Technical Correspondence

A LOCALLY ADAPTIVE DATA COMPRESSION SCHEME

I would like to mention that the main results of the paper "A Locally Adaptive Data Compression Scheme" by Bentley, Sleator, Tarjan, and Wei [1] coincides with many of the results of my paper "Data Compression by Means of a 'Book Stack'" [3].

Both papers deal with the encoding of words over an alphabet A. Each letter is given a code in order to make the code length of a word as small as possible. The frequencies of letters are not known a priori.

The same coding procedure was developed in both papers. It is called "book stack" in my paper and "move-to-front" in theirs. It is as follows: the alphabet A coincides as a stack of books. As soon as a letter occurs, the corresponding "book" is taken from the stack output on its top, etc. If a letter is located on the ith place in the stack, then its code length is $f(i)$. There are codes with $f(i) = (1)\ 1 + 2\lfloor \log i \rfloor$, $(2)\ 1 + \log i \rfloor + 2\lfloor \log(1 + \log i) \rfloor$, $(3)\ 1 + \lfloor \log i \rfloor + \lfloor \log(1 + \log i) \rfloor + 2\lfloor \log(1 + (1 + \log i)) \rfloor$, $(4)\ \lfloor \log i \rfloor + \lfloor \log(1 + \log n) \rfloor$, and $(5)\ \lfloor \log i \rfloor + \lfloor \log(1 + \log i) \rfloor + \lfloor \log(1 + \log(1 + \log n) \rfloor$.

The optimal universal code for monotonic source from my paper "Encoding of a Source with Unknown but Ordered Probabilities" [2] is used in their paper. For that encoding $f(i) = \log i + \log\log n + 0(1)$. If A is a countable alphabet, then $f(i) = \log i + 0(\log\log i)$. So, in both papers, the code length equals $\log i$ to within an addend $0(\log\log)$.

There are upper bounds for the average code length: $H + \log\log n + 0(1)$ in theirs; and $H + 1 + 2\log(1 + H)$, where H denotes the entropy, in mine.

I believe that the discussed coding procedure has many attractive features: It is simple to implement, it does not require a priori knowledge of source statistics, and it yields the average code length arbitrarily closed to the entropy (if applied to the blocks of letters). Those features are described in both papers.

B.Y. Ryabko
Kropotkin Street 118-426
Novosibirsk-2
USSR, 630 002

REFERENCES

1. Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr. 1986), 320–330.
2. Ryabko, B.Y. Encoding of a source with unknown but ordered probabilities. *Prob. Inf. Transm.* (1979).
3. Ryabko, B.Y. Data compression by means of a "book stack." *Prob. Inf. Transm. 16*, 4 (1980). (There is an English translation by Consultants Bureau, New York, 1981.)

A recent paper [1] describes a data compression scheme that resembles one we have investigated [7]. Although our paper was not published, the main results appeared shortly afterwards in the proceedings of a conference [8]. Some of our experimental results should be of interest to readers because they address some of the questions raised in the paper. We also have some observations to make about the paper.

Like Bentley et al., we used a list of words and encoded words using their index positions in the list. We also organized the list so that the words most likely to be used were at the front and used a coding scheme for index positions so that small index values had shorter encodings. We observed that maintaining a list of words or tokens where a maximum length restriction is imposed on the list is very similar to the page replacement problem in virtual-memory computer systems [5]. Almost any page replacement algorithm is usable for deciding which words or tokens should be kept in the list and for determining the order of items in the list. The move-to-the front heuristic, examined in [1], is identical to the least-recently-used (LRU) page replacement policy. In our paper we experimentally compared the compression performance using the well-known paging policies of LRU, FIFO (first in, first out), LFU (least frequently used), second chance, and climb (also known as the transposition heuristic) [6, 10]. In the cases of FIFO and second chance, the policies do not impose a natural ordering on items in the list. We simply ordered the list according to age—the words or tokens added to the list most recently are kept at the front. The comparative performance of the five methods are summarized in Table I. (These results appeared as Table 3 in [7].) The table shows the percent reduction in size as a function of three factors, namely, the replacement policy, the maximum size of the list, and the kind of data being compressed. On the whole, FIFO does not perform as well as the other policies. There is little choice, however, among these five policies. In our opinion, the policy that is the easiest to implement should be used and we believe this policy is climb. With the climb policy, a word or token advances one position closer to the front of the list each time it is used. It causes fewer implementation headaches because no more than two tokens must have their index positions updated in the hash table with each access.

We have some further comments to add. First, in the two-pass experimental version of their compression scheme, Bentley et al. use Huffman coding to encode index positions in the list. Later in the paper, a number of static coding schemes that use about $\log(i)$ bits to encode the integer i are suggested. The use of any fixed

coding scheme corresponds to making an assumption about the distribution of accesses to different list positions. We suggest that adaptive Huffman coding is the ideal method to use. Character-level adaptive Huffman coding is also appropriate in encoding words or tokens when they are encountered for the first time.

Second, in the "Remarks" section of their paper, a scheme is suggested for making dynamic Huffman coding locally adaptive. This involves incrementing the weight for a word each time it is used and periodically multiplying all weights by a factor less than 1. (This has the effect of making recent uses of a word count more than less recent uses.) This multiplication generally requires redetermination of the entire set of Huffman codes, due to the truncation error in multiplying the counts by a factor less than 1. The truncation error also results in codes that are suboptimal. We would like to point out that the same effect can be achieved without information loss if weights are incremented by ever-increasing amounts. For example, the increments could be 1, 2, 4, 8, etc. Algorithms for maintaining Huffman codes when weights increase by non-unit amounts appears in [3].

Finally, we question whether or not the performance results are good enough to justify the complexity of the implementation. Although the scheme beats Huffman coding by a reasonable margin, its compression performance is inferior to that of Ziv–Lempel compression [11], a compression method that is both simpler to implement and executes much faster. Newer compression methods by Cleary and Witten [2] and Cormack and Horspool [4, 9] achieve still better compression. Although considerably slower than the Ziv and Lempel, the latter scheme is competitive in speed with the method described here.

R. Nigel Horspool
Department of Computer Science
University of Victoria
P.O. Box 1700
Victoria, British Columbia, Canada V8W 2Y2

and

TABLE I. Compression Factors

Replacement policy	Test file	Maximum length of list 16	32	64	128	256
Least recently used	T	42.8	46.1	49.6	52.4	56.3
	P	59.6	63.6	68.0	71.2	72.8
	C	45.6	53.2	57.0	59.1	60.2
	F	43.2	49.7	53.4	56.3	58.9
	B	20.6	21.2	21.6	22.0	22.4
Least frequently used	T	40.7	43.8	47.9	50.6	54.8
	P	56.2	58.3	60.1	62.9	66.8
	C	44.2	45.3	48.5	54.7	61.3
	F	40.3	43.4	46.2	50.0	56.0
	B	20.4	21.8	21.9	23.0	25.1
First in, first out	T	38.6	41.4	42.7	43.9	47.2
	P	56.0	60.0	62.9	65.1	66.9
	C	43.4	48.7	51.4	54.6	57.4
	F	40.3	44.9	46.5	47.9	53.0
	B	16.1	13.5	11.1	9.0	11.2
Second chance	T	40.5	44.1	50.0	52.7	56.6
	P	57.6	63.7	68.5	71.7	73.3
	C	44.6	53.2	57.2	59.5	59.0
	F	41.8	49.0	53.4	56.1	58.8
	B	17.3	17.2	17.9	19.8	24.5
Climb	T	42.2	46.0	48.2	50.5	55.2
	P	56.8	59.8	61.4	67.3	70.4
	C	44.4	45.4	47.0	52.4	58.6
	F	41.2	44.6	48.4	49.7	55.0
	B	20.3	21.3	21.5	21.2	21.8

The test files were

T formatted text (73,385 bytes),
P Pascal source code (143,126 bytes),
C C source code (17,161 bytes),
F Fortran source code (40,975 bytes), and
B binary object code (18,432 bytes).

Gordon V. Cormack
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

REFERENCES

1. Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr. 1986), 320–330.
2. Cleary, J.G., and Witten, I.H. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun. COM-32*, 4 (Apr. 1984), 396–402.
3. Cormack, G.V., and Horspool, R.N. Algorithms for adaptive Huffman codes. *Inf. Process. Lett. 18*, 3 (Mar. 1984), 159–166.
4. Cormack, G.V., and Horspool, R.N.S. Data compression using dynamic Markov modelling. *Comput. J.* To be published.
5. Denning, P.J. Virtual memory. *ACM Comput. Surv. 2*, 3 (Sept. 1970), 153–189.
6. Franaszek, P.A., and Wagner, T.J. Some distribution-free aspects of paging algorithm performance. *J. ACM 21*, 1 (Jan. 1974), 31–39.
7. Horspool, R.N., and Cormack, G.V. Data compression based on token recognition. Oct. 1983. Unpublished manuscript.
8. Horspool, R.N., and Cormack, G.V. A general purpose data compression technique with practical applications. In *Proceedings of the CIPS Session 84* (Calgary, Alberta, May 9–11). 1984, pp. 138–141.
9. Horspool, R.N., and Cormack, G.V. Dynamic Markov modelling—A prediction technique. In *Proceedings of the 19th Hawaii International Conference on System Sciences* (Honolulu, Hawaii, Jan. 7–10). 1986, pp. 700–707.
10. Rivest, R. On self-organizing sequential search heuristics. *Commun. ACM 19*, 2 (Feb. 1976), 63–67.
11. Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory IT-24*, 5 (Sept. 1976), 530–536.

CORRECTIONS TO "COMBINATORIALLY IMPLOSIVE ALGORITHMS"

Set covering is an important tool for modeling abductive reasoning in artificial intelligence. It can be applied to a wide range of problems such as diagnostic problem solving [8, 11, 12], sequential error classification [1], and natural-language processing [3], to mention but a few. Extending on the ideas of set covering, a new language for abductive logic, ABLOG-IC, is currently being developed [2]. Several ideas of set covering have been used in the past for developing expert systems (e.g., [7, 9, 10, 14]). Set-covering concepts have also been extended to fuzzy sets [15], and work has been done on developing a theory of diagnosis based on logic that is similar to the set-covering approach [13].

Recently, in reviewing the literature on this topic, the authors came across the predicate covering algorithms proposed by Kornfeld [6]. On careful review of the proposed parallel algorithm `BottomUp` described therein, several errors were encountered. This letter specifically addresses those errors and gives the corrected version of the algorithm. The errors are discussed in two stages: First, the typographical errors are pointed out and corrected, followed by a list of logical errors with corresponding corrections that the authors recommend. The original version of the `BottomUp` algorithm as it appears in [6] is presented for reference, and some of the terms and concepts used in the algorithm are defined.

A set of clauses is said to *work* with respect to a predicate clause P, if P implies the disjunction of the clauses in the set. A set of clauses that works is also *minimal* if no subset of this set works (with respect to the same predicate clause). A set that works is called a *cover* and is called a *minimal cover* if it is also minimal. An *activity* is a locus of control with some purpose. A *sprite* is a pattern-invoked procedure associated with an activity, which watches for assertions that match its pattern in a global database. It is specified as a `(when [—pattern—] —body—)` statement, where the body is executed whenever a matching pattern is asserted. An activity may spawn other subactivities. When an activity is *stifled*, it and all its subactivities stop processing. The processing of an activity constitutes executing the bodies of those sprites that are triggered by assertions in that activity, and any `Execute` command associated with the activity. The `Execute` command takes two arguments: a procedure with its arguments, and an activity object that uniquely identifies an activity. It executes the procedure in the activity associated with the activity object. Activities and their identifying activity objects are created by the `NewActivity` command. More about the above terms and concepts can be found in [4]–[6].

The `BottomUp` algorithm uses the `Test-Workingness` procedure that takes P, a predicate, and `set`, a set of predicates, as its arguments. `Test-Workingness` checks if `set` works with respect to P, and asserts either `Works{set}` or `Notworks{set}`, which may then trigger some sprites. The `BottomUp` algorithm that appeared in [6] is presented below.

```
1   (define BottomUp (P set allpreds)
2     (foreach Q ∈ allpreds
3       (let ((activity (NewActivity)))
4         (if P ∉ set
5           (Execute (TestWorkingness
              P {set ∪ {Q}}) activity)
6           (when [(NotWorks {set ∪ {Q}})]
7             (Stifle activity)
8             (BottomUp P {set ∪ {Q}}
                allpreds)
9           (when [(NotWorks {set ∪ {Q}})]
10            (Stifle activity))
11          (when [(Works {set ∪ {Q}})]
12            (Assert (WorkAndMinimal
                {set ∪ {Q}})))))))))
```

Referring to the algorithm above, line 4 is redundant since the condition $P \notin$ `set` is always true. When the first call is made to `BottomUp`, `set` is $\emptyset$ and so, trivially, $P \notin$ `set`. A subsequent call to `BottomUp` is made in the first sprite, which calls `BottomUp` recursively, with P, `set` $\cup \{Q\}$, and `allpreds` as arguments, provided `set` $\cup \{Q\}$ does not work (lines 6–8). Now, if $P \in$ `set` $\cup \{Q\}$, then `set` $\cup \{Q\}$ would have worked, and hence the recursive call itself would not have been made. Perhaps the original intention was to have the test $Q \notin$ `set` instead. This would eliminate fruitless recursive function calls such as `(BottomUp` P `{set` $\cup$