arXiv:2304.07240v1 [cs.DB] 14 Apr 2023

GreedyGD：面向物联网直接分析的增强型广义去重

预印本

Aaron Hurst、Daniel E.Lucani 和 Qi ZhangDIGIT 中心与奥胡斯大学电气与计算机工程系，奥胡斯，丹麦 电子邮箱：{ah,daniel.lucani,qz}@ece.au.dk

2023年4月13日

摘要

物联网当前生成的数据量呈指数级增长，这给数据通信、数据存储和分析带来了重大挑战，并导致希望利用其数据的组织承担高昂成本。因此，需要采用新颖的技术，从整体上提高物联网系统中数据存储和分析的效率。新兴的广义去重（GD）压缩技术已被证明能够实现高压缩率，并支持直接对压缩数据进行分析，同时仅需较低的存储和内存资源。在本文中，我们提出了一种面向分析而设计的新型基于GD的数据压缩算法，称为 GreedyGD。与现有版本的GD相比，GreedyGD能够以更少的数据实现更可靠的分析，同时运行速度提高11.2×，并提供更好的压缩效果。

关键词 数据压缩、数据分析、广义去重、聚类、物联网

1 引言

通过物联网（IoT）收集的传感器数据越来越多地被用于制定重要的商业决策、提高效率并保障人们的安全 [1]。然而，以传统的集中式方式管理这些数据，成本日益高昂；随着数据生成速度超过硬件性能的提升，这不仅增加了存储和计算方面的成本，也带来了更高的延迟。同样，边缘服务器虽然能够降低通信成本和延迟 [2], ，但其计算能力较低、单位存储成本较高，也限制了其应用。总体而言，传输、存储和分析物联网数据的成本，可能迫使组织丢弃数据或仅短期保留数据，从而限制物联网系统的效用和可持续性 [3, 4]。因此，需要整合框架来降低物联网数据存储成本，同时改善分析延迟。

新兴的压缩技术广义去重（GD） [5–7] 为满足这一需求提供了有效解决方案。具体而言，它能在物联网系统中实现高效的端到端无损压缩，同时无需解压缩即可进行高效的近似分析 [7–9]。此外，GD压缩数据的结构支持在边缘服务器和云服务器之间拆分数据，以优化存储成本和分析延迟 [9]。GD通过将数据块拆分为基（经过去重）和偏差（原样存储）来进行压缩（见图1）。偏差与一个标识符一同存储，该标识符将偏差关联到对应的基，从而实现无损解压缩。使用GD压缩数据集分为两个阶段：1）配置，即确定为基分配多少位；2）压缩，即将数据块拆分为基和偏差。

本文提出了 GreedyGD，这是一种基于GD的数据压缩算法，旨在直接对压缩数据运行分析。与以往的 GD 版本相比，它解决了若干影响压缩速度和分析质量的关键问题，包括配置时间和基样本顺序保持。GreedyGD 的压缩效果甚至优于现有的 GD 版本。本文的主要贡献包括：

1. 专为压缩数据直接分析而设计的 GreedyGD压缩算法 ，

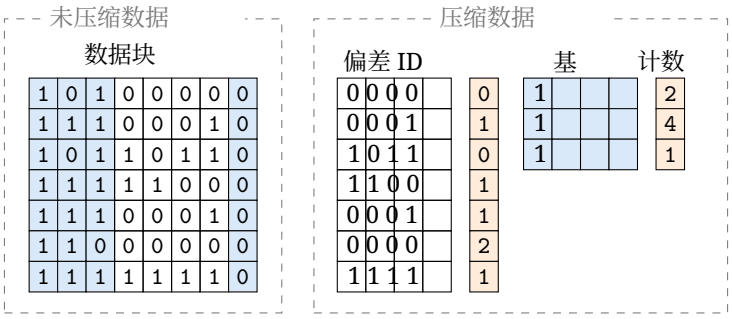

图1：将 GD 应用于 8 位数据块。前三位和最后一位（以蓝色突出显示）被分配为基并进行去重，其余位则是偏差，与将它们链接到相关基的标识符一起原样存储。

2. 针对 GD 的数据预处理，可提升浮点数据的压缩效果；3. 全面评估 GreedyGD 相较于现有 GD 版本和通用无损压缩器在压缩比、运行时间和分析方面的性能；4. 评估使用数据子集配置 GD 的效果。

本文结构如下。第2节讨论相关工作。第3节详述术语表。第4节介绍 GreedyGD。第5节评估 GreedyGD 的性能。最后，第6节对本文进行总结。

2 相关工作

数据压缩算法可分为有损和无损两类。有损算法通常能提供更高的压缩率，但会降低数据保真度，因此可能不适用于关键的物联网应用。因此，我们专注于无损压缩。现有多种通用无损压缩算法 [10–14], ，此外还有功能强大的物联网专用无损时间序列压缩算法，例如 Sprintz [15]。然而，现有算法在随机访问（对分析十分重要）、小数据块性能（与物联网传感器相关）以及在资源受限的物联网设备上运行的能力方面存在局限 [16]。

另一方面，基于GD的数据压缩在这些方面表现良好，同时还能提供较高的压缩比。为实现这一点，基位的选择至关重要，即选择映射到基的位索引集合；该选择应同时取决于数据和应用。 [7], 中介绍的GD版本（我们称之为 GD-INFO）根据位间相关性选择基位。它首先将全部位映射到基，并计算压缩数据大小。随后，将位间相关性最低的位从基移至偏差，并重新计算压缩数据大小。该过程持续进行，直到达到压缩数据大小的第一个局部最小值。此时的基位随后用于压缩。为限制运行时间，配置过程最多在前 10^6 个样本上运行。

尽管GD-INFO具有良好的性能，但仍存在一些显著局限，包括：1）没有高效计算压缩数据大小的方法；2）位间相关性并不总能准确反映去重潜力；3）在第一个局部最小值处终止通常不是最优的；4）缺乏针对多维数据的标准方法；以及5）压缩数据分析性能较差，如 [9]所示。使用数据子集（即前 10^6 个样本）进行配置所产生的影响也尚未得到研究。
在 [9], 中，提出了一个针对分析优化的 GD 版本，我们称之为 GD-GLEAN。与 GD-INFO 不同，这种压缩器牺牲了一部分压缩效果，以提升分析性能；但除此之外，它也存在与 GD-INFO 相同的局限性。

ALACRITY [17] 和 DigitHist [18]是两种与 GD 有显著相似之处的相关算法。尽管这两种算法都通过从数据中提取类似基的“分箱”来实现压缩，但与 GD 相比，它们的局限在于对哪些位可以分配给这些分箱施加了严格限制。此外，与 GD 不同，DigitHist 使用有损压缩。

本文通过提出 GreedyGD 来解决上述局限。GreedyGD 能在压缩数据上实现显著更准确的分析，同时运行速度更快，压缩效果也更好。

表1：术语表

符号	含义	符号	含义
n	数据样本数	S	压缩数据大小
d	维度数	Δ	最大偏差
n_b	基数	λ	平衡因子
B	基位集合	α	探索因子
l_c	每块比特数	$\mathcal{C}$	GreedySelect 成本
l_b	每个基位的比特数	CR	压缩比
l_d	每个偏差的比特数	ADR	分析数据比
l_{id}	每个基位标识的比特数	AR	近似比
l_{bc}	每个基数的比特数	AMI	调整互信息。

3 术语表

我们的术语汇总见表1。我们假设一个数据集包含 n 个样本和 d 个维度。GD 分配给基的位索引集合称为基位，记为 B。去重后的基总数记为 n_b。二进制数据块 由一个数据样本在各维度上的数据串接而成，长度为 l_c 位。每个基的比特数和偏差分别记为 l_b 和 l_d。因此，$l_c = l_b + l_d$。GD 分别使用 $l_{id} = \lceil \log_2(n_b) \rceil$ 位和 $l_{bc} = \lceil \log_2(n) \rceil$ 位存储基 ID 和基数计数。

“最大偏差” $\boldsymbol{\Delta}$，是映射到同一基的样本之间的最大差值。例如，图1中映射到第一个基（偏差位全为 0）的最小值为 10100000，即 160；而最大值（偏差位全为 1）为 10111110，即 190。因此，最大偏差为 00011110，即 30，也就是将所有偏差位置为 1。请注意，对于多维数据，$\boldsymbol{\Delta}$ 是一个向量；对于浮点数据，由于尾数比特的值取决于指数，因此其值会随基而变化。

GD 的总压缩数据大小 S 等于：

$$S = n_b \left(l_b + l_{bc}\right) + n \left(l_{id} + l_d\right) + S_{params}, \tag{1}$$

其中，S_{params} 是 GD 的参数大小（通常可忽略不计）。压缩性能使用压缩比（CR）衡量，定义如下：

$$\mathrm{CR} = \frac{\text{size of compressed data}}{\text{size of uncompressed data}}. \tag{2}$$

因此，数值越低（低于 1）越好。此外，要直接对 GD 压缩数据运行分析所需访问的（压缩）数据量，使用分析数据比（ADR）衡量，其值为：

$$\mathrm{ADR} = \frac{\text{size of data used for analytics}}{\text{size of uncompressed data}} \approx \frac{n_b(l_b + l_{bc})}{n l_c}. \tag{3}$$

我们使用 k-均值聚类 [19] 作为 GreedyGD 分析性能的测试案例。我们采用三个性能指标。第一个指标是近似比（AR），其根据误差平方和（SSE）定义如下：

$$\mathrm{AR} = \frac{\text{clustering SSE using compressed data}}{\text{clustering SSE using uncompressed data}}, \tag{4}$$

因此，AR ≥ 1，数值越低表示性能越好。数值为 1 表示，与在未压缩数据上运行聚类相比，在压缩数据上运行聚类不会导致聚类质量损失。

我们使用的第二个指标是调整互信息（AMI），这是一种用于衡量两个聚类结果相似性的信息论度量。在本例中，AMI 的取值始终介于 0 和 1 之间，数值越高越好，即表示聚类结果越相似。

第三个指标是轮廓系数，用于衡量数据点与同一簇中的数据点相比，与下一个最合适簇中的数据点的契合程度。更正式地说，对于给定的数据点 $\boldsymbol{x}$，如果 $a(\boldsymbol{x})$是 $\boldsymbol{x}$ 与同一簇中其他数据点之间的平均距离，而 $b(\boldsymbol{x})$是 $\boldsymbol{x}$ 与下一个最佳簇中数据点之间的平均距离，那么 $\boldsymbol{x}$ 的轮廓系数为：

$$s(\boldsymbol{x}) = \frac{b(\boldsymbol{x}) - a(\boldsymbol{x})}{\max\left(a(\boldsymbol{x}), b(\boldsymbol{x})\right)}. \tag{5}$$

算法1 GreedyGD

输入： 数据集 $\mathcal{D}$，探索因子 α，平衡因子 λ
输出： 压缩数据集 $\mathcal{D}_{GD}$

$\mathcal{D}' \leftarrow$ 预处理 $\mathcal{D}$ // 小节 *4.3*
$D \leftarrow$ 从 $\mathcal{D}'$提取子集 //第*4.4*小节（可选）
$B \leftarrow$ GreedySelect(D', α, λ) //第*4.2*小节
根据 B对 $\mathcal{D}'$ 应用 GD

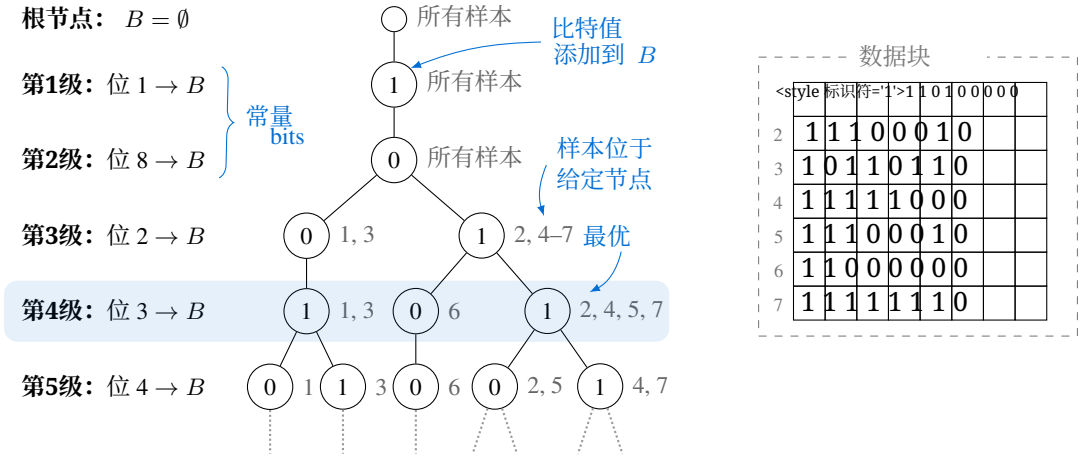

图2：将 BaseTree 部分应用于图1中的数据块，为方便起见，这些数据块也在图的底部重复显示。突出显示的最优配置第4级，对应于图1中基位和偏差位的分配。

整个聚类的轮廓系数是全部数据点的轮廓值的平均值。轮廓系数的范围为 ± 1，数值越高越好；接近 1 的值表示簇密集且分离良好。

4 GreedyGD算法

基于GD的压缩的关键在于选择一组有效的基位。由于可能的组合数量众多，这是一项具有挑战性的任务。例如，d维32位数据有 2^{32d} 种可能的配置，即每个维度超过40亿种配置。由于必须针对每种可能的组合计算不同基的数量，而这会非常耗时，因此枚举所有配置并不可行。

GD-INFO按照位间相关性对位进行排序，并在压缩数据大小首次达到局部最小值时终止，从而解决了这一问题。这使枚举的配置数量最多限制为 l_c种。然而，尽管这使GD-INFO在计算上可行，但前述限制使其无法达到最优。

概述于算法1中的GreedyGD压缩算法旨在找到更优的基位集合，并更快地对其进行评估。简而言之，GreedyGD的工作流程为：1）采用新颖的数据预处理方法；2）可选地选择数据的一个子集，以加速配置；3）使用贪心算法选择基位；4）使用GD压缩数据。

本节余下部分的结构如下。第4.1小节介绍用于快速统计基的 BaseTree 算法。第4.2小节介绍 GreedySelect，该算法基于 BaseTree，用于选择高质量的基位。第4.3小节详细说明预处理如何提升压缩效果，第4.4小节则详细说明如何利用数据子集缩短运行时间。最后，第4.5小节分析 GreedyGD 的时间复杂度。

4.1 基底计数

为加速 GD 配置的基底计数，我们提出了 BaseTree 算法，如图 2 所示。其思路是将给定基底比特集对应的基分布存储在树结构中。如图 2 所示，根

算法 2 GreedySelect

输入： 数据集 $\mathcal{D}$，探索因子 α，平衡因子 λ
输出： 基位 B

1: //初始化
2: $B \leftarrow$ $\mathcal{D}$中的常量位
3: 使用 B 扩展 BaseTree
4: $B_{\text{best}} \leftarrow B$
5: $\mathcal{C}_{\text{best}} \leftarrow \infty$
6: // 迭代添加基位，直至终止
7: **while** B 中的位尚未全部满足**do**
8: $b_{\text{loc}} \leftarrow -1$ // 重置用于局部优化的最低成本位
9: $\mathcal{C}_{\text{loc}} \leftarrow \infty$ // 重置用于局部优化的最低成本
10: **对于** $i = 1$ 到列数**执行**
11: $b_i \leftarrow$ 列 i 中不在 B中的最高有效位
12: $n_{b,\ i} \leftarrow n_b$ 将 b_i 添加到 B之后 //使用 *BaseTree*
13: $S_i \leftarrow$ 添加 b_i 到 B后的压缩数据大小 // 公式 *1*
14: $\Delta_i' \leftarrow \Delta_i$ 添加 b_i 到 B后 // 公式 *6*
15: $\mathcal{C}_i \leftarrow$ 添加 b_i 到 B后的成本 // 公式 *7*
16: **如果** $\mathcal{C}_i < \mathcal{C}_{\text{loc}}$ **那么** // 当前迭代中代价最低的位
17: $b_{\text{loc}} \leftarrow b_i$
18: $\mathcal{C}_{\text{loc}} \leftarrow \mathcal{C}_i$
19: **结束如果**
20: **结束循环**
21: **如果** $\mathcal{C}_{\text{loc}} > (1+\alpha) \cdot \mathcal{C}_{\text{best}}$ **那么** // 提前终止
22: **返回** B_{best}
23: **结束 if**
24: 将 b_{loc} 追加到 B
25: 使用 b_{loc}扩展 BaseTree
26: **如果** $\mathcal{C}_{\text{loc}} < \mathcal{C}_{\text{best}}$ **则** //更新全局最优值
27: $B_{\text{best}} \leftarrow B$
28: $\mathcal{C}_{\text{best}} \leftarrow \mathcal{C}_i$
29: **结束 if**
30: **结束 while**
31: **返回** B_{best}

节点对应于 $B = \emptyset$，并包含全部数据样本。每个后续层级都对应于向 B 添加一个额外位，并可能产生额外节点，具体取决于所选位的取值。例如，在第2级中，只创建了一个节点，因为位 8 始终为 0（即常量）。相比之下，在第3级中，数据被分到两个子节点，因为位 3 同时包含 0 和 1。在第4级中，左侧父节点只生成一个子节点，因为样本 1 和 3 的位 3 均为 1；而右侧父节点则生成两个子节点，因为样本 2 和 4–7 中，位 3 既有 0（左侧子节点），也有 1（右侧子节点）。
此结构可用于快速计算给定基位集合 B 的基底数量，因为树的宽度对应于 n_b。因此，基底计数只需：1）将所需基位添加到树中；2）统计叶节点的数量。请注意，树的高度对应于基位数量 l_b，而根到叶路径对应于各个基底。

4.2 选择基位

如前所述，选择合适的基位对 GD 的性能至关重要。为此，我们提出了 GreedySelect 算法，旨在优化压缩数据分析性能，同时保持良好的压缩效果。如算法 2 所述，GreedySelect 首先将所有常量位（即所有样本中取值相同的位）添加到 B，并相应地扩展 BaseTree（第 2–3 行）。随后，它迭代地向B 添加位，同时最小化代价函数，直至终止。每次迭代都包括一个局部优化过程，随后是一个全局优化步骤。

在局部优化中，GreedySelect 会检查数据的每一列，并找出当前映射到偏差的最高有效位 b_i（即 $b_i \notin B$，第 11 行）。然后使用 BaseTree 计算将 b_i 添加到 B 后的基底数量 $n_{b,\ i}$（第 12 行），并计算由此产生的压缩数据大小 S_i（第 13 行）。

然后计算如果将 b_i 移至基底 Δ'_i，第 i 列中的最大偏差（第 14 行）。 s:

$$\Delta'_i = \Delta_i \oplus 2^{b_i}, \tag{6}$$

其中， Δ_i 是列 i 中的当前最大偏差， $\oplus$ 是二进制异或运算符。最后，使用以下代价函数计算将 b_i 添加到 B 后得到的代价 $\mathcal{C}_i$（第 15 行）：

$$\mathcal{C}_i = \left(1 - \lambda \left(\Delta'_i/\Delta_i^{(0)}\right)^2\right) \cdot S_i, \tag{7}$$

其中， $\Delta_i^{(0)}$ 是仅将常量位添加到 B后维度 i 中的最大偏差， λ 称为平衡因子，其定义满足 $0 \leq \lambda < 1$。

公式 7 中的代价函数主要取决于 S，因此倾向于较高的压缩率。不过，该函数也会根据最大偏差进行缩放，从而平衡从各个维度添加到 B 的位数，以提升分析性能。在第一次迭代中，由于所有维度的 $\Delta'_i/\Delta_i^{(0)}$ 均为 $1/2$，缩放不会产生影响。在后续迭代中，对于已向 B添加位的维度，比例 $\Delta'_i/\Delta_i^{(0)}$ 会更低。因此，对于已向 B 添加较多位的维度，项（$1 - \lambda$（$\Delta'_i/\Delta_i^{(0)}$ ）^{2}）会更大（导致代价更高）；对于已向 B添加较少位的维度，该项则会更小（代价更低）。这种影响的程度由平衡因子 λ控制。我们发现，将 $\lambda = 0.02$ 设为相应值，对大多数数据集都能取得良好效果。在此设置下，每个维度添加 3–4 位后，平衡效果便会变得可以忽略不计，且 $\mathcal{C}_i \approx S_i$。局部优化过程通过跟踪最低代价 $\mathcal{C}_{\text{loc}}$及其对应的位 b_{loc} 来完成（第 16–19 行）。

GreedySelect 全局优化阶段的第一步是检查终止条件（第 21 行）。终止由探索因子 $\alpha > 0$控制，它限制了对局部最小值之外区域的探索程度。如果 $\mathcal{C}_{\text{loc}}$ 超过当前全局最优成本 $\mathcal{C}_{\text{best}}$的 $1 + \alpha$ 倍，算法就会终止。我们发现，将$\alpha = 0.1$ 设为该值对大多数数据集效果良好。若未终止，GreedySelect 会将 b_{loc} 添加到 B，更新 BaseTree，并在需要时更新 $\mathcal{C}_{\text{best}}$ （第 24–29 行）。总体而言，最多会检查 $l_c d$ 种基位组合（每次迭代检查 d 种，最多迭代 l_c 次）。

通过按重要性从高到低选择基位，GreedySelect 保证顺序保持 [8]。这意味着，对于任意两个数据样本 x_1 和 x_2，以及它们对应的基 base(x_1) 和 base(x_2)，

$$x_1 < x_2 \Rightarrow \text{base}(x_1) \leq \text{base}(x_2). \tag{8}$$

顺序保持对分析性能至关重要。若不保持顺序，结果可能会差几个数量级。总体而言，通过保证顺序保持、平衡各维度的基位数量，并依据压缩数据大小设定成本函数，GreedySelect 有效平衡了压缩数据分析与压缩这两个目标。

4.3 数据预处理

将数据转换为更适合处理的形式，是压缩算法中的典型步骤。对于 GD，通过缩放并将浮点数据转换为整数，可以显著提升压缩效果。例如，考虑图 3 所示的数据，该数据来自 *Aarhus CityLab* 数据集 [20]。原始数据（图 3(a)）只有两位小数，但只有一位恒定位。缩放以去除小数位后（图 3(b)），更多位变为恒定位，这意味着 GD 可以将它们分配给基，从而提升压缩性能。最后，将数据转换为整数会产生更多恒定位。

4.4 数据子集上的配置

对于大型且复杂的数据集，配置 GD 仍可能耗费大量时间。在这种耗时难以接受的情况下，我们遵循 [7] 的建议，使用数据子集。不过，我们注意到，这会带来两个重要风险：1）数据集中的小数位数可能多于数据子集中的小数位数，导致预处理不准确；2）数据子集中的恒定位在数据集其他位置可能有所不同，从而可能违反顺序保持。鉴于这些风险，我们建议使用完整数据集来选择预处理方式并确定恒定位，然后在数据子集上运行 GreedySelect 的其余部分。

4.5 复杂度

配置 GreedyGD（即运行 GreedySelect）的时间复杂度如下。首先，必须识别常量位，这需要检查全部 nl_c 位。接下来，必须将这些位添加到基树中，但

```
 0.39   00111110110001111010111000010100
37.83   01000010000101110101000111101100
98.92   01000010110001011101011100001010
```

(a) 原始数据（1 个恒定位）

```
   39.   01000010000111000000000000000000
3,783.   01000101011011000111000000000000
9,892.   01000110000110101001000000000000
```

(b) 缩放后（14 个恒定位）

```
   39   00000000000000000000000000100101
3,783   00000000000000000000000011011000111
9,892   00000000000000000000010011010100100
```

（c）整数（18 个常量位）

图 3：从 *Aarhus Citylab* 数据集 [20] 的第一个维度中选取的数据点，以十进制和 IEEE-754 [21] 浮点表示形式展示：(a) 缩放前；(b) 缩放后；(c) 以整数表示。整个数据集中恒定的位以蓝色突出显示。

由于这些位都是常量，因此无需拆分树，只需添加常量位对应的节点。最坏情况下，这需要 l_c（全部位均为常量）。在所有位都加入 B之前，GreedySelect 中的主 while 循环最多运行 l_c 次。每次迭代中，for 循环运行 d 次。除查找 $n_{b,\,i}$ 外，for 循环中的所有步骤耗时均为 $O(1)$；查找该项需要检查 BaseTree 中的每个节点是否需要拆分。最坏情况下，这需要 $O(n)$，因为需要针对一个位进行检查，而该位可能涉及每个样本。全局更新步骤的时间复杂度为常量，但更新 BaseTree 除外，最坏情况下需要 $O(n)$。因此，while 循环最坏情况下的时间复杂度为 $O(ndl_c)$。所以，GreedyGD 配置的总时间复杂度最坏情况下为 $O(nl_c + l_c + ndl_c)$ 或 $O(ndl_c)$。注意到 l_c 是 d 的倍数（$32d$ 适用于单精度数据，$64d$ 适用于双精度数据），可将其简化为 $O(nd^2)$。在实践中，while 循环的运行次数远少于 l_c 次，因为常量位的数量和提前终止都会减少迭代次数。如果将一个或多个维度的所有位都添加到基中，局部优化循环的迭代次数也可能减少。

在数据子集上配置 GreedyGD 可以降低其时间复杂度。假设使用完整数据集进行预处理和常量位识别，之后使用数据子集，则 GreedySelect 的时间复杂度为 $O(nl_c + l_c + ndrl_c)$ 或 $O(nl_c(1 + dr))$，其中 $0 < r < 1$ 是该子集所含数据的比例。如果 $r < 1/d$，则最坏情况下的时间复杂度可降至 $O(nd)$。

压缩过程相对简单，只需将基位与偏差位分开，并添加基 ID 和计数即可。不过，这需要访问完整的二进制数据，因此时间复杂度为 $O(nd)$。

5 性能评估

我们从压缩（第5.1小节）、压缩数据分析（第5.2小节）、配置运行时间（第5.3小节）以及使用数据子集进行配置（第5.4小节）等方面评估了 GreedyGD 的性能。评估共使用了 18 个数据集（表2），涵盖了多种规模、维度、数据类型和精度。请注意，非数值属性以及包含缺失值的行均已省略。芝加哥海滩天气数据集也按单精度浮点数列和整数列进行了拆分，并且省略了芝加哥出租车行程 数据集中的一些派生字段。所有评估均在一台笔记本电脑上进行，配备 Intel Core i7-10510U 1.80 GHz 中央处理器和 16 GB 内存，使用 Python 3.8.10。

首先，GreedyGD 与 GD 的先前版本进行了比较，即 GD-INFO [7] 和 GD-GLEAN [9]。请注意，为了将 GD-INFO 应用于多维数据，有必要修改其终止条件，使其能够稍微越过局部最小值继续探索（与 GreedyGD 类似）。我们还将其与经过增强的 GD-INFO 和 GD-GLEAN 版本进行了比较；这些版本使用预处理，并通过 BaseTree 统计基。我们分别将这些压缩器称为 GD- INFO+ 和 GD-GLEAN+。为了使用 BaseTree，有必要反转迭代顺序，使 GD-INFO+ 从 $B = \emptyset$ 开始并逐步添加位，而不是从 B 中的全部位开始并逐步移除位。在整个评估过程中，我们为 GreedyGD 设置了 $\alpha = 0.1$ 和 $\lambda = 0.02$。

表2：用于评估 GreedyGD 的数据集。

数据集	Type	精度	n	d	大小（kB）
Aarhus Citylab [20]	单精度浮点数	单精度	26,387	4	422
奥胡斯污染 172156 [22]	int	单精度	17,568	5	351
Aarhus 污染 204273 [22]	int	单精度	17,568	5	351
芝加哥海滩水 I [23]	单精度浮点数	单精度	39,829	5	797
芝加哥海滩水质 II [23]	单精度浮点数	单精度	10,034	6	241
芝加哥海滩天气 [24]	单精度浮点数	单精度	86,694	9	3,121
芝加哥海滩天气 [24]	int	单精度	86,763	5	1,735
芝加哥出租车行程 [25]	单精度浮点数	双精度浮点数	3,466,498	10	277,320
CMU IMU 加速度 [26]	单精度浮点数	单精度	134,435	3	1,613
CMU IMU 速度 [26]	单精度浮点数	单精度	134,435	3	1,613
CMU IMU 磁性 [26]	单精度浮点数	单精度	134,435	3	1,613
CMU IMU 位置 [26]	单精度浮点数	单精度	134,435	4	2,151
CMU IMU 全部 [26]	单精度浮点数	单精度	134,435	13	6,991
COMBED 市电 [27]	单精度浮点数	双精度浮点数	82,888	3	995
COMBED 不间断电源 [27]	单精度浮点数	双精度浮点数	86,199	3	1,035
墨尔本城市气候 [28]	单精度浮点数	单精度	56,570	3	679
燃气轮机排放 [29]	单精度浮点数	单精度	36,733	11	1,616
家庭用电量 [30]	单精度浮点数	单精度	2,049,280	7	57,380

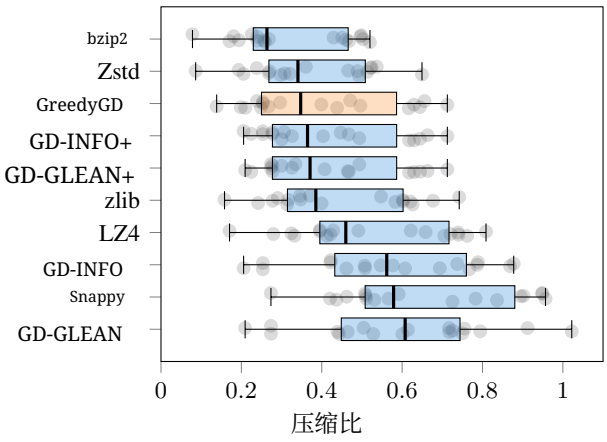

图4：箱线图　每种压缩器的压缩比箱线图，依据中位数压缩比从最佳（顶部）到最差（底部）排列 .

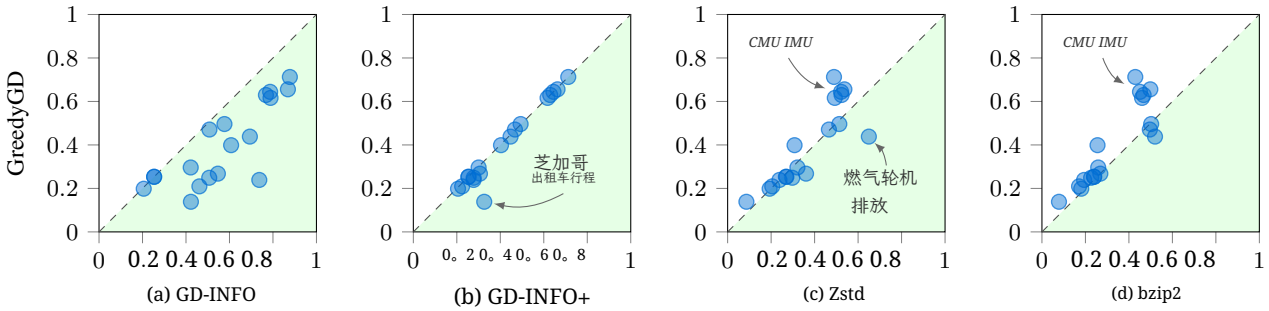

图5：GreedyGD 与选定压缩器在全部 18 个数据集上的压缩比两两比较，并标注了值得注意的数据集。对角线下方（上方）的数据点表示 GreedyGD 的压缩效果更好（更差）。

5.1 压缩比

我们将 GreedyGD 的压缩性能与上述其他 GD 版本以及五种广泛使用的通用压缩器进行了比较，分别是 zlib [10], Zstd [11], Snappy [12], bzip2 [13] 和 LZ4 [14]。所有通用

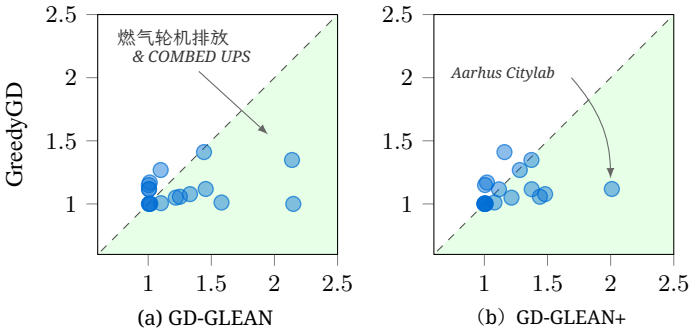

图 6：在全部 18 个数据集上，使用 GreedyGD 和 (a) GD-GLEAN、(b) GD-GLEAN+ 对压缩数据进行 k-means 聚类时近似比的两两比较。对角线下方（上方）的数据点表示 GreedyGD 的分析性能更好（更差）。

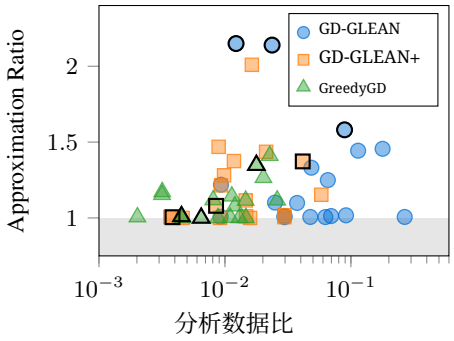

图 7：GreedyGD 和两个 GD-GLEAN 版本在全部数据集上的分析数据比（ADR）与聚类近似比（AR）的联合分布。GD-GLEAN 表现最差的三个数据集以黑色边框标出，适用于全部三个压缩器。

表 3：压缩与分析结果汇总。最佳（次佳）表现以粗体（斜体）显示。

压缩器	CR	ADR	AR	AMI	轮廓系数
GreedyGD	**0.348**	*0.011*	1.109	**0.758**	*0.309*
GD-INFO+	*0.365*	**0.009**	1.246	0.630	0.294
GD-GLEAN+	0.371	0.013	**1.097**	*0.753*	0.295
GD-INFO	0.562	0.014	1.806	0.368	0.162
GD-GLEAN	0.608	0.048	*1.099*	0.639	**0.313**

压缩器均以单次模式运行，并采用最大压缩设置。将这些压缩器应用于全部 18 个数据集后得到的压缩比（CR）分布如图 4 所示，其中灰点表示各个数据集的 CR 值。可以看出，GreedyGD 的表现可与最优秀的通用压缩器相媲美，并优于其他所有 GD 版本。

图 5 展示了更细致的比较。这些图中的每个数据点分别表示 GreedyGD（纵轴）和另一种压缩器（横轴）在单个数据集上的压缩比（CR）。因此，对角线下方的数据点表示 GreedyGD 的表现更好。如图 5(a) 所示，GreedyGD 的表现显著优于 GD-INFO；在压缩方面，GD-INFO 是现有最先进的 GD 版本。事实上，GreedyGD 的压缩比中位数为 0.348，而 GD-INFO 为 0.562，因此 GreedyGD 平均可多压缩 1.6× 。图 5(b) 比较了 GreedyGD 与 GD-INFO+，结果显示，即使经过数据预处理，GD-INFO 的表现也从未优于 GreedyGD。图 5(c) 和 (d) 分别比较了 GreedyGD 与 Zstd 和 bzip2。可以看出，对于大多数数据集，GreedyGD 与这些压缩器的 CR 非常接近，IMU 数据除外。具体而言，GreedyGD 的表现与 Zstd 非常接近，后者的压缩比中位数为 0.341。请注意，如果对基 ID 进行熵编码，GD 还能实现更高的压缩率，但这会牺牲随机访问能力，从而降低分析效率 [7]。

5.2 压缩数据上的分析

无需解压，即可直接从GD压缩数据中获得各种分析任务的近似结果。具体做法是对基底进行分析，并以其计数加权 [8]。这种方法的一大优势是，所需分析的数据点数量仅为原来的很小一部分，因为基底数量通常远少于样本数量，即 $n_b << n$。这意味着运行时间和内存开销都能降低 [9]。因此，为评估压缩数据分析的效果，我们同时考察近似分析结果的质量 以及执行分析所需访问的数据量 。

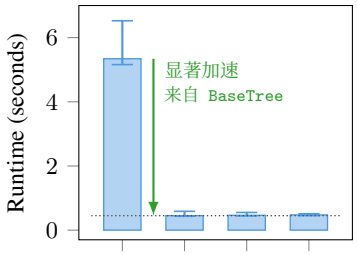

图 8：*COMBED* 主电源 数据集的基于GD的压缩器配置运行时间。

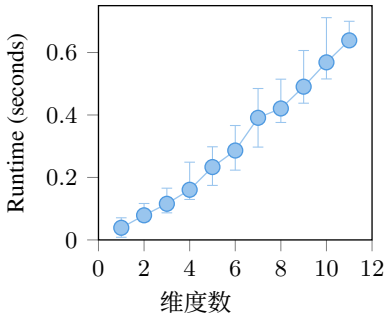

图 9：GreedyGD 配置运行时间与维度的关系，在燃气轮机排放数据集各维度的过采样子集上取平均。

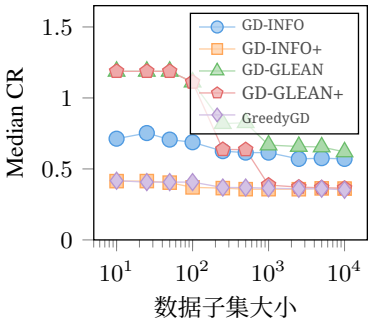

图 10：完整数据集的中位数压缩比与用于配置各压缩器的训练集大小之间的关系。

表3展示了各基于GD的压缩器执行分析所需数据量的中位数，以分析数据比率（ADR）表示。可以看出，GreedyGD仅需使用未压缩数据的1.1%即可执行分析，不到GD-GLEAN所需数据量的四分之一，并且比GD-GLEAN少15%+。

已有研究表明，GD压缩器能够支持较高的聚类性能 [8, 9]。因此，我们使用了 k-均值聚类 [19] 来评估 GreedyGD 的分析性能。我们采用了与 [9]，相同的方法，直接从各基计算聚类中心，并用这些中心对原始数据点进行聚类。随后，利用原始数据的聚类结果计算轮廓系数，并与未压缩数据上的聚类结果进行比较，以计算近似比和调整互信息。请注意，每次聚类均重复 10 次，每次进行 100 次初始化，以尽量减小方差；此外，使用大小为 $n = 10{,}000$ 的样本计算轮廓系数，以避免运行时间过长。表 3 展示了全部 18 个数据集的中位数结果。如表所示，GreedyGD 在各项聚类性能指标上均表现最佳或接近最佳。

我们还进一步考察了 GreedyGD 与 GD-GLEAN 和 GD-GLEAN+ 的性能对比。图 6(a) 和 (b) 分别展示了各数据集上使用 GreedyGD（纵轴）与 GD-GLEAN 和 GD-GLEAN+（横轴）时的近似比。如图所示，GreedyGD 通常能达到相近或更好的性能，其近似比的分布也更加稳定，没有离群值。

总体而言，GreedyGD 在我们测试的所有分析相关指标上都表现稳健。为更全面地说明这一点，我们还给出了图 7，该图绘制了 GreedyGD、GD-GLEAN 和 GD-GLEAN+在所有数据集上的 AR 与 ADR。图中，位置越靠左下，性能越好。可以看出，GreedyGD 的 AR/ADR 值分布最接近最优。每种压缩器在 GD-GLEAN 下表现最差的三个数据集也以黑色边框标出。从图中还可以看出，从 GD-GLEAN 到 GD-GLEAN+，以及从 GD-GLEAN+ 到 GreedyGD，AR 和 ADR 均有所改善。

5.3 配置运行时间

所提出的 BaseTree 算法显著缩短了 GD 的配置运行时间。图 8 展示了在 50 次试验中，选定压缩器处理 *COMBED* 市电 数据集时的配置运行时间中位数。误差线表示 50 次试验中的最短和最长运行时间。可以看出，应用 BaseTree 后速度显著提升。事实上，对比 GD-INFO（5.341 秒）和 GD-INFO+ （0.452 秒）可知，BaseTree 将速度提升了 11.8×倍。GD-GLEAN+（0.463 秒）和 GreedyGD（0.475 秒）由于针对分析进行了优化，速度略慢于 GD-INFO+，但仍分别实现了 11.5×倍和 11.2×倍的速度提升。鉴于数据集大小为 995 kB，GreedyGD 的运行时间对应的吞吐量为 2.1 MB/s。请注意，我们的实现并未针对速度进行优化，且使用 Python 编写，因此绝对运行时间数值不能作为衡量算法能力的依据。

如第4.5小节所述，GreedyGD 的最坏情况配置时间复杂度为 $O(nd^2)$。不过，正如前文所指出的，实际运行时间通常会快得多。为考察配置运行时间与维度之间的关系，我们测量了在燃气轮机排放 数据集的列随机子集上配置 GreedyGD 所需的时间。对于每个维度（1 到 11），我们从数据集的列中随机选取 50 种组合（$d = 1$、10 和 11 维时，组合数不足 50 种），并对每种组合进行 10 次试验。图9展示了每个维度下所有组合和试验运行时间的中位数。误差线表示 5th 和

95th 百分位数。可以看出，GreedyGD 在实际应用中的扩展性远胜于二次增长。也就是说，$d = 11$ 所需的运行时间（0.638 秒）仅为 $d = 1$ （0.039 秒）的 16.4 倍，接近线性增长，而非二次增长。

5.4 数据子集上的配置

我们通过压缩比评估了在数据子集上配置GD的效果，子集大小范围为10至10,000个样本。子集是随机选取的，并且按照第4.4小节的建议，数据预处理和常量位均根据整个数据集确定。图10显示了各子集大小下所有数据集压缩比的中位数。可以看出，GreedyGD和GD-INFO+在所有子集大小下的压缩比都非常稳定。事实上，即使只有250个样本，GreedyGD的压缩比中位数也达到0.368，仅比使用整个数据集进行配置时差5.7%。使用10,000个样本时，GreedyGD的压缩比中位数仅比在完整数据集上配置时的0.353差1.4%。

6 结论

本文提出了一种新的基于GD的压缩算法GreedyGD，旨在实现高精度的压缩数据直接分析，同时保持良好的压缩效果。与现有GD方法相比，所提出的方案提升了分析性能，同时运行速度显著更快，压缩效果也更好。我们还证明，GreedyGD可以在非常小的数据子集上可靠地完成配置，而不会显著影响压缩效果。未来，我们计划研究将GreedyGD用于物联网网络中的在线压缩以及其他分析任务。

7 致谢

本研究由丹麦独立研究委员会资助的压缩物联网数据直接分析（Light-IoT）项目（资助编号：0136- 00376B）以及奥胡斯大学的 DIGITCentre 提供支持。

参考文献

[1] S. Kumar、P. Tiwari 和 M. Zymbler，《物联网是未来技术发展的革命性方法：综述》，《大数据期刊》，第 6 卷，第 1 期，2019 年。[2] W. Yu、F. Liang、X. He、W. G. Hatcher、C. Lu、J. Lin 和 X. Yang，《物联网边缘计算综述》，《*IEEE Access*》，第 6 卷，第 6900–6919 页，2018 年。[3] E. Sisinni、A. Saifullah、S. Han、U. Jennehag 和 M. Gidlund，《工业物联网：挑战、机遇与发展方向》，载于《*IEEE*工业信息学汇刊》，第 14 卷，第 11 期，2018 年，第 4724–4734 页。[4] Y. Lee、E. Hwang 和 J. Choi，《AMI中智能电表读数压缩与认证的统一方法》，载于《*IEEE Access*》，第 7 卷，2019 年，第 34 383–34394 页。[5] R. Vestergaard、Q. Zhang 和 D. E. Lucani，《广义去重：海量小型物联网数据的无损压缩》，载于《欧洲无线通信会议》，2019 年。[6] ——，《采用广义去重对时间序列数据进行无损压缩》，载于《*IEEE*全球通信大会》，2019 年。[7] R. Vestergaard、D. E. Lucani 和 Q. Zhang，《一种面向时间序列数据的随机访问无损压缩方案》，载于《*IEEE INFOCOM*》，2020 年。[8] A. Hurst、Q. Zhang、D. E. Lucani 和 I. Assent，《对广义去重压缩的物联网数据进行直接分析》，载于《*IEEE*全球通信大会》，2021 年。[9] A. Hurst、D. E. Lucani、I. Assent 和 Q. Zhang，《GLEAN：支持近似边缘分析的广义去重》，*IEEE*《物联网期刊》，2022 年。[10] P. Deutsch 和 J.-L. Gailly，《ZLIB压缩数据格式规范3.3版》，1996 年，RFC 1950。[在线]。网址：https://datatracker.ietf.org/doc/html/rfc1950[11] Facebook，《Zstandard——实时数据压缩算法》，访问日期：2022年4月7日。 [在线]。网址：https://facebook.github.io/zstd/[12] Google，《Snappy：快速压缩器/解压缩器》，2021 年。 [在线]。网址：https://github.com/google/ snappy

[13] J. Seward, 《bzip2》, 2019 年。 [在线]。网址：https://sourceware.org/bzip2/[14] Y. Collet、F. Handte、I. Rosen 和 R. Odaira, 《LZ4——极速压缩》, 访问日期：2022年5月4日。[在线]。网址：https://lz4.github.io/lz4/[15] D. Blalock、S. Madden 和 J. Guttag, 《Sprintz：面向物联网的时间序列压缩》, ACM《交互、移动、可穿戴与普适技术》, 第 2 卷, 第 3 期, 第 1–23 页, 2018 年。[16] R. Vestergaard、Q. Zhang、M. Sipos 和 D. E. Lucani, 《Titchy：支持随机访问的物联网在线时间序列压缩》, IEEE《物联网期刊》, 第 8 卷, 第 24 期, 第 17 568–17 583 页, 2021 年。[17] J. Jenkins、I. Arkatkar、S. Lakshminarasimhan、D. A. Boyuka、E. R. Schendel、N. Shah、S. Ethier、C.-S. Chang、J. Chen、H. Kolla、S. Klasky、R. Ross 和 N. F. Samatova, 《 ALACRITY：用于快速原位索引、存储与查询的分析驱动型无损数据压缩》, 载于《大型数据与知识中心系统论丛》第 X 卷, 2013 年, 第 95–114 页。[18] M. Shekelyan、A. Dign¨os 和 J. Gamper, 《DigitHist：具有紧误差界的基于直方图的数据摘要》, 载于《VLDB学报》, 第 10 卷, 第 11 期, 2017 年, 第 1514–1525 页。[19] J. A. Hartigan 和 M. A. Wong, 《算法 AS 136：一种k均值聚类算法》, 《应用统计》, 第 28 卷, 第 1 期, 第 100 页, 1979 年。[20] 奥胡斯市政府, 《传感器数据》, 2017 年。 [在线]。网址：https://www.opendata.dk/city-of-aarhus/sensordata[21] 《IEEE浮点算术标准》, IEEE 754-2019标准（修订 IEEE 754-2008 ）, 2019 年。[22] M. I. Ali 、 F. Gao 和 A. Mileo, 《 CityBench：使用智慧城市数据集评估 RSP引擎的可配置基准》, 载于《语义网——ISWC 2015》, 2015 年, 第 374–389 页。[23]芝加哥市, 《海滩水质——自动传感器》, 2022 年。 [在线]。网址：https: // data.cityofchicago.org/Parks-Recreation/Beach-Water-Quality-Automated-Sensors/qmqz-2xku[24] ——, 《海滩气象站——自动传感器》, 2022 年。 [在线]。网址：https://data.cityofchicago.org/Parks-Recreation/Beach-Weather-Stations-Automated-Sensors/k7hf-8y75[25]——, 《出租车行程——2020年》, 2022 年。 [在线]。网址：https://data.cityofchicago.org/Transportation/Taxi-Trips-2020/r2u4-wwk3[26] F. D. la Torre、J. Hodgins、A. Bargteil、X. Martin、J. Macey、A. Collado 和 P. Beltran, 《卡内基梅隆大学多模态活动（CMU-MMAC）数据库指南》, 卡内基梅隆大学, 技术报告, 2009 年, 技术报告 CMU-RI-TR-08-22。[27] N. Batra、O. Parson、M. Berges、A. Singh 和 A. Rogers, 《商业及住宅建筑非侵入式负荷监测方法比较》, 2014 年, arXiv:1408.6595。[28]墨尔本市, 《传感器读数：8个地点每5分钟记录一次温度、光照和湿度（试点，2014至2015年）》, 2020 年。 [在线]。网址：https://data.melbourne.vic.gov.au/Environment/Sensor-readings-with-temperature-light-humidity-ev/ez6b-syvw[29] H. Kaya 和 P. T¨ufekci, 《燃气轮机CO和NOx排放数据集数据集》, 2019 年。 [在线]。网址：https://archive.ics.uci.edu/ml/datasets/GasTurbineCOandNOxEmissionDataSet[30]G. Hebrail 和 A. Berard, 《个体家庭用电量数据集》, 2012 年。 [在线]。网址：https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption