UNIVERSITY of CALIFORNIA
SANTA CRUZ

DIFFERENTIAL COMPRESSION: A GENERALIZED SOLUTION FOR BINARY FILES

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Randal C. Burns

December 1996

The thesis of Randal C. Burns is approved:

Prof. Darrell D. E. Long, Chair

Prof. Glen G. Langdon, Jr.

Dr. Ronald Fagin

Dean of Graduate Studies

Abstract

Differential Compression: A Generalized Solution for Binary Files

by

Randal C. Burns

This work presents the development and analysis of a family of algorithms for generating differentially compressed output from binary sources. The algorithms all perform the same fundamental task: given two versions of the same data as input streams, generate and output a compact encoding of one of the input streams by representing it as a set of changes with respect to the other input stream. Differential compression provides a computationally efficient compression technique for applications that generate versioned data and we often expect differencing to produce a significantly more compact file than more traditional compression techniques.

The greedy algorithm for file differencing is presented and this algorithm is proven to produce the optimally compressed differential output. However, this algorithm requires execution time quadratic in the size of the input files. We next present an algorithm to approximate the greedy algorithm in linear time and constant space. Finally, we present several advanced techniques that improve the performance of the linear algorithm and modify the greedy algorithm to run in linear time with minimal consequences on compression.

All algorithms were run against large data sets from a network file system in order to establish the viability of binary differential compression for operating system applications such as versioning file systems, document version control, file system backup and restore, consistency for distributed file system clients, and software distribution.

Contents

List of Figures

List of Tables

Acknowledgements

I am indebted to Dr. Ronald Fagin, Dr. Larry Stockmeyer and Dr. Miklos Ajtai of the IBM Almaden Research Center for their guidance and supervision. The majority of the innovation in this work resulted from our many sessions at the white board. It has been an honor to work with scientists of their caliber and the pleasure I took in collaborating with this group has rekindled my faith in the research process.

I would also like to offer special thanks to my advisor, Professor Darrell Long. The value of his input on this project can only be described as immeasurable. Darrell is a true mentor in that he not only provides knowledge, but also the guidance and other entrapments necessary for a complete education.

This line of research was introduced by Dr. Robert Morris who continually offered considerable insight into the implementation and application of binary differencing.

This work was conducted through the support of a grant from IBM and also during my employment at the IBM Almaden Research Center. I would like to thank Norm Pass who envisioned the potential value of this work and helped realize the support that IBM offered. This includes providing equipment, funding and experimental data.

I would like to extend my gratitude to the members of my committee who, in addition to their other contributions, continually helped improve the content and presentation of this work.

Finally, my thanks to all of my colleagues and friends who offered the comments and criticism necessary to produce a work of this nature. These contributors include: Theodore R. Haining, Thomas M. Kroeger, T. Spencer Burns, Cadir B. Lee, Bruce Sherrod, Tracey L. Sconyers, David P. Helmbold, J. Stewart Burns, Kevin Hagan, Jennifer L. Panders, Jane D. Burns, Joseph S. Burns and Alex D. Cronin.

Chapter 1

Introduction

1.1 Differencing Algorithms and Delta Files

Differencing algorithms compress data by taking advantage of statistical correlations between different versions of the same data sets. Strictly speaking, differencing algorithms achieve compression by finding common sequences between two versions of the same data that can be encoded using a copy reference. The term *file* will be used to indicate a linear data set to be addressed by a differencing algorithm. While this terminology is conventional, differencing applies more generally to any versioned data and need not be limited to files.

We define a *differencing algorithm* to be an algorithm that finds and outputs the changes made between two versions of the same file by locating common sequences to be copied and unique sequences to be added explicitly. A *delta file* (Δ) is the encoding of the output of a differencing algorithm. An algorithm that creates a delta file takes as input two versions of a file, a base file and a version file to be encoded, and outputs a delta file representing the incremental changes made

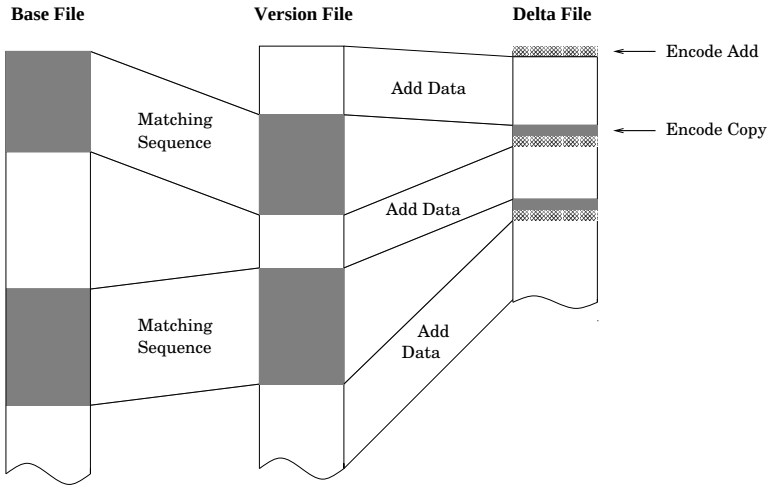


Figure 1.1: An example of encoding delta files with editing directives.

between versions.

$$F_{\text{base}} + F_{\text{version}} \rightarrow \Delta_{(\text{base,version})} \tag{1.1}$$

Reconstruction, the inverse operation, requires the base file and a delta file to rebuild a version.

$$F_{\text{base}} + \Delta_{(\text{base,version})} \rightarrow F_{\text{version}} \tag{1.2}$$

One encoding of a delta file consists of a linear array of editing directives (figure 1.1). These directives are copy commands, references to a location in a base file where the same data exists, and add commands, instructions to add data into the version file followed by the data to be added. There are other representations including those that represent delta files as linked data structures such as B/B+ trees or lists [16, 2], and one based upon matrix algebra [5]. In any representation scheme, a differencing algorithm must have found the copies and adds to be encoded. So, for our purposes, any encoding technique is compatible with the methods that we present.

1.2 Differential Algorithms Applied

Several potential applications of version differencing motivate the need for a compact and efficient differencing algorithm. Such an algorithm can be used to distribute software over a low bandwidth network such as a modem or the Internet. Upon releasing a new version of software, the version is differenced with respect to previous version. With compact versions, a low bandwidth channel can effectively distribute a new release of dynamically self updating software in the form of a binary patch. This technology has the potential to greatly reduce time to market on a new version and ease the distribution of software customizations.

For replication in distributed file systems, differencing can reduce by a large factor the amount of information that needs to be updated by transmitting deltas for all of the modified files in the replicated file set.

In distributed file system backup and restore, differential compression would reduce the time to perform file system backup, decrease network traffic during backup and restore, and lessen the storage to maintain a backup image [10]. Backup and restore can be limited by both bandwidth on the network, often 10 MB/s, and poor throughput to secondary and tertiary storage devices, often 500 KB/s to tape storage. Since resource limitations frequently make backing up the just the changes to a file system infeasible over a single night or even weekend, differential file compression has great potential to alleviate bandwidth problems by using available processor cycles to reduce the amount of data transferred. This technology can be used to provide backup and restore services on a subscription basis over any network including the Internet.

1.3 Previous Work

Differencing has it origins in both longest common subsequence (LCS) algorithms [3, 12] and the string-to-string correction problem [17]. Some of the first applications of differencing updated the screens of slow terminals by sending a set of edits to be applied locally rather than retransmitting a screen full of data. Another early application was the UNIX diff utility which used the LCS method to find and output the changes to a text file. diff was useful for source code development and primitive document control.

LCS algorithms find the longest common sequence between two strings by optimally removing symbols in both files leaving identical and sequential symbols.[1] While the LCS indicates the sequential commonality between strings, it does not necessarily detect the minimum set of changes. More generally, it has been asserted that string metrics that examine symbols sequentially fail to emphasize the global similarity of two strings [6]. Miller and Myers [9] established the limitations of LCS when they produced a new file compare program that executes at four times the speed of the diff program while producing significantly smaller deltas.

The *edit distance* [14] proved to be a better metric for the difference of files and techniques based on this method enhanced the utility and speed of file differencing. The edit distance assigns a cost to edit operations such as “delete a symbol”, “insert a symbol”, and “copy a symbol”. For example, one longest common subsequence between strings ***xyz*** and ***xzy*** is ***xy***, which neglects the common symbol ***z***. Using the edit distance metric, ***z*** may be copied between the two strings producing a smaller change cost than LCS. In the string-to-string correction problem [17], an algorithm minimizes the edit distance to minimize the cost of a given string transformation.

[1] A string/substring contains all consecutive symbols between and including its first and last symbol whereas a sequence/subsequence may omit symbols with respect to the corresponding string.

Tichy [14] adapted the string-to-string correction problem to file differencing using the concept of block move. Block move allows an algorithm to copy a string of symbols rather than an individual symbol. He then applied the algorithm to source code revision control package and created RCS [15]. RCS detects the modified lines in a file and encodes a delta file by adding these lines and indicating lines to be copied from the base version. We term this differencing at *line granularity*. The delta file is a line by line edit script applied to a base file to convert it to the new version. Although the SCCS version control system [13] precedes RCS, RCS generates minimal line granularity delta files and is the definitive previous work in version control.

Source code control has been the major application for differencing. These packages allow authors to store and recall file versions. Software releases may be restored exactly and changes are recoverable. Version control has also been integrated into a line editor [7] so that on every change a minimal delta is retained. This allows for an unlimited undo facility without excessive storage.

While line granularity may seem appropriate for source code, the concept of revision control needs to be generalized to include binary files. This allows binary data, such as edited multimedia, to be revised with the same version control and recoverability guarantees as text. Whereas revision control is currently a programmer's tool, binary revision control systems will enable the publisher, film maker, and graphic artist to realize the benefits of data versioning. It also enables developers to place image data, resource files, databases and binaries under their revision control system. Some existing version control packages have been modified to handle binary files, but in doing so they impose an arbitrary line structure. This results in delta files that achieve little or no compression as compared to storing the versions uncompressed.

Recently, an algorithm appeared that addresses differential compression of arbitrary byte streams [11]. The algorithm modifies the work of Tichy [14] to work on byte-wise data streams

rather than line oriented data. This algorithm adequately manages binary sources and is an effective developer's tool for source code control. However, the algorithm exhibits execution time quadratic in the size of the input, $\mathrm{O}(M \times N)$ for files of size M and N. The algorithm also uses memory linearly proportional to the size of the input files, $\mathrm{O}(M + N)$. To find matches, the algorithm implements the greedy method, which we will show to be optimal under certain constraints. The algorithm will then be used as a basis for comparison.

As we are interested in applications that operate on all data in a network file system, quadratic execution time renders differencing prohibitively expensive. While it is a well known result that the majority of the files are small, less than 1 kilobyte [4], a file system has a minority of files that may be large, ten to hundreds of megabytes. In order to address the differential compression of large files, we devise a differencing algorithm that runs in both linear time, $\mathrm{O}(M + N)$, and constant space, $\mathrm{O}(1)$. In Chapter 2, we outline basic methods for binary differencing and develop simple algorithms from these methods. In Chapter 3, we describe advanced techniques for binary differencing and using these techniques modify and improve our basic algorithms. Chapter 4 presents experimental data on the compressibility of a file system using these algorithms. We conclude in Chapter 5.

Chapter 2

Algorithms for Binary Differencing

Binary differencing algorithms all perform the same basic task. At the granularity of a byte, encode a set of version data as a set of changes from a base version of the same data. Due to their common tasks, all of the algorithms we examine share certain features. All binary differencing algorithms partition a file into two classes of variable length byte strings, those strings that appear in the base version and those that are unique to the version being encoded.

Before delving into the differencing algorithms themselves, we develop some language and techniques that will be common to the whole family of algorithms. We then explain and analyze two algorithms for binary differencing. We will present a *greedy algorithm* for binary differencing and prove that it finds the optimally compressed encoding of a version, but requires time quadratic in the size of the input files. Then we present a linear time, constant space algorithm that approximates the greedy algorithm. This linear algorithm sacrifices a degree of compression to achieve its performance bounds.

2.1 Methods for Binary Differencing

2.1.1 Data Streams

The binary algorithms under consideration operate on *data streams*. We term a *data stream* to be a data source that is byte addressable, allows random access, and stores consecutive data contiguously. The data stream abstraction is more appropriate for this application than the file abstraction, as the file abstraction provides a greater level of detail than the algorithms require. Files consists of multiple blocks of data which may exist on multiple devices in addition to being non-contiguous in storage or memory. In UNIX parlance, this is called the *i-node interface*. Files also lack byte addressability. Reads on a file are generally performed at the granularity of a file block, anywhere from 512 bytes to 64 kilobytes.

Many systems, such as UNIX, offer a byte addressable, seek-able and virtually contiguous file interface in the kernel. The UNIX read, write, open, close, and seek functions allow an application to treat file data as a stream. For the remainder of this work, the term file will be used to indicate a linear data source that meets the properties of a data stream.

For our purpose, data streams and consequently files will be assumed to have array semantics, *i.e.* the n^{th} offset in file A can be referred to as $A[n]$. This convention corresponds to the concept of memory mapped I/O, where the bytes of a file are logically mapped to a contiguous portion of the virtual address space.

2.1.2 Matching Strings

A data stream or file is composed of successive symbols from an alphabet, where symbols are a fundamental and indivisible element of data. For our purposes, symbols may be considered bytes and the alphabet is the set of all bytes, all combinations of 8 bits. While bytes are not truly

Table 2.1: Our method of encoding files use a single byte to indicate *add*, *copy* and *end* codewords. If required, a codeword may also specify additional bytes to follow.

- **ADD — 0nnnnnnn**
  The seven bits (**nnnnnnn**) trailing the **0** specify the number of bytes following the codeword that need to be added to the version file.

- **COPY — 1kknnnnn**
  All codewords starting with a **1** copy bytes from the base file to reconstruct the version file.
  nnnnn specifies the 5 lower bits for the copy length.
  kk selects from four formats for a copy command.

kk	following bytes	offset bits	length bits	max offset	max length
00	ss	16	5	64 KB	32 bytes
01	ssl	16	13	64 KB	8 KB
10	sssl	24	13	16 MB	8 KB
11	ssslll	32	29	4 GB	512 MB

  An `s' indicates a following byte used to encode the offset in the base version and an `l' indicates a following byte used to encode the length of the copy.

- **END — 00000000**
  Terminate the processing of a delta file.

indivisible, they do represent a fundamental unit for write, read and copy operations in the data streams that we address. Any combination of sequential and contiguous bytes comprise a *string*.

A differencing algorithm finds the changes between two versions of the same data by partitioning the data into strings that have changed and strings that have not changed. Those strings that have not changed may be compressed by encoding them with a reference to the same data in the other file. The quality of a differencing algorithm depends upon its ability to find the maximum number of matching strings. The algorithm that produces the minimal delta finds a maximum total length of strings to be copied between files. In a minimal delta, the amount of data not copied represents the changed data between versions.

2.1.3 Delta Files – Encoding the Changes

Having found a partitioning of a version, the data stream must then be encoded in an output stream. In order to better compare different techniques, all of the algorithms we develop use the same file encoding [11]. This encoding consists of three types of codewords. There is an *add* codeword, which is followed by the length of the string to add and the string itself, a *copy* codeword, which is followed by the length of the copy and an offset in the base version that references the matching string, and an *end* codeword, which indicates the end of input. These codewords are summarized in table 2.1.

For our purposes, the choice of this encoding, as compared to an equally good or better encoding, has a negligible effect on the algorithmic performance. Consider a worst case scenario where no strings can be copied from the base version and all data must be added to the file explicitly. In this case, the algorithm pays 1 byte, for the add codeword, to every 128 bytes of data. The codeword overhead is then less than 1%. No encoding can possibly have less overhead than the logarithm of the length of data that it encodes[1] and attainable encodings have overhead comparable to our selection.

2.1.4 Footprints – Identifying Matching Strings

An algorithm that differences files needs to match strings of symbols that are common between two versions of the same file, a base file, the reference version for the difference, and a version file, the file to be encoded. In order to find these matching strings, the algorithm remembers strings that it has seen previously. However, the common strings may not be stored explicitly as this is no smaller than the file being encoded.

[1] The minimum encoding of the value n uses $\log(n)$ bits.

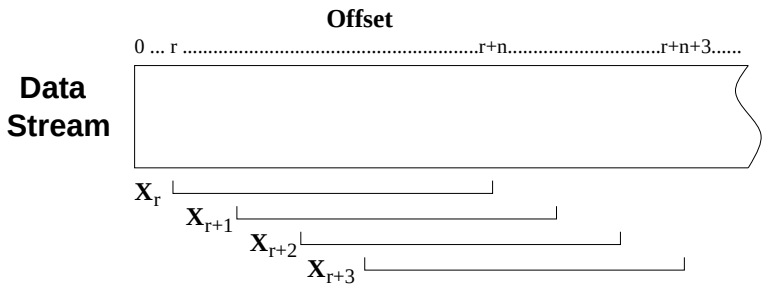


Figure 2.1: Footprints are generated by running a hashing function over strings of length n at successive symbols offsets.

In order to compactly identify a fixed length string of symbols, that string will be reduced to a large integer by some hashing function. This large integer is the string's *footprint*. A footprint does not uniquely represent a string, but does exhibit the following property: two matching strings will always express matching footprints, or equivalently, footprints that do not match always imply that the strings they represent differ. Note that it cannot be said that matching footprints imply matching strings. Since a footprint reduces the amount of information required to represent a given string, there are by definition fewer footprint values than possible combinations of strings. To determine if strings are identical in the presence of matching footprints, the strings themselves must be examined symbol by symbol.

Differencing algorithms will use footprints to remember and locate strings that have been seen previously. These algorithms use a hash table with size equal to the cardinality of the set of footprints, *i.e.* there is a one to one correspondence between potential footprint values and hash table entries. Each hash table entry holds at a minimum a reference to the string that generated the footprint. When a string hashes to a value that already has an entry in the hash table, a potential match has been found. To verify that the strings match, an algorithm will look up the strings using the stored offsets and perform a symbol-wise comparison. Strings that match may be encoded as

copies and strings that differ are *false matches*, different strings with the same footprint, and should be ignored.

2.1.5 Selecting a Hash Algorithm

Footprints are generated by a hashing function. A good hashing function for this application must be both run time efficient and generate a near uniform distribution of footprints over all footprint values. A non-uniform distribution of footprints results in differing strings hashing to the same footprint with higher probability.

Many hashing functions meet the requirement for a uniform distribution of keys [11, 1]. Differencing algorithms often need to calculate footprints at successive symbol offsets over a large portion of a file (Figure 2.1). This additional requirement makes Karp–Rabin hashing functions [8] more efficient than other methods.

Karp–Rabin techniques permit the incremental calculation of footprints. As successive string prefixes differ by only a single symbol, one implementation has the relation given by table 2.2. This method takes the original footprint, subtracts the value that the non-overlapping symbol added and uses this value, in combination with the symbol that was not in the original string prefix, to generate the new footprint. All arithmetic may be done modulo the number of hash entries, as addition and consequently multiplication are congruent over the modulus operation.

When calculating successive footprints, Karp–Rabin hashing dramatically improves the execution time of footprint generation and is consequently a significant performance benefit for differencing algorithms.

Table 2.2: One possible Karp–Rabin hash function calculated directly (2.1) and incrementally (2.2). q is the number of possible footprints and d is a multiplier typically equal to the cardinality of the symbol alphabet.

For the string of symbols $X_r = x_r, x_{r+1}, ..., x_{r+n}$

$$H(X_r) = \left(\sum_{i=1}^{n} x_i d^{n-i} \right) \mod q \tag{2.1}$$

$$H(X_{r+1}) = \left(H(X_r) - d^{n-1} x_r + x_{r+n+1} \right) \mod q \tag{2.2}$$

2.2 The Greedy Algorithm

Greedy algorithms often provide simple solutions to optimization problems by making what appears to be the best decision, the greedy decision, at each step. For differencing files, the greedy algorithm takes the longest match it can find at a given offset on the assumption that this match provides the best compression. It makes a locally optimal decision with the hope that this decision is part of the optimal solution over the input.

For file differencing, we prove the greedy algorithm provides an optimal encoding of a delta file and show that it requires time proportional to the product of the sizes of the input files. Then we present an algorithm which approximates the greedy algorithm in linear time and constant space by finding the match that appears to be the longest without performing exhaustive search for all matching strings.

2.2.1 Delta Compression with Greedy Techniques

Given a base file and another version of the same file, the greedy algorithm for constructing differential files finds and encodes the longest copy in the base file corresponding to the first offset in the version file. After advancing the offset in the version file past the encoded copy, it

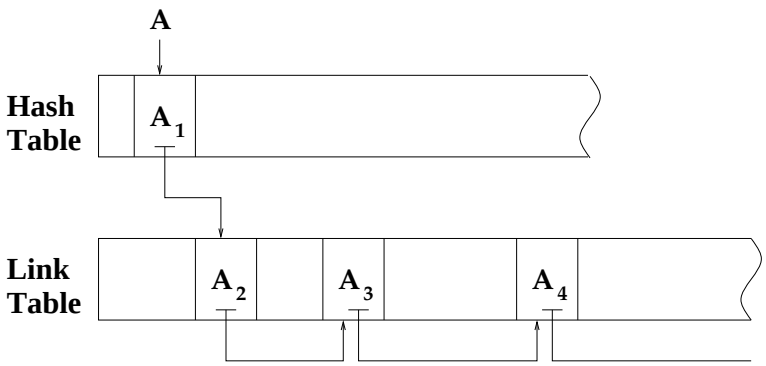


Figure 2.2: Data structures for the greedy algorithm.

looks for the longest copy starting at the current offset. If at a given offset, it cannot find a copy, the symbol at this offset is marked to be added and the algorithm advances to the following offset.

The first task the algorithm performs is to construct a hash list and a link list (Figure 2.2) out of the base version of the input files. The hash table allows an algorithm to store or identify the offset of a string with a given footprint. The link list stores the offsets of the footprints, beyond the initial footprint, that hash to the same value. In this example, strings at offset A_1, A_2, A_3, and A_4 all have a footprint with value A. The link list effectively performs as a re-hash function for this data structure. These data structures are assembled by the function **BuildHashList** in Figure 2.3.

The algorithm then finds the matching strings in the file. The **FindBestMatch** function (Figure 2.3) hashes the string at the current offset and returns the longest match that contains the string identified by the footprint. The function exhaustively searches through all strings that have matching footprints by fully traversing the link list for the matched hash entry. If the current offset in the version file *verFile* has footprint A, the function looks up the A^{th} element in the hash table to find a string with footprint A in the base file. In $hashtable[A]$, we store the offset of the string with a matching footprint. The string at the current offset in the version file is compared with the string at $hashtable[A]$ in the base file. The length of the matching string at these offsets is recorded. The

function then moves to $linktable[hashtable[A]]$ to find the next matching string. Each successive string in the link table is compared in turn. The longest matching string with offset $copy\_start$ and length $copy\_length$ is returned by the function **FindBestMatch**.

Alternatively, if **FindBestMatch** finds no matching string, the current offset in the version file (*ver_pos*) is incremented and the process is repeated. This indicates that the current offset could not be matched in the base version (*baseFile*) and will therefore be encoded as an add at a later time.

Once the algorithm finds a match for the current offset, the unmatched symbols previous to this match are encoded and output to the delta file using the **EmitAdd** function and the matching strings are output using the **EmitCopy** function. When all input from *verFile* has been processed, the algorithm terminates by outputting the end code to the delta file with the **EmitEnd** function.

We now prove that the greedy algorithm is optimal for a simplified file encoding scheme. In this case an optimal algorithm produces the smallest output delta. For binary differencing, symbols in the file may be considered bytes and a file a stream of symbols. However, this proof applies to differencing at any granularity. We introduce and use the concept *cost* to mean the length (in bits) for the given encoding of a string of symbols.

Claim Given a base file B, a version of the same file V, and an alphabet of the symbols Σ, by making the following assumptions:

- A copy of any length may be encoded with a unit cost = c.
- All symbols in the alphabet Σ appear in the base file B.
- Copying a string of length l with maximum cost $c \times l$ provides an encoding as compact as adding the same string.

we can state:

```
procedure GreedyDifference (footprintLength : number; baseFile, verFile, diffFile : stream )

local ver_pos, add_start, copy_start : number;
local hashtable[HASHTABLESIZE] : number;
local linktable[length(baseFile)] : number;

begin
   ver_pos ← 0;
   add_start ← 0;

** Fill the Hash Table and Link List with Footprints from the baseFile **

   BuildHashTable (baseFile, hashtable, linktable);

   while (ver_pos ≤ length(verFile ) - footprintLength)

** FindBestMatch hashes a footprint at ver_pos in verFile, looks at all matching **
** strings in the hash table and link list, and sets copy_start and copy_length **
** to the offset and length of that string in baseFile **

      FindBestMatch (ver_pos, copy_start, copy_length, hashtable, linktable, verFile, baseFile);

      if (copy_len ≥ footprintLength)
         if (add_start < verPos)
            EmitAdd (add_start, ver_pos − add_start, verFile, diffFile);
         end if

         EmitCopy (ver_pos, copy_start, copy_length, verFile, baseFile, diffFile);
         ver_pos ← ver_pos + copy_len;
         add_start ← ver_pos;
      else
         ver_pos ← ver_pos + 1;
      end if
   endwhile

   EmitEnd (diffFile);
end
```

Figure 2.3: Pseudo-code for the greedy algorithm as adapted from the work of Reichenberger.

Theorem 1 *The greedy algorithm finds an optimal encoding of the version file with respect to the base file.*

Proof Since all symbols in the alphabet Σ appear in the base file B, a symbol or string of symbols in the version file V may be represented in a differential file D exclusively by a copy or series of copies from B. Since we have assumed a unit cost function for encoding all copies and this cost is less than or equal to the cost of adding a symbol in the version file, there exists an optimal representation P, of V with respect to B, which only copies strings of symbols from B. In order to prove the optimality of a greedy encoding G, we require the intermediate result of Lemma 1.

Lemma 1 *For an arbitrary number of copies encoded, the length of version file data encoded by the greedy encoding is greater than or equal to the length of data encoded by optimal encoding.*

Proof *(by induction)* We introduce p_i to be the length of the i^{th} copy in the optimal encoding P and g_i to be the length of the i^{th} copy in the greedy encoding G. The length of data encoded in P and G after n copies are respectively given by $\sum_{i=1}^{n} p_i$ and $\sum_{i=1}^{n} g_i$.

1. At file offset 0 in V, P has a copy command of length p_1. G encodes a matching string of length g_1 which is the longest string starting at offset 0 in V. Since G encodes the longest possible copy, $g_1 \geq p_1$.

2. Given that G and P have encoded $n-1$ copies and the current offset in G is greater than the current offset in P, we can conclude that after G and P encode an n^{th} copy that the offset in G for n copies is greater than the offset in P.

$$\sum_{i=1}^{n-1} g_i \geq \sum_{i=1}^{n-1} p_i \implies \sum_{i=1}^{n} g_i \geq \sum_{i=1}^{n} p_i \tag{2.3}$$

G encodes a copy of length g_n and P encodes a copy of length p_n. If equation 2.3 did not hold, P would have found a copy of length p_n at offset $\sum_{i=1}^{n-1} p_i$ that is greater than $g_n + \sum_{i=1}^{n-1} g_i - \sum_{i=1}^{n-1} p_i$. A substring of this copy would be a string starting at $\sum_{i=1}^{n-1} g_i$ of length greater than g_n. As G always encodes the longest matching string, in this case g_n, this is a contradiction and equation 2.3 must hold. ■

Having established Lemma 1, we conclude that the number of copy commands that G uses to encode V is less than or equal to the number of copies used by P. However, since P is an optimal encoding, the number of copies P uses to encode V is less than or equal to the number that G uses. We can therefore state that, $\text{size}(G) = \text{size}(P) = c \times N$ where N is the number of copy commands in greedy encoding. ■

We have shown that the greedy algorithm provides an optimal encoding of a version file. Practical elements of the algorithm weaken our assumptions. In particular, the selection of codewords for encoding both adds and copies invalidates the unit cost assumption. However, we argued in section 2.1.3 that the choice of a particular encoding technique has a minimal impact on compression. Consequently, the greedy algorithm consistently reduces files to near optimal and should be considered a minimal differencing algorithm.

2.2.2 Analysis of Greedy Methods

Common strings may be quickly identified by common footprints, the value of a hash function over a fixed length prefix of a string. The greedy algorithm must examine all matching footprints and extend the matches in order to find the longest matching string. The number of matching footprints between the base and version file can grow with respect to the product of the sizes of the input files, *i.e.* $\text{O}(M \times N)$ for files of size M and N, and the algorithm uses time

proportional to the number of matching footprints.

In practice, many files elicit this worst case behavior. In both database files and executable files, binary zeros are stuffed into the file for alignment. This "zero stuffing" creates frequently occurring common footprints which must all be examined by the algorithm.

Having found a footprint in the version file, the greedy algorithm must compare this footprint to all matching footprints in the base file. This requires it to maintain a canonical listing of all footprints in one file, generally kept by computing and storing a footprint at all string prefix offsets [11]. Consequently, the algorithm uses memory proportional to the size of the input, $\mathrm{O}(N)$, for a size N file.

2.3 A Simple Linear Differencing Algorithm

Having motivated the need to difference all files in a file system and understanding that not all file are small [4], we improve upon both the run-time performance bound and run-time memory utilization of the greedy algorithm. Our algorithm intends to find matches in a greedy fashion but does not guarantee to execute a greedy policy exactly.

2.3.1 Linear versus Greedy – An Overview

The linear algorithm modifies the greedy algorithm in that it attempts to take the longest match at a given offset by taking the longest matching string at the first matching prefix beyond the offset at which the previous match was encoded. We call this the *next match* policy. In effect, it encodes the first matching string found rather than searching all matching footprints for the best matching string. For versioned data, matching strings are often sequential, *i.e.* they occur in the same order in both files. When strings that match are sequential, the next matching footprint

approximates the best match extremely well. In fact this property holds for all changes that are insertions and deletions (Figure 2.6).

2.3.2 The Linear Algorithm Described

The linear algorithm differences in a single pass over both files. Starting at offset zero in both files, *ver_pos* in the version and *base_pos* (Figure 2.4) in the base file, generate footprints for the strings at these offsets and store these footprints in the hash table so that they may be used later to find matching strings. The algorithm then increments the pointers and continues hashing at the following offsets. Data is collected in the hash table until the algorithm finds *colliding* footprints between the base and version file. Footprints *collide* when a new string has a footprint that has the same value as a string's footprint already stored in the hash table. The strings represented by the colliding footprints are checked for identity using the **Verify** function, and, if identical, the matching strings are encoded for output using the **EmitCodes** function.

The **EmitCodes** function (Figure 2.5) outputs all of the data in the version file between the end of the previous copy and the offset of the current copy as an add command. The footprints from this data were not matched in the base file and therefore need be explicitly added to the delta file. Then, starting with the matching strings, the function attempts to extend the match as far as possible. Note that the match may be longer than the footprint length. The longest matching strings from these offsets are encoded as a copy and output to the delta file.

After the copy of strings is encoded, the algorithm updates the current offsets in both files to point to the end of the encoded copy. If the files are versions of each other, the copies should represent the same data in both files and the end of both copies should be a point of file pointer synchronization. A point of synchronization in this case is defined to be the relative offsets of the

```
procedure LinearDifference (footprintLength : number; baseFile, verFile, diffFile : stream)

local baseh, verh, base_pos, ver_pos, ver_start : number;
local hashtable[HASHTABLESIZE] : struct hash_ entry;
local base_active : flag

begin
   ver_pos ← 0;
   base_pos ← 0;
   base_active ← TRUE;

   while (ver_pos ≤ length(verFile) − footprintLen)

      verh ← Footprint (verFile[ver_pos]);

      if (base_active)
         baseh ← Footprint (baseFile[base_pos]);
      end if

      if ((BASEFILE = hashtable[verh].file) and (Verify (baseFile[hashtable[verh].offset], verFile[ver_pos])))
         length ← EmitCodes (hashtable[verh].offset, ver_pos, ver_start, baseFile, verFile, diffFile);
         base_pos ← hashtable[verh] + length;
         ver_pos ← ver_pos + length;
         ver_start ← ver_pos;
         FlushHashTable(hashtable);
         continue;
      else
         hashtable[verh].offset ← ver_pos;
         hashtable[verh].file ← VERFILE;
      end if

      if ((base_pos ≤ length(baseFile) − footprintLen) and (base_active))
         if ((VERFILE = hashtable[baseh].file) and
               (Verify (verFile[hashtable[baseh].offset], baseFile[base_pos])) and
               (ver_start ≤ hashtable[baseh].offset))

            length ← EmitCodes (base_pos, hashtable[baseh].offset, ver_start, baseFile, verFile, diffFile);
            ver_pos ← ver_pos + length;
            base_pos ← base_pos + length;
            ver_start ← ver_pos;
            FlushHashTable(hashtable);
            continue;
         else
            hashtable[baseh].offset ← ver_pos;
            hashtable[baseh].file ← BASEFILE;
         end if
      else
         base_active ← FALSE;
      end if

      ver_pos ← ver_pos + 1;
      base_pos ← base_pos + 1;
   end while

   EmitCodes (length(baseFile), length(verFile), ver_start, baseFile, verFile, diffFile);
   EmitEnd (diffFile);
end
```

Figure 2.4: Pseudo-code for the linear time differencing algorithm.

```
number procedure EmitCodes (basePos, verPos, verStart : number; baseFile, verFile, diffFile : stream)

local copy_length : number;

begin

   if (verPos > verStart)
      EmitAdd (verStart, verPos − verStart, verFile, diffFile);
   end if

** Find the longest identical string between the matching footprints **

   copy_length ← ExtendMatch (baseFile[basePos], verFile[verPos]);
   length ← EmitCopy (verPos, basePos, copy_length, verFile, baseFile, diffFile);

   return length;
end
```

Figure 2.5: Pseudo-code for the EmitCodes function.

same data in the two file versions. The task of the linear differencing algorithm can be described as the detection of points of synchronization and subsequently copying from synchronized offsets. We use the **Footprint** function and the hash table to find points of synchronization and term this the "hashing mode" of the algorithm. Once synchronized offsets have been found, the **EmitCodes** function uses a byte identity check to extend the match for as long as the offsets are synchronized, *i.e.* the strings are the same. This phase of the algorithm is termed "identity mode". When the byte identity test fails, the respective file pointers are "out of synch" and the algorithm re-enters hashing mode.

2.4 Analysis of the Linear Time Algorithm

We often expect the changes between two versions of a file to be edits, insertions of information and deletions of information. This property implies that the common strings that occur

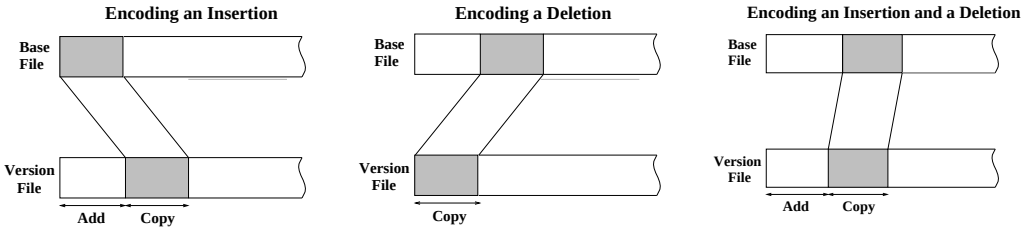


Figure 2.6: Simple file edits consist of insertions, deletions and combinations of both.

in these files are sequential. An algorithm can then find all matching strings in a single pass over the inputs files. After finding a match, we can limit our search space for subsequent matches to only the file offsets greater than the end of the previous matching string.

Many files exhibit insert and delete only modifications, in particular, mail files and database files. Mail files have messages deleted out from the middle of the file and data appended to the end. Relational database files operate on tables of records, appending records to the end of a table, modifying records in place, and deleting records from the middle of the table. System logs have an even more rigid format as they are append only files. Under these circumstances, we expect the linear algorithm to find all matches and compress data as efficiently as the greedy algorithm.

2.4.1 Performance Analysis

The presented algorithm operates both in linear time and constant space. At all times, the algorithm maintains a hash table of constant size. After finding a match, hash entries are flushed and the same hash table is reused to find the next matching footprint. Since this hash table neither grows nor is deallocated, the algorithm operates in constant space, roughly the size of the hash table, on all inputs.

Since the maximum number of hash entries does not necessarily depend on the input file size, the size of the hash table need not grow with the size of the file. The maximum number of

hash entries is bounded by twice the the number of bytes between the end of the previous copied string and the following matching footprint. On highly correlated files, we expect a small maximum number of hash entries, since we expect to find matching strings frequently.

The algorithm operates in time linear in the size of the input files as we are guaranteed to advance either the base file offset or the version file offset by one byte each time through the inside loop of the program. In identity mode, both the base offset and the version offset are incremented by one byte at each step. Whereas in hashing mode, each time a new offset is hashed, at least one of the offsets is incremented, as matching footprints are always found between the current offset in one file and a previous offset in another. Identity mode guarantees to advance the offsets in both files at every step, whereas hashing mode guarantees only to advance the offset in one file. Therefore, identity mode proceeds through the input at as much as twice the rate of hashing mode. Furthermore, the byte identity function is far easier to compute than the Karp–Rabin [8] hashing function. On highly correlated files, the algorithm spends more time in identity mode than it would on less correlated versions. We can then state that the algorithm executes faster on more highly correlated inputs and the simple linear algorithm operates best on its most common input, similar version files.

2.4.2 Sub–optimal Compression

The algorithm achieves less than optimal compression when either the algorithm falsely believes that the offsets are synchronized, the assumption that all changes between versions consist of insertions and deletions fails to hold, or when the implemented hashing function exhibits less than ideal behavior.

Due to the assumption of changes being only inserts and deletes, the algorithm fails to find

rearranged strings. Upon encountering a rearranged string, the algorithm takes the next match it can find. This leaves some string in either the base file or in the version file that could be compressed and encoded as a copy, but will be encoded as an add, achieving no additional compression. In Figure 2.7, the algorithm fails to find the copy of tokens ABCD since the string has been rearranged. In this simplified example we have selected a prefix for footprints of length one. The algorithm encodes EFG as a copy and flushes the hash table, removing symbols ABCD that previously appeared in the base file. When hashing mode restarts the match has been missed and will be encoded as an add.

The algorithm is also susceptible to spurious hash collisions, as a result of taking the next match rather than the best match. These collisions indicate that the algorithm believes that it has found synchronized offsets between the files when in actuality the collision just happens to be between two matching strings at least as long as the footprint length. In Figure 2.7, the algorithm misses the true start of the string ABCDEF in the base file (best match) in favor of the previous string at AB (next match). Upon detecting and encoding a "spurious" match, the algorithm achieves some degree of compression, just not the best compression. Furthermore, the algorithm never bypasses "synchronized offsets" in favor of a spurious match. This also follows directly from choosing the next match and not the best match. This result may be generalized. Given an ideal hash function, the algorithm never advances the file offsets past a point of synchronization.

Hashing functions are, unfortunately, not ideal. Consequently, the algorithm may also experience the *blocking* of footprints. When a fixed length string hashes to a footprint, if there is another footprint from a non-matching string in the same file already occupying that entry in the hash table, we say that the footprint is being *blocked*. In the simple linear algorithm the second footprint is ignored and the first one retained. This is the correct procedure to implement next match assuming that all footprints represent a unique string. However, hash functions generally

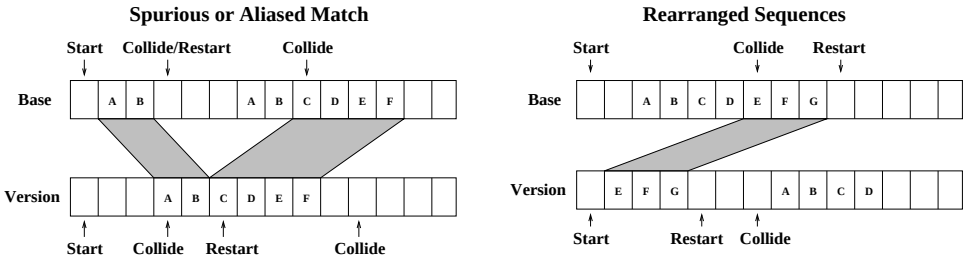


Figure 2.7: Sub-optimal compression may be achieved due to the occurrence of spurious matches or rearranged strings. The encoded matches are shaded.

hash a large number of inputs to a smaller number of keys and are therefore not unique. Strings that hash to the same value may differ and the algorithm loses the ability to find strings matching the discarded footprint.

Footprint blocking could be addressed by any rehash function or hash chaining. However, this solution would destroy the constant space utilization bound on the algorithm. It also turns out to be unnecessary as we will show in Chapter 3. The upcoming solution, called “undoing the damage”, solves this problem expressly without relying on a probabilistic method. However, the following solution is adequate for basic algorithms. Instead of a rehash function, we propose to address footprint blocking by scanning both forwards and backwards in identity mode. This simple modification allows the algorithm to go back and find matches starting at a footprint that was hash blocked. The longer the matching string, the less likely that the match will be blocked as this requires consecutive blocked footprints. Under this solution, the algorithm still operates in constant space, and although matches may still be blocked, the probability of blocking a match decreases geometrically with the length of the match.

Chapter 3

More Advanced Algorithms for Binary Differencing

In Chapter 2, we defined some basic techniques useful for differencing arbitrary byte streams and established two algorithms using these methods. In this chapter, we introduce advanced techniques useful for binary differencing and reformulate our previous algorithms to take advantage of these methods.

The first method we term “undoing the damage”. When a differencing algorithm runs, it finds strings to be copied and strings to be added and outputs them to a delta file. We modify this scheme and send the output encodings to a buffer. This buffer can be best thought of as a first in first out queue *(FIFO)* that caches the most recent encodings made by the algorithm. By caching encodings, an algorithm has the opportunity to recall a given encoding and exchange it for a better one. In many cases, an algorithm that uses this technique can make a quick decision as to an encoding, and if this decision turns out to not be the best decision, the encoding will be undone in lieu of a more favorable encoding. This technique is not orthogonal to other methods

in the algorithm and generally an algorithm needs to be significantly modified to take advantage of undoing the damage.

We also introduce a technique called “checkpointing” which reduces the amount of information that an algorithm needs to consider. Checkpointing takes a subset of all possible footprint values and calls these *checkpoints*. All footprints that are not in this subset are discarded and the algorithm runs on only the remaining checkpoints. This allows the file size and consequently the execution time to be reduced by an arbitrarily large factor. There is, unfortunately, a corresponding loss of compression with the runtime speedup. The technique is orthogonal to our other methods and can be applied to any of these algorithms.

3.1 Repairing Bad Encodings

A linear run time differencing algorithm often has to encode stretches of input without complete information. The algorithm may have found a common string between the base and version files which represents the best known encoding seen in the files up to this point. However, as the algorithm passes over more of the input files, it may find a longer common string that would encode the same region of the file more compactly. Under these circumstances, it becomes beneficial to let the algorithm change its mind and re-encode a portion of the file. This is termed “undoing the damage” and allows the algorithm to recover from previous bad decisions.

In general, an algorithm performs the best known encoding of some portion of a version file as its current version file pointer passes through that region. If it later encounters a string in the base file that would better encode this region, the old encoding is discarded in favor of the new encoding.

For our differencing algorithms, the hash table acts as a short term memory and allows

the algorithm to remember strings of tokens, so that when it sees them again, it may encode them as copies. This occurs when the algorithm finds a prior string in the base file that matches the current offset in the version file. Undoing the damage uses the symmetric case: matching strings between the current offset in the base file and a previous offset in the version file. The short term memory also allows the algorithm to recall and examine previous encoding decisions by recalling strings in the version file. These may then be re-encoded if the current offset in the base file provides a better encoding than the existing codewords.

To implement undoing the damage, the algorithm buffers codewords rather than writing them directly to a file. The buffer, in this instance, is a fixed size first in first out (*FIFO*) queue of file encodings called the "codeword lookback buffer". When a region of the file is logically encoded, the appropriate codewords are written to the lookback buffer. The buffer collects code words until it is full. Then, when writing a codeword to a full buffer, the oldest codeword gets pushed out and is written to the file. When a codeword "falls out of the cache" it becomes immutable and has been committed to the file.

3.1.1 Editing the Lookback Buffer

Our algorithm performs two types of undoing the damage. The first type of undoing the damage occurs when the algorithm encodes a new portion of the version file. If the algorithm is at the current offset in the file being encoded, new data will be encoded and added to the lookback buffer. The algorithm attempts to extend that matching string backwards from the current offset in the version file. If this backward matching string exceeds the length of the previous codeword, that encoding is discarded and replaced with the new longer copy command. The algorithm will "swallow" and discard codewords from the top of the lookback buffer as long as the codewords in

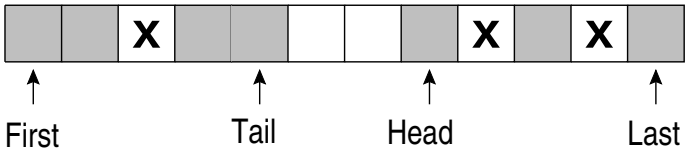


Figure 3.1: The fixup buffer implemented as a circular queue. Shaded squares contain encodings and **X**s mark dummy encodings.

question are either:

- A copy command that may be wholly re-encoded. If the command may only be partially re-encoded, the codeword may not be reclaimed and no additional compression can be attained.
- Any add command. Since add commands are followed by the data to be added, reclaiming partial add commands benefits the algorithm. While no codewords are reclaimed, the length of the data to be added is reduced and the resulting delta file decreases in size proportionally.

The second type of undoing the damage is more general and may change any previous encoding, not just the most recent encoding. If a matching string is found between the current offset in the base file and a previous offset in the version file, the algorithm determines if the current encoding of this offset of the version file may be improved using this matching string. The algorithm searches through the buffer to find the first codeword that encodes a portion of the version file where the matching string was found. The matching string is then used to re-encode this portion, reclaiming partial add commands and whole copy commands.

3.1.2 Implementing the Codeword Lookback Buffer

Undoing the damage requires that the codeword lookback buffer be both searchable and editable, as the algorithm must efficiently look up previous encodings and potentially modify or

erase those entries. The obvious implementation of the codeword lookback buffer is a linked list that contains the codewords, in order, as they were emitted from a differencing algorithm. This data structure has the advantage of simply supporting the insert, edit and delete operations on codewords. However, linear search of a linked list can be time consuming. Consequently, we implemented the codeword lookback buffer as a *FIFO* built on top of a fixed size region in contiguous memory (Figure 3.1). This region is divided into fixed sized elements and each element is an entry in the codeword lookback buffer. An element in the lookback buffer contains the necessary data to emit its codeword. It also contains the version offset, the region of the version file that this entry encodes. The version offsets in this data structure are increasing and unique. Therefore, any codeword in the n elements in this data structure can be looked up by version offset using binary search, which takes $\mathrm{O}(\log n)$ time. With linear search, we would require $\mathrm{O}(n)$ time for an n element linked list.

The circular queue uses a fixed amount of memory. The pointers **first** and **last** mark the boundaries of the allocated region. Within this region, the data structure maintains pointers **head** and **tail**, which are the logical beginning and end of the *FIFO*. These pointers allow the queue to wrap around the end of the allocated region as it does in Figure 3.1. Simple pointer arithmetic around these four pointers supports the access of any element in the queue in constant time.

This implementation of a first in first out queue suffers from no obvious support for insert and delete operations. Fortunately, our algorithms have special needs for insert and delete and can reasonably be limited to a few operations on the *FIFO*.

The algorithms require the append operation to fill the queue with encodings. We support this operation by incrementing the **tail** pointer. When appending an element on the end, if the queue is full, we must expel the element pointed to by the **head** pointer and increment that pointer to make room for the new encoding.

Since our implementation does not support insert, all other operations are prohibited from increasing the number of elements in the queue. When editing the lookback buffer, we allow the algorithm to replace an element, logically delete an old encoding and insert a new encoding in its place, by editing the values of the codeword.

We also support the delete operation by marking a current encoding as a *dummy* codeword. An algorithm will ignore this codeword for all subsequent operations in the queue. For example, when encountering a dummy element while performing binary search, an algorithm ignores this codeword and takes the closest valid encoding. When a dummy codeword is flushed from the queue, an algorithm outputs no data to its delta file. Whenever an algorithm inserts a dummy, the usable length of the *FIFO* is reduced by one until that entry is flushed.

When undoing the damage, we are trying to minimize the size of the delta file encoding. In general, this implies reducing the number of codewords that encode any given region and undoing the damage can be supported with the replace and delete operations that our implementation provides. Consider an operation that merges two adjacent codewords into a single codeword. This operation performs two deletes in the *FIFO* and one insert that occurs where the elements were deleted. We perform this action by editing one of the codewords to contain the new longer copy command and the other is marked a dummy codeword.

There is one case in undoing the damage that is excluded by our implementation. Consider that we have encoded an add command, and we later find that a portion of that add command can be re-encoded as a copy command. This operation reduces the size of the output delta file while increasing the number of codewords. Since there is no insert operation, our implementation fails to support this unless we are lucky enough to either find a dummy codeword adjacent to the add or the copy we find can swallow an adjacent codeword in addition to the add codeword we are modifying.

We feel that this limitation is a desirable tradeoff since we achieve asymptotically superior search time.

3.2 The One and a Half Pass Algorithm

Having developed the undoing the damage technique which improves the quality of the encodings that an algorithm can make, we modify our previous algorithms using this method.

The greedy algorithm always guarantees to find the best encoding by performing exhaustive search through its data structures for the longest matching string at any given footprint. At first glance it would seem that this method cannot be improved with undoing the damage. However, the greedy algorithm suffers from using both memory and and execution time inefficiently. As a consequence of linear memory growth and quadratic execution time growth, the greedy algorithm fails to scale well and cannot be used on arbitrarily large files.

The one and a half pass algorithm modifies the greedy algorithm by altering data structures and search policies to achieve execution time that grows linearly in the size of the input. Linear runtime comes at a price and the modifications reduce the one and a half pass algorithm's ability to compactly represent versions. We can then use the undoing the damage technique to improve the compression that the algorithm achieves. The resulting algorithm compresses data comparably to the greedy algorithm and executes faster on all inputs.

The significant modification from the greedy algorithm in the one and a half pass algorithm is that it uses the first matching string that it finds at any given footprint rather than searching exhaustively through all matching footprints. The algorithm discards the link table that was used in the greedy algorithm (Figure 2.2). Using the hash table only, the algorithm maintains a single string reference at each footprint value. By storing only a single string reference for each footprint, the

algorithm implements a first matching string rather than a best matching string policy when comparing footprints. This could be potentially disastrous, as the algorithm would consistently be selecting inferior encodings. Yet, by undoing the damage the algorithm avoids incurring the penalties for a bad decision. By choosing a first match policy, the algorithm spends constant time on any given footprint resulting in linear execution time. By maintaining only a single hash table of fixed size, the algorithm operates in constant space.

Let us consider a long matching string of length L and suppose our algorithm chooses instead a poor encoding. If we have a footprint of size F, the algorithm has $L - F$ different colliding footprints with which to find the long matching string. If it fails to find the string, this would imply that each and every of the $L - F$ footprints were overridden by another footprint. On long matches this occurs with geometrically decreasing probability.

We notice that the previous argument assumes that both footprinting and hashing are well behaved. This is a very reasonable assumption when the input data falls within the tolerable parameters of the algorithm, but does not hold for all inputs.

In particular, hashing ceases to behave well when the hash table becomes densely populated. So, our first requirement is that the total number of stored footprints, *i.e.* the length of the input file, is smaller than the number of storage bins in our hash table. We also require a suitably long footprint length so that the footprints well represent the strings they identify, but this condition must hold for any algorithm that uses a footprinting technique.

3.2.1 One and a Half Pass – Step by Step

The algorithm first passes over the base file, *baseFile* (Figure 3.2), footprinting a string prefix at every byte offset and storing these footprints for future lookup in a hash table. Having

```
procedure OneAndAHalfPass (prefixLength : number; baseFile, verFile, diffFile : stream )

local ver_pos, add_start, copy_start : number;
local hashtable[HASHTABLESIZE] : number;

begin
   ver_pos ← 0;
   add_start ← 0;

** Fill the Hash Table with Footprints from the baseFile **

   BuildHashTable (baseFile, hashtable);

   while (ver_pos ≤ length(verFile ) - prefixLength)

** FindFirstMatch hashes a footprint at ver_pos in verFile, looks in the hash table for a matching **
** string, and sets copy_start and copy_length to the offset and length of that string in baseFile **

      FindFirstMatch (ver_pos, copy_start, copy_length, hashtable, verFile, baseFile);

      if (copy_len ≥ prefixLength)
         if (add_start ≤ verPos)
            FixupBufferInsertAdd (add_start, ver_pos − add_start, verFile, diffFile);
         end if

         FixupBufferInsertCopy (ver_pos, copy_start, copy_length, verFile, baseFile, diffFile);
         ver_pos ← ver_pos + copy_len;
         add_start ← ver_pos;
      else
         ver_pos ← ver_pos + 1;
      end if
   endwhile

   FixupBufferInsertEnd (diffFile);
   FlushFixupBuffer (verFile, diffFile);
end
```

Figure 3.2: Pseudo-code for the One and a Half Pass Algorithm.

processed the base file, the algorithm footprints the first offset in the version file, *verFile*. The algorithm examines the hash table for a colliding footprint. If no footprints collide, we advance to the next offset by incrementing *ver_pos* and repeat this process.

When footprints collide the algorithm uses the **Verify** function to check the strings for identity. Strings that pass the identity test are then encoded and output to the fixup buffer. All symbols in the version file between the end of the last output codeword, *add_start*, and the beginning of the matching strings, *ver_pos*, are output as an add command. The matching strings are then output to the fixup buffer using the **FixupBufferInsertCopy** function.

The function **FixupBufferInsertCopy** (Figure 3.3) not only outputs the matching strings to the fixup buffer, it also implements undoing the damage. Before encoding the matching strings, the algorithm determines if they match backwards. If they do, it deletes the last encoding out of the queue and re-encodes that portion of the version file by integrating it into the current copy command. Having reclaimed as many backwards code words as possible, the function simply dumps a copy command in the buffer and returns. This one type of undoing the damage is adequate in this case as the algorithm has complete information about the base file as it encodes the version file.

We term this algorithm one and a half pass as it processes the base file twice and the version file once. Initially, this technique takes a single pass over the base file in order to build the hash table. Then, as the algorithm encodes the version file, random access is performed on the matching strings in the base file, inspecting only those strings whose footprints collide with footprints from the version file.

```
procedure FixupBufferInsertCopy (verPos, basePos, copyLen : number; baseFile, verFile, diffFile : stream)

local back_copy_length, reclaimed_length : number;
local current_el* : buffer_el;

begin
   reclaimed_length ← 0;
   back_copy_length ← ExtendMatchBackwards (verFile[verPos], baseFile[basePos]);

** LastBufferEl returns the top element in the Fixup Buffer **

   current_el ← LastBufferEl ();

   while (back_copy_length ≥ current_el.length)
      reclaimed_length ← reclaimed_length + current_el.length;
      current_el ← PopBufferEl (current_el);
   end while

** Encode the copy offset and length in the Fixup Buffer **

   current_el ← PushBufferEl (current_el);
   current_el.length ← copyLength + reclaimed_length;
   current_el.offset ← basePos − reclaimed_length;
   current_el.position ← verPos − reclaimed_length;
end
```

Figure 3.3: Pseudo-code for the FixupBufferInsertCopy subroutine.

3.2.2 Algorithmic Performance

We informally show that the algorithm runs in linear time by examining each step. In Figure 3.2, the algorithm generates a hash key for a footprint at each offset. The generation of a hash key takes constant time and must be done once for each footprint in the file, requiring total time linearly proportional to the size of the base file. Then, the version file is encoded. At each byte offset in the file, the algorithm either generates a hash key for the footprint at that offset, or uses the identity function to match the symbol as a copy of another symbol in the base file. In either case, the algorithm uses a constant amount of time at every offset for total time proportional to the size of the version file.

This algorithm has the potential to encode delta files as well as the greedy algorithm when the decision of choosing the first match is equally as good as choosing the best match. We can assert that the first match well represents the best match when the footprint hashing function generates “false matches” (see section 2.1.4) infrequently. Therefore, to achieve good compression, with respect to the greedy algorithm's compression, we must select a suitably long footprint. If the footprints uniquely represent the strings, the algorithms behave identically. However, the one and a half pass algorithm guarantees linear performance on all inputs and cannot be slowed by many strings with the same footprint.

3.3 The One Pass Algorithm

To implement the one pass algorithm, we modify the simple linear differencing algorithm with the advanced methods introduced in this chapter. The one pass algorithm improves the compression of the simple linear differencing algorithm without a significant depreciation in the execution time.

We recall that the simple linear differencing algorithm flushed its hash table discarding the available footprints. This was necessary in order to synchronize the pointers in the base and version file. To see that this is necessary, consider a frequently occurring string at the beginning of the base file. This string would match often in the version file and the pointer in the base file would never advance significantly beyond the occurrence of the common string. We therefore flush the hash table to ensure that no string matches more than once and consequently the file pointers are guaranteed to advance.

However, by flushing the hash table, the algorithm discards information that could later be valuable. If the algorithm was to make an encoding that was not from a point of synchronization,

```
procedure OnePass (prefixLength : number; baseFile, verFile, diffFile : stream)

local baseh, verh, base_pos, ver_pos, ver_start : number;
local verhashtbl[HASHTABLESIZE] basehashtbl[HASHTABLESIZE] : number;
local base_active, version_active : flag

begin
  ver_pos ← 0; base_pos ← 0;
  base_active ← TRUE; version_active ← TRUE;

  while (base_active or version_active)

    if (ver_pos ≤ length(verFile) - prefixLength)
      verh ← Footprint (verFile[ver_pos]);
      verhashtbl[verh] ← ver_pos;
      if ((SENTINEL ≠ basehashtbl[verh]) and (Verify (baseFile [basehashtbl[verh]], verFile[ver_pos])))
        length ← EmitCodes (basehashtbl[verh], ver_pos, ver_start, baseFile, verFile, diffFile);
        base_pos ← max (base_pos, basehashtbl[verh] + length);
        ver_pos ← ver_pos + length;
        ver_start ← ver_pos;
        continue;
      end if
    else
      version_active ← FALSE;
    end if

    if (base_pos ≤ length(baseFile) - prefixLength)
      baseh ← Footprint (baseFile[base_pos]);
      basehashtbl[baseh] ← base_pos;
      if ((SENTINEL ≠ verhashtbl[baseh]) and (Verify (verFile[verhashtbl[baseh]], baseFile[base_pos])))
        if (ver_start ≤ verhashtbl[baseh])
          length ← EmitCodes (base_pos, verehashtbl[baseh], ver_start, baseFile, verFile, diffFile);
          base_pos ← base_pos + length;
          ver_pos ← verhashtbl[baseh] + length;
          ver_start ← ver_pos;
          continue;
        else
          FixupEncoding (base_pos, verhashtbl[baseh], baseFile, verFile, diffFile);
        end if
      end if
    else
      base_active ← FALSE;
    end if
    ver_pos ← ver_pos + 1;
    base_pos ← base_pos + 1;
  end while

  EmitCodes (length(baseFile), length(verFile), ver_start, baseFile, verFile, diffFile);
  FixupBufferInsertEnd (diffFile);
  FlushFixupBuffer (verFile, diffFile);
end
```

Figure 3.4: Pseudo-code for the One Pass Algorithm.

```
number procedure EmitCodes (basePos, verPos, verStart : number; baseFile, verFile, diffFile : stream)

local copy_length : number;

begin

   if (verPos > verStart)
      FixupBufferInsertAdd (verStart, verPos − verStart, verFile, diffFile);
   end if

** Find the longest identical string between the matching footprints **

   copy_length ← ExtendMatch (baseFile[basePos], verFile[verPos]);
   length ← FixupBufferInsertCopy (verPos, basePos, copy_length, verFile, baseFile, diffFile);

   return length;
end
```

Figure 3.5: Pseudo-code for the EmitCodes function as modified to incorporate "undoing the damage".

the chance to later find a point of synchronization from that string is lost. The one pass algorithm does not flush the hash table in order to find potentially missed points of synchronization. The algorithm must then avoid the pitfall of not incrementing the file pointer when matching a frequently occurring common string. The algorithm does this by guaranteeing that the file pointers in both files are non-decreasing always and that when offsets are hashed, the pointers in both files advance. So, rather than trying to find the exact point of synchronization, the algorithm collects data about all previous footprints. The data that it accumulates arrives incrementally as it advances through the input files. The algorithm uses a replacement rule to update the hash table when there are identical footprints from the same file. This rule discards old information and preferentially keeps information close to the point of synchronization. The algorithm need not worry about making a bad encoding. Returning to the example of having a probable string in the base file, we notice two things. First, that any bad encodings made using this string can later be repaired by undoing the

```
number procedure FixupEncoding (basePos, verPos : number; baseFile, verFile, diffFile : stream)

local ret_value, copy_length, reclaimed_length : number;
local current_el* : buffer_el;

begin
   reclaimed_len ← 0;

** Find the longest matching string that contains these matching footprints **

   copy_len ← ExtendMatch (baseFile[basePos], verFile[verPos]);

** Locate the entry that encodes start of the match in the codeword buffer **

   current_el ← FindPreviousEncoding (verPos);
   if (current_el = NULL)
      return 0;
   endif

** Reclaim as many codewords as possible and remove them from the cache **

   do
      ret_value ← Reclaim (copy_length, current_el, verFile, diffFile);
      copy_len ← copy_len − ret_val;
      reclaimed_length ← reclaimed_length + ret_value;
      if ( retval ≠ current_el.length )
         break;
      endif
      current_el ← NextBufferEl (current_el);
   while (1);

** Re-encode the reclaimed codewords as a single copy **
** InsertBufferEl finds and returns the adjacent dummy codeword **

   if (reclaimed_length > 0)
      current_el ← InsertBufferEl (current_el);
      current_el.length ← reclaimed_length;
      current_el.offset ← basePos;
      current_el.position ← verPos;
   endif

   return reclaimed_length;
end
```

Figure 3.6: Pseudo-code for the FixupEncoding subroutine.

damage. Also, if the string is a probable match in the version file, *verFile*, (Figure 3.4), it should also occur frequently in the base file, *baseFile*. Each time the same footprint occurs in the same file, the reference to the string that generated the old footprint is purged from the hash table in favor of the new string. Our forward replacement rule prevents any single probable footprint from preventing the file pointers from advancing.

3.3.1 One Pass Step by Step

The one pass algorithm starts at offset zero in both files, generates footprints at these offsets and stores them in the hash tables. Footprints from *verFile* go into *verhashtbl* and footprints from *baseFile* in *bashashtbl*. It continues by advancing the file pointers, *ver_pos* and *base_pos*, and generating footprints at subsequent offsets. When the algorithm finds footprints that match, it first ensures that the strings these footprints represent are identical using the **Verify** function. For identical strings, it outputs the matched data to the fixup buffer using the **EmitCodes** (Figure 3.5) subroutine. We notice that the **EmitCodes** subroutine has been modified from its previous incarnation (Figure 2.5) to output codewords to the fixup buffer rather than outputting data directly to the file. The data that precedes the start of the copy is encoded in an add command using the function **FixupBufferInsertAdd**. The matched data is then output using the function **FixupBufferInsertCopy**.

FixupBufferInsertCopy implements one type of undoing the damage. Before encoding the current copy, the string is checked to see if it matches backwards. If the match extends backwards, the function re-encodes the previous codewords, if it produces a more compact encoding.

The one pass algorithm also implements undoing the damage when the current offset in *baseFile* matches a previous offset in *verFile*. This case of undoing the damage is different as it

attempts to repair an encoding from an arbitrary point in the cache, rather than just re-encoding the last elements placed in the codeword fixup buffer. In fact, the target codeword may have fallen out of the cache and not even be in the fixup buffer. The function **FixupEncoding** performs this type of undoing the damage (Figure 3.6). After finding the first codeword that encodes a portion of the string found in the version file, as many encodings as possible are reclaimed to be integrated into a single copy command.

We notice that the outer loop in the routine **OnePass** only runs when either the *base_active* or *version_active* flag is set. These flags indicate whether the file pointer has reached the end of the input. It is necessary to read the whole version file in order to complete the encoding. It is also necessary to finish processing of the base file, even if the version file has been wholly read, as the algorithm may use this information to undo the damage. This also differs from the simple linear differencing algorithm which completes after finishing processing in the version file. The simple linear differencing algorithm has no motivation to continue footprinting the base file after the version file has been encoded as it cannot modify previous encodings.

3.3.2 Windows into the Past

The per file hash tables in the one pass algorithm remembers the most recent occurrence of each checkpoint in each file. This results as the algorithm elects to replace existing footprints in the hash table with conflicting new occurrences of the same footprints. The hash tables tend to have complete information for the footprints from the most recent offsets. For older offsets, the hash table becomes incomplete with these footprints being overwritten. It is appropriate to consider this "memory" of previous strings through footprints as a window into the most recent offsets in each file. This window is the region over which the algorithm can act at any given time. A footprint

that has been expelled from this window cannot be used to create a copy command or to undo the damage.

Since data is replaced by conflicting footprints, the window in the past does not consist of contiguous data, but data about past footprints that gets less dense at offsets further in the past from the current file offset. This window dictates the effectiveness of the algorithm to detect transposed data. Consider two data streams composed of long strings A and B. One version of this data can be described by AB and the other by BA. We term this a transposition. This type of rearranged data can be detected and efficiently encoded assuming that the window into the past covers some portion of the transposed data. It is thereby beneficial for encoding transpositions to have a hash table that can contain all of the footprints in the base file.

3.4 One Pass and One and a Half Pass Compared

These algorithms are strikingly similar in their use of the same methods and data structures. Both algorithms use hash tables and footprinting to locate matching strings. Both algorithms implement undoing the damage to allow them to make hasty and efficient decisions. Perhaps the significant difference between the algorithms is the manner in which they access data in the input streams. The one and a half pass algorithm accesses data sequentially in the base file when building the hash table and accesses data sequentially in the version file when encoding. It only performs random access when verifying that colliding footprints are identical. This algorithm also only uses one hash table, so it uses memory slightly more efficiently.

The one pass algorithm may perform random access in either file but on highly correlated inputs this access should always be near the current file pointers and not to distant offsets in the past. What distinguishes the one pass algorithm from other algorithms is its on-line nature. Since the

algorithm starts encoding the version file upon initiation, it does not fill a hash table with footprints from the base file before encoding the version file, the algorithm emits a constant stream of output data. In fact, the algorithm can be described as having a data rate. This is a very important feature if one uses the algorithm to serve a network channel or for any other real time application.

The one pass algorithm behaves well under arbitrarily long input streams in that it only loses the ability to detect transposed data. The same cannot be said of the one and a half pass algorithm. Since it has only a single hash table with no ability to re-hash, when that hash table is full, the algorithm must discard footprints. This results in pathologically poor performance of inputs that overflow the one and a half pass algorithm's hash table. Note that both algorithms fail to perform optimally when the input is such that their hash tables are filled. In the next section, we will address this problem using a method called checkpointing.

3.5 Using Checkpoints to Reduce Information

In our analysis of the advanced algorithms presented in this chapter, we notice that both algorithms have performance limitations associated with the size of the input file. These limitations arise as a result of the hash tables these algorithms use becoming overloaded. As increasing the size of the hash table is not a scalable solution, we present a method to reduce the amount of information in a file that is compatible with the one pass and one and a half pass algorithms.

The checkpointing method declares a certain subset of all possible footprints checkpoints. The algorithm will then only operate on footprints that are in this checkpoint subset. We still need to run the hashing function at every offset, but only those footprints that are in the checkpoint subset participate in finding matches. This reduces the entries in the hash table and allows algorithms to accept longer inputs without the footprint entry and lookup operations breaking down.

This method allows us to reduce the file size by an arbitrary factor chosen so that our algorithm exhibits its best performance. We then need to address the issues of selecting checkpoints and integrating checkpointing into the existing algorithms.

3.5.1 Selecting Checkpoints

We choose the number of checkpoints in a file in order to achieve good behavior out of the hash table for storage and look up. A heuristic for selecting checkpoints is to choose a value so that the number of checkpoints found in a given input stream will number approximately half the size of the hash table, *i.e.* the hash table is populated to half capacity. Letting F be the set of all possible footprints, we select a set C of checkpoints such that $C \subset F$.

For an input stream of length L and a hash table of size H, we choose to have $H/2$ checkpoints occur in the input stream. We expect on average to obtain a checkpoint every $|F|/|C|$ [1] tokens. So, an algorithm must choose $|C|$ such that the number of checkpoints that appear over an input stream of length L produces $H/2$ hash entries. A rule to approximate this condition chooses $|C|$ such that

$$|C| \leq \frac{H|F|}{2L} \tag{3.1}$$

This only approximates the constraint because our argument is probabilistic and we cannot guarantee that one of our checkpoints will not be very popular and occur frequently. Such behavior is not problematic for our algorithms as they only store one string at any given checkpoint. This will not produce undesirable behavior in the footprinting storage and lookup operations. Instead, this checkpoint will generally not generate copy encodings in the algorithm as we have stored only one

[1] For a set S, we use $|S|$ to indicate the cardinality of that set

of its many occurrences.

An algorithm must also ensure that the set C of all checkpoints can address every element in the hash table, *i.e.* $|C| \geq H$. To satisfy this requirement, an algorithm must choose an appropriately large footprint size. An algorithm can select a minimum cardinality of the set F to ensure this bound on $|C|$. As the footprint length in bits is the logarithm of $|F|$, we choose a footprint of length l such that:

$$l = \lceil \log(|F|) \rceil : F \geq 2L \tag{3.2}$$

Having constrained the number of desired checkpoints and the minimum footprint length, we now turn to the checkpoint selection process. With the goals of efficiency and simplicity, we choose checkpoints using test equality with the modulo operation. So given $|C|$ checkpoints and $|F|$ footprints, a given footprint $f \in F$ is a checkpoint if

$$f \mod (|F|/|C|) = k \tag{3.3}$$

for some integer $k \neq 0$ chosen from the interval $[0, |F|/|C|)$. We select a non-zero value for k to ensure that the string of all zeros is not in the checkpoint set. Many types of data stuff zeros for alignment or empty space. Therefore, this string, with the corresponding checkpoint equal to zero, is frequently occurring and therefore not beneficial.

3.5.2 Integrating Checkpoints with Differencing Algorithms

We perform checkpointing in an on-line fashion and implement checkpointing as a conditional test that guards the inside loop of an algorithm. Our algorithms perform checkpointing by

testing every footprint as we hash it. When generating a footprint, if it meets the criterion described in equation 3.3, continue the algorithm normally. If it fails this test, advance to the next offset and continue execution. This implementation is orthogonal to the algorithms that use it and can be isolated to the one step where the algorithm generates the next footprint.

3.5.3 Checkpoints and the One and a Half Pass Algorithm

Checkpointing alleviates the failure of the one and a half pass algorithm operating on large input files. By choosing an appropriate number of checkpoints (equation 3.1), the algorithm can fit the contents of any file into its hash table. Of course, nothing comes for free: checkpointing has a negative effect on the ability of the algorithm to detect small matching strings between file versions. If an algorithm is to detect and encode matching strings, one of the footprints of this string must be a checkpoint. Matching strings approximately the size of the footprint length will have few colliding footprints and will consequently be missed with greater likelihood. On the other hand, for versioned data, we expect highly correlated input streams and can expect long matching strings which contain checkpoints with increasing probability.

We also note that the checkpointing technique relies upon undoing the damage and performs better on the one and a half pass algorithm than the greedy algorithm. Since checkpointing does not look at every footprint, an algorithm is likely to miss the starting offset for matching strings. With undoing the damage, this missed offset is handled transparently, and the algorithm finds the true start of matching strings without additional modifications to the code.

3.5.4 Checkpointing and the One Pass Algorithm

The one pass algorithm has problems detecting transpositions when its hash table becomes over-utilized. This feature is not so much a mode of failure as a property of the algorithm. Applying checkpointing as we did in the one and a half pass algorithm allows such transpositions to be detected. Yet, if the modification of the data does not exhibit transpositions, then the algorithm sacrifices the ability to detect fine grained matches and gains no additional benefit.

With the one pass algorithm, the appropriate checkpoint value depends on the nature of the input data. For data that exhibits only insert and delete modifications (section 2.4), checkpointing should be disregarded altogether. Any policy decision as to the number of checkpoints is subject to differing performance, and the nature of the input data needs to be considered to formulate such a policy. In our opinion, it can rarely be correct to choose a policy as drastic as equation 3.1, because the algorithm will then never fill its hash table and never use its full string matching capabilities. Perhaps a more appropriate heuristic would be to choose enough checkpoints so that the window into the past (section 3.3.2) covers more than half of the input data stream.

Chapter 4

Experimental Results for Binary Differencing

4.1 Compression Results

All of the presented algorithms were run on the same data sets in order to compare their compression performance. Since we are interested in file system applications, we targeted our test on a large and varied suite of files. For mostly security reasons, access to every file in a file system was not available. Instead, experiments were run against different versions of binaries and source code distributions. These distributions included the gnu tools distribution and Linux kernels among other things.

These files have many of the desirable properties for our experiments, including having many small files and few large files as we would expect in a file system [4]. It also contains a wide variety of files including images, text, source, libraries and binary files. What it does not include are the active user and system files such as mail files, database, configuration files, and system

Algorithm	Versions (bytes)	Deltas (bytes)	Size Reduction	Compression Factor
Greedy	17130984	1881619	11.0%	9.10
Simple Linear		3989045	23.3%	4.29
One Pass		3032184	17.7%	5.65
One and a Half Pass		2068500	12.1%	8.28

Table 4.1: A comparison of the compression performance of all algorithms.

logs. Nonetheless, we feel that this is a diverse and representative data set. All told we tested the algorithm on over 3,000 files. We also note that since we are comparing versions that are different releases of the same piece of software, and that we only examine the files that have been modified, the compression achieved over this data set is much less than that over a typical pair of sequential versions in a file system. This is true because a release of software combines the cumulative changes to a set of files over the period between releases. Most applications are concerned with daily changes or, for version control, changes between check-in. These periods are shorter than a software release cycle and consequently reflect fewer modifications.

We can see by table 4.1 that the greedy algorithm provides the best compression over all files. Not only does it provide the best compression, it also provides the best compression on every file. This is consistent with our analysis in chapter 2. The greedy algorithm is near optimal and will be used as a basis for compression comparison for all other algorithms.

We notice right away that the single pass or “streaming algorithms”, such as the simple linear and one pass algorithm, perform significantly less well than the one and a half and greedy algorithms. Greedy and one and a half pass start encoding the version with a full hash table, and can make good encodings on the first byte. The streaming algorithms start with an empty hash table and have to collect data before the encodings improve. While having an “on line” algorithm for file

differencing is extremely desirable for applications with real time requirements, these algorithms pay a significant compression penalty on many inputs.

The comparison of one pass and simple linear does delimit the benefits of undoing the damage and checkpointing. We conclude from table 4.1 that undoing the damage and checkpointing improve compression by over 5.5%. This improvement can be attributed almost entirely to undoing the damage, as the one pass algorithm behaves well with or without checkpointing. In figure 4.1, we see a file by file comparison of the simple linear and one pass algorithms and see that undoing the damage provides an improvement on many files. Data below the unit slope line indicate a more compact delta produced by the one pass algorithm. Many data points lie along the unit slope line showing that undoing the damage is not always helpful. We also note that points may lie above the unit slope line. In this case, the strict pointer synchronization of the simple linear algorithm provides better compression. From this we conclude that the casual synchronization of the one pass algorithm may lead to a bad decision that cannot be repaired. However, the simple linear algorithm never significantly outperforms the one pass algorithm, whereas the one pass algorithm does provide significant compression benefits on many inputs.

The one and a half pass algorithm performs almost as well as the greedy algorithm. We find this result particularly remarkable given that the one and a half pass algorithm does not search exhaustively for good matches and implements a simple and efficient form of undoing the damage. Cumulatively, the one and a half pass algorithm compresses within 1.1% of the greedy method. In figure 4.2, we see that the one and a half pass algorithm performs almost as well as the greedy algorithm consistently.

One concern with the one and a half pass algorithm is its scalability. This algorithm relies heavily on checkpointing to handle input streams that are larger than the hash table that it uses. We

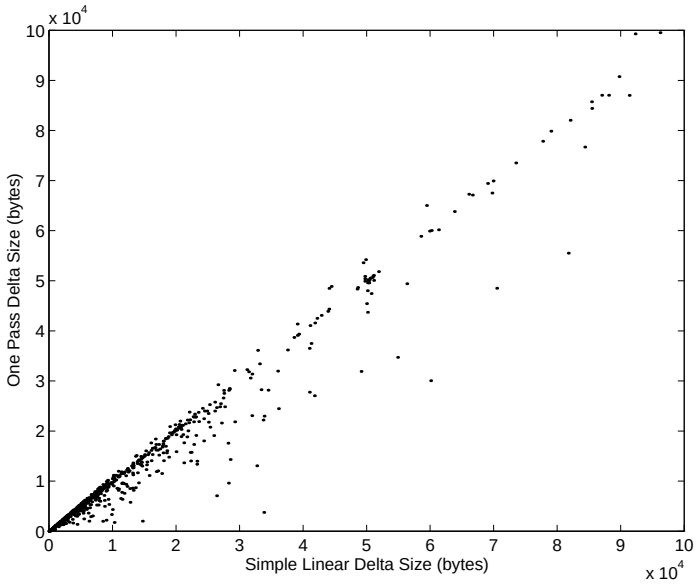


Figure 4.1: Compression comparison of the one pass and the simple linear algorithms.

are concerned with the corresponding loss of compression as inputs grow larger. The experiments we ran, while having large files, did not explore the scaling limits of this algorithm. It seems conceivable that for some relatively large inputs, the one pass algorithm's unlimited scalability will allow it to compress as well as the one and a half pass algorithm.

4.2 Best and Worst Case Execution Time

In order to probe the execution time limits of the four presented algorithms, we conduct a best case and worst case experiment.

The best case experiment ran the algorithms against versions of files that are identical. This is the best case for all algorithms as they all maximize the length of matching strings upon detecting colliding footprints and encode the whole file upon finding the first matching footprint. The best case results for the simple linear and the one pass algorithm show slow linear growth as

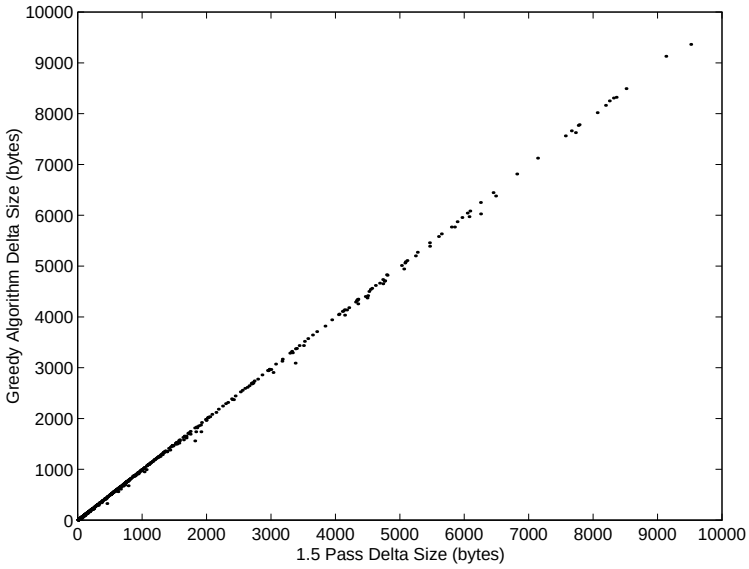
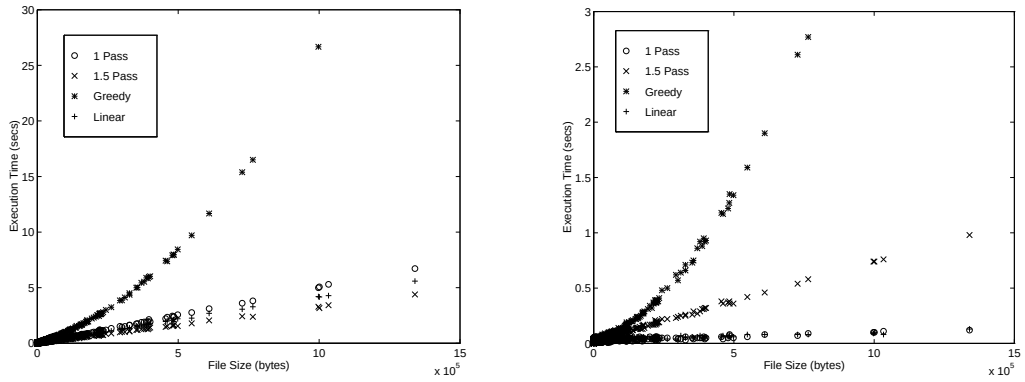


Figure 4.2: Compression comparison of the one and a half pass and greedy algorithms.

the input file size increases. This represents the time required to perform the byte identity function, *i.e.* the data rate of identity mode. These algorithms have a best case data rate of approximately 10 MB/s. The one and a half pass algorithm exhibits a data rate of 1.4MB/s. All runtime measurements were taken on an IBM 43P with a 133MHz processor and a 10MB/s fast wide SCSI disk interface. Recalling that the one and a half pass algorithm hashes the whole file before starting to encode the version file, the data rate of the one and a half pass algorithm represents the execution speed of the Karp–Rabin incremental footprinting function added to the byte identity function. Finally, the greedy algorithm exhibits quadratic execution time. For small values the curve closely approximates the one and a half pass algorithm as it also performs incremental hashing and byte identity. However, once the hash table becomes full, the algorithm needs to build the link table as well. Insertions in the link table (figure 2.2) require the algorithm to pass over all previous insertions of the same footprint. For n insertions this requires $1 + 2 + 3 + ... + n = n(n-1)/2$ steps.



(a) Worst case (b) Best case

Figure 4.3: Execution time results for four binary differencing algorithms

The worst case experiment ran the algorithms against randomized data. To generate randomized data, files were run through the IDEA encryption algorithm using two different keys. The output files should then have the same length, but any given bit should be completely independent of the same bit in the other file. Random data represents the worst case for all algorithms as it means that the algorithms almost never perform the byte identity function, and consequently footprint every offset.

The greedy algorithm exhibits its quadratic execution time in the worst case and the runtime cost of differencing becomes exorbitant for files much larger than 1 MB. This results as the algorithm performs quadratic work to both build and search in the link list. All of the other algorithms exhibit linear execution time in the worst case. The linear differencing algorithm exhibits a streaming data rate of approximately 160 KB/s and the one pass algorithm has a streaming data rate of 135 KB/s. The one and a half pass algorithm has a cumulative data rate of 180 KB/s.

The best and worst case results show the improved behavior of the linear runtime algo-

rithms on all inputs. We note that the streaming algorithms outperform the one and a half pass algorithm in the best case. This is again because these algorithms do not hash a whole file before they start encoding data. So, when the versions are identical, they can enter the byte identity function right away. Perhaps more interesting is the performance in the worst case. The one and a half pass algorithm outperforms all others. This is due mainly to its simplicity. When data is random, all algorithms hash nearly all offsets in both files. The algorithm with the simplest operations at each offset performs best. We also note that the simple linear algorithm runs faster than the one pass algorithm as it never tries to undo the damage. The one pass algorithm, even with no matching strings in files, will detect colliding footprints with non-zero probability and spend effort trying to encode this data. Also, undoing the damage, even when not invoked, costs an algorithm a few operations at every byte offset.

Both of these experiments represent reality and are therefore important to understanding algorithmic performance. The best case scenario we expect to appear consistently as many files change slightly or display append only modifications. In these situations, algorithms run the byte identity function almost exclusively. The worst case appears less frequently, when a file has been deleted and replaced with completely different data, or most of the file's bytes have been modified. While worst case inputs are not the common situation, it is important to ensure good behavior at all times.

Chapter 5

Conclusions and Afterthoughts

In this work, we have reviewed the previous art and introduced new methods for file differencing. The previous methods include footprinting for finding matching strings, Karp–Rabin hashing functions, and delta file encoding. To these existing methods, we add a checkpointing technique for reducing the information in a file and an undoing the damage technique to allow algorithms to repair sub-optimal encodings. Having developed improved methods, we formulate new algorithms for file differencing. The algorithms we present are not remarkable except that they provide a means to explore advanced techniques in file differencing.

The greedy algorithm was presented and its compression performance was shown to be optimal under some simplifying assumptions. We later use this algorithm's compression performance as a model to which we compare other algorithms. The greedy algorithm does exhibit quadratic execution time growth in the size of the input. As a consequence, it does not scale to accept large files. We experimentally show that the completion of this algorithm takes a prohibitive amount of time on files as small as 1 MB and conclude that the algorithm's performance is unacceptable for file system applications.

Using only previously known techniques, the simple linear algorithm provides a scalable differencing solution for binary inputs of any size. Analysis of the algorithm shows it to perform well on some known inputs, insertion and deletions, and poorly on other inputs, rearranged sequences. The simple linear algorithm also has an “on line” property in that it provides a reliable output data stream. This feature can be exploited to use this algorithm for real time applications that serve data channels. Our experiments show this algorithm to provide significantly less compression than the greedy algorithm.

Neither the greedy nor simple linear algorithm has an adequate combination of performance and compression to be a generalized differencing solution for file system applications. Consequently, we applied checkpointing and undoing the damage to both methods to improve the compression of the simple linear algorithm and the performance of the greedy algorithm.

Applying these techniques to the simple linear algorithm, we create the one pass algorithm. Undoing the damage in the one pass algorithm improves the compression performance of the linear algorithm by over 5.5% and only marginally increases the execution time. The increased execution time can be attributed solely to the complexity that undoing the damage and checkpointing add to the algorithm. As the “on line” property and scalability of the simple linear algorithm are preserved, the one pass algorithm provides a superior solution for applications with real time requirements or arbitrarily large inputs.

We modified the greedy algorithm to produce the one and a half pass algorithm. The algorithm achieves linear run-time by making hasty encodings, and then relies on undoing the damage to correct its worst decisions. The algorithm achieves compression nearly identical to the stated optimal greedy algorithm and has the best worst case run time performance. A combination of exceptional performance and simplicity make this algorithm stand out as clearly superior to previous

methods and experimentally better than the one pass algorithm. There are, however, some outstanding concerns about the scalability of this algorithm under large inputs that need to be resolved by further experimentation.

Having presented a family of efficient and general differencing algorithms, we establish file differencing as a viable data compression method for any application that versions files. We envision differencing as an enabling technology that will amplify the performance of network applications on low bandwidth channels and help mitigate resource limitations for distributed computing and Internet applications.

Bibliography

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers, principles, techniques, and tools*. Addison-Wesley Publishing Co., Reading, MA, 1986.

[2] ALDERSON, A. A space-efficient technique for recording versions of data. *Software Engineering Journal 3*, 6 (June 1988), 240–246.

[3] APOSTOLICO, A., BROWNE, S., AND GUERRA, C. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science 92*, 1 (1992), 3–17.

[4] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. Measurements of a distributed file system. In *Proceedings of the 13th Annual Symposium on Operating Systems* (Oct. 1991).

[5] BLACK, A. P., AND CHARLES H. BURRIS, JR. A compact representation for file versions: A preliminary report. In *Proceedings of the 5th International Conference on Data Engineering* (1989), IEEE, pp. 321–329.

[6] EHRENFEUCHT, A., AND HAUSSLER, D. A new distance metric on strings computable in linear time. *Discrete Applied Mathematics 20* (1988), 191–203.

[7] FRASER, C. W., AND MYERS, E. W. An editor for revision control. *ACM Transactions on Programming Languages and Systems 9*, 2 (Apr. 1987), 277–295.

[8] KARP, R. M., AND RABIN, M. O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development 31*, 2 (1987), 249–260.

[9] MILLER, W., AND MYERS, E. W. A file comparison program. *Software – Practice and Experience 15*, 11 (Nov. 1985), 1025–1040.

[10] MORRIS, R. Conversations regarding differential compression for file system backup and restore, Feb. 1996.

[11] REICHENBERGER, C. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management, Trondheim, Norway, 12-14 June 1991* (June 1991), ACM, pp. 144–152.

[12] RICK, C. A new flexible algorithm for the longest common subsequence problem. In *Proceedings of the 6th Annual Symposium on Combinatorial Patterm Matching Espoo, Finland, 5-7 July 1995* (1995).

[13] ROCHKIND, M. J. The source code control system. *IEEE Transactions on Software Engineering SE-1*, 4 (Dec. 1975), 364–370.

[14] TICHY, W. F. The string-to-string correction problem with block move. *ACM Transactions on Computer Systems 2*, 4 (Nov. 1984).

[15] TICHY, W. F. RCS – A system for version control. *Software – Practice and Experience 15*, 7 (July 1985), 637–654.

[16] TSOTRAS, V., AND GOPINATH, B. Optimal versioning of objects. In *Proceeedings of the Eight International Conference on Data Engineering, Tempe, AZ, USA, 2-3 Feb. 1992* (Feb. 1992), IEEE, pp. 358–365.

[17] WAGNER, R., AND FISCHER, M. The string-to-string correction problem. *Journal of the ACM 21*, 1 (Jan. 1973), 168–173.