УДК 004

Реализация арифметического кодирования с применением целых чисел

Вихляев Дмитрий Романович
Приамурский государственный университет имени Шолом-Алейхема
Студент

Лучанинов Дмитрий Васильевич
Приамурский государственный университет имени Шолом-Алейхема
Старший преподаватель кафедры информационных систем, математики и правовой информатики

Аннотация
В данной статье рассмотрен алгоритм арифметического сжатия и его модификация использующая целые числа. Описана программа данного алгоритма с помощью языка программирования Python. Результатом исследования станет реализация целочисленного арифметического алгоритма и оценка качества его работы.
Ключевые слова: арифметическое кодирование, сжатие данных, теория информации.

Implementation of arithmetic coding using integers

Vikhlyaev Dmitry Romanovich
Sholom-Aleichem Priamursky State University
Student

Luchaninov Dmitry Vasilyevich
Sholom-Aleichem Priamursky State University
Senior lecturer of the Department of Information Systems, Mathematics and Law Informatics

Abstract
This article discusses the algorithm of arithmetic compression and its modification using integers. The program of this algorithm is described using the Python programming language. The result of the study will be the implementation of an integer arithmetic algorithm and an assessment of the quality of its work.
Keywords: arithmetic coding, data compression, information theory.

1 Введение

1.1 Актуальность

Блочная стратегия сжатия является одной из базовых методов архивирования данных. Чем больше и однороднее данные и память, тем эффективнее блочные методы. Данная стратегия основана на использовании

статистических данных. Алгоритмы Хаффмана, Шеннона-Фано, и арифметическое кодирование относятся к данной стратегии. Многие архиваторы включают алгоритм Хаффмана в свои реализации, однако он не всегда является эффективным. Метод Хаффмана приближает относительные частоты появления символов в потоке частотами, кратными степени двойки. Если относительные частоты не являются степенями двойки, сжатие становится менее эффективным в сравнении с оптимальным сжатием по теореме Шеннона. Арифметическое алгоритм даёт сжатие близкое к оптимальному результату, за счёт кодирования всех символов всего в одном числе с плавающей точкой. Среди модификаций, существует метод целочисленного арифметического сжатия, который позволяет сжимать данные и сохранять точность в независимости от разрядности переменных хранящих код на компьютере.

1.2Обзор исследований

И.С.Сергеев, Н.Е.Балакирев провели сравнение алгоритмов сжатия звуковой информации алгоритмом Хаффмана и арифметическим кодированием [1]. М.В.Овчаренко, А.В.Винокуров рассмотрели алгоритмические решения повышения эффективности метода арифметического кодирования [2]. А.Ф.Оськин, В.И.Шайков провели исследования алгоритмов сжатия текстовой информации [3]. Е.А.Беляев, А.М.Тюрликов оценили вероятности появления символа при адаптивном двоичном арифметическом кодировании в задачах сжатия видеоинформации [4]. А.М.Гришина, Т.Н.Ничушкина осуществили анализ алгоритмов сжатия данных и оценка их эффективности [5].

1.3Цель исследования

Цель исследования – реализовать и сравнить классический и целочисленный методы арифметического кодирования, а также оценить качество их работы.

2 Материалы и методы

Для реализации алгоритма используете язык программирования python. Арифметическое кодирование представлено в классическом методе и с использованием целых чисел.

3 Результаты и обсуждения

Арифметический алгоритм относится к блочному методу сжатия, что значит, он сжимает данные в два прохода. В первый раз для сбора частот символов. Во второй непосредственно для кодирования. Процесс декомпрессии основывается на знании декодером кода и таблицы вероятностей символов.

В арифметическом сжатии кодируемый текст, представляется в виде дроби в промежутке от 0 до 1. На первом этапе составляется таблица вероятностей появления каждого символа в исходном тексте. В зависимости

от частоты появления, каждый символ имеет свой собственный под интервал. Пример построения таблицы для строки «SWISS_MISS» (таблица 1).

Таблица 1. Таблица частот символов

Символ	Вероятность	Диапазон
S	0.5	[0.5; 1.0)
W	0.1	[0.4; 0.5)
I	0.2	[0.2; 0.4)
M	0.1	[0.1; 0.2)
_	0.1	[0.0; 0.1)

Данная таблица должна быть известна компрессору и декомпрессору. Кодирование заключается в уменьшении рабочего интервала. Первый символ имеет в распоряжении весь промежуток от 0 до 1. Для каждого следующего символа берётся диапазон соответствующий текущему кодируемому символу. Его длина пропорционально вероятности появления этого символа в потоке. Длина рабочего интервала уменьшается, а точка начала сдвигается пропорционально вероятности текущего символа. В результате окончательная длина интервала равна произведению вероятностей всех встретившихся символов (рис.1).

```
def encode1(word):
    Low=0
    High=1
    code=""
    CumFreq=frequency(word)
    for char in word:
        newLow=Low+(High-Low)*CumFreq[char]["Low"]
        newHigh=Low+(High-Low)*CumFreq[char]["High"]

        Low,High=round(newLow,14),round(newHigh,14)

    code=str(Low)
    return code,CumFreq
```

Рис. 1. Реализация арифметического кодирования

В связи с имеющимися погрешностями, при работе с числами с плавающей точкой, необходимо постоянно округлять результат до разумного количества знаков после запятой. Процесс вычисления на каждой итерации цикла представлен ниже (таблица 2).

Таблица 2. Процесс работы арифметического сжатия

Символ		Вычисление переменных Low и High	Результат
S	L	0.0 + (1.0 – 0.0) × 0.5	0.5
	H	0.0 + (1.0 – 0.0) × 1.0	1.0
W	L	0.5 + (1.0 – 0.5) × 0.4	0.70
	H	0.5 + (1.0 – 0.5) × 0.5	0.75
I	L	0.7 + (0.75 – 0.70) × 0.2	0.71
	H	0.7 + (0.75 – 0.70) × 0.4	0.72
S	L	0.71 + (0.72 – 0.71) × 0.5	0.715

	H	0.71 + (0.72 – 0.71) × 1.0	0.72
S	L	0.715 + (0.72 – 0.715) × 0.5	0.7175
	H	0.715 + (0.72 – 0.715) × 1.0	0.72
–	L	0.7175 + (0.72 – 0.7175) × 0.0	0.7175
	H	0.7175 + (0.72 – 0.7175) × 0.1	0.71775
M	L	0.7175 + (0.71775 – 0.7175) × 0.1	0.717525
	H	0.7175 + (0.71775 – 0.7175) × 0.2	0.717550
I	L	0.717525 + (0.71755 – 0.717525) × 0.2	0.717530
	H	0.717525 + (0.71755 – 0.717525) × 0.4	0.717535
S	L	0.717530 + (0.717535 – 0.717530) × 0.5	0.7175325
	H	0.717530 + (0.717535 – 0.717530) × 1.0	0.717535
S	L	0.7175325 + (0.717535 – 0.7175325) × 0.5	0.71753375
	H	0.7175325 + (0.717535 – 0.7175325) × 1.0	0.717535

Процесс декомпрессии служит для восстановления цепочки. Перебором всех возможных элементов из таблицы частот, декодер распознаёт символ, сравнивая код с диапазоном каждого символа. Далее декодер удаляет эффект символа из кода с помощью вычитания нижнего конца интервала и деления на длину этого интервала (рис.2).

```
def decode1(code,CumFreq,length):
    dec=""
    code=float(code)
    for k in range(length):
        for i in CumFreq:
            if(code>=CumFreq[i]['Low']):
                dec+=i
                code=(code-CumFreq[i]['Low'])/(CumFreq[i]['High']-CumFreq[i]['Low'])
                break

    print(dec)
    return dec
```

Рис. 2. Реализация арифметического декодирования

Для увеличения производительности можно проводить сравнение только с нижней границей интервала, код должен быть больше либо равен ей. Процесс вычисления на каждой итерации цикла представлен ниже (таблица 3).

Таблица 3. Процесс работы арифметического декомпрессора

Символ	Code – Low	Результат	Область	Результат/Область
S	0.71753375 – 0.5	0.21753375	0.5	0.4350675
W	0.4350675 – 0.4	0.0350675	0.1	0.350675
I	0.350675 – 0.2	0.150675	0.2	0.753375
S	0.753375 – 0.5	0.253375	0.5	0.50675
S	0.50675 – 0.5	0.00675	0.5	0.0135
-	0.0135 – 0	0.0135	0.1	0.135
M	0.135 – 0.1	0.035	0.1	0.35
I	0.35 – 0.2	0.15	0.2	0.75
S	0.75 – 0.5	0.25	0.5	0.5
S	0.5 – 0.5	0	0.5	0

Когда весь входной файл будет обработан, выходом алгоритма объявляется любая точка, которая однозначно определяет текущий интервал.

После каждого обработанного символа текущий интервал становится все меньше, поэтому требуется все больше бит, чтобы выразить его.

Приведённый выше алгоритм может сжимать только достаточно короткие цепочки из-за ограничений разрядности переменных. Чтобы избежать этих ограничений, нужно использовать целые числа. Например использовав 16 битные числа от 0 (ox0) до 65536 (ox10000). При этом с потерей точности можно бороться, отслеживая сближения верней и нижней границ. Как только самые левые цифры переменных Low и High становятся одинаковыми, они уже не меняются в дальнейшем. Поэтому когда старшие цифры у обеих границ совпадают, тогда происходит их отбрасывание в другой файл, а оставшиеся цифры сдвигаются влево. При этом в конец переменной Low добавляются нули, в High добавляются единицы (для двоичных чисел) или девятки (для десятичных чисел).

Реализация данного алгоритма требует наличие дополнительных функций и констант (рис.3,4).

```
BIT_LIMIT = 16 #разрядность числа
BASE      = 2 # система счисления
MAX_SIZE  = BASE**BIT_LIMIT-1 #максимальное число
BIT_MOVE  = 1 # сдвиг битов

First_qtr=16384 # Четверть
Half = 32768 # половина
Third_qtr=49152 # 3 четверти
```

Рис. 3. Объявление констант

```
# битовый сдвиг влево заполнение 0 снизу
def bml(num,pos,base):
    return (num<<pos)%base

# битовый сдвиг влево заполнение 1 снизу
def bmlt(num,pos,base):
        num<<=pos
        for j in range(pos):
            num=num|(1<<j)
        return num%base

# заполнение кода
def BitsPlusFollow(bit,bits_to_folow):
    while(bits_to_folow>0):
        bits_to_folow-=1
        code=str(int(bit))
        return code

# Создание таблицы вероятностей символов
def frequency(word):
    High=1
    length=len(word)
    CumFreq={}
    for char, repit in Counter(word).items():
        possible=repit/length

        CumFreq[char]={
            "Low": round(High-possible,14) if High-possible > 1e-14 else 0,
            "High":round(High,14)
        }
        High-=possible

    return CumFreq
```

Рис. 4. Дополнительно используемые функции

Минимизация потерь по точности достигается благодаря тому, что длина целочисленного интервала всегда не менее половины всего интервала. Когда Low или High, одновременно находятся в верхней или нижней половине интервала (Half), тогда просто записываются их одинаковые верхние биты в выходной поток, вдвое увеличивая интервал. Если Low и High, приближаются к середине интервала, оставаясь по разные стороны от его середины, вдвое увеличивается интервал, записывая биты "условно". "Условно" означает, что реально эти биты, выводятся в выходной файл позднее, когда становится известно их значение (рис.5).

```
def encode(word):
    Low=0
    High=MAX_SIZE
    bits_to_folow=1 # количество добавляемых бит
    code=""
    CumFreq=frequency(word) # Таблица частот символов

    for char in word:
        newLow=Low+CumFreq[char]["Low"]*(High-Low+1)
        newHigh=Low+CumFreq[char]["High"]*(High-Low)
        Low,High=int(newLow),int(newHigh)

        while(1):
            if(High<Half):   #если старший бит 0
                code+=BitsPlusFollow(0,bits_to_folow)
                Low = bml(Low,1,MAX_SIZE+1)
                High=bmlt(High,1,MAX_SIZE+1)
                break
            elif(Low>=Half):  #если старший бит 1
                code+=BitsPlusFollow(1,bits_to_folow)
                Low = bml(Low,1,MAX_SIZE+1)
                High=bmlt(High,1,MAX_SIZE+1)
                break
            elif((Low>=First_qtr) and (High<Third_qtr)): #если старшие биты разные
                bits_to_folow+=1
                Low-=First_qtr
                High-=First_qtr
    code+=bin(Low+0)[2:]
    return code,CumFreq
```

Рис. 5. Процедура целочисленного арифметического кодирования

Применение данного кодирования, относится к любой системе счисления. Ниже приведены результаты данного метода с десятичными цифрами в диапазоне от 0 до 10000 (таблица 4).

Таблица 4. Процесс работы целочисленного арифметического сжатия

Символ		Вычисление переменных Low и High	Результат	Изменённый масштаб	Цифры, посылаемые на выход	Сдвиг влево
S	L=	0.0 + (1.0 – 0.0) × 0.5	0.5	5000		5000
	H=	0.0 + (1.0 – 0.0) × 1.0	1.0	9999		9999
W	L=	0.5 + (1.0 – 0.5) × 0.4	0.7	7000	7	0000
	H=	0.5 + (1.0 – 0.5) × 0.5	0.75	7499	7	4999
I	L=	0.0 + (0.5 – 0.0) × 0.2	0.1	1000	1	0000
	H=	0.0 + (0.5 – 0.0) × 0.4	0.2	1999	1	9999
S	L=	0.0 + (1.0 – 0.0) × 0.5	0.5	5000		5000
	H=	0.0 + (1.0 – 0.0) × 1.0	1.0	9999		9999
S	L=	0.5 + (1.0 – 0.5) × 0.5	0.75	7500		7500
	H=	0.5 + (1.0 – 0.5) × 1.0	1.0	9999		9999

-	L=	0.75 + (1.0 – 0.75) × 0.0	0.75	7500	7	5000
	H=	0.75 + (1.0 – 0.75) × 0.1	0.775	7749	7	7499
M	L=	0.5 + (0.75 – 0.5) × 0.1	0.525	5250	5	2500
	H=	0.5 + (0.75 – 0.5) × 0.2	0.55	5499	5	4999
I	L=	0.25 + (0.5 – 0.25) × 0.2	0.3	3000	3	0000
	H=	0.25 + (0.5 – 0.25) × 0.4	0.35	3499	3	4999
S	L=	0.0 + (0.5 – 0.0) × 0.5	0.25	2500		2500
	H=	0.0 + (0.5 – 0.0) × 1.0	0.5	4999		4999
S	L=	0.25 + (0.5 – 0.25) × 0.5	0.375	3750	3750	3750
	H=	0.25 + (0.5 – 0.25) × 1.0	0.5	4999		4999

Декодер работает в обратном порядке. В начале сделаны присвоения, и (первые четыре цифры сжатого файла). Эти переменные обновляются на каждом шаге процедуры декодирования. Low и High приближаются друг к другу и к Code до тех пор, пока их самые значимые цифры не будут совпадать. Тогда они сдвигаются влево, что приводит к их разделению, Code тоже сдвигается. На каждом шаге вычисляется переменная value, которая используется для нахождения диапазона символа из таблицы частот, позволяющего определить текущий символ (рис.6).

```
def decode(Code,CumFreq,length):
    dec=""
    index=BIT_LIMIT
    if(len(Code)%BIT_LIMIT!=0): #если длина кода не кратна 16 дополнить нулями снизу
        Code+=('0'*(len(Code)%BIT_LIMIT))
    value=int(Code[:index],2)
    Low=0
    High=MAX_SIZE
    for k in range(length): #количество символов исходного файла
        freq=((value-Low))/(High-Low+1)
        for i in CumFreq:
            if(freq>=CumFreq[i]['Low']):
                dec+=i
                newLow=Low+(High-Low+1)*CumFreq[i]['Low']
                newHigh=Low+(High-Low)*CumFreq[i]['High']
                Low,High=int(newLow),int(newHigh)
                while(1):
                    if(High<Half):
                        Low = bml(Low,1,MAX_SIZE+1)
                        High=bmlt(High,1,MAX_SIZE+1)
                        value=int((bin(value)[2:]+Code[index:index+1]),2)
                        index+=1
                        break
                    elif(Low>=Half):
                        Low = bml(Low,1,MAX_SIZE+1)
                        High=bmlt(High,1,MAX_SIZE+1)
                        value=int((bin(value)[3:]+Code[index:index+1]),2)
                        index+=1
                        break
                    elif((Low>=First_qtr) and (High<Third_qtr)):
                        Low-=First_qtr
                        High-=First_qtr
                        value-=First_qtr

                break
    return dec
```

Рис. 6. Процедура целочисленного арифметического декодирования

С неточностями арифметики необходимо бороться, выполняя отдельные операции над верхней и нижней границей, синхронно в компрессоре и декомпрессоре. Незначительные потери точности (доли процента при достаточно большом файле) и, соответственно, уменьшение степени сжатия по сравнению с идеальным алгоритмом происходят во время

операции деления, при округлении относительных частот до целого, при записи последних битов в файл.

Для того чтобы выяснить степень компрессии, достигаемой с помощью арифметического сжатия, необходимо переводить все результаты в двоичную форму перед тем, как оценивать эффективность компрессии. Кодируемая строка «SWISS_MISS» имеет двоичный вид «10110111101100000100». Строка из 10 символов сжата в строку из 20 битов. Энтропия исходной строки примерно равна «19.6», что значит 20 бит – это минимальное число, необходимое для практического кодирования этой строки.

Алгоритм Хаффмана для данной строки выдал длину в 20 бит, а алгоритм Шеннона-Фано 22 бита.

Таким образом, можно сказать, что метод арифметического кодирования является оптимальным, а применение целочисленной арифметики даёт возможность использовать его на практике.

Выводы

В результате исследования были приведены реализации арифметического кодирования. Проведено сравнение двух разных методов. Описаны преимущества и недостатки сжатия данного алгоритма.

Библиографический список

1. Сергеев И.С., Балакирев Н.Е. Сравнение алгоритмов сжатия звуковой информации алгоритмом Хаффмана и арифметическим кодированием // Наукосфера. 2022. № 8-2. С. 31-35.
2. Овчаренко М.В., Винокуров А.В. Алгоритмические решения повышения эффективности метода арифметического кодирования // В сборнике: Теоретические и прикладные проблемы науки и образования в 21 веке. сборник научных трудов по материалам Международной заочной научно-практической конференции: в 10 частях. 2012. С. 102-103.
3. Оськин А.Ф., Шайков В.И. Алгоритм сжатия текстовой информации // Информационный бюллетень ассоциации История и компьютер. 1996. № 17. С. 157-158.
4. Беляев Е.А., Тюрликов А.М. Оценка вероятности появления символа при адаптивном двоичном арифметическом кодировании в задачах сжатия видеоинформации // Цифровая обработка сигналов. 2007. № 3. С. 20-24.
5. Гришина А.М., Ничушкина Т.Н. Анализ алгоритмов сжатия данных и оценка их эффективности // Технологии инженерных и информационных систем. 2021. № 2. С. 48-55.
6. Фастов В.С., Атакищев О.И., Старостенко И.В. Основные особенности алгоритма двоичного арифметического кодирования // В сборнике: Молодежь и XXI век - 2016. Материалы VI Международной молодежной научной конференции: в 4-х томах. Ответственный редактор Горохов А.А., 2016. С. 129-131.
7. Череданова Е.М., Мамченко Е.А. Алгоритм сжатия текстовых файлов //

Молодой ученый. 2017. № 44 (178). С. 24-26.
8. Хашин С.И., Хашина Ю.А. Оптимизация алгоритма сжатия методами булевой алгебры // Вестник Ивановского государственного университета. 2003. № 3. С. 138-140.
9. Киамов И.И., Галимуллина Г.М. Сравнительный анализ алгоритмов сжатия // В сборнике: тенденции и закономерности развития современного российского общества: экономика, политика, социально-культурная и правовая сферы. материалы всероссийской научно-практической конференции с международным участием: в 2-х частях. 2016. С. 29.