基于 XOR 的浮点时间序列压缩算法中有效位计数的压缩

赵振山，瞿有利
（北京交通大学交通大数据与人工智能教育部重点实验室，北京 100044）

摘要：本文针对基于 XOR 运算的浮点时间序列压缩算法在中心有效位数量表示上存在的空间冗余问题，提出了一种优化算法 OSBC。首先通过目标算法扫描待压缩序列，随后应用 OSBC 算法确定中心有效位的最佳表示法，并基于此更新目标算法的压缩器，最终利用更新后的压缩器对浮点时间序列进行压缩。结果显示，在中心有效位数量表示方面，经 OSBC 算法优化过的算法比其原始版本节省了大量存储空间。

关键词：浮点时间序列、数据压缩、时序数据库
中图分类号：TP315

Significant bits counting compression in XOR-based floating-point time series compression algorithms

Zhao Zhenshan, Qu Youli
(Key Laboratory of Big Data & Artificial Intelligence in Transportation, Beijing Jiaotong University, Beijing 100044)

Abstract: In this paper, an optimization algorithm OSBC is proposed to address the spatial redundancy in the representation of the number of centrally significant bits in the floating-point time series compression algorithm based on the XOR operation. firstly, the sequence to be compressed is scanned by the target algorithm, then the OSBC algorithm is applied to determine the optimal representation of the centrally significant bits, based on which the compressor of the target algorithm is updated, and the updated compressor finally compresses the floating-point time series. The updated compressor is finally used to compress the floating point time series. The results show that the optimized OSBC algorithm saves a lot of storage space compared with the original version in terms of the number of centre-significant bits.

Key words: floating-point time series; data compression; time seires database

0 引言

时间序列数据在金融市场分析、气候变化监测[1]、医疗健康跟踪[2]和工业生产过程控制[3]等多个领域发挥着关键作用。信息技术的快速进步导致时间序列数据量指数增长，这对数据存储、传输和处理效率提出了重大挑战。

对数据进行压缩是常见的应对策略，好的压缩算法不仅能降低数据存储需要的空间，还能降低数据存储及传输成本。根据数据在压缩过程中是否会丢失原始信息，数据压缩算法分成有损压缩和无损压缩。有损压缩会对数据造成精度损失，因此不适用于科学计算。根据具体的应用场景，压缩算法可以分为通用压缩算法和专门用于时间序列的压缩算法。由于时间序列具有高频率、高密度以及时间戳为关键字等特点，通用压缩算法在压缩时间序列时往往

作者简介：赵振山（1998-），男，硕士，数据压缩
通信联系人：瞿有利（1974-），男，副教授，硕导，人工智能及应用，计算机技术. E-mail: ylqu@bjtu.edu.cn

表现出性能瓶颈和效率低下问题。因此专门用于时间序列的压缩算法[4]成为近年来的研究热点。时间序列数据有浮点数、整数、布尔、字符串等数据类型，其中浮点数应用范围最广。因此，本文重点研究用于浮点数时间序列的压缩算法。

大部分针对浮点数时间序列的压缩算法均采用了异或（XOR）运算。两个浮点数进行 XOR 运算的结果一般包括前导零、有效位和尾部零三部分，分别对这三部分进行压缩通常可以实现较好的压缩效果。FPC 算法[5]通过预测器预测下一个浮点数值，与实际值进行异或运算后进行压缩。Gorilla 算法[6]将当前值与前一个值 XOR 后压缩结果。这两种方法都能生成大量前导零，并通过较少比特的表示达到压缩效果。Chimp 算法[7]根据 XOR 结果尾部零的数量是否超过六个来分类处理。此外，算法还优化了基于前导零数量分布的优化方法。Chimp128[7]算法用当前值和前 128 个数值进行异或，选取最优异或值进行压缩，进而实现了更好的压缩效果。

各种基于 XOR 的浮点时间序列压缩算法以达到较好的压缩效果，但多数算法对 XOR 结果中心位的表示方法存在一定的冗余，本文重点研究基于 XOR 的浮点时间序列压缩算法中有效位计数的压缩。

1 相关工作

1.1 IEEE754 标准

计算机中的浮点数存储普遍采用 IEEE754 标准[8]，依照精度不同分为单精度（32 位）和双精度（64 位）。图 1 呈现了 64 位浮点数按 IEEE754 标准的结构。符号位占 1 位，0 代表正数，1 代表负数，用于表示浮点数的正负。指数位长 11 位，以 1023 为偏移值来表示浮点数的指数值。尾数位占 52 位，表示浮点数的尾数部分及其精度。

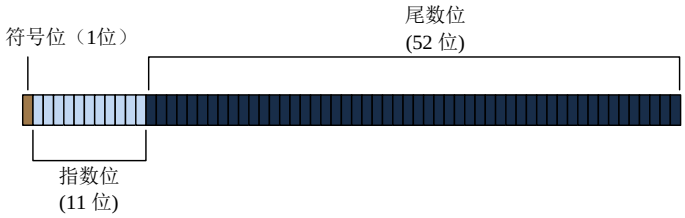

图 1 IEEE754 双精度浮点数[7]2

Fig.1 IEEE754 Double Precision Floating Point[7]2

以 3.5 为例，其二进制形式为 11.1，等同于 1.11×2^1。此数为正数，故符号位为 0。指数值为 1，加上偏置值 1023，得到的 11 位指数位为 10000000000。省略小数点前的 1 后，尾数部分为 11，其后跟随 50 个连续零。

IEEE754 标准明确了规约形式、非规约形式的浮点数以及特殊值，具体详情请参考 IEEE754 文档。专门针对浮点数时间序列的压缩算法通常提供 64 位与 32 位两种格式的压缩器，主要区别在于表示浮点数精度的不同。为了便于说明，本文将以 64 位浮点数为例进行阐述。

1.2 异或（XOR）运算

时间序列是按时间顺序排列的数据点，一般由传感器或物联网设备收集。由于设备连续采集，时间序列中的相邻数据点通常变化范围较小，呈现出时间相关性。例如，股票价格在连续采集时段内变化不大，城市降雨量在短时间范围内也相对稳定。对当前值和前一个数值执行 XOR 运算，结果通常包含许多连续的前导零。以浮点数从 3.25 变为 3.5 为例，其指数往往保持不变。根据 IEEE754 标准，前 12 位分别代表符号和指数，因此相邻数据的这些位通常相同。这导致 XOR 运算后结果包含大量前导零。

64.21: 0 10000000101 0000000011010111000010100011110101110000101000111100

64.25: 0 10000000101 0000000100000000000000000000000000000000000000000000

xor ： 0 00000000000 0000000111010111000010100011110101110000101000111100

图 2 异或运算示例

Fig.2 Examples of XOR operations

图 2 展示了异或运算过程，其中每个浮点数后跟随其 64 位 IEEE754 标准的二进制表示。异或运算的结果（xor）包含前导零、中心有效位和尾部零，分别用绿色、蓝色和红色表示。由于两个浮点数之间的差异较小，它们的 XOR 结果通常包含大量前导零。这一操作在先进的浮点数时间序列压缩算法中常作为数据处理方案，对浮点数时间序列的压缩具有重要意义。

1.3 Gorilla 算法

Gorilla 由 Facebook 公司的 Pelkonen 等研究人员在 2015 年提出，旨在开发一种快速、可扩展且完全基于内存的时间序列数据库。在处理浮点数时间序列压缩时，该时序数据库采用了一种创新的压缩技术，即 Gorilla 算法。在压缩前，Gorilla 首先对浮点序列做 XOR 操作。然后用以下可变长度的编码方案对处理后的 XOR 值进行编码

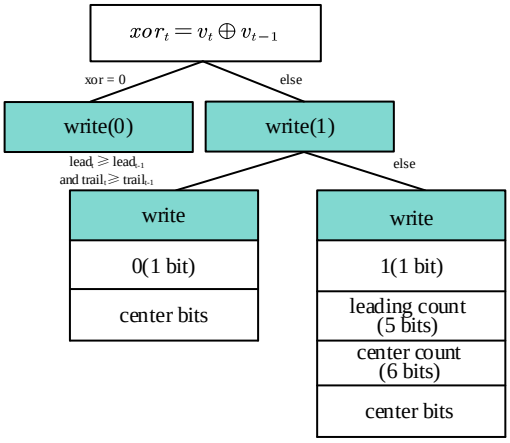

图 3 Gorilla 算法压缩器[7]5

Fig.3 Gorilla Algorithm Compressor[7]5

Gorilla 算法压缩浮点序列的过程是：

1. 第一个值直接用 64 位存储，不做压缩。
2. 如果与前一个值的 XOR 结果为零（两个值相同），存储 1 个比特 0。
3. 否则，计算该 XOR 值的前导零和尾部零的个数，存储 1 比特 1，然后：

3.1. 如果当前 XOR 值的前导零和尾部零个数不少于前一个 XOR 值对应零的个数，存储 1 比特 0，然后仅存储有效中心位。

3.2. 否则，首先用 5 比特表示前导零个数，然后用 6 个比特表示中心有效位。

如图 3 所示，该编码过程分成三种情况，分别用三种前缀码 0、10 和 11 来区分。对于情况 0，XOR 的结果为零，表明当前值与前一个值完全相同，从而避免为存储重复数值而占用大量存储空间的情况。对于情况 10，尽可能地复用之前表示前导零和中心位数量的信息，使得仅需存储中心位，有效节省存储空间。对于情况 11，Gorilla 论文指出，在绝大多数情况下，前导零的数量不会超过 32 个，故使用 5 比特表示前导零数量。如果前导零的数量超过 32 个，则超出的部分被视为中心有效位。随后，使用 6 比特来表示中心有效位的数量，以涵盖所有可能的情况。最后，存储中心有效位。尾部零的数量可以由前导零和中心位的数量推导得出。

以前一节中的两个浮点数 64.21 和 64.25 为例，它们的 XOR 运算结果包括 19 个前导零、43 个中心有效位和 2 个尾部零。采用 Gorilla 编码方法，19 个前导零可以用 5 个比特（10011）来表示，而中心位的数量则用 6 个比特（101011）表示，接着用 43 个比特来表示中心位本身。通过这种方式，原本需要 64 个比特存储的浮点数，现在只需要 54（5+6+43）比特，从而实现数据压缩。

1.4 Chimp 算法

Chimp 算法由 Liakos 等人于 2022 年提出，旨在为时序数据库设计一种高效的浮点数时间序列无损压缩算法。Chimp 算法对 Gorilla 算法进行了较多的改进，同样地，Chimp 在压缩浮点时间序列前先进行异或（XOR）处理，然后按以下规则对这些 XOR 值进行编码：

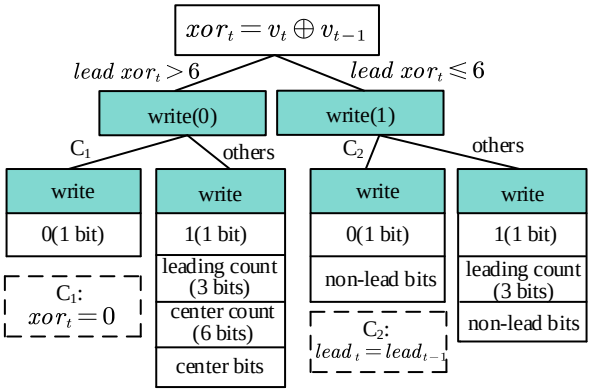

图 4 Chimp 算法压缩器[7]6

Fig.4 Chimp Algorithm Compressor[7]6

1. 第一个值直接存储，不进行压缩。
2. 如果与前一个值的 XOR 结果的尾部零个数超过 6 个，存储 1 个比特 0，然后分两种情况处理：

2.1. 如果 XOR 结果是零，存储 1 个比特 0，表示前后两个值相同。

2.2. 否则，首先存储 1 个比特 1，然后用 3 个比特表示前导零的数量，接着用 6 个比特表示中心有效位的数量，最后存储中心有效位。

3. 如果 XOR 结果的尾部零数量不超过 6 个，存储 1 个比特 1，然后分两种情况处理：

3.1. 如果当前 XOR 值的前导零个数等于上一个 XOR 值前导零的个数，存储 1 个比特 0，然后直接存储中心位。

3.2. 否则，存储 1 个比特 1，然后用 3 比特表示前导零个数，最后存储中心位。

Chimp 对 14 个浮点数时间序列数据集上的 XOR 运算结果中尾部零的数量分布进行了统计。结果表明，在大多数数据集中，XOR 结果的尾部零个数少于 6 个。基于此，Chimp 算法将编码过程分为两大类，每类进一步细分为两种情况，共四种不同的处理方法。这些方法分别用四个前缀码 00、01、10 和 11 来区分，如图 4 所示。

情况 00 和情况 01 的前提条件是 XOR 结果的尾部零个数多于 6 个。对于情况 00，Chimp 保留了以最小代价存储重复值的策略，即用 2 比特表示当前值和前一个值完全相同。对于情况 01，即 XOR 值的尾部零数量大于 6 且 XOR 结果不为零的情况，算法首先用 3 个比特表示前导零的数量，然后用 6 个比特表示中心有效位的数量，最后存储中心有效位。Chimp 指出，用于表示前导零数量的数值中，较小的数值出现次数较少，多数集中在中间范围。因此，Chimp 采用 3 个比特来表示前导零的数量，这 3 个比特共能表示 8 个数值，范围从 0 到 7，分别映射为一组增量指数衰减的值：0、8、12、16、18、20、22 和 24。XOR 结果前导零的数量均用该组数值表示，多出的部分划入中心有效位。例如，如果前导零的数量为 15，则取该组数中的 12 表示前导零个数，余下的 3 个前导零则计入中心有效位，数值 12 映射到 0-7 的结果为 2，用 3 比特表示该结果为 010。Chimp 指出，该做法为每个值提供了 0.51 比特的平均压缩增益。

情况 10 和情况 11 的前提条件是 XOR 结果的尾部零个数不多于 6 个。对于情况 10，若当前 XOR 结果中前导零的数量在映射后与上一结果相同，说明可以复用此信息，此时只需存储去除前导零后的比特位。值得注意的是，这种情况不再使用 6 比特来表示中心位的数量。这是因为尾部零的数量少于 6 个，直接存储中心有效位及尾部零所需的存储空间不会超过先使用 6 比特表示中心位数量、再存储有效位所需的空间。这种方法在大多数情况下能够节约存储空间，从而提高空间性能。对于情况 11，表示当前 XOR 结果中前导零的数量在映射后与上一结果不同，即不能复用该信息，此时用 3 比特表示前导零数量，然后存储去除前导零后的比特位。

同样以 64.21 和 64.25 为例，对 Chimp 编码过程进行阐述，并以 Gorilla 算法作为比较对象。这两个数值的 XOR 结果包含 19 个前导零、43 个中心位和 2 个尾部零。根据 Chimp 编码的规则，前导零数量 19 映射为数字 18（占用 3 比特），而 43 个中心位则被调整为 46 个（增加了 1 个前导零和 2 个尾部零）。对于本例，相比于 Gorilla 算法，Chimp 算法使得表示前导零的比特数减少了 1 个；而中心有效位的表示比特数增加了 3 个；由于不需要使用额外的 6 比特来表示中心有效位，节约了 6 个比特。总体上节省了 4 个比特，显然这一改进的效果是显著的。

Chimp 算法是基于现实世界中浮点序列的 XOR 结果的各部分的数量分布特征设计的。

它注意到 XOR 值的前导零数量在现实世界中分布不均，因此对前导零的表示进行了调整。同时，基于对 XOR 值尾部零数量（通常不超过 6 个）的观察，它对中心位的表示也做了相应调整。基于以上两点，Chimp 算法实现了更高效的数据压缩。

Chimp128 是对 Chimp 算法的一种改进，区别在于，它从当前值前的 128 个数值中选择一个与当前值进行 XOR 运算，以最大化 XOR 结果的尾部零数量。因此，Chimp128 能显著提升压缩率。Chimp 论文的实验结果表明，增加 XOR 结果的尾部零数量对提升时间序列压缩率起到了关键作用。

2 基于 XOR 的浮点时间序列压缩算法中有效位计数的压缩

2.1 算法设计思路

Gorilla 算法利用 6 比特表示浮点序列 XOR 结果中心有效位的数量，以覆盖 1 至 64 的所有数值。在前缀码 10 下，Chimp 算法对记录中心有效位数量的处理方法相同。当中心有效位分布均匀时，使用 6 比特覆盖所有可能数值的做法效果较好。但在现实场景中，中心有效位数量的分布通常不均匀。尽管 6 比特可以覆盖所有数值，但在 64 个数字中，大多数未被使用或使用概率低，导致多数场景下使用 6 比特表示中心有效位存在精度冗余。这种精度冗余为数据压缩提供了较大的空间。

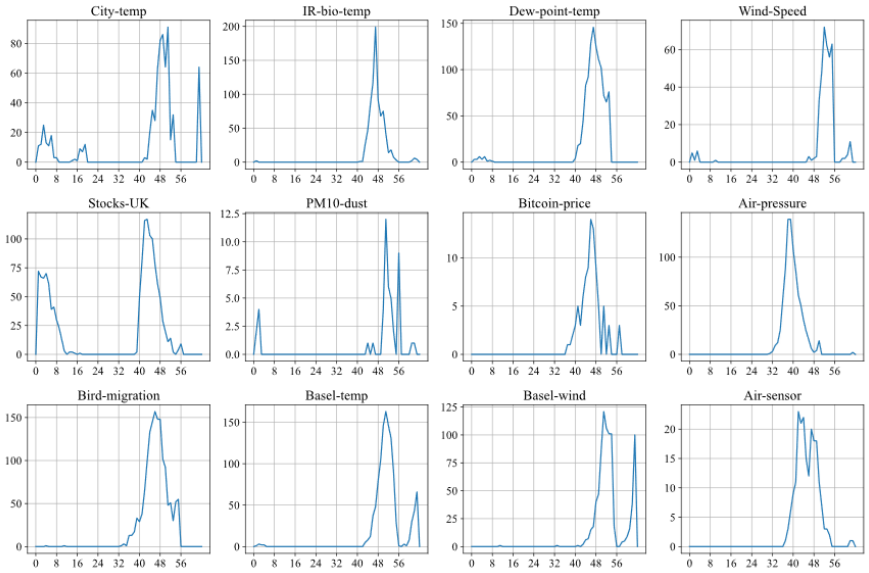

图 5 Gorilla 算法中心位分布

Fig. 5 Distribution of Gorilla algorithm center bits

图 5 展示了使用 Gorilla 算法在 12 个浮点时间序列数据集中心位数量的统计情况。数据集的介绍见实验部分。统计方法是：用 Gorilla 算法扫描浮点序列一次，但不执行压缩操作。扫描时，记录每个中心有效位位数，特别是前缀码“10”对应的情况。图中的数据全部对应于前缀码“10”的中心有效位数量。图表明，大多数浮点时间序列中心有效位数量的分布各异，且不遵循高斯分布。例如，在 Bird 数据集中，32 个以下的中心有效位的占比极低，几乎为

零，说明许多数值未被使用或使用频率低。因此，根据这些情况，通过优化中心有效位的表示，可以进一步实现压缩。

图 6 展示了 Chimp 算法在 12 个数据集上的统计结果。观察可知，Chimp 算法的中心位数量分布与 Gorilla 算法有所不同。造成这一差异的原因是，由于 Gorilla 和 Chimp 算法的不同，前缀码“10”对应情况的出现概率也不同，进而影响了中心有效位数量的分布。同样的，Chimp128 算法的中心有效位数量分布也与上述两种算法不同。

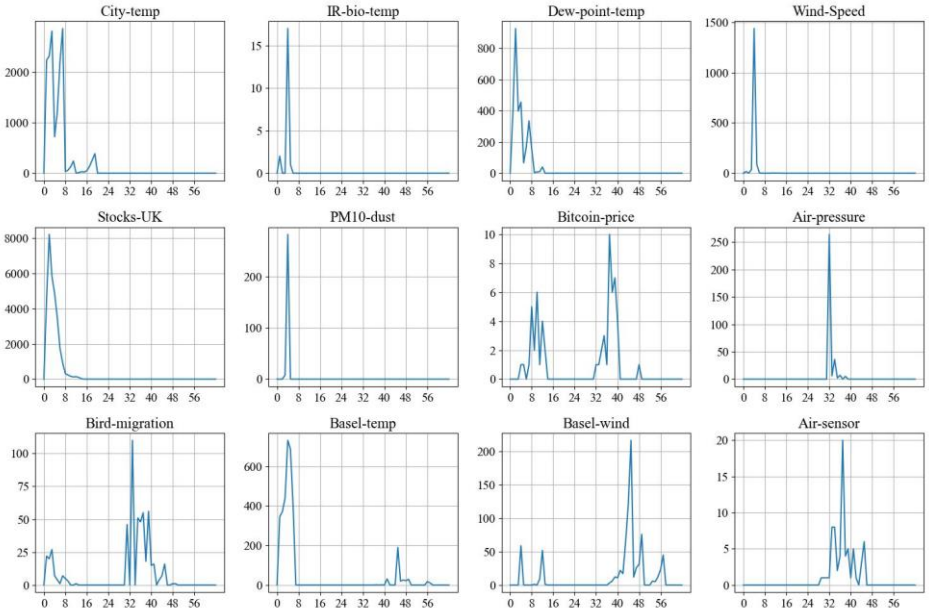

图 6 Chimp 算法中心位分布

Fig. 6 Distribution of Gorilla algorithm center bits

鉴于上述分析，本研究提出了一种中心有效位计数优化算法，旨在改进基于 XOR 的浮点数时间序列压缩算法。此算法被命名为 OSBC（Optimal Significant Bits Counting）算法，核心在于优化中心位的表示方式，而不作为一种独立的浮点数时间序列压缩算法。下面讲述算法的实现过程。

2.2 算法流程与实现

2.2.1 算法整体流程

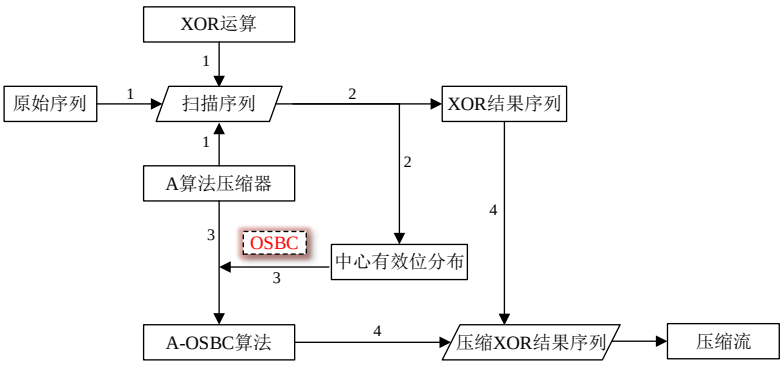

图 7 OSBC 算法优化目标算法

Fig. 7 OSBC algorithm optimize objective algorithm

图 7 描绘了 OSBC 算法优化目标算法的流程。首先，通过异或运算及目标算法 A 的压缩器对初始序列执行第一轮扫描，得到 XOR 结果序列和中心有效位数量分布。基于此分布，OSBC 算法优化 A 算法的中心有效位表示，创建 A-OSBC 压缩器。然后，用 A-OSBC 压缩器对 XOR 结果序列进行第二轮扫描压缩。OSBC 算法由统计和压缩两个阶段构成，首先统计中心有效位分布，然后应用优化后的压缩器进行压缩。过程中仅执行一遍 XOR 运算。OSBC 算法作为一种目标算法的优化部件，对目标算法的中心有效位表示进行优化。

2.2.2 算法实现

本节以 Gorilla 压缩器为例，介绍 OSBC 算法的核心计算过程：优化中心有效位的表示以节省空间。首先，基于出现频率，将中心有效位数量降序排列，并计算前 2、4、8 和 16 个数字的累计出现频次，分别对应 1、2、3 和 4 比特表示方式。以 4 个最频繁的中心有效位（2 比特表示）为例，其累计出现概率为 p，其余的概率总和为 1-p。OSBC 算法使用 2 比特和 6 比特表示中心有效位数量，并用 1 位前缀码区分，用 3 比特和 7 比特替代 Gorilla 的 6 比特表示方法。若浮点数的中心有效位数量是频率最高的 4 个之一，则用 3 比特表示，节省 2 比特；若不是，则用 7 比特表示，比 Gorilla 多用 1 比特。OSBC 算法根据频率选择最节省空间的比特数表示中心有效位。

表 1 不同表示方法对比

Tab. 1 Comparison of different representations

比特数	中心有效位个数	平均空间贡献	出现频率	空间贡献
1	2	4	P_2	$4\times P_2-(1-P_2)$
2	4	3	P_4	$3\times P_4-(1-P_4)$
3	8	2	P_8	$2\times P_8-(1-P_8)$
4	16	1	P_{16}	$1\times P_{16}-(1-P_{16})$

表 1 对比了使用不同比特数表示中心有效位数量的方法。以 4 比特为例，可表示前 16 个最常见的中心位数量。与 Gorilla 算法相比，在该范围内的浮点数平均节省 1 比特，而范围外的平均多耗费 1 比特。因此，采用此方法节省存储空间的条件是 $P_{16}>0.5$，即 $1\times P_{16}-(1-P_{16})>0$。OSBC 算法优选那种能最大化空间节省的压缩方式，从 1、2、3 和 4 比特表示方法中挑选出节省空间最多的一个。

在确定最优比特数表示中心有效位数量后，需要在压缩前注明哪些中心有效位数量出现次数最多。1、2、3 和 4 分别通过二进制 00 到 11 进行标识，故首先需写入两个比特来映射 1 至 4 的值。如果采用 3 比特表示中心有效位数量，则需额外写入 8 个整数，每个占用 6 比特，总共 48 比特顺序表示出现次数最多的前 8 个中心有效位数量。

本文提出的 OSBC 算法旨在用较少的比特表示目标算法中 XOR 结果的中心位数量，进而使目标算法进一步提高空间性能。OSBC 算法不作为独立的浮点序列压缩算法，因此不再给出具体的编码和解码说明，详情见目标算法。

3 实验与分析

3.1 数据集与相关算法

为方便对比实验，本文选取了 14 个浮点数时间序列数据集，与 Chimp 算法使用的浮点数时间序列的相同，详情参见 Chimp 论文[7]9。表 1 展示了数据集的具体信息，括号内为数据集简称。本文选取了 Gorilla、Chimp 以及 Chimp128 算法作为实验对比算法，由于 FPC 算法不记录 XOR 结果的有效位数量，故不与 FPC 算法进行对比分析。

表 2 数据集详情

Tab. 2 Dataset Details

数据集	记录数	时间跨度
City-temp(CT)	2,905,887	25 years
IR-bio-temp(IR)	380,817,839	7 years
Wind-speed(WS)	199,570,396	6 years
Stocks-UK(SUK)	115,146,731	1 year
Stocks-USA(SUSA)	374,428,996	1 year
Dewpoint-temp(DT)	5,413,914	3 years
PM10-dust(PM10)	222,911	5 years
Stocks-DE(SDE)	45,403,710	1 year
Bitcoin-price(BP)	2,741	1 month
Air-pressure(AP)	137,721,453	6 years
Bird-migration(BM)	17,964	1 year
Basel-wind(BW)	124,079	14 years
Basel-temp(BT)	124,079	14 years
Air-sensor(AS)	8664	1 hour

3.2 实验结果与分析

Datasets	XOR-based floating-point time series compression algorithms					
	Gorilla	Gorilla-OSBC	Chimp	Chimp-OSBC	Chimp128	Chimp128-OSBC
CT	4812	3892	93996	65565	475632	330354
IR	4866	3505	120	45	26742	17671
DT	6672	5192	17730	12006	511824	355508
WS	2250	1572	9486	3452	114384	43533
SUK	8016	7012	183600	120704	443904	275764
SUSA	7374	5480	1494	803	195588	119686
SDE	7632	5724	5376	3012	212130	126465
PM10	288	204	1746	582	9264	3133
BP	540	423	354	278	7206	5584
AP	5280	3976	1920	740	19122	10404
BM	8856	7257	3246	2542	53352	38490
BT	7236	5667	19902	13655	428580	263978
BW	5058	3795	4998	3830	48804	25933
AS	1242	1002	426	317	414	321
Average	5009	3907	24600	16252	181925	115487

图 8 实验结果

Fig.8 Experimental results

图 8 展示了三种算法及其经过 OSBC 算法优化版本的实验结果对比。表格记录了每种

压缩算法在各数据集上记录中心有效位数量所需的比特数。以 Gorilla 算法为例，在 CT 数据集中，记录中心有效位数量时原算法需要 4812 比特，而 OSBC 优化后的版本仅需 3892 比特。平均来看，在中心有效位数量的表示方面，Gorilla-OSBC 算法比 Gorilla 算法节省 22%的空间；Chimp-OSBC 算法比 Chimp 算法节省 34%的空间；Chimp128-OSBC 算法比 Chimp128 算法节省 37%的空间。

在压缩速度方面，OSBC 算法专注于优化中心有效位数量的表示。理论上，无论编码还是解码，OSBC 算法优化后只增加了少量判断条件，其他步骤未变，故压缩与解压速度与原始算法相当。

4 结论

本研究优化了基于 XOR 运算的浮点时间序列压缩算法中有效位数量的表示，提出了 OSBC 算法。实验结果显示，OSBC 算法在中心有效位数量的表示上，相较于原算法节省了 22%至 37%的空间。OSBC 算法有效地为基于 XOR 运算的浮点时间序列压缩算法在表示中心有效位数量上节省大量空间。

[参考文献] (References)

[1] Wang Q. Time series simulation method of meteorological elements based on ARIMA model[C]//2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE). IEEE, 2021: 358-362.

[2] 庄淑莲,乔妙,袁钰妍等.基于超声射频流的射频时间序列信号对乳腺良恶性病变的鉴别诊断效能分析[J].新医学,2024,55(02):138-142.

[3] Huang J, Liu C, Yang Y, et al. A GAN-Based Power Quality Anomaly Detection Method for Imbalanced Multivariate Time Series[C]//2023 IEEE 6th International Conference on Computer and Communication Engineering Technology (CCET). IEEE, 2023: 187-191.

[4] Chiarot G, Silvestri C. Time series compression survey[J]. ACM Computing Surveys, 2023, 55(10): 1-32.

[5] Burtscher M, Ratanaworabhan P. High throughput compression of double-precision floating-point data[C]//2007 Data Compression Conference (DCC'07). IEEE, 2007: 293-302.

[6] Pelkonen T, Franklin S, Teller J, et al. Gorilla: A fast, scalable, in-memory time series database[J]. Proceedings of the VLDB Endowment, 2015, 8(12): 1816-1827.

[7] Liakos P, Papakonstantinopoulou K, Kotidis Y. Chimp: efficient lossless floating point compression for time series databases[J]. Proceedings of the VLDB Endowment, 2022, 15(11): 3058-3070.

[8] Cowlishaw M. IEEE standard for floating-point arithmetic[J]. IEEE, New York, 2008: 1132-1138.