Алгоритмы сжатия информации

Д. Мастрюков

Часть 4. Алгоритм LZW

Введение

Алгоритмы группы LZ, описанные в предыдущей статье этой серии, имеют существенный недостаток. Так как они используют в качестве словаря только небольшой фрагмент сообщения, то нет возможности кодировать повторяющиеся подстроки, расстояние между которыми в сообщении больше, чем размер словаря. Кроме того, алгоритмы ограничивают размер подстроки, которую можно закодировать. Например, в приведенном в предыдущей статье примере размер подстроки ограничен 17 байтами. Очевидно, что указанные факторы снижают эффективность кодирования.

Чисто механический подход к улучшению характеристик кодера LZ за счет увеличения размеров словаря и буфера обычно дает обратные желаемым результаты. Допустим, что мы увеличим размер словаря до 64 К и размер буфера до 1 К. Это приведет к тому, что для кодирования смещения в словаре понадобится 16 битов вместо 12, а для кодирования длины совпадения будет нужно 10 битов вместо 4. Таким образом каждая фраза будет закодирована в 27 битов вместо 17, что, во-первых, приведет к значительному снижению степени сжатия на коротких сообщениях, а во-вторых, сделает невозможным кодировать повторяющиеся подстроки длиной менее 4 байтов (например «the» в английском языке). Кроме того, увеличение размеров словаря с 4 К до 64 К в алгоритме LZ77 приведет к увеличению в 16 раз времени поиска самого длинного совпадения, а в алгоритме LZSS, где благодаря двоичному дереву поиска время этой операции пропорционально двоичному логарифму размера окна, затраты времени возрастут примерно на 30 процентов. Так как и в LZ77, и в LZSS происходит посимвольное сравнение содержимого буфера с подстрокой в словаре, то при увеличении размеров буфера с 16 до 1024 байтов эта операция будет выполняться в 64 раза дольше. Таким образом, теоретически более мощный LZ кодер становится практически невозможно использовать.

Появление алгоритма LZ78

Поскольку перечисленные проблемы не могли не беспокоить создателей алгоритма Якоба Зива и Абрахама Лемпела, в 1978 году они в работе [2] предложили новую версию своего алгоритма, которая далее будет называться LZ78.

LZ78 уходит от идеи скользящего по тексту окна. В отличие от LZ77, в LZ78 словарем является потенциально бесконечный список уже просмотренных ФРАЗ, а не подстрок, как в LZ77.

В LZ78 и кодер, и декодер начинают работу с «почти пустого» словаря, содержащего только одну закодированную строку — НОЛЬ строку. Когда кодер считывает очередной символ сообщения, символ добавляется к текущей строке. До тех пор пока текущая строка соответствует какой-либо фразе из словаря, процесс продолжается. Но рано или поздно текущая строка

перестает соответствовать какой-либо фразе словаря. В этот момент, когда текущая строка представляет собой «последнее совпадение» со словарем плюс только что считанный символ сообщения, кодер LZ78 выдает код, состоящий из индекса совпадения и следующего за ним символа, нарушившего совпадение строк. Кроме того, новая фраза, состоящая из индекса совпадения и следующего за ним символа, добавляется в словарь. В следующий раз, когда эта фраза появится в сообщении, она может быть использована для построения более длинной фразы, что повышает степень сжатия информации.

Алгоритм LZW

Как и LZ77, алгоритм LZ78 был опубликован в сугубо научном журнале и сначала воспринимался читателями скорее как абстракция, нежели то, что можно положить в основу программного продукта. Так было до 1984 года, когда Терри Уэлч (Terry A. Welch) опубликовал свою работу [3]. Модификация алгоритма LZ78, предложенная Уэлчем, получила название LZW (Lempel-Ziv-Welch).

> Алгоритм LZW построен вокруг словаря фраз, который отображает строки символов сжимаемого сообщения в коды фиксированной длины.

LZW отличается от LZ78 примерно тем же, чем LZSS отличается от LZ77. Кодер LZW никогда не выдаст сами символы сжимаемого сообщения, только коды фраз.

LZW построен вокруг таблицы фраз (словаря), которая отображает строки символов сжимаемого сообщения в коды фиксированной длины (в данном случае 12-битовые). Таблица обладает так называемым свойством предшествования, то есть для каждой фразы словаря, состоящей из некоторой фразы w и символа К, фраза w тоже содержится в словаре.

Кодер LZW

Алгоритм работы кодера LZW можно описать следующим образом:

```
Проинициализировать словарь односимвольными
фразами, соответствующими символам входного
алфавита (обычно это 256 ASCII символов)

Прочитать первый символ сообщения в текущую фразу
w;

Шагалгоритма:
    Прочитать очередной символ сообщения К;
    Если КОНЕЦ_СООБЩЕНИЯ
        Выдать код w;
        ВЫХОД;
    Конец Если
    Если фраза wK  уже есть в словаре,
        Заменить w на код фразы wK;
        Повторить Шаг алгоритма;
    Иначе
        Выдать код w;
        Добавить wK в словарь;
        Повторить Шаг алгоритма;
    Конец Если;
```

Пример работы кодера LZW приведен в таблице 1.

Описанный алгоритм не пытается оптимально выбирать фразы для добавления в словарь или оптимально разбирать сообщение. Однако в силу его простоты он может быть эффективно реализован. Способ организации словаря («массив структур») хорошо подходит к хешированным методам доступа, что также повышает быстродействие кодера.

Декодер LZW

Очевидно, что декодер LZW использует тот же словарь, что и кодер, строя его по аналогичным правилам при восстановлении сжитого сообщения. Каждый считываемый код разбивается с помощью словаря на предшествующую фразу w и символ К. Затем рекурсия продолжается для предшествующей фразы w до тех пор, пока она не окажется кодом одного символа, что и завершает декомпрессию этого кода. Обновление словаря происходит для каждого декодируемого кода, кроме первого. После завершения декодирования кода его последний символ, соединенный с предыдущей фразой, добавляется в словарь. Новая фраза получает то же значение кода (позицию в словаре), что присвоил ей кодер. Так шаг за шагом декодер восстанавливает тот словарь, который построил кодер.

Алгоритм декодирования LZW может быть описан следующим образом:

```
КОД = Прочитать первый код сообщения();
ПредыдущийКОД = КОД;
Выдать символ К, у которого код(К)  == КОД;
Следующий код:
    КОД = Прочитать очередной код сообщения();
    ВходнойКОД = КОД;
    Если КОНЕЦ_СООБЩЕНИЯ
        ВЫХОД;
    Конец Если;
Следующий символ:
    Если КОД == КОД(wK)
        Выдать К;
        КОД = код(w);
```

```
        Повторить Следующий символ;
    Иначе если КОД == код(К)
        Выдать К;
        Добавить в словарь (ПредыдущийКОД, К);
        ПредыдущийКОД = ВходнойКОД;
        Повторить СледующийКОД;
    Конец Если;
```

У этого алгоритма есть два существенных недо-

Таблица 1

Шаги работы кодера LZW на примере трехсимвольного алфавита (а, б, в)

Символ	wK	Выход	Добавление в словарь (фраза - позиция)
а	1		
б	б	1	1б (4)
а	2а	2	2а (5)
б	1б		
в	4в	4	4в (6)
б	3б	3	3б (7)
а	2а		
б	5б	5	5б (8)
а	2а		
б	5б		
а	8а	8	8а (9)
а	1а	1	1а (10)
а	1а		
а	10а	10	10а (11)
а	1а		
а	10а		
а	11а	11	11а (12)
		1	

Таблица 2

Словарь, построенный кодером LZW для примера из таблицы 1

Фразы, добавленные в словарь при инициализации	
а	1
б	2
в	3
Фразы, добавленные при разборе сообщения	
1б	4
2а	5
4в	6
3б	7
5б	8
8а	9
1а	10
10а	11
11а	12

статка. Во-первых, он выдает символы в обратном порядке, во-вторых, он не будет работать в исключительной ситуации.

Изменить порядок выдачи раскодированных символов несложно, для этого достаточно использовать стандартный LIFO стек.

Обработка исключительной ситуации несколько сложнее. Исключительная ситуация складывается тогда, когда кодер пытается закодировать сообщение KwKwK, где фраза Kw уже присутствует в словаре. Кодер выделит Kw, выдаст код(Kw) и добавит KwK в словарь. Затем он выделит KwK и пошлет только что созданный код (KwK). Декодер при получении кода(KwK) еще не добавил этот код в словарь, потому что еще не знает символ-расширение предыдущей фразы. Тем не менее, когда декодер встречает неизвестный ему код, он может определить, какой символ выдавать первым. Это символ-расширение предыдущей фразы, который будет последним символом текущей фразы, который был последним символом предыдущей фразы, который был последним раскодированным символом.

Модифицированный алгоритм декодирования выглядит следующим образом:

```
КОД = Прочитать первый код сообщения();
ПредыдущийКОД = КОД;
Выдать символ К, у которого код(К) == КОД;
ПоследнийСимвол = К
Следующий код:
    КОД = Прочитать очередной код сообщения();
     ВходнойКОД = КОД;
     Если КОНЕЦ_СООБЩЕНИЯ
         ВЫХОД;
     Конец Если;
     Если Неизвестен(КОД) // Обработка
                              исключительной ситуации
         Выдать(ПоследнийСимвол)
         КОД = ПредыдущийКОД
         ВходнойКОД = код (ПредыдущийКОД,
                              ПоследнийСимвол)
     Конец Если;
Следующий символ:
    Если КОД == код(wK)
         В_СТЕК ( К );
         КОД = код(w);
         Повторить Следующий символ;
     Иначе если КОД == код (К)
         Выдать К;
         ПоследнийСимвол = К;
         Пока стек не пуст
             Выдать ( ИЗ_СТЕКА() );
         Конец пока;
         Добавить в словарь (Предыдущий КОД, К);
         ПредыдущийКОД = ВходнойКОД;
         Повторить СледующийКОД;
     Конец Если;
```

В алгоритме декодирования LZW нет необходимости в хешировании, так как обращение к словарю происходит непосредственно по коду фразы (то есть номеру в словаре). Обычно декодирование LZW намного быстрее кодирования.

Вопросы реализации

При попытке запрограммировать алгоритм LZW (листинг) необходимо решить целый ряд проблем. Наиболее простая из них — выбрать размер кодов алгоритма в битах так, чтобы добиться компромисса между эффективностью сжатия и затратами оперативной памяти на поддержание словаря. В этой реализации размер кода алгоритма равен 12 битам, что, по мнению автора, дает неплохое сжатие и позволяет обойтись примерно 25 К оперативной памяти на поддержание словаря. Следующий вопрос — как быть, когда словарь полностью заполнен, а обработка сообщения еще не завершена. В данной реализации, чтобы не загромождать пример программы, словарь в этот момент «замораживается», то есть кодирование перестает быть адаптивным. В принципе возможно поддерживать более гибкую схему работы со словарем.

Наиболее критичный для производительности программы вопрос — это как быстро определить, есть ли в словаре фраза wK, и если есть, то где. В данной реализации для этого используется хешированньтй доступ к словарю. Хеш-функция аналогична используемой в известной программе COMPRESS для ОС UNIX. Она формирует 12-битовое хеш-значение из 12-битового значения кода и 8-битового символа (пара wK). Функция поиска, основанная на этом хеш-функции, либо находит индекс заданной пары в словаре, либо индекс незанятого элемента словаря, если такой пары в словаре нет. Механизм борьбы с коллизиями при хеш-доступе требует, чтобы размер словаря (в его элементах) был простым числом, большим, чем количество возможных элементов словаря. В данном примере при размере кода 12 битов (4096 возможных элементов словаря) размер словаря равен 5021 элементу. Эксперименты показывают, что при такой комбинации параметров кодера поиск соответствия со словарем происходит в среднем за 2-3 операции сравнения пары wK и элемента словаря. Согласитесь, что это намного лучше, чем поиск полным перебором. Как уже отмечалось, декодеру не нужно пользоваться хеш-доступом к словарю, так как он обращается к его содержимому по номеру элемента.

Осторожно: патенты!

Автор LZW Терри Уэлч в свое время сумел запатентовать свой алгоритм в США. В настоящее время патент принадлежит компании Unisys, которая открыто заявляет о своем намерении защищать свои права на интеллектуальную собственность. Алгоритм LZW определяется как часть телекоммуникационного стандарта V. 42bis, и Unisys зафиксировала жесткие условия лицензирования алгоритма для производителей модемного оборудования. Однако не ясно, касаются ли эти ограничения производителей других типов продуктов. В силу таких неопределенностей, чтобы избежать судебных разбирательств, фирмы-разработчики программного обеспечения стараются не использовать LZW в своих коммерческих продуктах. •

Рекомендуемая литература

1. Ziv J., Lempel A. An Universal Algorithm for Sequential Data Compression// IEEE Transactions on Information Theory. - 1977. -Vol. 23. -№ 3.
2. Ziv J., Lempel A. Compression of Individual Sequences via Variable Rate Coding// IEEE Transactions on Information Theory. - 1978. -Vol. 23. -№3.
3. Welch T. A. A Technique for High-Performance Data Compression// Computer. -1984. -Vol. 17. -№ 6.
4. Storer J. A. Data Compression: Methods and Theory, - USA: Computer Science Press, 1988.
5. Nelson M. The Data Compression Book. -USA: M&T Publishing, 1991.
6. Арапов Д. Пишем упаковщик// «Монитор». -1993. - №1.

В следующем номере

Такие программные продукты, как PKZIP, LHA, ARJ и так далее, достаточно хорошо известны. Но в последнее время все большее распространение получают так называемые дисковые компрессоры - драйверы устройств, сжимающие и разжимающие информацию при доступе к диску «на лету». В силу того, что драйвер в значительно большей степени, чем обычный архиватор, ограничен по памяти и быстродействию, алгоритмы сжатия, используемые такими программами, отличаются от алгоритмов типовых архиваторов. В следующем номере «Монитора» вниманию читателей будет предложен алгоритм, разработанный автором этой статьи, который он использовал в пакете «Супердиск».