A compact representation for file versions: a preliminary report

Andrew P. Black
Digital Equipment Corporation

Charles H. Burris, Jr.
Seattle Pacific University

Abstract

This paper presents a new system for the compact representation of multiple versions of a file. The presentation is in terms of vectors and matrices, which results in conceptual simplicity. Algebraic transformations enable the retrieval process to be optimized for any given version or set of versions, in contrast to always optimizing for the most recent or least recent version. Moreover, any version can be added or deleted without affecting any other. File differencing and dictionary compaction are unified, and data compression can be included.

A compact representation for the (sparse) matrices is presented, and the main algorithms are described in terms of this representation.

I. Introduction

Many forms of information processing involve dealing with successive versions of a file: production of a document or a program are obvious examples. Many file systems recognize this fact and provide direct support for versions: Digital's VAX/VMS™ operating system is a typical example. In those file systems that do not provide versioning (notably, UNIX®), application level software has been developed to maintain multiple versions in a single (file system level) file: AT&T's Source Code Control System (SCCS)[7] and Tichy's Revision Control System (RCS) [10] are examples. Such software serves two purposes: *control* and *compaction*.

From the software engineering viewpoint, version *control* is the major problem. It is important to be able to select a consistent set of file versions to compile a program or document. Some of the more successful systems for the control of multiple versions do no compaction at all; the interested reader is referred to references [4] and [8].

The work reported here was supported in part by the National Science Foundation under grant DCR-8420945, and by a Digital Equipment Corporation External Research Grant. The work was initiated at the University of Washington.

Author's current addresses: Andrew P. Black, Digital Equipment Corporation, Cambridge Research Laboratory, One Kendal Square, Bldg 700, Cambridge, MA 02139; Charles H. Burris, Jr., Seattle Pacific University, Seattle, WA 98119.

™ VAX/VMS is a trademark of Digital Equipment Corporation

® Unix is a registered trademark of AT&T Bell Laboratories

This paper is concerned solely with version *compaction*, the problem of efficiently representing multiple (scores, perhaps hundreds) versions of a file efficiently. By "efficiently" we mean both in terms of storage space and time taken to store and retrieve versions. One approach to this problem is data compression, using techniques such as Huffman Coding [1] or the Ziv-Lempel algorithm [11]. These techniques exploit the non-random nature of data *within* a file to reduce its bulk by as much as seventy-five percent [5]. In contrast, the techniques to be described in this paper attempt to capitalize on the similarities *between* different versions of the same file. It is perfectly feasible to apply data compression techniques to the data that results from version compaction.

II. Current Compaction Schemes

The development of selection matrices has been influenced by three existing version compaction systems – SCCS, RCS, and AVL dags.

SCCS [7] stores all the versions in a single composite file called the *s-file*. Segments from various versions are bracketed by editing directives that state which versions of the file should contain that segment. The reconstruction process involves one complete pass through the s-file, looking at every editing directive and applying those which pertain to the version being reconstructed. The time to retrieve any version is dependent upon the total number of editing directives for all versions; as the number of versions increases it takes longer to reconstruct each version. Once text has been deleted, it is excluded from all future versions. Therefore, if a section of a file is deleted in one version and re-inserted in the next, that section will be stored twice in the file: once in the delete command, and again in the insert command. This happens even if the text is re-inserted in exactly the place from which it was deleted.

RCS [9, 10] stores the most recent version in plain text and uses a series of reverse deltas to generate each of the other versions. A delta is composed of editing directives which transform a "source file" into a "target file"; RCS stores the delta necessary to transform the i^{th} version into the $i-1^{\text{th}}$ version. These deltas (and the plain text) are all stored in a single file, but the deltas are separate rather than interleaved as in SCCS.

If the current version is the n^{th}, in order to retrieve the m^{th} version all the intermediate versions $n-1, n-2, \cdots, m+1$ must be created. The time required to generate any version therefore depends upon how many intermediate versions must be created. (In the actual implementation of RCS, the full text of the intermediate versions is represented as a "piece table", a table of pointers into the RCS file [10]. Application of a delta transforms piece table PT_r into PT_{r-1}. This avoids the excessive handling of large strings, but does not change the concepts.)

AVL dags represent a completely different approach. They were introduced by Myers [6] as a way of efficiently representing multiple versions of data structures in applicative languages. If an array is represented as a binary tree, and a new array is generated that differs from the old in just one element, then it is not necessary to store both trees in their entirety. One needs only two root nodes, and two distinct paths to the element that differs; the rest of the structure can be shared. Clearly, duplication of nodes is minimized if the trees are balanced, hence the choice of AVL trees. Fraser and Myers [2] later adapted the same idea to the storing of versions of a file.

Since text is stored in each node, a change to a leaf node in an AVL Dag of height h will result in h sections of text being duplicated. Since over half of the text is stored in the leaves of a binary tree, this will occur, on average, in over half the changes. The expected number of duplicated nodes for a random change is approximately $h-1$.

III. Requirements

From a study of SCCS, and RCS , we can derive the following desiderata:

(1) Use a separate delta for each version.

(2) Construct each version directly using only one delta.

These requirements for deltas can be summed up by stating that any version must be reconstructable *independently* of the other deltas. This is the central idea of the selection matrix approach. Out of this we immediately get the following:

(1) Store each delta in a separate file. This is much faster than reading one large file and searching for the desired delta.

(2) Place all text in a separate file and call this the basis. Recall that SCCS interleaved text and deltas and so separation was impossible. RCS began to separate the deltas from the text, but brought along any new text to be inserted as part of the delta. Both these approaches result in duplicated text since text in woven into the editing directives. By placing all text in one file, duplication of text need not occur.

(3) As a result of 1 and 2, it becomes clear that the editing directives of the deltas have been reduced to a sequence of selection operations.

For example, if the basis consists of the characters a, b, c, d, e, then the selection operations:

(select 3 characters, starting with character 1)
(select 4 characters, starting with character 2)

will combine the two sequences *abc* and *bcde* into the version *abcbcde*.

This example points out another advantage of representing all versions as selections from a single basis. The block of text *bc* is repeated in the version *abcbcde*, yet it is stored only once in the basis.

The combination of basis and selection operations provides the following advantages:

- Separate Deltas
- Direct Deltas
- A block of deleted and reinserted text is stored only once.
- A block of repeated text is stored only once.

These ideas are formalized in the following section. The selection operations are represented as a sparse matrix; this provides a well-understood formal framework from which useful insights and results can be borrowed.

IV. Selection Matrices

We will frame our discussion in terms of "atoms"; atoms may be words, lines, or even single characters, depending on the application.

A *basis* $\mathbf{b}, \mathbf{c}, \cdots$ is a row vector of unique atoms, and is used to contain the text of our files. For example, define

$$\mathbf{b} = [\ \textit{"the "}\ \textit{"man "}\ \textit{"bit "}\ \textit{"dog "}\ ].$$

A *selection matrix* $S, T, \cdots$ is a matrix of 0s and 1s such that each column contains a single 1. For example,

$$S = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

is a 4×5 selection matrix. With the usual notation,

S is an $m \times n$ selection matrix $\equiv \forall j \in [1 \cdots n]\ \exists!\, i$ s.t. $S_{ij} = 1$.

The product of a basis and a selection matrix is defined in the usual way, with $\times$ denoting repetition and $+$ concatenation. For example, the first element of $\mathbf{b}S$ is obtained by multiplying $\mathbf{b}$ by S_{*1}:

$$\left[\mathbf{b}S\right]_1 = [\ \textit{"the "}\ \textit{"man "}\ \textit{"bit "}\ \textit{"dog "}\ ] \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= \textit{"the "}.$$

Multiplying $\mathbf{b}$ times the columns of S yields

$$\mathbf{b}S = [\ \textit{"the "}\ \textit{"dog "}\ \textit{"bit "}\ \textit{"the "}\ \textit{"man "}\ ].$$

It is convenient to define an operator TEXT that takes a row vector of atoms and produces a string:

$$\text{TEXT}(\mathbf{c}) = +/\mathbf{c}$$
$$= c_1 + c_2 + \cdots + c_n$$

Thus,

$$\text{TEXT}(\mathbf{b}S) = \textit{"the dog bit the man "}$$

We now have the most rudimentary tools for version compaction. Consider a sequence of all of the unique strings (atoms) from the set of versions, $\mu_1, \mu_2, \cdots$ to be compacted. This is a basis **b**, of size m. For each version μ there exists a selection matrix S_μ such that

$$\mathrm{TEXT}(\mathbf{b}S_\mu) = \mu$$

Now, imagine that a new version ν is added. If ν contains no new text, i.e., every atom in ν is already in **b**, then there is a corresponding selection matrix S_ν and all is well. More realistically, however, we should assume that ν contains i new atoms $a_1, \cdots, a_i$. We now need a new basis: $\mathbf{b}^+$ of length $m+i$ will do, where

$$b_j^+ = \begin{cases} b_j & \text{if } j \in [1 \,..\, m] \\ a_{j-m} & \text{if } j \in [m+1 \,..\, m+i]. \end{cases}$$

The basis $\mathbf{b}^+$ is called an extension of **b**.

What change, if any, must be made to an existing selection matrix when the basis is extended? The "domino rule" for matrix multiplication requires that the matrix must also be extended, so that it has as many rows as there are elements in the new basis. However, this is a trivial task. If $\mathrm{TEXT}(\mathbf{b}S_\mu) = \mu$, then the extended selection matrix S_μ^+ satisfying $\mathrm{TEXT}(\mathbf{b}^+S_\mu^+) = \mu$ can be obtained from S_μ simply by adding i zero rows to the bottom, one for each new atom in the extended basis.

V. Permutation Matrices

So far we have discussed one type of modification to a basis – extension. It is natural to ask what other sorts of modification might be made to a basis while still retaining the ability to construct a selection matrix that will generate the original file.

Clearly, it is not possible to replace one basis by another arbitrarily: the new basis must include all of the atoms in the old basis that are selected by any selection matrix. However, one transformation of the basis that is always possible is permutation of the atoms. Permutation matrices, which are square matrices with a single 1 in each row and each column, provide a conceptually simple way to reorder both the atoms in a basis and the rows of selection matrices.

Consider a basis **b** with m atoms, an $m \times n$ selection matrix S, an $m \times m$ permutation matrix P, and an $m \times m$ identity matrix I. Then from standard matrix theory

$$\mathbf{b}S = \mathbf{b}IS = \mathbf{b}PP^{-1}S = \mathbf{b}^r S^r$$

where $\mathbf{b}^r = \mathbf{b}P$ represents a reordering of the atoms in **b**, and $S^r = P^{-1}S$ is the rearranged selection matrix. Since the inverse of a permutation matrix is its transpose, $S^r = P^{-1}S = P^TS$.

To see how this works, consider the following example.

$$\mu = \textit{"the dog bit the man "}$$

$$\nu = \textit{"the man loved the dog "}$$

$$\mathbf{b} = [\,\textit{"the " "man " "bit " "dog " "loved "}]$$

$$S_\mu = \begin{bmatrix} 1&0&0&1&0 \\ 0&0&0&0&1 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} \text{ and } S_\nu = \begin{bmatrix} 1&0&0&1&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&1 \\ 0&0&1&0&0 \end{bmatrix}$$

Suppose it is desired to switch the second and fourth atoms in **b**. Choose permutation matrix

$$P = \begin{bmatrix} 1&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&1 \end{bmatrix}$$

which is the result of switching the second and fourth columns of a 5×5 identity matrix. Then

$$\mathbf{b}^r = \mathbf{b}P = [\,\textit{"the " "dog " "bit " "man " "loved "}\,],$$

$$S_\mu^r = P^TS_\mu = \begin{bmatrix} 1&0&0&1&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&0&1 \\ 0&0&0&0&0 \end{bmatrix},$$

$$S_\nu^r = P^TS_\nu = \begin{bmatrix} 1&0&0&1&0 \\ 0&0&0&0&1 \\ 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \end{bmatrix}$$

Thus

$$\mu = \mathrm{TEXT}(\mathbf{b}S_\mu) = \mathrm{TEXT}(\mathbf{b}^rS_\mu^r)$$

and

$$\nu = \mathrm{TEXT}(\mathbf{b}S_\nu) = \mathrm{TEXT}(\mathbf{b}^rS_\nu^r)$$

as required.

To see why basis reordering can be a useful operation, consider a large basis, kept in secondary store, supporting many versions. Suppose version ν uses only a small fraction of the atoms in the basis and is reconstructed quite frequently. Reorder the basis to put the atoms of ν near the beginning. Then reconstruction of ν would proceed as follows: First get the selection matrix for ν and determine the highest row number which contains a 1. Retrieve only that many atoms from the basis and piece these together as per the selection matrix to form the version ν.

Since a significant task in reconstruction is getting the atoms from secondary store, this could be a dramatic improvement.

VI. Submatrix Notation

The storage space for the selection matrix of ν can be reduced, which may further speed up the reconstruction process. The recognition that any selection matrix can be viewed as a set of identity submatrices suggests an alternative notation. For example, recall S_μ^r of the previous section. This matrix contains three indentity submatrices as shown below, where I_i is an identity matrix of size i.

$$S_\mu^r = \begin{bmatrix} 1&0&0&1&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&0&1 \\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} &&&I_1&0 \\ &I_3&&0&0 \\ &&&0&0 \\ 0&0&0&0&I_1 \\ 0&0&0&0&0 \end{bmatrix}$$

Now, represent each one of these identity submatrices by a triple which contains the location of the upper left corner and the size. Then

$$S_\mu^r = [(1\,1\,3)\ (1\,4\,1)\ (4\,5\,1)]$$

where each submatrix (r, c, l) is an identity matrix located at row r, column c, and of diagonal length l. An advantage of this representation is that it relates directly to both the selection operations needed in the reconstruction, and to the matrix notation. Also, if the basis is extended as a new version is added, the representation of existing selection matrices does not need to be changed.

A further reduction in the size of the representation can be achieved by recalling that each column of a selection matrix contains one non-zero value. If the identity sub-matrices are kept in order, then the column number is not needed. However, in order to keep the notation clear, the column numbers will be included in this paper.

The necessary algorithms for a simple version compaction system can now be expressed in terms of the row vector representation for the basis and the identity submatrix representation for selection matrices. The five high level commands of most interest are listed in Table 1.

Table 1: Version Compaction System Commands

Command	Description
store	Store a version in the version compaction system.
retrieve	Retrieve a copy of a previously stored version.
reorder	Reorder the basis and rearrange the rows of the existing selection matrices.
delete	Remove some versions from the set stored by the version compaction system
purge	Reduce the size of the basis by removing inactive atoms (atoms not required by any version). This command is used to recover storage after a series of delete operations.

From Table 1 we can extract the following list of elementary operations:

(1) **Construction** of a selection matrix S from a version μ and a basis $\mathbf{b}$, including a possible basis extension $\mathbf{b}^+$. (store)

(2) **Basis** $\times$ **Selection Matrix** – the product $\mathbf{b}S \rightarrow \mu$. This includes Basis $\times$ Permutation Matrix ($\mathbf{b}P \rightarrow \mathbf{b}^r$) since a permutation matrix is a special case of a selection matrix. (retrieve, reorder)

(3) **Transpose** of a Permutation matrix, $P \rightarrow P^T$. (reorder)

(4) **Permutation Matrix Transpose** $\times$ **Selection Matrix** – the product $P^T S \rightarrow S^r$. (reorder)

(5) **Discrimination** between active and inactive basis atoms, $\mathbf{b}_{active}$ and $\mathbf{b}_{inactive}$, given a basis $\mathbf{b}$ and a subset V_s of the selection matrices supported by $\mathbf{b}$. (purge)

(6) **Partition** – constructing a permutation matrix P that partitions $\mathbf{b}$ into active atoms followed by inactive atoms: $\mathbf{b}P \rightarrow [\mathbf{b}_{active}\ \mathbf{b}_{inactive}]$. (purge)

(7) **Optimization** of the basis ordering ($\mathbf{b}^r$) and the creation of the associated permutation matrix P such that $\mathbf{b}P \rightarrow \mathbf{b}^r$. (reorder)

Algorithms for the first six of these elementary operations are described below. The seventh operation, optimization, is the subject of section 7.

Data Structure – Basis

Description

Represent an atom as a string of characters. The atoms are stored in one large character array when in primary storage, and can be located given either the row number or the value of the atom. The end of an atom is identified by looking at the next entry in the *row index* table. See Figure 1.

By using the hash function, both of these operations (get the row number given the value, and get the value given the row number) can be performed in $O(1)$ time.

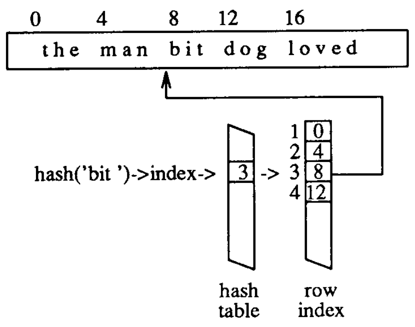

Figure 1: Data Structure for the Basis

The hash table and row index are built as the character strings are read from a file. If there are n atoms, then this operation takes $O(n)$ time and space.

Algorithm 1 – Construction

Description
Generate a Selection Matrix from a version and a basis.

Input
vec – a list of atoms representing the version μ.
basis – a list of unique atoms, possibly empty.

Output
a selection matrix
the (possibly extended) basis

Process
Locate each atom of *vec* in *basis* and append its location to an initially empty list. (This list forms the row numbers of the selection matrix). If the atom is not present in *basis*, extend *basis*.

Next, form a list of pairs by adding a length of 1 to each row number. Combine those pairs which are both adjacent in the list and whose row numbers represent adjacent atoms in the basis.

Last, turn the (*row*, *length*) pairs into triples if desired (see section VI).

If n is the number of atoms in *vec* then Algorithm 1 is $O(n)$ in time and space.

(end of algorithm 1)

Algorithm 2 – Basis × Selection Matrix

Description
Form the product of a row-vector and a Selection Matrix or of a row-vector and a Permutation Matrix

Input
basis – a non-empty list of unique atoms
mat – a selection matrix or permutation matrix

Output
μ – a list of atoms equal to the product $\mathbf{b}S$

Process
For each triple (r, c, l) in *mat*, the corresponding l atoms $vec_r, \cdots, vec_{r+l-1}$ are appended to the growing list of atoms μ.

This algorithm performs in $O(n)$ time and space where n is the number of atoms in μ.

(end of algorithm 2)

Algorithm 3 – Transpose

Description
Transpose a Permutation Matrix

Input
mat – a list of triples forming a permutation matrix.

Output
The transpose (and therefore the inverse) of *mat*

Process
Two steps are required here: first switch the row and column values of each triple in *mat*, and second sort the resulting triples by their new column numbers.

Due to the fact that the new column numbers are unique, this algorithm performs in $O(m)$ time and space where m is the number of rows in *mat*.

(end of algorithm 3)

Algorithm 4 – Permutation Matrix Transpose × Selection Matrix

Description
Compute the product of a permutation matrix transpose and a selection matrix, thus reordering the selection matrix. It is actually unnecessary to transpose the permutation matrix as a separate initial step; one can simply treat the row and column numbers in the triples that represent the permutation matrix as column and row numbers.

Input
p_mat – a permutation matrix
s_mat – a selection matrix

Output
The reordered selection matrix ($p\_mat^T \times s\_mat$).

Process
The first step is to construct a vector V (indexed from 1 to the number of rows in *s_mat*) so that the i^{th} entry in V contains the new location of row i of *s_mat*. For example, if row 3 of *s_mat* is to be moved to row 7, then $V[3] = 7$. V is constructed as follows: for each triple (r, c, l) in *p_mat*, assign $V[r] \leftarrow c$, $V[r+1] \leftarrow c+1$, ... for a total of l successive locations in V. (If given p_mat^T instead, assign $V[c] \leftarrow r$, etc.).

The next step is to reorder *s_mat* using the information in V. Each triple (r, c, l) of *s_mat* is converted to $(V[r], c, l)$. If the length l of the triple is greater than 1, then the reordering may cause that triple to split into several smaller triples. This happens when successive rows in a triple do not correspond to successive new rows in the vector. For example, consider the triple (3, 5, 1). Since $V[3] = 7$, this triple becomes (7, 5, 1). Now consider (3, 5, 3). If $V[4] = 8$ and $V[5] = 9$ then we can simply generate a new triple (7, 5, 3). However, suppose that $V[4] = 4$; it is then necessary to split (3, 5, 3) into (3, 5, 1) and (4, 6, 2). The same process must now be applied to (4, 6, 2).

After all triples of *s_mat* have been assigned new row numbers, combine any adjacent triples to form the reordered selection matrix.

This algorithm performs in $O(n)$ time and space where n is the number of columns in *s_mat*.

(end of algorithm 4)

Algorithm 5 – Discrimination

Description
Generate a list of active and inactive atoms from a list of selection matrices.

Input
s_mat_list – a list of selection matrices.

Output

active – a boolean vector indicating locations (in the basis) of active (*true*) and inactive (*false*) atoms.

Process

Initialize *active* to *false* for each atom in the *basis*.

For each triple (r, c, l) of each selection matrix in *s_mat_list* set locations $r, r+1, \ldots, r+l-1$ of *active* to *true*.

If m is the total number of columns in the selection matrices of *s_mat_list*, and n is the number of atoms in the *basis*, then this algorithm performs in $O(m)$ time and $O(n)$ space.

(end of algorithm 5)

Algorithm 6 – Partition

Description

Construct a permutation matrix which will reorder the atoms of the basis to partition active atoms from inactive atoms.

Input

active – an array of boolean values such that $active[i] = true$ if and only if $basis_i$ is an active atom.

Output

p_mat – a permutation matrix which can be used to reorder the basis (and the associated selection matrices) to place all active atoms at the beginning of the basis. Inactive atoms can then be dropped from the basis if so desired.

Process

Traverse the boolean array *active* from beginning to end. For each string of *true* values, append the pair (*starting location*, *length*) to an initially empty selection matrix *p_mat_active*. For each string of *false* values, append the pair (*starting location*, *length*) to an initially empty *p_mat_inactive*.

Finally, append the pairs in *p_mat_inactive* to the end of *p_mat_active* and return this as *p_mat*. Recall that column numbers are optional.

This algorithm performs in $O(n)$ time and space where n is the number of atoms in the *basis*.

(end of algorithm 6)

VII. Optimal Basis Ordering

A key feature of the selection matrix approach to version compaction is the capability of reordering the atoms in the basis and the rows of those selection matrices that depend upon it. Reordering the basis obviously leaves the size of the basis itself unaffected, but it can reduce the number of triples in a selection matrix, thus reducing both the storage space required for that version and the time required for its reconstruction.

In order to study the relationship between the order of the atoms in the basis and the size of its selection matrices, consider a basis consisting of the atoms a, b, c and d and a version $v = a\ b\ c\ d$. Table 2 lists several possible orders of the basis atoms and the resulting selection matrices. The term "Expanded Selection Matrix" refers to the result of expressing each 1 in the array as a submatrix of size one. Thus the number of triples is equal to the number of columns.

As the atoms in the basis are arranged to agree more closely with their order in the version, adjacent triples can be combined. Each time a pair of atoms in the basis matches a pair in the version, two adjacent triples in the selection matrix can be combined into one. By examining the row numbers of the triples (the first value in each triple) and looking for runs (a sequence of integers), triples to be combined can be identified.

In the first example, one pair of triples can be combined, and so the expanded selection matrix of four triples is reduced to three. The last example in Table 2 shows four triples whose row numbers form a run. Thus the expanded matrix of four triples is reduced to one triple.

The process of minimization can be viewed graphically. Given a version η, construct the complete, weighted digraph G whose nodes represent atoms and whose edge weights are assigned as follows:

If x and y are any two nodes (atoms), then the weight of the directed edge from x to y is the number of adjacent pairs x, y in the version η.

The order of atoms in the basis corresponds to a path γ in the complete weighted digraph G (see Figure 2). This path must contain each atom exactly once. The sum of the edge weights on this path represents the amount that the expanded selection matrix can be reduced. Find the path with maximum edge weight total and it represents the optimal basis order. Clearly, in Figure 2, the optimal basis order is $[a\ b\ c\ d]$ with edge weight total = 3.

To see how this graphic approach works with multiple versions, recall the example of the previous section:

Table 2: Effect of Basis Order upon Selection Matrix for Version $a\ b\ c\ d$.

Basis	Expanded Sel Mat	Combined Sel Mat
$[b\ a\ c\ d]$	(211) (121) (331) (441)	(211) (121) (332)
$[d\ a\ b\ c]$	(211) (321) (431) (141)	(213) (141)
$[a\ b\ c\ d]$	(111) (221) (331) (441)	(114)

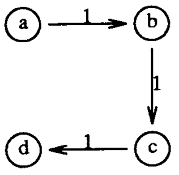
$$\begin{tikzcd} a \arrow[r, "1"] & b \arrow[d, "1"] \\ d & c \arrow[l, "1"] \end{tikzcd}$$

Figure 2. Complete Weighted Digraph G for Version $a\ b\ c\ d$.
(Zero weight edges not shown)

$$\mu = \text{"}\textit{the dog bit the man}\text{"}$$

$$\nu = \text{"}\textit{the man loved the dog}\text{"}$$

$$\mathbf{b} = [\ \text{"}\textit{the}\text{" "}\textit{man}\text{" "}\textit{bit}\text{" "}\textit{dog}\text{" "}\textit{loved}\text{"}]$$

$$S_\mu = [(1\ 1\ 1)\ (4\ 2\ 1)\ (3\ 3\ 1)\ (1\ 4\ 2)]$$

$$S_\nu = [(1\ 1\ 2)\ (5\ 3\ 1)\ (1\ 4\ 1)\ (4\ 5\ 1)].$$

The current basis ordering results in a total of eight triples for both versions. In order to see if this can be reduced, draw the complete weighted digraph G for versions μ and ν and find the path with maximum edge weight. Please see Figure 3.

The difference between the total number of triples in the expanded selection matrices for μ and ν and the total number of triples in the compressed forms for μ and ν is equal to the total edge weight of that path in the graph of Figure 3 which represents the order of the atoms in basis **b**.

The problem of reordering the basis to reduce the number of triples in the versions μ and ν can now be studied as a graph maximization problem. Example 1 in Table 3 represents the order chosen for the basis **b** originally. The second example is the order used to illustrate permutation matrices in Section 5. If the basis is now reordered to correspond to either example 3 or 4 of Table 3, the total number of triples in the compressed versions can be reduced from ten to five.

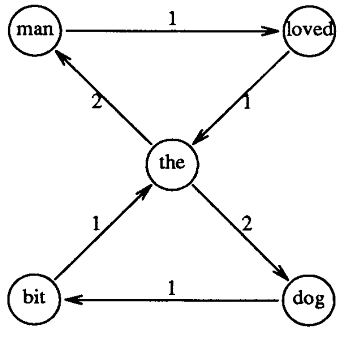

Figure 3: Complete Weighted Digraph G for Versions μ and ν.

Table 3: Edge Weight Totals for the Graph of Figure 3.

Ex.	Paths	Weight
1	[*"the " "man " "bit " "dog " "loved "*]	2
2	[*"the " "dog " "bit " "man " "loved "*]	4
3	[*"dog " "bit " "the " "man " "loved "*]	5
4	[*"man " "loved " "the " "dog " "bit "*]	5

To construct the permutation matrix necessary to reorder **b** as in example three of Table 3, make a table of the current and reordered locations for each atom (see Table 4).

Table 4: Current and Reordered Basis Atom Locations

Atom	Current Location	Reordered Location	Permutation Matrix
the	1	3	$I_{*1} \rightarrow P_{*3}$
man	2	4	$I_{*2} \rightarrow P_{*4}$
bit	3	2	$I_{*3} \rightarrow P_{*2}$
dog	4	1	$I_{*4} \rightarrow P_{*1}$
loved	5	5	$I_{*5} \rightarrow P_{*5}$

Now, beginning with a 5 × 5 identity matrix I with columns

$$I = [I_{*1} \mid I_{*2} \mid I_{*3} \mid I_{*4} \mid I_{*5}]$$
$$= [(1\ 1\ 1)\ (2\ 2\ 1)\ (3\ 3\ 1)\ (4\ 4\ 1)\ (5\ 5\ 1)],$$

rearrange the columns according to the rearranged locations of the atoms to form the permutation matrix P.

$$P = [I_{*4} \mid I_{*3} \mid I_{*1} \mid I_{*2} \mid I_{*5}]$$
$$= [(4\ 1\ 1)\ (3\ 2\ 1)\ (1\ 3\ 1)\ (2\ 4\ 1)\ (5\ 5\ 1)]$$

In matrix form, P and P^T are

$$P = \begin{bmatrix} 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&1&0&0&0 \\ 1&0&0&0&0 \\ 0&0&0&0&1 \end{bmatrix} \quad P^T = \begin{bmatrix} 0&0&0&1&0 \\ 0&0&1&0&0 \\ 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&1 \end{bmatrix}$$

Note that the reordered location column of Table 4 corresponds exactly to P^T.

Then

$$\mathbf{b}P = \mathbf{b}^r = [\ \text{"}\textit{dog}\text{" "}\textit{bit}\text{" "}\textit{the}\text{" "}\textit{man}\text{" "}\textit{loved}\text{"}\ ]$$

and the reordered selection matrix for version μ becomes

$$S_\mu^r = P^T S_\mu = \begin{bmatrix} 0&0&0&1&0 \\ 0&0&1&0&0 \\ 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&1 \end{bmatrix} \times \begin{bmatrix} 1&0&0&1&0 \\ 0&0&0&0&1 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&1&0&0&0 \\ 0&0&1&0&0 \\ 1&0&0&1&0 \\ 0&0&0&0&1 \\ 0&0&0&0&0 \end{bmatrix}$$

$$S_\mu^r = [(3\ 1\ 1)\ (1\ 2\ 1)\ (2\ 3\ 1)\ (3\ 4\ 1)\ (4\ 5\ 1)]$$
$$= [(3\ 1\ 1)\ (1\ 2\ 4)]$$

Similarly,

$$S_\nu^r = [(3\ 1\ 1)\ (4\ 2\ 1)\ (5\ 3\ 1)\ (3\ 4\ 1)\ (1\ 5\ 1)]$$
$$= [(3\ 1\ 3)\ (3\ 4\ 1)\ (1\ 5\ 1)]$$

and thus the total number of triples has been reduced by five as predicted.

The preceeding analysis has focused entirely upon reducing the *total* number of triples in the selection matrices. This achieves minimum storage requirements for the selection matrix approach with unique atoms, while also reducing the amount of processing necessary in reconstruction.

However, the speed of reconstruction may be more important than storage space. In that case, it is necessary to know the probability of reconstruction for each version. Then the expected number of selection operations can be computed by (first), multiplying the number of triples in each version by its probability of reconstruction, and (second), totaling these products. Since this corresponds directly to reconstruction time, it should be minimized.

Recall the previous example and suppose the probabilities of reconstruction for the versions are $p(\mu) = 0.4$ and $p(\nu) = 0.6$. Also let $N(x, y, \nu)$ = the number of occurances of the pair *xy* in version ν. Then, construct the complete weighted digraph G for versions μ and ν (see Figure 4) with edge weights defined as follows:

If x and y are any two nodes in G then

$$w(x, y) = N(x, y, \mu) \times p(\mu) + N(x, y, \nu) \times p(\nu)$$

Now, identify the path or paths which have maximum edge weight total. This total will represent the amount by which the expected number of triples (for the expanded selection matrices) can be reduced (see Table 5).

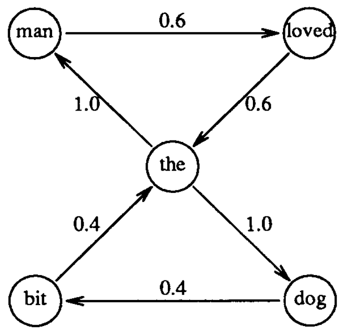

Figure 4: Complete Weighted Digraph G for Versions μ and ν with probabilities of reconstruction.

Table 5: Edge Weight Totals for the Graph of Figure 4

Ex.	Path	Weight
1	[*"the " "man " "bit " "dog " "loved "*]	1.6
2	[*"the " "dog " "bit " "man " "loved "*]	2.0
3	[*"dog " "bit " "the " "man " "loved "*]	2.4
4	[*"man " "loved " "the " "dog " "bit "*]	2.6

Constructing the permutation matrix P and its transpose P^T as before, and computing the matrix products $S_\mu^r = P^T S_\mu$ and $S_\nu^r = P^T S_\nu$ gives

$$S_\mu^r = [(3\ 1\ 3)\ (3\ 4\ 1)\ (1\ 5\ 1)]$$

and

$$S_\nu^r = [(3\ 1\ 1)\ (1\ 2\ 4)].$$

Now, the expected number of triples for the expanded matrices is

$$p(\mu) \times 5 + p(\nu) \times 5 = 5.0$$

and the expected number of triples for the compacted versions of the reordered matrices is

$$p(\mu) \times 3 + p(\nu) \times 2 = 0.4 \times 3 + 0.6 \times 2 = 2.4$$

which represents a reduction of 2.6 as was predicted by the path selected from Table 5.

From the preceding discussion it is clear that the optimum ordering of the basis depends upon two things:

(1) A set of user assigned probabilities.

(2) The identification of an atom ordering for the basis which maximizes the edge weight total of the associated digraph.

The first requirement is actually a strength of this method since if enhances user control of the performance of the version compaction system. The second requirement is comparable to solving the traveling salesperson problem, and could take a substantial amount of time to reach an otimum solution. Other methods which give "good" (but not optimum) results perform in much less time. For more details, see reference [3].

VIII. Future Work

To be a contribution to the state of the art, a new versioning system must be faster, or more compact, or more elegant than existing techniques; we may hope that it will be all three.

Naturally, we believe that the selection matrix approach is more elegant, since it has a simple and well understood mathematical basis, as well as a framework that can accommodate both dictionary-like schemes for data compression and file-differencing apprahes to versioning. However, any claim to elegance is by its very nature subjective, so our current work is aimed at investigating the speed and space efficiency of our approach.

The algorithms of Section VI have been prototyped in Franz Lisp, and the sizes of the compacted version sets compared with those produced by RCS and SCCS. For these tests, an atom was defined to be a line from the source file; this seemed to offer the most meaningfull comparisons, since both SCCS and RCS use lines as the unit of differencing.

Not counting the version control information maintained by SCCS and RCS (e.g., log messages and access control lists), the file sizes obtained for the selection matrix approach were for the most part within two percent of the sizes used by SCCS and RCS. The one exception to this was when a section of deleted text was reinserted; both SCCS and RCS duplicate this section, whereas our approach keeps it in the basis just once. In this case the version set based on selection matrices was one third smaller than the RCS file.

Comparing the running time of our Lisp prototype with the time taken by the production-quality C code in SCCS is futile. In addition, our current prototype uses simplistic algorithms, e.g., it locates atoms in the basis by sequential search. In general, the same sort of technique that speeds up the implementaion of RCS, such as hash-coding of lines in order to find matches, can also be applied to selection matrices, except that in our case comparision is with the basis vector rather than with another version.

Neither is the comparison of asymptotic worst case times very helpful, since in practice it is the constant factor in the average case that is of interest. Thus, we cannot yet offer any definitive verdict on the relative speed of our method compared to RCS and SCCS, and this is an area that we are continuing to investigate. Nevertheless, we are optimistic that our approach will be time efficient, for a number of reasons:

(1) The time taken to reconstruct a given version does not depend on the age of the version or on the number of versions stored.

(2) The ability to reorder the basis to minimize the expected number of selection operations, together with the optimal reordering results of Section VII, provide the ability to minimize the number of submatrices for a particular version or for a group of versions.

(3) The restriction introduced in Section IV, where we definied a basis as a row vector of *unique* atoms, can be relaxed. Although it is true that the size of the basis will be increased by allowing repetitions, the sizes of some selection matrices and thus the time required to reconstruct the version may be reduced.

Another parameter that affects both time and space requirements is the definition of an atom. Our initial tests defined an atom to be a line, to correspond to SCCS and RCS. An alternative is to define an atom to be a word, as was done for the examples in this paper. This might be beneficial for program texts, where reserved words and identifiers occur many times, but whole lines are duplicated infrequently. It also corresponds more closely to a dictionary approach to data compression. However, we have not yet fully investigated this approach; it is possible that the growth in the size of the selection matrices more than counterbalances the reduction in the size of the basis.

References

[1] Aronson, J. P. *Data Compression – A Comparison of Methods*. Institute for Computer Science and Technology, National Bureau of Standards, Washington, DC, June 1977. Gov. ordering no. NBS SP 500-12.

[2] Fraser, C. W. and Myers, E. W. "An Applicative Editor for Revision Control". Univ. of Arizona Tech. Rep. 84-20, October 1984.

[3] E. L. Lawler, J. K. Lenstra, A. H. G. RinnooyKan and D. B. Shmoys, eds., *The Traveling Salesman Problem: A guided Tour of Combinatorial Optimization*. John Wiley and Sons, New York, 1985.

[4] Marzullo, K. and Wiebe, D. "Jasmine: A Software System Modelling Facility". *Proc. ACM Software Eng. Notes/SIGPLAN Software Eng. Symp. on Practical Programming Development Environments*, Palo-Alto, CA, December 1986, pp.121-130. (SIGPLAN Notices Notices **22**, (1)).

[5] Miller, V. S. and Wegman, M. N. "Variations on a Theme by Ziv and Lempel", in *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil (editor). Springer Verlag, 1985, pp.131-140.

[6] Myers, E. W. "Efficient Applicative Data Types". *Conf. Rec. 11th ACM Symp. on Prin. of Prog. Lang.*, January 1984 , pp.66-75.

[7] Rochkind, M. J. "The Source Code Control System". *IEEE Trans. on Software Eng.* **SE-1**, *4* (December 1975), pp.364-370.

[8] Schmidt, E. "Controlling Large Software Developement in a Distributed Environment". CSL-82-7, Xerox PARC, December 1982. (Ph.D. Thesis UCB).

[9] Tichy, W. "Design, Implementation and Evaluation of a Revision Control System". *6th Intl. Conf. on Softw. Eng.*, September 1982, pp.58-67.

[10] Tichy, W. F. "RCS – A System for Version Control". *Software—Practice & Experience* **15**, *7* (July 1985), pp.637-654.

[11] Ziv, J. and Lempel, A. "Compression of individual sequences via variable-rate coding". *IEEE TRANS Inform. Theory* **IT-24**, *5* (September 1978), pp.530-536.