АЛГОРИТМ СЖАТИЯ ИНФОРМАЦИИ С ИСПОЛЬЗОВАНИЕМ НУМЕРАЦИИ ПЕРЕСТАНОВОК

Могилевец Д.Э., студент гр.353505, Максименков В.А., студент гр.353502

Белорусский государственный университет информатики и радиоэлектроники
г. Минск, Республика Беларусь

Калугина М.А. – канд. физ.-мат. наук, доцент

Аннотация. В данной работе приведено описание алгоритма сжатия информации без потерь с использованием идеи нумерации подпоследовательности различных подряд идущих элементов в исходной последовательности данных, доказана его корректность с обоснованием вычислительной сложности и внесены предложения по области применения.

Ключевые слова. Последовательность, перестановка, кодирование, сжатие информации, лексикографический номер, бит, множество.

Алгоритмы сжатия данных важны как для увеличения скорости их передачи, так и для уменьшения используемой памяти. По сохранности информации можно выделить два основных класса: алгоритмы сжатия с потерями и без потерь. В представленной работе рассматривается алгоритм сжатия данных без потерь, основанный на разбиении исходной последовательности на перестановки уникальных элементов и дальнейшем сжатии этих перестановок.

Сначала определимся с терминологией. Под последовательностью будем понимать упорядоченное множество целых неотрицательных чисел, а длиной такой последовательности будем называть число её элементов. Используя под понятием разрядности числа количество бит в его записи, будем считать битовой длиной последовательности число бит, необходимое для записи всей последовательности. В таком случае битовую длину последовательности получим, умножив разрядность максимального её элемента на длину самой последовательности. Назовём частичной перестановкой последовательность, в которой каждое число уникально. Так, например, [9,6,7], [3,4,16,9], [7,8,0,3] – частичные перестановки, а [9,3,2,9], [5,6,9,5,7,12] – нет. Тогда под перестановкой (полной перестановкой) будем понимать частичную перестановку длины K , где каждое число не превосходит $K - 1$. Как можно заметить, введенное понятие не противоречит классическому определению перестановки. Например, [2,1,0,3], [0,6,5,3,2,1,4] – перестановки (или полные перестановки).

Из перечислительной комбинаторики известно, что число полных перестановок длины K равно $K!$. Найдём число частичных перестановок длины L , состоящих из чисел, меньших K . Понятно, что оно равно числу размещений L элементов по K местам. Поэтому искомое значение определяется по формуле

$$A_K^L = \frac{K!}{(K-L)!}. \quad (1)$$

Тогда общее количество частичных перестановок из чисел, не превосходящих K , можно найти как сумму

$$\sum_{L=1}^K \frac{K!}{(K-L)!} = K! \sum_{L=1}^K \frac{1}{(K-L)!} = K! \sum_{L=0}^{K-1} \frac{1}{L!}. \quad (2)$$

Из теории информации (см., например, [1, с. 22]) известно, что для хранения одного элемента множества мощности R необходимо не менее $\log_2(R)$ бит при равновероятном распределении его элементов. Таким образом, для хранения всей перестановки понадобится не менее $\log_2\left(K! \sum_{L=0}^{K-1} \frac{1}{L!}\right)$ бит. Сумма, на которую умножается $K!$, как известно, достаточно быстро сходится к e при росте K . Тогда для логарифма, используя формулу Стирлинга для асимптотической оценки факториала, можно получить верхнюю оценку:

$$\log_2\left(K! \sum_{L=0}^{K-1} \frac{1}{L!}\right) \leq \log_2(eK!) \approx \log_2(e) + \log_2\left(\sqrt{2\pi K} \left(\frac{K}{e}\right)^K\right) \approx K \log_2(K) - K + 3, \quad (3)$$

что меньше, чем $K \log_2(K)$ бит, которые необходимо применить при явном кодировании последовательности.

Идея алгоритма и его краткое описание

Пусть каждое из чисел в перестановке не превосходит 2^n и для хранения каждого из них используется n бит. Тогда для хранения полной перестановки требуется $n2^n$ бит. Однако по формуле (3) видно, что на самом деле достаточно всего $(n-1) \cdot 2^n$ бит при $n \geq 2$. Тогда экономится 2^n бит, а с учётом грубости оценки – ещё больше.

Для получения этого выигрыша на практике предлагается выстроить все перестановки в лексикографическом порядке и использовать в качестве кодирования просто номер этой самой перестановки. Возникает проблема: получить из частичной перестановки её номер в словаре всех перестановок и, наоборот, из номера перестановки в словаре всех перестановок получить её саму. Эти задачи решают процедуры 1 и 2, описанные ниже.

Приведем описание алгоритма сжатия:

- отыщем наилучшую на данный момент частичную перестановку (самую длинную из них) при помощи процедуры 3;
- заменим данную частичную перестановку на её код;
- в последовательности, из которой удалена текущая наилучшая перестановка, вновь отыщем самую длинную;
- если она включает в себя место, где на прошлом ходу была удалена перестановка, то запишем, помимо кода текущей перестановки, также и индекс, где была расположена удалённая перестановка;
- в случае, если перестановка слишком мала, чтобы её замена на её код давала выигрыш по количеству бит, будем считать кодом данной перестановки её саму.

Данный процесс будем повторять до тех пор, пока у нас не останется последовательность длины 0. Для раскодирования необходимо будет добавить в начало каждого сжатого блока бит, означающий, применено ли кодирование при помощи номера перестановки, а также n бит, означающих длину сжатой перестановки.

Процедура получения номера частичной перестановки (Процедура 1)

Пусть дана частичная перестановка $perm$ длины n . Необходимо найти её номер в лексикографическом словаре всех частичных перестановок.

Чтобы найти её лексикографический номер, достаточно на каждом шаге посмотреть, каким по счёту среди ещё не использованных элементов является текущий элемент. Затем умножить этот индекс на факториал оставшегося количества позиций и суммировать результаты.

Процедура 1 принимает список $perm$ длины n , а также число K .

```

1:  $elems \leftarrow [0, 1, \dots, K - 1]$ 
2:  $counts \leftarrow [1]$ 
3:  $curr_{sa} \leftarrow 1$ 
4:  $for\ i \leftarrow 1\ K - 1\ do$ 
5:  $curr_{sa} \leftarrow (i + 1) * curr_{sa} + 1$ 
6:  $countsss.append(curr_{sa})$ 
7:  $endfor$ 
8:  $number \leftarrow 0$ 
9:  $for\ i \leftarrow 0\ n - 1\ do$ 
10:  $index \leftarrow elems.index(perm[i])$ 
11:  $remaining \leftarrow K - i - 1$ 
12:  $number \leftarrow number + index * counts[remaining]$ 
13:  $elems.remove(index)$ 
14:  $endfor$ 
15:  $return\ number + n - 1$ 

```

Первая часть процедуры ответственна за вычисление $\sum_{i=0}^n A_n^i$ по всем n от 1 до $K - 1$. Для вычисления используется рекуррентное соотношение $\sum_{i=0}^n A_n^i = n \sum_{i=0}^{n-1} A_{n-1}^i + 1$, которое легко доказывается при непосредственной записи A_n^i . Далее с использованием вычисленных значений находится номер перестановки.

Вклад (в смысле того, насколько больше он делает номер) каждого отдельного элемента перестановки можно оценить как количество перестановок, начинающихся на любой элемент меньше, при условии, что все последующие элементы могут быть любыми (или вообще отсутствовать). Но тогда количество элементов, меньше добавившихся из-за нового элемента, можно найти как $m \sum_{i=0}^n A_n^i$, где m – сам этот

элемент, а n – число не встречавшихся ранее в перестановке чисел из множества $\{1, \dots, K\}$. В конце к искомому номеру добавляется длина последовательности без единицы, данный вклад вносит последнее число.

Общая сложность алгоритма может быть оценена как $O(n + K)$, поскольку он состоит из двух циклов: первый по K , а второй – по n . В реальной программной реализации цикл по K можно вынести за тело процедуры и передавать в неё итоговый массив. Тогда сложность процедуры уменьшится до $O(n)$.

Процедура получения частичной перестановки из её номера (Процедура 2)

Пусть даны длина перестановки n и её номер num в лексикографическом порядке (номер считается от 0). Необходимо восстановить перестановку. Для этого последовательно «вырезаем» из списка доступных элементов тот, который соответствует текущему блоку перестановок размера $\sum_{i=0}^n A_n^i$. Сначала определяем, какой элемент идёт на первую позицию, потом — на вторую и так далее. Алгоритм работает путем последовательного вычисления индекса элемента, который должен быть на текущей позиции, и уменьшения номера оставшейся перестановки.

Процедура 2 принимает число элементов K и номер перестановки num .

```

1:  $elems \leftarrow [0, 1, \dots, K - 1]$ 
2:  $perm \leftarrow [ ]$ 
3:  $remaining_{num} \leftarrow num$ 
4:  $counts \leftarrow [1]$ 
4:  $curr_{sa} \leftarrow 1$ 
5:  $for\ i \leftarrow 1\ K - 1\ do$ 
6:  $curr_{sa} \leftarrow (i + 1) * curr_{sa} + 1$ 
7:  $countss.append(curr_{sa})$ 
8:  $endfor$ 
9:  $for\ i \leftarrow 0\ K - 1\ do$ 
10:  $if\ elems\ is\ empty\ then$ 
11:  $break$ 
12:  $for\ candidate \in elems\ do$ 
13:  $n \leftarrow K - (i + 1)$ 
14:  $\sum \leftarrow counts[n]$ 
15:  $if\ remaining_{num} \geq \sum\ then$ 
16:  $remaining_{num} \leftarrow remaining_{num} - \sum$ 
17:  $continue$ 
18:  $else$ 
19:  $perm.append(candidate)$ 
20:  $elems.remove(candidate)$ 
21:  $break$ 
22:  $endfor$ 
23:  $if\ remaining_{num} = 0\ then$ 
24:  $break$ 
25:  $if\ (K - (i + 1)) > 0\ then$ 
26:  $remaining_{num} \leftarrow remaining_{num} - 1$ 
27:  $endfor$ 
28:  $return\ perm$ 

```

Процесс начинается с создания списка всех доступных элементов и массива $counts$ для хранения числа перестановок, которые можно собрать с оставшимися элементами. На каждом шаге алгоритм вычисляет, сколько перестановок начинается с каждого возможного элемента. Для этого используется факториал, как и в первой процедуре, но в данном случае задача заключается в том, чтобы вычленил именно тот элемент, который соответствует текущему номеру в лексикографическом порядке.

После нахождения нужного элемента его удаляют из списка доступных, а оставшийся номер перестановки уменьшается, что позволяет точно определить следующий элемент. Этот процесс повторяется до тех пор, пока не будет восстановлена вся перестановка.

Время выполнения второй процедуры оценивается как $O(n + K)$, где n – длина перестановки, а K – количество доступных элементов. Алгоритм состоит из двух основных частей: вычисления значений в массиве $counts$ (что требует $O(K)$ времени) и восстановления самой перестановки (каждый элемент выбирается за $O(n^2)$ времени, так как на каждом шаге нужно пройти по списку доступных элементов).

Таким образом, общая сложность процедуры пропорциональна сумме этих двух операций, что даёт оценку $O(n^2 + K)$.

Поиск лучшей перестановки для сжатия за линию (Процедура 3)

Дана последовательность длины n . В ней необходимо отыскать самую длинную непрерывную подпоследовательность с уникальными элементами.

Для этого будем использовать следующий алгоритм, работающий за $O(n)$, где n – длина последовательности. Будем использовать скользящее окно.

Процедура 3 принимает последовательность *sequence* длины n .

```
1: 0
2: ← 0
3:  $max_{len} \leftarrow 0$ 
4:  $best_{pair} \leftarrow (0,0)$ 
5:  $seen \leftarrow \emptyset$ 
6: while < ndo
7: if sequence[ ]  $\notin$  seen then
8: seen.add(sequence[ ])
9: ← +1
10: if –  $max_{len}$  then
11:  $max_{len} \leftarrow$  –
12:  $best_{pair} \leftarrow ()$ 
13: else
14: seen.remove(sequence
15: 1
16: endwhile
17: return  $best_{pair}$ 
```

Поясним работу процедуры. Инициализируем два индекса: один будет указывать на левую границу текущего окна, другой – на правую. Будем поддерживать множество тех элементов, которые в данный момент находятся в скользящем окне, с помощью структуры данных, способной за амортизированное $O(1)$ добавлять, удалять уникальные элементы и проверять наличие заданного элемента. Этим требованиям удовлетворяет, например, хэш-таблица.

На каждом шаге алгоритма будем сдвигать правую границу вправо на единицу. Если нового элемента нет, то добавим его во множество и продолжим. В случае, когда элемент, на который сейчас указывает правый индекс, уже присутствует во множестве – запишем текущие правую и левую границу и будем сдвигать левую границу до тех пор, пока не окажемся на позиции после первого вхождения повторившегося элемента. Параллельно будем удалять все встретившиеся элементы из множества.

Повторяем это, пока правая граница не дойдёт до конца списка. Далее из всех выписанных пар левых и правых индексов выберем такую, где разность будет максимальной. Это и будет искомая перестановка.

Количество шагов алгоритма равно количеству различных значений, которые будут принимать левый и правый индексы. Поскольку и правая и левая границы принимают каждое из значений от 1 до n не более одного раза, а все операции вставки и удаления происходят за $O(1)$, то конечная сложность алгоритма – $O(n)$.

Область применения

Во многих прикладных задачах данные естественным образом представляют собой последовательности достаточно длинных перестановок или подпоследовательностей уникальных элементов. В таких случаях предложенный метод сжатия информации с использованием лексикографической нумерации перестановок может быть особенно эффективен. Укажем наиболее перспективные направления.

При организации индексов или хранении упорядоченных ключей. Представление данных в виде перестановок и их компактная нумерация позволяют сократить объём хранимой информации и ускорить операции поиска.

В системах мониторинга и сбора логов часто возникают последовательности уникальных событий (например, запросов, команд, состояний), которые интерпретируются как перестановки. Их сжатие описанным методом повышает эффективность хранения и анализа.

В ряде задач (например, ранжирование, рекомендации, комбинаторная оптимизация) данные представлены перестановками объектов. Использование сжатия на основе лексикографических номеров может быть полезно при передаче данных между компонентами системы или для компактного хранения обучающих выборок.

61-я научная конференция аспирантов, магистрантов и студентов БГУИР

Для описанных алгоритмов были разработаны программные реализации на языке программирования Python. Планируется проведение экспериментального исследования для сравнительного анализа временной сложности процедур и полученных теоретических оценок.

Список использованных источников:

1. Уэлстид С. *Фракталы и вейвлеты для сжатия изображений в действии* : учеб. пособие / С. Уэлстид – Москва: Триумф, 2003 -320 с.: ил.

DATA COMPRESSION ALGORITHM USING PERMUTATIONS NUMBERING

Mahiliavets D.E, Maksimenkau U.A.

Belarusian State University of Informatics and Radioelectronics, Minsk, Republic of Belarus

Kalugina M.A. – PhD in Physics and Mathematics, Associate Professor

Annotation. This paper describes a lossless data compression algorithm using the idea of numbering a subsequence of different consecutive elements in the original data sequence, proves the correctness of this method and options for its use in various areas of information technology.

Keywords. Sequence, permutation, coding, compression, lexicographic number, bit, set.