An Approach to Phrase Selection for Offline Data Compression

A. Turpin **W. F. Smyth**

School of Computing
Curtin University of Technology
GPO Box U1987, Perth, Western Australia, 6845.
Email: {andrew,smyth}@cs.curtin.edu.au

Abstract

Recently several *offline* data compression schemes have been published that expend large amounts of computing resources when encoding a file, but decode the file quickly. These compressors work by identifying phrases in the input data, and storing the data as a series of pointer to these phrases. This paper explores the application of an algorithm for computing all repeating substrings within a string for phrase selection in an offline data compressor. Using our approach, we obtain compression similar to that of the best known offline compressors on genetic data, but poor results on general text. It seems, however, that an alternate approach based on selecting repeating substrings is feasible. *Keywords:* strings, offline data compression, textual substitution, repeating substrings

1 Introduction

If data is to be stored on CD, DVD or in a static database it will be compressed once, but often decompressed many times. Given this scenario, a compression scheme can afford to spend several hours of computing time, make multiple passes over the input, and consume many megabytes of RAM during the compression process in order to make the compressed representation as small as possible. Decompression, however, should be both fast and memory efficient. Such a compression scheme is said to be *offline*.

One way to meet the demand for fast decoding and high compression levels is to identify a suitable *phrase book* such that the input data can be stored as a series of pointers to entries in the phrase book. For example, Figure 1 shows how the simple string "How much wood can a woodchuck chuck if a woodchuck could chuck wood?" is compressed using three different phrase books. The first representation favours phrases that appear frequently in the string, hence the space character forms a phrase by itself. The second representation looks to include the space character at the start or end of a word to form a phrase. The third greedily chooses the longest repeating phrase that it can, which is similar to the strategy employed in compressors based on the LZ77 schemes [Ziv & Lempel, 1977], such as *gzip*, *winzip*, and *pkzip*.

In the final file both the phrase book and the series of pointers must be stored. It is difficult to tell by inspection of our example which of these three phrase books will yield the best compression. The phrase book in representation 1 contains only 27 characters, but has 26 pointers. The phrase books in representation two and three have more characters in their phrase books, but significantly less pointers. Unfortunately the variables involved in choosing a phrase book are much more complicated than merely the number of pointers and number of characters in the phrase book. Assuming that some sort of statistical coder (for example, Huffman coding or arithmetic coding) will be used to actually encode the pointers and the phrase book, the frequency distribution or self entropy of the two components are better indicators of the fitness of a phrase book. In this particular example, the cost of a zero-order Huffman code on characters in the phrase book and pointers in the data portion, shown in the last row of the Figure 1, indicates that the first phrase book leads to the smallest representation of 21 bytes. Even these calculations are only an approximation of the final compression levels obtained with a code of this nature as necessary information that describes the Huffman codes employed (a *prelude*) is not included in these estimates.

Offline compression through the use of a phrase book is not a new idea [Rubin, 1976, Storer & Szymanski, 1982, Nevill-Manning & Witten, 1994], but with the increased availability of cheap, powerful computers, computationally intensive techniques are now viable during encoding in order to improve compression levels through the construction of good phrase books. The task of identifying the best possible phrase book on any input has been shown to be NP-complete [Storer & Szymanski, 1982], but using heuristics and a lot of machine power, compression levels superior to alternate techniques have been achieved on some data sets.

Nevill-Manning & Witten introduced an approach that induces a context free grammar from the text, using their grammar rules to describe the phrase book for compression [Nevill-Manning & Witten, 1994]. Both Larsson & Moffat and Cannane & Williams explore the use of repeated pairing of characters in order to build a phrase book, with the emphasis on small and large data sets respectively [Larsson & Moffat, 2000, Cannane & Williams, 2001]. Bentley & McIlroy describe an efficient algorithm for finding long repeating substrings to place in the phrase book [Bentley & McIlroy, 1999].

This paper expands on work of Apostolico & Lonardi. Recently they introduced the OFFLINE compressor [Apostolico & Lonardi, 2000], which calculates a measure of compression *gain* for all possible non-overlapping substrings of a string. A high gain factor indicates that if the substring was to be chosen as a phrase for the phrase book, good compression would result. Similarly, a low gain score for a substring indicates that the particular substring should not be chosen as a phrase in the phrase book. The compression algorithm used in OFFLINE is outlined in Figure 2.

Representation 1		Representation 2		Representation 3	
How much	1	How much	1	How much	1
	2	wood	2	wood	2
wood	3	could	3	c	3
	2	a woodchuck	4	ould	4
could	4	chuck	5	a	5
	2	if	6	wood	2
a	5	a woodchuck	4	chuck	6
	2	could	3	c	3
wood	3	chuck	5	huck	7
chuck	6	wood	2	if	8
	2	?	7	a	5
chuck	6			wood	2
	2			chuck	6
if	7			c	3
	2			ould	4
a	5			c	3
	2			huck	7
wood	3			wood	2
chuck	6			?	9
	2				
could	4				
	2				
chuck	6				
	2				
wood	3				
?	8				
Phrases	12		18		16
Pointers	9		4		8
Total	21		22		24

Figure 1: The string "How much wood could a woodchuck chuck if a woodchuck could chuck wood?" represented with three possible phrase books. The first occurrence of each phrase is shown in gray for each case, and are numbered in order of first occurrence. The final three rows show the cost of Huffman encoding the phrase book and pointers in bytes.

As alluded to in the above example, calculating the exact gain in compression for any given substring is a difficult task. At the commencement of encoding there is no way of knowing how many phrases will end up in the phrase book, or what the probability distributions of characters in the phrase book or pointers in the data component will be. Accordingly, Apostolico and Lonardi experimented with three approximate gain formulations. Using this simple approach they achieve excellent levels of compression on some genetic sequences, and competitive compression levels on general data [Apostolico & Lonardi, 2000]. Details of their results can be found at `www.cs.purdue.edu/homes/stelo/Off-line/`. The implementation of Offline relies on a suffix tree data structure, which is a trie that holds all possible suffixes of a string [Ukkonen, 1995, and references therein].

As they acknowledge, however, a suffix tree is a large and slow data structure for this task. In this paper we introduce an alternate approach for performing compression using the Offline algorithm, based on a string processing algorithm for finding all repeating substrings in a string. By focusing only on the repeating substrings, rather than all suffixes of a string, we hypothesise that the time taken to perform gain calculations and string manipulations using the Offline approach can be significantly reduced.

Section 2 describes Crochemore's algorithm [Crochemore, 1981] for finding repeating substrings within a string, and explains how we use it to select phrases in our offline compression scheme, Crush. Section 3 describes experimental results for both compression levels and timing for Crush and Offline. Finally, Section 4 discusses our results, and their implications.

INPUT	String to compress.
Step 1	Calculate the gain for all possible non-overlapping substrings of the input string to be compressed.
Step 2	Choose the substring with the highest gain factor, and add it to the phrase book.
Step 3	Remove all occurrences of the chosen substring from the string, and store a pointer to the original phrase for each occurrence.
Step 4	Recalculate the gain measure for all substrings of the input string that have not been covered by a chosen phrase.
Step 5	While there is still a positive gain factor, repeat from Step 2 on the remaining uncovered string.
OUTPUT	Phrase book and list of pointers representing the input string.

Figure 2: The basic algorithm employed in Offline.

2 Methods

The Crush compressor consists of two stages. The first analyses the input string using Crochemore's algorithm to generate a two dimensional array C, which stores information on all substrings up to a given length. This data structure is then traversed to calculate a gain measure for all substrings occurring in the leftmost uncompressed position of the input string, and the highest gain substring is chosen for the phrase book. Note that this approach deviates from the Offline algorithm as we make a local choice at the leftmost uncovered position, rather than a global choice over all possible uncovered positions remaining in the string. These two stages are explained in detail in the following two subsections, and summarised in Figure 3.

2.1 Stage 1—String analysis

Crochemore's algorithm [Crochemore, 1981] for finding all repeating substrings in an input string begins by grouping all positions in the string that have the same character into a single class. Each of these classes is then *refined* into subclasses to get repeating substrings of length two. In turn these classes are refined to get substrings of length three, and so on.

For example, consider the input string

$$\begin{array}{cccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ S = & a & b & a & a & b & a & b & a & a & b & a & a & b. \end{array}$$

The positions that form the initial classes for strings of length one are

$$L = 1 \quad \underset{a}{\{1,3,4,6,8,9,11,12\}} \quad \underset{b}{\{2,5,7,10,13\}}$$

This first stage can be accomplished in $O(n)$ time, where n is the number of characters in the input string, assuming that the alphabet from which characters of the string are drawn is indexable (for example, ASCII).

The next stage of the algorithm splits each class into classes that represent the starting position of substrings of length two. The first class, a, splits into the classes aa and ab, and class b splits into the classes ba and $b\$$, where \$ is the "end of string" symbol:

$$L = 2 \quad \underset{ab}{\{1,4,6,9,12\}} \quad \underset{aa}{\{3,8,11\}} \quad \underset{ba}{\{2,5,7,10\}} \quad \underset{b\$}{\{13\}}$$

Using a naïve approach, this stage can be accomplished in $\Theta(n)$ time, simply by checking the character following each position in each class. For example, in order to refine class

$$\underset{a}{\{1,3,4,6,8,9,11,12\}}$$

it would be necessary to check positions $\{2,4,5,7,9,10,12,13\}$ of S. In this case

$$S[4] = S[9] = S[12] = a \text{ and}$$
$$S[2] = S[5] = S[7] = S[10] = S[13] = b$$

so $\{3,8,11\}$ must form the class for aa, and $\{1,4,6,9,12\}$ forms the class for ab.

The process of refinement continues, ignoring any class that contains only a single position, as that must not represent a substring that repeats, until no refinements are possible:

$$\begin{array}{llll}
L = 2 & \underset{ab}{\{1,4,6,9,12\}} & \underset{aa}{\{3,8,11\}} & \underset{ba}{\{2,5,7,10\}} \\
L = 3 & \underset{aba}{\{1,4,6,9\}} & \underset{aab}{\{3,8,11\}} & \underset{baa}{\{2,7,10\}} \\
L = 4 & \underset{abaa}{\{1,6,9\}} & \underset{aaba}{\{3,8\}} & \underset{baab}{\{2,7,10\}} \\
L = 5 & \underset{abaab}{\{1,6,9\}} & \underset{baaba}{\{2,7\}} & \\
L = 6 & \underset{abaaba}{\{1,6\}} & &
\end{array}$$

If the naïve approach to refinement is adopted at each stage, then the total running time is $O(n^2)$, as there could be $O(n)$ levels, each requiring $\Theta(n)$ time. Crochemore offers two insights that allows this time to be reduced to $O(n \log n)$ [Crochemore, 1981].

The first is that it is not necessary to refer back to the original string S in order to refine a class; the refinement can achieved with respect to other classes at the same level. In order for members of a class in level L to be refined into the same class in level $L+1$, they must share the same character in their $L+1$st position. The naïve approach checks this character directly in S for each class member. However, if members of a class share the same $L+1$st character, their length L suffixes must also be identical. For example, substrings aba that all have a b in the next position share the three character suffix bab. For these substrings of length $L+1$ to share a suffix of length L, their positions plus one must all appear in another class at level L.

For example, if the substring aba occurs in position i, and the substring baa occurs in position $i+1$, then taking into account the overlap of the two character suffix of aba and the two character prefix of baa, we can deduce that the string $abaa$ must occur at positions i. This is precisely what happens when refining the class for $aba = \{1,4,6,9\}$ at level 3. We can check which of the positions $\{2,5,7,10\}$ fall in the same class on level 3, and deduce that such strings form a class at level 4. In this case, 2, 7, and 10 all inhabit the class baa, so $\{1,6,9\}$ forms a class at level 4. Similarly, $\{5\}$ is in a class of its own on level 3, so $\{4\}$ forms a class at level 4.

This observation alone does not reduce the running time of the algorithm, but when used in conjunction with the observation that not all classes need be refined at each level, the running time comes down. Consider again the example of refining $aba = \{1,4,6,9\}$ at level 3 into classes on level 4. How many other classes on level three do we need to inspect in order to perform this refinement? As discussed above, each of the other classes must have a prefix of ba so that it overlaps with the suffix of aba. If we inspect the refinements that took place on level 2 to produce level 3 we see that the class $ba = \{2,5,7,10\}$ was split into 2 classes on level 3, namely $baa = \{2,7,10\}$ and $bab = \{5\}$. These are precisely the two classes we need to consider when refining $aba = \{1,4,6,9\}$. This in turn means that if we use one of them to perform the refinement of $aba = \{1,4,6,9\}$, the remaining positions must fall into the other class at level 4. In this case we can either refine $\{1,4,6,9\}$ using $\{5\}$, to get a class $\{4\}$ and a remaining class of $\{1,6,9\}$, or we can refine $\{1,4,6,9\}$ using $\{2,7,10\}$, to get a class $\{1,6,9\}$ and a remaining class of $\{4\}$. Obviously we should choose the smallest classes against which to refine, leaving the largest class as the "left over", with no processing required. This is precisely the approach adopted by Crochemore's algorithm; at each stage only the "small" classes are refined.

Observe that when a class at level L is refined into two or more classes at level $L+1$, the longest of the smallest classes cannot be greater than half of the size of the parent class. So any character in a string can appear in a "small" class at most $O(\log_2 n)$ times, hence can only be involved in a refinement $O(\log n)$ times. Seeing as there are n characters, the overall running time of Crochemore's algorithm is $O(n \log n)$.

This very brief description of the intuition behind Crochemore's algorithm hides some of the complex and intricate details required to achieve a fast, memory efficient implementation of this algorithm. The implementation used in this paper operates in $O(n)$ space, storing only a list of classes for each level of refinement, discarding lists from previous levels. The constant factor is quite high in this space bound, with the current implementation requiring $44n$ bytes of memory.

2.2 Stage II—Phrase selection

In order to use the results from Crochemore's algorithm for phrase selection, our current implementation of Crush stores the class information for each level as it is derived. As memory conservation during encoding is not a primary aim of Crush, a simple array of n integers is used to hold a circular list of class members for each level. More formally, element $C[L][i]$ of array C is a pointer to the next member of the class containing position i on level L, with the final class member pointing back to the first member. In the above example of Crochemore's algorithm, C would be:

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$C[1][i]$	3	5	4	6	7	8	10	9	11	13	12	1	2
$C[2][i]$	4	5	8	6	7	9	10	11	12	2	3	1	13
$C[3][i]$	4	7	8	6	5	9	10	11	1	2	3	12	13
$C[4][i]$	6	7	8	4	5	9	10	3	1	2	11	12	13
$C[5][i]$	6	7	3	4	5	9	2	8	1	10	11	12	13
$C[6][i]$	6	2	3	4	5	1	7	8	9	10	11	12	13

The number of levels is restricted to K, a parameter to Crush, so total space requirement for C is $O(Kn)$.

Once array C exists, phrase selection can begin. Unlike Offline, Crush makes its phrase selections out of the set of substrings beginning at the leftmost uncovered position which do not overlap an already covered position. The offline compressor, however, considers all possible non-overlapping substrings at each phrase choice. CRUSH chooses the phrase p with the highest gain measure G_p out of the set of possible substrings. If $G_p \leq 0$ then the character at the uncovered position is skipped, and left to a final stage of processing. The final stage simply treats all uncovered characters as single letter phrases with infinite G_p, and stores the single letter in the phrase book and its uncovered occurrences as pointers.

Apostolico & Lonardi reported that computing G_p as the cost of storing all occurrences of a phrase with a zero-order character model less the cost of storing a single copy and a series of pointers to the copy gave the best results in their experiments [Apostolico & Lonardi, 2000]. Accordingly, CRUSH uses a similar gain measure.

Let H be the cost in bits of storing a single character in the input string. Using a simple character based model and a statistical coder (for example, Huffman coding or Arithmetic coding), H would be around 2 to 3 bits, while an ASCII code has $H = 8$. Quantity H can be estimated by a preliminary scan of the data which records the probability of each character, p_i, and then setting $H = -\sum p_i \log_2 p_i$, which is Shannon's lower bound on compression levels [Shannon, 1948]. This is the approach adopted by CRUSH.

Let f_p be the frequency with which phrase p occurs in the text, and l_p the number of characters in phrase p. The cost of storing the f_p copies of phrase p uncompressed in the text is approximated by Hf_pl_p bits. If phrase p is chosen for the phrase book, one copy is required at a cost of approximately Hl_p bits for the phrase, plus H bits to store either the length of the phrase, or a terminating symbol for the phrase, in the phrase book. Apart from the phrase book copy of p, it is also necessary to store f_p pointers to that phrase. The cost of a pointer to the new phrase can be estimated by $\lceil \log_2(P + 1)\rceil$, where P is the number of phrases already in the phrase book [Apostolico & Lonardi, 2000]. This is not a very accurate estimate of pointer cost as it amounts to the cost of a flat binary code for the pointers currently in the phrase book. Of course as CRUSH continues, P, the number of phrases will increase, and so the net effect is to slowly make the cost of adding a phrase more expensive. The total gain in compression if phrase p is to be included in the phrase book, therefore, is:

$$\begin{aligned} G_p &= \text{uncompressed representation} \\ &\quad -\text{phrase book entry cost} \\ &\quad -\text{pointer costs} \\ &= Hf_pl_p - H(l_p + 1) - f_p\lceil \log_2(P + 1)\rceil . \end{aligned}$$

Figure 3 shows pseudo code for the complete algorithm used in CRUSH. Steps 1 and 2 simply run Crochemore's algorithm and create the C array, Step 4 performs phrase selection, and Step 5 finishes off any *skipped* positions for which there was no positive gain during the Step 4 processing. The time required by CRUSH is dominated by the traversals of the C lists in Step 4.3.2. For each possible substring, of which there may be $K - 1$, the entire pointer chain of $O(n)$ items must be traversed in order to calculate the frequency of a substring. Step 4.6 also sees the chain of pointers relating to a selected phrase traversed a second time to record pointers and mark the positions as covered. Note Steps 4.3.2 and 4.6 must also exclude self overlapping positions from consideration.

3 Results

An implementation of CRUSH as described above, and an implementation of OFFLINE as downloaded from `www.cs.purdue.edu/homes/stelo/Off-line/` were run on the Purdue corpus [Purdue, 2001]. Table 1 shows the compression and speed results achieved using a Pentium III 800MHz CPU with 640Mb of RAM, 256Kb primary cache, and running Linux. The C code was compiled using `gcc` version `egcs-2.91.66` with full optimisations. Phrases in crush were limited to $K = 40$ characters in length. The values reported in Table 1 used a gain formula of

$$G_p = Hf_pl_p - H(l_p + 1) - f_p\lceil \log_2(P + 1)\rceil - 2f_pl_p .$$

The final term was added in order to bias the phrase selection towards single chars: that is, to reduce the number of phrases chosen. Our initial experiments showed that CRUSH using the gain measure stated in the previous section was too aggressive in its phrase selection, a problem we discuss below. Compression values assume a Huffman coder is used in coding both the phrase book and pointer lists, but prelude-costs for both codes are not included. For both codes, the number of codewords was small (less than 100 in all cases), and so prelude size has negligible effect on final compression levels.

As Table 1 shows, our compression results were competitive on most files of the corpus, which is unusual given our local rather than global approach to phrase selection. One obvious failure of CRUSH is to find good phrases in the file Spor_All_2x, which is the file Spor_All repeated twice. This is an example of the short-comings of the local choice approach we have adopted. The Spor_All_2x file, as for all the files in the Purdue Corpus, consists of 258×2 blocks of about 14 lines of genetic data as shown in Figure 4. On this file, OFFLINE first chooses

```
upstream sequence, from -800 to -1\n
```

as its highest gain phrase, and then proceeds to choose 200 phrases all of length 800 characters (the maximum allowed) and that occur four or less times. CRUSH, on the other hand, must first deal with the characters

```
RTS2   RTS2
```

before it can select the phrase

```
upstream sequence, from -800 to -1\n.
```

In fact, CRUSH determines that

```
S2 upstream sequence, from -800 to -1\n
```

is its first decent phrase, leaving the preceding characters to be encoded as singletons. Amongst CRUSH's phrase choices for this file are the phrases

```
1 upstream sequence, from -800 to -1
2 upstream sequence, from -800 to -1
0 upstream sequence, from -800 to -1
3 upstream sequence, from -800 to -1
4 upstream sequence, from -800 to -1
9 upstream sequence, from -800 to -1
7 upstream sequence, from -800 to -1
8 upstream sequence, from -800 to -1
5 upstream sequence, from -800 to -1
6 upstream sequence, from -800 to -1,
```

which clearly could be improved. Once CRUSH gets to line 2222 of the file, the location of the block OFFLINE designates as its second best phrase, most of that block has already been covered by earlier choices of smaller phrases, and so is not available as a choice to CRUSH.

A similar problem occurred on general text. We ran CRUSH on the small text files from the Calgary [Calgary, 2001] and Canterbury [Canterbury, 2001]

Input	String $S[1 \ldots n]$ to be compressed, and
	K, the maximum length phrase to consider for the phrase book.
Step 1	Create level one classes for Crochemore's algorithm.
Step 2	Run Crochemore's algorithm to level K, storing each level in the C array such that $C[k][i]$ points to the next member of the class containing position i on level k.
Step 3	Set all positions of S to uncovered.
Step 4	While there are uncovered positions in S
Step 4.1	Let i be the smallest uncovered position in S.
Step 4.2	Let j be the min(smallest covered position $> i$, $i + K$).
Step 4.3	For each level $2 \leq k \leq j - i$
Step 4.3.1	Set $f \leftarrow 0$.
Step 4.3.2	For each position c in the list rooted at $C[k][i]$
	If the k positions {c,c+1,. . . ,c+k-1} are all uncovered set $f \leftarrow f + 1$.
Step 4.3.3	Set $G_k \leftarrow Hfk - H(k+1) - f\lceil \log_2(P+1) \rceil$.
Step 4.4	Find $G_m = \max(G_k)$, for all $2 \leq k \leq j - i$.
Step 4.5	If $G_m \leq 0$ then record position i as *skipped*, mark it as covered, and goto Step 4.
Step 4.6	For each position c in the list rooted at $C[m][i]$
Step 4.6.1	If the k positions {c,c+1,. . . ,c+k-1} are all uncovered
	Record a pointer in position c to the new phrase.
	Set positions $\{c, c+1, \ldots, c+m-1\}$ to covered.
Step 5	For each position i recorded as *skipped* in Step 4.5
Step 5.1	If the single character at position i is not a phrase, add it to the phrase book.
Step 5.2	Record a pointer to the phrase at position i.
Output	Phrase book and list of pointers into the phrase book.

Figure 3: The algorithm used in Crush

File Name	Size (bytes)	`bzip2` (bpc)	Offline (bpc)	Crush (bpc)	Offline (secs)	Crush (secs)
Spor_EarlyII	25008	2.894	2.782	2.217	6.2	1.1
Spor_EarlyI	31039	1.882	1.835	2.222	8.5	1.9
Helden_CGN	32871	2.319	2.264	2.219	9.8	2.1
Spor_Middle	54325	2.281	2.176	2.196	21.3	9.0
Helden_All	112507	2.261	2.116	2.227	75.0	58.5
Spor_All	222453	2.218	1.953	2.195	278.5	291.2
All_Up_400k	399615	2.249	2.136	2.275	989.7	1034.8
Spor_All_2x	444906	1.531	0.148	2.194	1133.7	1046.7

Table 1: Compression and timing results for the Purdue corpus.

corpora, but the compression results were abysmal, averaging around four bits per character.

Table 1 also shows that the running time on the Purdue corpus, was not as low as we had anticipated. Running times on the Calgary and Canterbury corpora were exceptionally fast, but this is hardly surprising given the poor compression results. The reason for the low running time on general text is that short phrases are initially chosen which cover much of the input string. Subsequent processing only need look at the remaining substrings, of which there are few.

Table 2 shows a breakdown of the running time on the Purdue corpus generated using the `gprof` software. Profiling of the code indicated that by far the majority of the time was spent in counting the frequency of phrases: Step 4.3.2 in Figure 3. The string processing portion of Crush with Crochemore's algorithm was extremely fast.

File	Steps 1 and 2	Step 4.3.2
Spor_EarlyII	2%	78%
Spor_EarlyI	2%	82%
Helden_CGN	2%	85%
Spor_Middle	1%	91%
Helden_All	0%	96%
Spor_All	0%	98%
All_Up_400k	0%	99%

Table 2: Percentage of running time taken in Crochemore's algorithm (Steps 1 and 2) and frequency counting (Step 4.3.2) by Crush on the Purdue Corpus.

4 Discussion

This paper has reported on a simple attempt to apply Crochemore's algorithm for finding repeating substrings to phrase selection for offline data compression. We have followed the model of Apostolico & Lonardi, greedily choosing phrases at each stage that maximise an approximation of the compression gain expected if the phrase is included in the phrase book [Apostolico & Lonardi, 2000]. Rather than a global approach to phrase selection, however, we trialled a local approach, which has been shown to work well for string covering algorithms [Yang, 2000]. The

```
>RTS2   RTS2 upstream sequence, from -800 to -1
TGAATTCTGCGACGATAGCGGTATTGATGTACTCAACGAGTTTTTTTATCTTTTATGAT
AAAGTTTGTGAACGCAAAATCGTCGGTTTGATTTATGCATGGATCGTTTTTCAAGAAACA
GTAATGGTTCATTTCTTTAATAGCCTTCCATGACTCTTCTAAGTTGAGTTTATCATCAGG
CGATTTCCTTTTGGTAGGGAGTTCGTTTTCTTTGCTGTCTAATGTCACTGATTTTCCTTG
TTTCTCCAAATCACTATTTGACTTCTTTAAGGCAGACAACTTTAAACTCAATGAAACACT
ATGCTGGTTCGGGGTAAAGTCGCTTCTTTTTACCTTGTCTTTTGCAAGTAGCGTCTTCAG
ATCCTTCCTTTTTGTCTCAGCATTGTTATCTTTAGTTTCTCTATTCTTATGTAGTCTTTC
TAGTAAGGATGCACTTTTCGATGTACTATGAGACTGGTCCGCACTTAAAAGGCCTTTAGA
TTTTTCTATTCTTTTTCTTGCTAATTGCTCTAATAACGACATGTCAGGTGGTACTGTAAG
TTTCGAAGACCACCTCCTCGTACGTGTATTGTAGAAGGGTCTCTAGGTTTATACCTCCAA
TCTGTTATAGTACATATTATAGTACACCAATGTAAATCTGGTCCGGGTTACACAACACTT
TGTCCTGTACTTTGAAAACTGGAAAAACTCCGCTAGTTGAAATTAATATCAAATGGAAAA
GTCAGTATCATCATTCTTTTCTTGACAAGTCCTAAAAAGAGCGAAAACACAGGGTTGTTT
GATTGTAGAAAATCACAGCG
>MEK1   MEK1 upstream sequence, from -800 to -1
GTTTAAATCATCAATGTCTTTTTCATCTAGATCATTTAAGTTGTTCAAGTCCAGTTGCTC
...
ACAGAAAGAAGAAGAGCGGA
>NDJ1   NDJ1 upstream sequence, from -800 to -1
GTACGGCCCATTCTGTGGAGGTGGTACTGAAGCAGGTTGAGGAGAGGCATGATGGGGGTT
CTCTGGAACAGCTGATGAAGCAGGTGTTGTTGTCTGTTGAGAGTTAGCCTTAGTGGAAGC
...
```

Figure 4: Beginning of the file Spor_All_2x

string processing portion of our compressor based on Crochemore's algorithm [Crochemore, 1981] is fast, but the subsequent processing stage is limited by a poor data structure.

Several techniques offer hope for improvement to the running time of CRUSH. The first is to replace the pointer chain data structure, which stores the results of Crochemore's algorithm for later processing, with an alternate structure. Recently Smyth & Tang have shown that all the repeating substring information required by CRUSH can be stored in $\Theta(n)$ space arrays [Smyth & Tang, 2001]. Their structure plays the same role as a suffix tree, but is generated directly from Crochemore's algorithm, hence stores only repeating substrings. The overhead for its construction is minimal, and so should not increase the running time of the first stage of CRUSH. Using these arrays will significantly reduce the memory requirements of CRUSH, and speed up the processing of the repeating substring information. Importantly, it should allow the frequency of phrases to be calculated more efficiently, which is a major bottleneck. As shown in Table 2, typically over 90% of the time is spent calculating phrase frequencies in the current implementation.

Another avenue for resource savings is in an alternate implementation of Crochemore's algorithm. Future versions of our software will make use of a new array-based implementation of Crochemore's algorithm [Baghdadi et al, 2001] that in practice runs much faster than the standard implementation and reduces space requirements from $44n$ to $12n$ bytes.

One final avenue worth exploring in this context is a recent algorithm due to [Smyth & Tang, 2001] that calculates all repeating substrings that are nonextendible to both the left and right. Currently Crochemore's algorithm supplies a set of all substrings that are nonextendible only to the right. By running Crochemore's algorithm on the reverse of the string, and collating the results with a run on the string itself, the set of candidate strings for the phrase book should be reduced, while at the same time the utility of the remaining string should be enhanced. We anticipate a substantial improvement in compression results once this approach is implemented.

A major point of deviation between our approach and that of Apostolico & Lonardi is that at each phrase choice, CRUSH considers only those phrases starting at the leftmost uncovered position in the input string—a local approach. This is clearly a major contributing factor to our poor compression levels on general text. Examining the phrase choices made by CRUSH and OFFLINE, for example, on `progc` of the Calgary Corpus, shows that many "good" phrases selected by OFFLINE are unavailable to CRUSH because they have been partially covered by an earlier phrase choice. Indeed, CRUSH only chooses four phrases that are not single characters, hence the poor compression results. This was the reason for the introduction of the $-2f_pl_p$ term in the gain calculation formula. Biasing the gain towards single characters limits the early selection of short phrases that occur very frequently; the very phrases whose selection prevents the use of longer matches later in the processing. The downside is, of course, that it is extremely difficult for any infrequent, long phrases to be chosen. For a file in the Purdue corpus, H is typically around 2.5 bits per character and so

$$G_p \approx 0.5f_pl_p - 2.5l_p - f_p\lceil\log_2(P+1)\rceil .$$

For a phrase of length 800 to be selected, it must occur at least 6 times, whereas on Spor_All_2x, OFFLINE routinely chooses phrases of length 800 with a frequency less than 6.

It is interesting to note that our local approach, however, still gave good compression on the majority of the Purdue corpus. The reason that CRUSH performs well on the Purdue corpus, rather than on other text, is that in general the high gain phrases in the Purdue corpus occur towards the start of the data files. This allows their early selection by CRUSH unlike on general text, where high gain substrings are never considered as possible phrase candidates because parts of those phrases have been covered by an earlier, local choice. Implementing the global approach using the current pointer chain data structure in CRUSH would be prohibitively expensive. Once the tree data structure of [Smyth & Tang, 2001] is incorporated, however, the global approach may become a feasible option. The compression results on the Purdue corpus indicate that a global approach would improve the compression performance on general data.

Another technique that we intend to incorporate in CRUSH is a more accurate gain measure. As the generation of repeating substring information is fast, with a more appropriate data structure to store this information we can afford to spend more time estimating the gain of each phrase. An approximation based on the self entropy of both pointers and the phrase book representation should lead to improved compression levels on a wide range of data. Also an iterative approach that makes multiple passes over the data to improve gain estimates is worth investigating [Cannane & Williams, 2001].

A further avenue for exploration is allowing phrases to overlap. If phrases are allowed to overlap then the data section of the compressed file can no longer be a simple sequential list of pointers to phrases in the phrase book. Each pointer must also be paired with an indication of how much the phrase it represents overlaps the previous text. If Huffman coding or similar is used for storing this information, so that fast decoding is guaranteed, at least one bit per pointer must be added to the final file. Therefore, to maintain the compression levels of the non-overlapping implementation, the average size of the pointers must reduce by one bit. A reduction in pointer size would occur if a suitable change in the frequency distribution of pointers occurred when overlap was allowed, or there was a large reduction in the number of pointers. Our preliminary experiments allowing phrase overlap on the Purdue corpus seem to indicate that allowing overlap is not beneficial.

5 Acknowledgments

Thanks to Lu Yang [Yang, 2000] for making available code for the k-cover algorithm and to F. Franek [Franek & Smyth, 2001] for making available code for a very efficient implementation of Crochemore's algorithm. Thanks also to the anonymous referees for their helpful comments.

References

[Apostolico & Lonardi, 2000] Apostolico, A. & Lonardi, S. (2000). Off-line compression by greedy textual substitution. *Proc. IEEE*, 88(11):1733–1744.

[Baghdadi et al, 2001] Baghdadi, L., Franek, F., Smyth, W. F. & Xiao, X. (2001). A fast space-efficient approach to substring refinement. Preprint.

[Bentley & McIlroy, 1999] Bentley, J. & McIlroy, D. (1999). Data compression using long common strings. In Storer, J. A. & Cohn, M., editors, *Proc. IEEE Data Compression Conference*, pages 287–295, Snowbird, Utah. IEEE Computer Society Press, Los Alamitos, California.

[Calgary, 2001] The Calgary Corpus (2001). http://links.uwaterloo.ca/calgary.corpus.html.

[Cannane & Williams, 2001] Cannane, A. & Williams, H. (2001). General-purpose compression for efficient retrieval. *Journal of the American Society for Information Science*, 52(5):430–437.

[Canterbury, 2001] The Canterbury Corpus (2001). http://corpus.canterbury.ac.nz/.

[Crochemore, 1981] Crochemore, M. (1981). An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250.

[Franek & Smyth, 2001] Franek, F. & Smyth, W. F. (2001). Crochemore's algorithm revisited — a fast space-efficient approach. Preprint.

[Larsson & Moffat, 2000] Larsson, N. & Moffat, A. (2000). Offline dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732.

[Nevill-Manning & Witten, 1994] Nevill-Manning, C. G. & Witten, D. L. M. (1994). Compression by induction of hierarchical grammars. In Storer, J. A. & Cohn, M., editors, *Proc. IEEE Data Compression Conference*, pages 244–253, Snowbird, Utah. IEEE Computer Society Press, Los Alamitos, California.

[Purdue, 2001] The Purdue Corpus (2001). http://www.cs.purdue.edu/homes/stelo/Off-line/.

[Rubin, 1976] Rubin, F. (1976). Experiments in text file compression. *Communications of the ACM*, 19(11):617–623.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423, 623–656. Reproduced in [Shannon & Weaver, 1971].

[Shannon & Weaver, 1971] Shannon, C. E. & Weaver, W. (1971). *The Mathematical Theory of Communication*. The University of Illinois Press, Urbana, Illinois.

[Smyth & Tang, 2001] Smyth, W. F. & Tang, Y. (2001). Computing all repeats using $O(n)$ space and $O(n \log n)$ time. Preprint.

[Storer & Szymanski, 1982] Storer, J. A. & Szymanski, T. G. (1982). Data compression via textual substitution. *Journal of the ACM*, 29:928–951.

[Turpin & Moffat, 2000] Turpin, A. & Moffat, A. (2000). Housekeeping for prefix coding. *IEEE Transactions on Communications*, 48(4):622–628.

[Ukkonen, 1995] Ukkonen, E. (1995). Online construction of suffix trees. *Algorithmica*, 14(3):249–260.

[Yang, 2000] Yang, Lu (2000). Computing a k-cover of a string. M.Sc. thesis, Department of Computing & Software, McMaster University, Hamilton, Ontario, Canada.

[Ziv & Lempel, 1977] Ziv, J. & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT–23(3):337–343.