

Практическая работа №2. Статистическое кодирование

Цель – закрепить знания об общих принципах статистического кодирования, изучить выбранную схему кодирования и научиться оценивать эффективность сжатия

Задача – реализовать схему кодирования и декодирования в виде функций

На выбор даётся список кодов из числа пройденных в курсе. Каждый студент реализует свой вариант. В конце будет возможность сравнить эффективность методов.

Работа выполняется в среде Matlab. По согласованию возможно выбрать другой язык программирования, однако усилия по переносу данных и функций под него нужно будет взять на себя. Файл **statistics_coding.m** содержит заготовку рабочего скрипта. В ходе работы запускается этот скрипт. Папка **data/** содержит исходные данные для кодирования в файлах формата данных Matlab в виде символьных массивов. Выбор файла производится в аргументе функции `load(...)`.

Примечание: символьный массив №7 не следует использовать для тестирования работоспособности программы из-за его длины. Его следует использовать в рабочей программе для сбора статистики, так как кодирование может быть долгим.

При реализации можно не учитывать вопросы оптимизации кода. При написании программ допустимо пользоваться синтаксисом, аналогичным Си, однако помните, что код, написанный с использованием матричных операций и прочих возможностей Matlab, выглядит короче и проще, а также нередко работает ощутимо быстрее, чем программа, использующая множество циклов.

По результатам моделирования заполняется таблица по адресу: https://docs.google.com/spreadsheets/d/1nI58nm2Df42BokNro_AsRw_CozGiRTisgrPTRbm_t8A/edit?usp=sharing. При необходимости приводить характеристики при различных настройках алгоритма можно создавать строки внутри своего раздела. Для словарных методов обязательно требуется произвести сравнение эффективности при разных настройках размерностей индексов/длин буферов для текстовых массивов (№3, №7). На места пропусков выписывается *средняя длина кода на символ сообщения*. Для неадаптивных алгоритмов желательно отдельно вычислять объём метаданных и основных данных. Все величины приводить к *битам*.

Для сдачи нужно не только написать рабочую пару кодера и декодера, но и верно проинтерпретировать результаты, получаемые для разных последовательностей, а также ответить на вопросы по теории.

1. Реализуемые функции

Реализация состоит в написании следующих функций:

1. `function [encoded_data] = encodeData(data)`

Принимает на вход сообщение `data`. Сообщение есть линейный массив символов (`char`) в стандартной 8-битной кодировке.

Сообщение кодируется и возвращается в аргументе `encoded_data`. Если метод предполагает предварительный анализ контента (полуадаптивные методы), строится информационная модель и по ней формируется способ кодирования. Необходимые декодеру метаданные для восстановления кода (подробнее в разделе ниже) при наличии в битовом представлении включаются в состав аргумента `encoded_data`.

Оба выходных аргумента должны представлять одномерные массивы численного (0/1/2), символьного ('0'/'1'/'2') либо логического (`false/true`) типа.

2. `function decoded_data = decodeData(encoded_data)`

Читает сжатое сообщение `encoded_data`, возможно содержащее метаданные. Сообщение декодируется и возвращается: `decoded_data`.

2. Список методов кодирования к выбору

Уникально-префиксные коды (коды на деревьях)

1. Полуадаптивный код Шеннона-Фано [1,2,4]
2. Полуадаптивный код Хаффмана [1-4]
3. Полуадаптивный код Хаффмана для выходного троичного алфавита [3]
**Символы троичного алфавита называются тритами*
4. Адаптивный код Хаффмана (метод ФГК) [4]
5. Адаптивный код Хаффмана (метод Виттера) [4]
6. Адаптивный код Хаффмана с фиксированным начальным распределением
Начальное распределение должно регулироваться в программе. Оно считается известным обеим сторонам и должно быть заведомо достаточным. Символы не добавляются, нет esc-символа, но дерево перестраивается на каждом шаге. Обновление дерева должно производиться после каждого символа и без полного перестроения дерева (аналогично методам ФГК и Виттера).
7. Полуадаптивный код Танстола [3]

Словарное кодирование (параметры длин сделать настраиваемыми, учесть случай заполнения буфера)

8. LZ-77 [1-4]
9. LZSS [4]
10. LZ-78 [1-4]
11. LZW [1-4]
12. Другой метод кодирования семейства LZ (по согласованию) [4]

Другие методы (по согласованию). Примеры:

1. Полуадаптивное арифметическое кодирование [1,2,4]
*В лекциях рассмотрена необходимая используемая на практике целочисленная реализация метода. Подробно разобрана здесь: [4]:2.14, [5]:5.
При желании можно реализовать адаптивный (динамический) вариант алгоритма, использующий начальное распределение с равными (ненулевыми) начальными весами и увеличивающий веса по мере поступления символов.*
2. Контекстное кодирование
3. Комбинаторное кодирование
4. Статистическое моделирование для арифметического кода

3. Указания по реализации

3.1. О битах

Многие методы способны работать с выходными алфавитами, отличными от двоичных. На практике, при записи информации в память компьютера, сегодня мы пользуемся именно битовым представлением. Однако, когда рассматривается передача символов через канал связи, речь вполне может идти и о передаче недвоичных символов, хотя такой случай редок. В случае с троичным алфавитом нужно только привести длины к битовым длинам при анализе результатов.

3.2. Представление метаданных

Часто, говоря о процессе декодирования, мы сразу начинаем разговор с того, что имеется информационная модель (например, алфавит и таблица вероятностей) или способ кодирования (например, алфавит и кодовое дерево). Однако проблема всех полуадаптивных методов, производящих анализ контента перед кодированием, а не во время кодирования, в том, что эти данные (*метаданные*) каким-то образом нужно передать декодеру. В связи с этим необходимо возникают накладные расходы, снижающие эффективность сжатия. При этом стоит понимать, что и эти данные хранятся/передаются в виде обратимо закодированного массива бит.

В случае методов, использующих в виде моделей деревья, наиболее экономным способом является сжатие именно дерева. По очереди обходятся все вершины дерева и для каждой записывается «0», в случае если это узел ветвления, а если это лист – «1» с последующим 8-битным кодом соответствующего символа. Здесь используется знание о том, что в наших методах дочерних вершин либо нет, либо их заранее оговоренное количество (параметр алгоритма). Стоит напомнить, что существует множество способов обхода вершин дерева. Например, можно выделить *поиск в глубину* (рис. 1) и *поиск в ширину* (рис. 2).

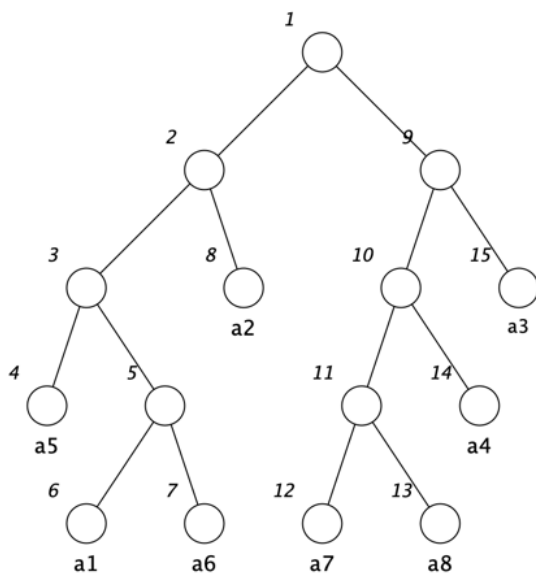


Рис. 1. Поиск в глубину

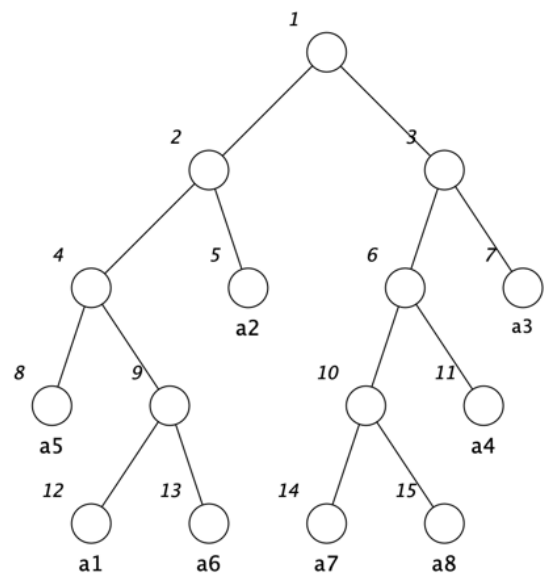


Рис. 2. Поиск в ширину

Коды, соответствующие деревьям выше:

- 1) 0001'A5'01'A1'1'A6'1'A2'0001'A7'1'A8'1'A4'1'A3' (поиск в глубину)
- 2) 00001'A2'01'A3'1'A5'001'A4'1'A1'1'A6'1'A7'1'A8' (поиск в ширину)

Количество символов отдельно задавать не требуется, так как они будут получены из этой записи.

Для кода Танстолла предлагается использовать следующее представление:

- 1) 8 бит, описывающие размер алфавита $K \in \{1, \dots, 256\}$ – имеются в виду реально используемые в сообщении символы;
- 2) исходные 8-битные коды этих K символов;
- 3) структура, аналогичная описанной выше, но уже без указания кодов символов (то есть просто описание структуры дерева).

При этом удобно будет в качестве выходных кодов постоянной длины использовать двоичные представления номеров вершин в порядке обхода, выбранном в п. 3. Число бит в выходном блоке восстанавливается как $\lceil \log_2 K \rceil$.

Для арифметического кодирования предлагается кодировать распределение вероятностей. Рекомендуются после получения распределения в исходном сообщении привести его к виду с фиксированной запятой: распределение умножается на 255 и округляется до ближайших целых (обратите внимание, что после округления сумма может быть не равна 255). Именно такое округлённое распределение нужно использовать при кодировании. В параметрах нужно выписать:

- 1) 8 бит, описывающие размер алфавита $K \in \{1, \dots, 256\}$ – имеются в виду реально используемые в сообщении символы;
- 2) 8-битные коды этих K символов;
- 3) полученные 8-битные веса.

Методы словарного кодирования, как и другие адаптивные коды, не требуют метаданных.

Примечание: полезные функции: `dec2bin(,)` / `bin2dec()`

3.3. О размещении контента в памяти

В будущем мы будем говорить о контейнерах – способах размещения уже сжатых данных внутри файла и передачи потоком. Упрощённо, с точки зрения файловой системы, ваши данные выглядят так:

011011001000101010110111000101010010011000111001...



здесь начинаются ваши данные



здесь кончаются ваши данные

Таким образом, границы файла известны (хотя при этом часто длина файла явно указывается и внутри файла). Чтобы эти данные можно было прочитать, необходимо каким-то образом понять, где в этом потоке лежат какие части данных и с какими параметрами они закодированы. Для этих целей обычно вводятся заголовки. В нашем случае было бы

целесообразно в заголовке прописать эти метаданные, а перед ними – длину самого заголовка.

Интерфейсом хранения у нас будет не файл, а строка. Если файловая система подсказывала нам, какой длины был файл, тут аналогично контейнер Матлаба подсказывает, какой длины строка. Но если требуется в этой строке передать несколько блоков информации (метаданные и основные данные, например), вам нужно самим обеспечить выделение этих частей. Длины, поскольку их немного, можно кодировать кодом заведомо достаточной постоянной длины.

3.4. Реализация деревьев на Matlab

Определённо, Matlab не самый удобный язык для работы с деревьями. Однако это вовсе не делает работу с ними невозможной. Ниже предлагаются лишь некоторые идеи реализации дерева на рис. 1-2 (на случай недвоичных деревьев все реализации легко обобщаются). Оптимальный выбор зависит от решаемой задачи. Выбор наиболее удобной реализации за студентом.

Первый вариант. Пользоваться по максимуму средствами Matlab. В нашем случае, когда вершина либо является родителем для других вершин, либо содержит символ, можно использовать универсальную структуру, способную в себя включать любой тип данных – ячейку ({}). В ячейку-узел будем записывать либо символы (концевой узел), либо массив других ячеек (узел ветвления):

```
tree={{{'a5',{'a1','a6'}},'a2'},{{{ 'a7','a8'},'a4'},'a3'}};
```

Обращение к листу, содержащему 'a7' и замена его на узел ветвления, из которого исходят листы 'a7' и 'a9':

```
tree{2}{1}{1}{1}={'a7','a9'};
```

Второй вариант. Попробовать перенести традиционные для Си структуры. Здесь нет указателей, из-за чего приходится думать, чем их заменять. Как вариант, вместо ссылок на элементы записывать копии элементов (нумерация для удобства, согласно рис. 2):

```
tree.left.left.left.value = 'a5';
tree.left.left.right.left.value = 'a1';
tree.left.left.right.right.value = 'a6';
tree.left.right.value = 'a2';
tree.right.left.left.left.value = 'a7';
tree.right.left.left.right.value = 'a8';
tree.right.left.right.value = 'a4';
tree.right.right.value = 'a3';
```

Обращение к листу, содержащему 'a7' и замена его на узел ветвления, из которого исходят листы 'a7' и 'a9':

```
tree.right.left.left.left= ...
    struct('left',struct('value','a7'), ...
        'right',struct('value','a9'));
```

Либо:

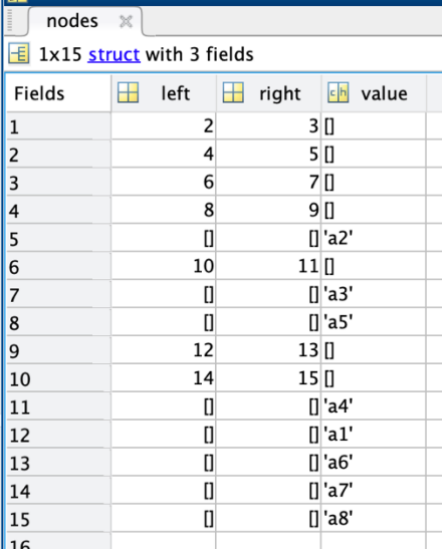
```
tree.right.left.left.left.left.value='a7';
tree.right.left.left.left.right.value='a9';
tree.right.left.left.left = ...
    rmfield(tree.right.left.left.left,'value'); % if necessary
```

Можно вводить отдельные поля для содержимого, а можно, вспомнив, что детей и содержимого одновременно быть не может, в качестве листьев использовать не структуры, а сразу массивы символов (tree.left.left.left = 'a5'). Тогда будет полезна функция isstruct(node).

Можно отличать узлы ветвления от листьев по содержимому какого-то из полей или по наличию каких-то полей: `isfield(node,fieldname)`. Можно вводить поля для определения типа узла.

Третий вариант. Всё-таки не заниматься копированием и попробовать сделать что-то более похожее на указатели. Почувствовать себя в роли `malloc` и выделять для новых узлов место в памяти (в некотором массиве однотиповых структур), а вместо ссылок использовать индексы. Естественно, по той же логике индекс привязывается к вершине, а при обмене меняются ссылки между узлами (в то время как индексы в массиве перестанут совпадать с индексом при обходе в ширину/глубину).

```
nodes(1)=struct('left',2,'right',3,'value',[]);
           % nodes(1).left=2; nodes(1).right=3;
nodes(2)=struct('left',4,'right',5,'value',[]);
nodes(3)=struct('left',6,'right',7,'value',[]);
nodes(4)=struct('left',8,'right',9,'value',[]);
nodes(5).value='a2'; % =struct('left',[],'right',[],'value','a2');
nodes(6)=struct('left',10,'right',11,'value',[]);
nodes(7).value='a3';
nodes(8).value='a5';
nodes(9)=struct('left',12,'right',13,'value',[]);
nodes(10)=struct('left',14,'right',15,'value',[]);
nodes(11).value='a4';
nodes(12).value='a1';
nodes(13).value='a6';
nodes(14).value='a7';
nodes(15).value='a8';
tree=nodes(1);
```



Fields	left	right	value
1	2	3	
2	4	5	
3	6	7	
4	8	9	
5			'a2'
6	10	11	
7			'a3'
8			'a5'
9	12	13	
10	14	15	
11			'a4'
12			'a1'
13			'a6'
14			'a7'
15			'a8'

Рис. 3. Что лежит в nodes

Обращение к листу, содержащему 'a7' и замена его на узел ветвления, из которого исходят листья 'a7' и 'a9':

```
nodes(nodes(nodes(nodes(tree.right).left).left).left) = ...
    struct('left',numel(nodes)+1, ...
          'right',numel(nodes)+2, ...
          'value',[]);
nodes(end+1).value='a7';
```



```
nodes (end+1) .value='a9';
```

3.5. Выполнение операций в дереве

Часто возникает необходимость выполнять операции для каждого узла дерева, при этом может быть важен порядок обхода или путь по дереву от корня до данного узла. Разумеется, на практике не понадобятся такие огромные ручные команды, как выше. Более того, количество команд, которое должно быть выполнено, зависит от входных данных. Можно использовать рекурсию в порядке поиска в глубину:

```
выходные_параметры = операция (узел) {
    если узел концевой
        сформировать выходные_параметры
    иначе {
        выходные_параметры_1 = операция (дочерний_узел_1)
        выходные_параметры_2 = операция (дочерний_узел_2)
        сформировать выходные_параметры по выходные_параметры_1
            и выходные_параметры_2
    }
}
```

При необходимости можно добавить аргумент *входные_параметры*, по ним формировать *входные_параметры_1* и *входные_параметры_2* на вход операций с дочерними узлами и каким-то образом использовать их при формировании *выходные_параметры* в листе.

Литература

1. Дворкович В.П., Дворкович А.В. Цифровые видеоинформационные системы (теория и практика). - М.: Техносфера, 2012. - 1008 с.
2. Семенюк В.В. Экономное кодирование дискретной информации. - СПб.: ИТМО, 2001.
3. Лекции по теории информации [Текст] : учеб. пособие для вузов / Э. М. Габидулин, Н. И. Пилипчук ; М-во образования и науки, Моск. физ.-техн. ин-т (гос. ун-т), Каф. радиотехники. — М. : Изд-во МФТИ, 2007. — 214 с.
4. Salomon D. Data Compression: The Complete Reference. 4th Edition. - Springer, 2006. - 1118 pp.
Переведённое, но сокращённое издание:
Сэломон Д. Сжатие данных изображений и звука. - М.: Техносфера, 2004. - 368 с.
5. Nelson M., Gailly J.-L. The Data Compression Book. 2nd Edition. - M&T Books, N.Y., 1995. — 541 pp.