providing no error protection. This unique error protection shall result in a reduction of storage requirements or a reduction in transmission bandwidth because the use of a lower level of error protection or no error protection will reduce the introduction of redundant data into the data link or storage medium while still providing error resiliency.

As known to those skilled in the art, a variety of proposed codes can be separated into a prefix and suffix fields as described above. See, for example, E. R. Fiala and D. H. Greene, "Data Compression with Finite Windows," *Communications of the ACM*, Vol. 32, No. 4, pp. 490-505 (1989). However, the proposed codes have not previously been separated in order to provide error resiliency as provided by the method and apparatus of the present invention.

It will be apparent to those skilled in the art that split field coding can be applied to data sets which are not characterized by a well-behaved distribution. This application can be accomplished by initially sorting the data set to produce a re-ordered monotonic distribution. This approach will result in error resilience in the sense that a bit error in the suffix field will not result in a loss of code word synchronization. However, the resulting error in the decoded value will not be constrained to a particular range since the sorting of the data set will destroy the contiguity of the superbins associated with specific prefix field values.

The relative positions of the significant coefficients can also be encoded in a variety of manners, such as run length coding as described above. The resulting run length values can, in turn, also be entropy encoded using an approach such as Huffman coding or the split field coding method described above. Alternatively, the positions of the significant coefficients can be encoded by other methods known to those skilled in the art, such as tree structures or coefficient maps, without departing from the spirit and scope of the present invention.

Once the plurality of code words representative of the quantized coefficients and the encoded run lengths have been generated, the run length code words and the prefix fields of the quantized coefficient code words are preferably error protected at an appropriately high level of error protection, as shown in block **38**. The run length code words are preferably afforded protection because a misdecoded run length value can potentially introduce catastrophic distortion into the reconstructed image. However, the suffix fields of the quantized coefficient code words are preferably error protected at a relatively lower level of error protection, if at all. As shown schematically in Figure 1, the data encoder **16** can therefore include unequal error protection means **29** for providing appropriate levels of error protection to the encoded data as described above.

Regardless of the error protection means, error protection adds redundancy to the encoded data and increases the storage and transmission requirements. Accordingly, by providing a reduced level of error protection or no error protection to the suffix fields of the quantized coefficient code words, the storage and transmission requirements can be reduced by the method and apparatus of the present invention while limiting the effects of bit errors incident upon the suffix fields of the quantized coefficient code words.

Following the data compression process described above, the encoded data can be efficiently stored. For example, the run length code words and the prefix fields of the quantized coefficient code words can be stored in a first data block **66** defined by a storage medium **18**, such as a magnetic disk storage which is error protected as shown in Figure 6. In addition, the respective suffix fields of the quantized coefficient code words can be stored in a second data block **68** defined by a storage medium which includes a reduced level of error protection or no error protection. Thus, the suffix fields can be more efficiently stored within the second data block.

Likewise, the compressed and encoded data can be efficiently transmitted, such as via first and second data links. In particular the error resilient method and apparatus of the present invention can include a transmitter **20** which transmits the respective run length code words and the prefix fields of the quantized coefficient code words via a first data link **22** which is error protected, and which transmits the respective suffix fields of the quantized coefficient code words via a second data link **24** which is not error protected or is error protected to a lesser degree than the first data link. Thus, the suffix fields can be more efficiently transmitted (with reduced or no redundancy) using the second data link.

Upon reception of the compressed data, the prefix fields of the quantized coefficient code words can be decoded (as shown in Figure 7) and the lengths of the suffix fields can be determined based on the decoded prefix fields. If one or more bit errors are incident upon the suffix field of a quantized coefficient code word, the code word synchronization is not lost because the length of the suffix field is known. As a result, the resulting error in the decoded coefficient value will be constrained to the range of coefficient values for the superbin corresponding to the associated prefix field. Accordingly, the effects of the error on the reconstructed image will be limited and will not be catastrophic. Following the transmission of the encoded data and the possible detection and correction of any storage and

Z/n

transmission errors by means of channel decoding known to those skilled in the art, the compressed data, including both the quantized values for the significant coefficients and the relative positions of the significant coefficients, is decoded, dequantized, and inverse transformed, as known to those skilled in the art, so as to provide a reconstructed image based upon the original image as shown in Figure 7.

The error resilient method and apparatus for compressing data, including the data transformer **12**, the data quantizer **14**, the data encoder **16** and the unequal error protection means **29**, are preferably implemented by a combination of hardware and software. For example, the method and apparatus for compressing data can be implemented by a computer having one or more controllers which operate under the control of software to provide the data transformation, quantization and encoding processes described above.

In the drawings and the specification, there has been set forth a preferred embodiment of the invention and, although specific terms are employed, the terms are used in a generic and descriptive sense only and not for purpose of limitation, the scope of the invention being set forth in the following claims.

THAT WHICH IS CLAIMED IS:

1. An error resilient method of encoding data comprising the steps of:

generating a plurality of code words representative of respective portions of the data, wherein each code word comprises a first portion and an associated second portion, and wherein said code word generating step comprises the steps of:

generating the first portion of each code word, the first portion including information representative of a predetermined characteristic of the associated second portion; and

generating the second portion of each code word, the second portion including information representative of the respective portion of the data; and

providing error protection to at least one of the first portions of the plurality of code words while maintaining any error protection provided to the respective second portion associated with the at least one first portion at a lower level than the error protection provided to the respective first portion.

2. An error resilient method of encoding data according to Claim 1 wherein said step of generating a plurality of code words comprises the step of entropy coding the data to thereby reduce the size of the resulting code words.

3. An error resilient method of encoding data according to Claim 1 wherein said step of generating the second portion of each code word comprises the step of generating second portions having predetermined numbers of characters, and wherein said step of generating the first portion of each code word comprises the step of generating first portions which

include information representative of the predetermined number of characters which comprise the associated second portion.

4. An error resilient method of encoding data according to Claim 3 further comprising the step of determining the probability with which respective ones of the plurality of code words are generated, wherein said step of generating second portions having predetermined numbers of characters comprises the step of generating a plurality of second portions having the same predetermined number of characters, and wherein the plurality of second portions which have the same predetermined number of characters comprise portions of respective code words which have corresponding probabilities of generation within a predetermined range of probabilities.

5. An error resilient method of encoding data according to Claim 1 wherein said step of providing error protection comprises the steps of:

storing the at least one first portion of the plurality of code words in a first data block of a storage medium, wherein the first data block is error protected; and

storing the respective second portion associated with the at least one first portion in a second data block of the storage medium, wherein any error protection provided by the second data block is at a lower level than the error protection provided by the first data block.

6. An error resilient method of encoding data according to Claim 1 wherein said step of providing error protection comprises the steps of:

Z0

transmitting the at least one first portion of the plurality of code words via a first data link, wherein the first data link is error protected; and

transmitting the respective second portion associated with the at least one first portion via a second data link, wherein any error protection provided by the second data link is at a lower level than the error protection provided by the first data link.

7. A data encoding apparatus comprising:

code word generating means for generating a plurality of code words representative of respective portions of the data, wherein each code word comprises a first portion and an associated second portion, and wherein said code word generating means comprises:

first generating means for generating the first portion of each code word, the first portion including information representative of a predetermined characteristic of the associated second portion; and

second generating means for generating the second portion of each code word, the second portion including information representative of the respective portion of the data; and

error protection means for providing error protection to at least one of the first portions of the plurality of code words while maintaining any error protection provided to the respective second portion associated with the at least one first portion at a lower level than the error protection provided to the respective first portion.

8. A data encoding apparatus according to Claim 7 wherein said code word generating means comprises entropy coding means for entropy coding the data to thereby reduce the size of the resulting code words.

9. A data encoding apparatus according to Claim 7 wherein said second generating means generates second portions having predetermined numbers of characters, and wherein said first generating means generates first portions which include information representative of the predetermined number of characters which comprise the associated second portion.

10. A data encoding apparatus according to Claim 7 wherein said error protection means comprises a storage medium for storing the plurality of code words, said storage medium being partitioned into a first data block which is error protected and a second data block, wherein any error protection provided by the second data block is at a lower level than the error protection provided by the first data block, wherein the at least first portion of the plurality of code words is stored in the first data block of the storage medium, and wherein the respective second portion associated with the at least one first portion is stored in the second data block of the storage medium.

11. A data encoding apparatus according to Claim 7 wherein said error protection means comprises:

first data link transmitting means for transmitting the at least one first portion of the plurality of code words via a first data link, wherein the first data link is error protected; and

second data link transmitting means for transmitting the respective second portion associated with the at least one first portion via a second data link, wherein any error protection provided by said second data link is at a lower level than the error protection provided by said first data link.

U |

12. An error resilient method of compressing data comprising the steps of:

transforming the data based upon a predetermined transformation function;

quantizing the transformed data such that the quantized data has fewer unique coefficients than the transformed data; and

encoding the quantized data, said encoding step comprising the steps of:

generating a plurality of code words, representative of respective portions of the data, which have respective first and second portions, wherein the first portion includes information representative of a predetermined characteristic of the associated second portion, and wherein the second portion includes information representative of a respective portion of the data; and

providing error protection to at least one of the first portions of the plurality of code words while maintaining any error protection provided to the respective second portion associated with the at least one first portion at a lower level than the error protection provided to the respective first portion.

13. An error resilient method of compressing data according to Claim 12 wherein said step of encoding the quantized data comprises the step of entropy coding the quantized data to thereby reduce the size of the resulting code words.

14. An error resilient method of compressing data according to Claim 12 wherein said step of generating a plurality of code words comprises the steps of:

generating second portions having predetermined numbers of characters; and generating first portions which include information representative of the predetermined number of characters which comprise the associated second portion.

15. An error resilient method of compressing data according to Claim 14 further comprising the step of determining the probability of occurrence of respective ones of the quantized data values, wherein said step of generating second portions having predetermined numbers of characters comprises the step of generating a plurality of second portions having the same predetermined number of characters, and wherein the plurality of second portions which have the same predetermined number of characters comprise portions of respective code words which represent quantized data values having corresponding probabilities of generation within a predetermined range of probabilities.

16. An error resilient method of compressing data according to Claim 12 wherein said step of providing error protection comprises the steps of:

storing the at least one first portion of the plurality of code words in a first data block of a storage medium, wherein the first data block is error protected; and

storing the respective second portion associated with the at least one first portion in a second data block of the storage medium, wherein any error protection provided by the second data block is at a lower level than the error protection provided by the first data block.

17. An error resilient method of compressing data according to Claim 12 wherein said step of providing error protection comprises the steps of:

112

transmitting the at least one first portion of the plurality of code words via a first data link, wherein the first data link is error protected; and

transmitting the respective second portion associated with the at least one first portion via a second data link, wherein any error protection provided by the second data link is at a lower level than the error protection provided by the first data link.

18. An error resilient method of compressing data according to Claim 12 wherein said transforming step comprises the step of transforming the data based upon a wavelet transform.

19. An error resilient method of compressing data according to Claim 18 wherein said transforming step comprises the step of transforming the data based upon a biorthogonal wavelet transform.

20. An error resilient method of compressing data according to Claim 12 wherein the transformed data includes a plurality of transformed coefficients, and wherein said quantizing step comprises the step of detecting transformed coefficients below a predetermined clipping threshold.

21. An error resilient method of compressing data according to Claim 20 further comprising the step of establishing a clipping threshold such that the ratio of the number of detected coefficients to the number of transformed coefficients which are not detected is at least as great as a predetermined clipping ratio.

Sub a4

22. An error resilient data compression apparatus comprising:

1114

a data transformer for transforming the data based upon a predetermined transformation function;

a data quantizer for quantizing the transformed data such that the quantized data has fewer unique coefficients than the transformed data; and

a data encoder for encoding the quantized data, said data encoder comprising:

code word generating means for generating a plurality of code words, representative of respective portions of the data, which have respective first and second portions, wherein the first portion includes information representative of a predetermined characteristic of the associated second portion, and wherein the second portion includes information representative of a respective portion of the data; and

error protection means for providing error protection to at least one of the first portions of the plurality of code words while maintaining any error protection provided to the respective second portion associated with the at least one first portion at a lower level than the error protection provided to the respective first portion.

23. An error resilient data compression apparatus according to Claim 22 wherein said data encoder comprises entropy coding means for entropy coding the quantized data to thereby reduce the size of the resulting code words.

24. An error resilient data compression apparatus according to Claim 22 wherein said code word generating means comprises:

second generating means for generating second portions having predetermined numbers of characters; and

first generating means for generating first portions which include information representative of

11

the predetermined number of characters which comprise the associated second portion.

25. An error resilient data compression apparatus according to Claim 22 wherein said error protection means comprises a storage medium for storing the plurality of code words, said storage medium being partitioned into a first error protected data block and a second data block, wherein any error protection provided by said second data block is at a lower level than the error protection provided by said first data block, wherein the at least first portion of the plurality of code words is stored in the first data block of the storage medium, and wherein the respective second portion associated with the at least one first portion is stored in the second data block of the storage medium.

26. An error resilient data compression apparatus according to Claim 22 wherein said error protection means comprises:

first data link transmitting means for transmitting the at least one first portion of the plurality of code words via a first data link, wherein the first data link is error protected; and

second data link transmitting means for transmitting the respective second portion associated with the at least one first portion via a second data link, wherein any error protection provided by the second data link is at a lower level than the error protection provided by the first data link.

27. An error resilient data compression apparatus according to Claim 22 wherein said data transformer comprises a wavelet transformer for transforming the data based upon the wavelet transformer.

WID

Sub 15

28. A computer readable memory for storing error resilient encoded data, the computer readable memory comprising:

a storage medium for storing the error resilient encoded data, said storage medium being partitioned into a first error protected data block and a second data block, wherein any error protection provided by said second data block is at a lower level than the error protection provided by said first data block; and

a plurality of code words, representative of respective portions of the original data, which have respective first and second portions, wherein the first portion includes information representative of a predetermined characteristic of the associated second portion, and wherein the second portion includes information representative of a respective portion of the original data,

wherein at least one of the first portions of the plurality of code words is stored in the first data block of said storage medium such that the at least one first portion is error protected, and wherein the respective second portion associated with the at least one first portion is stored in the second data block of said storage medium such that any error protection provided to the respective second portion associated with the at least one first portion is at a lower level than the error protection provided to the respective first portion.

29. A computer readable memory for storing error resilient encoded data according to Claim 21 wherein the second portion of each code word has a predetermined number of characters, and wherein the first portion of each code word includes information representative of the predetermined number of

characters which comprise the associated second portion.

30. A computer readable memory for storing error resilient encoded data according to Claim 29 wherein each of the plurality of code words occurs according to a predetermined probability, wherein a plurality of second portions of code words have the same predetermined number of characters, and wherein the plurality of second portions which have the same predetermined number of characters comprise portions of respective code words which have corresponding probabilities of occurrence within a predetermined range of probabilities.

08/633896

BT
18MAR98

~~ERROR RESILIENT METHOD AND APPARATUS FOR ENTROPY CODING~~

ABSTRACT OF THE DISCLOSURE

The error resilient method and apparatus for encoding data includes an encoder including a code word generator for generating a plurality of code words representative of respective portions of the data. The code word generator encodes data pursuant to split field coding in which each code word includes a prefix field and an associated suffix field. The prefix field includes information representative of a predetermined characteristic of the associated suffix field, such as the predetermined number of characters which form the associated suffix field. In addition, the suffix fields include information representative of at least some of the original data. Consequently, if the prefix field of a code word is decoded correctly, i.e, without the occurrence of bit error, the error resilient method and apparatus can correctly determine the length of the associated suffix field and the range of coefficient values to be represented by the associated suffix field such that the associated suffix field is resilient to errors. In order to increase the probability that the prefix field will be correctly decoded, the method and apparatus protects the prefix and suffix fields of the encoded data to greater and lesser degrees, respectively, such that the data can be more efficiently compressed.

~~223119~~

BT
5JULY97

PRINT OF DRAWINGS
AS ORIGINALLY FILED

08/633896

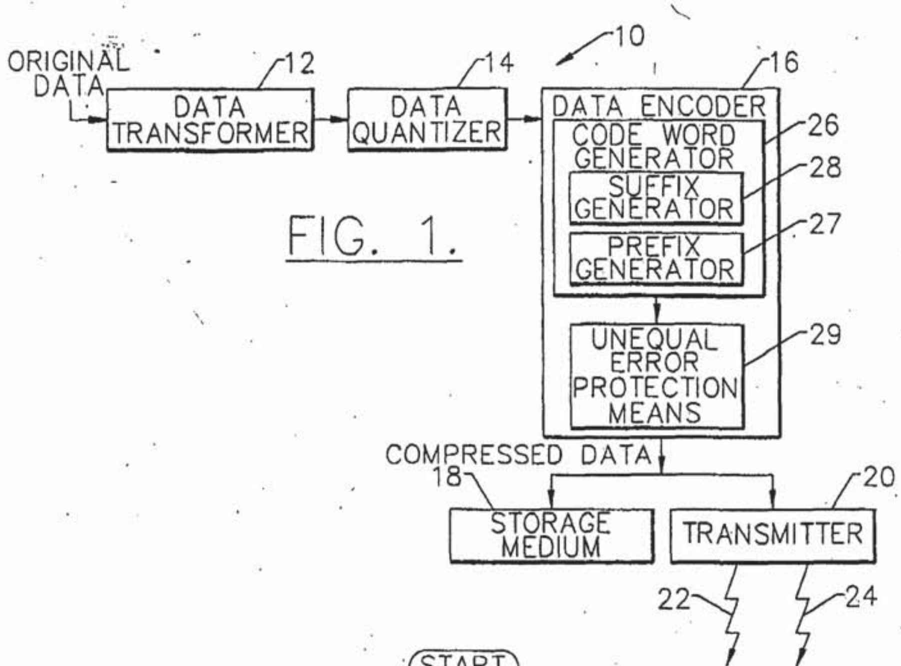


FIG. 1.

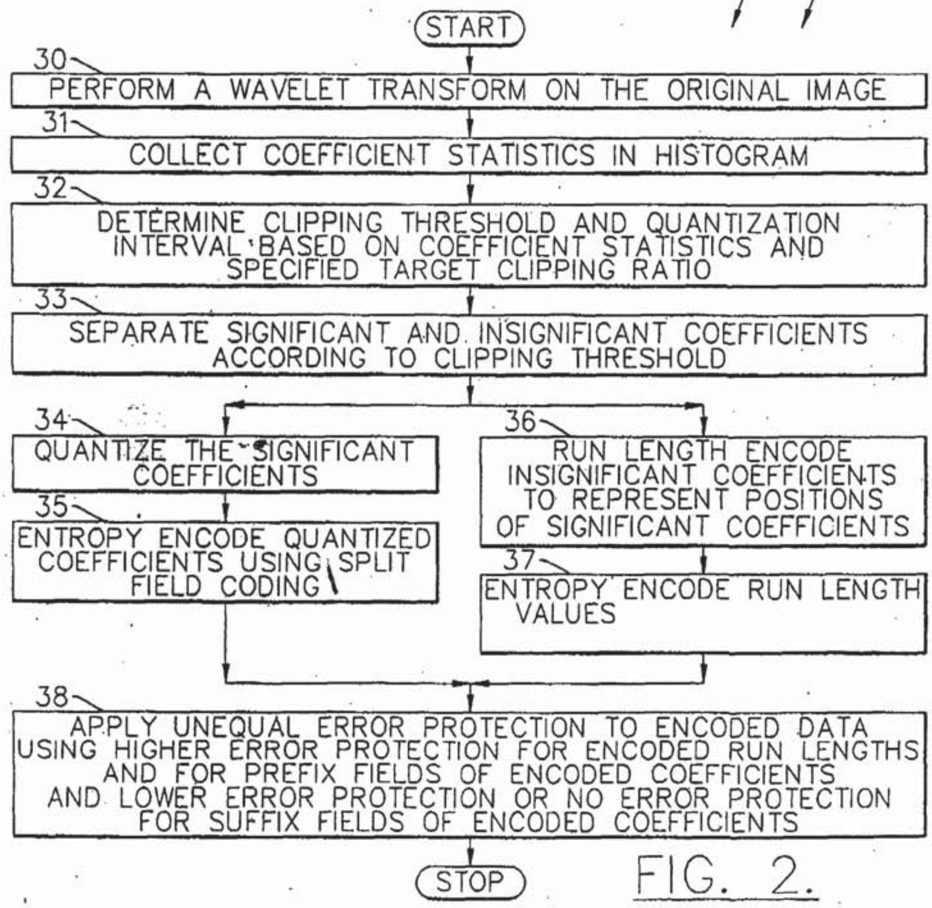


FIG. 2.

PRINT OF DRAWINGS
AS ORIGINALLY FILED

08/633896

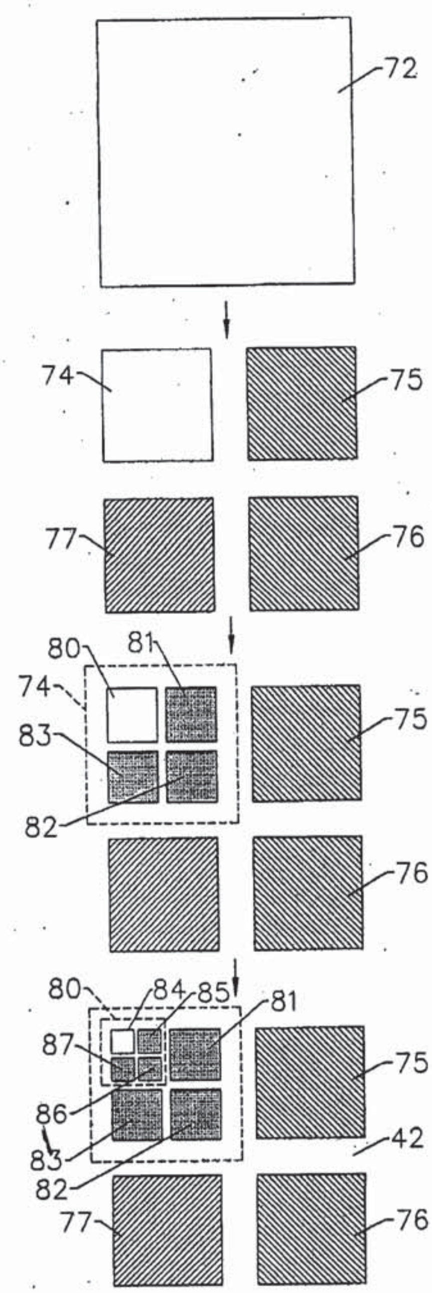


FIG. 3.

PRINT OF DRAWINGS
AS ORIGINALLY FILED

08/655552

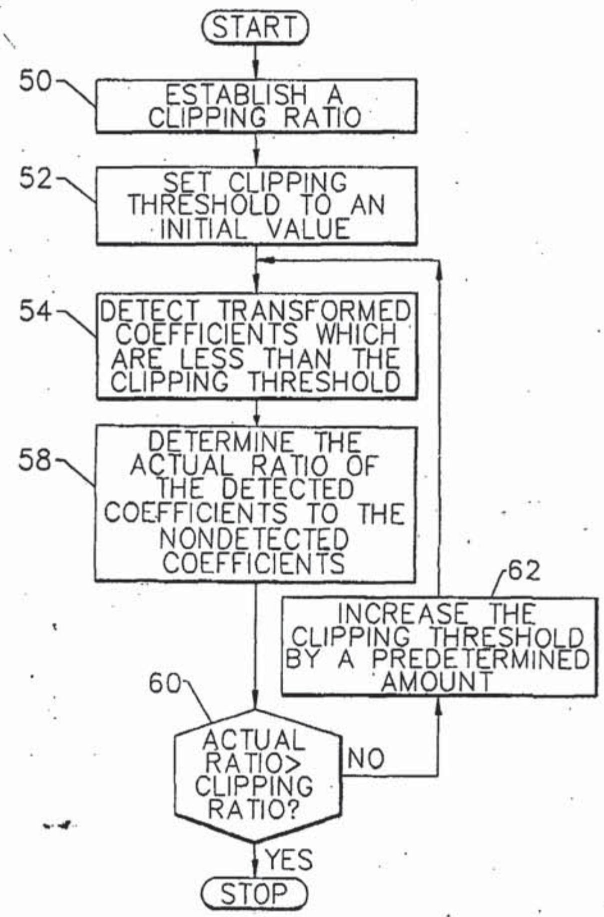


FIG. 4.

PRINT OF DRAWINGS
AS ORIGINALLY FILED

08/633896

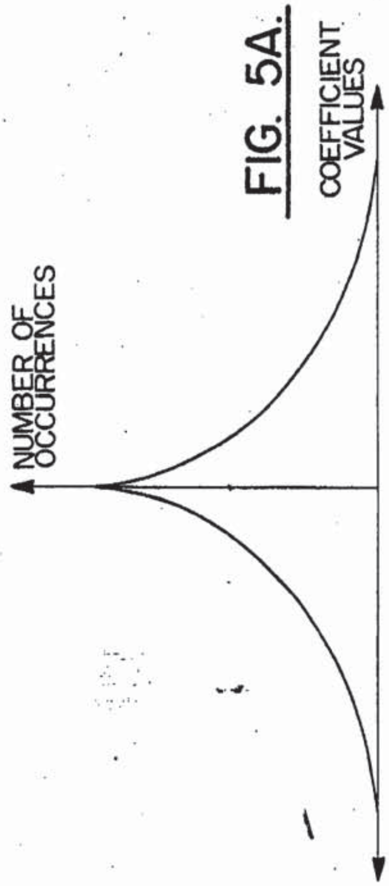


FIG. 5A.

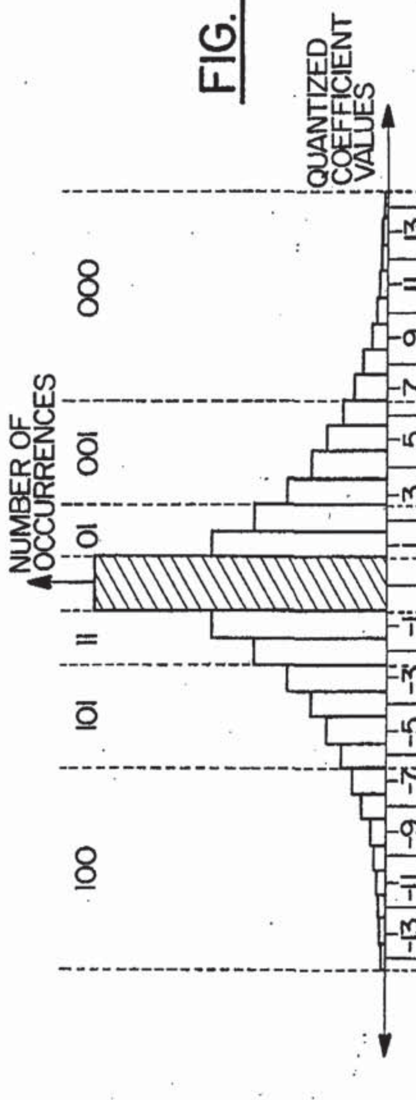


FIG. 5B.

PRINT OF DRAWINGS
AS ORIGINALLY FILED

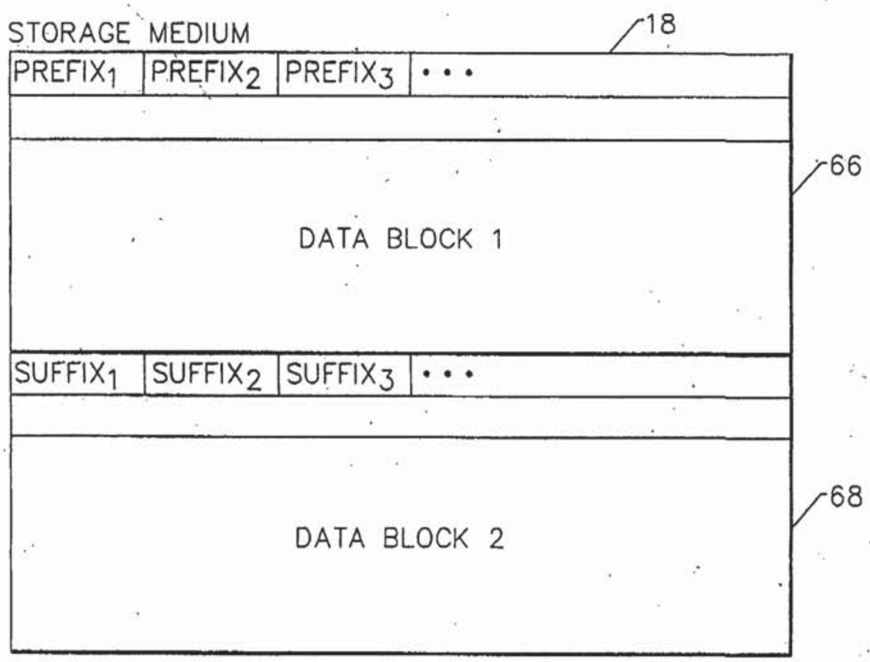


FIG. 6.

PRINT OF DRAWINGS
AS ORIGINALLY FILED

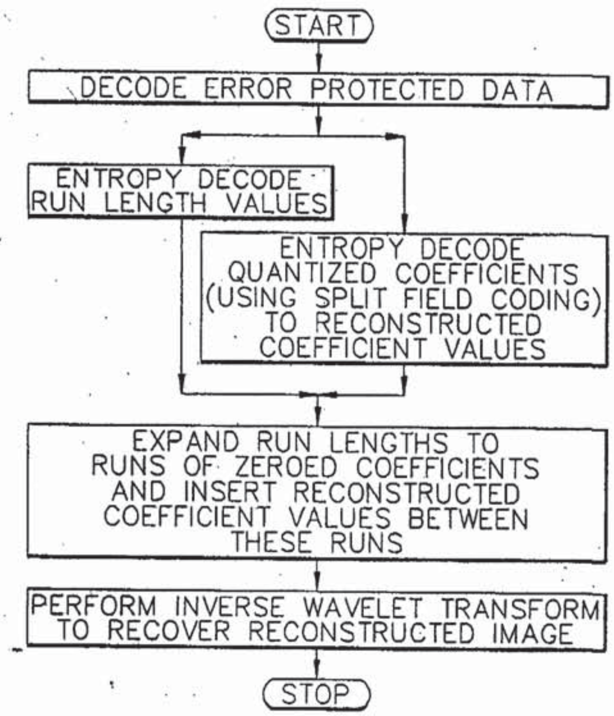
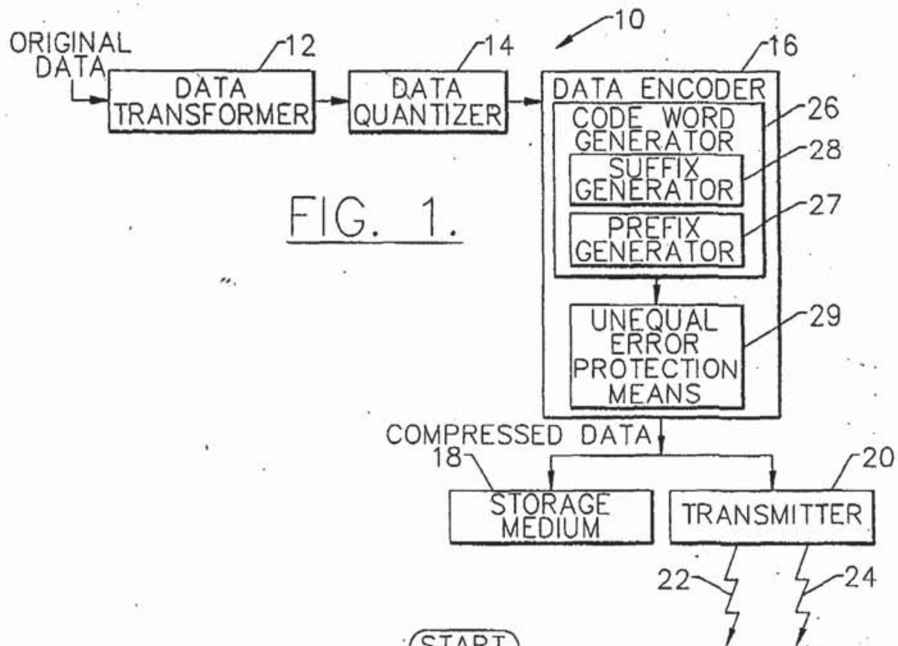
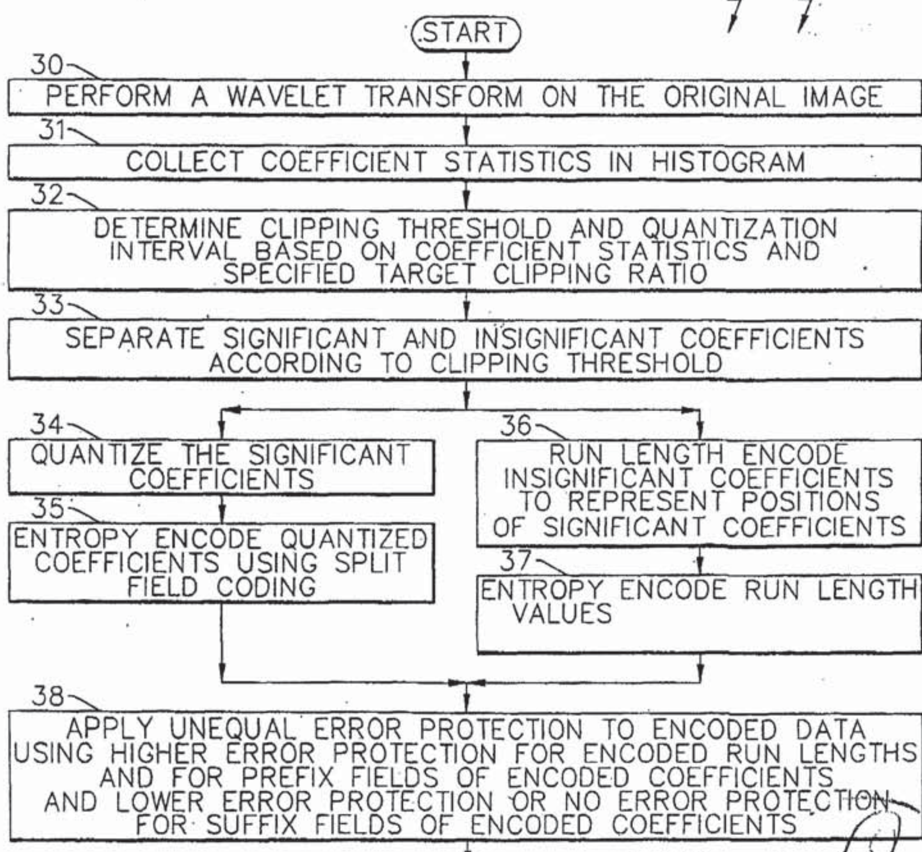


FIG. 7.



ORIGINAL DATA
12
DATA TRANSFORMER
14
DATA QUANTIZER
10
16
DATA ENCODER
CODE WORD GENERATOR
26
SUFFIX GENERATOR
28
PREFIX GENERATOR
27
UNEQUAL ERROR PROTECTION MEANS
29
COMPRESSED DATA
18
STORAGE MEDIUM
20
TRANSMITTER
22
24

FIG. 1.



START
30
PERFORM A WAVELET TRANSFORM ON THE ORIGINAL IMAGE
31
COLLECT COEFFICIENT STATISTICS IN HISTOGRAM
32
DETERMINE CLIPPING THRESHOLD AND QUANTIZATION INTERVAL BASED ON COEFFICIENT STATISTICS AND SPECIFIED TARGET CLIPPING RATIO
33
SEPARATE SIGNIFICANT AND INSIGNIFICANT COEFFICIENTS ACCORDING TO CLIPPING THRESHOLD
34
QUANTIZE THE SIGNIFICANT COEFFICIENTS
35
ENTROPY ENCODE QUANTIZED COEFFICIENTS USING SPLIT FIELD CODING
36
RUN LENGTH ENCODE INSIGNIFICANT COEFFICIENTS TO REPRESENT POSITIONS OF SIGNIFICANT COEFFICIENTS
37
ENTROPY ENCODE RUN LENGTH VALUES
38
APPLY UNEQUAL ERROR PROTECTION TO ENCODED DATA USING HIGHER ERROR PROTECTION FOR ENCODED RUN LENGTHS AND FOR PREFIX FIELDS OF ENCODED COEFFICIENTS AND LOWER ERROR PROTECTION OR NO ERROR PROTECTION FOR SUFFIX FIELDS OF ENCODED COEFFICIENTS
STOP

FIG. 2.

(8)

08/633896

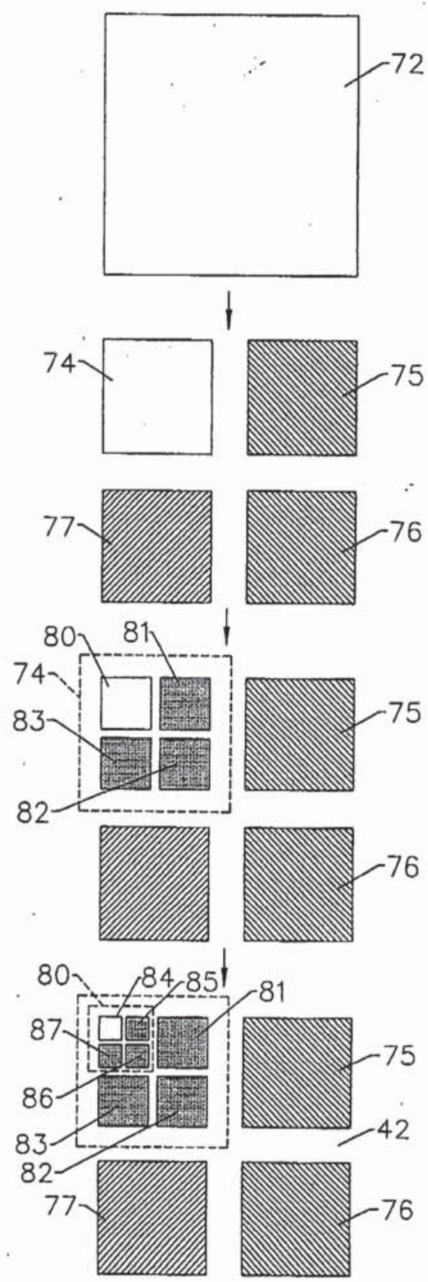


FIG. 3.

08/633896

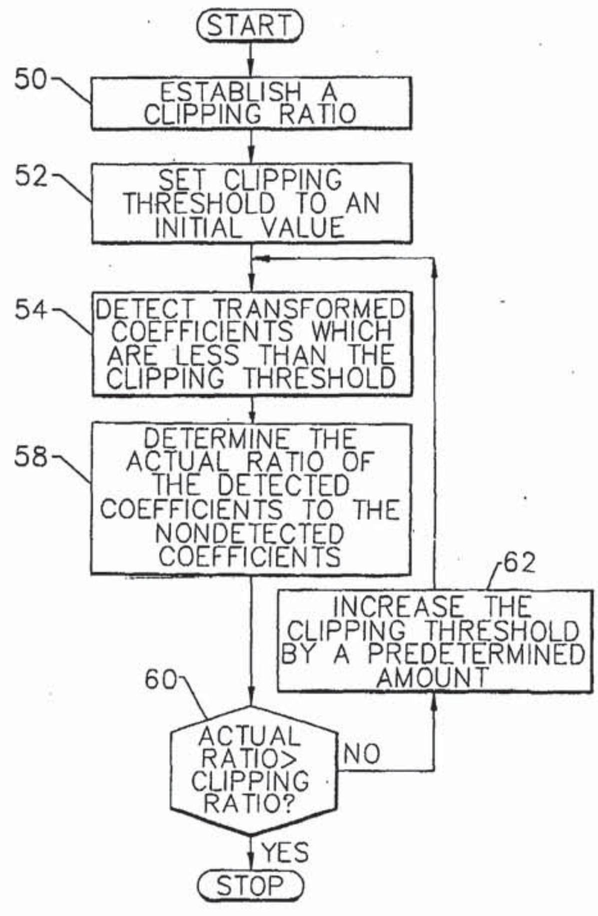


FIG. 4.

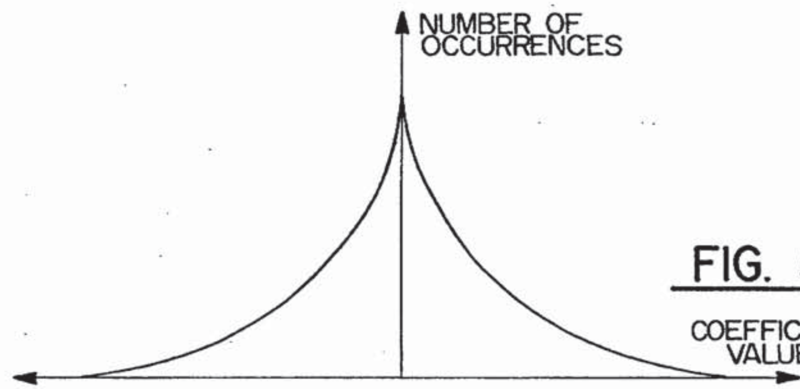


FIG. 5A.

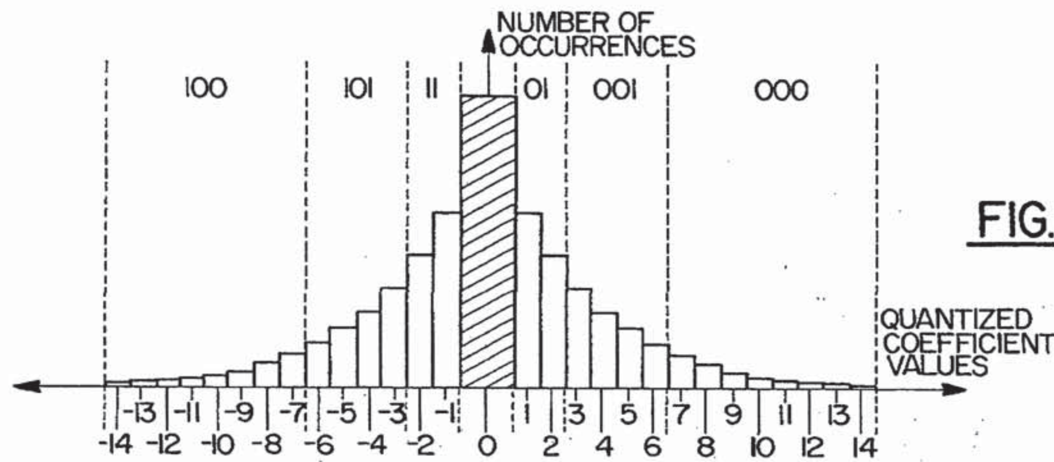


FIG. 5B.

08/633896

File History Content Report

The following content is missing from the original file history record obtained from the United States Patent and Trademark Office. No additional information is available.

Document Date - 1996-04-17

Document Title - Drawings

Page(s) - FIG 6

08/688896

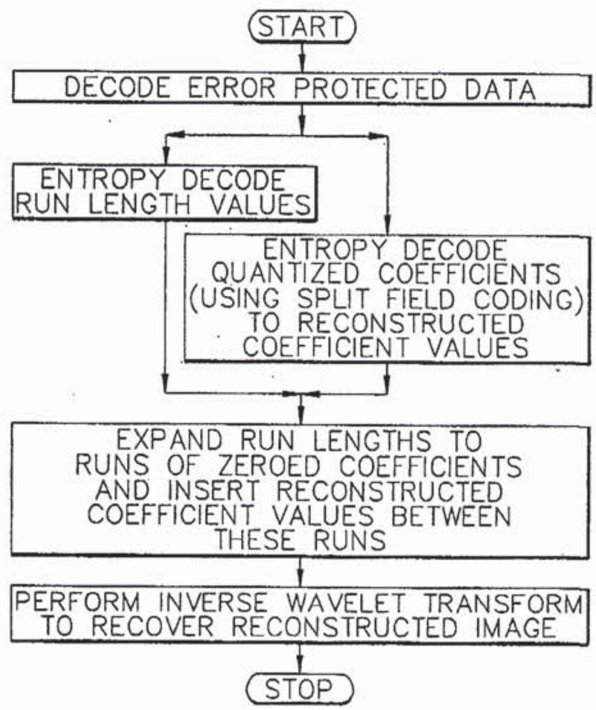


FIG. 7.

08/633896

MAIL ROOM APR 17 1996 PAT. & TRADEMARK OFF.

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

[...]s J. Meany and
Christopher J. Martens
Serial No. To Be Assigned
Filed: Concurrently Herewith
For: ERROR RESILIENT METHOD AND
APPARATUS FOR ENTHROPY CODING

April 17, 1996

Assistant Commissioner
of Patents and Trademarks
Washington, DC 20231

APPLICATION FILED UNDER 37 CFR 1.41(c)

Sir:

The above-identified application is being filed on behalf of the inventors, residents of Des Peres, St. Louis County, Missouri; and Creve Coeur, St. Louis County, Missouri, respectively, under the provisions of 37 CFR 1.41(c). A Declaration and Power of Attorney from the inventors will follow, 37 CFR 1.63.

Respectfully submitted,

Guy R. Gosnell
Registration No. 34,610

Bell, Seltzer, Park & Gibson
Post Office Drawer 34009
Charlotte, North Carolina 28234
Telephone: (704) 331-6000
Our File No. 8190-43
204780

"Express Mail" mailing label number TB877930020US

Date of Deposit April 17, 1996

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Commissioner of Patents and Trademarks, Washington, DC 20231.

Gwen Frickhoeffer

BEST COPY

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

APPLICATION NUMBER	FILING DATE	FIRST NAMED APPLICANT	ATTY. DOCKET NO./TITLE
08/633,896	04/17/96	MEANY	J 8190-43

#2

0252/0606

BELL SELTZER PARK AND GIBSON
POST OFFICE DRAWER 34009
CHARLOTTE NC 28234

0000

DATE MAILED: 06/06/96

NOTICE TO FILE MISSING PARTS OF APPLICATION
FILING DATE GRANTED

An Application Number and Filing Date have been assigned to this application. However, the items indicated below are missing. The required items and fees identified below must be timely submitted **ALONG WITH THE PAYMENT OF A SURCHARGE** for items 1 and 3-6 only of $130 — for large entities or $65 — for small entities who have filed a verified statement claiming such status. The surcharge is set forth in 37 CFR 1.16(e).

If all required items on this form are filed within the period set below, the total amount owed by applicant as a ☑ large entity, ☐ small entity (verified statement filed), is $130 —.

Applicant is given **ONE MONTH FROM THE DATE OF THIS LETTER, OR TWO MONTHS FROM THE FILING DATE** of this application, **WHICHEVER IS LATER,** within which to file all required items and pay any fees required above to avoid abandonment. Extensions of time may be obtained by filing a petition accompanied by the extension fee under the provisions of 37 CFR 1.136(a).

1. ☐ The statutory basic filing fee is: ☐ missing ☐ insufficient. Applicant as a ☐ large entity ☐ small entity, must submit $__________ to complete the basic filing fee.

2. ☐ Additional claim fees of $__________ as a ☐ large entity, ☐ small entity, including any required multiple dependent claim fee, are required. Applicant must submit the additional claim fees or cancel the additional claims for which fees are due.

3. ☑ The oath or declaration:
   ☑ is missing.
   ☐ does not cover items omitted at time of execution.

   An oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date is required.

4. ☐ The oath or declaration does not identify the application to which it applies. An oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date, is required.

5. ☐ The signature(s) to the oath or declaration is/are: ☐ missing; ☐ by a person other than the inventor or a person qualified under 37 CFR 1.42, 1.43, or 1.47. A properly signed oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date, is required.

6. ☐ The signature of the following joint inventor(s) is missing from the oath or declaration:

   __________ An oath or declaration listing the names of all inventors and signed by the omitted inventor(s), identifying this application by the above Application Number and Filing Date, is required.

7. ☐ The application was filed in a language other than English. Applicant must file a verified English translation of the application and a fee of $__________ under 37 CFR 1.17(k), unless this fee has already been paid.

8. ☐ A $__________ processing fee is required since your check was returned without payment. (37 CFR 1.21(m)).

9. ☐ Your filing receipt was mailed in error because your check was returned without payment.

10. ☐ The application does not comply with the Sequence Rules. See attached Notice to Comply with Sequence Rules 37 CFR 1.821-1.825.

11. ☐ Other.

Customer Service

Direct the response and any questions about this notice to, Attention: Application Processing Division, Special Processing and Correspondence Branch (703) 308-1202.

A copy of this notice MUST be returned with the response.

MAIL ROOM
24 JUL 8 1996
PAT. & TRADEMARK OFF.

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Meany and Martens
Serial No. 08/633,896
Filed: April 17, 1996
For: ERROR RESILIENT METHOD
AND APPARATUS FOR
ENTHROPY CODING

July 3, 1996

Assistant Commissioner
of Patents and Trademarks
Washington, DC 20231

SUBMITTAL OF DECLARATION UNDER 37 CFR 1.63

Sir:

Enclosed is a Declaration and Power of Attorney for the above-identified application which has been executed by the named inventors.

A check in the amount of $130 is also enclosed to cover the surcharge under 36 CFR 1.16(e). Any additional fee or credit may be charged to our deposit account No. 16-0605.

Respectfully submitted,

Guy R. Gosnell
Registration No. 34,610

Bell, Seltzer, Park & Gibson
Post Office Drawer 34009
Charlotte, North Carolina 28234
Telephone: (704) 331-6000
Our File No. 8190-43
Order No. 14879

232684

I hereby certify that this correspondence is being deposited with the United States Postal Service by first class mail in an envelope addressed to: Assistant Commissioner of Patents and Trademarks, Washington, D.C. 20231, on

July 3, 1996 by: Gwen Flickhoffer Signed: July 3, 1996

(3)

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

Attorney Docket No: 8190-43

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name,

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled ERROR RESILIENT METHOD AND APPARATUS FOR ENTROPY CODING, the specification of which

[ ] is attached hereto.

[ ] was filed on April 17, 1996 as

Application Serial No. 08/633,896

and was amended on ______________________
(if applicable)

I hereby state that I have reviewed and understand the contents of the above-identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the examination of this application in accordance with Title 37, Code of Federal Regulations, Section 1.56(a).

I hereby claim foreign priority benefits under Title 35, United States Code, Section 119 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

Prior Foreign Application(s) — Priority Claimed

(Number)	(Country)	(Day/Month/Year Filed)	Yes	No
(None)			[ ]	[ ]
			[ ]	[ ]
			[ ]	[ ]

I hereby claim the benefit under Title 35, United States Code, § 119(e) of any United States provisional application(s) listed below.

(None)	
(Application Number)	(Filing Date)
(Application Number)	(Filing Date)

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code, Section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, Section 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

(None)		
(Appln. Serial No.)	(Filing Date)	(Status) (patented/pending/aban.)
(Appln. Serial No.)	(Filing Date)	(Status) (patented/pending/aban.)

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

POWER OF ATTORNEY: As a named inventor, I hereby appoint the following attorneys to prosecute this application and transact all business in the Patent and Trademark Office connected therewith.

Timothy H. Courson
Registration No. 33,300

Charles B. Elderkin
Registration No. 24,357

Mitchell S. Bigel
Registration No. 29,614

Guy R. Gosnell
Registration No. 34,610

(4)

Send correspondence to: Guy R. Gosnell
Bell, Seltzer, Park & Gibson
Post Office Drawer 34009
Charlotte, North Carolina 28234

Direct telephone calls to: Guy R. Gosnell
(704) 331-6000

Full name of first inventor: James J. Meany

1-00 Inventor's Signature: James J. Meany Date: 4/30/96

Residence: Des Peres, Missouri MO

Citizenship: United States of America

Post Office Address: 2782 Diamond Ridge Lane
Des Peres, MO, USA 63131

Full name of second inventor: Christopher J. Martens

2-00 Inventor's Signature: Chris Martens Date: 4-30-96

Residence: Creve Coeur, Missouri MO

Citizenship: United States of America

Post Office Address: 1168 Town & Four
Creve Coeur, MO, USA 63141

223625

130/105

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office
Address: COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

MAIL ROOM JUL 8 1996 PAT. & TRADEMARK OFF.

APPLICATION NUMBER	FILING DATE	FIRST NAMED APPLICANT	ATTY. DOCKET NO./TITLE
08/633,896	04/17/96	MEANY J	8190-43

0252/0606

BELL, SELTZER PARK AND GIBSON
POST OFFICE DRAWER 34009
CHARLOTTE NC 28234

0000
DATE MAILED: 06/06/96

NOTICE TO FILE MISSING PARTS OF APPLICATION
FILING DATE GRANTED

An Application Number and Filing Date have been assigned to this application. However, the items indicated below are missing. The required items and fees identified below must be timely submitted **ALONG WITH THE PAYMENT OF A SURCHARGE** for items 1 and 3-6 only of $130 for large entities or $65 for small entities who have filed a verified statement claiming such status. The surcharge is set forth in 37 CFR 1.16(e).

If all required items on this form are filed within the period set below, the total amount owed by applicant as a ☑ large entity, ☐ small entity (verified statement filed), is $130.

Applicant is given **ONE MONTH FROM THE DATE OF THIS LETTER, OR TWO MONTHS FROM THE FILING DATE** of this application, **WHICHEVER IS LATER,** within which to file all required items and pay any fees required above to avoid abandonment. Extensions of time may be obtained by filing a petition accompanied by the extension fee under the provisions of 37 CFR 1.136(a).

1. ☐ The statutory basic filing fee is: ☐ missing ☐ insufficient. Applicant as a ☐ large entity ☐ small entity, must submit $______ to complete the basic filing fee.

2. ☐ Additional claim fees of $______ as a ☐ large entity, ☐ small entity, including any required multiple dependent claim fee, are required. Applicant must submit the additional claim fees or cancel the additional claims for which fees are due.

3. ☑ The oath or declaration:
   ☑ is missing.
   ☐ does not cover items omitted at time of execution.

   An oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date is required.

4. ☐ The oath or declaration does not identify the application to which it applies. An oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date, is required.

5. ☐ The signature(s) to the oath or declaration is/are: ☐ missing; ☐ by a person other than the inventor or a person qualified under 37 CFR 1.42, 1.43, or 1.47. A properly signed oath or declaration in compliance with 37 CFR 1.63, identifying the application by the above Application Number and Filing Date, is required.

6. ☐ The signature of the following joint inventor(s) is missing from the oath or declaration:

   ______ An oath or declaration listing the names of all inventors and signed by the omitted inventor(s), identifying this application by the above Application Number and Filing Date, is required.

7. ☐ The application was filed in a language other than English. Applicant must file a verified English translation of the application and a fee of $______ under 37 CFR 1.17(k), unless this fee has already been paid.

8. ☐ A $______ processing fee is required since your check was returned without payment. (37 CFR 1.21(m)).

9. ☐ Your filing receipt was mailed in error because your check was returned without payment.

10. ☐ The application does not comply with the Sequence Rules. See attached Notice to Comply with Sequence Rules 37 CFR 1.821-1.825.

340 UT 07/18/96 08633896
1 105 130.00 CK

11. ☐ Other.

[signature]

Direct the response and any questions about this notice to, Attention: Application Processing Division, Special Processing and Correspondence Branch (703) 308-1202.

A copy of this notice MUST be returned with the response.

MAIL ROOM 24 JUL 8 1996 PAT. & TRADEMARK OFFICE

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

#4
JT
8-26-96

Meany and Martens
Serial No. 08/633,896
Filed: April 17, 1996
For: ERROR RESILIENT METHOD AND APPARATUS FOR ENTHROPY CODING

Group Art Unit: 2616

Examiner:

July 3, 1996

Assistant Commissioner of
Patents and Trademarks
Washington, DC 20231

INFORMATION DISCLOSURE STATEMENT UNDER 37 CFR 1.97

Sir:

Attached is a form PTO-1449 listing several documents which may be considered material to the examination of the above-identified application. A copy of each document is also enclosed. It is requested that these documents be considered by the Examiner and officially made of record in accordance with the provisions of 37 CFR 1.97 and Section 609 of the MPEP.

Respectfully submitted,

Guy R. Gosnell
Registration No. 34,610

Bell, Seltzer, Park & Gibson
Post Office Drawer 34009
Charlotte, North Carolina 28234
Telephone: (704) 331-6000
Our File No. 8190-43

232678

I hereby certify that this correspondence is being deposited with the United States Postal Service by first class mail in an envelope addressed to: Commissioner of Patents and Trademarks, Washington, D.C. 20231, on

July 3, 1996 by: Gwen Frickhoeffer Signed: July 3, 1996

Sheet 1 of 2

FORM PTO-1449 U.S. Department of Commerce Patent and Trademark Office LIST OF DOCUMENTS CITED BY APPLICANT (Use several sheets if necessary)	Attorney Docket Number 8190-43	Serial Number 08/633,896
	Applicant: Meany and Martens	
	Filing Date: April 17, 1996	Group 2616

MAIL ROOM JUL 8 1996 PAT. & TRADEMARK OFFICE

U. S. PATENT DOCUMENTS

Examiner Initial		Document Number	Date	Name	Class	Subclass	Filing Date if Appropriate
BT	A	4,817,182	03/1989	Adelson, et al.	382	56	MAY87
BT	B	5,014,134	05/1991	Lawton, et al.	358	261.3	SEP89
BT	C	5,101,446	03/1992	Resnikoff, et al.	382	56	MAY90
BT	D	5,315,670	05/1994	Shapiro	382	56	NOV91
BT	E	5,321,776	06/1994	Shapiro	382	56	FEB92
	F						
	G						
	H						
	I						
	J						
	K						

FOREIGN PATENT DOCUMENTS

		Document Number	Date	Country	Class	Subclass	Translation Yes | No
	L						
	M						
	N						
	O						
	P						

OTHER DOCUMENTS (Including Author, Title, Date, Pertinent Pages, Etc.)

BT	Q	Edward R. Fiala and Daniel H. Greene, *Data Compression with Finite Windows*, Communications of the ACM, Vol. 32, No. 4, April 1989, pp.490-505.
BT	R	Timothy C. Bell, John G. Cleary and Ian H. Witten, *Text Compression*, Prentice Hall, Copyright 1990, pp. 290-95.
BT	S	Imran Ali Shah, Olu Akiwumi-Assani and Brian Johnson, *A Chip Set for Lossless Image Compression*, IEEE Journal of Solid-State Circuits, Vol. 26, No. 3, March 1991, pp. 237-44.

EXAMINER	DATE CONSIDERED
DR. B. Taddyan	5 JULY 97

*EXAMINER Initial if reference considered, whether or not citation is in conformance with MPEP 609; Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

Sheet 2 of 2

FORM PTO-1449	U.S. Department of Commerce Patent and Trademark Office	Attorney Docket Number 8190-43	Serial Number 08/633,896
LIST OF DOCUMENTS CITED BY APPLICANT		Applicant: Meany and Martens	
(Use several sheets if necessary)		Filing Date: April 17, 1996	Group 2616

MAIL ROOM 24 JUL 1996 PAT. & TRADEMARK OFF.

		OTHER DOCUMENTS (Including Author, Title, Date, Pertinent Pages, Etc.)
BT	T	Gregory K. Wallace, *The JPEG Still Picture Compression Standard*, Communications of the ACM, Vol. 34, No. 4, April 1991, pp. 30-45.
BT	U	Olivier Rioul and Martin Vetterli, *Wavelets and Signal Processing*, IEEE SP Magazine, October 1991, pp. 14-38.
BT	V	Albert Cohen, *Biorthogonal Wavelets*, Wavelets-A Tutorial in Theory and Applications, Copyright 1992, pp. 123-52.
BT	W	A. S. Lewis and G. Knowles, *Image Compression Using the 2-D Wavelet*, IEEE Transactions on Image Processing, Vol. 1, No. 2, April 1992, pp. 244-50.
BT	X	Mladen Victor Wickerhauser, *High-Resolution Still Picture Compression*, April 19, 1992, pp. 1-33. (NO TITLE OF PUBLICATION).
BT	Y	Marc Antonini, Michel Barlaud, Pierre Mathieu and Ingrid Daubechies, *Image Coding Using Wavelet Transform*, IEEE Transactions on Image Processing, Vol. 1, No. 2, April 1992, pp. 205-20.
BT	Z	Naoto Tanabe and Nariman Farvardin, *Subband Image Coding Using Entropy-Coded Quantization over Noisy Channels*, IEEE Journal on Selected Areas in Communications, Vol. 10, No. 5, June 1992, pp. 926-43.
BT	AA	Amir Said and William A. Pearlman, *Reversible image compression via multiresolution representation and predictive coding*, SPIE, Vol. 2094, 1993, pp. 664-74.
BT	AB	Jerome M. Shapiro, *Embedded Image Coding Using Zerotrees of Wavelet Coefficients*, IEEE Transactions on Signal Processing, Vol. 41, No. 12, December 1993, pp. 3445-62.
BT	AC	Michael L. Hilton, Björn D. Jawerth and Ayan Sengupta, *Compressing Still and Moving Images with Wavelets*, Multimedia Systems, Vol. 2, No. 3, April 18, 1994, pp. 1-20.
BT	AD	Keith A. Birney and Thomas R. Fischer, *On the Modeling of DCT and Subband Image Data for Compression*, IEEE Transactions on Image Processing, Vol. 4, No. 2, February 1995, pp. 186-93.
BT	AE	Parthasarathy Sriram and Michael W. Marcellin, *Image Coding Using Wavelet Transforms and Entropy-Constrained Trellis-Coded Quantization*, IEEE Transactions on Image Processing, Vol. 4, No. 6, June 1995, pp. 725-33.
BT	AF	Björn Jawerth and Wim Sweldens, *An Overview Of Wavelet Based Multiresolution Analyses*, pp. 1-39. (NO TITLE OF PUBLICATION) (NO DATE OF PUBLICATION)
BT	AG	Wim Sweldens, *The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Constructions*. (NO PLACE OR DATE OF PUBLICATION & NO PAGE #.)
BT	AH	Ahmad Zandi, James D. Allen, Edward L. Schwartz and Martin Boliek, *CREW: Compression with Reversible Embedded Wavelets*. (NO PLACE OR DATE) & (NO PAGE #.)
	AI	
	AJ	

EXAMINER Dr B. Tadayon	DATE CONSIDERED 5 JULY 97

*EXAMINER Initial if reference considered, whether or not citation is in conformance with MPEP 609; Draw line through citation if not in conference and not considered. Include copy of this form with next communication to applicant. 232651

Algorithms and Data Structures

Daniel Sleator Editor

Data Compression with Finite Windows

EDWARD R. FIALA and DANIEL H. GREENE

ABSTRACT: *Several methods are presented for adaptive, invertible data compression in the style of Lempel's and Ziv's first textual substitution proposal. For the first two methods, the article describes modifications of McCreight's suffix tree data structure that support cyclic maintenance of a window on the most recent source characters. A percolating update is used to keep node positions within the window, and the updating process is shown to have constant amortized cost. Other methods explore the tradeoffs between compression time, expansion time, data structure size, and amount of compression achieved. The article includes a graph-theoretic analysis of the compression penalty incurred by our codeword selection policy in comparison with an optimal policy, and it includes empirical studies of the performance of various adaptive compressors from the literature.*

1. INTRODUCTION

Compression is the coding of data to minimize its representation. In this article, we are concerned with fast, one-pass, adaptive, invertible (or lossless) methods of digital compression which have reasonable memory requirements. Such methods can be used, for example, to reduce the storage requirements for files, to increase the communication rate over a channel, or to reduce redundancy prior to encryption for greater security.

By "adaptive" we mean that a compression method should be widely applicable to different kinds of source data. Ideally, it should adapt rapidly to the source to achieve significant compression on small files, and it should adapt to any subsequent internal changes in the nature of the source. In addition, it should achieve very high compression asymptotically on large regions with stationary statistics.

All the compression methods developed in this article are *substitutional*. Typically, a substitutional compressor functions by replacing large blocks of text with shorter references to earlier occurrences of identical text [3, 5, 29, 34, 36, 39, 41–43]. (This is often called Ziv-Lempel compression, in recognition of their pioneering ideas. Ziv and Lempel, in fact, proposed two methods. The unqualified use of the phrase "Ziv-Lempel compression" usually refers to their second proposal [43]. In this article, we will be primarily concerned with their first proposal [42].) A popular alternative to a substitutional compressor is a statistical compressor. A symbolwise statistical compressor functions by accurately predicting the probability of individual symbols, and then encoding these symbols with space close to $-\log_2$ of the predicted probabilities. The encoding is accomplished with either Huffman compression [17] which has recently been made one-pass and adaptive [11, 22, 37], or with arithmetic coding, as described in [1, 14, 20, 25, 26, 31–33]. The major challenge of a statistical compressor is to predict the symbol probabilities. Simple strategies, such as keeping zero-order (single symbol) or first-order (symbol pair) statistics of the input, do not compress English text very well. Several authors have had success gathering higher-order statistics, but this necessarily involves higher memory costs and additional mechanisms for dealing with situations where higher-order statistics are not available [6, 7, 26].

It is hard to give a rigorous foundation to the substitutional vs. statistical distinction described above. Several authors have observed that statistical methods can be used to simulate textual substitution, suggesting that the statistical category includes the substitutional category [4, 24]. However, this takes no account of the simplicity of mechanism; the virtue of textual substitution is that it recognizes and removes coherence on a large scale, oftentimes ignoring the smaller scale statistics. As a result, most textual substitution compressors process their compressed representation in larger blocks than their statistical counterparts, thereby gaining a significant speed advantage. It was previously believed that the speed gained by textual substitution would necessarily cost something in compression achieved. We were surprised to discover that with careful attention to coding, textual substitution compressors can match the compression performance of the best statistical methods.

Consider the following scheme, which we will improve later in the article. Compressed files contain two types of codewords:

literal x	pass the next x characters directly into the uncompressed output
copy $x, -y$	go back y characters in the output and copy x characters forward to the current position.

So, for example, the following piece of literature:

```
IT WAS THE BEST OF TIMES,
                IT WAS THE WORST OF TIMES
```

would compress to

```
(literal 26)IT WAS THE BEST OF TIMES,
  (copy 11-26)(literal 3)WOR(copy 11-27)
```

The compression achieved depends on the space required for the copy and literal codewords. Our simplest scheme, hereafter denoted A1, uses 8 bits for a literal codeword and 16 for a copy codeword. If the first 4 bits are 0, then the codeword is a literal; the next 4 bits encode a length x in the range $[1 \ldots 16]$ and the following x characters are literal (one byte per character). Otherwise, the codeword is a copy; the first 4 bits encode a length x in the range $[2 \ldots 16]$ and the next 12 bits are a displacement y in the range $[1 \ldots 4096]$. At each step, the policy by which the compressor chooses between a literal and a copy is as follows: If the compressor is idle (just finished a copy, or terminated a literal because of the 16-character limit), then the longest copy of length 2 or more is issued; otherwise, if the longest copy is less than 2 long, a literal is started. Once started, a literal is extended across subsequent characters until a copy of length 3 or more can be issued or until the length limit is reached.

A1 would break the first literal in the above example into two literals and compress the source from 51 bytes down to 36. A1 is close to Ziv and Lempel's first textual substitution proposal [42]. One difference is that A1 uses a separate literal codeword, while Ziv and Lempel combine each copy codeword with a single literal character. We have found it useful to have longer literals during the startup transient; after the startup, it is better to have no literals consuming space in the copy codewords.

Our empirical studies showed that, for source code and English text, the field size choices for A1 are good; reducing the size of the literal length field by 1 bit increases compression slightly but gives up the byte-alignment property of the A1 codewords. In short, if one desires a simple method based upon the copy and literal idea, A1 is a good choice.

A1 was designed for 8-bit per character text or program sources, but, as we will see shortly, it achieves good compression on other kinds of source data, such as compiled code and images, where the word model does not match the source data particularly well, or where no model of the source is easily perceived. A1 is, in fact, an excellent approach to general purpose data compression. In the remainder of this article, we will study A1 and several more powerful variations.

2. OVERVIEW OF THE DATA STRUCTURE

The fixed window suffix tree of this article is a modification of McCreight's suffix tree [28] (see also [21, 34, 38]), which is itself a modification of Morrison's PATRICIA tree [30], and Morrison's tree is ultimately based on a Trie data structure [22, page 481]. We will review each of these data structures briefly.

A Trie is a tree structure where the branching occurs according to "digits" of the keys, rather than according to comparisons of the keys. In English, for example, the most natural "digits" are individual letters, with the lth level of the tree branching according to the lth letter of the words in the tree.

In Figure 1, many internal nodes are superfluous, having only one descendant. If we are building an index for a file, we can save space by eliminating the superfluous nodes and putting pointers to the file into the nodes rather than including characters in the data structure. In Figure 2, the characters in parentheses are not actually represented in the data structure, but they can be recovered from the (position, level) pairs in the nodes. Figure 2 also shows a suffix pointer (as a dark right arrow) that will be explained later.

Figure 2 represents some, but not all, of the innovations in Morrison's PATRICIA trees. He builds the trees with binary "digits" rather than full characters, and this allows him to save more space by folding the leaves into the internal nodes. Our "digits" are bytes, so the branching factor can be as large as 256. Since there are rarely 256 descendants of a node, we do not reserve that much space in each node, but instead hash the arcs. There is also a question about when the strings in parentheses are checked in the searching process. In what follows, we usually check characters immediately when we cross an arc. Morrison's scheme can avoid file access by skipping the characters on the arcs, and doing only one file access and comparison at the end of the search. However, our files will be in main memory, so this consideration is unimportant. We will use the simplified tree depicted in Figure 2.

For A1, we wish to find the longest (up to 16 character) match to the current string beginning anywhere in the preceding 4096 positions. If all preceding 4096 strings were stored in a PATRICIA tree with depth $d = 16$, then finding this match would be straightforward. Unfortunately, the cost of inserting these strings can be prohibitive, for if we have just descended d levels in the tree to insert the string starting at position i then we will descend at least $d - 1$ levels inserting the string at $i + 1$. In the worst case this can lead to $O(nd)$

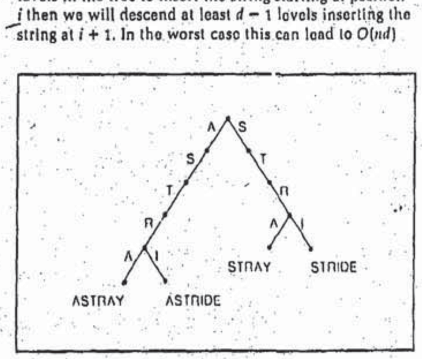


FIGURE 1. A Trie

insertion time for a file of size n. Since later encodings will use much larger values for d than 16, it is important to eliminate d from the running time.

To insert the strings in $O(n)$ time, McCreight added additional suffix pointers to the tree. Each internal node, representing the string aX on the path from the root to the internal node, has a pointer to the node representing X, the string obtained by stripping a single letter from the beginning of aX. If a string starting at i has just been inserted at level d we do not need to return to the root to insert the string at $i + 1$; instead, a nearby suffix pointer will lead us to the relevant branch of the tree.

Figure 3 shows how suffix links are created and used. On the previous iteration, we have matched the string aXY, where a is a single character, X and Y are strings, and b is the first unmatched character after Y. Figure 3 shows a complicated case where a new internal node, α, has been added to the tree, and the suffix link of α must be computed. We insert the next string XYb by going up the tree to node β, representing the string aX, and crossing its suffix link to γ, representing X. Once we have crossed the suffix link, we descend again in the tree, first by "rescanning" the string Y, and then by "scanning" from δ until the new string is inserted. The first part is called "rescanning" because it covers a portion of the string that was covered by the previous insert, and so it does not require checking the internal strings on the arcs. (In fact, avoiding these checks is essential to the linear time functioning of the algorithm.) The rescan either ends at an existing node δ, or δ is created to insert the new string XYb; either way we have the destination for the suffix link of α. We have restored the invariant that every internal node, except possibly the one just created, has a suffix link.

For the A1 compressor, with a 4096-byte fixed window, we need a way to delete and reclaim the storage for portions of the suffix tree representing strings further back than 4096 in the file. Several things must be added to the suffix tree data structure. The leaves of the tree are placed in a circular buffer, so that the oldest leaf can be identified and reclaimed, and the internal nodes are given "son count" fields. When an

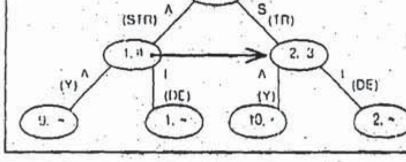


FIGURE 2. A PATRICIA Tree with a Suffix Pointer

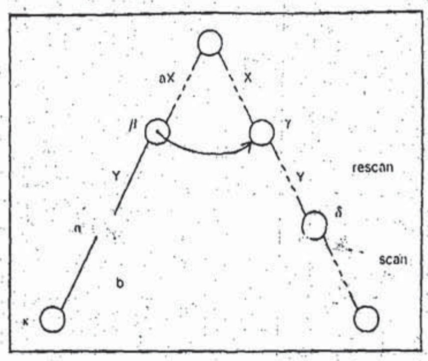


FIGURE 3. Building a Suffix Tree

internal "son count" falls to one, the node is deleted and two consecutive arcs are combined. In Section 3, it is shown that this approach will never leave a "dangling" suffix link pointing to deleted nodes. Unfortunately, this is not the only problem in maintaining a valid suffix tree. The modifications that avoided a return to the root for each new insertion create havoc for deletions. Since we have not always returned to the root, we may have consistently entered a branch of the tree sideways. The pointers (to strings in the 4096-byte window) in the higher levels of such a branch can become out-of-date. However, traversing the branch and updating the pointers would destroy any advantage gained by using the suffix links.

We can keep valid pointers and avoid extensive updating by partially updating according to a percolating update. Each internal node has a single "update" bit. If the update bit is true when we are updating a node, then we set the bit false and propagate the update recursively to the node's parent. Otherwise, we set the bit true and stop the propagation. In the worst case, a long string of true updates can cause the update to propagate to the root. However, when amortized over all new leaves, the cost of updating is constant, and the effect of updating is to keep all internal pointers on positions within the last 4096 positions of the file. These facts will be shown in Section 3.

We can now summarize the operation of the inner loop, using Figure 3 again. If we have just created node α, then we use α's parent's suffix link to find γ. From γ we move down in the tree, first rescanning, and then scanning. At the end of the scan, we percolate an update from the leaf, moving toward the root, setting the position fields equal to the current position, and setting the update bits false, until we find a node with an update bit that is already false, whereupon we set that node's update bit true and stop the percolation. Finally, we go to the circular buffer of leaves and replace the oldest leaf with the new leaf. If the oldest leaf's parent has only one remaining son, then it must also be deleted; in this case, the remaining son is attached to its grandparent, and the deleted node's position is percolated upward as before, only at each step the position being percolated and the position already in the node must be compared and the more recent of these sent upward in the tree.

3. THEORETICAL CONSIDERATIONS

The correctness and linearity of suffix tree construction follows from McCreight's original paper [28]. Here we will concern ourselves with the correctness and the linearity of suffix tree destruction—questions raised in Section 2.

THEOREM 1. *Deleting leaves in FIFO order and deleting internal nodes with single sons will never leave dangling suffix pointers.*

PROOF. Assume the contrary. We have a node α with a suffix pointer to a node δ that has just been deleted. The existence of α means that there are at least two strings that agree for l positions and then differ at $l + 1$. Assuming that these two strings start at positions i and j, where both i and j are within the window of recently scanned strings and are not equal to the current position, then there are two even younger strings at $i + 1$ and $j + 1$ that differ first at l. This contradicts the assumption that δ has one son. (If either i or j are equal to the current position, then α is a new node, and can temporarily be without a suffix pointer.)

There are two issues related to the percolating update: its cost and its effectiveness.

THEOREM 2. *Each percolated update has constant amortized cost.*

PROOF. We assume that the data structure contains a "credit" on each internal node where the "update" flag is true. A new leaf can be added with two "credits." One is spent immediately to update the parent, and the other is combined with any credits remaining at the parent to either: 1) obtain one credit to leave at the parent and terminate the algorithm or 2) obtain two credits to apply the algorithm recursively at the parent. This gives an amortized cost of two updates for each new leaf.

For the next theorem, define the "span" of a suffix tree to be equal to the size of its fixed window. So far we have used examples with "span" equal to 4096, but the value is flexible.

THEOREM 3. *Using the percolating update, every internal node will be updated at least once during every period of length "span."*

PROOF. It is useful to prove the slightly stronger result that every internal node (that remains for an entire period) will be updated twice during a period, and thus propagate at least one update to its parent. To show a contradiction, we find the earliest period and the node β farthest from the root that does not propagate an update to its parent. If β has at least two children that have remained for the entire period, then β must have received updates from these nodes: they are farther from the root. If β has only one remaining child, then it must have a new child, and so it will still get two updates. (Every newly created arc causes a son to update a parent, percolating if necessary.) Similarly, two new children also cause two updates. By every accounting, β will receive two updates during the period, and thus propagate an update—contradicting our assumption of β's failure to update its parent.

There is some flexibility on how updating is handled. We could propagate the current position upward before rescanning, and then write the current position into those nodes passed during the rescan and scan; in this case, the proof of Theorem 3 is conservative. Alternatively, a similar, symmetric proof can be used to show that updating can be omitted when new arcs are added so long as we propagate an update after every arc is deleted. The choice is primarily a matter of implementation convenience, although the method used above is slightly faster.

The last major theoretical consideration is the effectiveness of the A1 policy in choosing between literal and copy codewords. We have chosen the following one-pass policy for A1: When the encoder is idle, issue a copy if it is possible to copy two or more characters; otherwise, start a literal. If the encoder has previously started a literal, then terminate the literal and issue a copy only if the copy is of length three or greater.

Notice that this policy can sometimes go astray. For example, suppose that the compressor is idle at position i and has the following copy lengths available at subsequent positions:

$$\begin{array}{cccccc} i & i+1 & i+2 & i+3 & i+4 & i+5 \\ 1 & 3 & 16 & 15 & 14 & 13 \end{array} \qquad (1)$$

Under the policy, the compressor encodes position i with a literal codeword, then takes the copy of length 3, and finally takes a copy of length 14 at position $i + 4$. It uses 6 bytes in the encoding:

(literal 1)X(copy 3 − y)(copy 14 − y)

If the compressor had foresight it could avoid the copy of length 3, compressing the same material into 5 bytes:

(literal 2)XX(copy 16 − y)

The optimal solution can be computed by dynamic programming [36]. One forward pass records the length of the longest possible copy at each position (as in equation 1) and the displacement for the copy (not shown in equation 1). A second backward pass computes the optimal way to finish compressing the file from each position by recording the best codeword to use and the length to the end-of-file. Finally, another forward pass reads off the solution and outputs the compressed file. However, one would probably never want to use dynamic programming since the one-pass heuristic is a lot faster, and we estimated for several typical files that the heuristically compressed output was only about 1 percent larger than the optimum. Furthermore, we will show in the remainder of this section that the size of the compressed file is never worse than $^5/_4$ the size of the optimal solution for the specific A1 encoding. This will require developing some analytic tools, so the non-mathematical reader should feel free to skip to Section 4.

The following definitions are useful:

Definition. *$F(i)$ is the longest feasible copy at position i in the file.*

Sample $F(i)$'s were given above in equation 1. They are dependent on the encoding used. For now, we are assuming that they are limited in magnitude to 16, and must correspond to copy sources within the last 4096 characters.

Definition. *$B(i)$ is the size of the best way to compress the remainder of the file, starting at position i.*

$B(i)$'s would be computed in the reverse pass of the optimal algorithm outlined above.

The following Theorems are given without proof:

THEOREM. $F(i + 1) \geq F(i) - 1$.

THEOREM. *There exists an optimal solution where copies are longest possible (i.e., only copies corresponding to $F(i)$'s are used).*

THEOREM. *$B(i)$ is monotone decreasing.*

THEOREM. *Any solution can be modified, without affecting length, so that (literal x_1) followed immediately by (literal x_2) implies that x_1 is maximum (in this case 16).*

We could continue to reason in this vein, but there is an abstract way of looking at the problem that is both clearer and more general. Suppose we have a non-deterministic finite automaton where each transition is given a cost. A simple example is shown in Figure 4. The machine accepts $(a + b)^*$, with costs as shown in parentheses.

The total cost of accepting a string is the sum of the transition costs for each character. (While it is not important to our problem, the optimal solution can be computed by forming a transition matrix for each letter, using the costs shown in parentheses, and then multiplying the matrices for a given string, treating the coefficients as elements of the closed semiring with operations of addition and minimization.) We can obtain a solution that approximates the minimum by deleting transitions in the original machine until it becomes a deterministic machine. This corresponds to choosing a policy in our original data compression problem. A policy for the machine in Figure 4 is shown in Figure 5.

We now wish to compare, in the worst case, the difference between optimally accepting a string with the non-deterministic machine, and deterministically accepting the same string with the "policy" machine. This is done by taking a cross product of the two machines, as shown in Figure 6.

In Figure 6 there are now two weights on each transition; the first is the cost in the non-deterministic graph, and the second is the cost in the policy graph. Asymptotically, the relationship of the optimal solution to the policy solution is dominated by the smallest ratio on a cycle in this graph. In the case of Figure 6, there is a cycle from 1, 1' to 1, 2' and back that has cost in the non-deterministic graph of $2 + 1 = 3$, and cost in the policy graph of $3 + 3 = 6$, giving a ratio of $^1/_2$. That is,

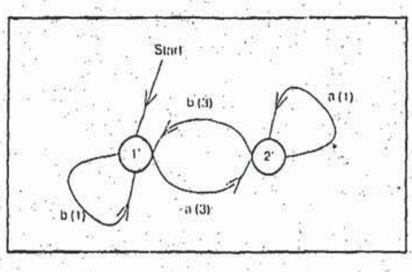


FIGURE 5. A Deterministic "Policy" Automaton for Figure 4

the policy solution can be twice as bad as the optimum on the string *abababab*. . . .

In general, we can find the cycle with the smallest ratio mechanically, using well known techniques [8, 27]. The idea is to conjecture a ratio r and then reduce the pairs of weights (x, y) on the arcs to single weights $x - ry$. Under this reduction, a cycle with zero weight has ratio exactly r. If a cycle has negative weight, then r is too large. The ratio on the negative cycle is used as a new conjecture, and the process is iterated. (Negative cycles are detected by running a shortest path algorithm and checking for convergence.) Once we have found the minimum ratio cycle, we can create a worst case string in the original automata problem by finding a path from the start state to the cycle and then repeating the cycle indefinitely. The ratio of the costs of ac-

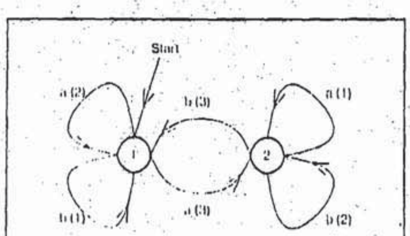


FIGURE 4. A Nondeterministic Automaton with Transition Costs

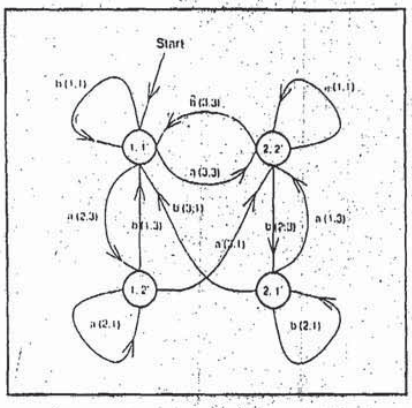


FIGURE 6. The Cross Product

copying the string non-deterministically and deterministically will converge to the ratio of the cycle. (The path taken in the cross product graph will not necessarily bring us to the same cycle, due to the initial path fragment; we will, nevertheless, do at least as well.) Conversely, if we have a sufficiently long string with non-deterministic to deterministic ratio r, then the string will eventually loop in the cross product graph. If we remove loops with ratio greater than r we only improve the ratio of the string, so we must eventually find a loop with ratio at least as small as r.

The above discussion gives us an algorithmic way of analyzing our original data compression problem. The possible values of $F(i)$ are encoded in a 17 character alphabet $p_0 \ldots p_{16}$, representing the length of copy available at each position. The compression algorithm is described by a non-deterministic machine that accepts strings of p_i; this machine has costs equal to the lengths of the codewords used by the algorithm. There are two parameterized states in this machine: l_x means that there is a literal codeword under construction with x spaces still available; c_y means that a copy is in progress with y characters remaining to copy. The idle state is $l_0 \equiv c_0$. In the non-deterministic machine, the possible transitions are:

$$\begin{array}{lll} l_0 \xrightarrow{p_*(2)} l_{15} & \text{start a literal} & \\ l_x \xrightarrow{p_*(1)} l_{x-1} & \text{continue a literal } (x \geq 1) & \quad (2) \\ l_* \xrightarrow{p_i(2)} c_{i-1} & \text{start a copy} & \\ c_y \xrightarrow{p_*(0)} c_{y-1} & \text{continue a copy} & \end{array}$$

(An asterisk is used as a wild card to denote any state.) Based on the theorems above we have already eliminated some transitions to simplify what follows. For example,

$$c_y \xrightarrow{p_*(2)} l_{15} \text{ start a literal from inside a copy } (y \geq 1) \quad (3)$$

is unnecessary. The deterministic machine, given below, eliminates many more transitions:

$$\begin{array}{ll} l_0 \xrightarrow{p_i(2)} l_{15} & \text{start a literal if } i \leq 1 \\ l_x \xrightarrow{p_i(1)} l_{x-1} & \text{continue a literal if } x \geq 1 \text{ and } i \leq 2 \\ l_x \xrightarrow{p_i(2)} c_{i-1} & \text{start a copy if } i \geq 3 \text{ or } x = 0 \text{ and } i = 2 \\ c_y \xrightarrow{p_*(0)} c_{y-1} & \text{continue a copy} \end{array}$$

Finally, we add one more machine to guarantee that the strings of p_i are realistic. In this machine, state s_i means that the previous character was p_i, so the index of the next character must be at least p_{i-1}:

$$s_i \xrightarrow{p_j} s_j \quad (j \geq i - 1) \quad (5)$$

The cross product of these three machines has approximately 17K states and was analyzed mechanically to prove a minimum ratio cycle of 4/5. Thus the policy we have chosen is never off by more than 25 percent, and the worst case is realized on a string that repeats a p_i pattern as follows:

1	2	3	4	5	6	7									
p_{10}	p_{10}	p_9	p_8	p_7	p_6	p_5									(6)
							8	9	10	11	12	13	14	15	
							p_4	p_3	p_2	p_1	p_2	p_{10}	p_{10}	p_9	...

(There is nothing special about 10; it was chosen to illustrate a long copy and to match the example in Appendix A.) The deterministic algorithm takes a copy of length 10 in the first position, and then switches to a literal for positions 11 and 12. Five bytes are used in each repetition of the pattern. The optimal solution is one position out of phase. It takes a copy of length 10 in the second position, and then finds a copy of length 2 at position 12, for a total of four bytes on each iteration.

We have abstracted the problem so that the possible copy operations are described by a string of p_i, and we have shown a pathological pattern of p_i that results in 5/4 of the optimal encoding. There might still be some doubt that such a string exists, since the condition that our third machine (5) guarantees, $F(i + 1) \geq F(i) - 1$, is a necessary but not sufficient condition. Nevertheless, the details of an actual pathological string can be found in Appendix A.

4. A SIMPLER DATA STRUCTURE

Although the quantity of code associated with A1 is not enormous, it is complicated, and the data structures are fairly large. In this section, we present simpler methods for finding the suffix and for propagating the window position.

The alternative to a percolating update is to update the positions in all nodes back to the root whenever a new leaf is inserted. Then no updates are needed when nodes are deleted. The *update* flags can be eliminated.

The alternative to suffix pointers is more complicated. The cost of movement in a tree is not uniform; moving deeper requires a hash table lookup, which is more expensive than following a parent pointer. So we can determine the suffix by starting at the suffix leaf and following parent pointers back toward the root until the suffix node is reached. The suffix leaf is known because the string at i matched the string at some earlier window position j; the suffix leaf $j + 1$ is the next entry in the leaf array. With this change, the suffix pointers can be eliminated.

From a theoretical perspective, these modifications, which have $O(nd)$ worst case performance for a file of size n and cut-off depth d, are inferior to the $O(n)$ performance of the suffix tree. For A1, with a cutoff of 16, these modifications improve average performance, but the A2 method discussed in the next section has such a deep cut-off that suffix pointers and percolated updates are preferable.

5. A MORE POWERFUL ENCODING

The 4,096-byte window of A1 is roughly optimal for fixed size copy and literal codewords. Longer copies would, on average, be found in a larger window, but a larger displacement field would be required to encode them. To exploit a larger window, we must use a variable-width encoding statistically sensitive to the fact that recent window positions are more likely to be used by copy codewords than those positions further back. Similarly, it is advantageous to use variable-width encodings for copy and literal lengths.

There are several approaches we might use for variable-length encoding. We could use fixed or adaptive Huffman coding, arithmetic encoding, a variable-length encoding of the integers, or a manageable set of hand-designed codewords. We eliminated from consideration adaptive Huffman and arithmetic coding because they are slow. Moreover, we felt they would provide (at best) a secondary adaptive advantage since the "front end" textual substitution is itself adapting to the input. We experimented with a fixed Huffman encoding, a hand-designed family of codewords, and a variable-length encoding of the integers, so we will compare these options briefly:

Hand-Designed Codewords. This is a direct generalization of A1, with short copies that use fewer bits but cannot address the full window, and longer copies that can address larger blocks further back in the window. With a few codewords, this is fast and relatively easy to implement. However, some care must be taken in the choice of codewords to maximize compression.

Variable-Length Integers. The simplest method we tried uses a unary code to specify field width, followed by the field itself. Copy length and displacement fields are coded independently via this technique, so any correlations are ignored. There are more elaborate codings of the integers (such as [9], [10], or [13]), that have been used by [15], and [34] in their implementations of Lempel-Ziv compression. These encodings have nice asymptotic properties for very large integers, but the unary code is best for our purposes since, as we will see shortly, it can be tuned easily to the statistics of the application. The unary code has the additional advantage of a simple hardware implementation. We will return to the unary code in more detail shortly.

Fixed Huffman. Ideally, a fixed Huffman encoder should be applied to source consisting of the copy length and displacement concatenated together (to capture the correlation of these two fields). However, since we wish to expand window size to 16384 and maximum copy length to 2000, the realities of gathering statistics and constructing an implementation dictate that we restrict the input of the fixed Huffman compressor to a size much smaller than 2000 × 16384 by grouping together codes with nearly equal copy lengths and displacements. To improve speed we use tables to encode and decode a byte at a time. Nevertheless, the fixed Huffman approach is the most complex and slowest of the three options compared here.

To decide how much compression could be increased with a Fixed Huffman approach, we experimented with several groupings of nearly equal copy lengths and displacements, using a finer granularity for small values, so that the input to the Fixed Huffman compressor had only about 30,000 states, and we computed the entropy to give a theoretical bound on the compression. The smallest entropy we obtained was only 4 percent more compact than the actual compression achieved with the unary encoding described below; and any real implementation would do worse than an entropy bound. Consequently, because the Fixed Huffman approach did not achieve significantly higher compression, we favor the simpler unary code, though this is not an overwhelmingly clear choice.

Define a (start, step, stop) unary code of the integers as follows: The nth codeword has n ones followed by a zero followed by a field of size start + n · step. If the field size is equal to stop then the preceding zero can be omitted. The integers are laid out sequentially through these codewords. For example, (3, 2, 9) would look like:

Codeword	Range
0*xxx*	0–7
10*xxxxx*	8–39
110*xxxxxxx*	40–167
111*xxxxxxxxx*	168–679

Appendix B contains a simple procedure that generates unary codes.

The A2 textual substitution method encodes copy length with a (2, 1, 10) code, leading to a maximum copy length of 2044. A copy length of zero signals a literal, for which literal length is then encoded with a (0, 1, 5) code, leading to a maximum literal length of 63 bytes. If copy length is non-zero, then copy displacement is encoded with a (10, 2, 14) code. The exact maximum copy and literal lengths are chosen to avoid wasted states in the unary progressions; a maximum copy length of 2044 is sufficient for the kinds of data studied in Section 8. The A1 policy for choosing between copy and literal codewords is used.

Three refinements are used to increase A2's compression by approximately 1 to 2 percent. First, since neither another literal nor a copy of length 2 can immediately follow a literal of less than maximum literal length, in this situation, we shift copy length codes down by 2. In other words, in the (2, 1, 10) code for copy length, 0 usually means literal, 1 means copy length 2, etc.; but after a literal of less than maximum literal length, 0 means copy length 3, 1 means copy length 4, etc.

Secondly, we phase-in the copy displacement encoding for small files, using a $(10 - x, 2, 14 - x)$ code, where x starts at 10 and descends to 0 as the number of window positions grows; for example, $x = 10$ allows $2^0 + 2^2 + 2^4 = 21$ values to be coded, so when the number of window positions exceeds 21, x is reduced to 9; and so forth.

Finally, to eliminate wasted states in the copy displacement encoding, the largest field in the $(10 - x, 2, 14 - x)$ progression is shrunk until it is just large enough to hold all values that must be represented; that is, if v values remain to be encoded in the largest field then smaller values are encoded with $\lfloor \log_2 v \rfloor$ bits and larger values with $\lceil \log_2 v \rceil$ bits rather than $14 - x$ bits. This trick increases compression during startup, and, if the window size is chosen smaller than the number of values in the displacement progression, it continues to be useful thereafter. For example, the compression studies in Section 8 use an A2 window size of 16,384 characters, so the (10, 2, 14) code would waste 5,120 states in the 14-bit field without this trick.

Percolating update seems preferable for the implementation of A2 because of the large maximum copy length; with update-to-root, pathological input could slow the compressor by a factor of 20. Unfortunately, the percolating update does not guarantee that the suffix tree will report the nearest position for a match, so longer codewords than necessary may sometimes be used. This problem is not serious because the tree is often shallow, and nodes near the root usually have many sons, so updates propagate much more rapidly than assumed in the analysis of Section 3. On typical files, compression with percolated update is 0.4 percent less than with update-to-root.

6. A FASTER COMPRESSOR

A2 has very fast expansion with a small storage requirement, but, even though compression has constant amortized time, it is 5 times slower than expansion. A1 and A2 are most appropriate in applications where compression speed is not critical and where the performance of the expander needs to be optimized, such as the mass release of software on floppy disks. However, in applications such as file archiving, faster compression is needed. For this reason, we have developed the B1 and B2 methods described here, which use the same encodings as A1 and A2, respectively, but compute window displacement differently. Copy codewords are restricted to start at the beginning of the yth previous codeword or literal character emitted; they can no longer address every earlier character, but only those where literal characters occurred or copy codewords started; we refer to displacements computed this way as "compressed displacements" throughout. Copy length is still measured in characters, like A1. By inserting this level of indirection during window access, compression speed typically triples, though expansion and the rate of adaptation are somewhat slower.

With "compressed displacements," suffix pointers and update propagation are unnecessary and a simpler PATRICIA tree can be used for the dictionary. Entries are made in the tree only on codeword boundaries, and this can be done in linear time by starting at the root on each iteration. It is useful to create an array of permanent nodes for all characters at depth 1. Since copy codewords of length 1 are never issued, it doesn't matter that some permanent nodes don't correspond to any window character. Each iteration begins by indexing into this node array with the next character. Then hash table lookups and arc character comparisons are used to descend deeper, as in A1. The new window position is written into nodes passed on the way down, so update propagation is unnecessary.

In short, the complications necessary to achieve constant average time per source character with A2 are eliminated. However, one new complication is introduced. In the worst case, the 16,384 window positions of B2 could require millions of characters, so we impose a limit of 12 × 16,384 characters; if the full window exceeds this limit, leaves for the oldest window positions are purged from the tree.

Because of slower adaptation, B2 usually compresses slightly less than A2 on small files. But on text and program source files, it surpasses A2 by 6 to 8 percent asymptotically; the crossover from lower compression to higher occurs after about 70,000 characters! A2 codewords find all the near-term context, while B2 is restricted to start on previous codeword boundaries but can consequently reach further back in the file. This gives B2 an advantage on files with a natural word structure, such as text, and a disadvantage on files where nearby context is especially important, such as scanned images.

We also tried variations where the tree is updated more frequently than on every codeword boundary and literal character. All variations up to and including A2 can be implemented within the general framework of this method, if speed is not an issue. For example, we found that about 1 percent higher compression can be achieved by inserting another compressed position between the two characters represented by each length 2 copy codeword and another 0.5 percent by also inserting compressed positions after each character represented by length 3 copy codewords. However, because these changes slow compression and expansion we haven't used them.

7. IMPROVING THE COMPRESSION RATIO

In Section 6 we considered ways to speed up compression at the cost of slower adaptation and expansion. In this section we will explore the other direction: improving the compression ratio with a slight cost to the running time of the algorithm.

When a string occurs frequently in a file, all the methods we have considered so far waste space in their encoding: when they are encoding the repeating string, they are capable of specifying the copy displacement to multiple previous occurrences of the string, yet only one string needs to be copied. By contrast, the data structures we have used do not waste space. The repeating strings share a common path near the root. If we base the copy codewords directly on the data structure of the dictionary, we can improve the compression ratio significantly. (This brings us closer to the second style of Ziv and Lempel's textual substitution work [19, 29, 43], where a dictionary is maintained by both the compressor and expander. However, since we still use a window and an explicit copy length coding, it is natural to view this as a modification of our earlier compressors, in the style of Ziv and Lempel's first textual substitution work.)

The C2 method uses the same PATRICIA tree data structures as B2 to store its dictionary. Thus it takes two pieces of information to specify a word in the dictionary: a node, and a location along the arc between the node and its parent (since PATRICIA tree arcs may correspond to strings with more than one character). We will distinguish two cases for a copy: if the arc is at a leaf of the tree, then we will use a *LeafCopy* codeword, while if the arc is internal to the tree we will use a *NodeCopy* codeword. Essentially, those strings appearing two or more times in the window are coded with *NodeCopies*, avoiding the redundancy of A2 or B2 in these cases.

The C2 encoding begins with a single prefix bit that is 0 for a *NodeCopy*, 1 for a *LeafCopy* or *Literal*.

For *NodeCopy* codewords, the prefix is followed by a node number in [0 .. *maxNodeNo*], where *maxNodeNo* is the largest node number used since initialization; for most files tested, *maxNodeNo* is about 50 percent the number of leaves. Following the node number, a displacement along the arc from the node to its parent is encoded; for most *NodeCopy* codewords the incoming arc is of length 1, so no length field is required. If a length field is required, 0 denotes a match exactly at the node, 1 a displacement 1 down the arc from the parent node, etc. Rarely is the length field longer than one or two bits because the arc lengths are usually short, so all possible displacements can be enumerated with only a few bits. For both the node number and the incoming arc displacement, the trick described in Section 5 is used to eliminate wasted states in the field; that is, if v values must be encoded, then the smaller values are encoded with $\lfloor \log_2 v \rfloor$ bits and larger values with $\lceil \log_2 v \rceil$ bits.

LeafCopies are coded with unary progressions like those of A2 or B2. A (1, 1, 11) progression is used to specify the distance of the longest match down the leaf arc from its parent node, with 0 denoting a literal; this progression leads to a maximum copy length of 4094 bytes. Since another literal never occurs immediately after a literal of less than maximum literal length, the *LeafCopy* arc distance progression is shifted down by 1 when the preceding codeword was a literal (i.e., arc displacement 1 is coded as 0, 2 as 1, etc.). On a cross section of files from the data sets discussed later, distance down the leaf arc was highly skewed, with about half the arc displacements occurring one character down the leaf arc. Because of this probability spike at 1 and the rapid drop off at larger distances, the average length field is small. Following the length field, the window position is coded by gradually phasing in a (10, 2, 14) unary progression exactly like B2's.

Literals are coded by first coding a *LeafCopy* arc displacement of 0 and then using a (0, 1, 5) unary progression for the literal length exactly like B2.

Unlike A2 and B2, the expander for C2 must maintain a dictionary tree exactly like the compressor's tree to permit decoding. Notice that this is not as onerous as it might seem. During compression, the algorithm must search the tree downward (root toward leaves) to find the longest match, and this requires a hash table access at each node. By contrast, the expander is told which node was matched, and can recover the length and window position of the match from the node. No hash table is required, but the encoding is restricted: a copy codeword must always represent the longest match found in the tree. In particular, the superior heuristic used by B2 to choose between Literal and Copy codewords must be discarded; instead, when the longest match is of length 2 or more, a copy codeword must always be produced. With this restriction, the expander can reconstruct the tree during decoding simply by hanging each new leaf from the node or arc indicated by the *NodeCopy* or *LeafCopy* codeword, or in the case of literals, by hanging the leaf from the permanent depth 1 node for each literal character.

8. EMPIRICAL STUDIES

In this section, we compare the five compression methods we have developed with other one-pass, adaptive methods. For most other methods, we do not have well-tuned implementations and report only compression results. For implementations we have tuned for efficiency, speed is also estimated (for our 3 MIP, 16-bit word size, 8 megabyte workstations). The execution times used to determine speed include the time to open, read, and write files on the local disk (which has a relatively slow, maximum transfer rate of 5 megabits per second); the speed is computed by dividing the uncompressed file size by the execution time for a large file.

We tested file types important in our working environment. Each number in Table I is the sum of the compressed file sizes for all files in the group divided by the sum of the original file sizes. Charts 1–3 show the dependency of compression on file size for all of the

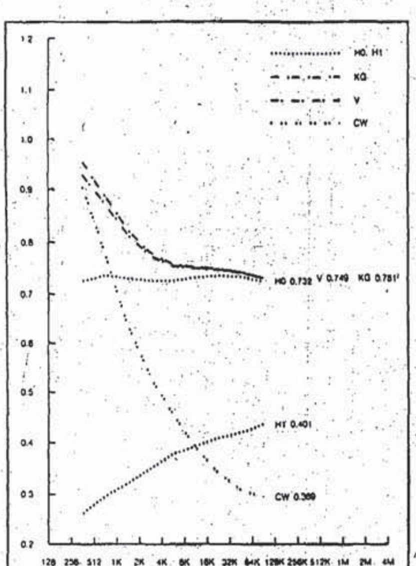


Chart 1. Compression vs. File Size, Data Set SC

compression methods tested on the source code (SC) data set.

Data Sets

SC Source Code. All 8-bit Ascii source files from which the boot file for our programming environment is built. Files include some English comments, and a densely-coded collection of formatting information at the end of each file reduces compressibility. The files themselves are written in the Cedar language. (1185 files, average size 11 Kbytes, total size 13.4 Mbytes)

TM Technical Memoranda. All files from a directory where computer science technical memoranda and reports are filed, excluding those containing images. These files are 8-bit Ascii text with densely-coded formatting information at the end (like the source code). (134 files, average size 22 Kbytes, total size 2.9 Mbytes)

NS News Service. One file for each work day of a week from a major wire service; these files are 8-bit Ascii with no formatting information. Using textual substitution methods, these do not compress as well as the technical memoranda of the previous study group, even though they are much larger and should be less impacted by startup transient; inspection suggests that the larger vocabulary and extensive use of proper names might be responsible for this. (5 files, average size 459 Kbytes, total size 2.3 Mbytes)

CC Compiled Code. The compiled-code files produced from the SC data set. Each file contains several different regions: symbol names, pointers to the symbols, statement boundaries and source positions for the debugger, and executable code. Because each region is small and the regions have different characteristics, these files severely test an adaptive compressor. (1220 files, average size 13 Kbytes, total size 16.5 Mbytes)

BF Boot File. The boot file for our programming environment, basically a core image and memory map. (1 file, size 525 Kbytes)

SF Spline Fonts. Spline-described character fonts used to generate the bitmaps for character sets at a variety of resolutions. (94 files, average size 39 Kbytes, total size 3.6 Mbytes)

RCF Run-coded Fonts. High-resolution character fonts, where the original bitmaps have been replaced by a run-coded representation. (68 files, average size 47 Kbytes, total size 3.2 Mbytes)

SNI Synthetic Images. All 8 bit/pixel synthetic image files from the directory of an imaging researcher. The 44 files are the red, green, and blue color separations for 12 color images, 2 of which also have an extra file to

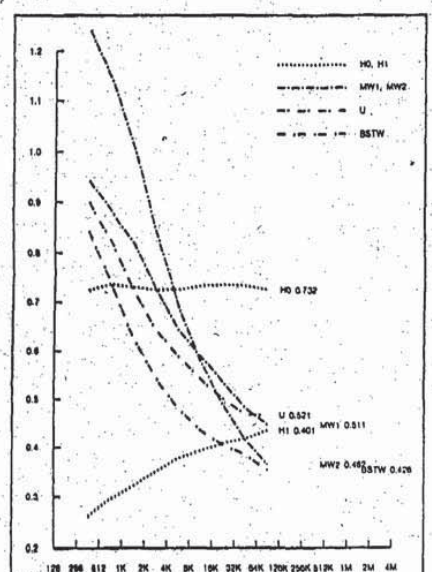


Chart 2. Compression vs. File Size, Data Set SC

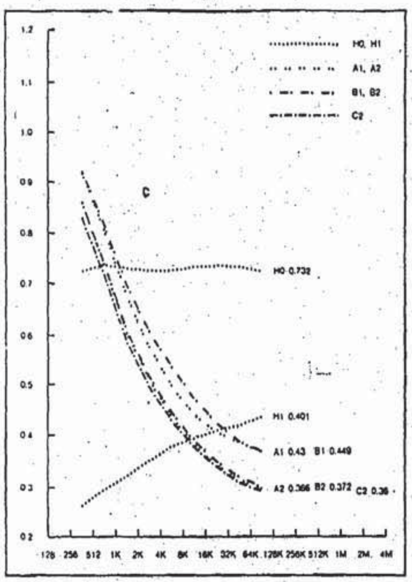


Chart 3. Compression vs. File Size, Data Set SC

encode background transparency; in addition, there are 6 other grey scale images. (44 files, average size 328 Kbytes, total size 14.4 Mbytes)

SCI Scanned Images. The red separations for all 8 bit/pixel scanned-in color images from the directory of an imaging researcher. The low-order one or two bits of each pixel are probably noise, reducing compressibility. (12 files, average size 683 Kbytes, total size 8.2 Mbytes)

BI Binary Images. CCITT standard images used to evaluate binary facsimile compression methods. Each file consists of a 148-byte header followed by a binary scan of 1 page (1728 pixels/scan line × 2376 scan lines/page). Some images have blocks of zeros more than 30,000 bytes long. Because these files are composed of 1-bit rather than 8-bit items, the general-purpose compressors do worse than they otherwise might. (8 files, average size 513 Kbytes, total size 4.1 Mbytes)

The special-purpose CCITT 1D and 2D compression methods reported in [18] achieve, respectively, 0.112 and 0.064 compression ratios on these standard images when the extraneous end-of-line codes required by the facsimile standard are removed and when the extraneous 148-byte header is removed. The special-purpose CCITT 2D result is significantly more compact than any general purpose method we tested, and only CW and C2 surpassed the 1D result.

Measurements and Compression Methods

H0 and H1. These are entropy calculations made on a per file basis according to:

$$H_0 = -\sum_{i=0}^{n-1} P(x = c_i)\log_2 P(x = c_i), \tag{7}$$

$$H_1 = -\sum_{i,j=0}^{n-1} P(x = c_i)P(y = c_j \mid x = c_i) \cdot \log_2 P(y = c_j \mid x = c_i) \tag{8}$$

where x is a random symbol of the source, xy is a randomly chosen pair of adjacent source characters, and c_i ranges over all possible symbols. Because of the small file size, the curves in Charts 1 to 3 drop off to the left. In theory, this small sampling problem can be corrected according to [2], but we have found it difficult to estimate the total character set size in order to apply these corrections. Nevertheless, Chart 1 shows that H0 is a good estimator for how well a memoryless (zero-order) compressor can do when file size is a large multiple of 256 bytes, and H1 bounds the compression for a first-order Markov method. (None of our files were large enough for H1 to be an accurate estimator.)

KG and V. These adaptive methods maintain a Huffman tree based on the frequency of characters seen so far in a file. The compressor and expander have roughly equal performance. The theory behind the KG approach appears in [11] and [23]. The similar V method, discussed in [37] should get better compression during the startup transient at the expense of being about 18 percent slower. It is also possible to bound the performance of Vitter's scheme closely to that of a fixed non-adaptive compressor. Except on the highly compressible CCITT images, these methods achieve compression slightly worse than H0, as expected. But because of bit quantization, the compression of the CCITT images is poor—arithmetic coding would compress close to H0 even on these highly compressible sources.

CW. Based on [6], this method gathers higher-order statistics than KG or V above (which we ran only on zeroth-order statistics). The method that Cleary and Witten describe keeps statistics to some order o and encodes each new character based on the context of the o preceding characters. (We've used $o = 3$, because any higher order exhausts storage on most of our data sets.) If the new character has never before appeared in the same context, then an escape mechanism is used to back down to smaller contexts to encode the character using those statistics. (We've used their escape mechanism A with exclusion of counts from higher-order contexts.) Because of high event probabilities in some higher-ordered contexts and the possibility of multiple escapes before a character is encoded, the fractional bit loss of Huffman encoding is a concern, so [6] uses arithmetic encoding. We have used the arithmetic encoder in [40].

Page 111 of 437

TABLE I. Comparison of Compression Methods

	Text			Binary		Fonts		Images		
Method	BC	TM	NS	CO	BF	SF	RCF	SNI	SCI	ST
H0	.732	.612	.590	.780	.752	.626	.756	.397	.845	.148
H1	.401	.424	.467	.540	.573	.380	.597	.181	.510	.101
KG	.751	.625	.595	.804	.756	.637	.767	.415	.850	.205
V	.749	.624	.595	.802	.756	.637	.768	.414	.850	.205
CW	.369	.358	.326	.768	.544	.516	.649	.233	.608	.106
MW1	.508	.470	.487	.770	.626	.558	.705	.259	.728	.117
MW2	.458	.449	.458	.784	.594	.526	.692	.270	.774	.117
UW	.521	.476	.442	.796	.638	.561	.728	.255	.697	.118
BSTW	.426	.434	.465	—	.684	—	.581	—	—	—
A1	.430	.461	.520	.741	.608	.502	.657	.351	.766	.215
A2	.366	.395	.436	.676	.538	.460	.588	.259	.709	.123
B1	.449	.458	.501	.753	.616	.505	.676	.349	.777	.213
B2	.372	.403	.410	.681	.547	.459	.603	.255	.714	.117
C2	.360	.376	.375	.668	.527	.445	.578	.238	.662	.105

As Table I shows, CW achieves excellent compression. Its chief drawbacks are its space and time performance. Its space requirement can grow in proportion to file size; for example, statistics for $o = 3$ on random input could require a tree with 256^4 leaves, though English text requires much less. The space (and consequently time) performance of CW degrades dramatically on "more random" data sets like SNI and SCI. A practical implementation would have to limit storage somehow. Even on English, Bell, Cleary, and Witten estimate that Moffat's tuned implementation of CW is 3 times slower compressing and 5 times slower expanding than C2 [4].

MW1. This method, described in [29], is related to the second style of Lempel-Ziv compression, alluded to in the introduction. It uses a Trie data structure and 12-bit codes. Initially (and always) the dictionary contains 256 one-character strings. New material is encoded by finding the longest match in the dictionary, outputting the associated code, and then inserting a new dictionary entry that is the longest match plus one more character. After the dictionary has filled, each iteration reclaims an old code from among dictionary leaves, following a LRU discipline, and reuses that code for the new dictionary entry. The expander works the same way. MW1 is simple to implement and is balanced in performance, with good speed both compressing and expanding (250,000 bits/sec and 310 bits/sec respectively). The original method used 12-bit codes throughout for simplicity and efficiency. However, our implementation starts by using 9-bit codewords, increasing to 10, 11, and finally to 12 bits as the dictionary grows to its maximum size; this saves up to 352 bytes in the compressed file size. On text and source code, Miller and Wegman determined that the 12-bit codeword size is close to optimal for this method.

MW2. One drawback of MW1 is the slow rate of buildup of dictionary entries. If, for example, the word *abcdefghi* appears frequently in a document, then *ab* will be in the dictionary after the first occurrence, *abc* after the second, and so on, with the full word present only after 8 occurrences (assuming no help from similar words in the document). A1 below, for example, would be able to copy the whole word *abcdefghi* after the first occurrence, but it pays a penalty for the quick response by having a length field in its copy codeword. The idea of MW2 is to build dictionary entries faster by combining adjacent codewords of the MW1 scheme. Longer words like *abcdefghi* are built up at an exponential rather than linear rate. The chief disadvantage of MW2 is its increased complexity and slow execution. Our implementation follows the description in [29] and uses an upper limit of 4096 dictionary entries (or 12-bit codewords). We did not implement the 9–12 bit phase-in that was used in MW1 so the size-dependent charts underestimate MW2's potential performance on small files.

UW. This is the Compress utility found in the Berkeley 4.3 Unix, which modifies a method described in a paper by Welch [39]; the authors of this method are S. Thomas, J. McKie, S. Davies, K. Turkowski, J. Woods, and J. Orost. It builds its dictionary like MW1, gradually expanding the codeword size from 9 bits initially up to 16 bits. The dictionary is frozen after 65,536 entries, but if the compression ratio drops significantly, the dictionary is discarded and rebuilt from scratch. We used this compressor remotely on a Vax-785, so it is difficult to compare its running time and implementation difficulties with the other methods we implemented. Nevertheless, because it does not use the LRU collection of codes, it should be faster than MW1. However, it has a larger total storage requirement and gets worse compression than MW1 on most data sets studied.

BSTW. This method first partitions the input into alphanumeric and non-alphanumeric "words," so it is specialized for text, though we were able to run it on some other kinds of data as well. The core of the compressor is a move-to-front heuristic. Within each class, the most recently seen words are kept on a list (we have used list size 256). If the next input word is already in the word list, then the compressor simply encodes the position of the word in the list and then moves the word to the front of the list. The move-to-front heuristic means that frequently used words will be near the front of the list, so they can be encoded with fewer bits. If the next word in the input stream is not on the word list, then the new word is added to the front of the list, while another word is removed from the end of the list, and the new word must be compressed character-by-character.

Since the empirical results in [5] do not actually give an encoding for the positions of words in the list or for the characters in new words that are output, we have taken the liberty of using the V compressor as a subroutine to generate these encodings adaptively. (There are actually four copies of Vitter's algorithm running, one to encode positions and one to encode characters in each of two partitions.) Using an adaptive Huffman is slow; a fixed encoding would run faster, but we expect that a fixed encoding would slightly reduce compression on larger files while slightly improving compression on small files. We could not run BSTW for all of the data sets, since the parsing mechanism assumes human-readable text and long "words" appear in the other data sets. When the unreadable input parsed well, as in the case of run-coded fonts, the compression was very good.

A1. This is our basic method described earlier. It has a fast and simple expander (560,000 bits/sec) with a small storage requirement (10,000 bytes). However, the compressor is much slower and larger (73,000 bits/sec, 145,000 bytes using scan-from-leaf and update-to-root). The encoding has a maximum compression to $\frac{1}{8}$ = 12.5 percent of the original file size because the best it can do is copy 16 characters with a 16-bit codeword.

Caveat: As we mentioned above, the running times reported include the file system overhead for a relatively slow disk. To provide a baseline, we timed a file copy without compression and obtained a rate of 760,000 bits per second. Thus, some of the faster expansion rates we report are severely limited by the disk. For example, we estimate that without disk overhead the A1 expander would be about twice as fast. On the other hand, removing disk overhead would hardly affect the compression speed of A1.

A2. This method, discussed in Section 5, enlarges the window to 16,384 characters and uses variable-width unary-coded copy and literal codewords to significantly increase compression. The running time and storage requirements are 410,000 bits/sec and 21,000 bytes for expansion and 60,000 bits/sec and 630,000 bytes for compression (using suffix pointers and percolated update).

B1. This method, discussed in Section 6, uses the A1 encoding but triples compression speed by updating the tree only at codeword boundaries and literal characters. The running time and storage requirements are 470,000 bits/sec and 45,000 bytes for expansion and 230,000 bits/sec and 187,000 bytes for compression.

B2. This method, discussed in Section 6, uses the same encoding as A2 but triples compression speed by updating the tree only at codeword boundaries and literal characters. The compressor and expander run at 170,000 and 380,000 bits/sec, respectively, and have storage requirements of 792,000 and 262,000 bytes.

C2. This method, discussed in Section 7, uses the same data structures as B2 but a more powerful encoding based directly upon the structure of the dictionary tree. Compression is about the same and expansion about 25 percent slower than B2; the compressor uses about the same storage as B2, but the expander uses more (about 529,000 bytes).

Table I highlights some differences between textual substitution methods like C2 and statistical methods like CW. (Time and space performance differences have been discussed earlier.) There are several data sets where these methods differ dramatically. On NS, CW is significantly better than C2. We believe that this is because NS shows great diversity in vocabulary: a property that is troublesome for textual substitution, since it cannot copy new words easily from elsewhere in the document, but is benign for CW, since new words are likely to follow the existing English statistics. On CC, for example, C2 is significantly better than CW. We believe that this is because CC contains several radically different parts, e.g. symbol tables, and compiled code. C2 is able to adjust to dramatic shifts within a file, due to literal codewords and copy addressing that favors nearby context, while CW has no easy way to rapidly diminish the effect of older statistics.

For all of our methods, A2, B2, and C2, window size is a significant consideration because it determines storage requirements and affects compression ratios. Chart 4 shows compression as a function of window size for the NS data set (concatenated into a single file to avoid start-up effects), and for the BF boot file. These two data sets were typical of the bimodal behavior we observed in our other data sets: large human-readable files benefit greatly from increasing window size, while other test groups show little improvement beyond a window size of 4K.

9. CONCLUSIONS

We have described several practical methods for lossless data compression and developed data structures to support them. These methods are strongly adaptive in the sense that they adapt not only during startup but also to context changes occurring later. They are suitable for most high speed applications because they make only one pass over source data, use only a con-

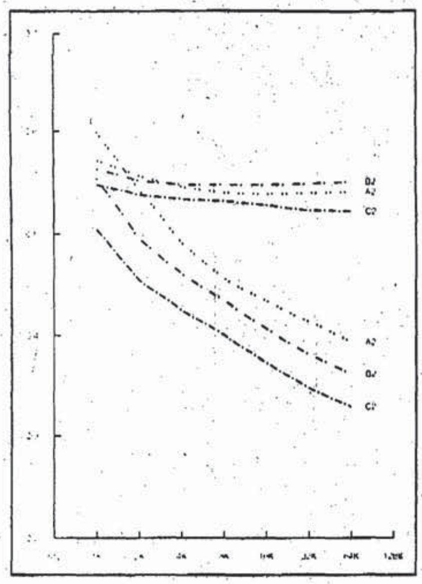


Chart 4. Compression vs. Window Size, Data Set NS (bottom) Data Set BF (top)

stant amount of storage, and have constant amortized execution time per character.

Our empirical studies point to several broad generalizations. First, based on the H0 and H1 theoretical limits, textual substitution via A2, B2, or C2 surpasses memoryless or first-order Markov methods applied on a character-by-character basis on half the data sets. On the other half, even the CW third-order method can't achieve the H1 bound. This suggests that, to surpass textual substitution for general purpose compression, any Markov method must be at least second-order, and to date, all such methods have poor space and time performance.

Secondly, the methods we've developed adapt rapidly during startup and at transitions in the middle of files. One reason for rapid adaptation is the use of smaller representations for displacements to recent positions in the window. Another reason is the inclusion of multi-character literal codewords. Together the literals and short displacements allow our methods to perform well on short files, files with major internal shifts of vocabulary or statistical properties, and files with bursts of poorly compressing material—all properties of a significant number of files in our environment.

Thirdly, it appears that the displacement-and-length approach to textual substitution is especially effective on small files. On 11,000-byte program source files, for example, A2 and B2 were over 20 percent more compact than textual substitution methods which did not use a length field (UW, MW1, and MW2). This is not surprising because the particular advantage of the length field in copy codewords is rapid adaptation on small files. However, even on the largest files tested, A2 and B2 usually achieved significantly higher compression. Only on images did other methods compete with them; our most powerful method, C2, achieved higher compression than any other textual substitution method we tested on all data sets. The effect of a length field is to greatly expand dictionary size with little or no increase in storage or processing time; our results suggest that textual substitution methods that use a length field will work better than those which do not.

Fourthly, studies of A2, B2, and C2 using different window sizes showed that, for human-readable input (e.g., English, source code), each doubling of window size improves the compression ratio by roughly 6 percent (for details see Chart 4). Furthermore, the data structures supporting these methods scale well: running time is independent of window size, and memory usage grows linearly with window size. Thus increasing window size is an easy way to improve the compression ratio for large files of human-readable input. For other types of input the window size can be reduced to 4096 without significantly impacting compression.

Going beyond these empirical results, an important practical consideration is the tradeoff among speed, storage, and degree of compression; speed and storage have to be considered for both compression and expansion. Of our own methods, A2 has very fast expansion with a minimal storage requirement; its weakness is slow compression; even though the suffix tree data structure with amortized update has constant amortized time per character, compression is still seven times slower than expansion. However, in applications which can afford relatively slow compression, A2 is excellent; for example, A2 would be good for mass distribution of software on floppy disks or for overnight compression of files on a file server. Furthermore, if the parallel matching in the compression side of A2 were supported with VLSI, the result would be a fast, powerful method requiring minimal storage both compressing and expanding.

B2 provides nearly three times faster compression than A2 but has somewhat slower expansion and adaptation. Thus B2 is well suited for communication and archiving applications.

A1 and B1 do not compress as well as A2 and B2, respectively, but because of their two-codeword, byte-aligned encodings they are better choices for applications where simplicity or speed is critical. (For example, J. Gasbarro has designed and implemented an expansion method like A1 to improve the bandwidth of a VLSI circuit tester [12].)

C2 achieves significantly higher compression than B2, but its expander is somewhat slower and has a larger storage requirement. In the compression study reported in Section 8, C2 achieved the highest compression of all methods tested on 6 of the 10 data sets.

We believe that our implementations and empirical results demonstrate the value of window-based textual substitution. Together the A, B and C methods offer good options that can be chosen according to resource requirements.

APPENDIX A
A Pathological Example

We now show a string that has the F pattern of equation (6) of Section 3:

$$\begin{array}{ccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & & & & & & & & \\ p_{10} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & & & & & & & & \\ & & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & & & & & \\ & & p_4 & p_3 & p_2 & p_1 & p_2 & p_{10} & p_{10} & p_9 & \ldots & & & & \end{array} \quad (6)$$

Hereafter we will stop abstracting the string by its copy lengths; capital letters are strings, small letters are single characters, and i, j, r, p, b are integers. The pathological string follows the pattern:

$$M_0M_1 \ldots M_{r-1}M_0M_1 \ldots M_{r-1}M_0M_1 \ldots \quad (9)$$

where the parameter r is chosen large enough so that one iteration exceeds the finite window (this prevents direct copying from the beginning of one M_0 to a subsequent M_0). Within each M_i we have groups,

$$M_i = G_{i0}G_{i1}G_{i2} \ldots G_{i(n/p-1)}, \quad (10)$$

and each group is:

$$G_{ij} = s_{jp}B_0 s_{(j+i)p} c_i s_{jp+1} B_1 s_{(j+i)p+1} c_i s_{jp+2} \\ B_2 s_{(j+i)p+2} c_i s_{jp+3} \ldots B_{p-1} s_{(j+i)p+p-1} c_i. \quad (11)$$

We have introduced two more parameters: p is the number of minor blocks B_i, and n is the number of s characters. All of the s subscripts in the above formula are computed mod n. The groups skew so that, for example, the beginning of $G_{10} = s_1 B_1 s_{p+1} \ldots$ will not match entirely with the beginning of $G_{00} = s_1 B_1 s_1 \ldots$. It will, however, match in two parts: the prefix $s_1 B_1$ appears in both strings, and the suffix $G_{10} = \ldots B_1 s_{p+1} \ldots$ will match with the suffix of $G_{01} = \ldots B_1 s_{p+1}$. If, for example, B_1 has 9 characters, this gives two consecutive locations where a copy of size 10 is possible, in the pattern of equation 6.

It remains to create the match of length 2 at position 12 in equation (6). For this purpose, each of the c_i above are either e_i or o_i. They will always precede respectively even and odd numbered s_j, and match in pairs with their following s_j's. For example, the e_0 in $G_{00} = s_1 B_1 s_1 e_0 s_2 B_2 s_2 \ldots$ will match with s_2. The $e_0 s_2$ match is hidden in a minor block segregated from the odd numbered s_i:

$$\begin{aligned} B_0 &= x e_0 s_0 e_0 s_2 e_0 s_4 \ldots e_0 s_{b-2} \\ B_1 &= x e_0 s_b e_0 s_{b+2} e_0 s_{b+4} \ldots e_0 s_{2b-2} \\ &\vdots \\ B_{n/b} &= x e_1 s_0 e_1 s_2 e_1 s_4 \ldots e_1 s_{b-2} \\ &\vdots \\ B_{p/2} &= x o_0 s_1 o_0 s_3 o_0 s_5 \ldots o_0 s_{b-1} \\ &\vdots \end{aligned} \quad (12)$$

This causes p and n to be related by:

$$pb = 2nr$$

In the case of our running example, where the finite window is size 4096 and the maximum copy length is 16 characters, an appropriate setting of the above parameters is:

$$r = 2, \quad b = 8, \quad p = 100, \quad n = 200 \quad (13)$$

We need to take some care that the heuristic does not find the optimal solution. It turns out that if we just start as in equation (9), then the first M_0 will not compress well, but the heuristic will start the behavior we are seeking in M_1. Asymptotically we achieve a worst case ratio of $4/3$ between the optimal algorithm and the policy heuristic.

APPENDIX B
Computing a Unary-Based Variable Length Encoding of the Integers

In Section 5 we defined a (start, stop, stop) unary code of the integers as a string of n ones followed by a zero followed by a field of j bits, where j is in the arithmetic progression defined by (start, stop, stop). This can be defined precisely by the following encoder:

Encode Var:

```
PROC [out: CARDINAL, start, stop, last: CARDINAL] ~ {
  UNTIL out < Power2[start] DO
    PutBits[1,1];
  out ← out - Power2[start];
  start ← start + stop;
```

```
ENDLOOP;
IF start < last THEN PutBits[out, start + 1]
                 —0 followed by field of size "start"
ELSE IF start > last THEN ERROR
ELSE PutBits[out, start]; —save a bit
];
```

```
PutBits: PROC[out: CARD, bits: INTEGER] --
  Output the binary encoding of "out" in a field of size
  "bits."
```

Notice that the encoder is able to save one bit in the last field size of the arithmetic progression.

Acknowledgments. We would like to thank Jim Gasbarro, John Larson, Bill Marsh, Dick Sweet, Ian Witten, and the anonymous referees for helpful comments and assistance.

REFERENCES

1. Abramson, N. *Information Theory and Coding*. McGraw-Hill, 1963, 61.
2. Basharin, G.P. On a statistical estimate for the entropy of a sequence of independent random variables. *Theory Prob. Appl.* 4, (1959), 333–336.
3. Bell, T.C. Better OPM/L text compression. *IEEE Trans. Commun.* COM-34, 12 (1986), 1176–1182.
4. Bell, T.C., Cleary, J.G., and Witten, I.H. *Text Compression*. In press with Prentice-Hall.
5. Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (1985), 320–330.
6. Cleary, J.G., and Witten, I.H. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun. COM-32*, 4 (1984), 396–402.
7. Cormack, G.V., and Horspool, R.N.S. Data compression using dynamic markov modelling. *The Computer Journal*, 30, 6 (1987), 541–550.
8. Dantzig, G.B., Blattner, W.O., and Rao, M.R. Finding a Cycle in a Graph with Minimum Cost to Time Ratio with Application to a Ship Routing Problem. *Theory of Graphs*, P. Rosenstiehl, ed. Gordon and Breach, 1966.
9. Elias, P. Universal codeword sets and representations of the integers. *IEEE Trans. Info. Theory IT-21*, 2 (1975), 194–203.
10. Even, S., and Rodeh, M. Economical Encoding of Commas Between Strings. *Commun. ACM 21* (1978), 315–317.
11. Gallager, R.G. Variations on a theme by Huffman. *IEEE Trans. Info. Theory IT-24*, 6 (1978), 668–674.
12. Gasbarro, J. An Architecture for High-Performance VLSI Testers. Ph.D. dissertation, Department of Electrical Engineering, Stanford University, 1988.
13. Golomb, S.W. Run-Length Encodings. *IEEE Trans. Info. Theory IT-12* (1966), 399–401.
14. Guazzo, M. A General Minimum Redundancy Source-coding Algorithm. *IEEE Trans. Info. Theory IT-26*, 1 (1980), 15–25.
15. Guoan, G., and Hobby, J. Using string matching to compress Chinese characters. *Stanford Tech. Rep.* STAN-CS-82-914, 1982.
16. Horspool, R.N., and Cormack, G.V. Dynamic Markov Modelling—A Prediction Technique. In *Proceedings of the 19th Annual Hawaii International Conference on System Sciences* (1986), 700–707.
17. Huffman, D.A. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E. 40* (1952), 1098–1101.
18. Hunter, R., and Robinson, A.H. International digital facsimile coding standards. In *Proceedings of the IEEE 68*, 7 (1980), 854–867.
19. Jakobsson, M. Compression of character strings by an adaptive dictionary. *BIT 25* (1985), 593–603.
20. Jones, C.B. An efficient coding system for long source sequences. *IEEE Trans. Info. Theory IT-27*, 3 (1981), 280–291.
21. Kempf, M., Bayer, R., and Güntzer, U. Time Optimal Left to Right Construction of Position Trees. *Acta Informatica*, 24 (1987), 461–474.
22. Knuth, D.E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, second printing, 1975.
23. Knuth, D.E. Dynamic Huffman coding. *J. Algo. 6* (1985), 163–180.
24. Langdon, G.G., Jr. A note on the Ziv-Lempel model for compressing individual sequences *IEEE Trans. Info. Theory IT-29*, 2 (1983), 284–287.
25. Langdon, G.G., Jr., and Rissanen, J. Compression of Black-White Images with Arithmetic Coding. *IEEE Trans. Commun. COM-29*, 6 (1981), 858–867.
26. Langdon, G.G., Jr., and Rissanen, J. A Double Adaptive File Compression Algorithm. *IEEE Trans. Commun. COM-31*, 11 (1983), 1253–1255.
27. Lawler, E.L. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.
28. McCreight, E.M. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2 (1976), 262–272.
29. Miller, V.S., and Wegman, M.N. Variations on a theme by Ziv and Lempel. *IBM Res. Rep.* RC 10630 (#47798), 1984. *Combinatorial Algorithms on Words, NATO ASI Series F, 12* (1985), 131–140.
30. Morrison, D.R. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM 15*, 4 (1968), 514–534.
31. Pasco, R.C. *Source Coding Algorithms for Fast Data Compression*. Ph.D. dissertation, Department of Electrical Engineering, Stanford University, 1976.
32. Rissanen, J., and Langdon, G.G., Jr. Arithmetic Coding. *IBM Journal of Research and Development* 23, 2 (1979), 149–162.
33. Rissanen, J., and Langdon, G.G., Jr. Universal Modeling and Coding. *IEEE Trans. Info. Theory IT-27*, 1 (1981), 12–23.
34. Rodeh, M., Pratt, V.R., and Even, S. Linear algorithm for data compression via string matching. *J. ACM 28*, 1 (1981), 16–24.
35. Shannon, C.E. A mathematical theory of communication. *The Bell System Technical Journal* 27, 3, 379–423 and 27, 4 (1948), 623–656.
36. Storer, J.A., and Szymanski, T.G. Data compression via textual substitution. *J. ACM 29*, 4 (1982), 928–951.
37. Vitter, J.S. Design and analysis of dynamic Huffman coding. *Brown University Technical Report No. CS-85-13*, 1985.
38. Weiner, P. Linear Pattern Matching Algorithms. *Fourteenth Annual Symposium on Switching and Automata Theory*, 1–11, 1973.
39. Welch, T.A. A technique for high performance data compression. *IEEE Comp. 17*, 6 (1984), 8–19.
40. Witten, I.H., Neal, R.M., and Cleary J.G. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (1987), 520–540.
41. Ziv, J. Coding theorems for individual sequences *IEEE Trans. Info. Theory IT-24*, 4 (1978), 405–412.
42. Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Info. Theory IT-23*, 3 (1977), 337–343.
43. Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Info. Theory IT-24*, 5 (1978), 530–536.

CR Categories and Subject Descriptors: E.4 [Data]: Coding and Information Theory—*data compaction and compression*; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—*computations on discrete structures, pattern matching*

General Terms: Algorithms, Design, Experimentation, Theory

Additional Key Words and Phrases: Amortized efficiency, automata theory, minimum cost to time ratio cycle, suffix trees, textual substitution

ABOUT THE AUTHORS:

EDWARD R. FIALA is a member of the research staff in the Computer Sciences Laboratory at the Xerox Palo Alto Research Center. He has general system interests including data compression and digital video. Author's Present Address: Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304; ARPA Internet: Fiala.pa@Xerox.com

DANIEL H. GREENE is presently a member of the research staff at Xerox's Palo Alto Research Center (PARC). He received his Ph.D. in 1983 from Stanford University. His current research interests are in the design and analysis of algorithms for data compression, computational geometry, and distributed computing. Author's Present Address: Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304; Greene.pa@Xerox.com

TEXT COMPRESSION

Timothy C. Bell
University of Cantebury

John G. Cleary
University of Calgary

Ian H. Witten
University of Calgary

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

BELL, TIMOTHY C., (Date)
Text compression / Timothy C. Bell, John G. Cleary, Ian H. Witten.
p. cm. — (Prentice Hall advanced reference series)
Includes bibliographical references (p. 303)
ISBN 0-13-911991-4
1. Text processing (Computer science) I. Cleary, John G.
II. Witten, I. H. (Ian H.) III. Title. IV. Series.
QA76.9. T48B45 1990
005—dc20 89-23206
CIP

Editorial/production supervision
and interior design: *Carol L. Atkins*
Cover design: *Karen A. Stephens*
Manufacturing buyer: *Ray Sintel*

Prentice Hall Advanced Reference Series

The publisher offers discounts on this book when ordered in bulk quantities. For more information, write:
Special Sales/College Marketing
Prentice-Hall, Inc.
College Technical and Reference Division
Englewood Cliffs, NJ 07632

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-911991-4

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

1

A

VARIABLE-LENGTH REPRESENTATIONS OF THE INTEGERS

A.1 CODING ARBITRARILY LARGE INTEGERS

The common binary fixed-length representation of the integers requires a prespecified number of bits for each integer, say n, and is able to represent integers in the range 1 to 2^n. Sometimes a maximum value is not known, so a variable-length representation must be used instead. Also, it is often desirable for small integers to have a shorter code than large ones. The representations here can code any integer value greater than zero, and the length of the encoding grows as the value being encoded increases.

Information theory tells us that if the representation of the integer i uses $|C(i)|$ bits, it is optimal if the probability of i is $2^{-|C(i)|}$. Thus the different representations given here imply different probability distributions, and the most appropriate for an application is the one whose probability distribution best approximates that of the integers being encoded. Figure A-1 shows the probability distributions implied by two of the codes described below, C_α and C_γ. The C_α code length increases linearly with the value being encoded, so the probability decreases exponentially. The length of the codes for C_γ (and all the other codes described here) increases logarithmically, so the implied probability decreases linearly. Consequently, C_α is suitable when almost all of the integers to be coded are very small, while the other codes are considerably more generous to larger integers. It is possible to generate an optimal set of codes (each with an

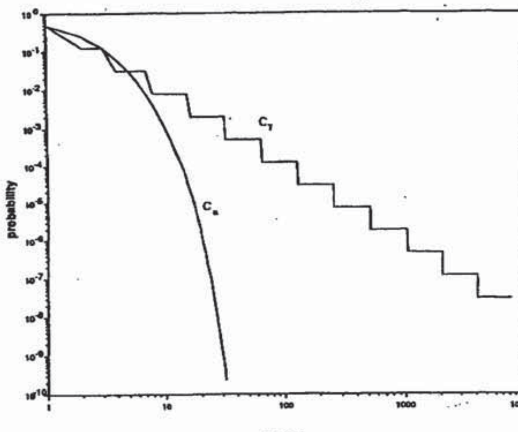


Figure A-1 Implied probabilities of the codes C_α and C_γ

integer number of bits) using Huffman's algorithm, but this requires that a probability distribution be provided, and it is slower.

Three of these codes (C_γ, C_δ and C_ω) are universal, which means that assigning messages in order of decreasing probability to the codewords gives an average codeword length within a constant of the optimal average length. Unfortunately, the constant may be too large for the code to be used in preference to one based on the probabilities. The codes C_δ and C_ω have the stronger property that the average code length approaches the entropy as the entropy increases. This means that they perform best when no message is much more likely than any other, which can be seen intuitively from the fact that the code lengths increase very slowly for large integers, corresponding to a flat probability distribution.

Examples of the following codings are given in Table A-1.

C_α: C_α is unary coding; that is, $\alpha(i)$ is $(i-1)$ zeros followed by a 1; or recursively, $\alpha(1) = 1$, $\alpha(i+1) = 0.\alpha(i)$. $|\alpha(i)| = i$.

TABLE A-1 EXAMPLES OF VARIABLE-LENGTH CODINGS OF THE INTEGERS; COLONS HAVE BEEN ADDED FOR CLARITY

i	$\alpha(i)$	$\beta(i)$	$\hat\beta(i)$	$\gamma'(i)$	$\gamma(i)$	$\delta(i)$	$\omega(i)$	$\omega'(i)$	$\omega''(i)$
1	1	1		1:	1	1:	0	001:0	000:
2	01	10	0	01:0	001	001:0	10:0	010:0	001:
3	001	11	1	01:1	011	001:1	11:0	011:0	010:
4	0001	100	00	001:00	00001	011:00	10:100:0	100:0	011:
5	00001	101	01	001:01	00011	011:01	10:101:0	101:0	100:0
6	000001	110	10	001:10	01001	011:10	10:110:0	110:0	101:0
7	0000001	111	11	001:11	01011	011:11	10:111:0	111:0	110:0
8	00000001	1000	000	0001:000	0000001	00001:000	11:1000:0	100:1000:0	111:0
9	000000001	1001	001	0001:001	0000011	00001:001	11:1001:0	100:1001:0	100:1000:0
10	0000000001	1010	010	0001:010	0001001	00001:010	11:1010:0	100:1010:0	100:1001:0
11	00000000001	1011	011	0001:011	0001011	00001:011	11:1011:0	100:1011:0	100:1010:0
12	—	1100	100	0001:100	0100001	00001:100	11:1100:0	100:1100:0	100:1011:0
13		1101	101	0001:101	0100011	00001:101	11:1101:0	100:1101:0	100:1100:0
14		1110	110	0001:110	0101001	00001:110	11:1110:0	100:1110:0	100:1101:0
15		1111	111	0001:111	0101011	00001:111	11:1111:0	100:1111:0	100:1110:0
16		10000	0000	00001:0000	000000001	00011:0000	10:100:1[illegible]0	101:10000:0	100:1111:0
17		10001	0001	00001:0001	000000011	00011:0001	10:100:1[illegible]:0	101:10001:0	101:10000:0
18		10010	0010	00001:0010	000001001	00011:0010	10:100:1[illegible]0	101:10010:0	101:10001:0
19		10011	0011	00001:0011	000001011	00011:0011	10:100:1[illegible]1:0	101:10011:0	101:10010:0
20		10100	0100	00001:0100	000100001	00011:0100	10:100:1[illegible]0	101:10100:0	101:10011:0
21		10101	0101	00001:0101	000100011	00011:0101	10:100:1[illegible]:0	101:10101:0	101:10100:0
22		10110	0110	00001:0110	000101001	00011:0110	10:100:1[illegible]0	101:10110:0	101:10101:0
23		10111	0111	00001:0111	000101011	00011:0111	10:100:[illegible]11:0	101:10111:0	101:10110:0
24		11000	1000	00001:1000	010000001	00011:1000	10:100:1[illegible]0	101:11000:0	101:10111:0
25		11001	1001	00001:1001	010000011	00011:1001	10:100:1[illegible]:0	101:11001:0	101:11000:0
26		11010	1010	00001:1010	010001001	00011:1010	10:100:1[illegible]0	101:11010:0	101:11001:0
27		11011	1011	00001:1011	010001011	00011:1011	10:100:1[illegible]1:0	101:11011:0	101:11010:0
28		11100	1100	00001:1100	010100001	00011:1100	10:100:[illegible]0	101:11100:0	101:11011:0
29		11101	1101	00001:1101	010100011	00011:1101	10:100:1[illegible]1:0	101:11101:0	101:11100:0
30		11110	1110	00001:1110	010101001	00011:1110	10:100:1[illegible]0	101:11110:0	101:11101:0
31		11111	1111	00001:1111	010101011	00011:1111	10:100:1[illegible]11:0	101:11111:0	101:11110:0
32		100000	00000	000001:00000	00000000001	01001:00000	10:101:[illegible]0	110:100000:0	101:11111:0
64		1000000	000000	0000001:000000	0000000000001	01011:000000	10:110:[illegible]000:0	111:1000000:0	110:111111:0
128		10000000	0000000	00000001:0000000	000000000000001	000001:0000000	10:111:[illegible]0000:0	100:1000:10000000:0	111:1111111:0
256		100000000	00000000	000000001:00000000	00000000000000001	000011:00000000	11:1000:[illegible]0000000:0	100:1001:100000000:0	100:1000:11111111:0

C_β: C_β is binary coding; that is, $\beta(1) = 1$, $\beta(2i) = \beta(i).0$, $\beta(2i+1) = \beta(i).1$. This coding is not decodable unless the length of the code is known in advance. Note that the most significant bit of $\beta(i)$ is always a 1. $|\beta(i)| = \lfloor \log i \rfloor + 1$.

$C_{\hat\beta}$: $C_{\hat\beta}$ is C_β with the most significant bit removed. It is not useful on its own, but is used to build other codes. $|\hat\beta(i)| = \lfloor \log i \rfloor$.

$C_{\gamma'}$: $C_{\gamma'}$ is a unary code for the number of bits in the binary coding of the integer, followed by the the binary coding of the integer with the most significant bit removed; that is, $\gamma'(i) = \alpha(|\beta(i)|).\hat\beta(i)$. $|\gamma'(i)| = 2\lfloor \log i \rfloor + 1$.

C_γ: C_γ can be viewed as $\lfloor \log i \rfloor$ zeros followed by a 1, and then $\hat\beta(i)$. C_γ is a rearrangement of $C_{\gamma'}$, with each of the $\lfloor \log i \rfloor$ zeros followed by a bit from $\hat\beta(i)$, and ending with a 1. Thus $|\gamma'(i)| = |\gamma(i)|$, but C_γ may be easier to implement than $C_{\gamma'}$.

C_δ: C_δ is the C_γ coding of the length of the integer, followed by the integer in binary; that is, $\delta(i) = \gamma(|\beta(i)|)\hat\beta(i)$ and $|\delta(i)| = 1 + \lfloor \log i \rfloor + 2\lfloor \log(1 + \lfloor \log i \rfloor)\rfloor$.

C_ω: With $\omega(i)$, first the code $\beta(i)$ is written. To the left of this the C_β representation of $(|\beta(i)| - 1)$ is written. This process is repeated recursively, with each earlier code being the binary encoding of the length, less 1, of the following code. The process halts on the left with a coding of 2 bits. A single zero is appended on the right to mark the end of the code.

$C_{\omega'}$: $C_{\omega'}$ is very similar to C_ω, but codes begin with 3 bits rather than 2, and the integer zero can be coded.

$C_{\omega''}$: $C_{\omega''}$ is $C_{\omega'}$ altered so that the integers range from 1 instead of 0, and with a redundant bit removed from the codes for the integers 1 to 4.

A.2 PHASING IN BINARY CODES

If an upper limit can be specified for the value of an integer, it can be coded using a normal binary representation. For example, if an integer is known to be in the range 0 to 15, it can be coded as a 4 bit binary number. In general, if the integer i is known to be in the range $0 \le i < \rho$, it can be coded in $k = \lceil \log \rho \rceil$ bits. This type of coding is often used to select entries in a growing dictionary, with the limit ρ being the number of items currently in the dictionary, rather than its maximum size. The codes become longer as the dictionary grows, so the code for a full dictionary is being "phased in."

Now, if ρ is not a power of 2, codes are available that are never used—for example, the range 0 to 8 requires 4-bit codes, but in this case the codes for 9 to 15 are also available. Table A-2 shows a more efficient set of codes, most of which are only 3 bits. In general, if $i < 2^k - \rho$, it is coded in $k-1$ bits; otherwise, a k-bit coding must be employed. A $(k-1)$-bit number is coded using its normal binary representation; the k-bit coding of i is the binary representation of $i + 2^k - \rho$. Of course, this slightly improved coding will increase the encoding and decoding time.

TABLE A-2 EXAMPLE OF EFFICIENT METHOD OF PHASING IN BINARY CODES

Integer	Code
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	1110
8	1111

A.3 START–STEP–STOP CODES

This family of codes has three parameters, (*start*, *step*, *stop*), and each selection yields a different coding of the integers. A great variety of codes can be generated in this manner, including C_β and $C_{\gamma'}$. The parameters can also impose a maximum value on the integers being coded.

A start–step–stop code employs n simple binary codes; the first code is *start* bits, the second *start* + *step*, then *start* + 2 × *step*, and so on, up to *start* + *stop* bits. Thus $n = (stop - start)/step + 1$. To show that a code belongs to set number s $(1 \le s \le n)$ it is prefixed by the unary code (C_α) for s. (If $i = n$, the final 1 of the unary code can be omitted.) The resulting codes are assigned to the integers in order of increasing size.

For example, Table A-3 shows a (1,2,5) code. Three binary codes are used: 1-bit (0 to 1), 3-bit (2 to 9), and 5-bit (10 to 41), and these are distinguished by prefixes of 1, 01, and 00, respectively. Other values for *start* and *step* will vary with the way the code length increases with the coded value, and *stop* determines the maximum value that can be encoded. The number of different codes available is

$$\frac{2^{stop+step} - 2^{start}}{2^{step} - 1}$$

Since this increases exponentially with *stop*, very high limits can be set with relatively small parameters—a (1,1,20) code can represent over 2 million integers. Two special cases of the start–step–stop code have already been described in Section A-1: the parameters $(k,1,k)$ yield a simple k-bit binary code (C_β), while $(0,1,\infty)$ is the $C_{\gamma'}$ code.

If the upper limit on the value to be encoded is increasing, better coding is obtained by *phasing* in the start–step–stop coding, in a similar manner to the method described in Section A-2 for binary coding. For example, one could start with a (0,2,4) code capable of coding 21 different values, then change to a (1,2,5) code when more than 21 values can occur, then a (2,2,6) code when the limit is greater than 42, and so on. Even in this case there will often be unused codes; for example, a (1,2,5) code (see Table A-3) with a limit of 22 codes is wasting 20 of the codes. An economy is to apply

TABLE A-3 START–STEP–STOP CODE FOR PARAMETERS (1,2,5)

Integer	Code
0	1:0
1	1:1
2	01:000
3	01:001
4	01:010
5	01:011
6	01:100
7	01:101
8	01:110
9	01:111
10	00:00000
11	00:00001
12	00:00010
13	00:00011
...	...
39	00:11101
40	00:11110
41	00:11111

the binary phasing-in technique of Section A-2 to the largest codes. In the example, these are 5-bit numbers intended to code the values 10 to 41. The range 22 to 41 will never be used, and the efficient phase-in can be applied to code the numbers 10 to 19 in 4 bits (plus the unary prefix), and 20 and 21 in 5 bits.

NOTES

The definitions of C_α, C_β, $C_{\beta'}$, $C_{\gamma'}$, C_γ, C_δ, and C_ω are from Elias (1975). $C_{\gamma'}$ was proposed independently by Bentley and Yao (1976). $C_{\omega'}$ is described by Rodeh et al. (1981). The efficient method of phasing in binary codes is described by Tischer (1987) and by Fiala and Greene (1989). The start–step–stop codes are due to Fiala and Greene (1989).