Отчёт по лабараторной работе №1

Лев Довжик, M4139c

Описание алгоритма

Базовым алгоритмом был классический арифметический кодер, поверх которого реализовывалось нумерационное кодирование. Однако в реальности нам неизвестен размер текста (N) мы записываем его в первый 8 байт сжатого файла.

Так же в реальности мы не можем оперировать вещественными числами бесконечно точности, так что реализация пользуется целочисленной арифетикой. В классическом АК у нас есть два числа $low = 0.0$ и $hig = 1.0$. Однако заметим, что $0.(1) = 1$, так как $2 \cdot 0.(1) - 0.(1) = 1.0$. Исходя из этого будет симулировать бесконечную последовательность 0 и бескнечную последовательность 1 с помощью целочисленных типов, а вместо кумулятивной вероятности символа будем поддерживать кумулятивную частоту.

Теоритеские оценки показывают, что разрядности регистра для математических операций должна быть как миниум равна суммарной разрядности кумулятиной вероятности и кодового буффера, $FREQUENCY\_BITS$ и $CODE\_BITS$ соотвественно. При это также для корректной работы необходимо выполение условия $CODE\_BITS \geq FREQUENCY\_BITS + 2$. В данной работе используется 64-битный тип данных для арифметики, в следствии чего $CODE\_BITS = 33$, $FREQUENCY\_BITS = 31$, что делает масимальный размер сжимаемых файлов $2^{31} - 1$ байт. Однако это условие можно ослабить, если использовать нестандартные расширешия для 128-битных типов данных или более медленной длинной арифметики.

Кодер и декодер в программе при обработке каждого символа принимают текущую модель алфавита в качетве аргумента, что позволяет, например, без проблем начать кодировать символы с ненулевой встречаемостью сразу после их частот.

Для ускорения работы с равномерным распледеним был реализован класс **uniform_model**, который хранит лишь суммарное количество элементов, а всё отсальное считает на ходу на его основе. Для работы же с распредлением байтов исходного текста существует класс **byte_model**, который внутри себя хранит дерево Фенвика, позволяющее узнавать сумму на префиксе (то есть кумулятивную частоту) и делать точеные изменения за $O(\log n)$, где n — количество элементов. В силу того что в нумерационном кодировании мы при кодировании или декодировании очередного

символа уменьшаем его частоту, данная стурктура позволяет успокрить работу по сравнению с префиксными суммами, выполняющими те же операции за $O(1)$ и $O(n)$ соотвественно.

Так же стоит немного добавить про завершение кодирования. В конце кодер остаётся в трёх возможных состояниях:

1. $high = 11xxx,\ low = 01yyy$
2. $high = 11xxx,\ low = 00yyy$
3. $high = 10xxx,\ low = 00yyy$

Нам нужно вывести несколько битов, которые буду образовывать собой значение между low и $high$. Для этого в случах 2 и 3 достаточно вывести **01**, а в случае 1 — **10**. Однако у нас так же могут оставаться биты в $pending\_bits$, так что мы просто увеличим его на 1, чтобы гарантированно покрыть описанные выше, а затем позовём стандратную функцию для вывода битов, с нужным битом в зависимости от значения low.

Результаты работы

Файл	**H(X)**	**H(X|X)**	**H(X|XX)**	**Затраты на символ**	**Размер сжатого в байтах**
BIB	5.201	3.364	2.307	5.209	72450
BOOK1	4.527	3.585	2.814	4.528	435171
BOOK2	4.793	3.745	2.756	4.795	366100
GEO	5.646	4.254	3.458	5.669	72569
NEWS	5.190	4.092	2.922	5.193	244785
OBJ1	5.948	3.463	1.400	6.044	16247
OBJ2	6.260	3.870	2.265	6.270	193455
PAPER1	4.983	3.646	2.332	5.002	33240
PAPER2	4.601	3.522	2.513	4.613	47402
PIC	1.210	0.823	0.705	1.213	77815
PROGC	5.199	3.603	2.134	5.225	25869
PROGL	4.770	3.212	2.044	4.784	42840
PROGP	4.869	3.188	1.755	4.888	30173
TRANS	5.533	3.355	1.930	5.545	64940

Суммарный размер всех сжатых файлов: 1723056 байт.