,i,j)=norm(a'*b)/norm(a)*norm(b);
      end
   end
end
accuracy(q,1)=min(min(min(P)));
end

function [A,j] = accumCon(A)
[U,S,V]=svd(A);
sum=0;
for i=1:256%算一下累计贡献率
   sum=sum+S(i,i);
end
sum2=0;
con=zeros(1,256);
for i=1:256%算一下累计贡献率
   sum2=sum2+S(i,i);
   con(i)=sum2/sum;%累计贡献率
end
%记录到哪一个奇异值贡献率超过0.9
j=0;
for i=1:256
   if con(i)>=0.99
      j=i;
      break;
   end
end
A=U(:,1:j)*S(1:j,1:j)*V(:,1:j)';
end

function [V1] = suiji(A)
Q=[];
```

```
w=normrnd(0,1,size(A,2),1);
y=A*w;
q=y;
q_new=q/norm(q);
Q=horzcat(Q,q_new);
count1=0;
while norm(A-Q*Q'*A)>0.000001
   w=normrnd(0,1,size(A,2),1);
   y=A*w;
   q=(eye(size(A,1))-Q*Q')*y;
   q_new=q/norm(q);

   Q=horzcat(Q,q_new);
   count1=count1+1;
end
B=Q'*A;
[u,s,V1]=svd(B);

end

function [W_new11] = cunchuW(num1)
W_new11=zeros(64,2,4,384);
for i=1:4
   for j=1:384
      W_new11(:,:,i,j)=num1((i-1)*64+1:i*64,(j-1)*2+1:j*2);
   end
end
end
```

```
def main():
   output()
if __name__ == '__main__':
   main()

from ReadData import *
import numpy as np
import pandas as pd
import openpyxl as op
import os
```

```
import xlwt
import scipy.io

# 读取.文件，并将矩阵储存起来mat
# 表示第几个表data_num
# 表示表中矩阵的行数row
# 表示表中矩阵的列数col
# 返回表中所有矩阵
def read_mat_file(data_num):
    data = np.array(read_data_mat()[data_num])
    # 用于存储整个矩阵
    matrix_V = []
    for i in range(4):
        # 用于存储矩阵的每一行
        matrix_V_line = []
        for j in range(384):
            V_ij = data[i*64:(i+1)*64, j*2:(j+1)*2]
            matrix_V_line.append(V_ij)
        matrix_V.append(matrix_V_line)
    return np.array(matrix_V)

# 获取的方法v_k
# 表示第几个表data_num
# 表示第几列的，不大于，表示第一列KVk3830
def get_v_k(data_num, K):
    data = read_mat_file(data_num)
    v_k = data[0][K]
    for i in range(1,4):
        v_k = np.hstack((v_k,data[i][K]))
    return v_k

# 计算矩阵的共轭转置
def conj_t(A):
    return np.conj(np.array(A)).T

# 奇异值分解
def svd(matrix):
    svd_matrix = []
    u,d,v = np.linalg.svd(np.array(matrix))
```

```
    svd_matrix.append(u)
    svd_matrix.append(d)
    svd_matrix.append(v)
    return svd_matrix

# 利用求解线性方程组svdAx=b
def solve_equations_by_svd(A,b):
    b = np.array(b)
    svd_matrix = svd(A)
    u,d,v = svd_matrix[0],svd_matrix[1],svd_matrix[2]
    u = np.array(u)
    d = np.array(d)
    v = np.array(v)
    y = np.ones([np.shape(v)[0],1])
    b = np.matmul(conj_t(u),b)
    for i in range(np.shape(y)[0]):
        y[i] = b[i] / d[i]
    return np.matmul(conj_t(v),y)

# 简单的矩阵乘法实现A*B
def matrix_multiply(A, B):
    res = [[0] * len(B[0]) for i in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                res[i][j] += A[i][k] * B[k][j]
    return res

# 矩阵乘常数
def matrix_multiply_a(matrix_a,a):
    shape_A = np.shape(matrix_a)
    for i in range(shape_A[0]):
        for j in range(shape_A[1]):
            matrix_a[i][j] *= a
    return matrix_a

# 简单的矩阵相加
def matrix_add(matrix_a, matrix_b):
    rows = len(matrix_a)
```

```
    columns = len(matrix_a[0])
    matrix_c = [list() for i in range(rows)]
    for i in range(rows):
        for j in range(columns):
            matrix_c_temp = matrix_a[i][j] + matrix_b[i][j]
            matrix_c[i].append(matrix_c_temp)
    return matrix_c

# 简单的矩阵相减
def matrix_minus(matrix_a, matrix_b):
    rows = len(matrix_a)
    columns = len(matrix_a[0])
    matrix_c = [list() for i in range(rows)]
    for i in range(rows):
        for j in range(columns):
            matrix_c_temp = matrix_a[i][j] - matrix_b[i][j]
            matrix_c[i].append(matrix_c_temp)
    return matrix_c

# 矩阵的分块
def matrix_divide(matrix_a, row, column):
    length = len(matrix_a)
    matrix_b = [list() for i in range(length // 2)]
    k = 0
    for i in range((row - 1) * length // 2, row * length // 2):
        for j in range((column - 1) * length // 2, column * length //
            2):
            matrix_c_temp = matrix_a[i][j]
            matrix_b[k].append(matrix_c_temp)
        k += 1
    return matrix_b

# 矩阵的拼接
def matrix_merge(matrix_11, matrix_12, matrix_21, matrix_22):
    length = len(matrix_11)
    matrix_all = [list() for i in range(length * 2)]
    for i in range(length):
        matrix_all[i] = list(matrix_11[i]) + list(matrix_12[i])
    for j in range(length):
```

```
            matrix_all[length + j] = list(matrix_21[j]) + list(matrix_22[j])
        return matrix_all

# 算法计算矩阵的乘法Strassen
def strassen(matrix_a, matrix_b):
    rows = len(matrix_a)
    if rows == 1:
        matrix_all = [list() for i in range(rows)]
        matrix_all[0].append(matrix_a[0][0] * matrix_b[0][0])
    elif rows <= 16:
        matrix_all = matrix_multiply(matrix_a, matrix_b)
    else:
        s1 = matrix_minus((matrix_divide(matrix_b, 1, 2)),
            (matrix_divide(matrix_b, 2, 2)))
        s2 = matrix_add((matrix_divide(matrix_a, 1, 1)),
            (matrix_divide(matrix_a, 1, 2)))
        s3 = matrix_add((matrix_divide(matrix_a, 2, 1)),
            (matrix_divide(matrix_a, 2, 2)))
        s4 = matrix_minus((matrix_divide(matrix_b, 2, 1)),
            (matrix_divide(matrix_b, 1, 1)))
        s5 = matrix_add((matrix_divide(matrix_a, 1, 1)),
            (matrix_divide(matrix_a, 2, 2)))
        s6 = matrix_add((matrix_divide(matrix_b, 1, 1)),
            (matrix_divide(matrix_b, 2, 2)))
        s7 = matrix_minus((matrix_divide(matrix_a, 1, 2)),
            (matrix_divide(matrix_a, 2, 2)))
        s8 = matrix_add((matrix_divide(matrix_b, 2, 1)),
            (matrix_divide(matrix_b, 2, 2)))
        s9 = matrix_minus((matrix_divide(matrix_a, 1, 1)),
            (matrix_divide(matrix_a, 2, 1)))
        s10 = matrix_add((matrix_divide(matrix_b, 1, 1)),
            (matrix_divide(matrix_b, 1, 2)))
        p1 = strassen(matrix_divide(matrix_a, 1, 1), s1)
        p2 = strassen(s2, matrix_divide(matrix_b, 2, 2))
        p3 = strassen(s3, matrix_divide(matrix_b, 1, 1))
        p4 = strassen(matrix_divide(matrix_a, 2, 2), s4)
        p5 = strassen(s5, s6)
        p6 = strassen(s7, s8)
        p7 = strassen(s9, s10)
```

```
        c11 = matrix_add(matrix_add(p5, p4), matrix_minus(p6, p2))
        c12 = matrix_add(p1, p2)
        c21 = matrix_add(p3, p4)
        c22 = matrix_minus(matrix_add(p5, p1), matrix_add(p3, p7))
        matrix_all = matrix_merge(c11, c12, c21, c22)
    return matrix_all

# 基于算法的求逆算法Strassen
def inv_strassen(matrix_a):
    rows = len(matrix_a)
    if rows >= 2:
        if rows <= 4 or rows % 2 == 1:
            matrix_all = np.linalg.inv(matrix_a)
        else:
            M1 = inv_strassen(matrix_divide(matrix_a, 1, 1))
            M2 = strassen(matrix_divide(matrix_a, 2, 1), M1)
            M3 = strassen(M1, matrix_divide(matrix_a, 1, 2))
            M4 = strassen(matrix_divide(matrix_a, 2, 1), M3)
            M5 = matrix_minus(M4, matrix_divide(matrix_a, 2, 2))
            M6 = inv_strassen(M5)
            C12 = strassen(M3, M6)
            C21 = strassen(M6, M2)
            M7 = strassen(M3, C21)
            C11 = matrix_minus(M1, M7)
            C22 = np.array(M6).dot(-1)
            matrix_all = matrix_merge(C11, C12, C21, C22)
        return matrix_all
    else:
        return np.linalg.inv(matrix_a)


# 基于算法的求逆优化算法Strassen
def inv_strassen_neo(matrix_a):
    matrix_a = np.array(matrix_a)
    rows = len(matrix_a)
    cols = len(matrix_a[0])
    if (rows <= 4) or (rows % 2 == 1 or cols % 2 == 1):
        matrix_a_np = np.array(matrix_a)
        matrix_all = np.linalg.inv(matrix_a_np)
```

```
    else:
        M1 = inv_strassen_neo(matrix_divide(matrix_a, 1, 1))
        M2 = strassen(matrix_divide(matrix_a, 2, 1), M1)
        M3 = conj_t(M2)
        M4 = strassen(matrix_divide(matrix_a, 2, 1), M3)
        M5 = matrix_minus(M4, matrix_divide(matrix_a, 2, 2))
        M6 = inv_strassen_neo(M5)
        C12 = strassen(M3, M6)
        C21 = conj_t(C12)
        M7 = strassen(M3, C21)
        C11 = matrix_minus(M1, M7)
        C22 = np.array(M6).dot(-1)
        matrix_all = matrix_merge(C11, C12, C21, C22)
    return matrix_all

# Coppersmith-算法Winograd
def coppersmith_winograd(matrix_a, matrix_b):
    rows = len(matrix_a)
    if rows == 1:
        matrix_all = [list() for i in range(rows)]
        matrix_all[0].append(matrix_a[0][0] * matrix_b[0][0])
    elif rows <= 16:
        matrix_all = np.matmul(matrix_a, matrix_b)
    else:
        S1 = matrix_add(matrix_divide(matrix_a, 2,
            1),matrix_divide(matrix_a, 2, 2))
        S2 = matrix_minus(S1,matrix_divide(matrix_a, 1, 1))
        S3 = matrix_minus(matrix_divide(matrix_a, 1,
            1),matrix_divide(matrix_a, 2, 1))
        S4 = matrix_minus(matrix_divide(matrix_a, 1, 2),S2)
        T1 = matrix_minus(matrix_divide(matrix_b, 1,
            2),matrix_divide(matrix_b, 1, 1))
        T2 = matrix_minus(matrix_divide(matrix_b, 2, 2),T1)
        T3 = matrix_minus(matrix_divide(matrix_b, 2,
            2),matrix_divide(matrix_b, 1, 2))
        T4 = matrix_minus(T2,matrix_divide(matrix_b, 2, 1))

        M1 = coppersmith_winograd(matrix_divide(matrix_a, 1,
            1),matrix_divide(matrix_b, 1, 1))
```

```
        M2 = coppersmith_winograd(matrix_divide(matrix_a, 1,
           2),matrix_divide(matrix_b, 2, 1))
        M3 = coppersmith_winograd(S4,matrix_divide(matrix_b, 2, 2))
        M4 = coppersmith_winograd(matrix_divide(matrix_a, 2, 2),T4)
        M5 = coppersmith_winograd(S1,T1)
        M6 = coppersmith_winograd(S2,T2)
        M7 = coppersmith_winograd(S3,T3)

        U1 = matrix_add(M1,M2)
        U2 = matrix_add(M1,M6)
        U3 = matrix_add(U2,M7)
        U4 = matrix_add(U2,M5)
        U5 = matrix_add(U4,M3)
        U6 = matrix_minus(U3,M4)
        U7 = matrix_add(U3,M5)
        matrix_all = matrix_merge(U1, U5, U6, U7)
    return matrix_all

# 基于Coppersmith-算法的求逆算法Winograd
def inv_coppersmith_winograd(matrix_a):
    matrix_a = np.array(matrix_a)
    rows = len(matrix_a)
    cols = len(matrix_a[0])
    if (rows <= 4) or (rows % 2 == 1 or cols % 2 == 1):
        matrix_a_np = np.array(matrix_a)
        matrix_all = np.linalg.inv(matrix_a_np)
    else:
        M1 = inv_coppersmith_winograd(matrix_divide(matrix_a, 1, 1))
        M2 = coppersmith_winograd(matrix_divide(matrix_a, 2, 1),M1)
        M3 = coppersmith_winograd(M1,matrix_divide(matrix_a, 1, 2))
        M4 = coppersmith_winograd(matrix_divide(matrix_a, 2, 1),M3)
        M5 = matrix_minus(M4,matrix_divide(matrix_a, 2, 2))
        M6 = inv_coppersmith_winograd(M5)
        C12 = coppersmith_winograd(M3,M6)
        C21 = coppersmith_winograd(M6,M2)
        M7 = coppersmith_winograd(M3,C21)
        C11 = matrix_minus(M1,M7)
        C22 = np.array(M6).dot(-1)
        matrix_all = matrix_merge(C11, C12, C21, C22)
```

```
    return matrix_all

# 计算的方法Wk
# 将拆分后存储到一个列表中Wk
def compute_w_k(matrix_v_k):
    list_w_k = []
    matrix_v_k_mul = np.matmul(conj_t(matrix_v_k), matrix_v_k)
    o2i = matrix_multiply_a(np.eye(len(matrix_v_k_mul)), 0.01)
    add = matrix_add(matrix_v_k_mul, o2i)
    u = inv_coppersmith_winograd(add)
    w_k = np.array(np.matmul(matrix_v_k, u))
    for i in range(4):
        list_w_k.append(w_k[:,i*2:(i+1)*2])
    return list_w_k

# 批量计算的方法Wk
# data_one_base = data_all_six_base[i]
# W_ij = data_one_base[j][i]
def compute_all_w_k():
    # 用来储存所有个数据库的矩阵6
    data_all_six_base = []
    # 读取个6.文件mat
    for data_num in range(6):
        # 用于储存一个数据库中的数据
        data_one_base = []
        # 数据库的第几列即第几个(V_K)
        for col in range(384):
            # 用于储存数据库的一列Wk
            data_col = []
            v_k = get_v_k(data_num,col)
            list_w_k = compute_w_k(v_k)
            for i in range(4):
                data_col.append(list_w_k[i])
            data_one_base.append(data_col)
        data_all_six_base.append(data_one_base)
    return data_all_six_base

# 将中的数据组成一个大数组data_one_base
def combine_all_matrix(data_one_base):
```

```
    list_all = []
    for i in range(4):
        list_row = np.array(data_one_base[0][i])
        for j in range(1,384):
            list_row =
                np.hstack((list_row,np.array(data_one_base[j][i])))
        list_all.append(list_row)
    matrix_1 = np.vstack((np.array(list_all[0]),np.array(list_all[1])))
    matrix_2 = np.vstack((np.array(list_all[2]),np.array(list_all[3])))
    matrix = np.vstack((matrix_1,matrix_2))
    return np.array(matrix)

# 将所有数据输出
def output():
    data_all_six_base = compute_all_w_k()
    for k in range(6):
        data_one_base_matrix = combine_all_matrix(data_all_six_base[k])
        matrix = {'mat': data_one_base_matrix}
        scipy.io.savemat(os.getcwd() + '\\data' + str(k+1) + '.mat',
            matrix)

import csv
import numpy as np
from scipy.io import loadmat
# 读取数据
def read_data(filepath):
    matrix = []
    with open(filepath,'r') as csv_file:
        reader = csv.reader(csv_file)
        for line in reader:
            line_matrix = []
            for line_data in line:
                line_data = line_data.replace('i','j')
                myoutput = complex(line_data)
                line_matrix.append(myoutput)
            matrix.append(line_matrix)
    return matrix

# 读取.数据mat
```

```
def read_data_mat():
    data = []
    data1 = loadmat('V1.mat')
    data2 = loadmat('V2.mat')
    data3 = loadmat('V3.mat')
    data4 = loadmat('V4.mat')
    data5 = loadmat('V5.mat')
    data6 = loadmat('V6.mat')
    data.append(data1['data1'])
    data.append(data2['data2'])
    data.append(data3['data3'])
    data.append(data4['data4'])
    data.append(data5['data5'])
    data.append(data6['data6'])
    return data
```

```
//此处为结点、树和编码的代码Huffmanjava
//Huffman 结点定义
package ch05;

public class HuffmanNode {
 public int weight;
 public int flag;
 public HuffmanNode parent,lchild,rchild;
 public HuffmanNode()
 {
  this(0);
 }
 public HuffmanNode(int weight)
 {
  this.weight=weight;
  flag=0;
  parent=lchild=rchild=null;
 }

}

//树及编码HuffmanHuffman
package ch05;
```

```
public class HuffmanTree {
 public int[][] huffmancoding(int[] W)
 {
  int n=W.length;
  int m=2*n-1;
  HuffmanNode[] HN=new HuffmanNode[m];
  int i;
  for(i=0;i<n;i++)
  {
   HN[i]=new HuffmanNode(W[i]);
  }
  for(i=n;i<m;i++)
  {
   HuffmanNode min1=selectMin(HN,i-1);
   min1.flag=1;
   HuffmanNode min2=selectMin(HN,i-1);
   min2.flag=1;
   HN[i]=new HuffmanNode();
   min1.parent=HN[i];
   min2.parent=HN[i];
   HN[i].lchild=min1;
   HN[i].rchild=min2;
   HN[i].weight=min1.weight+min2.weight;
  }
  int[][] HuffCode=new int[n][n];
  for(int j=0;j<n;j++)
  {
   int start=n-1;
   for(HuffmanNode c=HN[j],p=c.parent;p!=null;c=p,p=p.parent)
   {
    if(p.lchild.equals(c))
    {
     HuffCode[j][start--]=0;
    }
    else
    {
     HuffCode[j][start--]=1;
    }
```

```
   HuffCode[j][start]=-1;
  }
 }
 return HuffCode;
}
private HuffmanNode selectMin(HuffmanNode[] HN,int end)
{
 HuffmanNode min=HN[end];
 for(int i=0;i<=end;i++)
 {
  HuffmanNode h=HN[i];
  if(h.flag==0&&h.weight<min.weight)
   min=h;
 }
 return min;
}

public static void main(String[] args) {
 // TODO Auto-generated method stub
 int [] W= {23,11,5,3,29,14,7,8};
 HuffmanTree T=new HuffmanTree();
 int[][] HN=T.huffmancoding(W);
 System.out.println("哈夫曼编码为:");
 for(int i=0;i<HN.length;i++)
 {
  System.out.print(W[i]+" ");
  for(int j=0;j<HN[i].length;j++)
  {
   if(HN[i][j]==-1)
   {
    for(int k=j+1;k<HN[i].length;k++)
     System.out.print(HN[i][k]);
    System.out.print(" ");
    break;
   }
  }
  System.out.println();
 }
```

```
 }

}
```