arXiv:2303.10784v1 [cs.DS] 19 Mar 2023

稀疏偏迹

Julio Candanedo

物理系，亚利桑那州立大学，坦佩，亚利桑那州
85287，美国
电子邮箱：jcandane@asu.edu

Abstract: 矩阵以及更一般的多维数组，构成了计算研究的基础。本文展示了通过对稀疏数组进行部分求迹/张量缩并来提升计算效率。已有研究表明，稀疏数组实际上由 3 个稠密数组（稠密形状、索引数组和数据数组）构成。通过对这些组成数组进行稠密数组操作，可以确定最终的部分求迹。由于计算数组广泛应用于各种不同的研究中，这些方法具有广泛的适用性。

目录

1 引言 **1**

2 规范格式 12.1 良序 12.2 复合、子索引和排列 22.3 数组标记 22.4 笛卡尔积 22.5 直积 22.5.1 稠密数组/张量 22.5.2 稀疏数组/张量 42.6 内部交集/切片 4

3 重塑3.1 修改后的形状 **5**
3.2 元组到整数 3.3 整数 5
到元组 3.4 元组到元组 6
3.5 重塑 6
7
7

4 直接交集 **7**
4.1 鞋带 84.2 多重交集（超边） 84.3 边链 104.3.1 树搜索与光线追踪 10

5 满射 **10**5.0.1 稠密外部映射 10

6 二数组纯稀疏算法 **11**

7 二阶数组：稀疏与稠密迹13

8 结果138.1 矩阵乘积148.2 矩阵乘积算子迹148.3 投影纠缠对算子迹15

9 总结 **15**

稀疏数组16
A.1 属性 16
A.1.1 形状 17
A.1.2 索引 17
A.1.3 data 17
A.2 稀疏性 17
A.3 良序 17

B 字典序 **18**
B.1 元组数组 18B.2 元组间比较 18B.2.1 元组间相等 18B.2.2 元组到数字 18

B.3 字典序的定义 19
B.3.1 字典序排序算法 19
B.3.2 计算时间复杂度 20
B.3.3 定理 20

---

1 引言

计算数组在科学领域无处不在，用于数据存储，近年来也用于数据处理。在许多情况下，待存储或处理的数据具有稀疏性这一特征。从计算角度来看，这意味着许多元素恒为零。稀疏矩阵早已得到应用，但直到近年来，随着机器学习的发展及其在[5] 和 [3], 等领域的应用，稀疏张量 才被定义。然而值得注意的是，许多计算研究中使用的偏迹算法尚未扩展到这种数据结构：稀疏张量/稀疏数组。本文弥补了这一能力缺失。至少在理论层面如此，因为该实现采用高级计算语言 Python 编写，并可通过命令行使用 Python 索引进行访问，命令为：pip install pyftt（ftt 是暂定名称：快速张量迹，快速 是因为它随稀疏元素数量线性扩展；该名称借鉴了快速傅里叶变换的缩写）。结果表明，对于稀疏数组，该实现的速度可能超过 [2]的 einsum，其性能与前述数组的稀疏性直接相关。

2 规范格式

2.1 良序

稀疏数组只有在其索引数组按行排列的元素均唯一时，才能忠实地（双射地）表示稠密数组。为验证任意

定义的索引数组是否满足这一条件，会将其按字典序规范排序（参见附录§B），以确保其确实是良序。 良序是指严格排序的列表，即使用关系 $<$。使得对于所有条目 a， $b \in A$ (A 它是一个列表/数组)，其元素索引为 i 和 j ，那么 $a[i] < b[j]$对于 $i < j$。一种较弱的说法是部分良序，这表示可能存在重复项，但仍保持字典序，即 $\leq$。讨论对重复元组元素求和时，我们将数组称为良序，同时保持字典序。

2.2 复合、子索引与排列

给定两个一维数组 A 和 B，我们可以用其中一个对另一个进行复合/子索引。从数学上讲，这可以用带有 $A \circ B$的函数表示；或者，我们也可以使用方括号记法，例如 $A[B]$。如果 B 可以与 A建立双射，且 B中的所有元素都唯一，那么这称为排列。通常，为了对数组排序（例如 A），我们会确定排列数组 B，使得复合后 $A[B]$的结果是有序的。这在讨论稀疏数组时很有用，因为它们具有相互连接的数组结构。

2.3 数组标记

我们可以为张量/数组的每个轴指定一个字符/字母。例如，不失一般性地说，如果 A 是一个四维数组，那么 $A[i, j, k, \ell]$ 表示对 A的标记，标记为 $\{i, j, k, \ell\}$。这些字符完全是任意的，仅用于将某些索引/轴与其他索引/轴对应起来，从而创建边（更一般地说是超边，使得每条边都与一种字符相关联）。

2.4 笛卡尔积

笛卡尔积 是由 A 中的所有元素（第一个元组项）与 B 中的所有元素（第二个元组项）组成的所有可能的二元组。笛卡尔积可推广到多个 n 元组列表的乘积，从而得到一个由 n个元组构成的元组列表。元组的元组可以直接简化为更高阶的 m元组，因此 $m > n$。此外，关于此操作的详细信息见 [1] 第 5 节。最初，这是集合论中的一种运算；应用于列表/数组时，只有当两个原始数组都是良序的，它们才会保持良序性 。如果考虑偏序数组上的笛卡尔积，其结果既不是良序的，也不是偏序的。我们天真地希望该乘积能生成一个偏序元组列表。无需诉诸天真的排序，也可以进行排列校正。

2.5 直积

2.5.1 稠密数组/张量

数组上最一般的运算或许是直积（又称 张量积或克罗内克积），即将一个数组中的所有元素分别与另一个数组中的所有元素相乘。结果是一个新的张量/数组，其秩等于各组成部分秩的总和。以下是一个例子

对两个二维数组进行运算：

$$A_{\mu\nu} = \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0n} \\ A_{10} & A_{11} & \cdots & A_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m0} & A_{m1} & \cdots & A_{mn} \end{pmatrix} \qquad B_{\kappa\lambda} = \begin{pmatrix} B_{00} & B_{01} & \cdots & B_{0\ell} \\ B_{10} & B_{11} & \cdots & B_{1\ell} \\ \vdots & \vdots & \ddots & \vdots \\ B_{k0} & B_{k1} & \cdots & B_{k\ell} \end{pmatrix} \quad .$$

那么，直积 $\otimes$ 的明确定义如下：

$$A_{\mu\nu} \otimes B_{\kappa\lambda} = \begin{pmatrix} A_{00} \begin{pmatrix} B_{00} & B_{01} & \cdots & B_{0\ell} \\ B_{10} & B_{11} & \cdots & B_{1\ell} \\ \vdots & \vdots & \ddots & \vdots \\ B_{k0} & B_{k1} & \cdots & B_{k\ell} \end{pmatrix} & \cdots & \cdots & A_{0n} \begin{pmatrix} B_{00} & B_{01} & \cdots & B_{0\ell} \\ B_{10} & B_{11} & \cdots & B_{1\ell} \\ \vdots & \vdots & \ddots & \vdots \\ B_{k0} & B_{k1} & \cdots & B_{k\ell} \end{pmatrix} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ A_{m0} \begin{pmatrix} B_{00} & B_{01} & \cdots & B_{0\ell} \\ B_{10} & B_{11} & \cdots & B_{1\ell} \\ \vdots & \vdots & \ddots & \vdots \\ B_{k0} & B_{k1} & \cdots & B_{k\ell} \end{pmatrix} & \cdots & \cdots & A_{mn} \begin{pmatrix} B_{00} & B_{01} & \cdots & B_{0\ell} \\ B_{10} & B_{11} & \cdots & B_{1\ell} \\ \vdots & \vdots & \ddots & \vdots \\ B_{k0} & B_{k1} & \cdots & B_{k\ell} \end{pmatrix} \end{pmatrix} \quad .$$

这可以用符号表示为：

$$A\left[\mu,\nu\right] \otimes B\left[\kappa,\lambda\right] = AB\left[\mu,\nu,\kappa,\lambda\right] \qquad .$$

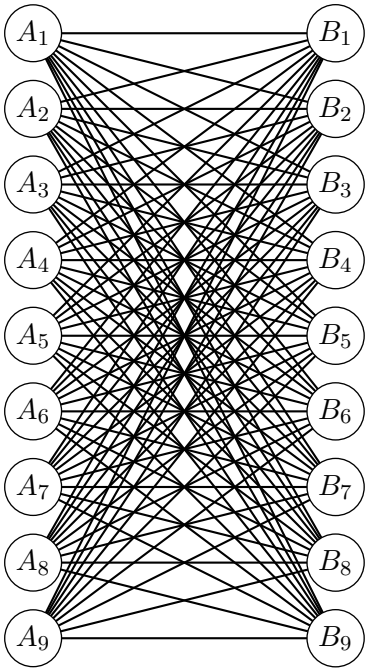
n 维数组的一般直积同样表示为：

$$A^{(0)}\left[m_0\right] \otimes A^{(1)}\left[m_1\right] \otimes \cdots \otimes A^{(n)}\left[m_n\right] = C\left[m_0, m_1, \cdots, m_n\right] \qquad .$$

这一运算可以用网络/图表示。如果为 A 中的每个数据项分配一个节点，并对 B 也进行同样的处理，那么将它们连接起来的边所对应的乘积，就由完全二部图表示。对于涉及

n 张量/数组的一般情形，所得图就是完全-n-部图。

$$\begin{pmatrix} A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 \\ A_7 & A_8 & A_9 \end{pmatrix} \otimes \begin{pmatrix} B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 \\ B_7 & B_8 & B_9 \end{pmatrix} \quad \Rightarrow$$



.

2.5.2 稀疏数组/张量

假设我们有两个良序的 A & B 稀疏数组，那么它们的直积为：对各自的元组列表取二元笛卡尔积，并对其数据取重塑/展平后的直积，同时拼接形状元组：

$$\begin{aligned} A^{\text{index}} \times B^{\text{index}} &= C^{\text{index}} \\ \text{reshape}\left(A^{\text{data}} \otimes B^{\text{data}}\right) &= C^{\text{data}} \\ A^{\text{shape}} \,||\, B^{\text{shape}} &= C^{\text{shape}} \end{aligned} \quad .$$

这可以直接推广到 n-稀疏数组， $A^{(n)}$:

$$\begin{aligned} A^{(0)}_{\text{index}} \times A^{(1)}_{\text{index}} \times \cdots \times A^{(n)}_{\text{index}} &= C_{\text{index}} \\ \text{reshape}\left(A^{(0)}_{\text{data}} \otimes A^{(1)}_{\text{data}} \otimes \cdots \otimes A^{(n)}_{\text{data}}\right) &= C_{\text{data}} \\ A^{(0)}_{\text{shape}} \,||\, A^{(1)}_{\text{shape}} \,||\, \cdots \,||\, A^{(n)}_{\text{shape}} &= C_{\text{shape}} \end{aligned} \quad .$$

2.6 内部交集/切片

现在假设给定一个张量/数组，我们想要通过 ¨§2.3 中提到的标签，将某些列/轴彼此对应起来。这种减少数组自由度的操作称为切片或内部求交。由于这些都是内部操作，最好先执行，因为内部求交只会使规模缩小。遗憾的是，如果相交的列彼此不相邻，或者内部求交索引位于内部，就必须重新良序排列（至少要从第一个内部交集列/索引/轴开始，对其余部分重新排列）。此操作需要对给定稀疏数组的索引数组中的所有行进行逐元素搜索。

3 重塑

本节将介绍如何像处理稠密数组那样重塑稀疏数组。重塑 是合并或划分给定数组的轴。与在实空间内存中连续存储数据的稠密数组不同，稀疏数组将数据存储在 3 个稠密数组中。其中两个稠密数组：形状和元组数组，用于控制数据的结构；本节将介绍如何对这两个数组执行稠密运算，以改变稀疏数据的结构。稀疏数据的结构由元组数组管理。在本文的讨论中，设 A 和 B 为元组数组： $A,\ B \subset \prod_i \mathbb{Z}_i$ （即 由 $\mathbb{Z}$的多个整数数组子集构成的笛卡尔积）。因此，我们想定义一个函数 h，将一个元组数组映射到另一个元组数组：

$$h : A \longrightarrow B \qquad .$$

我们选择采取一个中间步骤，转而解决更简单的问题：将元组映射为整数（§3.2）以及将整数映射为元组（§3.3）：

$$A \longrightarrow \mathbb{Z}_N \longrightarrow B \qquad .$$

最终，我们需要一个有限配对函数，令 $\mathbb{Z}_N \subset \mathbb{N}$ 满足 $|\mathbb{Z}_N| \neq \infty$ （即$\mathbb{Z}_N$ 具有有限个元素），那么配对函数为：

$$\begin{aligned} f &: A \longrightarrow \mathbb{Z}_N \\ g &: \mathbb{Z}_N \longrightarrow A \end{aligned} \qquad ,$$

满足： $f(g(\mathbb{Z}_N)) = \mathbb{Z}_N$或 $g(f(A)) = A$。稠密元素的总数为： $N = \prod_i s[i]$。整数范围（按顺序）为： $\mathbb{Z}_N = \begin{pmatrix} 0 & 1 & 2 & \cdots & N-1 \end{pmatrix}$。此外，值得注意的是，这里定义的重塑函数不依赖任何类型的排序。配对函数有很多种，我们选择了 *C* 语言重塑顺序 [1]。

3.1 修改后的形状

对于元组到整数和整数到元组映射，我们都需要将形状数组 $s[i]$ 修改为辅助数组 $\tilde{s}[i]$。在这两种情况下，形状数组分别表示：当前数组形状与目标数组形状的异或。这需要以特定方式对形状数组进行排列，

[1]令 i 表示行， j 表示 A 的列，则：

$$A_{ij} = A[i,j] = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \xrightarrow{\text{“C”-order}} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix} = A[i'] \qquad .$$

并计算累积乘积。修改 $s[i]$的方法如下[2]:

$$s_i = \begin{pmatrix} s_0, s_1, s_2, s_3, \cdots, s_{n-1} & s_n \end{pmatrix}$$
$$\tilde{s}_i = \begin{pmatrix} \prod_{i\in[1,n]} s_i & \prod_{i\in[2,n]} s_i & \prod_{i\in[3,n]} s_i & \cdots & s_{n-1}s_n & s_n & 1 \end{pmatrix}$$

因此，单射性源于该辅助形状数组严格递减，也就是说， $\tilde{s}[i] > \tilde{s}[j]$ 对于 $j > i$。而满射性则源于辅助形状数组的精确间距。假设我们有： $a_i = \begin{pmatrix} a_0 & a_1 & a_2 & \cdots & a_n \end{pmatrix}$，其中每个 a_i 值的上限为 s_i:

$$z_i = a_0 \left(\prod_{i\in[1,n]} s_i \right) + a_1 \left(\prod_{i\in[2,n]} s_i \right) + a_2 \left(\prod_{i\in[3,n]} s_i \right) + \cdots + a_{n-1}s_n + a_n \quad .$$

3.2 元组到整数

元组到整数的转换可以通过稠密矩阵-向量乘积实现，使用适当修改后的形状（s_i）、元组数组 A^i_I （其中包含 i 元组索引）以及整数数组 a_I:

$$a_I = A^i_I \tilde{s}_i \quad .$$

因此，该运算的时间复杂度为 $\mathcal{O} \sim nN$，其中 n 是列（轴）的数量，而 N 是稀疏元素的数量；通常情况下，这将呈线性扩展$n \ll N$。

3.3 整数到元组

给定修改后的形状数组，参见§3.1，我们可以执行以下稠密运算，将整数数组（或 1 元组）转换为新的元组数组:

$$\begin{aligned} A[I, i] &= A^I \otimes \mathbf{1}^i \\ A[:, 1:] &= A[:, 1:] \mod \tilde{s}[:-1] \\ A[I, i] &= \left\lfloor \frac{A[I, i]}{\tilde{s}[i]} \right\rfloor \\ A[i, I] &= A[I, i] \quad . \end{aligned}$$

其中包括：将一个整数数组与一个由全 1 构成、大小与所需元组相同的数组进行整数直积，$\mathbf{1}^i$。结果是 A^{Ii} 一个二维

[2]实际操作中，可以这样做:

$s[0] = 1 s[i] =$ 翻转$(s[i])$

$s[i] =$ 滚动$(s[i],\ 1)$

$s[i] =$ 累积乘积$(s[i]) s[i] =$ 翻转$(s[i])$ 。

$(s_0, s_1, \cdots, s_{n-1}, s_n) \to (s_n, s_{n-1}, \cdots, s_1, s_0)$

通过右移进行排列

数组。然后，沿第二轴对修改形状后的数组应用模除（余数）（沿该轴的一个子集条目，即A 的第一个/第零个条目[:,0] 会被忽略，形状的最后一个元素也会被忽略）。接着，沿同一轴应用补函数——整数除法，但这次作用于所有元素。最后，将所得数组转置，以获得规范 形式的元组数组（稀疏数组数据条目索引之后的轴¨），因此 $A[i, I]$。与§3.2类似，此操作的时间复杂度呈线性增长： $\mathcal{O} \sim nN + nN + nN + 1 \sim 3nN$。

3.4 元组到元组

很容易证明，使用中间整数将一个元组映射到另一个大小不同的元组，是一个双射。这是因为众所周知，双射函数的复合本身也是双射。

3.5 重塑

使用本节介绍的技术，可以将任何稀疏数组转换为稀疏矩阵。因此，我们可以将稀疏矩阵算法应用于稀疏数组。

4 直接交集

现在，我们希望实现交集间运算，即数组/张量之间元素的交集。直接交集是两个（或更多）张量/数组中相交元素的直积，并删除非相交元素。或者说，它是完整直积的某个特定切片/内部交。从视觉上看，对于两个数组，这可以表示为许多完全二分图的不连通并，每个完全二分图都作用于相交元素。

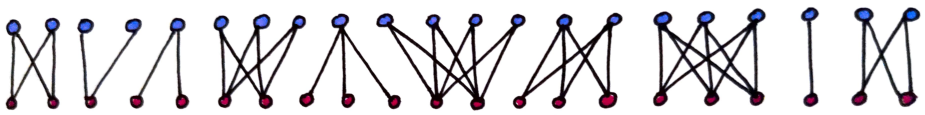


图1。此图展示了两个数组（红色和蓝色）的元素（即 节点）之间直接交集（所有边的集合）的示例，该交集由若干互不相交的完全二分图之和表示。

在集合论中，用于确定同时属于两个集合的子集的运算称为交集。设 A 和 B 为集合，则 $A \cap B = C \subseteq A \,\&\, B$。在我们的讨论中，交集很有用，因为数组间偏迹只会针对数值相同的内部条目进行。对于数组（具有任意顺序、且可能含有重复条目的集合），我们需要的是（来自各数组的）那些对交集元素有贡献的索引，也就是自变量交。也就是说，给定两个数组 A 和B，以及它们之间的 1 个相交元素 C， I_A 和 I_B 表示该元素出现的索引

在 A 和 B中出现的位置， C。因此，复合/子索引到 A 和 B中：

$$A[I_A] = B[I_B] = C \qquad .$$

现在，如果 A和 B有多个相交元素， $C^0, C^1, \cdots, C^n$，那么我们需要一个索引列表 $[I_A^{(0)}, I_A^{(1)}, \cdots, I_A^{(n)}]$，以及 B的索引所对应的 $[I_B^{(0)}, I_B^{(1)}, \cdots, I_B^{(n)}]$，使得 $A[I_A^{(j)}] = B[I_B^{(j)}] = C^{(j)}$。如果每个相交元素都存在重复项，那么每个 $I_A^{(i)}$ 本身都是索引列表。与条目不同，索引列表是唯一的，因此如果它们是良序的，其笛卡尔积也将是良序的。由此得到两个数组的直接交集的最终结果：所有索引相同的索引对（二元组）。由于数组对重复元素没有限制，因此将集合交集自然扩展到数组上，就是直接交集。

到目前为止，我们一直在考虑两个无标签数组（宽度相同）。如果 A 和B分别带有标签 ℓ_A 和 ℓ_B，那么标签交集 $\ell_A \cap \ell_B = e$会产生一条边。这条边决定了 A 和 B 中哪些列会进行直接交集。对于 n个数组的一般结果是得到另一个稀疏数组，其中列枚举了参与运算的数组，而行表示对应数组中的行索引。

$$\bigcirc\!\!\!\!\bigcap_{\text{edges}} A^{(n)}$$

结果是所有可能对偏迹有贡献的乘积。

4.1 鞋带

假设有两个数组 A 和 B，我们想计算它们的直接交集。为此，需要对每个 A 和 B进行排序，以便在两个数组上使用二分查找。我们先从其中一个数组开始，通过左侧和右侧二分查找，在另一个数组中搜索第一个元素。如果找到，则左右两侧的搜索会得出匹配值的范围；如果未找到，则左右两侧的搜索会匹配到一个值，该值称为插入点。这种区间搜索可以在两个数组之间交替进行。如果假设两个数组都已排序，那么在两个数组之间执行二分查找时，若找到的元素跨过了许多元素，则根据定义，这些元素都不会同时存在于两个数组中。即使搜索的元素未找到，这一点也成立，因为插入点会为后续搜索提供另一项约束。此外，这些数组不必包含唯一元素，只需经过排序即可。而且，所有后续的二分查找都会受到越来越多的约束。因此，如果 A 和 B的交集中有 M 个唯一条目，$|A \cap B| = M$，且每个数组总共有 N 个条目，那么该算法的时间复杂度应为 $\mathcal{O} \sim M \log N$。

4.2 多重交集（超边）

前述算法适用于二节点边，也就是两个二数组之间的情况。它可以很容易地推广到 n节点超边，也就是位于 n个数组之间的情况。这里的主要思路是，将数组按照任意的一维[3] 顺序排列，并选取两个枢轴数组。这两个枢轴

[3]假设我们有图 2 中所示的 4 个数组，按 $ABCD$、 $ACDB$、 $CDBA$ 的顺序排列，或按任意排列，都会得到相同的结果。

数组比较特殊，因为它们负责引导搜索。假设我们从一个边数组A开始搜索某个元素 $a \in A$，我们先在相邻数组中搜索；如果找到，就继续在它的相邻数组中搜索，如此重复，直到最后一个数组（一个边数组， D）。如果搜索贯穿所有数组，那么就将它们的相交元素直接求笛卡尔积并保存。否则（如果任意一次搜索失败），则在 B 中搜索 $a \in A$ 的某个插入点 i。这个插入点决定下一次搜索的位置 $d = D[i]$，然后算法朝着边数组 A的方向重复执行。连接边与边的算法可以称为扫描。

该算法如图 2 所示，展示了 4 个元组列表/数组的情况。图中所有线条（点线、虚线、实线）均表示二分查找 $\mathcal{O} \sim \log N$，虚线表示“失败”的搜索，得到插入点（两次搜索都得到同一位置）。搜索失败后，直接比较一条边与另一条边，该边中的下一个唯一元素（或多个唯一元素）开始新的搜索。点线和实线分别计算左侧和右侧搜索，对应于成功的搜索。当到达任意数组的末尾时，算法即告完成。

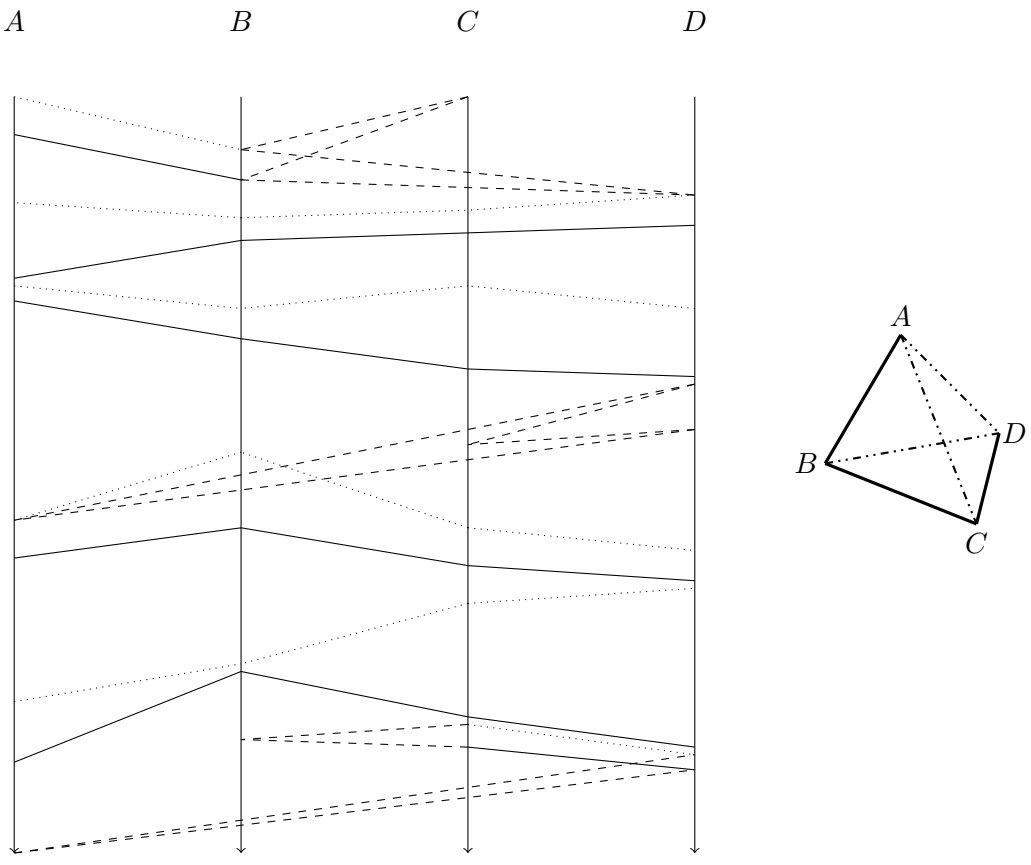


图 2。（左）在 4 个列表/数组上应用编织算法的示例。（右）展示收缩超边和搜索路径（如左侧所示），实线表示该路径。

4.3 边链

现在假设我们有 n 个数组，每个数组包含许多边（或超边），也就是说，如标签交集所示的网络/图。很容易证明，要对整个网络的直接交集作出贡献，任何给定元素都必须通过网络中的所有边与某些其他元素相连。必须存在一条遍历所有边的路径（不一定是欧拉路径）。这条路径定义了一个直积以及一个属于直接交集的元素。

我们可以对网络中的每条边求直接交集，从而得到相交的稀疏数组。只对.....求直接交集。通过依次应用直接交集，可以得到整个网络的直接交集（所有元素，即 n 元组，都是唯一的）。

4.3.1 树搜索与射线追踪

我们需要一种方法，以连通的方式依次遍历网络中的所有边，这称为路径 γ。这条路径从一条特殊的枢轴边开始。我们可以沿着这条路径启动一次侦察二分搜索，并判断它能否一路遍历整个网络。如果依次遍历了整个网络（每次搜索都至少得到一个元素），那么整个树积就计算完成了。否则，枢轴边中的下一个相交元素会启动一次新的侦察二分搜索。尝试完枢轴边中的所有相交元素后，算法即告完成。树积是单边直积。乘积的数量等于树中最大的层数。

5 满射

到目前为止，直接交集计算的是与有序数组相关的相交索引。但这并不完全符合我们的需求，因为参与交集的内部索引通常与外部索引的顺序不同。实际情况并非如此，因为外部索引往往才是我们关注的目标，而我们必须直接对内部索引求交。因此，我们希望根据相应的有序外部索引来获取直接交集。

5.0.1 稠密外部映射

一个重要但微妙的要素，是外部索引与内部索引之间的满射。假设我们将内部索引和外部索引（F）一起排序，使内部索引（D）按字典序排列 $D^{\leq}$, $D^{\leq}||F$ （令 $||$ 表示拼接），一般而言，这意味着外部索引 F 并未排序。由于外部索引决定了我们的结果，因此它们必须是良序的，进而必须有序。这可以通过一个排列 i 来实现，从而得到偏序。不过，我们还必须进一步确定偏序数组中哪些索引是唯一的；这些索引由一个排序数组 u 给出。将其满射复合／子索引到 $F^{\leq}$ 后，便得到$F^{<}$。这个数组 u 不能直接用于 $D^{\leq}$ 中的元素，因为它们的顺序和大小不同。因此，我们可以生成一个顺序与 $F^{\leq}$ 相同的辅助数组，但是

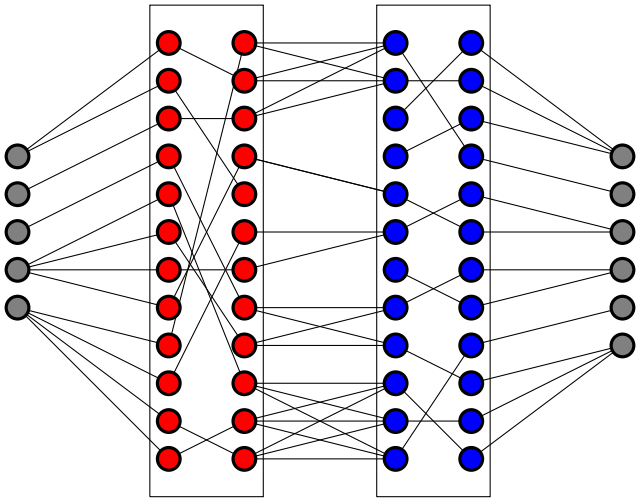


图3。此图展示了两个满射、两种排序和一个直接交集映射之间的完整操作。

其中的条目根据其在 $F^{\leq}$ 中对应的唯一条目的索引确定。例如，不失一般性地说， $F^{\leq}$ 中第 8 个唯一元素对应的所有条目都是 8。这个辅助数组与 $F^{\leq}$ 双射对应，称为 $f^{\leq}$。排序置换数组 i^{-1} 的逆可以复合进来，从而将其还原为 $D^{\leq}$ 的顺序。

$$D^{\leq}||F \xrightarrow{i} F^{\leq} \xrightarrow{u} F^{<}$$
$$F^{\leq} \xrightarrow{1-1} f^{\leq} \xrightarrow{i^{-1}} f \quad ,$$

注意， f 的顺序与 F的原始顺序一致，因此其排序方式与 $D^{\leq}$相同。因此， f 为从 $D^{\leq}$中的每个元素到 $F^{<}$中某个元素的满射：

$$f : D^{\leq} \to F^{<} \quad .$$

这一点很重要，因为外部索引给出的结果必须是良序的，而它们可能属于多个乘积。因此可以证明，若$d^{(0)} \subset D^{(0)\leq}$ 和 $d^{(1)} \subset D^{(1)\leq}$ 是两个不同稀疏数组元组列表中的有序子集，且它们对应的外部元组互不相同，则：

$$d^{(0)}[f^{(0)}] \times d^{(1)}[f^{(1)}] \subset F^{(0)<} \times F^{(1)<} \quad ,$$

是结果的一个良序子集。

6 二阶数组纯稀疏算法

这里我们将考虑应用于两个纯稀疏数组的偏迹算法。这通常已经足够，因为对于多个稠密数组，最优的偏迹算法往往也是一次对两个数组进行求迹。由于只考虑两个数组，

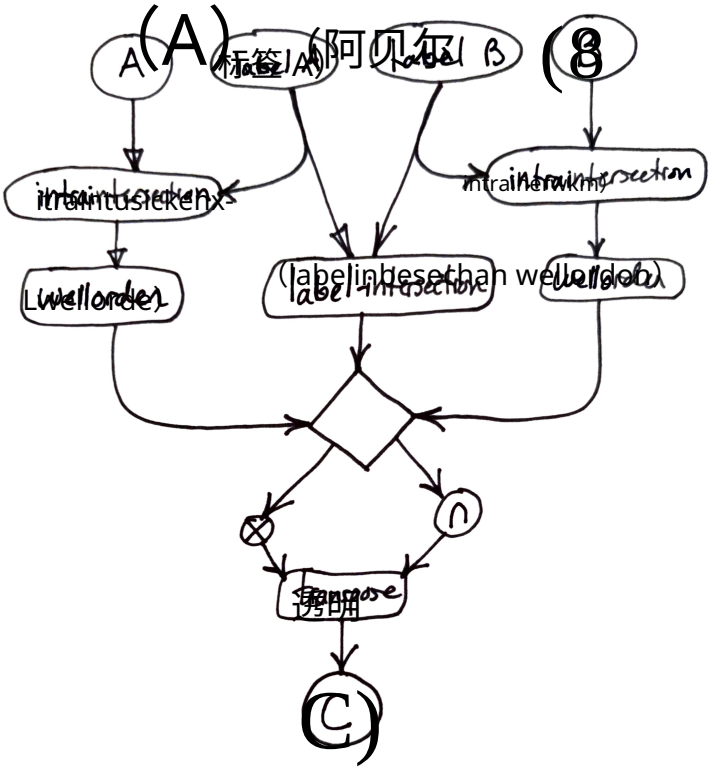


图4。对两个稀疏数组进行部分求迹的算法流程图。

不可能出现超边（同时涉及三个或更多数组的迹）。见图4。

直接交集是该算法的关键部分。问题在于，直接交集通常会导致元素数量迅速膨胀，因为它涉及部分求迹中所有可能的乘积。直接交集中的元素数量也是该算法的时间复杂度（例如 在朴素¨矩阵乘积中，我们有 $\sim N^3$个乘积）。因此，我们希望先执行所有数组内操作，而不处理可能很大的直接交集。这包括对 §2.6 进行内部交集（移除冗余自由度），并建立§2的规范格式。此规范格式要求对每个内部索引进行排序。

下一个操作是标签交集，我们需要知道哪些轴¨彼此相交。如果没有轴¨相交，那么唯一的操作就是直积。假设我们有一个稀疏数组 A，包含 n 个轴¨，这意味着索引数组 A^{index} 有 n 列。如果给定一个标签，它可以确定哪些索引是外部索引，哪些是内部索引。我们希望将稀疏数组整理成列，使其分别包含外部索引或内部索引。多个外部索引可以归为一组，因为它们不会与其他任何轴¨交互。重叠的（通常是内部）索引则由 $O^{(i)}$表示，每个重叠（内部）索引对应一个。竖线表示拼接（跨列/轴），我们

有两个元组列表/索引数组分解的示例：

$$A^{\text{index}} = F \,||\, O^{(0)} \,||\, O^{(1)} \,||\, \cdots \,||\, O^{(n-1)}$$
$$B^{\text{index}} = O^{(0)} \,||\, O^{(1)} \,||\, \cdots \,||\, O^{(n)} \quad .$$

每个内部元组数组对应一条迹化/收缩边。它们按元组连接起来。这种分解很重要，因为输出数组的大小至多为外部索引笛卡尔积的大小。继续进行直接交集运算，我们希望确定哪些稀疏索引会相乘，这可以通过§4中的方法，利用二分查找来实现。然后，我们对相交的子数组元素进行直积。对于这些元素中的每一个，都有与之关联的外部索引，这些索引通过§3中的满射得到。对相同的外部索引求和，即可得到结果数组。

7 二阶数组：稀疏与稠密迹

这里展示如何对一个纯稀疏数组（S）和一个稠密数组（D）进行部分求迹，从而得到另一个稠密数组 O。假设稀疏数组（S）和稠密数组（D）都具有两类索引：外部索引和内部索引。分别用 $\mathbb{I}_E$ 和 $\mathbb{J}_E$ 表示稀疏数组和稠密数组的外部索引集合，用 $\mathbb{I}_I$ 和 $\mathbb{J}_I$表示它们的内部索引集合。这样，两个数组都可以写成[4]：$S[\mathbb{I}_E, \mathbb{I}_I]$ 和 $D[\mathbb{J}_E, \mathbb{J}_I]$的形式。于是，得到的稠密数组 O 为 $O[\mathbb{I}_E, \mathbb{J}_E]$，其元素就是复合的乘积之和，也就是对每个稀疏项 I求和的迭代求和：

$$O[\mathbb{I}_E, \mathbb{J}_E] \overset{\sum}{=} D[\mathbb{J}_E, \hat{\mathbb{I}}_I] S^{\text{data}}[I] \quad .$$

请注意，稀疏内部索引元组 $\mathbb{I}_I$ 被注入/组合到稠密数组中。子数组 $D[\mathbb{J}_E, \hat{\mathbb{I}}_I]$ 与 $S^{\text{data}}[I]$相乘，并累加到 $O[\mathbb{I}_E, \mathbb{J}_E]$的现有项中。

8 结果

这里我们将稠密收缩方法与本文提到的稀疏收缩方法进行比较。我们将考虑非零项比例低至

[4]由于语法限制，此处考虑的实现会将给定数组的所有内部索引重塑为一个轴，并将所有外部索引重塑为另一个轴。因此，稀疏数组会变成稀疏矩阵（一种标准数据结构，*i.e.* $\mathbb{I}_E \to I_E$），而稠密数组会变成稠密矩阵。随后，便在此设置下应用相同的复合过程。

$$D \xrightarrow{\text{swapaxes}} D[\mathbb{J}_E, \mathbb{J}_I] \xrightarrow{\text{reshape}} D[\mathbb{J}_E | J_I]$$
$$S \xrightarrow{\text{swapaxes}} S[\mathbb{I}_E, \mathbb{I}_I] \xrightarrow{\text{reshape}} S[\mathbb{I}_E, I_I] \xrightarrow{\text{reshape}} S[I_E, I_I]$$
$$\left\{ O[\mathbb{J}_E, I_E] \overset{\Sigma}{=} S^{\text{data}}[I] D[\mathbb{J}_E, I_E] \right\}_I \longrightarrow O \quad .$$

10^{-4}。该算法目前使用 Python（一种高级语言）实现，因此存在优化方面的问题（仍有很大的改进空间）。尽管如此，在某些情况下，它仍能胜过经过优化的程序（使用低级语言优化）。由于稠密迹算法不受稀疏性的影响，因此会单独多次计算它们，作为对照。在以下结果中，稠密偏迹/收缩由`einsum` 提供，来自 [2]。我们结果中使用的数组均为随机生成。该算法的一个潜在应用是张量网络。因此，我们考虑矩阵乘积算子（MPO）的收缩（§8.2）以及投影纠缠对算子（PEPO）的收缩（§8.3）

8.1 矩阵乘积

这里我们考察两个二维数组之间的乘积 $W_{(1)AB}W_{(2)BC}(\mathcal{O} \sim N^3)$。由于稀疏矩阵已有成熟的理论，这种比较很有用。本文提到的算法并未使用底层语言进行优化，因此将其与成熟代码进行比较，或许能让我们了解优化后的性能。成熟的稀疏矩阵代码由 [4]提供。结果表明，在稠密和极稀疏的情况下，我们的实现比 SciPy 的稀疏矩阵代码慢约 50 倍。图 5显示，我们算法的稠密-稀疏交叉点约为 0.01，而 SciPy 实现的交叉点为 ≈ 0.50。

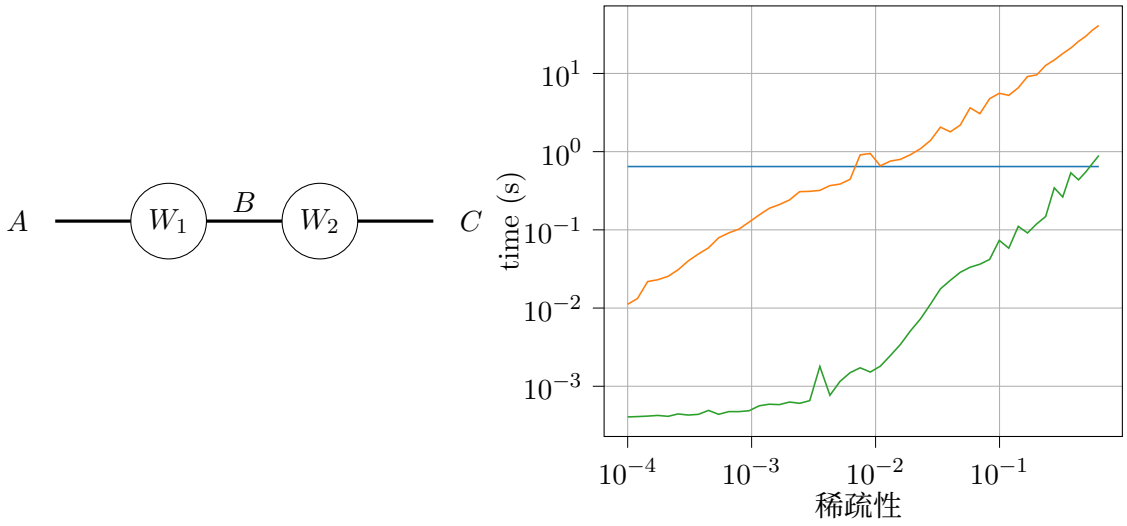


图 5。两个矩阵的偏迹。（左）表示矩阵乘积的示意图。（右）显示稀疏性时间复杂度的图表，其中 einsum 用蓝色表示，我们的算法用橙色表示，SciPy 的稀疏矩阵乘积用绿色表示。

8.2 矩阵乘积算子迹

接下来我们考虑两个稀疏四指标矩阵乘积算子（MPO）的偏迹：$W_{(1)ABab}W_{(2)BCcd} = C_{ACabcd}(\mathcal{O} \sim N^7)$。在我们的测试中，每个四指标算子的形状为 $s[i] =$

(20,20, 20, 20)。图6展示了我们的结果。稠密-稀疏交叉点约为 0.05，时间复杂度显示与稀疏性呈直接比例关系。

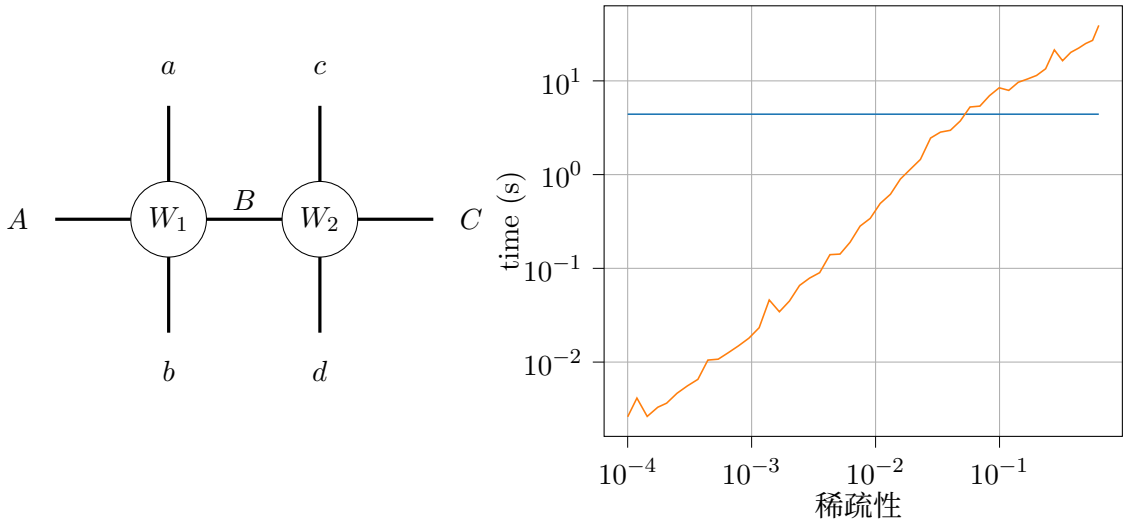


图6。两个 MPO 元素的偏迹。（左）偏迹示意图。（右）稀疏性时间复杂度图，其中 einsum 用蓝色表示，我们的算法用橙色表示。

8.3 投影纠缠对算子的迹

接下来，我们考虑两个稀疏的 6 指标投影纠缠对算子（PEPO）的偏迹：$W_{(1)ABCDab}W_{(2)DEFGcd} = C_{ABCEFGabcd}(\mathcal{O} \sim N^{11})$。在我们的测试中，每个 4 指标算子的形状为 $s[i] = (8, 8, 8, 8, 8, 8)$。结果如图 7所示，表明该算法的时间复杂度仍随稀疏性线性增长。

9 总结

正如我们的结果所示，稀疏数组可以利用某些优势（部分求迹），这是稠密数组所不具备的。不过，有些操作看起来速度较慢。与稠密数组操作相比，稠密数组算法可以在 $\mathcal{O}(1)$ 时间内访问元素，而该算法则需要 $\mathcal{O}(\log\ N)$ 时间；重塑稠密数组可以在 $\mathcal{O}(1)$ 时间内完成，而稀疏数组则需要 $\mathcal{O}(N)$。此外，稀疏数组需要沿每个索引排序，对于包含 n 列/索引和 N 个条目的稀疏数组，这会产生 $\mathcal{O}(nN\log\ N)$ 次操作。排序开销是可加的，因此最多与初始元素数量成比例，并且所有操作都可以并行完成。在我们的结果中，对于所考察的所有测试，稀疏性相关的指数缩放因子约为 1（分别为 0.86、1.10 和 1.17）。这意味着算法的时间复杂度直接取决于稀疏性。仍有许多工作要做，包括使用低级编程语言优化算法、利用对称性进行部分求迹，以及全面扩展对块稀疏数据结构的支持。

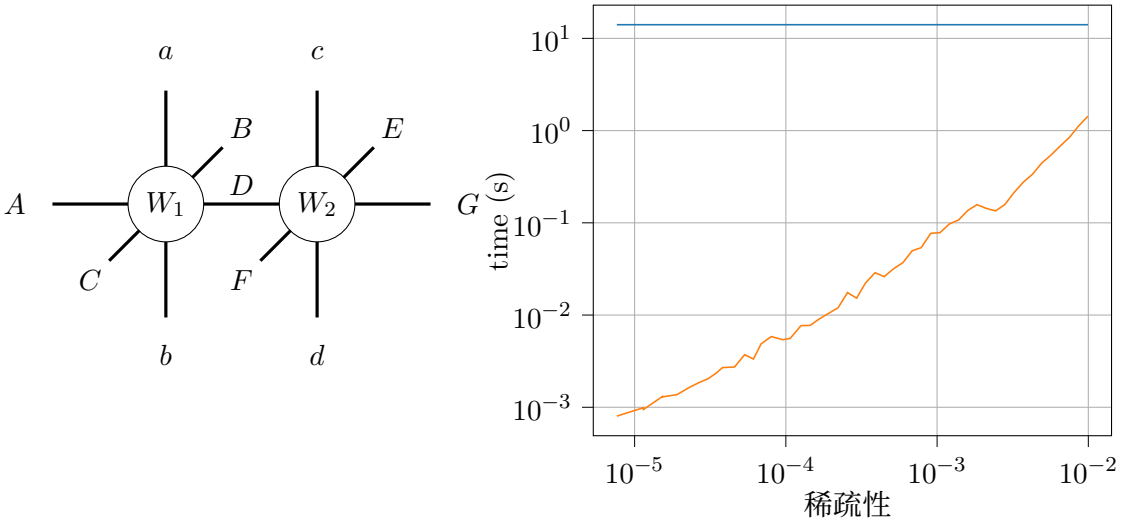


图 7。两个 PEPO 元素的偏迹。（左）偏迹示意图。（右）稀疏性时间复杂度图，其中 einsum 用蓝色表示，我们的算法用橙色表示。

A. 稀疏数组

A.1 属性

稀疏数组旨在完美表示稠密数组，其前提是忽略值为零的元素[5]。稀疏数组可分为两个主要部分：一个二维稠密索引数组和一个一维稠密数据数组。稀疏索引数组的表示方式如下：

$$\text{Sparse Index-Array} \equiv A[\ \ |\ \ ] \qquad ,$$

竖线 $|$ 将这个二维数组的两个索引分隔开。左侧表示稠密索引列，右侧则对稀疏元组条目的枚举进行索引。为了与稠密表示相对应，通常会明确标出稠密索引。例如，对于张量 A，其稠密索引为 $ijkl$，稀疏索引为 I（枚举所有非零值）：

$$\begin{aligned}\text{Dense Representation} &\equiv A[i,j,k,l] \\ \text{Sparse Representation} &\equiv A[i,j,k,l\,|\,I] \qquad .\end{aligned}$$

请注意，稠密表示没有 I（稀疏）索引。此外，还有一项少量信息，即数组稠密表示的形状。因此，稀疏数组的完整属性包括：形状数组（s）、元组数组/索引数组（A_{index}）以及数据数组（A_{data}）。稀疏数组的完整结构如图8所示。

[5]有时近似为零，此时该表示是近似的。

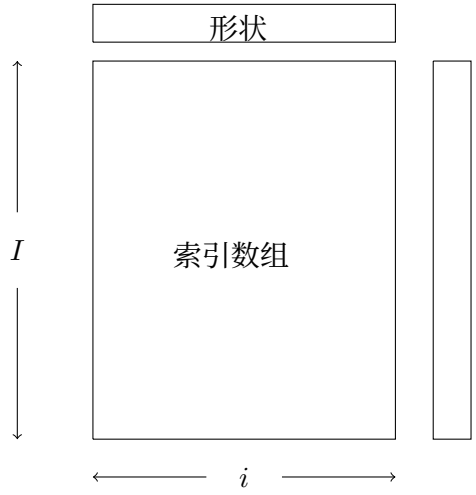


图 8。稀疏数组分解为形状、元组数组和数据数组。

A.1.1 形状

形状数组是一维数组，用于给出数组在其稠密表示中的形状。这对于稀疏数组与稠密数组之间的映射很有用。

A.1.2 索引

按照惯例，稀疏数组的索引数组（或索引 或元组数组 ）按字典序排列（ssort，按惯例采用升序）。这有助于识别重复项以及执行其他稀疏数组操作。索引数组记录了数组的结构。

A.1.3 数据

对我们而言，数据数组 是一维的，其长度与索引数组相同，并且与索引数组一一对应。也就是说，列表索引（索引数组的）与数据数组索引相匹配。这些数组通常采用浮点数据类型，对重复项没有限制，槽表示为 $A_{\text{data}}[\quad]$。

A.2 稀疏性

稀疏性定义如下（假设 A_{index} 格式正确，且仅包含唯一条目）：

$$\begin{aligned}\text{sparsity} &= \frac{\text{total elements in proper sparse-tensor}}{\text{total elements in dense-tensor}} \\ &= \frac{\text{len}(A_{\text{data}})}{\prod_i \text{shape}[i]} \quad .\end{aligned}$$

A.3 良序

假设我们有一个已排序的索引数组 $A_{\text{index}}[\![\quad|\quad]\!]$，以及它的唯一值数组 u （表示唯一元素在已排序数组中的首次出现位置）。那么，重复元素

就是已排序稀疏张量中两个相邻唯一元素之间的所有元素，$u[i]:\ u[i+1]$（不包括 $u[i+1]$）。因此，我们对 u 中每一对相邻元素所对应的 $A[\![\ \ |\ \ ]\!]$ 中所有这些元素求和。这样就会得到一个数组，其中每个唯一索引元组都对应一个唯一的数据值：

$$\begin{aligned} A_{\text{index}}[\![\ \ |\ \ ]\!] &= A_{\text{index}}[\![\ \ |\ u\ ]\!] \quad , \\ A_{\text{data}}[\ \ ] &= \sum_{i \in |u|} A_{\text{data}}[\ u[i] : u[i+1]\ ] \qquad . \end{aligned}$$

注意 $u[i+1] = |u|$ 最后一个唯一数组的索引，$i = |u|$。

B 字典序

B.1 元组数组

数值字典序 定义在二维数组（即某种数据类型上的矩阵）上。我们希望沿着给定的轴（例如列）进行排序，同时保留行。或者，我们也可以将这种数据结构称为元组数组，以便区分元组的顺序（行）得以保留，而我们对数组进行置换，即对列进行置换。我们可以将这个二维元组数组表示为：$A[n|I]$。对于数组索引（记为I），按行方向作用时指第 I 个元组；而按列方向作用时，则使用元组索引（记为 n）。

B.2 元组间比较

现在，我们想要定义一个元组小于或大于另一个元组的概念，并将这种比较记为 $\prec$ 异或 $\preceq$。假设有两个长度相同且有限的元组，其中填入了 $n+1$ 项 $a,\ b \in \mathbb{N}^{n+1}$，那么 $a < b$ 如果我们进行逐元素减法$b - s = c$，并且 c （从左到右）第一个非零项为正数。等价地，按照 [1]:

$$(a[0], a[1], \cdots, a[n]) \prec (b[0], b[1], \cdots, b[n])$$

star从左到右查看，如果 $a[i] < b[i]$，否则为 $a_i = b_i \,\&\, a[i+1] < b[i+1]$（对于 $0 < i \leq n$） .

B.2.1 元组间相等

假设我们有两个元组（大小相同，填入了 $n+1$ 条目） $a,\ b \in \mathbb{N}^{n+1}$，那么$a = b$ 当且仅当 $a[i] = b[i]$ 对所有 i （ a 的第 i 个元素与 b 的第 i 个元素相匹配）。

B.2.2 元组到数字

接下来，我们给出一种比较方法，它有助于将元组映射到数制中，同时保持顺序（给定两个元组，如果一个大于另一个，相应的数也会遵循这一关系）。假设某个数制的基数为 r。那么，我们可以通过内积将一个元组映射为该数制中的一个数（一一映射）（这里的上标表示乘方运算）：

$$\text{num}_r(a) = a[n] * r^n + a[n-1] * r^{n-1} + \cdots + a[1] * r^1 + a[0] * r^0 \qquad .$$

例如，二进制数制（$r = 2$）、十进制（$r = 10$）、十二进制（$r = 12$）、十六进制（$r = 16$）。可以证明，如果 $a < b$，那么以基数 r 表示的数也满足 $\mathrm{num}_r(a) < \mathrm{num}_r(b)$。

B.3 字典序／字典顺序的定义

如果元组列表（索引数组）中的所有元组对都符合元组间比较 $\prec$，则称该列表按字典序排序。请注意，这不包括相等的情况，因此所有条目都必须唯一；我们将这种顺序记作 $sssort$，或称为数学意义上的良序。如果将比较条件放宽为允许相等，则可以包含重复项，并使用 $\preceq$比较；这种顺序记作$ssort$ 顺序。

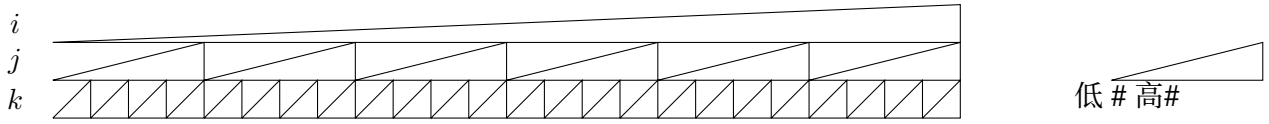


图9。示意图展示了一个包含 3 个索引的 $A[i, j, k|I]$ 中，元组列表数组的排序方式：沿着 i 数值按升序排序。随后，在 i （前一列）的重复值中，将 j 按升序排序。最后，在 i 和 j （前两个数组）的重复值中，将 k 按升序排序。

B.3.1 字典序排序算法

假设我们有一个通用的、未排序且可能包含重复条目的元组列表数组（列标签为 $\mathbf{0, 1, 2, 3, \cdots, N}$）：

$$A\left[\ \mathbf{0,\ 1,\ 2,\ 3,\ \cdots, N}\ |\ I\ \right] \qquad .$$

接下来，我们引入一个名为域的辅助数组 u，它用于约束连续（列/索引）排序的范围。例如，排序会在域数组中的连续元素内执行（列表索引的范围为 $[u[n], u[n+1])$ ），这是一种约束排序：

$$A\left[\ \mathbf{0,\ 1,\ 2,\ 3,\ \cdots, N}\ |\ u[n]\ :\ u[n+1]\ \right] \qquad .$$

对每个索引（$\mathbf{0, 1, 2, 3, \cdots, N}$）进行排序后，排序约束会发生变化，并且必须递增。最初没有任何约束。随后，排序后索引中各元素之间的差异会形成域，用于约束后续排序。重要的是，域 u^n 必须与先前的域 u^m （$m < n$）合并，以遵守先前的边界：

$$u = \bigcup_{\eta} u^{(\eta)} \qquad .$$

如果能对小域进行 argsort，并行处理这些小域，就会很方便。对于归并排序来说，这相当自然，因为它会先对数据集进行分区，再进行合并。不同之处在于，我们希望能够指定这些分区，而分区的大小可能各不相同。然后根据分区大小实现排序算法：要么使用 插入排序 异或归并排序。

B.3.2 计算时间复杂度

设 $N = M^n$ （M 是稠密张量一条边的长度），那么我们必须依次执行以下受约束排序：

$$\begin{aligned}\mathcal{O} &\sim M^n \left(\log\left(\frac{M^n}{M^0}\right) + \log\left(\frac{M^n}{M^1}\right) + \cdots + \log\left(\frac{M^n}{M^n}\right)\right) = \frac{n+1}{2} M^n \log\left(M^n\right) \\ &\sim \frac{n+1}{2} N \log N \qquad .\end{aligned}$$

将其与： $\mathcal{O} \sim nN\log\ N$（对每一列进行独立排序）进行比较。

B.3.3 定理

以下内容很容易证明：

- 在 sssort 顺序的右侧添加任意（随机）列，不会改变原有的 sssort 顺序。
- 删除任意一行（一个元组）后，仍保持 sssort 顺序。
- 从 sssort 顺序中删除任意一列会使其变为 ssort 顺序（从而可能产生重复项）。

致谢

The 作者感谢亚利桑那州立大学物理系的支持 t.

参考文献

[1] Munkres, J. R.（2000）。拓扑学，第 2 卷。Prentice Hall，上萨德尔河。[2] NumPy、Harris, C. R.、Millman, K. J.、van der Walt, S. J.、Gommers, R.、Virtanen, P.、Cournapeau, D.、Wieser, E.、Taylor, J.、Berg, S.、Smith, N. J.、Kern, R.、Picus, M.、Hoyer, S.、van Kerkwijk, M. H.、Brett, M.、Haldane, A.、del R´ıo, J. F.、Wiebe, M.、Peterson, P.、G´erard-Marchant, P.、Sheppard, K.、Reddy, T.、Weckesser, W.、Abbasi, H.、Gohlke, C. 和 Oliphant, T. E.（2020）。《自然》，585(7825):357–362。[3] PyTorch、Paszke, A.、Gross, S.、Massa, F.、Lerer, A.、Bradbury, J.、Chanan, G.、Killeen, T.、Lin, Z.、Gimelshein, N.、Antiga, L.、Desmaison, A.、Kopf, A.、Yang, E.、DeVito, Z.、Raison, M.、Tejani, A.、Chilamkurthy, S.、Steiner, B.、Fang, L.、Bai, J. 和 Chintala, S.（2019）。Pytorch：一种命令式风格的高性能深度学习库。载于 Wallach, H.、Larochelle, H.、Beygelzimer, A.、d'Alch´e-Buc, F.、Fox, E. 和 Garnett, R. 编，神经信息处理系统进展 *32*，第 8024–8035 页。Curran Associates, Inc. [4] SciPy、Virtanen, P.、Gommers, R.、Oliphant, T. E.、Haberland, M.、Reddy, T.、Cournapeau, D.、Burovski, E.、Peterson, P.、Weckesser, W.、Bright, J.、van der Walt, S. J.、Brett, M.、Wilson, J.、Millman, K. J.、Mayorov, N.、Nelson, A. R. J.、Jones, E.、Kern, R.、Larson, E.、Carey, C. J.、˙Polat, I.、Feng, Y.、Moore, E. W.、VanderPlas, J.、Laxalde, D.、Perktold, J.、Cimrman, R.、Henriksen, I.、Quintero, E. A.、Harris, C. R.、Archibald, A. M.、Ribeiro, A. H.、Pedregosa, F.、van Mulbregt, P. 和 SciPy 1.0 贡献者（2020）。《自然·方法》，17:261–272。

[5] TensorFlow、Abadi, M.、Agarwal, A.、Barham, P.、Brevdo, E.、Chen, Z.、Citro, C.、Corrado, G. S.、Davis, A.、Dean, J.、Devin, M.、Ghemawat, S.、Goodfellow, I.、Harp, A.、Irving, G.、Isard, M.、Jia, Y.、Jozefowicz, R.、Kaiser, L.、Kudlur, M.、Levenberg, J.、Man´e, D.、Monga, R.、Moore, S.、Murray, D.、Olah, C.、Schuster, M.、Shlens, J.、Steiner, B.、Sutskever, I.、Talwar, K.、Tucker, P.、Vanhoucke, V.、Vasudevan, V.、Vi´egas, F.、Vinyals, O.、Warden, P.、Wattenberg, M.、Wicke, M.、Yu, Y. 和 Zheng, X.（2015）。*USENIX*。软件可从 tensorflow.org 获取。