Пишем упаковщик

Д. Арапов

Введение

Большинству программистов известны программы-архиваторы: PKPAK, PKARC, PKZIP, LHARC, ARJ и т.п. Они сжимают хранящуюся в файлах информацию и помещают ее в файл-архив, где она хранится до тех пор, пока не понадобится. Восстановить первоначальный вид данных можно, воспользовавшись процедурой распаковки. Изучению сравнительных достоинств архиваторов посвящено множество статей. Напомним, что для оценки архиваторов обычно используют два критерия: коэффициент сжатия и скорости упаковки и распаковки. Эти характеристики обычно зависят от характера информации: исполняемые программы, графика, текст, записи базы данных и так далее.

Цель данной статьи - познакомить читателя с идеями, положенными в основу современных упаковщиков, и предложить программу компрессии-декомпрессии, представляющую собой вариацию на тему алгоритма Лемпела-Зива. Этот алгоритм позволяет паковать данные любых типов, однако прежде всего он ориентирован на сжатие текстовой информации. Приведенный модуль читатель может поместить в собственный набор инструментов и использовать в своих программах (скажем, в инсталляторах).

Выделяются два типа организации алгоритмов упаковки: одно- и двухпроходные. И те и другие используют словарь, в котором накапливают статистику о повторяющихся символах и подстроках. В случае двухпроходной организации на первом проходе строится словарь, а на втором (при помощи этого словаря) производится упаковка. Как правило, двухпроходные алгоритмы позволяют достичь чуть более высокой степени сжатия, однако организация дополнительного прохода требует больше времени, кроме того, чтобы обеспечить распаковку, приходится запоминать вместе с текстом и словарь.

Алгоритм Лемпела-Зива

Алгоритм Лемпела-Зива, которому посвящена статья, имеет однопроходную организацию. Как в процессе упаковки, так и при распаковке создается словарь, так что сохранять его вместе с текстом не нужно. В словарь заносятся подстроки исходного текста. При упаковке они заменяются номерами, под которыми они найдены в словаре. Опуская детали, алгоритм сжатия можно представить на псевдокоде таким образом:

```
Инициализировать словарь.
Установить текущую позицию в начало
текста.
ПОКА не весь текст запакован:
    Выделить наиболее длинную, начинающуюся в текущей позиции подстроку s, для
    которой есть соответствие в словаре.
    Записать в архив номер позиции P найденной подстроки в словаре. Посмотреть
    следующий за подстрокой в тексте символ K.
    Занести в словарь строку sK. Продвинуть
    текущую позицию, чтобы она указывала на
    символ K.
КОНЕЦ ПОКА
```

Алгоритм распаковки выглядит так:

```
Инициализировать словарь.
Установить позицию на начало архива.
ПОКА не весь архив распакован:
 Прочесть из архива очередное число P. Найти
 строку s, лежащую на позиции P в словаре.
 Записать s в конец распакованного текста.
 Прочесть из архива следующее за P число Q и
 найти в словаре строку u, находящуюся на
 месте Q. Записать в словарь su[1].
 Продвинуть текущую позицию на символ Q.
КОНЕЦ ПОКА
```

Принцип одинакового изменения словаря

Основной принцип, на основе которого построены все рассматриваемые вариации алгоритма, состоит в том, что упаковщик и распаковщик изменяют словарь одинаковым образом. Благодаря этому получаемая при декомпрессии из словаря по номеру n строка совпадает со строкой, которую упаковщик зашифровал тем же числом n.

Отметим, что приведенный выше алгоритм не вполне точен. Проблема состоит в том, что если упаковщик может немедленно занести расширенную строку в словарь, то при распаковке этого сделать нельзя, необходимо прочесть следующую строку. Поэтому для обеспечения одинаковой последовательности манипуляций со словарем при упаковке расширенную строку помещают для начала во вспомогательный буфер, а в словарь заносят только на следующем витке цикла.

Строки вида KsKsK

В большинстве случаев этот прием никак не влияет на качество сжатия. Однако это не так, если в пакуемом тексте, начиная с текущей позиции, находится строка вида KsKsK, где K - некоторый символ, а s -подстрока (возможно пустая). Пусть подстрока Ks ранее была занесена в словарь и ей был присвоен некоторый номер x, в то же время строка KsK в словаре отсутствует. Если следовать исправленному алгоритму, то строка KsKsK будет кодироваться как xxy, где y - код подстроки, начинающейся с третьего вхождения символа K. Модифицируем алгоритм, введя дополнительную проверку, не находится ли подстрока в буфере. Подстроку KsK будем кодировать особым числом z, не являющимся правильным номером словаря. Это позволит закодировать исходную строку KsKsK в виде xz. При декомпрессии, обнаружив номер z, мы не должны обращаться к словарю. Вместо этого нам следует воспользоваться полученной на предыдущем витке цикла строкой.

Замечание. Вся приведенная выше "кухня" не заслуживала бы внимания, если бы не весьма распространенные строки вида 'K...KKK'. На них использование приведенного выше приема позволяет увеличить плотность кодирования до 2 раз.

Кодирование позиций

До сих пор мы не говорили о том, в каком виде номера входов словаря записываются в сжатом представлении (архиве). Если число входов в словаре меньше 2**n, то каждый вход можно закодировать в виде цепочки из n битов. Обратите внимание, что неравенство строгое, поскольку одну комбинацию битов нужно зарезервировать для описанного выше случая. Обычно используются словари размером от 2**10-1 до 2**14-1 входов. Однако тривиальная кодировка позиций не самая экономная. Гораздо лучше кодировать, используя цепочки битов переменной длины: короткие для часто встречающихся номеров -позиций, а длинные для редко встречающихся. При этом, однако, потребуется по крайней мере один до-полнительный бит для кодирования длины цепочки.

Организация словаря

Словарь должен позволять искать позицию по строке (при упаковке) и строку по позиции (для распаковки). Кроме этого, словарь должен быть организован так, чтобы позиция для наиболее употребимых в последнее время строк кодировалась с помощью меньшего числа знаков. Рассмотрим организацию словаря в виде приоритетного списка: в головной части (с номерами 1, 2 и так далее) будут помещаться наиболее употребительные подстроки, а относительно редкие строки попадут в хвост списка (tail). При этом для изменения словаря будут использоваться 3 операции:

Add (включить)	- добавляет в словарь новую строку;
Delete (исключить)	- удаляет из словаря строку;
Promote (продвинуть)	- перемещает ранее помещенную в сло варь строку ближе к началу списка.

Две операции - Add и Promote - вызываются непосредственно из процедур упаковки (распаковки), а Delete - изнутри операции Add в том случае, когда словарь уже целиком заполнен: она выбирает и удаляет из словаря наименее полезную, с ее точки зрения, строку. При этом следует соблюдать 2 принципа:

1. Нельзя исключать из словаря односимвольные строки.

2. Нельзя исключать из словаря строку, которая является префиксом другой строки словаря или добавляемой строки.

Нарушение первого принципа (инварианта) может остановить процесс сжатия: алгоритм Лемпела-Зива основав на том, что, пока текст полностью не упакован, мы можем отщепить от него непустую строку, • которая уже присутствует в словаре. Для этого там должны быть как минимум все односимвольные строки. Нарушение второго принципа приведет к тому, что в словаре появятся недостижимые строки.

В рассматриваемой программе мы будем исключать первый с конца элемент словаря, не попадающий под перечисленные запреты.

Адаптивность

Операции Add и Promote могут быть закодированы по-разному. От выбранного способа зависит степень

адаптивности кодирования. Дело в том, что сжимаемые файлы могут быть неоднородны. Например, текст этой статьи практически не содержит строк из латинских букв, а включенный в нее пример программы -знаков кириллицы. Способность словаря быстро подстраиваться к изменениям частотных характеристик входного текста называется адаптивностью.

Если нужно внести в словарь новую строку, то ее можно добавить в хвост списка. Однако после того как словарь заполнен этот подход практически не позволяет внести в него новую строку, поскольку с очень большой вероятностью при вставке следующего элемента эта строка будет удалена, ведь она находится в самом конце списка. Таким образом в словаре возникнут "застойные явления" и он не будет адаптироваться к изменениям во входном тексте, в свою очередь, это приведет к тому, что соответствующая программа сжатия будет плохо работать на длинных и неоднородных текстах.

Другой крайностью будет включение нового элемента в начало списка. Безусловно, этот способ очень адаптивен, однако в начало списка очень часто будут попадать случайные строки, оттесняя с выгодных позиций в середину списка часто повторяющиеся элементы. В результате последние будут кодироваться при помощи более длинных последовательностей битов и общее качество упаковки снизится.

Аналогичные рассуждения применимы и к операции Promote. Новые элементы предлагается включать в позицию Count div m, где Count - число элементов в списке, операция Promote будет каждый раз передвигать элемент из позиции n в позицию n div m. В результате экспериментов с различными текстами было обнаружено, что лучшие результаты обычно достигаются при m = 3.

Поиск в словаре

Исторически ахиллесовой пятой упаковщиков была низкая скорость поиска строки в словаре. Кроме этого, манипуляция со строками переменной длины обычно подразумевает использование динамически распределяемой области памяти (все равно, стандартной или реализованной самостоятельно) и, соответственно, операций new и dispose, требующих сравнительно много времени. Во многих упаковщиках последняя проблема решена довольно просто: фрагмент пакуемого текста (длиной обычно 4 или 8К), непосредственно предшествующий текущей позиции компрессора (декомпрессора) хранится в циклическом буфере. Есть и другой способ. Хранения строк переменной длины можно избежать, если учесть следующие особенности.

1. Если в словаре имеется некоторая строка, то он должен содержать и все ее префиксы. Как следствие, каждую строку длины l можно представить как пару из последнего символа и ссылки на префикс длиной l - 1.
2. Если на некотором шаге алгоритма упаковки производится поиск строки sK, то на предыдущем шаге был найден ее максимальный префикс - подстрока s.

В нашей программе для обеспечения приемлемой скорости используется структура данных, состоящая из записей фиксированной длины. При этом каждая запись содержит символьное поле и 4 ссылки на другие записи. Все записи помещены в массив, а ссылки реализуются с помощью индексов массива. Кроме этого, допустимым значением ссылки будет специальное значение NullLink (ссылка "в никуда"). В целом построенная структура данных будет иметь вид дерева.

```
type TNode  =  record
                  Up, Down,
                  Left, Right   : TLink;
                  Symbol    : Char end;
```

Рассмотрим назначение полей. Ссылка Up указывает на узел, содержащий предыдущую литеру кодируемой строки (то есть на максимальный префикс строки) . Таким образом записи типа TNode (узел) взаимно однозначно соответствует строка словаря, и мы можем далее не различать их. Для односимвольных строк ссылка Up будет указывать "в никуда". Left и Right ссылаются на узлы, представляющие строки той же длины, которые отличаются от данной только последним символом. Эти поля служат для организации бинарного поиска среди строк с одним и тем же максимальным префиксом. Ссылка Down будет указывать на одну из строк, для которых данная будет максимальным префиксом (рис. 1).

		Up	Down	Left	Right	Sym
1	a	2	5	0	0	a
2	b	0	0	1	3	b
3	c	2	0	0	4	c
4	d	3	0	0	0	d
5	ab	1	7	0	6	b
6	ad	5	0	0	0	d
7	abb	5	0	9	8	b
8	abc	7	0	0	0	c
9	aba	7	0	0	0	a

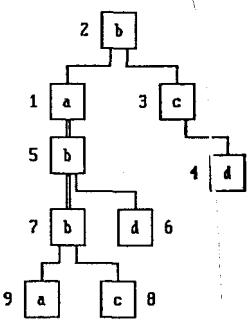

Рис. 1. Представление строк в словаре. Двойной линией обозначены ссылки Down

К сожалению, операции Add, Delete и Promote не ограничиваются изменением позиции одного элемента. Входы словаря постоянно меняют свои места. Корректировка ссылок может потребовать очень много времени. Для решения этой проблемы можно воспользоваться двумя способами.

1. Использовав представление словаря в виде двух массивов: массива записей типа Node (узлов дерева), в котором каждая запись от момента создания до мо-

мента уничтожения не меняет своего места, и массива ссылок на элементы этого массива. Операции Add, Delete и Promote изменяют взаимное расположение ссылок на элементы словаря, а не расположение самих элементов.

Дотошный читатель заметит подвох и спросит: "А как организован поиск позиции ссылки на строку в приоритетном списке?" Увы, линейным просмотром. Однако на IBM PC этот просмотр можно сделать весьма быстрым, запрограммировав его на ассемблере и воспользовавшись командой REP SCASW (поиск слова в массиве). К нашей выгоде действуют также два обстоятельства: во-первых, искомое значение заведомо присутствует в массиве, а во-вторых, в силу приоритетности списка, расположено скорее всего ближе к началу списка (рис. 2).

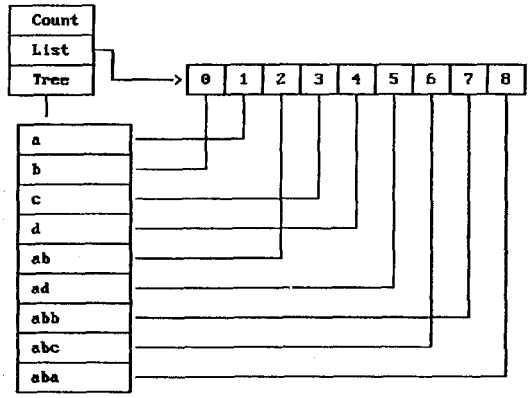

Рис. 2. Словарь представлен в виде массива Tree узлов дерева, на которые идут ссылки из приоритетного списка List

2. Имеется и другой более быстрый (по крайней мере, теоретически) алгоритм: заранее разделить список на две части (в принципе, можно и на большее число частей, но выигрыша в плотности упаковки это практически не дает): на маленькую и большую. И та и другая представляют собой двунаправленные списки элементов массива. Новые узлы включаются с головы списка в большую часть. Если какой-либо элемент из большой части встречается чаще, чем самый долго не встречавшийся из малой, то они обмениваются местами. Только в случае таких обменов приходится перестраивать ссылки Up, Down, Left и Right (рис. 3).

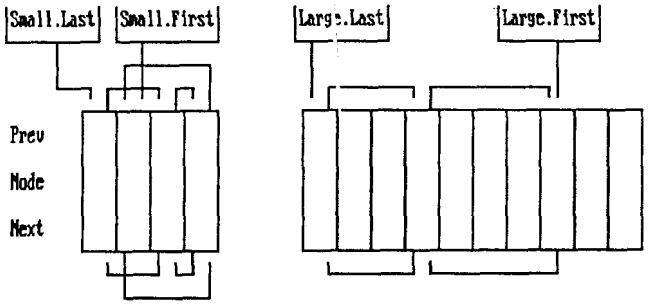

Рис.3. Представление словаря в виде двух частей: малой (Small) и большой (Large). Обратите внимание, что в части Large не все элементы включены в двунаправленный список

В приводимой программе малая часть состоит из 127 входов, а большая из 3966. Каждый элемент из малой части может быть представлен 7 битами, а из большой - 12, кроме этого, требуется по одному биту для того чтобы различать коды.

Мы уже заметили, что для каждой строки, которую мы ищем в словаре при упаковке, ее префикс уже найден в словаре. Благодаря рассмотренной организации словаря время поиска строки не зависит от количества входов в словарь.

Инициализация словаря

Первоначально в словарь заносятся 256 различных односимвольных строк. Если известно, что программа будет работать с особыми текстами (например, с програм-мами на Паскале), то при инициализации словаря имеет смысл занести характерные для таких текстов строки, разумеется, со всеми их префиксами. Не забудьте, что инициализация словаря для программы сжатия и для распаковки должна производиться одинаково.

Конфликты

Алгоритм LZ всегда отщепляет от пакуемого текста начальную подстроку максимальной длины. В большинстве случаев это наиболее выгодная стратегия. Однако часто встречаются ситуации, когда, отщепив вначале более короткий префикс, можно выиграть. Это происходит в тех случаях, когда строка, идущая вслед за коротким префиксом, значительно длиннее, чем строка, начинающаяся вслед за префиксом максимальной длины, и сумма длин двух последовательных подстрок больше, чем сумма длин двух последовательных подстрок, выделяемых алгоритмом LZ. Назовем такую ситуацию конфликтом. Приведем практический пример возникновения конфликта: пакуется текст программы на Паскале. В словаре имеются все 256 символов, а также строки: 'p', 'pr', 'pro', 'proc'. Начиная с текущей позиции, текст имеет вид ' proc'. Если отщепить префикс максимальной длины (' p'), то этот фрагмент можно будет закодировать четырьмя входами словаря. Если же наш алгоритм умеет заглядывать вперед, то тот же фрагмент можно представить в виде всего двух строк словаря: ' ' + 'proc'.

Алгоритм с разрешением конфликтов выбирает тот из префиксов, для которого сумма его длины и длины самой большой начинающейся после него строки максимальна. Его следует рекомендовать только в том случае, когда плотность упаковки имеет решающее значение. Использование его замедляет работу в несколько раз (коэффициент замедления приблизительно равен отношению плотностей упакованного и исходного текстов, то есть 2-3).

Короткие тексты и энергичное пополнение словаря

Алгоритм Лемпела-Зива предназначен для упаковки сравнительно больших текстов, объем которых в

Символах существенно больше числа входов словаря. Маленькие тексты он пакует хуже. Это связано с тем, что в начале работы словарь практически пуст. Кодирование одной строки расширяет словарь на 1 элемент. Если программу упаковки предполагается применять для работы с относительно короткими текстами, скажем, в информационно-поисковой системе, то имеет смысл более активно пополнять словарь. Предлагается добавлять в словарь строки вида s + t, где t -следующая подстрока текста, найденная в словаре. Разумеется, чтобы избежать противоречия принципу доступности, в словарь следует занести и все префиксы, которые длиннее s. Основным преимуществом будет быстрое попадание в словарь длинных, но часто повторяющихся подстрок.

Так, при кодировании программы на Паскале алгоритму Лемпела-Зива потребуется минимум 8 раз встретить слово 'function', чтобы закодировать его при помощи одного числа. Предложенный алгоритм способен сделать это уже с 4 попытки.

Недостаток этого подхода очевиден: быстрое разбухание словаря. У этого разбухания имеется 2 негативных последствия: во-первых, могут быть вытолкнуты более короткие и полезные подстроки, во-вторых, потребуется больше двоичных знаков для кодирования входов. Когда речь идет о коротких текстах, первым можно пренебречь: количество строк, попавших в словарь, не превысит числа символов в исходном тексте. Второе последствие также не столь существенно при использовании кодов переменной длины: активные входы будут по-прежнему заменяться короткими цепочками битов, а длина кодов для строк, которые были занесены в словарь, но больше не встретились, на объем упакованного текста не влияет.

Заключение

Представленная программа позволяет упаковывать текст приблизительно так, как это делали архиваторы первого поколения: PKPAK, ранние версии программы ZIP. Она может оказаться полезной для тех, кто по каким-либо причинам не пользуется существующими архиваторами, скажем, при создании какой-нибудь своей программы. •

Литература

1. Lempel A., Ziv J. A Universal Algorithm for Sequential Data Compression// IEEE Trans. Info. Theory.-1977.-N 23.-P.337-343.
2. Lempel A., Ziv J. Compression of Individual Sequences Via Variable-Rate Coding// IEEE Trans. Info. Theory. -1978.-N24.-P.530-536.
3. Welsh T.A. A Technique for High-Performance Data Compression// Computer.-1984.- N 17.- P.8-19.
4. Chang D.K. Data Compression Using Hierarchical Dictionaries// The Journal of Systems and Software.-1991.-Vol. 15.-N 3. - P. 233-238.